UNIVERSITAT DE LES ILLES BALEARS

DEPARTAMENT DE CIÈNCIES MATEMÀTIQUES I INFORMÀTICA
PROGRAMA DE DOCTORAT EN INFORMÀTICA

PH.D. THESIS

A document submitted in partial satisfaction
of the requirements for the degree *Doctor*

Improving Error Containment and Reliability of Communication Subsystems Based on Controller Area Network (CAN) by Means of Adequate Star Topologies

Manuel Barranco

SUPERVISORS

Julián Proenza
Luís Almeida

Fem constar que aquesta memòria ha estat realitzada, sota la direcció de Julián Proenza Arenas i Luís Miguel Pinho de Almeida, per Manuel Alejandro Barranco González i que constitueix la seva tesi doctoral.

Palma, març de 2010

Signat: Manuel Alejandro Barranco González
Estudiant de doctorat

Signat: Julián Proenza Arenas
Co-director de la tesi
Professor Titular d'Universitat
Departament de Ciències Matemàtiques i Informàtica
Universitat de les Illes Balears

Signat: Luís Miguel Pinho de Almeida
Co-director de la tesi
Professor Associado
Faculdade de Engenharia
Dep. de Eng. Electrotècnica e de Computadores
Universidade do Porto

To my parents and Elia.

There was Eru, the One, who in Arda is called Ilúvatar and he made first the Ainur, the Holy Ones, that were the offspring of his thought, and they were with him before aught else was made. And he spoke to them, propounding to them themes of music; and they sang before him, and he was glad. But for a long while they sang only each alone, or but few together, while the rest hearkened; for each comprehended only that part of the mind of Ilúvatar from which he came, and in the understanding of their brethren they grew but slowly. Yet ever as they listened they came to deeper understanding, and increased in unison and harmony.

En el principio estaba Eru, el único, que en Arda es llamado Ilúvatar, y primero hizo a los Ainur, los Sagrados, que eran vástagos de su pensamiento, y estuvieron con él antes de que se hiciera alguna otra cosa. Y les habló y les propuso temas de música; y cantaron ante él y él se sintió complacido. Pero por mucho tiempo cada uno de ellos cantó solo, o junto con unos pocos, mientras el resto escuchaba; porque cada uno sólo entendía aquella parte de la mente de Ilúvatar de la que provenía él mismo, y eran muy lentos en comprender el canto de sus hermanos. Pero cada vez que escuchaban, alcanzaban una comprensión más profunda, y crecían en unisonancia y armonía.

First fragment of ***Ainulindalë - The Music of the Ainur****, J.R.R. Tolkien (1892-1973)*

Resum

Els busos de camp han estat amplament utilitzats en sistemes de control distribuïts, incloent sistemes amb garanties de funcionament, ja que aquesta tecnologia proporciona robustesa elèctrica i baix cost. Aquest és el cas del protocol *Controller Area Network* (CAN), el qual és extremadament popular en aquest context degut principalment al seu baix preu, la seva facilitat de configuració, la seva robustesa elèctrica, i perquè proporciona una latència determinista d'accés al medi, així com bons mecanismes de detecció i contenció d'errors. Fins ara, els busos de camp típicament s'han basat en una topologia de bus simple, la qual planteja dubtes sobre la idoneïtat d'aquestes tecnologies per a aplicacions amb altes garanties de funcionament. Un dels principals problemes d'una topologia de bus és que els seus components estan interconnectats sense els mecanismes adequats de contenció d'errors. Això fa que una única fallada en qualsevol component de la xarxa (com per exemple un controlador de comunicacions, un transceptor, un connector o un cable) pugui generar errors que es propaguin per tot el subsistema de comunicacions, provocant una avaria generalitzada de comunicació. Aquesta característica persisteix inclús quan s'utilitzen de forma separada o conjunta tant busos replicats que proporcionen tolerància a fallades, com mecanismes addicionals per a incrementar la contenció d'errors. Aquesta limitació es deu principalment a les conegudes com fallades de proximitat espacial i de mode comú. Aquestes fallades afecten simultàniament a més d'un component del sistema, reduint o inhabilitant els seus mecanismes de detecció d'errors i tolerància a fallades.

A diferència dels busos, les topologies d'estrella poden oferir una millor contenció d'errors i independència de fallades i, per tant, poden conduir potencialment a l'obtenció de sistemes de comunicació més robustos. Concretament, el centre d'una topologia d'estrella, per exemple un concentrador, té una visió privilegiada de les comunicacions i podria fer-se servir per a detectar i aïllar fallades que ocorrin als medis i als nodes. Més encara, les topologies d'estrella no sofreixen ni fallades de proximitat espacial ni de mode comú. Per a moltes aplicacions, aquests beneficis potencials compensen el major cost que típicament tenen les topologies

d'estrella. De fet, el domini de les xarxes d'àrea local (LAN) ja fa temps que ha adoptat topologies d'estrella per a Ethernet, una tecnologia que avui en dia s'està adaptant a l'entorn de l'automatització industrial. Moviments similars s'han donat en el domini dels sistemes encastats, on noves tecnologies de busos de camp, com ara el Time-Triggered Protocol TTP/C i FlexRay en sistemes de control dins vehicles, han evolucionat devers topologies d'estrella. Aquestes dues tecnologies ofereixen acobladors (concentradors) d'estrella simple i replicada per tal de proporcionar una millor contenció d'errors i/o tolerància a fallades. Fins ara, CAN, que és probablement la xarxa més amplament utilitzada en sistemes distribuïts encastats, s'ha mantingut bàsicament com una xarxa de bus simple. Diferents investigadors també han proposat topologies d'estrella per a CAN. Malauradament, aquestes estrelles no ataquen el problema de la contenció d'errors ni la tolerància a fallades, o simplement tracten un conjunt reduït de possibles fallades. Fins i tot moltes d'elles ni tan sols són compatibles amb CAN o presenten problemes i limitacions elèctriques.

Aquests avantatges potencials de les estrelles que encara no han estat explotats per a CAN, ens motivaren a declarar que és possible millorar la contenció d'errors i la tolerància a fallades de sistemes basats en aquest protocol, canviant la seva topologia de bus simple per topologies d'estrella adequades. A més, nosaltres afirmem que aquesta millora de contenció d'errors i tolerància a fallades pot realment augmentar la fiabilitat d'aquests sistemes; on fiabilitat s'entén com la probabilitat de qué un sistema proporcioni el seu servei de forma ininterrompuda durant un determinat interval de temps. Estem especialment interessats en la fiabilitat perquè les aplicacions més noves contínuament requereixen un nivell més elevat d'aquest atribut, tal i com succeeix en els sistemes crítics on una fallada o un funcionament incorrecte pot causar un gran perjudici a les persones, a l'entorn, a equipaments molts costosos i, fins i tot, la pèrdua de vides humanes.

Per tal de validar aquestes afirmacions, en aquesta dissertació dissenyem i construïm dues topologies d'estrella inèdites, CANcentrate i ReCANcentrate. Des d'un principi es va imposar un requisit al seu disseny, a saber, preservar totes les especificacions de CAN, per tal de mantenir totes les bones propietats relacionades amb les garanties de funcionament que aquest protocol ja aconsegueix per sí mateix, així com per a assegurar la compatibilitat d'aquestes estrelles amb components CAN comercials i aplicacions i protocols basats en CAN. La primera de nostres propostes, CANcentrate, és una topologia d'estrella simple el concentrador de la qual incorpora mecanismes inèdits per a detectar i contenir errors als seus ports d'origen, de tal forma que millora la contenció d'errors de CAN més enllà de les capacitats de qualsevol altra estrella simple que hagi estat proposada prèviament per a aquest protocol. La nostra segona proposta, ReCANcentrate,

és una topologia d'estrella replicada que inclou dos concentradors similars al de CANcentrate. Addicionalment, cadascun dels concentradors de ReCANcentrate pot contenir els errors generats per l'altre concentrador. Més encara, ReCANcentrate inclou mecanismes per a tolerar fallades que afecten no tan sols a un dels concentradors, si no també als enllaços i als controladors de comunicacions dels nodes.

Finalment, per tal de corroborar quantitativament els beneficis que les topologies d'estrella poden aportar a la fiabilitat del sistema, hem modelat i avaluat, utilitzant *Stochastic Activity Networks* (SANs), la fiabilitat de sistemes basats en CAN, CANcentrate i ReCANcentrate. És important tenir en compte que tot i que hi ha hagut un interès creixent en la utilització de topologies d'estrella en tecnologies de busos de camp diferents de CAN, fins on nosaltres sabem, aquest és el primer intent de quantificar apropiadament l'increment de garanties de funcionament que les estrelles poden atènyer en comparació amb els busos.

Abstract

Field-buses have been extensively used in distributed control systems, including dependable ones, as this technology offers electrical robustness and low cost. This is the case of the Controller Area Network (CAN) protocol, which is extremely popular in this context mainly due to its low cost, simple configuration, electric robustness, deterministic medium-access delay, and good error-detection and containment features. However, field-buses have typically relayed on a simplex bus topology, which poses some uncertainties about their suitability for highly dependable applications. One of the main problems of a bus topology is that since it attaches components to each other without the appropriate error-containment mechanisms, a single fault in any network component (such as a communication controller, transceiver, connector, or wire) can generate errors that propagate throughout the communication subsystem, leading to a generalized failure of communication. This characteristic persists when using replicated buses to provide fault tolerance, additional facilities to increase error containment in a bus, or both. The main problem is the so-called spatial-proximity and common-mode failures. These failures simultaneously affect more than one component in the system, reducing or disabling its error-detection and fault-tolerance mechanisms.

In contrast, star topologies can provide better error containment and fault independence, thus potentially leading to more robust communication systems. Specifically, the center of a star topology, e.g. a hub, has a privileged view of the communication, and could be used to detect and isolate faults occurring at media and nodes. Moreover, star topologies do not suffer from spatial-proximity and common-mode failures. These potential benefits, in several applications, compensate for star topologies' typical higher cost. The local area network (LAN) domain, for example, has long moved to star topologies with Ethernet, a technology nowadays being adapted to the industrial automation domain. Similar moves occurred in the embedded systems domain, where newer field-bus technologies, such as the Time-Triggered Protocol TTP/C and FlexRay for in-vehicle systems, have also evolved to star topologies. These two technologies offer simplex or replicated

star couplers (hubs) for providing better error-containment and/or fault-tolerance. However, CAN, which is probably the most widely used network in distributed embedded systems, remained essentially a bus-only network. Therefore, researchers have also proposed star topologies for CAN. Unfortunately, these stars address neither error containment nor fault tolerance, or deal with only a small set of possible faults. Moreover, some of them are not even fully compatible with the CAN protocol or present electrical problems and limitations.

The unexploited potential benefits of stars in CAN motivated us to claim that it is possible to improve error containment and fault tolerance in CAN-based systems by changing its simplex bus topology to adequate star topologies. Moreover, we state that this improvement of error containment and fault tolerance can actually enhance the reliability of these systems; where reliability is understood as the probability with which a system continuously delivers its intended service throughout a given interval of time. We are specially interested on reliability since the level of this attribute required by newer applications is continuously increasing, as happens in critical ones where failure or malfunction may result in serious injury to people, environment, high-priced equipment or even in loss of lives.

In order to validate these claims, in this dissertation we design and implement two novel star topologies, CANcentrate and ReCANcentrate. One requirement was imposed from the beginning on their design, namely to preserve all the specifications of CAN, so that they keep all the dependability properties already achieved by CAN and they are compatible with commercial-off-the-shelf (COTS) CAN components and CAN-based applications and protocols. The first one of our proposals, CANcentrate, is a simplex star topology whose hub incorporates novel mechanisms to detect errors and contain them at their ports of origin, thereby enhancing error-containment of CAN beyond the capacities of any other simplex star topology previously proposed for this protocol. Our second approach, ReCANcentrate, is a replicated star topology that includes two hubs similar to the CANcentrate's one. However each one of them can also contain errors generated by the other hub. Moreover, ReCANcentrate includes mechanisms to tolerate faults occurring not only at one of the hubs, but also at links and in the communication controllers of the nodes.

Finally, to quantitatively corroborate the system reliability benefits of adequate star topologies, we modelled and evaluated, using Stochastic Activity Networks (SANs), the reliability of systems relying on CAN, CANcentrate and ReCANcentrate. It is important to note that despite the growing interest in using stars in fieldbus technologies other than CAN, to our best knowledge, this is the first attempt to appropriately quantify the dependability benefits stars achieve when compared with buses.

Acknowledgments

Al empezar a escribir estas líneas, me doy cuenta de lo difícil que es agradecer en tan poco espacio tantas cosas a tantas personas. Con total seguridad, me olvidaré de alguien o de algo, pues incluso hay quien con sus acciones nos ayuda, aun cuando nosotros no nos demos cuenta.

No obstante, en primer lugar quisiera intentar agradecer lo mucho que me han aportado las personas con las que he trabajado más estrechamente: mis supervisores Julián Proenza y Luís Almeida, así como Guillermo Rodríguez-Navas. Todos ellos me recibieron con los brazos abiertos y me arroparon desde un principio, guiando mis pasos por el estimulante y muchas veces arduo mundo de la investigación y la escritura. Han sido y son mi soporte y ejemplo, y rara es la vez en la que no amplío o cambio mi punto de vista tras conversar con ellos. De verdad que hace tiempo que perdí la noción de la cantidad de horas que, por ejemplo, Julián ha estado ahí, escuchando, leyendo y corrigiendo cuanto escribo; o las veces que Luís me ha ayudado a crear, a impulsar y a redondear trabajos. Es más, he encontrado en ellos a verdaderos amigos con los que compartir viajes (a veces surrealistas), alegrías, penas, conversaciones filosóficas y, en el caso de Guillermo, hasta consejos sobre Yoga.

Así mismo, he tenido mucha suerte de poder trabajar y compartir espacio con las personas del *Departament de Matemàtiques i Informàtica* de la UIB, en especial con los demás miembros del *Grup de Sistemes, Robòtica i Visió* (SRV). Fortuna como la de compartir despacho con José Guerrero, con el que he sobrevivido a microclimas invernales, del que siempre recibo una bienvenida y del que siempre encuentro respuesta a preguntas técnicas. Le estoy igualmente agradecido a personas como Alberto Ortiz, Yolanda González y Javier Antich, que me han apoyado y apoyan en el área de docencia y que siempre han estado disponibles cuando las he necesitado. Por supuesto, tampoco me puedo olvidar de Biel Oliver, Antoni Burguera, Francesc Bonin, Xisco Bonnin, Bartolomé Garau y Óscar Valero, cuya permanente alegría e ironía son contagiosas. Finalmente, también me gustaría agradecer a David Geßner su interés, dedicación y profesionalidad en la real-

ización de tareas técnicas directamente relacionadas con esta tesis; trabajar con él es sinónimo de inspiración, optimización y mejora.

En este punto recuerdo con gran cariño a muchas personas del *Departamento de Electrònica Telecomunicaçoes e Informática* de la *Universidade de Aveiro* en Portugal. Con ellos me sentí siempre como en casa; pues su hospitalidad no parece tener límites. En especial me gustaría nombrar a Ricardo Marau, Valter Silva, Arnaldo Oliveira y Pedro Duarte magníficos profesionales y mejores humanos si cabe, con un gran sentido de la amistad y cuyos consejos técnicos fueron vitales en las tareas relacionadas con los prototipos presentados en este documento. También me gustaría agradecer a José Alberto Fonseca, Paulo Pedreiras, Joaquim Ferreira, Fernando Morgado, Frederico Santos y Francisco Borges su amistad, su hospitalidad y, como no, su ayuda para conocer la cultura portuguesa (y brasileña). Sin lugar a dudas, todas estas personas me han hecho pasar muchos de los mejores momentos de mi vida.

In fact, the time that I spent in Aveiro was so intense that it is impossible not to smile when I remember people like Tullio and Agnese. You are just fantastic! Thank you very much for your friendship, hospitality and for all those good trips we made together through Portugal and Mallorca. Another very important person that helped me a lot is Elisabete Souza. In fact, she was my first friend in Aveiro and she is one of the best and more emotive persons I have ever met. She was also the person who firstly taught me Portuguese and I am very grateful to her for introducing me to the Brazilian culture. Of course, I cannot forget to thank Matthias Herrmann for everything. You are that kind of person that is surrounded by a special aura of friendship, freedom and adventure. In particular, you showed me the most funny and amusing lifestyle of Aveiro; you and the people from *Casa dos Loucos* are just unforgettable.

Finalmente, en lo que concierne a las personas que hicieron de mi estancia en Aveiro un sueño irrepetible, no sé cómo expresar mi gratitud y nostalgia hacia Doña Maria Do Santo Cristo. Ella es para mí como una segunda madre, pues me recibió en su casa como si me conociese de toda la vida y, de hecho, no sé hasta qué punto no era así en realidad. Le estoy muy agradecido por la cantidad de cosas que aprendí en su hogar y en el de su familia, desde los rudimentos de la preciosa lengua portuguesa, algunas de cuyas frases hechas me vienen a la cabeza casi todos los días, hasta la variedad, sutileza y calidad de su gastronomía, pasando por ese humor ácido y escatológico que les caracteriza y que tanto me recordó siempre al Mediterráneo. Siempre dice que le hice mucha compañía, pero en realidad vivir con usted fue como disfrutar de una segunda infancia.

I am also deeply indebted to all the anonymous reviewers of several publica-

tions for their comments and advices, some of which help me in improving important parts of the work carried out in the context of this dissertation, such as those concerning some paramount aspects of dependability evaluation; to Nicolas Navet from INRIA for the encouraging and clarifying conversations about automotive embedded systems; to Juan Pimentel from Kettering University for his valuable opinions concerning CANcentrate and ReCANcentrate, as well as for his personal invitation to participate at the SAE World Congress; to Francisco Vasques from the University of Porto, Thilo Sauter from the Vienna University of Technology, and Thomas Nolte from the Mälardalen University for encouraging me in publishing the results of my research.

Me gustaría también agradecer el cariño y la alegría de todas las personas con las que he compartido parte de mis viajes a congresos. Me acuerdo especialmente de Marga Marcos, Elisabet Estévez y Federico Pérez de la Universidad del País Vasco; saber que uno se va a encontrar con gente tan divertida y natural es un aliciente extra para investigar.

Ya fuera del ámbito universitario, me gustaría agradecer el apoyo de mis amigos más cercanos. Especialmente a Miguel Ángel Nieto por su amistad incondicional, su interés por mi trabajo, y nuestras discusiones sobre la utilidad y los límites de la ciencia y la tecnología.

En cuanto a mi familia, tengo que decir que es irrepetible. Le doy las gracias a Mamaíta y al Abuelo (que Dios tenga a ambos en su gloria), así como a Juanfra por creer en mí y por hacerme sentir tan querido. De la misma forma, le estoy muy agradecido a mi familia de Cartagena, que siempre se interesa y está orgullosa de mí; sobretodo a Isabel, la Yaya, Antonio y David. En especial me acuerdo de mi primo Tono, que en paz descanse; siempre echaré de menos nuestras marchas, conversaciones y muestras de cariño de ingeniero a ingeniero ;-). También le doy las gracias a mi familia "política", por su abundante hospitalidad gallega y por el aprecio que siempre me demuestran.

No sé cómo agradecer a mi madre y a mi padre todo lo que han hecho y hacen por mí. Sin ellos no sería nada. Desde el día en que nací no me han fallado ni una sola vez. Lo mismo puedo decir de mis hermanos José y Cristina y de mi cuñado Óscar, mis mejores amigos, que siempre me empujan a evadirme momentáneamente para hacer más llevadero el trabajo. Sólo espero poder corresponderos, aunque sé que es casi imposible.

Por último, quisiera agradecerle a Elia su amor y su paciencia. A pesar de ser tan transparente, ella es para mí un misterio y una fuente de inspiración que nunca llegaré a entender. Ojalá hubiese forma de agradecerte tu alegría y todo lo que haces y soportas por mí.

Contents

List of Figures **xvii**

List of Tables **xx**

1 Introduction **1**
1.1 Problem statement 1
1.2 The role of star topologies in improving dependability of CAN . . 3
1.3 The thesis 4
1.4 Main contributions 5
1.4.1 The simplex star CANcentrate 5
1.4.2 The replicated star ReCANcentrate 6
1.4.3 Quantifying the dependability improvement 7
1.5 Organization of the document 11

2 An overview on dependability **13**
2.1 Introduction 13
2.2 Basic concepts and terminology 13
2.3 Fault tolerance basics 19
2.4 Fundamentals of fault-tolerant systems design 22
2.5 Dependability concepts introduced for this work 23
2.6 Conclusions 24

3 Controller Area Network (CAN) protocol **27**
3.1 Introduction 27
3.2 CAN Physical Layer 28
3.3 CAN Data Link Layer 29
3.3.1 Frame format 29
3.3.2 Bit-wise arbitration mechanism 31
3.3.3 Frame encoding 31
3.3.4 Error detection and signalling 31

3.3.5 Fault treatment 34
3.3.6 Overload signalling 35
3.4 Types of faults in CAN networks 36
3.5 Conclusions 37

4 Potential solutions for improving dependability in CAN 39
4.1 Introduction 39
4.2 Replicated bus topology 40
4.3 Reconfigurable bus topology 40
4.4 Bus guardian 41
4.5 Star topologies 42
4.5.1 Passive star couplers 43
4.5.2 Active star couplers 44
4.5.3 Bridge star couplers 45
4.6 Conclusions 47

5 CANcentrate basics 49
5.1 Introduction 49
5.2 Fault model 49
5.3 Design rationale 50
5.4 Internal structure of the hub 52
5.5 Hub synchronization in the presence of errors at the coupled signal 55
5.6 Considerations on the cabling and bit rate 58
5.7 Conclusions 61

6 CANcentrate error-detection and fault-treatment mechanisms 63
6.1 Introduction 63
6.2 Error-detection and fault-treatment rationale 64
6.3 Current State signals for the Enabling/Disabling units 66
6.4 Stuck-at-recessive faults 67
6.5 Stuck-at-dominant faults 68
6.6 Bit-flipping faults 69
6.7 Reintegration policy 70
6.8 Conclusions 72

7 CANcentrate mechanisms for detecting bit-flipping errors 75
7.1 Introduction 75
7.2 Bit-flipping error-detection rationale 75
7.3 Error detection during normal transmission 77
7.3.1 Error detection on the transmitter contribution 78
7.3.2 Error detection on a receiver contribution 81

7.4 Error detection upon the occurrence of an error 83
7.4.1 Error detection after an error occurs in the *resultant frame* 83
7.4.2 Error detection after an error occurs on a port contribution 87
7.5 Error detection during an error signaling 89
7.6 Error detection during an overload signaling 93
7.7 Conclusions 94

8 Analysis of the mechanisms that deal with bit-flipping faults 97
8.1 Introduction 97
8.2 Complexity of the mechanisms 98
8.3 Advantages of the mechanisms 99
8.3.1 Enhanced error detection 100
8.3.2 Enhanced fault treatment 102
8.4 Limitations of the mechanisms and further enhancements 103
8.4.1 Unfair error detection during the error signaling 104
8.4.2 Unfair error detection after an *arbitration misunderstanding* 106
8.5 *Penalization policy* of the BFC Manager 108
8.6 Conclusions 111

9 CANcentrate prototype 115
9.1 Introduction 115
9.2 Description of the prototype 115
9.3 Experimental platform 118
9.4 Functional tests 121
9.5 Performance measurements 123
9.6 Conclusions 124

10 Reliability evaluation of CANcentrate 127
10.1 Introduction 127
10.2 Metrics 130
10.3 Modelling limitations 132
10.4 Modelling assumptions 133
10.4.1 Implementation assumptions 134
10.4.2 System components and entities 136
10.4.3 Basic statistical fault properties 137
10.4.4 Failure mode assumptions 139
10.4.5 Coverage assumptions 146
10.5 Modelling formalism 153
10.6 Modelling rationale 154
10.6.1 A dedicated SAN submodel per *entity* 155

10.6.2 A dedicated SAN submodel per *entity* type 158
10.6.3 A dedicated SAN submodel per *region* type 160
10.7 CANcentrate model 167
10.7.1 *nodeKernelsT* submodel 168
10.7.2 *nodeConnsT* submodel 171
10.7.3 *hubKernel* submodel 176
10.7.4 *branchesFailureEval* submodel 176
10.7.5 *CANcentrateFaiEval* submodel 177
10.8 CANbus model 179
10.8.1 *nodeKernelsB* submodel 180
10.8.2 *nodeConnsB* submodel 183
10.8.3 *inBusSections* and *edBusSections* submodels 187
10.8.4 *CANbusFaiEval* submodel 188
10.9 Quantitative assessment 189
10.9.1 NFT/AR vs number of nodes 191
10.9.2 FT/AR_1 vs number of nodes 192
10.9.3 FT/AR_1 vs system fault-tolerance coverage 194
10.9.4 FT/AR_1 vs fail-silent node proportion 197
10.9.5 FT/AR_1 vs bit-flipping coverage of the hub 201
10.9.6 FT/AR_1 vs bit-flipping proportion 203
10.9.7 FT/AR_1 vs out-of-fault-model proportion 208
10.9.8 FT/AR_1 vs wires and connectors' failure rates 211
10.9.9 FT/AR_1 vs Hub Core failure rate 216
10.10Conclusions 218

11 ReCANcentrate **227**
11.1 Introduction 227
11.2 Redundancy approaches in CAN 228
11.3 Fault model of ReCANcentrate 230
11.4 Design rationale 231
11.5 Internal structure of the hub 235
11.6 Error-detection and fault-treatment mechanisms of the hub 237
11.7 Node's media management strategy 238
11.7.1 Faults and discrepancies 240
11.7.2 Treatment of discrepancies and fault-tolerance strategy . . 242
11.8 Considerations on the cabling and bit rate 242
11.9 Prototype implementation 244
11.10Functional tests 245
11.10.1 Experiments under fault-free conditions 246
11.10.2 Experiments under the presence of faults 246

Stuck-at recessive faults at links and interlinks 247
Stuck-at-dominant and bit-flipping faults at links 248
Stuck-at-dominant and bit-flipping faults at interlinks . . . 249
Stuck-at-dominant and bit-flipping faults at a hub 250
11.11 Performance measurements 251
11.12 Conclusions 252

12 Reliability evaluation of ReCANcentrate **255**
12.1 Introduction 255
12.2 Metrics 256
12.3 Modelling assumptions 257
12.3.1 Implementation assumptions 257
12.3.2 System components and entities 258
12.3.3 Failure mode assumptions 258
12.3.4 Coverage assumptions 259
12.4 ReCANcentrate model 262
12.4.1 Modelling rationale 263
12.4.2 Some important preliminary remarks 267
12.4.3 *ReCANcentrateFaiEval* submodel 269
12.4.4 *nodeKernelsR* submodel 273
12.4.5 *nodeConnsR* submodel 282
12.4.6 *hubInConns* submodel 287
12.4.7 *hubKernels* submodel 292
12.4.8 *fauLPevalAtNode* submodel 298
12.4.9 *fauLPsEvalAtHubs* submodel 303
12.4.10 *fauLPevalAtHub* submodel 307
12.4.11 *fauIPevalAtHubs* submodel 311
12.4.12 *fauHubEvalAtHub* submodel 315
12.4.13 *fauHubEvalAtNodes* submodel 319
12.4.14 *ofmFauEval* submodel 326
12.5 Quantitative assessment 332
12.6 Conclusions 335

13 Conclusions and future work **341**
13.1 Thesis validation and contributions 342
13.1.1 First assertion 342
13.1.2 Second assertion 344
13.1.3 Third assertion 346
13.1.4 Fourth assertion 347
13.2 Publication of results 349

13.2.1 Preliminary publications 349
13.2.2 Publication of results presented in this dissertation 349
13.2.3 Publication of future work's first results 353
13.3 Applicability of the contributions 353
13.4 Future research 355

Bibliography **359**

Index **367**

List of Figures

1.1 Examples of failures of the communication system 2

2.1 Hardware component's mortality curve 17

3.1 CAN base frame format (CAN 2.0 A) of the Data frame 29
3.2 CAN base frame format (CAN 2.0 A) of the Remote frame 29

4.1 Common-mode failure in replicated buses 40
4.2 Spatial-proximity failures in replicated buses 41
4.3 Common-mode failure when using a bus guardian 42
4.4 A useless bus guardian in the presence of a medium failure 42

5.1 Architecture of CANcentrate 51
5.2 Configuration of the transceivers to connect a node to its link . . . 52
5.3 Internal structure of the hub 53
5.4 Hybrid topology combining CANcentrate and CAN 58
5.5 Comparison between cabling lengths in a star and in a bus 59

6.1 Internals of the Enabling/Disabling Unit 65
6.2 Reintegration policy schema of CANcentrate 71

9.1 Hub prototype 116
9.2 Basics of the CANcentrate node prototype 117
9.3 Faulty node prototype 119
9.4 Experimental platform 120

10.1 Bus layouts 135
10.2 Basic internal architecture of the Philips SJA1000 CAN controller 143
10.3 Error-containment capabilities of a CAN controller in a CAN bus . 149
10.4 Error-containment capabilities of a CAN controller in CANcentrate 151
10.5 Basic structure of an entity submodel 156

10.6 Basic structure of a regions submodel 162
10.7 CANcentrate model 168
10.8 nodeKernelsT submodel 169
10.9 nodeConnsT submodel 171
10.10 hubKernel submodel 176
10.11 BranchesFailureEval submodel 177
10.12 CANcentrateFaiEval submodel 178
10.13 CANbus model 180
10.14 nodeKernelsB submodel 181
10.15 nodeConnsB submodel 184
10.16 inBusSections and edBusSections submodels 187
10.17 CANbusFaiEval submodel 189
10.18 NFT/AR vs number of nodes 192
10.19 FT/AR_1 vs number of nodes 193
10.20 FT/AR_1 vs system' fault-tolerance coverage for 3 nodes 195
10.21 FT/AR_1 vs system' fault-tolerance coverage for 15 nodes 196
10.22 FT/AR_1 vs fail-silent node proportion for 3 nodes 199
10.23 FT/AR_1 vs fail-silent node proportion for 15 nodes 200
10.24 FT/AR_1 vs bit-flipping coverage of the hub for 3 nodes 202
10.25 FT/AR_1 vs bit-flipping coverage of the hub for 15 nodes 203
10.26 FT/AR_1 vs bit-flipping proportion for 3 nodes 206
10.27 FT/AR_1 vs bit-flipping proportion for 15 nodes 207
10.28 FT/AR_1 vs ofm proportion for 3 nodes 209
10.29 FT/AR_1 vs ofm proportion for 15 nodes 210
10.30 FT/AR_1 vs cable and connector's failure rate for 3 nodes 213
10.31 FT/AR_1 vs cable and connector's failure rate for 15 nodes 214
10.32 FT/AR_1 vs hub's failure rate for 3 nodes 216
10.33 FT/AR_1 vs hub's failure rate for 15 nodes 218

11.1 Example of a network partition in a replicated star topology 230
11.2 Architecture of ReCANcentrate 232
11.3 Architecture of a ReCANcentrate's node 233
11.4 Possible simplified architecture of a ReCANcentrate's node 234
11.5 New internal structure of the hub 236
11.6 Analogy between ReCANcentrate and CAN with two controllers . 239
11.7 Injection of stuck-at-dominant/bit-flipping faults at links 249
11.8 1st injection of stuck-at-dominant/bit-flipping faults at interlinks . 250
11.9 2nd injection of stuck-at-dominant/bit-flipping faults at interlinks . 251

12.1 ReCANcentrate model 263

12.2 Paths of the coverage process 265
12.3 ReCANcentrateFaiEval submodel 270
12.4 nodeKernelsR submodel 276
12.5 nodeConnsR submodel 283
12.6 hubInConns submodel 289
12.7 hubKernels submodel 294
12.8 fauLPevalAtNode submodel 299
12.9 fauLPsEvalAtHubs submodel 304
12.10 fauLPevalAtHub submodel 309
12.11 fauIPevalAtHubs submodel 312
12.12 fauHubEvalAtHub submodel 317
12.13 fauHubEvalAtNodes submodel 321
12.14 ofmFauEval submodel 327
12.15 NFT/AR vs number of nodes 333
12.16 FT/AR_1 vs number of nodes 334

List of Tables

10.1 FS proportion as a function of the node's error-containment coverages . 198
10.2 20% bit-flipping proportion sensitivity analysis configuration 204
10.3 Attachments' failure rates configuration for 15 nodes 212
10.4 CAN bus, CANcentrate and ReCANcentrate models' common parameters 224
10.5 Parameters specific to the CAN bus model 225
10.6 Parameters specific to the CANcentrate model 226

12.1 Parameters specific to the ReCANcentrate model 339

Chapter 1

Introduction

1.1 Problem statement

The Controller Area Network (CAN) protocol [ISO93] is a field-bus which fulfills the communication requirements of many distributed embedded systems. In particular, CAN provides electrical robustness [FOFF04] and good real-time performance with very low cost. Due to this, the CAN protocol is nowadays used in a wide range of applications, such as in-vehicle communication and factory automation. Furthermore, there is still a high interest in researching new solutions based on the mentioned properties of CAN [PPA+09] [GAW09] [Cav05] [NNH05] [PF04].

However, the use of CAN in applications that require a high level of dependability, e.g. in safety-critical systems, has been controversial due to some dependability limitations [PPA+09] [Fre02]. Some of these limitations are caused by the bus topology it relies on [Kop03] [BKS03]. The main drawback of any protocol using a bus topology is that the structure of the network presents multiple components, i.e. cables, connectors and circuits in nodes, which have direct electrical connections to each other without proper *error containment* [Lap92] [Lap01] [Kop03] [ABST03]. As a consequence, a fault in the bus interface of one node may generate errors that propagate to the remaining nodes and effectively prevent further communication to take place, leading to a global failure of the communication subsystem. This situation is depicted in case A of Figure 1.1 in which a fault in the medium access circuitry of node 2, e.g. with the transmitted bits stuck at a fixed value, blocks the communication channel and none of the nodes can communicate with each other. Similar situations can happen with short-circuits in the bus transmission medium or the connectors.

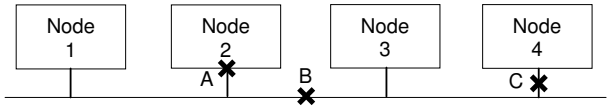

Figure 1.1: Examples of failures of the communication system

Moreover, a bus is shared by all communication paths between every subset of nodes. Consequently, a physical disruption in just one point necessarily destroys many of those paths, thereby dividing the network in several subnetworks or partitions [Kop03]. This is depicted in case B of Figure 1.1 in which a disruption in the bus mid point splits the network into two partitions and blocks any further communication between nodes 1 and 2 with nodes 3 and 4. Moreover, even if both partitions can continue operating independently, i.e. the respective nodes can still communicate with each other, the global communication capabilities may have been substantially reduced.

Finally, case C shows the situation in which there is a partition in the local connection of node 4 with the bus that does not affect the bus integrity and which leaves the inputs of the node's reception port floating. Consequently, node 4 becomes isolated but the communication among the remaining nodes is unaffected. From the communication system point of view, this is the desired behavior when a fault occurs on one node or on one node's bus interface, because it exhibits the least impact on the communication system itself.

It can be deduced from the above discussion that a bus topology presents multiple points where a single fault in any of them generates errors that cannot be contained and that, thus, propagate throughout the communication subsystem. This multiplicity represents a problem, because a higher probability of error propagation should negatively affect the dependability of the overall system. This is even clearer if we notice that each one of those points represents a *single point of failure*, i.e. a point whose failure causes the failure of the overall system. In fact, the presence of a single point of failure could be unacceptable for many highly reliable systems, which raises an additional objection against the use of CAN for these applications. As a consequence, it would be necessary to provide CAN with additional fault-tolerance mechanism aimed at removing all single points of failure from the communication system.

Moreover, note that the lack of adequate error-containment and/or fault-tolerance mechanisms persists when adopting replicated buses, bus guardians or both. Replicated buses are used to tolerate both the propagation of errors through bus replicas and faults happening at those replicas. The use of bus guardians is devoted to

containing errors generated at nodes. More specifically, a bus guardian controls a node's transmissions and blocks them when they are not legal in both the time domain, such as transmissions carried out in the wrong instant, and the value domain, such as transmissions using wrong data. The main problem of replicated buses and bus guardians is that they may suffer from spatial-proximity and common-mode failures. A spatial-proximity failure occurs when a fault affects several components because of their physical proximity. For instance, if a mechanical action destroys a node, it is likely that such action will also partition all the replicated buses to which that node is connected, or that this action also affects the corresponding bus guardian. Similarly, a common-mode failure happens when different components of the system fail in the same way. It can be caused by spatial proximity of components or because different components share the same resources. For example, nothing prevents a faulty node from issuing errors to all bus replicas, thereby blocking any further communication. Another example could be a bus guardian that shares components with the node it controls, such as the clock oscillator or the power supply. In such a situation, a single fault in one of these components will affect the node and its bus guardian equally, so that the bus guardian will likely no longer detect and isolate certain faults.

1.2 The role of star topologies in improving dependability of CAN

Conversely to bus guardians and replicated buses, star topologies might represent an effective solution for containing errors and tolerating faults. In a simplex star topology, each node is connected to a central element or hub by its own link. The most important advantage of this feature is that it is possible to design the hub as an element that has a privileged view of the system, knowing the contributions received from each node through the corresponding links. In this way, the hub can disable any faulty node's contribution in order to contain the corresponding errors at their port of origin. Moreover, a replicated star can be used in order to provide tolerance to hub and link failures. In this sense, an adequate hub can play a key role since it can provide the degree or error-containment needed to prevent errors from propagating from a faulty hub or a faulty link to the rest of the system.

Finally, another important advantage of a star is that links come into spatial proximity only at the star's center, significantly reducing the probability that links suffer from spatial-proximity failures. Moreover, since the hub can be designed to be independent from the nodes, it is hardly possible that the hub and the nodes exhibit spatial-proximity or common-mode failures.

A broader analysis of communication system dependability aspects can be found in [Kop02] concerning a comparison between TTP/C [KG94] and FlexRay [Fle05], two recent field-bus communication technologies proposed for automotive systems. This analysis also discusses the specific issue of network topology and shows that the qualitative benefits of a star topology over a bus are clear. Moreover, [PPA+09], which deals with dependability aspects in the particular case of CAN for automotive systems, addresses the advantages of using star topologies for this technology. In fact, benefits of stars over buses are the reason why the LAN domain has long ago moved to star topologies with Ethernet, a technology that is now extensively used in the industrial automation and large embedded systems domains.

Different simplex star topologies have already been proposed for CAN. However, as will be explained later, although they reduce the impairment provoked by spatial-proximity and/or common-mode failures, none of them takes full advantage of the potential error-containment and fault-tolerance capacities of the star topology. Thus, the general objective of the work herein presented is to improve the dependability of CAN-based systems by means of adequate star topologies.

1.3 The thesis

This document presents the work we have conducted in order to demonstrate the truthfulness of the following thesis:

"It is possible to improve the CAN features related to dependability by means of adequate star topologies. Specifically it is possible to improve error containment of CAN by using an adequate simplex star topology whose hub is provided with adequate mechanisms that contain errors at their ports of origin. Moreover, this improvement can be bigger than the one achieved for CAN with previously proposed simplex star topologies. Thanks to its better error containment, an adequate simplex star topology is suitable to increase the reliability of a CAN-based system that already accepts or tolerates node failures or disconnections. Additionally, it is possible to enhance both error containment and fault tolerance of CAN and of CAN-based simplex star topologies by means of an adequate replicated star topology that includes two hubs that can contain errors at their ports of origin and also errors generated by the other hub, and that includes mechanisms to tolerate faults at links and at one of the hubs. Furthermore, thanks to these improved dependability features, the use of an adequate replicated star topology is appropriate to increase the reliability of both CAN-based systems that do not accept or tolerate node failures or disconnections and CAN-based systems that can do that.".

1.4 Main contributions

1.4.1 The simplex star CANcentrate

Our first main contribution that is aimed at improving dependability of CAN-based systems is the design and implementation of a simplex star topology we call CANcentrate [BPRNA06]. CANcentrate includes an active hub provided with enhanced mechanisms that detect and contain errors generated by faults affecting nodes and/or links. The most innovative feature of CANcentrate is that its hub can contain errors that no other existing star topology for CAN is able to detect. Moreover, it presents neither the implementation limitations nor the incompatibilities with the CAN protocol exhibited by other CAN-based stars.

In fact, two requirements were imposed from the beginning on the CANcentrate's design. First, to ensure a complete compatibility with the CAN protocol. As a consequence of this compatibility with the standard CAN specification, *commercial off-the-shelf* (COTS) CAN components can be used for implementing the nodes of a CANcentrate network. Moreover, CANcentrate can be the communication infrastructure of any CAN-based application and CAN-based protocol. The second requirement was to preserve all the characteristics of the CAN protocol that are related to dependability. Since CANcentrate is fully compatible with CAN, this requirement was accomplished by taking advantage of the dependability-related features already provided by CAN. For instance, particular care was taken to maintain the frame format and all mechanisms for channel error detection and signalling exactly as they are defined in CAN [ISO93].

In order to formalize the error-containment capabilities of a given topology such as the CAN bus or CANcentrate, we define the concept of *k-severe failure of communication* as a situation in which few than $N - k$ (with $k \geq 0$) nodes of the N nodes that constitute the complete system can operate and communicate among them. By extension a *point of k-severe failure of communication* is a point such that a single fault in it may cause a k-severe failure of communication. This definition subsumes the commonly referred to as *single point of failure*, since a single point of failure is a point of k-severe with $k = N$.

In this sense, we can say that a CAN bus presents multiple points of k-severe failure, since the errors generated by a single fault are likely to propagate and thus prevent more than k nodes from communicating. Conversely, notice that in CANcentrate each node is connected to the hub by means of a dedicated link, so that a fault affecting a node or its link can be isolated by disabling only the respective hub port. As a consequence, in CANcentrate, each fault prevents a maximum of one node from communicating, except if the fault affects the hub.

This implies that a CANcentrate star does not include multiple points of k-severe failure, but only one: the hub itself.

The existence of a point of k-severe failure has a negative impact on the system dependability in the sense that, although the system may be resilient to the failure of such a point, it could not deal with a situation in which the errors propagate causing the failure of other parts of the system. However, although the hub still represents a point of k-severe failure, we consider the simplex star topology to be a good choice. Note that a simplex star has one point of k-severe failure, whereas the bus includes multiple of these points. Thus, the star should yield dependability benefits, since it minimizes the number of subsystems that are affected by errors when faults occur. Certainly, the capacity of a system to take profit from this star's advantage depends on its ability to deal with subsystems that are polluted with errors. In particular, a system should obtain greater benefits from the simplex star as the number of subsystems that are not essential for its correct operation increases.

Moreover, as pointed out above, a point of k-severe failure can actually represent a single point of failure. In this sense notice that, in fact, not only the hub, but also all the points of k-severe failure of a bus topology are single points of failure. Therefore, when a fault occurs, it is more likely that an overall failure takes place in a system that relies on a bus topology.

Finally, notice that it is also possible to improve the error containment and thus the system dependability, if the probability of error propagation is reduced by increasing the reliability of the points of k-severe failure. This is because to increase the reliability of a point of k-severe failure is equivalent to decrease the probability with which this point fails and then causes a harmful propagation of errors. The star topology also represents an advantage over the bus in this aspect, since it is easier to improve reliability for just one point of k-severe failure, i.e. the hub, than for the multiple points of k-severe failure in a bus topology.

1.4.2 The replicated star ReCANcentrate

However, it is still possible to further improve the error containment by means of a star topology, if spatial redundancy is used at the hub level, so that the errors provoked by a hub failure are also contained. Therefore, the second main contribution of our work is the design and implementation of an adequate replicated star topology we call ReCANcentrate [BAP05]. As in the case of CANcentrate, this replicated star has been designed to be fully compatible with CAN and to preserve the good features of CAN related to dependability.

ReCANcentrate includes two hubs and each node is normally connected to both of them by means of two dedicated links. ReCANcentrate is provided with mechanisms that are able not only to prevent the propagation of errors generated by faulty nodes and/or links, but also to contain the errors that any of the two hubs may issue through any of its ports. Notice that the fact that a hub issues errors through a given port does not necessarily mean that the hub is completely faulty. A hub can be partially affected by a fault, so that it can correctly deliver its services to some nodes, while sending errors to other nodes.

However, the most innovative feature of ReCANcentrate is that, besides error-containment, it further provides mechanisms to tolerate faults at the communication subsystem level. On the one hand, ReCANcentrate tolerates faults that completely affect one the two hubs (no matter which one) and that prevent all nodes from communicating through it, e.g. faults that lead one of the two hubs to constantly issue errors through all its ports. On the other hand, ReCANcentrate tolerates faults happening at the connections of the nodes to the hubs, so that each node is able to communicate as long as it is connected to a non-faulty hub port.

The ability of ReCANcentrate to tolerate faults at the communication subsystem can yield important benefits in terms of system dependability. Firstly, the capacity of ReCANcentrate to tolerate the complete failure of a hub implies that it definitively eliminates all points of k-severe failure from a CAN network. This is because to tolerate a fault that provokes a complete hub failure is equivalent to contain the errors generated by a fault that, otherwise, will propagate to all nodes. Secondly, to tolerate the failure of one of the connections (no matter which one) of a node to a hub goes beyond the capacity for preventing that other nodes are indirectly affected by the errors associated with such a failure. Conversely, what we additionally obtain by tolerating the failure of these connections is to increase the capacity of each node for communicating when it suffers from faults that directly affect its connections with the rest of the system.

1.4.3 Quantifying the dependability improvement

Nevertheless, despite the good properties of star topologies, no one has appropriately quantified the dependability benefits they achieve when compared with buses. Stars are inherently more resilient to spatial-proximity and common-mode failures, can yield better error containment and, in particular, replicated stars can even provide fault tolerance. However, they also include more hardware components, thereby increasing the probability that faults and errors occur. Therefore, it is necessary to quantify the effect that the above-mentioned advantages of star topologies, e.g. better error-containment, have on the system dependability. More-

over, to quantify the dependability improvement that a star topology can achieve in the context of field-bus communications in general, and in the case of CAN in particular, it is essential to justify the interest of the designs presented in this dissertation.

Therefore, the third main contribution of the work herein presented is the quantification of the benefits that the dependability-related mechanisms of CANcentrate and ReCANcentrate, i.e. their error-containment and fault-tolerance features, yield in terms of specific dependability attributes when they are compared with the CAN bus. To our best knowledge, no one has appropriately quantified the dependability improvement of stars over buses. For instance, no one has taken into account the ability of hubs to contain errors, or the fact that more hardware is normally needed to connect a node to a hub than to connect a node to a bus. Moreover, it is noteworthy that such an appropriate quantification has not been carried out for technologies other than CAN either.

As it will be explained later, dependability is a broad concept that embraces several attributes such as for example the *maintainability*, *availability* and *reliability*. A star topology can yield benefits in terms of those and other dependability attributes. For example, as concerns *maintainability*, the hub can be designed to provide information about what ports have been diagnosed as faulty, thereby facilitating the work of the maintenance personnel. Moreover, the information provided by the hub concerning the location of faults can reduce the time needed to repair the system. Thus, a star topology can also yield benefits in terms of system *availability*, since short repair times increase the probability with which the system is ready to deliver its service at a given instant of time.

However, notice that it is not always possible to have access to a distributed embedded control system in order to carry out repair activities. Moreover, sometimes it is not even acceptable that a system momentarily interrupts its service. The most important dependability attribute in those cases is the *reliability*, which stands for the probability with which a system continuously delivers its intended service during a given interval of time.

The benefits that the error-containment capabilities of a star can yield in terms of reliability depend on the ability of the system to correctly deliver its service when only a subset of nodes can still operate and communicate among them. This was already pointed out before, when we discussed about the potential advantages that a system can obtain from the fact that a star topology eliminates the multiple points of k-severe failure present in a bus. At this point, notice that to contain errors in a star implies the disconnection of a given hub port and, thus, of the corresponding node. Hence, error containment becomes useless if the system is not able to accept

or tolerate that, in order to prevent the propagation of errors, at least one node is isolated. The same happens if the system does not accept or tolerate the failure of at least one node. In general, to isolate M nodes in order to contain errors is useless if the system does not accept or tolerate the isolation or the failure of those M nodes.

Certainly, there are systems that do not accept the failure of any node and that require that all nodes communicate with each other. Because of the reasons just explained above, error containment is useless for them and, thus, the mechanisms of CANcentrate (or of any simplex star topology) intuitively do not represent any advantage for those systems in terms of reliability when compared with the CAN bus. In fact, it is expected that a simplex star topology reduces the reliability of those system since, in general, it needs more hardware than a bus to interconnect a given ensemble of nodes, e.g. the hub and extra cabling, and thus, it is expected that faults are more likely to occur in the star.

Nevertheless, as concerns ReCANcentrate, it is not a priori so evident whether it reduces or improves the reliability of those systems. This is because, besides error containment, ReCANcentrate also provides fault-tolerance mechanisms that could compensate its extra hardware, and that could even increase the probability with which all nodes are able to communicate with each other. For instance, imagine a ReCANcentrate star and a CAN bus where each node is connected to both hubs by means of two dedicated links, and where a CAN node is connected to the bus just by means of a single stub. Since a link is generally larger and includes more wire than a stub, the probability of failure of the former is bigger. Then, a situation in which a node loses its connection to one of the hubs is more likely than a situation in which a node loses its connection to the bus. However, a ReCANcentrate node that suffers from a fault in one of its two links can still communicate using its remaining non-faulty link, whereas a CAN node is no longer able to communicate if its stub fails.

Anyway, although it seems quite clear that a better error containment does not improve reliability in the above-mentioned systems, it is not so obvious that it cannot increase the reliability of systems that can continue delivering their services as long as a minimum number of nodes can still operate and communicate among them. In fact, it is possible that error containment could significantly improve the reliability of these systems. To better understand this issue, image two systems, one that does not include adequate error-containment mechanisms and another one that does include those mechanisms, but at the expense of being more prone to faults. Notice that a single fault can be enough to provoke the failure of the first system, because the errors that fault generates are not correctly contained and, as a consequence, they can affect too many nodes. Conversely, in the second system,

the number of nodes that are affected by the errors a fault generates is minimized and, hence, more than one fault are needed to affect too many nodes and lead to a system failure. This ability of the second system to minimize the impact of faults is very valuable, because the occurrence of multiple faults is much less probable than the occurrence of just one. Therefore, although the second system is more prone to faults, its probability of failure can be significantly lower than the probability of failure of the first system.

Fortunately, there are plenty of systems that can just accept the failure or the disconnection of some nodes and, then, deliver their services in a degraded mode. An example could be a factory plant with N automated production lines working in parallel, in which it can be accepted that up to k production lines are inoperative, as long as a minimum throughput is guaranteed. Another example is the intra-building communication system of a hotel or a house, whose main objective is to provide service to the maximum number of rooms, even when faults occur.

Moreover, there are also examples of systems that can fully deliver their services, while tolerating the loss of up to k of N nodes. This is the case of highly reliable (safety-critical) systems that use mechanisms to tolerate node failures. Fault tolerance is normally achieved in those systems in a systematic way by means of redundancy, either of hardware, software, information or time. One typical way of introducing such a redundancy is to include several circuits that provide the same service in parallel for the system to be able to continue its operation despite the failure of a specific maximum number of these redundant circuits. Each one of these redundant circuits is called a replica. Error containment becomes thus fundamental to guarantee fault independence among replicas in those systems. Otherwise, errors generated by a fault can propagate to several replicas, thereby provoking their failure and possibly the failure of the whole system. An example of this kind of systems can be the air-conditioning control system of an aircraft, which extensively uses redundancy to tolerate node, actuator and sensor failures. Similar examples could be x-by-wire control applications, which are devoted to substituting the mechanical and hydraulic control mechanisms in vehicles by an electronic control system.

Among all the dependability attributes, the quantitative evaluation presented in this work focuses on the *reliability*. Notice that although CAN has been extensively used in a broad range of applications, including highly reliable ones such as aerospace systems, e.g. [KHJN03], the level of reliability required by newer applications is continuously increasing. Those new reliability requirements have raised considerable uncertainties about the suitability of CAN for those systems, as pointed out before. This situation is clear in the avionics and the automotive industry domain, for which alternative highly reliable protocols have been recently

developed, e.g. TTP/C [KG94] and FlexRay [Fle05], in order to complement or compete with CAN (as can be seen in [HR09] or [DZY09] for instance). In fact, those newer protocols have also adopted star topologies, given the reliability benefits that stars are supposed to provide.

However, this interest in reliability does not mean that this work is exclusively devoted to highly reliable and fault tolerant systems. Conversely, error containment and reliability are desirable for many other applications, as mentioned above. In fact, non highly reliable applications are also demanding increasing levels of error containment and reliability, since the requirements in number of nodes and services are also growing. This can be seen, for instance, in the automotive or in the home automation domains, where comfort or entertainment features are gaining in importance.

1.5 Organization of the document

The rest of the document is organized as follows. Firstly, Chapters 2 and 3 respectively give an overview on the dependability concepts and on the CAN protocol features that are relevant to this dissertation.

Chapter 4 briefly describes previous solutions proposed in the literature to improve dependability of CAN networks, and highlights their main pros and cons. Then, Chapter 5 presents CANcentrate, explains its main characteristics and discusses its performance when compared with CAN. The internal mechanisms of the CANcentrate hub to deal with errors and faults are thoroughly explained in Chapter 6.

Since the detection and containment of errors generated by a specific type of fault, called *bit-flipping fault*, is particularly complex, the two next chapters are dedicated to this issue. Specifically, Chapter 7 deeply describes the mechanisms the hub of CANcentrate includes for detecting bit-flipping errors, whereas Chapter 8 further analyzes the pros and cons of the hub mechanisms for both detecting bit-flipping errors and diagnosing bit-flipping faults, and proposes some further enhancements.

After the explanation of the CANcentrate design, Chapter 9 describes its first prototype implementation. It also shows some experiments we conducted using this prototype in order to evaluate the mechanisms CANcentrate includes to deal with errors and faults, as well as to measure its performance. Finally, Chapter 10 describes the work we carried out in order to quantitatively evaluate the improvement of reliability achieved when using CANcentrate instead of CAN.

As concerns ReCANcentrate, Chapter 11 describes its main characteristics, focusing on the mechanisms it includes to tolerate faults, as well as on its main differences when compared with CANcentrate. This chapter also presents a first prototype implementation of ReCANcentrate and the experiments we carried out to check its main error-containment and fault-tolerance capabilities and to measure its performance. Afterwards, Chapter 12 thoroughly explains the reliability models of ReCANcentrate and quantitatively compares the reliability achievable by equivalent systems relying on CAN, CANcentrate and ReCANcentrate.

Finally, Chapter 13 summarizes the work presented in this document, explains how the thesis has been validated, highlights the main contributions of this dissertation and their practical applicability, specifies the publications that resulted from the work herein presented, and proposes future work.

Chapter 2

An overview on dependability

2.1 Introduction

As already explained, the work presented in this dissertation is devoted to improving both error containment and fault tolerance of CAN networks by means of adequate star topologies. This better error containment and fault tolerance can yield benefits in terms of some dependability attributes, such as reliability.

However, dependability embraces other attributes and it is related to a wide range of different concepts. Thus, although in previous chapter we briefly outlined some of these concepts that give an idea about our objectives and how we are going to achieve them, it is still necessary to further clarify some aspects to precisely understand the work herein presented.

This section is devoted to providing an overview on these concepts. Particularly, we adopted the terminology proposed by Laprie in [Lap92] [Lap01], which is widely accepted by the research community that works on dependability issues.

2.2 Basic concepts and terminology

It is quite difficult to find a concise definition that reflects the meaning of *dependability*. Maybe a good definition of it can be the one provided in [Lap92] [Lap01]:

"Dependability is defined as the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers [Car82]".

This definition tries to encompass the attributes that constitute dependability: *reliability*, *availability*, *safety*, *security*, *performability*, *maintainability* and *testabil-*

ity. In this document we focus on the attribute of *reliability*, which can be defined as the ability of a system to continuously deliver its intended service throughout a given interval of time [Sho02].

Note that reliability differs from other dependability attributes. For instance, *availability*, which stands for the ability of a system for being ready to deliver its service at a given instant of time [Sho02], does not require the system to uninterruptedly provide its service. Another concept different from reliability is the *performability*, which is a measurement of how well a system operates during an interval of time in the presence of faults [Sho02]. In this sense, note that a system can still operate in a degraded but acceptable way in the presence of faults. However, a system that operates in a degraded mode cannot bet always considered as delivering its intended service.

There are also several impairments to dependability and it is necessary to clearly understand how they are related to each other. These impairments are *faults*, *errors*, and *failures*, and there exists a cause-effect relationship between them. A *fault* is a defect in the behavior of a system or in the way the system is designed or built. A fault can generate an *error* (or errors), which is an incorrect result delivered by the system. Finally, an error can lead to a *failure*, which implies that the comportment of the system deviates from the service it is intended to. A system that *presents a failure* is said to be *faulty*, whereas a system that *does not present a failure* is *correct* or *non-faulty*.

Notice that if we decompose a system into different subsystems or components, then the failure of a given subsystem can be considered as a fault from the point of view of the overall system. In this way, a faulty subsystem represents a fault that can generate errors that lead the entire system to fail. Moreover, if a entire system S_A forms part of a bigger system S_B, then the failure of S_A can be considered as a fault in S_B and so on. This is the reason why sometimes it is not so easy to differentiate between a fault and a failure. In fact, to talk about one or the other depends on whether we are considering a subsystem or a whole system, respectively.

A fault can be *active* or *dormant* [Lap92] [Lap01], meaning that it generates errors or it does not respectively. Depending on the frequency with which a fault cycles between the active and the dormant states, it can be classified as *permanent*, *intermittent* or *transient*[1]. A *permanent fault* is always active and constantly generates errors. In contrast an *intermittent* and a *transient fault* cycle between the active and the dormant states. The difference between these two last types of faults is that the former cycles quickly when compared with the mechanisms devised to handle

[1]This notation actually corresponds to the one used in [DT89] for classifying errors depending on their cycling between the effective and the latent states

the errors it generates, whereas the second one does not. In addition, a *transient fault* can also be a fault that is the consequence of a temporary external influence.

At this point, it is necessary to make some brief remarks about how faults can be characterized by means of reliability measures. First, note that the time that elapses until a system fails is normally referred to as the *Time To Failure* of that system (TTF). More specifically, if X is a random variable that corresponds to the TTF of a system, then $F(t)$ is called the *Cumulative Distribution Function* of X (or the *Time To Failure distribution*) and can be expressed as:

$$F(t) = \textit{Probability}(X \leq t)$$

where F(t) has the following properties:

$$\begin{aligned} &F(t) = 0 \text{ for } t < 0 \\ &0 \leq F(t_1) \leq F(t_2) \text{ if } t_1 < t_2 \\ &lim_{t\to\infty} F(t) = 1 \end{aligned}$$

If we focus on the probability of failure within an infinitesimal interval of time, then we can write $F(t)$ in terms of its *probability density function*, $f(t)$, as:

$$\textit{F(t)} = \int_0^t f(\text{t'})\, d\text{t'}$$

Another important function that is used to characterize faults is the *reliability function* of the system, $R(t)$, i.e. the function that expresses the probability that the system correctly operates throughout the interval of time $[0, t]$. Note that $R(t)$ is a continuous monotonic nonincreasing function that is defined in the interval $[0, \infty)$ and whose values range between 0 and 1. Moreover, it is assumed that $lim_{t\to\infty} R(t) = 0$, so that we can write $R(t)$ as:

$$\textit{R(t)} = \int_t^\infty f(\text{t'})\, d\text{t'} = 1 - F(t)$$

The functions $R(t)$ and $f(t)$ allow defining an important measure of reliability called *failure rate*. The failure rate of a system is the instantaneous probability with which the system fails, conditioned by the fact that the system has not failed so far. It can be written as:

$$\lambda(t) = \frac{f(t)}{R(t)}$$

It is worth highlighting the difference between $f(t)$ and $\lambda(t)$. $f(t)\Delta t$ is the unconditional probability with which a system fails within the interval $[t, t + \Delta t]$; whereas $\lambda(t)\Delta t$ is the conditional probability that the system fails within the interval $[t, t + \Delta t]$, given that it has not failed up to t.

$\lambda(t)$, which is also referred to as the *hazard rate* or the *age-dependent failure rate* [STP96][2], is one of the mostly used measures to analyze the reliability of hardware components, such as electronic ones. Thus, this measure is of paramount importance in the context of this dissertation since, as will be explained later in Chapters 10 and 12, we analyze the reliability of a system that relies on a CAN bus, CANcentrate and ReCANcentrate exclusively considering permanent hardware faults.

If we study the failure rate of a hardware component as a function of time, we obtain what is called the *mortality curve* [STP96]. Figure 2.1 depicts the typical shape of this curve, which is based on empirical experiences. As can be seen in this figure, the curve is divided into three regions. The *infant mortality period* is the extent of time during which the component may exhibit faults related to intrinsic imperfections or defects normally due to the manufacturing processes. This period is characterized by a high failure rate that rapidly decreases with time. The second period is known as the *steady-state operation period*. It is the largest period and it is characterized by a low an almost constant failure rate. Faults occur in this period normally as a consequence of external conditions. The last period is called the *wearout period* and corresponds to the last stage of the component's life, where the failure rate monotonically increases with time. Nevertheless, notice that the characteristics of the wearout period are based on the experience with mechanical components, which start to wear with age. In contrast, data concerning electronic devices that has been acquired during many years indicate that the failure rate of an electronic component is not expected to exhibit the properties of this third period, but to remain low and almost constant during all its life [KNM90].

Besides those remarks about different reliability measures, it is necessary to discuss the effect of faults (either permanent, intermittent or transient) on the service delivered by a system or a subsystem. In this sense, notice that the way in which a failure manifests is commonly referred to as its *failure mode*. For instance, a faulty node in a network can simply stop communicating, or it can fail by corrupting the

[2]There are authors that distinguish between the concepts of *hazard rate* and *failure rate* [KNM90]. Specifically, they consider that the former is devoted to characterizing components' failures in non-maintainable systems, whereas the *failure rate* aims at characterizing these failure in maintainable ones. In this sense, what we understand as *failure rate* in the context of this dissertation corresponds to the concept of *hazard rate* in [KNM90]. However, these two concepts are numerically equivalent under specific conditions. For more details concerning this issue refer to [KNM90]

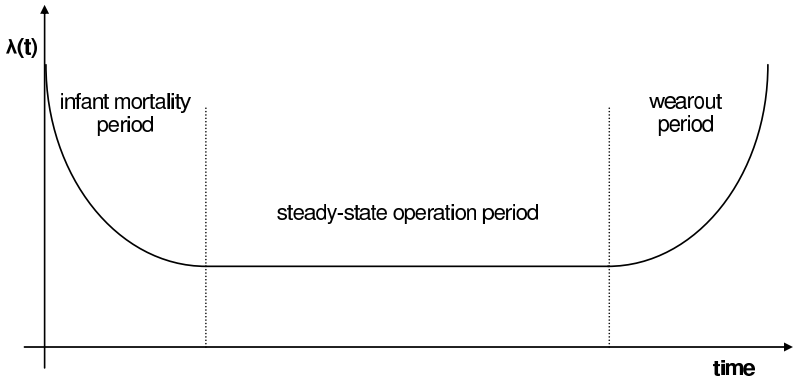

Figure 2.1: Hardware component's mortality curve

data conveyed through the medium. For practical reasons, it is common to classify the failure modes following the hierarchy specified in [Pol96]. This hierarchy specifies the following types of failures.

- **Byzantine or arbitrary failures**. A system that suffers from this failure mode exhibits an incorrect behavior with no restriction neither in the value nor in the time domain. This type of failure is also referred to as *fail-uncontrolled* or even *malicious*. An arbitrary failure can be malicious in the sense that the system may exhibit a "two-faced" behavior or an *inconsistent failure*, delivering different results to different users, so that users have a different perception of the service delivered by the system. Moreover, an arbitrary failure can also be considered as malicious when it leads the system to deliver forged results or messages that belong to other systems, i.e. when the system exhibits a *masquerading failure*.

- **Authentification detectable Byzantine failures**. A system that presents this failure mode behaves in a *fail-uncontrolled* manner, but with one restriction: it cannot forge results or messages that are calculated or delivered by other systems. In other words, this failure mode is equivalent to the Byzantine failure mode, but it cannot provoke that the system impersonates another one.

- **Incorrect computation failures**. When a system shows this failure mode, it fails by issuing incorrect values either in the value or in the time domain. This failure mode is actually not included in [Pol96], but introduced in [BMD93] as *incorrect computation faults*.

- **Performance failures**. A system that suffers from this type of failure delivers results that are correct at the value domain, but that are incorrect in the time domain. In other words, the system delivers results that are either too early or too late. This failure mode is also referred to as *timing failures* in [BMD93].

- **Omission failures**. A system exhibits an omission failure when it does not deliver a requested result, i.e. when it delivers an expected result with an infinite delay. Notice that an omission failure does not necessarily imply that the system permanently stops delivering its service. In this sense, the system may respond or not when it is requested for something.

- **Crash failures**. A system *crashes* or *is crashed* when it permanently stops delivering its service. Hence, it can be considered that a crashed system omits any further requested service.

- **Stopping (Fail-stop) failures**. When a system suffer from a *stopping failure* it exhibits a special case of a crash failure. Specifically, it permanently issues the same result or value to any request, e.g. the last value that was correctly delivered.

Each one of these categories specifies a set of failure modes with different degrees of harshness. In this way, Byzantine and stopping failures are the *least benevolent* [Pro07] and the *most benevolent* ones, respectively. Also notice that a given category includes all the failure modes of the categories that are below in the hierarchy, e.g. performance failures comprise omission, crash and stopping failures.

This categorization is useful when designing a dependable system, since it is necessary to specify what failure modes the system and its subsystems are going to exhibit. On the one hand, the designers of the system must know what will be the behavior of the system when it fails, in order to devise the mechanisms that are necessary to deal with such a situation. For instance, in an ultra-reliable system embedded in a vehicle, it could be necessary to use a mechanical back-up system that takes over the control from the electronic system when the electronic system completely fails in a specific manner. On the other hand, as concerns the subsystems and components that constitute the system, to know the ways in which they fail is essential, because the mechanisms a system must include to address these faults, e.g. for detecting them, depend on the way in which they manifest.

The failure modes a system or a subsystem can exhibit are generally referred to as its *failure semantics*. A more formal definition of this concept can be found in [Pol96]:

"A system exhibits a given failure semantics if the probability of failure modes which are not covered by the failure semantics is sufficiently low".

Notice that the definition of failure semantics includes the concept of *coverage*, which is commonly known as *assumption coverage* [Pol96] or *failure mode assumption coverage* [Pow92]. This concept is formalized in [Pow92] as the probability that the specific way(s) in which a system or a subsystem is supposed to fail proves to be true in practice, conditioned on the fact that the failure actually occurs. This probability is a main concern when devising a dependable system, since a subsystem that fails in a manner that is not covered, i.e. that violates the failure mode assumptions, is considered as provoking the failure of the overall system.

In fact, to help to fulfil the application requirements, one of the first steps carried out when designing a dependable system is to gather all the failure modes of its components into a so called *fault model*. In this sense, a fault that violates the failure mode assumptions is said to be *out of the fault model*.

Finally, notice that a great effort is normally carried out to *restrict the failure semantics* of the subsystems that compose a whole system. If this is done, it is possible to simplify the mechanisms that will allow non-faulty subsystems to continue operating when other subsystems fail. For instance, in fault-tolerant distributed control systems, it is very common to devise mechanisms to enforce each node to exhibit a *fail-silent* behavior, i.e. to ensure that each node can only suffer from a *crash failure*.

2.3 Fault tolerance basics

At this point, one may ask what are the mechanisms that can be used to provide dependability. Certainly, there are different ways to achieve it: *fault prevention*, *fault tolerance*, *fault forecasting* and *fault removal* [Pro07]. Among them, this dissertation focuses on *fault tolerance*, i.e. on methods used to allow a system to deliver its service even in the presence of faults.

Fault tolerance is carried out in two phases: *error processing* and *fault treatment* [AL81]. *Error processing* is intended to eliminate errors from the system, whereas *fault treatment* aims at preventing faults from generating errors again. Among the different ways that can be used to carry out error processing, this document addresses two of them called *error detection* and *error recovery*. *Error detection* is performed in order to discover errors, whereas *error recovery* is used to lead the system to an operational state that does not present errors.

Regarding *fault treatment*, it is performed by means of *fault diagnosis* and *fault*

passivation. *Fault diagnosis* aims at finding out the fault that generates the errors, whereas *fault passivation* aims at preventing a fault from causing errors again. For instance, the hubs we propose in this dissertation passivate a fault happening at a node or at a link by disabling the corresponding hub port, which prevents this fault from causing errors in the communication among the other nodes.

These two phases of fault tolerance, i.e. error processing and fault treatment, can be carried out following two different strategies: *application-specific fault tolerance* and *systematic fault tolerance* [Pol96]. The first one consists in using the knowledge about the characteristics of the system to detect faults and to tolerate them. In the second one, *redundancy* is used to detect faults when different *replicas* disagree on a given state or result, as well as to continue delivering the intended service using only non-faulty replicas.

The designs presented in this dissertation use both application-specific and systematic fault tolerance. For instance, as will be explained later in Chapter 7, the hub of our star topologies detects errors at any of its ports using its knowledge about how nodes should behave according to the CAN protocol. As concerns systematic fault tolerance, notice that the replicated star topology we propose (Chapter 11) includes two hubs in order to tolerate the failure of one of them. Notice that in this last example, redundancy is used to eliminate the *single point of failure* the hub represents. As already pointed out, a *single point of failure* is a single subsystem or component whose failure provokes the failure of the overall system. For obvious reasons, the presence of a single point of failure is normally not accepted in a fault-tolerant system.

Redundancy is typically the preferred alternative to provide final fault tolerance, since to deploy redundancy is much more simple and more effective than to exploit the knowledge about some properties of the system. There are different types of redundancy, but all them can be classified as *spatial* or *temporal redundancy*. The first one consists in introducing extra hardware, software or even extra information, e.g. a cyclic redundancy code in a message; whereas the second one consists in using extra time to perform a given function in order to enable error detection and fault tolerance. Independently of the type of redundancy used, it is necessary to provide the system with the appropriate *redundancy management* mechanisms. These are indispensable for coordinating the different replicas either in the presence or in the absence of faults.

An important aspect that must be taken into account to deploy redundancy is whether it is needed to deal with *related faults* or only with *independent faults*. Related faults are those that are due to the same cause, whereas independent faults do not have the same origin. Related faults may occur either when different subsys-

tems are close to each other, so that an external cause affects some of them at the same time, or when different subsystems are equally affected by a fault happening in a resource they share. When failures happen as a consequence of physical proximity they are referred to as *spatial-proximity failures*. In addition, if related faults lead the affected subsystems to exhibit the same failure mode, then it is said that *common-mode failures* occurs.

It is necessary to guarantee as much as possible that different replicas cannot exhibit related faults, i.e. it is vital to provide *fault independence*. Otherwise, redundancy becomes useless. For example, if replicas are used to detect faults by comparing their outputs, then, it is necessary to ensure that they will not exhibit a common-mode failure that lead them to deliver the same erroneous result. This could be accomplished, for instance, by preventing them from sharing the same resources, e.g. a power supply.

However, although replicas can be designed to ensure that they independently fail, errors generated by faults in other subsystems may propagate and affect different replicas. In fact, the propagation of errors is one of the main problems to be solved when designing a fault-tolerant system, since errors can corrupt several subsystems, thereby leading to a global failure. Thus, it is mandatory to provide the system with *error-containment* mechanisms, i.e. mechanisms to prevent errors from propagating from the faulty components to the rest of the system. In this sense, what is normally done is to divide the system into several *error-containment regions* [Kop97] or *fault-containment regions*, i.e. regions that can be isolated to prevent error propagation. Notice that in this dissertation we propose to contain errors by disabling the corresponding hub ports. This means that each node and its corresponding link represent an error-containment region.

A particular case of error propagation that provokes a system failure may happen in the presence of Byzantine failures that lead different subsystems to receive *inconsistent data*, i.e. to receive different data when it is supposed that all them receive the same. For instance, a node that is the responsible for indicating the other nodes in which operational mode the overall system should behave may fail by specifying different modes to different nodes.

In order to overcome the problems that inconsistencies pose, it is necessary to devise some kind of mechanisms that enforce an *agreement* on the received data. In the particular case of CAN, this agreement is supposed to be enforced at the communication level. Specifically, it is said that CAN presents the so called *data consistency* property. This property ensures that every frame is quasi-simultaneously accepted by all nodes or by none of them [ISO93]. The mechanisms CAN uses to try to provide data consistency will be further addressed in Section 3.3.4.

Finally, notice that any mechanism devised to process or contain errors, as well as to treat faults cannot be perfect. For instance, if an error-detection mechanism must deal with a huge range of different scenarios involving errors, it is actually impossible to guarantee that all errors will be always detected. Therefore, when designing and evaluating a dependable system, it is necessary to take into account the *coverages* of its fault-tolerance mechanisms. Furthermore, these coverages are of capital interest since the dependability of a system strongly depends on their value. This strong influence have long ago been demonstrated or highlighted by many authors, e.g. [BCS69] [Arn73] [DT89].

Notice that the mentioned coverages are, in fact, an abstraction of the probability of success of the different processes that constitute the fault-tolerance mechanisms of a given system. Thus, the specific coverages to be considered depend on the system itself and on the level of abstraction used to design and to evaluate it. More specifically, the coverages depend on the number of phases the process of tolerating a fault is supposed to be divided into. For instance, [Pow92] considers just one coverage called *coverage of the error-processing mechanisms*; whereas [DT89] compares system abstractions from different authors, each one considering its own set of coverages, e.g. error-detection, error-location and error-recovery coverages. It is important not to confuse these coverages with the *failure mode assumption coverage*, which we explained above.

2.4 Fundamentals of fault-tolerant systems design

Besides the approach used to provide fault tolerance, application-specific and/or systematic, the development of a complex fault-tolerant system requires the use of a "systematic" and practical strategy. In our case, we roughly followed the paradigm that Prof. Avižienis proposed for developing a fault tolerant system [Avi95]. This paradigm divides the tasks carried out to build a fault-tolerant system into three activities: *specification*, *design* and *evaluation*.

Specification activities basically consist in determining the functional and dependability requirements of the system, e.g. the fault model, the degree of desired reliability, etc.

Design activities embrace different tasks aimed at defining the system architecture: what subsystems compose the overall system and how they interact with each other; the structure and functionalities of each subsystem, e.g. how each subsystem performs error containment; and the integration of all subsystems.

Finally, evaluation activities are performed during the different phases of the

design process for two purposes. First, to guide the design process. Second, to assess whether or not the designed system or subsystems fulfill their functional and dependability requirements.

There are two types of evaluation: *qualitative evaluation* and *quantitative evaluation*. The first one is devoted to verifying that the system is able to deal with all the faults included in its fault model; whereas the second one aims at numerically corroborating that the system fulfills its dependability requirements, e.g. its intended reliability.

There are different techniques to perform both types of evaluation. *Model checking* techniques [CGP99], for instance, can be suitable for qualitatively evaluating intricate parts of a system. More specifically, model checking allows formally verifying system properties. The first step consist in building a model of the system, typically in the form of a set of interconnected automatons. Then, the user asks, by means of queries, whether or not the modelled system fulfils certain properties. Finally, a software tool, called *model checker*, exhaustively analyzes all the possible states of the model and determines whether or not each property/query holds.

Quantitative evaluation techniques usually also rely on the specification of a model of the system. Different formalisms, such as for example *Markov Chains* or *Petri Nets* [TMGT93] [Pet81], can be used for this purpose. These models normally include a considerable amount of parameters, some of which have a great impact on the final results. For instance, as already said, this is the case of the coverages of the system's fault-tolerance mechanisms.

Different alternatives are available to estimate those parameters, e.g. simulations, experimentation with prototypes, etc. For the case of the coverage, it is very common to build a prototype and then to evaluate its behavior in the presence of faults that are introduced in the system by means of *fault injection* techniques [ACL89] [GKT89].

2.5 Dependability concepts introduced for this work

Before concluding this chapter about dependability aspects, let us further discuss some concepts we have introduced for the work herein presented.

As said above, errors generated by a faulty subsystem may propagate to other subsystems and even provoke a global failure. In particular, if we consider a CAN node as a subsystem, then the errors issued by a faulty node can prevent other nodes from communicating. Thus, to contain errors in a network such as CAN is particularly important for systems that can accept the failure of some nodes as long

as a minimum set of non-faulty nodes can communicate with each other.

In order to characterize the dependability of a network that can be suitable for those systems, we defined the concepts of *k-severe failure of communication* and *point of k-severe failure of communication*. If we consider a network composed of N nodes, then a *k-severe failure of communication* occurs when few than $N - k$ nodes can operate and communicate with each other. A *point of k-severe failure of communication* is a component or a subsystem whose failure provokes a k-severe failure of communication. For $k = 0$, a *point of k-severe failure of communication* corresponds to the concept of a single point of failure in the context of a communication network. For the sake of conciseness, we will normally use the terms *severe failure* and *severe point of failure* instead of *k-severe failure of communication* and *point of k-severe failure of communication* respectively.

2.6 Conclusions

This chapter gives an overview on some dependability concepts that are essential to understand the work presented in this dissertation.

Firstly, we provided a definition of dependability and of some of its attributes, such as reliability. We focused on the cause-effect relationship between faults, errors and failures, and we discussed the concepts of failure mode, failure semantics, failure mode assumption coverage and fault model.

Then, we briefly described some fault-tolerance basics. Firstly, we addressed the phases of fault tolerance (error processing and fault treatment), focusing on the concepts of fault passivation and redundancy. In particular, we pointed out that we chose redundancy as the approach to provide final fault tolerance, and we presented in few words the different types of redundancy. Secondly, we addressed the concepts of independence of faults and we introduced the problem of dealing with spatial-proximity and common-mode failures. We also put special emphasis on the problem of the error propagation, as well as on the need of error-containment mechanisms. Thirdly, we presented the problem of inconsistencies in the presence of Byzantine failures, as well as the necessity for agreement mechanisms such as protocols that provide data consistency. Finally, we highlighted the importance of the coverages of the fault-tolerance mechanisms and their impact on the system's dependability.

After this overview on fault tolerance, we outlined the paradigm that Prof. Avi$\check{z}$ienis proposed for developing fault tolerant systems. We briefly explained each phase of this paradigm, pointing out some evaluation techniques such as model

checking, Petri Net formalisms and fault injection.

Finally, we defined the concepts of k-severe failure and point of k-severe failure, which are specific to the work herein presented and that will be used throughout this dissertation.

Chapter 3

Controller Area Network (CAN) protocol

3.1 Introduction

The Controller Area Network (CAN) protocol is a field-bus communication technology created in the early 80s by Bosch GmbH in Germany. Its main advantages are its low cost, simple configuration, electric robustness, prioritized medium access arbitration mechanism, as well as error-detection and containment features. Thus, although CAN was initially aimed to reduce the wiring cost in in-vehicle communications, soon after it became extremely popular in other distributed embedded control systems. Nowadays it is widely used as the communication infrastructure of a wide range of applications such as factory automation, robotics, intra-building communication, medical equipment, etc.

CAN comprises the *Physical Layer* and the *Data Link Layer* [ISO03a] of the ISO *Open Systems Interconnection Basic* (OSI) *Reference Model*. Its data link layer was firstly specified by Bosch in 1991 [Gmb91]. Afterwards, ISO standardized both layers in 1993 [ISO93] and updated that specification in 2003 [ISO03a]. Moreover, due to increased interest in using CAN, other physical layers standards have also been specified.

This chapter summarizes the aspects of the CAN physical and data link layers that are more relevant to this document. Additionally, it specifies the types of faults that can affect a CAN network, some of which are the aim of the solutions we propose in this document for improving its dependability.

3.2 CAN Physical Layer

There exists several CAN physical layers specifications. The most relevant ones are: ISO 11898-2 (high-speed) [ISO03b], ISO 11898-3 (fault-tolerant), SAE J2411 (single-wire) and ISO 11992 (point-to-point). Among them, maybe the most popular is the ISO 11898-2 (high-speed); hence this document only addresses this standard.

In compliance with this standard, a CAN network relies on a simplex bus topology whose medium is constituted by a two-wire differential line that is specially resistant to *Electromagnetic Interference* (EMI). Additionally, in order to prevent signal reflections, the bus is terminated at both its ends with impedances of 120 Ohm and its stub lines are configured as short as possible.

One of the most important features of the medium of CAN is that it implements a wired-AND function of every node contribution. This is the basis of the dominant / recessive transmission property of CAN. This property guarantees that whenever one of the nodes transmits a bit with a dominant value, i.e. a logical '0', this value is received by all the nodes in the network. In contrast, a bit with the recessive value, i.e. a logical '1' is only received as long as every node issues a recessive value.

Moreover, CAN communication relies on a complex *bit synchronization mechanism* which guarantees that nodes have a quasi-simultaneous view of every single bit on the channel, i.e. the so-called *in-bit response*. This mechanism uses the recessive to dominant transitions of the signal on the channel in order to keep the nodes of the network synchronized with respect to the node that is transmitting (the so-called *leading transmitter*).

Notice that the in-bit response property implies that the transmission of a bit traverses all the network and electrically stabilizes and only then the next bit can be transmitted. Thus, the bit synchronization of CAN forces an inverse relationship between the bit rate and the achievable bus length. For example, if a CAN network operates at its higher bit rate, i.e. at 1 Mbit/s, the maximum achievable bus length is around 40 m [CiAa]. In contrast, a CAN network operating at 125 Kbit/s or at 10 Kbit/s can respectively achieve a bus length of 500 m and 5 Km [CiAa].

Although the bit synchronization of CAN limits the maximum bus length and/or the bit rate of the network, at the same time it allows definition of a number of additional mechanisms we will describe later on, e.g. *bit-wise arbitration*, which yield important benefits in terms of dependability and real-time.

3.3 CAN Data Link Layer

The data link layer of CAN provides a set of mechanisms that allow nodes to correctly exchange data even in the presence of errors and/or some permanent faults. Next, we present a brief overview on these mechanisms.

3.3.1 Frame format

CAN includes four types of frames: *data frames*, *remote frames*, *error frames* and *overload frames*. Current section describes the format of the two first ones. The format of the error frame and the overload frame are respectively explained in Sections 3.3.4 and 3.3.6.

A node uses a data frame to transmit data, whereas it uses a remote frame to request data from other node. The appearance of both types of frames is almost the same, as shown in Figures 3.1 and 3.2. More specifically, these figures show the *CAN base frame format* (CAN 2.0 A) for both types of frames. However, notice that it also exists another frame format called *CAN extended frame format* (CAN 2.0 B). The only difference between CAN 2.0 A and CAN 2.0 B is that in the former one the identifier is constituted by 11 bits, whereas in the second one the identifier is 23 bits long. This document only deals with CAN 2.0 A, but all the ideas herein presented are applicable to CAN 2.0 B as well.

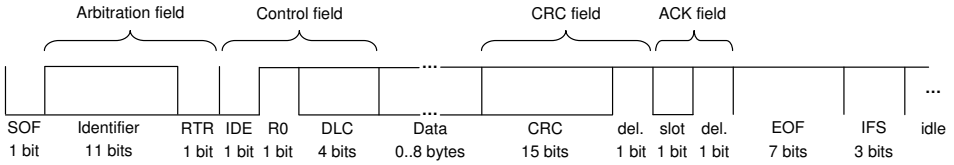

Figure 3.1: CAN base frame format (CAN 2.0 A) of the Data frame

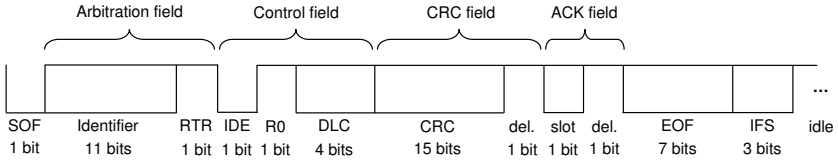

Figure 3.2: CAN base frame format (CAN 2.0 A) of the Remote frame

When the channel is free (no frame is being broadcast), any node willing to trans-

mit firstly sends a dominant bit called *Start Of Frame* (SOF). All nodes that listen to that SOF synchronize with that transmitting node. However, it is possible that more than one node quasi-simultaneously send a SOF. In such a case a contention takes place, during which all transmitting nodes decide which one finally gains the access to the medium for transmitting. Moreover, a node can decide transmitting just after detecting a SOF. If this occurs, the node is normally allowed to start sending its own frame (in the next bit without transmitting a SOF), thereby initiating or joining in a contention.

A contention is solved by using a *bit-wise arbitration mechanism* that relies on the *arbitration field* of the frame. This field includes the *identifier* and the *Remote Transmission Request* (RTR) bit. The arbitration mechanism, which is described later on, ensures that after the *arbitration field* only the node with the lowest identifier and RTR continues transmitting. The identifier is constituted by 11 bits whose value identifies the frame itself. Notice that an identifier does not belong to a given node, since CAN nodes have not addresses. Regarding the RTR bit, it is '0' to indicate that the frame is a data frame, or '1' to indicate that it is a remote frame.

The *arbitration field* is followed by the *control field*, which includes the *IDentifier Extension* (IDE) bit, a reserved bit called R0, as well as the *Data Length Code*. The former is '0' or '1' to respectively indicate whether the frame follows the *CAN base frame* or the *CAN extended frame* format. The R0 bit is reserved for future protocol extensions. The DLC is a 4-bit field that indicates the number of bytes that will be included in the *Data field*, in the case the frame is a data frame; or the number of requested data bytes, in the case of a remote frame.

In order to allow the receiving nodes to check the integrity of the frame, the transmitter sends a *Cyclic Redundancy Code* (CRC) after de Data field. This code is included within the so called *CRC field*, which is constituted by 16 bits. The 15 first bits of this field are the value of the CRC itself, whereas the last one is a recessive bit called *CRC delimiter*.

The *ACKnowledge (ACK) field* is used by the receiving nodes to indicate whether or not they consider that the part of the frame that has been broadcast so far is correct. The transmitting node always sends a recessive value at the first bit of this field, i.e. at the *ACK slot*. In contrast, the receiving nodes that want to acknowledge the frame send a dominant bit, called *ACK bit*, in the ACK slot, thereby overwriting the recessive sent by the transmitter. The second bit of the *ACK field* is always recessive and it is called *ACK delimiter*.

Finally, the frame ends with the *End Of Frame field* (EOF), which consists of 11 consecutive recessive bits. After this field, an *Intermission Frame Space* (IFS) of 3 recessive bits follows and, afterwards, the channel becomes free or *Idle* (at the

recessive state) as long as no node initiates a new transmission.

3.3.2 Bit-wise arbitration mechanism

As explained above, if more than one node decide to star transmitting at the same time, a contention occurs after the start-of-frame (SOF). When this happens, in order to decide which node gains the access to the medium, all transmitters initiate the so called *bit-wise arbitration mechanism* of CAN.

Specifically, each transmitting node observes the actual bits on the bus while transmitting its arbitration field. If a transmitting node monitors a dominant bit, '0', while it is issuing a recessive bit, '1', then it assumes that it has lost the contention, backs off, becomes a receiving node, and retries after the current frame.

At the end of the arbitration phase only the node with the lowest identifier field continues transmitting. Also notice that since the RTR bit of the arbitration field of a data frame is '0' and the RTR of a remote frame is '1', data frames have a higher priority than remote ones.

3.3.3 Frame encoding

CAN uses a *Non Return to Zero* (NRZ) bit coding, so that the signal level is kept at the same value during all the bit time. However, as said before, nodes use the recessive to dominant transitions of the signal on the channel to keep synchronized with the transmitting node. Therefore, it is necessary to limit the time the signal remains at the same level, so that nodes can keep correctly synchronized.

This is achieved by using the *stuff rule*. This rule basically specifies that the transmitter must insert a complementary bit, called *stuff bit*, whenever it has already transmitted five consecutive bits of the same polarity (including stuff bits). All the frame is encoded using the stuff rule, except the CRC delimiter, the ACK field and the EOF.

3.3.4 Error detection and signalling

Each CAN node includes a set of mechanisms that provide in-bit detection of bit-stream errors. Any CAN node is able to detect five different error types [ISO93]: *stuff error*, *format error*, *bit error*, *CRC error* and *ACK error*.

Such errors are detected by means of several *error-detection mechanisms* that check the correctness of the frame that the node transmits or receives. These mech-

anisms respectively are: *stuff rule check*, *frame check*, *monitoring*, *CRC check* and *ACK check*. Each node uses them as follows.

- Stuff Error. Both, the transmitter and the receivers perform a *stuff rule check* to test if the stream being broadcast fulfils the stuff rule. As explained before, this rule basically specifies that a complementary bit, called *stuff bit*, must follow every five consecutive bits of the same polarity (including stuff bits).
- Format error. Both, the transmitter and the receivers perform a *frame check* to test if the frame obeys the format rules. These rules define the characteristics of each field of the frame: order within the frame, length and allowed bit values.
- Bit error. Each node (either transmitter or receiver) performs a *monitoring* of the signal on the channel in order to check that whenever it transmits a dominant bit, the resultant bit in the channel is actually a dominant bit. Moreover, the transmitter also checks that whenever it transmits a recessive bit, the resultant bit is also recessive (except in the *arbitration field* and in the ACK slot).
- CRC error. As explained before, the transmitter calculates a 15 bit Cyclic Redundancy Code (CRC) based on the bits of the frame it has already transmitted and, next, it transmits such CRC in the last but two field of the frame. The receivers also calculate the CRC and carry out an acceptance test to check if it matches with the CRC received i.e. they perform a *CRC check*.
- ACK error. As already explained, receiving nodes send a dominant bit value at the *ACK slot* if they wish to acknowledge the correct reception of the frame. The transmitter performs an *ACK check* to test if there is a dominant bit value in the ACK slot and thus, to detect if at least one node has acknowledged the frame.

As will be explained later, depending on the number of errors detected up to a given instant of time, a node is in the *error-active* state, in the *error-passive* state or in the *bus-off* state. Thus, one can consider that a node is error-active, error-passive or bus-off respectively.

A node detecting an error initiates the *error-signaling mechanism*, which consists in signaling an error by transmitting an *error frame*. The node starts transmitting that frame in the next bit after detecting the error. Except in the case of a receiving node that detects a CRC error. In that case, the node starts sending the error frame at the first bit of the EOF.

An error frame is formed from an *error flag* followed by an *error delimiter*, and its actual appearance depends on some factors such as, for example, on wether nodes are error-active or error-passive[1]. An error-active node signals an error by transmitting an *active error flag*, which is composed of 6 consecutive dominant bits. Conversely, an error passive node signals an error by sending a *passive error flag*, which is constituted by 6 consecutive recessive bits. An active error flag always violates the stuff rule and provokes all the nodes to detect an error and to signal it too. In such a way, the error is globalized and the frame that was being transmitted is rejected by all the nodes, i.e. it is said that an *error globalization* occurs. Nevertheless, a passive error flag does not always force the other nodes to detect an error and thus, no globalization is ensured when an error-passive node signals an error.

In case an *error flag* (active or passive) provokes a globalization, all nodes cooperatively transmit an *error delimiter* after transmitting their own error flags. Thus, the final appearance of an *error frame* is the one that results from the overlapped error flags, followed by the cooperative error delimiter.

The cooperative error delimiter is constituted by a minimum of 8 consecutive recessive bits, and it is built as follows. After transmitting its own error flag, each node sends its error delimiter, which is constituted by consecutive recessive bits, until it monitors a pattern of 8 consecutive recessive bits in the channel. This pattern indicates the end of the error delimiter and, thus, of the error frame. Notice that since dominant values always overwrite recessive values, all nodes will monitor dominant bits as long as there is a node that is still transmitting its active error flag. Therefore, all nodes will observe the beginning and the end of the sequence of 8 consecutive recessive bits at the same time, i.e. they will quasi-simultaneously detect the end of the error frame.

Finally, the error frame is followed by the Intermission Frame Space and then the channel becomes idle.

At this point, it is worth noting that the CAN error-signalling mechanism is designed for providing error-active nodes with *data consistency*. As said in Section 2.3, if *data consistency* is ensured for a set of nodes, then it is guaranteed that each frame is quasi-simultaneously accepted by all these nodes or by none of them [ISO93]. This property is very valuable for dependable systems and, therefore, data consistency has been considered as one of the most important advantages of the CAN protocol. However, it is worth noting that CAN actually does not enforce data consistency in some situations. On the one hand, data consistency is not enforced if there is any error-passive node. This is due to the fact that a passive

[1] A bus-off node does not communicate

error flag cannot always force an error globalization. On the other hand, data consistency is violated in the presence of some *inconsistency scenarios* identified in [RVA+98] and [PMJ00]. As will be explained, although the solutions we propose in this document do not totally overcome these problems, at least they do not make them worse.

3.3.5 Fault treatment

The CAN standard specifies some fault-treatment mechanisms to diagnose and passivate faults occurring at nodes. Specifically, each CAN node includes two error counters [ISO93]: the *Transmission Error Counter* (TEC) and the *Reception Error Counter* (REC). Depending on the value of these counters, the node changes its way of behaving in order to minimize its impact on the communication.

A CAN node increases its TEC/REC almost every time it detects an error in the channel, e.g. a stuff error, a format error, etc. Specific rules [ISO93] are followed to decide how to increase an error counter. Basically, these rules penalize more the errors the node detects when it is acting as the transmitter than when it is acting as a receiver. In this way, the transmitter normally increases the TEC by 8 units, whereas a receiver usually increases the REC by 1 unit.

Additionally, the node further increases the TEC or the REC (depending on whether it is acting as a transmitter or a receiver) in 8 units when it suspects that it is the responsible for an error. This basically happens when it detects a *primary error* [ISO93], i.e. when it monitors a dominant bit after its own error flag. That is because a node that detects this dominant bit can assume that it was one of the nodes (or the only node) that firstly detected an error and started to signal it, thereby provoking the other nodes to detect an error and to perform an error signaling too. In other words, a node detecting a primary error means that the node did not detect an error as a consequence of the error signaled by other node, but probably due to a local error.

Besides the rules for increasing the TEC/REC, the node also follows specific rules for decreasing them. Normally, the TEC (or the REC in the case of a receiver) is decreased by 1 unit (unless its is already 0) when a frame is successfully transmitted (or received). The only case in which the REC is decreased by more than 1 unit is when its value is greater than 127; in such a case the REC is set to a value between 119 and 127.

As said before, the node takes into account the values of the TEC/REC to reduce its impact on the communication. A CAN node is initially in the *error-active state*, meaning that it is involved in bus activities without any restriction. However, when

any of its error counters becomes higher than 127, the node goes into the *error-passive state*. The single difference between this state and the error-active state is that an error-passive node signals an error by means of a passive error flag. As explained before, a passive error flag does not necessarily provoke an error globalization, since it is constituted by 6 consecutive recessive bits. Therefore, when an error-passive node detects and signals a *local error*, i.e. an error that is only detected by one node locally, it does not necessarily corrupt an on-going frame. This actually reduces the propagation of errors generated by a fault that only affects that node. Moreover, a node goes into the *bus-off state* whenever its TEC becomes greater than 255. In such state the node is no longer involved in the communication and, thus, this situation corresponds to a node diagnosing itself as being permanently faulty.

Finally, note that an error-passive or a bus-off node can become error-active again. This allows reintegrating nodes that are not permanently faulty. Specifically, an error-passive node becomes error-active when both its TEC and REC are less than or equal to 127. A bus-off node becomes error-active after observing 128 *CAN bus-free occurrences*, i.e. 128 sequences of 11 consecutive recessive bits [ISO93].

3.3.6 Overload signalling

CAN specifies two kinds of overload conditions [ISO93]. The first condition occurs when a receiving node needs an extra delay before a new data or remote frame can be transmitted on the bus. When this happens, the node starts signaling an *overload flag*, constituted by 6 consecutive dominant bits, at the first bit of the Intermission Frame Space (IFS).

The second overload condition happens when a node detects a dominant bit during the IFS. When this occurs, the node must react by signaling an overload. Notice that this behavior is the mechanism that allows globalizing an overload previously triggered by the first condition.

After transmitting their own overload flags, all nodes cooperatively transmit an *overload delimiter* of at least 8 consecutive recessive bits, i.e. the format of an *overload frame* is the same as the format of an active error frame.

3.4 Types of faults in CAN networks

Faults occurring at different components of a CAN network can manifest in several ways. Unfortunately, as explained in Section 1.1, the error-detection and fault-treatment mechanisms of the CAN protocol present important limitations. Moreover, it has also been pointed out that CAN lacks fault-tolerance facilities. Thus, the occurrence of one fault can be enough to cause a severe failure of communication. More specifically, to better understand how these faults can jeopardize the communication in a CAN network, let us classify them into the following two main categories.

On the one hand, we consider faults that generate errors that corrupt the bit values that are broadcast through the medium and that, thus, lead CAN frames to be incorrect from a syntactical point of view. These faults can be further classified into two categories:

- *Stuck-at* fault. It occurs whenever a given node or a component of the medium is damaged and issues a constant bit value. These faults can arise, for example, from short-circuits to ground or battery, or malfunctioning or isolated controllers. Two types of stuck-at fault exist: stuck-at-dominant and stuck-at-recessive faults, depending on whether the stuck-at bit is dominant or recessive respectively. Since the physical layer of CAN is equivalent to a logic-AND of every node's contribution, only a stuck-at-dominant node may cause a severe failure of communication (the recessive bit is implemented as the logical '1' value). In contrast, a stuck-at fault affecting the medium leads to a global failure, independently of whether the medium becomes stuck-at-dominant or stuck-at-recessive.

- *Bit-flipping* fault. This occurs whenever a component of the network (forming part of either a node or a medium) exhibits a *fail-uncontrolled* behavior and starts sending erroneous and random bits with no restrictions in the value domain. In this case, even if a node is trying to send a correct bit stream, this is destroyed by the dominant bits of the bit-flipping stream. Some potential causes of this fault are: a damaged node that sends random bit values; a bad welding on a medium connector that generates random bit values, etc.

On the other hand, components in a CAN network can also suffer from faults that lead CAN frames to be incorrect just from a semantical point of view. For instance, a fault could lead a node to transmit frames that are timely-incorrect, e.g. frames that are too early or too late, which is a typical problem of communication systems. Among these, there has been an important concern about *babbling-idiot* faults in

the context of real-time distributed embedded systems. This type of timing fault is defined in the literature as a situation in which a node sends messages in a way that it consumes more resources that it really needs, thereby starving other nodes of the appropriate resources for communicating [BB03]. A babbling-idiot fault may happen as a consequence of several causes. For instance, a fault manifests as babbling idiot if it affects the internals of a CAN controller and compels it to constantly send a frame placed in its transmission buffer. Another example could be a software fault in a node that results in an infinite loop that constantly sends messages that are incorrect in the time domain. Finally, a semantic fault that deserves careful consideration in this dissertation is the *Network partition* fault. It occurs whenever the network is broken into several subnetworks, which are called *network partitions*. Since any two nodes located in a different partition can no longer communicate with each other, the nodes of the system have an inconsistent view of which ones are available for communicating. In a bus topology, a network partition is typically provoked by a physical disruption of the medium.

As can be inferred from the above sections, the error-detection and fault-treatment mechanisms the CAN protocol is provided with are only able to deal with the former above-mentioned types of faults, i.e. with syntactic faults. This fact, the limited effectiveness of the mechanisms it does include for dealing with stuck-at and bit-flipping faults, and its lack of fault-tolerance facilities encourage the investigation of solutions that improve its error-containment and fault-tolerance capabilities.

3.5 Conclusions

The Controller Area Network (CAN) protocol is a field-bus communication subsystem that has demonstrated to be ideally suited for a wide range of distributed control systems. Part of its success is due to the fact that it represents an excellent trade-off between dependability and cost.

In this chapter we have described the most important features of CAN. We showed that it relies on a differential bus line that is very resilience to Electromagnetic Interference (EMI). We also explained that the medium of CAN implements a wired-AND function of every node contribution, thereby enabling the use of in-bit response together with dominant/recessive transmission. These two last properties are the basis for the definition of a set of mechanisms that yield important benefits in terms of dependability and real-time performance. On the one hand, CAN provides a prioritized medium access arbitration mechanism that avoids indeterministic medium access delays. On the other hand, CAN includes in-bit detection of bit-stream errors, as well as error signalling and globalization mechanisms that

are aimed at providing data consistency.

Nevertheless, CAN relies on a simplex bus topology that poses important limitations on the containment of syntactic errors and that lacks fault-tolerance mechanisms. To better understand these limitations we classified syntactic faults into stuck-at-recessive, stuck-at-dominant and bit-flipping faults. Moreover, we also explained that CAN does not include any mechanism for dealing with faults that cause semantically-incorrect frames (either in the value or the time domain), e.g. babbling-idiot faults.

In conclusion, despite the good characteristics of CAN concerning dependability and real-time, its error-containment and fault-tolerance limitations justify the research on new solutions for CAN that keep the good properties already presented by this technology. In particular, since some of these limitations are imposed by its simplex bus topology, it is worth investigating new topologies for CAN that do not present these drawbacks and that could even ease the inclusion of new error-detection, fault-treatment and fault-tolerance mechanisms.

Chapter 4

Potential solutions for improving dependability in CAN

4.1 Introduction

Some of the faults that can cause a severe failure of communication in CAN (see Section 5.2) can be confined in bus-based systems, up to a certain extent, using techniques that are already known. These techniques rely on the use of replicated transmission media, *bus guardians* and reconfigurable bus topologies. However, due to the characteristics that are inherent to the bus topology, said techniques do not prevent the existence of multiple components such that a single fault in any of them may cause a severe failure of the communication system.

In contrast, the interest in using star topologies has been growing, given their better error-containment capabilities and fault resilience. For example, the LAN domain has long moved to star topologies with Ethernet, a technology that is now extensively used in the industrial automation and large embedded systems domains. In this latter case, particularly concerning in-vehicle systems, we can find other examples of transition topologies such as with TTP/C [BKS03] and FlexRay [Fle05]. Different star topologies have also been proposed for Controller Area Network (CAN), but they still do not take a full advantage of the potential benefits of this topology.

This chapter aims to provide an overview on the technics that have been proposed for improving dependability of CAN, while identifying their main drawbacks.

4.2 Replicated bus topology

The use of a replicated bus topology generally allows nodes to detect a faulty or an error-polluted bus by comparing the values received from each of the replicas [RVA99] [Rus03] [SFF06]. In this way, nodes can disable a faulty bus so that the communication system can still provide a correct service.

Nevertheless, this solution does not prevent a faulty node (a stuck-at-dominant node, for instance) from causing a failure of the whole communication system by sending erroneous information to all replicas, i.e. all bus replicas can exhibit *common-mode failures*. An example of such a situation is depicted in Figure 4.1, in which a faulty node blocks both buses by sending a stuck-at-dominant value to both of them.

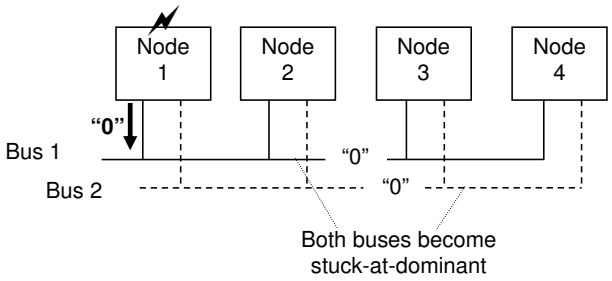

Figure 4.1: Common-mode failure in replicated buses

Moreover, this solution has a more subtle weakness; regardless of the routing of the replicated buses, they have to come together near every node of the system. This can be a potential cause of *spatial-proximity failures* of the replicated media system. For instance, as showed in Figure 4.2, a smash near any node may cause a physical disruption in all media, thus provoking a network partition and a severe failure of communication [Kop03].

4.3 Reconfigurable bus topology

A different architecture, which is used in RedCAN [Fre02], connects nodes by means of a special ring in which one of its sectors is redundant and left inactive, so that the resultant topology is a bus. The ring can be reconfigured by shutting down one or more adjacent sectors and activating the redundant one. This is carried out when a stuck-at fault of the medium occurs in one or some active and adjacent

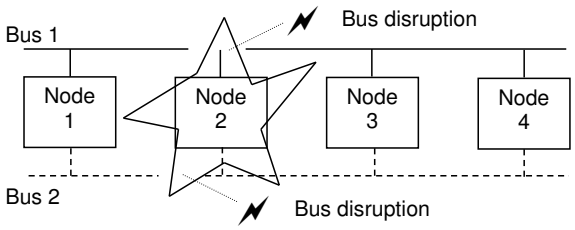

Figure 4.2: Spatial-proximity failures in replicated buses

sectors or if the node or nodes that connect adjacent sectors crash.

The main disadvantage of RedCAN is that it only deals with faults occurring in adjacent sectors or contiguous nodes. Moreover, this solution increases the complexity of the network nodes, thus increasing their probability of failure, and uses specific RedCAN hardware.

4.4 Bus guardian

Bus guardians have been proposed to prevent the propagation of errors from any node, thereby enforcing a fail-silent behavior [BB03] of the nodes. A bus guardian is basically a device which supervises the output of a node to its bus interface in order to detect incorrect behaviors. In this way, a faulty node, such as a stuck-at-dominant or a bit-flipping node, can be easily detected and isolated from the rest of the system. Moreover, a bus guardian could even include information concerning the scheduling of the messages in order to detect and isolate babbling-idiot faults.

Nevertheless, the weak point of this approach is that fault independence between a node and its corresponding bus guardian is not completely ensured, so that they can exhibit common-mode failures. These can be caused either by spatial proximity of a node and its bus guardian, or by sharing resources or procedures, e.g. power supply, system clock, clock synchronization algorithm. Figure 4.3 shows an example of this situation in which a node and its guardian share a clock supply that fails and stops working. In such a case, the output of both the node and its guardian can become stuck-at-dominant, thereby permanently blocking the bus.

Moreover, the use of bus guardians is useless for containing error propagation from a faulty medium, as depicted in Figure 4.4. As can be seen in this example, if the node stub is shorted to ground or to battery, the guardian can do nothing to prevent this from leading the whole bus to be stuck-at a given logical value.

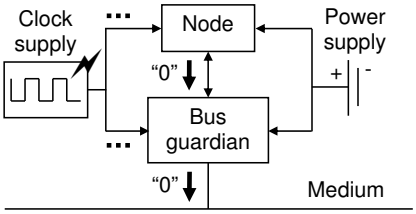

Figure 4.3: Common-mode failure when using a bus guardian

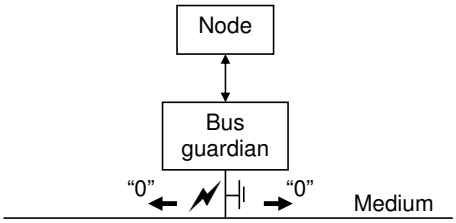

Figure 4.4: A useless bus guardian in the presence of a medium failure

4.5 Star topologies

From the discussion above we can conclude that even though the use of replicated media as well as bus guardians significantly improves the dependability characteristics of CAN, these mechanisms -even if they are used together in the same system- still allow multiple components to cause severe failures of the communication system, and therefore they do not fulfill the aim of this work. Thus, alternative solutions have been researched, namely those based on a star topology.

In a star topology, each node is connected to a central element, the *hub*, by its own *link*. On the one hand, this provides a natural way of enforcing *confinement*[1] of faulty transmission media by isolating the respective links at the respective hub ports. Moreover, the links of a star topology only come into spatial proximity at the center of the star. On the other hand, the hub has a privileged view of the system, as it simultaneously knows the contribution from every node and thus, it can play the role of bus guardian of each node. In this way, spatial-proximity and common-mode failures between a node and its corresponding bus guardian are avoided.

[1] *Fault confinement* is a concept introduced in the CAN specification [ISO93] to refer to the mechanisms CAN includes to passivate faulty nodes. This term roughly corresponds to the concept of error-containment introduced in Section 2.3.

Even though the star topology provides a good basis to improve the dependability of the communication system, the adoption of such a topology is not enough. Additional mechanisms should be included in the hub in order to detect and isolate faulty components and achieve the behavior of what we call the *ideal star*, that is a star-based system in which the hub includes all the mechanisms which are necessary to ensure containment of all errors which may cause a severe failure of communication.

Nevertheless, it is obvious that the main drawback of a star topology is that the hub represents a single point of failure. In addition, a star topology includes more cables than a bus, and the error-detection and fault-treatment mechanisms that could be included in the hub would make its complexity higher than for simpler components. This extra hardware and hub complexity can imply that the probability with which faults occur in a star is higher than in a bus topology. Fortunately, different strategies can be adopted in order to face these problems. For instance, the hub reliability can be increased by placing it in a well-protected zone inside the physical system or by investing in its quality. Moreover, the overall network's fault-resilience can also be improved by adopting a replicated star topology.

For the particular case of CAN, the interest in using star topologies to improve dependability is reflected in the considerable amount of star topologies that have been already proposed for this protocol in the literature. Next, we present an overview on these stars, focusing on their pros and cons.

4.5.1 Passive star couplers

In [Ruc94] [CiAb], a passive star network topology for CAN is presented. This solution relies on the use of a central element, the *passive star coupler*, which acts as a concentrator where all the incoming signals are electrically coupled. The result of this coupling is then broadcast to the nodes.

As concerns dependability, the only advantage this solution presents when compared with a bus-based CAN system is the reduction of the spatial-proximity problem between different links. This reduction is possible since the links only come into physical proximity at the center of the star.

However, this kind of star couplers shows some technical drawbacks that discourage its use from the practical point of view. On the one hand, large coupling loses impose strong limitations on the star radius (5 to 10 meters) and hence force nodes to communicate at low bit rates. On the other hand, the coupling of the incoming signals causes some electrical problems, such as resonances, harmonics or disturbances, which require the use of complex hardware solutions.

4.5.2 Active star couplers

Besides passive star topologies, active ones have also been proposed for CAN: [CiAb], [IXX09], [CDV01]. The central element of an active star topology is the so called *active star coupler*. Conversely to the case of passive star couplers, an active one does not merely electrically couple the incoming signals. Instead, it translates the physical signals it receives at its ports to logical values and, then, it combines them to obtain a resultant signal that is broadcast to all nodes. More specifically, an active star coupler basically receives the incoming signals from the nodes bit by bit, implements a logical AND function, and retransmits the result to all nodes. By means of this coupling these stars overcome the technical problems of passive star topologies.

In the first one of the referred active stars, [CiAb], each node is connected to the star coupler by an independent link constituted by two optical paths. The coupler includes a transceiver for each link (a so-called *node-coupler transceiver*) as well as an internal CAN bus with few centimeters of length. In a first stage, signals from each link are received by the respective node-coupler transceiver and transmitted without any processing into the internal CAN bus. In a second stage, the resultant signal from the internal CAN bus is received by each node-coupler transceiver and retransmitted towards the corresponding node. Since the star coupler includes neither any error-detection nor any fault-treatment mechanism, from the dependability point-of-view it just reduces the spatial-proximity problem between different links.

In contrast, the active star topologies available in [IXX09] do present error-detection and fault-treatment mechanisms for detecting and passivating ports that suffer from a permanent stuck-at-dominant fault. Since there are not technical information available describing the mechanisms included in these star couplers, it is difficult to evaluate the performance of their error-detection and fault-treatment mechanisms. However, notice that they use only one link to connect each node to the coupler and, therefore, they have to deal with the fact that nodes' contributions are not separated in the space. This leads us to think that the time needed to diagnose which port is actually stuck-at-dominant is considerably bigger than in a case in which each node contribution can be independently monitored before it is coupled with the others. Besides this possible disadvantage, it is also worth noting that these stars do not deal with stuck-at-recessive, bit-flipping and babbling-idiot faults.

The last active star topology referred above is called *StarCAN* [CDV01]. The main goal of this solution is not network dependability, but network performance. In particular, StarCAN achieves either an extension more than 10 times longer or a bit rate 10 times higher than a typical CAN network. Nevertheless, in order

to fulfill this goal, StarCAN sacrifices one of the most important characteristics of CAN, the in-bit response [ISO93]. This decision has an enormous impact on the dependability properties of the network. On the one hand, the lack of in-bit response jeopardizes the so-called *data consistency* of the CAN network, since inconsistency scenarios [RVA+98] [PMJ00] turn out to be more likely. On the other hand, despite keeping some CAN mechanisms, e.g. arbitration and error signaling, off-the-shelf CAN controllers cannot be used, raising issues about the practicality of the solution.

4.5.3 Bridge star couplers

A recent star topology proposed for CAN is ESCAPE CAN [HPDS08]. This is, however, a different kind of star coupler with respect to the previous ones since it does not operate at the physical layer but at the data link layer instead, being capable of storing and forwarding frames. In fact, the hub does not couple the contributions of the different nodes but manages each port as an independent CAN bus segment through which it performs the appropriate transmissions.

ESCAPE CAN addresses babbling-idiot and masquerading faults due to either hardware or software faults. More specifically, the main goal of ESCAPE CAN is to enhance the robustness of the arbitration and the acknowledge mechanisms of CAN. It is not intended to provide error detection and fault treatment for stuck-at and bit-flipping faults.

As concerns the arbitration mechanism, the hub of ESCAPE CAN aims at detecting and isolating a node that tries to send a not allowed identifier frame field, without aborting the on-going arbitration. To achieve this, the hub does not couple each node contribution, but monitors each one of its ports independently, and includes mechanisms to align the bit-streams of the different nodes with each other during the arbitration. If a port issues a not allowed identifier, the hub simply discards it without injecting an error in any port.

To emulate the CAN arbitration mechanism, the hub sends the lowest prioritized CAN identifier through all its ports. In this way, it leads each contending node to believe that it is winning the arbitration. However, the last bit of the identifier is reserved for the hub, so that no node is allowed to send a dominant value for this bit. This allows the hub to know, at the last-but-one bit of the identifier field, which node has won. Then, it uses the last (reserved) bit of the identifier to force all nodes, except the winner, to lose the arbitration.

When the arbitration phase ends, the hub continues delivering a *dummy frame* to the receiving nodes and continues monitoring the frame that the node that won

the arbitration (the transmitter) is sending. As soon as possible, the hub aborts the dummy frame by signaling an error in each port corresponding to a receiver, and stars retransmitting to the receiving nodes the frame the transmitter is sending. Once the transmitting node ends its frame, the hub forces it to signal an overload condition in such a way that the end of that overload coincides in time with the end of the frame the hub is retransmitting to the receiving nodes. In this way, all nodes reach the Intermission Frame Space and, afterwards, the idle state at the same time.

Regarding the other objective of ESCAPE CAN, i.e. to enhance the acknowledgement mechanism, the hub behaves as follows. Firstly, since the nodes' contributions are not coupled, the hub sends the ACK bit to the transmitter to make it to believe that the receiving nodes are acknowledging its frame. Secondly, later on, when the hub is retransmitting that frame, the hub observes the value that each receiving node sends at the ACK slot. In this way, once the frame is completely retransmitted, the hub knows which receiving nodes accepted that frame. Finally, just after the retransmission ends, the hub transmits to all nodes a new frame whose payload includes information concerning which nodes accepted the frame, i.e. what in the context of ESCAPE CAN is referred to as the *acknowledgement vector*. This information can then be used by the software running at nodes as a basis for a membership algorithm.

Although interesting, ESCAPE CAN exhibits important shortcomings. Firstly, it is not clear whether or not ESCAPE CAN improves the acknowledgement mechanism of CAN. Note that the hub sets up the *acknowledgement vector* taking into account the nodes that, from its point of view, have accepted the frame. Certainly, there is not enough available information about how the hub comes to a conclusion about whether or not a node accepts a frame. However, if this decision is only based on the reception of a dominant value during the ACK slot, then a single error that corrupts the ACK bit sent by a node will lead the hub to an incorrect conclusion. Moreover, due to the inconsistency scenarios of CAN [RVA$^+$98] [PMJ00], the hub cannot be sure about what nodes (including the transmitting one) actually accepted a given frame. In fact, since the acknowledgement mechanism proposed in ESCAPE CAN is more complex that the one included in CAN, it could make it possible the occurrence of scenarios concerning inconsistencies that have not been appropriately investigated yet. Finally, notice that the frame that carries the *acknowledgement vector* can also be inconsistently received.

The second disadvantage of ESCAPE CAN is that it reduces the compatibility with existing CAN applications. On the one hand, ESCAPE CAN substantially restricts the amount of possible values that can be used as a frame identifier. Specifically, only the hub can use the lowest prioritized identifier; the first and the last bit of any identifier are also reserved for the hub; and it is not possible to use identi-

fiers that include stuff bits. On the other hand, this compatibility is also negatively affected by the hub leveraging the priority of the identifier of every frame it retransmits. This obliges the application to recalculate the original value of the identifier of each frame it receives.

Thirdly, note the hub of ESCAPE CAN includes mechanisms that allow it to store and forward a CAN frame, to leverage a frame identifier, and to recalculate CRCs. This implies that ESCAPE CAN may exhibit failure modes that are more typical of switches rather than of hubs, e.g. it may fail by creating and sending false CAN frames. Moreover, since the hub manages each port as an independent CAN bus segment, it must include a considerable amount of hardware to communicate through each port independently from the others. This implies that the complexity of the hub should be quite high when compared with other active hubs. In conclusion, the higher failure rate and the possible failure modes of the hub are issues of main concern that should be further investigated to justify ESCAPE CAN benefits.

Finally, note that a secondary objective of ESCAPE CAN is to improve the performance of CAN. Nonetheless, it is not clear if it actually degrades it. Authors claim that the key to improve the performance is to decouple the nodes' contributions. This allows the hub to avoid aborting an on-going transmission when, during the arbitration phase, it needs to isolate a candidate that is sending a not allowed identifier. Nevertheless, this decoupling also obliges the hub to abort the dummy frame it sends to the receiving nodes (by injecting an error frame), and then, to retransmit the frame the transmitter is sending. This behavior actually increases the time needed to broadcast each frame through the network, thereby probably reducing the overall performance of CAN.

4.6 Conclusions

The use of CAN in highly-dependable applications has been controversial due to a few factors, such as its bus topology, which includes multiple points of severe failure.

Solutions based on either simplex, reconfigurable or replicated bus topologies suffer from several impediments to enforce error containment, even if they are used together with bus-guardians. Replicated buses and bus guardians still may exhibit spatial-proximity and common-mode failures. Moreover, bus-guardians cannot contain errors provoked by faults in the media, whereas reconfigurable buses can only deal with a few faults.

In contrast, star topologies may represent an effective solution to prevent the

existence of multiple severe points of failure. In a simplex star topology, each node is connected to a central element, the hub, by its own link. One advantage of a simplex star topology is that links only come into spatial proximity at the center of the star and, thus, the probability that different links suffer from spatial-proximity failures is significantly reduced. But the most important advantage is that the center of the star, i.e. the hub, can be designed to have a privileged view of the system, knowing the transmissions from each node through the corresponding links. Thus, the hub can act as the bus-guardian of every node while not exhibiting common-mode failures. Because of all these advantages, a star topology allows reducing the number of components whose failure can cause a severe failure of the communication system, to a unique single point of failure, i.e. the hub.

Some star topologies have already been proposed for CAN. Unfortunately, they either do not address fault confinement; only deal with a small set of possible faults or with faults not related to our fault model; are not compatible with CAN; or are not even implemented yet. Therefore, we can conclude that none of these stars fulfills our goal of preventing the existence of multiple components such that a single fault in any of them may cause a severe failure of communication. In fact, almost all the studied star topologies behave as a bus with enhanced resilience to spatial-proximity failures. This justifies the design of a new star topology for CAN, with special focus on achieving an ideal star.

Chapter 5

CANcentrate basics

5.1 Introduction

In Chapter 4, it was shown that neither bus topologies nor existing star topologies do fulfill the strong dependability requirements of many systems, since they allow a single fault in any of multiple network components to cause a severe failure of communication. Due to this, we have proposed a new star topology, called CANcentrate, which does not exhibit this drawback.

In the CANcentrate architecture, each node is connected through a dedicated link to a different port of a central hub and, therefore, a node together with its link can be considered as an error-containment region. Moreover, from the hub perspective, a fault within a given error-containment region manifests as a faulty port. Thus, from now on, we will say that the hub contains errors by detecting and isolating the appropriate *faulty ports*.

This chapter is devoted to describing the CANcentrate architecture, paying special attention to the internal structure of the hub. The error-detection and fault-treatment mechanisms the hub includes are thoroughly discussed in Chapters 6, 7 and 8.

5.2 Fault model

As explained in Section 2.2, one of the first steps towards the design of a dependable system is to specify the type of faults it has to deal with, i.e. its fault model. Notice that we propose to enhance the reliability of CAN-based systems by means

of star topologies that prevent faults from causing a severe failure of communication. Thus, in principle the fault model we consider in the rest of this dissertation gathers all the different kinds of faults that may happen in the components of a CAN network and that may cause such a failure. These faults where thoroughly explained in Section 3.4.

However, in the context of this dissertation we focus on solutions for CAN that are independent of the application. Therefore, although semantic faults may provoke a severe failure of the communication system, we postpone the treatment of those that require information specific to the application. In this sense, the only semantic faults we deal with are network partitions, which as will be explained later on can be addressed by means of topological features of a star. In contrast, we rule out the treatment of semantic faults such as babbling-idiot ones, which would require information concerning the scheduling of messages. Anyway, note that for instance it is suitable to include in the hub a bus-guardian, similar to the one proposed in [BB03], for dealing with faults in the time domain.

In conclusion, our fault model includes faults that manifests as stuck-at-recessive, stuck-at-dominant, bit-flipping or as a network partition.

5.3 Design rationale

Probably the most important characteristic of CAN is the dominant/recessive transmission. As explained in Section 3.2, this means that a recessive bit value is received by all nodes in the network only if every node issues a recessive bit; it is enough that one node transmits a dominant bit value to force all nodes to receive a dominant bit. Moreover, we also explained therein that CAN communication relies on a complex bit synchronization mechanism that enforces *in-bit response*, which guarantees that nodes have a quasi-simultaneous view of every single bit on the channel. This synchronization mechanism uses the recessive to dominant transitions of the signal on the channel to keep the nodes of the network synchronized with respect to the one which is transmitting, i.e. with respect to the *leading transmitter*. As already explained, this bit synchronization limits the maximum bit rate of the network, but at the same time allows definition of a number of additional mechanisms, e.g. bit-wise arbitration, error signalling and globalization, which significantly improve the dependability and real-time properties of CAN networks [ISO93]. Due to the relevance of these mechanisms, it is very important to preserve them even if a star topology is used instead of a bus.

Assuming that the typical assignment is done, i.e. logical ‘1’ to recessive value and logical ‘0’ to dominant value, in order to keep the dominant/recessive transmis-

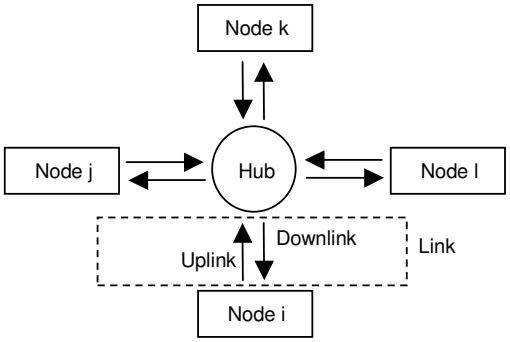

Figure 5.1: Architecture of CANcentrate

sion, the hub must implement a logical AND function of the contributions received from every node. Moreover, and in order to preserve the in-bit response, this logical AND must be performed within a fraction of one bit time, despite the extra delay which the internal circuitry of the hub may cause.

Furthermore, the hub must include some mechanisms in order to identify faulty ports. These mechanisms, which are thoroughly described later on, require the hub to be able to discriminate the signal that any node transmits from the signal resulting of the logical AND that the hub broadcasts to the nodes. A simple way to separate both signals is through the use of two different cables for each link that connects each node to the hub. Figure 5.1 shows the corresponding architecture in which there are only point-to-point unidirectional electrical connections.

The cable that carries the signal from a node to the hub is called the *uplink*, whereas the cable that carries back the resulting signal from the hub to the node is called the *downlink*. Each cable is of the same type as the twisted copper wiring used for implementing typical CAN buses, which have a good resilience against electromagnetic interferences. Moreover, each cable is terminated at both its ends, the node and the hub.

Therefore, two transceivers are required at the end of each link; one for the uplink and another one for the downlink. Figure 5.2 illustrates how the transceivers are connected at the end of the node. Note that the *receive data output* pin (RxD) of the uplink transceiver is left open whereas the *transmit data input* pin (TxD) of the downlink transceiver is forced to have a recessive level (the logical ‘1’ value). It is important to remark that the CANcentrate architecture can be implemented with both off-the-self CAN controllers and off-the-shelf CAN transceivers. This makes the solution practical and relatively low-cost. Nevertheless, the hub requires some

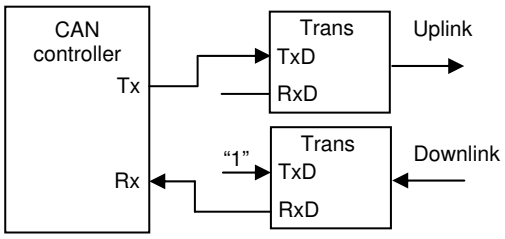

Figure 5.2: Configuration of the transceivers to connect a node to its link

specifically designed hardware, as discussed next.

5.4 Internal structure of the hub

The hub plays a crucial role in the star topology since it performs two fundamental functions. On the one hand, it implements the logical AND function which allows preservation of the dominant/recessive transmission of CAN as well as the rest of dependability mechanisms of CAN. On the other hand, it includes a number of mechanisms to detect error at the hub ports and to isolate any of them that becomes faulty.

The hub is divided into three modules, namely the *Input/Output Module*, the *Coupler Module*, and the *Fault-Treatment Module*. The structure and interconnections of these modules are depicted in Figure 5.3.

The Input/Output Module is made up of a number of transceivers; two for each link. As Figure 5.3 shows, one transceiver is assigned to every uplink in order to convert the physical signal received from each node into a logical value that the hub can process, $B_{1..n}$. Moreover, one transceiver is assigned to every downlink so that the logical output of the hub, the resultant *coupled signal* B_0, is converted into a physical signal that is broadcast to every node.

The Coupler Module is made up of an AND gate, which performs the coupling of the uplink signals, and a number of OR gates, one per link, which allow the hub to disable the contribution to the global AND from a specific uplink. In particular, the contributions that are disabled are those from ports that have been diagnosed as being faulty. Since the AND gate replaces the wired-AND functionality of the CAN bus, this means that the output of the Coupler Module, B_0, would be the same of a CAN bus where there were no faulty component. The frame that results from coupling the frames from the enabled ports (and, therefore, that would result

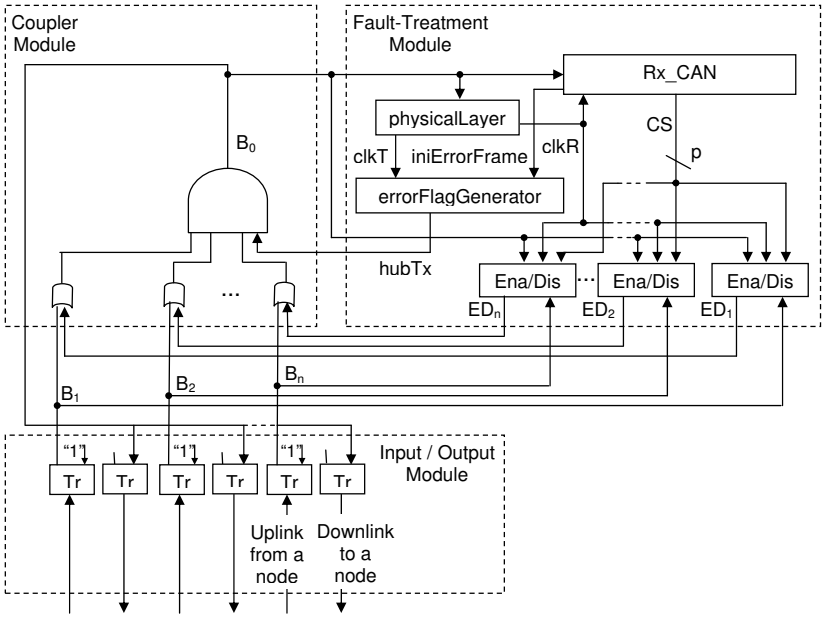

Figure 5.3: Internal structure of the hub

in a CAN bus without faulty nodes) is called the *resultant frame* hereafter.

This configuration causes an additional delay on the signal that the nodes receive. For the bit synchronization of the nodes, this additional delay has to be taken into account as a part of the *propagation time* [ISO93]. For all purposes it is similar to the extra delay caused by an equivalently longer cable in a bus system.

Note that the output of the AND gate is connected to each and every one of the downlink transceivers. In this way, the output of the hub (i.e. the coupled signal) does not interfere with the signals received through the uplinks, so the contribution of every node remains separated and further mechanisms can be applied in order to identify a faulty port.

The Fault-Treatment Module is devoted to performing *error detection* and *fault treatment*. Notice that the name of this module only reflects the concept of fault treatment for the shake of succinctness. The Fault-Treatment Module performs error detection in order to identify when a port issues incorrect data. This is essential to carry out fault treatment actions, which include both *fault diagnosis* and *fault passivation* (see Section 2.3). In the context of CANcentrate, fault diagnosis aims at finding out when a port is faulty; whereas fault passivation aims at isolating the faulty port from the system.

The error-detection mechanisms of the Fault-Treatment Module require the identification of the contributions from every uplink as well as knowledge of the *current state* of the *resultant frame*. This current state represents what all nodes are supposed to have received from the hub until this moment, and therefore permits to forecast which should be the proper contribution of each node for the following bit. Fortunately, the use of two cables for each link keeps the contribution from each link separated, and therefore the physical source of the errors can be more easily established.

However, this architecture does not allow the hub to discriminate between errors that are caused by a faulty transmission medium and faults that are caused by a faulty node. Therefore, as stated before in Section 5.1, from the point of view of the hub, either a faulty medium or a faulty node are viewed as a *faulty port*.

The current state of the *resultant frame* describes the meaning of the bit of the *resultant frame* that is currently being broadcast to all ports. The knowledge of such current state requires to keep the synchronization of the hub with the *resultant frame* at bit level as well as at frame level. The synchronization at bit level allows the hub to agree with all the nodes about the beginning and the end of each bit time in order to perform a correct sampling of the bit value; whereas the synchronization at frame level allows the hub to agree with all the nodes about the location of each bit inside the frame, i.e. about the frame field the bit belongs to.

On the one hand, the *Physical Layer Module* uses the typical CAN synchronization mechanisms [ISO93] for allowing the hub to synchronize with the bit stream at bit level and this generates the reception and the transmission clocks (clkR and clkT respectively in Figure 5.3). As in a normal CAN node, the reception clock indicates to the hub the instant of time at which the input signal from the medium (the coupled signal and each port contribution in the case of the hub) must be sampled; whereas the transmission clock indicates the instant of time at which a transmission bit value could be issued to the medium if the hub needs to transmit a bit.

On the other hand, the *Rx_CAN Module* observes the bit stream at the coupled signal in order to achieve the synchronization at frame level. As a result of this synchronization, the Rx_CAN Module generates a set of signals, we call *Current State* signals (CS), that together with B_0 describes the current state of the *resultant frame*. However, as will be explained in Section 5.5, the Rx_CAN Module must keep the synchronization at bit and at frame level with the other nodes in spite of the presence of errors that are observed in the resultant coupled signal. It is very important not to confuse the errors that are observed in the coupled signal with the errors that are observed at a given hub port. Errors at the coupled signal are broadcast to all nodes, leading the hub and nodes to desynchronize with each

other. In contrast, an error at a hub port is not necessarily propagated to the coupled signal and, hence, it is not always broadcast or provokes a desynchronization. To deal with this situation and force the hub and all nodes to re-synchronize when the hub or any node detects an error in the coupled signal, the hub includes some of the standard error management mechanisms of CAN. Section 3.3.4 thoroughly discussed what are these mechanisms. Basically they consist in signaling and globalizing any detected error by transmitting an active error flag. Specifically, when the Rx_CAN Module detects an error in the coupled signal, it orders the *Error Flag Generator Module* within the Fault-Treatment Module (*errorFlagGenerator* in Figure 5.3) to transmit the active error flag. Then this module transmits the flag through a dedicated contribution, *hubTx*, driven into the global AND.

The ultimate error detection, fault diagnosis and fault passivation are carried out by the *Enabling / Disabling* units (*Ena/Dis* in Figure 5.3). Each one of these units uses the set of signals CS, which are better described in Section 6.3, and the resultant coupled signal, B_0, to know the current state of the *resultant frame*. This information is used together with the contribution from its corresponding port (either B_1, B_2, etc.) in order to detect errors and to diagnose whether its port is faulty or not.

Whenever a given Enabling/Disabling Unit diagnoses its corresponding hub port as being faulty, it removes the contribution of this port from the system by issuing a logical '1' to the corresponding *Enabling/Disabling* signal, $ED_{1..n}$, which is connected to the OR gate that corresponds to the faulty port (see Figure 5.3). This effectively removes the contribution of this port to the global AND, being equivalent to disconnecting the link, and the corresponding node, from the hub. In general, this mechanism is similar to the one proposed in [RVA99] to manage, locally in each node, the media redundancy in a replicated bus topology.

The mechanisms that have been devised in order to detect errors at the hub ports, as well as to diagnose them as faulty are thoroughly described in Chapters 6, 7 and 8.

5.5 Hub synchronization in the presence of errors at the coupled signal

As stated above, the error-detection and fault-treatment mechanisms of the Fault-Treatment Module require knowledge of the current state of the *resultant frame*. However, errors that corrupt the coupled signal can lead the hub and nodes to desynchronize at bit and/or at frame level, i.e. to disagree on the logical value

of the bit that is being broadcast, and/or on the frame field this bit belongs to. If the hub and a node are desynchronized with each other, the hub can misinterpret the contribution it receives through the hub port corresponding to that node and, thus, it can incorrectly believe that this contribution is erroneous or even faulty. This desynchronization can be provoked for various reasons. For instance, an electromagnetic disturbance may affect a bit in a way that not all nodes (and the hub) sample the same logical value.

Since a desynchronization can lead the hub to misinterpret the contribution of a given hub port, the hub needs mechanisms to keep synchronized with all nodes in spite of the presence of errors at the coupled signal. Fortunately, the standard CAN [ISO93] embraces error management mechanisms that force nodes to re-synchronize after any of them detects an error. The hub includes some of these mechanisms in order to force itself and all nodes to re-synchronize whenever it or any node detects an error in the coupled signal.

In order to better understand how the hub can keep synchronized with all nodes, let us summarize the error management mechanisms of CAN, which were outlined in Section 3.3.4. Any CAN node is able to detect five different error types [ISO93]: *stuff error*, *format error*, *bit error*, *CRC error* and *ACK error*. Whenever a node detects an error, it signals it by means of an active error flag or a passive error flag, depending on whether the node is error-active or error-passive.

An active error flag always violates the stuff rule and forces all nodes to detect an error and to signal it too. In such a way, the error is globalized and the frame that was being transmitted is rejected by all the nodes, i.e. we say that an *error globalization* occurs. Nevertheless, a passive error flag does not always force the other nodes to detect an error [ISO93] and thus, no globalization is ensured when an error-passive node signals an error. This is an important issue since *data consistency* can only be ensured if all nodes which detect an error are able to globalize it (see Section 3.3.4).

Moreover, as already explained, after an active error flag provokes an error globalization, all nodes cooperatively transmit an error delimiter after transmitting their own error flags. Since all nodes recognize the end of an error delimiter at the same time, they become re-synchronized at frame level at that time. This is very important because it guarantees that all nodes are re-synchronized before any node tries to transmit a new frame.

In contrast, it is not guaranteed that an error passive node that signals an error becomes re-synchronized with the other nodes before a new frame is transmitted. For example, imagine a situation in which a receiving node detects an error during the End Of Frame field (EOF) of a given frame. The node signals the error, but it

is not globalized, thereby leading to an inconsistency scenario in which all nodes except itself accept the frame. This situation is certainly undesirable, but it can be even worse. Notice that after transmitting the passive error flag, this node will continue monitoring the *resultant frame* trying to detect the pattern of 8 consecutive recessive bits that indicate the end of the error passive delimiter (see Section 3.3.4). Then, imagine that other node starts transmitting a new frame before the error passive node detects this pattern. In such a situation, and considering that no more errors occur, the error passive node will only be able to recognize a sequence of 8 consecutive recessive bits when monitoring the EOF of this new frame. As a consequence, not only the fist frame was inconsistently accepted, but also the second one. Thus, one can conclude that it is necessary to re-synchronize all nodes as soon as possible after an error occurs.

All these considerations regarding the CAN error-signaling mechanism influence the design of the hub. As stated above, the hub includes some of the error management mechanisms of CAN to keep synchronized with all nodes in spite of the presence of errors at the coupled signal. Specifically, since only an active error flag ensures the globalization of any error, the hub must behave as an error-active node when signalling an error. This means that the hub always signals an error by transmitting and active error flag, regardless the number of errors it has detected in the coupled signal so far. Specifically, as indicated in Section 5.4, the Rx_CAN Module is the responsible for detecting errors at the coupled signal. Thus, when this happens, the Rx_CAN Module simply orders the Error Flag Generator Module to transmit an active error flag.

At this point, it is also noteworthy that in order to reduce the probability of *data inconsistency* the hub not only signals errors as an error-active node, but it also considers that error-passive nodes are faulty, i.e. it only allows contributions of error-active nodes.

As concerns the types of errors the hub signals, notice that it is never the original transmitter of a message; so that it observes the coupled signal from the point of view of a receiver. Thus, it can only detect in the coupled signal those errors that a receiving CAN node would detect. More specifically, the hub is able to detect a subset of the errors that are detectable in CAN: stuff error, format error and CRC error. Notice that although a receiving CAN node can also detect a bit error, this type of error is not included in this subset. The hub could also detect it by just monitoring the downlinks. However this monitoring is not required because any error in the downlink actually desynchronizes the corresponding node at frame level, which implies that, sooner or later, this node will cause a stuff error, a format error or a CRC error in the *resultant frame*.

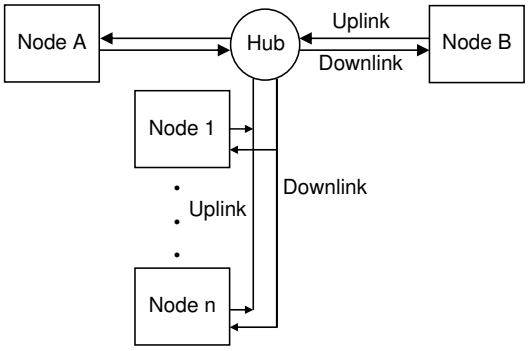

Figure 5.4: Hybrid topology combining CANcentrate and CAN

Finally, note that the error globalization mechanism provided by the Error Flag Generator allows the hub to abort the transmission of a frame at any moment. Therefore, it allows inclusion of further fault-treatment mechanisms, which are not included in the CAN protocol, e.g. to abort the transmission of a forged message due to a masquerading fault. Nevertheless, these issues will be addressed in future work.

5.6 Considerations on the cabling and bit rate

The cost and the length of the cabling, as well as the achievable bit rate, are important factors in distributed embedded systems. Star topologies generally lead to longer cabling than corresponding bus topologies, and thus higher costs, but not necessarily. In fact, the gains or losses in cabling length are highly dependent on the network physical layout. Moreover, the benefits of dependability that can be obtained when using star topologies could be a good reason for choosing them when dependability is an issue.

However, a flexible approach would be the best choice, which combines dependable mechanisms only where needed with less expensive mechanisms where dependability is not an issue. CANcentrate allows this kind of flexibility because it brings the possibility of setting up an *hybrid topology* combining a bus and a star topology (Figure 5.4). This can be done by connecting a set of nodes with lower dependability requirements to one hub port, sharing the same uplink and downlink.

The second important factor regarding the cabling is the achievable bit rate. As explained in Section 3.2, in CAN, due to the synchronization at the bit level among

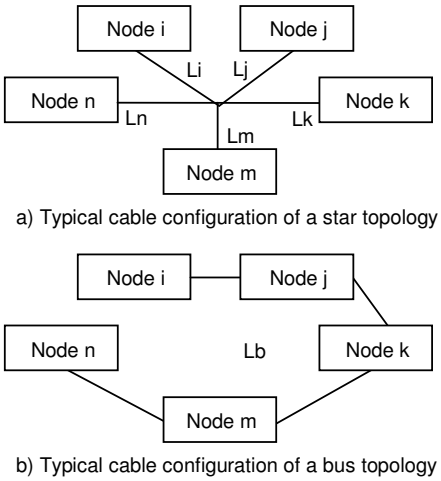

Figure 5.5: Comparison between cabling lengths in a star and in a bus

all nodes, there is an inverse relationship between the bit rate and the maximum bus length [ISO93]. In CANcentrate, these relationship is preserved as the bit level synchronization of CAN is maintained.

However, since the signals travel to the hub and then in parallel in all links back to the nodes, the maximum length applies only to every pair of links. This feature may represent a substantial increase in the capacity to interconnect nodes when compared with the bus topology. To better understand this issue, imagine a system with N nodes separated in space, as depicted in Figure 5.5. The total length of the bus that interconnects such nodes is Lb (Figure 5.5b). On the other hand, consider all nodes interconnected by means of a hub with link i having length L_i (Figure 5.5a). Despite depending on the nodes placement, for the general case, $Lb >> L_i + L_j, \forall_{i,j}$. This is a major benefit of the star topology. Obviously, also for the general case, $Lb < \sum_i(L_i)$ meaning that the total length of the cabling system is longer in the star topology. But the superior connectivity of the star may allow using higher bit rates than with a bus due to the stronger limitation on the bus length.

Finally, let us compare the relationship between the bit rate and the maximum separation between two nodes in a bus and in a star topology. Regarding the length of each star link, the bit level synchronization imposes a limitation on the sum of the lengths of every pair, as stated above. Let this limitation be $Lmax_s$, the *star diameter*, i.e. the sum of the lengths of the two longest links in the star. In order to have the lengths of all links independent of each other, the previous constraint

implies that $\forall_i L_i < Lmax_s/2$.

To derive $Lmax_s$, the maximum diameter of the star, we need to analyze the propagation of the electrical signals from end-to-end. With respect to a bus topology, the star presents an extra delay caused by the hub (additional transceivers and internal gates). This delay is dominated by the former factor since the gate delays are negligible (order of 1 ns or less using modern technologies) when compared with the transceiver delay (approximately 150 ns for fast transceivers, including bus to reception pin and transmission pin to bus [Inf02]). For a given bit rate B, the bit time $1/B$ has now to account for both propagation effects as in a bus plus hub delay. For the former aspect, consider all the parts that contribute to establish the bit time in CAN using the normal bus topology. Let this be t_{pb} (notice that $t_{pb} = 1/B$ by definition). In a star, all these parts related to propagation effects also have to be considered, taking t_{ps}. However, the bit time now also includes the hub delay t_h, thus $t_{ps} = 1/B - t_h$. Note that since a signal must go through the hub two times (from the transmitting node to the receiving node and viceversa), t_h includes twice the time a signal is delayed when crossing the hub in one way.

Therefore, from the point of view of signal transmission, we can define a star equivalent bus, with propagation effects taking t_{ps} and operating at a bit rate B' so that:

$$B' = \frac{1}{t_{ps}} = \frac{1}{1/B - t_h} = \frac{B}{1 - B \cdot t_h} > B \tag{5.1}$$

The previous equation shows that a star is, from an electrical signal transmission point of view, equivalent to a bus operating at a higher bit rate. Moreover, the higher the bit rate, the larger the difference. Therefore, the maximum diameter of the star $Lmax_s$, operating at bit rate B, is the maximum length of standard CAN operating at bit rate B'. For example, given $t_h = 2 \cdot 150 = 300$ ns (according to the figure of hub delay referred above), a star operating at $B = 1$ Mbit/s has a maximum diameter equal to the length of a bus operating at 1.43 Mbit/s. On the other hand, if $B = 125$ Kbit/s then the maximum diameter of the star equals the length of a bus operating at 129.9 Kbit/s which implies a negligible reduction in length. To calculate the effective bus length for these transmission rates refer to [CiAa]

5.7 Conclusions

CANcentrate is a star topology that includes an active hub provided with enhanced fault-treatment mechanisms. Each node is connected to the hub by means of a dedicated link containing an uplink and a downlink. Each node and its link can be considered as region or a hub port that can be isolated to prevent error propagation.

The hub is made up of three modules: the Input/Output Module, the Coupler Module and the Fault-Treatment Module. The Input/Output Module is composed of a number of transceivers that convert the physical signal received from each hub port into a logical value that the hub can process, and that translate the resultant coupled signal to a physical signal that is broadcast to every node.

The Coupler Module substitutes the wired-AND functionality of the CAN bus by a logical AND gate and includes a number of OR gates, one per link, which allow the hub to disable the contribution to the global AND from a specific uplink. The coupling of every hub port is performed in a fraction of the bit time. This allows preserving the CAN low level properties, i.e. the dominant/recessive transmission and the in-bit response, as well as the rest of mechanisms of CAN. Moreover, as a consequence of this compatibility with the standard CAN specification, off-the-shelf CAN components can be used in the nodes of CANcentrate.

The hub evaluates each port independently to detect errors. For this purpose, it includes a dedicated Enabling/Disabling Unit per port within the Fault-Treatment Module. Each one of these units is supplied with information about the meaning of each bit that is being broadcast. Basically, this information specifies the frame field the bit belongs to. The Enabling/Disabling unit uses this information to evaluate whether or not the contribution received from its corresponding hub port is correct. The independence among the Enabling/Disabling units allows easily adding a new hub port. This can be done by basically including a new Enabling/Disabling Unit in the fault treatment module and its corresponding OR gate in the Coupler Module.

In order to supply the Enabling/Disabling units with the information that describes the meaning of each bit that is being broadcast, the Fault-Treatment Module includes the Rx_CAN Module. This module acts as a CAN receiver, synchronizing with the resultant coupled signal at bit level and at frame level, so that it agrees with all nodes about the location of each bit inside the frame. To enforce this agreement even in the presence of errors in the channel, the Rx_CAN Module is also able to detect the same kind of errors as a standard CAN receiver and to trigger the transmission of an active error frame.

Finally, regarding the cabling of CANcentrate, two key factors have to be considered in distributed embedded systems: its cost/length and the achievable bit

rate. Star topologies generally lead to longer cabling and higher costs than corresponding bus topologies, but not necessarily. In fact, the gains or losses in cabling length are highly dependent on the network physical layout. Moreover, CANcentrate allows building hybrid topologies by, for instance, attaching to a hub port a set of CAN nodes that are interconnected by means of a bus topology. Since star topologies can yield dependability benefits when compared with bus topologies, star topologies could be the choice when dependability is an issue.

Regarding the achievable bit rate, CANcentrate preserves the inverse relationship between the bit rate and the maximum separation between two nodes. This is because it maintains the bit level synchronization of CAN. Fortunately, in CANcentrate such compromise applies to its diameter only. However, the presence of the hub causes an extra delay that must be taken into account when dimensioning the bit time. In practice, CANcentrate will require a longer bit time for a given diameter than a CAN bus with an equivalent length. This means that the maximum bit rate attainable in the star is lower than that achievable in the bus. To minimize such a difference, it is important to use fast transceivers in the hub ports.

Chapter 6

CANcentrate error-detection and fault-treatment mechanisms

6.1 Introduction

As we have already explained, CANcentrate aims at improving dependability of CAN networks by providing enhanced error-containment capabilities. To achieve this, the hub of CANcentrate incorporates error-detection and fault-treatment mechanisms that allow it to deal with errors and to contain them at their port of origin. These mechanisms are implemented within the Fault-Treatment Module (see Section 5.4).

In Sections 5.4 and 5.5 we explained that errors are detected at two different levels. On the one hand, the Rx_CAN Module includes most of the error-detection mechanisms of CAN in order to detect errors at the coupled signal. Moreover, it also includes the CAN error-signaling mechanism to participate with all other nodes in recovering when an error at the coupled signal occurs. This recovery basically consists in forcing the hub and all nodes to re-synchronize at bit and at frame level.

On the other hand, the hub also detects errors at each hub port. Specifically, each hub port is monitored by a dedicated Enabling/Disabling Unit within the Fault-Treatment Module. A given Enabling/Disabling Unit aims at detecting errors at its corresponding hub port in order to diagnose if that port is faulty. When this occurs, the Enabling/Disabling Unit passivates the fault by driving a logical '1' at the corresponding Enabling/Disabling signal (ED_i), which effectively isolates the port contribution, as indicated in Section 5.4.

Current chapter is devoted to discussing the basics of how the Enabling/Disabling Unit detects errors and treats faults. Details concerning the way in which errors at the coupled signal are processed were explained in previous Sections 5.4 and 5.5.

6.2 Error-detection and fault-treatment rationale

The Enabling/Disabling Unit is able to deal with all faults included in our fault model, which is specified in Section 5.2: stuck-at-dominant, stuck-at-recessive and bit-flipping faults. Note that in the present chapter we will not refer to network partition faults. As explained in Section 3.4, in a bus topology, this type of fault is provoked by a physical disruption affecting the medium. However, a physical disruption affecting the media of a simplex star topology, e.g. occurring at an uplink, cannot provoke a network partition, since each node has its own dedicated connection to the hub. In other words, a simplex star topology inherently prevents network partitions from happening. Moreover, a physical disruption can also lead the medium to be stuck-at or bit-flipping. For instance, signal reflections at an open extremity of a link may cause channel errors and, hence, manifest as a bit-flipping fault. Therefore, we can say that the hub indirectly treats physical disruptions when dealing with stuck-at and bit-flipping faults.

The internals of the Enabling/Disabling Unit are shown in Figure 6.1. On the one hand, each Enabling/Disabling Unit has a dedicated *event counter* and an associated *manager module* for each type of fault that must be detected. A given event counter is used to count up suspicious situations in which the corresponding hub port is possibly behaving incorrectly due to a specific type of fault. In fact, one can consider the event counter as a kind of error counter. Regarding a given manager module, it is the responsible for detecting suspicious situations at a given hub port (for detecting possible errors) and, then, for deciding how to increase/decrease its associated event counter. For that, the manager module basically analyzes the coupled signal, B_0, the port contribution B_i, as well as the Current State signals (CS) from Rx_CAN. The manager module makes its decisions depending on the type of fault it is responsible to deal with.

More specifically, the Enabling/Disabling Unit includes the following event counters and their corresponding manager modules: the *Dominant Bit Counter* (DBC) and the *DBC Manager Module* for stuck-at-dominant faults; the *Non-Acknowledge Counter* (NACKC) and the *NACKC Manager Module* for the stuck-at-recessive; and the *Bit-Flipping Counter* (BFC) and the *BFC Manager Module* for the bit-flipping faults.

On the other hand, the Enabling/Disabling Unit includes a *Threshold Control*

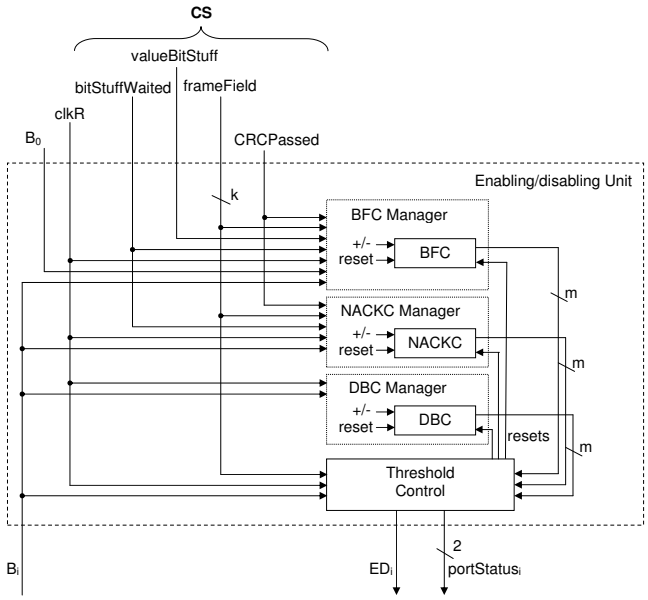

Figure 6.1: Internals of the Enabling/Disabling Unit

Module that is aimed at declaring the port as faulty when it corresponds and, then, at isolating its contribution. The Threshold Control Module takes into account the value registered by each event counter and is programmed with a specific threshold for each one of them: the *Dominant Bit Threshold* (DBT), the *Non-Acknowledge Threshold* (NACKT) and the *Bit-Flipping Threshold* (BFT). Whenever any of the event counters exceeds its corresponding threshold, the Threshold Control Module isolates the port contribution by setting the corresponding ED_i signal to '1'.

However, in order to increase the tolerance to transient faults, the Threshold Control Module may use a specific *reintegration policy* to re-enable the port contribution and to allow the operation of all managers again, after a given period of *inactivity* is observed at the port. This reintegration policy will be explained in Section 6.7.

Finally notice that, as explained in Section 3.3.4, although the CAN protocol is supposed to ensure data consistency, it actually cannot enforce it if a CAN node is in the error-passive state. This is because an error-passive node cannot always force the globalization of an error. In order to mitigate this problem, the Enabling/Disabling units consider the behavior of error-passive nodes as incorrect (see Section 5.5). This will eventually lead to the isolation of any hub port corre-

sponding to an error-passive node.

6.3 Current State signals for the Enabling/Disabling units

As indicated in Section 5.4 and in Figure 5.3, the Rx_CAN Module, besides being responsible for ensuring the synchronization at frame level, also provides the Enabling/Disabling units with a set of signals called Current State signals, CS, that gathers all the information that, together with B_0, is needed to know the meaning of the bit that is currently broadcast to all nodes, i.e. the current state of the *resultant frame*. Next, these signals are explained (see Figure 6.1). Nevertheless, the reason why these are the signals required by the Enabling/Disabling units for describing the current state of the *resultant frame* will be more easily understood later on in Chapter 7.

First, the meaning of a bit takes into account whether or not it is a stuff bit. Signal *bitStuffWaited* indicates this. Additionally, in case it is a stuff bit, the signal *valueBitStuff* specifies the expected correct bit value according to the stuff rule.

Second, the type of frame and the specific field of the frame in which the bit is located also determines the meaning of the bit. As explained in Section 3.3.1, CAN specifies four kinds of frames: data frames, remote frames, error frames and overload frames. However, we have re-defined the types of frames for practical reasons.

- *Data frame*. This type of frame coincides with the data frame specified in the CAN standard [ISO93]. It is the frame a node uses to transmit data.

- *Remote frame*. This kind of frame is also included in the CAN standard. It does not carry data, but a node uses it to request for the transmission of a specific data frame from another node.

- *Overload frame*. This type of frame is specified in the CAN standard as well. An overload frame is transmitted to achieve an extra delay between two different data or remote frames.

- *Active error frame*. This frame, which is not defined in the CAN standard, is the one that results when error-active nodes signal an error. It is composed of the superposition of the active error flags sent by these nodes (and by the hub in the case of CANcentrate), followed by the error delimiter they all cooperatively transmit.

- *Passive error frame*. This frame is not included in the CAN standard. We defined it as being the one that results when error-passive nodes signal an error. It is composed of the superposition of the passive error flags sent by these nodes, followed by the error delimiter they cooperatively transmit.

- *Inter frame*. This type of frame is also specified for practical reasons. We defined it to embrace the bits that form part of the *Intermission Frame Space* (IFS) and the *Idle* period [ISO93]. As explained in Section 3.3.1, the CAN standard specifies the IFS as a sequence of three consecutive recessive bits that follows every frame. After the IFS the channel remains at a recessive value (Idle) until a dominant bit is observed at it as a consequence of an error or because of a node starts transmitting a frame. In this way, our inter frame consists of two fields: the *Intermission field*, which is composed of three consecutive recessive bits, and the *Idle field*, which comprises no bit or some consecutive recessive bits.

The kind of frame and the specific frame field the bit that is being broadcast belongs to is codified by means of the vector of signals (a signal of k bits) called *frameField*.

Finally, as will be explained later on in Sections 7.3.1, 7.3.2, 7.4.2 and 7.4.1, some error-detection mechanisms included in the Enabling/Disabling units also need to know whether or not the frame sent through the hub has passed the CRC check (this check is performed by the Rx_CAN Module, see Section 5.5). This information is provided by means of the *CRCPassed* signal.

6.4 Stuck-at-recessive faults

Due to the AND function that the hub implements, a port suffering a stuck-at-recessive fault does not interfere the communication among the rest of the nodes in the star. Therefore, this kind of fault does not generate a severe failure of the communication system. Nevertheless, detection of such faults may still be useful in order to implement additional fault-tolerance mechanisms at higher levels of the system architecture, for example to detect a crashed or absent node.

The detection of stuck-at-recessive faults poses an additional difficulty because a CAN node may be without transmitting, which actually means sending recessive values, for a long time. Therefore, it would be theoretically impossible to differentiate between a stuck-at recessive node and an operational but non-transmitting node. Nevertheless, the CAN protocol specifies that every CAN controller must

transmit a dominant bit in the ACK slot of every frame it correctly receives, i.e. the ACK bit (see Section 3.3.1). Therefore, the absence of this bit can be used to detect stuck-at-recessive ports.

For each port, such detection is carried out by a specific NACKC Manager Module. Whenever the NACKC Manager detects, thanks to the CS signals, that the current state of the *resultant frame* is the ACK slot and that the frame has passed the CRC check, it checks in B_i if the node is sending a dominant value to acknowledge the frame. If this dominant value is not sent, then the NACKC Manager increases the NACKC (Non-Acknowledge Counter).

The NACKC Manager decreases the NACKC whenever a dominant bit is issued through the port. It is important to note that by decreasing the counter, instead of resetting it when detecting a dominant bit value, the hub can detect not only stuck-at-recessive failures, but also nodes that tend to be stuck-at-recessive.

When the NACKC exceeds the Non-Acknowledge Threshold (NACKT), the corresponding Threshold Control Module does not isolate the port, but it merely notifies the user about the inactivity of the port by means of a LED. The specific value for the NACKT can be configured depending on how strict we want to be when considering a port as being stuck-at-recessive. For instance, since an error-active node should send an active error flag after omitting an ACK bit, even a NACKT value equal to 2 can be considered if we want to be very strict when detecting a crashed or absent node.

6.5 Stuck-at-dominant faults

In order to detect a stuck-at-dominant fault at a given hub port, the corresponding DBC Manager Module counts the number of consecutive dominant bits that are received through that port (through the uplink of that port). The DBC Manager increases the corresponding DBC in one unit each time it observes a dominant bit value, and it resets that DBC as soon as it observes a recessive bit.

The associated Threshold Control Module compares the value of the DBC with the Dominant Bit Threshold (DBT). Whenever the DBC exceeds the DBT, the Threshold Control Module isolates the port.

The DBT is configured in order to maximize the chances to differentiate between situations in which a stuck-at-dominant fault really exists and situations in which the channel is occupied by many consecutive dominant bits, although there is not a stuck-at-dominant fault. The maximum number of allowed consecutive dominant bits before diagnosing a stuck-at-dominant fault takes into account two different

contributions:

$$\text{DBT} = (T_{\mathit{stuff}} + 1) + N \cdot T_{\mathit{errorFlag}}$$

The first term, $T_{\mathit{stuff}}+1$, specifies the minimum number of consecutive dominant bits that violates the stuffing rule in a CAN network (6 bits). This term includes the maximum number of consecutive dominant bits allowed in CAN, T_{stuff}, plus the additional dominant bit needed for violating the stuff rule. Whenever the stuff rule is violated, it is expected that all nodes start to send an active error flag immediately after this error occurs. Nevertheless, it is possible that a node detects a second error during its own error flag and restarts the transmission of the active error flag, thereby prolonging the sequence of consecutive dominant bits. In the worst case, a node will see this second error in the last bit of its first error flag, and will send a consecutive active error flag. The second term, $N \cdot T_{\mathit{errorFlag}}$, is intended to covering these situations. It specifies the maximum number of consecutive dominant bits that are considered as overlapped or consecutive active error flags. In other words, it indicates the maximum amount of time a node is allowed to transmit overlapped or consecutive error flags, measured in number of bits.

Note that for $N = 2$ the threshold coincides with the one proposed in [RVA99]. In that case, the threshold can be exceeded if two additional errors occur in the error flag that follows a violation of the stuff rule, leading to an erroneous diagnosis of a stuck-at-dominant fault. Using a higher value of N reduces the probability of performing an erroneous stuck-at-dominant diagnosis.

The value of N can be configured depending on the application. For instance, in a hazarding environment, we may consider that $N = 4$ is tolerant enough and does not imply a significant loss of reactivity in diagnosing stuck-at-dominant faults.

6.6 Bit-flipping faults

As said before, a bit-flipping fault occurs whenever a component of the network sends erroneous and random bits with no restrictions in the value domain. From the hub point of view, such kind of fault manifests as any of its ports receiving too many arbitrarily erroneous sequences of bits.

The CAN standard specifies some fault-treatment mechanisms that can be used to diagnose and passivate bit-flipping faults occurring at nodes. These mechanisms were briefly described in Section 3.3.5. Basically, each CAN node includes a *Transmission Error Counter* (TEC) and a *Reception Error Counter* (REC). These counters are increased and decreased following some rules established in the CAN

specification. When any of these counters exceeds a given threshold, the corresponding CAN node reduces its impact on the communication process by going into the *error-passive* state. Moreover, a CAN node may disconnect itself from the network by entering into the *bus-off* state [ISO93] if a second threshold is also exceeded. These fault-treatment mechanisms are aimed at preventing further propagation of local errors. See Section 3.3.5 for a further explanation about the error-passive and bus-off states.

Nevertheless, these mechanisms based on the TEC/REC included in each CAN node present some deficiencies that make little advisable for the hub to rely on these mechanisms for achieving error containment. First, normal CAN nodes can fail in arbitrary ways and for this reason may stop performing fault confinement. Second, if a medium is the source of bit-flipping faults affecting all nodes, it cannot be isolated by the nodes. Finally, the accuracy of the error detection strategy followed by the TEC/REC is limited by the restricted vision that the bus imposes, in which the contributions of all nodes are mixed. Thus, we decided to implement in each Enabling/Disabling Unit a dedicated Bit-Flipping Counter (BFC) and its associated BFC Manager Module.

Each BFC Manager is aimed at detecting errors in its port contribution. The BFC Manager increases and decreases its BFC depending on the errors it detects. Whenever the BFC exceeds a given Bit-Flipping Threshold (BFT), the Threshold Control Module diagnoses its port as being faulty and isolates it by means of the corresponding ED signal.

The BFC Manager evaluates the correctness of its corresponding port by checking whether or not its contribution (B_i) deviates from the expected behavior according to the current state of the resultant frame (CS and B_0). Because of the behavior of a CAN node is quite complex and a bit-flipping fault can provoke a huge number of error scenarios, the details of the error-detection mechanisms implemented by the BFC Manager are thoroughly explained in Chapter 7. Additionally, the specific values for increasing and decreasing the BFC, as well as the value for the BFT are addressed in Section 8.5.

6.7 Reintegration policy

In Section 6.2 it was explained that each Enabling/Disabling Unit has a Threshold Control Module that isolates its port contribution when detects that any of its event counters exceeds a specific threshold. However, it was also said that in order to increase the tolerance to transient faults, the hub implements an automatic reintegration policy of isolated ports.

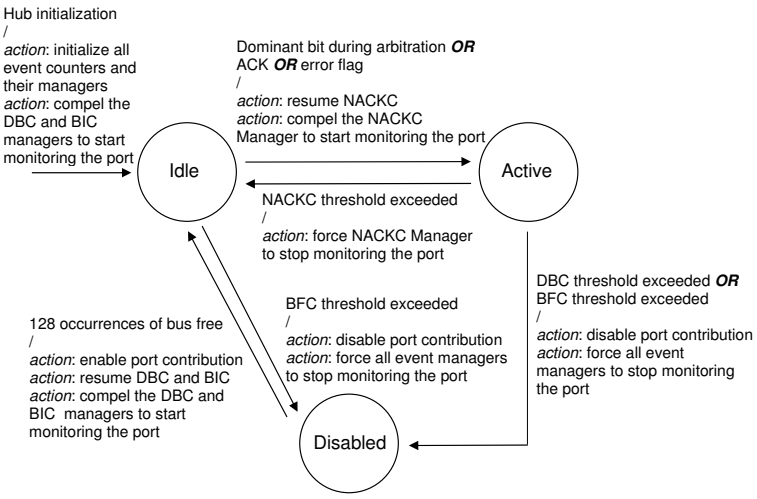

Figure 6.2: Reintegration policy schema of CANcentrate

The reintegration mechanism is implemented by the Threshold Control Module, and it basically consists in re-enabling the contribution of any port after a given period of *inactivity* is observed at the port. The state machine that describes this reintegration policy is depicted in Figure 6.2.

When the hub is initialized, each Threshold Control Module sets its port to the *idle* state. During this state the port contribution is enabled. Notice that as long as the hub port is in idle, the DBC and the BIC managers monitor its contribution in order to detect possible errors. However, the NACK Manager does not. This is because and idle port is not considered as being regularly participating in the communication process and, thus, it is not expected to receive an ACK bit through it.

As soon as the hub receives a *meaningful* contribution from a port, e.g. a dominant bit contribution during the arbitration, an ACK bit, or an error flag transmission, the corresponding Threshold Control Module sets that port to the *active* state. The only difference between the idle and active states is that the second one indicates that the node is regularly participating in the communication process. Therefore, the NACK Manager is compelled to start monitoring the hub port whenever it becomes active. Also notice that if the hub port is diagnosed as being stuck-at-recessive during the active state, the hub port returns to idle and, thus, the NACK Manager must stop monitoring the port.

Whenever the stuck-at-dominant or the bit-flipping thresholds are exceeded, the

Threshold Control Module sets the port to the *disabled* state, regardless the state of the port at that moment. To be in this state actually implies that the contribution of the port is disabled, and that the managers stop monitoring the port.

Once a port is in the disabled state, the corresponding Threshold Control Module waits to observe a constant recessive contribution during 128 CAN bus-free occurrences, i.e. 128 occurrences of 11 consecutive recessive bits. This coincides with the number of consecutive recessive bits that a CAN node in the bus-off state must observe before being able to re-enter in the error-active state (see Section 3.3.5). After detecting this period of *inactivity*, the port will be set again to the idle state; its contribution will be re-enabled; the DBC and BIC will resume, and their managers will start monitoring the port again.

The reintegration policy allows an autonomous performance of the hub because it is able to return to normal operation by itself.

6.8 Conclusions

This chapter discuses the basics of how each Enabling/Disabling Unit detects errors and treats faults to enhance error containment in a CAN network.

In order to be able to detect errors at its port contribution, the Enabling/Disabling Unit needs to know the meaning of the bit that is currently being broadcast to all nodes, i.e. the current state of the *resultant frame*. It gets this information from a set of signals, CS, which are provided by the Rx_CAN Module, and from the current bit value of the coupled signal. The set of signals CS basically includes information about whether or not the bit being broadcast is a stuff bit, as well as about the type of frame and the frame field this bit belongs to. It is important to note, at this point, that every bit of the stream that is being broadcast can be considered as forming part of a frame. Therefore, for practical reasons, we consider the interfame space as a type of frame.

The Enabling/Disabling Unit copes with all the faults that are included in our fault model, even though it only explicitly detects and treats stuck-at-recessive, stuck-at-dominant and bit-flipping faults. This is because a simplex star topology inherently avoids network partitions and, in the worst case, a physical disruption at a given point of the media manifests as a stuck-at or as a bit-flipping fault at the corresponding hub port.

Stuck-at-recessive, stuck-at-dominant and bit-flipping faults are addressed separately. For each one of these types of faults, the Enabling/Disabling Unit includes an event counter and an associated event counter manager. The manager monitors

the hub port in order to detect suspicious situations in which the port is possibly behaving incorrectly, i.e. to detect possible errors; and it increases/decreases its associated event counter following specific rules.

In addition, the Enabling/Disabling Unit is provided with a Threshold Control Module, which takes into account a different threshold for each type of fault. Whenever the Threshold Control Module detects that the number of detected possible errors related to either a stuck-at-dominant or a bit-flipping fault exceeds the corresponding threshold, it diagnoses the port as being faulty and isolates it by setting the appropriate ED_i signal to '1'. In contrast, the Threshold Control Module does not isolate the port if it diagnoses it as being stuck-at-recessive. This is because a stuck-at-recessive port does not generate errors that can propagate to other ports.

The detection of stuck-at-recessive faults poses an additional difficulty because a CAN node may be without transmitting, which actually means sending recessive values, for a long time. We propose a simple but effective solution to overcome this problem. It consist in taking advantage of the fact that in CAN every operational receiving node contributes to all frames in transmission with a dominant bit, the ACK bit, within the ACK slot. In this way, a port is diagnosed as stuck-at-recessive when it has omitted a predefined number of ACK bits.

A stuck-at-dominant fault is easily detected by counting up the number of consecutive dominant bits issued through the port. However, special care must be taken not to confuse a stuck-at-dominant port with a port occupied by many consecutive dominant bits, e.g. with a port that receives many overlapped or consecutive active error flags. Specifically, we propose to consider that the maximum number of allowed consecutive dominant bits depends on the maximum number of consecutive active error flags that are expected in the network.

We have seen that to deal with suck-at faults is quite simple. However, the detection of a bit-flipping fault is much more complicated, since such a fault can lead to the corresponding manager module to observe a huge number of error scenarios at the hub port. This justifies the presence of Chapters 7 and 8, which are entirely dedicated to discuss the details of the error-detection and fault-diagnosis mechanisms for dealing with bit-flipping faults respectively.

Finally, we have also described the reintegration policy that each Threshold Control Module performs in order to increase the tolerance to transient faults. This policy basically consists in re-enabling the contribution of the port after it exhibits a given period of *inactivity*. The reintegration policy allows an autonomous performance of the hub because it is able to return to normal operation by itself.

Chapter 7

CANcentrate mechanisms for detecting bit-flipping errors

7.1 Introduction

As explained in Chapter 6, a bit-flipping fault can manifest itself in a huge number of different manners. On the one hand, this implies that the mechanisms responsible for detecting bit-flipping errors must try to encompass this wide range of error scenarios. On the other hand, some difficulties arise for diagnosing a port as being bit-flipping.

The present chapter is devoted to describing the details of the error-detection mechanisms for dealing with bit-flipping faults; whereas a deep discussion of the fault-diagnosis mechanisms will be provided in Chapter 8.

Notice that, as indicated before, the error-detection and fault-diagnosis mechanisms are included in each Enabling/Disabling Unit, which operate independently at each hub port. Particularly, within the Enabling/Disabling Unit, the *Bit-Flipping Counter Manager* (BFC Manager) is the responsible for detecting bit-flipping errors. This module counts up these errors using its corresponding *Bit-Flipping Counter* (BFC).

7.2 Bit-flipping error-detection rationale

As explained in Section 6.6, each BFC Manager evaluates the correctness of its corresponding hub port by checking that its contribution does not deviate from

the correct behavior that is expected according to the current state of the *resultant frame*. For instance, the BFC Manager does not expect to receive a dominant bit through its port if that port corresponds to a receiving node, unless the current bit that is being broadcast is the ACK bit and the frame that is being broadcast has passed the CRC check. Other illustrative example is that the BFC Manager expects that the stream received through its port correctly follows the stuff rule, in the case its hub port corresponds to the transmitting node.

Concerning the concepts of transmitting and receiving node, notice that these are the two possible roles that a CAN node can play during communication. On the one hand, it is considered that a node is a receiving node as long as it does not attempt to transmit. This means that a node acts as a receiver during the intermission field, the idle field, as well as during the frame any other node is transmitting. On the other hand, whenever the node attempts to transmit, i.e. when it sends a Start Of Frame (SOF), it is considered as a transmitter in principle. The node keeps this role until it ends transmitting, i.e. until the first bit of the intermission field, unless it loses the arbitration. Concerning this last possibility, notice that, as explained in Section 3.3.2, any conflict that arises when more than one node tries to gain access to the medium for transmitting is resolved by the CAN bit-wise arbitration mechanism. Thus if a transmitting node loses the arbitration, it is immediately considered as a receiver.

In the case of CANcentrate, the BFC Manager assumes that its corresponding node has lost the arbitration and becomes a receiver whenever its bit contribution is recessive and the resultant coupled signal has a dominant value. Under normal circumstances, after the arbitration phase only one node considers itself as the transmitter and only the BFC Manager corresponding to such node considers it to be the transmitter. However, this may be not ensured in the presence of a special type of error scenarios, which will be explained later on in Section 8.4.

Notice that since a node together with its corresponding link are considered as an error-containment region (see Section 5.1), the BFC Manager assumes that the bit issued through the port is the bit the node wishes to send. Hence, from now on we will talk about the behavior of a hub port and the behavior of a node interchangeably.

Although the behavior of a CAN node is quite complex in the general case, we have been able to identify five independent types of behaviors:

- The behavior of a transmitting node during a data frame and a remote frame in which no error has been detected so far, i.e. during normal transmission (see Section 6.3 for a further explanation about each type of frame).

- The behavior of a receiving node during a data frame, a remote frame and an inter frame[1] in which no error has been detected so far, i.e. during normal transmission.
- The behavior of a node upon an error condition is detected.
- The behavior of a node during an error signalling.
- The behavior of a node during an overload signalling.

It is very important to note here that when we say that an error condition is detected, we refer again to errors that are detected at two different levels. As it was explained in Sections 5.4 and 5.5 and, afterwards, again in Section 6.1, the first level corresponds to errors detected in the coupled signal, i.e. in the *resultant frame*, whereas the second one refers to errors detected in a port contribution.

It is important to highlight that a BFC Manager does not monitor the contribution of other hub ports. Therefore, it only becomes aware of an error that happens in other port when that error affects the *resultant frame*. Notice that it would not have sense that the BFC Manager knows that other port is behaving incorrectly if the error generated by that port does not propagate to the *resultant frame*. This is because a CAN node will not react to an error issued by other node if that error does not affect the correctness of the *resultant frame*. In contrast, the BFC Manager must detect any error affecting its hub port to forecast what is going to be the contribution of its node, independently of whether or not that error affects the *resultant frame*.

Next subsections are devoted to describing the rules the BFC Manager follows to detect that its CAN node does not behave as expected. First, we will focus on how the BFC Manager detects that its node behaves incorrectly before an error is detected. Then, we will address the way in which the BFC Manager checks the contribution of its node upon detection of an error or an overload.

7.3 Error detection during normal transmission

This section describes how the BFC Manager checks whether or not its node behaves correctly as long as an error has not been detected so far in the *resultant frame* or in its port contribution.

We differentiate between the way in which the BFC Manager evaluates its node contribution when that node is acting as the transmitter or as the receiver.

[1] Note that all the nodes are considered as receivers during the intermission frame space and the idle period.

To evaluate the correctness of the contribution of the transmitter is quite complex since it is rather free to send the bit values it wishes. Fortunately, the BFC Manager can adapt most of the error-detection mechanisms of the CAN protocol to detect errors in the contribution of the transmitting node. Contrary to the case of a transmitting node, to detect errors on the contribution of a receiving node is easier. This is because a receiver is only allowed to send recessive bits, except in particular locations of the frame and under specific circumstances. Thus, the BFC Manager of a receiving node basically checks that its port only issues a dominant bit when allowed.

7.3.1 Error detection on the transmitter contribution

The first type of errors the BFC Manager can detect is based on the expected behavior of the transmitting node during a data frame or a remote frame in which no error has been detected so far.

The error detection for evaluating this type of behavior is not trivial because although a transmitting node must respect the restrictions imposed by the CAN standard, it is rather free to send the bit values it wishes. Fortunately, the most complete set of error-detection mechanisms available for detecting errors in the contribution of a transmitter is already specified in the CAN protocol [ISO93]. These mechanisms are: *stuff rule check*, *frame check*, *monitoring*, *CRC check* and *ACK check* (see Section 3.3.4). However, notice that these error-detection mechanisms are based on the observation of the *resultant frame*, in which the contribution of the transmitter is mixed with the contributions of all the other nodes. Therefore, the BFC Manager needs to adapt these error-detection mechanisms in order to directly detect errors in the contribution of the transmitting node.

Note that to implement the referred CAN error-detection mechanisms, a typical node needs to observe the bit value it wishes to send and the current value on the bus. By observing the bus, the CAN node is able to know which is the resultant bit in the bus (the bus acts as a logical AND gate), as well as to calculate, bit by bit, the current state of the *resultant frame*.

Thus, similarly, the BFC Manager observes its corresponding port's contribution, B_i, the coupled signal B_0, and the set of signals CS provided by the Rx_CAN Module, in order to detect errors. The B_i signal allows the BFC Manager to know which is the value of the bit sent by the node. Additionally the BFC Manager uses the signal B_0 to know the value of the resultant bit that (as in a bus) all the nodes should see. Finally, the set of signals CS complete the description of the current state of the *resultant frame*.

Also note that since after the arbitration phase only one node is considered as the transmitter (see Section 7.2), only one BFC Manager will consider its corresponding contribution as the transmitting node's contribution during the rest of the data or remote frame. Next, it is specifically explained how the BFC Manager corresponding to the transmitting node adapts each CAN error-detection mechanism.

(1) *Stuff rule check*. The BFC Manager corresponding to the transmitting node performs a *stuff rule check* on the stream sent by the transmitter. The signal *bitStuffWaited* included in CS indicates whether the current bit of the *resultant frame* is a stuff bit or not, whereas the signal *valueBitStuff* included in CS indicates the correct bit value expected for fulfilling the stuff rule. Whenever the signal *bitStuffWaited* indicates that the current bit is a stuff bit, the BFC Manager corresponding to the transmitter checks if the stuff rule is fulfilled by analyzing whether the bit issued through its corresponding port, B_i, matches with the bit value indicated in the signal *valueBitStuff*.

(2) *Frame check*. The BFC Manager corresponding to the transmitter checks that the transmitter's contribution respects the frame format (during a data or a remote frame) specified in the CAN protocol, i.e. the BFC Manager performs a *frame check* on the transmitter's contribution (see Section 6.3 for a further explanation of the different types of frames). For data and remote frames, the transmitter is allowed to send dominant bits and recessive bits depending on the current field being transmitted. As concerns the signals involved to perform such *frame check*, the vector *frameField* included in CS indicates which is the kind of frame and the field within the frame the current bit belongs to. The BFC Manager uses its knowledge about the CAN protocol in order to check whether the contribution of its corresponding port, B_i, respects the frame format according to the kind of frame and the field indicated by the vector *frameField*.

(3) *Monitoring*. As explained in Section 3.3.4, the transmitting node in a CAN bus performs a *monitoring* of the *resultant frame* in order to check that whenever it transmits a bit, such bit value is the value seen in the *resultant frame* (except during the arbitration field and in the ACK slot, where it is allowed that a dominant bit overwrites a recessive bit sent by the transmitter). In other words, the transmitting node checks if a bit error has occurred.

Nevertheless, the BFC Manager cannot adapt this mechanism to detect that its hub port provokes a bit error. This is because it cannot be sure about which is the real bit value the node sent nor about whether or not that bit is corrupted in the uplink/downlink of that node.

Fortunately, notice that if a bit error actually occurs, the BFC Manager corresponding to the transmitter will detect, sooner or later within the frame, that the

transmitter had any kind of problem. This is because the transmitting node should signal an error when detecting the bit error. Such error signaling forces the BFC Manager corresponding to the transmitter to detect an error by means of any of the CAN error-detection mechanisms it adapts. In addition, even if the transmitter does not signal the bit error due to extra errors (the transmitter may not detect that the bit value it sent changed when reaching the hub), the BFC Manager will also detect an error later on in the frame. That is because the error-detection mechanisms of the CAN protocol (which the BFC Manager adapts) ensure that an error will be detected if few than 5 errors occur within the same frame [ISO93]. For instance, a bit-error that is not signaled by the transmitter will provoke a CRC error.

(4) *CRC check*. As said before in Section 3.3.4, the transmitting node calculates and sends within the frame a 15-bit Cyclic Redundancy Code based on the bits of the frame it has already transmitted. Note that if the contribution of any port corresponding to a receiving node changes any of the bits that constitute the content of the CRC or that are included in the set of bits from which the CRC is calculated, then the transmitting node should detect such situation as a bit error and should abort the transmission of the frame by signaling an error. Therefore, it is assumed that the responsible for the correctness of the value of the CRC that can be seen in the *resultant frame* is the transmitter. Thus, only the BFC Manager corresponding to the transmitter checks if the CRC is correct in order to identify that its port has behaved as bit-flipping. Regarding the signals involved in the CRC error detection, the signal *CRCPassed* included in CS indicates whether the frame has passed the CRC check performed by the Rx_CAN Module or not. The vector *frameField* included in CS indicates the frame and the field that is currently being transmitted and, indirectly, when the CRC field ends. The BFC Manager corresponding to the transmitter observes the vector *frameField* in order to know when the CRC field ended, and then, checks by means of the signal *CRCPassed* if the frame has passed the *CRC check*.

(5) *ACK check*. In CAN, when no receiving node acknowledges the frame, the transmitter detects an ACK error by means of the *ACK check*. In fact, an ACK error indirectly indicates to the transmitter that the frame it sent probably did not pass the CRC check. Nevertheless, to use the *ACK check* in order to detect a CRC error in the transmitter contribution is not appropriate. An *ACK error* can also occur, even if the contribution of the transmitter is correct, if the receiving nodes (or their links) have any problem and are not able to acknowledge a correct frame. Therefore, to use the ACK error to conclude that the transmitting node is behaving incorrectly would be unfair. Since the aim of the hub is to isolate permanently faulty ports and not to isolate correct ones, the BFC Manager does not use the *ACK check* for detecting CRC errors in the transmitter's contribution. Note that this is

not a problem since the BFC Manager checks if the CRC sent by the transmitter is correct, as just explained above.

7.3.2 Error detection on a receiver contribution

As stated in Section 7.2, the BFC Manager is able to detect a second type of errors, namely errors in the contribution of a receiving node during normal transmission, i.e. during data frames, remote frames and inter frames in which no error has been detected so far.

Contrary to the case of error detection on the contribution of a transmitting node (explained above in Section 7.3.1), the error detection on the contribution of a receiving node is easier. This is because a receiver is only allowed to send recessive bits, except in three cases in which it can send a dominant bit: during the ACK slot within a data and a remote frame if the frame has passed the CRC; in the first bit of the intermission field of an inter frame; and during the idle field within an inter frame (see Section 6.3 for an explanation of the types of frames we consider for detecting errors).

Next, it is explained how the BFC Manager corresponding to a receiving node specifically checks its node contribution during the ACK slot, the intermission field and the idle field.

As concerns the ACK slot, it is important to note that a receiving node must only acknowledge a frame if it has not detected any error (including the CRC error). In the case a receiving node detects an error in the CRC, it does not acknowledge the frame, but signals the error in the first bit of the End Of Frame field (EOF) (see Section 3.3.4). The BFC Manager corresponding to a receiving node expects that the result of the CRC check performed by the receiver is equal to the result of the CRC check performed by the Rx_CAN Module. Therefore, a receiving node that sends a dominant bit during the ACK slot when the frame has not passed the CRC check of Rx_CAN is considered as a bit-flipping error.

In contrast, although a receiving node that does not send a dominant bit during the ACK slot when it should do that is behaving incorrectly, the BFC Manager does not consider, in principle, that omission as a bit-flipping error. This is because the lack of the ACK bit is used for detecting stuck-at-recessive faults (see Section 6.4). Only in the case the node actually starts signaling an error (a CRC error) at the first bit of the EOF, the BFC Manager will consider the ACK omission as a bit-flipping error.

As regards the signals that are involved in the detection of an error in the receiving node contribution during the ACK slot, the vector *fieldFrame* included in

CS indicates which kind of frame and which field within the frame the current bit belongs to, whereas the signal *CRCPassed* indicates whether the frame has passed the CRC check performed by the Rx_CAN Module. When the *fieldFrame* indicates that the current bit belongs to the ACK slot, the BFC Manager corresponding to the receiver will expect its corresponding node to send a dominant bit in the ACK slot if the signal *CRCPassed* indicates that the frame has passed the CRC check; otherwise, it will expect its corresponding receiving node to send a recessive bit in the ACK slot.

Note that since each BFC Manager corresponding to a receiving node independently checks its node contribution, a receiving node omitting the acknowledge can be detected even though in the *resultant frame* the ACK slot has a dominant value. This represents an improvement when compared with CAN, where it is impossible to detect a node omitting an ACK if any other node is acknowledging the frame.

As explained above, besides the case of the ACK slot, other exception in which the BFC Manager corresponding to a receiver allows its node to send a dominant bit is at the first bit of the intermission field. This is because a receiving node can send that bit to trigger an overload signaling (see Section 3.3.6). When this occurs, the BFC Manager checks if the contribution of its node constitutes a correct overload signaling, as will be explained in Section 7.6.

Finally the last case in which a receiving node is allowed to send a dominant bit is during the idle field of the inter frame. During such field, any receiving node[2] that wishes to send a frame starts the transmission of such frame by means of a dominant bit that constitutes the Start Of Frame (SOF) (see Sections 3.3.1 and 7.2). When this occurs, the BFC Manager of the receiving node will not consider it as an error, but instead that the receiving node becomes a transmitter. Hence, after monitoring a SOF through its port, the BFC Manager will apply the error-detection mechanisms for the contribution of a transmitting node described above in Section 7.3.1.

However, the BFC Manager will consider that its corresponding node becomes a receiver again if during the arbitration phase its node loses the arbitration. This happens if the BFC Manager observes that its node sends a recessive bit and at the same time a dominant bit is detected in the coupled signal. When this happens, the BFC Manager checks that its node acts as a receiver during the rest of the frame.

[2]Note again that during intermission and idle all nodes are considered as receiving nodes.

7.4 Error detection upon the occurrence of an error

Section 7.3 described the rules the BFC Manager follows to check if its CAN node behaves correctly as long as an error has not been detected. Current section explains what kind of behavior the BFC Manager expects from its node just after an error is discovered during normal transmission.

Normally, what the BFC Manager will expect is that its node signals the error that has been detected by means of an active error flag. However, there are some exceptions that must be taken into account in order not to incorrectly expect that a node will signal an error. This section is actually devoted to clarifying when the BFC Manager must expect an error signaling upon an error is detected. Details concerning the correct behavior of a node during an error signaling, however, will be addressed in Section 7.5.

At this point, it is important to note again that an error can happen at two different levels: at the *resultant frame* and at any port contribution. Most of the errors that affect the *resultant frame* are identified by the Rx_CAN Module, which is able to detect almost all types of errors that a receiving CAN node would detect (see Section 5.5 for further details). However, as will be explained later, the BFC Manager can also detect errors at the *resultant frame* beyond the capabilities of the Rx_CAN Module and the nodes. On the other hand, an error affecting a given port is discovered by its corresponding BFC Manager by means of the error-detection mechanisms just explained in Section 7.3.

Current section firstly addresses how the BFC Manager evaluates its port contribution after an error is detected in the *resultant frame*. Then, it discusses how the BFC Manager performs such evaluation upon observing an error in its own port contribution.

7.4.1 Error detection after an error occurs in the *resultant frame*

This section is devoted to explaining what behavior the BFC Manager expects from its node, upon an error occurs in the *resultant frame* during normal transmission.

As already explained, any CAN node is able to detect five different error types [ISO93]: *stuff error*, *format error*, *bit error*, *CRC error* and *ACK error*. Such errors are detected by means of several error-detection mechanisms that check the correctness of the frame that the node transmits or receives. These mechanisms, also specified in [ISO93], respectively are: *stuff rule check*, *frame check*, *monitoring*, *CRC check* and *ACK check*. See Section 3.3.4 for further details.

Every CAN node detecting an error signals it by means of an active error flag. In the particular case of CANcentrate, a node will also signal any error it detects in the *resultant frame*. Hence, in order to correctly forecast that a node is going to signal an error that affects the *resultant frame*, it is necessary to identify when a CAN node detects it. Specifically, stuff, format and bit errors will be detected by all nodes; a CRC error should be detected by all receiving nodes; whereas an ACK error must be detected by the transmitting node.

The Rx_CAN Module can identify a stuff, a format and a CRC error happening in the *resultant frame*. Therefore, each BFC Manager can use the information provided by this module to forecast when its node is going to signal an error. However, the Rx_CAN Module does not detect ACK or bit errors. As explained in Section 5.5, this is because the hub is never the original transmitter of a message; so that the Rx_CAN observes the coupled signal from the point of view of a receiver. Fortunately, the BFC Manager can observe the resultant coupled signal to further know both when an ACK error has occurred and, in some cases, also to detect when its corresponding node has detected a bit error.

Next, we explain how these errors at the *resultant frame* can be identified, as well as what is the behavior that each BFC Manager expects from its node when the error is detected.

(1) *Stuff or format error*. The Rx_CAN Module detects these two errors by means of the *stuff rule check* and *frame check* mechanisms respectively. The CAN protocol specifies that whenever a node (transmitter or receiver) detects any of these errors, it must start to signal it in the next bit. Therefore, as soon as the Rx_CAN Module detects a stuff error or a format error in the *resultant frame*, each BFC Manager assumes that its corresponding node also has detected the error. Thus, the BFC Manager will check that its port starts signaling an active error flag in the bit following the error.

The signals involved in this case are used as follows. Once the Rx_CAN Module detects a stuff or a format error in the *resultant frame*, it will indicate (by means of the vector *frameField*) at the next bit that the state of the *resultant frame* is accounting the transmission of an active error flag (as said in Section 5.5, the Rx_CAN Module orders the Error Flag Generator to transmit an active error flag when detecting an error). Notice that if all nodes are synchronized with each other at frame level, they must detect an error in the *resultant frame* at the same time and they have to start signaling it at the same bit. Therefore, when the BFC Manager observes by means of the vector *frameField* that the transmission of an active error flag has been started in the *resultant frame*, it checks by means of B_i that its corresponding node has also started sending an active error flag.

(2) *CRC error*. The Rx_CAN Module detects this error by means of the *CRC check* mechanism. As already indicated, receiving nodes are the only ones that can detect this error. Moreover, in Section 3.3.4, it was also explained that any receiving node detecting a CRC error must start signaling it in the first bit of the End Of Frame field (EOF). Therefore, when the *resultant frame* does not pass the CRC check performed by the Rx_CAN Module, each BFC Manager corresponding to a receiver expect its node to start sending an active error flag in such bit of the EOF.

For checking the CRC error signaling, each BFC Manager corresponding to a receiver observes the signal *CRCPassed* to know if the *resultant frame* has passed the CRC check; and the vector *frameField* for knowing when the EOF begins. When the BFC Manager corresponding to a receiver detects that the frame has not passed the CRC check, it will observe the vector *frameField* and the contribution of its port, B_i, in order to check that its corresponding node starts sending an active error flag in the first bit of the EOF field.

(3) *ACK error*. This type of error is detected by the BFC Manager corresponding to the transmitter, which is the only node that actually can detect it. The CAN protocol specifies that a transmitting CAN node detects an ACK error when monitors a recessive bit during the ACK slot. The protocol also specifies that whenever a transmitter detects an ACK error, it must start to signal it in the bit following the ACK slot. Therefore, when a recessive bit is observed at the *resultant frame* during the ACK slot, the BFC Manager corresponding to the transmitter expects its node to detect an ACK error and, therefore, to start to send an active error flag in the next bit.

In order to detect the ACK error, the BFC Manager corresponding to the transmitter first observes the vector *frameField* for detecting when the ACK slot is being broadcast. In addition, the BFC Manager also has to observe the coupled signal, B_0, for detecting whether the ACK slot has a dominant bit or a recessive bit. If the BFC Manager corresponding to the transmitter detects a recessive bit in B_0 during the ACK slot, it assumes that an ACK error occurred.

When the BFC Manager corresponding to the transmitter detects an ACK error, it will observe its port contribution, B_i, to check that the transmitter starts sending an active error flag in the next bit.

(4) *Bit error*. This type of error can be detected by the BFC Manager to some extent. As explained above in Section 3.3.4, a CAN node detects a bit error whenever it sends a dominant bit, but observes a recessive bit. Furthermore, a CAN transmitting node also detects a bit error when it transmits a recessive bit and observes a dominant bit during a data or a remote frame (except during the arbitration phase

and at the ACK slot field, see Section 3.3.4). This can happen, for example, if a receiving node sends a dominant bit in a frame field during which it is not allowed to do that. A node that detects a bit error must signal it in the bit after the error is detected.

To identify that a node is going to detect a bit error because it has sent a dominant, but it has observed a recessive bit could be done, in principle, as follows. If the BFC Manager observes a dominant bit at its port contribution B_i and, nevertheless, the bit value of the coupled signal B_0 is recessive, it could assume that its node detects a bit error. Nevertheless, notice that it has not sense to decide that a node must detect a bit error when it issues a dominant bit and a recessive is observed at the coupled signal. This is because such a situation can only occur if a fault affects the internal circuitry of the hub so that the coupled signal is corrupted or even not correctly calculated. However, as a fault may also lead the BFC Manager to incorrectly monitor the coupled signal, to assume as valid the opinion of the BFC Manager could be incorrect. To diagnose which part of the hub behaves incorrectly as a consequence of an internal fault would require mechanisms that are beyond the scope of this work, e.g. internal hub redundancy.

In contrast, it is reasonable to detect a bit error in the case in which the recessive bit sent by a transmitting node is overwritten by a dominant bit during a frame field other than the ones related to the arbitration or the ACK slot field. This is because one dominant bit issued from the port corresponding to a receiver is enough to force the value of the coupled signal to be dominant.

As concerns the signals used for detecting a bit error in this case, the BFC Manager corresponding to the transmitter observes the contribution of its port, B_i, the vector *frameField* included in CS and the coupled signal, B_0. The BFC Manager uses the signal *frameField* and the CAN format rules to know when the transmitter and the receivers are allowed to send dominant and recessive bits. When the transmitter is allowed to send dominant and recessive bits, but the receiver is only allowed to send recessives, the BFC Manager corresponding to the transmitter compares the port contribution of the transmitter with the coupled signal. If a recessive bit is issued through the port of the hub corresponding to the transmitting node and the coupled signal has a dominant value, then the BFC Manager corresponding to the transmitter assumes that a receiver is sending an incorrect dominant bit which should trigger a bit error detection at the transmitter (it is enough a dominant bit issued from one port to force a dominant bit at the coupled signal).

7.4.2 Error detection after an error occurs on a port contribution

As specified at the beginning of Section 7.4 an error detected during a normal transmission, i.e. during a data frame, a remote frame or an inter frame in which no error has been detected so far, can be localized at two different levels: at the *resultant frame* and at any port contribution. In previous section we explained what behavior the BFC Manager expects from its node upon an error affects the *resultant frame*. Thus, current section explains what should be that behavior if the error is detected at its port contribution.

The specific contribution the BFC Manager expects just after detecting such an error depends on the role played by the node corresponding to the port which causes the error and the value of the bit issued through that port.

Let us discuss what is the expected behavior depending on the role of the node. If the node is acting as a transmitter, the possible errors that can be detected at its ports are the stuff error, the format error and the CRC error (see Section 7.3.1). If the transmitter violates the stuff rule or the format by issuing an unexpected dominant bit, that error will propagate to the *resultant frame*. This implies that the transmitting node at least should detect the error that affects the *resultant frame* and signal it at the next bit. However, if the bit that violates the stuff rule or the format is recessive, it may happen that the error does not propagate to the *resultant frame*. Specifically, this may occur if the recessive bit is overwritten by an unexpected dominant bit issued through the hub port corresponding to a receiver. Therefore, in the case of detecting a stuff or a format error on the transmitter contribution, its BFC Manager expects that the node correctly signals the error by means of an active error flag at the next bit, only if the error has actually propagated to the *resultant frame*. Otherwise, the BFC Manager simply expects that the transmitter continues communicating as if no error has occurred so far.

Regarding the other type of error that can be detected at the port of a transmitter, i.e. the CRC error, notice that a transmitting node does not signal a CRC error, even though it is the responsible for that error (it signals an ACK error, instead). Thus, although the BFC Manager considers that its node has been bit-flipping when the Rx_CAN detects a CRC error, it does not expect that the transmitter starts signaling an error. Instead, the BFC Manager just expects that the transmitter continues transmitting normally. Afterwards, if an ACK error finally occurs at the *resultant frame* as a consequence of the CRC error, the BFC Manager will expect that the transmitter signals the error, as explained before in Section 7.4.1.

For the case in which the node that provokes the error at the port is acting as a receiver notice that, as indicated in Section 7.3.2, the type of errors that can be detected at its contribution are: a dominant bit issued out of the ACK slot (except

when this bit is the beginning of an overload signaling or a SOF); a dominant bit at the ACK slot when the frame has not passed the CRC check; and a recessive bit at the ACK slot when the frame has passed this check.

A dominant bit sent by the receiver out of the ACK slot can either provoke an error at the *resultant frame* or not. For example, such a dominant bit provokes that the *resultant frame* violates the stuff rule if it overwrites a recessive stuff bit sent by the transmitter. In contrast, such a dominant bit does not provoke an error in the *resultant frame* if, for instance, it coincides with a dominant bit sent by the transmitter.

If a dominant bit sent by a receiver out of the ACK slot does actually lead to an error in the *resultant frame*, the receiving node should detect it; thus, its BFC Manager simply expects that it correctly signals the error by means of an active error flag at the next bit. In contrast, what to expect upon such a dominant bit does not provoke an error in the *resultant frame* is more complicated. Notice that this incorrect dominant bit can probably be the first bit of an active error flag the node sends in order to signal a local error (an error that only it has detected so far). But, it could also be just a bit-flipping bit generated by its uplink. Since, it is impossible to know, a priori, if the node is signaling a local error, the BFC Manager cannot be sure that its node will continue signaling an error in the next bit. Therefore, the BFC Manager proceeds as follows. In principle, it expects that its node signals an error in the next bit. But if the node actually does not, the BFC Manager will assume that the incorrect dominant bit actually was a bit-flipping bit generated by the uplink. In this case, the BFC Manager increases the bit-flipping counter (BFC) and simply expects that its node continues behaving correctly as if the error had not occurred. Notice that, in this way, the BFC Manager will not unfairly expect the node to signal an error when the dominant bit actually was a bit-flipping bit generated by the uplink.

The other possible error at the port of a receiver is a dominant bit at the ACK slot when the frame has not passed the CRC check. This can happen if the receiver incorrectly considers that the frame has passed the CRC check. However, as in the previous case, that dominant bit could be also the first bit of an active error flag the node sends to signal a local error, or even an incorrect bit generated by the uplink. Therefore, the BFC Manager acts as in the previous case.

The last error that can be detected at the contribution of a receiver consists in a recessive bit at the ACK slot of a frame that has passed the CRC check. This error occurs if the node incorrectly detected a CRC error; in which case it will signal the error at the first bit of the EOF. However, there can also be other causes for this error. For example, the port could be stuck-at-recessive, or even the uplink could

corrupt a correct dominant ACK bit sent by the receiver.

In order to deal with all these cases, what the BFC Manager will expect after detecting an incorrect recessive ACK bit is that the node behaves correctly as if the error had not occurred. Notice that this is the best choice. First, if the node actually incorrectly calculated the CRC, it will signal it at the first bit of the EOF. Since the BFC Manager will not expect this error signaling, it will consider the first bit of this signaling as a bit-flipping bit. This is desirable because it allows the BFC Manager to differentiate between a stuck-at-recessive and a bit-flipping fault (in Section 7.3.2, we explained that the BFC Manager does not consider, in principle, an ACK omission as a bit-flipping error). Second, if the recessive bit was actually a dominant ACK bit that became corrupted, the most probable is that the receiving node will not detect and signal it, because other receiving nodes will overwrite the recessive bit with their own dominant ACK bits. Thus, not to expect an error signaling is the better choice also for this case. Moreover, even if the node does detect that its dominant ACK bit was corrupted, the strategy of the BFC Manager is appropriate: the BFC Manager will consider the first bit of the active error flag the node will send as bit-flipping, thereby detecting that the node had a problem (that its dominant ACK bit was corrupted).

Finally, note that in this section we have just explained when and where the BFC Manager expects its node to signal an error, after it has detected an error in the contribution of that node. In the case the BFC Manager assumes that the node has to signal the error, it will further check that this node correctly performs the signaling. However, the specific format of the expected error frame will be thoroughly discussed next in Section 7.5.

7.5 Error detection during an error signaling

In Section 7.4 we described the cases in which the BFC Manager expects that its port signals an error. Current section explains how the BFC Manager checks that its port actually signals the error in a correct way.

As explained in 5.5, we only accept as valid nodes that are error-active. Thus, the BFC Manager expects that its node sends an active error frame. Notice again that an error-active node signals an error by transmitting an active error flag composed of 6 consecutive dominant bits, followed by a cooperatively error delimiter that is formed from consecutive recessive bits (see Section 3.3.4 for more details). For the sake of clarity, let us differentiate between the behavior the BFC Manager expects during the transmission of the error flag and the error delimiter.

The BFC Manager always checks that the error flag is composed of 6 consecutive dominant bits. This implies that the BFC Manager must check both that the error flag is not too short and that the error flag is not too large.

On the one hand, to check that the active error flag includes at least 6 consecutive dominant bits, the BFC Manager basically counts up the number of consecutive dominant bits it observes at its port, starting at the bit where the error signaling is supposed to begin. However, the BFC Manager cannot do that such simple. Imagine for example a situation in which the transmitting node detects a local error during the data field of a frame, and starts signaling it by sending an active error flag. Consider that this error flag provokes a stuff error (at both the *resultant frame* and the transmitter port) at its 6th bit. Then, the BFC Manager detects this stuff error and assumes that the transmitting node should start signaling it in the next bit (refer to Section 7.4.2 for more details). If the BFC Manager starts counting up the number of consecutive dominant bits the transmitter sends thereafter, it will only monitor recessive bits (which belong to the error delimiter the transmitter sends) and will unfairly assume that the transmitter is not correctly signaling the error.

In conclusion, the BFC Manager can unfairly believe that the error flag is too short when, at a given bit, it assumes that the node is starting to signal an active error flag when, in fact, the node started to signal it some bits before. In order to deal with this kind of situations, the BFC Manager considers as forming part of the active error flag all those consecutive dominant bits received through the port just before the bit that is supposed to be the beginning of the error flag.

The only disadvantage of this strategy is that the BFC Manager will not be able to detect some situations in which the error-flag is actually too short. For example, imagine the case of a transmitter that sends 3 correct consecutive dominant bits during the data field and that, due to a local error, does not monitor the third dominant bit (so that it detects a bit error). Then, the transmitter starts sending an active error flag. However, imagine that the last 3 bits of the flag are corrupted by the uplink so that they do reach the hub port as recessives. In this case, although the error flag was truncated, the BFC Manager is not able to detect it. This is because the BFC Manager will believe that the error flag was composed of the three dominant bits the node sent before signaling the error plus the three first dominant bits of the flag that were not corrupted. Nevertheless, despite the possible existence of this type of scenarios, we consider that it is more important not to unfairly penalize a correct port than to try to detect all possible bit-flipping errors.

On the other hand, in order to check the correctness of the active error flag, the BFC Manager also evaluates whether or not the flag is too large. This is important because a too large active error flag indicates that, due to local faults, the node has

detected bit errors during the transmission of its own active error flag (any bit error will compel the node to re-send its error flag).

Specifically, the BFC Manager accepts up to 6 consecutive dominant bits after the bit that is supposed to be the beginning of the error flag. When the BFC Manager detects the 7th consecutive dominant bit it assumes that the node has detected a bit error (an additional bit-flipping error) during the error signaling and that is transmitting a new active error flag. The BFC Manager checks again that this new active error flag is composed of a maximum of 6 consecutive dominant bits and so on.

The reason why the BFC Manager accepts an additional group of 6 consecutive dominant bits when detecting that the flag is too large is because, in the worst case, the node may detect an error in the last bit of an error flag, thereby initiating the transmission of a new error flag just after the previous one ends.

Also notice that for checking the maximum length of an error flag, the BFC Manager does not take into account the consecutive dominant bits a node may send before the first error flag is supposed to begin. For example, if during the data field, the BFC Manager detects a unexpected dominant bit at the node contribution and the three bits previous to that error were also dominant, the BFC Manager does not consider the three previous bits nor the erroneous bit as forming part of the active error flag. This is because the BFC Manager cannot be sure about whether or not these four consecutive dominant bits form part of the error flag.

Besides the error flag, the BFC Manager checks the correctness of the other part of the error frame: the error delimiter. First of all it is important to clarify when the BFC Manager considers that its node is sending the error delimiter. In principle, the BFC Manager assumes that its node is sending the error delimiter as soon as it detects in its contribution a recessive bit after the active error flag (independently of whether the flag is too short or too large).

However, there is one exception: the case of a receiving node that sends a too short error flag that does not provoke an error in the *resultant frame*. This may happen if the dominant bits that constitute that error flag coincide with a set of correct dominant bits sent by the transmitter. We made this decision because a too short error flag sent by a receiver can be just a set of consecutive dominant bits generated by a short disturbance of its uplink. Therefore, when a too short error flag sent by a receiver does not corrupt the *resultant frame*, its BFC Manager considers that this node will behave as if no error has occurred so far. Notice that this cannot happen if the node is the transmitter, since any unexpected dominant bit sent by the transmitter that leads its BFC Manager to assume that is signaling an error will also corrupt the *resultant frame*.

In conclusion, the BFC Manager assumes that its node is sending the error delimiter as soon as it detects a recessive bit in its contribution and an error has already occurred in the *resultant frame*.

When the BFC Manager assumes that its node is sending the error delimiter, it checks that the node only sends recessive bits. In fact, it will expect recessive bits as long as the *resultant frame* does not reach the idle field.

If the BFC Manager monitors a dominant bit at its port contribution when it is considering that its node is transmitting the error delimiter, the BFC Manager assumes that a bit-flipping error has occurred. The behavior the BFC Manager expects after detecting that dominant bit depends on the state of the *resultant frame*. If this state is the error delimiter, it means that all nodes are signaling the error delimiter and that, consequently, the dominant bit is going to corrupt that delimiter. Therefore, since the node should at least detect that error at the *resultant frame*, its BFC Manager expects that it starts signaling it at the next bit.

In contrast, if the node sends a dominant bit during its error delimiter and the state of the *resultant frame* is the error flag, then the BFC Manager cannot assume that its node is going to signal an error in the next bit. This is because that dominant bit could be just an erroneous bit generated by the uplink and, in such a case, since it coincides with a dominant bit of the error flag that is being broadcast, the node will not detect it. Therefore, if the node actually does not signal an error in the next bit, the BFC Manager simply expects that its node continues behaving correctly (transmitting its error delimiter) as if the error had not occurred. Notice that this strategy is equivalent to the one explained in Section 7.4.2, where we explained what the BFC Manager of a receiver expects when its node sends an unexpected dominant bit that does not provoke an error in the *resultant frame*.

The last check the BFC Manager performs on its node contribution when that node is transmitting the error delimiter is the following one. If the BFC Manager observes that a dominant bit not issued through its port (but from another one) corrupts the error delimiter being broadcast at the *resultant frame*, it expects its node to detected it and, therefore, to start signaling it in the next bit.

In all cases in which the BFC Manager assumes that its node stops transmitting its error delimiter to signal an active error flag, the BFC Manager evaluates the correctness of the expected error flag following the rules already explained in this section.

7.6 Error detection during an overload signaling

The fifth type of expected contribution identified in Section 7.2 corresponds to the behavior of a node during an overload signalling.

As indicated in Section 3.3.6, there are two kinds of overload conditions [ISO93]. The first overload condition consists in a node that needs extra delay before a new data or remote frame is transmitted on the bus. When this happens, the node start sending an overload frame at the first bit of intermission. The format of an overload frame is the same as the format of an active error frame [ISO93]: it includes an overload flag of 6 consecutive dominant bits followed by a cooperative overload delimiter of at least 8 consecutive recessive bits. The second overload condition ensures that the first condition, which is locally detected by a node, is globalized to the rest of nodes. Specifically, any node detecting a dominant bit during intermission must react by sending an overload frame.

The behavior the BFC Manager expects from its port after detecting an overload condition (after detecting that its node sends a dominant bit during intermission) depends on whether that condition is triggered by its node or not, i.e. on whether or not its node has initiated the overload to achieve an extra delay before a new data or remote frame is transmitted.

On the one hand, if the BFC Manager of a receiver detects that its corresponding port issues a dominant bit in the first bit of the intermission field, it will assume that its node is triggering an overload condition. The BFC Manager detects this when the vector *frameField* indicates that the current bit being broadcast is the first bit of the intermission field and, in addition, it observes that the value of its corresponding port contribution, B_i, is dominant. When this happens, the BFC Manager assumes that this dominant bit is the first bit of the overload flag and, hence, it expects its node to send 5 more consecutive dominant bits. Afterwards, the BFC Manager checks that its node cooperatively transmits the overload delimiter.

On the other hand, the BFC Manager considers that an overload condition is triggered by another node if it observes a dominant bit at the coupled signal, B_0, and a recessive bit at its port contribution, B_i, during the first bit of the intermission field. In this case, the BFC Manager expects that its port issues the 6 consecutive bits that constitute the overload flag and, thereafter, it checks that its node cooperatively transmits the overload delimiter.

Since the format of an overload frame is the same as the format of an active error frame, the specific way in which the BFC Manager checks that its node correctly sends the overload frame (flag and delimiter) is the same as the way in which it evaluates that its node correctly sends an active error frame. How the BFC Manager

evaluates its node during an error signaling was explained in Section 7.5. Basically, what the BFC Manager checks is that the overload flag is not too short or too large, as well as that the node does not send dominant bits during its error delimiter. If the BFC Manager detects that its port behaves incorrectly or that the overload delimiter is corrupted at the *resultant frame*, it proceeds as explained in Section 7.4.

7.7 Conclusions

This chapter thoroughly describes the way in which the BFC Manager detects bit-flipping errors in its hub port contribution. Since a node together with its corresponding link are considered as an error-containment region, the BFC Manager assumes that the bit issued through the port is the bit the node wishes to send. Hence, we talk about the behavior of a hub port and the behavior of a node interchangeably.

Each BFC Manager evaluates the correctness of its corresponding hub port by checking that its contribution does not deviate from the correct behavior expected according to the current state of the *resultant frame*. Although the behavior of a CAN node is quite complex in the general case, we have been able to identify few independent types of behaviors.

We basically differentiate between the behavior of a node when no error has been detected so far and when an error has occurred. On the one hand, when no error has occurred so far, i.e. during normal transmission, the behavior of a node depends on the role that it is playing, i.e. transmitter or receiver. To evaluate the correctness of the contribution of the transmitter is quite complex since it is rather free to send the bit values it wishes. Fortunately, the CAN protocol already specifies the most complete set of error-detection mechanisms for detecting errors that corrupt the data conveyed through the bus (and thus that corrupt the *resultant frame*). Since these mechanisms are designed to detect errors in a signal where the contributions of all nodes are mixed, the BFC Manager has to adapt them in order to directly detect errors in the contribution of the transmitting node. Contrary to the case of a transmitting node, to detect errors on the contribution of a receiving node is easier. This is because a receiver is only allowed to send recessive bits, except in particular locations of the frame and under specific circumstances. Thus, the BFC Manager of a receiving node basically checks that its port only issues a dominant bit when allowed.

On the other hand, when an error occurs, the expected behavior of a node does not only depend on the role it is playing, but also on the type of the error as well as on where the error is detected: on the *resultant frame* or on its port contribu-

tion. When an error affects the *resultant frame*, the BFC Manager needs to identify whether or not its node detects it in order to forecast when that node is going to signal the error. Specifically, all nodes should detect and signal any stuff, format and bit error; only receivers should detect and signal a CRC error; whereas exclusively the transmitter should detect and signal an ACK error. Since the Rx_CAN Module can identify a stuff, a format and a CRC error happening in the *resultant frame*, the BFC Manager can use the information provided by this module to forecast when its node is going to signal any of these errors. Additionally, the BFC Manager can observe the resultant coupled signal to further know both when an ACK error has occurred and, in some cases, also to detect when its corresponding node has detected a bit error.

The other location where an error issued by a node can be detected is at its port contribution. An error affecting a given port is discovered by its corresponding BFC Manager by means of the error-detection mechanisms that evaluate whether or not the contribution of a transmitter or a receiver is correct when no error has been detected so far. If the erroneous bit issued by a node provokes an error in the *resultant frame*, that node should detect this situation and signal it when appropriate. Thus, the corresponding BFC Manager can easily forecast what should be the correct contribution of its node. However, if the erroneous bit does not corrupt the *resultant frame*, the BFC Manager cannot know a priori which will be the contribution of the node thereafter. This is because it is impossible to elucidate a priori which is the cause for an erroneous bit, e.g. it could be just generated by the uplink, or it could form part of an active error flag the node is transmitting to signal a local error. To overcome this problem, the BFC Manager adapts what it expects from its node to the bits that monitors after the erroneous bit.

Finally, the BFC Manager also evaluates whether or not its node correctly signals an error and an overload. The BFC only accepts as valid nodes that are error-active, hence it expects that its node sends an active error frame (an overload frame has the same format as an active error frame). Basically, what the BFC Manager checks is that the error/overload flag is not too short or too large, as well as that the node does not send dominant bits during its error/overload delimiter (unless it should signal a new error that affects the *resultant frame*). However, due to the huge number of different manners in which a bit-flipping can manifest, it is not possible to know what should be the correct contribution of a node every time it issues an unexpected bit during the error/overload signaling. We explained the strategy the BFC Manager follows to try to encompass this wide range of error scenarios. Since to pretend to detect all possible bit-flipping bits would be impossible, at least this strategy tries not to take decisions that would unfairly penalize the nodes

Chapter 8

Analysis of the mechanisms that deal with bit-flipping faults

8.1 Introduction

The error-detection and fault-treatment mechanisms the hub includes for dealing with bit-flipping faults were first presented in Section 6.6. We explained that a bit-flipping fault can manifest itself in a huge number of different manners. This implies that the hub must be able to detect bit-flipping bits in the presence of an enormous amount of scenarios involving errors. Additionally, the wide variety of patterns in which a bit-flipping fault can manifest poses some problems for correctly diagnosing a port as being permanently bit-flipping.

Note that, as explained before, all these mechanisms are included in each Enabling/Disabling Unit, which operate independently at each hub port. Specifically, within the Enabling/Disabling Unit, the *Bit-Flipping Counter Manager* (BFC Manager) is the responsible for detecting bit-flipping errors. This module increases and decreases the BFC (*Bit-Flipping Counter*) following specific rules that try to efficiently diagnose when the hub port is actually bit-flipping. Whenever the BFC exceeds a given *Bit-Flipping Threshold* (BFT), the Threshold Control Module diagnoses the port as being faulty and isolates it by means of the corresponding ED signal.

Due to the complexity of detecting bit-flipping errors, Chapter 7 was entirely dedicated to describe the way in which the hub detects these errors at its port contributions. Current chapter further discusses how the complexity of the bit-flipping error-detection and fault-treatment mechanisms influences dependability. Then, it

explains what are the advantages and limitations of these mechanisms and proposes some further enhancements. Finally, it addresses how to configure some parameters of the fault-diagnosis mechanisms in order to properly diagnose bit-flipping faults.

8.2 Complexity of the mechanisms

The complexity of the circuitry included in a device is an important aspect that has to be taken into account when devising dependable systems. The probability of failure of a device augments as its complexity increases. As said before, the hub is the most critical element concerning dependability in CANcentrate, since it is the single point of failure of the communication system. Thus, we took special care when devising the error-detection and fault-treatment mechanisms of the hub in order to increase as less as possible its complexity in terms of circuitry.

As described before, the mechanisms for detecting and treating bit-flipping faults are mainly included in each BFC Manager, which independently operates over a given port. Each BFC Manager basically observes its port contribution, B_i, the coupled signal B_0, the set of signals CS from Rx_CAN and uses its acknowledge about the CAN protocol to check if its contribution is correct according to the current state of the *resultant frame*.

The first characteristic of the error-detection and fault-treatment mechanisms that allows minimizing the circuitry complexity is the way in which the BFC Manager gets the current state of the *resultant frame*. As explained in Chapters 5 and 6, the BFC Manager does not calculate this current state by observing the resultant coupled signal, B_0. Instead, the Rx_CAN Module is aimed at calculating bit by bit this current state by observing the coupled signal. Then, the Rx_CAN Module provides the different BFC Managers with a set of signals, CS, that, together with B_0, describes this state. Hence, since the logic for calculating the current state of the *resultant frame* is implemented once in the hub (in the Rx_CAN Module), the cost in terms of circuitry is less than it would be if each BFC Manager had to implement it.

This way of minimizing the circuitry is even more important if one takes into account that, in order to detect bit-flipping errors, it is necessary to identify several types of errors in the *resultant frame*: stuff error, format error, CRC error and ACK error (see Sections 7.4 and 7.5). Since the BFC Managers do not need to include the logic for identifying these errors, an important amount of circuitry is saved. Among all these error-detection mechanisms, the most important reduction is achieved when not including in each BFC Manager the logic needed for checking

the CRC sequence, which is expensive in terms of circuitry.

The second characteristic of the fault-diagnosis mechanisms that makes possible reducing the amount of circuitry needed for implementing the error-detection and fault-treatment mechanisms is that, as explained in Chapter 7, each BFC Manager does not monitor the activity of the other BFC Managers. The major benefits of this are that it actually reduces the number of needed interconnections inside the hub and the complexity of the state machines included in each BFC Manager. Moreover, this also makes the hub design more flexible and extensible for further improvements. For instance, to add new ports and their respective Enabling/Disabling units will not require to change the circuitry of the existing modules and units within the Fault-Treatment Module.

8.3 Advantages of the mechanisms

This section explains what are the advantages of the mechanisms the hub of CANcentrate includes for dealing with bit-flipping faults. On the one hand, the hub can detect bit-flipping errors beyond the possibilities of the error-detection mechanisms implemented by a typical CAN node. Moreover, it also increases the accuracy with which these errors are detected in CAN. On the other hand, the hub overcomes some limitations of the CAN fault-treatment mechanisms.

In order to better understand how the hub improves error detection and fault treatment, let us briefly summarize how a CAN node detects errors and diagnoses faults (see Sections 3.3.4 and 3.3.5 for further details). Firstly, the CAN node includes a set of mechanisms that allow it to detect five different error types [ISO93]: stuff error, format error, bit error, CRC error and ACK error. Such errors are detected by means of several error-detection mechanisms that check the correctness of the frame that the node transmits or receives. These mechanisms, also specified in [ISO93], respectively are: *stuff rule check*, *frame check*, *monitoring*, *CRC check* and *ACK check*.

Concerning the way in which a fault (a faulty node) is diagnosed and passivated in CAN [ISO93], each CAN node is provided with two error counters: the *Transmission Error Counter* (TEC) and the *Reception Error Counter* (REC), which are increased and decreased following specific rules. As explained in Section 3.3.5, when either the TEC or the REC reach a given threshold, the node enters the *error passive state* which actually reduces the impact of the node on the communication [ISO93]. A second threshold is used if the error passive state is not enough. If the TEC/REC reaches this second threshold, the node enters the *bus-off state*, so that it is not involved in bus activities. This last situation corresponds to a node

diagnosing itself as being faulty.

A CAN node increases its TEC/REC almost every time it detects an error. As explained in Section 3.3.5, the rules used to increase the TEC/REC are more strict with the errors the node detects when it is acting as the transmitter than when it is acting as a receiver. The transmitter normally increases the TEC by 8 units, whereas a receiver usually increases the REC by 1 unit.

Additionally, the TEC or the REC (depending on the role of the node) are further increased in 8 units when the node seems to be the responsible for the error condition. Specifically, a node additionally increases its TEC/REC by 8 when it detects a *primary error* [ISO93], i.e. when it monitors a dominant bit after its own error flag. Refer to Section 3.3.5 for more details about the primary error.

8.3.1 Enhanced error detection

In CAN the error detection is only performed on the *resultant frame*. In contrast, the hub can also detect errors on the contribution of each port. This allows the hub to detect bit-flipping errors beyond the capacity of the error-detection mechanisms of CAN, as well as to increase the accuracy with which bit-flipping errors are detected.

More specifically, the first error-detection enhancement is that the hub can identify when a node should start transmitting an error and, then, to check if this actually happens. This is an important improvement since CAN does not include any mechanism to check if a node starts signaling an error when expected. For example, imagine that an ACK error is observed at the *resultant frame*. When this occurs, the transmitter must signal the error at the next bit: the ACK delimiter. In CANcentrate, the BFC Manager corresponding to that node will check that it actually starts signaling the error. Conversely, in CAN no mechanism evaluates if this happens; hence it would not be considered as erroneous that the transmitter does not signal the ACK error. The details of how each BFC Manager evaluates whether or not its node signals an error as expected can be found in Sections 7.4 and 7.5.

The second enhancement the hub presents in terms of error detection is that the hub is able to identify incorrect bits issued from a node that would be masked (and hence not detected) in CAN by the contribution of other nodes. Firstly, in CAN, an incorrect dominant bit issued by a receiver can be masked by a correct dominant bit issued by a transmitter. For example, an incorrect dominant bit sent by a receiver during the data field is not detected in CAN if that bit coincides with a dominant bit sent by the transmitter. Secondly, a dominant bit sent by a transmitter can also be masked. Specifically, this occurs when the transmitter sends a dominant bit

during the ACK slot, since to observe a dominant bit at this slot of the resultant frame is not an error. Thirdly, an incorrect recessive bit can be also masked in CAN. For instance, a recessive bit sent by the transmitter during the data field will provoke an error if it violates the stuff rule. However, if this recessive bit coincides with an incorrect dominant bit issued by a receiver, then no error will be encountered. In contrast, in CANcentrate no incorrect bit can be masked, since each node contribution is monitored separately before they are coupled.

The last enhancement the hub presents concerning error detection is that it increases the accuracy with which bit-flipping errors are identified. Specifically, the hub overcomes the lack of accuracy of the ACK check mechanism of CAN. As indicated in Section 7.3.1, the ACK check is the mechanism that the CAN transmitting node uses to indirectly detect that its frame has not passed the CRC check. In other words, the transmitter does not perform the CRC check of the frame it is transmitting, but it decides that it is the responsible for a CRC error when observes an ACK error, i.e. that no receiver acknowledges the frame.

Unfortunately, as explained in Section 7.3.1, to use the ACK check to detect that the transmitter contribution is incorrect may be unfair. Firstly, an ACK error can occur even when the contribution of the transmitter is correct, if the receiving nodes (or their links) have any problem so that they do not acknowledge a correct frame. If this occurs, the transmitter will signal an error in the ACK delimiter and, thus, it will unfairly detect a primary error. Moreover, any receiver that did not acknowledge the frame because it incorrectly calculated the CRC will not detect a primary error. This is because any receiver that incorrectly decides that the CRC is erroneous plans to start signaling the CRC error at the first bit of the EOF; however, it will observe that the transmitter initiates the error signaling before it (at the ACK delimiter).

Secondly, to use the ACK check to detect that the transmitter has behaved incorrectly could also provoke that correct receivers unfairly detect a primary error. Imagine a situation in which a receiving node does not correctly perform the CRC check and comes to the conclusion that the CRC is correct when, in fact, it is not. Then, that receiver will acknowledge the frame and the transmitter will not detect and signal an ACK error. Therefore, the nodes that will eventually corrupt the frame will be the correct receivers, which detected the CRC error. Specifically, these receivers will signal the CRC error at the first bit of the EOF. This implies that all these correct receivers will detect a primary error, whereas the incorrect receiver (the one that acknowledged the frame) will not.

Conversely to what happens in CAN, the hub of CANcentrate assumes that the transmitter issues an error whenever the frame does not pass the CRC check, but not

when no receiver acknowledges the frame. On the one hand, this allows detecting an error in the transmitter contribution whenever it is actually the responsible for the CRC error. On the other hand, it allows detecting an error on the contribution of both the receivers that do not acknowledge a correct frame and the receivers that acknowledge an incorrect frame. Moreover, the hub does not unfairly detect errors on the receivers in the circumstances explained above. Refer to Sections 7.3 and 7.4 for more details concerning the way in which the hub uses the CRC and the ACK check for detecting bit-flipping errors.

8.3.2 Enhanced fault treatment

The second advantage of the mechanisms of CANcentrate for dealing with bit-flipping faults is that its hub overcomes some deficiencies of the mechanisms CAN includes for treating faults. As explained in Section 6.6, the hub cannot rely on the fault-treatment mechanisms that each CAN node includes for treating bit-flipping faults: first, faulty nodes may stop performing fault-confinement operations, second, a bit-flipping fault located in a medium bothers all nodes so that none of them can isolate the fault, and third, the bus imposes a mixed vision of all nodes' contributions, thus, reducing the accuracy of the fault-diagnosis mechanisms. In contrast, the hub we have devised does not present most of such deficiencies.

First, notice that the contribution of all nodes have to traverse the hub, which operates independently from them. Therefore, the fault-diagnosis capacities of the hub do not depend on the correct error-containment operations performed by the nodes. Hence, passivation of faulty ports is guaranteed even if nodes stop performing fault-treatment operations.[1]

Second, the impossibility of dealing with faulty media in CAN is overcome in CANcentrate, since each link is dedicated and, together with its corresponding node, constitutes a single fault-containment region that the hub can treat.[2]

Third, the hub is also able to improve the accuracy of the fault-diagnosis mechanisms implemented by typical CAN nodes. As already explained, these mechanisms are based on the *Transmission Error Counter* (TEC) and the *Reception Error Counter* (REC). The problem is that the rules for increasing/decreasing the TEC/REC present some limitations that reduce the accuracy with which nodes diagnose faults. First, a transmitting node is always penalized, even when it is not the cause of the error. This may lead to unfairly isolate a CAN node that is not faulty. For example, it may happen that each one of the subsequent frames a node

[1] Note that this advantage applies not only to bit-flipping faults, but also to stuck-at faults.

[2] Note that this advantage applies not only to bit-flipping faults, but also to stuck-at faults.

tries to transmit is corrupted by the failure of a different receiving node. Although each faulty receiver will eventually isolate itself, the subsequent increments of the TEC of the transmitter can lead it to shut-down too.

The second limitation of the rules for increasing/decreasing the TEC/REC is that they rely on the detection of a primary error to identify (and hence, to additionally penalize) the node that is the responsible for an error globalization. The problem is that the node needs to wait until the end of its error flag to detect a primary error and, hence, the node may incorrectly believe that it is the responsible for an error if extra errors occur during the transmission of the error flag.

One possible case in which this can occur arises when a node that started to signal an error situation, as a consequence of a local error, detects an additional local error during its own active error flag (it monitors a recessive instead of a dominant). In this situation the node immediately starts the transmission of an additional active error flag, leading the other nodes to incorrectly detect a primary error. Other example of an incorrect detection of the primary error occurs when a node that detects a local error does not monitor a dominant bit after its own active error flag due to an extra local error. In this case the node, which actually should have been detected a dominant bit belonging to the error flags of other nodes, will incorrectly not detect a primary error.

Due to the inaccurate strategy for detecting the CAN node that is the responsible for an error condition, the fault-diagnosis mechanisms specified in CAN are inaccurate too. In contrast, the hub improves the detection of guilty nodes since it does not use the primary error mechanism, but it has an independent vision of each node contribution. On the one hand, each BFC Manager detects that its port is guilty of an error condition at the instant of time it issues an incorrect contribution. On the other hand, each BFC Manager is able to detect that its port is the responsible for an error condition, regardless of the contribution of any other port. Therefore, a guilty node is detected independently of whether or not any other node (or itself) detects extra errors. See Chapter 7 for a thorough explanation of the error-detection mechanisms the BFC Manager implements.

8.4 Limitations of the mechanisms and further enhancements

In Section 8.3 we explained the advantages of the mechanisms the hub includes for dealing with bit-flipping faults when compared with CAN. However, the error-detection and fault-treatment mechanisms of the hub also present some limitations.

Current section aims at analyzing these limitation and to provide some further enhancements.

8.4.1 Unfair error detection during the error signaling

As explained in Chapter 7, the mechanisms for detecting errors provoked by bit-flipping faults are included in the BFC Managers. Each BFC Manager basically checks that the contribution of its port is correct according to the current state of the *resultant frame*.

The contribution of a node can be incorrect due to several causes: the bit value the node sends can be changed by a fault in its uplink; the bit value broadcast to the node can be changed by a fault in the downlink, thereby compelling the node to react by sending an unexpected bit value; the node can fail by misinterpreting the state of the *resultant frame* and then issue a bit whose value is incorrect; an electromagnetic disturbance or an internal node fault may provoke that the node losses the synchronization at bit level and, thus, at frame level, etc. Of all these causes, to lose the synchronization at bit level poses some additional difficulties in detecting bit-flipping errors.

As it was explained, the synchronization at bit level allows all nodes and the hub to agree about the beginning and the end of each bit time in order to perform a correct sampling of the bit value. Thus, a node that is not synchronized at bit level with the hub and the other nodes usually also loses the synchronization at frame level. This normally provokes that the node issues an unexpected bit value, e.g. a de-synchronized receiver could send its acknowledge at the CRC delimiter (one bit before the ACK slot).

During a data, a remote or an inter frame, the BFC Manager corresponding to a de-synchronized node will notice any unexpected bit value at its port and, thus, it will correctly detect that the node issues a bit-flipping error. Moreover, the BFC Manager could also expect that the de-synchronized node signals the error (see Section 7.4 for further details about when a BFC Manager expects its node to signal an error). However, at this point, the BFC Manager could encounter some difficulties in evaluating whether or not the contribution of its node is correct.

As explained in Section 7.5, one of the aspects that the BFC Manager checks when evaluates how its node signals an error is that it sends an active error flag that is not too short nor too large (an active error flag should be constituted of 6 consecutive dominant bits). The problem is that since the node and the hub are de-synchronized at bit-level with each other, the hub is not always able to sample at the correct instants of time the bits that constitute the error signaling of the node. In

other words, the BFC Manager can sample each bit at the wrong time and, hence, it cannot always correctly interpret the value of the bits the node sends. In particular, as a consequence of this limitation, the BFC Manager could unfairly detect that the error flag is too short or too large. For example, the BFC Manager could sample as recessive the last dominant bit of the active error flag of its node, thereby unfairly assuming that the flag is too short.

Furthermore, a fault such as an electromagnetic disturbance can provoke not only that one node loses the synchronization at bit level, but that many nodes and the hub lose this synchronization with each other. If this occurs, the hub could incorrectly evaluate the way in which each one of these nodes signals an error. This is specially hazardous, since the BFC Manager could detect bit-flipping errors on the contributions of nodes that do not suffer from any fault.

This problem is worsened even further if the bit-flipping fault that provokes the loss of synchronization manifests during several bits of the error signaling; specially, while all nodes and the hub are cooperatively transmitting the error delimiter. If this occurs, the bit-flipping bits generated by the fault will corrupt the delimiter thereby forcing all nodes and the hub to signal the error again. Since all nodes and the hub will be still not synchronized with each other, the hub will unfairly penalize them again. This problem could manifest many times if the fault endures, as long as it is not isolated.

One possible solution to mitigate the effects of a fault that provokes a loss of synchronization is to be less strict when evaluating if the active error flag sent by a node has the appropriate length. On the one hand, the BFC Manager could be modified to accept as valid an active error flag that is constituted of 5 consecutive dominant bits. In this way, if due to a de-synchronization, the hub is not able to correctly monitor the last dominant bit of an active error flag, at least the BFC Manager will not consider that flag as being too short. We believe that to accept as correct error flags of few than 5 consecutive dominant bits is not a good choice. This is because a de-synchronization can lead the sample points of different nodes to be out of phase, but it cannot provoke that nodes consider very different bit times. Therefore, the sample points of two de-synchronized nodes cannot differ in more than one bit time.

On the other hand, in order to be less strict when evaluating the length of an active error flag, the BFC Manager could accept as valid flags that consist of more than 6 consecutive dominant bits. In Section 7.5, we explained that the BFC Manager considers that an active error flag is too long when it is composed of more than 6 consecutive dominant bits. Specifically, the BFC Manager assumes that its node has experienced an extra error while sending its flag, every time the manager

monitors that this flag is composed of a new sequence of 7 consecutive dominant bits. Because of the sample points of two de-synchronized nodes cannot differ in more than one bit time, we could change the behavior of the BFC Manager so that it assumes that a new error affected its node while sending its error flag, whenever it monitors a new sequence of 8 consecutive dominant bits.

8.4.2 Unfair error detection after an *arbitration misunderstanding*

The second limitation of the mechanisms for dealing with bit-flipping faults is related to the error-detection and fault-diagnosis after an arbitration phase where the roles of the nodes are not correctly determined.

As explained in Section 7.2, each Enabling/Disabling Unit monitors its port contribution during the arbitration phase in order to decide if its port is acting as a receiver or as a transmitter. An Enabling/Disabling Unit assumes that its corresponding port is a receiver in two cases. First, if that port does not send any dominant bit during the arbitration phase. Second, if that port is attempting to transmit by sending dominant bits during the arbitration phase, but it issues a recessive bit and, at the same time, a dominant bit is observed at the coupled signal. This is because if another node is contending for gaining the access to the channel and sends a dominant bit, this bit will be reflected at the coupled signal.

Nevertheless, notice that since the bits that contending nodes send and receive can become corrupted, it is possible that an *arbitration misunderstanding* occurs after the arbitration phase finishes. An *arbitration misunderstanding* occurs when the hub and all nodes do not agree on the role of each node after the arbitration. For example, a contending node that actually loses the arbitration at the last bit of the arbitration phase can incorrectly believe that it has won, if it does not observe that the recessive bit it sends is overwritten by the dominant bit other node issues. Similarly, a BFC Manager corresponding to a node that loses the arbitration at the last bit can incorrectly conclude that its node has won, if the recessive bit the node sends is corrupted at the uplink so that the BFC Manager monitors a dominant bit.

As can be deducted from the above examples, an *arbitration misunderstanding* can happen even if only one error occurred during the arbitration phase. However, if the BFC Manager and its node do not agree on the role the node is playing, then the BFC Manager will consider the contribution of that node as being bit-flipping during the rest of the frame. This is because the BFC Manager will expect a contribution of a transmitter when its node is actually acting as a receiver, or viceversa. Therefore, even if the *arbitration misunderstanding* was provoked by only one error during the arbitration, a BFC Manager can unfairly detect many

bit-flipping errors in its contribution.

The amount of bit-flipping errors a BFC Manager detects after an *arbitration misunderstanding* happens depends on whether or not the hub (the Rx_CAN Module), or any node, detects an error in the *resultant frame* and decides to abort the frame by signaling it. If more than one node considers itself as being the transmitter as a consequence of the *arbitration misunderstanding*, then it is likely that their contributions will eventually collide, leading them to abort the frame. In such a case, the number of times the BFC Manager unfairly detects a bit-flipping error is low. Unfortunately, it is possible that no more than one node considers itself as the transmitter, even though there is a BFC Manager corresponding to a receiver that incorrectly believes that its node won the arbitration. In this situation, the frame will probably not be aborted and this BFC Manager will unfairly detect a high number of bit-flipping errors.

One possible solution for reducing the impact of the *arbitration misunderstanding* is that the hub aborts the frame (by sending an active error flag) not only when the Rx_CAN Module (see Sections 5.4 and 5.5) detects an error in the *resultant frame*, but also when any BFC Manager detects an error at its port contribution. In this way, if a BFC Manager does not agree with its node in the role that node is playing, then it will abort the frame as soon as it detects an erroneous contribution.

However, this solution negatively affects the performance of the network, since a BFC Manager could abort frames that would not be aborted in a CAN bus. For example, a BFC Manager could incorrectly believe that its node is the transmitter when, in fact, it is a receiver. In this case, the BFC Manager will abort a frame that in CAN will be successfully transmitted.

Therefore, we propose to adopt an alternative solution. It consists in restricting the number of times that the BFC Manager increases its BFC during a data, a remote frame or an inter frame. To do that, the BFC Manager additionally uses a *Bit-Flipping Detection Counter* (BFDC) and a *Bit-Flipping Detection Threshold* (BFDT).

At the beginning of each data or remote frame, the BFC Manager resets the BFDC, but increases it each time that it increases the BFC during such frame. Whenever the value of the BFDC exceeds the BFDT, the BFC Manager orders the Error Flag Generator Module (see Sections 5.4 and 5.5) to signal an error. This will abort the frame and will lead the nodes to send their active error flags. Each BFC Manager (including the one corresponding to the port whose BFDC exceeded its BFDT) continues using its BFC without any restriction during the error signaling, in order to check if its node correctly signals the error (see Sections 7.4.2 and 7.5 for an explanation of how a BFC Manager checks this).

This solution based on the use of the BFDC and BFDT mitigates the loss of performance that can be provoked by aborting frames that would not be aborted in a standard CAN system. First, notice that the only case in which a frame would be unnecessarily aborted when using this alternative solution may happen if a bit-flipping fault only generates dominant bits that coincide with correct dominant bits sent by the transmitter, or with an acknowledgement dominant bit sent by any receiver. In such situations no error will be detected in the *resultant frame*, but only in the hub port where the fault is located.

Fortunately, the probability of a bit-flipping fault not provoking an error in the *resultant frame* detectable by the hub or the nodes during many bits, but in contrast leading the BFDC to exceed the BFDT, must be taken as negligible. In average, a CAN node should send the same number of dominant bits and recessive bits during a data or a remote frame. Thus, during a data or a remote frame, the probability that a dominant bit generated by a bit-flipping fault coincides with a correct dominant bit send by other node (thus not provoking an error in the *resultant frame*) is of 0.5. As a consequence, the probability that N dominant bits generated by a bit-flipping source only coincide with correct dominant bits during a data or a remote frame can be calculated as 0.5^N. Since a bit-flipping fault should generate several erroneous dominant bits in order to lead the BFDC of the corresponding port to exceed the BFDT, the probability that a frame is unnecessarily aborted should be very low.

The second reason why the adoption of the solution based on the BFDC and the BFDT mitigates the loss of performance is that the hub will eventually isolate any bit-flipping port. Thus, a possible loss of performance provoked by the use of this solution must be taken as temporary.

Finally, notice that it can be considered that the solution based on the BFDC and the BFDT may, in some cases, improve the performance. Consider the case in which, during a data or a remote frame, a source of bit-flipping bits will generate an error in the *resultant frame* sooner or later. Note that if the bit-flipping bits are masked by the contribution of the transmitting node, in the worst case, the bit-flipping bits may only provoke an error next to the end of the frame. Therefore, to use the BFDC and the BFDT to abort the frame earlier will save bandwidth.

8.5 *Penalization policy* of the BFC Manager

As already explained, the BFC Manager diagnoses its port as being permanently faulty when the value of the *Bit-flipping Counter* (BFC) exceeds a given *Bit-flipping Threshold* (BFT). Additionally, in Section 8.4 it has been explained that each BFC Manager could restrict the number of times it increases its corresponding BFC in

each data, remote or inter frame in order to reduce the impact of unfair error detections in its port contribution. For this purpose, the BFC Manager could use a dedicated *Bit-Flipping Detection Counter* (BFDC) and a dedicated *Bit-Flipping Detection Threshold* (BFDT).

The present section is aimed at discussing what should be the specific values for increasing and decreasing the counters, as well as the specific values of the thresholds involved in the diagnosis of bit-flipping faults, i.e. it is devoted to discussing the *penalization policy*.

The BFT and the number of units that the BFC has to be increased or decreased can depend on how restrictive the application is. For instance, a highly-dependable application may claim for specific issues that enhance the dependability of the communication system. Concretely, as explained in Section 6.2, it is important not to allow a node to be in the error-passive state in order to reduce the probability of *data inconsistency*. The fault-diagnosis mechanisms of the hub consider the behavior that characterizes the error-passive nodes as incorrect (i.e. passive error flags are considered as erroneous contributions) and then, such nodes are eventually isolated. However, it may be also necessary to apply a tight *penalization policy* in order to minimize as much as possible the probability of any node entering in the error-passive state.

We consider that a first good approach is to adopt a *penalization policy* based on the standard CAN [ISO93], but introducing slightly modifications in order to be more strict. Such policy presents the following rules.

- When the BFC Manager detects a bit-flipping error in its corresponding contribution, it increases its BFC in 8 units. This corresponds with the value a CAN node must increase the TEC or the REC when detecting a primary error in a CAN bus. As explained in Section 8.3, the BFC Managers can detect that their nodes issue incorrect bits in situations that cannot be detected in a CAN bus and, in addition, they improve the accuracy of detecting which nodes are the responsible for an error situation. Thus, although the BFC Managers use the same values used in CAN to increase the error counters of guilty nodes, the BFC Managers are stricter.

- When the BFC Manager detects in its corresponding contribution an error related to the format of the active error frame, it will not increase its BFC in 8 units as it was just said, but in 16 units. This is because extra errors during an error signaling are a good indication of a bit-flipping behavior. Even more, if a node reaches the error passive state, this stronger penalization during an incorrect error signaling allows the corresponding BFC Manager to reduce the

time needed to isolate that node. See Section 7.5 for a detailed explanation about the mechanisms for detecting errors during an error signaling.

- When the state of the *resultant frame* reaches the idle field (i.e. no transmission is observed on the channel) after a data or remote frame has been successfully broadcast, all the BFC are decreased in 1 unit. Notice that, this asymmetric approach of increments and decrements is intended to require high reliability of the nodes and links.

- The bit-flipping threshold can be set to 127 units. Which corresponds to the threshold specified in the CAN protocol to lead a node to enter in the *error passive* state. Note that in the case the accuracy of the error detection performed by the BFC Manager was the same as the accuracy of the error detection of a CAN node, this threshold could prevent a BFC Manager from disabling the contribution of its node before it enters the error passive state. However, since the error detection performed by the BFC Manager is more accurate than the one performed by a CAN node, it is even less likely that a CAN node enters in the error passive state before its corresponding BFC Manager disables its contribution.

Even when this policy is more exacting than the policy specified in CAN, it is important to note that the hub cannot ensure that a node never enters in the error-passive state by simply adopting a stricter *penalization policy*. It should be necessary to do a further analysis in order to find a proper policy that ensures that the hub isolates any node before reaching this state.

Fortunately, an additional solution may be used without addressing this further and complex analysis. This solution is based on the fact that it is possible to force a CAN node to enter in the error-active state after a reset action. Thus, it is possible to avoid that a node communicates while being error-passive by building the software executed in each node so that it resets the node whenever it enters into such state. Note that to restart the operation of a faulty node in this way does not negatively affect the communication. This is because, anyway, the hub will eventually isolate the port corresponding to an error-passive node.

Finally, as regards the *Bit-Flipping Detection Counter* (BFDC) and the *Bit-Flipping Detection Threshold* (BFDT), it would be necessary to choose values for them that allow overcoming the problems derived from an arbitration misunderstanding, while mitigating the loss of performance of a CAN network.

Notice that in Section 8.4.2 we explained that this loss of performance only happens if the hub aborts frames that would not be aborted in a standard CAN system.

Therefore, the values of the BFDC and the BFDT should ensure that the probability of aborting such frames is negligible. Specifically, we propose to increase the BFDC in 1 unit and to set the BFDT to 5 units. On the one hand, this choice should not noticeably reduce the performance, since the probability that 5 erroneous bits do not provoke the abortion of a data or a remote frame in CAN is around $0.5^5 = 0.031$ (see Section 8.4.2). Furthermore, note that the standard CAN ensures that a data frame or a remote frame is aborted if the number of errors during the frame are lower or equal to 5. If a frame is affected by more than 5 errors, the standard does not ensure that the frame will be detected as corrupted and, then, aborted. Therefore, if the BFDT is set to 5 units and the BFDC in increased in 1 unit, any BFC Manager will force the abortion of a frame only when the error-detection mechanisms of CAN are not able to do it. Actually, this should imply that the performance will not be negatively affected.

8.6 Conclusions

This chapter further discusses some issues concerning the error-detection and fault-treatment mechanisms the hub includes for dealing with bit-flipping faults.

We briefly showed that despite the complexity of these mechanisms, we implemented them in a way that we minimize the amount of internal circuitry of the hub. Firstly, each BFC Manager does not calculate the current state of the *resultant frame*. Instead, the Rx_CAN Module calculates this state and provides them with a set of signals that describe it. Secondly, the Rx_CAN Module also detects some types of errors that affect the *resultant frame* and informs the BFC Managers about them. This saves an important quantity of circuitry; specially because each BFC Manager does not include the logic needed for checking the CRC sequence. Thirdly, the fact that each BFC Manager does not monitor the activity of the other BFC Managers reduces the number of interconnections as well as the complexity of their state machines. Moreover, this also makes the hub design more flexible and extensible for further improvements, e.g. to add new ports and their respective Enabling/Disabling units will not require to change the circuitry of the existing modules.

After this brief discussion about how the complexity of the error-detection and fault-treatment mechanisms affect the amount of hub circuitry, we explained the advantages of these mechanisms. One of the advantages is that the BFC Manager can detect bit-flipping errors beyond the capacity of the typical error-detection mechanisms of CAN. The key issue that enables this is that each BFC Manager observes its port contribution before it is coupled with the other port contributions.

First, this allows the BFC Manager to check if its node starts signaling an error when the node should do that. Second, the BFC Manager can observe erroneous bits sent by its node that, otherwise, would not be detected on the coupled signal because they would be masked by the contribution of other nodes. Third, when its node is acting as a receiver, the BFC Manager can evaluate whether or not it correctly acknowledges (or does not acknowledge) the frame being broadcast. Finally, the BFC Manager overcomes the lack of accuracy the CAN error-detection mechanisms exhibit when they try to detect when the transmitter has sent a frame that does not pass the CRC check.

The other advantage of the mechanisms for dealing with bit-flipping faults is that they overcome some CAN limitations concerning fault-diagnosis. On the one hand, isolation of faulty ports is guaranteed even if nodes stop performing fault-treatment operations or the fault is located at the media. On the other hand, they provide a better accuracy of fault-diagnosis than CAN. The problem of the fault-diagnosis mechanisms of CAN is that they additionally increase the error counter of the transmitter, as well as of any node detecting a primary error. This could be unfair because of two reasons. First, because the transmitter is not always the cause of an error. Second, because a node cannot rely on the detection of a primary error to conclude that it is the cause of an error: additional errors during the error signaling (not necessarily affecting that node) can lead it to incorrectly come to the conclusion about whether or not it has provoked the error.

Once we explained the referred advantages, we described what are the limitations of the mechanisms for dealing with bit-flipping faults, and we proposed some further enhancements. The problem is that in some circumstances the BFC Manager cannot correctly interpret the contribution of its port and, as a consequence, it can unfairly detect errors at its port contribution. On the one hand, a BFC Manager that is de-synchronized with its node at bit level cannot correctly sample its contribution. This has not negative consequences if it leads the BFC Manager to detect an error at its node contribution during a data, a remote or an inter frame in which no error has been detected so far. In fact, to detect an error in this situation is desirable since it implies that the BFC Manager has correctly detected that its node has a problem. However, if the de-synchronization happens during the error signaling, the BFC Manager can incorrectly evaluate whether or not the active error flag its node should send has the correct length. Due to the fact that a de-synchronization cannot lead the hub and the node to consider very different bit sizes, their sample points cannot differ in more than one bit time. Thus, it is enough to slightly modify the BFC Manager so that it not only accepts as valid active error flags composed of 6 consecutive dominant bits, but also those that consist of 5 or 7 consecutive dominant bits.

On the other hand, the BFC Manager can unfairly detect errors at its port contribution if an *arbitration misunderstanding* occurs, i.e. if after the arbitration phase the hub and all nodes do not agree on the role each node plays (transmitter or receiver). To minimize the number of errors that are unfairly detected, we proposed to restrict the number of times that the BFC Manager increases its BFC during a data, a remote or an inter frame. To do that, the BFC Manager should additionally use a Bit-Flipping Detection Counter (BFDC) and a Bit-Flipping Detection Threshold (BFDT). At the beginning of each data or remote frame the BFC Manager resets the BFDC, but it increases this counter each time that it increases the BFC. Whenever the value of the BFDC exceeds the BFDT, the BFC Manager forces the hub to signal an error. This aborts the frame and, in case the errors were caused by an *arbitration misunderstanding*, it allows the BFC Manager to re-evaluate the role of its node in the next arbitration phase.

Finally, we discussed the *penalization policy* of the BFC Manager. The BFT and the number of units that the BFC has to be increased or decreased depends on how restrictive the application is. However, we consider that it is important to apply a tight penalization policy in order to minimize as much as possible the probability of any node reaching the error-passive state. Since the hub improves the capacity and the accuracy of the error-detection mechanisms of CAN, it should be enough a penalization policy similar to the one implemented in CAN. Thus, we propose to adopt a CAN-like penalization policy, but further penalizing nodes that incorrectly signal an error frame. This is because extra errors during an error signaling are a good indication of a bit-flipping behavior. Even more, if a node signals and error-passive flag (which is considered as incorrect), this stronger penalization will allow the corresponding BFC Manager to reduce the time needed to isolate that node. Anyway, it is possible to avoid that a node communicates while being error-passive by building the software executed in each node so that it resets the node whenever it enters into such state. Regarding the Bit-Flipping Detection Counter (BFDC) and the Bit- Flipping Detection Threshold (BFDT), we proposed to use values for them that do not imply a loss of performance when compared with a standard CAN system.

Chapter 9

CANcentrate prototype

9.1 Introduction

Previous chapters explain the motivation, the architecture, the error-detection and the fault-treatment mechanisms of CANcentrate. This chapter describes the fundamentals of its first COTS-based prototype.

The hub of this prototype includes all the mechanisms described in above sections, except the further enhancements that were proposed in Section 8.4 to overcome some limitations of the error-detection and fault-treatment mechanisms of the hub. In other words, the hub does not include any of the mechanisms we proposed to address unfair error detections during an error signaling and after an *arbitration misunderstanding*.

We built this prototype not only for demonstrating the feasibility of the ideas addressed above, but also to evaluate the effectiveness of the error-detection and fault-treatment mechanisms, as well as to carry out a first assessment of the performance of a CANcentrate network. Thus, this chapter also discusses the basics of the experimental platform we have set up and the tests we carried out to verify the correct behavior and the performance of the prototype.

9.2 Description of the prototype

The prototype is divided into several parts. Each of them corresponds to a given part or parts of the CANcentrate architecture. The details of such architecture can be found in Chapter 5. When building our prototype, we differentiated the

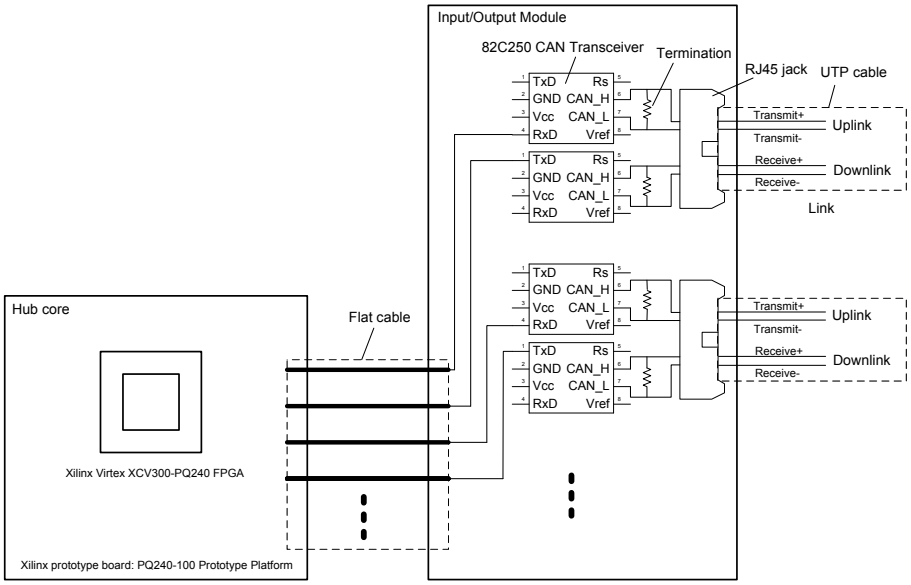

Figure 9.1: Hub prototype

following parts: the Coupler and the Fault-Treatment modules (referred hereafter as the *hub core*), the Input/Output Module, the links, and the CAN nodes. Next, a general description of the characteristics of each part implementation is given.

Figure 9.1 shows the hub prototype. For the sake of clarity only the most important connections are depicted. The hub core has been implemented using the VHSIC Hardware Description Language (VHDL) and the *Xilinx Virtex XCV300-PQ240* FPGA (Field-Programmable Gate Array), which is placed in the Xilinx prototype board *PQ240-100 Prototype Platform* (*HW-AFX-PQ240-100* version).

A dedicated board has been used for implementing the Input/Output Module, following the wire-wrap technique. This board mainly contains four pairs of Philips PCA82C250 high-speed CAN transceivers [PHI00] and four RJ45 jacks (one jack for each pair of transceivers), so that up to four CAN nodes can be connected to the hub at the same time. The pin CANL (LOW level CAN voltage input/output) and the pin CANH (HIGH level CAN voltage input/output) of the transceiver are then connected to the appropriate pins of the corresponding RJ45 jack. The interconnection between the Input/Output Module and the hub core is made by means of a flat cable, which connects the specific reception and transmission pins of the CAN transceivers with the corresponding pins of the Xilinx prototype board.

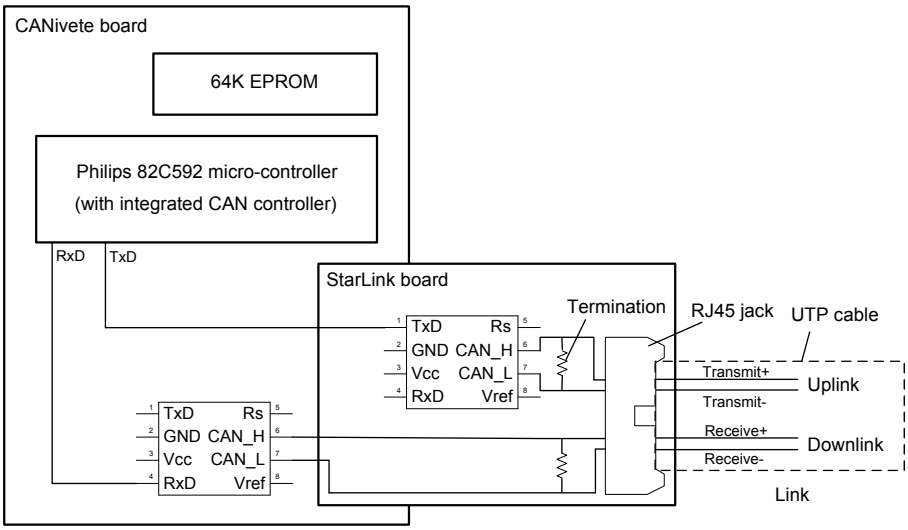

Figure 9.2: Basics of the CANcentrate node prototype

One UTP (Unshielded Twisted Pair) Category 5/5e/6 ethernet cable is used for implementing each link, which is constituted by an uplink and an independent downlink (as explained in Section 5.3). Both the uplink and the downlink use two-wire differential lines. The uplink uses the Transmit pair while the downlink uses the Receive pair of the Ethernet cable. On the one hand, the CAN_H and the CAN_L wires of the uplink are implemented with the Transmit+ and the Transmit- ethernet wires respectively. On the other hand, the CAN_H and the CAN_L wires of the downlink are implemented with the Receive+ and the Receive- ethernet wires as well.

The CAN nodes have been implemented using commercial-off-the-self (COTS) components. Figure 9.2 shows the basics of the node prototype. Each node is constituted by two different boards that are attached to each other: a *CANivete* board and a *starLink* board. The CANivete board is a previous development of the Universidade de Aveiro (UA) for standard CAN applications and implements a typical CAN node. In contrast, the starLink board was specifically designed for this project. It includes all the additional components needed for modifying the CAN interface of the CANivete in order to build the schema of double transceivers needed for connecting each CAN node to the uplink and the downlink of CANcentrate (see Figure 5.2 and Section 5.3 for a description of such schema).

On the one hand, the CANivete is based on a printed board where the compo-

nents are welded. Its main components are a Philips 82C592 microcontroller which integrates a CAN controller; several sets of input/output pins that are connected to different parts of the board for digital or analog I/O; an external EPROM memory of 64k (for storing the program); two RS-232 drivers, one connected to the internal UART of the microcontroller and another one connected to the I/O pins of the board; and a Philips PCA82C250 high-speed CAN transceiver, which is connected to the CAN controller located within the 82C592 microcontroller.

On the other hand, the starLink is a wire-wrap board which contains a Philips PCA82C250 high-speed CAN transceiver and a RJ45 jack. The transceiver located within the CANivete is used for connecting the CAN node to the downlink, whereas the transceiver located within the starLink board is used for the uplink. The printed track of the CANivete which connected the transmit data pin (TxD) of the CAN controller to the TxD pin of the PCA82C250 transceiver was modified to be connected to the TxD pin of the transceiver of the starLink; the transmit TxD pin of the transceiver of the CANivete was left open. The pins CANH and CANL of both transceivers are then connected to the appropriate pins of the RJ45 jack.

9.3 Experimental platform

The prototype of CANcentrate was extensively tested to check its correct operation under error-free conditions and in the presence of faults, as well as to measure its performance. To perform these tests, an experimental platform was built. Specifically, the issues that were taken into account when devising this platform are the application that is executed at the CAN nodes, the configuration of the network and the fault-injection mechanisms.

All CAN nodes run the same application, which is constantly trying to send data frames with different identifiers and different data lengths, in order to test different frames. However, although the CAN nodes execute the same application, each one of them uses a different set of frame identifiers. In this way, it is impossible that two nodes try to send a frame with the same identifier at the same time (this is a general requirement for any CAN application).

In addition, since we wanted to assess the correctness of the CAN arbitration mechanism, we force an arbitration at the beginning of the transmission of each frame. For this purpose the application follows two basic rules: it must trigger a new transmission whenever it successfully transmits a frame and it must restart the CAN controller whenever, due to errors, it reaches the *bus-off* state (in such state, a CAN controller is not involved in bus activities, see Section 3.3.5).

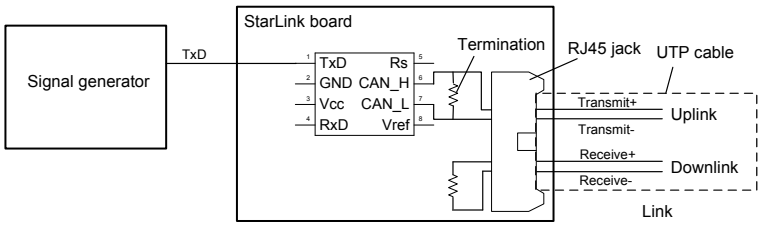

Figure 9.3: Faulty node prototype

Moreover, by means of these rules we achieve the maximum network utilization with a given bit rate. This allowed us to assess if the hub is able to correctly analyze any two consecutive frames that are separated by a minimum delay (given a specific bandwidth).

With regard to the second issue of the experimental platform, namely the configuration of the network, it covers several aspects that are related to the nodes, to the links and to the bit rate. Concerning the nodes, it is worth noting that at least three CANivete nodes are needed for forcing an arbitration to take place in the transmission of each frame. This is because our CAN nodes are not able to perform a new transmission just after finishing a previous one (they have a single transmission buffer and, thus, an extra delay is needed for configuring and ordering a new transmission). As stated before in Section 9.2, the Input/Output Module has been built to allow the connection of four CAN nodes at the same time. However, although we could include four CANivete nodes, one of the ports of the hub was reserved for fault-injection purposes as will be explained later in this section. Therefore, the network finally included three CANivete nodes.

Regarding the other aspects covered in the network configuration, the links and the bit rate, several Ethernet cables of different lengths, as well as different bit rates were used in order to measure the performance of the network depending on the star diameter and the bit rate. Nevertheless, due to implementation limitations on the clock oscillators of the CAN nodes, the maximum bit rate that was used for testing the performance is 690 Kbit/s.

Finally, the last issue related to the experimental platform is the set of fault-injection mechanisms that are used to validate the fault-treatment capabilities of the hub. As explained in Chapter 6, the hub is able to detect permanently faulty ports that present stuck-at-recessive faults, as well as to diagnose and isolate permanently faulty ports that present stuck-at-dominant or bit-flipping faults. Stuck-at-recessive faults can be easily injected by disconnecting the link of an operational CAN node

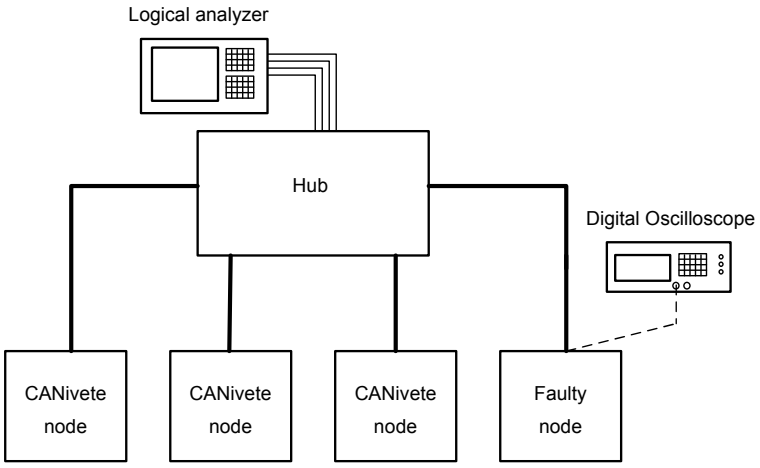

Figure 9.4: Experimental platform

from the hub. However, a more complex fault-injection mechanism is needed for stuck-at-dominant and bit-flipping faults.

For injecting both stuck-at-dominant and bit-flipping faults, a special CAN node, called *Faulty node*, has been implemented (see Figure 9.3). Such node is implemented with a stand-alone starLink board that is connected to a signal generator device (see Section 9.2 for a detailed explanation of such board). The *Faulty node* is connected as any other node (by means of an Ethernet cable) to the port of the hub that is reserved for fault-injection purposes. Note that since the *Faulty node* only has one transceiver, which is connected to the uplink within the cable, the downlink is left open at the end of the *Faulty Node*.

The transmit data input pin of the transceiver of the *Faulty node* is connected to the signal generator device. In this way, different bit stream patterns, consisting of a periodic signal that alternates from the recessive to the dominant value with a given frequency, can be transmitted to the hub. How to use the *Faulty node* to inject stuck-at-dominant and bit-flipping faults is explained in the next section.

Figure 9.4 shows a schema of the experimental platform. Notice that it includes a logical analyzer and a digital oscilloscope, which are used to monitor different parts of the network, e.g. the state of some internal state machines of the hub, or the value of the faulty stream the *Faulty node* injects.

9.4 Functional tests

As explained in Section 9.3, the correct operation of the prototype under error-free conditions, as well as in the presence of faults was checked by means of several functional tests. The aspects that have been tested under error-free conditions are the correctness of the:

- Operation of the different state machines that constitute the hub.
- Calculation of the *resultant frame* upon all node contributions.
- Correct synchronization at bit level and at frame level.
- Assignation of the roles of the nodes after the arbitration phase.

In contrast, the aspects that have been tested in the presence of faults are the correctness of the:

- Increase and decrease of the different error counters during different fault scenarios.
- Detection of ports suffering stuck-at-recessive faults, as well as the isolation of ports suffering stuck-at-dominant or bit-flipping faults.
- Reintegration of ports following the policy explained in Section 6.7.

All the issues indicated above were tested at two different levels: at the level of the VHDL design of the hub and at the level of the physical network. However, the tools that have been used at each of these two levels impose different limitations. Thus, the different aspects listed above have been tested in different depths at the two levels.

The first level of testing, the functional testing of the VHDL design of the hub, has been done by means of the simulation tool *ModelSim XE II 5.7g* (provided by Mentor Graphics Corporation). Several simulations were done in order to check all the issues specified above and, in all cases, the operation of the hub was correct. Special attention has been paid to check the correct operation of the different state machines that constitute the hub, as well as their correct mutual interaction.

As concerns the second level of testing, physical limitations in the layout of the FPGA board discourage an exhaustive testing of all the state machines that constitute the hub. In contrast, many more fault scenarios can be injected at physical level than at simulation level.

For this physical level of testing, different parts of the physical network have been observed by means of a logical analyzer and a digital oscilloscope. In particular, the ports of the hub were observed in order to know which is the contribution of each node as well as the value of the coupled signal. Since the Rx_CAN Module and the Enabling/Disabling units are key modules for synchronizing the hub at bit level and at frame level, as well as for diagnosing and isolating faulty ports, respectively, they have also been observed.

For physically testing the correct operation of the network under error-free conditions (like during the phase of the tests of the VHDL design), the correct calculation of the coupled signal, the correct synchronization at bit and at frame level and the correct assignment of the roles during the arbitration phase have been checked. With regard to the physical testing of the fault-treatment mechanisms the hub implements, extensive tests that include stuck-at-recessive, stuck-at-dominant, bit-flipping faults and the reintegration policy have been performed.

Specifically, in order to physically testing the actions carried out by the hub in the presence of stuck-at-recessive faults, the link of a previously operating node has been mechanically disconnected. When the link is disconnected a transient bit-flipping behavior is observed in its corresponding port. However, these erroneous bits are not enough for leading the hub to isolate the port. In contrast the contribution of the port quickly stabilizes to the recessive value and then, the hub indicates that the port is at the *idle* state. Which actually means that the port is stuck-at-recessive (see Section 6.7).

For physically testing the operations the hub performs in the presence of stuck-at-dominant faults, the *Faulty node* described in Section 9.3 was used to transmit a periodic signal that keeps the dominant value during many frames. It was observed that the hub correctly increases the DBC and isolates the corresponding port whenever the configured Dominant Bit Threshold (DBT) is exceeded.

As concerns the fault-diagnosis and fault-isolation operations the hub performs in the presence of bit-flipping faults, two kinds of techniques for injecting them have been used. On the one hand, a bit stream, which has random values, was injected by means of mechanically connecting/disconnecting a given link into its plug. On the other hand, the *Faulty node* was used for injecting a bit stream that changes from the recessive to the dominant value with different frequencies that do not match with the bit rate the nodes use for communicating. Notice that in both cases, the beginning of the bit-flipping injection was randomly chosen. Many tests were performed with both techniques and in all situations the results were correct.

Finally, for the physical testing of the reintegration policy, the state (*idle*, *active* or *disabled*) of a given port was observed in different situations (see Section 6.7

for an explanation of the different states of the ports). After the system start-up the port was in the *idle* state. When the node sent an ACK bit or when it tried to send a frame, the port state changed to the *active* state. If the node was at the *active* state and its link was disconnected from the hub, the port returned to the *idle* state. After the port was isolated due to a stuck-at-dominant or a bit-flipping fault, a recessive value was forced in this port by disconnecting its link or by compelling its node to send recessive bits. In these cases, the hub re-enabled the contribution of the port, which agrees with the expected behavior related to the reintegration policy.

9.5 Performance measurements

Regarding the performance tests, some measurements have been made. The values of the FPGA device utilization needed for implementing the hub prototype (with 4 ports) are: 758 out of 3072 slices, 278 out of 6144 Flip-flops, 1396 out of 6144 LUTs, 91 out of 170 IOBs, 4 out of 4 GLCKs.

The extra delay introduced by the hub core is 35 ns, whereas the average value of the extra delay introduced by the entire hub is 155 ns. Notice that the value of the extra delay introduced by all the hub is of the order of 1/6 of the bit time when operating at the higher bit rate allowed in CAN [CiAa] (1 Mbit/s).

In addition, the hub core has been also built with 16 ports. The values of the FPGA device utilization in this case are: 2534 out of 3072 slices, 869 out of 6144 Flip-flops, 4662 out of 6144 LUTs, 91 out of 170 IOBs, 4 out of 4 GLCKs. It has been observed that the extra delay introduced by the hub core does not visibly depend on the number of ports it is provided with.

Finally, several Ethernet cables of different lengths, as well as different bit rates have been used in order to measure the performance of the network depending on the star diameter and the bit rate. As said before, due to implementation limitations, the maximum bit rate that has been used is 690 Kbit/s. At this bit rate, the maximum star diameter that was used without generating errors is 41 meters which implies a small reduction in length when compared with a CAN bus operating at the same bit rate (maximum length of approximately 68 meters) [CiAa].

Moreover, if we use Equation 5.1 considering that $t_h = 310$ ns, i.e. two times the measured hub extra delay, we see that the bit rate, B', of an equivalent CAN bus with a bus length equal to the CANcentrate diameter approximately is 878 Kbit/s. The maximum CAN bus length achievable at this bit rate is in the range of 30 to 50 m [CiAa], whose average point (40 m) almost coincides with the diameter at which CANcentrate starts experiencing sporadic errors.

Finally, it is noteworthy that the delay introduced by the hub could be reduced using specific high speed transceivers in the hub. In this way the current delay of the hub, 155 ns, could be reduced until 55 or 65 ns, without losing compatibility with COTS components when building nodes and links.

9.6 Conclusions

This chapter describes the basics of the implementation of a first prototype of CANcentrate, as well as the experimental platform and the tests we carried out to assess its correct behavior and performance.

The entire prototype was built using COTS components, except the *hub core*, which includes the Coupler and the Fault-Treatment modules, and which has been synthesized on an FPGA.

We verified the correct behavior of the hub at the level of the VHDL design and at the level of the physical network, in the presence of both error-free conditions and faults. At the level of the VHDL design, we exhaustively verified the correct operation of all the automata that constitute the hub via simulation. However, it was not possible to inject many scenarios involving faults at this level. In contrast, we were able to inject many more fault scenarios at the physical network. In this case, due to limited number of probes of the logical analyzer, it was not possible to monitor all the state machines of the hub. Fortunately, to monitor the state of the main machines of the Rx_CAN Module and the manager modules, as well as the error counters was enough for verifying the correctness of the error-detection and fault-treatment mechanisms of the hub.

Regarding the performance of the prototype, several cables of different lengths, as well as different bit rates have been used in order to measure the performance of the network depending on the star diameter and the bit rate. It has been observed that the extra delay introduced by the hub implies a negligible reduction of length when compared with a CAN bus operating at the same bit rate. Furthermore, it has been found that the extra delay introduced by the hub does not depend on the number of ports. In fact, the transceivers of the Input/Output Module represent the major part of the extra delay introduced by the hub. Thus, this delay could be reduced using specific high speed transceivers in the hub, without losing compatibility with COTS components when building nodes and links.

Finally, notice that some implementation decisions were merely taken for practical reasons. The Input/Output Module of the hub was built using the wire-wrap technique. Although this technique is not the appropriate for building a robust

device, it allowed quickly implementing the hub. Similarly, the links were not built with typical CAN cables, but with Ethernet ones. Despite creating a small impedance mismatch, Ethernet cables were easier to deploy. Also notice that for implementing each CAN node we used an already available CANivete board, just adding a small board that includes the components needed to connect the node to the uplink and the downlink. Although this allowed us to save time when developing the prototype, limitations on the clock oscillators of the CANivete boards restricted the maximum bit rate that could be used for testing the performance of CANcentrate. Anyway, despite the mentioned implementation limitations, this first prototype represents a proof of concept of the feasibility and the potential advantages of CANcentrate.

Chapter 10

Reliability evaluation of CANcentrate

10.1 Introduction

As explained in Sections 1.2 and 4.5, star topologies can represent a step towards improving dependability of communication networks, in general, and of CAN in particular. Specifically, the interest in using star topologies has been growing, given their better error containment capabilities and their resilience to spatial-proximity and common-mode failures [Kop03]. For example, Ethernet has long ago moved to star topologies that are nowadays used in industrial automation and the embedded systems domains. In the particular case of in-vehicle systems, we can find other examples of transition to star topologies such as with TTP/C [BKS03] and FlexRay [Fle05]. Moreover, as explained in Section 4.5, different star topologies have also been proposed for CAN.

Despite this growing trend towards the use of star topologies, it is not so clear whether or not stars improve the dependability of systems that rely on them. In fact, stars can provide better error containment but they also include more hardware components, thereby increasing the probability that faults and errors occur. Furthermore, the enhancement of system dependability that can be achieved when using a star instead of a bus has never been appropriately quantified. On the one hand, previous work, e.g. [RD88], [Cao97], [FLS02], [ADS03], [LJ90], that quantitatively analyzes the dependability of different network topologies by means of mathematical or statistical models abstract away many important details: the differences between the failure rates of the nodes of a star and a bus; the different components' failure modes; the specific capacity of the hub for con-

taining errors caused by different types of faults, etc. On the other hand, fault injection experiments carried out for some technologies, e.g. for TTP/C [ABST03] and FlexRay [DLMSS08] [MSH08], quantitatively demonstrate that a star is better suited to prevent the propagation of errors than a bus. However, data collected from these experiments have not been used so far to quantify the dependability improvement that can be actually achieved by means of the stars' error-containment capacities.

Among all the dependability attributes (see Section 2.2), we are interested in quantifying the benefits that stars yield in terms of reliability. As explained in Section 1.4.3, we are specially interested in this attribute because CAN can still play a key role in newer applications that are demanding increased levels of reliability, and for which alternative solutions have been proposed to complement or to replace this protocol. These applications embrace different domains such as automotive and home automated systems.

The objective of this chapter is twofold. On the one hand, it is devoted to carrying out a fair comparison of the system reliability that can be achieved with CAN and CANcentrate, by means of models that include parameters for all the relevant aspects of a system relying on these infrastructures. In this sense, we determined the values of our models' parameters that characterize CAN and CANcentrate with special care not to favor the star in the comparison. For instance, since there are many ways of implementing a CAN bus and a CANcentrate star, each yielding different reliability results, we always choose the implementation options that favor the case of CAN. Thus, results herein presented are likely to be lower bounds to the reliability that can be achieved with CANcentrate. On the other hand, this chapter aims at carrying out sensitivity analyses with respect to some of the system's aspects that are parameterized, e.g. the ability of the hub to contain errors, in order to assess how they influence the reliability that can be achieved with CAN and CANcentrate.

In order to model the reliability of a system that relies on both CAN and CANcentrate we used the Stochastic Activity Networks (SANs) formalism. This formalism is an extension to stochastic Petri Nets [CFJ+91] [TMGT93] [SoT04]. More details about SANs are given later on in Section 10.5.

We take as a reference for comparing CAN and CANcentrate the reliability requirements of a typical highly reliable application. This is because although we are interested in using CANcentrate as a general field-bus network, one of the motivations of this dissertation is to enhance the error containment of CAN to make it suitable for distributed control systems that demand an increasing level of reliability. More specifically, since CAN was initially designed for in-vehicle communi-

cations, we consider the reliability requirements of the less demanding x-by-wire applications in cars [MK05], e.g. of throttle-by-wire and of some brake-by-wire systems. As explained in Section 1.4.3, x-by-wire systems are devoted to substituting the mechanical and hydraulic control mechanisms in vehicles by electronic parts.

Notice that since star topologies were mainly proposed to address the issue of permanent hardware faults, this chapter focuses on these faults and leaves aside transient ones (see Section 2.2). Moreover, transient faults, by definition, cannot prevent nodes from communicating indefinitely. Instead, they cause temporary unavailability of the communication system thereby negatively affecting its performance, e.g. deadline violations, increased average response times, packet losses, etc. Therefore, the impact of transient faults on reliability is necessarily application dependant and, thus, they are beyond the scope of this dissertation. For example, deadline violations can lead to a generalized failure in hard real-time systems; but this strongly depends on the specific set of messages and the scheduling [HNNP02]. Response times can also affect the reliability of some types of control system, e.g. of life-critical ones [TMGT93]. But similarly to what happens with deadline violations, the way in which performance affects reliability depends on the application.

Nonetheless, this chapter is not intended to bring a definitive assessment in the presence of permanent faults either. Firstly, as explained above, we characterized system's aspects by means of parameters whose values guarantee that results are not biased towards CANcentrate and that, thus, lead to obtain lower bounds of the system reliability achievable by the star. However, in some cases, these kind of decisions may have been too detrimental for CANcentrate.

Secondly, we consider that semantically-incorrect frames, e.g. timely-incorrect frames issued by babbling-idiot nodes (see Section 3.4) are not contained. However this type of failures could be treated by simply including a bus guardian within the hub, whereas in the CAN bus it would be necessary to include extra hardware in each node. Thus, the potential benefits that CANcentrate could yield when dealing with these failures are not reflected in the results.

Thirdly, we do not take into account faults happening at the software executed at nodes. This fact should favor the CAN bus in the comparison, since software faults are the main source of semantically-incorrect frames, which could be treated by the hub as just said. Moreover, since a hub cannot suffer from software faults, the negative reliability impact of including the hub as a new source of faults when compared with a bus topology diminishes.

And fourthly, the minimization of the impact of spatial-proximity faults that is

inherent to a star topology is not included in the models. This is because it is impossible in practice to find a reasonable value for the probability with which an external event, e.g. a blow on a vehicle's side, affects a given number of components in a bus or in a star.

Due to the mentioned limitations, results presented in this chapter are just an evidence of how CANcentrate can improve the reliability of CAN-based distributed control systems; but they do not reflect its full potential.

Finally, it is noteworthy that although this chapter is devoted to comparing the reliability of a system that relies on the CAN bus with an equivalent system that relies on CANcentrate, we will also compare them with the system reliability achievable with ReCANcentrate (see Chapter 12). As will be explained, this means that, in fact, the major part of of the modelling decisions explained in this chapter were made to be also appropriate for the case of ReCANcentrate.

10.2 Metrics

As just said, among all the attributes of dependability, e.g. availability, maintainability, etc., we are interested in reliability. Notice again that reliability is defined as the probability with which a system continuously delivers its intended service throughout a given interval of time (see Section 2.2). Such a definition implies that the reliability is an attribute that is strongly related to the particular characteristics of a system, since for each system the delivery of the intended service is defined differently.

In this sense, as already explained in Section 1.4.3, we can differentiate between two types of systems. On the one hand, there are systems that can only deliver their services as long as all their nodes are non-faulty and can communicate with each other. We refer to them as *non-fault-tolerant/accepting systems* (NFT/A systems). On the other hand, we can find systems that are able to accept or tolerate the failure or the disconnection of up to k of N nodes. We call them *fault-tolerant/accepting systems* (FT/A systems). Examples of systems that accept the failure or disconnection of several nodes could be a factory plant in which it can be accepted that up to k of N production lines are inoperative, or the intra-building communication subsystem of a hotel or a house. As concerns systems that tolerate nodes' failures and/or disconnections, clear examples are safety-critical ones, such as those used in avionics, which normally provide fault tolerance by means of redundancy.

This classification allows to better understand the role that the dependability-related features of a simplex star topology such as CANcentrate can play in order

to improve system reliability. On the one hand, the resilience to spatial-proximity and common-mode failures and the potential error-containment mechanisms of a simplex star are useless to improve reliability of NFT/A systems. This is because to isolate a fault in a simplex star entails isolating a given node, thereby inexorably provoking the failure of an NFT/A system. Moreover, since a star topology includes more hardware than a bus to interconnect a given ensemble of nodes, it is expected that faults are more likely to occur in the former, thereby reducing the reliability of an NFT/A system.

In order to quantify this reduction, we define a metric called *non-fault-tolerant / accepting system reliability* (NFT/AR) as the *probability with which all nodes of a system can correctly operate and communicate with each other throughout a given interval of time*. As will be described later on, we use this metric to quantify the reliability of an NFT/A system that relies on CAN and of an equivalent system that relies on CANcentrate and, then, we compare them.

On the other hand, although a simplex star is likely to provoke a reduction in terms of NFT/AR, it is still possible that the mentioned features of a star can improve the reliability of FT/A systems. We discussed this issue in Section 1.4.1, where we explained that those features allow reducing the multiple points of k-severe failure of a bus to one point of k-severe failure (the hub), thereby minimizing the number of nodes that are affected by the errors that faults generate. Examples of those FT/A systems could be the intra-building communication system of a hotel, in which the main objective is to provide service to the maximum number of rooms, as well as highly reliable distributed control systems that tolerate faulty or disconnected nodes by, for example, replicating them.

In order to quantify the improvement of reliability CANcentrate can yield for FT/A systems, we define a new metric we call *fault-tolerant/accepting system reliability* (FT/AR). More specifically, let us specify a parameter called k for this metric, so that we can denote it as FT/AR$_k$. Thus, the FT/AR$_k$ is formally defined as the *probability with which at least $N - k$ of the N nodes of a system can correctly operate and communicate among them throughout a given interval of time*. In other words, the FT/AR$_k$ is the probability of not suffering a k-severe failure. As already said, a k-severe failure occurs when few than $N - k$ of N nodes can operate or communicate among them.

Particularly, we are interested in the FT/AR$_k$ when $k = 1$, i.e. we focus on the FT/AR$_1$. In this way, we study the potential benefits that CANcentrate can yield in terms of reliability for FT/A systems that can accept or tolerate that at most one node fails or cannot communicate. We selected this value of k because it is the one with which CANcentrate intuitively yields the least reliability benefits. To better

understand this issue, notice that when the number of nodes that fail or cannot communicate reaches the value of k, then the ability of the hub to contain errors becomes useless, since a new fault will inevitably lead to a generalized (k-severe) failure even though the hub can isolate it. Therefore, the hub of CANcentrate has more opportunities to contribute to the system reliability as the value of k increases.

10.3 Modelling limitations

Dependability modelling is a discipline whose application presents some limitations that must be taken into account to understand its usefulness and to adequately interpret the models and results presented in this chapter.

Maybe the most important limitation of dependability modelling is the impossibility of reflecting all the characteristics of a real system, so that the model is actually an abstraction. This is because the computational resources needed to solve a model and obtain numerical results easily become unattainable as more details are included in the models. As a consequence, every model relies on a given set of assumptions and it is inherently inaccurate.

Although there is not a definitive solution to overcome this problem, it is still possible to model the essential characteristics of a system in order to extract conclusions about how different design alternatives and assumptions affect a given dependability attribute such as the system reliability. In this sense, we have explored and compared different modelling strategies in order to model as many system's details as possible, while keeping a reasonable computation time. In fact, to our best knowledge, no one has previously modelled a system relying on a bus or a star topology taking into account the details included in the models herein proposed.

Another important limitation of dependability modelling is the fact that it is hardly possible to find numerical values that accurately quantify specific system's characteristics related to dependability such as, for example, the failure rate of the components, the proportions of the different failure modes and the coverage of some error-containment mechanisms. In order to mitigate this limitation, we made a great effort to find values that can be considered as realistic for these characteristics. Specifically, we took into account real implementation and technological aspects and, when possible, we also used widely spread prediction standards.

Nevertheless, in order to avoid biased results towards CANcentrate, assumptions for system's characteristics were always taken in favor of CAN and, in some cases, this may have been too detrimental for CANcentrate. Therefore, it becomes essential to analyze how the system reliability varies depending on the value of its

characteristics. Moreover, this analysis is very important for identifying which ones have a bigger influence on reliability. Because of these reasons, as will be explained later, we have built our models so that the value of different system's characteristics are parameterized. Even more, this parametrization allows our models to be more easily adaptable to assess the reliability improvement achieved when using a star in technologies other than CAN.

From the above discussion concerning the limitations of dependability modelling, it is easy to understand that the objective of this discipline is not to provide absolute figures for different dependability attributes. In contrast, its objective is to guide the design and implementation of a system by analyzing how different options and decisions affect its dependability [TMGT93]. Therefore, the comparison carried out in this chapter aims not only at assessing whether or not it is possible to improve the reliability of CAN-based systems; but also at analyzing how different aspects of the system and the communication infrastructure affect this improvement.

10.4 Modelling assumptions

This section introduces the assumptions our models rely on. It is important to note again that each one of the assumptions our models are grounded on has been made guaranteeing that the results are not biased towards CANcentrate. Moreover, when we had to make a choice for any of these assumptions we always took the option that was favorable for the CAN bus. Some of these decisions may have been too detrimental for the CANcentrate results.

Some modelling assumptions condition the structure of the models, whereas other ones are reflected as model parameters. As already explained, these parameters are specially valuable since they allow performing sensitivity analysis with respect to fundamental dependability-related characteristics of the system for which it is hardly possible to find exact values.

Table 10.4 shows the parameters that are common to the models of the CAN bus and CANcentrate. Moreover, these parameters also belong to the model of ReCANcentrate. In addition, Table 10.5 shows the parameters that are specific to the model of the CAN bus; whereas Table 10.6 presents the parameters that belong to the model of CANcentrate. Finally, Table 12.1 specifies the parameters that belong to the case of ReCANcentrate. The parameters of this last table are beyond the scope of this chapter and will be addressed in Chapter 12. Each one of the three first tables shows the name of each parameter, its default value and a short explanation. The meaning of the parameters will be better understood in the next

sections.

As will be explained in Section 10.9, we use the default values specified in those tables to initially set up a case of reference in which we compare the reliability of a system relying on CAN and on CANcentrate. Then, we perform some sensitivity analysis with respect to the major part of the parameters, in order to assess how they affect the benefits of CANcentrate.

Default values are those we will propose as reasonable in the following subsections. In this sense, figures we estimate in these sections should not be taken as real values, but as initial reference values that can be considered as realistic.

10.4.1 Implementation assumptions

Firstly, it is necessary to establish what features of the different CAN physical layer standards are included in the implementation of the CAN bus and CANcentrate. This is because different CAN physical layer standards exhibit different degrees of fault tolerance and electrical robustness and, hence, a fair comparison between a CAN bus and a CANcentrate star must consider that both use the same physical layer standard. We consider the ISO 11898-2 High-speed CAN standard [ISO03b], which is the most widespread one. This standard specifies a two-wire differential bus line terminated at both ends with impedances of 120 Ohm (see Section 3.2). High-speed CAN does not compel to tolerate faults affecting any of the two wires or any of the bus line terminations. Thus, we assume that a fault affecting any of the wires or terminations will not be tolerated.

Secondly, we need to address the length and layout of both the bus and the star; how nodes are attached to the medium; the quantity of wires included in each cable; and how the terminations are implemented. We assume that the CAN bus length and the CANcentrate diameter are of 100 m, which is the maximum bus length that can be achieved operating at 500 Kbit/s [CiAa] (a half of the maximum CAN bit rate). Moreover, in the case of the CAN bus we consider that nodes are equidistant, whereas in CANcentrate we suppose that every pair of nodes is separated by a length equal to the star diameter (100 m) and that all links have the same length (50 m). Note that such layout is pessimistic for CANcentrate, since a star could use much less cable to cover the area occupied by the nodes the bus interconnects.

There are several possible ways to attach a node to the bus line in the case of CAN and to the link in the case of CANcentrate. Figure 10.1 depicts two examples of bus layouts. Case A shows a daisy chain configuration with each pair of adjacent nodes connected to each other using a CAN cable with a straight connector at both its ends. Conversely, case B shows a situation in which each node has its own

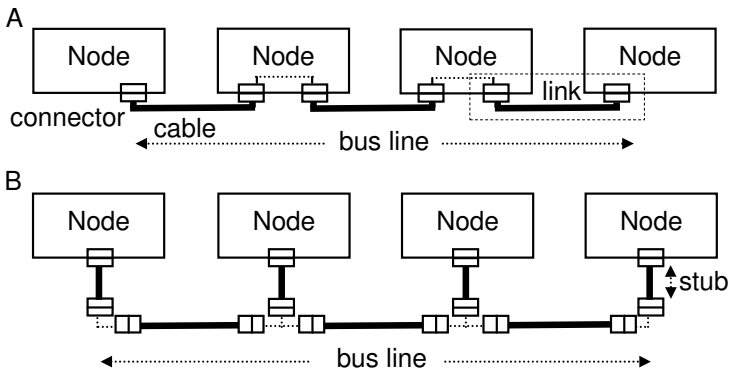

Figure 10.1: Bus layouts

straight connector, which attaches it to the corresponding stub. Each stub is then connected to the bus line by means of a T-connector. The first configuration is the most reliable option for the CAN bus since it includes no stubs and minimizes the number of connectors. Thus, in order to obtain more optimistic results concerning the reliability of the bus, we assumed this configuration for CAN.

For CANcentrate, each link (including an uplink and a downlink) connecting a node to the hub could also be implemented using a single CAN cable with one straight connector at each end. However, for implementing the link in such a way it would be necessary to use a cable with extra wires and straight connectors with more pins than for CAN. Using different types of connectors and cables for CAN and CANcentrate could pose some uncertainties concerning the fairness of the comparison. Hence, we decided to consider that each uplink is implemented separately from its downlink, so that each one of them uses a CAN cable and a pair of connectors equal to those used for connecting each pair of adjacent nodes in the CAN bus.

As concerns the number of wires included in each cable, a higher number of wires implies a worse cable reliability. Usually, CAN cables have one or two mandatory supply wire(s), e.g. GND, V+/GND or V+/V-, in addition to the pair of wires used for the differential transmission. Hence, in order to cope with the most general case, we decided to consider that a cable consists of four wires: CAN_H, CAN_L, GND (or V-) and V+.

Notice that, as indicated above, in order to consider the same type of connectors for CAN and CANcentrate, we have assumed that in each link the uplink and the downlink are independently implemented in separated CAN cables. Thus,

each link of the star includes an extra V+/GND pair that is unnecessary in practice, thereby biasing the results in favor of the CAN bus. In other words, this is a pessimistic assumption concerning the cabling of CANcentrate since the uplink/downlink pair could be implemented in a single CAN cable that would include the two differential-wire pairs, but only one GND/V+ pair.

Concerning the terminations, we consider the typical approach of using special connectors that already have the terminations inside and attach them at both ends of the bus or of each uplink/downlink.

Thirdly, we need to take into account the way in which the hub is built and its features that are to be modelled. For the former aspect we decided to follow the basic characteristics of our first CANcentrate prototype hub (see Chapter 9), which consists of a core that implements the wired-AND functionality and fault-treatment capabilities, i.e. the Coupler and the Fault-Treatment modules, and an interface. We assume that the core is implemented in a dedicated IC and that the interface is basically composed of a set of COTS transceivers. For the latter aspect, we consider all the hub functionalities except the CANcentrate's reintegration policy (see Section 6.7) because we only take into account permanent faults.

10.4.2 System components and entities

In order to model the reliability of a system relying on CAN or on CANcentrate it is also necessary to decide what are its constituent components. This step is quite important since to guarantee a fair comparison between CAN and CANcentrate, it is necessary to identify what are the extra components introduced by CANcentrate. This is because, as already explained, although CANcentrate provides more error-containment mechanisms than CAN, it also includes more hardware, thereby increasing the probability that faults occur.

However, it is impossible in practice to model the dependability-related properties of each individual component. Instead, there must be a compromise between the level of abstraction a system is modelled with and how close to the reality the model is. In this sense, we gathered the components of the system into different assemblies or *entities*.

The first entity is the *Node Core*, which includes an oscillator; a microprocessor; a 32Kb SRAM and a 32Kb EEPROM; the corresponding integrated circuit (IC) sockets, and a piece of a printed circuit board (PCB), whose amount of connections depends on the number of components attached to it. This piece of PCB represents the area of the node's PCB the Node Core occupies.

Besides the Node Core we consider an entity called *Controller*, which is com-

posed of a CAN controller, its socket and the corresponding area of the node's PCB.

Similarly, we consider two entities, called *Node IO* and *Hub IO*, which include the components needed to interface a node and a hub port, respectively, with the medium. In fact, these two entities include the same components: one CAN transceiver and its corresponding socket and PCB area. However, we differentiate between these two entities in order to make the model more flexible. For instance, one could decide to use components with a different quality in the node than in the hub and, therefore, these two entities could have different dependability properties, e.g. different failure rates.

Regarding the medium, we consider the *Attachment* and the *Termination* entities. The first one includes a CAN cable and a pair of straight connectors. In CANcentrate, it represents the uplink or the downlink so that two of these entities are necessary to connect a node to the hub. In the CAN bus, an Attachment constitutes a bus section that connects two adjacent nodes. The other entity, the Termination, is merely constituted by a resistor. A pair of Terminations is used in each uplink/downlink, as well as in the bus line to prevent signal reflections.

Finally, we also consider an entity that represents the hub core. As explained in Section 9.2, the hub core includes the Couple and the Fault-Treatment modules. We call this entity *Hub Core* as well, and it comprises a dedicated IC that implements these modules, its socket, an oscillator and the necessary PCB area.

10.4.3 Basic statistical fault properties

A capital aspect that needs to be considered when modelling the dependability of a systems is the statistical properties of faults.

We consider that all faults are permanent and that component failures are independent. The hypothesis that faults are independent is typically made in the context of dependability evaluation because it simplifies the way in which models are mathematically solved. However, this independence should be verified if faults are suspected of being correlated [TMGT93]. In the case of a star topology, its resilience to spatial-proximity and common-mode failures supports the hypothesis that faults are independent in CANcentrate. Conversely, since a bus topology does not exhibit this advantage, to assume that faults are independent is maybe optimistic for the CAN bus. This means, that results may be biased in favor of the CAN bus, which is desirable in the analysis herein presented.

As concerns the characterization of the components reliability, notice that we are interested in evaluating the reliability of the system during its *steady-state opera-*

tion period, which implies that its components have left behind their *infant mortality period* and still have not reached their *wearout period*. See Section 2.2 for an explanation of the phases of a hardware component's life. Based on data collected during many years, the exponential distribution is commonly assumed to be the most appropriate for modelling the Time To Failure distribution of components during the steady-state operational period [KNM90] [Sho02]. Moreover, such a distribution simplifies the way in which the models are mathematically treated to obtain a numerical solution.

Therefore, in order to characterize the reliability of each component during the steady-state operation period we model its Time To Failure distribution, F(t), as an exponential distribution with mean $1/\lambda$, where λ is the constant failure rate of the component expressed in number of failures per hour.

$$F(t) = 1 - e^{-\lambda t} \quad (t \geq 0 \;\; \lambda > 0)$$

Notice that the above expression corresponds to a Non-Defective Time To Failure distribution [MT95]. In this way, if the failure time is X, then the probability with which the component fails at or before time t, F(t) = Probability($X \leq t$), is 0 when $t = 0$, $1 - e^{-\lambda t}$ when $0 < t < \infty$, and 1 when $t = \infty$.

In order to obtain the failure rates of the components we use a software of prediction of failure rates, called Relex [Cor06], taking into account the MIL-HDBK-217 model [DOD95] and the Tellcordia *Method I Case I* calculation method [Cor06].

The MIL-HDBK-217 is the widest accepted model for calculating failure rates, even though it is sometimes considered as pessimistic. The Tellcordia methodology is widely used by commercial organizations. In particular, the *Tellcordia Method I Case I* is used when not all specific data regarding components are available, which is our case. More specifically, this method specifies that the failure rate of a given subsystem (an entity in our case) is obtained by adding the failure rates of its constituent components.

For all components we assume non optimistic technological characteristics, e.g. commercial quality level, CMOS technology, etc. For calculating the failure rate of the IC that constitutes the Hub Core, it was also necessary to specify its number of logic gates, which depends on its number of ports. For this purpose we did not make any assumption, but we used real values obtained when synthesizing it in an FPGA (see Section 9.5).

Finally, some characteristics related to the environment where the system under evaluation is supposed to operate are also needed to predict the failure rate of the components. On the one hand, the *Tellcordia Method I Case I* assumes 40 degrees

Celsius as the operating temperature, and 50 percent rated stress. On the other hand, notice again that, as indicated in Section 10.1, we use as a reference the dependability requirements of in-vehicle communications. Therefore, in addition to the Tellcordia environmental parameters, we decided to use the *Ground Mobile* (GM) operating environments provided by the MIL-HDBK-217, which is better suited for those systems.

10.4.4 Failure mode assumptions

Besides predicting the failure rate of components, it is necessary to take into account the possible failure modes they can exhibit, as well as the proportions of these modes. The concept of failure mode was introduced in Section 2.2 as the effect of a fault on the service delivered by a system or a subsystem. For the particular case of this chapter, we consider that the failure mode of a given component is the way in which a fault affecting that component manifests *from the channel point of view*.

In principle, one could consider that faulty components can only lead to faults included in the fault model the system is able to deal with. In our case, this would mean that a component failure can only provoke a stuck-at-recessive, a stuck-at-dominant, a bit-flipping or a network partition failure (see Section 5.2 for more details about the fault model of CANcentrate).

However, note that, in a real system, components can actually fail in manners that are not included in a given fault model. As explained in Section 2.2, this fact can be formalized by means of the concept of *failure mode assumption coverage* [Pow92], defined as the probability that a component failure mode assumption proves to be true in practice. In order to take this coverage into account, we consider that a component can also exhibit a new type of failure we call *out-of-fault-model* (ofm) failure mode. This mode gathers all faults that are beyond our fault model and that, thus, can be treated by neither a CAN controller nor a hub.

More specifically, we consider that ofm faults are those that lead to the transmission of frames that are syntactically correct, e.g. that do not violate the CAN frame format, but that are incorrect from a semantic point of view. An example could be a fault that changes the data stored in the transmission buffer of a CAN controller before this data is encapsulated and sent within a CAN frame. In this case, although the frame is syntactically correct, it carries semantically incorrect data that can lead nodes to function improperly and provoke a generalized failure. Another example of an ofm fault could be a CAN controller that fails in a *babbling-idiot* manner by continuously sending a message stored in its transmission buffer. The

only semantic faults we do not consider as being out of our fault model are network partition faults, since a simplex star topology inherently prevents them from happening.

Note that we do not consider as ofm faults those that cannot be isolated by means of mechanisms that act at the logical level, e.g. by means of mechanisms similar to the ones included in a CAN controller or in a hub, which isolate an erroneous bit-stream by disconnect the CAN controller itself or by disabling a hub port. This means that faults such as electrical problems or instabilities, e.g. a short circuit of the GND wire to the V+ wire of a CAN cable that causes a fire that propagates along the cabling installation, are out of the scope of this dissertation. We believe that to rule out these type of faults is adequate for several reasons. First, to our best knowledge, reliability analyses of digital systems do not take into account these faults. Second, these faults are typically treated by means of types of mechanisms other than those we address in this work, e.g. by means of fuses, circuit breakers, cooling devices or fireproof cable coatings. Moreover, these failure modes are not only specific to a digital control system, but they are also related to other systems with which the control system interacts, e.g. to the lighting subsystem of a car.

Regarding the proportions with which components fail in specific manners, it is worth noting that there is not a real consensus on that issue. Different analysis centers, e.g. the *Reliability Analysis Center* (RAC), make their own analysis on failure data, thereby coming to different conclusions. This can be seen in [Mee95], which even recommends the reader to interpret the available data concerning failure modes for its own application.

Because of these limitations, we decided to make our own interpretations, but trying to find reasonable values. On the one hand, we consider that, in principle, components exhibit a 0% of ofm failures. Notice that, otherwise, an ofm greater than 0% would prevent us from analyzing the reliability achievable by the CAN bus and our stars, since their contribution to the reliability would be masked by the effect of faults they do not address. In fact, in order to fully benefit from our stars, the system should include mechanisms that deal with ofm faults, since it is impossible to increase the system reliability by improving only one of its parts. In this sense, an ofm proportion of 0% is equivalent to assuming that these mechanisms are 100% effective. Also notice that some of these mechanisms could be included in our stars. However, our hubs do not present them, since their design is application-independent and the knowledge necessary to address most of these faults, e.g. babbling idiot ones, is strongly related to the application. Anyway, as will be seen in Section 10.9.7, we carry out sensitivity analyses with respect to the ofm proportion of specific components in order to analyze the importance of including the corresponding additional mechanisms.

On the other hand, we consider that the failure modes that are included in our fault model are equiprobable. However, when possible, we assume higher proportions for failure modes that favor the case of the CAN bus, as explained next. First, we suppose that any fault happening in a component placed within the Hub Core causes the failure of all the system. This is a pessimistic assumption for CANcentrate since the Hub Core could suffer from a more benign fault, e.g. it could stop performing fault confinement or it could unfairly isolate a correct port.

Second, the cables and connectors that constitute an Attachment may fail in many different ways [Mee95]. For instance, cables may be shorted, broken, opened, fractured, arcing, worn, etc., whereas connectors may suffer from high resistance, intermittent and/or poor connections, open circuits, short circuits, mechanical failure of solder joints, etc. We consider that these failure modes lead an Attachment to exhibit, with the same probability, either a stuck-at-recessive, a stuck-at-dominant, a bit-flipping fault or just a physical disruption of the medium. The last one of these faults has different consequences depending on whether it happens in a star link or in a bus section. In CANcentrate, an uplink/downlink that suffers from a disruption is considered to actually generate bit-flipping errors, because such a failure can cause signal reflections at the open extremities of the medium. This is somehow a pessimistic assumption for CANcentrate, because a physical disruption may also lead the uplink/downlink to be stuck-at (as stated in Section 6.2), which can be treated by the hub in an easier way. Note that the specific consequence of a physical disruption is internally translated by the model. In the case of CANcentrate, it is transformed into a bit-flipping proportion (see Section 10.7.2).

As concerns the consequences of a physical disruption in a bus section, we also assume that it generates bit-flipping errors, since such a disruption will split the bus into two parts in such a way that each one of them will have an open extremity that will provoke signal reflections. Note that we do not model the case in which a physical disruption provokes a network partition instead of signal reflections. This is because we consider that CAN (and CANcentrate) are implemented in accordance with the ISO 11898-2 High-speed CAN standard, which does not compel to tolerate faults affecting any of two bus line terminations (see Section 10.4.1). Moreover, to assume that a physical disruption of a bus section always manifests as bit-flipping instead of considering that it could also manifest as stuck-at is not pessimistic for CAN. This is because, anyway, all nodes would be also prevented from communicating if the disruption leads the medium (and then the entire bus) to be stuck-at.

Similarly to the case of the Attachment, we assume that the failure of the resistor that implements a Termination leads the medium to be stuck-at-recessive, stuck-at-dominant, bit-flipping, or it can just imply that the medium losses the termination.

We consider all these modes to be equiprobable too. However, notice that the loss of a termination leaves one extremity of the bus and of the uplink / downlink opened, indirectly provoking again a bit-flipping fault. Therefore, our models intrinsically assume that a termination loss provokes a bit-flipping fault. For instance, see Section 10.7.2 for more details about how this assumption is modelled.

Third, we consider that an IO entity, i.e. a Node IO and a Hub IO, fails provoking, with the same probability, a stuck-at-recessive, a stuck-at-dominant and a bit-flipping fault. We believe that this assumption is reasonable, given the fact that an IO entity is directly connected to the medium and, thus, it can directly deliver errors to the channel without any restriction.

In contrast, it would not be reasonable to assume that stuck-at and bit-flipping faults are equiprobable when a Node Core or a Controller fail. The Node Core has not direct connection with the CAN transceiver, but uses the CAN controller as an interface. Thus, we believe that the Node Core cannot manipulate the CAN controller in a way that leads that controller to permanently deliver dominant or bit-flipping values. Such assumption favors the case of CAN when compared with CANcentrate. This is because if a Node Core never generates errors that can propagate causing a severe failure, the benefits of the error-containment capabilities of the hub become less relevant.

As concerns the Controller entity, to come to a conclusion about the proportions with which it exhibits different failure modes poses additional difficulties. This is because it includes both, components that can deliver errors to the channel without any restriction and components whose errors cannot always reach the channel. More specifically, the socket and the PCB area of the Controller entity can generate erroneous stuck-at-dominant or bit-flipping streams that reach the channel through the CAN transceiver. In contrast, a fault affecting the CAN controller will not always generate these errors. Notice that a CAN controller has a complex internal structure that includes some internal modules or parts whose failure cannot generate errors that reach the channel, but that can only lead the CAN controller to remain silent, i.e. to permanently deliver a stuck-at-recessive bit stream.

To overcome this problem we proposed a simple but reasonable approach. First, we rule out the influence that the failure modes of the socket and the PCB area have on the failure mode proportions of the Controller entity. We can do this simplification because the way in which a faulty CAN controller manifests has the major influence on the failure mode proportions of the Controller entity, given its much higher complexity and failure rate. More specifically, in order to rule out the failure modes of the socket and the PCB area we consider their failure rates as part of the failure rate of the CAN controller itself. Thus, from now on we will refer to the

CAN controller and the Controller entity without distinction.

For calculating the failure mode proportions of the CAN controller, we divide it into its main parts. We assume all these parts to have approximately the same complexity and, thus, the same influence on the failure rate of the CAN controller. For the sake of simplicity, we also suppose that all parts fail independently from each other.

For each part, we analyze whether its failure can only lead the CAN controller to be stuck-at-recessive or, in contrast, it can also lead it to deliver a stuck-at-dominant or a bit-flipping stream. In the later case, we consider that the part failure provokes these three failure modes with the same probability.

Once we have analyzed all parts, we calculate the CAN controller failure mode proportions. For that, we weight the failure mode proportions of every part taking into account its influence on the failure rate of the CAN controller. As said above, all parts have the same influence on that rate. Therefore, the proportion with which the CAN controller exhibits a specific failure mode is calculated dividing the number of parts that can lead to that failure by the total number of parts.

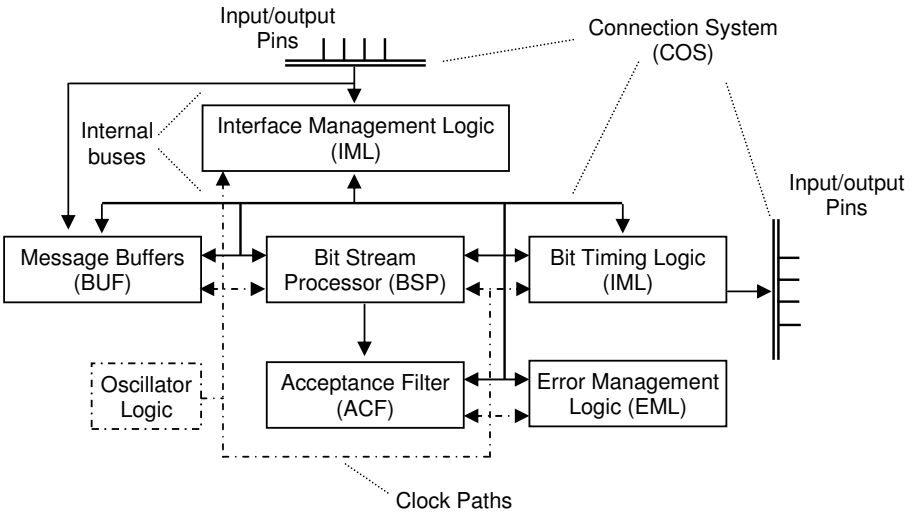

Figure 10.2: Basic internal architecture of the Philips SJA1000 CAN controller

More specifically, we carried out this analysis for the Philips SJA1000 CAN controller [SJA00], whose architectural basics are shown in Figure 10.2. We consider it to be constituted by the following parts: *Interface Management Logic* (IML); *Message Buffers* (BUF); *Acceptance Filter* (ACF); *Error Management Logic* (EML);

Bit Stream Processor (BSP); *Bit Timing Logic* (BTL); *Clock System* (CLK); and the *Connection System* (COS), which includes the internal bus and the input/ouput pin connections.

The IML is the interface between the microcontroller and the CAN controller: it interprets commands from the CPU and provides interrupts and status information. The BUF includes the transmission and the reception buffers. The ACF includes the logic needed to implement the acceptance reception filter functionality of a typical CAN controller: the identifier of each frame received from the network is compared with certain filter values to decide whether or not to accommodate it on the reception buffer. The EML takes into account the number of errors detected so far and decides when the CAN controller is in the *error-active*, *error-passive* or *bus-off* states (see Section 3.3.5). The BSP controls the data stream flow between the transmission/reception buffer and the channel. It also implements the arbitration, stuff rule, error-detection and error-signalling mechanisms. The BTL is the responsible for implementing the CAN bit synchronization mechanism. The CLK includes the oscillator logic and the clock paths that propagate the clock signals throughout the device. Finally, the COS includes the internal buses that the different submodules use to communicate with each other, as well as the pin connections of the device.

We consider that a fault in the IML, BUF, ACF, and EML can only lead the CAN controller to deliver a stuck-at-recessive stream[1]. Similarly to what we said before for the case of the Node Core, we believe that it is extremely unlikely that the IML sends commands, to the rest of the device parts, that lead the CAN controller to constantly send a stuck-at-dominant or a bit-flipping stream. Similarly, the BUF cannot send incorrect commands to other modules. At most, it can fail by corrupting any frame stored in the transmission buffer. But this cannot even provoke the transmission of stuck-at-dominant or bit-flipping bits. The content of this buffer is not directly delivered to the channel, but the BSP uses this content as a piece of data it encapsulates within a frame. Regarding the ACF, it has not influence on the bit values the CAN controller outputs. Finally, if the EML fails, the CAN controller can incorrectly be in the *error-active*, *error-passive* or in the *bus-off* states. If the CAN controller incorrectly enters in any of the two latest states, we consider that the controller is stuck-at-recessive. This is because, as we explained in Section 8.5, if the CAN controller is *error-passive*, the microcontroller should disconnect it to prevent inconsistencies to occur. Likewise, a *bus-off* CAN controller disconnects itself. In contrast, a CAN controller that does remain *error-active*, when it should have disconnected itself to contain a fault that affects

[1]Note that a CAN controller that is disconnected (by the microcontroller or by itself) is considered as stuck-at-recessive, since it does not deliver dominant bits through its transmission pin

its output, cannot be exclusively considered as stuck-at-recessive. It can also be stuck-at-dominant or even bit-flipping depending on the values of the incorrect bits it delivers. However, notice that such a situation can only happen if, at least, two faults affect the CAN controller: one that affects its EML and another fault that compels it to output stuck-at-dominant or bit-flipping bits. Since the event of two faults affecting a CAN controller is very unlikely, we rule out this possibility and, thus, we definitively consider that a fault affecting the EML can only manifest as a stuck-at-recessive.

But not all faults affecting a CAN controller manifest as stuck-at-recessive. We consider that a fault affecting either the BSP, BTL, CLK or COS can lead the CAN controller to be stuck-at-recessive, stuck-at-dominant or bit-flipping with the same probability. The BSP and BTL can stop operating when the CAN controller is issuing a dominant bit. If this occurs, the controller will permanently output a dominant value. Similarly, the BPS and BTL can fail by performing random transitions, which will lead the controller to send random bit values as well. Regarding the CLK subsystem, it can also stop or even perform random transitions, thereby leading the controller to send a stuck-at-dominant or a bit-flipping stream. Finally, regarding the COS, a fault in an internal bus can lead the BSP to transmit erroneous data or the BTL to behave incorrectly; a faulty transmission pin can output any value; and a faulty reception pin will lead the CAN controller to misinterpret the logical value of the channel, thereby compelling the controller to transmit syntactically incorrect data.

As explained before, the proportion with which a CAN controller exhibits a given failure mode is obtained dividing the number of parts that can lead to that failure by the total number of parts. Thus, based on the above discussion, the proportion with which it is stuck-at-recessive is:

$$\text{Stuck-at-recessive proportion} = \frac{4}{8} + \frac{4}{8} \cdot \frac{1}{3} = \frac{2}{3} \simeq 66.6\%$$

where the 1/3 is the proportion with which the BSP, BTL, CLK and COS fail provoking a stuck-at-recessive. This fraction is also the proportion with which these parts provoke a stuck-at-dominant and a bit-flipping, thus:

$$\text{Stuck-at-dominant proportion} = \text{Bit-flipping proportion} = \frac{4}{8} \cdot \frac{1}{3} = \frac{1}{6} \simeq 16.7\%$$

These proportions are reflected in the parameters *ctrlStrProp*, *ctrlStdProp* and *ctrlFlipProp* of Table 10.4.

10.4.5 Coverage assumptions

Until this point we have explained our models' assumptions that are related to implementation issues, constituent components and entities, statistical properties of faults, and the proportion with which the different entities of the system exhibit different failure modes. However, as said in Section 2.3, there is a crucial aspect that must be considered when evaluating a dependable system: the coverages of its fault-tolerance mechanisms. As we already explained in Section 2.4, these coverages are of capital importance, since dependability is extremely sensitive to them. Current section is thus devoted to deciding what coverages should be taken into account as well as what are the most reasonable values for them. Anyway, given the expected big impact of coverages, the present chapter analyzes the sensitivity of the system reliability with respect to the major part of the coverages herein proposed.

First of all, notice that we distinguish between the fault-tolerance mechanisms of the system itself and the fault-tolerance mechanisms of the communication subsystem (CAN or CANcentrate) it relies on. The first set of fault-tolerance mechanisms refer to the ability of the system to accept or tolerate the failure or the disconnection of a given number of nodes. Since we do not make any proposal in order to improve those system's mechanisms, we reflect their effectiveness by means of a single coverage we call *sysFauTolCov* (see Table 10.4). Broadly speaking, *sysFauTolCov* can be considered as a numerical value that reflects the probability with which the system accepts or tolerates the failure or the disconnection of a node, provided that the system is able to accept or tolerate such a situation.

In order to better understand the meaning of this coverage, it is necessary to recall that we mainly differentiate between two types of systems: systems that do not accept or tolerate the failure or disconnection of any node, and systems that are able to accept or tolerate the failure or the disconnection of up to k of N nodes. We call them *non-fault-tolerant/accepting* (NFT/A) and *fault-tolerant/accepting* (FT/A) systems respectively (see Section 10.2). Obviously, *sysFauTolCov* is not connected with an NFT/A system, since such a kind of system has no mechanism to tolerate the failure or disconnection of any node. Therefore, as will be explained later in Sections 10.7.5 and 10.8.4, we built our models in such a way that the NFT/AR, i.e. the metric we use to measure the reliability of an NFT/A system, does not depend on this parameter.

Conversely, *sysFauTolCov* is strongly related to an FT/A system. Imagine an FT/A system that is able to accept or tolerate the failure or disconnection of up to k of N nodes. Then, *sysFauTolCov* represents the probability with which this FT/A system accepts or tolerates the failure or disconnection of a new node, provided

that the number of nodes that have failed or that have become disconnected so far (including the new one) does not exceed the value of k.

As can be deduced from the above definition, *sysFauTolCov* does influence the value of the metric we use to measure the reliability of FT/A systems, i.e. the FT/AR$_k$. In order to decide what should be the default value of *sysFauTolCov*, it is necessary to further differentiate between FT/A systems that merely accept situations in which up to k nodes are faulty or disconnected, and FT/A systems that tolerate these situations. Examples of these two types of FT/A systems were already specified in Section 10.2. Notice that the value of *sysFauTolCov* can be considered of the 100% for the former type of FT/A systems; since they intrinsically accept situations in which not all nodes are operative and can communicate. Conversely, *sysFauTolCov* cannot be perfect for an FT/A system of the second type, since to build a fault-tolerance mechanism that yields such a coverage is impossible in practice.

In order to choose the most general option for measuring the FT/AR$_k$, we decided that the default value of *sysFauTolCov* is 100%. On the one hand, as just said, it is the appropriate value for the first type of FT/A systems, i.e. for FT/A systems that intrinsically accept the failure or disconnection of a given number of nodes. On the other hand, the coverage of the fault-tolerance mechanisms of an important number of the second type of FT/A systems is expected to be very close to this default value. This is the case of highly-reliable systems that include redundancy to tolerate faults (and for which it is very interesting to assess the benefits a star topology can yield). For instance, the coverage of the fault-tolerance mechanisms of any redundant subsystem that forms part of the *Self-Repairing Flight Control System* (SRFCS) of a military aircraft typically ranges from 99%, 99.99% and 99.9992% up to virtually 100%, depending on the number of replicas the redundant subsystem is provided with [Wu02] [Wu04]. Moreover, notice again that we do not propose any fault-tolerance mechanism at the level of the system that relies on CAN or CANcentrate. Thus, to find specific values for the fault-tolerance coverages of different FT/A systems is beyond the scope of the analysis herein presented. Anyway, as will be seen later in this chapter, we perform a sensitivity analysis of the FT/AR$_k$ with respect to the value of *sysFauTolCov*, in order to assess what is the minimum coverage for which the use of a star yields benefits in terms of this metric.

Regarding the coverage of the fault-tolerance mechanisms of the communication subsystem, notice that as indicated in Section 2.3, each author defines her own specific coverages, depending on the subsystem itself and on the level of abstraction from which it is addressed. In our case, we are not interested in modelling the details of the error-processing and fault-treatment mechanisms of a CAN bus and a

CANcentrate star. Thus, we do not consider a coverage for each one of the phases carried out to process errors and treat faults in those networks. This would be necessary in case we were interested in analyzing the efficacy of their fault-tolerance mechanisms with respect to decisions concerning the design of those mechanisms themselves. In contrast, we are interested in how the effectiveness of those mechanisms affect the reliability of a system that relies on them. More specifically, since CANcentrate is devoted to improving reliability by means of appropriate error containment, in the context of this chapter, we focus on the so called *error-containment coverage*. We define this coverage as the probability of detecting and isolating a fault included in our fault model, provided that this fault occurs. Do not confuse this coverage with the failure mode assumption coverage, which we considered when talking about the failure mode assumptions.

In the CAN bus, the only available error-containment mechanisms are those implemented at each node; whereas in CANcentrate the hub provides additional mechanisms to contain errors at the hub ports. Next, we explain what are those mechanisms, as well as what are the default values we consider as reasonable for them.

The error-containment capabilities of a CAN node rely on the fault-treatment mechanisms of its CAN controller. As explained in Chapter 3, the CAN controller can detect errors and stop transmitting when diagnosing itself as the responsible for a permanent fault, thereby forcing local faults to manifest as the transmission of a stuck-at-recessive stream. However, we cannot assume that a CAN controller provides a 100% error-containment coverage because it is a practical impossibility. On the one hand, it is necessary to analyze what faults the CAN controller can isolate to prevent the propagation of errors. On the other hand, it is necessary to decide what is the reasonable probability with which the controller successfully diagnoses the faults it is able to isolate.

In a CAN bus, a CAN controller can potentially isolate faults affecting its Node IO entity (its transceiver, basically), as well as some of its internal parts. In particular, a CAN controller can isolate a fault affecting its Node IO entity only if the fault does not compel the transceiver to deliver incorrect bit values to the medium. Otherwise, the CAN controller can do nothing to prevent the propagation of errors. In order to take into account this fact, we define an error-containment coverage called *nodeIOInFauProp*, which specifies the percentage of Node IO's faults that are delivered to the controller or to the medium. More specifically, we suppose that *nodeIOInFauProp%* of Node IO's faults deliver errors to the CAN controller, whereas $1 - $ *nodeIOInFauProp* of these faults transmit errors to the medium (see Figure 10.3). Since we do not know what is the actual value of *nodeIOInFauProp*, we assume a neutral default value of 50% for it (see Table 10.4).

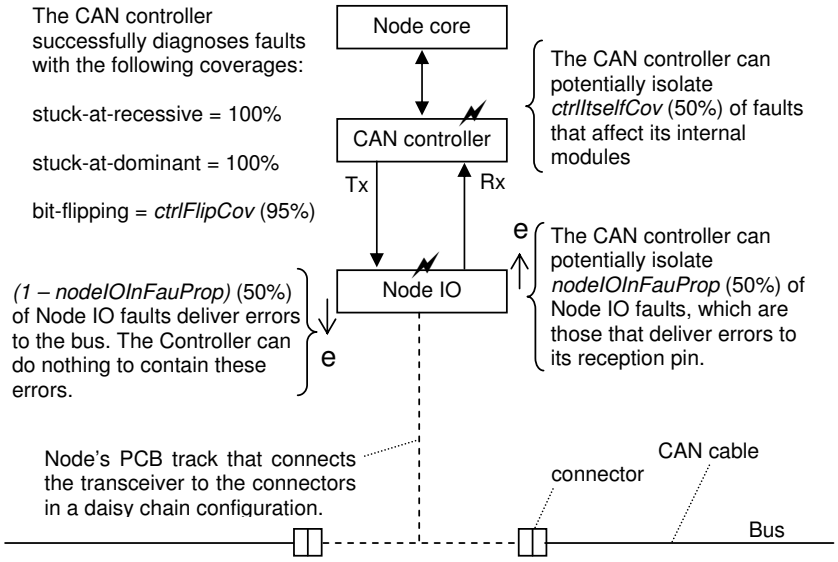

Figure 10.3: Error-containment capabilities of a CAN controller in a CAN bus

As said above, the CAN controller can also potentially isolate faults that affect its internal parts. This is useful when the fault leads it to deliver a stuck-at-dominant or a bit-flipping stream. Otherwise, if the fault compels the controller to deliver a stuck-at-recessive stream, there is no need to isolate it. Anyway, notice that what the CAN controller does in order to prevent the propagation of its own errors is to stop operating when it diagnoses itself as faulty. This mechanism limits the capacity of the CAN controller for isolating an internal fault. A clear example could be a case in which the only way to isolate an internal fault consists in stoping the faulty internal part, but this part does not stop operating when it is required to do so. Moreover, the fault isolation mechanism of the CAN controller can be ineffective even if an internal fault can be theoretically isolated by stopping the operation of other non-faulty internal parts. For instance, if a fault affects the clock signal or the internal buses of the controller, the non-faulty internal parts can become desynchronized or they cannot intercommunicate properly and, hence, they may ignore or misinterpret the commands sent to them for stopping. Finally, notice that when a fault leads the clock signal to stop, the CAN controller cannot perform any action. If such a fault leaves the controller's transmission pin stuck-at-dominant, the controller becomes irretrievably stuck-at-dominant. This last limitation was also pointed out in [NSSLW05]. In order to model the just mentioned drawbacks of the fault isolation mechanism of the CAN controller, we suppose that a controller that

diagnoses itself as faulty can successfully isolate an internal fault with a coverage of *ctrlItselfIsoCov*. Again, since we actually do not know what is the real value of this probability, we assume a neutral default value of 50% for it (see Table 10.4).

In CANcentrate, the fault-isolation capabilities of the CAN controller are more or less the same as in the CAN bus. Firstly, the CAN controller is able to potentially isolate faults affecting its internal parts, but showing the same limitations explained just before.

Secondly, as depicted in Figure 10.4, the CAN controller can potentially isolate faults affecting any of the components that connect its node to the hub port corresponding to the downlink. More specifically, the CAN controller can isolate faults affecting the Node IO or the Hub IO entities connected to its node's downlink; the Attachment entity that represents the downlink itself; and the Terminations of the downlink. Notice that all errors generated by any of these entities are observed only by the CAN controller and, thus, the hub becomes aware of these error only if the CAN controller transmits an erroneous contribution as a consequence of them. Therefore, the CAN controller can prevent the propagation of these errors by simply stop operating. This fact leaded us to assume that when the CAN controller diagnoses a fault in any of the referred entities, it is able to isolate it with no restriction.

Finally, Figure 10.4 also shows what are our assumptions concerning the ability of the CAN controller to isolate faults affecting the components that connect the node to the hub port corresponding to the uplink. Specifically, we consider that the CAN controller can do nothing to isolate those faults, since it has no mechanism to prevent the propagation of errors they generate. This is slightly pessimistic for CANcentrate, because we could suppose that, as in the case of the CAN bus, a fault in the Node IO connected to the uplink does not deliver errors to the medium with a probability of *nodeIOInFauProp*.

Up to this point, we have explained the capacity of the CAN controller to isolate faults. However, a stated before, in order to characterize the error-containment capabilities of the CAN controller, it is also necessary to assess what is the probability with which it successfully diagnoses the faults it can isolate. The effectiveness of fault-diagnosis mechanisms of the CAN controller depend on the type of fault. On the one hand, we suppose that the CAN controller successfully diagnoses stuck-at faults with a probability of 100%. This is because such faults will lead the CAN controller to detect an error in almost every bit transmission and, thus, it can be expected that the CAN controller quickly reaches the *error-passive* or the *bus-off* state. For instance, consider a fault in a transceiver that leads the corresponding controller to receive a permanent stuck-at-recessive stream. The CAN controller

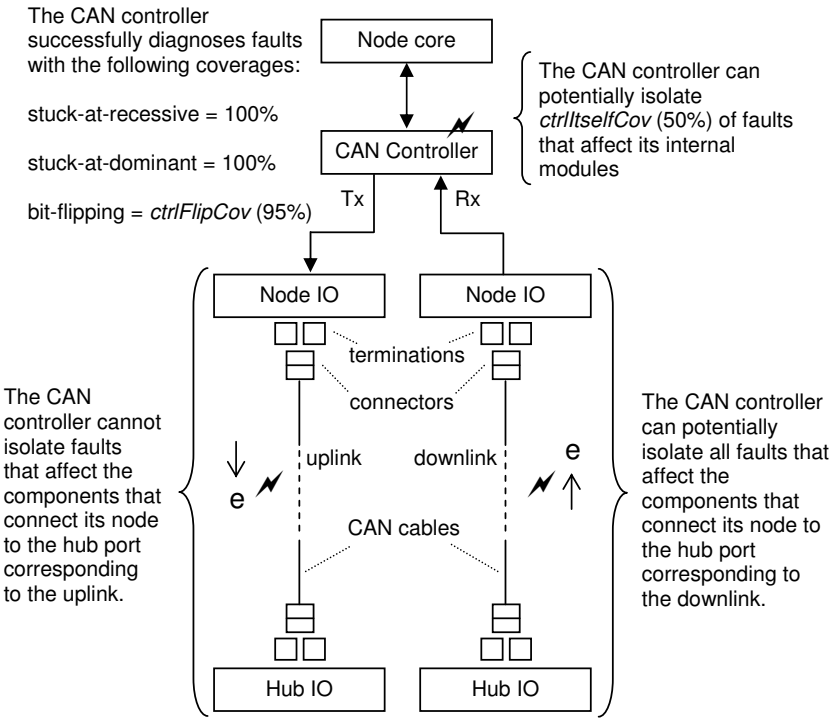

Figure 10.4: Error-containment capabilities of a CAN controller in CANcentrate

will detect a bit error as soon as it tries to send a dominant bit value. Then, the CAN controller will start signalling an active error flag, but it will detect an additional bit error in every bit transmission. As a consequence, its error counter will quickly increase compelling the controller to reach the *error-passive* or the *bus-off* state. On the other hand, we do not assume a perfect diagnosis coverage for bit-flipping faults, since such a kind of fault can lead to a huge amount of error scenarios. Instead, we suppose that the CAN controller diagnoses bit-flipping faults with a coverage of *ctrlFlipCov*. In particular, we consider a default value of 95% for this coverage (see Table 10.4). As indicated above, this is a coverage default value commonly assumed in the literature related to the dependability evaluation. Notice that Figures 10.3 and 10.4 reflect our assumptions concerning the effectiveness with which the CAN controller diagnoses different types of faults. For instance, these figures show that the coverage with which the CAN controller successfully diagnoses bit-flipping faults is of *ctrlFlipCov%*.

So far, we have analyzed the effectiveness of the error-containment mechanisms

implemented at the CAN nodes. Next, we do the same for the error-containment mechanisms the hub is provided with. These mechanisms are thoroughly described in Chapters 6, 7 and 8.

Given its privileged location within the network, the hub of CANcentrate is able to detect errors generated by any faulty component located outside of the hub core. Moreover, since the mechanism it uses to isolate faults, i.e. the OR gates, is independent from any of those components, the hub can always isolate any fault it successfully diagnoses.

However, the hub cannot diagnose faults with a perfect coverage. Again, the effectiveness of its error-detection and fault-diagnosis capabilities depends on the way in which the fault manifests and, thus, on the difficulty off its detection. Firstly, we consider that the hub can always diagnose suck-at-recessive faults since, even in the case that the hub cannot detect them, they have no negative impact in the communication among the other nodes.

Secondly, notice that the mechanism the hub uses to detect and diagnose stuck-at-dominant faults is trivial: a counter that monitors the number of consecutive dominant bits and a specific threshold. Hence, we consider that stuck-at-dominant faults are diagnosed with a perfect coverage. This is a realistic assumption because if such a failure occurs and the Hub Core is not faulty, then the related counter will eventually reach its threshold.

Finally, as said above, a bit-flipping fault can lead to a huge amount of scenarios involving errors. Therefore, since it is not reasonable to assume that the hub provides a perfect fault-diagnosis coverage for it, we consider that the hub can diagnose bit-flipping faults with a limited effectiveness of *flipLnkCov*. Notice that Table 10.6 describes the meaning of this parameter as being the coverage with which the hub isolates a bit-flipping fault at the uplink hub ports. This is because what the hub actually does to isolate a given fault in a branch is to disable the corresponding faulty contribution it receives through the uplink. In particular, Table 10.6 indicates that the default value we assumed for *flipLnkCov* is 95%. We believe that this is not an optimistic assumption for CANcentrate, given that the hub always correctly detected and isolated all bit-flipping faults we have injected in our CANcentrate prototype so far. Moreover, this coverage can even be considered pessimistic for CANcentrate. This is because we supposed that a CAN controller is able to diagnose bit-flipping faults with the same effectiveness as the hub, despite the fact that the error-detection and fault-diagnosis mechanisms of the hub are more efficient (see Section 8.3).

10.5 Modelling formalism

As already indicated in 10.1, we built our models using the *Stochastic Activity Network* (SAN) formalism, which is an extension to stochastic Petri Nets [CFJ+91] [TMGT93] [SoT04].

Notice that we could use other formalisms to built our models, instead of SANs. As will be seen later on, we model the system as being in different states that represent where faults have occurred and how they have been isolated and / or tolerated. This implies that each one of our models basically changes its state when a fault occurs or when an error-containment or a fault-tolerance action takes place. On the one hand, the Time To Failure of a given component is exponentially distributed, which implies that the time that elapses until a fault occurs in the system is exponentially distributed too. On the other hand, error-containment and fault-tolerance actions are modelled to be performed instantaneously (they do not consume time in comparison with the occurrence of faults), so that the system can also immediately change from one state to another one. The fact that state transitions are exponentially distributed or are instantaneous allows to model the system as a *Continuous Time Markov Chain* (CTMC) [STP96]. Nevertheless, to model a system by means of a CTMC poses some difficulties. On the one hand, the number of states of a CTMC tends to explode when the complexity of the system to be modelled increases. On the other hand, a CTMC provides scarce modelling primitives, so that a CTMC model is normally not intuitive.

In contrast, Petri Nets offer more primitives than a CTMC and allow to build up a compact model that is more easy to understand. The different Petri Net formalisms that are available differ in the modelling facilities they provide. In this sense, SANs is one of the most powerful Petri Nets' formalisms [AM02]. A SAN includes *tokens*, *places*, *activities*, *input gates* and *output gates*. The number of tokens located in each place, i.e. the marking of the places, determines the state of the modelled system. An activity is connected to one or more source places and has one or more *cases*, each one connected to one or more destination places. Each activity can be enabled or disabled. When enabled, it *fires* immediately or in accordance with a given statistical distribution to change the marking of places, thereby modelling the system transitions through different states, e.g. the failure of a given component. Activities that fire immediately are referred to as *instantaneous activities*, whereas the others are called *timed activities*. An input gate defines a condition for an activity to fire, which depends on the marking of its source places. This gate also specifies how to change the marking of the source places when the activity fires. Besides, an activity selects one of its cases to change the marking of specific destination places. An output gate is connected to a given case and specifies the set

of marking changes to be performed depending on some conditions.

The SANs formalism also provides two primitives to build a model as a hierarchical composition of submodels: the *Join* primitive, which allows interconnecting different submodels by sharing places, and the *Rep* primitive, which can be used to replicate a given submodel in order to model different instances of the same submodel.

All the above mechanisms are used to specify the structure of the stochastic process that models a given system behavior, i.e. they are used to build up the *structure state model* of the system [STP96]. But in addition to these mechanisms, the SANs formalism offers the possibility of specifying a *reward model*. A given reward model is associated to specific states of the structure state model and it is aimed at calculating a specific metric or attribute such as, for example, reliability, availability, throughput, etc. Particularly, the SANs formalism offers the possibility of specifying a reward model by means of the so-called *reward variables*. A reward variable is related to a specific aspect of the stochastic process. There are basically two categories of reward variables: *impulse reward variables*, which are associated to the number of times activities fire; and *rate reward variables*, which are associated to specific markings.

Finally, it is important to highlight the importance of being able to assume exponentially distributed Time To Failures. As already explained in Section 10.4.3, this is a realistic assumption that further simplifies the mathematical treatment of the models. Specifically, such an assumption implies that the stochastic process underlying the corresponding SANs model can be characterized by means of an CTMC that can be analytically solved[2]. There are different software tools that offer the possibility of modelling SANs, that automatically translate them into the corresponding CTMC and that provide facilities to solved them [AM02]. In particular, we used the Moëbius software [SoT04] to build and analytically solve all our SANs models.

10.6 Modelling rationale

The current section aims at providing a general view of the strategy we followed to model the reliability of a system that relies on the CAN bus and of an equivalent system that relies on CANcentrate. Specifically, this section focuses on the level of abstraction and the basic structure of these models. However, it is impor-

[2]See [STP96] for a basic explanation of this procedure and [SoT04] for a more detailed specification of the requirements needed to transform a SANs into a CTMC

tant to note again that we will compare the reliability of CAN and CANcentrate with ReCANcentrate in Chapter 12. This means that, in fact, we modelled CAN, CANcentrate and ReCANcentrate following the same strategy. Thus, some of the modelling decisions explained here were made to be also appropriate for the case of ReCANcentrate.

As concerns the level of abstraction, it is necessary to keep the complexity of the model within reasonable limits, while guaranteing an accurate description of the real system. The main abstraction we carried out consists in not modelling the failure of single components but of groups of them called entities, as explained in Section 10.4.2. These entities are: Node Core, Controller, Node IO, Hub IO, Attachment, Termination and Hub Core. This abstraction allowed us to calculate basic entities' dependability properties, such as failure rates, failure modes and different error-containment coverages, in a reasonably simple way. See Tables 10.4, 10.5, 10.6 and Section 10.4 for more details.

Regarding the basic structure of the models, we explored different approaches in order to find one that can be analytically solved in a reasonable amount of time. Specifically, we studied three different possibilities. Among these three approaches, only the third one provides a reasonable computation time for the models of CAN, CANcentrate and ReCANcentrate and, thus, it is the strategy we finally adopted. Anyway, in order to make it easier for the reader to understand this third approach, next we summarize all three.

10.6.1 A dedicated SAN submodel per *entity*

At a first attempt, we modelled the CAN bus and CANcentrate following the strategy presented in [BPA06], which is similar to those proposed in [MT95] and [PdS04]. It consists in modelling each entity of the system by means of a dedicated SAN submodel, we call *entity submodel*.

The basic structure of an entity submodel is depicted in Figure 10.5. It basically has a place, called *okEntity*, that initially has a token that indicates that the entity is not faulty; an activity called *entityFailure*, that models the failure of the entity, and which has a case for each failure mode; and a set of places that represent how the fault manifests (fm_1, fm_2, etc.).

The activity *entityFailure* models the Time To Failure of the entity as an exponential distribution whose rate is the failure rate of that entity. When the entity fails, the token is erased from *okEntity* and one of the activity's cases is selected. The probability with which each case is selected is the proportion with which the entity shows the corresponding failure mode.

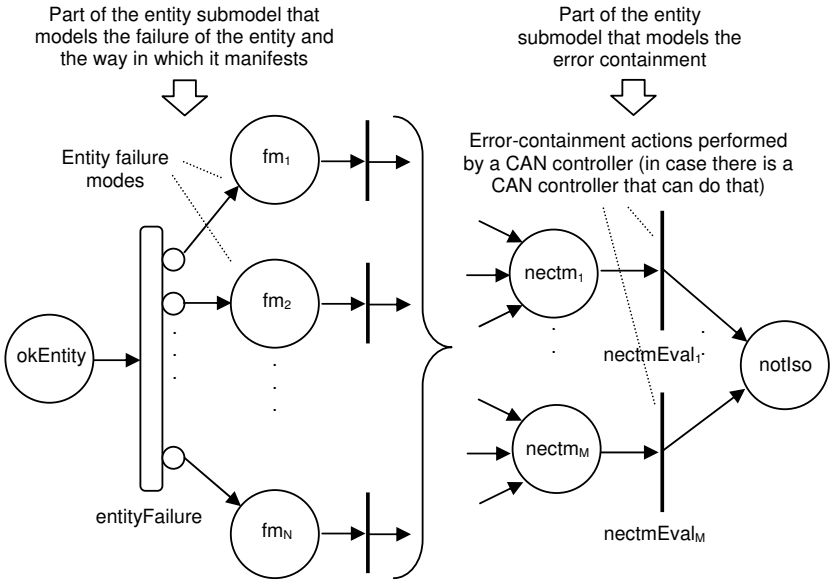

Figure 10.5: Basic structure of an entity submodel

Once a case is selected, a token is written at the place that represents the failure mode corresponding to that case (fm_i) and, then, the entity submodel initiates a sequential process that models whether or not the fault is isolated. We refer to this process as the *coverage process*. As explained in Section 10.4.5, we must consider two types of error-containment mechanisms: those implemented at the CAN controllers and the mechanisms included in the hub of CANcentrate.

In case the fault can be isolated by means of a given CAN controller's mechanisms, the corresponding entity submodel models the part of the coverage process related to these mechanisms. This is shown in the Figure 10.5, where a token in a place that represents a failure mode is instantaneously transferred to a place, $nectm_i$, that represents a specific CAN controller's error-containment mechanism. When a token reaches any of these places, a dedicated *nectmEval* instantaneous activity decides if the corresponding mechanism contains the errors. On the one hand, if the errors are contained, then the token is basically deleted from the place $nectm_i$. In this case, the process that evaluates how the fault is isolated finishes, since it is not necessary to further evaluate how the errors propagate, or whether or not they can be contained by additional mechanisms. On the other hand, if the activity $nectmEval_i$ decides that the errors are not contained, it transfers the token from $nectm_i$ to a place that indicates this situation: *notIso*.

When the fault is not isolated, either because there is no CAN controller's mechanism that can potentially do it, or because the CAN controller does not successfully achieve it, it is necessary to differentiate between what happens in a CAN bus and in a CANcentrate star. If the first case, the communication subsystem (and the whole system) can be considered as faulty, since a CAN bus does not include any error-containment mechanism apart from those implemented at the CAN controllers. In contrast, in the second case (in the case of CANcentrate), it is necessary to further evaluate if the hub can contain the errors. For this purpose, the CANcentrate model includes an additional SAN submodel that models the error-containment capabilities of the hub. We refer to this submodel as the *coverage submodel*. All the entity submodels of CANcentrate share the place *notIso* with the coverage submodel, so that this submodel takes over when any of the entity submodels sets a token in it.

Besides these submodels, both the CAN bus model and the CANcentrate model have what we call an *evaluator submodel*. It takes into account the faults that have happened, as well as whether or not each one of them has been successfully isolated, to elucidate what is the number of nodes that can still communicate among them. Then, it decides if the system is faulty. This decision will depend on what is the minimum number of nodes that must communicate among them to consider that the system is not faulty. This number is configurable by means of the parameter *kSevere* (see Table 10.4), which represents the value of k of the concept of k-severe failure. In this way, the models can be set up to measure the NFT/AR and different degrees of FT/AR$_k$, i.e. the FT/AR for different values of k. For instance, if one is interested in measuring the NFT/AR, the minimum number of nodes that must communicate is the total number of nodes and, thus, *kSevere* must be specified as 0.

When the *evaluator submodel* considers that the system is faulty, it writes a token at a place called *generalized failure*. This place is shared among all submodels and allows the *evaluator submodel* to indicate to the rest of submodels that the system is faulty. When this happens, all submodels stop evolving and the whole SANs model do not change of state since then. Such a behavior reduces the size of the state space of the underlying stochastic process.

Finally, in order to quantify the NFT/AR and the FT/AR$_k$, we associate a rate reward variable with the marking of the place *generalized failure*. We call this variable *probNonSev* and its expressions is:

$$probNonSev = 1.0 - generalizedFailure \rightarrow Mark()$$

where $generalizedFailure \rightarrow Mark()$ represents the marking of the place *gener-*

alizedFailure, i.e. the number of tokens of this place. Since the marking of *generalizedFailure* can be only 0 or 1, the value of the variable is 1 as long as the system is not faulty. In this way, once the model is analytically solved, the value of this variable at a particular point in time indicates the probability with which the system has not failed until that instant of time. Particularly, this variable quantifies the NFT/AR and the FT/AR$_k$ for different values of k, depending on the value of the parameter *kSevere*, as explained above.

Notice that the whole system (relying on CAN or on CANcentrate) is modelled as a composition of these SANs submodels, using the *Rep* and *Join* primitives. Specifically, we used the *Rep* primitive to obtain several instances of each one of the SAN submodels that represents a given entity, e.g. to obtain several Node Cores. This was very useful because we could specify every SAN submodel once and, then, replicate each submodel to represent all entities of the system. We used the Join primitive just to let submodels to share the appropriate places.

This strategy was suitable for modelling systems based on CAN and CANcentrate that include a small number of nodes. Unfortunately, when more nodes were considered, this strategy led the state space transformation of the SANs into Markov Chains to become extremely inefficient in terms of computation time. We found out that to model the failure of each entity separately becomes impracticable when the number of entities increases. Notice that this happens even though the *Rep* primitive is used to obtain several instances of the different SAN submodels. Similar problems have been reported when using the Petri Net formalism to model complex and large systems [CFJ+91] [AM02].

10.6.2 A dedicated SAN submodel per *entity* type

In order to overcome the extreme performance inefficiency of the solution explained just above, we modeled CAN and CANcentrate following a second approach that does not model each entity failure by means of a dedicated SAN submodel. Instead, we used a SAN submodel to model faults occurring at any entity of a given type. We call these SANs *entity group* submodels. For example, there is one entity group submodel that represents all Node Cores and that models the failure of any of them.

The entity group submodel has almost the same structure as the entity submodel explained before (see Figure 10.5). It has one place whose number of tokens represents the number of entities of a given type that have not failed so far; an activity that models the failure of any of these entities and that has a set of cases representing different failure modes; and a set of places that represent the failure mode with

which a fault manifests.

When an entity fails, the corresponding entity group submodel decides in which way the fault manifests and, then, initiates the coverage process, which models whether or not the fault is isolated. As in the previous approach, this process is carried out by the entity group submodel itself and, in the case of CANcentrate, also by a *coverage submodel* that represents the error-containment capabilities of the hub.

Besides the entity group and the coverage submodels, this second strategy still uses an *evaluator submodel* to elucidate when the system is faulty, as well as the *probNonSev* reward variable to quantify the NFT/AR and the FT/AR$_k$.

It is worth noting that although this alternative approach reduces the number of SANs submodels, it still models the failure of every single entity. This is implicitly done at every entity submodel by means of the activity that models the failure of any of the entities of a given type. Specifically, this activity models both the Time To Failure distribution and the failure mode proportions of the entities that the entity submodel represents. On the one hand, the activity represents a Time To Failure distribution that takes into account the number of surviving entities of the corresponding type, i.e. the number of entities of that type that have not failed so far. Specifically, since the Time To Failure distribution of each entity is exponentially distributed and each entity can independently fail, the time elapsed until any surviving entity fails is also exponentially distributed, with a failure rate:

$$\lambda_{entityGroup} = \lambda_{entity} \cdot \textit{entitiesOk} \rightarrow \textit{Mark()}$$

where λ_{entiy} is the failure rate of the entity, and *entitiesOk→Mark()* is the marking of the place that represents the number of surviving entities at a given instant of time.

On the other hand, each one of the cases of these activities represents the proportion with which a given failure mode manifests, when any surviving entity of the corresponding type fails. Notice that all the entities of the same type will exhibit the same failure modes with the same proportions. Therefore, the proportion with which each case is selected is just the proportion with which an entity of that type exhibits the corresponding failure mode.

This new approach yields very good results for the case of CAN and CANcentrate in terms of computation time. Moreover, the time needed to transform the SANs into Markov Chains when using this strategy is independent of the number of nodes. Nevertheless, we found this approach to be very inefficient for the case of ReCANcentrate. In fact, conversely to the case of CAN and CANcentrate, the per-

formance of this strategy is unacceptably degraded in the case of ReCANcentrate as more nodes are considered.

10.6.3 A dedicated SAN submodel per *region* type

Due to the efficiency problems of the first and second modelling strategies explained above, we decided to further reduce the number of SANs submodels needed to model faults occurring at individual entities. For this purpose, we define the concept of *region* as an ensemble of entities that constitute an error-containment region. In other words, a *region* can be understood as the ensemble of entities that are isolated to prevent the propagation of errors generated by any entity of this ensemble. To better understand this issue, notice that normally, when an entity fails, it is not possible to contain the errors it generates by isolating just that entity, but by isolating a whole region that includes other entities. For example, a fault that affects a downlink in a CANcentrate network is isolated by disabling the appropriate hub port; which means that not only the Attachment entity that represents the downlink is isolated, but also the rest of the entities of its branch: the Node Core, the Node IOs, etc.

We decided what *regions* constitute the system taking into account what are the ones that are appropriate for modelling not only the reliability of a system relying on CAN or CANcentrate, but also of a system relying on ReCANcentrate. In this sense, notice that, as will be explained later in Chapter 11, ReCANcentrate tolerates faults at hubs, links and even at CAN controllers. This means that the error-containment regions of ReCANcentrate are smaller than the ones of the CAN bus and of CANcentrate. For example, conversely to what happens in CANcentrate, to disable a hub port in ReCANcentrate does not imply to isolate the corresponding Node Core. After analyzing the CAN bus, CANcentrate and ReCANcentrate, we decided to divide up the system into the following regions.

- *Node Kernel*. It only includes the entity Node Core.
- *Node Connection*. It comprises all the entities a node needs to be connected to the bus line or to a given hub. In the case of the bus it includes one Controller and one Node IO. In CANcentrate (and in ReCANcentrate), it comprises one Controller, two Node IOs, two *Attachments* (one that represents the uplink and another one that represents the downlink), two Hub IOs and four Terminations.
- *Internal Bus Section*. It contains the Attachment that connects two adjacent nodes in a CAN bus, when none of these nodes is located at a bus extremity.

- *Edge Bus Section*. It contains the Attachment and the Termination that constitute the bus section that connects a node located at a bus extremity with the next node in the bus line.
- *Hub Kernel*. It only includes the entity Hub Core.

We use a single SAN submodel to represent all regions of a given type. We refer to each one of these SANs as a *regions submodel*. In this sense, there is one regions submodel that represents all Node Kernels, other regions submodel that represents all Node Connections and so on.

The structure of a regions submodel is similar to the one of the entity and entity group submodels of previous modelling strategies. As shown in Figure 10.6, this means that a regions submodel still has a place, *okRegions*, whose marking represents the number of regions that do not include any faulty entity; an activity, *regionFailure*, that models the occurrence of an entity failure in any of those regions and that has different cases each of which represents a given failure mode; and places that represent how the fault manifests: fm_1, fm_2, etc. When an entity fails, the corresponding regions submodel erases one token from *okRegions* to model that the region where the entity is placed is faulty. Notice that by erasing this token, the model rules out all the entities that are within the same region as the entity that has failed, so that *regionFailure* will not model the failure of these entities from then on. This can be done since it is not necessary to model these potential additional faults. On the one hand, if the region is successfully isolated, all the entities located within it will be isolated too and, thus, there is no need to model that they can fail afterwards. On the other hand, if the fault cannot be isolated, then the whole system becomes faulty and, thus, it is not necessary to continue modelling entity failures either.

Once it erases the token from the place *okRegions*, the regions submodel selects one of the cases of the activity *regionFailure*, in order to decide the way in which the fault manifests. Notice that now this decision is more complicated than in the previous modelling strategies. On the one hand, the cases of the activity *regionFailure* take into account the proportions with which each entity located in each region (of the type being modelled) exhibits different failure modes. On the other hand, in some regions submodels, the cases of *regionFailure* also reflect that the errors generated by the fault can be contained by a given CAN controller, which forces the fault to manifest as stuck-at-recessive. This aspect is depicted in Figure 10.6, in which the part that models the error-containment actions performed by the corresponding CAN controller is represented as being embedded in the expression that specifies the proportion with which each case is selected. As will be seen later, the only regions submodels that take into account the error-containment mechanisms

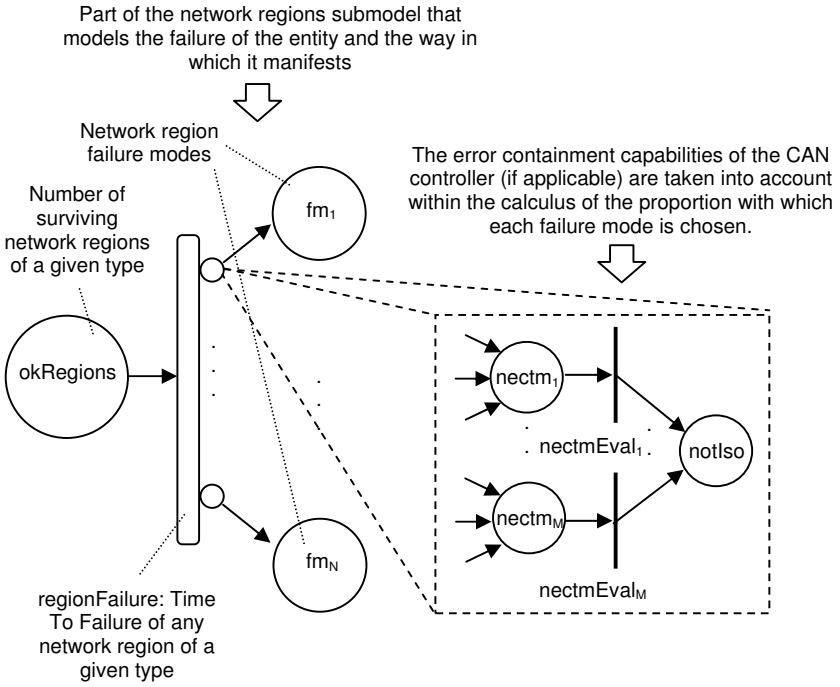

Figure 10.6: Basic structure of a regions submodel

of the CAN controller are those that represent all Node Connections in the CAN bus, CANcentrate or in ReCANcentrate. This is because a CAN controller is only able to potentially contain errors generated by faults affecting some of the entities located in its own Node Connection (see Section 10.4.5 for a detailed explanation concerning the error-containment capabilities of a CAN controller).

Like in the former modelling strategies, the occurrence of a fault triggers a process that evaluates whether or not the fault is isolated, i.e. it starts what we call the *coverage process*. Notice that to include the CAN controller's error-containment capabilities within the calculus of the proportions of the cases of the activity *regionFailure* of some regions submodels implies that part of the coverage process is partially carried out by these submodels. In fact, since the CAN bus does not include any error-containment mechanism different from those implemented at CAN controllers, the only actions related to the coverage process are those modelled by the activity *regionFailure* of the submodel that represents all its Node Connections. Conversely, in the model of CANcentrate, it is necessary to carry out the part of the coverage process that is related to the error-containment capabilities of

the hub. This part of the coverage process is carried out by a dedicated coverage submodel, which shares the places fm_i with the regions submodels. Similarly, as will be explained in Chapter 12, the model of ReCANcentrate carries out this process by means of several coverage submodels, where each of which represents a different error-containment or fault-tolerance mechanism of this communication infrastructure.

Finally, as concerns the way in which this third strategy models the failure of the overall system and measures its probability, it is done as in the previous approaches, i.e. using an evaluator submodel and the *probNonSev* reward variable (see Section 10.6.1).

As can be inferred from the above discussion, the actual innovative feature of this third strategy is the way in which the regions submodel models both the Time To Failure and the failure modes of every individual entity of each one of the regions the submodel represents. Therefore, next we explain this issue in more detail.

As just mentioned above, the Time To Failure of each individual entity is implicitly modelled by means of the activity *regionFailure*, as depicted in Figure 10.6. Notice that this activity represents the time that elapses until a fault occurs in any entity of any of the surviving regions of a given type, i.e. in any region (of a given type) that does not include any faulty entity. Therefore, the Time To Failure distribution modelled by the activity *regionFailure* can be easily calculated as follows. First, suppose that a region is composed of E entities, each of which independently fails following an exponential Time To Failure distribution with failure rate λ_i, where $i \in [1, E]$. Therefore, the Time To Failure distribution of a given region can be written as:

$$F_{region}(t) = (1 - e^{-\lambda_{region} \cdot t}) \tag{10.1}$$

where λ_{region} is the failure rate of the whole region, and it is calculated as the summation of the failure rates of each one of its individual entities:

$$\lambda_{region} = \sum_{i=1}^{E} \lambda_i \tag{10.2}$$

Second, notice that all the surviving regions represented by the regions submodel are equal to each other. This implies that each one of them has the same entities and, thus, the same Time To Failure distribution. As a consequence, the Time To

Failure distribution represented by the activity *regionFailure* can be calculated as:

$$F_{regionType}(t) = (1 - e^{-\lambda_{regionType} \cdot t}) \tag{10.3}$$

where $\lambda_{regionType}$ is the rate with which any surviving region the submodel represents fails. Specifically, if *okRegions→Mark()* represents the number of surviving regions, i.e. the marking of the place *okRegions*, then the value of $\lambda_{regionType}$ can be obtained as:

$$\begin{aligned} \lambda_{regionType} &= okRegions \rightarrow Mark() \cdot \lambda_{region} \\ &= okRegions \rightarrow Mark() \cdot \sum_{i=1}^{E} \lambda_i \end{aligned} \tag{10.4}$$

The activity *regionFailure* is also the responsible for modelling the failure modes of every individual entity of each one of the regions the submodel represents. As already explained, *regionFailure* does this, implicitly, by means of its cases, which in a submodel that represents all Node Connections also take into account the error-containment abilities of the corresponding CAN controller (see Figure 10.6). Anyway, for the sake of clarity, let us explain first how *regionFailure* takes into account the entities' failure modes. Afterwards, we will describe how it includes the error-containment capacities of the CAN controller in the calculus of the cases' proportions.

Since each one of the regions that a regions submodel represents includes exactly the same entities, it is possible to calculate the proportions of the *regionFailure*'s cases in terms of the failure mode proportions of the entities of just one region. Also notice that we consider that faults are not near coincident in time, so that a region's failure modes are determined assuming that only one of its entities has failed. In this sense, let us consider again that a region is constituted by E entities that exhibit M different failure modes. Moreover, imagine that a given failure mode is called fm_j with $j \in [1, M]$, so that the proportion with which an entity i exhibits the failure mode fm_j is denoted by $fmp_{i,j}$ with $fmp_{i,j} \in [0, 1]\ \forall i \in [1, E]$ and $\forall j \in [1, M]$. Then, the proportion with which the region exhibits the failure mode fm_j (and, thus, the proportion with which the case that represents this mode is selected), is a function that depends on the failure rates of its constituent entities, λ_i, as well as on the proportions with which each one of these entities exhibits that failure mode, $fmp_{i,j}$. We call this function Fmp_j and it is denoted by:

$$\mathit{Fmp}_j(\lambda_1, ..., \lambda_E, \mathit{fmp}_{1,j}, ..., \mathit{fmp}_{E,j})$$

Given this function, the probability with which a region fails exhibiting the failure mode fm_j at or before time t can be written as:

$$F_{\mathit{region}_j}(t) = F_{\mathit{region}} \cdot \mathit{Fmp}_j = (1 - e^{-\lambda_{\mathit{region}} \cdot t}) \cdot \mathit{Fmp}_j \quad \forall j \in [1, M] \tag{10.5}$$

Moreover, since the region cannot exhibit more than one failure mode simultaneously, the Time To Failure distribution of the region can be expressed in terms of F_{region_j} as:

$$\begin{aligned} F_{\mathit{region}}(t) &= \sum_{j=1}^{M} F_{\mathit{region}_j}(t) = \sum_{j=1}^{M} [(1 - e^{-\lambda_{\mathit{region}} \cdot t}) \cdot \mathit{Fmp}_j] \\ &= (1 - e^{-\lambda_{\mathit{region}} \cdot t}) \cdot \sum_{j=1}^{M} \mathit{Fmp}_j \end{aligned} \tag{10.6}$$

Notice that if we take into account Equation 10.1 and Equation 10.6, we corroborate that:

$$\sum_{j=1}^{M} \mathit{Fmp}_j = 1 \tag{10.7}$$

At this point, we can expand Equation 10.1 in such a way that it includes the entities' failure rates and failure mode proportions (λ_i and $\mathit{fmp}_{i,j}$) and, then, we can compare the resultant expression with Equation 10.6 to obtain the expression of Fmp_j in terms of λ_i and $\mathit{fmp}_{i,j}$. Specifically, in order to expand Equation 10.1, we can use an expression that specifies $\lambda_{\mathit{region}}$ in terms of the entities' failure rates and failure mode proportions:

$$\lambda_{\mathit{region}} = \sum_{i=1}^{E} \lambda_i = \sum_{i=1}^{E} \left(\lambda_i \cdot \sum_{j=1}^{M} \mathit{fmp}_{i,j}\right) \tag{10.8}$$

More specifically, Equation 10.1 can be expanded as follows:

$$
\begin{aligned}
F_{region}(t) &= (1 - e^{-\lambda_{region} \cdot t}) \\
&= (1 - e^{-\lambda_{region} \cdot t}) \cdot 1 \\
&= (1 - e^{-\lambda_{region} \cdot t}) \cdot \frac{\lambda_{region}}{\lambda_{region}} \\
&= (1 - e^{-\lambda_{region} \cdot t}) \cdot \frac{\sum_{i=1}^{E} (\lambda_i \cdot \sum_{j=1}^{M} fmp_{i,j})}{\lambda_{region}} \\
&= (1 - e^{-\lambda_{region} \cdot t}) \cdot \frac{\sum_{\substack{i=1..E \\ j=1..M}} \lambda_i \cdot fmp_{i,j}}{\lambda_{region}} \\
&= (1 - e^{-\lambda_{region} \cdot t}) \cdot \sum_{\substack{i=1..E \\ j=1..M}} \frac{\lambda_i \cdot fmp_{i,j}}{\lambda_{region}} \\
&= (1 - e^{-\lambda_{region} \cdot t}) \cdot \sum_{j=1}^{M} \left(\frac{\sum_{i=1}^{E} \lambda_i \cdot fmp_{i,j}}{\lambda_{region}} \right)
\end{aligned}
\tag{10.9}
$$

At this point, if we compare this expression with Equation 10.6, we deduce that Fmp_j can be written as:

$$
Fmp_j = \frac{\sum_{i=1}^{E} \lambda_i \cdot fmp_{i,j}}{\lambda_{region}} = \frac{\sum_{i=1}^{E} \lambda_i \cdot fmp_{i,j}}{\sum_{i=1}^{E} \lambda_i} \quad \forall j \in [1, M] \tag{10.10}
$$

This expression of Fmp_j indicates that the proportion with which a surviving region (and all the surviving regions of a given type together) exhibits a given failure mode is the weighted arithmetic mean of the proportions with which each entity of the region manifests that failure mode. Specifically, the proportion with which each entity exhibits the failure mode is weighted considering the contribution of its failure rate to the failure rate of the region.

Finally, let us explain how we take into account the error-containment capabilities of a CAN controller when calculating the proportions of the cases of the activity *regionFailure* of a regions submodel that represents all Node Connections. For this purpose, notice that a CAN controller prevents the propagation of errors just by stop operating when it diagnoses a fault that affects an entity of its Node Connection. This implies that a given percentage of an entity failure that leads the Node Connection to manifest as stuck-at-dominant, as well as a given percentage of that

entity failure that leads the Node Connection to manifest as bit-flipping actually will lead the Node Connection to manifest as stuck-at-recessive. These percentages are obtained from the coverages that characterize the probability with which the CAN controller successfully contains stuck-at-dominant and bit-flipping errors generated by that entity respectively (see Section 10.4.5 for a detailed description concerning these coverages). In general terms, these coverages are reflected in the calculus of the Node Connection's failure mode proportions in accordance with the following equations:

$$\begin{aligned} Fmp_{str} &= \frac{\sum_{i=1}^{E} \lambda_i \cdot fmp_{i,str}}{\sum_{i=1}^{E} \lambda_i} + \\ &\frac{\sum_{i=1}^{E} (\lambda_i \cdot fmp_{i,std} \cdot stdCov_i + \lambda_i \cdot fmp_{i,flip} \cdot flipCov_i)}{\sum_{i=1}^{E} \lambda_i} \\ Fmp_{std} &= \frac{\sum_{i=1}^{E} \lambda_i \cdot fmp_{i,std} \cdot (1 - stdCov_i)}{\sum_{i=1}^{E} \lambda_i} \\ Fmp_{flip} &= \frac{\sum_{i=1}^{E} \lambda_i \cdot fmp_{i,flip} \cdot (1 - flipCov_i)}{\sum_{i=1}^{E} \lambda_i} \end{aligned} \tag{10.11}$$

where now Fmp_{str}, Fmp_{std} and Fmp_{flip} are the proportions with which the Node Connection exhibits a stuck-at-recessive, a stuck-at-dominant and a bit-flipping failure respectively; $fmp_{i,str}$, $fmp_{i,std}$ and $fmp_{i,flip}$ are the proportions with which entity i manifests as stuck-at-recessive, stuck-at-dominant and bit-flipping respectively; and $stdCov_i$ and $flipCov_i$ are the coverages with which the CAN controller detects and isolates the entity i when it manifests as stuck-at-dominant and bit-flipping respectively.

10.7 CANcentrate model

This section describes the model of the reliability of a system that relies on CANcentrate. Figure 10.7 depicts the general structure of this model, which is a composition of submodels interconnected by means of the Join primitive. The model adheres to the third modelling approach explained above. In this sense, notice that it comprises three regions submodels: *nodeKernelsT*, *nodeConnsT* and the *hubKernel*, which respectively represent all Node Kernel regions, all Node Connection regions and the Hub Kernel region.

Besides these submodels we can find the *branchesFailureEval* submodel, which acts as the *coverage submodel*. When a Node Kernel or a Node Connection fails,

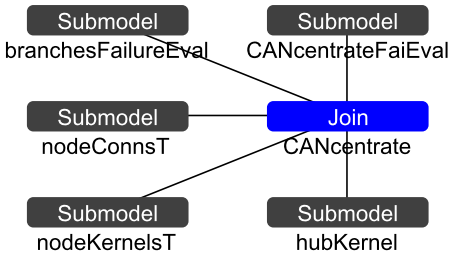

Figure 10.7: CANcentrate model

this model evaluates whether or not the hub is able to successfully isolate the corresponding hub port. Notice that, from now on, we will refer to a CANcentrate hub port as either a hub uplink port or a branch.

Finally, there is also one *evaluator submodel* called *CANcentrateFaiEval*. It evaluates the number of nodes that can still operate and communicate among them to decide whether or not the system is faulty.

Next, all these submodels are explained in more detail. The meaning and the default values of the parameters included in the following description can be found in Sections 10.4.4 and 10.4.5, as well as in tables 10.4, 10.5 and 10.6.

10.7.1 *nodeKernelsT* submodel

The *nodeKernelsT* submodel, which is depicted in Figure 10.8, represents all the Node Kernel regions of CANcentrate. As explained before, a Node Kernel region only includes the entity Node Core, whose constituent components are specified in Section 10.4.2.

The marking of the place *okNodeKernels* denotes the number of Node Kernel regions that are not faulty; whereas a token in any of the places *stuckBranch*, *flipBranch*, and *outFauMod* indicates that a Node Kernel has failed leading its related branch to suffer from the corresponding type of failure.

The activity *nkFailure* models the time that elapses until any non-faulty Node Kernel region fails. This time is exponentially distributed in accordance with Equation 10.4. Specifically, taking into account this equation and that a Node Kernel is constituted by the Node Core entity only, the rate with which a Node Kernels fails is:

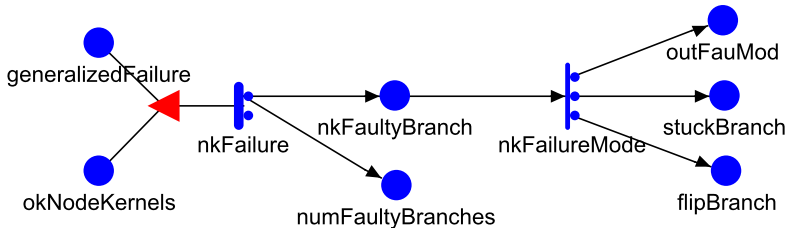

Figure 10.8: nodeKernelsT submodel

$$okNodeKernels \rightarrow Mark() \cdot nodeCoreFRate$$

where *okNodeKernels→Mark()* is the marking of the place *okNodeKernels* and *nodeCoreFRate* is the failure rate of the entity Node Core.

When the activity *nkFailure* fires, a token is erased from *okNodeKernels* and one of its two cases, represented by circles at the right edge of the activity, is chosen. The first case (the upper one) represents a fault that affects a Node Kernel of a node that is located in a non-faulty branch; whereas the second one models a fault happening in a Node Kernel of a node that is placed at a branch that was already faulty, i.e. a Node Kernel attached to an already isolated hub port. The proportion with which *nkFailure* chooses each case reflects the probability that the Node Kernel is placed (or not) at an already faulty branch. For instance, the first case's proportion is:

$$\frac{(numBranches - numFaultyBranches \rightarrow Mark())}{okNodeKernels \rightarrow Mark()}$$

Parameter *numBranches* indicates the number of total branches of CANcentrate, whereas *numFaultyBranches* is a place that submodels share and whose marking indicates the number of branches that are faulty. Notice that the Node Kernel of a non-faulty branch is always non-faulty (otherwise the branch would be faulty). Therefore, the above expression can calculate the probability that a surviving Node Kernel is placed in a non-faulty branch by simply dividing the number of surviving branches by the number of Node Kernels that were not faulty. Likewise the proportion of the second case, which is the probability that the surviving Node Kernel that fails is placed at an already faulty branch, is:

$$\frac{(okNodeKernels \rightarrow Mark() - (numBranches - numFaultyBranches \rightarrow Mark()))}{okNodeKernels \rightarrow Mark()}$$

Concerning the actions taken after the activity *nkFailure* selects a case, notice that the second one is not connected to any place. This is because since the branch was already faulty and thus, isolated, the Node Kernel failure has no further impact on the communication. Thus, no more actions are performed within the model when this case is chosen.

Conversely, the first case is connected to places *numFaultyBranches* and *nkFaultyBranch*. When this case is selected, the marking of *numFaultyBranches* is increased in one unit to reflect that the branch corresponding to the faulty Node Kernel becomes faulty. Additionally, a token is placed at *nkFaultyBranch*, thereby enabling the activity *nkFailureMode*. This activity is instantaneous, so that it immediately fires transferring the token to one of the three places: *outFauMod*, *stuckBranch* and *flipBranch*, which respectively represent that the fault manifests as an ofm, a stuck-at and a bit-flipping failure at the hub port.

The first case of *nkFailureMode* is selected with probability *nodeCoreOfmProp*, which is the proportion with which the entity Node Core exhibits an ofm failure, i.e. a failure mode that is not included in the fault model and that leads to the failure of the whole system. When this occurs, a token is placed in *outFauMod*. This place is shared with submodels *nodeConnsT*, *hubKernel* and *CANcentrateFaiEval*. As will be explained later, whenever *CANcentrateFaiEval* detects a token in this place, it sets a token in place *generalizedFailure* to indicate to all submodels that the whole system is faulty. This leads all submodels stop evolving to reduce the state space. Specifically, as can be seen in Figure 10.8, place *generalizedFailure* is connected to the input gate corresponding to the activity *nkFailure*. This allows stopping the *nodeKernelsT* submodel by disabling this activity when a generalized failure occurs. The same strategy is used to stop the submodels *nodeConnsT* and *hubKernel*.

Regarding the second case of *nkFailureMode*, it is selected with proportion:

$$nodeCoreStrProp + nodeCoreStdProp$$

which is the probability with which the Node Core exhibits a stuck-at-recessive or a stuck-at-dominant failure mode (the Node Kernel is only constituted by a Node Core entity). Notice that there is no need to differentiate between the situation in which the Node Kernel exhibits a stuck-at-recessive and a stuck-at-dominant failure, since the hub isolates both types of faults with a perfect coverage. As concerns

the third case of *nkFailureMode*, it is selected with proportion *nodeCoreFlipProp*, which is the one with which the Node Core fails provoking a bit-flipping failure.

Finally, it important to note that we have differentiated between a stuck-at-recessive/dominant and a bit-flipping failure when, in fact, in Section 10.4.4 we specified that we are convinced that it is almost impossible that a Node Core fails in a way that compels the node's CAN controller to transmit a stuck-at-dominant or a bit-flipping fault. We reflect this assumption by specifying a value of 0 for the corresponding model parameters, i.e. for *nodeCoreStdProp* and *nodeCoreFlipProp*, but the structure of the model still contemplates the possibility that the Node Core exhibits a stuck-at-dominant or a bit-flipping failure.

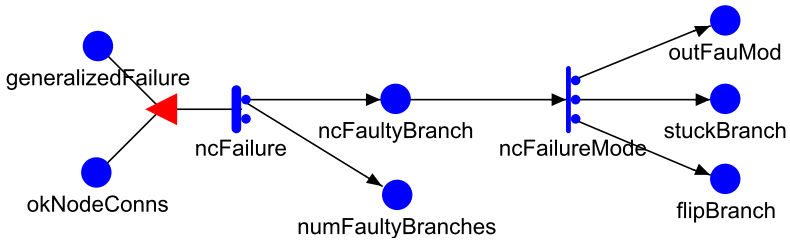

Figure 10.9: nodeConnsT submodel

10.7.2 *nodeConnsT* submodel

The *nodeConnsT* submodel models all Node Connection regions of CANcentrate. As depicted in Figure 10.9 its structure is the same as the structure of the *nodeKernelsT* submodel. The difference between both submodels lies in the fact that a Node Connection region is constituted by several entities, whereas a Node Core region is only formed from the entity Node Core. This complicates the way in which the failure rate and the failure mode proportions are specified in the corresponding activities.

The failure rate of the exponential Time To Failure distribution of the activity *ncFailure* is calculated adding the failure rates of all the entities that compose each Node Connection region, as indicated before in Equation 10.4. This rate is:

$$okNodeConns \rightarrow Mark() \cdot (ctrlFRate + 2 \cdot (nodeIOFRate + lnkAttchFRate + hubIOFRate + 2 \cdot termFRate))$$

Notice that the failure rate of each entity is added as many times as it is included

within the Node Connection. For instance, the failure rate of the entity Attachment, *lnkAttchFRate*, is added twice, since there are two Attachment entities within a Node Connection region, i.e. one Attachment for the uplink and another one for the downlink.

Regarding the proportions of the cases of the activity *ncFailure*, they are calculated as in the *nodeKernelsT* submodel. The proportion of the first one, which represents the situation in which the Node Connection that fails is located in a non-faulty branch, is:

$$\frac{(numBranches - numFaultyBranches \rightarrow Mark())}{okNodeConns \rightarrow Mark()}$$

Whereas the proportion of the second case of *ncFailure*, which models the situation in which the Node Connection that fails is located in an already faulty and isolated branch, is calculated as:

$$\frac{(okNodeConns \rightarrow Mark() - (numBranches - numFaultyBranches \rightarrow Mark()))}{okNodeConns \rightarrow Mark()}$$

Finally, let us explain in detail the activity *ncFailureMode*, which decides the way in which the Node Connection failure manifests at the corresponding hub uplink port. The probability with which this activity chooses its first case is the proportion with which the Node Connection region fails exhibiting an *out-of-fault-model* (ofm) failure. According to Equation 10.10, this proportion is:

$$2 \cdot lnkAttchOfmProp \cdot lnkAttchFRate + 2 \cdot nodeIOOfmProp \cdot nodeIOFRate + ctrlOfmProp \cdot ctrlFRate + 2 \cdot hubIOOfmProp \cdot hubIOFRate + 4 \cdot termOfmProp \cdot termFRate$$

Notice that it is not necessary to divide the above expression by the failure rate of the Node Connection region in order to adhere to Equation 10.10. This is because Moëbius normalizes the case proportions. Thus, such a division is not necessary as long as the sum of all the case proportions is that failure rate (which is the case indeed).

The second and third cases of *ncFailureMode* model a stuck-at and a bit-flipping fault respectively. In order to calculate the proportions of these cases we apply Equation 10.10, but taking into account that the CAN controller that forms part

of the Node Connection is able to contain, to some extent, the errors generated by faults affecting the Node Connection. More specifically, the CAN controller is able to diagnose faults happening at other entities of the Node Connection it belongs to and, then, to stop operating in order to prevent error propagation. See Section 10.4.5 for a detailed explanation of what are the entities whose failure the CAN controller can diagnose and isolate, as well as what are the coverages with which the CAN controller can do so.

As already said in Section 10.6.3, if the CAN controller isolates the entity responsible for the failure of the Node Connection, then the Node Connection will exhibit a stuck-at-recessive failure, i.e. a fail-silent failure. This implies that a given percentage of the entity failures that lead the Node Connection to manifest as stuck-at-dominant, as well as a given percentage of the entity failures that lead the Node Connection to manifest as bit-flipping actually will lead the Node Connection to manifest as stuck-at-recessive. These percentages are the coverages with which the CAN controller contains stuck-at-dominant and bit-flipping errors generated by that entity respectively, and they are included in the calculus of the Node Connection's failure modes' proportions following Equations 10.11.

In the particular case of a Node Connection region of CANcentrate, the two first equations specified in 10.11 are applied as follows in order to calculate the proportion with which *ncFailureMode* selects its first case, i.e. the case that models the situation in which the Node Connection manifests a the stuck-at-recessive or as stuck-at-dominant. These two failure modes are modelled by means of a single case, because the hub isolates both types of faults with a perfect coverage and, thus, it is not necessary to differentiate between them.

$$
\begin{aligned}
&(\mathit{hubIOStrProp} + \mathit{hubIOStdProp}) \cdot \mathit{hubIOFRate} + \\
&(\mathit{termStrProp} + \mathit{termStdProp}) \cdot \mathit{termFRate} \cdot 2.0 + \\
&(\mathit{lnkAttchStrProp} + \mathit{lnkAttchStdProp}) \cdot \mathit{lnkAttchFRate} + \\
&(\mathit{nodeIOStrProp} + \mathit{nodeIOStdProp}) \cdot \mathit{nodeIOFRate} + \\
\\
&[\mathit{hubIOFlipProp} \cdot \mathit{hubIOFRate} + \\
&(\mathit{termLossProp} + \mathit{termFlipProp}) \cdot \mathit{termFRate} \cdot 2.0 + \\
&(\mathit{lnkAttchDisProp} + \mathit{lnkAttchFlipProp}) \cdot \mathit{lnkAttchFRate} + \\
&\mathit{nodeIOFlipProp} \cdot \mathit{nodeIOFRate}] \cdot \mathit{ctrlFlipCov} + \\
\\
&(\mathit{ctrlStrProp} + \mathit{ctrlStdProp}) \cdot \mathit{ctrlFRate} + \\
&\mathit{ctrlFlipProp} \cdot \mathit{ctrlFRate} \cdot (\mathit{ctrlFlipCov} \cdot \mathit{ctrlItselfIsoCov}) + \\
\\
&(\mathit{nodeIOStrProp} + \mathit{nodeIOStdProp}) \cdot \mathit{nodeIOFRate} + \\
&(\mathit{termStrProp} + \mathit{termStdProp}) \cdot \mathit{termFRate} \cdot 2.0 + \\
&(\mathit{lnkAttchStrProp} + \mathit{lnkAttchStdProp}) \cdot \mathit{lnkAttchFRate} + \\
&(\mathit{hubIOStrProp} + \mathit{hubIOStdProp}) \cdot \mathit{hubIOFRate}
\end{aligned}
$$

Notice that it is not necessary to divide the above expression by the failure rate of the Node Connection region to adhere to the first one of Equations 10.11. As already said, this is because the sum of the proportions of the three cases of the activity *ncFailureMode* is this failure rate and Moëbius automatically normalizes the case proportions.

In order to better understand the above expression, we have written it in separated blocks. This allows us to highlight different aspects concerning the capacity of the CAN controller and the hub to contain errors. See Section 10.4.5 for a detailed discussion on that issue. The first and second blocks of lines correspond to faults of entities that connect the node to the downlink hub port. The first block represents these faults when they manifest as a stuck-at. The second block corresponds to these faults when they generate bit-flipping streams that the CAN controller of the node diagnoses and isolates, thereby forcing the faults to manifest as a stuck-at-recessive at the hub uplink port. In this sense, notice that *ctrlFlipCov* is the coverage with which the CAN controller successfully diagnoses bit-flipping faults. Also notice that this blocks reflects that a physical disruption of an Attachment entity and the loss of a Termination are considered as bit-flipping faults; as explained

in Section 10.4.4. The third block of the expression represents the internal CAN controller faults: the ones that manifest as the transmission of a stuck-at stream, and the ones that manifest as a bit-flipping stream but that the CAN controller successfully isolates. Notice that this block implicitly considers a value of the 100% for the error-containment coverage of the CAN controller internal faults that manifest as a stuck-at-dominant stream. This is because although the CAN controller does not always isolate an internal fault that leads it to send a stuck-at-dominant stream, the hub diagnoses and isolates this type of failure with an effectiveness of the 100% at the corresponding uplink hub port. The last block corresponds to the entities that connect the node to the uplink hub port. The CAN controller can do nothing to isolate bit-flipping fault affecting these entities, so that the expression only includes the cases in which these entities exhibit a stuck-at-fault.

Regarding the second case of the activity *ncFailureMode*, it is selected with the proportion with which the Node Connection exhibits a fault that manifests as bit-flipping at the uplink hub port. This proportion adheres to the third one of Equations 10.11. Next we write this proportion's expression split into different blocks, each one corresponding to a specific aspect that should be highlighted:

$$
\begin{aligned}
&(\mathit{hubIOFlipProp} \cdot \mathit{hubIOFRate} + \\
&(\mathit{termLossProp} + \mathit{termFlipProp}) \cdot \mathit{termFRate} \cdot 2.0 + \\
&(\mathit{lnkAttchDisProp} + \mathit{lnkAttchFlipProp}) \cdot \mathit{lnkAttchFRate} + \\
&\mathit{nodeIOFlipProp} \cdot \mathit{nodeIOFRate}) \cdot (1.0 - \mathit{ctrlFlipCov}) + \\
&\\
&\mathit{ctrlFlipProp} \cdot \mathit{ctrlFRate} \cdot (1.0 - \\
&\mathit{ctrlFlipCov} \cdot \mathit{ctrlItselfIsoCov}) + \\
&\\
&\mathit{nodeIOFlipProp} \cdot \mathit{nodeIOFRate} + \\
&(\mathit{termLossProp} + \mathit{termFlipProp}) \cdot \mathit{termFRate} \cdot 2.0 + \\
&(\mathit{lnkAttchDisProp} + \mathit{lnkAttchFlipProp}) \cdot \mathit{lnkAttchFRate} + \\
&\mathit{hubIOFlipProp} \cdot \mathit{hubIOFRate})
\end{aligned}
$$

The first block represents the entities that connect the node to the downlink hub port when they fail exhibiting a bit-flipping fault that the CAN controller does not successfully diagnose. As can be seen, these bit-flipping faults are not diagnosed (and thus not isolated) with probability $1.0 - \mathit{ctrlFlipCov}$. Similarly, the second block corresponds to faults that affect the internals of the CAN controller compelling it to transmit a bit-flipping stream that the CAN controller does not isolate.

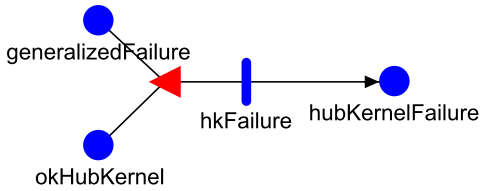

Figure 10.10: hubKernel submodel

The last block gathers the entities that connect the node to the uplink hub port and that fail exhibiting a bit-flipping failure.

10.7.3 *hubKernel* submodel

The last regions submodel is the *hubKernel*. Its structure, which is depicted in Figure 10.10, is very simple. Initially, there is one token in the place *okHubKenel* to indicate that the only Hub Kernel region of the system is not faulty. The activity *hkFailure* models the exponential Time To Failure distribution of the Hub Kernel. Since this region only includes the entity Hub Core, the failure rate of the distribution is *hubCoreFRate*.

Moreover, as already said, we assume that every Hub Core failure leads to the failure of the overall system. Therefore, the activity *hkFailure* does not differentiate between different failure modes. Instead, whenever the activity *hkFailure* fires, it transfers the token from *okHubKernel* to the place *hubKernelFailure*. As will be explained later, this compels the *CANcentrateFaiEval* submodel to diagnose the overall system as faulty.

10.7.4 *branchesFailureEval* submodel

Submodel *branchesFailureEval* (Figure 10.11) evaluates whether or not the hub isolates each fault happening in a Node Kernel or in a Node Connection region. For this purpose, it shares the places *stuckBranch* and *flipBranch* with the submodels *nodeKernelsT* and *nodeConnsT*. When a token is set in any of these places, *branchesFailureEval* decides whether or not the corresponding failure is successfully confined by the hub, and then it transfers the token to *faultyCoveredBranches* or to *faultyNonCoveredBranch* accordingly.

Specifically, the marking of place *faultyCoveredBranches* indicates the number of branches that are faulty but successfully isolated by the hub. Conversely, a to-

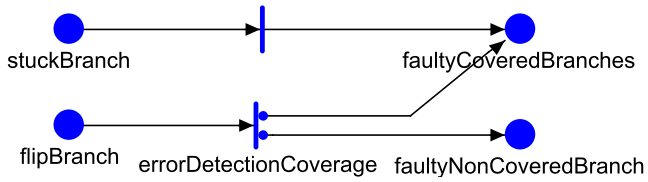

Figure 10.11: BranchesFailureEval submodel

ken in place *faultyNonCoveredBranch* means that a branch has failed and that the hub was unable to isolate it. Since the error-containment coverage of stuck-at failures is of the 100%, a token in *stuckBranch* is always transferred to *faultyCoveredBranches*. In contrast, the error-containment coverage of a bit-flipping fault is not perfect. Submodel *branchesFailureEval* models this fact by transferring any token set in *flipBranch* to *faultyCoveredBranches* or to *faultyNonCoveredBranch* with probabilities *flipLnkCov* and $1.0 - flipLnkCov$ respectively; where *flipLnkCov* is the coverage with which the hub isolates a bit-flipping fault at the corresponding hub uplink port (see Section 10.4.5 and Table 10.6).

10.7.5 *CANcentrateFaiEval* submodel

CANcentrateFaiEval submodel, which is depicted in Figure 10.12, is devoted to deciding when the system fails as a whole. When this happens, it sets a token in the place *generalizedFailure*, thereby compelling the regions submodels to stop evolving, as explained in Section 10.7.1.

The *CANcentrateFaiEval* submodel becomes aware of failures by means of different places it shares with other submodels: *outFauMod* with all regions submodels; place *hubKernelFailure* just with the *hubKernel* submodel; and *faultyCoveredBranches* and *faultyNonCoveredBranch* with the *branchesFailureEval* submodel.

Each one of these places is connected to place *generalizedFailure* by means of an input gate and an *instantaneous activity*. Each input gate enables the activity it is connected to, depending on the marking of its incoming place. In particular, whenever a token is received in *faultyNonCoveredBranch* or in *outFauMod*, the corresponding input gate enables the firing of its activity, so that a token is placed at *generalizedFailure*. This is because a fault that cannot be isolated propagates thereby provoking the failure of the overall system. Similarly, since the hub is a single point of failure, when a token is set in the place *hubKernelFailure*, the corresponding activity instantaneously writes a token at *generalizedFailure*.

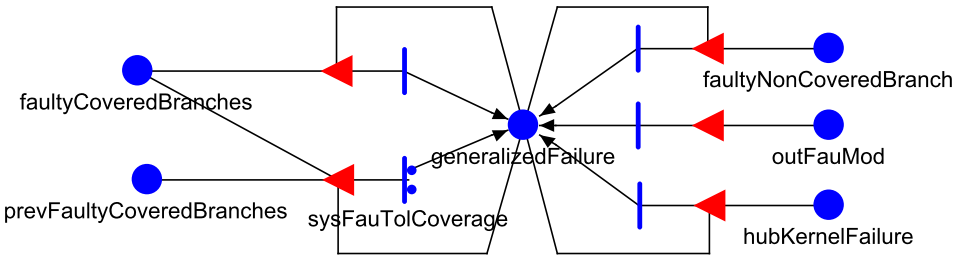

Figure 10.12: CANcentrateFaiEval submodel

In contrast, the number of tokens in *faultyCoveredBranches* that are necessary to enable its associated activity to fire depends on the minimum number of nodes that must communicate among them to consider that the system is not faulty. As explained in Section 10.6.1, this minimum number of nodes is configured by means of the parameter *kSevere* (see Table 10.4). If the marking of *faultyCoveredBranches* exceeds the value of *kSevere*, then the activity fires transferring a token to *generalizedFailure*. If one token is enough to enable this activity, *kSevere* $= 0$, then the probability of not having a token in *generalizedFailure* is the non-fault-tolerant/accepting system reliability (NFT/AR). Conversely, if more than one token is necessary to enable the activity, *kSevere* $= k$ with $k > 0$, then the probability of not having a token in *generalizedFailure* is the fault-tolerant/accepting system reliability (FT/AR$_k$). In other words, a value of *kSevere* equal to 0 must be used for measuring the reliability of an NFT/A system, whereas greater values of this parameter must be specified to calculate the reliability of an FT/A system. Please, refer to Section 10.2 for more details concerning this issue.

Finally, as concerns an FT/A system, notice that *CANcentrateFaiEval* also models the capacity of this type of system to successfully accept or tolerate the failure or disconnection of a new node, provided that the system can do so. More specifically, when the hub isolates a new faulty branch and the number of isolated branches (including the new faulty branch) still does not represent a k-severe failure, *CANcentrateFaiEval* evaluates whether or not the system actually accepts or tolerates the disconnection of that new branch.

For this purpose *CANcentrateFaiEval* includes the place *prevFaultyCoveredBranches*, the activity *sysFauTolCoverage* and the corresponding input gate. The marking of *prevFaultyCoveredBranches* tracks the marking of the place *faultyCoveredBranches*, which as already said represents the number of branches that have been isolated so far. When a new branch fails and the hub successfully isolates it, a new token is set in *faultyCoveredBranches*, so that the marking of this place

becomes greater than the one of *prevFaultyCoveredBranches*. Then, if a k-severe failure has not occurred as a consequence of the fault, the difference between the markings of *faultyCoveredBranches* and *prevFaultyCoveredBranches* enables the activity *sysFauTolCoverage*, which instantaneously fires in order to evaluate if the FT/A system accepts or tolerates the fault. Specifically, the expression that enables the activity *sysFauTolCoverage*, and which is evaluated by its corresponding input gate, is:

$$faultyCoveredBranches \rightarrow Mark() > prevFaultyCoveredBranches \rightarrow Mark() \text{ and}$$
$$faultyCoveredBranches \rightarrow Mark() \leq kSevere$$

Notice that when *sysFauTolCoverage* fires, its input gate forces the marking of *prevFaultyCoveredBranches* to be equal to the marking of *faultyCoveredBranches*, thereby ensuring that *sysFauTolCoverage* does not fire again for the same fault.

As regards the actions taken by *sysFauTolCoverage*, it has to choose one of its two cases. The upper one represents the situation in which the FT/A system does not successfully accept or tolerate the failure of a new branch and, therefore, it sets a token in the place *generalizedFailure*. The other case models the opposite situation and it is not connected to any place. *sysFauTolCoverage* selects its first and second cases with probabilities $1 - sysFauTolCov$ and *sysFauTolCov* respectively; where *sysFauTolCov* is the coverage with which the FT/A system accepts or tolerates the failure or disconnection of a node, provided that it is able to do that (see Section 10.4.5 and Table 10.4 for more details concerning this coverage).

10.8 CANbus model

We have modelled the reliability of a system relying on a CAN bus by means of the *CANbus* model, which also adheres to the modelling approach explained in Section 10.6.3. As shown in Figure 10.13, its structure is very similar to the one of the *CANcentrate submodel*: it includes some regions submodels and an *evaluator submodel* that are joined by means of the *Rep* primitive.

Like in the *CANcentrate* model, the *nodeKernelsB* and the *nodeConnsB* submodels represent the Node Kernel and the Node Connection regions respectively. However, notice that a Node Connection region contains fewer entities in the case of the CAN bus than in the case of CANcentrate, as described in Section 10.6.3.

The *CANbus* model also includes two additional regions submodels when compared with the *CANcentrate* model: the *inBusSections* and the *edBusSections* sub-

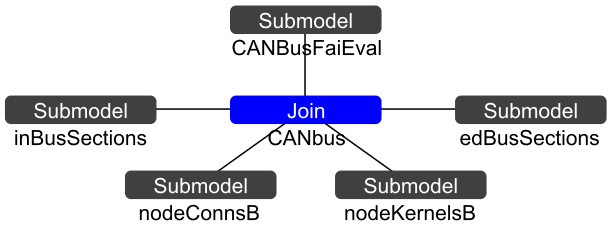

Figure 10.13: CANbus model

models. The first one represents all the Internal Bus Section regions of the CAN bus. As explained in Section 10.6.3, an Internal Bus Section region includes the Attachment entity that represents the section of the CAN bus line that connects two adjacent nodes that are not located at the extremities of the bus. If a bus includes N nodes, then it has $N - 2$ Internal Bus Section regions. The other submodel, the *edBusSections*, corresponds to the two Edge Bus Section regions of the CAN bus. As described in Section 10.6.3, each Edge Bus Section region includes one Termination entity and the Attachment entity that represents the section of the CAN bus line that connects a node located at a extremity of the bus with the next node in the bus line (in a daisy chain configuration).

The last submodel is an *evaluator submodel* called *CANBusFaiEval*. It takes into account the faults that have occurred so far to diagnose when the overall system is faulty. To facilitate this diagnosis we further classify the failures modes that are within the fault model into *exclusion failures* and *blocking failures*. The former refers to a situation in which the fault leads only one node to be prevented from communicating; whereas the second occurs when the errors generated by the faulty entity prevent all nodes from communicating. For instance, a Node Connection that fails transmitting a stuck-at-recessive stream only prevents its node from communicating and, thus, it is an exclusion failure. In contrast, a stuck-at-recessive Internal Bus Section does prevent all nodes from communicating and, hence, it is a blocking failure.

10.8.1 *nodeKernelsB* submodel

The *nodeKernelsB* submodel, whose structure is depicted in Figure 10.14, models all the Node Kernel regions of the CAN bus. Thus, its role is analogous to the one of the *nodeKernelsT* submodel of CANcentrate.

As in CANcentrate, since the Node Kernel region is exclusively constituted by

the Node Core entity, the failure rate of the activity *nkFailure* is calculated taking into account the failure rate of this entity only.

$$okNodeKernels \rightarrow Mark() \cdot nodeCoreFRate$$

The activity *nkFailure* has three cases. The first one corresponds to a Node Kernel failure that manifests in a way that is beyond our fault model; whereas the other two cases represent a Node Kernel failure that is within that fault model. Specifically, the second one models the situation in which the fault of the Node Kernel manifests as an exclusion failure, whereas the third case represents the situation in which the Node Kernel fault can lead to a blocking failure.

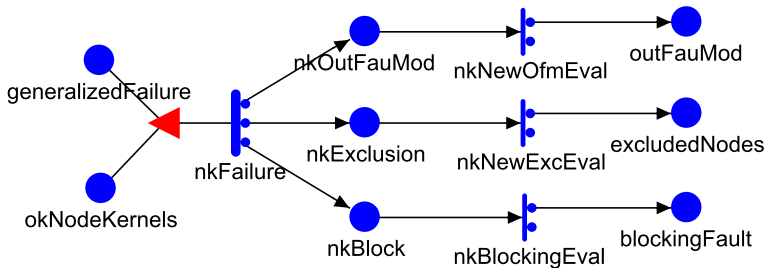

Figure 10.14: nodeKernelsB submodel

The activity *nkFailure* selects the first case with probability *nodeCoreOfmProp*, since this is the proportion with which a Node Core presents an ofm failure. Regarding the proportions of the second and third cases of *nkFailure*, notice that a Node Kernel exhibits an exclusion failure when it compels its node to be stuck-at-recessive, whereas it exhibits a potential blocking failure when it can lead its node to transmit a stuck-at-dominant or a bit-flipping stream. Thus, the second case is selected with proportion *nodeCoreStrProp* and the third one with *nodeCoreStdProp*+ *nodeCoreFlipProp*. As pointed out in Section 10.7.1, although we assume that it is almost impossible that a Node Kernel failure leads its node to transmit a stuck-at-dominant or a bit-flipping stream, the structure of our models still contemplates this possibility.

If the first case of the activity *nkFailureMode* is selected, the token is transferred to the place *nkOutFauMod* and the instantaneous activity *nkNewOfmEval* becomes enabled. This activity evaluates whether or not the Node Kernel that has failed belongs to a node that was already prevented from communicating. Specifically, the node is already prevented from communicating if its Node Connection region

had already suffered from an exclusion failure (a stuck-at-recessive fault). If this is what actually happened, the activity *nkNewOfmEval* chooses its second case and the model does not perform any further action (the case is unconnected). This is because the Node Connection is the interface between the Node Kernel and the medium and, hence, if this interface is stuck-at-recessive, the node will continue transmitting a stuck-at-recessive stream to the medium independently of what the Node Kernel tries to transmit. Otherwise, if the node to which the Node Kernel belongs was not prevented from communicating, the activity *nkNewOfmEval* selects its first case and it transfers the token from the place *nkNewOfmEval* to *outFauMod*, which compels *CANBusFaiEval* to diagnose the overall system as faulty.

In order to calculate the proportions with which the activity *nkNewOfmEval* selects its cases, we followed a strategy similar to the one we used for calculating the proportions with which a Node Kernel (or a Node Connection) region is placed in an already faulty branch in CANcentrate (see Section 10.7.1). More specifically, the proportion with which the activity *nkNewOfmEval* selects its first case is calculated by dividing the number of non-faulty Node Kernels that were placed at nodes that were not prevented from communicating, by the total number of Node Kernels that were not faulty before the fault occurred. This expression is:

$$\frac{numNodes - excludedNodes \rightarrow Mark()}{kNodeKernels \rightarrow Mark() + 1}$$

where *numNodes* is the parameter that specifies the number of nodes connected to the CAN bus. The place *excludedNodes* is shared by the *nodeKernelsB*, the *nodeConnsB* and the *CANBusFaiEval* submodels, and it represents the number of nodes that have been prevented from communicating so far.

In order to better understand how the above expression is obtained, it is necessary to take into account the following aspects. First, we know that a Node Kernel can be placed either in a node that is prevented from communicating or in a node that is not, but that a faulty Node Kernel always belongs to a node that is prevented from communicating. Thus, the number of non-faulty Node Kernels that are placed at nodes that are not prevented from communicating can be calculated by subtracting $excludedNodes \rightarrow Mark()$ from the total number of nodes: *numNodes*. Second, it is noteworthy that when the activity *nkNewOfmEval* has to select one of the cases, the marking of place *okNodeKernels* has been already decreased by the activity *nkFailure*. Thus, the number of Node Kernels that were not faulty before the fault occurred is not $okNodeKernels \rightarrow Mark()$ but $okNodeKernels \rightarrow Mark() + 1$.

As concerns the proportion of the second case of the activity *nkNewOfmEval*, it is obtained by dividing the number of non-faulty Node Kernels that were placed

at nodes that were already prevented from communicating, by the total number of Node Kernels that were not faulty before the fault occurred. Thus, the resulting expression for the proportion of the first case is:

$$\frac{(okNodeKernels \rightarrow Mark() + 1 - (numNodes - excludedNodes \rightarrow Mark()))}{(okNodeKernels \rightarrow Mark() + 1)}$$

Notice that the number of non-faulty Node Kernels that were placed at nodes that were already prevented from communicating is calculated by subtracting *numNodes* $-$*excludedNodes* $\rightarrow$ *Mark()* from the number of nodes that were not faulty before the fault occurred.

Up to this point, we have discussed what is modelled when the activity *nkFailureMode* selects the case that represents an ofm failure; but it is still necessary to describe the actions the model performs when this activity decides that the Node Kernel fails exhibiting an exclusion failure or a potential blocking failure, i.e. when it selects its second and third cases respectively. Basically, what the model does for each one of these two cases is analogous to what we have explained for the first case of *nkFailureMode*. When the second case is chosen, the instantaneous activity *nkNewExcEval* decides if the node to which the Node Kernel belongs was already prevented from communicating. For this purpose, it uses the same two expressions just explained above. If affirmative, *nkNewExcEval* increases the marking of *excludedNodes* in one unit in order to reflect that a new node is prevented from communicating. Otherwise, the model does not need to perform any further action and, thus, the second case of *nkNewExcEval* is left unconnected.

Finally, the activity *nkBlockingEval* also decides whether or not the node was already prevented from communicating, using for this decision the same expressions as *nkNewOfmEval* and *nkNewExcEval*. Its first case sets a token in the place *blockingFault*, whereas its second one is left unconnected. The place *blockingFault* is shared among all submodels and indicates that the communication is completely blocked. As will be explained later, this automatically leads the *CANBusFaiEval* submodel to diagnose that the overall system is faulty.

10.8.2 *nodeConnsB* submodel

The *nodeConnsB* submodel represents all the connection regions of the CAN bus. As can be seen in Figure 10.15, it is very similar to the *nodeKernelsB* submodel. There are two main differences between these two submodels. The first difference is that the Node Connection region includes several entities and, thus, the failure

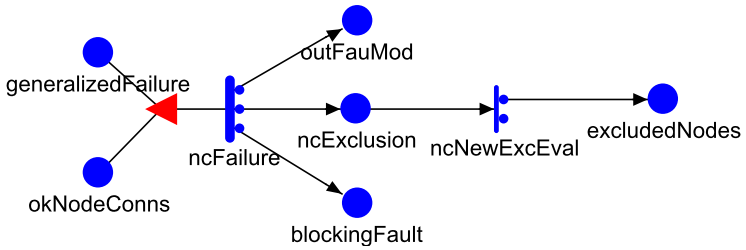

Figure 10.15: nodeConnsB submodel

rates and the failure mode proportions of all these entities must be considered when calculating the failure rate and the cases' proportions of the activity *ncFailure*.

More specifically, the failure rate of the Time To Failure distribution modelled by *ncFailure* is:

$$okNodeConns \rightarrow Mark() \cdot (ctrlFRate + nodeIOFRate)$$

where *ctrlFRate* and *nodeIOFRate* are the failure rates of the Controller and the Node IO entities respectively. In this sense, notice again that, conversely to the Node Connection region of CANcentrate, a Node Connection of the CAN bus only has one Node IO entity, which corresponds to the only transceiver a node needs to connect to the bus.

Regarding the cases of the activity *ncFailure*, the first one models an ofm Node Connection failure, whereas the second and the third cases respectively represent a Node Connection that exhibits an exclusion failure (a stuck-at-recessive) and a blocking failure (stuck-at-dominant or bit-flipping).

The proportion of the first case is calculated following Equation 10.10:

$$ctrlOfmProp \cdot ctrlFRate + nodeIOOfmProp \cdot nodeIOFRate$$

Whereas the proportions of the second and third cases adhere to Equations 10.11, in order to take into account the capacity of the CAN controller to isolate errors generated by faults affecting itself and the Node IO entity placed within its Node Connection region (see Section 10.4.5 for a detailed explanation of the error-containment capabilities of the CAN controller). The second case's proportion is calculated following the first one of Equations 10.11 as:

$$
\begin{aligned}
&\mathit{nodeIOStrProp} \cdot \mathit{nodeIOInFauProp} \cdot \mathit{nodeIOFRate} + \\
&\mathit{nodeIOStdProp} \cdot \mathit{nodeIOInFauProp} \cdot \mathit{nodeIOFRate} + \\
&\mathit{nodeIOFlipProp} \cdot \mathit{nodeIOInFauProp} \cdot \mathit{ctrlFlipCov} \cdot \mathit{nodeIOFRate} + \\
&\\
&\mathit{ctrlStrProp} \cdot \mathit{ctrlFRate} + \\
&\mathit{ctrlStdProp} \cdot \mathit{ctrlFRate} \cdot (1.0 \cdot \mathit{ctrlItselfIsoCov}) + \\
&\mathit{ctrlFlipProp} \cdot \mathit{ctrlFRate} \cdot (\mathit{ctrlFlipCov} \cdot \mathit{ctrlItselfIsoCov}) + \\
&\\
&\mathit{nodeIOStrProp} \cdot (1.0 - \mathit{nodeIOInFauProp}) \cdot \mathit{nodeIOFRate})
\end{aligned}
$$

It is noteworthy that conversely to what is done in the *nodeConnsT* submodel of CANcentrate (see Section 10.7.2), stuck-at-recessive Node Connection failures are not modelled together with stuck-at-dominant Node Connection ones by means of the same case. This is because a CAN bus does not include a hub that is able to isolate stuck-at-dominant faults. In fact, since stuck-at-dominant Node Connections are blocking failures, they are modelled together with bit-flipping Node Connection's ones by means of the third case of the activity *ncFailure*, as will be explained later on in this section.

We have written the above expression in separated blocks in order to put emphasis on some aspects related to the capacity of the CAN controller to contain errors. The first block of lines represents the faults of the Node IO entity that do not deliver errors to the medium, but to the CAN controller reception port. The percentage of Node IO faults that do not deliver errors to the medium is given by the parameter *nodeIOInFauProp*. Notice that this block also reflects the fact that the CAN controller always diagnoses faults that lead it to receive errors in the form of stuck-at streams, but that it is only able to diagnose faults that generate bit-flipping errors with a coverage of *ctrlFlipCov*. The second block of the expression corresponds to faults that affect the internals of the CAN controller. It reflects the error-containment capacities of the CAN controller to treat these faults. First, notice that the CAN controller successfully diagnoses the stuck-at-dominant and the bit-flipping fault with a coverage of 1.0 and *ctrlFlipCov* respectively. Second, the CAN controller is able to isolate faults affecting its internals with a probability of *ctrlItselfIsoCov*. The third block of the expression corresponds to the faults of the Node IO entity that deliver errors to the medium in the form of a stuck-at-recessive stream.

The third case of the activity *ncFailure* models the situation in which the Node

Connection region fails exhibiting a blocking failure, i.e. by issuing a stuck-at-dominant or a bit-flipping stream to the medium. It is chosen taking into account the second and the third ones of Equations 10.11:

$$
\begin{aligned}
&\mathit{nodeIOFlipProp} \cdot \mathit{nodeIOInFauProp} \cdot (1.0 - \mathit{ctrlFlipCov}) \cdot \mathit{nodeIOFRate} + \\
&\\
&\mathit{ctrlStdProp} \cdot \mathit{ctrlFRate} \cdot (1.0 - 1.0 \cdot \mathit{ctrlItselfIsoCov}) + \\
&\mathit{ctrlFlipProp} \cdot \mathit{ctrlFRate} \cdot (1.0 - \mathit{ctrlFlipCov} \cdot \mathit{ctrlItselfIsoCov}) + \\
&\\
&\mathit{nodeIOStdProp} \cdot (1.0 - \mathit{nodeIOInFauProp}) \cdot \mathit{nodeIOFRate} + \\
&\mathit{nodeIOFlipProp} \cdot (1.0 - \mathit{nodeIOInFauProp}) \cdot \mathit{nodeIOFRate}
\end{aligned}
$$

The first block (line) of the above expression represents the faults of the Node IO entity that deliver bit-flipping errors to the reception port of the CAN controller and that, additionally, are not contained by that controller. The second block corresponds to the CAN controller's internal faults that the CAN controller does not successfully diagnose or does not isolate. The last blocks reflects the faults that lead the Node IO entity to deliver a stuck-at-dominant or a bit-flipping stream to the medium.

Up to this point we have explained the difference between the *nodeKernelsB* and the *nodeConnsB* submodels as concerns the way in which the failure rate and the failure mode proportions are calculated. But, as pointed out before, there is another difference between these submodels. As explained in Section 10.8.1, when a Node Kernel fails, i.e. when the activity *nkFailure* fires, the *nodeKernelsB* submodel evaluates whether or not the Node Connection of its node had already suffered from an exclusion failure. If the Node Connection had already suffered from such a failure, the *nodeKernelsB* submodel does not perform any further action to model the propagation of the errors the Node Kernel generates. This is because the value of the Node Connection's output prevails over the output of the Node Kernel. In contrast, notice that *nodeConnsB* does not always need to check whether or not the Node Kernel corresponding to the Node Connection had already suffered from an exclusion failure. Specifically, *nodeConnsB* does not carry out this check when the Node Connection exhibits an ofm or a blocking failure. This is because the errors a faulty Node Connection generates always propagate through the bus, independently of whether or not its corresponding Node Kernel has already failed. The only situation in which *nodeConnsB* checks if the Node Kernel had already suffered from an exclusion failure is when the Node Connection exhibits an exclusion failure. This check is done by the activity *ncNewExcEval*. In particular, if the

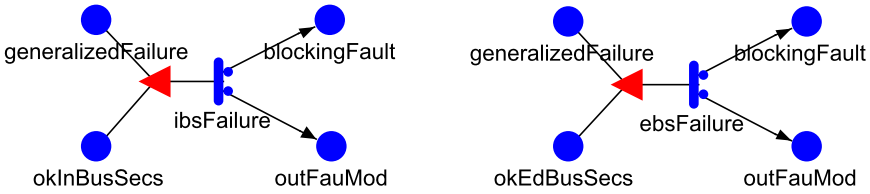

Figure 10.16: inBusSections and edBusSections submodels

Node Kernel had already suffered from such a failure, then *ncNewExcEval* does not add a token to *excludedNodes*, since the exclusion of the node to which the Node Connection belongs to was already reflected by the *nodeKernelsB* submodel when the Node Kernel failed.

More specifically, the proportions with which the activity *ncNewExcEval* chooses its two cases are analogous to the ones with which the activity *nkNewExcEval* selects its first and second cases (see Section 10.8.1):

$$\frac{(numNodes - excludedNodes \rightarrow Mark())}{(okNodeConns \rightarrow Mark() + 1)}$$

and

$$\frac{(okNodeConns \rightarrow Mark() + 1 - (numNodes - excludedNodes \rightarrow Mark()))}{(okNodeConns \rightarrow Mark() + 1)}$$

10.8.3 *inBusSections* and *edBusSections* submodels

Submodels *inBusSections* and *edBusSections* respectively model all the Internal Bus Sections and the two Edge Bus Sections of the CAN bus. Both submodels have exactly the same structure, as shown in Figure 10.16.

The marking of the places *okInBusSecs* and *okEdBusSecs* respectively represent the number of Internal and Edge Bus Section regions that are not faulty. The initial marking of *okInBusSecs* is the value of the parameter *numInSections*, which must be equal to $numNodes - 2$, where *numNodes* is the parameter that specifies the number of nodes connected to the CAN bus. In contrast, the initial marking of the place *okEdBusSecs* is 2.

When the activity *ibsFailure* or the activity *ebsFailure* fire, a token is erased from *okInBusSecs* and *okEdBusSecs* respectively. The failure rate of the activity *ibsFailure* is calculated as $busAttchFRate \cdot inBusSecsOk \rightarrow Mark()$. However, since an edge region includes an Attachment entity and a Termination entity, the failure rate of the activity *ebsFailure* takes into account the failure rate of both of them: $(busAttchFRate + termFRate) \cdot edBusSecsOk \rightarrow Mark()$.

The two cases of these activities respectively represent the situation in which the failure mode the bus section exhibits is included in our fault model and the situation in which it is not. Notice that any bus section failure leads to the failure of the overall communication subsystem (and the whole system), since the medium of the bus becomes faulty. This means that to differentiate between these two cases is not actually necessary. However, we preferred to differentiate between the failure modes that are included in our fault model and those that are not because, in this way, we can better integrate the *inBusSections* and *edBusSections* submodels with the rest of submodels.

Activity *ibsFailure* selects the first and the second cases with proportions: $1.0 - busAttchOfmProp$ and $busAttchOfmProp$. In contrast, due to the fact that an Edge Bus Section includes an Attachment and a Termination entities, the activity *ebsFailure* chooses these cases with proportions:

$$1.0 - (busAttchOfmProp \cdot busAttchFRate + termOfmProp \cdot termFRate)$$

and

$$busAttchOfmProp \cdot busAttchFRate + termOfmProp \cdot termFRate$$

10.8.4 *CANbusFaiEval* submodel

As already pointed out, the *CANbusFaiEval* submodel is the responsible for diagnosing when the overall system is faulty.

As shown in Figure 10.17, it shares the places *outFauMod*, *excludedNodes* and *blockingFault* with the regions submodels to become aware of what kind of faults have happened in the system. On the one hand, when a regions submodel sets a token in the place *oufFauMod* or in *blockingFault*, the corresponding instantaneous activity of the *CANbusFaiEval* submodel immediately transfers that token to the place *generalizedFailure*, thereby indicating that the system is faulty.

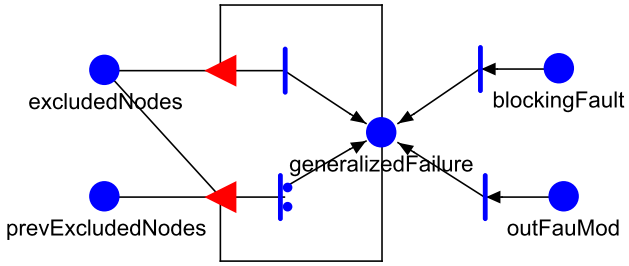

Figure 10.17: CANbusFaiEval submodel

On the other hand, the *CANbusFaiEval* submodel writes a token on the place *generalizedFailure* whenever the marking of the place *excludedNodes* exceeds the value of the parameter *kSevere*. As with the model of CANcentrate, it is possible to measure the NFT/AR as well as different degrees of FT/AR$_k$, by specifying the appropriate value for the parameter *kSevere*. For instance, if *kSevere* $= 0$, then the probability of having a token in *generalizedFailure* is the NFT/AR of a system that relies on the CAN bus.

Moreover, *CANbusFaiEval* also models the coverage with which an FT/A systems actually accepts or tolerates the failure or disconnection of a given node, provided that it can do so. For this purpose, it includes the place *prevExcludedNodes*, the activity *sysFauTolCoverage* and the corresponding input gate. These elements are analogous to the ones *CANcentrateFaiEval* includes for carrying out the same evaluation when the FT/A system relies on CANcentrate. Thus, for more details concerning the way in which *CANbusFaiEval* uses these elements, please refer to Section 10.7.5.

Finally notice that the *CANbusFaiEval* submodel shares the place *generalizedFailure* with all the other submodels. When a token is set at this place each regions submodel stops evolving. As in the case of CANcentrate, each regions submodel stops by using an input gate connected to the activity that represents the Time To Failure distribution of the regions it represents.

10.9 Quantitative assessment

As explained in Section 10.2, in order to quantify the reliability of a system that relies on CAN and of an equivalent system that relies on CANcentrate, we use two metrics: the *non-fault-tolerant/accepting system reliability* (NFT/AR) and the *fault-tolerant/accepting system reliability* (FT/AR$_k$) for $k = 1$. The first one

is devoted to measuring the reliability of a *non-fault-tolerant/accepting system* (NFT/AR), whereas the second aims at calculating the reliability of a *fault-tolerant / accepting system* that is robust to the failure or disconnection of at most 1 out of N nodes. As already said in Section 10.2, we choose this value for k, since it is the case that least reflects the benefits of CANcentrate for FT/AR systems.

In particular, we analyze the NFT/AR and the FT/AR$_1$ throughout the interval of time during which they are equal or greater than a certain value. This is because a reliable application normally requires that the system it relies on guarantees a specific and well-defined degree of reliability. In this sense, as already explained in Section 10.1, we consider just as a reference the reliability requirements of the less demanding x-by-wire applications in cars. Specifically, we analyze the NFT/AR and the FT/AR$_k$ as long as they are ≥ 0.99999, which is the value of reliability required by a throttle-by-wire system [MK05].

In order to compare the reliability that can be achieved when using CAN and CANcentrate, we assess the maximum amount of time during which a system that relies on them exhibits a reliability (NFT/AR and FT/AR$_1$) equal or greater than the above-indicated reliability degree. We refer to this amount of time as the *mission time*. This concept is defined in [MK05] as the expected length of operation of one mission for a system. Just as a reference and following the example cited before, notice that the least demanding x-by-wire applications require a reliability ≥ 0.99999 during a mission time of 10 hours [MK05].

It is also noteworthy that, in principle, we compare the reliability that can be achieved with CAN and CANcentrate considering the default values we have proposed as reasonable for our models' parameters. These parameters and their default values are described in Tables 10.4, 10.5 and 10.6. However, in order to study how different aspects influence the reliability of a system relying on both networks, we perform additional sensitivity analyses with respect to some of these parameters, such as the number of nodes, the fault-tolerance coverages, the hub failure rate, etc. Specifically, those additional analyses focus on the FT/AR$_1$, since NFT/A systems cannot actually benefit from the potential advantages of a simplex star topology.

Each one of these sensitivity analyses is basically conducted with respect to one parameter only. However, we consider 3 and 15 nodes for each one of these analyses (except for the one that assesses the reliability with respect to the number of nodes itself). We selected 3 as it is the minimum number of nodes needed to tolerate a fault that prevents one of them from communicating. Additionally, we consider 15 nodes, since this is the typical average number of nodes of an in-vehicle CAN subnetwork, e.g. in the body network of a vehicle [BHN07].

Finally notice that as stated before, all dependability parameters were deter-

mined, and options taken, with specific care not to favor CANcentrate. Moreover, we have not even considered some potential dependability advantages of stars, such as for example its resilience to spatial proximity failures. This means that the results herein presented are likely to be lower bounds to the system reliability achievable with CANcentrate.

10.9.1 NFT/AR vs number of nodes

As explained in Section 10.2, since CANcentrate includes more hardware than a CAN bus to interconnect the same number or nodes, it is expected that CANcentrate reduces the system NFT/AR when compared with the bus. Figure 10.18 quantitatively corroborates this intuitive idea. It shows the NFT/AR of a system that relies on CAN and of an equivalent system that relies on CANcentrate for different number of nodes (from 3 to 20). As already said, 3 is the minimum number of nodes needed to tolerate a fault that prevents one of them from communicating. Besides this minimum, we consider a maximum of 20, which is the typical maximum number of nodes of an in-vehicle CAN subnetwork [Pau04]. For instance, [BHN07] considers up to 20 nodes[3] when presenting NETCARBENCH, one of the most recent benchmarks devoted to assessing in-vehicle communication networks.

As depicted in Figure 10.18, the CAN bus is more reliable than CANcentrate for any number of nodes. This is an expected result given the extra hardware of CANcentrate with respect to CAN and the fact that an NFT/A system cannot benefit from the error-containment provided by a simplex star topology. If we focus on the achievable mission times, we see that the reduction provoked by CANcentrate remains almost constant for any number of nodes (around the 35%). For instance, for 3 nodes, a system that relies on CAN and an equivalent system that relies on CANcentrate presents an NFT/AR ≥ 0.99999 during 0.63 and 0.41 hours respectively.

Finally, Figure 10.18 also indicates that a system that relies on CAN or CANcentrate only achieves some minutes of mission time; a value that is quite far from the 10 hours required by the least demanding x-by-wire systems. Nevertheless, it is worth noting that the system does not show this poor reliability as a consequence of using CAN or CANcentrate as the underlying communication infrastructure. In fact, the NFT/AR does not only reflect the dependability properties of the network, but also the reliability of the nodes themselves, which are the least reliable

[3]To be more specific, a node in an in-vehicle network is basically constituted by an *Electronic Control Unit* (ECU)

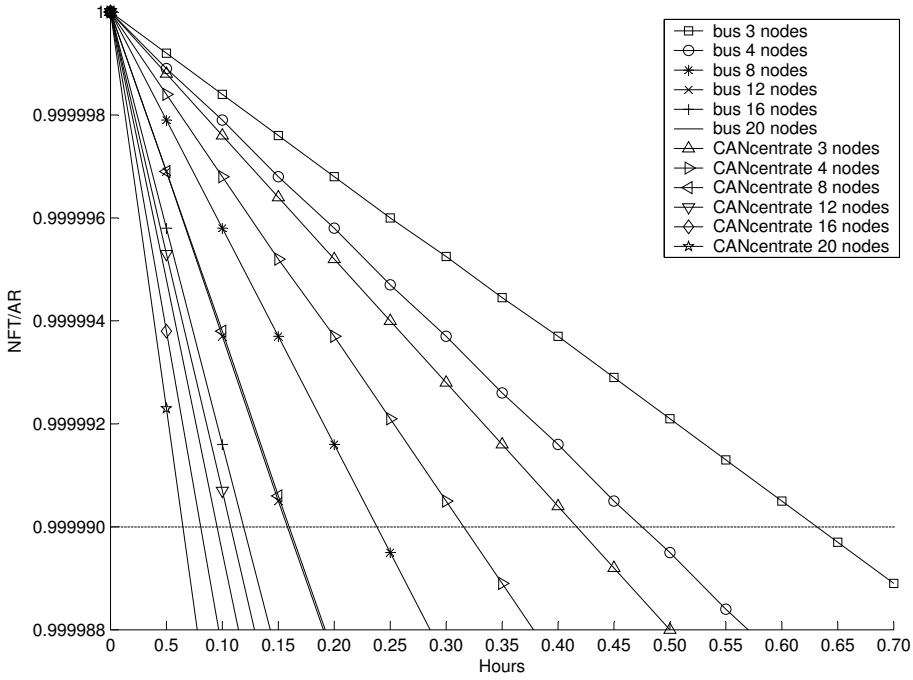

Figure 10.18: NFT/AR vs number of nodes

elements of the system. This means that a system appropriate for x-by-wire applications should be provided with additional mechanisms or better components to increase the reliability of the nodes themselves and/or of the hub. Another option would be to include mechanisms at the system level in order to tolerate the failure or the disconnection of nodes. If this last option is chosen, then the FT/AR$_k$ is the appropriate metric for measuring the reliability of the resultant FT/A system.

10.9.2 FT/AR$_1$ vs number of nodes

Although using CANcentrate implies a loss of system NFT/AR when compared with CAN, its hub can yield error-containment benefits. This is specially valuable for improving the reliability of any system that can continue delivering its service even when not all nodes can communicate, i.e of any FT/A system. As explained before, we use the *fault-tolerant/accepting system reliability* (FT/AR$_k$), i.e. the probability of not suffering a k-severe failure, for measuring the reliability of these systems. More specifically, notice that here and in the rest of sensitivity analyses, we assess the FT/AR for $k = 1$ (FT/AR$_1$), i.e. we measure the reliability of systems

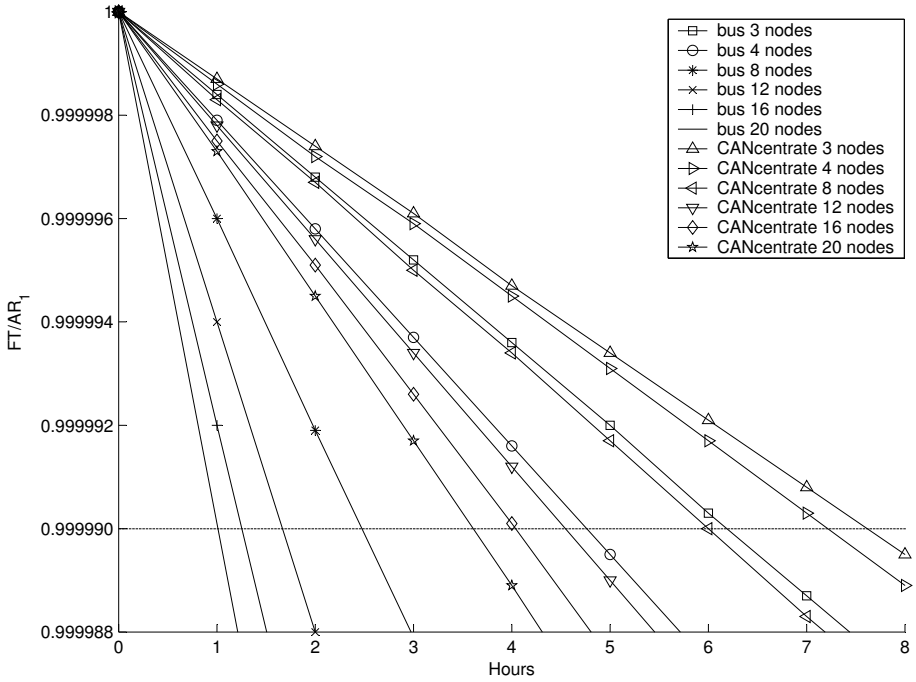

Figure 10.19: FT/AR_1 vs number of nodes

that are robust to the loss of communication with at most 1 out of N nodes. As already said in Section 10.2, we choose this value for k, since it is the case that least reflects the benefits of CANcentrate.

Figure 10.19 depicts the FT/AR_1 of a system relying on CAN and of an equivalent system that relies on CANcentrate for different numbers of nodes. Results are, to some extent, the opposite of what we obtained for the NFT/AR. CANcentrate is better than CAN for any number of nodes. For instance, the mission times yielded by CAN and CANcentrate are, respectively, 6.2 and 7.6 hours for 3 nodes, as well as 1.0 and 3.6 hours for 20. This corroborates that reliability of FT/A systems can be improved by means of the enhanced error-containment of a simplex star. Moreover, a higher number of nodes implies a bigger difference between the mission time achievable with CAN and CANcentrate. For instance, from the just mentioned mission times it can be seen that CANcentrate improves the mission time of CAN around the 22% and the 260% for 3 and 20 nodes respectively.

Notice that these results demonstrate that the CAN bus is much more sensitive than CANcentrate to the number of nodes. Actually, this greater sensitivity of

the CAN bus is also reflected on the fact that, when the number of nodes is high enough, CANcentrate exhibits the same or a greater FT/AR than CAN even when the bus includes fewer nodes than the star. For instance, a system provided with 20 nodes that relies on CANcentrate reaches a higher mission time than a system that includes 8 nodes and that relies on CAN. This means that CANcentrate supports an increase in the number of nodes that can be included in the network while ensuring an FT/AR$_1 \geq 0.99999$ during a given mission time (an increase of the quantity of nodes of more than the 150% in the example just mentioned).

Finally, notice that the mission times provided by CAN and CANcentrate are still quite low when compared with the requirements of the least demanding x-by-wire systems. However, as it was explained in the previous section for the case of the NFT/AR, this result does not mean that the reliability of these two networks cannot be further improved. Additional fault-tolerance mechanisms, can be included to enhance the nodes and the hub reliability, e.g. the nodes and the hub could be provided with internal redundancy. Moreover, the reliability of the components can be also improved, since we are currently considering the lowest quality for them, i.e. a commercial quality level (see Section 10.4.3).

10.9.3 FT/AR$_1$ vs system fault-tolerance coverage

As explained in Sections 2.4 and 10.4.5, the dependability of a system is extremely sensitive to the coverage of its fault-tolerance mechanisms. Therefore, it is important to assess how these coverages affect the reliability of a system that relies on CAN and of an equivalent system that relies on CANcentrate.

In particular, as indicated in Section 10.4.5, we have to differentiate between the fault-tolerance capacities of the system itself and the fault-tolerance mechanisms of the communication subsystem (CAN or CANcentrate) it relies on. Current section addresses the coverage of the firstly mentioned capacities, whereas the coverages of the other ones will be addressed later in Sections 10.9.4 and 10.9.5.

When we talk about the fault-tolerance capacities of the system itself we are referring to the ability of an FT/A system to accept or tolerate the failure or the disconnection of a node. More specifically, in order to characterize this ability we proposed a fault-tolerance coverage we call *sysFauTolCov*. This coverage was introduced in Section 10.4.5 and we formally defined it as the probability with which an FT/A system accepts or tolerates the failure or disconnection of a new node, provided that the number of nodes that have failed or that have become disconnected so far (including the new one) does not exceed the value of k.

Notice that we are considering FT/A systems that are able to accept the failure

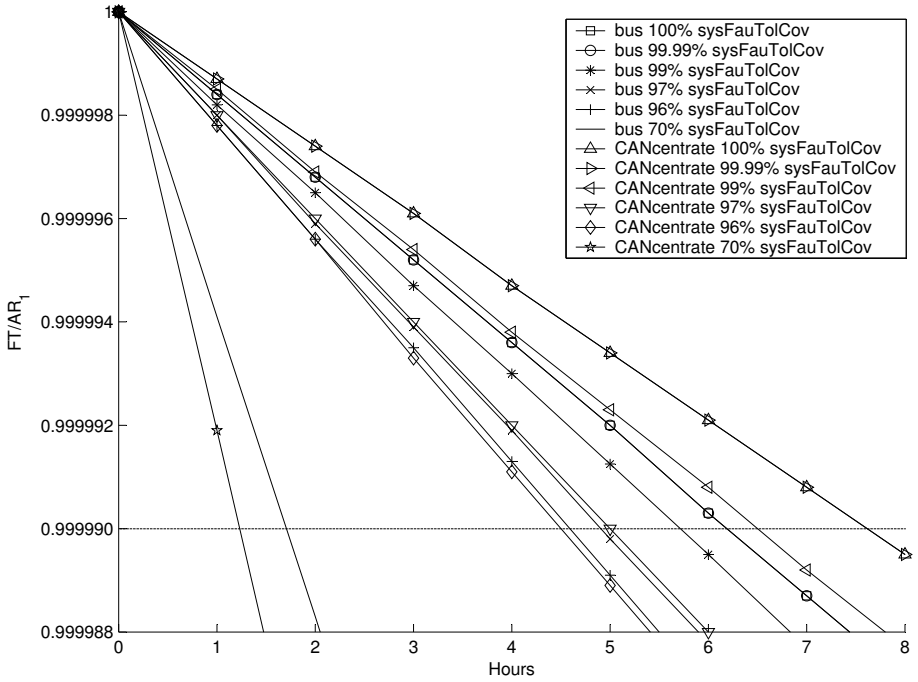

Figure 10.20: FT/AR$_1$ vs system' fault-tolerance coverage for 3 nodes

or disconnection of up to $k = 1$ of N nodes. Therefore, *sysFauTolCov* can be understood here as the probability with which an FT/A system actually accepts or tolerates that one node fails or becomes disconnected, provided that this FT/A system can potentially do that.

We proposed a default value for this coverage of 100%. As also explained in Section 10.4.5, this value is the one that appropriately characterizes all FT/A systems that intrinsically do not require that all nodes operate and communicate. Moreover, this default value is also well suited for an important quantity of FT/A systems that deliberately include mechanisms to tolerate the failure or disconnection of a given number of nodes. This is the case of highly-reliable systems, which explicitly include fault-tolerance mechanisms whose coverages range from 99% up to virtually 100%.

However, there are FT/A systems that do not achieve such a high fault-tolerance coverage and, hence, it is necessary to quantify what is the minimum value of this coverage that should be guaranteed in order to improve the system reliability by means of a simplex star topology such as CANcentrate.

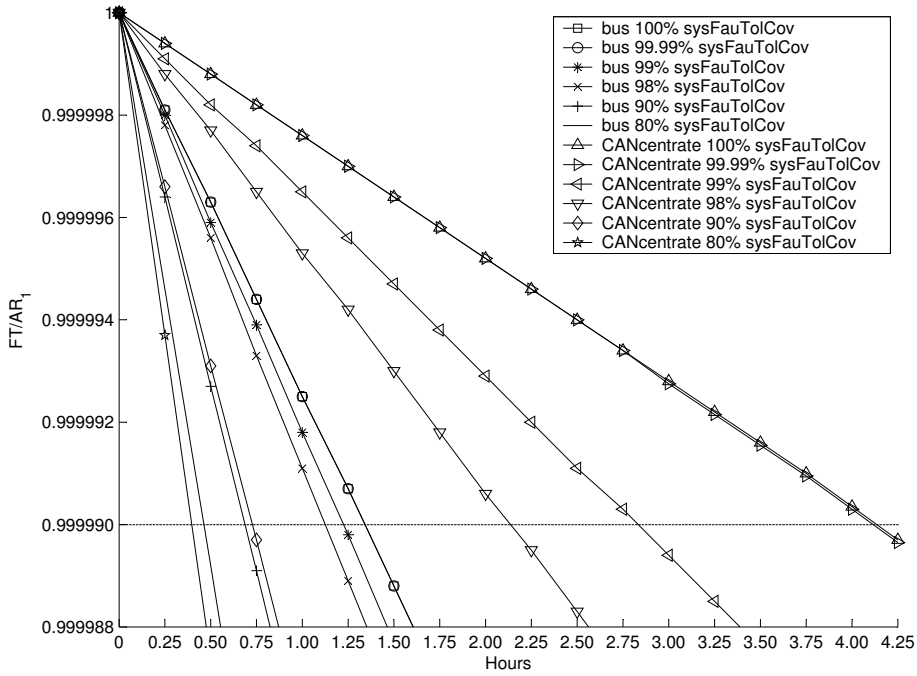

Figure 10.21: FT/AR_1 vs system' fault-tolerance coverage for 15 nodes

Figure 10.20 shows the FT/AR_1 of a system relying on a CAN bus and on CANcentrate for different values of *sysFauTolCov* when 3 nodes are considered. The first aspect that can be seen is that the FT/AR_1 is very sensitive to this coverage. However, it is important to highlight that it is not necessary to attain a *sysFauTolCov* greater than 99.99, since the FT/AR_1 that is obtained with this value is equal to the FT/AR_1 achieved with a *sysFauTolCov* of 100%. This last result means that a simplex star topology achieves its maximum potential for a typical highly-reliable system such as, for example, the *Self-Repairing Flight Control System* (SRFCS) of a military aircraft, which is able to achieve fault-tolerance coverages of the order of 99.99% and 99.9992% [Wu02].

Another important result observed in Figure 10.20 is that the minimum *sysFauTolCov* that must be accomplished in order to take profit from the error-containment capabilities of CANcentrate is around 97%. Certainly, this threshold is quite lower than the typical fault-tolerance coverage of highly-reliable systems. But it is still high enough to compel designers to devise sufficiently efficient fault-tolerance mechanisms, if they want to use a simplex star topology for improving the reliability of an FT/A system.

These results become slightly different if we consider a bigger system, e.g. a system that includes 15 nodes. As can be seen in Figure 10.21, the minimum *sysFauTolCov* for which CANcentrate improves the system reliability (FT/AR$_1$) decreases down to the 90%. This low threshold indicates that when an average number of nodes is considered, a simplex star topology improves the reliability of not only highly-reliable FT/A systems, but also of almost every FT/A system.

10.9.4 FT/AR$_1$ vs fail-silent node proportion

Previous section analyzed the FT/AR$_1$ of the system with respect to the coverage of the fault-tolerance mechanisms of the system itself. The current and the following section assess the FT/AR$_1$ of the system depending on the coverages of the fault-tolerance mechanisms of the communication subsystem. More specifically, they evaluate the FT/AR$_1$ with respect to the effectiveness of the error-containment mechanisms implemented at each CAN controller and of the mechanisms the hub is provided with.

As concerns the error-containment mechanisms of the CAN controller, notice again that they are devoted to containing errors generated by faults occurring at the node it is located at, i.e. ocurring at the Controller itself and at the Node IOs. Moreover, the CAN controller can also contain errors it receives from faults affecting its local connection to the star (see Section 10.4.5). In this section we focus on the ability of the CAN controller to contain errors generated by faults happening at the node itself (at the Controller and at the Node IOs entities), so that, in the rest of this section, we refer to the capacity of the CAN controller to contain errors as the ability of the CAN node to fail in a fail-silent way, i.e. to fail by transmitting a stuck-at-recessive stream. By extension, we refer to the analysis carried out in this section as the sensitivity analysis with respect to the *fail-silent node* (FS) proportion.

In Section 10.4.5 we characterized the error-containment capacity of the CAN controller by means of several coverages: *nodeIOInFauProp*, *ctrlItselfIsoCov* and *ctrlFlipCov* (see Table 10.4, which gathers the default values we have proposed for them). Thus, it is possible to obtain different values of the FS proportion by varying the value of these coverages. This strategy is depicted in Table 10.1, which shows the correspondence between different values of the CAN controller's error-containment coverages and the FS proportion. It is noteworthy that the default values of the CAN controller's error-containment coverages imply that, so far, we were considering that the proportion with which the node fails exhibiting a fail-silent behavior is around 91.4%. This proportion is maybe slightly overestimated, and thus biased in favor of the CAN bus.

ctrlItselfIsoCov	ctrlFlipCov	nodeIOInFauProp	FS (%)
0.00	0.00	0.00	83.30
0.40	0.77	0.50	90.00
0.50	0.85	0.50	91.00
0.60	0.90	0.50	92.00
0.70	0.95	0.50	93.00
0.80	0.99	0.50	94.00
0.92	1.00	0.50	95.00
1.00	1.00	0.54	96.00
1.00	1.00	0.65	97.00
1.00	1.00	0.77	98.00
1.00	1.00	0.89	99.00
1.00	1.00	1.00	100.0

Table 10.1: FS proportion as a function of the node's error-containment coverages

Figure 10.22 depicts the sensitivity of the FT/AR_1 with respect to the FS proportion when 3 nodes are considered. Note, again, that this is the number of nodes most unfavorable to CANcentrate whose relative advantages over the bus get stronger with increasing number of nodes. The legend specifies (as a percentage) the approximate value of the FS proportion.

The figure shows that when the FS proportion is 100%, the CAN bus is better than CANcentrate. This was obviously expected since with such a proportion the fault-treatment mechanisms of the hub become irrelevant. However, it can be seen that a drop in the proportion of silent faults dramatically affects the FT/AR_1 of a system that relies on the CAN bus, but not the FT/AR_1 of a system relying on CANcentrate. Specifically, Figure 10.22 shows that a CAN-based system reaches a lower mission time that an equivalent system that relies on CANcentrate when the FS proportion is equal or lower than the 93% approximately.

In fact, Figure 10.22 shows that the FT/AR_1 of a CANcentrate-based system seems to be almost insensitive to the FS proportion. For instance, a CAN-based system achieves a mission time of 36.0 and 5.4 hours when the FS proportion is of the 100% and of the 90% respectively, whereas a CANcentrate-based system achieves 7.7 and 7.5 hours respectively for these same FS proportions. This means that when the FS proportion is decreased from the 100% to the 90%, the mission time is reduced around the 85% with CAN and only around the 3% with CANcentrate. The lower sensitivity of the FT/AR_1 of a CANcentrate-based system when compared with an equivalent CAN-based one relies on the fact that the hub is able to isolate faulty nodes that are not able to isolate themselves, whereas in the CAN

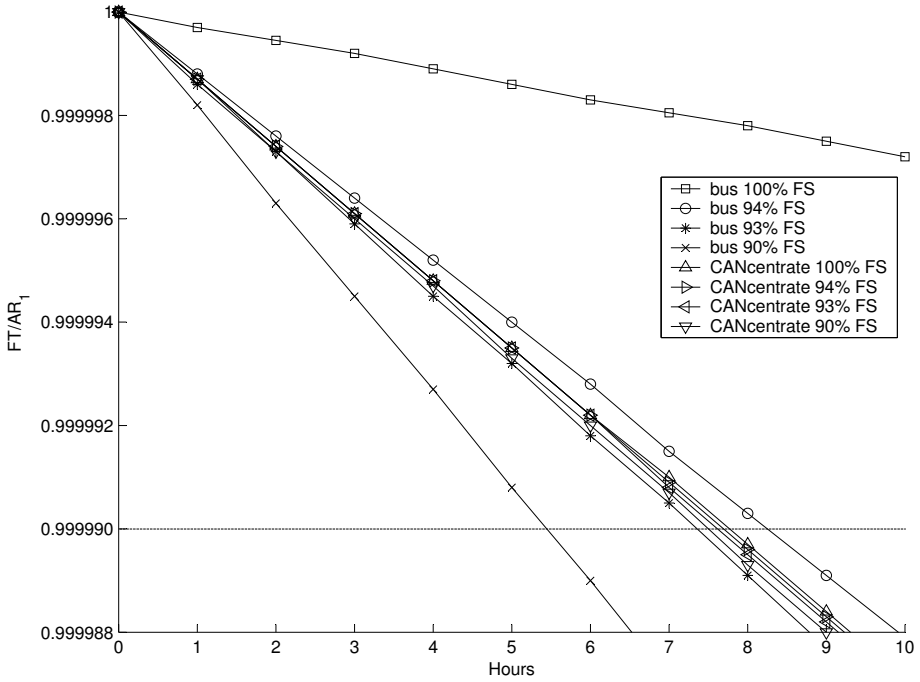

Figure 10.22: FT/AR$_1$ vs fail-silent node proportion for 3 nodes

bus a non fail-silent node inevitably provokes a severe failure.

Figure 10.23 depicts the FT/AR$_1$ of a system that relies on CAN and of an equivalent system relying on CANcentrate with respect to the FS proportion, but considering 15 nodes. As indicated before, 15 is the typical number of nodes of an in-vehicle CAN subnetwork. In general terms, the results are equivalent to those depicted in the previous figure. On the one hand, Figure 10.23 further demonstrates that the FT/AR$_1$ of a CAN-based system is more sensitive than the FT/AR$_1$ of an equivalent CANcentrate-based one, even when the system includes an average number of nodes. For instance, the achievable mission time of a CAN-based system is around 12.0 and 1.1 hours when the FS proportion is of the 100% and of the 90% respectively, whereas a CANcentrate-based system achieves 4.3 and 3.9 hours respectively. This means that the mission time is reduced around the 91% with CAN and only around the 9% with CANcentrate when the FS proportion is decreased from the 100% to the 90%. Also notice that the sensitivity of FT/AR$_1$ with respect to the FS proportion increases with the number of nodes in both a CAN-based and a CANcentrate-based system (although this increase in terms of sensitivity is slightly bigger when using CANcentrate).

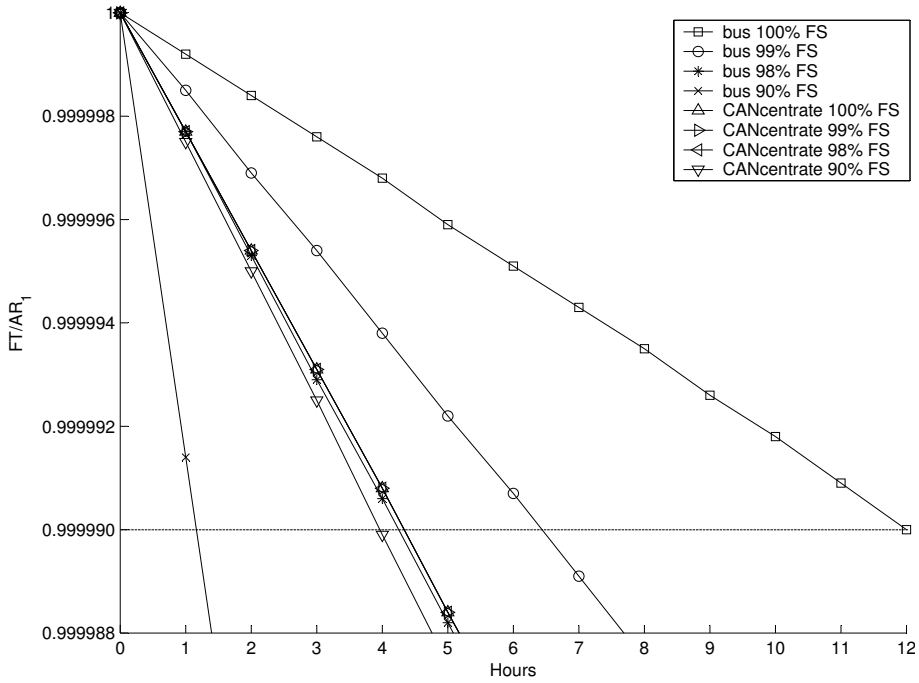

Figure 10.23: FT/AR_1 vs fail-silent node proportion for 15 nodes

On the other hand, Figure 10.23 reinforces the idea about the fact that the benefits that CANcentrate yields in terms of system FT/AR_1 are bigger as the number of nodes grows. In particular, Figure 10.23 shows that, when 15 nodes are considered, it is enough that the FS proportion of nodes drops to around the 98% to turn a CAN-based system worse than an equivalent CANcentrate-based one.

Finally, it is important to highlight that the analyses presented in this section are important not only because the real value of the FS proportion is unknown and, thus, a fair comparison between CAN and CANcentrate requires a sensitivity analysis with respect to it. In contrast, this analysis is also interesting to estimate the relevance of using star topologies in communication protocols in general, depending on the ability of nodes to contain their own errors.

In particular, note that this section's results constitute a preliminary assessment of the system reliability benefits that can be achieved with a CANcentrate star, when compared with a CAN bus where the fail silent behavior of each node is enforced by a dedicated bus guardian. See Section 4.4 for an explanation on the use of bus guardians to improve dependability. Specifically, notice that the above

results are obtained without taking into account the failure rate of the bus guardian in the estimation of the node failure rate. This implies that the reliability we are considering for the nodes is more optimistic as the FS proportion increases, since it is expected that the complexity (and thus the failure rate) of the bus guardian itself increases with its effectiveness. Therefore, a future comparison of CANcentrate and bus guardians should consider this fact in addition to other bus guardian's limitations such as spatial-proximity and common-mode failures.

10.9.5 FT/AR$_1$ vs bit-flipping coverage of the hub

As pointed out above, it is also necessary to assess the sensitivity of the FT/AR$_1$ with respect to the coverage of the error-containment mechanisms the hub of CANcentrate is provided with.

The default values we have considered so far for the coverages that characterize these mechanisms are explained in Section 10.4.5. On the one hand, we consider that the hub can contain a stuck-at stream at any of its ports with a perfect coverage. As already explained in the mentioned section, this assumption is completely realistic and, therefore, there is no need to perform a sensitivity analysis with respect to this coverage. On the other hand, we suppose that the hub can detect and isolate bit-flipping streams at any of its ports with a 95% coverage (see parameter *flipLnkCov* of Table 10.6). However, as also indicated in that section, the real value of this coverage is unknown and a coverage of 95% is maybe underestimated for CANcentrate. In particular, notice again that we assumed the same default value for the coverage with which the CAN controller diagnoses a bit-flipping fault, i.e. the default value of the parameter *ctrlFlipCov* is also 95%. This is detrimental for CANcentrate, because the error-detection and fault-diagnosis mechanisms of the hub are more efficient than the ones included in a CAN controller (see Section 8.3 for more details concerning this issue).

In order to asses how the bit-flipping coverage of the hub determines the achievable FT/AR$_1$ of a system that relies on CANcentrate, we measure the FT/AR$_1$ of such a system for different values of the parameter that specifies this coverage: the *flipLnkCov* (see Table 10.6). We do not only assess the FT/AR$_1$ when *flipLnkCov* is greater than 95%, but also when the value of *flipLnkCov* decreases.

Figure 10.24 shows the FT/AR$_1$ of a CANcentrate-based system provided with 3 nodes for different values of *flipLnkCov*, which are indicated in the legend as percentages. Notice that, as a reference, the figure also shows the FT/AR$_1$ of an equivalent system relying on the CAN bus. Results indicate that the FT/AR$_1$ of a system that relies on CANcentrate is quite sensitive to the value of *flipLnkCov*.

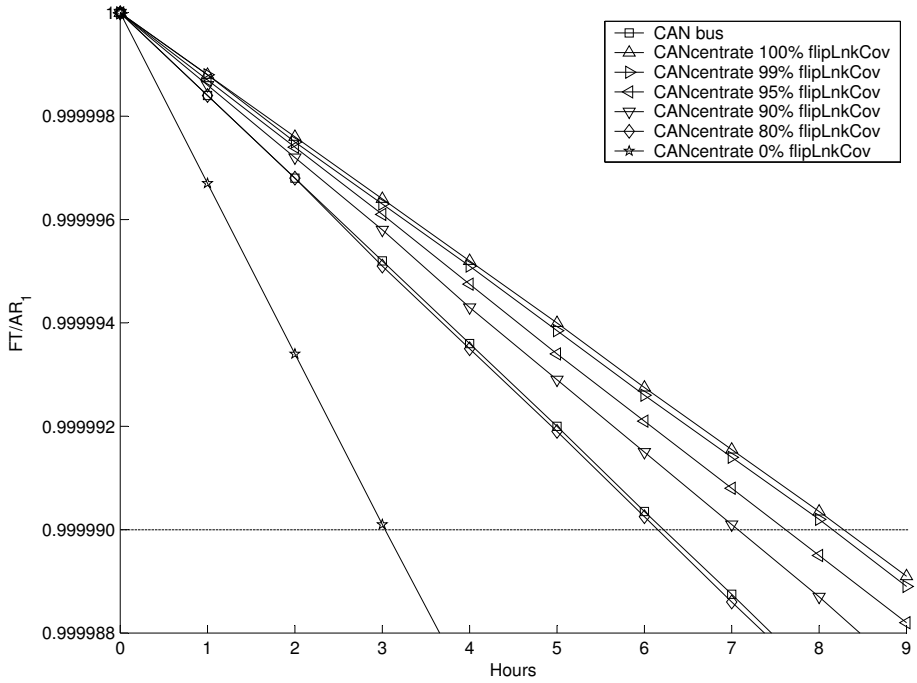

Figure 10.24: FT/AR$_1$ vs bit-flipping coverage of the hub for 3 nodes

However the implications of this fact are a bit subtle. Results suggest that the bit-flipping coverage the hub must provide in order to improve the FT/AR$_1$ when compared with the CAN bus is not too high. Specifically, notice that CANcentrate yields a worse mission time than the CAN bus only if *flipLnkCov* is lower or equal to 80%. Moreover, results also indicate that it is not so important to achieve a perfect coverage, which in turn is a practical impossibility. As can be seen in Figure 10.24, the mission time a system can achieve with a CANcentrate network whose hub provides a perfect bit-flipping coverage can be almost attained if this coverage is of the 99%.

Anyway, it is noteworthy that these results are based on the assumption that bit-flipping faults represent only a third of the faults that manifest in a way that is included in our fault model. This means that the sensitivity of the FT/AR$_1$ of CANcentrate should be greater as the proportion of bit-flipping faults increases.

Figure 10.25 shows the results when 15 nodes are considered. In this case, the results are slightly different. On the one hand, it is necessary that *flipLnkCov* drops down to the 45% in order to make a CANcentrate-based system worse than an

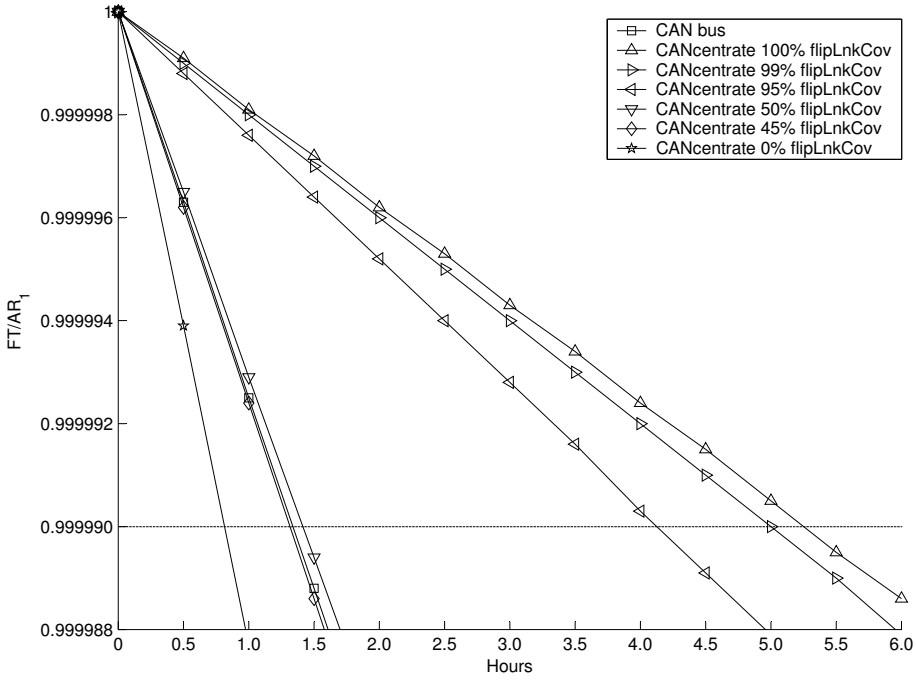

Figure 10.25: FT/AR_1 vs bit-flipping coverage of the hub for 15 nodes

equivalent CAN-based one in terms of mission time. This further supports the idea about the fact that the benefits of CANcentrate increase with the number of nodes; since for an average number of nodes, e.g. 15, the star is better than the bus even if the error-containment mechanisms of its hub are not very efficient. On the other hand, notice that the difference of mission time between the case in which *flipLnkCov* is 99% and the case in which it is 100% slightly increases when compared with what is depicted in Figure 10.24. This means that the sensitivity of the FT/AR_1 (with respect to *flipLnkCov*) of a system that relies on CANcentrate grows with the number of nodes. This is a quite predictable result, since the probability of fault occurrences, and hence of non-covered faults, increases with the number of nodes.

10.9.6 FT/AR_1 vs bit-flipping proportion

Besides the error-containment coverages, there are also other parameters for which we have assumed approximate, but intuitive values: the proportion of the different failure modes that are within our fault model. Therefore, it is important to carry

out a sensitivity analysis of the FT/AR$_1$ of a system that relies on CAN and on an equivalent system relying on CANcentrate with respect to these failure modes' proportions.

What is important in this analysis is to study how the achievable FT/AR$_1$ is affected by the proportion of the failure modes that are more harmful to the communication among the nodes. This is because, from an intuitive point of view, the advantages of the star become stronger as the proportion of these faults increase. For instance, in Section 10.9.4 it has been just demonstrated that the benefits of CANcentrate decrease as the proportion of fail-silent node failures increases. In this sense, notice that among all the failure modes that are within our fault model, bit-flipping failures are the most harmful ones, since neither the CAN controller nor the hub can provide a perfect error-containment coverage for them. Therefore, here we perform a sensitivity analysis with respect to the proportion of bit-flipping faults, while assuming that the rest of failure modes that are within our fault model are equiprobable.

Parameter	**Value**
nodeIOOfmProp	0.0
nodeIOStrProp	0.40
nodeIOStdProp	0.40
nodeIOFlipProp	0.20
termOfmProp	0.00
termStrProp	0.40
termStdProp	0.40
termFlipProp	0.20
termLossProp	0.0
ctrlOfmProp	0.00
ctrlStrProp	0.70
ctrlStdProp	0.20
ctrlFlipProp	0.10

Table 10.2: 20% bit-flipping proportion sensitivity analysis configuration

Table 10.2 illustrates how the parameters that specify the failure mode proportions of the entities have been varied to perform this analysis. Specifically, this table shows, as an example, the failure mode proportions of the Node IO, the Termination and the Controller when the FT/AR$_1$ is assessed for a 20% of bit-flipping faults.

The Node IO is an example of entity that can exhibit three different failure modes included in our fault model: stuck-at-recessive, stuck-at-dominant and bit-flipping.

For this type of entities, the proportion of bit-flipping failures is specified as the 20% and the proportion of stuck-at-recessive and stuck-at-dominant faults are calculated as $(1.0 - 0.20) \cdot \frac{1}{2} = 0.40$; which means that one half of Node IO's non-bit-flipping faults that are within our fault model manifest as stuck-at-recessive, whereas the other half manifest as stuck-at-dominant.

The Termination is an example of entity that can exhibit four failure modes included in our fault model: a stuck-at-recessive, a stuck-at-dominant, a bit-flipping and a Termination loss. The proportion of the first three failure modes are specified as in the case of the Node IO. On the other hand, notice from Table 10.2 that the parameter that specifies the proportion with which a Termination fails by leading the medium to loss that termination is specified as the 0%. This is because, as explained in Section 5.2, a Termination loss indirectly compels the medium to become bit-flipping. This line of reasoning is also applied to the case of the Attachment entity, so that the proportion with which it suffers from a physical disruption is also specified as the 0%.

As regards the Controller, notice that it is a type of entity that is different from the others, in the sense that the parameters that specify its failure modes' proportions must be calculated taking into account that faults affecting some of its internal modules cannot manifest as bit-flipping, but can only lead the Controller to transmit a stuck-at stream (see Section 10.4.4). In fact, this is the reason why the default proportions of stuck-at-recessive, stuck-at-dominant and bit-flipping failures are not equiprobable in the case of the Controller (see Table 10.4). In particular, Table 10.2 shows the resultant Controller's failure mode proportions when it is considered that the internal modules of the CAN controller that can provoke a bit-flipping failure do manifest it with a probability of the 20%. For instance, following the indications of Section10.4.4, the proportion with which the Controller exhibits a stuck-at-recessive fault (the value of the parameter *ctrlStrProp*) is calculated as: $\frac{4}{8} + \frac{4}{8} \cdot 0.20 = 0.70$.

At this point and before continuing, it is important to highlight again that we still assume that when a Node Core exhibits a failure mode included in our fault model, this failure mode is stuck-at-recessive.

Figure 10.26 shows the FT/AR_1 of a system that relies on CAN and of an equivalent system that relies on CANcentrate when 3 nodes are considered. As can be seen there, the FT/AR_1 of a CAN-based system is more sensitive than the FT/AR_1 of a CANcentrate-based one to the proportion of bit-flipping faults, i.e. to the *flip proportion*. For instance, a CAN-based system achieves a mission time of 8 and 4.3 hours when the flip prop is of the 0% and of the 100% respectively, whereas a CANcentrate-based system achieves 8.3 and 6.6 hours respectively for these same

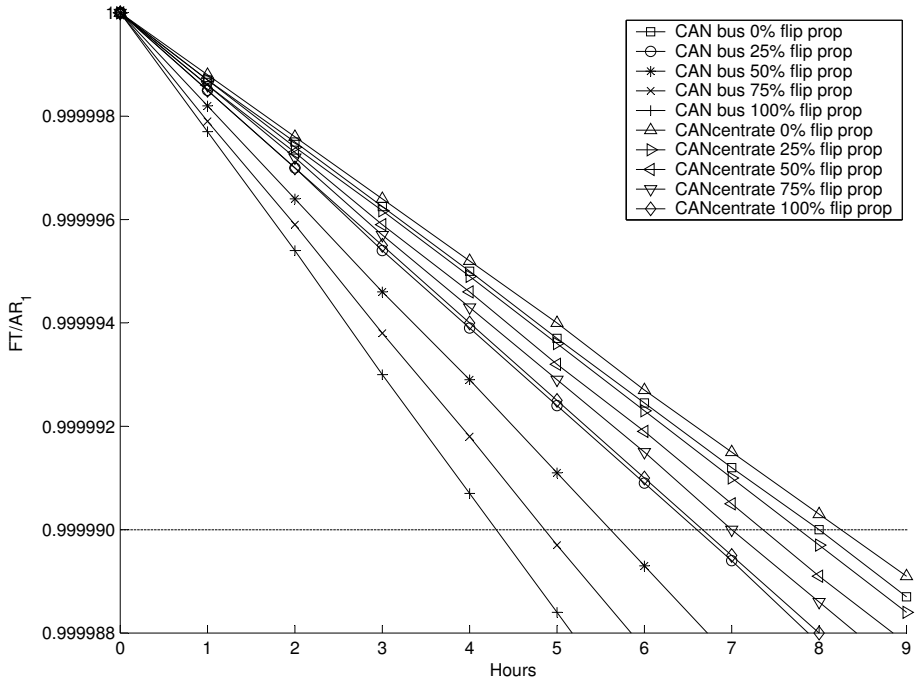

Figure 10.26: FT/AR$_1$ vs bit-flipping proportion for 3 nodes

proportions. This means that when the flip prop is increased from the 0% to the 100%, the mission time is reduced around the 46% with CAN and around the 20% with CANcentrate. Another example that illustrates the bigger sensitivity of the FT/AR$_1$ of the CAN bus with respect to the flip prop is the fact that the mission time achievable with a CANcentrate star where the flip proportion is of the 100% is slightly better than the mission time achievable with a CAN bus where such proportion is only of the 25%.

To better understand these results, notice that although we assume that the CAN controller and the hub can diagnose bit-flipping faults with the same coverage, i.e. with a 95% of effectiveness, the CAN controller is not always able to isolate such a type of fault. For instance, when the CAN controller diagnoses a bit-flipping fault in its transceiver, it cannot contain it if such a fault compels the transceiver to directly deliver bit-flipping bits to the medium. In contrast, the hub is always able to isolate a bit-flipping fault when it diagnoses it at any of its ports. As a consequence, the hub can actually contain bit-flipping bits with a higher coverage than the CAN controller and, thus, CANcentrate is more resilient to these errors.

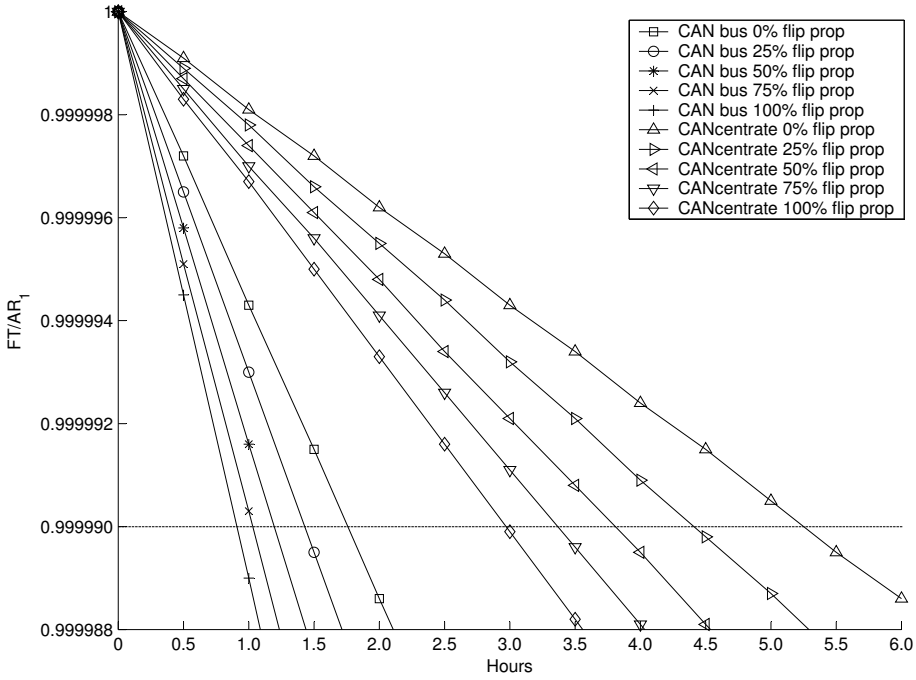

Figure 10.27: FT/AR_1 vs bit-flipping proportion for 15 nodes

Figure 10.27 analyzes the FT/AR_1 of a system relying on CAN and of an equivalent system that relies on CANcentrate with respect to the flip proportion for 15 nodes. Following the first example mentioned before, a CAN-based system achieves a mission time of 1.7 and 0.9 hours when the flip prop is of the 0% and of the 100% respectively, which implies a reduction of mission time around the 47%. This indicates that the sensitivity of the FT/AR_1 of a CAN-based system with respect to the flip prop remains almost constant for any number of nodes.

In contrast, if we analyze the case of an equivalent CANcentrate-based system, we can observe that its mission time is reduced by 44%: from 5.2 hours with a flip prop of the 0% down to 2.9 hours with a flip prop of the 100%. This means that the sensitivity of the FT/AR_1 of a CANcentrate-based system with respect to flip prop increases as the number of nodes grows. However, results also indicate that this bigger sensitivity of the star is compensated by the stronger benefits CANcentrate yields with an increasing number of nodes. For example, Figure 10.27 shows that a CAN-based system with a 0% of bit-flipping faults achieves a lower mission time than an equivalent CANcentrate-based system in which the flip proportion is of the 100%.

Finally, note that it would be also interesting to study how the proportion of the other failure modes that are included in our fault model affect the FT/AR_1 that can be achieved with CAN and CANcentrate. For instance, it would be valuable to analyze the FT/AR_1 with respect to the proportion of stuck-at-recessive faults, which are the ones that have the least impact on the communication on a bus topology. Anyway, this last study concerning stuck-at-recessive faults was somehow carried out in Section 10.9.4. This is because the nodes are the least reliable elements of the system and, thus, to assess the FT/AR_1 with respect to the fail-silent node proportion is equivalent to analyzing the FT/AR_1 with respect to the proportion of almost all stuck-at-recessive faults that cannot prevent nodes from communicating in a bus (note that a stuck-at-recessive bus section will block the bus).

10.9.7 FT/AR_1 vs out-of-fault-model proportion

In the previous section we highlighted the necessity of performing a sensitivity analysis of the FT/AR_1 with respect to the proportion of the different failure modes. In particular, that section focused on the proportion of the failure modes that are included in our fault model. Conversely, this section analyzes how the proportion of out-of-fault-model (ofm) faults affects the FT/AR_1.

Notice that up to this point we have assumed that this proportion is of the 0%. As explained in Section 10.4.4, this is necessary for assessing what would be the reliability benefits of CANcentrate in systems that include the appropriate mechanisms to deal with faults that are beyond the scope of this star. However, to evaluate the effect of ofm failures on the system reliability is interesting, since it will determine the importance of dealing with them, e.g. the importance of providing the system with mechanisms for babbling-idiot faults.

In order to carry out this analysis we varied the proportion of ofm failures of the entities that can actually fail in this way, i.e. of the entities that can fail by generating syntactically correct frames that are erroneous from a semantic point of view (see Section 10.4.4 for a detailed explanation of what we consider as an ofm failure). In this sense, we only change the ofm proportions of the entities Node Core and Controller, whereas we keep the ofm proportions of the rest of entities as 0.0. Notice that this is reasonable, since it is almost impossible that a fault affecting any of the rest of entities, e.g. a cable, builds a syntactically correct CAN frame, or changes some bits of a CAN frame that is being transmitted in such a way that they remain undetected by the native error-detection mechanisms of CAN.

In order to measure the FT/AR_1 for a given proportion of ofm faults, we set the parameters that specify the proportion of this kind of fault in each entity to the

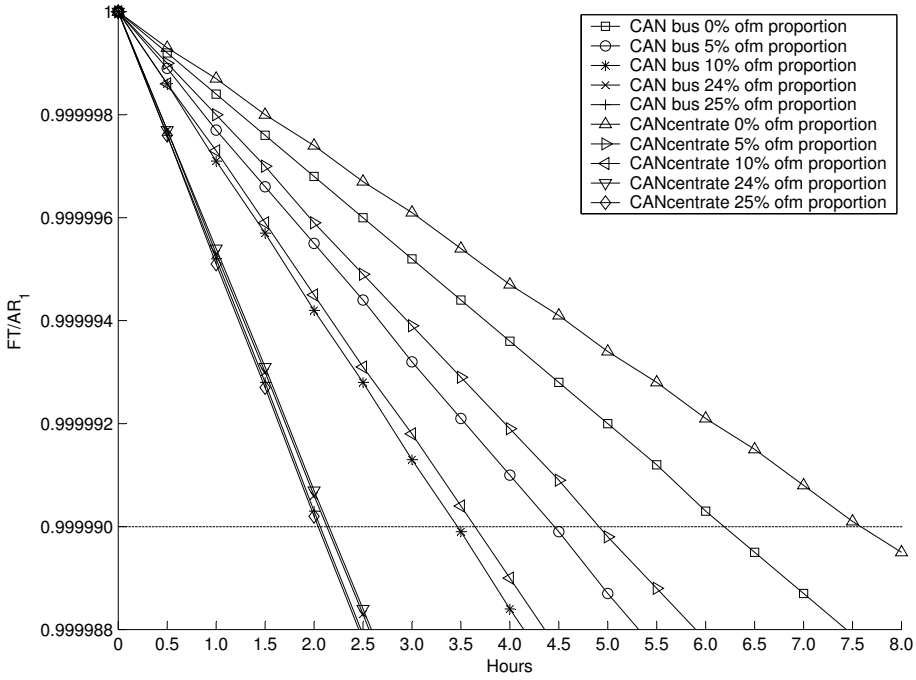

Figure 10.28: $\mathrm{FT/AR}_1$ vs ofm proportion for 3 nodes

desired proportion. For instance, if we consider that the proportion of ofm faults is of the 20%, then the parameters *nodeIOOfmProp*, *nodeCoreOfmProp*, *ctrlOfmProp*, etc. are set to 0.2. Then, for each entity (except for the Node Core and the Controller), the rest of the parameters that specify the proportions of the failure modes that are within our fault model are re-calculated so that those failure modes are equiprobable. In contrast, the proportions of the failure modes of the Node Core and the Controller that are within our fault model are re-calculated in a different way. Since the only type of non-ofm failures that a Node Core can exhibit is a stuck-at-recessive, the parameters that establish the proportion with which the Node Core exhibits a stuck-at-dominant or a bit-flipping failure are kept as 0.0, whereas the parameter that specifies the proportion of stuck-at-recessive Node Core failures, i.e. *nodeCoreStrProp*, is specified as $1.0 - \mathit{nodeCoreOfmProp}$. Regarding the Controller entity, the proportions of its failure modes that are included in our fault model are also supposed to be equiprobable. Nevertheless, the parameters that specify the actual proportion with which a Controller exhibits a stuck-at-recessive, a stuck-at-dominant and a bit-flipping failure are re-calculated taking into account that some of its internal modules can only provoke a stuck-at-recessive failure. See

Section 10.4.4 for a detailed explanation of the way in which we suppose that the internal modules of the CAN controller actually fail.

Figure 10.28 shows the FT/AR_1 of a system that relies on CAN and of an equivalent system that relies on CANcentrate for different values of the proportion of ofm faults, i.e. to the *ofm proportion*, when 3 nodes are considered. As can be seen in this figure, the FT/AR_1 of both a CAN-based system and an equivalent CANcentrate-based one are very sensitive to this parameter. However, conversely to what we observed with the proportion of bit-flipping faults, CANcentrate is slightly more sensitive to the ofm proportion than the CAN bus. For example, when the ofm proportion is greater than 24%, then the CANcentrate-based system achieves a lower mission time than the CAN-based one.

To understand these results it is necessary to recall again that the nodes are the most unreliable elements of the system. As a consequence, the benefits of CANcentrate when compared with CAN strongly depend on the effectiveness with which the hub isolates faults occurring at nodes. In this sense, notice that a similar effect on the CANcentrate's FT/AR_1 was observed when we studied it with respect to the bit-flipping coverage of the hub (see Section 10.9.5).

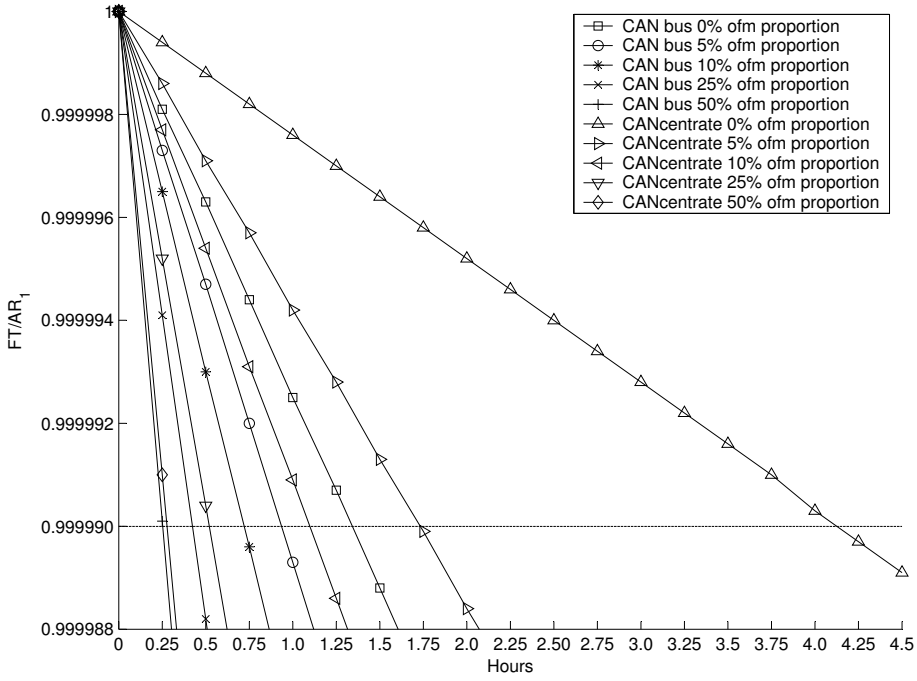

Figure 10.29: FT/AR_1 vs ofm proportion for 15 nodes

Figure 10.29 supports those results and shows that CANcentrate becomes even more sensitive as the number of nodes grows. Nevertheless, it shows that the FT/AR_1 of CANcentrate becomes worse than the FT/AR_1 of the CAN bus only for an ofm proportion greater than the 50%. This was somehow expected since in previous sections we have observed that the benefits of CANcentrate over CAN are more evident as the number of nodes grows (Section 10.9.2).

In conclusion, in order to benefit from the reliability advantages of a simplex star topology such as CANcentrate, it can be valuable to provide the system with mechanisms that isolate node's faults that manifest in a semantic manner. In particular, these mechanisms should be specially effective for systems that include a low number of nodes. However, the need for including these additional mechanisms depends on the actual proportion of semantic failures. In this sense, this inclusion is justified only if the proportion of semantic faults a high-enough and, thus, a study that experimentally assessed this proportion would be very useful.

10.9.8 FT/AR_1 vs wires and connectors' failure rates

As already explained, one of the main disadvantages of using CANcentrate instead of CAN is the fact that the star requires more hardware components than the bus to interconnect a given ensemble of nodes, thereby increasing the probability that faults occur and limiting the star's reliability benefits. On the one hand, CANcentrate includes more quantity of hardware components per node. For instance, the star includes extra CAN cables and connectors for implementing the Attachment entities that represent the link (the uplink and the downlink) of each node. Moreover, the star also includes, for each node, hardware for building one extra Node IO, two Hub IOs, and four Terminations (two Terminations for the uplink and another two Terminations for the downlink). On the other hand, the star has an additional entity that is not present in the bus: the Hub Core.

Although this disadvantage of CANcentrate reduces the system NFT/AR when compared with CAN, we have also seen that the better error-containment capabilities of the star compensate its extra hardware and allow improving the system FT/AR_1. Thus, at this point, it raises the question of whether or not it is possible to further improve the FT/AR_1 of a star-based system by investing in the reliability of the star's extra components. In order to partially answer this question, this section is devoted to carrying out a sensitivity analysis with respect to the reliability of the CAN cables and connectors, i.e. with respect to the reliability of the Attachment entities. We believe that it is not strictly necessary to carry out a dedicated sensitivity analysis with respect to the reliability of each one of the other entities whose amount is bigger in CANcentrate than in CAN, i.e. with respect to the reliability of

the Node IO, the Hub IO or the Termination. Notice that the failure rates of those entities are similar to the failure rate of an Attachment entity. Thus, the sensitivity of the FT/AR$_1$ with respect to the failure rate of each one of these entities can be indirectly studied by analyzing the FT/AR$_1$ with respect to the Attachment's failure rate. As concerns the Hub Core, it represents a single point of failure and, thus, to perform a sensitivity analysis with respect to its reliability is an issue of paramount importance that will be addressed in the next section.

Wire failure rate (failures/hour/km)	**Connector failure rate (failures/hour)**	**busAttchFRate (failures/hour)**	**lnkAttchFRate (failures/hour)**
$1.00000 \cdot 10^{-5}$	$2.07774 \cdot 10^{-6}$	$4.44310 \cdot 10^{-6}$	$6.15738 \cdot 10^{-6}$
$1.00000 \cdot 10^{-6}$	$2.07774 \cdot 10^{-7}$	$4.46023 \cdot 10^{-7}$	$6.17452 \cdot 10^{-7}$
$1.00000 \cdot 10^{-7}$	$2.07774 \cdot 10^{-8}$	$4.63159 \cdot 10^{-8}$	$6.34588 \cdot 10^{-8}$
$1.00000 \cdot 10^{-8}$	$2.07774 \cdot 10^{-9}$	$6.34519 \cdot 10^{-9}$	$8.05948 \cdot 10^{-9}$
$1.00000 \cdot 10^{-9}$	$2.07774 \cdot 10^{-10}$	$2.34812 \cdot 10^{-9}$	$2.51955 \cdot 10^{-9}$

Table 10.3: Attachments' failure rates configuration for 15 nodes

Coming back to the point at issue, we analyze the system FT/AR$_1$ with respect to the reliability of the Attachment entities of CAN and CANcentrate. As already explained, an Attachment entity includes a piece of a CAN cable (which includes four wires), two connectors and the portion of the PCB areas where the connectors are attached to (see Sections 10.4.1 and 10.4.2). Thus, we varied the reliability of the Attachment entity by considering different failure rates for some of these components. More specifically, we changed the order of magnitude of the default failure rates of the wires and the connectors, while keeping the default PCB areas' failure rates. Notice that the default failure rates we have considered so far for the Attachment's components are: $1.0 \cdot 10^{-7}$failures/hour/km for each wire located within a CAN cable, $2.07774 \cdot 10^{-8}$failures/hour for each connector, and $9.52 \cdot 10^{-10}$failures/hour for each connector's PCB (see Tables 10.5 and 10.6).

Table 10.3 specifies the Attachment's failure rates we have obtained for a system composed of 15 nodes, when considering different failure rates for the Attachment's components just mentioned. In particular, *busAttchFRate* specifies the failure rate of the Attachment entity that constitutes a bus section, whereas *lnkAttchFRate* is the failure rate of the Attachment entity that constitutes a link of CANcentrate. Note that the value of *busAttchFRate* depends on the number of nodes because this number determines the length of each bus section. More specifically, since we assume that nodes are equidistant in a bus-based system, we obtain this length by dividing the total bus length (100 m) by the quantity of nodes minus one (see Section 10.4.1 for a detailed explanation of the assumptions we have

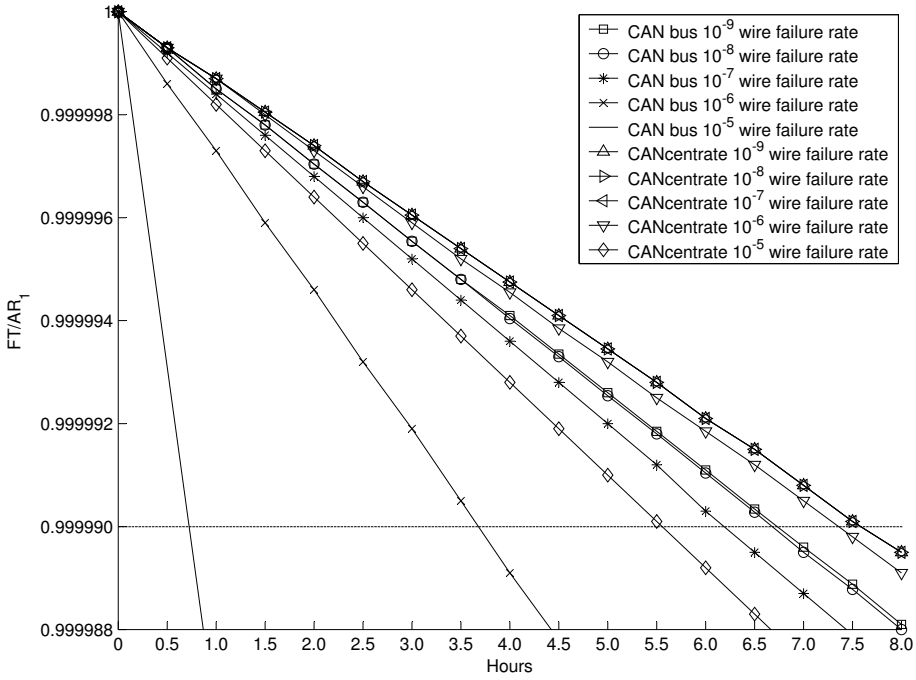

Figure 10.30: FT/AR_1 vs cable and connector's failure rate for 3 nodes

made concerning different important implementation issues of CAN and CANcentrate). In particular, Table 10.3 refers to a system that includes 15 nodes, thus the *busAttchFRate*'s values this table specifies are calculated supposing that the length of each bus section is of $\frac{100}{15-1}$ meters. Similarly, the value of *lnkAttchFRate* depends on the star's diameter (100 m) and, more specifically, on the links' length. However, in this case, the length of each link does not depend on the number of nodes, since we have assumed that all links have the same length and that this length is half the diameter, i.e. 50 m. Thus, conversely to the case of *busAttchFRate*, the values of *lnkAttchFRate* Table 10.3 specifies are obtained assuming that the length of a link is always of 50 m.

Figure 10.30 shows the FT/AR_1 of a system that relies on CAN and of an equivalent system relying on CANcentrate for the wires and connectors' failure rates specified in the two first columns of table 10.3, when 3 nodes are considered. For the sake of succinctness, each element included in the legend of this figure only specifies the order of magnitude of the wire's failure rate. Results indicate that a CANcentrate-based system achieves a higher FT/AR_1 than an equivalent CAN-based one for all the failure rates we have considered for the wires and connectors.

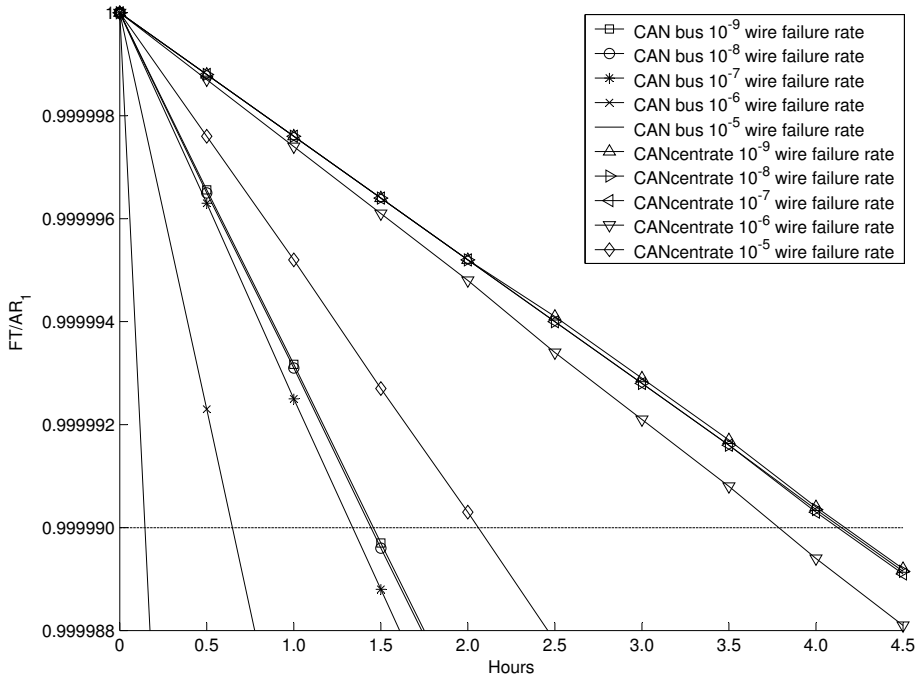

Figure 10.31: FT/AR_1 vs cable and connector's failure rate for 15 nodes

The figure also shows that the FT/AR_1 of the CAN-based system is much more sensitive to the reliability of wires and connectors than the CANcentrate-based one. Notice that when the failure rates of the wires and connectors are lower than the default values we have considered so far for them, only the FT/AR_1 of the CAN-based system is drastically reduced. This result indicates that the star should be the choice when the system includes a small number of nodes and it is not possible to use reliable-enough cables and connectors.

Furthermore, the low sensitivity of a CANcentrate-based system is also reflected on the fact that it does not achieve a higher mission time when the order of magnitude of the wires and connectors' failure rates is reduced with respect to our default case. In contrast, the mission time of an equivalent CAN-based system can be improved by around half an hour if the failure rates of the wires and connectors are decreased in this way. This indicates that, in principle, when the system is constituted by a small amount of nodes, to use highly-reliable wires and connectors is an issue deserving of attention when the system relies on CAN, but not when it relies on CANcentrate.

However, to try to improve the reliability of a CAN-based network by investing in the reliability of its cables and connectors has an important practical limitation. Note that it is almost impossible in practice to achieve wires and connectors' failure rates whose orders of magnitude are lower than those we consider in the default case, i.e. lower than 10^{-7} and 10^{-8} respectively. On the one hand, the only wire's failure rate considered by the MIL-HDBK standard is $1.00000 \cdot 10^{-7}$ failures/hour/km, whereas other sources have established even higher wire's failure rates, e.g. 1.15-1.81 $\cdot 10^{-6}$ failures/hour in [Lat05]. On the other hand, the MIL-HDBK standard yields a connector's failure rate of $1.03887 \cdot 10^{-8}$ when the highest quality level is considered for this type of component.

The second figure, Figure 10.31, qualifies part of the above remarks. Specifically, it shows that, when an average number of nodes, i.e. 15, is considered, the FT/AR_1 of a system that relies on CANcentrate becomes also very sensitive to the wires and connectors' failure rates. Fortunately, this sensitivity only manifests when these failure rates are quite high. In particular, the improvement of the mission time of a CANcentrate-based system that could be achieved with highly-reliable wires and connectors is not noticeable if it is compared with the mission time that can be attained with CANcentrate when the wires and connectors present the default failure rates we assumed for them.

Moreover, the increased sensitivity of the CANcentrate-based system is widely compensated by the fact that the relative FT/AR_1 benefits of CANcentrate (with respect to CAN) grow with the number of nodes. In this sense, notice that the mission time reached by a CAN-based system that presents hypothetical highly-reliable wires and connectors, e.g. wires with a failure rate of $1.0 \cdot 10^{-9}$ failures/hour/km, is lower than the mission time achieved by an equivalent CANcentrate-based system where the reliability of those components is quite poor, e.g. where wires present a failure rate of $1.0 \cdot 10^{-5}$ failures/hour/km.

Finally, as mentioned at the beginning of this section, the results obtained in this analysis can be extrapolated to the case in which we were analyzing the reliability of other components whose amount is bigger in the star, e.g. of the components that make up the Node IO. It is important to pay attention to this fact because, conversely to what happens with the wires and connectors, it is possible to decrease the failure rate of the major part of the components that constitute the Node IO, the Hub IO and the Termination by two orders of magnitude. Note that these entities are mainly composed of electronic components, whose reliability is specially sensitive to the quality level. This means that to invest in the reliability of the electronic components that constitute any of these entities, e.g. in the transceivers, is an issue deserving of attention, specially for CAN-based system.

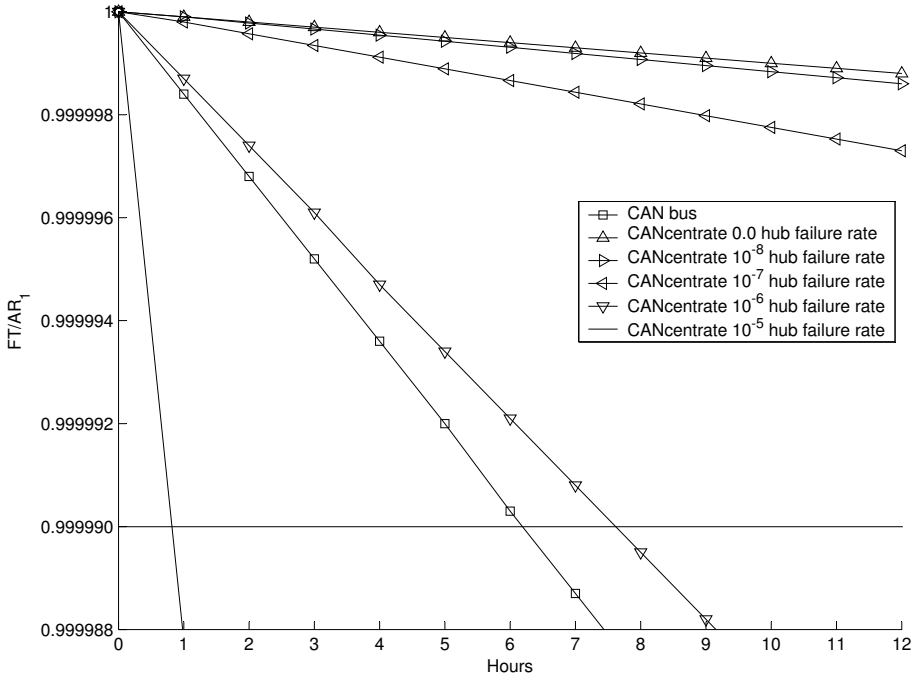

Figure 10.32: FT/AR$_1$ vs hub's failure rate for 3 nodes

10.9.9 FT/AR$_1$ vs Hub Core failure rate

As mentioned in the previous section, to analyze the sensitivity of the FT/AR$_1$ with respect to the reliability of the extra components that CANcentrate presents when compared with the CAN bus is valuable for clarifying if it is useful to invest in the reliability of those components. In particular, the hub reliability should have a big impact on the reliability benefits achievable with CANcentrate, since besides being an additional hardware element present in the star, it is also a single point of failure. As a consequence, to analyze the importance of investing in the hub reliability is an issue deserving of special attention.

Current section aims at analyzing the sensitivity of the FT/AR$_1$ of a CANcentrate-based system with respect to the reliability of the Hub Core, which is the part of the hub that actually represents the single point of failure. In order to carry out this analysis, we vary the order of magnitude of the Hub Core's failure rate. For example, the default failure rate of a Hub Core that interconnects 3 nodes is $1.20843 \cdot 10^{-6}$ failures/hour (see Table 10.6). Thus, for 3 nodes, we take into account the following Hub Core's failure rates: $1.20843 \cdot 10^{-5}$, $1.20843 \cdot 10^{-6}$,

$1.20843 \cdot 10^{-7}$ and $1.20843 \cdot 10^{-8}$. Additionally, we consider a Hub Core's failure rate of 0.0 failures/hour.

Figure 10.32 shows the FT/AR$_1$ achievable by a CANcentrate-based system for different Hub Core's failure rates when 3 nodes are considered. It also depicts as a reference the FT/AR$_1$ achieved by an equivalent CAN-based system. This figure demonstrates that the FT/AR$_1$ of the system that relies on CANcentrate is very sensitive to the failure rate of its Hub Core and that, hence, it is very important to invest in its reliability. For example, if the Hub Core's failure rate increases in one order of magnitude with respect to our default case, i.e. from $1.20843 \cdot 10^{-6}$ to $1.20843 \cdot 10^{-5}$ failures/hour, then the mission time is drastically reduced from 7.6 to 0.9 hours approximately. In contrast, if the reliability of the Hub Core is improved in such a way that its default failure rate is decreased in one order of magnitude, i.e. from $1.20843 \cdot 10^{-6}$ to $1.20843 \cdot 10^{-7}$ failures/hour, then the mission time is increased from 7.6 to 43 hours, which means that the mission time is improved by 466% approximately. But the improvement could be even greater. The figure indicates that if the Hub Core's failure rate was around $1.20843 \cdot 10^{-8}$ failures/hour, the CANcentrate-based system would reach a mission time of 77 hours, which would represent an improvement around the 913% when compared with our default case.

Note that since the Hub Core is mainly constituted by an electronic component, i.e. by the Hub Kernel, which is an IC, it is actually possible to achieve very low failure rates for it. More specifically, it is possible to reduce the Hub Core's failure rate to near $1.20843 \cdot 10^{-8}$ failures/hour by using components of the highest quality for its construction, e.g. components that are typical of military applications. This is an important result, because it means that it is feasible to achieve a mission time near 86 hours, which is the value that would be theoretically reached with a Hub Core's failure rate of 0.0 failures/hour. In conclusion, when a small amount of nodes is considered, the FT/AR$_1$ of a CANcentrate-based system can be boosted just by investing in the reliability of its Hub Core, while using commercial quality levels for the rest of the star's components.

Figure 10.33, which depicts the FT/AR$_1$ for 15 nodes, yields similar results. However, in this case the improvement of mission time that can be achieved by investing in the reliability of the Hub Core is lower than in the previous study. Specifically, the mission times that are achieved with Hub Core's failure rates of $1.87019 \cdot 10^{-6}$ (which is the default value for a hub that interconnects 15 nodes), $1.87019 \cdot 10^{-7}$ and $1.87019 \cdot 10^{-8}$ failures/hour are around 4.1, 12.8 and 15.9 hours respectively. This means that the mission time can be approximately improved by 212% and 287% with respect to the default case. This is because the contribution to the FT/AR$_1$ of the reliability of the components that do not compose the Hub

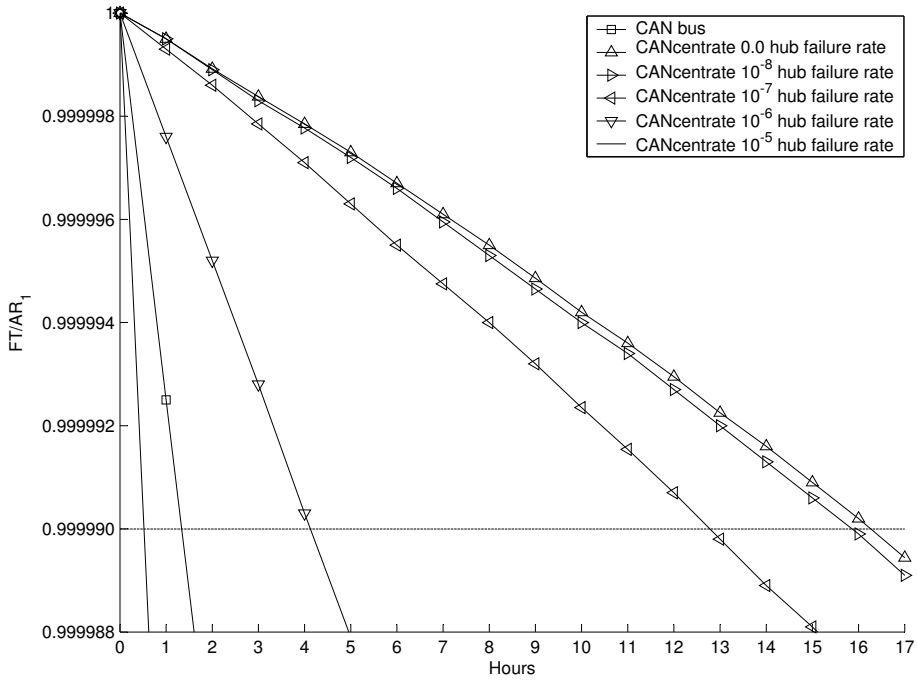

Figure 10.33: FT/AR_1 vs hub's failure rate for 15 nodes

Core grows with the number of nodes and, hence, the relevance of the Hub Core reliability decreases.

In any case, maybe it is still possible to further improve the FT/AR_1 of a system based on CANcentrate when an average number of nodes is considered, if the reliability of components different from those that constitute the Hub Core is also improved. In particular, given the impact that the quality has on the reliability of electronic components, it would be interesting to analyze the system FT/AR_1 when the quality of electronic components other than the Hub Kernel, such as the microcontroller, the CAN controller, and the transceivers, is also enhanced.

10.10 Conclusions

In this chapter we quantitatively compared the reliability achieved by equivalent systems relying on CAN and on CANcentrate when permanent hardware faults can occur. To our best knowledge, this is the first formal (quantitative) comparison of the system reliability that can be obtained when using a bus and a star that takes

into account the capacity of the hub and nodes to contain errors, different failure modes of the components, and implementation aspects.

We quantified the reliability that CAN and CANcentrate can yield for two different types of systems: *non-fault-tolerant/accepting systems* (NFT/A systems) and *fault-tolerant/accepting systems* (FT/A systems). For this purpose, we modelled using SANs the reliability of equivalent systems relying on CAN and on CANcentrate and, then, we compared their NFT/AR and their FT/AR$_1$, i.e. their *non-fault-tolerant/accepting system reliability* and their *fault-tolerant/accepting system reliability*.

Besides describing our SANs models in depth, we explained the assumptions they rely on. We showed that all these assumptions were made favoring, whenever possible, the CAN bus and always guaranteeing that results are not biased towards the star. Most assumptions are reflected as model parameters, for which we specified default values that can be considered as realistic.

Since CAN was initially designed as an in-vehicle network and there is an increasing interest in enhancing its reliability to make it suitable for highly-reliable distributed control systems, we take as a reference for the comparison, the reliability requirements of in-vehicle communications of x-by-wire systems. In particular, the reliability analyses herein presented focus on the mission time during which a system that relies on CAN and of an equivalent system that relies on CANcentrate exhibits a reliability ≥ 0.99999.

The first two analyses presented in this chapter compare the NFT/AR and the FT/AR$_1$ of CAN and CANcentrate for different number of nodes. We considered from 3 up to 20 nodes, thereby embracing typical node numbers in CAN-based body and powertrain subnetworks in vehicles. Results quantify the drawbacks and the benefits that a simplex star topology was supposed to yield in terms of NFT/AR and FT/AR$_1$ respectively.

On the one hand, results show that a simplex star slightly reduces the system NFT/AR, i.e. the reliability of NFT/A systems. However, the decrease of mission time that CANcentrate can cause remains almost constant for any number of nodes: around 35%.

On the other hand, as concerns the FT/AR$_1$, results quantitatively corroborate that it is possible to improve the reliability of FT/A systems by using CANcentrate instead of CAN. Moreover, the improvement of mission time yielded by CANcentrate increases as the number of nodes grows, e.g. by 22% and 260% for 3 and 20 nodes respectively. The stronger sensitivity of the CAN bus with respect to the number of nodes means that, when the FT/AR$_1$ is the main concern, CANcentrate allows the inclusion of more nodes while presenting an FT/AR$_1$ above a specific

threshold during a given mission time.

Besides the above-mentioned analyses, we quantitatively studied the sensitivity of the FT/AR$_1$ with respect to different relevant aspects, using for that purpose different values for the parameters of the models of CAN and CANcentrate. To our best knowledge, this is also the first time that it has been quantified how those aspects affect the relevance of using a star topology.

In order to obtain results that are valuable for a wide range of applications, we performed each sensitivity analysis considering that the system includes 3 and 15 nodes. On the one hand, 3 is the minimum number of nodes needed to tolerate the failure of one of them, and it could be the number of nodes of a small highly reliable embedded system. On the other hand, 15 is the average number of nodes of a typical in-vehicle CAN subnetwork.

The three first sensitivity analyses are devoted to quantifying how the coverage of different fault-tolerance mechanisms affect the system FT/AR$_1$ achievable with CAN and CANcentrate. In particular, we differentiated between the fault-tolerance capacity of the FT/A system itself and the effectiveness of the error-containment mechanisms of the communication subsystem it relies on.

As concerns the fault-tolerance ability of FT/A systems, we defined a coverage called *sysFauTolCov*, which characterizes the probability with which an FT/A system actually accepts or tolerates the failure or the disconnection of a node, provided that the system can potentially accept or tolerate this fault. The default value we assume for *sysFauTolCov* is of 100%, since this is the value that better represents the fault-tolerance capacity of FT/A systems: the ones that intrinsically accept communication-failed nodes, as well as highly-reliable FT/A systems that deliberately include highly-efficient fault-tolerance mechanisms. When we varied the value of this coverage we found out that the FT/AR$_1$ is quite sensitive to it. However we also showed that the minimum value of *sysFauTolCov* that should be attained in order improve the FT/AR$_1$ by means of CANcentrate strongly depends on the number of nodes. On the one hand, this coverage must be quite high, around 97%, for small networks (3 nodes). This result does not negatively affect the suitability of CANcentrate for highly-reliable FT/A systems, since they typically achieve fault-tolerance coverages $\geq 99.99\%$. But it highlights the necessity of designing quite efficient fault-tolerance mechanisms in order to use CANcentrate for improving the reliability of non-highly-reliable FT/A systems that include few nodes. On the other hand, for a network that includes an average number of nodes, e.g. 15, we showed that the minimum *sysFauTolCov* should be around $\geq 90\%$. This relatively low threshold indicates that CANcentrate is appropriate for improving the reliability of almost every medium-size FT/A system.

As regards the sensitivity of the FT/AR$_1$ with respect to the coverage of the error-containment mechanisms of the communication infrastructure, we firstly studied the nodes' ability to contain their own errors, i.e. the fail-silent node proportion. We found out that this proportion dramatically affects the FT/AR$_1$ yielded by CAN, but not the FT/AR$_1$ obtained with CANcentrate. In fact, if nodes are always able to contain their own errors, then the CAN bus is better than CANcentrate. However, a small decrease in the error-containment capacity of nodes provokes a strong drop in the mission time achieved by the bus. For example, when 15 nodes are considered, it is enough that the FS proportion of nodes drops to around the 98% to turn a CANbased system worse than an equivalent CANcentrate-based one.

Note that the error-containment capabilities of a node are implemented by its CAN controller. This implies that, in fact, this study quantifies the relevance of using a simplex star topology depending on the effectiveness of the error-containment mechanisms provided by the CAN protocol. We believe that the results herein presented concerning this issue are very valuable, since there is not a real consensus on the actual value of this effectiveness. Moreover, those results constitute a preliminary assessment of the suitability of using a simplex star topology instead of bus guardians in order to improve error-containment and thus reliability in CAN networks.

The second analysis related to the error-containment coverage addresses the ability of the hub to contain faults that manifest as the transmission of bit-flipping bits. Results suggest that, for a small number of nodes, it is important to guarantee a high enough bit-flipping coverage, i.e. a coverage $> 80\%$. However, results also indicate that for a bigger number of nodes CANcentrate yields benefits even with low bit-flipping coverages, e.g. until 45% for 15 nodes. Finally, another important result obtained from this analysis is that, in principle, it is not necessary to make a great effort to achieve a perfect bit-flipping coverage (which is actually impossible in practice). In particular, the maximum improvement that can be achieved in terms of mission time is almost achieved with a bit-flipping coverage of 99%.

Besides these studies related to the fault-tolerance coverages, we analyzed the sensitivity of the FT/AR$_1$ with respect to the proportion with which faults manifest as bit-flipping. This additional study is needed because although the default values we assumed for the component's failure mode proportions can be considered as realistic, they are actually intuitive. Results indicate that the CAN bus is much more sensitive than CANcentrate to the bit-flipping proportion. This was expected despite the fact that we assume that both the hub and the nodes detect bit-flipping fault with the same effectiveness (which is pessimistic for CANcentrate). The reason is that the hub is able to isolate bit-flipping faults that the CAN controller can detect but not passivate. In addition, results also show that the sensitivity of the

FT/AR_1 provided by CANcentrate increases with the number of nodes, but that this fact is compensated by the stronger reliability benefits the star yields as the number of nodes grows. For example, for 15 nodes, CANcentrate is better than CAN even if the bit-flipping proportion is 100% in the star and 0% in the bus.

Apart from analyzing the sensitivity of the FT/AR_1 to the bit-flipping proportion, we also assessed it for the proportion with which faults affecting the node (basically the microcontroller and the CAN controller) manifest in a way that is not included in our fault model (out-fault-model failures). More specifically, note that these faults are those that generate syntactically correct frames that are erroneous from a semantic point of view. This evaluation is interesting for assessing the importance of providing the system with mechanisms that deal with this type of fault, e.g. with babbling-idiot ones. Results show that CANcentrate is slightly more sensitive than CAN. This is because the nodes are the most unreliable elements of the system and, thus, the benefits of CANcentrate strongly depend on the effectiveness with which it can isolate faults occurring at them. However, the need for including these additional mechanisms is justified only if the proportion of semantic faults is high-enough. Thus, a study that experimentally assessed this proportion would be very useful.

The two last sensitive analyses we performed study the impact of the reliability of the CANcentrate's extra components on the star's FT/AR_1 benefits. Firstly, we measured the FT/AR_1 with regard to the reliability of the extra components the star includes per node. On the one hand, results indicate that the CAN bus is very sensitive to the reliability of those components, whereas the star is not. On the other hand, we showed that from both, a theoretical and a practical point of view, it is not possible to achieve a better mission time with the bus than with the star, just by investing in the reliability of the cables and connectors.

Secondly, we analyzed the FT/AR_1 with respect to the Hub Core reliability. Results confirm that the FT/AR_1 achievable by CANcentrate is extremely sensitive to this parameter. In fact, surprisingly, they show that the mission time of CANcentrate can be really boosted just by investing in the Hub Core reliability, e.g. by investing in its quality. This effect is particularly noticeable in the case of a CANcentrate network that interconnects a small number of nodes. For instance, for 3 nodes, the mission time of CANcentrate can be improved by 913% in practice. When more nodes are considered, the improvement in terms of mission time is lower, e.g. around 287% for 15 nodes. This is because the contribution of the Hub Core reliability to the reliability of the network decreases as the number of nodes increases. As a consequence, to invest in the reliability of other network components is necessary to really boost the mission time in those star based systems.

At this point, it is very important to highlight that the above-explained analyses are not intended to provide absolute figures for CAN or CANcentrate, but to compare the system reliability that can be achieved with both infrastructures in order to justify the interest of using CANcentrate. In this sense, this chapter pursues the objective of dependability evaluation, i.e. to guide the design and implementation of a system by analyzing how different options and decisions affect its dependability [TMGT93].

It is also worth noting that results presented in this chapter are likely to be lower bounds to the system reliability achievable by CANcentrate, given the special concern taken to ensure it was not favored in the analysis and the fact that several other advantages, such as the minimization of the impact of spatial proximity faults, were not even modelled, as explained in Section 10.3. Therefore, it can thus be inferred that even in the case that our analyses should yield a result whereby the FT/AR_1 of CAN is equal to the one of CANcentrate, the actual fact is that CANcentrate is superior.

Although these results refer to the case of CAN, and other technologies, such as TTP/C or FlexRay, deal with different failure modes, conclusions regarding the justification of using a star topology depending on the different aspects addressed in the sensitivity analyses can be extrapolated to these technologies. On the one hand, they use similar components with similar failure rates. On the other hand, failure modes can be abstracted so that what really matters is the proportion of failures that can be covered by the hub and the nodes.

Finally, as explained in Section 10.1, although there is an increasing interest in providing mechanisms to deal with transient faults, a reliability evaluation taking them into account is necessarily application dependant. Therefore, transient faults are beyond the scope of this dissertation and will be addressed in future work.

Parameter	Default value	Meaning
kSevere	0 (NFT/AR), 1 (FT/AR_1)	Value of k of the concept of k-severe failure.
sysFauTolCov	1.0	Coverage of the fault-tolerance mechanisms of the system that relies on CAN, CANcentrate or ReCANcentrate
ctrlFlipCov	0.95	Probability with which the CAN controller successfully diagnoses a bit-flipping fault
ctrlItselfIsoCov	0.5	Probability with which a CAN controller that diagnoses itself as faulty can successfully isolate its internal fault
nodeIOInFauProp	0.5	Proportion with which the *Node IO* does not deliver errors to the medium
nodeCoreFRate	$3.25312 \cdot 10^{-6}$	*Node Core* failure rate
nodeCoreOfmProp	0.0	*Node Core out-of-fault-model* proportion
nodeCoreStrProp	1.0	*Node Core* stuck-at-recessive proportion
nodeCoreStdProp	0	*Node Core* stuck-at-dominant proportion
nodeCoreFlipProp	0	*Node Core* bit-flipping proportion
ctrlFRate	$1.25537 \cdot 10^{-6}$	*Controller* failure rate
ctrlOfmProp	0.0	*Controller out-of-fault-model* proportion
ctrlStrProp	0.6666	*Controller* stuck-at-recessive proportion
ctrlStdProp	0.1666	*Controller* stuck-at-dominant proportion
ctrlFlipProp	0.1666	*Controller* bit-flipping proportion
nodeIOFrate	$6.73258 \cdot 10^{-7}$	*Node IO* failure rate
nodeIOOfmProp	0.0	*Node IO* out-of-fault-model proportion
nodeIOStrProp	0.3333	*Node IO* stuck-at-recessive proportion
nodeIOStdProp	0.3333	*Node IO* stuck-at-dominant proportion
nodeIOFlipProp	0.3333	*Node IO* bit-flipping proportion
termFRate	$7.38299 \cdot 10^{-8}$	*Termination* failure rate
termOfmProp	0.0	*Termination out-of-fault-model* proportion
termStrProp	0.25	*Termination* stuck-at-recessive proportion
termStdProp	0.25	*Termination* stuck-at-dominant proportion
termFlipProp	0.25	*Termination* bit-flipping proportion
termLossProp	0.25	*Termination* loss proportion

Table 10.4: CAN bus, CANcentrate and ReCANcentrate models' common parameters

Parameter	**Default value**	**Meaning**
numNodes	3, 15	Number of nodes connected to the bus.
numInSections	0, 12	Number of bus sections, except the two ones located at the extremities of the bus line. This parameter allows to indirectly specify the total number of bus sections needed to interconnect the amount of nodes specified by the parameter *numNodes*. The total number of sections is equal to $(numInSections + 2) = (numNodes - 1)$
busAttchFRate	$6.34588 \cdot 10^{-8}$, $4.63159 \cdot 10^{-8}$	*Attachment* failure rate when it represents a bus section. This rate decreases with the number of nodes, since the length of a bus section is calculated dividing the total bus length by the number of nodes. The first and second default values correspond to the failure rate of each bus section when the bus interconnects 3 and 15 nodes respectively
busAttchOfmProp	0.0	*Attachment out-of-fault-model* proportion when it represents a bus section
busAttchStrProp	0.25	*Attachment* stuck-at-recessive proportion when it represents a bus section
busAttchStdProp	0.25	*Attachment* stuck-at-dominant proportion when it represents a bus section
busAttchFlipProp	0.25	*Attachment* bit-flipping proportion when it represents a bus section
busAttchDisProp	0.25	*Attachment* physical disruption proportion when it represents a bus section

Table 10.5: Parameters specific to the CAN bus model

Parameter	Default value	Meaning
numBranches	3, 15	Number of star branches. In other words, the number of nodes connected to the star.
hubCoreFRate	$1.20843 \cdot 10^{-6}$, $1.87019 \cdot 10^{-6}$	*Hub Core* failure rate. This rate increases with the number of nodes, since the hub needs more hardware to connect a greater number of them. The first and second default values correspond to the failure rate the Hub Core for 3 and 15 nodes respectively
flipLnkCov	0.95	Probability with which the hub successfully diagnoses a bit-flipping fault at an uplink hub port
hubIOFRate	$6.73258 \cdot 10^{-7}$	*Hub IO* failure rate
hubIOOfmProp	0.0	*Hub IO out-of-fault-model* proportion
hubIOStrProp	0.3333	*Hub IO* stuck-at-recessive proportion
hubIOStdProp	0.3333	*Hub IO* stuck-at-dominant proportion
hubIOFlipProp	0.3333	*Hub IO* bit-flipping proportion
lnkAttchFRate	$6.34588 \cdot 10^{-8}$	*Attachment* failure rate when it represents the uplink or the downlink of CANcentrate
lnkAttchOfmProp	0.0	*Attachment out-of-fault-model* proportion when it represents the uplink or the downlink of CANcentrate
lnkAttchStrProp	0.25	*Attachment* stuck-at-recessive proportion when it represents the uplink or the downlink of CANcentrate
lnkAttchStdProp	0.25	*Attachment* stuck-at-dominant proportion when it represents the uplink or the downlink of CANcentrate
lnkAttchFlipProp	0.25	*Attachment* bit-flipping proportion when it represents the uplink or the downlink of CANcentrate
lnkAttchDisProp	0.25	*Attachment* physical disruption proportion when it represents the uplink or the downlink of CANcentrate

Table 10.6: Parameters specific to the CANcentrate model

Chapter 11

ReCANcentrate

11.1 Introduction

Previous sections focused on the motivation, design, implementation and dependability evaluation of CANcentrate. It has been explained that CANcentrate includes enhanced error-detection and fault-treatment mechanisms that allow it to deal with errors and to contain them at their ports of origin. Moreover, a quantitative dependability analysis of a system relying on the CAN bus and of an equivalent system based on CANcentrate demonstrates that CANcentrate can actually improve the system reliability when compared with the second one. Specifically, it has been shown that CANcentrate is more suited than a CAN bus for systems that accept or tolerate the failure or the disconnection of at most 1 out of N nodes, thereby quantitatively corroborating the advantage that the error containment capabilities of an adequate simplex star topology intuitively yield in terms of reliability.

However, in some applications the degree of system reliability achieved by CANcentrate could be not enough and the presence of a single point of failure unacceptable. In these cases, spatial redundancy at the hub level is required so as to tolerate permanent hub faults.

In order to provide this redundancy, we have proposed a new communication infrastructure, called ReCANcentrate, that relies on a replicated star topology that includes two hubs. Besides providing the same capacity of error containment as CANcentrate, ReCANcentrate further tolerates one hub failure, as well as faults that affect one of the connections of each node to the hubs (no matter to which hub).

This chapter is devoted to describing ReCANcentrate. First of all, it explains the

existing main approaches for providing CAN with redundancy, paying attention to the problems they pose and identifying their pros and cons. Then, it describes the design, the characteristics and the functionalities of ReCANcentrate and addresses issues related to the cabling length and the bit rate. Finally it also describes the implementation of a ReCANcentrate prototype and the tests that were conducted to check its functionalities and performance.

11.2 Redundancy approaches in CAN

Removing all severe points of failure from a communication system can only be achieved with redundancy, either temporal or spatial. However, permanent communication faults, such as a physical disruption of the medium, can only be tolerated with spatial redundancy. This can be found in several existing safety critical protocols, such as TTP [KG94], FlexRay [Fle05], or FlexCAN [PF04], which rely on replicated media architectures.

Regardless of the specific topology, we can differentiate between two main uses of media replication: increased throughput, when the replicated media are used independently to transmit different data; or fault tolerance, when the replicated media are used to transmit the same data [Bel02]. Each communication medium is commonly referred to as channel and in this dissertation we will focus on the use of replicated channels for fault-tolerance purposes.

The main problem of using replicated channels is that nodes must be able to manage the redundant frames they receive. Particularly, they must determine when two received frames are in fact copies of the same frame (duplicates), or when a frame received from one channel is omitted from the other (omissions). Synchronizing the transmission and the reception of frames across the network is a possible solution. This synchronization is easily achieved in time-triggered protocols due to their inherent transmission schema. In fact, each frame is expected to be transmitted quasi-simultaneously in both channels at predefined time slots. Hence, removal of duplicates and detection of omissions is done on the fly. This is the basic transmission mechanism specified in protocols such as TTP and FlexRay, which provide communication via dual channel either using a bus topology, a star topology or, in the case of FlexRay, also using an hybrid topology.

Unfortunately, since CAN is an event-triggered protocol, it does not provide any mechanism for synchronizing the frames in the different channels. Therefore, additional mechanisms have been proposed in the literature to provide some sort of synchronization, as in FlexCAN, SMART-1 [KHJN03], or the *Columbus Egg Idea* [RVA99]. FlexCAN uses a strategy based on triplicated CAN buses and

nodes. The nodes are coordinated by means of a software that uses timers to control the transmissions and the receptions on the channels. In [KHJN03] the CAN network used in the lunar mission SMART-1 (*Small Missions for Advanced Research in technology*) is described. This network includes replicated nodes and two replicated CAN buses, one active and other inactive that is used as a spare. Nodes detect when the active channel is faulty and then switch to the spare one, thus there is no need of synchronization at the frame level. Finally, the *Columbus Egg Idea* proposed in [RVA99] uses several CAN buses and eliminates the need of dealing with duplicates and omissions by coupling the streams received from all buses, at the bit level, in each node.

In principle, any of these solutions could be used to provide replicated CAN channels while dealing with duplicates and omissions. Nonetheless, although they provide synchronization between channels, they still rely on a bus topology, which still has severe points of failure, e.g. nothing prevents a faulty node from continuously sending erroneous information to all channels.

Instead, since CANcentrate provides improved fault-treatment capabilities with respect to CAN buses, we decided to replicate this star in order to definitively eliminate all severe points of failure. Moreover, notice that CANcentrate is transparent with respect to any application or CAN-based protocol executed at CAN nodes. Thus, any of the replication strategies referred above could be even used for replicating CANcentrate, replacing each bus by a hub.

However, we finally found out that none of them is the most suited for replicating CANcentrate. On one hand, systems such as FlexCAN or SMART-1 rely on a quite complex solution that increases the requirements of nodes in terms of hardware and software. On the other hand, the approach proposed in [RVA99] would limit the dependability features of the replicated star, as will be described later on in Section 11.4.

Therefore, we proposed to replicate CANcentrate in a way that the resultant infrastructure does not exhibit these limitations. We called this novel replicated star topology *ReCANcentrate*. ReCANcentrate takes advantage of the dependability properties already achieved by CANcentrate and provides nodes with an easy way to manage the replicated star. Despite replicated stars being available for protocols such as TTP and FlexRay, to the best of our knowledge, ReCANcentrate is the first one for CAN.

11.3 Fault model of ReCANcentrate

In principle, the fault model considered by ReCANcentrate just gathers the same kind of faults included in the fault model of CANcentrate (see Section 5.2). However, it is necessary to further discuss some aspects concerning the way in which faults manifest in ReCANcentrate.

Firstly, note that in this chapter we will explicitly refer to network partition faults. This is because, conversely to what happens in the case of a simplex star topology, particular combinations of faults occurring in different ports of the two hubs of a replicated star may lead nodes to have an inconsistent view of the nodes that are available for communicating. Figure 11.1 depicts such a situation, showing the failure of two different links connecting two different nodes to different hubs. Node A can communicate with nodes B and C. However, nodes B and C cannot communicate with each other.

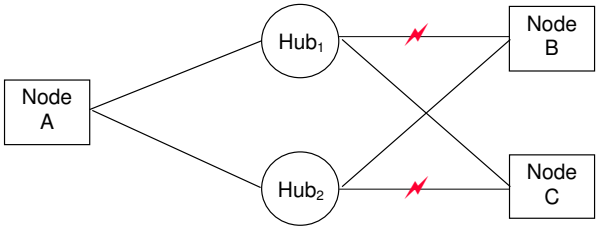

Figure 11.1: Example of a network partition in a replicated star topology

Secondly, it is noteworthy that in CANcentrate the unique fault assumption made was that the hub would not fail. Conversely, we relax this supposition for the case of ReCANcentrate and we consider that at most one of the hubs can fail. As a consequence, faults may occur not only at nodes and links, but also at one of the hubs. More specifically, note that a hub cannot build or buffer CAN frames. Thus, we suppose that a faulty hub can only fail by generating or propagating (in case it does not isolate a faulty port) syntactic incorrect data, i.e. stuck-at or bit-flipping bits.

Finally, it is important to note again that ReCANcentrate is provided with mechanisms that allow tolerating faults affecting the components that constitute any of the two connections of each node to the hubs. In particular, as will be seen later on in Sections 11.4 and 11.7, each one of these two connections includes its own CAN controller, so that the node can tolerate the failure of one of these controllers. Because of this reason, to specify that a CAN controller's fault manifests as the

transmission of stuck-at or bit-flipping bits is not enough. In addition, we need to specify the way in which a faulty CAN controller manifests from its node point of view. In this sense, the fault model of ReCANcentrate also includes the possibility that a CAN controller *crashes*. If this happens, the CAN controller stops performing any action, so that it only delivers recessive bits to the network and notifies its node about nothing (about no transmission, no reception, etc.).

11.4 Design rationale

The main architectural characteristic of ReCANcentrate is that it is constituted by two CANcentrate hubs interconnected by means of two dedicated links called *interlinks* (Figure 11.2). Nodes are connected to each hub via a link that contains an uplink and a downlink, as in CANcentrate.

A given interlink is also formed by two *sublinks* used by each hub to send the coupling of the contributions of its own nodes to the other hub. For the sake of simplicity, we will refer to this signal as the *contribution* of that hub. Then, the resulting signal that each hub broadcasts to its own nodes is the one that results from coupling its own contribution with the contribution received from the other hub. This coupling creates a *single logical broadcast domain* since both hubs behave like one, transmitting the same value bit by bit in their downlinks, i.e. in-bit response [ISO93] is enforced in the whole replicated domain.

This tight coupling has deep consequences on the whole communication system. Firstly, nodes can be either connected to both hubs, for improved fault tolerance, or they can be connected to one hub, only. In any case, the node transmissions will be broadcasted to all nodes. Therefore, regardless the hub or hubs a node is connected to, all nodes will have a coherent view of which nodes are available for communicating thus preventing network partitions to occur.

Secondly, nodes connected to both hubs receive the same frames within the same bit time thus easily distinguish between duplicates and omissions, which is one of the major problems when designing an event-triggered system that relies on a replicated communication system. Specifically, duplicated frames are always expected in each reception, whereas an omission can be easily detected by checking, at the reception of each frame, that two copies of the same frame are effectively received from both stars.

As concerns the fault-treatment capabilities of ReCANcentrate, note that each node can be considered as two CAN nodes, each one connected to a different hub. In such a way, a hub monitors the contribution of a CAN node, regardless the con-

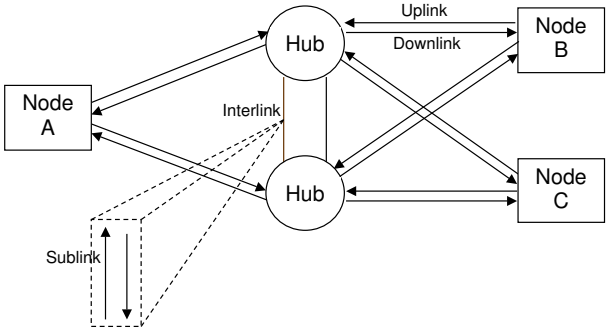

Figure 11.2: Architecture of ReCANcentrate

tribution the node sends to the other hub, and isolates the corresponding port when faulty. Therefore, the fault-treatment capabilities of CANcentrate with respect to faults occurring at nodes or links apply to the whole network by means of the actions performed by both hubs at their respective ports.

Additionally, each hub includes mechanisms for detecting and isolating faults occurring at the interlinks. Specifically, it monitors the two sublinks, within both interlinks, that carry the contribution from the other hub. When any of these sublinks fails, the hub isolates it, but continues using the other one. Therefore, hubs will communicate with each other as far as there are two non-faulty sublinks (regardless the interlinks they are located in) that make possible them to interchange bits in both directions.

Moreover, the hub also uses these mechanisms to detect and isolate a faulty hub. Specifically, this occurs when both contributions received from the other hub are diagnosed as being faulty. In such a way, the errors generated by the faulty hub cannot propagate through the non-faulty hub to the nodes.

Note that despite ReCANcentrate removes all severe points of failure, the properties derived from having both hubs coupled only apply when hubs can still correctly receive the contribution from each other. Otherwise, the communication system would be equivalent to have two independent CANcentrate stars. This still provides a valid replicated communication system but losing most of the features referred above. Namely, the in-bit response may be lost, the domains of each hub may be strictly isolated from each other and then duplicates could arrive at very different instants in time. However, communication is still possible and thus graceful degradation is provided.

Finally, different schemas can be used in order to connect a node to both hubs.

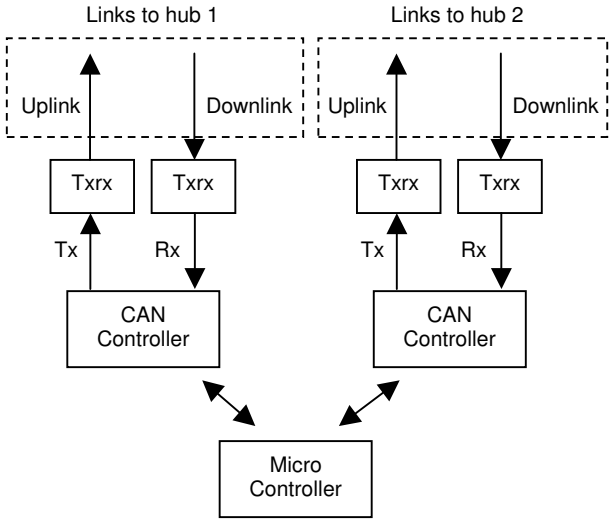

Figure 11.3: Architecture of a ReCANcentrate's node

However, in order to provide a high degree of fault tolerance, we propose the schema depicted in Figure 11.3. As can be seen there, each node has one microcontroller and two CAN controllers, each of which is connected to one hub, using one transceiver ($Txrx$) for the uplink and another one for the downlink. A node will only transmit using one of the CAN controllers at a given time, while receiving from both. This selective behavior in the node transmissions can be enforced by several ways. For instance, each node could monitor the state of its CAN controllers. When a given controller is not able to communicate through the hub it is connected to, the node switches to the other controller thereby communicating through the other hub. In Section 11.7 we will explain further details concerning the way in which the node manages the traffic it observes at both its CAN controllers.

Note that the schema shown in Figure 11.3 allows tolerating, per node, faults that affect any of the components the node uses to connect to one of the hubs (no matter which hub), e.g. a fault affecting one of the links of the node. Anyway, for simplicity of implementation, it is also possible to use a mechanism similar to the one proposed in [RVA99] to allow nodes to connect to both channels using a single CAN controller.

Figure 11.4 depicts this alternative schema. The node is composed of one microcontroller and one CAN controller, which connects to each hub using the cor-

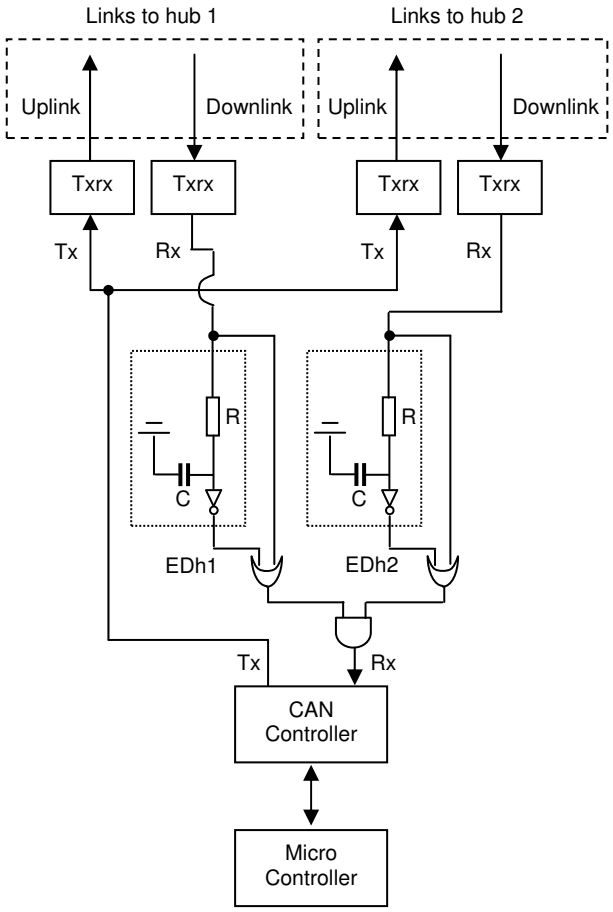

Figure 11.4: Possible simplified architecture of a ReCANcentrate's node

responding pair of transceivers. The transmission pin of the CAN controller is connected to the transmission pin of the two transceivers that correspond to the uplinks, so that the node transmits exactly the same bit stream to both hubs. Similarly, the reception pin of each one of the two transceivers that correspond to the downlinks are coupled by means of an AND gate, whose output is then driven into the reception pin of the CAN controller. However, the reception signal of each one of these two transceivers is not directly connected to the AND gate. In contrast, each one of them is previously driven into a specific OR gate together with a dedicated *EDhi* signal. This signal allows masking the stream received from a specific downlink, when this stream becomes stuck-at-dominant. Specifically, the value of an EDhi signal is basically calculated by means of a *resistor / capacitor circuit* (RC circuit), whose input is the contribution of the downlink. Note that when this contribution becomes stuck-at-dominant, the capacitor's plate that is connected to the inverter becomes completely discharged, so that the inverter outputs a logical '1' that masks the downlink's stream at the corresponding OR gate.

Unfortunately this approach is not recommendable when high fault tolerance is desired. On the one hand, it does not tolerate faults that manifest as a bit-flipping downlink. Second, note that this mechanism couples both downlinks and uplinks at the node. Thus, since the CAN controller will send error frames through both uplinks in response to errors it detects in any of the downlinks, both hubs will observe these error frames at the respective uplink ports. This will likely cause both hubs to isolate the node even when the fault only affects the connection of the node to one of the hubs.

11.5 Internal structure of the hub

In order to replicate CANcentrate, the internal structure of the hub has been slightly modified. As depicted in Figure 11.5, the ReCANcentrate hub continues being constituted by the same three main modules as in CANcentrate: the Coupler Module, the Input-Output and the Fault-Treatment Module.

The Coupler Module includes an additional AND gate and two additional OR gates. The first AND gate, AND_C, plays the same role as in CANcentrate. It couples the contributions from every node, $B_{1..n}$, that is connected to the hub. However, in this case, the output of this AND gate, i.e. the *hub contribution* (B_0), is not broadcasted to these nodes. Instead, it is sent to the other hub by means of two identical contributions, B_{00} and B_{01}, that are routed into different sublinks within distinct interlinks. By sending these two copies of B_0, the failure of the sublink that carries one of them is tolerated.

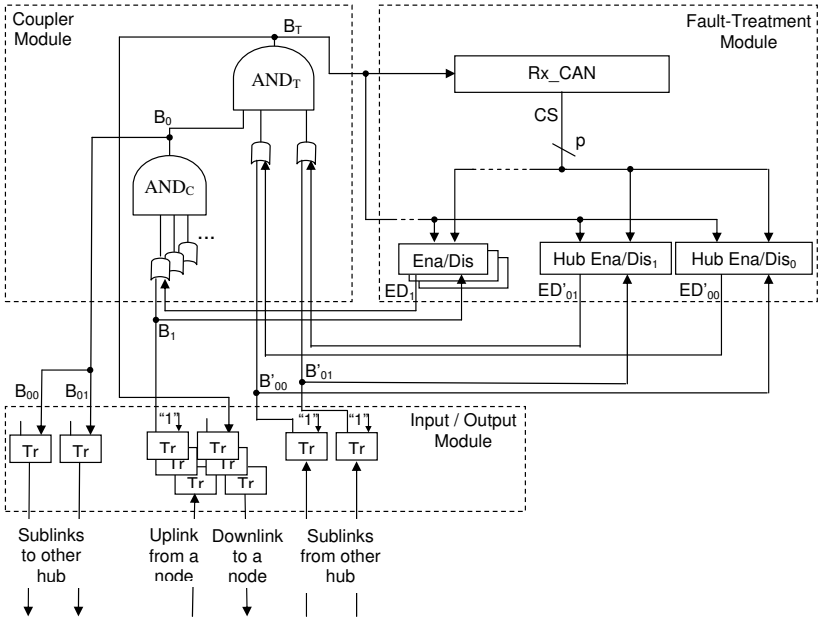

Figure 11.5: New internal structure of the hub

The second AND gate, AND$_T$, is aimed at coupling B_0 with the contribution of the other hub. Two copies of this contribution are received by means of two identical signals, B'_{00} and B'_{01}, routed within different sublinks. Finally, the resultant signal from coupling $B_{1..n}$, B'_{00} and B'_{01}, is then broadcast to the nodes, B_T.

As in CANcentrate, each OR gate connected to AND$_C$ is used to enable or disable the contribution of a specific node connected to the hub. In contrast, each one of the additional OR gates, which are connected to AND$_T$, is used to enable or disable one of the signals that carry the contribution from the other hub (either B'_{00} or B'_{01}).

Note that the output of the gate AND$_T$ within both hubs would be the same as a CAN bus interconnecting all non-faulty nodes connected to any of the hubs. Similarly to the case of CANcentrate (see Section 5.4), the frame that results from coupling the frames from all these nodes is called *resultant frame*. In such a way, nodes can consider both hubs as one unique CANcentrate hub. This allows ReCANcentrate to keep all the CAN properties related to dependability, as well as to enforce a synchronization between both stars at bit level.

Furthermore, the usage of two AND gates within the Coupler Module makes it

possible to separate the contributions of both hubs, thus allowing each hub to detect errors in the contribution of the other hub and isolate it when faulty.

Regarding the modifications in the Input/Output Module, simply note that four transceivers have been included for transmitting and receiving the contributions of both hubs. Two of these transceivers are used to send the copies of the hub contribution (B_{00} and B_{01}), whereas the other two are aimed at receiving the copies of the contribution of the other hub (B'_{00} and B'_{01}).

Finally, regarding the changes within the Fault-Treatment Module, the internal structure and functionalities of the modules the hub already included in CANcentrate (see Chapters 5 and 6) have not been modified. The only change performed on these modules is that now they monitor B_T, instead of B_0 (see Section 5.4), to know the value of each bit of the *resultant frame* that is broadcasted to the nodes.

For instance, each Enabling/Disabling Unit (*Ena/Dis*) in the Figure 11.5) continues using the coupled signal, now B_T, and the set of signals CS together with the contribution from its corresponding port, B_i, in order to diagnose whether its port is permanently faulty or not. For removing the contribution of its port, the Enabling/Disabling Unit also uses the appropriate Enabling/Disabling signal, $ED_{1..n}$.

Additionally, two new units have been added, namely *Hub Enabling/Disabling units* (*Hub Ena/Dis*$_0$ and *Hub Ena/Dis*$_1$). Each one of them is responsible for detecting errors in the contribution received from the other hub at a specific sublink (either B'_{00} or B'_{01}). When one of these signals is permanently faulty, the corresponding *Hub Ena/Dis* disables it by means of the appropriate Enabling/Disabling signal.

11.6 Error-detection and fault-treatment mechanisms of the hub

As explained above, a hub detects errors and treats faults occurring at nodes and links by means of the Enabling/Disabling Units, whereas it detects/treats errors and faults occurring at the sublinks by means of the Hub Enabling/Disabling Units.

The error-detection and fault-treatment functionalities performed by each Enabling/Disabling Unit are kept as they were in CANcentrate. Specifically, these functionalities consist of detecting and counting errors in the contribution of the corresponding port, produced by any of the types of faults gathered in the fault model. For each kind of fault there are specific rules for detecting the errors that may appear due to the fault and an associated threshold (see Chapters 6 and 7). When the number of errors related to a given fault exceeds the corresponding

threshold, the port is diagnosed as being permanently affected by that fault.

The Hub Enabling/Disabling Unit performs essentially the same functionalities as the Enabling / Disabling Unit, but introducing some small changes. Specifically, the Hub Enabling/Disabling Unit uses slightly different rules for detecting errors due to bit-flipping faults, i.e. bit-flipping errors; and it uses higher thresholds for diagnosing each kind of fault.

The Enabling/Disabling Unit detects bit-flipping errors by checking each one of the bits issued from its corresponding port. Particularly, the Enabling/Disabling Unit considers that the value of a bit is correct if it matches with the set of allowed values that are expected for that bit. As in CANcentrate, this set of values are calculated, bit by bit, taking into account which is the current state of the *resultant frame*, as well as which is the role currently played by the node that sends the bit, i.e. whether it is a transmitter or a receiver (see Section 7.2).

In contrast, the Hub Enabling/Disabling Unit must take into account that the contribution received from the other hub can be, in fact, the contribution of a transmitting node that sends its frame to the other hub already coupled with the contributions of several receiving nodes connected to that hub. This implies only few changes in the rules followed for calculating the set of correct values for each bit.

As concerns the differences about the thresholds, notice that as long as a given hub does not isolate a faulty port, it sends the errors issued through this port to the other hub. In such situations, the other hub will detect errors in the contribution received from this hub. Hence, if the thresholds of the Hub Enabling/Disabling Unit are not higher enough than the thresholds of the Enabling/Disabling Unit, then faults at some ports of a hub occurring near in time may lead the other hub to unfairly diagnose that the hub is faulty.

In the worst case, all ports of a hub may consecutively fail. Thus, it may be required to consider thresholds N times greater in the Hub Enabling/Disabling Unit, where N is the number of ports. Nevertheless, this would imply a big latency for detecting a real failure of a hub contribution. Fortunately, since port failures occurring very near in time are very unlikely, we can consider that a value of $N = 3$ is wise enough, while not increasing significatively the latency for diagnosing a real failure of a hub contribution.

11.7 Node's media management strategy

As explained above, in ReCANcentrate each star is a CAN channel that conveys a replica of the same data in parallel to provide tolerance to hub and link faults. How-

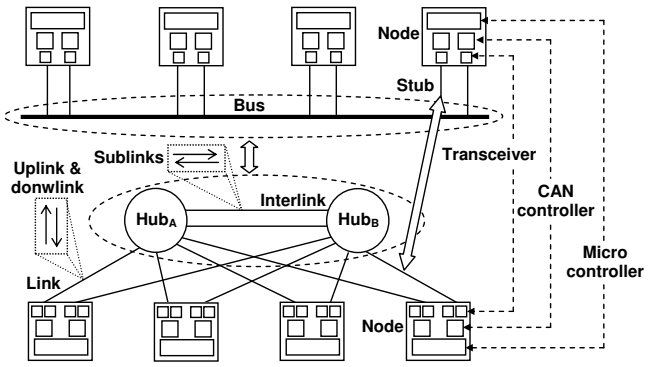

Figure 11.6: Analogy between ReCANcentrate and CAN with two controllers

ever, as pointed out in Section 11.2, to use parallel channels that rely on an event-triggered protocol such as CAN has an important drawback: there is no mechanism to synchronize the traffic between channels. As a consequence, nothing guarantees the traffic to be the same in all of them, so that each node needs to detect when frames received at different instants of time, each one through a different channel, are copies of the same one (duplicates); as well as when a frame received from one channel is omitted from the others (omissions). Moreover, an event-triggered transmission schema does not provide a node with mechanisms to diagnose when a fault prevents it from communicating through a channel.

To overcome these limitations, the hubs of ReCANcentrate perform a special AND-coupling within a fraction of the bit time, thereby creating a single logical broadcast domain that keeps the dominant/recessive transmission and the in-bit response properties of CAN. In this way, both hubs behave like one, transmitting the same value bit by bit in their downlinks. In fact, as depicted in Figure 11.6, ReCANcentrate is logically equivalent to a CAN network where nodes have two connections to a single bus: hubs are equivalent to the bus line, and each link corresponds to a stub.

As already said in Section 11.4, this coupling allowed us to define a strategy for each node to easily manage the replicated star, using the node architecture depicted in Figure 11.3. According to our media management strategy, the node triggers each transmission towards one of the hubs only, while receiving from both hubs at the same time. One controller acts as the *transmission controller* (*tx controller*), so that it is used to both transmit the frames of its node and receive frames sent by other nodes (the *tx controller* does not receive its own frames). The other controller, the *non-transmission controller* (*non-tx controller*), is used to receive

frames transmitted by its own node and by other nodes.

When a frame is successfully exchanged through the network, i.e. when a *delivery event* occurs, each node expects that its two controllers quasi-simultaneously notify of that event. This notification can occur in two different manners. First, if the node successfully transmits a frame, the *tx controller* and the *non-tx controller* notify of the transmission and reception of this frame respectively. Second, if the node receives a frame sent from another node, it is notified of this reception by its two CAN controllers.

Thus, in the absence of faults, the node manages transmissions and receptions as follows. First, if the node successfully transmits a frame, the *tx controller* and the *non-tx controller* notify of the transmission and reception of this frame respectively. Then, the node only needs to accept the notification of the transmission as valid and release the reception buffer of the *non-tx controller*. Second, if the node receives a frame sent from another node, it is notified of this reception by its two CAN controllers. When this happens, the node must merely consume the frame received at one of the controllers and, then, release the reception buffers of both controllers.

Note that as long as the hubs are coupled, a single communication domain is enforced and, thus, each node can also manage the traffic in a simple way when faults occurs. Certainly, this domain cannot be enforced if faults lead any hub to continue coupling the contributions of its own nodes, but not to couple with the other hub. This may happen if all interlinks are faulty, and thus isolated, or if a hub erroneously decides not to couple with the other hub. Fortunately, the probability of such situations is almost negligible since there are several interlinks and the hub could include internal redundancy to reduce the likelihood of an incorrect decoupling.

How nodes detect and manage two independent stars is beyond the scope of this dissertation. Next sections are devoted to describing how each node manages the replicated traffic in the presence of faults when the single communication domain is enforced.

11.7.1 Faults and discrepancies

Conversely to what typically happens in other replicated media schemes, the nodes of ReCANcentrate do not need to deal with discrepancies between channels, i.e. they do not need to execute a distributed or a complex algorithm to differentiate duplicates from omissions and to reach a consensus on what frames are actually exchanged through the network. In contrast, when a node observes that its two controllers differ in their visions of what delivery events take place in the network,

it can rely on the fact that one of its controllers has an incorrect view of what actually occurs in the single communication domain.

To analyze the discrepancies between the two controllers of a node, let us differentiate again between faults occurring at the media and at controllers. Media faults manifest as the generation of stuck-at or bit-flipping bits that usually generate errors that corrupt data. In general, these errors block the channel. Consequently, as long as the hub/s do not contain them by disabling the adequate hub ports, no controller notifies about a transmission or a reception and no discrepancy between controllers can take place. When the media fault is isolated, the channel becomes unblocked, but only the controllers that have not been isolated so far will communicate again. Thus, thereafter, each node that has an isolated controller will observe what we call a *notification omission discrepancy*, i.e. that only one of its controllers notifies of a *delivery event*.

Notice that there are some situations in which a media fault does not prevent all controllers from communicating, but that does lead them to inconsistently exchange frames. First, a frame is inconsistently exchanged in any of the error scenarios affecting the last-but-one bit of a frame that have been identified for CAN [PMJ00]. Second, a stuck-at-recessive fault may provoke an inconsistency if it prevents a controller from monitoring the traffic, or if it impedes that its contribution reaches its corresponding hub. For instance, if a downlink is stuck-at-recessive during the broadcast of a whole frame, the controller connected to that downlink will not observe it. The media management we propose does not take into account these situations. This is because the probabilities of the scenarios due to errors in the last-but-one bit are controversial [FOFF04], whereas the probability of an inconsistency due to a stuck-at-recessive fault should be also very low.

Regarding what discrepancies can be provoked by CAN controller faults, notice again that we consider that a controller can only exhibit a crash failure. When this happens its node will also observe a *notification omission discrepancy* each time a new frame is exchanged. Certainly, a controller could also suffer from a Byzantine failure. If this happens, the controller can also provoke a *notification omission discrepancy* if it arbitrarily omits a notification. Additionally, a Byzantine CAN controller could also forge notifications, thereby delivering semantically incorrect frames to its node. In this case, the fault could manifest as a *notification non-omission discrepancy*, which occurs when both controllers of a node notify of a *delivery event*, but they do not agree on the frame the event is related to. Anyway, since the detection of semantically incorrect frames requires knowledge about the application executed at nodes (and thus is beyond our fault model), we postponed the treatment of CAN controller's faults other than crashes for future work.

11.7.2 Treatment of discrepancies and fault-tolerance strategy

As just explained, the errors generated by a fault block the communication in both stars as long as the hubs do not isolate it by disabling the appropriate hub ports. Once isolated, it is necessary to carry out further actions in order to tolerate it. Specifically, it is needed that any node that cannot communicate through a given hub as a consequence of that fault (e.g. if a hub fault, no node will be able to communicate through that hub) does not stop communicating through the other hub. As explained before, a node that cannot communicate through a given hub will observe a *notification omission discrepancy*: when a *delivery event* occurs, the node observes that the controller connected to that hub erroneously does not notify that event. Thus, in principle, the node can tolerate a fault by simply accepting as valid the transmission/reception notified by the controller that has no problems.

Note that if the controller that cannot communicate is the *non-tx controller*, the node need not even diagnose that controller as faulty. However, if the controller that cannot communicate is the *tx controller*, the node must eventually diagnose it as faulty and, then, rule it out for communicating. Otherwise, the node will not be able to transmit anymore and the fault will not be tolerated. To overcome this problem, the node initiates a *transmission timer* when it requests a transmission. If the timer expires before the *tx controller* notifies of a successful transmission, the node rules it out and uses the other controller for transmitting/receiving.

Additionally, we propose to rule out a CAN controller whenever its *Transmission Error Counter* (TEC) or its *Reception Error Counter* (REC) [ISO93] reaches a given threshold. This only allows to rule out a controller that cannot communicate and that detects errors. But it enhances the node's fault diagnosis capabilities, which will allow us to improve in the future the management strategy to deal with a wider range of faults, e.g. CAN inconsistency scenarios [PMJ00] and forged transmission/reception notifications.

11.8 Considerations on the cabling and bit rate

As stated in Section 5.6, the length of the cabling is an important factor in a distributed embedded system, mainly due to its cost in terms of wire and the limitations it imposes on the bit rate. When compared with a bus, CANcentrate demands a higher amount of cabling, since every node is connected to the hub by means of two dedicated links. However, as already explained in Section 5.6, signals travel in parallel to both hubs and then in parallel in all links back to the nodes. Hence, the maximum length applies only two every pair of links. This feature may represent

a substantial increase in the capacity to interconnect nodes when compared with a bus topology.

The increment of cabling is even bigger in ReCANcentrate because nodes are normally connected to two hubs. Nevertheless, since in ReCANcentrate the hubs are coupled, nodes are not required to be connected to both hubs for communicating. This allows to achieve higher dependability than CANcentrate without needing to duplicate the cost of the cabling.

Regarding the limitations on the bit rate, CAN imposes an inverse relationship between the length of the cable an the maximum bit rate, as a consequence of the synchronization at bit level among all nodes [ISO93]. In both CANcentrate and ReCANcentrate this synchronization is preserved, thus the same kind of relationship applies.

In CANcentrate it is needed to take into account the extra delay introduced by the hub (additional transceivers and internal gates) when dimensioning the bit time. In particular, from the point of view of signal propagation, the hub is equivalent to have extra cable length. In Section 5.6 it is explained which is the maximum bit rate, B', that could be achieved in a CAN bus with a bus length equal to the diameter of a CANcentrate star operating at a maximum bit rate of B (see Equation 5.1).

The comparison between ReCANcentrate an a CAN bus is slightly different. Note that in ReCANcentrate two nodes communicate simultaneously through different paths, which can include one or both hubs. Hence, in order to allow the bit value to settle before sampling, the bit time must take into account the slowest communication path in the network. In such a way, we define a ReCANcentrate equivalent bus as a CAN bus whose length is equal to the slowest communication path between two any nodes. Therefore the referred equation must be slightly modified for accommodating a parameter, n_h, that specifies the number of hubs included within the slowest communication path. Specifically, $n_h = 2$ or $n_h = 1$ if this path includes both hubs or not respectively. Equation 5.1 can thus be rewritten as follows:

$$B' = \frac{1}{t_{ps}} = \frac{1}{1/B - n_h \cdot t_h} = \frac{B}{1 - B \cdot n_h \cdot t_h} > B \tag{11.1}$$

Where t_{ps} is the bit time of the replicated star equivalent bus, and t_h is the delay introduced by the hub (310 ns approximately). Note that since a signal must go through the hub two times (from the transmitting node to the receiving node and viceversa), t_h includes twice the time a signal is delayed when crossing the hub.

The discussion above shows that both CANcentrate and ReCANcentrate are,

from a electrical signal transmission point of view, equivalent to a bus operating at a higher bit rate. This actually means that the length of the slower communication path in ReCANcentrate, operating at bit rate B, is the maximum length of standard CAN operating at bit rate B'. Moreover, the higher the bit rate, the larger the difference. For instance, if we consider that both hubs are included in the slower communication path and that $t_h = 310$ ns, the maximum length of a communication path in ReCANcentrate operating at $B = 1$ Mbit/s is equal to the length of a CAN bus operating at $B' = 2.6$ Mbit/s. In contrast, when $B = 125$ Kbit/s, the maximum length of a communication path in ReCANcentrate is equal to the maximum length of a CAN bus operating at $B' = 135.5$ Kbit/s, which implies a lower reduction in length.

11.9 Prototype implementation

In order to experimentally verify the proposed replicated star topology, we built a ReCANcentrate prototype as well as an experimental platform very similar to those we set up for testing CANcentrate (see Chapter 9).

Specifically, the ReCANcentrate prototype was built including two hubs and three CAN nodes. The internal part of the hubs (the Coupler Module and the Fault-Treatment Module) was implemented using the VHSIC Hardware Description Language (VHDL) and the Xilinx Spartan-3 XC3S1000 Field Programmable Gate Array (FPGA), within the XSA-3S1000 Board [X E04]. Only the interface (the Input/Ouput Module) of each hub with the media was implemented on a dedicated board (using the wire-wrap technique) with four pairs of PCA82C250 high-speed CAN transceivers [PHI00] and four RJ45 plugs (one for each transceiver pair) providing the connection for four hub ports, i.e. allowing the connection of three nodes and one interlink. UTP (Unshielded Twisted Pair) Category 5/5e/6 cables were used for the links and the interlink. Each uplink / downlink pair used two wire differential lines within the same cable. Similarly, one cable was used for the interlink, using a two wire differential line for each sublink.

Again, each CAN node was totally implemented using commercial off-the-self components, and basically includes a PIC microcontroller [Mic04], which incorporates one CAN controller, and four CAN transceivers. Each pair of transceivers was aimed at connecting the node with one of the hubs, following the schema specified in Section 5.3 and Figure 5.2 for interfacing a CAN node with a CANcentrate hub.

For managing the replicated channels we used the simplified approach referred in Section 11.4 and depicted in Figure 11.4, which is similar to the mechanism

proposed in [RVA99]. Despite limiting the fault-tolerance properties of ReCANcentrate, this approach is very simple to deploy and still allows verifying the main features of this architecture, namely the error detection, isolation and masking capabilities applied to both nodes and hubs.

11.10 Functional tests

Two types of functional tests were carried out to assess the functionality of ReCANcentrate under both fault-free conditions and in the presence of faults. These tests were carried out at the level of the hub VHDL design using the *ModelSim XE II 5.7g* simulation tool and at the physical network level using the ReCANcentrate prototype with different links/interlinks configurations. The bit rate was of 333 Kbit/s for all the functional tests.

During the tests with the ReCANcentrate prototype an arbitration was forced when transmitting every CAN frame. To do this, 3 CAN nodes were programmed to transmit frames with different data and lengths continuously in an infinite loop. Upon each transmission, two nodes always contended to access the medium. Reception of CAN messages was handled by interrupts, clearing the appropriate reception buffer.

In order to assess the fault-treatment functionalities of ReCANcentrate, faults were both simulated at the VHDL design level and injected into the physical network in different ways. One basic mechanism used to inject physical faults in the links and interlinks was to mechanically connect and disconnect the respective cables. This allowed injecting stuck-at-recessive as well as random bit-flipping faults. However, for more controlled fault injection, we built a dedicated board we call *Faulty renode*. This board is very similar to the *Faulty node* we used to inject faults in the prototype of CANcentrate (see Section 9.3 and Figure 9.3). The only difference between these two fault injectors it that the *Faulty renode* includes two link interfaces, whereas the *Faulty node* includes only one link. A given link interface basically consists in a pair of CAN transceivers that allow connection to a given hub port. In this way, the *Faulty renode* allowed injecting stuck-at-dominant faults and steady bit flipping streams at one or two hub ports simultaneously. Finally, a specific feature was added to the VHDL design of the hubs that allowed bit-flipping bits and stuck-at (dominant or recessive) bits to be sent through different hub ports when pressing a specific button on the corresponding hub's FPGA board.

In all the experiments the *diagnosis latency* was measured, i.e. the time from the very first bit of a stuck-at or from when a bit-flipping stream is injected until the

corresponding hub port is diagnosed as being permanently faulty. This latency is also called *isolation latency* when there are stuck-at-dominant or bit-flipping faults because these imply the isolation of the corresponding hub port. The *reintegration latency* was also measured in some cases. This is the time that elapses from the end of the last bit of a stuck-at or a bit-flipping stream that is injected until the hub re-enables the contribution of the corresponding hub port.

11.10.1 Experiments under fault-free conditions

As in the case of CANcentrate, the main aspects that have been tested under fault-free conditions are:

- Operation of the different state machines that constitute the hub.
- Calculation of the resultant frame upon all node contributions.
- Correct synchronization at bit level and at frame level.
- Assignation of the roles of the nodes after the arbitration phase.

At the VHDL design level several simulations were performed to verify these aspects. In all cases the hub operated correctly. Special attention was devoted to checking the correct operation of the different state machines that constitute the hub, as well as their correct mutual interaction.

Furthermore, the correct operation of ReCANcentrate at the physical network level was also observed. The prototype of ReCANcentrate was configured with two hubs and three CAN nodes, and the hubs were interconnected by a single interlink. This configuration was used throughout the experiments except where stated otherwise. The test was performed during more than 168 hours, during which nodes were able to correctly communicate.

11.10.2 Experiments under the presence of faults

The fault treatment capabilities of ReCANcentrate were assessed by means of different fault scenarios, i.e. scenarios involving faults. Again, as in CANcentrate, the main issues that were checked are the correctness of:

- Operation of the state machines that let us know how the hub is performing error detection and fault diagnosis.

- Detection of ports suffering stuck-at-recessive faults, as well as the detection and isolation of ports suffering stuck-at-dominant or bit-flipping faults.

- Reintegration of ports that became non-faulty, following the policy explained in Section 6.7.

Some basic fault scenarios were simulated at the level of the VHDL design, checking the behavior and the interaction of the different self-operating mechanisms that constitute the hub. In particular, stuck-at-recessive, stuck-at-dominant and bit-flipping faults were simulated as occurring at different ports and at different bit positions in the frames. For example, in order to cause a bit-flipping fault in a port, a bit-flipping stream was initiated during different frame fields. In all cases, the hub behaved according to its specifications.

As to tests at the physical level, a logical analyzer and a digital oscilloscope monitored the test pads. They showed the internal operation of the hub, the contribution of each node received at the respective port, and the value of the coupled signal. The physical network allowed testing many more fault scenarios than the simulation tool, as discussed next.

Stuck-at recessive faults at links and interlinks

Stuck-at-recessive faults were injected at both links and interlinks by disconnecting the respective links, one at a time, during communication. It was observed that some bit-flipping bits were always injected at a port when the respective link was disconnected, as expected. However these bit-flipping bits were not enough to make the hubs diagnose a bit-flipping fault. After these bit-flipping bits, the affected hub port received recessive bits only, and the hub correctly diagnosed it as being stuck-at-recessive. The communication between nodes was never disrupted.

During the execution of these tests, it was also observed that nodes were able to communicate with each other as long as each of them was connected at least to one hub. In particular, some links were disconnected in order to achieve the configuration of the network shown in Figure 11.1, in order to try causing a network partition. These were never observed, given the coupling of the two hubs forced by ReCANcentrate.

For injecting stuck-at-recessive faults at interlinks a different configuration was used with two CAN nodes, each one connected to both hubs, and two interlinks connecting the hubs. Stuck-at-recessive faults were injected at a given interlink by disconnecting it. Such fault injection was repeated many times and it was observed

that nodes always continued communicating correctly, with both hubs still coupled by the remaining non-faulty interlink. Moreover, each hub always correctly indicated the stuck-at-recessive failure of the corresponding interlink.

Stuck-at-dominant and bit-flipping faults at links

In order to test the isolation of stuck-at-dominant and bit-flipping faults at links, each hub was configured to diagnose a stuck-at-dominant condition when monitoring 24 consecutive dominant bits and a bit-flipping fault when counting 24 bit-flipping errors.

The *Faulty renode* was used to transmit periodic dominant and recessive levels in two different ways, as depicted in Figure 11.7: to the port of one hub only (a), and simultaneously to the ports of both hubs (b). When the frequency of the signal transmitted by the *Faulty renode*, i.e. the *faulty signal*, was low enough compared to the bit rate, the recessive and dominant values of this signal lasted many bit times. The affected hub(s) alternately detected the port as being stuck at dominant and then reintegrated it, as expected. The faulty signal was injected continuously during 15 hours with the frequency ranging from 62 Hz to 6.6 Khz, thus injecting between 62 and 6600 stuck-at-dominant faults per second. All faults were correctly diagnosed and isolated by the affected hub(s). At the lower frequencies, the hub(s) were also able to reintegrate an isolated port during the recessive pulses. During all these tests, the global communication activity was disturbed only during the latency needed to diagnose an affected port as being faulty and isolate it. The diagnosis and reintegration latencies lasted 73 us and 5.2 ms, respectively.

The faulty signal was also injected at a higher frequency so that the dominant/recessive pulses lasted from anywhere between a few bit times to less than one bit time. In these situations, the affected hub(s) correctly diagnosed the port as being permanently bit-flipping. The experiments used 100 bit-flipping streams, each one causing a bit-flipping fault. The frequencies of the periodic bit-flipping signal ranged from 6.6 Khz to 2.5 Mhz, toggling between dominant and recessive levels in a few bits times to several times per bit. The diagnosis latency to detect the bit-flipping fault depended on the frequency. The following values were measured 20 times for each frequency: 609 us at 10 Khz, 150 us at 100 Khz and 246 us at more than 1 Mhz. Finally, bit-flipping ports were also reintegrated when the bit-flipping link was physically disconnected. In this case, the reintegration latency was also around 5.2 ms.

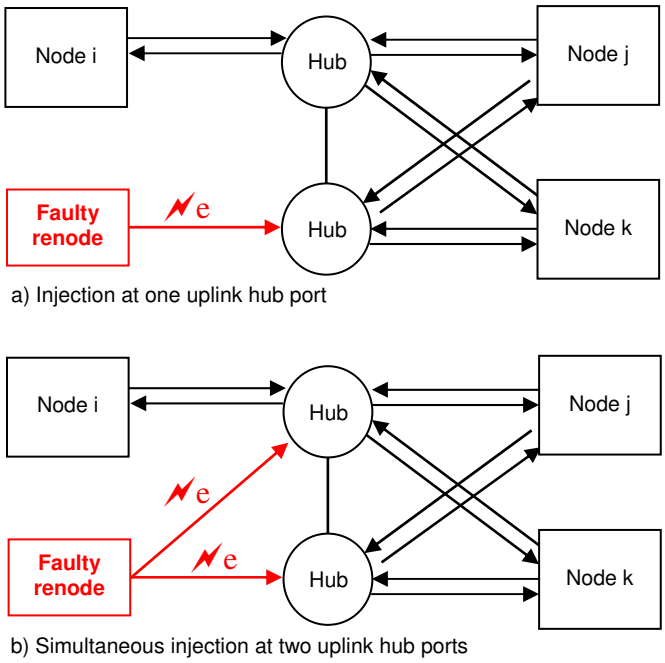

Figure 11.7: Injection of stuck-at-dominant/bit-flipping faults at links

Stuck-at-dominant and bit-flipping faults at interlinks

Stuck-at-dominant and bit-flipping faults were injected at interlinks using two different network configurations: one with 2 nodes and 2 interlinks, one of which was replaced by the *Faulty renode* (Figure 11.8); and the standard one with 3 nodes and 1 interlink that was replaced by the *Faulty renode* (Figure 11.9). For both configurations, each hub was programmed to diagnose a stuck-at-dominant and a bit-flipping fault at an interlink when detecting 72 consecutive dominant bits and 72 bit-flipping errors, respectively.

Using the first configuration, stuck-at-dominant and bit-flipping faults were injected at the hubs' interlink ports, either to one or both sublinks, while nodes were correctly communicating. Stuck-at-dominant faults were injected continuously during 7 hours, using a faulty signal frequency from 62 Hz to 2.3 Khz. The isolation latency was always around 216 us. In addition, 300 bit-flipping streams were injected with the faulty signal frequency ranging from 2.3 Khz to 2.5 Mhz. When a bit-flipping stream was injected in one sublink only, the affected hub correctly isolated and reintegrated the corresponding faulty port. When injecting in

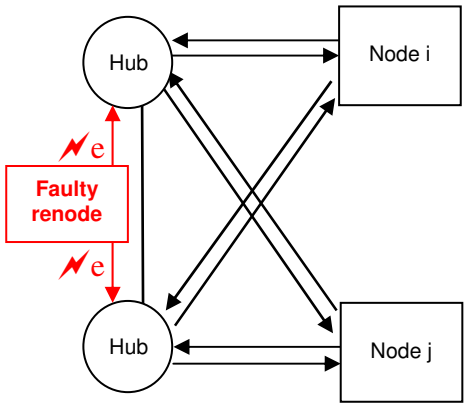

Figure 11.8: 1st injection of stuck-at-dominant/bit-flipping faults at interlinks

both sublinks, both hubs correctly isolated and reintegrated the faulty interlink, each one at its respective hub port, nearly simultaneously. The measured isolation latency also depended on the frequency of the faulty signal, being 2.6 ms at 10 Khz, 476 us at 100 Khz and 850 us at 1 Mhz. The reintegration latency was always about 4.2 ms after the end of either a stuck-at-dominant or bit-flipping fault.

Other experiments were conducted using the second configuration of the network (Figure 11.9). Initially, the three CAN nodes communicated using both hubs, even though the hubs were not coupled. This is because each node also transmitted simultaneously to both hubs, thus receiving the same bits through both hubs simultaneously. The *Faulty renode* started to transmit a stuck-at-dominant or a bit-flipping signal to both hubs simultaneously at a randomly chosen instant. The hubs always correctly isolated the faulty interlink at almost the same time. 50 stuck-at-dominant and 50 bit-flipping streams were injected. The isolation latencies were exactly the same as for the first network configuration.

Stuck-at-dominant and bit-flipping faults at a hub

Hub failures were also injected using the special feature implemented in VHDL, which allows the hub to send stuck-at and bit-flipping independent streams through each port. Nevertheless, the simplified schema implemented at each node for connecting it to both hubs (see Figure 11.4) limited the type of faults that actually could be injected at a hub. As already explained, in this schema the node couples both its downlinks by means of an AND gate and has the ability of isolating any one of them suffering from a stuck-at fault. However, the node has no mechanism

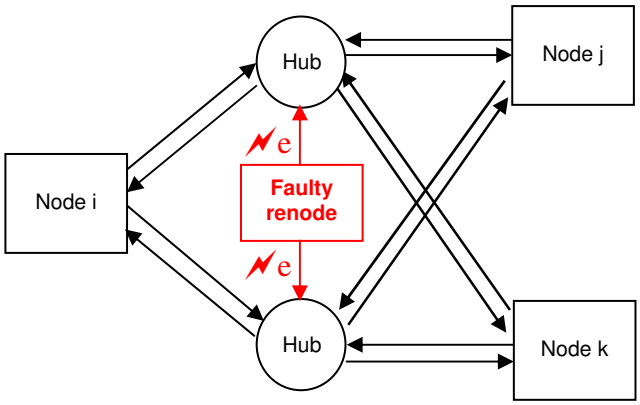

Figure 11.9: 2nd injection of stuck-at-dominant/bit-flipping faults at interlinks

to isolate a downlink that becomes bit-flipping. Thus, it was not possible to inject faults in a hub that led it to send bit-flipping bits to nodes.

Because of this limitation, the tests were carried out by first sending stuck-at bits through all ports, and secondly bit-flipping bits to the interlink ports only. In the first case, the non-faulty hub and the nodes always isolated the faulty hub. Additionally, the non-faulty hub indicated that the contributions of the other hub were stuck-at. In the second case, the non-faulty hub correctly isolated the faulty interlinks, diagnosing them as being bit-flipping. The isolation latencies observed at the non-faulty hub were equal to those measured when we directly injected faults at the interlinks. Moreover, analogously to the case of the hubs, the nodes were configured to isolate a stuck-at-dominant downlink when monitoring 24 consecutive dominant bits. This fact leaded us to observe that the time each node needed to isolate a stuck-at-dominant hub coincides with the latency with which a hub isolates a stuck-at-dominant uplink.

11.11 Performance measurements

As in the case of CANcentrate, we measured the extra delay introduced by the core and the transceivers of the ReCANcentrate hub. We obtained the same results as in CANcentrate: 35 ns for the core and 120 ns for each transceiver, which means that the total hub extra delay is of 155 ns (the total delay only includes one time the transceiver delay). Moreover, we observed again that this delay does not visibly depend on the number of hub ports.

The performance of the ReCANcentrate prototype was evaluated with the network configuration we used for the functional tests under fault-free conditions, i.e. the network included three nodes and the hubs were interconnected by means of one interlink. Arbitration was also forced for each transmitted frame, with the network bandwidth used at its maximum capacity. The tests basically assessed the maximum cable length, i.e. star diameter, which can be achieved with a given bit rate. The diameter was taken as the distance between the two farthest nodes, including the length of the interlink plus the length of the two longest links.

Due to practical limitations the maximum bit rate used was 625 Kbit/s. At this bit rate no errors were observed during two hours of constant operation with a star diameter of 25 m. We will refer to this two-hour period without observing errors as *normal communication*. Increasing the diameter to 30 m caused sporadic errors at an average rate of one every 5 ms. Notice that the maximum length of a CAN bus operating at 625 Kbit/s would be around 79 m [CiAa]. Moreover, note that if we apply Equation 11.1 (with $n_h = 2$ hubs and $t_h = 2 \cdot 155 = 310$ ns) to calculate the bit rate, B', of an equivalent CAN bus with a bus length equal to the ReCANcentrate diameter, we obtain that B' would be of 1 Mbit/s. The maximum CAN bus length achievable at this bit rate should be of 30 m [CiAa], which is similar to the diameter at which ReCANcentrate should start experiencing sporadic errors.

Another bit rate vs star diameter results are the following ones. At 500 Kbit/s a diameter of 57 m resulted in normal communication, while a sporadic error occurred every 10 ms on average with a diameter of 58 m and a disruption of the communication occurred for a diameter of 70 m (the maximum length of a bus operating at 500 Kbit/s would be around 100 m). At 454.5 Kbit/s normal communication was observed with 68 m while 70 m caused a sporadic error every 10 ms on average and 72 m disrupted the communication. Finally, at 416.6 Kbit/s normal communication was observed with 78 m while 80 m caused an error frame every 10 ms and 82 m disrupted the communication.

11.12 Conclusions

CANcentrate improves reliability of CAN-based systems by means of enhanced error-detection and fault-treatment mechanisms. More specifically, CANcentrate reduces the multiple severe points of failure that appear in a CAN bus to one single point of failure, i.e. the hub.

However, more demanding highly-dependable systems require to eliminate any single point of failure from the communication system. To achieve this while tak-

ing profit from the advantages already achieved by CANcentrate, we proposed a new communication infrastructure, called ReCANcentrate, that relies on a replicated star topology, and which overcomes the drawbacks of any other replicated communication system already proposed for CAN.

This infrastructure basically includes two interconnected CANcentrate hubs. In this way, it eliminates any single point of failure and extends the fault-treatment capabilities of CANcentrate to the overall of the communication system. Furthermore, ReCANcentrate is still compatible with commercial off-the-shelf CAN components and can be used with any CAN-based protocol.

Beyond the good properties of CANcentrate, ReCANcentrate exhibits additional advantages. First, it provides a synchronization mechanism for transmitting and receiving frames in both stars, which is very helpful for managing duplicates and omissions, as well as for treating faults in a replicated event-triggered communication system. Second, nodes may be able to communicate as long as one of its links remain non-faulty and is connected to a non-faulty hub. Third, it prevents network partition fault to occur. Finally, nodes are not required to be connected to both hubs for communicating, thereby achieving higher dependability than CANcentrate without needing to duplicate the cost of the cabling.

In this chapter, we describe the internal structure of the hub and its new functionalities; some possible node's architectures, focusing on the way in which each node should manage the replicated traffic and treat faults; some issues concerning the cabling and the bit rate; as well as the implementation and test of our first prototype of ReCANcentrate.

Chapter 12

Reliability evaluation of ReCANcentrate

12.1 Introduction

In Chapter 10 we quantified the improvement of system reliability that can be achieved when using CANcentrate instead of the CAN bus when permanent hardware faults can occur. The results quantitatively corroborate the benefits that the CANcentrate's error-containment mechanisms can yield in terms of reliability for *fault-tolerant/accepting* (FT/A) systems. However, they also show that a simplex star topology such as CANcentrate slightly reduces the reliability of *non-fault-tolerant / accepting* (NFT/A) systems. Moreover, as already mentioned, the degree of system reliability achieved by CANcentrate could be not enough for some applications and the presence of a single point of failure the hubs represents unacceptable.

Therefore, in order to improve the reliability of not only FT/A systems, but also of NFT/A ones, as well as to satisfy the requirements of highly reliable applications, we proposed a redundant active star topology called ReCANcentrate, whose design and first implementation were addressed in Chapter 11. The key feature of ReCANcentrate is that, besides error-containment, it provides tolerance to hub failures, as well as to faults that affect one of the connections of each node to a given hub (no matter to which hub). In this way, ReCANcentrate eliminates the single point of failure the hub of CANcentrate represents and it theoretically compensates the bigger probability of suffering from faults in a star topology.

As also mentioned, other technologies such as TTP/C [ABST03] and FlexRay

[Fle05] have already adopted a replicated star topology to improve error containment and fault tolerance. However, despite this interest, and as happens with simplex star topologies, no one has quantitatively demonstrated that replicated star topologies improve reliability when compared with other communication infrastructures. Current chapter is thus devoted to quantitatively assessing how a replicated star topology such as ReCANcentrate can improve the system reliability when compared with a bus and a simplex star topology such as CAN and CANcentrate.

For this purpose, we model the reliability of a system relying on ReCANcentrate using the Stochastic Activity Networks (SANs) formalism. In particular, we follow the same modelling strategy we proposed for assessing the reliability of a system relying on CAN and on CANcentrate. This fact was already pointed out in Chapter 10, where we explained that most of the modelling decisions and assumptions described therein were done to be also appropriate for the case of ReCANcentrate. Anyway, notice that we make additional assumptions that are exclusively related to a ReCANcentrate-based system and, again, these new assumptions are made ensuring that results are not biased towards ReCANcentrate.

It is noteworthy that, as we did in Chapter 10, we model the reliability of a system relying on ReCANcentrate only when permanent hardware faults can occur. As already explained, a dependability evaluation that takes into account transient faults is postponed to a later work. Similarly, the comparison between the system reliability achievable with ReCANcentrate and other topologies such as replicated buses is also proposed as a future work.

12.2 Metrics

In order to compare the reliability of a system relying on CAN, CANcentrate and ReCANcentrate we use the same metrics we adopted for comparing CAN and CANcentrate in Chapter 10. On the one hand and for quantifying the reliability of NFT/A systems, we use the *non-fault-tolerant/accepting system reliability* (NFT/AR), which we defined as the probability that all nodes can correctly operate and communicate with each other over time (see Section 10.2). On the other hand, we quantify the reliability of FT/A systems by means of the *fault-tolerant/accepting system reliability* (FT/AR$_k$), which stands for the probability with which at least $N - k$ of N nodes can correctly operate and communicate among them throughout time. In particular, as we did for CANcentrate, we measure the FT/AR$_1$, i.e. the reliability of FT/A systems that are robust to the failure or disconnection of at most 1 out of N nodes.

12.3 Modelling assumptions

In order to compare the reliability of equivalent systems that rely on CAN, CANcentrate and ReCANcentrate we use the models of CAN and CANcentrate described in Chapter 10 and we built an additional model for ReCANcentrate. For this new model, we consider as valid all modelling assumptions we made for the case of CANcentrate, which were thoroughly described in Section 10.4. However, we had to consider additional assumptions related to specific ReCANcentrate aspects, which are not connected with the case of CANcentrate.

We made all assumptions guaranteeing not only that results do not favor ReCANcentrate in the comparison with CAN, but also in the comparison with CANcentrate. Moreover, as we did for the models of CAN and CANcentrate, most of these additional assumptions are reflected as parameters, thereby allowing to perform sensitivity analyses with respect to them. The parameters the model of ReCANCentrate has in common with the models of the CAN bus and CANcentrate are specified in Table 10.4; whereas Table 12.1 shows the parameters that are specific to ReCANcentrate.

12.3.1 Implementation assumptions

The first additional assumption we made for ReCANcentrate consists in deciding what is the number of interlinks that interconnect its two hubs. Since the motivation of having several interlinks is to eliminate the single point of failure one interlink would represent, we propose to consider just two interlinks (see parameter *numInterlinks* of Table 12.1).

As concerns the length of the links and interlinks of ReCANcentrate, we follow the same approach as for CANcentrate (see Section 10.4.1). There, we considered that if an ensemble of nodes need to be interconnected by means of a CAN bus that is Lb meters long, then each link of CANcentrate needs to be $Lb/2$ meters long to interconnect all these nodes. For the case of ReCANcentrate we assume the same link length and, in addition, we consider that the length of each interlink is also equal to the half of Lb. These are pessimistic assumptions for ReCANcentrate. On the one hand, as explained in the section referred above, this link length is overestimated. On the other hand, the supposed interlink length is pessimistic because even with interlinks of 0 meters, a ReCANcentrate star whose links are $Lb/2$ meter long can interconnect an ensemble of nodes that require a bus length of Lb meters.

The last implementation assumption regards the schema that each node uses to connect to both hubs. As explained in Section 11.4, there are different ways to do

this connection. One possible option is to couple both downlinks and uplinks at the node, which only includes one CAN controller. This schema (see Figure 11.4) allowed us to easily build a first prototype of ReCANcentrate, with which we experimentally verified and assessed the main fault-tolerance features of both nodes and hubs. However, this node's architecture limits the degree of fault-tolerance that can be achieved with ReCANcentrate (see Section 11.4 for further details about this issue). For this reason, we advocate using the other architecture we proposed and which is depicted in Figure 11.3. As already explained, the node is basically constituted by one microcontroller and two CAN controllers (one for connecting to one hub and another one for connecting to the other hub). This schema allows each node to tolerate the failure of any of the components that connect it to a given hub. The details about the way in which a node that is based in this architecture manages the traffic it observes at both hubs, as well as how it tolerates faults can be found in Section 11.7.

Since this last connection strategy is the one that we proposed for achieving a high degree of fault tolerance, we assume that each ReCANcentrate node is based on it for communicating. In this sense, note that the reliability evaluation herein presented throws light on the suitability of this node architecture prior to the implementation of a new ReCANcentrate prototype that relies on it.

12.3.2 System components and entities

As regards the components and entities that are supposed to constitute the system, we do not introduce any change with respect to what is considered for CAN and CANcentrate (see Section 10.4.2 for a detailed explanation about the components and entities that constitute a system relying on CAN and on CANcentrate). However, it is important to highlight that now an Attachment entity, which embraces a piece of a CAN cable and a pair of connectors, is used to represent not only an uplink, a downlink or a bus section, but also a given sublink, i.e. one of the two independent and identical links that are included in an interlink and which carries the contribution of one hub to the other.

12.3.3 Failure mode assumptions

Regarding the failure mode assumptions we have made so far (see Section 10.4.4), it is not longer suitable to consider that a fault affecting the Hub Core provokes the failure of the overall system. Notice that, in ReCANcentrate, the failure of a hub can be treated by the hub that remains non-faulty and by the nodes. Therefore, since the ability of a hub and nodes to diagnose a hub as faulty depends on the

way in which the failure manifests, it is mandatory to model different hub's failure modes and their respective proportions.

In order to afford this new necessity, we model the Hub Core's failure modes following the same idea as for the rest of entities. This consists in supposing that the Hub Core exhibits a 0% of out-of-fault-model (ofm) failures, and that the rest of its failure modes, i.e. stuck-at-recessive, stuck-at-dominant, and bit-flipping, are equiprobable. More specifically, we consider that when the Hub Core fails in a way that is included in our fault model, it transmits a stream composed of the errors corresponding to the specific type of fault through all its outgoing ports, i.e. through the ports corresponding to its downlinks and to the sublinks that carry its contribution to the other hub. In principle, this assumption can be considered as reasonable, given that the Hub Core has no mechanism to treat its own faults and, hence, it cannot prevent that errors propagate through all its outgoing ports. However, this assumption may also be considered as pessimistic for ReCANcentrate, because a fault affecting the Hub Core could lead it to transmit the corresponding stuck-at or bit-flipping stream through some ports only. Moreover, as pointed out in Section 10.4.4, a Hub Core could even exhibit a more benign failure mode, e.g. it could unfairly isolate a correct port.

Another important aspect related to the existence of two hubs in ReCANcentrate and that must be analyzed is the way in which a hub propagates the errors it receives from a faulty port it cannot isolate. Notice that if the faulty port corresponds to an uplink, then the hub will broadcast the errors through all its downlinks and its outgoing sublinks. Conversely, in case the faulty port belongs to an incoming sublink, it will retransmit the errors through the downlinks only. In order to simplify the model, we do not differentiate between these two cases and we just consider the worst option for ReCANcentrate, i.e. that the hub propagates the errors through all its downlinks and its outgoing sublinks. Notice that this way of broadcasting errors coincides with the way in which the errors generated by a Hub Core failure pollute the system, i.e. the hub transmits error through all its outgoing ports. Therefore, from now on, we will consider that a hub is faulty not only when its Hub Core fails, but also when it is not able to isolate a faulty port.

12.3.4 Coverage assumptions

Finally, the last assumptions that are specific to ReCANcentrate are related to the fault-tolerance coverages that characterize it. Notice that we assume as valid all the fault-tolerance coverages proposed in Section 10.4.5 for the models of CAN and CANcentrate. On the one hand, we assume the same default value for the *sysFauTolCov* coverage, which reflects the capacity of an FT/A system to actually

accept or tolerate the failure or disconnection of a node, provided that it can do so. On the other hand, we accept as correct all default values we proposed for the error-containment coverages associated with the fault-treatment mechanisms of the CAN controller and the hub, i.e. the default values of *nodeIOInFauProp*, *ctrlItselfIsoCov*, *ctrlFlipCov* and *flipLnkCov* (see Tables 10.4 and 10.6).

However, since ReCANcentrate includes further fault-tolerance mechanisms, we define some additional coverages to characterize them. The first set of these new coverages quantify the capacity of the hub to contain errors at its sublink ports, i.e. at the ports corresponding to each one of its incoming sublinks. In this sense, we differentiate between the errors generated by a fault occurring at the Hub Core of the other hub, a fault affecting the sublink itself and a fault affecting one of the ports of the other hub. As concerns the ability of the hub to contain errors generated by a fault affecting the other hub's core or a sublink, notice that the units that are responsible for diagnosing faulty sublink ports are the Hub Enabling/Disabling units. The functionalities of these units are very similar to the ones performed by the Enabling/Disabling units, which are aimed at detecting faulty uplink ports. Therefore, we assume that the coverages with which the hub diagnoses faults affecting the other hub's core or the sublinks are the same with which it diagnoses faults affecting the uplinks ports: 100% for stuck-at faults and 95% for bit-flipping faults. Anyway, we parameterized the coverage that specifies the probability with which the hub diagnoses a bit-flipping Hub Core or sublink at the corresponding port/s. The corresponding parameter is called *flipSlnkCov* (see Table 12.1).

A deeper analysis is required to characterize the coverage with which a hub contains the errors generated by a fault that affects a port of the other hub. First, notice that errors only propagate from one hub to the other, if the hub that suffers from a fault in one of its ports is not able to isolate that port. Since a hub isolates stuck-at faults with a perfect coverage, the only faults whose errors can propagate from one hub to the other are those that manifest as bit-flipping. Thus, from now on, let us focus on bit-flipping faults only. Imagine that the two hubs of ReCANcentrate are referred to as *hub A* and *hub B*, and that the *hub A* does not successfully isolate a bit-flipping stream it receives through one of its ports. If this happens, the other hub, the *hub B*, will receive, though all the incoming sublinks, a bit-flipping contribution from the *hub A*. Thus, from then on, the *hub B* considers the *hub A* as faulty and, hence, one may wonder whether or not the *hub B* has any capacity for isolating the *hub A*. In principle, the *hub B* has the same capacity for diagnosing bit-flipping faults as the *hub A*. Thus, since the *hub A* could not diagnose the bit-flipping fault, it seems reasonable to assume that the *hub B* cannot do it either. However, the bit-flipping contribution that the *hub B* receives is not equal to the bit-flipping contribution that the *hub A* observes at its faulty port. Conversely, what

the *hub B* monitors at its incoming sublinks is the result of coupling, at the *hub A*, the original bit-flipping stream with the error frames transmitted by the *hub A* and the CAN controllers connected to it. Since the erroneous stream the *hub B* receives is different from the original one, we believe that the *hub B* is able to diagnose the bit-flipping fault at the sublinks with certain coverage. We call this coverage *flipSlnkPropCov* (see Table 12.1). In particular, we decided to assume a conservative value of 50% for this coverage. We think that a *flipSlnkPropCov* of the order of the 95% could be considered as optimistic, since it exits the possibility that the original bit-flipping stream does not compel the *hub A* and the CAN controllers connected to it to transmit error frames.

Besides the new coverages that quantify the capacity of the hub to isolate its sublink ports, we define coverages for other fault-tolerance mechanisms ReCANcentrate is provided with. Some of these fault-tolerance coverages represent the probability with which the node can tolerate a failure that prevents it from communicating through a given star, so that it can continue communicating using the other star. Such a fault can be either a fault that affects any of the entities that connect the node to one hub, e.g. the CAN controller, or a fault that affects the hub itself. In order to define these coverages and decide reasonable values for them, it is necessary to analyze the mechanisms that the node uses to tolerate this kind of faults. For this purpose, we have to differentiate between the situation in which the hubs are decoupled and the situation in which they are not, since the specific fault-tolerance mechanisms the node uses are different in both cases.

We define a coverage called *decConnCov* (see Table 12.1) for characterizing the probability with which a node tolerates a fault that prevents it from communicating through one star when the hubs are decoupled. Since we have not proposed any mechanism that allows a node to do that, we assume a default value of 0% for this coverage. However, our model makes it possible to carry out sensitivity analyses with respect to different values of this coverage in order to assess its influence on the reliability achievable by a ReCANcentrate-based system.

Similarly, we have defined a coverage called *connCov* (see Table 12.1) that specifies what is the probability that a node tolerates a fault that prevents it from communicating through one star when the hubs are coupled. The mechanisms the node uses to tolerate these faults are quite simple and they were described in Section 11.7.2. Basically, the node tolerates the situation in which it cannot communicate through a given hub by merely accepting as valid the transmission/reception notified by the controller that has no problems for communicating. Additionally, the node can diagnose, almost in all cases, that it cannot communicate through a given hub by simply using the fault-diagnosis capacities of its CAN controller that is connected to that hub. In particular, this diagnosis is essential for the node to

detect when its *transmission controller* cannot communicate and, then, to start using its surviving CAN controller as the new *transmission controller* (otherwise the node wouldn't be able to transmit any more). In this sense, recall that there is only one case in which a node cannot rely on the fault-diagnosis capabilities of its *transmission controller*. This happens when the *transmission controller* crashes, so that it does not notify its faulty status to the node. Fortunately, the node can also easily overcome this problem by simply associating a given time-out to every transmission it requests through its *transmission controller*. Given all these considerations, we believe that it is reasonable to assume a value for *connCov* equal to the probability with which a CAN controller diagnoses a bit-flipping fault, i.e. 95%. Moreover, this can also been considered as pessimistic for ReCANcentrate, because a node can be prevented from communicating through a given hub due to a stuck-at fault affecting one of its downlinks and, in this case, the corresponding CAN controller would diagnose the fault with a coverage of 100%.

The last new coverage that has to be taken into account for ReCANcentrate is the probability with which its nodes successfully star communicating with each other using two independent stars when the hubs become decoupled. We refer to this coverage as *decCov* (see Table 12.1). Again, since we have not proposed any mechanism that allows nodes to achieve this behavior, we assume a default value of 0% for *decCov*. Nevertheless, since we believe that it would be very valuable to evaluate how this coverage affects the reliability of a system relying on ReCANcentrate, our model allows performing sensitivity analyses with respect to values of *decCov* different from 0%.

12.4 ReCANcentrate model

Current section is devoted to thoroughly describing how we modelled the dependability of a system that relies on ReCANcentrate. As already mentioned, we specified this new model using the same formalism as for the previous ones, i.e. the *Stochastic Activity Network* (SAN) formalism. Once again, we used the Moëbius software [SoT04] to build and analytically solve the model of ReCANcentrate.

Next subsections are organized as follows. First, we begin explaining the general modelling strategy. Then, we will point out some important remarks that can help the reader in understanding the model. Finally, we describe each one the SANs submodels that compose the overall ReCANcentrate's model.

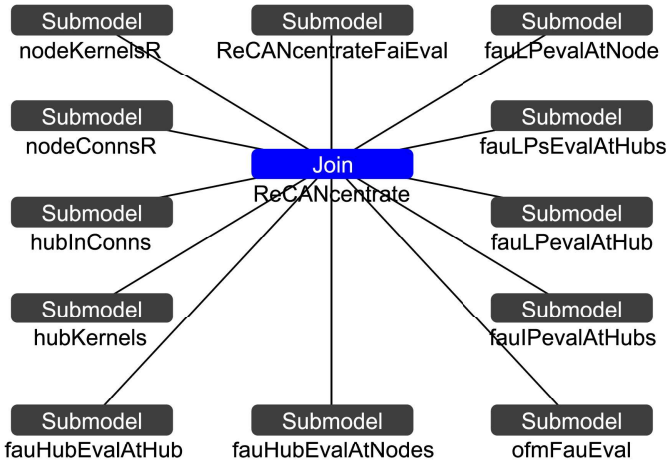

Figure 12.1: ReCANcentrate model

12.4.1 Modelling rationale

The general strategy we followed to model the dependability of a system that relies on ReCANcentrate is basically the same one we used to model the dependability of a system relying on CAN and CANcentrate, which we explained in Section 10.6.3. In this sense, the model of ReCANcentrate is a composition of three types of submodels we call *regions submodels*, *coverage submodels* and *evaluator submodels*. Figure 12.1 depicts the general structure of the ReCANcentrate model.

The *regions* submodels play the same role as in CAN and CANcentrate: each one of them represents all the error-containment regions of a given type (see Section 10.6.3). Specifically, as can be seen in Figure 12.1, the overall ReCANcentrate's model includes the following regions submodels: *nodeKernelsR*, *nodeConnsR*, *hubInConns* and *hubKernels*. The two first ones of these submodels respectively represent all the Node Kernel and all the Node Connection regions of ReCANcentrate. They are analogous to the *nodeKernelsT* and *nodeConnsT* of CANcentrate's model and to the *nodeKernelsB* and *nodeConnsB* of the model of the CAN bus. However, it is important to recall here that each Node Connection region in a star comprises all the entities a node needs to be connected to a given hub, i.e. one Controller, two Node IOs, two Attachments, two Hub IOs and four Terminations. This implies that a ReCANcentrate-based system includes two Node Connection regions per node, whereas a CANcentrate-based system includes only one for each node. As a consequence, the *nodeConnsR* submodel of ReCANcentrate models double the quantity of Node Connection regions than the *nodeConnsR*

submodel of CANcentrate.

The third submodel, *hubInConns*, represents faults happening at each one of the *Hub Interconnection* regions that interconnect both hubs. We defined the Hub Interconnection region specifically for ReCANcentrate. It includes all the entities that constitute a given sublink: one Attachment, two Hub IO and two Termination entities. More specifically, the Attachment entity represents the CAN cable and the pair of straight connectors of the sublink; each Hub IO entity includes the components that its hub uses to interface the sublink (a transceiver basically); and each Termination represents one of the two terminations that are used to prevent signal reflections at the extremities of the sublink.

The last regions submodel is *hubKernels*, which models faults affecting in any of the two Hub Kernel regions. Notice that conversely to the *hubKernel* submodel of CANcentrate, which represents faults happening in the single Hub Kernel of CANcentrate, *hubKernels* models faults occurring at any of the two Hub Kernels of ReCANcentrate.

The basic structure of a regions submodel of ReCANcentrate is depicted in Figure 10.6. As can be seen there, it is equal to the structure of this type of submodel in the case of CAN and CANcentrate (see Section 10.6.3). A regions submodel of ReCANcentrate still includes the place *okRegions*, whose marking represents the number of regions of a given type that are not faulty; the activity *regionFailure* that models the Time To Failure of an entity located at any of these regions; and the places, fm_i, that represent how the fault manifests.

Special emphasis should be put on these last places. When a fault occurs in a given region, the activity *regionFailure* decides the failure mode with which the fault manifests and sets a token in the corresponding fm_i. In order to choose the way in which the fault manifests, the cases' proportions of *regionFailure* are calculated taking into account the proportion with which the entities that constitute the region exhibit different failure modes. In addition, as indicated in Section 10.6.3, the activity *regionFailure* of the submodel that represents all the Node Connections of ReCANcentrate, i.e. of the *nodeConnsR* submodel, takes into account not only the failure mode proportions of the entities that constitute the Node Connection that fails, but also the fact that the errors generated by such a fault can be contained to some extent by the CAN controller placed at that Node Connection. This feature is depicted in Figure 10.6, where the error-containment capabilities of the CAN controller are also used for calculating the cases's proportions. More specifically, if a CAN controller successfully isolates a fault affecting its Node Connection, the activity *regionFailure* decides that the Node Connection manifests as stuck-at-recessive.

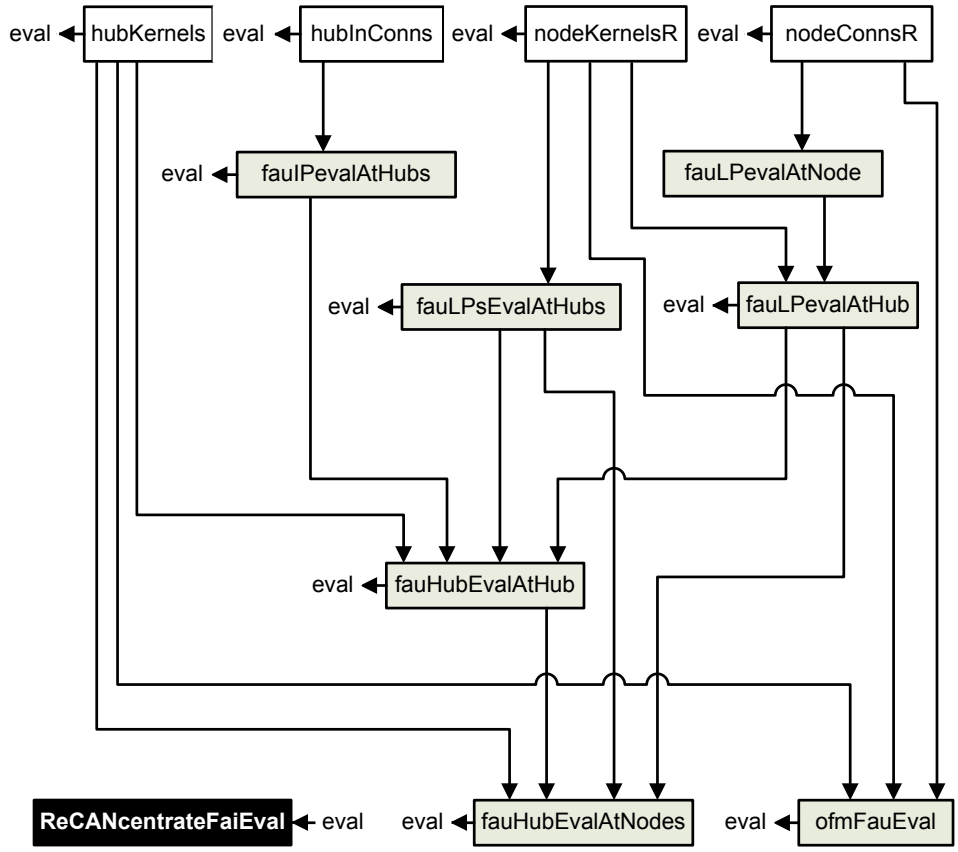

Figure 12.2: Paths of the coverage process

Anyway, like in the models of CAN and CANcentrate, the occurrence of any fault initiates a sequential process that models how the errors generated by that fault propagate, how they are contained and, if appropriate, how the fault is tolerated. We refer to this process as the *coverage process* (see Section 10.6.1). If the fault happens in a Node Connection, part of this process is carried out by the activity *regionFailure* of the corresponding submodel, which as just said takes into account the error-containment capabilities of the CAN controller placed at that Node Connection. The rest of the coverage process is carried out by different coverage submodels, each of which represents, more or less, a specific error-containment or fault-tolerance mechanism of ReCANcentrate. These models take over when the regions submodel set a token in any of the places fm_i.

Figure 12.2 depicts a diagram that represents the way in which the different

submodels of ReCANcentrate collaborate to implement the different paths of the coverage process that can be executed when a region fails. The regions and the coverage submodels are represented by means of clear and shaded boxes respectively. Each one of the paths of the diagram starts at a given regions submodel, since each coverage process begins when a fault occurs in a given region. Basically, each coverage submodel evaluates whether or not a given mechanism contains the errors, or whether or not such a mechanism tolerates the fault. Depending on the results obtained by the mechanism, the coverage submodel finishes the process (this is indicated in Figure 12.2 by an arrow connected to the word *eval*) or compels another appropriate coverage submodel to proceed with the evaluation. The coverage process continues until the errors are contained (or the fault is tolerated) by a given mechanism, or until all the error-containment and fault-tolerance mechanisms are exhausted. In next sections, we will explain what coverage submodels are involved in these processes when a fault affects each type of region, and the actions these submodels perform.

However, at this point, let us highlight some characteristics of the *ofmFauEval* coverage submodel, which is the one that evaluates whether or not the errors generated by an ofm fault propagate throughout the system. First, notice that the presence of *ofmFauEval* may seem counterintuitive, since there is no mechanism that can diagnose an ofm failure. However, the errors an ofm failure generates cannot pollute the system if it occurs in an already isolated region. For instance, the incorrect messages sent by a CAN controller that suffers from a babbling-idiot fault cannot affect any node if this CAN controller is already isolated by its node and the hub it is connected to. Second, it is noteworthy that *ofmFauEval* is the only coverage submodel that does not represent any specific fault-treatment or fault-tolerance mechanism. This is because there is no mechanism that can diagnose ofm faults. Finally, as can be seen in Figure 12.2, each regions submodel is connected to *ofmFauEval* (this connection is represented by the word *ofm*). This is because all regions are likely to suffer from an ofm fault.

During the coverage process and depending on whether or not the fault is successfully isolated and/or tolerated, *coverage submodels* update a set of places that make it possible to calculate how many nodes can communicate among them. These places are then used by the evaluator submodel *ReCANcentrateFaiEval* (see Figure 12.2) to decide wether or not the entire system is faulty. Specifically, it considers that an overall failure happens if there are few than a minimum number of nodes that can communicate among them. Once again, this number is set up by means of the parameter *kSevere* (see Table 10.4) so that the model can be configured to measure the NFT/AR and different degrees of FT/AR. The name and the meaning of each one of these places will be described in detail in the following

sections, specially in Section 12.4.3. Finally notice that *ReCANcentrateFaiEval* takes its decision at the end of the coverage process. This is shown in Figure 12.2, where *ReCANcentrateFaiEval* is represented as a black box where the word *eval* is connected to.

12.4.2 Some important preliminary remarks

Before continuing with the explanation of each ReCANcentrate's submodel, let us point out some important remarks. First of all, for the sake of clarity, let us refer to the two hubs of ReCANcentrate as *hub A* and *hub B* from now on. In particular, this notation will facilitate the explanation of some submodels of ReCANcentrate that are somehow symmetric. More specifically, there are submodels that reflect the actions carried out by both hubs. Thus, since both hubs are equal to each other, these submodels are symmetric in the sense that they model actions performed by a hub twice.

Second, as we pointed out in Section 12.3.3, a hub is considered as faulty in two different situations: (1) when its Hub Kernel region fails and (2) when the hub propagates errors as a consequence of not being able to isolate any of its ports.

Third, similarly to what we did for the case of the model of CANcentrate, we will call each port of the hub a node is connected to either a hub uplink port or a branch. Additionally, we will refer to each hub port connected to an incoming sublink as a hub sublink port or an *inbranch*.

It is also very important to understand when we consider that a branch and an inbranch are faulty. On the one hand, since a branch can be seen as an uplink hub port, we consider that it is faulty when the node connected to that port cannot communicate through it. This implies that a branch is faulty in the following cases: (1) when the Node Kernel of the node that uses the branch is faulty; (2) when the Node Connection region corresponding to the branch fails; (3) when the hub to which the branch belongs suffers from a fault or it cannot contain the errors received from other hub port. As will be explained, many submodels need to know how many branches are faulty in order to take their own decisions. For this purpose, the submodels share the place *numFaultyBranches*, whose marking indicates the number of branches that are faulty.

On the other hand, we consider that an inbranch is faulty when the hub that uses this inbranch to receive the contribution of the other hub receives an incorrect contribution. In this sense, an inbranch is faulty in the following cases: (1) when the Hub Interconnection region corresponding to the inbranch fails; (2) when any of the hubs fails (it kernel fails or it cannot contain the errors it receives from any of

its ports) and (3) when the hubs are decoupled. Concerning these two last cases, notice that when a hub fails, not only the inbranches that carry the contribution of the faulty hub can be considered as faulty, but also the inbranches that convey the contribution of the non-faulty hub. This is because, from the communication point of view, these last inbranches are isolated together with the faulty hub. Similarly, when the hubs are decoupled, their contributions are not related to each other because each hub constitutes an independent communication domain. Thus, even if the Hub Interconnection that constitutes a given inbranch is not faulty, the hub perceives the contribution it receives through that inbranch as erroneous.

Fourth, it is important to highlight when we consider that the hubs are decoupled. Specifically, we understand that the hubs are decoupled when both of them are not faulty, but there are no enough Hub Interconnection regions to carry the contribution of the hub A to the hub B or viceversa. In this sense, from now on we will say that the hubs are decoupled when each one of them represents a separate communication domain that the nodes can use for communicating.

Finally, it is important to note that many submodels based their own decisions on the capacity of each node of the system to communicate through the hub A and the hub B. This is because the impact of a fault on the overall system depends on the communication capabilities of each node that is affected by a fault. To better understand this issue, let as classify each node of ReCANcentrate into the following categories, depending on its capacity to communicate through the hub A and the hub B.

- *okAB node*. It is a node whose Node Kernel is not faulty (the node is operative) and that can communicate through hub A and hub B.
- *okA node*. It is a node whose Node Kernel is not faulty and that tolerated a fault that prevented it from communicating through hub B. Such a node can still communicate through hub A.
- *stopA node*. It is a node whose Node Kernel is not faulty and that did not tolerate a fault that prevented it from communicating through hub B. As a consequence, although the node could still communicate through hub A, it stops communicating.
- *okB node*. It is a node whose Node Kernel is not faulty and that tolerated a fault that prevented if from communicating through hub A. Such a node can still communicate through hub B.
- *stopB node*. It is a node whose Node Kernel is not faulty and that did not tolerate a fault that prevented it from communicating through hub A. As a

consequence, although the node could still communicate through hub B, it stops communicating.

- *noConn node*. It is a node whose Node Kernel is not faulty and that can use no hub for communicating. A noConn node differs from a stopA and a stopB node in the fact that a noConn node cannot potentially use any hub for communicating.

- *nonOk node*. It is a node whose Node Kernel is faulty. It does not matter whether or not it could potentially communicate through any hub; since its Node core is faulty, it is not operative and it does not communicate.

Attending to this classification it is easy to see that, for instance, the impact of a fault affecting a stopA or stopB node will be lower than the impact of a fault affecting an okA or an okB node. In the first case, the fault does not prevent any new node from communicating, whereas in the second case it does.

Due to this necessity of knowing the communication capabilities of each node in order to evaluate the impact of faults, the submodels share a set of places whose marking represent the number of nodes per category. Specifically, the model of ReCANcentrate includes a set of shared places, each of which represents the number of nodes that belong to one of the first five categories of nodes listed above. These places are: *okABNodes*, *okANodes*, *stopANodes*, *okBNodes* and *stopBNodes* respectively. Notice that it is not necessary to include a place that represents the number of *noConn* nodes and a place that models the quantity of *nonOk* nodes. On the one hand, the quantity of nodes that belong to any of these two categories can be calculated by subtracting the number of nodes that belong to the other categories from the total number of nodes. On the other hand, we do not need to differentiate between *noConn* and *nonOk* nodes, since a fault affecting a node of any of these two categories has the same impact on the communication.

12.4.3 *ReCANcentrateFaiEval* submodel

As already said, when a fault occurs and the coverage process finishes, the *ReCANcentrateFaiEval* submodel decides whether or not an overall failure has occurred. For the sake of clarity, Figure 12.3 depicts a simplified version of the *ReCANcentrateFaiEval* submodel. This version does not show how *ReCANcentrateFaiEval* takes into account the value of *sysFauTolCov*, i.e. the coverage with which an FT/A system accepts or tolerates the failure or disconnection of a new node, provided that the number of nodes that have failed or that have become disconnected so far

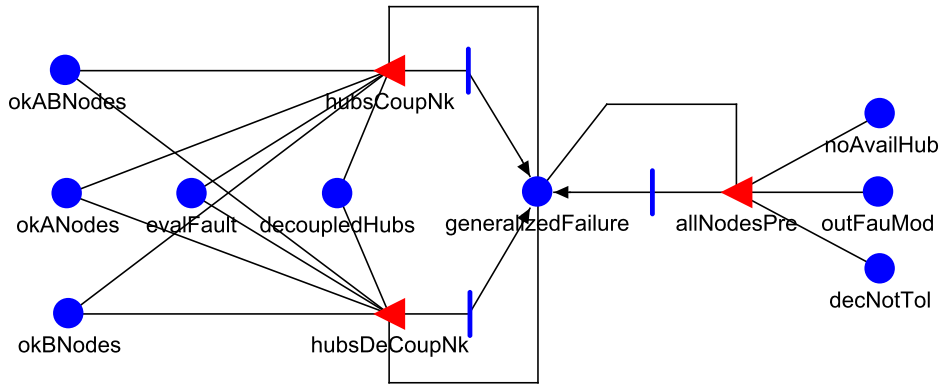

Figure 12.3: ReCANcentrateFaiEval submodel

(including the new one) can be theoretically accepted or tolerated by that FT/A system.

Leaving this aspect aside, notice that there are different circumstances that lead *ReCANcentrateFaiEval* to diagnose the failure of the overall system. Each one of these circumstances is detected by a dedicated input gate that monitors the marking of a set of places *ReCANcentrateFaiEval* shares with other submodels. When appropriate, each input gate enables an instantaneous activity that sets a token in the place *generalizedFailure*. As in the case of the models of the CAN bus and CANcentrate, a token in this place indicates that the overall system is faulty and stops all the submodels (including the *ReCANcentrateFaiEval* submodel itself), in order to reduce the state space of the underlying Markov process.

The first one of the referred input gates, *allNodesPre*, is devoted to detecting situations in which all nodes are prevented from communicating. Gate *allNodesPre* becomes aware about any of these situations when it receives a token in *noAvailHub*, *outFauMod* or *decNotTol* (see Figure 12.3). The specific role of each one of these places is as follows. *ReCANcentrateFaiEval* receives a token in the place *noAvailHub* when there is not any hub available for communicating, i.e. when both hubs are faulty. Notice again that a hub is considered as faulty in two different situations: when its Hub Kernel region fails and when the hub cannot isolate a faulty uplink hub port or a faulty sublink hub port. As concerns the place *outFauMod*, it receives a token when an out-of-fault-model (ofm) fault occurs. Finally, a token in the place *decNotTol* indicates that the hubs have become decoupled and that the nodes were not able to tolerate this situation. Notice that a token in any of these places implies the failure of the overall system. Thus, the input gate *allNodesPre*

compels its instantaneous activity to set a token in *generalizedFailure* when a token is set in any of them.

The other two input gates are aimed at detecting situations in which, although not all nodes are prevented from communicating, the number of nodes that can operate and communicate among them is not enough to consider that the system can deliver its services. The difference between both input gates is that *hubsCoupNk* is devoted to detecting such a situation when the hubs are coupled with each other, whereas *hubsDeCoupNk* aims at detecting this situation when the hubs are decoupled.

To better understand how each one of these gates calculates the number of nodes that are operative and that communicate, let us describe the role of the places that are connected to them. As can be seen in Figure 12.3, these places are: *decoupledHubs*, *okABNodes*, *okANodes* and *okBNodes*.

ReCANcentrateFaiEval receives a token in the place *decoupledHubs* when both hubs become decoupled, but nodes are able to continue communicating using both hubs independently.

The characteristics of the places *okABNodes*, *okANodes* and *okBNodes* were explained before in Section 12.4.2: they are shared among several submodels and their markings respectively reflect the number of okAB, okA and okB nodes. Notice that these types of nodes embrace every node that is able to communicate through a hub. Therefore, coverage submodels use the places *okABNodes*, *okANodes* and *okBNodes* to indirectly indicate to *hubsCoupNk* and *hubsDecCoupNk* how many nodes can communicate among them. In other words, these places are the ones we pointed out before in Section 12.4.1 and whose marking make it possible for the evaluator submodel to infer the number of intercommunicating nodes.

As concerns the place *evalFault*, it is used to indicate to *hubsCoupNk* and *hubsDecCoupNk* that the markings of the places *okABNodes*, *okANodes* and *okBNodes* are not coherent and, thus, that these places cannot be used for taking any decision. More specifically, as will be explained later, when a fault occurs, a token is set in the place *evalFault* in order to indicate that the markings of referred places are incoherent until all the corresponding coverage submodels assess how the errors are propagated and how they are contained or tolerated, i.e. until the coverage process ends. Afterwards, this token is erased when the coverage submodels finish. As already pointed out, this fact is reflected in Figure 12.2, where *ReCANcentrateFaiEval* appears at the end of each path of the coverage process.

Taking into account the above considerations, the expression of *hCoupNk* is as follows:

$$
\begin{aligned}
&\textit{generalizedFailure} \rightarrow \textit{Mark()} == 0 \textit{ and} \\
&\textit{decoupledHubs} \rightarrow \textit{Mark()} == 0 \textit{ and} \\
&\textit{okABNodes} \rightarrow \textit{Mark()} + \textit{okANodes} \rightarrow \textit{Mark()} + \\
&\textit{okBNodes} \rightarrow \textit{Mark()} < \textit{numNodes} - \textit{kSevere and} \\
&\textit{evalFault} \rightarrow \textit{Mark()} == 0
\end{aligned}
$$

The first term ensures that the input gate is not enabled if an overall failure has already occurred. The second one guarantees that *hubsCoupNk* remains disabled if the hubs are decoupled. The third term compares the number of nodes that operate and communicate among them, i.e. the quantity of intercommunicating nodes, with the minimum number of intercommunicating nodes the system needs in order to deliver its service. Notice that when both hubs are coupled, an operative node can communicate as long as it can transmit and receive through at least one hub, no matter which. Thus, the number of intercommunicating nodes can be calculated by simply adding up the marking of the places *okABNodes*, *okANodes* and *okBNodes*. Once this number is obtained, *hCoupNk* compares it with the value of *numNodes* − *kSevere*. If the number of intercommunicating nodes is few than *numNodes* − *kSevere*, then a generalized failure occurs. The parameter *numNodes* specifies the total number of nodes, whereas *kSevere* is the maximum number of non-intercommunicating nodes a system can accept or tolerate. Notice that the parameter *kSevere* allows to configure the model to measure the NFT/AR and different degrees of FT/AR_k. For instance, if one is interested in measuring the NFT/AR, the minimum number of intercommunicating nodes is the total number of nodes and, thus, *kSevere* must be specified as 0. Finally, the fourth term refers to the marking of the place *evalFault*. Gate *hCoupNk* remains disabled as long as *evalFault* contains a token, since the marking of places like *okABNodes*, *okANodes* and *okBNodes* are incoherent until the coverage process ends.

Finally, the expression of *hubsDecCoupNk* basically differs from the above expression in the way in which the number of nodes that can communicate among them is calculated:

$$
\begin{aligned}
&\textit{generalizedFailure} \rightarrow \textit{Mark()} == 0 \textit{ and} \\
&\textit{decoupledHubs} \rightarrow \textit{Mark()} == 1 \textit{ and} \\
&\textit{okABNodes} \rightarrow \textit{Mark()} + \textit{okANodes} \rightarrow \textit{Mark()}) < \textit{numNodes} - \textit{kSevere and} \\
&\textit{okABNodes} \rightarrow \textit{Mark()} + \textit{okBNodes} \rightarrow \textit{Mark()} < \textit{numNodes} - \textit{kSevere and} \\
&\textit{evalFault} \rightarrow \textit{Mark()} == 0
\end{aligned}
$$

In this case, since the hubs are decoupled, the okA nodes cannot communicate with the okB nodes. Thus, the input gate calculates the quantity of nodes that can communicate through the hub A and the hub B, independently. Specifically, the quantity of nodes that communicate through the hub A is calculated in the third term, whereas the quantity of nodes that communicate through the hub B is calculated in the fourth one. If both quantities of nodes are few than $numNodes - kSevere$, then a generalized failure occurs. This means that we suppose that the amount of nodes that communicate among them when the hubs are decoupled is the maximum of the number of nodes that can communicate through the hub A and the hub B. This can be accomplished, for instance, if the nodes of the system are provided with a mechanism that allows them to permanently know which is the hub that provides service to the maximum number of nodes and then, to communicate through that hub.

Notice that to assume such a mechanism for ReCANcentrate when the hubs are decoupled is optimistic, since we could assume a different communication strategy in which the number of nodes that intercommunicate when the hubs are decoupled is fewer than the referred maximum. However, the reliability results presented in this dissertation are not biased towards ReCANcentrate as a consequence of this assumption. This is because, as explained before, we suppose that nodes do not tolerate the situation in which hubs become decoupled, i.e. the default value of the parameter *decCov* is 0. In contrast, this assumption allows assessing what would be the maximum system reliability that can be achieved with ReCANcentrate if the nodes are able to tolerate hub decouplings.

12.4.4 *nodeKernelsR* submodel

The *nodeKernelsR* submodel models faults happening at the Node Kernel regions of ReCANcentrate. Additionally, since a faulty Node Kernel can generate errors that propagate to the hub through its uplinks, the submodel normally needs to initiate a path of the coverage process that evaluates how these errors are contained. More specifically, *nodeKernelsR* initiates a path of the coverage process if the fault affects any hub port that has not failed so far, i.e. a hub port that is not already isolated.

If the Node Kernel suffers from an ofm fault, it activates the *ofmFauEval* submodel, which will be thoroughly described in Section 12.4.14. In contrast, if the fault manifests in a way that is included in our fault model, *nodeKernelsR* compels two different coverage submodels to start the corresponding paths of the coverage process that evaluate how a faulty Node Kernel is treated and tolerated (see Figure 12.2) by means of the corresponding ReCANcentrate's mechanisms. These

coverage submodels are *fauLPsEvalAtHubs* and *fauLPevalAtHub*, which are described in detail in sections 12.4.9 and 12.4.10 respectively. The specific coverage submodel that *nodeKernelsR* compels to take over depends on the number of uplink hub ports (branches) affected by the errors the faulty Node Kernel generates. If the fault affects two ports (one placed a the hub A and the other at hub B), then *nodeKernelsR* activates the *fauLPsEvalAtHubs* coverage submodel, which evaluates whether or not each hub contains the errors. If the fault affects only the uplink port of one of the hubs, *nodeKernelsR* compels the *fauLPevalAtHub* coverage submodel to proceed. This submodel evaluates if the corresponding hub contains the errors.

The coverage process continues ahead if one of the hubs does not contain the errors. As already said, a hub that does not isolate a faulty port is considered as faulty, since it delivers errors through all its downlinks and sublinks. At this point, how to model the way in which the errors the hub propagates are treated depends on whether or not the hubs were coupled before the fault occurred. If the hubs were coupled, it is necessary to evaluate if the hub that remains non-faulty successfully isolates the faulty hub by disabling the corresponding sublinks. The coverage submodel involved in such an evaluation is the *fauHubEvalAtHub* submodel. This fact is depicted in Figure 12.2, where both *fauLPsEvalAtHubs* and *fauLPEvalHub* are connected to the this submodel. Moreover, if the non-faulty hub successfully isolates the faulty one, then it is also necessary to evaluate whether or not each node that was using the hub that fails for communicating is able to isolate it and, then, to continue communicating through the non-faulty hub. The *fauHubEvalAtNodes* coverage submodel performs this evaluation; as reflected in Figure 12.2, where *fauHubEvalAtHub* is connected to this coverage submodel.

In contrast, if the hubs were already decoupled when the fault occurred, then it has not sense to evaluate if the non-faulty hub is able to isolate the faulty one. This is because the non-faulty hub would have already isolated the sublinks through which it receives the contribution of the faulty hub. However, in this situation, the fact that the model had not stopped although the hubs were decoupled means that the nodes were still communicating using both hubs independently before the fault occurred. Therefore, it is necessary to reflect that the nodes will not be able to use the faulty hub for communicating any more. Once again, this is carried out by the *fauHubEvalAtHub* submodel, which is directly activated by the *fauLPsEvalAtHub* and by the *fauLPevalAtHub* submodels when necessary (see Figure 12.2).

As concerns the *nodeKernelsR* submodel itself, its structure is depicted in Figure 12.4. As in the case of the corresponding submodels of the CAN bus 10.8.1 and CANcentrate 10.7.1, the marking of the place *okNodeKernels* represents the number of Node Kernels that have not failed so far; whereas the place *generalized-*

Failure is used to disable the submodel when the overall system is faulty.

Moreover, the activity *nkFailure* has also the same role as in these two submodels: it represents the time that elapses until a non-faulty Node Kernel region fails. Thus, the failure rate of the Time To Failure distribution it models is, once again:

$$okNodeKernels \rightarrow Mark() \cdot nodeCoreFRate$$

The activity *nkFailure* also has three cases. The difference between these cases is that each one of them respectively represents (starting from the upper one) a fault occurring at an okAB node; at an okA, stopA, okB, or stopB node; and at a noConn node. The proportion with which the activity *nkFailure* selects each one of these cases is basically calculated by dividing the number of Node Kernels that belong to the corresponding type of node by the total number of non-faulty Node Kernels. Specifically, the proportion with which the activity *nkFailure* selects the first case is basically calculated by dividing the number of Node Kernels that belong to nodes that can communicate through both hubs, i.e. okAB nodes, by the total number of non-faulty Node Kernels:

$$\frac{okABNodes \rightarrow Mark()}{okNodeKernels \rightarrow Mark()}$$

Analogously, the second case of the activity *nkFailure*, which is the proportion with which the Node Kernel that fails belongs to an okA, stopA, okB, or stopB node, is calculated as:

$$(okANodes \rightarrow Mark() + stopANodes \rightarrow Mark() + okBNodes \rightarrow Mark() + stopBNodes \rightarrow Mark()) \cdot \frac{1}{okNodeKernels \rightarrow Mark()}$$

The proportion of the third case of the activity *nkFailure* is obtained using the same strategy. In this case, the number of non-faulty nodes that cannot communicate through any hub, i.e. noConn nodes, is calculated subtracting the number of nodes that can communicate through two or only one hub from the total number of nodes whose Node Kernel is not faulty:

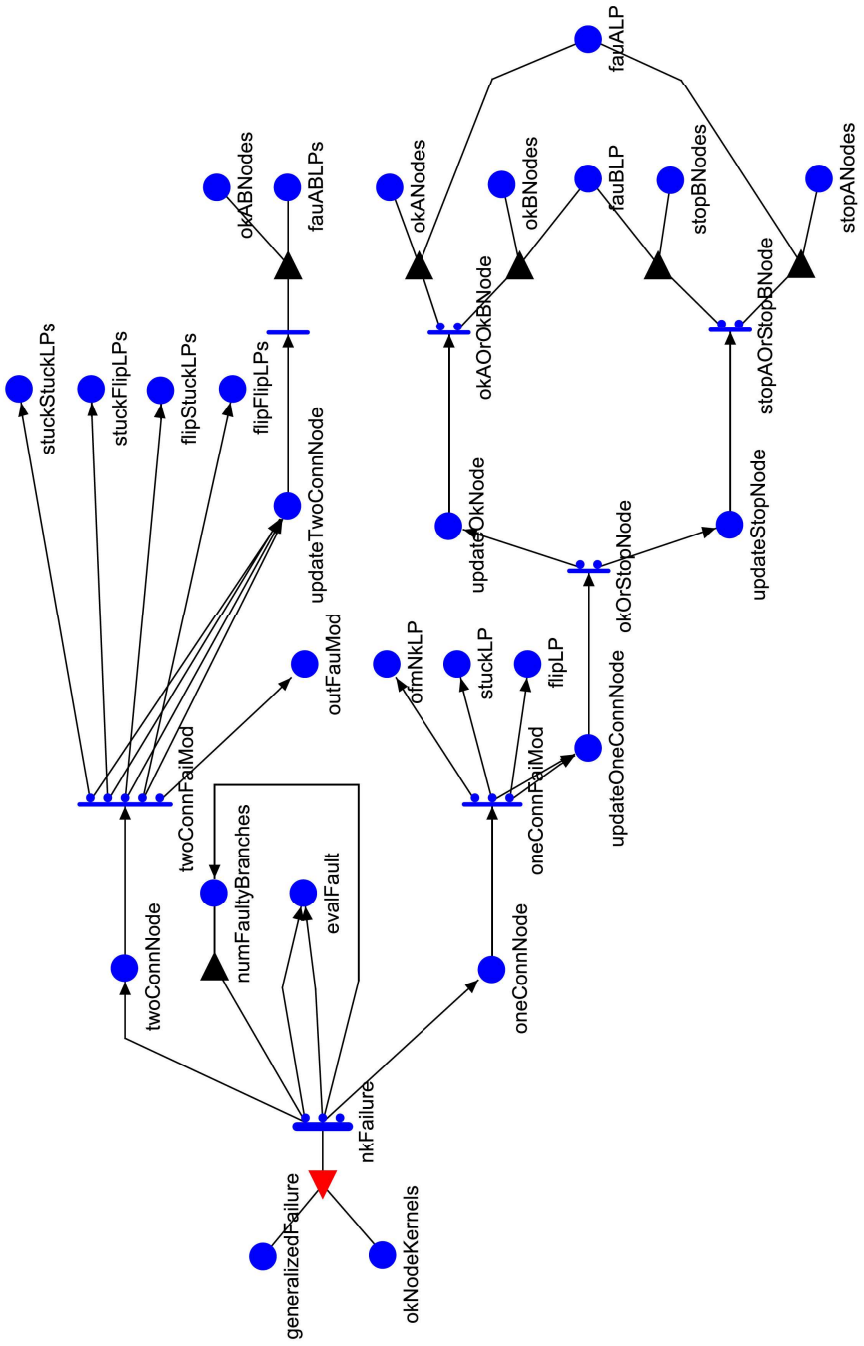

Figure 12.4: nodeKernelsR submodel

$$[okNodeKernels \rightarrow Mark() - (okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark() + okBNodes \rightarrow Mark() + stopBNodes \rightarrow Mark())] \cdot \frac{1}{okNodeKernels \rightarrow Mark()}$$

As can be seen in Figure 12.4, the first case of the activity *nkFailure* sets two tokens in the place *numFaultyBranches* (using an output gate for this purpose) and one token in the place *evalFault*. On the one hand, as explained in Section 12.4.2, the place *numFaultyBranches* is shared with other submodels and it indicates the number of branches that are faulty. In this sense, since the first case of the activity *nkFailure* represents a Node Kernel failure happening in a node that can communicate through both hubs, it is necessary to record that the two branches through which the node can communicate become faulty. On the other hand, the first case of the activity *nkFailure* sets a token in the place *evalFault*, in order to prevent the *ReCANcentrateFaiEval* submodel to evaluate the places *okABNodes*, *okANodes* and *okBNodes* before their markings are consistent. As explained in Section 12.4.3, the marking of these places are inconsistent until the corresponding evaluator submodels do not finish assessing how the errors generated by the fault are propagated and contained.

In addition, the first case of the activity *nkFailure* sets a token in the place *twoConnNode*, thereby enabling the instantaneous activity *twoConnFaiMod*, which has five cases. These cases cover the different ways in which the Node Kernel failure manifests at the two uplink hub ports at which it is connected to. In this way, the first case sets a token in the place *stuckStuckLPs*, which indicates that the failure manifests as stuck-at at both uplink hub ports; the second case sets a token in the place *stuckFlipLPs*, which indicates that the failure manifests as stuck-at at the uplink port of the hub A and as bit-flipping at the uplink port of the hub B; and so on.

However, the last one of these cases (the lower one) sets a token in *outFauMode*, which provokes that the system is diagnosed as faulty. Note that we could differentiate between each one of the situations in which the faulty Node Kernel manifests as ofm at one uplink hub port only, while exhibiting a fault that is included in our fault model at the other uplink hub port. Nevertheless, in order to simply the model, we assumed that a Node Kernel that exhibits an ofm failure at one of its Node Connections eventually sends ofm errors through its other Node Connection. This could actually happen if the application decides to send ofm information through both its Node Connections, as a consequence of observing ofm errors generated by its Node Kernel through one of them. Of course, this assump-

tion can be considered pessimistic for ReCANcentrate, since a fault that affects a Node Kernel and that sends ofm error through one Node Connection does not necessarily lead to a generalized failure. For instance, imagine a situation in which the hubs are decoupled and in which the only node that can communicate through both hubs suffers from a fault in its Node Kernel. If the faulty Node Kernel sends ofm information to one of the hubs only, then only the nodes connected to that hub fail. In this situation, a generalized failure does not occur if the number of nodes that remain connected to the other hub are enough for the system to deliver its service.

The proportion with which each one of the first four cases of the activity *twoConnFaiMod* is selected is calculated multiplying the probability with which the Node Core (the Node Kernel is composed of the Node Core exclusively) compels each one of its two CAN controllers to transmit the types of errors represented by that case. For instance, the proportion of the second case is calculated as:

$$(\mathit{nodeCoreStrProp} + \mathit{nodeCoreStdProp}) \cdot \mathit{nodeCoreFlipProp}$$

Which is the probability with which the Node Core compels the CAN controller connected to the hub A and the CAN controller connected to the hub B to respectively transmit a stuck-at and a bit-flipping stream. See the meaning of the above parameters at Table 10.4). Notice that in Section 10.4.4, we said that it is practically impossible that a fault affecting a Node Core leads a CAN controller to transmit a stuck-at-dominant or a bit-flipping stream. However, we decided to model this possibility for the sake of completeness, even though the parameters that specify the proportion with which the Node Core manifest as stuck-at-dominant and bit-flipping are set to 0.

As concerns the proportion of the last case of the activity *twoConnFaiMod*, i.e. the one that sets a token in *outFauMod*, it is calculated in a different way, as follows:

$$2 \cdot \mathit{nodeCoreOfmProp} - \mathit{nodeCoreOfmProp}^2$$

As just explained above, we suppose that a Node Kernel that has two non-faulty Node Connections provokes a generalized failure if it manifests as ofm in any of its uplink hub ports (no matter which). Thus, the probability with which *twoConnFaiMod* chooses its fifth case is the sum of the probability of the event in which the fault manifests as ofm at the hub A and the event in which the fault manifests as ofm at the hub B, i.e. $2 \cdot \mathit{nodeCoreOfmProp}$, minus the probability of the intersection of these two events, i.e. $\mathit{nodeCoreOfmProp}^2$.

Going back to the first four cases of *twoConnFaiMod*, notice that besides setting a token in one of the mentioned places, each one of them always writes a token at the place *updateTwoConnNode*. This action enables an instantaneous activity that is connected to the places *okABNodes* and *fauABLPs* by means of an output gate. On the one hand, this output gate decreases the number of okAB nodes of the system in one unit, since this point of the *nodeKernelsR* submodel is reached as a consequence of a fault affecting the Node Kernel of such a kind of node (the okAB node becomes a nonOk node). On the other hand, the output gate sets a token at the place *fauABLPs* in order to indicate to the *fauLPsEvalAtHubs* submodel that two new uplink ports (one of them belonging to the hub A and the other to the hub B) have become faulty. As said at the beginning of this section, when a faulty Node Kernel affects two uplink hub ports, *nodeKernelsR* compels the *fauLPsEvalAtHubs* coverage submodel to evaluate if each hub isolates the fault (see Figure 12.2).

Until this point we have discussed the actions that are performed after the activity *nkFailure* selects its first case. As concerns its second case, as explained above, it represents the situation in which the Node Kernel that fails is placed in an okA, stopA, okB, or stopB node. This case is also connected to the places *numFaultyBranches* and *evalFault*. However, it increases the marking of the place *numFaultyBranches* in one unit. This is because the node was already not able to communicate through one of the hubs, i.e. the corresponding Node Connection or Hub Kernel region was already faulty, and thus only one new branch must be recorded as faulty.

Analogously to the first case, the second case of the activity *nkFailure* additionally sets a token in the place *oneConnNode*, thereby enabling the instantaneous activity *oneConnFaiMod*. This activity has three cases. The first one is chosen to model that the Node Core exhibits an ofm failure. The second and third cases respectively represent the situation in which the Node Core compels the CAN controller, corresponding to the branch through which it can still communicate, to transmit an erroneous stuck-at and a bit-flipping stream. The proportion of the first case is calculated as *nodeCoreOfmProp*; whereas the proportions of the second and third ones are calculated by adding the proportions with which the faulty Node Core compels the CAN controller to transmit a stuck-at or a bit-flipping stream. Specifically, the proportions of the two cases are *nodeCoreStrProp* + *nodeCoreStdProp*, and *nodeCoreFlipProp* respectively.

If the activity *oneConnFaiMod* selects if first case, it sets a token in the place *ofmNkLP*. This place is used by *nodeKernelsR* to activate the *ofmFauEval* submodel when necessary. Notice that this point of the submodel is reached when the Node Kernel that fails has only one non-faulty Node Connection and, thus, the ofm errors generated by this Node Kernel only reach the uplink hub port of one of the

hubs. This implies that the fact that these ofm errors propagate throughout the system depends on some circumstances, e.g. on whether or not the hubs are coupled. These circumstances are analyzed by *ofmFauEval*, which decides whether the ofm errors pollute all the system or only part of it.

Similarly, when the activity *oneConnFaiMod* chooses its second or its third case, it respectively sets a token in the places *stuckLP* and *flipLP*, no matter it is an okA/okB or a stopA/stopB node. As mentioned above, when a Node Kernel fails in way that is included in our fault model and sends errors to one uplink hub port only, *nodeKernelsR* compels the *fauLPevalAtHub* coverage submodel to evaluate if the corresponding hub isolates the fault (see Section 12.4.10 for a detailed explanation of this submodel). Specifically, *nodeKernelsR* shares the places *stuckLP* and *flipLP* with *fauLPevalAtHub* in order to indicate to it how the Node Kernel failure manifests at the hub port. However, notice that a token in any of these places does not compel *fauLPevalAtHub* to take over. Instead, *nodeKernelsR* leads *fauLPevalAtHub* to proceed later on, by setting a token in the place *fauALP* or in the place *fauBLP*, as will be explained in this section.

The second and third cases of the activity *oneConnFaiMod* also write a token at the place *updateOneConnNode*, thereby initiating a set of actions aimed at deciding which specific kind of node the Node Kernel that fails is located at, i.e. at a okA, a stopA, a okB, or a stopB node, and then, to update the marking of the places okANodes, stopANodes, okBNodes and stopBNodes accordingly.

The first one of this actions is performed by the activity *okOrStopNode*, which decides whether the Node Kernel is placed at a node that does communicate through one hub or that does not. In this sense, the first case of this activity is chosen when the node is an okA or an okB node, whereas the second one is selected when the node is a stopA or a stopB node. The proportion of these cases is calculated by dividing the number of okA and okB nodes, or the number of stopA and stopB nodes, by the total number of these nodes, respectively. More specifically, the first case proportion is:

$$
\begin{aligned}
&(okANodes \rightarrow Mark() + okBNodes \rightarrow Mark()) \ / \\
&(okANodes \rightarrow Mark() + stopANodes \rightarrow Mark() + \\
&okBNodes \rightarrow Mark() + stopBNodes \rightarrow Mark())
\end{aligned}
$$

whereas the proportion of the second case is calculated as:

$$(stopANodes \rightarrow Mark() + stopBNodes \rightarrow Mark()) \ / \\ (okANodes \rightarrow Mark() + stopANodes \rightarrow Mark() + \\ okBNodes \rightarrow Mark() + stopBNodes \rightarrow Mark())$$

When selected, these cases transfer the token to the place *updateOkNode* and *updateStopNode* respectively. The actions carried out when the second case is selected are analogous to the actions performed when the first case is chosen. Thus, next we only describe what is done when the first case is taken. The activity *okAOrOkBNode* decides whether the affected node is an okA or an okB node. For that, it has two cases whose proportions are calculated using the same strategy just explained for the activity *okOrStopNode*; so that *okAOrOkBNode* selects the first and second cases with proportions:

$$\frac{okANodes \rightarrow Mark()}{(okANodes \rightarrow Mark() + okBNodes \rightarrow Mark())}$$

and:

$$\frac{okBNodes \rightarrow Mark()}{(okANodes \rightarrow Mark() + okBNodes \rightarrow Mark())}$$

Each one of the two cases of the activity *okAOrBNode* is connected to a dedicated output gate. On the one hand, the output gate decreases the marking of the place *okANodes* (or *okBNode*) in order to reflect that the okA (or the okB) node becomes a nonOk node. On the other hand, the output gate sets a token in the place *fauALP* (or *fauBLP*), thereby indicating to the *fauLPEvalAtHub* submodel that a new uplink port corresponding to the hub A (or the hub B) has become faulty.

Finally, to conclude this section, let us explain why the third case of the activity *nkFailure* is unconnected. Notice again that this case is chosen when the faulty Node Kernel is placed in a noConn node. This implies that the two branches of this node were already faulty and that, thus, their failure was already recorded in the place *numFaultyBranches*. Moreover, it does not matter the type of erroneous stream or frame that the faulty Node Kernel tries to transmit through both branches, since the respective uplink hub ports (or the hub itself) were already isolated. Because of these reasons, the third case of *nkFailure* is not connected to any place, which means that no further action is performed when this case is chosen.

12.4.5 *nodeConnsR* submodel

The *nodeConnsR* submodel is the responsible for modelling faults happening at the Node Connection regions of ReCANcentrate. In addition, as explained before, a regions submodel sometimes needs to compel specific coverage submodels to carry out a path of the coverage process that evaluates how the faults are treated. In the case of a fault happening in a Node Connection region, the corresponding coverage process's path is initiated in two different situations

First, *nodeConnsR* compels *ofmFauEval* to take over when the fault is not included in our fault model (see Figure 12.2). The role of *ofmFauEval* was introduced before and it will be thoroughly described in Section 12.4.14.

Second, *nodeConnsR* activates the *fauLPevalAtNode* submodel when the fault is included in our fault model and affects a branch that was not faulty (and thus not previously isolated). The *fauLPevalAtNode* submodel basically evaluates if the node to whose the Node Connection region belongs to tolerates the fault, i.e. it evaluates whether or not the node of the faulty Node Connection will be able to continue communicating using its other Node Connection region (see Section 12.4.8 for a detailed explanation about this issue). In addition, when *fauLPevalAtNode* finishes, it is necessary to evaluate if the hub corresponding to that Node Connection isolates the fault. Thus, as also depicted in the referred figure, *fauLPevalAtNode* further compels *fauLPevalAtHub* to evaluate this aspect. Notice that *fauLPevalAtHub* is the same coverage submodel *nodeKernelsR* activates when a Node Kernel fault that is included in our fault model affects only the uplink port of one of the hubs the node is connected to (see Section 12.4.4). This is because the error-containment actions a hub carries out when one of its uplink ports fails do not depend on whether the errors are provoked by a faulty Node Kernel or a faulty Node Connection. Moreover, also notice that if the hub does not isolate the uplink port corresponding to the faulty Node Connection, *fauLPevalAtHub* will activate the coverage submodels *fauHubEvalAtHub* and/or *fauHubEvalAtNodes* following the same strategy already explained at the beginning of the section that describes the *nodeKernelsR* submodel (Section 12.4.4).

The structure of the *nodeConnectionsR* submodel is depicted in Figure 12.5. It is similar to the structure of the *nodeKernelsR* submodel. The marking of the place *okNodeConns* is the number of Node Connection regions that are not faulty. Its initial value is two times the number of nodes, $2 \cdot numNodes$, since each node initially has two non-faulty Node Connection regions (one per hub). See Table 12.1 for the specific default values of the parameter *numNodes*.

The activity *ncFailure* models the Time To Failure distribution of all the surviving Node Connections as a whole. This distribution is exponential and its failure

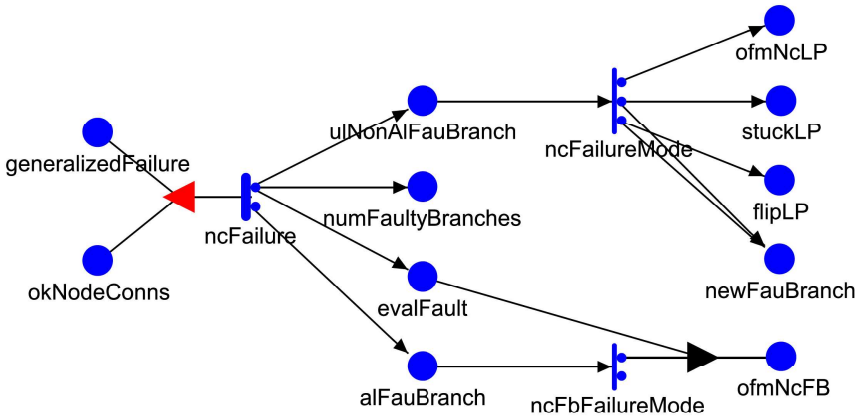

Figure 12.5: nodeConnsR submodel

rate is calculated following the same strategy used for all the regions we have explained so far (see Equation 10.4):

$$okNodeConns \rightarrow Mark() \cdot (ctrlFRate + 2 \cdot (nodeIOFRate + lnkAttchFRate + hubIOFRate + 2 \cdot termFRate))$$

Notice that this failure rate coincides with the failure rate of the activity *ncFailure* of the submodel that represents the Node Connection regions in CANcentrate: the *nodeConnsT* submodel (see Section 10.7.2). This is because a Node Connection region is composed of exactly the same entities in CANcentrate and in ReCANcentrate.

In fact, the structure of both submodels is very similar: compare Figures 10.9 and 12.5. The activity *ncFailure* of *nodeConnsR* has two cases, each of which has exactly the same meaning as in the equivalent activity of CANcentrate. The first case represents a fault that affects a Node Connection located in a non-faulty (and thus non-isolated) branch; whereas the second one models a Node Connection fault that happens in a branch that was already faulty (and thus, in a branch that is already isolated).

The proportions of both cases are calculated exactly as in the case of the Node Connection region submodel of CANcentrate. The expression of the first case is:

$$\frac{numBranches - numFaultyBranches \rightarrow Mark()}{okNodeConns \rightarrow Mark()}$$

And the proportion of the second one is:

$$\frac{(okNodeConns \rightarrow Mark() - (numBranches - numFaultyBranches \rightarrow Mark()))}{okNodeConns \rightarrow Mark()}$$

It is noteworthy that, as in CANcentrate, we use the value of the parameter *numBranches* and the marking of the place *numFaultyBranches*, to calculate the probabilities with which a Node Connection that fails belongs to a non-faulty branch and to an already faulty branch. For instance, the probability with which a Node Connection that fails is placed in a non-faulty branch is calculated by dividing the number of non-faulty Node Connections that are placed at non-faulty branches, by the total number of Node Connections that have not failed so far. Notice that the number of non-faulty Node Connections located at non-faulty branches coincides with the number of non-faulty branches, i.e. with $numBranches - numFaultyBranches \rightarrow Mark()$. This is because a faulty Node Connection leads its branch to be faulty and, hence, a faulty Node Connection cannot be placed in a non-faulty branch.

The actions that *nodeConnsR* performs after the activity *ncFailure* fires depend on the specific case this activity chooses. If it selects its first case (the upper one), *ncFailure* sets a token in three places: *numFaultyBranches*, *evalFault*, and *ulNonAlFauBranch*. Since this first case represents the failure of a Node Connection that is located in a non-faulty (and thus non-isolated) branch, it is necessary to record that a new branch is faulty by increasing the marking of *numFaultyBranches* in one unit. In addition, when this first case is selected, it is needed to evaluate how the errors generated by the fault propagate and are contained, i.e. it is necessary to start up the corresponding path of the coverage process. The first step needed to do this consists in inhibiting the *ReCANcentrateFaiEval* submodel, temporarily, so that it does not evaluate whether or not the communication system is faulty. As explained before, this is done by setting a token in the place *evalFault*.

But what really initiates the coverage process is the token that the first case of *ncFailure* writes at the place *ulNonAlFauBranch*. A token in this place enables the instantaneous activity *ncFailureMode*, which decides the way in which the Node Connection manifests at the uplink hub port. The first case of this activity corresponds to a Node Connection region that fails exhibiting an ofm failure mode. The proportion of this case is calculated as in the *nodeConnsT* submodel of CANcentrate, i.e. using the Equation 10.10:

$$2 \cdot \mathit{lnkAttchOfmProp} \cdot \mathit{lnkAttchFRate} + 2 \cdot \mathit{nodeIOOfmProp} \cdot \mathit{nodeIOFRate} + \mathit{ctrlOfmProp} \cdot \mathit{ctrlFRate} + 2 \cdot \mathit{hubIOOfmProp} \cdot \mathit{hubIOFRate} + 4 \cdot \mathit{termOfmProp} \cdot \mathit{termFRate}$$

A node that sends ofm errors to one hub only does not necessarily pollute all the system with these errors. This was already pointed out in Section 12.4.4, where we also indicated that the *ofmFauEval* submodel is the responsible for taking a final decision concerning this issue. Therefore, what the first case of the activity *ncFailureMode* does is to activate this submodel by setting a token in *ofmNcLP*.

The second case of *ncFailureMode* models the situation in which the fault manifests as stuck-at, whereas the third one corresponds to a bit-flipping failure mode. The proportions of each one of this cases are also calculated as in the *nodeConnsT* submodel of CANcentrate, i.e. using the Equations 10.11. The proportions of the first and the second case are (respectively):

$$\begin{aligned}
&(\mathit{hubIOStrProp} + \mathit{hubIOStdProp}) \cdot \mathit{hubIOFRate} + \\
&(\mathit{termStrProp} + \mathit{termStdProp}) \cdot \mathit{termFRate} \cdot 2 + \\
&(\mathit{lnkAttchStrProp} + \mathit{lnkAttchStdProp}) \cdot \mathit{lnkAttchFRate} + \\
&(\mathit{nodeIOStrProp} + \mathit{nodeIOStdProp}) \cdot \mathit{nodeIOFRate} + \\
&\\
&[\mathit{hubIOFlipProp} \cdot \mathit{hubIOFRate} + \\
&(\mathit{termLossProp} + \mathit{termFlipProp}) \cdot \mathit{termFRate} \cdot 2 + \\
&(\mathit{lnkAttchDisProp} + \mathit{lnkAttchFlipProp}) \cdot \mathit{lnkAttchFRate} + \\
&\mathit{nodeIOFlipProp} \cdot \mathit{nodeIOFRate}] \cdot \mathit{ctrlFlipCov} + \\
&\\
&(\mathit{ctrlStrProp} + \mathit{ctrlStdProp}) \cdot \mathit{ctrlFRate} + \\
&\mathit{ctrlFlipProp} \cdot \mathit{ctrlFRate} \cdot (\mathit{ctrlFlipCov} \cdot \mathit{ctrlItselfIsoCov}) + \\
&\\
&(\mathit{nodeIOStrProp} + \mathit{nodeIOStdProp}) \cdot \mathit{nodeIOFRate} + \\
&(\mathit{termStrProp} + \mathit{termStdProp}) \cdot \mathit{termFRate} \cdot 2 + \\
&(\mathit{lnkAttchStrProp} + \mathit{lnkAttchStdProp}) \cdot \mathit{lnkAttchFRate} + \\
&(\mathit{hubIOStrProp} + \mathit{hubIOStdProp}) \cdot \mathit{hubIOFRate}
\end{aligned}$$

and

$$
\begin{aligned}
&(\mathit{hubIOFlipProp} \cdot \mathit{hubIOFRate} + \\
&(\mathit{termLossProp} + \mathit{termFlipProp}) \cdot \mathit{termFRate} \cdot 2 + \\
&(\mathit{lnkAttchDisProp} + \mathit{lnkAttchFlipProp}) \cdot \mathit{lnkAttchFRate} + \\
&\mathit{nodeIOFlipProp} \cdot \mathit{nodeIOFRate}) \cdot (1.0 - \mathit{ctrlFlipCov}) + \\
&\\
&\mathit{ctrlFlipProp} \cdot \mathit{ctrlFRate} \cdot (1.0 - \mathit{ctrlFlipCov} \cdot \mathit{ctrlItselfIsoCov}) + \\
&\\
&\mathit{nodeIOFlipProp} \cdot \mathit{nodeIOFRate} + \\
&(\mathit{termLossProp} + \mathit{termFlipProp}) \cdot \mathit{termFRate} \cdot 2 + \\
&(\mathit{lnkAttchDisProp} + \mathit{lnkAttchFlipProp}) \cdot \mathit{lnkAttchFRate} + \\
&\mathit{hubIOFlipProp} \cdot \mathit{hubIOFRate})
\end{aligned}
$$

In fact, these expressions are equal to those used by the activity *ncFailureMode* of the *nodeConnsT* submodel. This is because a Node Connection is composed of the same entities in CANcentrate and in ReCANcentrate, and the error-containment capabilities of the CAN controller are exactly the same in both infrastructures. A detailed explanation of these expressions can thus be found in Section 10.7.2.

The second and the third cases of the activity *ncFailureMode* set a token in *stuckLP* and *flipLP* respectively. In this way, *nodeConnsR* indicates to the *fauLPevalAtHub* coverage submodel how the Node Connection failure manifests; so that *fauLPevalAtHub* can evaluate whether or not the corresponding hub isolates the fault. However, as we pointed out when explaining the *nodeKernelsR* submodel, *fauLPevalAtHub* does not proceed with the coverage process when it receives a token in *stuckLP* or *flipLP*. In contrast, it starts up when it observes a token in the place *fauALP*. In this sense, we showed in Section 12.4.4 that *nodeKernelsR* sets a token in this place when necessary. Conversely, *nodeconnsR* does not directly compel *fauLPevalAtHub* to take over by setting a token in *fauALP*. Instead, it activates the *fauLPevalAtNode* coverage submodel by writing a token in the place *newFauBranch*. Afterwards, *fauLPevalAtNode* will be the responsible for activating *fauLPevalAtHub* (see Figure 12.2). As explained at the beginning of current section, *fauLPevalAtNode* evaluates whether or not the node to which the Node Connection region belongs tolerates the failure and, thus, can continue communicating using its other Node Connection region. A detailed description of *fauLPevalAtNode* will be carried out in Section 12.4.8.

Going back to the activity *ncFailure*, let us explain what actions are carried out when it selects its second case, i.e. the case that models the failure of a Node Con-

nection that is placed at an already faulty (and isolated) branch. When this happens, *ncFailure* transfers one token from *okNodeConns* to the place *alFauBranch*. This token activates the instantaneous activity *ncFbFailureMode*, which evaluates whether the fault manifest as ofm or not. If affirmative, *ncFbFailureMode* selects its first case and initiates a path of the coverage process that is intended to evaluate if the ofm errors generated by the fault propagate throughout the system. For this purpose, *ncFbFailureMode* uses an output gate that sets a token in *evalFault* and in *ofmNcFB*. Notice that this last token is the one that actually compels the *oufFauEval* submodel to evaluate the propagation of the ofm errors. Certainly, to activate *oufFauEval* may seem counterintuitive because this point of the model is reached if the faulty Node Connection is placed at an already faulty (and isolated) branch and, thus, one may think that the errors generated by the ofm fault cannot propagate. However, notice that an ofm Node Connection could lead its Node Kernel to fail in an ofm manner, so that this Node Kernel sends errors through its other Node Connection. For instance, this could happen if the Controller placed at the Node Connection that fails exhibits an ofm failure and notifies its Node Kernel about the reception of nonexistent frames. Therefore, it is necessary that *ofmFauEval* analyzes whether or not the ofm errors propagate through that other Node Connection of the node.

Finally, if the activity *ncFbFailureMode* decides that the Node Connection fails in a way that is included in our fault model, it selects its second case and it does not perform any further action (the second case of *ncFbFailureMode* is left unconnected). This is basically because the errors generated by the faulty Node Connection cannot propagate through its corresponding hub port, since that port (branch) was already faulty and, thus, isolated. Moreover, since the fault is included in our fault model, it cannot provoke the failure of the corresponding Node Kernel and thus, conversely to what happens with an ofm Node Connection, the fault cannot indirectly provoke the transmission of errors through the other Node Connection of the node.

12.4.6 *hubInConns* submodel

The *hubInConns* submodel models faults happening at any of the Hub Interconnection regions of ReCANcentrate. As explained in Section 12.4.1, a Hub Interconnection region includes all the entities that constitute a sublink: one Attachment entity (the CAN cable and the two connectors of both its ends), two Hub IO entities (each one corresponding to a different hub) and two Termination entities. Notice that each interlink includes two Hub Interconnection regions, one for each direction. This means that a given Hub Interconnection region is used by one hub to

send its own contribution to the other hub.

If the fault is within our fault model and it affects a Hub Interconnection region that was not already isolated, the *hubInConns* submodel additionally compels the *fauIPevalAtHubs* coverage submodel to start the corresponding coverage process. On the one hand, *fauIPevalAtHubs* evaluates if the hub that receives the contribution of the other hub through that interconnection region isolates the fault. On the other hand, it evaluates if the Hub Interconnection failure implies that all the interconnections between both hubs are exhausted and, thus, whether or not the hubs become decoupled (the hubs become decoupled if there are not enough Hub Interconnections that allow them to exchange their contributions).

The details of *fauIPevalAtHubs* will be explained in Section 12.4.11. However, let us point out that this submodel can further compel the *fauHubEvalAtHub* submodel to proceed with the coverage process (see Figure 12.2). As will be described, this basically happens when the hub that receives the contribution from the faulty Hub Interconnection region does not isolate the fault, so that it is necessary to assess if the other hub can contain the errors this hub will propagate.

Figure 12.6 shows the structure of the *hubInConns* submodel. The marking of the place *okInConnsToA* represents the number of Hub Interconnection regions through which the hub A receives the contribution of the hub B. Analogously, *okInConnsToB* models the number of Hub Interconnection regions through which the hub B receives the contribution of the hub A. The initial marking of each one of these places is the number of interlinks, which is specified by means of the parameter *numInterlinks* (see Table 12.1).

As mentioned in Section 12.4.2, some submodels of ReCANcentrate are symmetric in the sense that they model actions carried out by a hub twice (they model the actions carried out by the hub A and the actions performed by the hub B). The *hubInConns* submodel is one of these submodels because it represents faults happening at Hub Interconnection regions that carry the contribution of the hub A, as well as at Hub Interconnection regions that convey the contribution of the hub B. Therefore, from now on we will explain only the actions *hubInConns* performs when a Hub Interconnection region through which the hub A receives the contribution of the hub B fails.

The activity *hiAFailure* models the Time To Failure of the surviving Hub Interconnection regions that carry the contribution of the hub B to the hub A. It is exponentially distributed and its failure rate is calculated as in the previous regions submodels, i.e. by multiplying the number of surviving regions by the sum of the failure rates of the entities that constitute the region:

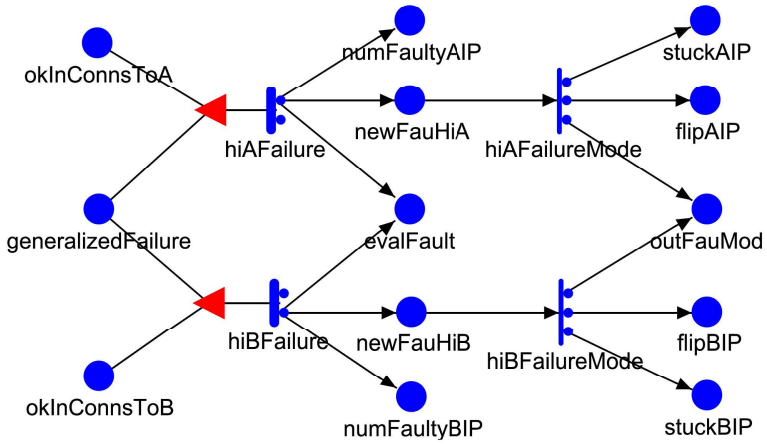

Figure 12.6: hubInConns submodel

$$okInConnsToA \rightarrow Mark() \cdot slnkAttchFRate + 2 \cdot hubIOFRate + 2 \cdot termFRate)$$

As can be seen in Figure 12.6, *hiAFailure* has two cases. The first one (the upper case) corresponds to a situation in which the Hub Interconnection region is placed at a non-faulty inbranch of the hub A, i.e. to a situation in which the Hub Interconnection region represents a non-faulty (and thus not isolated) sublink that conveys the contribution of the hub B to the hub A. The second one models the opposite situation, in which the inbranch was already faulty and thus isolated by the hub A. In principle, since *hiAFailure* only models faults happening at non-faulty Hub Interconnections, it may seem that *hiAFailure* should always select its first case. Nevertheless, an inbranch becomes faulty not only when the corresponding Hub Interconnection region's entities suffers from a fault, but also when (1) a hub fails, or (2) when the hubs become decoupled (see Section 12.4.2). Therefore, the number of non-faulty inbranches may be lower than the number of Hub Interconnection regions that are not faulty.

In order to know what is the actual number of faulty inbranches, we define the place *numFaultyAIP*, whose marking represents the total number of inbranches that carry the contribution of hub B to hub A and that have failed so far. Likewise, the marking of place *numFaultyBIP* represents the total number of faulty inbranches that are aimed at carrying the contribution of hub A to hub B. In this sense notice that the first case of *hiAFalure* sets a token on *numFaultyAIP*, thereby reflecting that a new inbranch fails due to the failure of a Hub Interconnection. Additionally, when

a hub fails or when the hubs become decoupled, the marking of *numFaultyAIP* (and of *numFaultyBIP*) is forced to be equal to the number of interlinks, in order to reflect that all the inbranches of the system are faulty. This is respectively done by the *fauHubEvalAtHub* and the *fauIPevalAtHubs* submodels, as will be explained in Sections 12.4.12 and 12.4.11.

From the above discussion, it is easy to see that the number of non-faulty inbranches that carry the contribution of the hub B to the hub A is $numInterlinks - numFaultyAIP$. Therefore, the expressions that specify the probabilities with which *hiAFalure* chooses its first and second cases are (respectively):

$$\frac{(numInterlinks - numFaultyAIP \rightarrow Mark())}{okInConnsToA \rightarrow Mark()}$$

and

$$\frac{(okInConnsToA \rightarrow Mark() - (numInterlinks - numFaultyAIP \rightarrow Mark()))}{okInConnsToA \rightarrow Mark()}$$

The fraction of the first expression is the probability that the inbranch corresponding to Hub Interconnection that suffers from a fault was not faulty, whereas the fraction of the second expression is the probability that this inbranch was already faulty. It is worth noting that the value of each one of these fractions can only be 0 or 1. On the one hand, the value of $numInterlinks - numFaultyAIP$ is equal to the marking of *okInConnsToA*, i.e. it is equal to the number of surviving Hub Interconnection regions, as long as the hub B does not fail and the hubs are coupled. Thus, in this case, the fraction of the first and second expressions are 1 and 0 respectively. On the other hand, if the hub B fails or the hubs are decoupled, then $numInterlinks - numFaultyAIP$ becomes 0 and, thus, the values of these fractions become 0 and 1 respectively.

The second case of *hiAFailure* is left unconnected. On the one hand, since the inbranch corresponding to the faulty Hub Interconnection was already faulty and isolated, there is no need to increase the marking of *numFaultyAIP* to reflect any change in the number of faulty inbranches that carry the contribution of the hub B to the hub A. On the other hand, the errors the faulty Hub Interconnection generates cannot propagate beyond an already isolated inbranch and, thus, there it is not necessary to initiate any path of the coverage process.

Conversely, the first case of *hiAFailure* is connected to three places. Specifically, the *hubInConns* submodel proceeds as follows when this case is selected. On the

one hand, it increases the marking of the place *numFaultyAIP* in one unit to record the failure of a new inbranch. On the other hand, it initiates a path of the coverage process that evaluates how the fault is treated. For this purpose it sets a token in the place *evalFault*, which disables the *ReCANcentrateFaiEval* submodel, and writes a token in the place *newFauHiA*. This last token enables the instantaneous activity *hiAFailureMode*, which decides if the fault manifests as stuck-at, bit-flipping or in a way that is not included in our fault model.

The proportions of the cases of *hiAFailureMode* are calculated following Equation 10.10. The proportions of the two first cases are (respectively):

$$
\begin{aligned}
&(\mathit{hubIOStrProp} + \mathit{hubIOStdProp}) \cdot \mathit{hubIOFRate} \cdot 2 + \\
&(\mathit{slnkMedStrProp} + \mathit{slnkMedStdProp}) \cdot \mathit{slnkMedFRate} + \\
&(\mathit{termStrProp} + \mathit{termStdProp}) \cdot \mathit{termFRate} \cdot 2
\end{aligned}
$$

and

$$
\begin{aligned}
&\mathit{hubIOFlipProp} \cdot \mathit{hubIOFRate} \cdot 2 + \\
&(\mathit{slnkMedDisProp} + \mathit{slnkMedFlipProp}) \cdot \mathit{slnkMedFRate} + \\
&(\mathit{termFlipProp} + \mathit{termLossProp}) \cdot \mathit{termFRate} \cdot 2
\end{aligned}
$$

When the activity *hiAFailureMode* selects its first and second cases, it sets a token in the place *stuckAIP* and in *flipAIP* respectively. It shares these places with the *fauIPevalAtHubs* coverage submodel, thereby compelling it to evaluate if the hub A is able to isolate the corresponding type of fault exhibited by the Hub Interconnection region at the corresponding inbranch. Notice that the *hubInConns* submodel also shares the places *stuckBIP* and *flipBIP* with *fauIPevalAtHubs*. The meaning of these places is analogous to the one of *stuckAIP* and *flipAIP*: when the *fauIPevalAtHubs* coverage submodel receives a token in any of these two last places, it evaluates if the hub B successfully isolates the corresponding failure mode at the inbranch. See Section 12.4.11 for a detailed description of the *fauIPevalAtHubs* submodel.

Finally, the expression of the last case of the activity *hiAFailure* reflects the proportion with which the Hub Interconnection region fails exhibiting an ofm fault. In accordance with Equation 10.10, this proportion is:

$$
\begin{aligned}
&\mathit{slnkAttchOfmProp} \cdot \mathit{slnkAttchFRate} + 2 \cdot \mathit{hubIOOfmProp} \cdot \mathit{hubIOFRate} + \\
&2 \cdot \mathit{termOfmProp} \cdot \mathit{termFRate}
\end{aligned}
$$

When *hiAFailure* chooses this third case, it merely sets a token in *outFauMod*, which leads the *ReCANcentrateFaiEval* submodel to diagnose the failure of the overall system. As already said in Section 12.3.3, we suppose that a hub that receives errors it cannot contain (which is the case of errors generated by an ofm fault) propagates them through all its outgoing ports. Therefore, the hub A does not only propagate the ofm errors to its own nodes, but also to the hub B and, then, to the nodes connected to this other hub.

12.4.7 *hubKernels* submodel

The *hubKernels* submodel models faults happening at any of the two Hub Kernel regions of ReCANcentrate. Additionally, *hubKernels* sometimes needs to initiate a path of the coverage process that evaluates how the faulty hub is treated and tolerated. This is depicted in Figure 12.2, where *hubKernels* compels the coverage submodels *ofmFauEval*, *fauHubEvalAtHub* or *fauHubEvalAtNodes* to take over, or it simply does not initiate any coverage process.

More specifically, *hubKernels* initiates the coverage process in the following situations. First, if one of the hubs exhibits an ofm failure and both hubs were not faulty before the fault occurred, then *hubKernels* compels *outFauMode* to assess if the corresponding ofm errors propagate throughout all the system.

Second, *hubKernels* compels *fauHubEvalAtHub* to take over when the fault is within our fault model and both hubs were coupled before the fault occurred. In this case, it is necessary to evaluate if the hub that remains non-faulty is able to contain the errors generated by the hub whose kernel has failed. Notice that *fauHubEvalAtHub* is also the submodel that proceeds with the coverage process when the hubs are coupled, but a hub is not able to isolate a fault in any of its uplink or sublink hub ports. This is because, as said in Section 12.3.3, a hub that is not able to isolate a fault, broadcasts the errors generated by that fault through all its downlinks and all its outgoing sublinks. Such a coincidence is shown in Figure 12.2, where there is a dedicated arrow connecting the submodels *fauLPsEvalAtHubs*, *fauLPevalAtHub*, *fauIPevalAtHubs* and *hubKernels* with *fauHubEvalAtHub*.

Finally, if the fault is within our fault model, but the hubs were decoupled, then the *hubKernels* submodel activates *fauHubEvalAtNodes* instead of *fauHubEvalAtHub*. Notice that the fact that both hubs were decoupled implies that all the sublink ports (all inbranches) were already disabled by both hubs. Thus, in this situation, as already mentioned in Section 12.4.4, it is not necessary to evaluate if the hub that remains non-faulty is able to contain the errors the hub that fails generates. However, also notice that the system was not faulty even though the hubs

were decoupled. This means that the nodes were able to tolerate the hub decoupling (when it occurred) and, hence, that they were using both hubs independently for communicating. Therefore, when a hub fails in this situation, it is necessary to reflect that the nodes will not be able to communicate through that hub. In order to do that, *hubKernels* compels *fauHubEvalAtNodes* to take over. Notice that the role of *fauHubEvalAtNodes* was already introduced in Section 12.4.4, since the submodels *fauLPevalAtHub* and *fauLPsEvalAtHubs* directly activate it when the hubs are decoupled and one hub is not able to isolate an uplink port.

The structure of the *hubKernels* submodel is shown in Figure 12.7. The marking of the place *okHubKernels* indicates the number of hubs that are not faulty. Its initial marking is, thus, equal to 2. The activity *hkFailure* models the Time To Failure distribution of the surviving hubs. This distribution is exponential and its failure rate is: $hubCoreFRate \cdot okHubKernels \rightarrow Mark()$.

When this activity fires, it is necessary to decide which path of the coverage process is initiated to model how the Hub Kernel failure is isolated and/or tolerated. This is done in several steps. The first one of these steps is carried out by the activity *hkFailure* itself. On the one hand, this activity checks if the Hub Kernels of both hubs were not faulty before the fault occurred. If one of the Hub Kernels was already faulty, then the Hub Kernel that has just failed would be the only one that was still operative. In such a situation, there is no need to initiate any path of the coverage process, as no hub will be available for communicating. This is modelled by the second case of *hkFailure*, which sets a token in the place *noAvailHub*, thereby compelling the *ReCANcentrateFaiEval* submodel to diagnose the overall system as faulty. On the other hand, if the Hub Kernels of both hubs were not faulty, *hkFailure* must decide which one of the two hubs the kernel that fails belongs to. In particular, the first case of *hkFailure* models the failure of the kernel of the hub A, whereas the third one represents the kernel failure of the hub B.

The activity *hkFailure* respectively calculates the proportions of its first, second and third cases by means of the following *if* clauses.

(Case 1)

if ($okHubKernels \rightarrow Mark() > 1$)
 return 0.5;
else
 return 0.0;

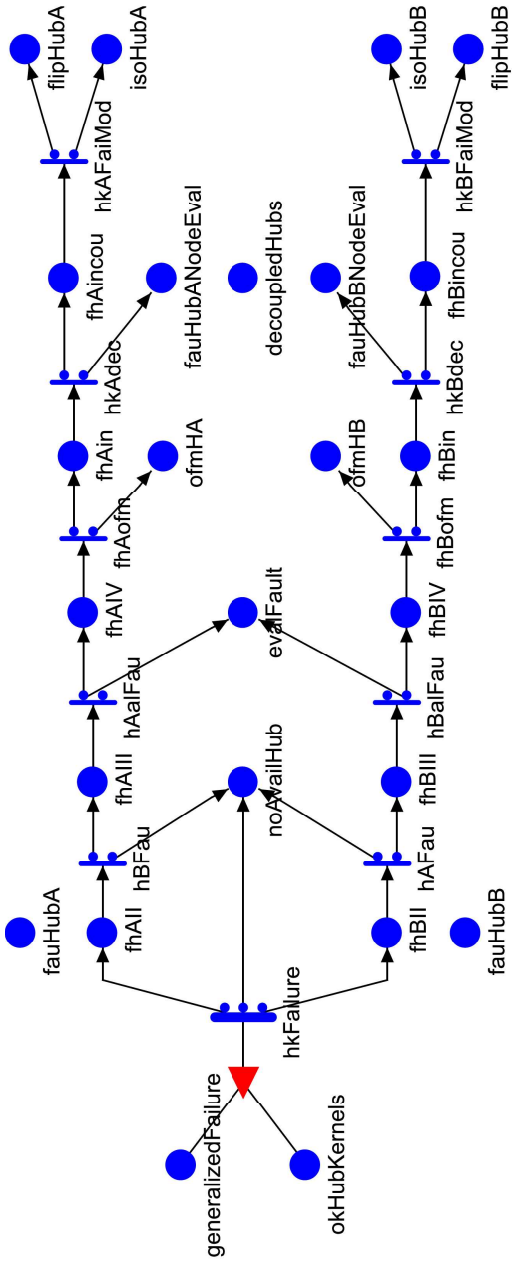

Figure 12.7: hubKernels submodel

(Case 2)

if (*okHubKernels* $\rightarrow$ *Mark()* $== 1$)
 return 1.0;
else
 return 0.0;

(Case 3)

if (*okHubKernels* $\rightarrow$ *Mark()* > 1)
 return 0.5;
else
 return 0.0;

If only one Hub Kernel was not faulty before the fault occurred, the marking of *okHubKernels* is equal to 1 when *hkFailure* fires. Thus, the proportion of the second case is 1.0, whereas the proportion of the other ones is 0.0. In contrast, if both Hub Kernels were non-faulty, the marking of *okHubKernels* is 2. Hence, the proportion of the second case of *hkFailure* is 0 whereas the proportions of the first and third cases are 0.5. Notice that in this last situation, the first and third cases are selected exactly with the same proportion. This is because both Hub Kernels have the same probability of failure.

If *hkFailure* selects its first or its third case, the *hubKernels* submodel carries out additional actions to decide with path of the coverage process it initiates. In this sense, notice that the actions carried out when *hkFailure* selects its first case are analogous to the ones performed when *hkFailure* chooses its third one. Thus, from now on, we will only focus on the actions that follow when *hkFailure* takes its first case, i.e. on the actions performed when *hkFailure* decides that the faulty kernel belongs to the hub A.

When the activity *hkFailure* selects its first case, it sets a token in the place *fhAII*. This token enables the instantaneous activity *hBFau*, which evaluates if the hub B is faulty. This is because the fact that the kernel of the hub B is non-faulty does not necessarily mean that this hub is non-faulty. Actually, the hub B is already faulty if, previously, it was not able to isolate a fault in any of its ports.

The first case of *hBFau* is selected if the hub B is non-faulty, whereas the second case is chosen otherwise. The proportions of the first and second cases of *hBFau* are respectively calculated as $1 -$ *fauHubB* $\rightarrow$ *Mark()* and *fauHubB* $\rightarrow$ *Mark()*; where *fauHubB* is a place shared among different submodels and whose marking

indicates whether or not the hub B is faulty. Specifically, as will be explained in Sections 12.4.13 and 12.4.14, a token is set in *fauHubA* and in *fauHubB*) when the hub A and the hub B become faulty respectively (this is done independently of whether the hub failure is provoked by the failure of its own kernel or because it is not able to isolate a fault in any of its ports).

If the activity *hBFau* selects its second case, it sets a token in the place *noAvailHub* and does not initiate any path of the coverage process, since both hubs are faulty. Otherwise, if it selects if first case, it sets a token in the place *fhAIII*. This token enables the instantaneous activity *hAalFau*, whose role is similar to the one of the activity *hBFau*; *hAalFau* evaluates if the hub A was already faulty before its kernel failed. Notice that the result of this evaluation is only affirmative if, previously, the hub A was not able to isolate a fault in any of its ports. The activity *hAalFau* bases its decision on the marking of the place *fauHubA*, whose meaning is analogous to the one of *fauHubB*. Specifically, the first case of *hAalFau* models the situation in which the hub A was not already faulty and it is selected with proportion $1 - \textit{fauHubA} \rightarrow \textit{Mark()}$. The second case of this activity represents the opposite situation and its proportion is calculated as $\textit{fauHubA} \rightarrow \textit{Mark()}$.

As can be observed in the Figure 12.7, the second case of the activity *hAalFau* is left unconnected. This is because if the hub A was already faulty, it was also already isolated and, thus, the errors generated by its kernel failure cannot pollute the system. In contrast, the first case of the activity *hAalFau* sets a token in the place *fhAIV*. This token enables the instantaneous activity *fhAofm*, which evaluates if the Hub Kernel exhibits an ofm failure. The first case of this activity is chosen when the kernel exhibits a failure mode that is included in our fault model, whereas its second case is selected otherwise. Their proportions are respectively calculated as $(1.0 - \textit{hubCoreOfmProp} \rightarrow \textit{Mark()})$ and $\textit{hubCoreOfmProp} \rightarrow \textit{Mark()}$.

When *fhAofm* selects its second case, it sets a token in the place *ofmHA*, in order to indicate to the *ofmFauEval* submodel that the kernel of the hub A exhibits an ofm failure. As already explained, the *ofmFauEval* submodel represents the path of the coverage process that models how the ofm errors propagate. In particular, when it receives a token in *ofmHA*, *ofmFauEval* evaluates if the ofm errors generated by the kernel of the Hub A propagate throughout both stars, or they only affect the nodes exclusively connected to that hub (see Section 12.4.14 for a detailed explanation concerning this issue).

Conversely, when *fhAofm* chooses its first case, it sets a token in the place *fhAin*, which activates the instantaneous activity *hkAdec*. This activity decides what is the path of the coverage process that must be initiated to evaluate how the errors generated by a faulty node kernel that manifest in a way that is included in our

fault model are treated and/or tolerated. Specifically, this activity checks if both hubs were coupled before the kernel of the hub A failed. For this purpose, *hkAdec* merely consults the marking of the place *decoupledHubs*. As explained in Section 12.4.3, the marking of this place is 1 if both hubs are decoupled and nodes are using them independently. More specifically, the first case of *hkAdec* is chosen when the hubs are coupled, whereas the second case is selected otherwise. Thus, the proportions of the first and second cases of *hkAdec* are respectively obtained as $1 - decoupledHubs \rightarrow Mark()$ and $decoupledHubs \rightarrow Mark()$.

When *hkAdec* selects its second case, i.e. when the hubs were decoupled, it sets a token in the place *fauHubANodeEval*. A token in this places activates the *fauHubEvalAtNodes* coverage submodel, which will be thoroughly explained in Section 12.4.13. As pointed out at the beginning of this section, the fact that the system is not faulty and that both hubs were decoupled and operative before the kernel of the hub A failed, implies that the nodes were using each hub as an independent communication domain. Therefore, what *fauHubANodeEval* does is to evaluate whether or not the nodes are able to isolate the faulty hub (the hub A in this case) in order to continue communicating through the other hub. Also notice that the path of the coverage process initiated by *hubKernesR* ends at *fauHubANodeEval*, which activates the *ReCANcentrateFaiEval* submodel, as depicted in Figure 12.2.

Regarding the first case of *hkAdec*, it is chosen when the hubs were coupled before the kernel of the hub A failed. This case sets a token in the place *fhAincou*, thereby enabling the activity *hkAFaiMod*. This activity instantaneously fires to decide which failure mode of those that are included in our fault model the kernel of the hub A exhibits (notice that this point of the *hubKernels* submodel is only reached if, previously, the activity *fhAofm* has decided that the failure of the kernel of the hub A does not manifest as ofm). The proportions of the two first cases of the activity *hkAFaiMod* are calculated in accordance with Equation 10.10. Specifically, the fist case corresponds to a Hub Kernel's bit-flipping failure, whereas the second one models a Hub Kernel's stuck-at. Thus, the expressions of these cases are (respectively) *hubCoreFlipProp* and $1.0 - hubCoreOfmProp - hubCoreFlipProp$). Notice that the parameters that specify the failure mode's proportions of a Hub Core entity do not distinguish between a stuck-at-recessive and a stuck-at-dominant fault, since we assume that the hubs and the nodes are able to isolate these two types of Hub Core's faults with the same coverage (see Table 12.1).

Finally, also notice that a token in *flipHubA* or *isoHubA* compels the *fauHubEvalAtHub* coverage submodel to continue with the coverage process. As it will be explained in Section 12.4.12, this submodel evaluates whether or not the hub B is able to isolate the corresponding type of fault at its incoming sublinks (at the

corresponding in branch).

12.4.8 *fauLPevalAtNode* submodel

The *fauLPevalAtNode* coverage submodel evaluates if a node is able to tolerate the failure of one of its Node Connection regions and, thus, if it is able to continue communicating using its other Node Connection region. This was mentioned in Section 12.4.5, where we explained that the *nodeConnsR* submodel compels *fauLPevalAtNode* to proceed with the coverage process when a Node Connection region placed at a non-faulty branch fails and exhibits a failure mode that is included in our fault model. Moreover, we also indicated therein that when *fauLPevalAtNode* finishes its evaluation, it compels the *fauLPevalAtHub* submodel to assess if the corresponding hub isolates the faulty Node Connection region. The interconnections among these submodels is shown in Figure 12.2.

Figure 12.8 shows the structure of the *fauLPevalAtNode* submodel. It starts performing whenever the *nodeConnsR* submodel sets a token in the place *newFauBranch*. This token enables the instantaneous activity *selFauLPHub*, which aims at deciding whether the Node Connection region that fails is connected to the hub A or to the hub B. Notice that the *nodeConnsR* submodel did not make this decision, it merely modelled how the Node Connection failure manifests. See Section 12.4.5 for more details.

If *selFauLPHub* decides that the Node Connection region is connected to the hub A, then it selects its first case; otherwise it chooses its second one. The proportions of each one of these cases is calculated dividing the number of Node Connection regions that are connected to the corresponding hub by the total number of Node Connection regions. It is important to note that we have to take into account only those Node Connection regions that were placed at non-faulty branches before the fault occurred. In this sense, notice that all the Node Connection regions that were placed in a non-faulty branch belonged to either an okAB, an okA, an okB, a stopA or a stopB node. Thus, we can use the markings of the places *okABNodes*, *okANodes*, *okBNodes*, *stopANodes* and *stopBNodes* (which still have not changed as a consequence of the fault) to indirectly calculate the number of Node Connection regions we are interested in. More specifically, *selFauLPHub* calculates the number of Node Connection regions connected to the hub A as:

$$okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark()$$

Similarly, the amount of Node Connection regions that are connected to the hub

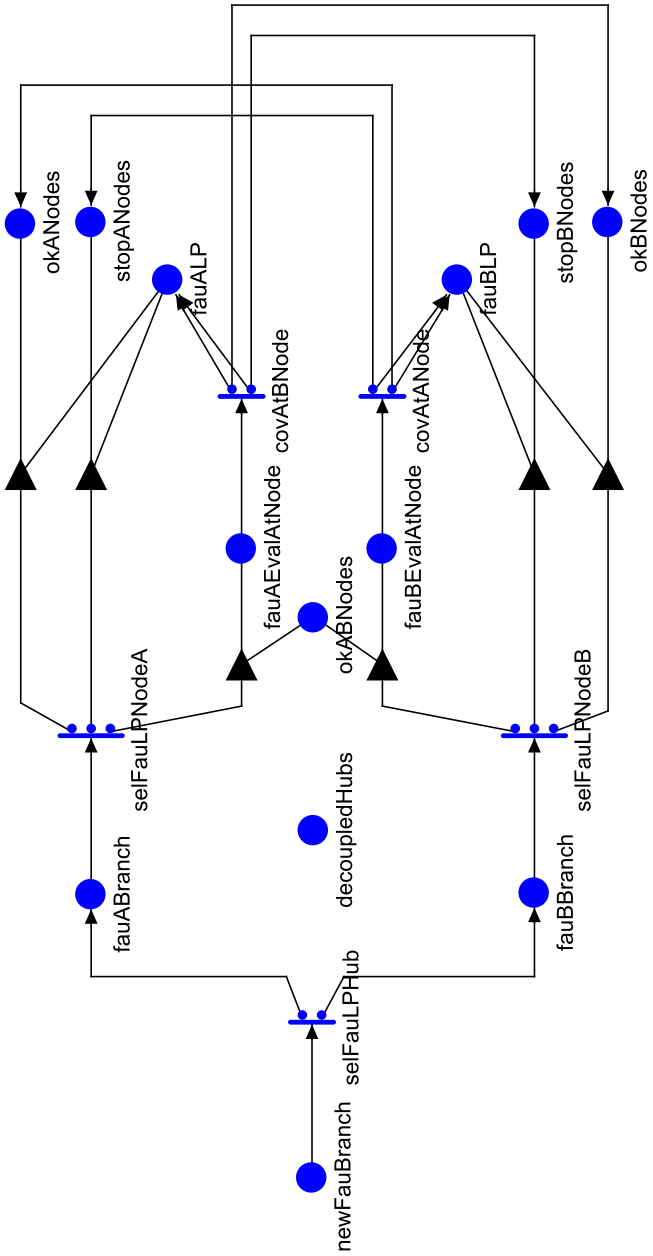

Figure 12.8: fauLPevalAtNode submodel

B and that must be taken into account is:

$$okABNodes \rightarrow Mark() + okBNodes \rightarrow Mark() + stopBNodes \rightarrow Mark()$$

As concerns the total number of Node Connection regions that were placed in a non-faulty branch, they can be obtained as:

$$2 \cdot okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark() + okBNodes \rightarrow Mark() + stopBNodes \rightarrow Mark()$$

Taking into account all these considerations, the proportions of the first and the second cases of the activity *selFauLPHub* are (respectively):

$$(okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark()) \; / \; (2 \cdot okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark() + okBNodes \rightarrow Mark() + stopBNodes \rightarrow Mark())$$

and

$$(okABNodes \rightarrow Mark() + okBNodes \rightarrow Mark() + stopBNodes \rightarrow Mark()) \; / \; (2 \cdot okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark() + okBNodes \rightarrow Mark() + stopBNodes \rightarrow Mark())$$

Similarly to what happens in some of the submodels we have explained so far, the actions the *fauLPevalAtNode* submodel carries out when the activity *selFauLPHub* chooses its first case are analogous to the ones it performs when it selects its second one. Thus, in the rest of this section, we will focus on the actions performed when the first case is taken.

As depicted in the Figure 12.8, the first case of *selFauLPHub* sets a token in the place *fauABranch*, thereby enabling the instantaneous activity *selFauLPNodeA*. This activity decides which is the specific kind of node the Node Connection belongs to. The first case corresponds to an okA node, the second one to a stopA node and the third one to an okAB node. Such a decision is indispensable for evaluating whether or not the node is able to tolerate the failure of the Node Connection

region. This is because, the capacity of a node for tolerating this failure depends on the node's type. More specifically, the node has the possibility of tolerating the fault only if it is an okAB node. This is because this is the only case in which, after the failure, the node still has a non-faulty Node Connection that it can potentially use for communicating.

The proportions of the three cases of the activity *selFauLPNodeA* are calculated following the same strategy as for the proportions of the cases of the activity *selFauLPHub*. The probability with which *selFauLPNodeA* selects if first case is the the number of okA nodes divided by the total number of nodes that have a Node Connection placed in a non-faulty branch of the hub A:

$$\frac{okANodes \rightarrow Mark()}{okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark()}$$

Analogously, the proportions of the second and the third cases are (respectively):

$$\frac{stopANodes \rightarrow Mark()}{okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark()}$$

and

$$\frac{okABNodes \rightarrow Mark()}{okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark()}$$

As depicted in Figure 12.8, each one of the three cases of the activity *selFauLPNodeA* is connected to a dedicated output gate. The output gate corresponding to the first case decreases the marking of *okANodes* in one unit and sets a token in *fauALP*. On the one hand, since the first case of *selFauLPNodeA* is chosen to model that the Node Connection region that fails belongs to an okA node, this output gate decreases the marking of *okANodes* to reflect that such a node becomes a noConn node. On the other hand, the output gate sets a token in *fauALP* to compel the *fauLPevalAtHub* submodel to proceed with the coverage process. As indicated at the beginning of this section, *fauLPevalAtNode* will compel *fauLPevalAtHub* to assess if the hub A isolates the faulty Node Connection region. The output gate of the second case of *selFauLPNodeA* is analogous to the gate of the first case. It decreases the marking of *stopANodes* in one unit and sets a token in *fauALP*

As concerns the output gate of the third case of *selFauLPNodeA*, its actions are slightly different from the ones performed by the previous output gates. On the one

hand, it decreases the marking of the place *okABNodes* to reflect that the node will not be able to use one of its Node Connections anymore. On the other hand, it set a token in the place *fauAEvalAtNode*, thereby enabling the instantaneous activity *covAtBNode*. Besides writing a token in the place *fauALP*, *covAtBNode* assesses whether or not the node is able to tolerate the fault of the Node Connection region. As a result, it increases the marking of the places *okBNodes* and *stopBNodes* accordingly. More specifically, the first case of *covAtBNode* represents the situation in which the node tolerates the fault, so that it is still able to communicate using its connection to the hub B. For this reason, this case increases the marking of *okBNodes* in one unit. The second case models the opposite situation, in which the node is not able to use its connection to the hub B even though such a connection is not faulty. Thus, this case adds a token to *stopBNodes*.

The proportions with which *covAtBNode* selects its first and second cases depend on whether or not the hubs are coupled. As explained in Section 12.3.4, the coverage with which a node tolerates a fault that prevents it from communicating through one of the stars, when the hubs are coupled, is called *connCov* and its default value is of the 95%. In contrast, if the hubs are decoupled, this coverage is called *decConnCov* and its default value is of the 0% (see Table 12.1). In order to use the appropriate coverage, the activity *covAtBNode* consults the marking of the place *decoupledHubs*, which, as already mentioned, has a token if the hubs are decoupled. Specifically, the proportions of the first and second cases are respectively expressed as:

(Case 1)

if ($decoupledHubs \rightarrow Mark() == 0$)
 return *connCov*;
else
 return *decConnCov*;

(Case 2)

if ($decoupledHubs \rightarrow Mark() == 0$)
 return $1.0 - connCov$;
else
 return $1.0 - decConnCov$;

12.4.9 *fauLPsEvalAtHubs* submodel

As described in Section 12.4.4, when a faulty Node Kernel manifest in a way that is included in our fault model and generates errors that propagate through its uplinks to the hub A and the hub B, the *nodeKernelsR* submodel compels the *fauLPsEvalAtHubs* coverage submodel to evaluate whether or not each hub is able to contain these errors. As mentioned in that section, the submodel *nodeKernelsR* notifies about this situation by setting a token in the place *fauABLPs*. Additionally, it specifies the way in which the Node Kernel failure manifests at the uplink port of the hub A and at the uplink port of the hub B, by writing a token in the corresponding place: *fauABLPs*, *stuckStuckLPs*, *stuckFlipLPs*, *flipStuckLPs* and *flipFlipLPs*.

The structure of the *fauLPsEvalAtHubs* submodel is depicted in Figure 12.9. When it receives a token in the place *fauABLPs*, its activity *covABLPs* immediately fires (consuming the token) to decide whether or not each hub is able to contain the errors at its corresponding uplink port. The first case of the activity *covABLPs* models the situation in which only the hub B contains the errors; the second case represents the situation in which both hubs contain the errors; the third one corresponds to the case in which only the hub A contains the errors; and the last case models the scenario in which none of the hubs is able to contain the errors.

The proportions of each one of these cases is calculated taking into account the mode in which the fault manifests at the uplink port of each hub and, then, the coverage with which each one of the hubs isolates the corresponding type of fault in its uplink. For instance, the proportion of the first case, i.e. the probability that only the hub B isolates the fault, is calculated as:

$$
\begin{aligned}
&\mathit{stuckStuckLPs} \rightarrow \mathit{Mark()} \cdot 0.0 + \\
&\mathit{stuckFlipLPs} \rightarrow \mathit{Mark()} \cdot 0.0 + \\
&\mathit{flipStuckLPs} \rightarrow \mathit{Mark()} \cdot (1.0 - \mathit{flipLnkCov}) + \\
&\mathit{flipFlipLPs} \rightarrow \mathit{Mark()} \cdot (1.0 - \mathit{flipLnkCov}) \cdot \mathit{flipLnkCov}
\end{aligned}
$$

Each one of the lines of the above expression represents the probability with which only the hub B isolates its uplink port, provided that the fault manifests at the hub A and at the hub B in a specific way. The correspondence between a given line and a specific failure mode is achieved by using the marking of the place that represents the failure mode as a multiplying factor in the line. In this way, the lines that do not correspond to the way in which the fault actually manifests, i.e. that do not correspond to the place that has the token, are simplified to 0. Specifically, a token in *stuckStuckLPs* indicates that the fault manifests as stuck-at at both hubs. In

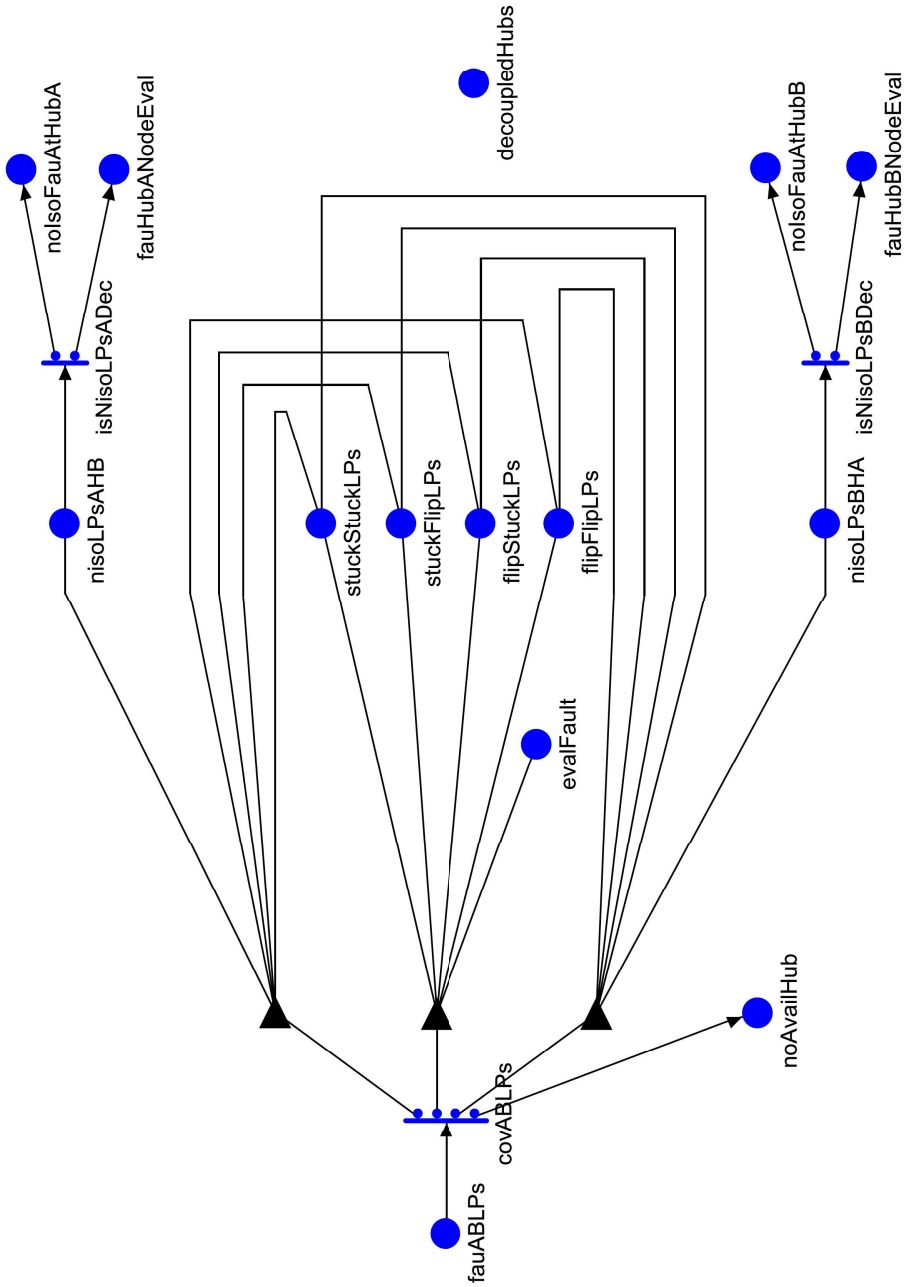

Figure 12.9: fauLPsEvalAtHubs submodel

such a situation, the line that calculates the probability that only the hub B isolates the fault is the first one. Since a hub always isolates a stuck-at port, it is impossible that only the hub B isolates the fault. Thus, the marking of *stuckStuckLPs* is multiplied by 0.0. The second line corresponds to the situation in which the fault manifests as stuck-at-recessive at the uplink port of the hub A and as bit-flipping at the uplink port of the hub B (the token is placed in *stuckStuckLPs*). Thus, the probability that only the hub B isolates the fault is 0 again. Conversely, when the fault manifest as bit-flipping at the hub A and as stuck-at at the hub B (the token is placed in *flipStuckLPs*), there is a possibility that only the hub B isolates the fault, since the hub A cannot isolate a bit-flipping fault with a perfect coverage. For this reason, in the third line, *flipStuckLPs* $\rightarrow$ *Mark()* is multiplied by the probability with which the hub A does not isolate a bit-flipping fault, i.e. by $1.0 -$ *flipLnkCov*. Finally, the last line is written following the same line of reasoning: *flipFlipLPs* $\rightarrow$ *Mark()* is multiplied by $(1.0 -$ *flipLnkCov*$) \cdot$ *flipLnkCov*, which is the probability with which the hub A does not isolate the bit-flipping fault and the hub B does.

The proportions of the rest of the cases of the activity *covABLPs* are calculated following the same strategy. For instance, the proportion of the second case, i.e. the case in which both hubs isolate their respective uplink ports, is:

$$
\begin{aligned}
&\textit{stuckStuckLPs} \rightarrow \textit{Mark()} \cdot 1.0 \cdot 1.0 + \\
&\textit{stuckFlipLPs} \rightarrow \textit{Mark()} \cdot 1.0 \cdot \textit{flipLnkCov} + \\
&\textit{flipStuckLPs} \rightarrow \textit{Mark()} \cdot \textit{flipLnkCov} \cdot 1.0 + \\
&\textit{flipFlipLPs} \rightarrow \textit{Mark()} \cdot \textit{flipLnkCov} \cdot \textit{flipLnkCov}
\end{aligned}
$$

In this case, in each line we multiply the marking of the place that specifies the way in which the Node Kernel fault manifests at each hub port by the coverage with which each hub isolates its uplink in each situation.

Next, we explain the actions that *fauLPsEvalAtHubs* carries out when its activity *covABLPs* fires and selects a case. If *covABLPs* chooses the first case, i.e. if only the hub B isolates the errors, a dedicated output gate performs the following actions. On the one hand, it resets the marking of the places that specify the failure modes in which the Node Kernel manifests at each uplink hub port. This is needed for letting the *nodeKernelsR* submodel to specify, in the future, how a new faulty Node Kernel manifests at both hubs.

On the other hand, since the hub A was not able to contain the errors, the output gate sets a token in *nisoLPsAHB* to continue with the path of the coverage process and to evaluate how these errors propagate and are treated. The activity *isNisoLPsADec* is the first one involved in the rest of this path. Specifically, it elucidates

whether or not the hubs are coupled with each other. The first case of this activity corresponds to the situation in which the hubs are coupled. In such a situation it is necessary to evaluate if the hub B isolates the sublinks through which it receives the errors propagated by the hub A. As already explained in Section 12.4.4, the submodel *fauHubEvalAtHub* carries out this evaluation (see Figure 12.2). The activity *isNisoLPsADec* compels this submodel to take over by setting a token in the place *noIsoFauAtHubA*. Also note from Figure 12.2 that if the hub B successfully contains the errors, the submodel *fauHubEvalAtHub* further compels the submodel *fauHubEvalAtNodes* to evaluate if the nodes connected to the hub A tolerate this hub failure, so that they can still communicate through the hub B. This will be explained later in Section 12.4.13.

As concerns the second case of the activity *isNisoLPsADec*, it corresponds to the situation in which the hubs are decoupled. In this situation, it is not necessary to evaluate if the hub B isolates the hub A. Nevertheless, as also explained in Section 12.4.4, the fact that both hubs are decoupled does not necessarily mean that the nodes are not able to communicate with each other using both hubs independently. In particular, if there are nodes that still use the hub A for communicating, it is necessary to asses if each one of these nodes is able to tolerate the failure of the hub A and can continue communicating using the hub B. The submodel *fauHubEvalAtNodes* is the responsible for this task and *isNisoLPsADec* compels it to take over by setting a token in the place *fauHubANodeEval*.

In order to calculate the probability with which the activity *isNisoLPsADec* must select its first and its second case, we use the marking of the place *decoupledHubs*. As already explained, a token in this place indicates that the hubs are decoupled, so that the nodes are using them independently. If there is a token in *decoupledHubs*, the activity *isNisoLPsADec* selects its first case; if there is no token, then it selects its second one. The expressions of both cases are: $1 - hubsDecoupled \rightarrow Mark()$ and $hubsDecoupled \rightarrow Mark()$ respectively.

Up to this point, we have explained what actions are performed when the activity *covABLPs* selects its first case. The meaning of the second case of the activity *covABLPs* was explained above: it corresponds to the situation in which both hubs isolate the faulty Node Kernel. Thus, when this case is chosen, the path of the coverage process that assesses how the errors generated by the Node Kernel are propagated and contained is finished. For this reason, as can be seen in Figure 12.9, the second case of the activity *covABLPs* is connected to an output gate that performs two simple actions. On the one hand, the output gate resets the marking of the places that specify the failure modes in which the Node Kernel manifests at each uplink hub port. As said before, this will allow the *nodeKernelsR* submodel to specify again how a new faulty Node Kernel manifests. On the other hand,

the output gate resets the marking of the place *evalFault*. As already explained in Section 12.4.4, the *nodeKernelsR* sets a token in this place to inhibit the *ReCANcentrateFaiEval* submodel from evaluating if an overall system failure occurs. Hence, by means of this action, the output gate enables again the *ReCANcentrateFaiEval* submodel, so that it can evaluate whether or not the Node Kernel failure has provoked an overall failure.

As concerns the third case of the activity *covABLPs*, i.e. the case in which only the hub A contains the errors generated by a faulty Node Kernel, notice that the actions carried out when it is selected are analogous to the actions performed when *covABLPs* selects its first case, i.e. when the hub B was the only hub that contains the errors.

Finally, as we also said before, the fourth case of the activity *covABLPs* models the situation in which no hub isolates the faulty Node Kernel. This means that from then on no node will be able to communicate. To model this fact, the fourth case of *covABLPs* sets a token in the place *noAvailHub*. As explained in Section 12.4.3, the *ReCANcentrateFaiEval* submodel immediately diagnoses an overall failure when a token is set in this place.

12.4.10 *fauLPevalAtHub* submodel

The *fauLPevalAtHub* coverage submodel evaluates whether or not a given hub is able to contain the errors it receives through an uplink port, when this errors are generated by a fault that is included in our fault model. At this point, it is important to highlight the difference between this submodel and the one just explained in previous section: *fauLPsEvalAtHubs*. Notice that *fauLPsEvalAtHubs* proceeds with the appropriate path of the coverage process when a non-ofm fault in a Node Kernel generates errors that reach the corresponding uplink ports of both hubs. In contrast, *fauLPevalAtHub* takes over when a non-ofm fault generates errors that affect the uplink port of one hub only. More specifically, *fauLPevalAtHub* proceeds with the coverage process in the following cases. On the one hand, as can be seen in Figure 12.2, *fauLPevalAtHub* is activated by *nodeKernelsR* when a faulty Node Kernel fails in such a way that it sends errors to just one hub (see Section 12.4.4 for more details). On the other hand, *fauLPevalAtHub* always participates in the path of the coverage process initiated when a Node Connection region fails. This is because the errors generated by a Node Connection can reach the uplink port of one hub only. In particular, as depicted in Figure 12.2, *fauLPevalAtHub* takes over after *fauLPevalAtNode* assesses if the corresponding node tolerates a Node Connection region failure (see Sections 12.4.5 and 12.4.8).

Figure 12.10 shows the structure of *fauLPevalAtHub*. When a fault that is included in our fault model occurs in a Node Kernel or in a Node Connection region, the *nodeKernelsR* and the *nodeConnsR* submodels respectively indicate to *fauLPevalAtHub* how the fault manifests at the uplink hub port. Specifically, a token in the place *stuckLP* means that the fault manifest as stuck-at, whereas a token in *flipLP* informs *fauLPevalAtHub* that the fault manifests as bit-flipping. However, a token in any of these places does not activate *fauLPevalAtHub*. In contrast, what activates the *fauLPevalAtHub* submodel is a token in the place *fauALP* or in *fauBLP*. More specifically, the *nodeKernelsR* and the *fauLPevalAtNode* (which is activated by *nodeConnsR*) submodels set a token in the place *fauALP* or *fauBLP* to indicate that the hub A or the hub B, respectively, receives the stuck-at/bit-flipping errors through one of its uplink ports.

As can be observed in Figure 12.10, the structure of *fauLPevalAtHub* is symmetric in the sense that the actions performed when a token is set in *fauALP* are analogous to those carried out when a token is set in *fauBLP*. The only difference is that in the first case the hub that receives the errors is the hub A, whereas in the second one it is the hub B. Therefore, the rest of this section only describes the actions *fauLPevalAtHub* performs when it detects a token in *fauALP*.

A token in this place enables the instantaneous activity *covALP*, which evaluates if the hub A is able to isolate the faulty uplink. The first case of this activity represents the situation in which the hub A is not able to isolate the fault. The second case models the opposite situation.

When *covALP* selects its first case, the corresponding output gate performs the following actions. On the one hand, it resets the marking of the places *stuckLP* and *flipLP*. In this way, it allows *nodeKernelsR* and *nodeConnsR* to respectively specify, when necessary, how a new faulty Node Kernel and a new faulty Node Connection region manifest at an uplink port. On the other hand, this output gate initiates a set of actions devoted to deciding how to continue with the path of the coverage process, in order to model how the errors propagated by hub A are treated.

The first one of these actions consists in elucidating whether or not the hub B is non-faulty. This is necessary because if the hub B was already faulty when the fault that leaded to the activation of *fauLPevalAtHub* occurred, then the fact that the hub A has become faulty implies that there is no hub available for communicating. The activity *isNisoLPAHB* is the responsible for finding out whether or not the hub B was faulty. It selects its first case when the hub B is non-faulty and its second one in the opposite situation. In order to know which case it has to select, *isNisoLPAHB* uses the marking of the place *fauHubB*. As mentioned in Section 12.4.7, a token in this place indicates that the hub B is faulty because its kernel is faulty or because

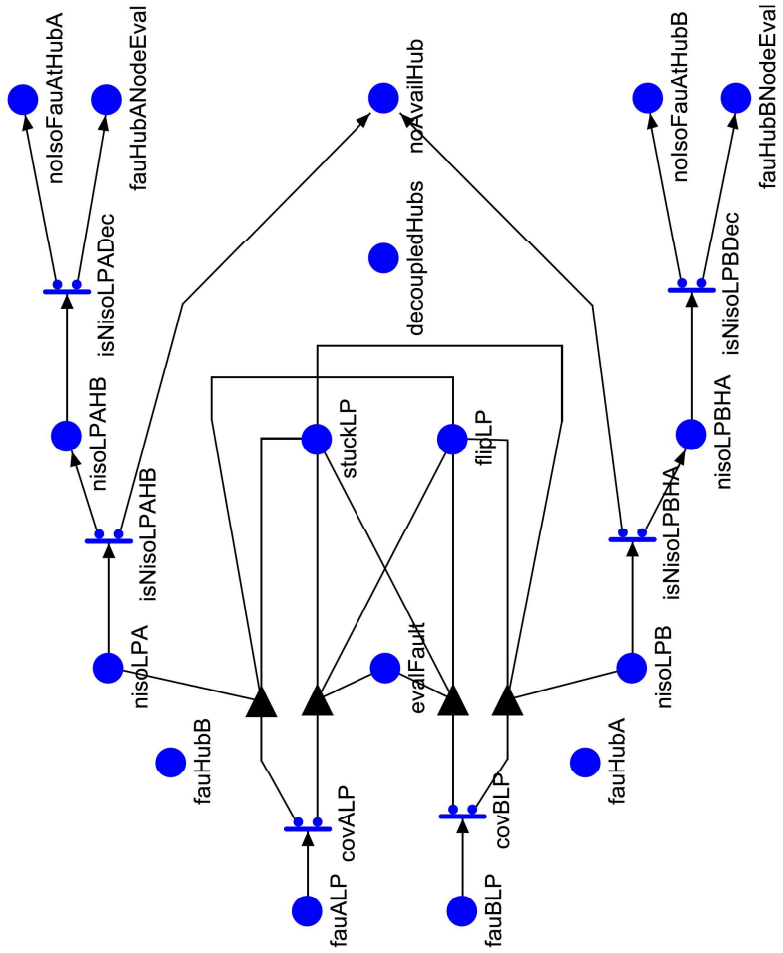

Figure 12.10: fauLPevalAtHub submodel

it was not able to contain a fault in any of its ports. If there is a token in *fauHubB*, *isNisoLPAHB* chooses its second case, otherwise it selects its first one.

If *isNisoLPAHB* selects its second case, it simply sets a token in the place *noAvailHub* in order to indicate that there is no hub available for communicating. As it has been explained before, this token leads the *ReCANcentrateFaiEval* submodel to diagnose that the whole system is faulty. In contrast, if *isNisoLPAHB* selects its first case, it sets a token in *nisoLPAHB* thereby enabling the instantaneous activity *isNisoLPADec*. This activity checks whether or not the hubs were coupled before the fault occurred. Then, it decides which submodel must proceed with the coverage process in order to assess how the errors the fault generates (and which the hub A propagates) are treated. On the one hand, if the hubs are coupled, it sets a token in *noIsoFauAtHubA*. This activates *fauHubEvalAtHub*, which assesses whether or not the hub B isolates the errors propagated by the hub A. On the other hand, if the hubs are decoupled, then it sets a token in *fauHubANodeEval*. This token activates *fauHubEvalAtNodes*, which is devoted to evaluating if each node that is connected to the hub A is able to tolerate this hub failure and, thus, if it is able to communicate exclusively through the hub B.

Notice that *isNisoLPADec* plays the same role as the activity *isNisoLPsADec* of the *fauLPsEvalAtHubs* submodel (see Section 12.4.9 and Figure 12.9). This is because *fauLPsEvalAtHubs* also models a situation in which the hub A does not contain errors it receives at one of its uplink ports and, therefore, *fauLPsEvalAtHubs* has also to decide, by means of *isNisoLPsADec*, which submodel (*fauDecHubANodeEval* or *fauDecHubEvalAtNodes*) must proceed with the coverage process. Moreover, the proportions of the cases of these activities are calculated in the same way in both submodels. See Section 12.4.9 for a detailed explanation of how these proportions are obtained.

It is also noteworthy that, conversely to what *fauLPevalAtHub* does, *fauLPsEvalAtHubs* enables *isNisoLPsADec* without checking whether or not the hub B was already faulty. Notice again that *fauLPsEvalAtHubs* is activated by *nodeKernelsR* when the Node Kernel of an *okAB* node suffers from a non-ofm fault, i.e. when a non-ofm fault happens in the Node Kernel of a node that is able to communicate through hub A and hub B. Therefore, *fauLPsEvalAtHubs* can be sure about the fact that both hubs were not faulty before the fault occurred.

Finally, as concerns the second case of the activity *covALP*, i.e. the case of *covALP* that corresponds to the situation in which the hub A successfully isolates the faulty uplink, notice that it is also connected to a dedicated output gate. On the one hand, this output gate resets the marking of the places *stuckLP* and *flipLP* for the same reasons mentioned above. On the other hand, it resets the marking of the

place *evalFault* to indicate to the *ReCANcentrateFaiEval* submodel that the coverage process is finished. This is because if the hub A isolates its uplink, then the errors are not propagated to the rest of the communication subsystem.

The proportions of the two cases of the activity *covALP* are calculated taking into account the way in which the fault manifests at the uplink hub port, as well as the coverage with which the hub can isolate the specific type of fault. The expressions of the first and the seconds cases of *covALP* are (respectively):

$$1.0 - (\mathit{stuckLP} \rightarrow \mathit{Mark()} \cdot 1.0 + \mathit{flipLP} \rightarrow \mathit{Mark()} \cdot \mathit{flipLnkCov})$$

and

$$\mathit{stuckLP} \rightarrow \mathit{Mark()} \cdot 1.0 + \mathit{flipLP} \rightarrow \mathit{Mark()} \cdot \mathit{flipLnkCov}$$

In each one of these expressions, the marking of *stuckLP* and *flipLP* is multiplied by the coverage with which the hub is able to isolate the corresponding type of fault. More specifically, since the hub is always able to isolate a stuck-at uplink port, the marking of *stuckLP* is multiplied by 1.0. In contrast, the hub can only isolate a bit-flipping uplink port with a *flipLnkCov* coverage (see Table 12.1) and, thus, the marking of *flipLP* is multiplied by the this coverage. Notice that when *covALP* fires, the token that indicates the failure mode is located in *stuckLP* or in *flipLP*, but it cannot be placed in both of them. Therefore, each one of the above expressions is simplified in such a way that it only retains the coverage associated with the failure mode the uplink port actually manifests. In particular, if the fault manifests as stuck-at, the proportions of the first and the second cases are 0.0 and 1.0 respectively, so that *covALP* selects the second one. Otherwise, *covALP* respectively chooses these cases with probabilities $1.0 - \mathit{flipLnkCov}$ and *flipLnkCov*.

12.4.11 *fauIPevalAtHubs* submodel

The *fauIPevalAtHubs* coverage submodel is aimed at evaluating how a non-ofm faulty Hub Interconnection region is isolated by the corresponding hub, i.e. by the hub that receives the contribution of the other hub through that interconnection. Moreover, depending on the number of Hub Interconnections regions that remain non-faulty, the *fauIPevalAtHubs* submodel decides if the hubs become decoupled. The location of *fauIPevalAtHubs* within the coverage process is depicted

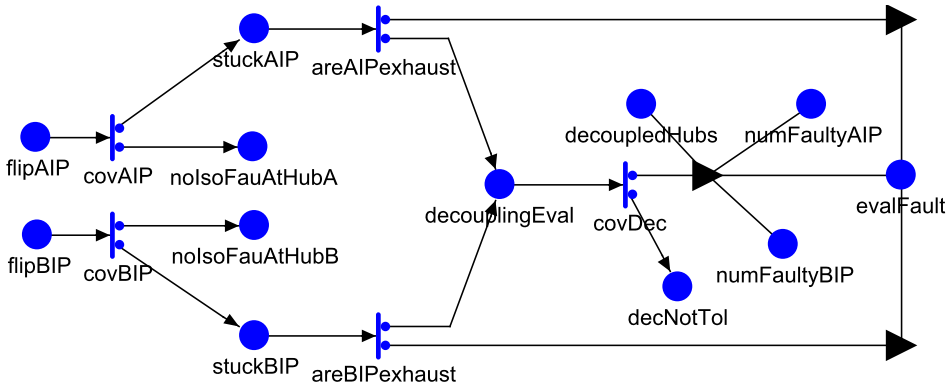

Figure 12.11: fauIPevalAtHubs submodel

in Figure 12.2, where the *hubInConns* submodel, which represents all the Hub Interconnection regions of ReCANcentrate, activates *fauIPevalAtHubs* when a Hub Interconnection region fails in a way that is included in our fault model.

Figure 12.11 shows the structure of the *fauIPevalAtHubs* submodel. As explained in Section 12.4.6, the *hubInConns* submodel activates *fauIPevalAtHubs* by setting a token in one of the following places: *stuckAIP*, *flipAIP*, *stuckBIP* and *flipBIP*. A token in *stuckAIP* and in *flipAIP* indicates that a Hub Interconnection region that carries the contribution of the hub B to the hub A exhibits a stuck-at and a bit-flipping fault respectively. A token in *stuckBIP* and in *flipBIP* also indicates that each one of these types of fault occurs. However, a token in any of these two last places indicates that the fault has occurred in a Hub Interconnection that conveys the contribution of the hub A to the hub B.

The set of actions *fauIPevalAtHubs* performs when it receives a token in *stuckBIP* or in *flipBIP* are analogous to those it carries out when it receives a token in *stuckAIP* or in *flipAIP*. The difference between both sets of actions is that in the first case the hub B is the one that must isolate the faulty Hub Interconnection, whereas in the second one it is the hub A. Thus, for the sake of succinctness, the rest of this section addresses the actions performed by *fauIPevalAtHubs* when a token is set in *stuckAIP* or in *flipAIP*, i.e. when the hub A is the responsible for isolating the fault.

As depicted in Figure 12.11, a token in the place *flipAIP* enables the instantaneous activity *covAIP*. This activity evaluates whether or not the hub A isolates a bit-flipping Hub Interconnection at the corresponding sublink port (inbranch). The first case represents the situation in which the hub A isolates the fault, whereas the second one models the opposite situation. The proportion of the first case is

merely the coverage with which a hub is able to isolate a bit-flipping Hub Interconnection, i.e. a fault occurring at a sublink: *flipSlnkCov* (see Section 12.3.4 and Table 12.1). The proportion of the second case is, thus, *1.0 - flipSlnkCov*.

If the hub A does not isolate the fault, *covAIP* sets a token in *noIsoFauAtHubA*. This compels the *fauHubEvalAtHub* submodel to take over in order to assess if the hub B isolates the hub A. Notice that this is the same place the *fauLPsEvalAtHubs* and *fauLPevalAtHub* submodels use to activate *fauHubEvalAtHub* when the hub A does not isolate a non-ofm fault at one of its uplink ports and the hubs were coupled when the uplink port failed (see Sections 12.4.9 and 12.4.10 and Figure 12.2). As already explained, *fauLPsEvalAtHubs* and *fauLPevalAtHub* do not activate *fauHubEvalAtHub* if the hubs were decoupled when the fault occurred because, in that case, all the Hub Interconnection regions are already isolated at their respective hub ports. However, also notice that, conversely to what those two submodels do, *fauIPevalAtHubs* activates *fauHubEvalAtHub* without corroborating that both hubs are coupled. Notice that activity *covAIP* does not need to check this condition because if the hubs were decoupled, then *fauIPevalAtHubs* would not have been activated by the *hubInConns* submodel.

To better understand why *fauIPevalAtHubs* would not have been activated if the hubs were decoupled, it is necessary to go back to Section 12.4.6. There we explained that the activity *hiAFailure* of *hubInConns* initiates the path of the coverage process that evaluates how a non-ofm fault happening in a Hub Interconnection is treated (and thus activates *fauIPevalAtHubs*), only if the inbranch corresponding to that Hub Interconnection was not already faulty. Notice again that an inbranch is faulty in the following conditions: (1) if its Hub Interconnection fails, (2) if a hub fails, or (3) if the hubs are decoupled. Therefore, the fact that *fauIPevalAtHubs* is active implies that none of these conditions is true and, in particular, that the hubs are not decoupled.

Up to this point, we have explained what happens when *covAIP* decides that the hub A does not isolate the faulty Hub Interconnection region, i.e. when the activity *covAIP* selects its second case. Regarding the first case of this activity, i.e. the one that is chosen when the hub A successfully isolates the faulty Hub Interconnection region, notice that it sets a token in the place *stuckAIP*. This is also the place at which *fauIPevalAtHubs* receives a token when a Hub Interconnection exhibits a stuck-at fault. This is because an isolated hub port is equivalent to a stuck-at-recessive hub port and, therefore, the *fauIPevalAtHubs* submodel must carry out the same actions in both cases.

The instantaneous activity *areAIPexhaust* carries out the first one of these actions. It consists in elucidating whether or not all the inbranches of hub A are

exhausted. This is necessary because the faulty Hub Interconnection region that has provoked the activation of *fauIPevalAtHubs* belonged to a non-faulty inbranch of hub A and, therefore, it is possible that the failure of this interconnection has affected the last surviving inbranch of that hub. If this is the case, then the hubs become decoupled as the hub A will no longer receive the contribution of the hub B. The activity *areAIPexhaust* selects its first case when there is at least one inbranch that allows the hub A to receive the contribution of the hub B. Otherwise, this activity chooses its second case. The proportions of both cases are respectively calculated as:

(Case 1)

if (*numFaultyAIP* $\rightarrow$ *Mark()* $==$ *numInterlinks*)
 return 0.0;
else
 return 1.0;

(Case 2)

if (*numFaultyAIP* $\rightarrow$ *Mark()* $==$ *numInterlinks*)
 return 1.0;
else
 return 0.0;

To better understand the above expressions notice that, as already said in Section 12.4.6, the total number of inbranches that carry the contribution of the hub B to the hub A (and viceversa) is equal to the number of interlinks. Moreover, we also explained there that the marking of the places *numFaultyAIP* and *numFaultyBIP* respectively represent the number of inbranches of the hub A and the hub B that have failed so far. In particular, we said that when a Hub Interconnection fails, the *hubInConns* submodel increases the marking of one of these places accordingly. Therefore, when *areAIPexhaust* inspects the marking of *numFaultyAIP*, it knows that this marking also reflects the new inbranch failure. In this way, *areAIPexhaust* only needs to compare this marking with the total number of interlinks (which is specified by means of the parameter *numInterlinks*), in order to know if all the inbranches of the hub A have become exhausted. As can be inferred from the above expressions, if the marking of *numFaultyAIP* is equal to the number of interlinks, the proportions of the first and second cases of *areAIPexhaust* are 0.0 and 1.0 respectively. If this marking is lower than *numInterlinks* then, these proportions are 1.0 and 0.0.

When *areAIPexhaust* selects its first case, then a dedicated output gate simply resets the marking of the place *evalFault* to finish the coverage process. This is because the hub A has isolated the faulty Hub Interconnection (so that errors do not propagate) and there are enough non-faulty inbranches that allow both hubs to receive the contribution of each other and, thus, to remain coupled.

Conversely, the second case of *areAIPexhaust* is chosen when both hubs become decoupled and, therefore, it is necessary to evaluate whether or not the nodes are able to continue communicating using both hubs independently. The instantaneous activity *covDec* evaluates this aspect as soon as it receives a token in *decouplingEval*. The first of the two cases of *covDec* corresponds to the situation in which the nodes tolerate the hub decoupling. When *covDec* selects it, the corresponding output gate carries out different actions. First, it sets a token in the place *decoupledHubs* to indicate that the hubs become decoupled. As described in the corresponding sections, different submodels consult the marking of this place to elucidate if the nodes are communicating using both hubs independently, e.g. the *ReCANcentrateFaiEval*, the *hubKernels* or the *fauLPevalAtNode* submodels. Second, the output gate forces the marking of the places *numFaultyAIP* and *numFaultyBIP* to their maximum value, i.e. to *numInterlinks*. This is necessary because, as already explained in Section 12.4.2 and mentioned in Section 12.4.6, the sublink hub ports of both hubs can be considered as faulty when the hubs become decoupled (each hub will perceive the contribution of the other hub as erroneous). Finally, the output gate also resets the marking of the place *evalFault*. Notice that the coverage process can be considered as finished at this point: the hub A has isolated the faulty Hub Interconnection, the hubs have become decoupled and the nodes start communicating using both hubs independently.

Regarding the second case of the activity *covDec*, it models the situation in which the nodes are not able to tolerate the hub decoupling. When *covDec* selects this case, it sets a token in the place *decNotTol*, thereby indicating such a situation to the *ReCANcentrateFaiEval* submodel and, thus, provoking that the whole system is diagnosed as faulty.

Finally, the proportions with which *covDec* selects its first and second cases are *decCov* and $1.0 - \mathit{decCov}$ respectively; where *decCov* is the coverage with which the nodes tolerate the hub decoupling (see Section 12.3.4 and Table 12.1).

12.4.12 *fauHubEvalAtHub* submodel

The *fauHubEvalAtHub* coverage submodel evaluates whether or not a hub is able to isolate the other hub when that other hub exhibits a non-ofm failure. As already

explained in Section 12.3.3, a hub is faulty when its kernel fails, as well as when it is not able to isolate a faulty uplink or a faulty sublink port. In both cases, the faulty hub broadcasts errors through all its downlinks and its outgoing sublinks. Also notice again that if the hubs were decoupled before one of the hubs fails, then it is not necessary to evaluate if the hub that remains non-faulty isolates the faulty one and, thus, *fauHubEvalAtHub* is not activated.

Figure 12.2 shows the location of *fauHubEvalAtHub* within the coverage process. If the non-faulty hub does not isolate the faulty one by disabling the corresponding inbranches, then *fauHubEvalAtHub* finishes the coverage process and *ReCANcentrateFaiEval* diagnoses that the whole system is faulty. Otherwise, since the faulty hub also broadcasts errors through its downlinks, *fauHubEvalAtHub* compels *fauHubEvalAtNodes* to further evaluate whether or not each node successfully isolates it and continues communicating using the non-faulty hub. This issue will be explained in detail in Section 12.4.13.

Figure 12.12 depicts the structure of the *fauHubEvalAtHub* submodel. As can be observed, it is symmetric because the way in which *fauHubEvalAtHub* models how the hub B isolates the hub A is equal to the way in which it models how the hub A isolates the hub B. Thus, we will only explain the first one of these cases.

As we have seen in previous sections, the different situations that lead a hub to become faulty are modelled by means of specific submodels, which compel *fauHubEvalAtHub* to proceed with the coverage process. These submodels are (Figure 12.2): *fauLPsEvalAtHubs*, *fauLPevalAtHub*, *fauIPevalAtHubs* and *hubKernels*. Regarding the *fauLPsEvalAtHubs*, *fauLPevalAtHub* and *fauIPevalAtHubs* submodels, notice again that they activate *fauHubEvalAtHub* by setting a token in *noIsoFauAtHubA*. More specifically, *fauLPsEvalAtHubs* and *fauLPevalAtHub* set this token when the hub A does not isolate a non-ofm fault affecting the port corresponding to an uplink, whereas *fauIPevalAtHubs* sets it when the hub A cannot isolate a non-ofm fault happening in a sublink port. See Sections 12.4.9, 12.4.10 and 12.4.11 for further details.

Note that a non-isolated port that suffers from a non-ofm failure can only be a bit-flipping port, since a hub isolates stuck-at faults with a perfect coverage. This means that *fauLPsEvalAtHubs*, *fauLPevalAtHub* and *fauIPevalAtHubs* only activate *fauHubEvalAtHub* (and thus set a token in *noIsoFauAtHubA*) when the hub A propagates bit-flipping errors. In this sense, as depicted in Figure 12.12, a token in *noIsoFauAtHubA* enables an instantaneous activity that evaluates if the hub B contains the bit-flipping errors that the hub A propagates through its outgoing sublinks. This activity is called *covFauHAProp*. Its first and second cases respectively model the situation in which the hub B isolates the hub A and the situation in

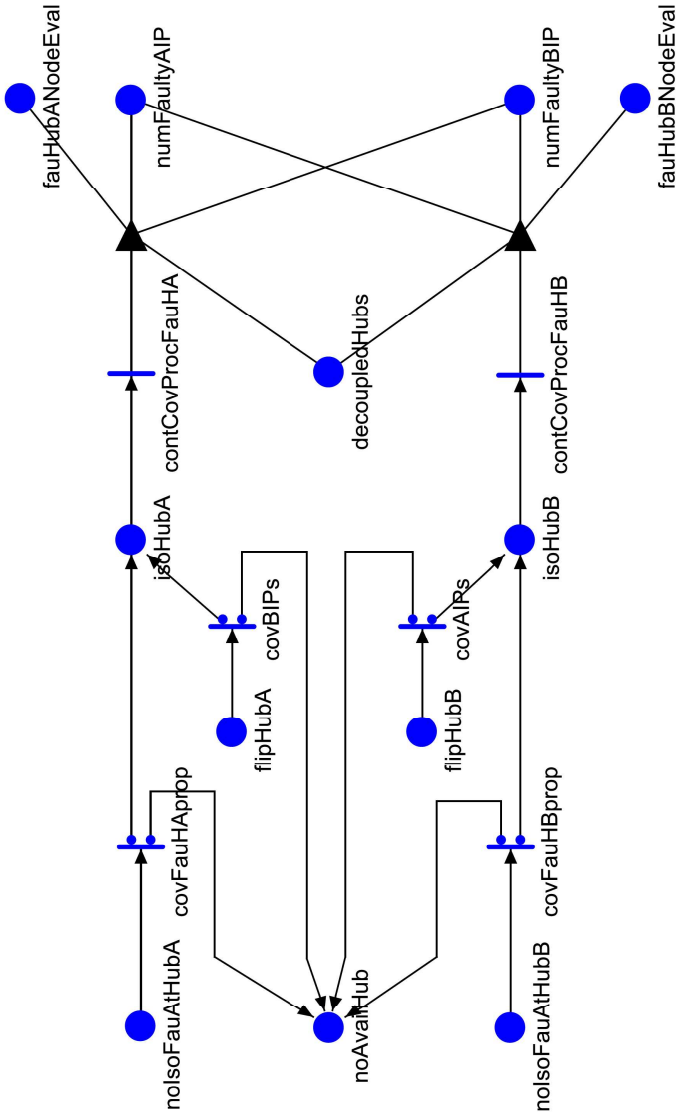

Figure 12.12: fauHubEvalAtHub submodel

which it does not. As described in Section 12.3.4, the probability with which a hub successfully isolates a bit-flipping stream propagated by the other hub is called *flipSlnkPropCov* (see Table 12.1). Thus, the proportions of the first and second cases of *covFauHAProp* are *flipSlnkPropCov* and $1.0 - $ *flipSlnkPropCov* respectively.

If the activity *covFauHAprop* selects its second case, it sets a token in the place *noAvailHub*. As explained in Section 12.4.3, a token in *noAvailHub* indicates to the *ReCANcentrateFaiEval* submodel that there is not any hub available for communicating. This leads *ReCANcentrateFaiEval* to diagnose the system as faulty. Otherwise, if *covFauHAprop* selects its first case, it sets a token in the place *isoHubA* to indicate the the hub B successfully isolates the hub A.

At this point and before explaining the actions *fauHubEvalAtHub* carries out when a token is set in *isoHubA*, it is important to recall that this place is one of the two places that *hubKernels* uses for activating *fauHubEvalAtHub* when the kernel of the hub A fails. More specifically, as explained in Section 12.4.7, *hubKernels* activates *fauHubEvalAtHub* by setting a token in *isoHubA* and *flipHubA*, when the Hub Kernel region of the hub A fails exhibiting a stuck-at and a bit-flipping failure respectively. The reason why *hubKernels* submodel directly uses *isoHubA* to indicate that the kernel of hub A is stuck-at-recessive is because the hub B is always able to isolate that type of fault.

In contrast, when *hubKernels* sets a token in *flipHubA*, it is necessary to evaluate whether or not the hub B isolates the bit-flipping bit stream the kernel of the hub A sends to it through the sublinks. The instantaneous activity *covBIPs* performs such an evaluation. Its first case represents the situation in which the hub B successfully isolates the hub A and, thus, it sets a token in the place *isoHubA*. The second case of this activity models the opposite situation and sets a token in the place *noAvailHub*.

The coverage with which the hub B isolates the bit-flipping failure of the kernel of the hub A is *flipSlnkCov*, i.e. the probability with which a hub isolates a bit-flipping Hub Core or sublink failure (see Section 12.3.4 and Table 12.1). Therefore, the proportions of the first and second cases of the activity *covBIPs* are *flipSlnkCov* and *1.0 - flipSlnkCov* respectively. Notice that the default value of this coverage is greater than the default value of *flipSlnkPropCov*, which is the probability that the activity *covFauHAprop* takes into account for deciding if the hub B successfully isolates a bit-flipping stream the hub A propagates. This is because *flipSlnkPropCov* corresponds to a situation in which the hub B needs to isolate a bit-flipping stream that is very similar to the stream the hub A was not previously able to isolate. See Section 12.3.4 for a further explanation on this issue.

Finally, let us explain what *fauHubEvalAtHub* does once a token is set in the place *isoHubA*, i.e. once it decides that the hub B successfully isolates the hub

A. When this happens, the instantaneous activity *contCovProcFauHA* fires and a dedicated output gate carries out the following actions. First, it forces the marking of the place *decoupledHubs* to 0, i.e. it resets *decoupledHubs* even if it has no token. As explained before, *decoupledHubs* is a place shared by several submodels and its marking is 1 if the hubs are decoupled and 0 otherwise. Notice again that we understand that the hubs are decoupled when both of them are not faulty, but they cannot exchange their contributions with each other, i.e. when each hub represents an independent communication domain (see Section 12.4.2). Therefore, in order to be consistent with this definition, the output gate makes sure that the marking of *decoupledHubs* is 0 when the hub A fails.

Second, the output gate forces the marking of the places *numFaultyAIP* and *numFaultyBIP* to their maximum value, i.e. to *numInterlinks*. As already explained in Section 12.4.11, the marking of these places respectively indicate the number of inbranches of hub A and hub B that have failed so far. Since one of the reasons that lead an inbranch to fail is the failure of any of the hubs (see Section 12.4.2), the output gate modifies *numFaultyAIP* and *numFaultyBIP* as indicated, in order to reflect that all the inbranches are faulty.

The last action that the output gate of the activity *contCovProcFauHA* performs is to set a token in the place *fauHubANodeEval*. This token compels the *fauHubEvalAtNodes* submodel to take over. As explained at the beginning of this section (and mentioned in other sections, e.g. in 12.4.4) when a hub fails and the other one successfully isolates it, then it is necessary that *fauHubEvalAtNodes* evaluates whether or not each node is able to continue communicating using the non-faulty hub.

12.4.13 *fauHubEvalAtNodes* submodel

The *fauHubEvalAtNodes* submodel is activated in two different situations. On the one hand, the *fauHubEvalAtHub* submodel activates *fauHubEvalAtNodes* when the hubs are coupled and one of them exhibits a non-ofm failure that the other hub successfully isolates (as just explained in Section 12.4.12). On the other hand, the *hubKernels*, *hubLPsEvalAtHubs* and *hubLPevalAtHub* submodels directly activate *fauHubEvalAtNodes* when the hubs are decoupled and one of them manifests a non-ofm failure as a consequence of a fault in its kernel, or because it was not able to isolate a non-ofm fault in any of its uplink ports. All these connections among the mentioned submodels are depicted in Figure 12.2.

When activated, the *fauHubEvalAtNodes* submodel evaluates if each node that is connected to the faulty hub tolerates the failure and, thus, if each node is able

to continue communicating using the non-faulty one. Notice that *fauHubEvalAtNodes* is the last coverage submodel that is involved in several paths of the coverage process. In this sense, as depicted in Figure 12.2, *fauHubEvalAtNodes* does not activate any coverage submodel. Instead it always compels *ReCANcentrateFaiEval* to proceed and to decide if the system is faulty.

The *fauHubEvalAtNodes* submodel needs to differentiate between the nodes that have any chance of tolerating the failure of one of the hubs and those that have not. In this sense, only the nodes that have a non-faulty Node Connection to the hub that remains non-faulty, and that were using that connection to communicate through it, have the possibility of tolerating the failure of the other hub. In other words, if the hub A fails, then the nodes that can potentially tolerate the hub failure are the okAB nodes and the okB nodes. Analogously, if the hub that fails is the hub B, then the nodes that must be considered are the okAB and the okA nodes.

As concerns the probability with which a node tolerates a hub failure, notice that this depends on the type of node, as well as on whether or not the hubs are coupled. The coverage with which an okAB node tolerates a fault that prevents it from communicating through one of the hubs is *connCov*, if the hubs are coupled, and *decConnCov* otherwise (see Section 12.3.4 and Table 12.1). Notice that such a fault can affect either the Node Connection or the hub itself. This is the reason why *connCov* and *decConnCov* are also the coverages that *fauLPevalAtNode* uses to decide if an okAB node can continue communicating when one of its Node Connections fails (see Section 12.4.8). Regarding the okA and the okB nodes, they can respectively tolerate the failure of the hub B and the hub A with a perfect coverage. This is because such a kind of nodes have already discarded the hub that fails for communicating, so that they were exclusively using the hub that remains non-faulty.

Figure 12.13 shows the structure of the *fauHubEvalAtNodes* submodel. As happens with other submodels, its structure is symmetric. The way in which *fauHubEvalAtNodes* evaluates how the nodes tolerate the failure of the hub B is analogous to the way in which it models how they tolerate the failure of the hub A. Thus, we only describe the case that corresponds to the failure of the hub A.

As already explained, *fauHubEvalAtNodes* becomes active when it receives a token in the place *fauHubANodeEval* (or *fauHubBNodeEval*). This token enables the instantaneous activity *okStopAExcl*, which is connected to a dedicated output gate. This gate performs the following actions. First, it updates the marking of the place *numFaultyBranches*, in order to reflect that all the surviving branches of the hub A become faulty. This is necessary because a faulty hub does not only issue errors through all its outgoing sublinks, but also through all its downlinks.

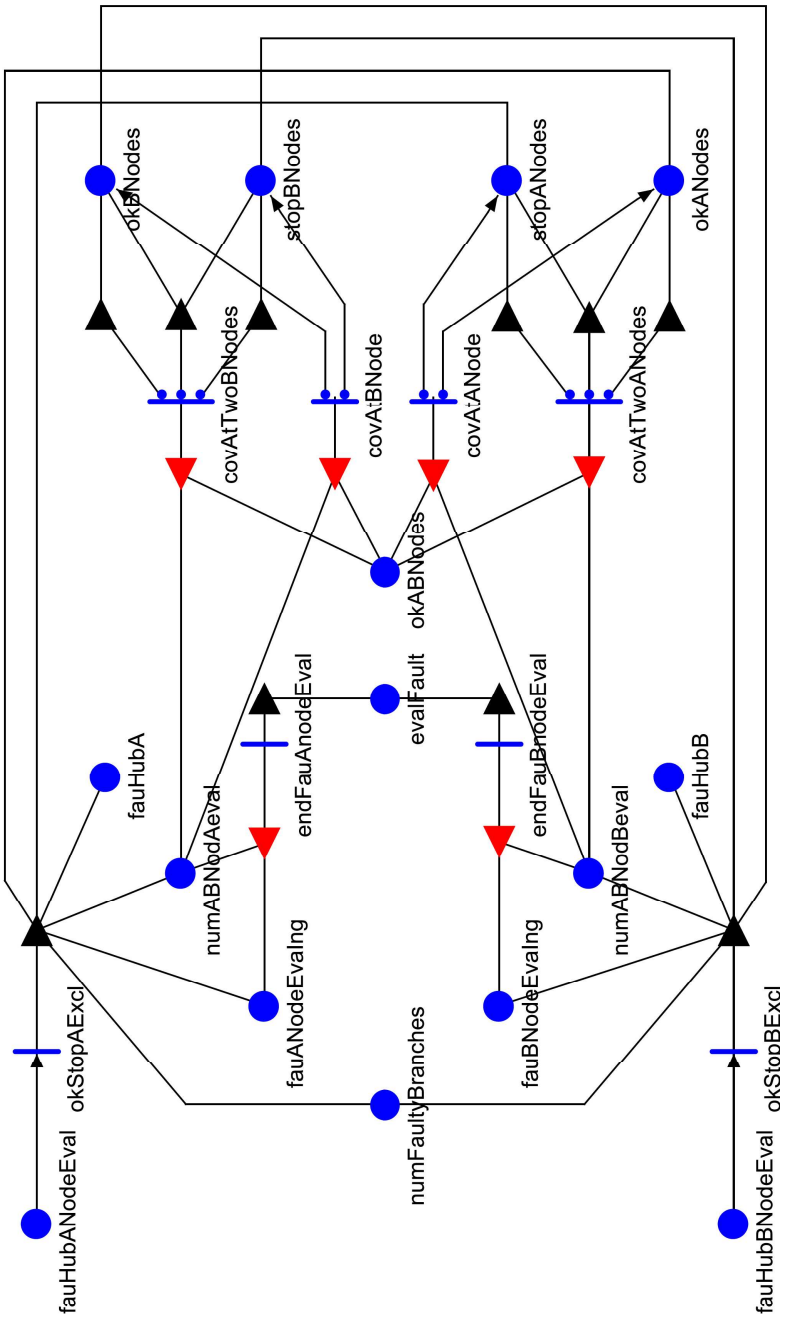

Figure 12.13: fauHubEvalAtNodes submodel

Specifically, the number of branches of the hub A that were not faulty are those that belong to an okAB, an okA or a stopA node; since only each one of these nodes had a non-faulty Node Connection to the hub A. Therefore, the output gate updates the marking of *numFaultyBranches* as follows:

$$numFaultyBranches \rightarrow Mark() = numFaultyBranches \rightarrow Mark() + okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark();$$

Second, the output gate resets the marking of the places *stopANodes* and *okANodes* in order to reflect that each okA and stopA node loses its Node Connection that is attached to the hub A. Notice that these nodes become *noConn* nodes and, as explained in Section 12.4.2, there is no need to explicitly record their quantity.

Third, the output gate forces the marking of *fauHubA* to 1 to indicate that the hub A is faulty. As explained before, submodels such as for example *hubKernels* and *fauLPevalAtHub* use this place to elucidate if the hub A is faulty and, then, to act accordingly.

Finally, the gate starts a process that evaluates whether or not the nodes that can potentially tolerate the failure of the hub A successfully do it. As just said above, these nodes are the okAB and the okB nodes. In fact, since okB nodes tolerate the failure of the hub A with a perfect coverage, it is only necessary to evaluate what okAB nodes do. In order to initiate the referred process, the output gate proceeds as follows. On the one hand, it forces the marking of the place *numABNodAeval* to be equal to the marking of the place *okABNodes*, i.e. to the number of okAB nodes. As will be explained later, the marking of *numABNodAeval* is used as a counter that indicates, during the process, what is the number of okAB nodes for which it is still necessary to decide if the failure of the hub A is tolerated. On the other hand, the output gate sets a token in the place *fauANodeEvaIng*. This indicates to the activity *endFauAnodeEval* that the *fauHubEvalAtNodes* submodel is carrying out the process that evaluates each okAB node. As will be explained at the end of this section, the activity *endFauAnodeEval* detects when this process finishes and, then, it resets the marking of *evalFault* to indicate to the *ReCANcentrateFaiEval* submodel that the whole coverage process is finished too.

As concerns the actions of the process itself, notice that the activities that actually evaluate if each okAB node tolerates the failure of the hub A are *covAtTwoBNodes* and *covAtBNode*. In principle, as long as the marking of *numABNodAeval* is ≥ 0, one of these two activities fires, does its task, and consumes part of the tokens of this place and of the place *okABNodes*. More specifically, the activity *covAtTwoBNodes* consecutively fires as long as the marking of *numABNodAeval*

is ≥ 2; whereas *covAtBNode* fires when there is only one token left in this place. The activity *fauHubEvalAtNodes* evaluates whether or not each one of two okAB nodes tolerates the failure, i.e. it evaluates two okAB nodes at a time. For each one of these two okAB nodes, *covAtTwoBNodes* increases the marking of *okBNodes* in one unit if the node tolerates the fault, or it adds a token to *stopBNodes* otherwise. Additionally, *covAtTwoBNodes* erases two tokens from both *numABNodAeval* and *okABNodes*. Likewise, when *numABNodAeval* has only one token, the activity *covAtBNode* evaluates whether or not the single okAB node this token represents tolerates the failure of the hub A. Then, it adds one token to *okBNodes* or to *stopBNodes* accordingly, and erases one token from *numABNodAeval* and *okABNodes*. Notice that this process could be done without the activity *covAtTwoBNodes*, since we could have used the activity *covAtBNode* to evaluate all the okAB nodes, one at a time. However, to evaluate two nodes at the same time by means of the activity *covAtTwoBNodes* reduces the state space of the underlying Markov process. Moreover, in fact, the *fauHubEvalAtNodes* includes an activity called *covAtThreeBNodes* in order to evaluate three okAB nodes at a time. However, this activity and its corresponding input and output gates are not depicted in Figure 12.13 for the sake of simplicity.

In order to control which one of these three activities fires, each one of them is enabled by means of a dedicated input gate. However, next we only show the expressions of the input gates of *covAtTwoBNodes* and *covAtBNode*. In this sense, notice that the expression of the input gate of *covAtTwoBNodes* we specify here is not exactly the real one, since we had to slightly modify it as the activity *covAtThreeBNodes* has been eliminated from the Figure 12.13. Anyway, the expressions herein specified still retain the underlying strategy we followed to enabled/disable the three mentioned activities.

Input gate of *covAtTwoBNodes*:

$$numABNodAeval \rightarrow Mark() \geq 2 \text{ and}$$
$$(okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + okBNodes \rightarrow Mark()) \geq$$
$$numNodes - kSevere$$

Input gate of *covAtBNode*:

$$numABNodAeval \rightarrow Mark() == 1 \text{ and}$$
$$(okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + okBNodes \rightarrow Mark()) \geq$$
$$numNodes - kSevere$$

The first term of each one of the above expressions allows to select the appropriate activity depending on the number of okAB nodes that need to be evaluated. If such a number, which is indicated by the marking of *numABNodAeval*, is ≥ 2, then only the activity *covAtTwoBNodes* is enabled. If this number is 1, then it is not *covAtTwoBNodes*, but *covAtBNode* the activity that fires. Finally, if the marking of *numABNodAeval* is 0, both activities are disabled.

Additionally, the second term, which coincides in both expressions, is used to disable both activities when the number of nodes that can communicate among them is not enough to guarantee that the system can deliver its services. Notice that as the evaluation of the okAB nodes progresses, each okAB node can become a stopB node and thus disconnected. As a consequence, it is possible that the number of communicating nodes decreases below the level that is acceptable to the system, before all the okAB nodes have been evaluated. If this is the case, to continue with the process would only generate useless states, since the overall system will be eventually diagnosed as faulty. Conversely, to disable the activities *covAtTwoBNodes* and *covAtBNode* in advance allows reducing the state space of the underlying Markov process. Specifically, in order to detect that the number of communicating nodes is not sufficient, the second term of each input gate uses the same strategy as the *ReCANcentrateFaiEval* submodel does (see Section 12.4.3). Basically, this is accomplished by comparing the number of nodes that can communicate among them with *numNodes* $-$ *kSevere*; where *numNodes* is the parameter that specifies the number of nodes of the system and *kSevere* is the value of k of the concept of k-severe failure (see Tables 10.4 and 12.1).

It is also noteworthy that the input gates of *covAtTwoBNodes* and *covAtBNode* do not take into account the value of *sysFauTolCov* to detect that the overall system fails in advance. To omit this value is not incorrect, because *ReCANcentrateFaiEval* will take it into account when *fauHubEvalAtNodes* finishes. However, it would be possible to further simplify the underlaying Markov process by taking this coverage into account in the mentioned input gates. For instance, the input gates could disable *covAtTwoBNodes* and *covAtBNode* when the system does not accept or tolerate that a new okAB node has become a stopB node, i.e. when the system does not accept or tolerate a new disconnected node, even though the number of nodes that have become disconnected so far can be theoretically accepted or tolerated.

As concerns the cases of the activities *covAtTwoBNodes* and *covAtBNode*, their roles and proportions are the following ones. The first case of *covAtTwoBNodes* models the situation in which the two okABNodes it is evaluating tolerate the failure of the hub A. Thus, its proportion is calculated as:

(Case 1 of *covAtTwoBNodes*)

```
if (decoupledHubs → Mark() == 0)
  return connCov · connCov;
else
  return decConnCov · decConnCov;
```

where, as indicated at the beginning of this section, *connCov* and *decConnCov* are the coverages with which an okAB node tolerates a fault that prevents it from communicating through a given hub when the hubs are coupled and decoupled respectively. When this case is selected, its dedicated output gate increases the marking of *okBNodes* in 2 units.

The second case of *covAtTwoBNodes* represents the situation in which only one of the two okAB nodes that *covAtTwoBNodes* is assessing tolerates the fault. As a consequence, its output gate increases the marking of both *okBNodes* and *stopBNodes* in 1 unit. The proportion of this case is obtained as:

(Case 2 of *covAtTwoBNodes*)

```
if (decoupledHubs → Mark() == 0)
  return 2 · connCov · (1.0 − connCov);
else
  return 2 · decConnCov · (1.0 − decConnCov);
```

The third case *covAtTwoBNodes* is chosen to model that the two okAB nodes that *covAtTwoBNodes* is evaluating do not tolerate the failure of the hub A. Its output gate adds 2 tokens to the place *stopBNodes* and its proportion is:

(Case 3 of *covAtTwoBNodes*)

```
if (decoupledHubs → Mark() == 0)
  return (1.0 − connCov) · (1.0 − connCov);
else
  return (1.0 − decConnCov) · (1.0 − decConnCov);
```

Regarding the activity *covAtBNode*, it has two cases. The first one models the situation in which the okAB node tolerates the failure of the hub A, whereas the

other one represents the opposite situation. The proportions of these two cases are *connCov* and $1.0-$*connCov* respectively, when the hubs are coupled. If the hubs are decoupled, these proportions are expressed as *decConnCov* and $1.0-$*decConnCov* respectively.

To conclude this section, let us explain how *fauHubEvalAtNodes* indicates to the *ReCANcentrateFaiEval* submodel that the path of the coverage process in which *fauHubEvalAtNodes* is involved finishes. As mentioned above, the activity *endFauAnodeEval* carries out this notification by setting a token in the place *evalFault*. More specifically, as can be seen in Figure 12.13, there is an input gate that decides when the activity *endFauAnodeEval* has to fire. This occurs when the activities *covAtTwoBNodes* and *covAtBNode* (and *covAtThreeBNodes*) stop evaluating whether or not each okAB node tolerates the failure of the hub A. Therefore, the input gate enables *endFauAnodeEval* in two cases: when there is no okAB node left to be evaluated, or when the number of communicating nodes is too low to guarantee that the system can deliver its services. The expression of the input gate is:

$$
\begin{aligned}
&\textit{fauANodeEvaIng} \rightarrow \textit{Mark()} == 1 \textit{ and} \\
&[\textit{numABNodAeval} \rightarrow \textit{Mark()} == 0 \textit{ or} \\
&(\textit{okABNodes} \rightarrow \textit{Mark()} + \textit{okANodes} \rightarrow \textit{Mark()} + \textit{okBNodes} \rightarrow \textit{Mark()}) < \\
&\textit{numNodes} - \textit{kSevere}]
\end{aligned}
$$

12.4.14 *ofmFauEval* submodel

The role of the *ofmFauEval* submodel was introduced in some of the previous sections: it is the responsible for evaluating how the errors generated by an ofm fault propagate through the system. As depicted in the Figure 12.2, this submodel can be activated by the submodels: *nodeKernelsR*, *nodeConnsR* and *hubKernels* (see sections 12.4.4, 12.4.5 and 12.4.7). When *ofmFauEval* finishes, it merely compels the *ReCANcentrateFaiEval* submodel to assess if the whole system is faulty.

Figure 12.14 shows the structure of the *ofmFauEval* submodel, which becomes active when it observes a token in *ofmNkLP*, *ofmNcLP*, *ofmNcFB*, *ofmHA* or *ofmHB*. More specifically, the *nodeKernelsR* submodel sets a token in the place *ofmNkLP* when the Node Kernel of a node that can use only one of its Node Connections for communicating, i.e. an okA, okB, stopA or a stopB node, suffers from a fault that manifests as ofm at the hub port corresponding to that connection. Notice again that an ofm Node Kernel placed in an okAB node will send ofm errors to both hubs, thereby provoking the failure of the overall system. Thus, as explained in Section 12.4.4, the *nodeKernelsR* submodel does not activate *outFauMod* in this

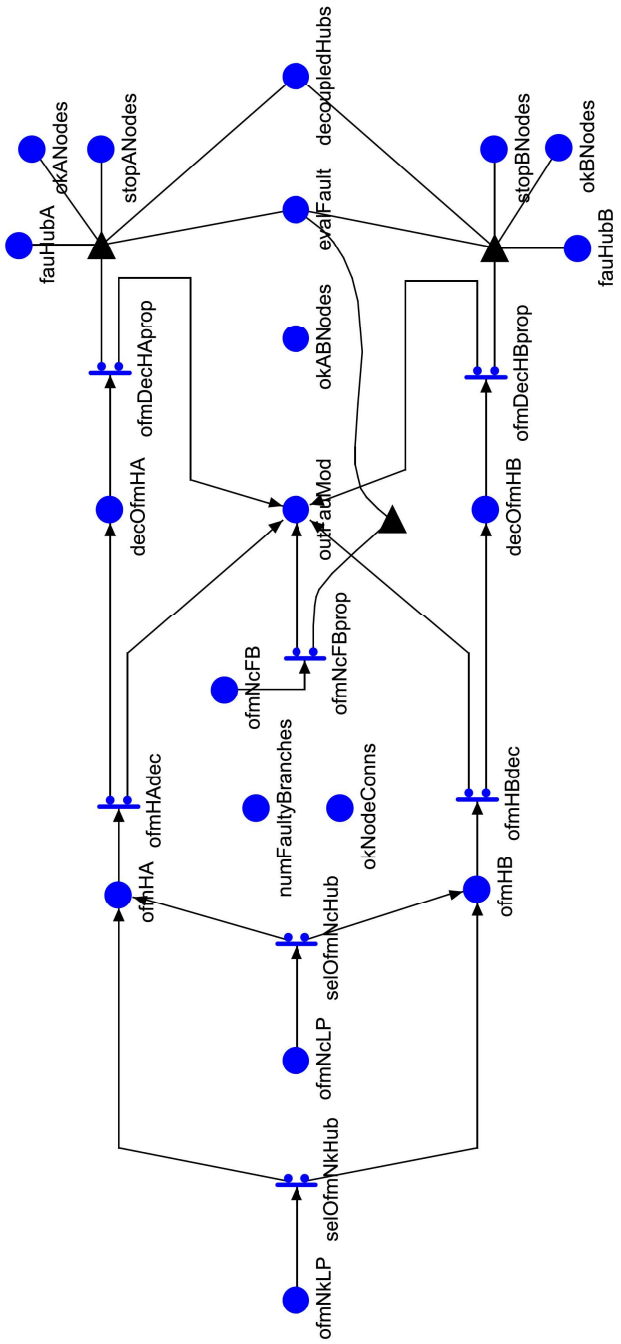

Figure 12.14: ofmFauEval submodel

last situation, but directly compels *ReCANcentrateFaiEval* to diagnose the whole system as faulty.

Regarding the place *ofmNcLP*, notice again that the *nodeConnsR* submodel sets a token in it when a Node Connection that is placed in a non-faulty branch exhibits an ofm failure. Similarly, the *nodeConnsR* submodel sets a token in the place *ofmNcFB* when a Node Connection that is placed at an already faulty (and isolated) branch suffers from an ofm failure.

Finally, the *hubKernels* submodel sets a token in the place *ofmHA* or in *ofmHB* when both hubs are not faulty and the kernel of one of them suffers from an ofm fault. More specifically, *hubKernels* sets a token in *ofmHA* when the kernel that fails belongs to the hub A, whereas it sets a token in *ofmHB* otherwise.

A token in *ofmNkLP* activates the instantaneous activity *selOfmNkHub*, which is devoted to selecting the hub to which the Node Kernel that suffers from the ofm failure is (exclusively) connected to. This is necessary since the *nodeKernelsR* submodel does not specify which is this hub. The activity *selOfmNkHub* calculates the proportion of the case that corresponds to a specific hub by dividing the number of Node Kernels that had only one available Node Connection attached to that hub before the ofm fault occurred, by the total number of Node Kernels that had only one available Node Connection attached to any hub, no matter which, before the ofm occurred. For instance, the proportion of the first case, which models the ofm failure of a Node Kernel that is exclusively connected to the hub A, is calculated as:

$$
\begin{aligned}
&(okANodes \rightarrow Mark() + stopANodes \rightarrow Mark()) \ / \\
&(okANodes \rightarrow Mark() + stopANodes \rightarrow Mark() + \\
&okBNodes \rightarrow Mark() + stopBNodes \rightarrow Mark())
\end{aligned}
$$

Notice that the marking of each place that appears in the above expression correctly reflects what was the quantity of nodes of a given type, before the Node Kernel suffered from the ofm fault. This is because the *nodeKernelsR* submodel does not modify the marking of these places when a Node Kernel fails.

Regarding the actions the activity *selOfmNkHub* carries out, note that it sets a token in *ofmHA* and in *ofmHB* when it selects its first and second cases respectively. These two places are also the ones that the *hubKernels* submodel uses to indicate to *ofmFauEval* that the hub A and the hub B exhibit an ofm failure respectively. This coincidence is due to the fact that a hub propagates ofm errors throughout all its outgoing ports, i.e. it exhibits an ofm failure itself, when it receives ofm errors through an incoming port that was not still isolated. For instance, the first case

of *selOfmNkHub* models the situation in which the faulty Node Kernel sends ofm errors to the Hub A, thereby provoking the ofm failure of this hub.

Moreover, for this same reason, note that the activity *selOfmNcHub* also transfers a token to *ofmHA* or *ofmHB* when it selects its first and second cases respectively. The first one of these cases models the situation in which an ofm Node Connection that is placed at a non-faulty branch is connected to the hub A, whereas the second one represents the situation in which that ofm Node Connection is connected to the hub B. The proportions of the cases of *selOfmNcHub* are calculated following a strategy similar to the one followed by *selOfmNkHub*. Specifically, *selOfmNcHub* obtains the proportions of its cases by dividing the number of non-faulty Node Connections that were correctly connected to a specific hub before the ofm fault occurred, by the total number of Node Connections that were non-faulty before this fault took place. For instance, the proportion of the first case is:

$$(okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark()) \; / \; (okABNodes \rightarrow Mark() + okANodes \rightarrow Mark() + stopANodes \rightarrow Mark() + okBNodes \rightarrow Mark() + stopBNodes \rightarrow Mark())$$

Conversely to the above-mentioned activities, *ofmNcFBprop* sets a token neither in *ofmHA* nor in *ofmHB*. As already pointed out before, *nodeConnsR* writes a token in *ofmNcFB* (thereby activating *ofmNcFBprop*) when an ofm Node Connection is placed at an already faulty branch. Thus, since the corresponding hub will not be directly affected by the errors this fault generates, it is not necessary to elucidate if this Node Connection is attached to the hub A or the hub B. What *ofmFauEval* does when receiving a token in *ofmNcFBprop* will be addressed later on in this section. At this point, however, let us firstly explain what are the actions that *ofmFauEval* carries out when a token is set in the place *ofmHA* or *ofmHB*, i.e. the actions that are performed once the *outFauMod* submodel knows which is the hub that directly receives the ofm errors generated by a Node Kernel, a Node Connection or a Hub Kernel.

These actions are devoted to evaluating whether or not these ofm errors propagate throughout all the system. Note from Figure 12.14 that the way in which this evaluation is carried out is independent of whether the hub that is directly affected by the ofm errors is the hub A or the hub B. Thus, we will only focus on the first case, i.e. when a token is written in *ofmHA*. A token in this place enables the instantaneous activity *ofmHAdec*, which elucidates if the hubs are coupled or not. For this purpose, it simply uses the marking of the place *decoupledHubs*. If the hubs are coupled, then *ofmHAdec* sets a token in the place *outFauMod*, thereby

compelling *ReCANcentrateFaiEval* to diagnose the whole system as faulty. This is because if the hubs are coupled, then the ofm errors will propagate not only to the nodes that are connected to the hub A, but also to the hub B and, then, to the nodes that are connected to that hub. In contrast, if the hubs are decoupled, the ofm errors are only received by the nodes that are connected to the hub A. In this later case, it is necessary to evaluate an additional aspects to decide if the entire system fails.

The activity *ofmDecHAprop* checks this aspect, which consists in elucidating whether or not there is a node connected to the hub A that can propagate the ofm errors through the hub B. Note that a node that is communicating through the hub A will receive ofm errors from that hub. Thus, its Node Kernel can also fail in an ofm manner, since these errors consist in data that are incorrect from the semantic point of view. When this actually happens, the Node Kernel of the node can send ofm errors to the hub B if it has a non-faulty Node Connection attached to that hub. In other words, a node that receives ofm errors from the hub A can pollute the hub B with this type of errors if it is an okAB node.

As a consequence, what *ofmDecHAprop* checks is if there is at least on okAB node. For this purpose, it uses the marking of the place *okABNodes*. If there is one or more tokens in this place, it chooses its second case and sets a token in *outFauMod*. Otherwise, *ofmDecHAprop* selects its first one and compels a dedicated output gate to carry out the following actions. Firstly, it sets a token in the place *fauHubA* in order to indicate that the hub A is faulty. As explained before, submodels such as for example *hubKernels* and *fauLPevalAtHub* use this place to elucidate if the hub A is faulty and, then, to act accordingly. Secondly, it forces the marking of *decoupledHubs* to 0. As already explained in Section 12.4.12, this is necessary to be consistent with the meaning of this place, i.e. with what we understood as a hub decoupling. Thirdly, the output gate resets the marking of the places *okANodes* and *stopANodes*, since the types of nodes these places represent become nonConn nodes. Notice that the output gate does not reset the marking of *okABNodes*, since this point of the submodel is reached if there is not any okAB node in the system (otherwise *ofmDecHAprop* would have set a token in *outFauMod*). Finally, the output gate also resets the marking of *evalFault* in order to compel the *ReCANcentrateFaiEval* submodel to take over.

In order to conclude this section, let us explain the actions that *ofmFauEval* performs when it receives a token in the place *ofmNcFB*, i.e. when the *nodeConnsR* submodel indicates that a Node Connection that is placed at an already faulty (and isolated) branch suffers from an ofm failure. Note that when this happens, the node that corresponds to the Node Connection that has failed cannot be an okAB node, but an okA, an okB, a stopA, a stopB, a nonConn, or a nonOk node. This is because an okAB node can communicate through two non-faulty branches and the

Node Connection that has failed was placed at an already faulty branch. Since the faulty Node Connection did not correspond to an okAB node and it was placed at an already faulty branch, one could think that the ofm errors it generates cannot pollute the system. However, note that the fact that the Node Connection was placed at an already faulty branch does not mean that its corresponding Node Kernel has rule out that Node Connection for communicating. For instance, imagine a branch that is already faulty because the hub it is attached to is stuck-at-recessive. If this branch is being used by a CAN controller that acts as the *non-tx controller* of a node, then that node does not rule out the branch for communicating, but it simply accepts the delivery event it receives through its other branch (see Section 11.7.2).

Since it is possible that the Node Connection that fails is still being used by its node, *ofmNcFBprop* evaluates if the Node Kernel of that node becomes ofm and sends ofm errors to the other hub (to the hub that corresponds to the other Node Connection of the node). Specifically, if the node is an okA or an okB node, then the activity *ofmNcFBprop* decides that the node sends the ofm errors to the other hub, takes its first case and sets a token in *outFauMod*. This is because only an okA or an okB node is using its other Node connection for communicating. Otherwise, if the node is not an okA or an okB node, *ofmNcFBprop* selects it second case, which simply resets the marking of the place *evalFault*, as the ofm errors do not propagate to the rest of the system.

The proportion of the first case of *ofmNcFBprop* is calculated by dividing the number of already faulty branches that belong to the okA and the okB nodes by the total number of branches that were already faulty before the Node Connection failed. The resultant expression is:

$$
\begin{aligned}
&(okANodes \rightarrow Mark() + okBNodes \rightarrow Mark()) \ / \\
&(okNodeConns \rightarrow Mark() + 1 - \\
&(numBranches - numFaultyBranches \rightarrow Mark()))
\end{aligned}
$$

Note that the total number of branches that were already faulty before the Node Connection failed is calculated almost as in the second case of the activity *ncFailure* of the *nodeConnsR* submodel (see Section 12.4.5). The only difference is that now it is necessary to add 1 to $okNodeConns \rightarrow Mark()$. This is because *ncFailure* decreased the marking of *okNodeConns* after firing.

Finally, the proportion of the second case of *ofmNcFBprop* is calculated by dividing the number of already faulty branches that do not belong to the okA and the okB nodes by the total number of branches that were already faulty before the Node Connection failed. Thus, the expression of this case is:

$$((okNodeConns \rightarrow Mark() + 1 - (numBranches - numFaultyBranches \rightarrow Mark())) - (okANodes \rightarrow Mark() + okBNodes \rightarrow Mark())) / (okNodeConns \rightarrow Mark() + 1 - (numBranches - numFaultyBranches \rightarrow Mark()))$$

12.5 Quantitative assessment

In this section we quantitatively assess the reliability that can be achieved by equivalent systems that rely on CAN, CANcentrate and ReCANcentrate. For this purpose, we use the reliability models of the CAN bus and CANcentrate proposed in Chapter 10 and the reliability model of ReCANcentrate we have just described.

In particular, we compare the system *mission time* [MK05] achievable with CAN, CANcentrate and ReCANcentrate, i.e. the maximum amount of time during which a system that relies on them exhibits a reliability equal or greater than a certain value, for different number of nodes. Specifically, we analyze the system NFT/AR and the FT/AR$_1$ as long as they are ≥ 0.99999. As already said, this value is just taken as a reference and it corresponds to the reliability required by a throttle-by-wire system [MK05].

As concerns, the value of the different models' parameters, we use the default values specified for them in Tables 10.4, 10.5, 10.6 and 12.1; except for those that strictly depend on the number of nodes, e.g. *numBranches*, *hubCoreFRate*, etc. Also note that, as already explained in Section 10.4.4, we consider a 0% of ofm failures, which allows assessing what would be the reliability benefits of our stars in systems that include the appropriate mechanisms to deal with faults that are beyond the scope of these stars. In this sense, it is worth noting again that to consider an ofm greater than 0% would prevent us from analyzing the system reliability achievable by the CAN bus and our stars, since their contribution to this reliability would be masked by the effect of faults they do not address. Anyway, the value of all dependability parameters, e.g. coverages, cables' failure rates, etc., have been determined considering assumptions that never favor the stars when compared with the CAN bus (see Sections 10.4 and 12.3). Therefore, the result herein presented are likely to be lower bounds to the reliability achievable with CANcentrate and ReCANcentrate.

Figure 12.15 depicts the NFT/AR of equivalent systems that rely on CAN, CANcentrate and ReCANcentrate for different number of nodes. Notice that the reli-

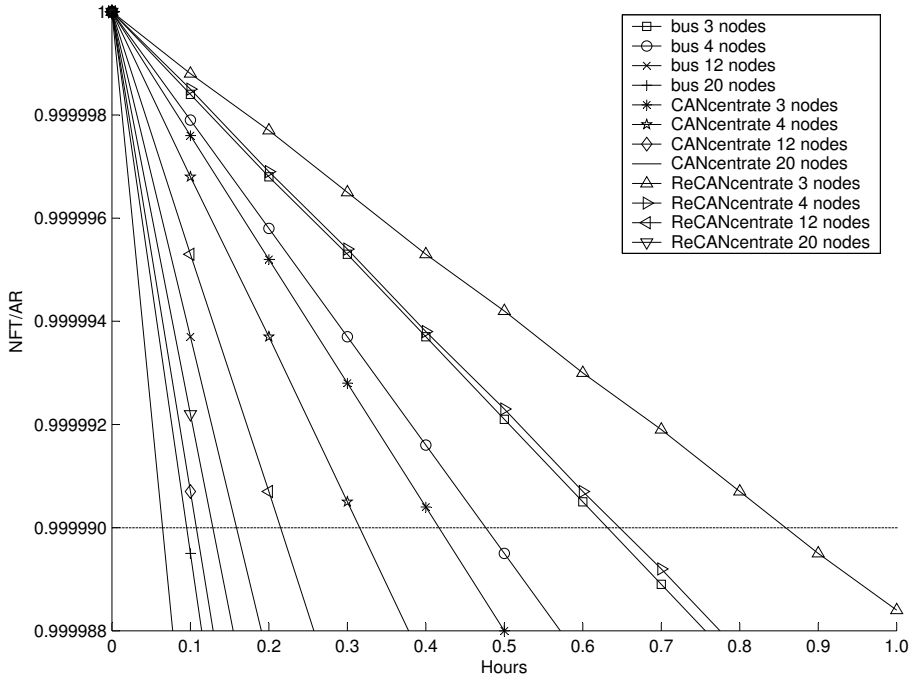

Figure 12.15: NFT/AR vs number of nodes

ability results of CAN and CANcentrate that appear in this figure were already discussed in Section 10.9.1. Specifically, for any number of nodes, CAN yields a bigger NFT/AR than CANcentrate. As already explained in the mentioned section, this is because a CANcentrate-based system includes more hardware than an equivalent CAN-based one, and an NFT/A system cannot benefit from the error-containment provided by a simplex star topology (see Section 10.2). However, notice again that the reduction of mission time provoked by CANcentrate is not outstanding in absolute terms and is kept almost constant (around the 35%) for any number of nodes, e.g. for 3 nodes CAN and CANcentrate achieve 0.63 and 0.41 hours respectively.

Regarding ReCANcentrate, this figure shows that the system NFT/AR achievable with it is higher than the one achieved with both CAN and CANcentrate for any number of nodes. These results, demonstrate that the fault-tolerance mechanisms of ReCANcentrate amply compensate its extra hardware, so that it can be well suited for improving the reliability of NFT/A systems. More specifically, note that the mission time improvement is not outstanding in absolute terms, but it is important from a relative point of view. For instance, when 3 nodes are considered, a ReCANcentrate-based system achieves a mission time 37% and 110% higher than

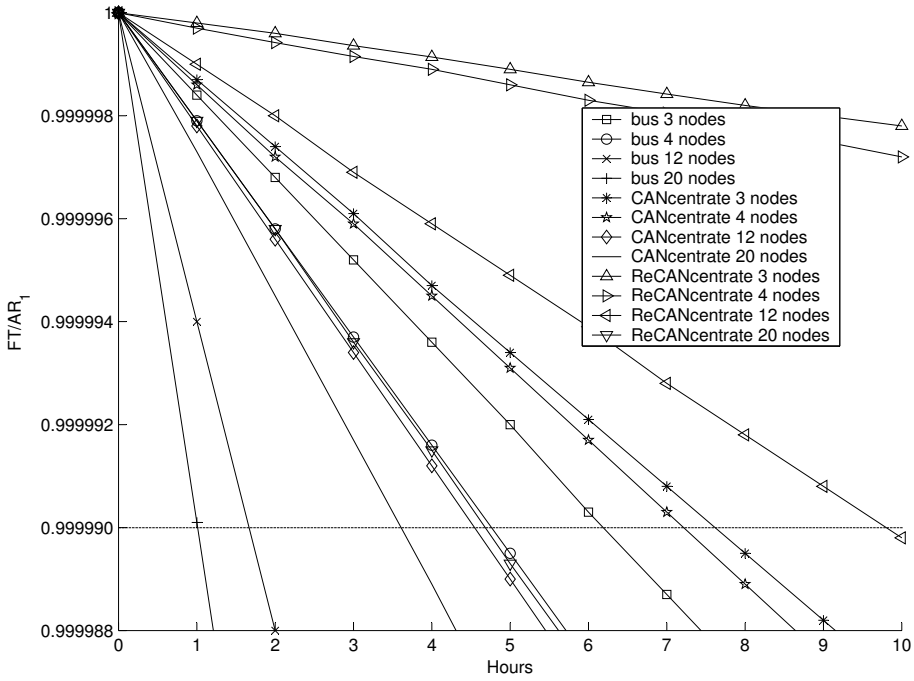

Figure 12.16: FT/AR$_1$ vs number of nodes

the mission time of an equivalent system that relies in CAN and CANcentrate, respectively. Similarly, for 20 nodes, ReCANcentrate improves the system mission time by 33% and 100% when compared with CAN and CANcentrate.

It is noteworthy that the relative mission time improvement achieved by ReCANcentrate slightly diminishes as the number of nodes increases. This suggests that special care should be taken to ensure a good enough reliability for the components whose amount are bigger in ReCANcentrate, e.g. of transceivers, specially when addressing a big number of nodes.

Figure 12.16 compares the FT/AR$_1$ of equivalent systems that rely on CAN, CANcentrate and ReCANcentrate. The results of CAN and CANcentrate were previously discussed in detail in Section 10.9.2. There we explained that CANcentrate achieves a higher FT/AR$_1$ than CAN in all cases and that the improvement in terms of mission time increases with the number of nodes. For instance, results indicate that CANcentrate improves the mission time of CAN around the 22% and the 260% for 3 and 20 nodes respectively.

As concerns the FT/AR$_1$ of a ReCANcentrate-based system, Figure 12.16 shows that it further improves the reliability of an equivalent FT/A CAN-based system for

any number of nodes. For example, ReCANcentrate yields mission times of 45.0, 7.1 and 4.6 hours for 3, 15 and 20 nodes respectively, whereas CAN yields 6.2, 1.3 and 1.0 hours for these same number of nodes. This implies that ReCANcentrate represents an improvement of the 626%, the 446% and of the 360%.

Moreover, note that ReCANcentrate improves reliability of FT/A systems much more than CANcentrate does. In fact, results show that a replicated star topology is suitable for improving the system reliability that can be achieved with a simplex star. For instance, for 3, 15 and 20 nodes, an FT/A ReCANcentrate-based system achieves mission times of 45.0, 7.1 and 4.6 hours respectively, whereas an equivalent CANcentrate-based one presents mission times of 7.6, 4.1, and 3.6 hours. This means that ReCANcentrate improves the mission time of CANcentrate by 534%, 73% and 28% for 3, 15 and 20 nodes respectively.

All these results concerning ReCANcentrate demonstrate that to include fault-tolerance mechanisms in addition to error-containment features, by means of a replicated star, can actually boost the reliability of FT/A systems; specially of those that include a small or an average number of nodes.

12.6 Conclusions

In this chapter we proposed a SANs model that allows measuring the reliability (when permanent hardware faults occur) of systems that rely on ReCANcentrate. In particular, we followed the same strategy we proposed in Chapter 10 for modelling the reliability of systems relying on CAN and on CANcentrate. In this sense, we decomposed the ReCANcentrate-based system into the same elementary components of a CAN and a CANcentrate-based one, and we kept the same assumptions we made for these two last types of systems. However, we had to specify some new entities (aggregations of components), since a ReCANcentrate-based system includes parts that are not present in the previous systems, e.g. the interlinks that interconnect both its hubs. Moreover, we had also to introduce new assumptions that are specific to ReCANcentrate, e.g. the length of the interlinks. Again, all these new assumptions are made taking special care not to favor ReCANcentrate in the comparison.

In particular, one important assumption that should be highlighted is related to the schema that each node uses to connect to both hubs. Specifically, we supposed that each node includes two CAN controllers and that uses each one of them to connect to a different hub. We took this option because it is the one we proposed for achieving a high degree of fault tolerance in ReCANcentrate (see Chapter 11). In this sense, the reliability evaluation herein presented throws light on the suitability

of this node's architecture prior to its inclusion in a new ReCANcentrate prototype.

Another aspect that reflects the fact that we followed the modelling strategy proposed in Chapter 10 is that despite being more complex, the structure of the ReCANcentrate's model is analogous to the ones of CAN and CANcentrate. The reason of this bigger model's complexity is that a ReCANcentrate-based system has a more complicated structure and its network is provided with more fault-tolerance mechanisms when compared with CAN and CANcentrate. In particular, this makes more difficult to model the propagation of the errors generated by faults as well as the way in which they are contained, i.e. the *coverage process* and hence the interaction between different submodels become much more complex.

We used the ReCANcentrate's model together with the models we proposed for CAN and CANcentrate in Chapter 10 to compare the reliability achievable by equivalent systems relying on ReCANcentrate, CAN and CANcentrate. On the one hand, results demonstrate that the additional fault-tolerance mechanisms of ReCANcentrate compensate its extra hardware, so that it is able to improve the reliability of NFT/A systems when compared with CAN (around the 35% in terms of mission time). This is an interesting result that quantitatively corroborates the advantage of the fault-tolerance mechanisms of a replicated star topology over the dependability-related features of both simplex bus and simplex star topologies. In fact, the results we obtained renew the interest in using star topologies for NFT/A systems since, as we quantitatively corroborated in Chapter 10, these kind of systems cannot benefit from the enhanced error-containment mechanisms of a simplex star topology. Moreover, this results become even more noticeable if we take into account that, to our best knowledge, no one has quantitatively assessed the reliability benefits of a replicated star topology.

On the other hand, results quantitatively demonstrate that a replicated star topology such as ReCANcentrate can really boost the reliability of FT/A systems in terms of mission time. For instance, a ReCANcentrate-based system improves the mission time of an equivalent CAN-based one by the 626%, the 446% and the 360% when 3, 15 and 20 nodes are considered respectively. Furthermore, this improvement is outstanding not only in relative terms, but also from an absolute point of view. Following the last example, a ReCANcentrate-based system achieves mission times of 45, 7.1 and 4.6 hours for 3, 15 and 20 nodes, whereas an equivalent CAN-based system achieves 6.2, 1.3 and 1 hours respectively.

Regarding the comparison between CANcentrate and ReCANcentrate, it is noteworthy that the results we obtained also quantitatively corroborate the advantage of using a replicated star topology instead of a simplex star for improving the reliability of FT/A systems. In fact, we believe that these results really advocate the

use of adequate replicated star topologies for the most demanding highly-reliable systems. For instance, ReCANcentrate improves the mission time of CANcentrate by 534%, 73% and 28% for 3, 15 and 20 nodes respectively.

Finally, notice that as already explained in Chapter 10 the results we obtained with our models can be extrapolated to other technologies. In this sense, the work presented herein quantitatively corroborates, for the very first time, the suitability of using replicated star topologies for improving the reliability of FT/A systems. Moreover, since the assumptions our models rely on may have been too detrimental for ReCANcentrate, and some potential dependability advantages of stars were not considered, the results we obtained are likely to be lower bounds to the system reliability achievable with a replicated star topology such as ReCANcentrate.

Parameter	Default value	Meaning
numNodes	3, 15	Number of nodes
numBranches	6, 30	Number of total branches of the network when 3 and 15 nodes are considered respectively (it is supposed that each node is connected to both hubs)
numInterlinks	2	Number of interlinks that interconnect the two hubs of ReCANcentrate
flipLnkCov	0.95	Probability with which the hub successfully diagnoses a bit-flipping fault at an uplink hub port
flipSlnkCov	0.95	Probability with which the hub successfully diagnoses a bit-flipping fault at its sublink port/s, when the errors received through that sublink/s are generated by a bit-flipping fault affecting the core of the other hub or a sublink
flipSlnkPropCov	0.5	Probability with which a hub successfully diagnoses a bit-flipping fault at its sublinks ports, when the errors received through that sublinks are generated by a bit-flipping fault that the other hub has not been able to diagnose and isolate
connCov	0.95	Probability with which a node is able to continue communicating through one hub only, when the hubs are coupled and a fault prevents it from communicating through one of them
decConnCov	0.0	Probability with which a node is able to continue communicating through one hub only, when the hubs are decoupled and a fault prevents it from communicating through one of them
decCov	0.0	Probability with which the nodes of ReCANcentrate can still communicate with each other using two independent stars when the hubs become decoupled
hubCoreFRate	$1.27560 \cdot 10^{-6}$, $1.94095 \cdot 10^{-6}$	*Hub Core* failure rate when it is provided with 3 and 15 uplink ports respectively
hubCoreOfmProp	0.0	*Hub Core out-of-fault-model* proportion
hubCoreFlipProp	0.3333	*Hub Core* bit-flipping proportion
hubIOFRate	$6.73258 \cdot 10^{-7}$	*Hub IO* failure rate

hubIOOfmProp	0.0	*Hub IO out-of-fault-model* proportion
hubIOStrProp	0.3333	*Hub IO* stuck-at-recessive proportion
hubIOStdProp	0.3333	*Hub IO* stuck-at-dominant proportion
hubIOFlipProp	0.3333	*Hub IO* bit-flipping proportion
lnkAttchFRate	$6.34588 \cdot 10^{-8}$	*Attachment* failure rate when it represents the uplink or the downlink of ReCANcentrate
lnkAttchOfmProp	0.0	*Attachment out-of-fault-model* proportion when it represents the uplink or the downlink of ReCANcentrate
lnkAttchStrProp	0.25	*Attachment* stuck-at-recessive proportion when it represents the uplink or the downlink of ReCANcentrate
lnkAttchStdProp	0.25	*Attachment* stuck-at-dominant proportion when it represents the uplink or the downlink of ReCANcentrate
lnkAttchFlipProp	0.25	*Attachment* bit-flipping proportion when it represents the uplink or the downlink of ReCANcentrate
lnkAttchDisProp	0.25	*Attachment* physical disruption proportion when it represents the uplink or the downlink of ReCANcentrate
slnkAttchFRate	$6.34588 \cdot 10^{-8}$	*Attachment* failure rate when the it represents a sublink of ReCANcentrate
slnkAttchOfmProp	0.0	*Attachment out-of-fault-model* proportion when the it represents a sublink of ReCANcentrate
slnkAttchStrProp	0.25	*Attachment* stuck-at-recessive proportion when the it represents a sublink of ReCANcentrate
lnkStdProp	0.25	*Attachment* stuck-at-dominant proportion when the it represents a sublink of ReCANcentrate
slnkAttchFlipProp	0.25	*Attachment* bit-flipping proportion when the it represents a sublink of ReCANcentrate
slnkAttchDisProp	0.25	*Attachment* physical disruption proportion when the it represents a sublink of ReCANcentrate

Table 12.1: Parameters specific to the ReCANcentrate model

Chapter 13

Conclusions and future work

CAN has been extensively used in a broad range of applications, including highly-dependable ones. However, the use of CAN in this last context has been controversial due to few factors, some of which arise from its simplex bus topology. The main drawback of any protocol using a bus topology is that the structure of the network presents multiple components, which have direct electrical connections to each other without proper error containment. As a consequence, a fault in any of them may generate errors that propagate and effectively prevent further communication to take place. Moreover, even if errors were properly contained in a simplex bus topology, it lacks mechanisms for tolerating the faults that generate them.

Despite the dependability limitations of CAN, there is still an important interest in using it, given its low-cost, electrical robustness, good real-time properties and widespread use. This situation is clear in the industrial automation and the automotive industry domains, where CAN has played a key role during many years specially due to its robustness. In particular, among all the dependability attributes, the work herein presented focuses on the *reliability*, since the level of this attribute required by newer applications such as those of the domains just mentioned is continuously increasing. In fact, alternative highly reliable protocols have been recently developed, e.g. TTP/C [KG94] and FlexRay [Fle05], in order to complement or compete with CAN in those areas.

Moreover, reliability is also desirable for many other applications such as domotics, where CAN has also enjoyed an important position. Nowadays, these kind of applications are also demanding increasing levels of reliability, since the requirements in number of nodes and services are also growing. This can be seen, for instance, in the automotive or in the home automation domains, where comfort or entertainment features are gaining in importance.

13.1 Thesis validation and contributions

In response to the above-mentioned limitations of CAN and given the interest in continuing using it in highly dependable applications, the thesis supported by this dissertation basically claims that it is possible to improve error containment, fault tolerance and then reliability of CAN-based systems by changing its simplex bus topology to adequate star topologies.

This document described the work we have conducted to validate this thesis, as well as the contributions that resulted from the different tasks we carried out to attain this objective. Next, we explain how we have validated the different specific assertions that constitute the above-mentioned thesis, focusing on the main contributions.

13.1.1 First assertion

The first assertion of the thesis basically states that it is possible to improve error containment of CAN by using an adequate simplex star topology whose hub is provided with adequate mechanisms that contain errors at their ports of origin.

The first step towards validating this assertion was to formalize the error containment limitations of CAN by defining the concept of k-severe failure. We explained that any network relying on a bus topology presents multiple of these points, since the fact of relying on a bus topology seriously limits the capacity of CAN nodes to thoroughly detect and contain errors. More specifically, the single way of containing errors in a CAN bus is to shut down the nodes that are faulty. This requires that each node diagnoses itself as being faulty when it is the cause of errors that prevent other nodes from communicating. However, contributions of all nodes are irreversibly mixed in a CAN bus. This limits the accuracy of the fault-diagnosis mechanisms nodes are provided with, which can unfairly penalize correct nodes. Moreover, CAN nodes can fail and stop performing these fault-passivation actions. Finally, shutting nodes down cannot contain the errors generated by faults that affect other parts of the network, e.g. connectors or cables.

We described some solutions that have been proposed for improving error containment in CAN. However, we showed that no one of them effectively achieves this purpose. Some of them are negatively affected by common-mode and/or spatial proximity failures, whereas other ones only deal with a narrow range of types of errors, or are not even compatible with CAN COTS components, CAN applications or CAN-based protocols.

Thus, in order to effectively improve error containment in CAN, while overcom-

ing the shortcomings of other solutions, we designed a CAN-compliant simplex star topology called CANcentrate. The hub of CANcentrate receives each node contribution through a dedicated uplink, couples all contributions and then broadcasts back the resultant coupled signal to each node through the corresponding downlink. Since this coupling is done in a fraction of the bit time, CANcentrate keeps the recessive/dominant transmission and the in-bit response features of CAN. This yields two important benefits. First, CANcentrate still presents all the dependability advantages of CAN that are related to these features. Second, its compatibility with any CAN-based application or protocol is guaranteed.

In addition, the hub of CANcentrate includes novel error-detection and fault-treatment mechanisms that go beyond the capacity of CAN and any other previously proposed solution. On the one hand, its hub is able to contain, at their port of origin, stuck-at and bit-flipping errors generated by faults that affect nodes and/or media. This is an important feature of CANcentrate since, as stated above, any existing CAN-based solution only deals with few types of errors or is not able to treat faults happening at certain locations, e.g. faults happening at the media. On the other hand, we showed that the use of a separated uplink/downlink per node allows the hub to distinguish each node contribution from the coupled signal. In this way, the hub can implement error-detection and fault-treatment mechanisms that are more effective than the ones included in any other CAN-based solution. In particular, this implies that CANcentrate detects errors and treats faults with a higher coverage than other communication subsystems based on CAN. Such a coverage is well known to be a fundamental parameter, since it has a great impact on the system dependability.

This document thoroughly described the internal structure of the hub of CANcentrate, as well as its error-detection and fault-treatment mechanisms. Special care was taken when explaining in detail the mechanisms for dealing with bit-flipping faults. In fact, the ability of the hub of CANcentrate to deal with this type of faults is one of the main contributions of this work. For this reason, this explanation takes up an important part of this document.

We also demonstrated the feasibility of the implementation of CANcentrate by presenting and describing its first prototype based on COTS components. Furthermore, this prototype also allowed us to carry out a first evaluation of the performance of CANcentrate, as well as of the effectiveness of the error-detection and fault-treatment mechanisms by using fault injection.

13.1.2 Second assertion

The second assertion of the thesis claims that the advanced error-containment capabilities of an adequate simplex star topology are suitable to increase the reliability of a CAN-based system that already accepts or tolerates node failures or disconnections.

Thus, in addition to the design and prototyping of CANcentrate, we quantitatively assessed, by means of *Stochastic Activity Networks* (SANs) models, the reliability benefits of using it as the underlying communication infrastructure. In particular, we compared the reliability of a system relying on CAN and of an equivalent system based on CANcentrate in the presence of permanent hardware faults. To our best knowledge, this is the first formal (quantitative) comparison between a simplex star and a simplex bus that takes into account the capacity of the hub and nodes to contain errors, different failure modes of the components, and implementation aspects.

In order to compare the reliability of equivalent systems relying on CAN and CANcentrate, we classified them into *non-fault-tolerant/accepting systems* (NFT/A systems) and *fault-tolerant/accepting systems* (FT/A systems). This classification is based on the system's requirements for operating, which considerably vary depending on the specific application the system is intended for. In this sense, NFT/A systems are those that can only deliver their services as long as all their nodes are non-faulty and can communicate with each other; whereas FT/A systems accept or tolerate the failure or the disconnection of up to k of N nodes. Since the requirements of both types of systems for operating are different, we defined two different metrics to quantify their reliability. On the one hand, for the case of NFT/A systems, we specified a metric called *non-fault-tolerant / accepting system reliability (NFT/AR)* as the probability with which all nodes of a system can correctly operate and communicate with each other throughout a given interval of time. On the other hand, in order to quantify the reliability of FT/A systems, we used a new metric called *fault-tolerant/accepting system k-reliability* (FT/AR$_k$), which we defined as the probability with which at least $N - k$ of the N nodes of a system can correctly operate and communicate among them throughout a given interval of time. Note that the FT/AR$_k$ can be understood as the probability of not suffering a k-severe failure. In particular, we measured the FT/AR$_1$, since it is the value for k with which CANcentrate intuitively yields the least reliability benefits.

The models we proposed for CAN and CANcentrate are complete as they include parameters for all the relevant aspects that can influence the system reliability. Moreover, these parameters allow to both refine the results as more system's details are known and carry out sensitivity analyses with respect to different sys-

tem's aspects. Anyway, we proposed default values for all model parameters in order to set up a case of reference with which we quantified the system reliability achievable with CAN and CANcentrate. These values should not be taken as real figures, but as initial reference values that can be considered as realistic.

In particular, notice that in order to compare the reliability achievable with each one of the referred communication infrastructures it is necessary to assume that the system includes 100% effective mechanisms that deal with the faults that are beyond the capacities of CAN and CANcentrate. Otherwise, the contribution of these two infrastructures to the system reliability would be masked by the effect of faults they do not address, e.g. by babbling-idiot faults. In fact, in order to fully benefit from CANcentrate, the system should include mechanisms that deal with these faults. Moreover, it is important to note that all model parameters that characterize the CAN-based and the CANcentrate-based systems were determined with special care not to favor the star in the comparison. Thus, results herein presented are likely to be lower bounds to the reliability that can be achieved with a simplex star such as CANcentrate.

The results quantitatively corroborated for the very first time the advantages that the error-containment capabilities of a simplex star topology such as CANcentrate yield in terms of FT/AR_1. Additionally, we carried out several analyses of the sensitivity of the FT/AR_1 with respect to several parameters such as, for example, the error-containment coverage of the hub and nodes, the hub's failure rate, etc. These analyses constitute an important contribution, since no one has previously quantified how these aspects influence the reliability benefits yielded by a simplex star topology over a simplex bus.

Furthermore, besides demonstrating that CANcentrate improves FT/AR_1, we also found out that it reduces the NFT/AR when compared with a CAN bus. This result was expected since a CANcentrate network includes more hardware than an equivalent CAN one, so that the probability of suffering from a fault is higher in the former. However, to quantitatively corroborate and characterize this issue is also an important contribution of this dissertation. For instance, we showed that the decrease of mission time that CANcentrate can cause when compared with CAN is not outstanding in absolute terms and that it remains almost constant for any number of nodes.

Finally, apart from this CANcentrate's limitation, we explained that although CANcentrate reduces the multiple points of severe failure that appear in a CAN bus to one single point of failure (the hub), the existence of such a point can still be unacceptable for some applications with high reliability requirements.

13.1.3 Third assertion

The third assertion of the thesis states that it is possible to enhance both error containment and fault tolerance of CAN and of CAN-based simplex star topologies, by means of an adequate replicated star topology that includes two hubs that can contain errors at their ports of origin and also errors generated by the other hub, and that includes mechanisms to tolerate faults at links and at one of the hubs.

This assertion constitutes our response to the above-mentioned limitations of CANcentrate. Specifically, in order to validate this claim we designed a CAN-compliant replicated star topology called ReCANcentrate. To our best knowledge, ReCANcentrate is the only replicated star topology that has been proposed for CAN so far.

ReCANcentrate includes two hubs that exchange their traffic through replicated interlinks. Each hub couples its own traffic with the traffic received from the other hub, so that both hubs create a single broadcast domain. As in CANcentrate, this coupling is done in a fraction of the bit time, so that ReCANcentrate presents recessive/dominant transmission together with in-bit response. Again, this keeps all the good dependability properties of CAN that are related to these features, and guarantees compatibility with CAN-based applications, protocols and COTS components.

Regarding the nodes, each one of them includes two CAN controllers and uses each one of them to transmit/received to/from a given hub. We explained that hub coupling forces both hubs to transmit the same value bit by bit in their downlinks, guaranteeing the traffic to be the same in both stars. In this sense, the hub coupling provides a synchronization mechanism for transmitting and receiving frames in both stars that simplifies the way in which each node manages the redundant traffic. This is also an important contribution of this dissertation, since to use event-triggered communication channels in parallel (such as CAN-replicated channels) poses serious difficulties in managing the traffic at each node.

Going back to the dependability-related features of ReCANcentrate, we showed that each one of its hubs includes the same mechanisms as a CANcentrate hub to contain stuck-at and bit-flipping errors at their ports of origin. Additionally, each hub is provided with mechanisms to contain these types of errors when they are generated by faults affecting the interlinks and/or the other hub. Moreover, ReCANcentrate further allows tolerating one faulty hub, faults at up to $M - 1$ of M interlinks, as well as one faulty link per node. In fact, it can tolerate the simultaneous failure of a hub and of one of the links of each node to the hubs as long as all nodes are connected to a non-faulty hub. Note that in this context we use the term *link* not only to refer to the components that constitute an uplink/downlink, but also

to the components the node uses for communicating through that uplink/downlink, e.g. to the CAN controller.

We explained the internals of the hub of ReCANcentrate, focusing on the main differences with respect to the hub of CANcentrate. Additionally, we outlined the strategy each node of ReCANcentrate uses to manage the transmissions and the receptions on the replicated star, as well as to tolerate faults.

Afterwards, we described a first prototype based on COTS components that demonstrates the feasibility of the implementation of ReCANcentrate. The interconnection of each node to the replicated star was simplified, but the prototype still allowed us to successfully evaluate the main ReCANcentrate error-containment and fault-tolerance properties, as well as its performance.

13.1.4 Fourth assertion

The last assertion of the thesis states that the improved dependability features of an adequate replicated star topology are appropriate to increase the reliability of both CAN-based systems that do not accept or tolerate node failures or disconnections and CAN-based systems that can do that.

As we did for CANcentrate, we quantitatively evaluated, by means of *Stochastic Activity Networks* (SANs), the reliability benefits of using ReCANcentrate. For that we compared the reliability of equivalent systems relying on CAN, CANcentrate and ReCANcentrate in the presence of permanent hardware faults. To our best knowledge, this is the first formal (quantitative) evaluation of the dependability benefits achievable by means of a replicated star topology when compared with both a simplex bus and a simplex star topology.

We thoroughly described the SANs model we proposed for measuring the reliability of a ReCANcentrate-based system. Specifically, we followed the same modelling strategy we proposed for the case of CAN and CANcentrate. Basically, we decomposed the ReCANcentrate-based system into the same elementary components of a CAN and a CANcentrate-based one, and we kept the same assumptions we made for these two last types of systems. However, we had to model some features and introduce new assumptions and parameters that are specific to ReCANcentrate. Again, all these new assumptions were made taking special care not to favor ReCANcentrate in the comparison.

In fact, we showed that the model of ReCANcentrate is much more complex than the models of CAN and CANcentrate. In particular, it was much more difficult to model how the errors propagate and how faults are isolated and/or tolerated. Moreover, before adopting a definitive strategy for modelling the reliability of systems

relying on CAN, CANcentrate and ReCANcentrate, we had to explore different alternatives. This was needed because the ReCANcentrate's model complexity can easily lead to inefficiency problems in terms of computation time when using SANs. In this sense, we believe that the model strategy finally adopted herein is not only suitable for addressing the referred systems, but it constitutes a good example that can help other researchers to evaluate the reliability that can be achieved with other communication topologies and/or technologies.

As regards the reliability results obtained when evaluating ReCANcentrate, note that they are likely to be lower bounds to the system dependability achievable by ReCANcentrate, given the special concern taken to ensure it was not favored in the analyses and the fact that several other advantages of ReCANcentrate, such as the minimization of the impact of spatial proximity faults, were not even modelled.

Results demonstrated that the ReCANcentrate's additional fault-tolerance mechanisms largely compensate the potential reduction of reliability caused by its extra hardware. On the one hand, they indicated that ReCANcentrate can improve the reliability of NFT/A systems when compared with CAN and CANcentrate. Although this improvement is not outstanding in absolute terms, it is an interesting result that quantitatively corroborates the advantage of the fault-tolerance mechanisms of a replicated star topology over the dependability-related features of both simplex bus and simplex star topologies. In fact, this result renews the interest in using star topologies for NFT/A systems since, as said before, these kind of systems cannot benefit from enhanced error-containment when they rely on a simplex star topology.

On the other hand, results quantitatively demonstrated that a replicated star topology such as ReCANcentrate can really boost the reliability of FT/A systems in terms of mission time, specially when compared with a CAN bus. Furthermore, this improvement is outstanding not only in relative terms, but also from an absolute point of view. In this sense, the work presented herein quantitatively supports, for the very first time, the interest in using replicated star topologies for improving the reliability of FT/A systems, specially of highly-reliable ones.

In fact, although our analyses refer to the case of CAN, and other technologies, such as TTP/C or FlexRay, deal with different failure modes, we believe that our results really advocate the use of adequate star topologies for improving reliability when using these technologies. On the one hand, they use similar components with similar failure rates. On the other hand, failure modes can be abstracted so that what really matters is the proportion of failures that can be covered by the hub and the nodes.

Finally, we would like to highlight that our reliability analyses are not intended

to provide absolute figures for CAN, CANcentrate or ReCANcentrate, but to compare the system reliability that can be achieved with these infrastructures in order to justify the interest in using adequate stars. In this sense, our analyses pursue the objective of dependability evaluation, i.e. to guide the design and implementation of systems by analyzing how different options and decisions affect their dependability.

13.2 Publication of results

This section gathers the different publications that resulted from the work herein described, as well as those publications that are directly related to this dissertation.

13.2.1 Preliminary publications

The following publications are prior to the work described in this dissertation. However, they are directly related to it as some parts of the work therein presented, e.g. the use of COTS components and programmable hardware for providing communication facilities at low cost, constitute a source of inspiration for some of the solutions proposed to validate our thesis.

Peer-reviewed papers published in international conferences:

- Guillermo Rodríguez-Navas, Manuel Barranco, Julián Proenza. *COTS-Based Hardware Support to Timeliness in CAN networks*. Proceedings of the 2003 IEEE International Conference on Emerging and Factory Automation (ETFA 2003), Lisboa, Portugal, 2003.
- Guillermo Rodríguez-Navas, Manuel Barranco, Julián Proenza. *Harmonizing Dependability and Real Time in CAN Networks*. Proceedings of the 2nd Euromicro International Workshop in Real-Time LANS in the Internet Age, Porto, Portugal, 2003.

13.2.2 Publication of results presented in this dissertation

Next follow different sets of publications that spread the main results and contributions presented in this dissertation.

The first set of publications are concerned with the design, implementation and tests of CANcentrate. These publications roughly cover Chapters 4, 5, 6, 7 and 8.

Peer-reviewed papers published in international journals:

- Manuel Barranco, Julián Proenza, Guillermo Rodríguez Navas and Luís Almeida. *An Active Star Topology for Improving Fault Confinement in CAN Networks*. IEEE Transactions on Industrial Informatics, vol. 2, issue 2, May 2006, pp.78-85.

Peer-reviewed papers published in international conferences:

- Manuel Barranco, Guillermo Rodríguez-Navas, Julián Proenza, and Luís Almeida. *CANcentrate: An active star topology for CAN networks*. Proceedings of the 5th IEEE International Workshop on Factory Communication System (WFCS 2004), Vienna, Austria, September 2004. BEST-PAPER AWARD.

Papers published in other international conferences or books:

- Manuel Barranco, Julián Proenza, Guillermo Rodríguez Navas and Luís Almeida. *A CAN hub with Improved Error Detection and Isolation*. 10th International CAN Conference (ICC 2005), Rome, Italy, March 2005.

The second set of publications explain the basics and main results of the work we conducted in order to quantify the improvement of system reliability that can be achieved with CANcentrate over CAN. These publications describe part of the content of Chapter 10.

Peer-reviewed papers published in international journals:

- Manuel Barranco, Julián Proenza and Luís Almeida. *Quantitative comparison of the error-containment capabilities of a bus and a star topology in CAN networks*. IEEE Transactions on Industrial Electronics. To be published in 2010. Digital Object Identifier: 10.1109/TIE.2009.2036642.

Peer-reviewed papers published in international conferences:

- Manuel Barranco, Julián Proenza and Luís Almeida. *First results of the assessment of the improvement of error containment achieved by CANcentrate*. Proceedings of the 6th IEEE International Workshop on Factory Communication Systems (WFCS 2006), Work-in-progress session, Torino, Italy, 2006.

The third set of publications describe the design, implementation and tests of ReCANcentrate. Note that the three last publications refer to the same paper. This paper was first published in the SAE (Society of Automotive Engineering) 2006 World Congress. Then, it was selected for inclusion in a SAE book that contains 40 SAE technical papers covering six years (2001-2006) of research on safety-critical automotive systems, as well as for inclusion in the 2006 SAE Transactions Journal of Passenger Cars. This third set of publications approximately covers Chapter 11.

Peer-reviewed papers published in international conferences:

- Manuel Barranco, Luís Almeida and Julián Proenza. *ReCANcentrate: A replicated star topology for CAN networks*. Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2005), Catania, Italy, September 2005.
- Manuel Barranco, Julián Proenza and Luís Almeida. *Designing and Verifying Media Management in ReCANcentrate*. Proceedings of the 13rd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2008), Work-in-progress session, Hamburg, Germany, September 2008.

Papers published in other international conferences or books:

- Manuel Barranco, Julián Proenza and Luís Almeida. *Management of Media Replication in ReCANcentrate*. 12th International CAN Conference (iCC 2008), Barcelona, Spain, March 2008.
- Manuel Barranco, Luís Almeida and Julián Proenza. *Experimental assessment of ReCANcentrate, a replicated star topology for CAN*. SAE 2006 World Congress, Detroit, Michigan, USA, April 2006.
- Manuel Barranco, Luís Almeida and Julián Proenza. *Experimental assessment of ReCANcentrate, a replicated star topology for CAN*. Safety-Critical Automotive Systems, Society of Automotive Engineers, USA, August 2006.
- Manuel Barranco, Luís Almeida and Julián Proenza. *Experimental assessment of ReCANcentrate, a replicated star topology for CAN*. SAE 2006 Transactions Journal of Passenger Cars: Electronic and Electrical Systems, Society of Automotive Engineers, USA, 2007.

The fourth set of publications describe the basics and the results of the reliability modelling of ReCANcentrate, which we carried out to quantify the system relia-

bility improvement achieved by this infrastructure when compared with CAN and CANcentrate. They correspond to Chapter 12.

Peer-reviewed papers published in international conferences:

- Manuel Barranco, Luís Almeida and Julián Proenza. *First quantitative results of the dependability improvement achieved by ReCANcentrate*. Proceedings of the 14th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2009), Work-in-progress session, Palma de Mallorca, Spain, September 2009.

- Manuel Barranco, Luís Almeida and Julián Proenza. *Reliability Improvement Achievable in CAN-based Systems by Means of the ReCANcentrate Replicated Star Topology*. To be published in the Proceedings of the 8th IEEE International Workshop on Factory Communication Systems (WFCS 2010), Nancy, France, 2010.

The fifth set of publications present both CANcentrate and ReCANcentrate in an integrated and balanced way. They summarize the most important aspects of Chapters 4, 5, 6, 7, 8 and 11.

Peer-reviewed papers published in international journals:

- Manuel Barranco, Julián Proenza and Luís Almeida. *Boosting the Robustness of Controller Area Networks: CANcentrate and ReCANcentrate*. IEEE Computer, vol. 42, issue 3, May 2009, pp. 66-77.

Papers published as international book chapters:

- Juan Pimentel, Julián Proenza, Luís Almeida, Guillermo Rodríguez-Navas, Manuel Barranco and Joaquim Ferreira. *Dependable Automotive CAN Networks*. Handbook on Automotive Embedded Systems. CRC Press. Edited by Nicolas Navet and Françoise Simonot-Lion from LORIA (France). 2009.

Papers published in international magazines:

- Manuel Barranco, Julián Proenza, Guillermo Rodríguez Navas and Luís Almeida. *Pushing error containment and fault tolerance in CAN by means of star topologies: CANcentrate and ReCANcentrate*. CAN Newsletter Automotive Networks 2006, CAN in Automation GmbH Germany, 2006.

13.2.3 Publication of future work's first results

The next publications present the results of some research tasks unveiled as future work of this dissertation. The first one of them proposes additional mechanisms to enforce data consistency in ReCANcentrate even when hubs become temporarily or permanently decoupled. The second one presents the basics and first results of a new prototype of ReCANcentrate that takes full advantage of the fault-tolerance capabilities of the media management we proposed in Chapter 11.

Peer-reviewed papers published in international conferences:

- Manuel Barranco, Julián Proenza and Luís Almeida. *Maintaining data consistency in ReCANcentrate during hub decouplings*. Proceedings of the 7th IEEE International Workshop on Factory Communication Systems (WFCS 2008), Work-in-progress session, Dresde, Germany, May 2008.

- Manuel Barranco, David Geßner, Julián Proenza and Luís Almeida. *Demonstrating the feasibility of media management in ReCANcentrate*. Proceedings of the 14th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2009), Work-in-progress session, Palma de Mallorca, Spain, September 2009.

13.3 Applicability of the contributions

This section briefly highlights the practical relevance of some of the contributions that result from the work conducted to validate our thesis.

On the one hand, we believe that the two main contributions of the work presented in this dissertation, i.e. the design and implementation of CANcentrate and ReCANCentrate, constitute an important step towards improving reliability in a wide range of applications, including those where CAN has ruled for many years.

A clear example where our stars are applicable are distributed embedded control systems for safety-critical applications, e.g. x-by-wire systems, which are now widespread in several domains, such as avionics and the automotive industry. Applications that require similar levels of reliability as these ones are robotics and health-care monitoring systems. In this sense, CANcentrate and ReCANcentrate can represent a first step towards providing CAN with most of the features concerning error-containment and/or fault-tolerance that are typical of recent highly-reliable protocols such as TTP and FlexRay. In particular, the reliability advantages of our stars can be further increased by including new fault-treatment capabilities

within their hubs or by investing in their quality. Moreover, given their compatibility with CAN, it is also possible to integrate them with other CAN-based solutions proposed for improving reliability and real-time performance such as, for example, with FTT-CAN.

Another important field where CANcentrate and ReCANcentrate are also applicable is factory automation, where CAN has played an important role. This is because applications of this domain normally require a communication infrastructure in which a minimum number of nodes can communicate with each other throughout a complete interval of time, e.g. in a factory plant it is required that a fault in any of its components jeopardizes the communication capabilities of the lower number of nodes as possible.

But CANcentrate and ReCANcentrate can be used in practice not only for improving dependability of highly-reliable applications, but also in other domains where CAN is nowadays widespread. This is mainly because both stars are fully compliant with CAN, being transparent for any existing CAN-based application using Commercial Off-The-Shelf (COTS) CAN components. This brings the opportunity for improving dependability in applications such as intra-building communication or home automation, where the main objective is to provide service to the maximum number of nodes even when faults occur. In this sense note that CANcentrate and ReCANcentrate bring to CAN valuable fault-diagnosis facilities, as well as similar features regarding error containment and/or fault tolerance than hubs bring to Ethernet networks.

Finally, it is noteworthy that due to their practical interest, CANcentrate and ReCANcentrate have been the subject of two patent filings:

- Julián Proenza, Guillermo Rodríguez Navas, Manuel Barranco, Luís Almeida. *Red de comunicaciones de protocolo CAN con topología en estrella*. Universitat de les Illes Balears (UIB), Spain, Universidade de Aveiro (UA), Portugal, 2004. Patent number: 200402207.

- Julián Proenza, Manuel Barranco, Luís Almeida. *Red de comunicaciones de protocolo CAN con topología en estrella replicada y procedimiento de acoplamiento de dicha red*. Universitat de les Illes Balears (UIB), Spain, Universidade de Aveiro (UA), Portugal, 2005. Patent number: 200502292.

On the other hand, apart from the applicability of CANcentrate and ReCANcentrate, we believe that the models we proposed for quantifying their reliability are also valuable in practice. Firstly, note that the results we obtained are significant not only from the theoretical, but also from the practical point of view, since

although there has been a growing interest in using star topologies in field-bus communications, e.g. such as in TTP or FlexRay, it was still not so clear whether stars achieve a higher degree of dependability than buses in the context of distributed embedded systems.

Secondly, as concerns the engineering contribution of these models, notice that they are useful for any researcher or engineer working on highly-dependable distributed embedded applications. Since our models allow performing sensitivity analyses with respect to some parameters directly related to electronic devices, e.g. the failure rate of components, they can guide engineers in the process of designing and implementing an adequate bus or star-based network for achieving a specific level of dependability in CAN-based systems.

Furthermore, our models also include parameters that allow characterizing different architectural or design options such as, for example, the number of interlinks or the coverage with which nodes tolerate a hub decoupling in ReCANcentrate. This means that our models allow engineers to evaluate if making an effort in improving certain architectural aspects or functionalities of a star topology is an issue deserving of attention.

Finally, we believe that our models are not only relevant in order to quantify the advantages of star topologies in CAN networks, but they are also valuable contributions towards helping other researchers to evaluate the reliability that can be achieved with other communication topologies and/or technologies.

13.4 Future research

The work described in this dissertation opens the door to carry out future research on different interesting issues.

Single cable CAN-compliant star topologies

The cost of the cabling is an important factor in distributed embedded systems. Despite the gains or losses in cabling length being highly dependent on the network physical layout, star topologies generally lead to longer cabling than corresponding buses, which results in higher costs and difficulties of installation. This problem is specially relevant in the case of CANcentrate and ReCANcentrate, since their fault-treatment mechanisms require a separated uplink/downlink per node in order to separate (and thus distinguish) each node contribution from the coupled signal.

Certainly, both CANcentrate and ReCANcentrate partially mitigate this problem as they bring the possibility of setting up an hybrid topology combining a bus and

a star topology. However, it would be interesting to further investigate if it is possible to design and implement CAN-compliant star topologies that use a single CAN cable for connecting a node to a given hub. In this sense, it would be necessary to find a mechanism that allows distinguishing each node's contribution by, for example, multiplexing in time the transmission of the nodes and the hub/s.

Design, implementation and dependability evaluation of further hub's fault-treatment mechanisms

In the context of this dissertation we focused on star-based solutions for CAN that are independent of the application. For this reason, the fault-treatment and fault-tolerance mechanisms of CANcentrate and ReCANcentrate basically deal with faults whose treatment does not require any knowledge about the application, i.e. with faults that manifest as the transmission of syntactically incorrect data.

However, it would be interesting to provide the hubs with mechanisms that allow them to treat and tolerate a wide range of faults that manifest from a semantical point of view. For example, it seems quite easy to include information within the hubs concerning the scheduling of the messages, so that they can detect and isolate babbling-idiot faults.

Design, implementation and dependability evaluation of time-triggered CAN-compliant star topologies

Maybe the main feature of ReCANcentrate is that both its hubs are coupled, so that they enforce a single communication domain that allows nodes to easily manage the replicated traffic as well as to tolerate faults. This characteristic is an advantage for event-triggered communication protocols such as CAN, since they do not provide any mechanisms to synchronize different channel replicas when they are used to transmit the same data in parallel.

However, this feature can also represent a disadvantage. Specifically, the hub coupling propagates errors generated by faults from one star to the other, thereby reducing the total bandwidth of the network. Fortunately, in time-triggered communication, e.g. in FTT-CAN, the synchronization among channels is inherently achieved due to the protocol's transmission schema and, thus, the hub coupling could be dispensable when using these protocols. Therefore, it would be interesting to investigate how to adapt ReCANcentrate in order to benefit from its error-containment and fault-tolerance capabilities while using a time-triggered communication schema.

Integration of CANcentrate and ReCANcentrate with other existing CAN-based dependability-related mechanisms

Apart from the error-containment and fault-tolerance limitations of CAN that are due to its simplex bus topology, it also presents other dependability limitations that discourage its use for highly-dependable systems; namely limited data consistency, limited support for node fault-tolerance, and lack of clock synchronization services [PPA+09]. Fortunately, there have been solutions proposed for overcoming these CAN limitations separately. This unveils propitious possibilities on the integration of our stars with those independent solutions with the goal of designing, implementing and validating a complete CAN-based infrastructure for supporting the execution of highly-dependable distributed control applications.

Further sensitivity analyses of the reliability of systems based on CAN, CANcentrate and ReCANcentrate

In spite of all the results presented in this document, the models we have proposed for measuring the reliability of systems relying on CAN, CANcentrate and ReCANcentrate can be further exploited to carry out additional sensitivity analyses. This is specially relevant for the case of ReCANcentrate, since its model allows configuring several architectural and functionality options for which we have assumed conservative values, and whose potential benefits on the system reliability could be an issue deserving of attention. For instance, it would be very valuable to assess how the reliability of a ReCANcentrate-based system can be improved by including more interlinks or by allowing nodes to continue communicating even when hubs become decoupled.

Quantitative comparison of the system reliability achievable by stars and other topologies different from a simplex bus

One of the objectives of the work presented in this dissertation was to validate the thesis's assertion that claims that it is possible to improve the reliability of CAN-based systems by changing its simplex bus to adequate star topologies. For this reason, the reliability analyses we carried out compare CANcentrate and ReCANcentrate with a simplex CAN bus.

However, it would be also interesting to quantitatively compare star topologies with other topologies. In particular, since ReCANcentrate includes redundancy, the next natural step would be to compare the system reliability achievable with this star with the one achievable with other redundant topologies such as replicated CAN buses.

Quantitative analysis of the performability of systems relying on CAN, CANcentrate and ReCANcentrate in the presence of transient faults.

Although transient faults affecting the communication subsystem cannot prevent nodes from communicating indefinitely, they negatively affect the system performability. Furthermore, these faults can even provoke a generalized failure. For instance, deadline violations provoke an overall failure in hard real-time systems. As already said, the impact of transient faults on dependability is, necessarily, application dependent, since deadline violations due to these faults strongly depend on the scheduling of the application-specific set of messages. Thus, it will be interesting to quantitatively compare the performability of the CAN bus, CANcentrate and ReCANcentrate under these faults for specific applications.

Quantitative assessment of the system reliability achievable by star topologies that are based on technologies other than CAN

To our best knowledge, in this dissertation we have provided the first formal (quantitative) comparison of the system reliability achievable with a simplex bus and stars taking into account different technological aspects, failure modes and error-containment and fault-tolerance coverages.

Another encouraging line of research is to perform this quantitative comparison in other field-bus communication technologies such as in TTP or FlexRay. In fact, as already mentioned, the models herein presented can be used for this purpose to some extent, thus being a good starting point to achieve this objective. Likewise, it would be also interesting to quantitatively compare the reliability of systems relying on CAN-stars and stars based on those alternative technologies.

Bibliography

[ABST03] Astrit Ademaj, Günther Bauer, Hakan Sivencrona, and Jan Torin. Evaluation of Fault Handling of the Time-Triggered Architecture with Bus and Star Topology. *IEEE International Conference on Dependable Systems and Networks (DSN 2003), San Francisco*, Jun. 2003.

[ACL89] J. Arlat, Y. Crouzet, and J.C. Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *Proceedings of the IEEE 19th Int. Symp. on Fault-Tolerant Computing. FTCS-19*, pages 348–355, Chicago, USA, June 1989.

[ADS03] Fulya Atiparmak, Berna Dengiz, and Alice E. Smith. Reliability Estimation Of Computer Communication Networks: ANN Models. In *Proceedings of the Eight IEEE International Symposium on Computers and Communication (ISCC 03)*, pages 1353–1358, June 2003.

[AL81] T. Anderson and P.A. Lee. *Fault Tolerance - Principles and Practice*. Prentice Hall, 1981.

[AM02] M. Abdollahi and A. Movaghar. Application of Stochastic Activity Networks on Network Modelling. *SoftCOM'02. 10th International Conference on Software, Telecommunications and Computer Networks, Split, Dubrovnik, Croatia*, 2002.

[Arn73] T. F. Arnold. The Concept of Coverage and Its Effect on the Reliability Model of a Repairable System. *IEEE Transactions on Computers*, 22(3):251–254, March 1973.

[Avi95] Algirdas Avižienis. Building dependable systems: How to keep up with complexity. *Special Issue of the IEEE 25th Int. Symp. Fault-Tolerant Compuitng. FTCS-25. Pasadena*, pages 4–14, June 1995.

[BAP05] M. Barranco, L. Almeida, and J. Proenza. ReCANcentrate: A replicated star topology for CAN networks. *ETFA 2005. 10^{th} IEEE International Conference on Emerging Technologies and Factory Automation, Catania, Italy*, 2005.

[BB03] I. Broster and A. Burns. An Analyzable Bus-Guardian for Event-Triggered Communication. In *Proceedings of the 24th Real-time Systems Symposium (RTSS)*, pages 410–419, Cancun, Mexico, Dec 2003. IEEE.

[BCS69] W. G. Bouricius, W. C. Carter, and P. R. Schneider. Reliability modeling techniques for self-repairing computer systems. In *Proceedings of the 1969 24th national conference*, pages 295–309, New York, NY, USA, 1969. ACM.

[Bel02] Ralf Belschner. FlexRay - Requirements Specification, 2002.

[BHN07] C. Braun, L. Havet, and N. Navet. NETCARBENCH: a benchmark for techniques and tools used in the design of automotive communication systems. In *Proceedings of the 7th IFAC International Conference on Fieldbuses and Networks in Industrial and Embedded Systems (FeT 2007)*, Toulouse, France, November 2007.

[BKS03] Günther Bauer, Hermann Kopetz, and Wilfried Steiner. The central guardian approach to enforce fault isolation in the time-triggered architecture. In *Proceedings of the The Sixth International Symposium on Autonomous Decentralized Systems (ISADS'03), Washington, DC, USA*, page 37. IEEE Computer Society, 2003.

[BMD93] Michael Barborak, Miroslaw Malek, and Anton Dahbura. The consensus problem in fault-tolerant computing. In *ACM Computing Surveys*, pages 171–220, June 1993.

[BPA06] M. Barranco, J. Proenza, and L. Almeida. First results of the assessment of the improvement of error containment achieved by CANcentrate. In *WFCS'06. IEEE Workshop on Factory Communication Systems, Torino, Italy*, 2006.

[BPRNA06] M. Barranco, J. Proenza, G. Rodríguez-Navas, and L. Almeida. An Active Star Topology for Improving Fault Confinement in CAN Networks. *IEEE Transactions on Industrial Informatics, vol. 2, issue 2*, 2:78–85, May 2006.

[Cao97] Feng Cao. Reliability Analysis of Partitioned Optical Passive Stars Networks. In *22nd Annual IEEE International Conference on Local Computer Networks (LCN'97)*, page 470, 1997.

[Car82] W.C. Carter. A time for reflection. In *Proceedings of the IEEE 12th Int. Symp. Fault-Tolerant Computing. FTCS-12. Santa Monica, California, USA*, June 1982.

[Cav05] Salvatore Cavalieri. Meeting Real-Time Constraints in CAN. *IEEE Transactions on Industrial Informatics*, pages 124–135, May 2005.

[CDV01] Gianluca Cena, Luca Durante, and Adriano Valenzano. A new CAN-like field network based on a star topology, July 2001.

[CFJ+91] Joshep A. Couvillion, Roberto Freire, Ron Johnson, W. Douglas Obal II, M. Akber Qureshi, Manish Rai, William H. Sanders, and Janet E. Tvedt. Performability Modelling with UltraSAN. *IEEE Software, vol. 8, issue 5*, pages 69–80, September 1991.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[CiAa] CiA. CAN data link layer. Technical report, CAN in Automation (CiA), Am Weichselgarten 26.

[CiAb] CiA. CAN physical layer. Technical report, CAN in Automation (CiA), Am Weichselgarten 26.

[Cor06] Relex Corporation. Relex Reliability Software, 2006.

[DLMSS08] M. Dehbashi, V. Lari, S.G. Miremadi, and M. Shokrollah-Shirazi. Fault Effects in FlexRay-Based Networks with Hybrid Topology. In *ARES 2008. Third International Conference on Availability, Reliability and Security, Barcelona, Spain*, March 2008.

[DOD95] DOD. *MIL-HDK-217F-2 Military Handbook, Reliability Prediction Of Electronic Equipment*. Department of Defense Washington DC, 1995.

[DT89] Joanne Bechta Dugan and Kishor S. Trivedi. Coverage modeling for Dependability Analysis of Fault-Tolerant Systems. *IEEE Transactions on Computers*, 38(5), June 1989.

[DZY09] Jianmin Duan, Liang Zhu, and Yongchuan Yu. Research on FlexRay communication of Steering-by-Wire system. In *IV 2009. IEEE Intelligent Vehicles Symposium, Xi'an, China*, June 2009.

[Fle05] FlexRay^{TM}. FlexRay Communications System - Protocol Specification, Version 2.1, 2005.

[FLS02] K. Fitzgerald, Shahram Latifi, and P. K. Srimani. Reliability Modeling and Assessment of the Star-Graph Network. *IEEE Transactions on Reliability*, 51(1):49–57, March 2002.

[FOFF04] J. Ferreira, A. Oliveira, P. Fonseca, and J. Fonseca. An Experiment to Assess Bit Error Rate in CAN. *Proceedings of 3rd International Workshop on Real-Time Networks, Catania, Italy*, 2004.

[Fre02] L.-B. Fredriksson. CAN for critical embedded automotive networks. *IEEE Micro, Special Issue on Critical Embedded Automotive Networks*, 22(4):28–35, July-August 2002.

[GAW09] Huaqun Guo, Jun Jie Ang, and Yongdong Wu. Extracting Controller Area Network data for reliable car communications. In *IV 2009. IEEE Intelligent Vehicles Symposium, Xi'an, China*, June 2009.

[GKT89] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proceedings of the IEEE 19th Int. Symp. on Fault-Tolerant Computing. FTCS-19*, pages 340–347, Chicago, USA, June 1989.

[Gmb91] Robert Bosch GmbH. CAN Specification, Version 2.0, 1991.

[HNNP02] Hans A. Hansson, Thomas Nolte, Christer Norström, and Sasikumar Punnekkat. Integrating Reliability and Timing Analysis of CAN-Based Systems. *IEEE Transactions on Industrial Electronics*, 49(6):1240–1250, December 2002.

[HPDS08] Brendan Hall, Michael Paulitsch, Kevin Driscoll, and Hakan Sivencrona. ESCAPE CAN Limitations. In *SAE 2007 Transactions Journal of Passenger Cars: Electronic and Electrical Systems, Detroit, USA*. Society of Automotive Engineers, USA, August 2008.

[HR09] C. Heller and R. Reichel. Enabling FlexRay for avionic data buses. In *DASC 2009. IEEE/AIAA 28th Digital Avionics Systems Conference, Orlando, USA*, October 2009.

[Inf02] Infineon(technologies). CAN-Transceiver TLE 6250, 2002.

[ISO93] ISO. ISO11898. Road vehicles - Interchange of digital information - Controller Area Network (CAN) for high-speed communication, 1993.

[ISO03a] ISO. ISO11898-1. Controller Area Network (CAN) – Part 1: Data link layer and physical signalling, 2003.

[ISO03b] ISO. ISO11898-2. Controller Area Network (CAN) – Part 2: High-speed medium access unit, 2003.

[IXX09] IXXAT. Innovative products for industrial and automotive communication systems, 2009.

[KG94] Hermann Kopetz and Günter Grunsteidl. TTP - A Protocol for Fault-Tolerant and Real-Time Systems. *IEEE COMPUTER*, January 1994.

[KHJN03] M. Törngren K. H. Johansson and L. Nielsen. Vehicle Applications of Controller Area Network. Technical report, Department of Signals, Sensors and Systems, Royal Institute of Technology, Stockholm, Sweden; Department of Electrical Engineering, Linkoping University, Sweden, 2003.

[KNM90] David J. Klinger, Yoshinao Nakada, and Maria A. Menendez. *ATT Reliability Manual*. Van Nostrand Reinhold, 1990.

[Kop97] Hermann Kopetz. Dependability. In Real-Time Systems: Design Principles for Distributed Embedded Applications. *Real-Time Systems. Engineering and Computer Science. Chapter 2.4. Kluwer Academic Publishers, Boston, Dordrecht, London*, pages 39–42, 1997.

[Kop02] H. Kopetz. Fault Containment and Error Detection in TTP/C and FlexRay. Research Report 23, Vienna University Of Technology, TU Wien, August 2002.

[Kop03] H. Kopetz. Time-Triggered Protocols for Safety-Critical Applications. Presentation, March 2003.

[Lap92] Jean-Claude Laprie. *Dependability: Basic Concepts and Terminology*. Springer-Verlag Wien New York, 1992.

[Lap01] J.C. Laprie. Fundamental concepts of dependability. Technical report, Technical Report 739, University of Newcastle upon Tyne, School of Computing Science, 2001.

[Lat05] Elizabeth Ann Latronico. *Reliability Validation of Group Membership Services for X-by-Wire Protocols*. Carnegie Mellon University, Electrical and Computer Engineering Department, May 2005.

[LJ90] Nanchang Lin and Charles B. Silio Jr. A Reliability Comparison of Single and Double Rings. In *INFOCOM 1990. IEEE International Conference on Computer Communications. San Francisco. USA*, pages 504–511, June 1990.

[Mee95] Victor Meeldijk. *ELECTRONIC COMPONENTS, Selection and Application Guidelines*. Wiley-Interscience Publication, JOHN WILEY & SONS, INC., 1995.

[Mic04] PIC18FXX8 Data Sheet - 28/40-Pin High-Performance, Enhanced Flash Microcontrollers with CAN Module, 2004.

[MK05] J. Morris and P. Koopman. Representing Design Tradeoffs in Safety-Critical Systems. *WADS. Workshop on Architecting Dependable Systems, St. Louis, Missouri, USA*, 2005.

[MSH08] P. Milbredt, A. Steininger, and M. Horauer. Automated Testing of FlexRay Clusters for System Inconsistencies in Automotive Networks. In *DELTA 2008. 4^{th} IEEE International Symposium on Electronic Design, Test and Applications, Hong Kong, China*, January 2008.

[MT95] M. Mahotra and K. S. Trivedi. Dependability Modeling Using Petri-Nets. *IEEE Transactions on Reliability*, 44(3), September 1995.

[NNH05] Thomas Nolte, Mikael Nolin, and Hans A. Hansson. Real-Time Server-Based Communication with CAN. *IEEE Transactions on Industrial Informatics*, pages 192–201, August 2005.

[NSSLW05] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert. Trends in Automotive Communication Systems. *Proceedings of the IEEE*, 93(6), 2005.

[Pau04] Ondrej Pauk. Powering Connectivity in Todays Automobiles. *Power Electronics Technology magazine*, 2004.

[PdS04] P. Portugal and A. da Silva. A Framework for Dependability Evaluation of Fieldbus Networks. *WFCS'04. IEEE Workshop on Factory Communication Systems, Vienna, Austria*, 2004.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1981.

[PF04] Juan R. Pimentel and Jose A. Fonseca. FlexCAN: A Flexible Architecture for Highly Dependable Embedded Applications. *The 3rd International Workshop on Real-Time Networks, Catania, Italy*, July 2004.

[PHI00] PHILIPS. Data sheet - PCA82C250 - CAN controller interface - Product Specification, 2000.

[PMJ00] J. Proenza and J. Miro-Julia. MajorCAN: A modification to the Controller Area Network to achieve Atomic Broadcast. *IEEE Int. Workshop on Group Communication and Computations, Taipei, Taiwan*, 2000.

[Pol96] S. Poledna. System model and terminology. In Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism, Real-Time Systems, chapter 3. In *Engineering and Computer Science*, pages 21–30. Kluwer Academic Publishers, Boston, Dordrecht, London, 1996.

[Pow92] D. Powell. Failure Mode Assumptions and Assumption Coverage. *Digest of Papers of the IEEE 22th Int. Symp. Fault-Tolerant Computing FTCS-22, Boston, Massachusetts-USA*, pages 386–395, July 1992.

[PPA+09] J. Pimentel, J. Proenza, L. Almeida, G. Rodríguez-Navas, M. Barranco, and J. Ferreira. *Dependable Automotive CAN Networks*. Handbook on Automotive Embedded Systems. CRC Press. Edited by Nicolas Navet and Françoise Simonot-Lion, 2009.

[Pro07] Julián Proenza. *Ph. D. Thesis. RCMBnet: A Distributed Hardware and Firmware Support for Software Fault Tolerance*. Universitat de les Illes Balears, Departament de Ciències Matemàtiques i Informàtica., January 2007.

[RD88] G. A. Ray and J. J. Dunsmore. Reliability of Network Topologies. In *INFOCOM 1988. IEEE International Conference on Computer Communications. New Orleans, USA.*, pages 842–850, March 1988.

[Ruc94] Michael Rucks. Optical layer for CAN. *1st International CAN Conference*, November 1994.

[Rus03] J. Rushby. A Comparison of Bus Architectures for Safety-Critical Embedded Systems. Contractor report, SRI International, Menlo Park, California, 2003.

[RVA+98] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. *FTCS-28, The 28th International Symposium on Fault-Tolerant Computing, Munich, Germany*, 1998.

[RVA99] José Rufino, Paulo Veríssimo, and Guilherme Arroz. A Columbus' Egg Idea for CAN Media Redundancy. *FTCS-29. The 29th International Symposium on Fault-Tolerant Computing, Winconsin, USA*, June 1999.

[SFF06] V.F. Silva, J.C. Ferreira, and J.A. Fonseca. Dynamic Topology Management in CAN. In *ETFA 2006. 11^{th} IEEE International Conference on Emerging Technologies and Factory Automation, Prague, Czech Republic*, 2006.

[Sho02] Martin L. Shooman. *Reliability of Computer Systems and Networks*. John Wiley & Sons, Inc., 605 Third Avenue, New York, USA, 2002.

[SJA00] DATA SHEET SJA1000 Stand-alone CAN controller, 2000.

[SoT04] W. Sanders and The Board of Trustees. Moebius User Manual Version 1.6.0, 2004.

[STP96] Robin A. Sahner, Kishor S. Trivedi, and Antonio Puliafito. *Performance and Reliability Analysis of Computer Systems*. Kluwer Academic Publisher, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061, USA, 1996.

[TMGT93] L. Tomek, V. Mainkar, R. Geist, and K. Trivedi. Reliability Modeling of Life-Critical, Real-Time Systems. *Procedings of the IEEE*, 1993.

[Wu02] N. Eva Wu. Reliability Analysis for AFTI-F16 SRFCS using ASSIST and SURE. In *Proceedings of American Control Conference*, pages 4975–4800, 2002.

[Wu04] N. Eva Wu. Coverage in fault-tolerant control. *ELSEVIER*, 40(4):537–548, April 2004.

[X E04] X Engineering Software Systems Corporation. *XSA-3S1000 Board V1.0 User Manual*, 2004.

Index

ACK bit sent by a receiving node in a CAN frame, 30
ACK check mechanism of CAN, 32
ACKnowledge (ACK) delimiter of a CAN frame, 30
ACKnowledge (ACK) error in CAN, 32
ACKnowledge (ACK) field of a CAN frame, 30
ACKnowledge (ACK) slot of a CAN frame, 30
active error flag of a CAN error frame, 33
active error frame of CANcentrate, 66
active fault, 14
active star coupler, 44
active state of a CANcentrate hub port, 71
age-dependent failure rate, 16
agreement mechanisms, 21
an out of the fault model fault, 19
application-specific fault tolerance, 20
arbitration field of a CAN frame, 30, 31
arbitration misunderstanding in CANcentrate, 106
assumption coverage, 19
authentification detectable Byzantine failure, 17
availability, 14

babbling-idiot fault, 36
BFC Manager Module of an Enabling / Disabling Unit, 64, 70
bit error in CAN, 32
bit synchronization mechanism of CAN, 28
Bit-Flipping Counter (BFC) of an Enabling / Disabling Unit, 64
Bit-Flipping Detection Counter (BFDC) of the CANcentrate hub, 107
Bit-Flipping Detection Threshold (BFDT) of the CANcentrate hub, 107
bit-flipping fault, 36
Bit-Flipping Threshold (BFT) of an Enabling / Disabling Unit, 65
bit-wise arbitration mechanism of CAN, 31
bitStuffWaited signal of the CANcentrate hub, 66
bus guardian, 41
bus-off CAN node, 32
bus-off state of a CAN node, 35
Byzantine or arbitrary failure, 17

CAN base frame format (CAN 2.0 A), 29
CAN bus-free occurrence, 35
CAN extended frame format (CAN 2.0 B), 29
CANcentrate, 49
CANivete board of a CANcentrate node prototype, 117
Columbus Egg Idea, 229
common-mode failures, 21, 40, 41
contention in CAN, 30

control field of a CAN frame, 30
COTS component, 5
coupled signal (B_0) of the hub of CANcentrate, 52
Coupler Module of the CANcentrate hub, 52
Coupler Module of the ReCANcentrate hub, 235
coverage process of the model of reliability of a system that relies on CAN, CANcentrate or ReCANcentrate, 156, 162
coverage submodel of the model of reliability of a system that relies on CAN, CANcentrate or ReCANcentrate, 157, 163
crash failure, 18
CRC check mechanism of CAN, 32
CRCPassed signal of the CANcentrate hub, 67
Current State (CS) signals of the CANcentrate hub, 54, 66
current state of the resultant frame in CANcentrate, 54
Cyclic Redundancy Code (CRC) delimiter of a CAN frame, 30
Cyclic Redundancy Code (CRC) error in CAN, 32
Cyclic Redundancy Code (CRC) field of a CAN frame, 30

daisy chain configuration of a CAN network, 134
data consistency, 21, 33
Data field of a CAN frame, 30
data frame of CAN, 29
data frame of CANcentrate, 66
Data Length Code (DLC) of a CAN frame, 30
DBC Manager Module of an Enabling / Disabling Unit, 64, 68
dependability, 13
disabled state of a CANcentrate hub port, 72
Dominant Bit Counter (DBC) of an Enabling / Disabling Unit, 64
Dominant Bit Threshold (DBT) of an Enabling / Disabling Unit, 65, 68
dominant value of a bit in CAN, 28
dominant/recessive transmission property of CAN, 28
dormant fault, 14
downlink of a CANcentrate network, 51
duplicate in a replicated channel, 228

Enabling/Disabling signal (ED_i) of the ReCANcentrate hub, 237
Enabling/Disabling signal (ED_i) of the CANcentrate hub, 55
Enabling/Disabling Unit (*Ena/Dis*) of the CANcentrate hub, 55
Enabling/Disabling Unit (*Ena/Dis*) of the ReCANcentrate hub, 237
End Of Frame (EOF) field of a CAN frame, 30
entity of a system relying on CAN / CANcentrate / ReCANcentrate, 136
error, 14
error containment, 1, 21
error delimiter of a CAN error frame, 33
error detection, 19
Error Flag Generator Module of the CANcentrate hub, 55
error flag of a CAN error frame, 33
error frame of CAN, 32
error globalization in CAN, 33
error processing, 19
error propagation, 21
error recovery, 19
error types detectable in CAN, 31

error-active CAN node, 32
error-active state of a CAN node, 34
error-containment coverage, 148
error-containment region, 21
error-containment region in CANcentrate, 49, 76
error-detection mechanisms of CAN, 31
error-passive CAN node, 32
error-passive state of a CAN node, 35
error-signaling mechanisms of CAN, 32
ESCAPE CAN star topology, 45
evaluator submodel of the model of reliability of a system that relies on CAN, CANcentrate or ReCANcentrate, 157
event counter of an Enabling / Disabling Unit, 64
extra delay of the propagation of the electrical signal in CANcentrate, 60

fail-silent failure, 19
fail-uncontrolled failure, 17, 36
failure, 14
failure mode, 16
failure mode assumption coverage, 19, 139
failure rate, 15
failure semantics, 18
fault, 14
fault confinement, 42
fault diagnosis, 20
fault independence, 21
fault injection, 23
fault model, 19
fault passivation, 20
fault tolerance, 19
fault treatment, 19
fault-containment region, 21
fault-tolerant/accepting system reliability (FT/AR), 131
fault-tolerant/accepting systems (FT/A systems), 130
Fault-Treatment Module of the CANcentrate hub, 53
Fault-Treatment Module of the ReCANcentrate hub, 237
Faulty node of the CANcentrate prototype, 120
faulty port in CANcentrate, 49, 54
FlexCAN, 228
format error in CAN, 32
FPGA, 116
frame check mechanism of CAN, 32
frameField vector of signals of the CANcentrate hub, 67

hazard rate, 16
hub core of the CANcentrate prototype, 116
Hub Enabling/Disabling Unit ($HubEna/Dis$) of the ReCANcentrate hub, 237
hub of a star topology, 42
hybrid topology using CANcentrate, 58

ideal star, 43
IDentifier Extension (IDE) bit of a CAN frame, 30
identifier of a CAN frame, 30
Idle state of a CAN channel, 30
idle state of a CANcentrate hub port, 71
impulse reward variable of a SANs model, 154
in-bit response property of CAN, 28
inconsistency scenarios of CAN, 34
inconsistent data, 21
inconsistent failure, 17
incorrect computation failure, 17
independent faults, 20
Input/Output Module of the CANcentrate hub, 52

Input/Output Module of the ReCANcentrate hub, 237
inter frame of CANcentrate, 67
interlink of a ReCANcentrate network, 231
Intermission Frame Space (IFS) of CAN, 30
intermittent fault, 14

Join primitive of a SAN, 154

k-severe failure of communication, 5, 24

leading transmitter in CAN, 28
link of a CANcentrate network, 51
link of a ReCANcentrate network, 231
link of a star topology, 42
local error, 35

malicious failure, 17
manager module of an Enabling / Disabling Unit, 64
Markov Chain, 23
masquerading failure, 17
MIL-HDBK-217 model, 138
mission time of a system, 190
model checker, 23
model checking, 23
monitoring mechanism of CAN, 32

NACK Manager Module of an Enabling / Disabling Unit, 64, 68
network partition, 37
network partition fault, 37
Non-Acknowledge Counter (NACKC) of an Enabling / Disabling Unit, 64
Non-Acknowledge Threshold (NACKT) of an Enabling / Disabling Unit, 65
non-fault-tolerant/accepting system reliability (NFT/AR), 131
non-fault-tolerant/accepting systems (NFT/A systems), 130

ofm: out-of-fault model, 139
omission failure, 18
omission in a replicated channel, 228
out-of-fault-model failure mode, 139
overload conditions in CAN, 35
overload delimiter of an overload CAN frame, 35
overload flag of an overload CAN frame, 35
overload frame of CAN, 35
overload frame of CANcentrate, 66

passive error flag of a CAN error frame, 33
passive error frame of CANcentrate, 67
passive star coupler, 43
penalization policy of the BFC Manager, 108
performability, 14
performance failure, 18
permanent fault, 14
Petri Net, 23
Physical Layer Module of the CANcentrate hub, 54
point of k-severe failure of communication, 5, 24
primary error in CAN, 34

qualitative evaluation, 23
quantitative evaluation, 23

rate reward variable of a SANs model, 154
ReCANcentrate, 227
receiving node in a CAN / CANcentrate network, 76
Reception Error Counter (REC) of a CAN node, 34

recessive value of a bit in CAN, 28
redundancy management, 20
region of the model of reliability of a system that relies on CAN, CANcentrate or ReCANcentrate, 160
reintegration policy of CANcentrate, 65, 70
related faults, 20
reliability, 14
reliability function, 15
remote frame of CAN, 29
remote frame of CANcentrate, 66
Remote Transmission Request (RTR) bit of a CAN frame, 30
Rep primitive of a SAN, 154
replicated bus topology, 40
reserved bit (R0) of a CAN frame, 30
resultant frame in CANcentrate, 53
resultant frame in ReCANcentrate, 237
reward model, 154
reward variable of a SANs model, 154
role of a node in a CAN / CANcentrate network, 76
Rx_CAN Module of the CANcentrate hub, 54

severe failure, 24
severe point of failure, 24
single broadcast domain enforced by ReCANcentrate, 231
single point of failure, 5, 20, 24
SMART-1, 229
spatial redundancy, 20
spatial-proximity failures, 21, 40
star diameter, 59
star topology, 42
StarCAN star topology, 44
starLink board of a CANcentrate node prototype, 117
Start Of Frame (SOF) field of a CAN frame, 30
Stochastic Activity Network (SAN) formalism, 128, 153
stopping or fail-stop failure, 18
structure state model, 154
stuck-at fault, 36
stuck-at-dominant fault, 36
stuck-at-recessive fault, 36
stuff bit in CAN, 31
stuff error in CAN, 32
stuff rule check mechanism of CAN, 32
stuff rule of CAN, 31
sublink of a ReCANcentrate network, 231
synchronization at bit level in a CAN / CANcentrate network, 54
synchronization at frame level in a CAN / CANcentrate network, 54
sysFauTolcov, 146
systematic fault tolerance, 20

Tellcordia methodology, 138
temporal redundancy, 20
Threshold Control Module of an Enabling / Disabling Unit, 65
Time To Failure (TTF), 15
Time To Failure distribution, 15
timing failure, 18
transient fault, 14
Transmission Error Counter (TEC) of a CAN node, 34
transmitting node in a CAN / CANcentrate network, 76

uplink of a CANcentrate network, 51

valueBitStuff signal of the CANcentrate hub, 66
VHDL, 116

wired-AND function of the medium of CAN, 28