Simulation concurrente de fautes comportementales pour des systèmes à événements discrets : Application aux circuits digitaux

Laurent Capocchi

▶ **To cite this version:**

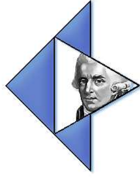
Laurent Capocchi. Simulation concurrente de fautes comportementales pour des systèmes à événements discrets : Application aux circuits digitaux. Modélisation et simulation. Université Pascal Paoli, 2005. Français. NNT : . tel-00165440

HAL Id: tel-00165440
https://theses.hal.science/tel-00165440

Submitted on 26 Jul 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



UNIVERSITÉ DE CORSE – PASQUALE PAOLI

U.F.R. SCIENCES ET TECHNIQUES

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE CORSE

ÉCOLE DOCTORALE ENVIRONNEMENT ET SOCIÉTÉ

Discipline : Sciences pour l'Environnement

Spécialité : Informatique

présentée par

M. Laurent CAPOCCHI

Simulation concurrente de fautes comportementales pour des systèmes à événements discrets : *Application aux circuits digitaux*

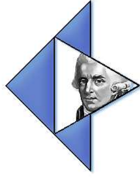
---

sous la direction du Professeur

Paul-Antoine Bisgambiglia

soutenue publiquement le 25 novembre 2005 devant le jury composé de :

Rapporteurs :	M. Norbert GIAMBIASI, *Professeur, Université d'Aix-Marseille III*
	M. Gabriel A. WAINER, *Professeur, Université de Carleton*
Examinateurs :	M. Gérard CAPOLINO *Professeur, Université de Picardie*
	M. Paul-Antoine BISGAMBIGLIA, *Professeur, Université de Corse*
	M. Jean-François SANTUCCI, *Professeur, Université de Corse*
	M. Dominique FEDERICI, *MCF*, *Université de Corse*
	M. Antoine AIELLO, *MCF HDR, Université de Corse*
Invitée :	Mme. Claudia FRYDMAN, *Professeur, Université d'Aix-Marseille III*



UNIVERSITÉ DE CORSE – PASQUALE PAOLI

U.F.R. SCIENCES ET TECHNIQUES

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE CORSE

ÉCOLE DOCTORALE ENVIRONNEMENT ET SOCIÉTÉ

Discipline : Sciences pour l'Environnement

Spécialité : Informatique

présentée par

M. Laurent CAPOCCHI

Simulation concurrente de fautes comportementales pour des systèmes à événements discrets : *Application aux circuits digitaux*

---

sous la direction du Professeur

Paul-Antoine Bisgambiglia

soutenue publiquement le 25 novembre 2005 devant le jury composé de :

Rapporteurs :	M. Norbert GIAMBIASI, *Professeur, Université d'Aix-Marseille III*
	M. Gabriel A. WAINER, *Professeur, Université de Carleton*
Examinateurs :	M. Gérard CAPOLINO *Professeur, Université de Picardie*
	M. Paul-Antoine BISGAMBIGLIA, *Professeur, Université de Corse*
	M. Jean-François SANTUCCI, *Professeur, Université de Corse*
	M. Dominique FEDERICI, *MCF*, *Université de Corse*
	M. Antoine AIELLO, *MCF HDR, Université de Corse*
Invitée :	Mme. Claudia FRYDMAN, *Professeur, Université d'Aix-Marseille III*

Remerciements

Je tiens avant tout à exprimer ma profonde reconnaissance à Monsieur Paul-Antoine Bisgambiglia, qui a assuré la direction de cette thèse, pour ses conseils judicieux, mais surtout pour le soutien, l'aide scientifique et morale qu'il m'a toujours apporté au cours de mes années d'études. Qu'il trouve ici l'expression de mon profond respect. J'adresse également mes remerciements à Monsieur Jean-François Santucci, pour m'avoir accueilli au sein de son équipe et pour m'avoir exprimé son soutien tout au long de mon travail de thèse.

Des remerciements très spéciaux pour Monsieur Dominique Federici pour toutes les discussions fructueuses, pour les critiques constructives qui ont contribué au contenu de cette thèse, ainsi que pour toute son amitié et son encouragement. Je le remercie très chaleureusement de m'avoir fait part de son expérience.

Des remerciements également très spéciaux pour Monsieur Fabrice Bernardi, avec qui j'ai fait mes premières expériences dans le domaine de la recherche, pour le travail que nous avons mené très étroitement ensemble, et pour ces méthodes rigoureuses de travail qu'il a su m'inculquer. Je le remercie également pour toute son amitié sincère et pour sa confiance en moi depuis le début de cette thèse.

Je tiens à remercier Monsieur Christophe Paoli et Mademoiselle Marie-Laure Nivet pour l'intérêt qu'ils ont manifesté à l'égard de mon travail, et aussi pour les conseils et leurs soutien au cours de ce travail.

Je tiens à remercier Monsieur Norbert Giambiasi et Monsieur Gabriel Wainer qui ont eu la gentillesse de bien vouloir être rapporteur de cette thèse. Je les remercie très sincèrement pour leurs commentaires et remarques précieuses, et également pour le temps qu'ils m'ont accordé.

Un très grand merci à mes parents Berthe et Charles-André Capocchi, et à Monsieur Ange De Cicco. Sans leur soutien je n'en serais pas arrivé là.

Da sinu à chi e radiche tireranu

suchju ind'è u terraghjone anticu

e talle di sta cultura crisceranu.

Paulu Santu Parigi

Table des matières

1 Introduction générale 1
1.1 Problématique et objectif 1
1.2 Aperçu de la démarche 2
1.3 Organisation du document 4

2 État de l'art 6
2.1 Le test de circuits 6
2.1.1 Le test dans le flot de conception 7
2.1.2 Les notions de base du test 9
2.1.2.1 Fautes et modèles de fautes 9
2.1.2.2 Vecteurs et séquences de test 10
2.1.2.3 La simulation de fautes 11
2.1.3 Le test à haut niveau 14
2.1.3.1 Le modèle de fautes de haut niveau 15
2.1.3.2 La simulation de fautes à haut niveau 16
2.1.4 Conclusion 17
2.2 La Simulation Comparative et Concurrente 18

2.2.1 Historique 18
2.2.2 Méthodologie de la SCC 19
2.2.3 Propriétés et avantages de la SCC 21
2.2.4 La Simulation de Fautes Concurrente 22
2.2.5 Conclusion 24
2.3 Le formalisme DEVS 25
2.3.1 La modélisation DEVS 26
2.3.1.1 La notion de modèle atomique 26
2.3.1.2 La notion de modèle couplé 29
2.3.2 La simulation DEVS 30
2.3.2.1 Algorithme d'un composant Simulateur 32
2.3.2.2 Algorithme d'un composant Coordinateur 34
2.3.3 Conclusion 37
2.4 Conclusion 38

3 Le formalisme BFS-DEVS 39
3.1 Introduction 39
3.2 Modèle de fautes comportementales 42
3.3 Le formalisme BFS-DEVS 43
3.3.1 La modélisation BFS-DEVS 43
3.3.2 La simulation BFS-DEVS 44
3.3.2.1 Protocole de communication 44
3.3.2.2 Modification algorithmique du composant simulateur 45
3.3.2.3 Modification algorithmique d'un composant coordinateur . . . 46
3.4 Mécanisme de Simulation BFS-DEVS 48
3.5 Conclusion 52

4 Le formalisme BFS-DEVS appliqué aux circuits digitaux 53
4.1 La modélisation BFS-DEVS 54

4.1.1 Le Langage VHDL 54
4.1.1.1 Bref historique des langages de description du matériel 54
4.1.1.2 Bref notion du langage VHDL 55
4.1.1.3 Sous ensemble étudié : les descriptions VHDL comportementales 56
4.1.2 Modèle de fautes proposé 57
4.1.3 Transformation VHDL/BFS-DEVS 60
4.2 La simulation concurrente BFS-DEVS 63
4.2.1 Définitions spécifiques aux descriptions VHDL comportementales 63
4.2.2 Le cycle de simulation VHDL 66
4.2.2.1 Le cycle de simulation sain 66
4.2.2.2 Le cycle de simulation de fautes 66
4.2.3 La technique de propagation des listes de fautes LF_i 73
4.2.3.1 Propagation intra-processus 73
4.2.3.2 Propagation inter-processus 76
4.3 Exemple : Le registre 8 bits 80
4.3.1 Liste des fautes 80
4.3.2 Cycle d'initialisation C_0 81
4.3.3 Cycle symbolique C_0^{+1} 86
4.4 Conclusion 89

5 Modélisation orientée objet 90
5.1 Approche de description statique 90
5.1.1 Le paquetage "*DEVSModels*" 91
5.1.2 Le paquetage "*Domain*" 92
5.1.3 Le paquetage "*X BFS-DEVS Library*" 93
5.1.4 Le paquetage *"VHDL BFS-DEVS Library"* 94
5.1.5 Le paquetage *"DEVSSimulator"* 96
5.2 Approche de description dynamique 96
5.2.1 La classe *"Generator"* 97

5.2.2 La classe "*Assignment*" . . . 98
5.2.3 La classe *"Conditional"* . . . 100
5.2.4 La classe "*Junction*" . . . 102
5.2.5 La classe "*ProcessEngine*" . . . 104
5.2.6 Gestion des messages d'un arbre de simulation . . . 107
5.3 Conclusion . . . 109

6 Expériences et résultats . . . 110
6.1 Les benchmarks ITC'99 . . . 110
6.2 Critère de couverture de fautes . . . 111
6.3 Validation du prototype BFS-DEVS . . . 112
6.3.1 Architecture du prototype . . . 113
6.3.2 Modification du code VHDL . . . 115
6.4 Simulation BFS-DEVS des séquences de test RAGE . . . 116
6.4.1 Le générateur automatique de séquences de test RAGE . . . 116
6.4.2 Simulation BFS-DEVS . . . 119
6.4.3 Équivalence des modèles de fautes . . . 122
6.5 Conclusion . . . 124

7 Conclusion et perspectives . . . 125
7.1 Conclusion générale . . . 126
7.2 Perspectives de recherche . . . 128

8 Annexes . . . 130
8.1 Le composant BFS-DEVS *"Generator"* . . . 130
8.1.1 Rôle dans la simulation de fautes . . . 131
8.1.2 Spécifications BFS-DEVS . . . 136
8.2 Le composant BFS-DEVS *"Conditional"* . . . 139
8.2.1 Rôle dans la simulation de fautes . . . 140
8.2.2 Spécifications BFS-DEVS . . . 146

8.3 Le composant BFS-DEVS *"Assignment"* 148
8.3.1 Rôle dans la simulation de fautes 149
8.3.1.1 Détermination de la liste L_O 149
8.3.1.2 Évaluation de l'influence de L 150
8.3.2 Spécifications BFS-DEVS 151
8.4 Le composant *"Junction"* 152
8.4.1 Rôle dans la simulation de fautes 153
8.4.2 Spécifications BFS-DEVS 157
8.5 Le composant BFS-DEVS *"ProcessEngine"* 158
8.5.1 Rôle dans la simulation de fautes 160
8.5.1.1 Redirection des messages fautifs : propagation inter-processus 160
8.5.1.2 L'observabilité des fautes 162
8.5.2 Spécifications BFS-DEVS 166

Bibliographie 169

Liste des publications 176

Liste des figures 178

Liste des tableaux 181

Liste des algorithmes 183

CHAPITRE 1

Introduction générale

L'étude présentée dans ce mémoire concerne la définition d'un simulateur concurrent pour des systèmes à événements discrets. L'objectif principal de nos travaux est de donner la possibilité de simuler de manière concurrente plusieurs expériences en une seule exécution. Le domaine d'application permettant de valider notre approche est celui de la simulation de fautes concurrente pour le test des circuits digitaux décrits à haut niveau d'abstraction.

1.1 Problématique et objectif

Au cours des 40 dernières années, la **simulation à événements discrets** a commencé à remplacer l'expérimentation physique des systèmes. En effet, une alternative à l'élaboration souvent coûteuse *(en temps et en moyens techniques)* d'une expérimentation sur un système physique réside dans la *modélisation en vue de le simuler*. La modélisation permet de donner une représentation facilement manipulable et réutilisable des systèmes pouvant ainsi être rapidement et indéfiniment simulés. Cependant, à partir de ces modèles, il est impossible de réaliser plusieurs traitements simultanés et les solutions consistent soit à paralléliser les algorithmes *(traitement en*

parallèle) soit à traiter une expérience à la fois *(traitement en série)*. Ces techniques deviennent fastidieuses du point de vue de l'implémentation des algorithmes dans le cas des traitements en parallèle ou gourmandes en temps d'exécution dans le cas des traitements en série. Dans tous les cas l'*analyse* et l'*observabilité* des résultats sont difficiles. Afin de dépasser ces limitations la **Simulation Comparative et Concurrente *(SCC)*** s'avère être une solution adaptée car elle permet d'effectuer plusieurs simulations d'un système en une seule exécution.

Une des premières applications de la SCC est la **Simulation de Fautes Concurrente *(SFC)*** pour *le test des circuits digitaux* en vue de leur conception. Le test de circuits digitaux permet de rendre compte d'éventuels défauts physiques pouvant induire un comportement irrégulier appelé *faute*. L'augmentation de la complexité et du nombre de portes logiques des circuits digitaux a amené les chercheurs à développer des outils de tests à des niveaux d'abstractions plus élevés. L'obtention d'un simulateur de fautes concurrent de haut niveau travaillant uniquement sur des descriptions matérielles nécessite l'implémentation des algorithmes de la SFC. Cependant, les particularités des langages de description matérielle ne permettent pas une intégration simple de ces algorithmes. Afin de simplifier cette tâche, nous proposons une modélisation des circuits digitaux plus adaptée à l'implémentation des algorithmes de la SFC. Cette nouvelle représentation est obtenue grâce à l'utilisation d'un formalisme permettant la modélisation et la simulation des systèmes à événements discrets.

L'objectif de nos travaux est donc de proposer un simulateur de fautes concurrent appliqué au domaine du test de circuits digitaux décrits à haut niveau en s'appuyant sur un formalisme de spécification des systèmes à événements discrets.

1.2 Aperçu de la démarche

La démarche mise en place pour d'atteindre l'objectif fixé repose sur les étapes suivantes :

1. **L'étude des algorithmes de la SFC** [Agrawal, 1981] avec la technique de *propagation de liste de fautes*.
2. **L'étude du formalisme DEVS *(Discrete EVent Specification)*** [Zeigler et al., 2000].

3. **L'intégration des algorithmes de la SFC à l'intérieur du formalisme DEVS** dans un nouveau formalisme *BFS-DEVS (Behavioral Fault Simulation-Discrete EVent Specification).*

4. **L'étude des représentations "textuelles"** des circuits digitaux décrits à l'aide d'un sous-ensemble comportemental du langage *VHDL (Very high speed integrated circuits Hardware Description Language) [sta, 2000, Ghosh, 2000].*

5. **L'utilisation du formalisme BFS-DEVS pour transformer** des descriptions comportementales VHDL en des *réseaux de modèles BFS-DEVS.*

Ces étapes, visibles sur la figure 1.1, permettent de construire le simulateur BFS-DEVS.

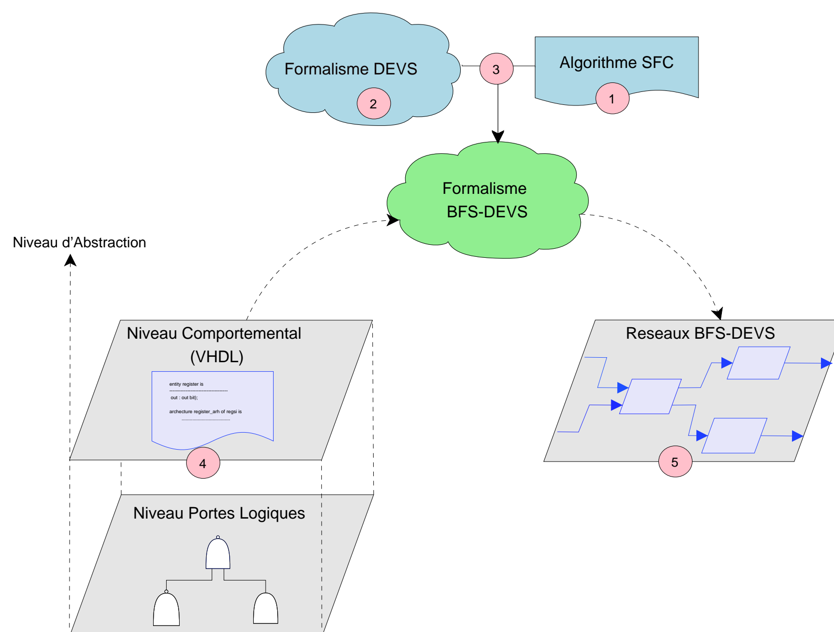


FIG. 1.1: *Aperçu de la démarche.*

Le formalisme DEVS a été introduit par le professeur B.P Zeigler au début des années 70. Il permet une modélisation *modulaire* et *hiérarchique* des systèmes à événements discrets et peut être vu, d'après le professeur H. Vangheluwe, comme le dénominateur commun des formalismes à événements discrets les plus utilisés [Vangheluwe et al., 2002]. Ce formalisme nous permet de transformer une description VHDL en une interconnexion de modèles comportementaux correspondant à chacune des instructions du circuit. A partir de ces modèles il est possible de représenter l'instruction sous-jacente afin de simuler son comportement normal ou irrégulier *(appelé faute)*,

ce qui est difficile à partir d'une description "textuelle". Une autre propriété importante du formalisme DEVS est qu'il permet de simuler ces modèles de manière automatique. Par conséquent, simuler des fautes comportementales dans un circuit digital à partir de sa description VHDL revient à simuler les comportements fautifs des modèles DEVS du réseau associé. Cependant, le formalisme DEVS ne possède aucun algorithme permettant d'accomplir une simulation de comportements fautifs de manière concurrente. Nous proposons donc d'intégrer à DEVS des algorithmes spécifiques afin de définir un nouveau formalisme capable de simuler de manière concurrente des fautes comportementales au sein de systèmes tels que les circuits digitaux.

1.3 Organisation du document

Ce document est organisé en cinq chapitres. Le **premier chapitre**, intitulé **"Etat de l'art"**, nous permet de définir les concepts utilisés. Dans une première partie, consacrée au *"Test de circuits digitaux"*, nous justifions la position et l'importance de la simulation de fautes dans le processus de conception d'un circuit. Nous expliquons plus précisément l'évolution du domaine de la conception des circuits digitaux amenant la simulation de fautes à évoluer vers des niveaux d'abstraction plus élevés. Dans une deuxième partie, nous donnons une définition de *"La Simulation Comparative et Concurrente"* et nous expliquons sa méthodologie en mettant en évidence ses avantages ainsi que ses propriétés. Nous introduisons ensuite la forme la plus répandue de la SCC : la SFC appliquée au domaine des circuits digitaux décrits au niveau portes logiques. La troisième partie définit *"Le formalisme DEVS"* en séparant deux vues complémentaires : la modélisation et la simulation des systèmes à événements discrets. Nous expliquons de quelle manière il est possible de représenter un système en utilisant les notions de hiérarchie et de modularité offertes par le formalisme DEVS. Enfin, nous montrons comment à partir de ces modèles, il est possible de générer automatiquement les algorithmes de simulation associés.

Dans le **deuxième chapitre**, intitulé **"Le formalisme BFS-DEVS"**, nous présentons l'approche générique nous permettant de simuler de manière concurrente des comportements fautifs quel que soit le domaine d'application. Nous montrons en particulier comment grâce au forma-

lisme DEVS qui permet de traiter les modèles indépendamment du noyau de simulation, il est possible de créer un environnement de simulation adaptable au domaine étudié. Dans la première partie nous introduisons les notions de comportements fautifs, de modèles de fautes et de librairies dédiées. Ces librairies contiendront les modèles BFS-DEVS propres aux différents domaines d'applications et dont l'interconnexion constitue une représentation du système. Dans la seconde partie, nous décrivons plus particulièrement les modifications apportées aux algorithmes de simulation DEVS afin d'intégrer les concepts de la SFC. Enfin, nous décrivons dans la troisième partie, le mécanisme de propagation de fautes BFS-DEVS basé sur *une technique de propagation des simulations fautives par découpage et réorientation de listes de fautes.*

Le **troisième chapitre**, intitulé **"Le formalisme BFS-DEVS appliqué aux circuits digitaux"**, est consacré à l'application de notre simulateur BFS-DEVS au domaine des circuits digitaux décrits dans un sous-ensemble du langage VHDL. Après avoir donné une brève description de ce sous ensemble, nous expliquons le processus de transformation de ces modèles "textuels" en modèles BFS-DEVS. Pour cela nous définissons quatre modèles de base qui constituent la librairie spécifique permettant de représenter les circuits digitaux. Dans la seconde partie nous présentons le modèle de fautes utilisé, ainsi que les mécanismes de propagation de listes de fautes. Nous terminons ce chapitre en détaillant la technique de simulation concurrente sur l'exemple d'une description VHDL décrivant un registre 8 bits.

Le **quatrième chapitre** de ce rapport intitulé **"Modélisation orientée objet"** décrit l'architecture orientée objet de notre outil à l'aide du langage UML. Nous présentons les aspects statiques et dynamiques du prototype grâce à l'utilisation de diagrammes de classes, de diagrammes de séquences et de diagrammes d'états-transitions.

Le **cinquième chapitre** intitulé **"Expérimentations et résultats"** rassemble les expériences et les résultats obtenus afin de valider notre approche et de montrer les avantages de la SFC dans le domaine des circuits digitaux. Nous avons effectué une série de simulation sur les dix premiers benchmarks ITC'99 dans le but d'obtenir et de comparer les évolutions des couvertures de fautes.

Nous terminons ce rapport en donnant les **conclusions** des travaux que nous avons développés ainsi que quelques **perspectives** de recherche.

CHAPITRE 2

État de l'art

Le but de ce chapitre est de définir le domaine d'étude dans lequel nous avons effectué nos travaux de recherche. Pour cela nous allons tout d'abord définir ce qu'est le test de circuit digitaux en précisant la position et la fonction d'un simulateur de fautes dans un tel processus. Dans un second temps, nous décrivons en détail la méthode de simulation comparative et concurrente ainsi que tous les concepts qui l'accompagnent. Ces concepts constituent l'essentiel des notions mises en oeuvre dans nos travaux. Une dernière partie est consacrée à la présentation du formalisme DEVS qui permet de modéliser et de simuler des systèmes à évènements discrets.

2.1 Le test de circuits

Le *test de circuit* est un processus indispensable de vérification intervenant à toutes les étapes du *flot de conception* d'un système digital [Landrault, 2004]. Le principe du test d'un circuit consiste à lui appliquer un ensemble de stimuli *(nommé vecteurs de test ou séquence de test)*, à mesurer sa réponse, à évaluer celle-ci en la comparant avec une réponse de référence, à prendre une décision de type correct/défectueux. Le premier critère de décision est la fonctionnalité glo-

bale du circuit et le test est appelé *test comportemental*. Cependant, lorsque le circuit devient important, cette procédure génère des séquences de test beaucoup trop longues. Pour diminuer le temps d'évaluation de ces séquences de test, une solution consiste à traiter les circuits, non plus de manière globale, mais comme un ensemble de blocs plus facilement testables. Ce test, qualifié de *test structurel,* donne des séquences de test minimales mettant en évidence le maximum de défauts au sein d'un modèle du circuit. Qu'il soit comportemental ou structurel, le test s'appuie sur [Miczo, 1986] :

– *les simulateurs de fautes* : ils permettent de s'assurer que la séquence de test que l'on va injecter au circuit détecte bien les fautes que l'on cherche à tester,
– *les générateurs automatique de séquences de test (ATPG pour Automatic Test Pattern Generation)* : ils produisent les stimuli à appliquer en fonction de la connaissance de la faute à tester et d'un modèle du circuit.

La simulation de fautes est une méthode qui permet de qualifier une séquence de test en terme de *couverture de fautes* [Agrawal, 1981] à partir d'un modèle du circuit et d'une liste de fautes susceptibles d'affecter ce circuit. Elle s'effectue en général sur un modèle logique du circuit et implique une définition des simulateurs de fautes et des *modèles de fautes* [Abramovici et al., 1990] au même niveau. Cependant, l'accroissement de la *complexité des circuits digitaux* a engendré une augmentation de la taille des séquences de test et des réponses à analyser. Ceci est à l'origine de la *migration* et de *l'automatisation* des outils de test vers des niveaux plus élevés du cycle de conception. C'est pourquoi les recherches se sont investies dans le développement de *simulateurs de fautes comportementales* [Santucci et al., 1993] travaillant, non plus sur des modèles logiques, mais sur des *modèles "textuels"* plus abstraits.

2.1.1 Le test dans le flot de conception

Le test d'un circuit intégré est une étape importante et indispensable intervenant tout au long du processus de conception jusqu'en fin de fabrication. Durant la phase de fabrication, les tests et le diagnostic sont variés et plusieurs méthodes sont développées afin de détecter les problèmes liés à la fabrication ou au vieillissement du circuit.

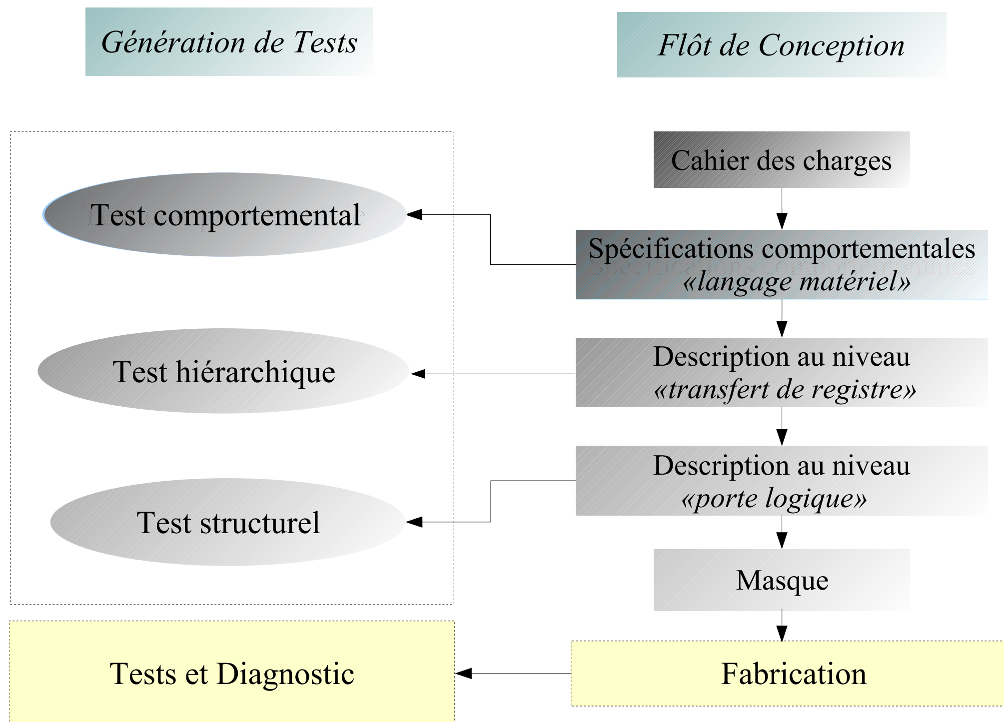


FIG. 2.1*: Place du test dans le flôt de conception d'un circuit.*

La figure 2.1 montre les étapes de génération de test présentes tout au long du *flot de conception* d'un circuit et pour être efficace, le test d'un circuit doit débuter très tôt dans le flot de conception. Le test de haut niveau peut utiliser une approche *structurelle,* une approche *hiérarchique* ou une approche *comportementale [Hurst, 1998, Buonanno et al., 1996] :*

- *Le test structurel* considère l'architecture et la topologie du circuit décrit au niveau porte logique. Défini par les fabricants, il exploite des fonctionnalités internes *(autotest, test en ligne,..)* souvent inaccessibles à l'usager. Il consiste à vérifier le bon fonctionnement des éléments de base du circuit à partir de sa structure interne. Cette approche permet l'obtention de séquences de test courtes et un niveau de qualité du test supérieur pour des hypothèses de dysfonctionnement données *(modèles de fautes).* L'inconvénient est que l'efficacité du test est limitée au choix de ces hypothèses.
- *Le test hiérarchique* : Les méthodes classiques de génération de vecteurs de test, basées sur la description au niveau porte logique atteignent très rapidement leurs limites dès que la taille du circuit dépasse quelques milliers de portes et en particulier pour les circuits séquentiels. La méthode hiérarchique de génération de vecteurs de test est basée sur des

informations issues à la fois du niveau RTL *(interconnexions de blocs)* et du niveau porte *(description interne des blocs)*. Cette technique s'appuie sur l'analyse de testabilité effectuée à haut niveau.

– *Le test comportemental* considère le circuit comme une "boîte noire" dont on ne connaît le fonctionnement qu'en présence de certains stimuli et que l'on ne peut vérifier qu'en contrôlant son comportement en fonction de l'application de ces stimuli. Il consiste à définir des vecteurs de test permettant de parcourir tous les modes de fonctionnement possibles du circuit, tels qu'ils sont spécifiés dans le cahier des charges *(ou dans les spécifications comportementales)*. Un tel test est très proche des simulations faites par un concepteur pour vérifier l'absence d'erreur de conception. Dans le cas d'un circuit très simple, purement combinatoire, cette approche revient à vérifier la table de vérité de la fonction globale réalisée par le circuit. Cependant elle présente deux inconvénients dans le cas d'un circuit complexe :
  – l'exhaustivité du test n'est généralement pas envisageable et la tentative de détecter le plus grand nombre de problèmes possibles conduit à *des séquences de test excessivement longues* ;
  – *aucune mesure fiable* n'existe pour indiquer le niveau de qualité atteint par un jeu de vecteurs de test fonctionnels.

2.1.2 Les notions de base du test

Le test structurel est très utilisé, mais compte tenu de la complexité croissante des circuits digitaux, le test comportemental devient une étape nécessaire dans le flot de conception. Quelque soit le niveau d'abstraction auquel on se réfère *(structurel ou comportementale)* les modèles utilisent des notions comme les modèles de fautes, les simulateurs de fautes et la génération de séquences de test. Ainsi, nous détaillons ces notions dans la suite de cette section.

2.1.2.1 Fautes et modèles de fautes

Le but du test de circuits digitaux est de rendre compte d'éventuels défauts physiques ; Ces défauts ne peuvent être détectés que s'ils induisent un comportement irrégulier que l'on appelle

faute. L'effet d'une faute est déterminé par une différence entre un état du "modèle sain" *(le modèle de référence)* et l'état correspondant du "modèle faux" *(modèle dans lequel est injectée une hypothèse de faute)*. Il convient de noter que les hypothèses de fautes dépendent du niveau d'abstraction utilisé pour la modélisation du circuit. Un modèle de faute est l'ensemble des hypothèses de fautes définies en terme de comportements défectueux des éléments de base du modèle du circuit.

Les outils de test utilisés travaillant en général au niveau logique, les modèles de fautes proposés sont des modèles logiques [Abramovici et al., 1990] :

– Collage *(stuck-at)* : c'est le collage d'un nœud du circuit à un état logique *(0 ou 1)* de manière permanente.
– Collage à l'état passant, Collage à l'état bloqué *(stuck-open, stuck-on)* : c'est le collage d'un transistor dans l'état passant ou l'état bloqué.
– Court circuit *(bridging fault)* : c'est une faute qui résulte du court-circuit entre plusieurs lignes du circuit intégré.
– Fautes de délai *(delay fault)* : ce sont des fautes qui modélisent les défauts affectant le temps de propagation du signal à travers une porte logique.

A partir de ces modélisations, on dispose de modèles qui, appliqués à un circuit, permettent de dresser une liste de fautes à prendre en compte lors du test : plus ces fautes se rapprochent de la réalité, meilleur est le test du circuit. Il a été montré depuis très longtemps [Landrault, 2004] que le modèle le plus représentatif des défauts physiques réels à l'intérieur des circuits est le modèle de collage. C'est la raison pour laquelle la majorité des outils de test proposent majoritairement ce modèle.

2.1.2.2 Vecteurs et séquences de test

La figure 2.2 schématise le principe d'application du test d'un circuit, ou d'un bloc à l'intérieur d'un circuit : des valeurs sont appliquées sur les entrées, et les résultats obtenus sur les sorties sont comparés à des valeurs prédéterminées.

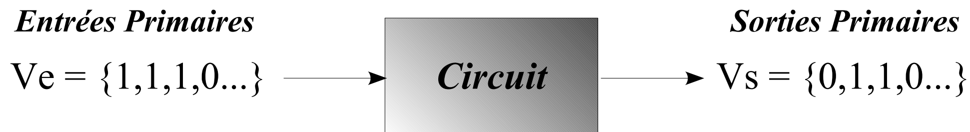


FIG. 2.2: *Vecteurs de test.*

Ceci correspond respectivement à un vecteur d'entrée V_e *(stimuli)* et un vecteur de sortie V_s *(référence)*. Un *vecteur de test* est composé de ces deux vecteurs : $V = (V_e, V_s)$. Les vecteurs V_e sont déterminés pour permettre la mise en évidence des différentes fautes que l'on cherche à détecter. Une séquence de test est définie comme un ensemble de vecteurs de test.

Les valeurs du vecteur d'entrée V_e peuvent être obtenues selon une approche comportementale ou par un algorithme permettant d'assurer la détection d'un ensemble de fautes donné. Nous parlerons dans ce cas de *test déterministe*, au sens où la séquence des valeurs d'entrée appliquées au circuit est définie de façon à détecter une liste précise de problèmes potentiels. Une autre approche consiste à envoyer sur les entrées du circuit des *valeurs choisies aléatoirement*. L'avantage du *test aléatoire* est le gain de temps obtenu, pendant la conception, en supprimant la détermination d'une séquence de test déterministe. Toutefois, la longueur de la séquence à appliquer, pour un niveau de qualité de test donné, est nettement plus longue pour un test aléatoire que pour un test déterministe. Cette approche est donc rarement utilisée pour un test externe en fin de fabrication.

2.1.2.3 La simulation de fautes

La méthodologie suivie par un simulateur de fautes classique consiste à simuler le circuit correct et les circuits fautifs afin de comparer les résultats sur les sorties primaires.

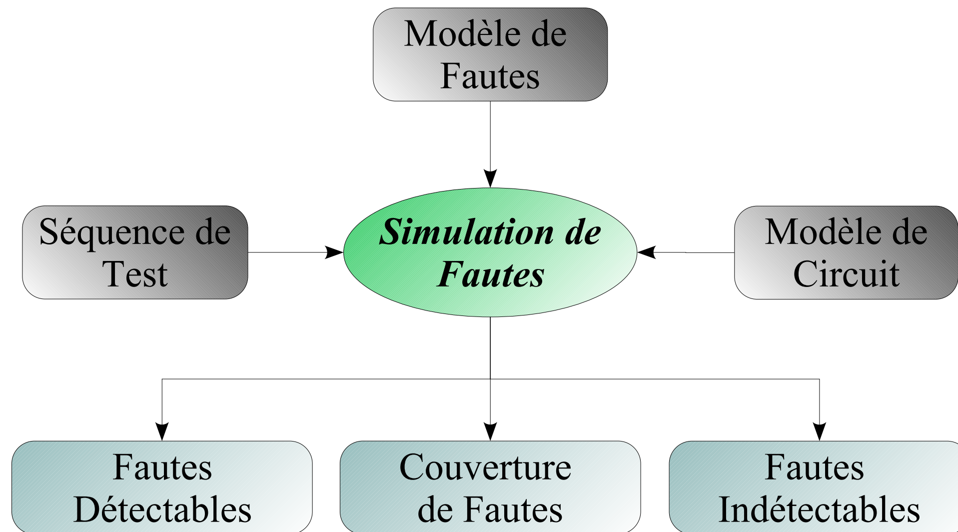


FIG. 2.3*: Méthodologie de la simulation de fautes.*

Comme il est montré sur la figure 2.3, le simulateur de fautes a besoin d'un *modèle du circuit* sous test *(par exemple un schéma portes logiques)*, d'un *modèle de fautes (par exemple le "stuck-at")* et d'un*e séquence de test* en entrée qui peut être générée de manière aléatoire. Une faute introduite dans le circuit défectueux impliquant une sortie primaire différente de celle obtenue par la simulation du circuit correct est dite *détect*ée. La *couverture de faute* est une métrique qui est obtenue en faisant le rapport du nombre de fautes détectées par le nombre total de fautes.

La simulation de fautes permet de qualifier un ensemble de vecteurs de test en terme de couverture de fautes. Plusieurs algorithmes de simulation de fautes au niveau portes logiques ont été développés comme :

– L'algorithme *en série :* il ne nécessite aucun développement d'un simulateur de fautes spécialisé et consomme très peu d'espace mémoire. Cependant il est très lent *(n simulations consécutives pour une liste de n fautes + une simulation du circuit correct).*
– L'algorithme *parallèle [Patil, 1991] :* il tire partie du traitement parallèle *(par mot)* de l'information binaire dans les ordinateurs et permet de simuler m-1 fautes *(avec m la taille du mot)*. Il est rapide mais complexe et nécessite plus de mémoire que l'algorithme en série.
– L'algorithme *déductif* [Armstrong, 1972] : il permet de simuler un nombre de fautes théoriquement infini *(contrairement à la simulation parallèle : m-1 fautes).* Il consiste à simuler uniquement le circuit correct afin de déduire les comportements des circuits défectueux. Il

permet la simulation de plusieurs fautes en un seul passage mais peut nécessiter un espace mémoire difficilement prévisible en pointe.

– L'algorithme *concurrent* [Demba et al., 1990] : il ne simule que les portions de circuits défectueux qui présentent une différence avec le circuit correct. Il consiste à associer une liste de configurations fautives à chaque élément du circuit. Il est rapide car il est basé sur la simulation simultanée de plusieurs portes fautives. Cependant il est complexe et demande beaucoup d'espace mémoire en début de simulation.

L'utilité du simulateur de fautes est double : il mesure la couverture de fautes pour savoir si la génération de vecteurs supplémentaires est nécessaire, il construit le dictionnaire des fautes détectées. Le dictionnaire contient la signature de chaque faute et est également utilisé pour le diagnostic du circuit [Agrawal et al., 1989]. Un tel usage du simulateur de fautes a été suggéré par Sesh et Freeman vers le début des années 60 [Kearney, 1984]. Ils utilisaient un simulateur à code compilé et les fautes étaient injectées en série mais cette méthode s'est vite révélée inefficace sur des circuits complexes.

La génération aléatoire de vecteurs de test a été utilisée dans les années 70 en combinaison avec les simulateurs de fautes uniquement dans le but simpliste de générer des vecteurs afin d'augmenter la couverture de fautes [Agrawal et Agrawal, 1972]. Cette stratégie a eu du succès avec quelques circuits combinatoires, mais la nécessité d'adopter des algorithmes déterministes s'est fait ressentir pour des circuits difficiles à tester. En effet, il faut souligner que lorsqu'une faute est particulièrement difficile à détecter, la méthode de génération aléatoire est inefficace car il faut parfois simuler consécutivement plusieurs vecteurs avant de la détecter. Ce problème à permis l'introduction d'un nouveau type d'application pour la simulation de fautes en exploitant l'activité interne des fautes. Plusieurs algorithmes concurrents et ATPG ont été développés, ils font l'objet de la section suivante.

Le début des années 70 a vu l'évolution des simulateurs de fautes vers des niveaux de description supérieurs au niveau porte logique. Ces simulateurs qui travaillent au niveau *RT (Register-Transfer level)*, fonctionnels ou comportementaux, sont moins précis que leurs prédécesseurs mais offrent une simulation facile à mettre en œuvre plus en amont dans le processus de conception.

2.1.3 Le test à haut niveau

Les circuits digitaux deviennent aujourd'hui de plus en plus complexes au niveau intégration et fonctionnalité et l'on est en mesure d'intégrer tout dans un même composant : c'est le concept de *single chip*.

Années	**1998**	**1999**	**2001**	**2002**	**2004**
Technologie	$0,25\mu m$	$0,18\mu m$	$0,15\mu m$	$0,23\mu m$	$0,09\mu m$
Complexité	1M	2-5M	5-10M	10-25M	>25M

TAB. 2.1*: Loi de Moore.*

Ceci est en fait lié à la loi de Moore qui stipule que pour une surface de silicium donnée, le nombre de transistors intégrés double tous les 18 mois ! Le tableau 2.1 montre cette évolution. La loi de Moore a radicalement changé la façon de concevoir les systèmes numériques. Les concepteurs travaillent maintenant au niveau système *(ou fonctionnalité)* et non au niveau porte logique.

L'évolution de la conception peut être résumée sur la figure 2.4.

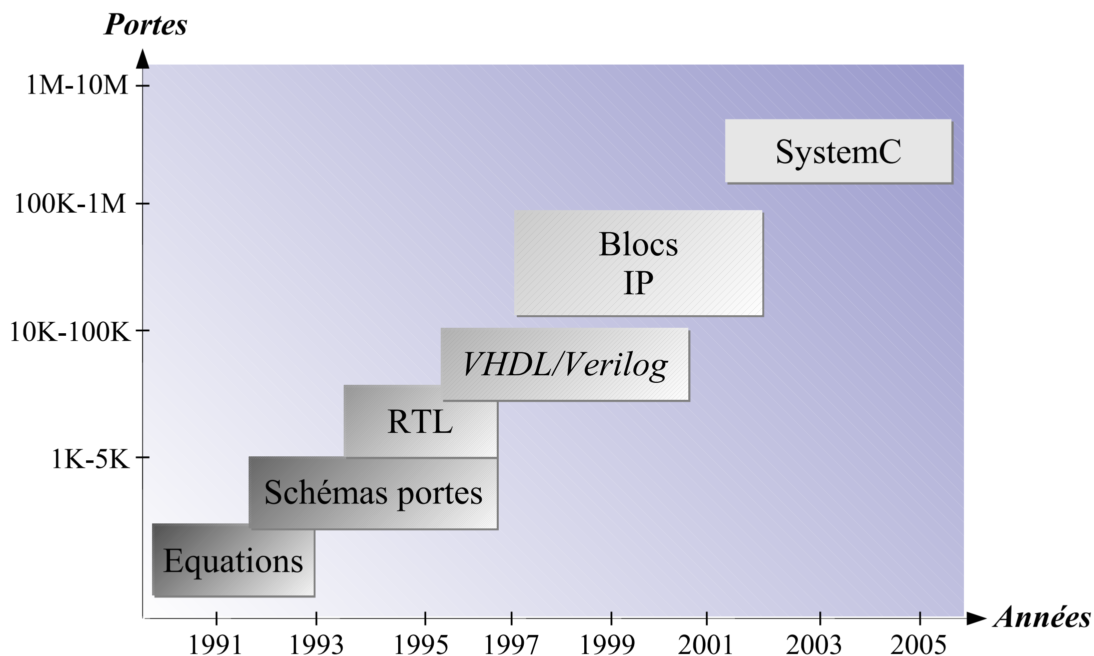


FIG. 2.4*: Évolution de la conception numérique.*

L'approche "schématique" au niveau porte logique ou fonctionnalités de base RTL *(Registre-Transfer-Level)* est délaissée pour la conception des systèmes complexes au profit d'une approche

"textuelle" mais reste bien sûr toujours valable pour la conception des systèmes numériques plus modestes. Pour cela, on utilise les langages de description matérielle comme VHDL *(Very high speed integrated circuit Hardware Description Language)* ou Verilog pour synthétiser une fonctionnalité numérique. Ces langages de description matérielle sont en fait de véritables langages de programmation et peuvent être utilisés comme tels par les informaticiens. Ils sont utilisés conjointement avec un compilateur ou un simulateur et ont permis de travailler avec un niveau d'abstraction plus grand.

De nos jours, le concepteur d'un circuit complexe veut avoir la possibilité de concevoir, simuler et synthétiser entièrement au niveau *RT*. C'est pour cette raison que les méthodes de conceptions à haut niveau, de la synthèse et de la génération de vecteurs de test devraient être proposées au sein des outils commerciaux. Durant le développement d'un projet, le concepteur veut avoir la possibilité de tester le circuit avant de débuter sa phase de synthèse logique. Durant ces dernières années, plusieurs publications faisant l'objet d'activités de recherche dans le domaine de la testabilité au niveau RT proposent des modèles de fautes [Devadas et al., 1996, Riesgo et Uceda, 1996, Thaker et al., 1999], des simulateurs de fautes [Fin et Fummi, 2000] ainsi que des ATPG [Ferrandi et al., 1998, Farzan Fallah et Devadas, 1999, Corno et al., 2000b]. Cependant, malgré de nombreux efforts, le domaine de la conception des circuits en vue de la testabilité à haut niveau reste un problème lié au choix d'un modèle de fautes réaliste et par conséquent est un problème difficile à solutionner.

2.1.3.1 Le modèle de fautes de haut niveau

Plusieurs modèles de fautes de haut niveau utilisant en majorité les fautes de type *"stuck-at"* ont été proposés mais ils ne peuvent être appliqués qu'à une classe restreinte de circuits. D'après [Goloubeva et al., 2002],il n'existe aucun modèle de fautes accepté de manière universelle pouvant fournir des résultats généraux et compréhensibles pour toutes les classes de circuits. Tant qu'aucun modèle de fautes réaliste ne sera accepté, les outils commerciaux de tests n'ont aucune raison d'investir dans le domaine du test à haut niveau.

De nombreuses approches de modélisation de fautes pour les descriptions VHDL comporte-

mentales ont été développées dans le domaine du test de logiciel [Beizer, 1990]. Ces modèles de fautes ont été ensuite étendus aux descriptions matérielles. Ceci explique comment les modèles de fautes de haut niveau correspondent à des *métriques* utilisées pour mesurer la validité des séquences de vecteurs d'entrées. L'un des modèles de fautes le plus utilisé est *"observability enhanced statement coverage"* proposé dans [Devadas et al., 1996, Fallah et al., 1998]. Ce modèle de fautes demande que toutes les instructions VHDL soient exécutées au moins une fois et que leurs résultats soient propagés sur au moins une des sorties primaires du circuit. La propagation est implicitement modélisée par l'effet de l'instruction fautive sur les valeurs de sortie. Une heuristique est cependant nécessaire pour résoudre les cas non-déterministes et influence fortement la couverture de fautes. Alors que cette approche peut être exploitée pour l'ATPG [Farzan Fallah et Devadas, 1999, Corno et al., 2000b] elle ne satisfait pas la simulation de fautes qui nécessite des résultats plus précis.

2.1.3.2 La simulation de fautes à haut niveau

Une fois le modèle de fautes choisi, une autre barrière au développement d'outils de test de haut niveau est le manque de simulateur de fautes de haut niveau. Les algorithmes de simulation de fautes pour les descriptions au niveau RT sont connus depuis une décennie. Cependant bien qu'ils pourraient être utilisés pour des descriptions comportementales, ils sont appliqués aux descriptions structurelles et les outils commerciaux refusent de les intégrer. *La complexité de ces algorithmes associée aux particularités du langage HDL rendent difficile leur intégration au sein des simulateurs HDL [Fulvio Corno, 2000].* L'obtention d'un simulateur de fautes, le plus souvent en série, est basé sur l'utilisation des simulateurs commerciaux associé à :

- *la modification du code VHDL* pour l'injection des fautes impliquant des temps de simulation importants [Fin et Fummi, 2000],
- *la modification du code source du noyau de simulation* nécessitant la possession de ce code,
- *l'interaction avec le noyau de simulation* au travers de scripts nécessitant la connaissance parfaite du logiciel de simulation [Fulvio Corno, 2000, Fin et Fummi, 2000].

Ces méthodes sont lourdes à implémenter et ne facilitent pas l'analyse *(observabilité et contrôlabilité)* des résultats issus d'une simulation de fautes.

2.1.4 Conclusion

Nous avons présenté dans cette partie les notions de test des circuits digitaux basées sur la simulation de fautes. Cette présentation permet de se familiariser avec les concepts utilisés par la suite pour étendre la simulation de fautes concurrente du niveau portes logiques vers le niveau comportemental. Les algorithmes de simulation de fautes concurrents développés au niveau portes logiques sont proches des algorithmes concurrents développés pour des niveaux d'abstraction supérieurs mais il ne sont pas implémentés au sein des outils de test commerciaux pour les raisons suivantes :

- Il n'existe aucun modèle de fautes général et réaliste de haut niveau,
- Les algorithmes de simulation de haut niveau sont complexes et difficiles à implémenter au sein des noyaux de simulation.

Nous laissons de coté la problématique du choix d'un modèle de fautes fiable pour axer nos recherches sur la définition d'un simulateur de fautes concurrent pour des descriptions VHDL comportementales de circuits digitaux.

2.2 La Simulation Comparative et Concurrente

Cette section évolution est consacrée à la *Simulation Comparative et Concurrente (SCC)* et plus particulièrement à la définition de la *Simulation de Fautes Concurrente (SFC)* dans le domaine des circuits digitaux . Après un bref historique nous expliquons le mécanisme de fonctionnement de la SCC. Nous mettons en évidence ces avantages et ces propriétés afin de dégager quelques concepts comme le *"rassemblement d'expériences*", la *"signature"* et "*l'observabilité"* . Enfin, nous abordons l'application la plus répandue de la SCC : la SFC appliquée au domaine des circuits digitaux.

2.2.1 Historique

La figure 2.5 compare deux formes de simulation concurrente *(SCC, SFC)* avec la Simulation à Evénements Discrets *(SED)*. La progression SED-SFC-SCC montre l'évolution et la généralisation de la simulation concurrente. Les augmentations de la rapidité visibles sur la figure 2.5 sont relatives au nombre d'expériences ou au nombre de fautes simulées.

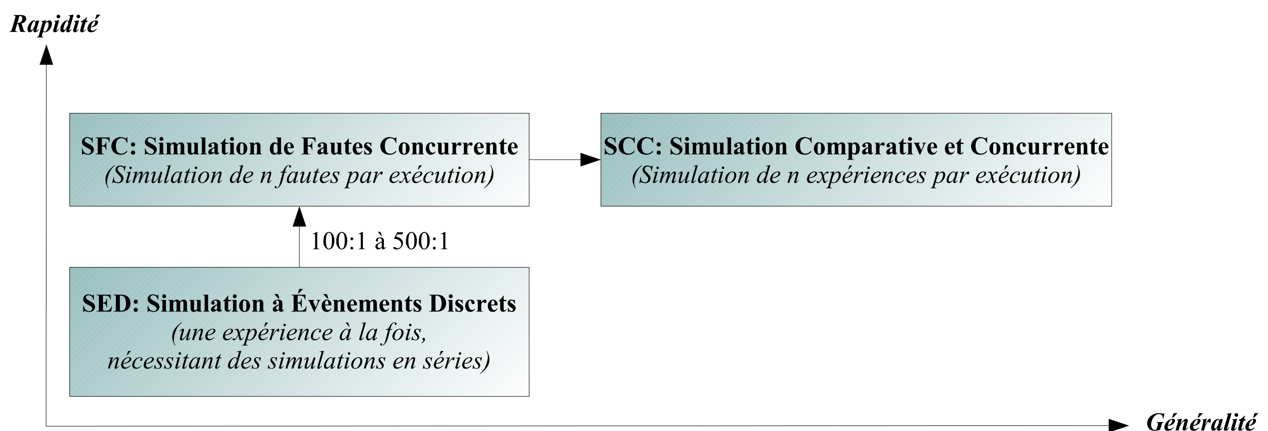


FIG. 2.5*: Évolution et généralisation de la simulation concurrente.*

D'après [Ulrich et al., 1994], les formes les plus évoluées de la SFC permettent des gains en rapidité de l'ordre de 100 :1 à 500 :1 par rapport à la SED, mais ces techniques sont limitées à la simulation des circuits digitaux décrits au niveau porte logique. La généralisation de la simulation concurrente a été possible grâce à la technique de propagation Transversale Multi-

Listes *(TML)* [Machlin, 1987, Gai et al., 1988a, Demba et al., 1990, Montessoro et Gai,] : DECSIM, MOZART, et CREATOR sont des simulateurs concurrents multi-niveaux *(gate, switch, etc)* les plus connues basés sur la TML.

2.2.2 Méthodologie de la SCC

La SCC permet la simulation concurrente de plusieurs expériences. Typiquement, les expériences concurrentes sont celles qui sont dues : à la présence de fautes au sein des systèmes digitaux, à l'existence de plusieurs exécutions d'un programme ou à l'exécution de différentes instructions au sein d'un même programme. La puissance essentielle de la SCC réside dans le fait que plusieurs expériences sont simulées de manière *implicite* au travers d'une seule simulation. La SCC débute avec la simulation de *référence (ou R-expérience)* qui sera à l'origine de toutes les expériences concurrentes (*ou C-expériences).* Dans tous les cas, les simulations manipulent des données et se propagent à travers les éléments ou les instructions de l'expérience. Les C-expériences peuvent diverger de la R-expérience soit parce qu'elles empruntent des *éléments différents de la R-expérience*, soit parce qu'elles affectent des données avec des *valeurs différentes* de la R-expérience.

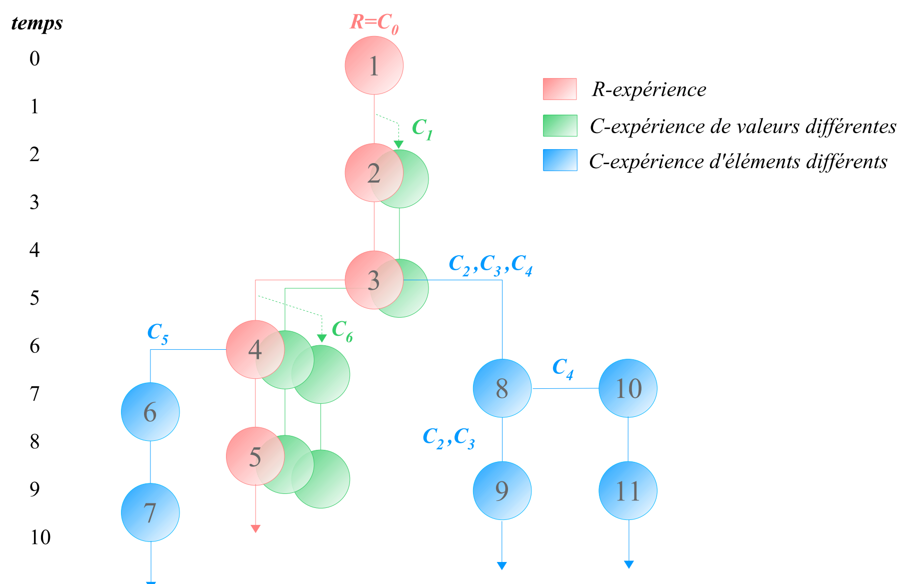


FIG. 2.6*: Exemple général d'une simulation comparative et concurrente.*

La figure 2.6, montrée en page suivante, illustre la simulation d'un nombre arbitraire *n* d'expériences $\{C_0, C_1, C_2, ..., C_n\}$. Jusqu'au temps 2, toutes les expériences sont implicitement simulées par rapport à la R-expérience C_0. Au temps 2, la C-expérience C_1 affecte des valeurs différentes aux données appartenant à C_0. Elle diverge de manière explicite de C_0 et parcour les mêmes *éléments que* C_0 *(i.e les éléments 2,3,4,5).* Au temps 5, 3 C-expériences d'éléments différents divergent de C_0 et se décomposent à nouveau au temps 7. Au temps 6, la C-expérience C_5 d'éléments différents *(6 et 7)* ainsi que la C-expérience C_6 de valeurs différentes divergent de $C0$. On remarque que C_6 possède une portion d'activité distincte de C_0 moins importante que C_1. Cette C-expérience demande donc moins d'espace mémoire et de temps CPU que C_1.

Durant la simulation, chaque C-expérience accumule dans une *signature (ou trace) :*

– la *taille* de l'expérience *(nombre d'éléments parcourus)*,
– la *liste des éléments* exécutés par l'expérience *(la liste [6,7] pour* C_5*)*,
– la *liste des données* modifiées par rapport à la R-expérience,
– le *temps et l'élément* de la création de l'expérience *(le temps 6 et l'élément 4 pour* C_5*).*

Les signatures sont importantes car l'exploitation de leur contenu facilite l'observabilité et la comparaison des expériences. Pour la SCC, il existe quatre sources d'efficacité [Ulrich et al., 1994] :

– *La similitude entre les expériences*. Si la simulation implique 500 expériences similaires, alors elles sont toutes représentées par une R-expérience qui s'exécute avec une efficacité de 500 :1 par rapport à la simulation en série conventionnelle.
– *La présence de C-expériences de données distinctes* mais travaillant sur les mêmes éléments. Ce sont des petites expériences nécessitant peu de temps CPU.
– *La présence de C-expériences d'éléments différents*. Elle peuvent être supprimées après un temps très court car elles sont facilement observables.
– *La similitude entre des C-expériences*. Si deux ou plusieurs C-expériences convergent vers les mêmes éléments, elles peuvent être simulées efficacement dans une seule C-expérience.

Les principes de fonctionnement de la SCC que nous venons d'exposer constituent les bases de nos travaux. Les notions de R-expérience, de C-expérience et de signature feront partie des concepts majeurs que nous allons intégrer dans notre approche.

2.2.3 Propriétés et avantages de la SCC

La SCC est une méthode algorithmique concurrente basée sur le domaine temporel qui s'applique et reste limitée aux systèmes à événements discrets [Breuer et Friedman, 1976, Ulrich, 1978, Phillips et Tellier, 1978]. Les résultats de l'exécution d'une SCC sont fortement proportionnels aux nombres d'expériences simulées. Cette méthode est parallèle/concurrente et minimise le travail manuel car elle est plus systématique et exhaustive que la méthode en série.

La simulation en série demande une synchronisation temporelle parfaite et beaucoup de travail manuel pour sa mise en œuvre. De ce fait, les expériences initiales reçoivent de l'utilisateur le maximum d'attention et les expériences suivantes suscitant trop peu d'attention sont souvent négligées voir non simulées. De plus, les résultats surgissent dans l'ordre des expériences simulées qui est arbitraire. Pour la SCC il n'existe pas d'arbitraire, toutes les expériences sont simulées et les résultats apparaissent en même temps et peuvent être analysés et comparés à chaque temps de simulation.

La SCC est similaire à une simulation multi-processeurs avec un processeur par expérience. Cependant il n'y a pas besoin de machine parallèle et de coûts de communication entre les processeurs. La SCC utilise un seul processeur réel et autant de processeur virtuels qu'il y a d'expériences. En étant une solution logicielle, elle est plus générale et plus flexible que la solution matérielle.

Les avantages principaux de la SCC sont :

1. La diminution du temps de développement car elle *rassemble* plusieurs expériences dans une seule exécution, évitant les interruptions et les travaux manuels entre les simulations en séries successives. Elle minimise les temps CPU car elle est basée sur la similarité et sur l'unicité des initialisations des simulations.

2. L'observation, comparativement au cas de la simulation en série, est simple car les expériences s'exécutent en parallèle. De plus, elle est automatique car elle est basée sur l'analyse et la maintenance des *signatures* de toutes les expériences.

3. Elle est utilisée dans plusieurs domaines d'application comme la *simulation concurrente de*

*logiciel*s. Dans ce cas, la SCC permet la simulation des différentes variantes d'exécution d'un programme informatique, le test et le débogage des sous-programmes.

4. Le contrôle automatique de l'exécution de la SCC car elle est basée sur l'observation des expériences et des signatures. Les *suppressions/additions* d'expériences ne nécessitent aucun travail manuel.

2.2.4 La Simulation de Fautes Concurrente

La simulation de fautes est caractérisée par le besoin de simuler des milliers d'expériences fautives sur des circuits digitaux. La *Simulation de Fautes Concurrente (SFC)* sur les circuits digitaux a été la première application de la SCC [Breuer et Friedman, 1976]. En effet, le principal domaine d'application dans lequel la SCC a montré la plus importante évolution est le domaine de la simulation de fautes des circuit digitaux décrits au niveau porte logique. Plusieurs simulateurs de fautes concurrents ont été développés en se basant sur des descriptions multi-niveaux comme le simulateur DECSIM qui peut simuler des fautes à quatre niveaux logiques différents *(swtich, portes logiques, mémoires et comportementale)*. MOZART et CREATOR [Gai et al., 1987, Gai et al., 1988b, Gai et al., 1988a, Demba et al., 1990, Montessoro et Gai,] sont d'autres simulateurs de fautes concurrent qui utilisent en plus une technique de propagation de liste de fautes appelée Transversal Multi-List *(TML)*.

L'algorithme concurrent repose sur l'activité des simulations fautives individuelles résultant des expériences fautives. Ces expériences fautives *(ou mauvaises portes)* se détachent de manière indépendante de l'expérience de référence R et peuvent diverger ou converger au cours de la simulation.

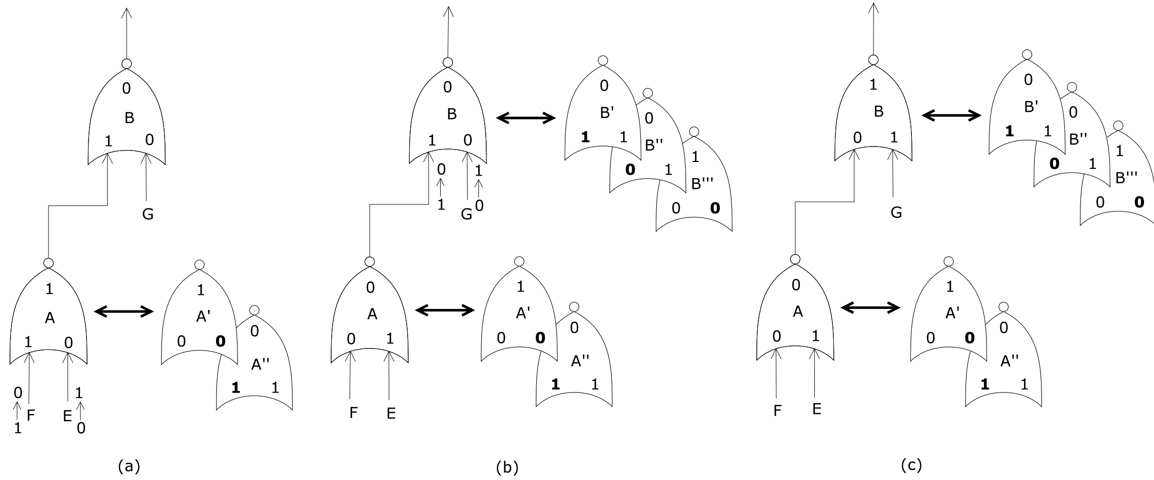


FIG. 2.7*: Propagation de l'effet d'une faute au niveau porte logique.*

La figure 2.7 illustre la simulation de fautes concurrente sur un réseau de deux portes logiques OU. Considérons la situation ou : le signal d'entrée E a l'activité $0 \rightarrow 1$ et le signal d'entrée F a l'activité $1 \rightarrow 0$. Ces activités impliquent l'évolution de la porte de référence A. Les deux fautes E_0 *(collage de E à* 0*)* et F_1 *(collage de F à* 1*)* sont à l'origine de la divergence de la porte A vers les deux portes fautives A' et A'' *(cf. figure 2.7 (a))*. La simulation de référence débutée à la porte A évolue vers la porte B. Les fautes E_0 et F_1 sont propagées et impliquant les mauvaises portes B' et B''. L'activité $0 \rightarrow 1$ du signal d'entrée G peut être à l'origine de la présence d'une nouvelle faute de collage à 0 sur le signal G noté G_0. Cette faute est à l'origine d'une troisième porte fautive B''' *(cf. figure 2.7 (b))*. En fin de simulation *(cf. figure 2.7 (c))*, les valeurs de sorties des portes B' et B'' étant différentes de la porte de référence B, les fautes E_0 et F_1 sont détectées. Les valeurs de sortie des portes B''' et B sont identiques, la faute G_0 n'est pas détectée. Cet exemple montre que la divergence des trois *C-expériences fautives* (A',B'),(A'',B'') et (A,B''') a lieu à partir de *l'expérience de référence (A,B)*.

Le mécanisme de *co-détection* des fautes reste souvent inexploité alors qu'il est possible dans la SFC [Ulrich, 1985]. Pour la simulation de fautes, le nombre de fautes simulées peut être réduit en considérant que certaines d'entre elles sont détectables par co-détection. Si deux fautes distinctes sont co-détectées *(en même temps et sur le même élément)* et si de plus leurs expériences fautives sont similaires alors la probabilité que ces deux fautes appartiennent à une même expérience est très élevée. Cette notion peut être mise en évidence sur l'exemple de la figure *2.7.*

Considérons la porte A comme défectueuse car elle possède un retard sur le “pin” d’entrée portant le signal E noté R_A. Supposons que ce retard soit supérieur au temps nécessaire pour obtenir la valeur de sortie sur la porte B en fin de simulation. Dans ce cas, les fautes R_A et E_0 causent les mêmes effets et appartiennent à la même simulation fautive. Les fautes E_0 et R_A sont deux fautes co-détectables et sont détectées en même temps.

2.2.5 Conclusion

Dans cette section nous avons donné un bref résumé de la SCC en présentant ses concepts, ses propriétés ainsi que ses avantages. La SCC permet la simulation concurrente de *plusieurs expériences en seule exécution*. Nous avons ensuite exposé les principes de la SFC afin de définir le contexte dans lequel nos travaux de recherche se sont insérés : *la simulation concurrente de fautes au sein des circuits digitaux*. Les notions d’expériences fautives divergentes, d’expérience de référence, de signature, de propagation des effets d’une expérience fautive, de co-détection constituent les bases de notre approche.

2.3 Le formalisme DEVS

La *théorie des systèmes* [Boulding, 1956] développée dans les années 60 apporte un formalisme fondamental et rigoureux du point de vue mathématique, permettant la représentation des systèmes dynamiques. Il existe deux principaux aspects orthogonaux de cette théorie : les *niveaux de spécifications* décrivant les mécanismes qui les font évoluer et les *formalismes de spécification* décrivant les approches de modélisation, continue ou discrète, que les modélisateurs peuvent employer pour établir les modèles.

Pour P.A Mûller, un modèle est *"une description abstraite d'un système ou d'un processus, une représentation simplifiée permettant de comprendre et de simuler"*. Les modèles peuvent être classifiés en deux catégories en fonction de leur comportement temporel : Les *modèles continus* pour lesquels le temps s'écoule sans discontinuités, et les *modèles discrets* évoluant par pas de temps. Les modèles à temps continu peuvent être représentés par des équations différentielles partielles ou par des formalismes à événements discrets. Pour les modèles à événements discrets, les événements arrivent de manière continue dans le temps et les changement d'états interviennent seulement à des instants précis. En effet, le temps évolue d'un événement à un autre de manière arbitraire.

Basé sur la théorie des systèmes, *le formalisme DEVS (Discrete EVent system Specification)* a été introduit par le professeur B.P.Zeigler vers la fin des années 70 [Zeigler, 1976]. Il permet une modélisation *modulaire* et *hiérarchique* des systèmes à événements discrets. Un système *(ou modèle)* est dit modulaire dans le sens ou il possède des ports d'entrée et de sortie permettant l'interaction avec son environnement extérieur. Dans le formalisme DEVS un modèle est vu comme une *"boîte noire"* qui reçoit et émet des messages sur ses ports d'entrées ou de sorties.

Cette section introduit le formalisme DEVS en distinguant la partie modélisation de la partie simulation. Dans la première partie nous mettons en évidence les concepts de modularité et de hiérarchie en présentant deux types de modèles : les *modèles atomiques* et les *modèles couplés*. Dans la seconde partie nous montrons comment il est possible à partir des modèles de générer *automatiquement* les algorithmes de simulation associés. Nous verrons plus particulièrement comment à partir d'une notion de *messages* il est possible de mettre en œuvre ces algorithmes.

2.3.1 La modélisation DEVS

Le formalisme DEVS définit deux catégories de modèles : les *modèles atomiques* et les *modèles couplés*. Les modèles atomiques sont chargés de représenter le(s) comportement(s) du système. Les modèles couplés sont définis par un ensemble de sous-modèles *(atomiques et/ou couplés)* et traduisent la structure interne des sous-parties du système grâce à la définition de couplages entre les sous-modèles.

2.3.1.1 La notion de modèle atomique

Le formalisme de base DEVS dit *"classique"* considère un modèle atomique comme un concept mathématique basé sur le temps, un ensemble de valeur caractérisant tous les stimuli possibles en entrée et en sortie du système ainsi que deux fonctions permettant de déterminer la réponse comportementale à ces stimuli. Dans le formalisme DEVS dit *"classique avec ports"* la notion de couple $(port, valeur)$ est introduite pour chaque port *(entrée ou sortie)* d'un modèle atomique. Les spécifications classiques d'un modèle atomique *MA* avec ports sont les suivantes :

$$\boxed{MA = < X, Y, S, \delta_{int}, \delta_{ext}, \lambda, t_a >}$$

où,

- $X = \{(p,v) | p \in Ports_entree, v \in X_p\}$ est la liste des ports et des valeurs d'entrées,
- $Y = \{(p,v) | p \in Ports_entree, v \in X_p\}$ est la liste des ports et des valeurs de sorties,
- S est l'ensemble des variables d'états,
- $\delta_{int} : S \rightarrow S$ est la fonction de transition interne,
- $\delta_{ext} : Q \times X \rightarrow S$ est la fonction de transition externe où,
  - $Q = \{(s,e) | s \in S, 0 \leq e \leq t_a(s)\}$ est l'ensemble des états,
  - e est le temps écoulé depuis la dernière transition.
- $\lambda : S \rightarrow Y$ est la fonction de sortie,
- $t_a : S \rightarrow \mathbb{R}^+$ est le temps de vie de l'état S *(réels positifs de 0 à ∞).*

Les modèles réagissent à deux types d'événements : les *événements externes* et les *événements internes*. Les événements externes proviennent d'un autre modèle, déclenchent la fonction de transition externe et mettent à jour le temps de vie du composant. Les événements internes correspondent à des changements d'états du modèle, déclenchent les fonctions de transitions internes et de sorties, le modèle calcul ensuite grâce à la fonction t_a la date du prochain événement interne.

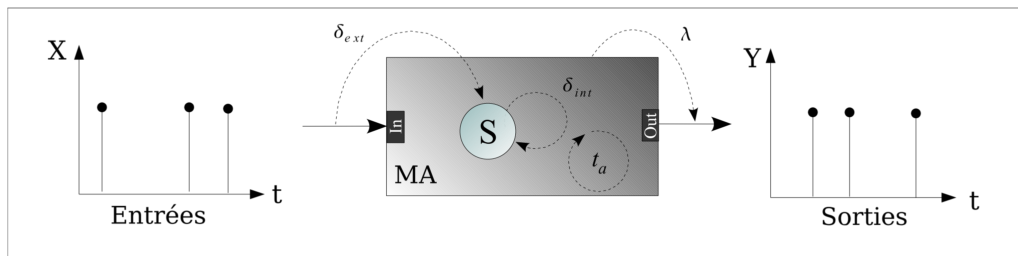


FIG. 2.8*: Modèle atomique en action.*

L'interprétation des ces éléments est illustré sur la figure 2.8. A chaque instant le système *(modèle atomique)* est dans un état s. Si aucun événement externe n'intervient, le système restera dans cet état pendant un temps donné par la fonction $t_a(s)$. Lorsque le temps de vie du modèle est expiré, i.e. lorsque qu'il s'est écoulé $e = t_a(s)$ le système active sa fonction de sortie $\lambda(s)$ et change d'état grâce à l'exécution de la fonction de transition interne $\delta_{int}(s)$. Si un événement externe $x \in X$ intervient avant que le temps ne soit expiré, i.e. quand le système est dans l'état total (s,e) avec $e \leq t_a(s)$, le système change d'état grâce à l'exécution de la fonction de transition externe $\delta_{ext}(s,e,x)$. Dans les deux cas, le système est alors dans un nouvel état s' avec un nouveau temps restant $t_a(s')$ et ainsi de suite.

Le temps de vie du composant peut être égal à zéro ou à l'infini. Dans le premier cas, la durée de l'état s est tellement courte qu'aucun événement externe ne peut intervenir avant l'arrivée du prochain changement d'état. Nous disons de s qu'il est dans un *état transitoire*. Dans le second cas, le système restera dans l'état s indéfiniment si aucun événement externe ne vient l'interrompre. Nous disons dans ce cas que s est un *état passif*.

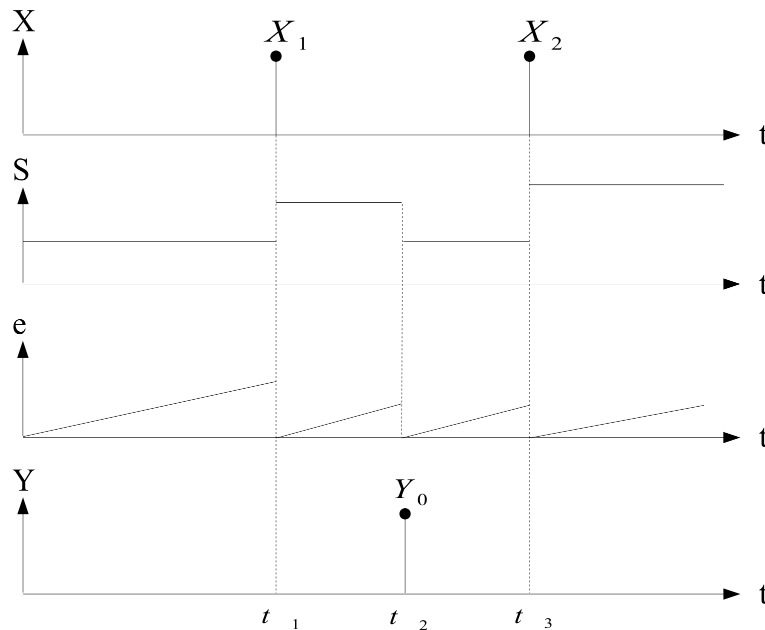


FIG. 2.9: *Trajectoires d'un modèle atomique.*

L'explication précédente de l'activité d'un modèle atomique DEVS suggère, mais ne décrit pas totalement, le fonctionnement d'un simulateur qui exécuterait de tels modèles pour produire leurs comportements. Cependant, le comportement de DEVS est bien défini et peut être représenté comme sur la figure 2.9. Ici la *trajectoire d'entrée X* représente une série d'événements occurrents au temps t_1 et t_3. Entre ces événements temporels peuvent intervenir des temps comme t_2 correspondant à des événements internes. La courbe de la *trajectoire d'état s* montre le changement d'état lors de la venue d'événements internes et externes. La *trajectoire du temps écoulé e* est en dent de scie et montre l'écoulement du temps par un compteur qui est remis à zéro à chaque changement d'événement. Finalement, *la trajectoire de sortie Y* montre les événements de sortie qui sont produits par l'exécution de la fonction de sortie après avoir appliqué la fonction de transition interne.

Notons que dans DEVS classique avec ports, un port p reçoit une seule valeur v d'un événement externe. Avec le formalisme *DEVS parallèle* [Chow et Zeigler, 1994, Chow et Zeigler, 2003], un modèle DEVS peut recevoir sur un ports d'entrée donné plusieurs valeurs simultanées. La gestion des événements simultanés est prise en compte au sein d'une fonction de conflit δ_{conv} dictant un ordre de priorité entre ces événements. Ce genre de systèmes n'est pas abordé dans nos travaux.

2.3.1.2 La notion de modèle couplé

Le formalisme DEVS utilise la notion de *hiérarchie de description* qui permet la construction de modèles dits “couplés” à partir d'un ensemble de sous-modèles et de trois relations de couplage avec ces sous-modèles. Un modèle couplé est constitué de sous-modèles qui peuvent être *atomiques ou couplés* et il possède les *trois relations de couplage* suivantes :

- *une relation de couplage interne* pour le couplage entre les ports des sous-modèles qui composent le modèle couplé *(en rouge sur la figure 2.10)*,
- *une relation de couplage des entrées externes* pour le couplage entre les ports d'entrées du modèle couplé et les ports d'entrées des sous-modèles *(en bleue sur la figure 2.10)*,
- *une relation de couplage des sorties externes* pour le couplage entre les ports de sorties du modèle couplé et les ports de sorties des sous-modèles *(en vert sur la figure 2.10)*.

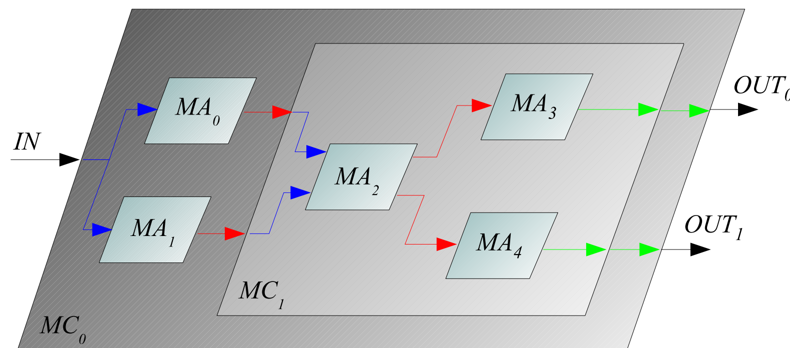


FIG. 2.10*: Hiérarchie de modélisation DEVS.*

La figure 2.10 montre un exemple de hiérarchie entre les sous-modèles d'un système. Le modèle couplé MC_0 modélise au plus haut niveau le système étudié. Il possède deux ports de sortie OUT_0 et OUT_1, un port d'entrée IN et il contient les deux modèles atomiques MA_0 et MA_1 ainsi qu'un modèle couplé supplémentaire MC_1. Les modèles atomiques MA_0 et MA_1 sont reliés par une relation de couplage d'entrées externe au modèle couplé MC_0 *(en bleue sur la figure 2.10)*, et par une relation de couplage interne au modèle couplé MC_1 *(en rouge sur la figure 2.10)*.

Dans le cas du formalisme DEVS classique avec port les spécifications d'un modèle couplé sont les suivantes :

$$\boxed{MC = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >}$$

où,

- $X = \{(p,v) \,|\, p \in ports_entree,\, v \in X_p\}$ est la liste des ports et des valeurs d'entrées.
- $Y = \{(p,v| \, p \in ports_sortie,\, v \in Y_p)\}$ est la liste des ports et des valeurs de sorties.
- D est la liste des composants constituant le modèle couplé.
- $M_i =< X_i, Y_i, S_i, \delta_{ext,i}, \delta_{int,i}, \lambda_i, t_{a,i} >$ est un modèle atomique,
- Pour chaque modèle $i \in D \cup \{MC\}$, I_i est l'ensemble des modèles qui influencent i,
- $Z_{i,j}$ est la fonction de translation des sorties de i vers j telle que :
  - $Z_{MC,j} : X_{MC} \rightarrow X_j$ est la fonction de couplage des entrées externes,
  - $Z_{i,MC} : Y_i \rightarrow Y_{MC}$ est la fonction de couplage des sorties externes,
  - $Z_{i,j} : Y_i \rightarrow X_j$ est la fonction de couplage interne.

La structure d'un modèle couplé doit répondre à des contraintes telle que, $\forall i \in D$:

1. $M_i =< X_i, Y_i, S_i, \delta_{ext,i}, \delta_{int,i} \lambda_i, t_{a,i} >$ est un modèle atomique,
2. Une seule fonction $Z_{i,j}$ contient l'ensemble des informations sur les couplages du modèle couplé,
3. I_i est un sous-ensemble de $D \cup \{MC\}$ et $i \notin I_i$.

La cohérence et la conservation des comportements du système entre ces niveaux hiérarchiques est résumée par la propriété dite*"closed under coupling"* [Zeigler, 1990]. Cette propriété assure qu'un modèle couplé, représenté par le couplage d'un ensemble de sous-modèles plus détaillés, est équivalent à un modèle atomique *(contrainte n°1)*.

2.3.2 La simulation DEVS

L'une des propriétés importantes du formalisme DEVS est qu'il fournit automatiquement un simulateur pour chacun des modèles. DEVS établit une distinction entre la modélisation et la simulation d'un système tel que n'importe quel modèle DEVS puisse être simulé sans qu'il ne soit nécessaire d'implémenter un simulateur spécifique. Chaque modèle atomique est associé à un *simulateur* chargé de gérer le comportement du composant et chaque modèle couplé est associé à un *coordinateur* chargé de la synchronisation temporelle des composants sous-jacents. L'ensemble de ces modèles est géré par un coordinateur spécifique appelé *"Root"*.

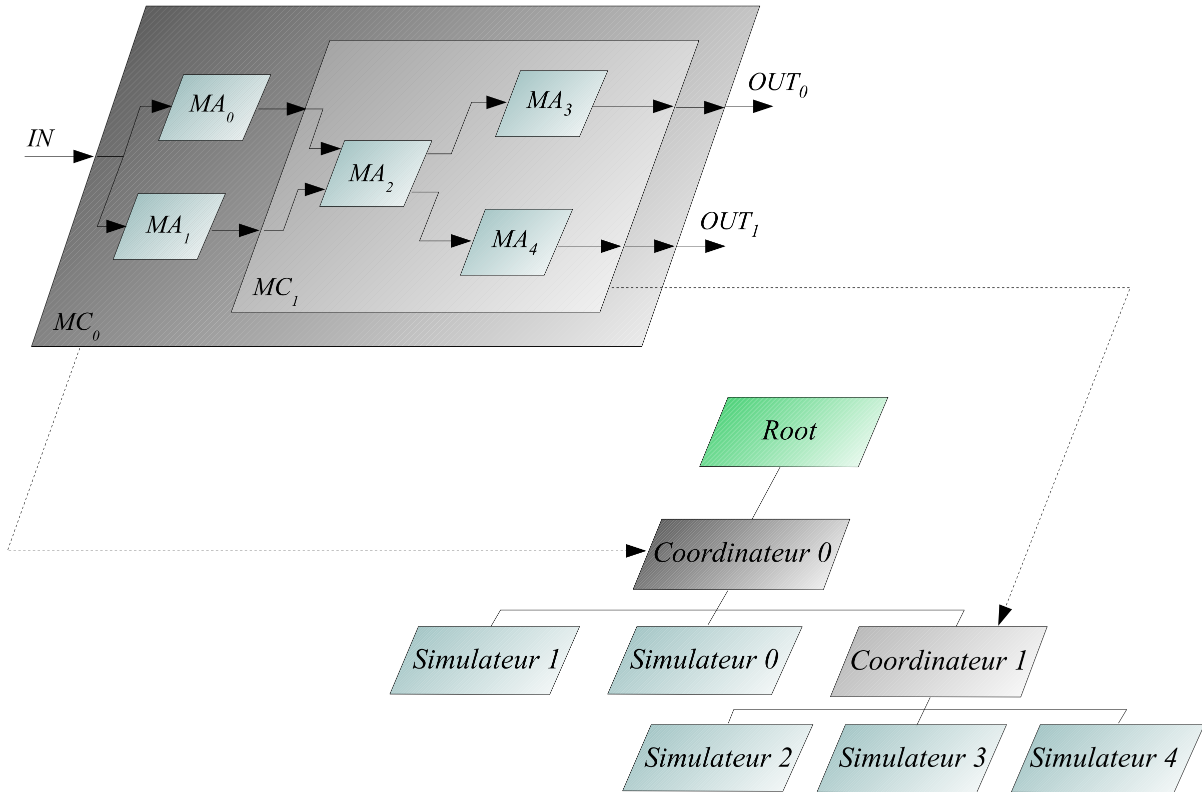


FIG. 2.11*: Arbre hiérarchique de simulation DEVS.*

La figure 2.11 montre la structure d'un simulateur associé à un modèle DEVS quelconque. La hiérarchie du simulateur DEVS est construite sur une arborescence constituée de simulateurs et de coordinateurs. Les deux coordinateurs "$Coordinateur\,0$" et "$Coordinateur\,1$" sont associés aux modèles couplés MC_0 et MC_1. Le "$Coordinateur\,0$" est chargé de gérer le "$Coordinateur\,1$" ainsi que les simulateurs "$simulateur\,0$" et "$Simulateur\,1$" associés aux deux modèles atomiques MA_0 et MA_1. De la même manière, comme le modèle couplé MC_1 encapsule les trois modèles atomiques MA_2, MA_3 et MA_4, le coordinateur "$Coordinateur\,1$" associé gère les trois simulateurs "$Simulateur\,2$", "$Simulateur\,3$" et "$Simulateur\,4$". Le coordinateur "*Root*" est chargé de coordonner la totalité des composants de l'arbre de simulation.

L'ordre d'exécution des fonctions comportementales (δ_{int},δ_{ext},λ,t_a) d'un modèle atomique DEVS de la figure 2.8 sous-entend la présence d'un mécanisme de simulation. Ce mécanisme est géré par le composant simulateur qui, comme le coordinateur, peut recevoir ou émettre 4 types de messages :

– Le **message d'initialisation (i,t)** permet à tous les acteurs d'effectuer une synchronisation

temporelle initiale.

- Le **message de transition interne (*,t)** permet la gestion d'un événement interne avec l'exécution de la fonction δ_{int}.
- Le **message de sortie (y,t)** permet le transport des sorties *(données par λ)* aux éléments parents et résulte de la réception d'un message (*,t).
- Le **message de transition externe (x,t)** permet la gestion d'un événement externe avec l'exécution de la fonction δ_{ext}.

Dans cette sous-section nous présentons les algorithmes de simulation des simulateurs et des simulateurs. Ces algorithmes vont nous permettre de mieux comprendre comment au travers de l'utilisation de ces messages, il est possible de coordonner l'exécution des modèles atomiques et couplés DEVS.

2.3.2.1 Algorithme d'un composant Simulateur

Une interprétation de la dynamique de DEVS est donnée en considérant un composant simulateur DEVS. Il utilise deux variables temporelles t_l *(last time)* et t_n *(next time)*. La première correspond au temps de simulation du dernier événement et la seconde au temps d'apparition du prochain événement. A partir de la définition du temps d'avancement t_a on a :

$$t_n = t_l + t_a$$

De plus, si le temps de simulation t est connu, le composant simulateur peut calculer à partir de cette variable le temps écoulé depuis le dernier événement,

$$e = t - t_l$$

Ainsi que le temps restant pour l'apparition du prochain événement,

$$\sigma = t_n - t = t_a - e$$

Le temps t_n est envoyé au composant coordinateur parent pour lui permettre d'effectuer une bonne synchronisation des événements.

Algorithme 1 Algorithme du simulateur DEVS.

Variables :

parent // *coordinateur parent*

t_l // *temps du dernier événement*

t_n // *temps du prochain événement interne*

$DEVS = < X, Y, S, \delta_{int}, \delta_{ext}, \lambda, t_a >$ // *modèle associé*

y // *sortie courante du modèle*

Réception i-message (i,t) au temps t :

$t_l = t - e$

$t_n = t_l + t_a(s)$

Réception ⋆-message (⋆,t) au temps t :

si $(t \mathrel{!}= t_n)$ alors

Erreur : mauvaise synchronisation

$y = \lambda(s)$ //*stockage de la sortie avant changement d'état*

envoie y-message (y,t) au parent coordinateur

$s = \delta_{int}(s)$

$t_l = t$

$t_n = t_l + t_a(s)$

Réception x-message (x,t) au temps t avec x en entrée :

si $!(t_l \leq t \leq t_n)$ alors

Erreur : mauvaise synchronisation

$e = t - t_l$

$s = \delta_{ext}(s, e, x)$

$t_l = t$

$t_n = t_l + t_a(s)$

Comme il est montré sur l'algorithme 1 [Zeigler et al., 2000], pour une initialisation correcte du simulateur un message d'initialisation (i,t) doit être reçu au début de chaque exécution d'une simulation. Quand le simulateur DEVS reçoit ce message, il initialise son temps t_l par soustraction du temps écoulé e au temps de simulation t apporté par le message. Le temps d'apparition du prochain événement t_n est calculé par addition du temps t_l et du temps donné par la fonction d'avancement du temps $t_a(s)$. Ce temps t_n est envoyé au coordinateur parent pour l'informer sur l'apparition du premier événement interne qui devra être exécuté par ce composant simulateur.

La réception d'un message de transition interne $(*,t)$ provoque l'exécution d'un événement interne. Quand le message $(*,t)$ est reçu par le simulateur DEVS, il calcule la sortie et exécute

la fonction de transition interne du modèle DEVS associé. La sortie est renvoyée au coordinateur parent via le message (y,t). Finalement le temps d'apparition du dernier événement t_l est égal au temps de simulation et le temps d'apparition du prochain événement t_n est égal à la somme du temps courant t et du temps donné par la fonction d'avancement $t_a(s)$.

Un message d'entrée (x,t) informe le composant simulateur de l'arrivée d'un événement d'entrée avec la valeur x au temps de simulation t. Cela provoque l'exécution de la fonction de transition externe $\delta_{ext}(s,e,x)$ du modèle atomique avec pour données d'entrée x et pour temps écoulé depuis le dernier événement e. Comme pour l'événement interne, le temps d'apparition du dernier événement t_l est égal au temps courant et le temps d'apparition du prochain événement t_n est égal à l'addition du temps courant et de la fonction d'avancement du temps $t_a(s)$.

2.3.2.2 Algorithme d'un composant Coordinateur

Un composant coordinateur est responsable du bon fonctionnement et de la synchronisation des ces subordonnés *(simulateurs et/ou coordinateurs qui le compose)*. Le coordinateur utilise une liste d'événements $liste_{event} = \{(d,t_{nd}) | d \in D, t_{nd} \in \Re_+\}$ pour assurer la synchronisation de ces composants. Le premier composant de la liste définit alors le prochain événement dans la mire du coordinateur. Le temps minimum, $t_n = min\{t_{nd} | d \in D\}$ est fourni aux parents du coordinateur comme le temps du prochain événement. De manière similaire, le temps de l'événement précédent du coordinateur est calculé par : $t_l = max\{t_{nd} | d \in D\}$.

La fonction $Z_{i,j}$ permet la transmission des valeurs entre les ports des modèles i et j. Par exemple, si la valeur y_i sort d'un port du modèle i et que ce port est relié à un port du modèle j qui attend une valeur x_j, on a : $Z_{i,j}(y_i) = x_j$. L'ensemble I_i est composé des modèles qui influencent le modèle i. Si un modèle i a deux ports d'entrées reliés à deux modèles j et k, on a : $I_j = \{j,k\}$.

Algorithme 2 Algorithme du coordinateur DEVS.

```
Variables :
 parent // coordinateur parent
 t_l // temps du dernier événement
 t_n //temps du prochain événement interne
 MC =< X,Y,D,{M_i},{I_i},{Z_{i,j}} >
 liste_event //liste des événements (d,t_nd) (triée par t_n croissant)
 d∗ //fils imminent sélectionné

Réception i-message (i,t) au temps t :
 pour chaque modèle d dans D faire :
  envoie un i-message (i,t) au fils d
 t_l = max{t_ld | d ∈ D}
 t_n = min{t_nd | d ∈ D}

Réception ⋆-message (∗,t) au temps t :
 si (t ≠ t_n) alors :
  Erreur : mauvaise synchronisation
 d∗ = premier(liste_event)
 envoie *-message (∗,t) à d∗
 t_l = t
 t_n = min{t_nd | d ∈ D}

Réception x-message (x,t) au temps t avec x en entrée :
 si !(t_l ≤ t ≤ t_n) alors :
  Erreur : mauvaise synchronisation
 //consultation du couplage externe pour obtenir les fils
   influencés
 receveur = {r | r ∈ D, MC ∈ I_r, Z_{MC,r}(x) ≠ ∅}
 pour chaque r dans receveur :
  envoie x-message (x_r,t) avec x_r = Z_{MC,r}(x) à r
 t_l = t
 t_n = min{t_nd | d ∈ D}

Réception y-message (y_d∗,t) au temps t de la part de d∗ :
 si d∗ ∈ I_MC et Z_{d∗,MC}(y_d∗) ≠ ∅ alors :
  envoie y-message (y_MC,t) avec y_MC = Z_{d∗,MC}(y_d∗) au parent
 receveur = {r | r ∈ D, d∗ ∈ I_r, Z_{d∗,r}(y_d∗) ≠ ∅}
 pour chaque r dans receveur :
  envoie x-message (x_r,t) avec x_r = Z_{d∗,r}(y_d∗) à r
```

Comme il est montré sur l’algorithme 2, un composant coordinateur répond et envoie les

mêmes types de messages qu'un composant simulateur.

Quand le composant coordinateur reçoit un i-message, il le transmet à ses fils. Quand chacun des fils à traité ces i-messages, il affecte son temps d'occurrence du précédent événement au maximum des temps d'occurrence des précédents événements de ces fils et son temps d'occurrence du prochain événement au minimum des temps d'occurrence des prochains événements de ces fils.

Quand le composant coordinateur reçoit un *-message, il est transmis au fils le plus imminent *(le premier de la liste* $liste_{event}$*)*. Le fils exécute alors une transition interne d'état et pourra envoyer le message (y,t) en amont. Suite à cette exécution, et à l'exécution de tous les événements externes causés par la sortie du composant imminent, le temps du prochain événement est calculé comme le minimum du prochain événement de ce fils.

Lorsque le coordinateur reçoit lui-même un x-message de ses coordinateurs parents, il consulte les couplages d'entrée externes du réseau DEVS pour générer le message d'entrée approprié pour son subordonné influencé par l'événement externe. Les messages d'entrée sont envoyés à tous les fils r influencés par l'entrée externe x. Enfin, la mise à jour temporelle est effectuée.

Quand le composant coordinateur reçoit un message de sortie (y,t) il consulte :

- le couplage externe d'entrées du réseau DEVS pour voir si le message doit être envoyé à ses parents coordinateur,
- le couplage interne pour obtenir les fils, ainsi que les ports d'entrées.

Dans le premier cas le message y est envoyé au père du coordinateur. Dans le second cas, le message de sortie (y,t) est converti en un message d'entrée (x,t) à destination des fils sélectionnés.

Algorithme 3 Algorithme du coordinateur *"Root"*.

```
Variables :
 t //temps de simulation
 fils //subordonné directe
 tn // temps du prochain événement du fils

 envoie un i-message (i,t) au fils
 boucle jusqu'à la fin de la simulation :
  envoie un *-message(*,t) au fils
  t = tn
```

Comme il est montré sur l'algorithme 4, le coordinateur *"Root"* gère le temps de simulation global t. Au début de la simulation, il envoie à son fils un message d'initialisation. Tant que la simulation n'est pas terminée *(temps de simulation atteint une valeur maximale)*, le *"Root"* envoie à son fils direct un *-message $(*,t)$.

2.3.3 Conclusion

Le formalisme DEVS est utilisé pour la description de systèmes à événements discrets. Il constitue un outil de modélisation et de simulation permettant de représenter de manière modulaire et hiérarchique les systèmes réagissant à des événements discrets et de les simuler. Bien que le formalisme DEVS dit *"classique avec ports"* ai été étendu vers le formalisme dit *"parallèle"* nos travaux sont basés sur la première version.

Les raisons essentielles pour lesquelles nous utilisons le formalisme DEVS sont :

- Il permet la modélisation d'un système sur *plusieurs niveaux* de description.
- Il permet la *distinction* entre la modélisation et la simulation d'un système.
- Il permet la simulation des systèmes *automatiquement* à partir de ces modèles.
- Il permet de simuler des modèles *continus*.
- Il rend *facile* la modification du comportement des modèles.

2.4 Conclusion

Dans ce chapitre nous avons présenté le domaine du test des circuits digitaux dans lequel nous avons mis en évidence la nécessité d'implémenter un simulateur concurrent de fautes comportementales de haut niveau. Ensuite, nous avons présenté la SCC afin de mettre en évidence les concepts et les algorithmes de la simulation concurrente que nous allons utiliser. Enfin, nous avons décrit en détail le formalisme DEVS car il constitue la base de notre démarche qui consiste en représenter et simuler les comportements fautifs de manière concurrente au sein des systèmes discrets.

Dans la suite de ce mémoire, nous montrons comment les concepts et les algorithmes concurrents propres à la SCC peuvent être interprétés dans le formalisme DEVS pour accomplir la SFC des systèmes discrets. Le domaine d'application que nous avons choisi d'étudier plus particulièrement est celui de la simulation de fautes concurrente sur des circuits digitaux décrits en langage VHDL comportemental.

CHAPITRE 3

Le formalisme BFS-DEVS

3.1 Introduction

La simulation de fautes concurrente au niveau porte logique est âgé d'à peu près 20 ans et c'est l'une des seules applications existantes qui utilise la simulation concurrente. Cependant la SCC peut être appliquée dans plusieurs domaines comme [Ulrich et al., 1994] : le contrôle de trafic aérien, le contrôle des centrales nucléaires, l'analyse de graphes, la simulation symbolique, etc.

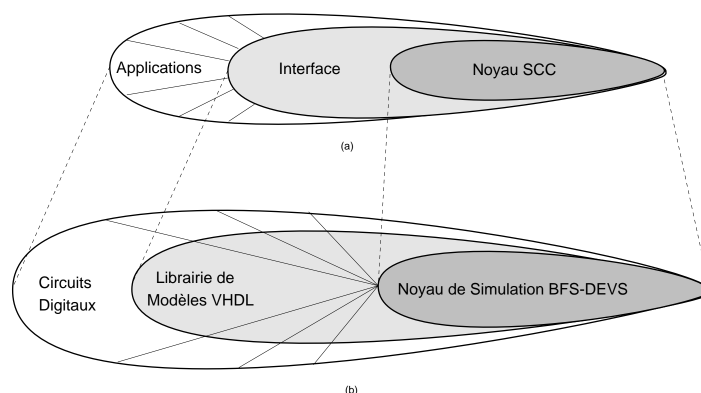


FIG. 3.1*: Intégration du formalisme BFS-DEVS.*

La figure 3.1 (a) illustre la relation entre le noyau de la SCC et ses domaines d'applications.

Les algorithmes du noyau de la SCC sont suffisamment généraux pour permettre la simulation des applications indépendamment de l'environnement dans lequel ils sont modélisés. Cependant, la relation entre le noyau et le domaine d'application est réalisée grâce à une *interface* de modélisation indispensable et spécifique au domaine d'application.

Les avantages de la SCC peuvent être mis en œuvre à partir du formalisme DEVS qui permet la séparation explicite des noyaux de modélisation et de simulation. Ce formalisme permet de modéliser des systèmes discrets et constitue l'interface idéale entre le noyau de simulation concurrent et le domaine d'application auquel appartient le système. De plus, le formalisme DEVS offre un noyau de simulation auquel nous pouvons facilement intégrer les algorithmes concurrents de la SCC.

Afin d'appliquer les algorithmes de la SCC dans le domaine de la simulation de fautes comportementales pour des systèmes discrets, nous définissons un noyau de simulation générique appelé *"BFS-DEVS" (Behavioral Fault Simulation for DEVS)*. L'utilisateur désireux de simuler les comportements fautifs d'un système construira une librairie de modèles spécifique au domaine d'étude sans se préoccuper de la partie simulation. Comme il est montré sur la figure 3.1 (b), il est possible de construire une librairie de modèles pour le domaine des circuits digitaux afin de simuler de manière concurrente les fautes comportementales au sein de ces systèmes.

Cette approche qui sépare le noyau de simulation de la représentation des modèles permet :

- d'**intégrer** les algorithmes de la SCC indépendamment des domaines d'applications,
- de **définir** des modèles spécifiques sans se soucier des algorithmes de simulation,
- de **spécifier** facilement le comportement fautif d'un modèle *(ou faute)*,
- de **réutiliser** les modèles définis dans les librairies.

Chaque application possède sa librairie de composant BFS-DEVS construite par l'utilisateur. Comme il est montré sur la figure 3.2, elle est composée de modèles BFS-DEVS *(atomiques et/ou couplés)* dont l'association constituera le modèle de l'application à simuler. Sa construction nécessite la connaissance d'un modèle de fautes indispensable à la définition des comportements fautifs des modèles. Elle nécessite également la représentation du système étudié en une interconnexion d'éléments *(structure de contrôle)* qui manipulent des données *(structure de données)* ainsi que des règles comportementales associées à ces deux structures. Ces règles permettent de définir le nombre d'éléments de base de la structure, leurs interfaces *(entrées/sorties)* et leurs comportements généraux.

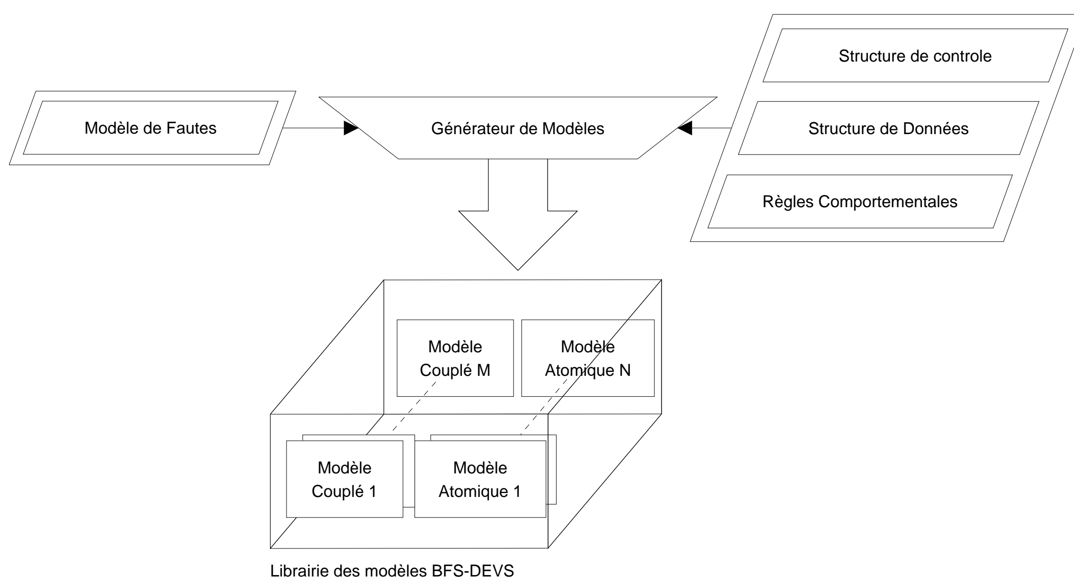


FIG. 3.2*: Construction de la librairie de modèles BFS-DEVS.*

La détermination de la structure de contrôle et de données d'une application demande de l'expérience. Elle découle de la modélisation à événements discrets et sera intégrée dans le nombre, le comportement et le couplage des modèles BFS-DEVS de la librairie. La construction d'une librairie propre a un domaine n'est donc pas une chose facile mais est la seule phase délicate de notre approche. En effet une fois cette tâche accomplie, le formalisme BFS-DEVS assure de manière automatique la simulation des modèles de la librairie.

3.2 Modèle de fautes comportementales

Une faute comportementale ou *une faute d'état* est représentée par [Kofman et al., 2000] comme une modification des fonctions de transitions et/ou des fonctions de sortie d'un modèle DEVS. Une faute a donc une influence sur le comportement *(ou l'état)* du modèle et peut conduire à un comportement que nous qualifierons de *"fautif"*. Nous pouvons alors définir **un modèle de fautes comportementales** comme l'ensemble des types de fautes qui modifie le comportement d'un modèle. Les fautes dépendent fortement de la *nature du système physique* que l'on veut modéliser, et elles doivent être les plus représentatives des défauts comportementaux. Le modèle de fautes dépend fortement des structures de contrôle et de données de l'application. Une fois choisi, il sera intégré au travers des comportements de chaque modèle BFS-DEVS. Dans [Kofman et al., 2000], l'auteur propose également un modèle de *fautes structurelles* permettant de considérer également des modifications au sein des couplages des modèles DEVS. De plus, le modèle de faute qu'il propose est évolutif dans le sens ou les hypothèses de fautes peuvent changé au cours de la simulation. Notre modèle de fautes ne considère aucun défaut structurel et reste fixe au cours de la simulation. Nous pouvons alors donner les définitions suivantes :

- **La signature d'une faute** ou *trace d'une faute* permet de résumer l'effet de la faute sur le comportement des modèles qu'elle a pu rencontrer. Chaque faute possède une signature dans laquelle est stockée le comportement fautif de *tous* les modèles qu'elle a contaminé au sein du réseau durant la simulation concurrente.
- **Une faute détectable**, est une faute dont *la signature* contient un comportement fautif du modèle différent de son comportement en l'absence de la faute.
- **Une faute localement observable** est une faute qui par sa présence s'observe localement sur un modèle par une modification de son comportement. Une faute reste localement observable tant que le comportement fautif reste différent du comportement sain.
- **L' hypothèse de faute unique** consiste à dire que les réseaux des modèles simulés ne peuvent être affectés que par une seule faute à la fois. Cependant, c'est la technique de *simulation concurrente BFS-DEVS basée sur la propagation des listes de fautes* qui nous permet de simuler plusieurs fautes tout en respectant cette hypothèse.

3.3 Le formalisme BFS-DEVS

3.3.1 La modélisation BFS-DEVS

Le nouveau formalisme BFS-DEVS permet de spécifier le comportement fautif d'un modèle. Les spécifications d'un modèle atomique BFS-DEVS sont équivalentes à celle d'un modèle atomique DEVS ajoutées des règles suivantes :

- la transition dans un **nouvel état fautif** résulte de l'arrivée sur un port d'une **valeur fautive**,
- le comportement fautif est spécifié à l'intérieur d'une **nouvelle fonction de transition externe fautive** δ_{fault}.

Un modèle atomique DEVS n'admet pas de comportement fautif et est considéré comme *"sain"*. Il est représenté par la structure classique décrite dans la partie 2.3.1.1 :

$$MA = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, t_a >$$

Afin de prendre en compte le comportement fautif d'un tel modèle, nous modifions sa structure de la façon suivante :

$$MA' = < X', Y', S', F, \delta'_{int}, \delta'_{ext}, \delta_{fault}, \lambda', t'_a >$$

ou,

- $X' = X \cup X^f$ est la liste des ports et des valeurs d'entrées avec $X^f = \{(p, v_f)\}$ le nouvel ensemble des ports et des *valeurs fautives* d'entrées.
- $Y' = Y \cup Y^f$ est la liste des ports et des valeurs de sorties avec $Y^f = \{(p, v_f)\}$ le nouvel ensemble des ports et des *valeurs fautives* de sortie.
- $S' = S \cup S^f$ est la liste des états avec S^f le nouvel ensemble des *états fautifs* pris par le modèle lorsqu'il reçoit une valeur fautive.
- $F = \{F_i\} \cup \oslash$, $i \in \mathbb{R}^+$ est l'ensemble des fautes F_i.
- $\delta'_{int} : S' \times F \rightarrow S$ est la fonction de transition interne avec la restriction : $\delta'_{int}(s, \oslash) = \delta_{int}(s)$.

- $\delta'_{ext} : Q \times X' \times F \rightarrow S$ est la fonction de transition externe qui vérifie : $\delta'_{ext}(s, \oslash, e, x) = \delta_{ext}(s, e, x)$ si $x \in X$.
- $\delta_{fault} : Q \times X^f \times F \rightarrow S^f$ est la *fonction de transition externe fautive* amenant le système dans un état fautif $s_f \in S^f$.
- $\lambda' : X' \times F \rightarrow Y'$ est la fonction de sortie qui vérifie : $\lambda'(s, \oslash) = \lambda(s)$.
- $t'_a : S' \times F \rightarrow \mathbb{R}^+_\infty$ est la fonction d'avancement du temps avec la restriction : $t'_a(s, \oslash) = t'_a(s)$.

Les spécifications BFS-DEVS sont similaires à celles du formalisme DEVS. La différence apparaît lorsque l'on considère l'arrivée d'une valeur fautive x_f $(x_f \in X^f)$ sur le modèle. Dans ce cas, le nouvel état fautif est déterminé par la fonction de transition externe fautive δ_{fault}. Si une valeur sa*ine* x $(x \in X)$ arrive sur le modèle, le nouvel état du système est toujours déterminé par la fonction de transition externe δ'_{ext}.

Les spécifications sur le couplage d'un modèle BFS-DEVS sont identiques à celle d'un modèle DEVS. Cependant, si le modèle de fautes utilisé contient des fautes pouvant modifier la structure du modèle *(fautes structurelles)*, le couplage peut être modifié au cours de la simulation [Kofman et al., 2000]. Nous considérons uniquement les fautes comportementales et la propriété de *"closure under coupling"* permettant la hiérarchie de composition entre les modèles BFS-DEVS est préservée.

3.3.2 La simulation BFS-DEVS

Nous présentons les algorithmes de simulation BFS-DEVS permettant d'effectuer une SFC sur des systèmes discrétisables. C'est en modifiant les algorithmes de simulation DEVS 1 et 2 de la partie 2.3.2 que nous offrons à l'utilisateur la possibilité de modéliser et de simuler de manière concurrente des fautes au sein d'un réseaux de composants BFS-DEVS grâce à une technique de propagation de liste de fautes.

3.3.2.1 Protocole de communication

C'est grâce à l'introduction d'un nouveau type de message (x_f, t) **représentant un événement externe fautif que nous sommes capable d'activer le comportement fautif d'un modèle**.

En effet, nous savons que la communication entre composant dans la hiérarchie de simulation se fait par le biais de quatre types de messages. Si nous voulons distinguer le comportement fautif au sein de la simulation il faut définir un nouveau type de message qui, au même titre que le *x-message* (x,t), permettra d'activer le modèle lorsqu'un événement fautif x_f est présent sur ces ports à l'instant t. Le coordinateur, par la connaissance de l'ensemble des valeurs d'entrées X', aura alors la possibilité d'envoyer deux types de messages externes :

- un *x-message* sur ces fils imminents si ceux-ci doivent avoir un comportement sain,
- un *f-message* sur ces fils imminents si ceux-ci doivent avoir un comportement fautif.

Grâce à cette distinction nous sommes capable d'effectuer les simulations saines et fautives en concurrence. Dans le cas d'une propagation concurrente, les expériences fautives *(C-expériences cf. partie 2.2.2)* se distinguent de l'expérience saine *(R-expérience cf. partie 2.2.2)* soit parce quelles empruntent un chemin différent *(chemin fautif)*, soit parce qu'elles modifient des données tout en gardant le même chemin que l'expérience saine *(chemin sain)*. Dans le premier cas, seul un *f-message* est propagé entre les modèles, dans le second cas un message sain *x-message* et un message fautif *f-message* sont propagés en même temps.

Sur un chemin sain la simulation de fautes ne peut se faire que si les valeurs saines sont connues, ce qui implique que les *x-messages* doivent être traités avant les *f-messages.* Dans ce cas les valeurs saines sont alors connus et peuvent servir de référence à la simulation de fautes. Par conséquent, sur un chemin sain la simulation de fautes est précédée de la simulation saine. C'est donc la nature des événements externes qui permet de distinguer les chemins de simulation.

3.3.2.2 Modification algorithmique du composant simulateur

La fonction principale d'un composant simulateur étant la gestion comportementale du modèle atomique auquel il est attaché, nous avons défini une nouvelle fonction de transition externe δ_{fault} qui sera exécuté à la réception d'un *f-message.* Bien que la représentation d'un comportement fautif du modèle implique la modification des fonctions de transition et de sortie *(essentiellement pour la détermination des chemins empruntés par les messages de sorties)*, c'est à l'intérieur de cette fonction de transition δ_{fault} que l'on exprime entièrement le comportement fautif du modèle.

Comme il est montré sur l'algorithme 4, c'est la seule modification apportée au sein du composant simulateur.

Algorithme 4 Compléments algorithmiques d'un simulateur.

```
Variables :
 parent
 t_l
 t_n
 DEVS = {X', Y', S', F, δ'_int, δ'_ext, δ_fault, λ', t'_a}
 y

Réception i-message (i,t) au temps t :
 inchangé (algorithme 1)

Réception *-message (*,t) au temps t :
 inchangé (algorithme 1)

Réception x-message (x,t) au temps t avec x en entrée :
 inchangé (algorithme 1)

Réception f-message (x_f,t) au temps t avec x_f en entrée :
 e = t − t_l
 s = δ_fault(s, e, x_f)
 t_l = t
 t_n = t_l + t_a(s)
```

3.3.2.3 Modification algorithmique d'un composant coordinateur

Comme il est montré sur l'algorithme 5 de la page suivante, la réception d'un *f-message* présente la même structure que la réception d'un *x-message* puisque ce sont des messages générés lors de l'arrivée d'un événement externe.

Algorithme 5 Compléments algorithmiques d'un coordinateur.

Variables :
inchangé (algorithme 2)

Réception i-message (i,t) **au temps t :**
inchangé (algorithme **2)**

Réception ⋆-message $(*,t)$ **au temps t :**
inchangé (algorithme **2)**

Réception x-message (x,t) **au temps t avec** x **en entrée :**
inchangé (algorithme **2)**

Réception y-message (y_{d*},t) **au temps t avec de la part de** $d*$ **:**
si $d* \in I_{MC}$ et $Z_{d*,MC}(y_{d*}) \neq \emptyset$ alors :
envoie y-message (y_{MC},t) avec $y_{MC} = Z_{d*,MC}(y_{d*})$ au parent
$receveur = \{r \,|\, r \in D, d* \in I_r, Z_{d*,r}(y_{d*}) \neq \emptyset\}$
pour chaque r dans *receveur* :
envoie un x-message (x_r,t) ***et un f-message*** $(\emptyset,t)$ ***avec*** $x_r = Z_{d*,r}(y_{d*})$ ***à r si*** $x_r \in X$
envoie un f-message (x_r,t) ***avec*** $x_r = Z_{d*,r}(y_{d*})$ ***à r si*** $x_r \in X_f$

Réception f-message (x_f,t) ***au temps t avec*** x_f ***en entrée :***
si $!(t_l \leq t \leq t_n)$ alors :
Erreur : mauvaise synchronisation
$receveur = \{r \,|\, r \in D, MC \in I_r, Z_{MC,r}(x_f) \neq \emptyset\}$
pour chaque r dans receveur :
envoie f-message (x_r,t) avec $x_r = Z_{Mc,r}(x_f)$ à r
$t_l = t$
$t_n = min\{t_{nd} \,|\, d \in D\}$

Lorsqu'un coordinateur reçoit un *f-message* il le transmet aux fils imminents appartenant à D. Concernant la réception d'un *f-message*, une partie supplémentaire doit être ajoutée à l'algorithme de base, cet ajout est justifié par le fait que l'on parallélise l'exécution des simulations saine et fautives. Si l'événement en entrée du receveur r et sain *(message du type sain sur chemin sain,* $x_r \in X$*)* alors un *x-message* (x_r,t) est suivit d'un *f-message* $(\emptyset,t)$ assurant ainsi la parallélisation des simulations. Le *f-message* ne contient pas de valeur fautive car il sert uniquement à la construction de liste de fautes sur le chemin sain. Si l'événement en entrée du receveur est fautif *(message du*

type fautif sur chemin faux, $x_r \in X_f$*)* alors un *f-message* est envoyé. Dans ce cas, le message fautif contient une valeur x_f qui est la liste des fautes à analyser sur le(s) chemin(s) fautif(s).

3.4 Mécanisme de Simulation BFS-DEVS

Le mécanisme de la simulation BFS-DEVS s'appuie sur les notions de la SCC *(cf. partie 2.2.2)* pour accomplir la simulation concurrente de fautes au sein des modèles BFS-DEVS. **Ce mécanisme consiste en une simulation saine *(R-expérience)* concurrencée par plusieurs simulations fautives *(C-expériences)* induites par des fautes comportementales pouvant intervenir à l'intérieur des modèles**. L'une des propriétés importantes de ce mécanisme est qu'il utilise une technique de propagation de liste de fautes permettant le rassemblement des simulations fautives pour une meilleur observabilité.

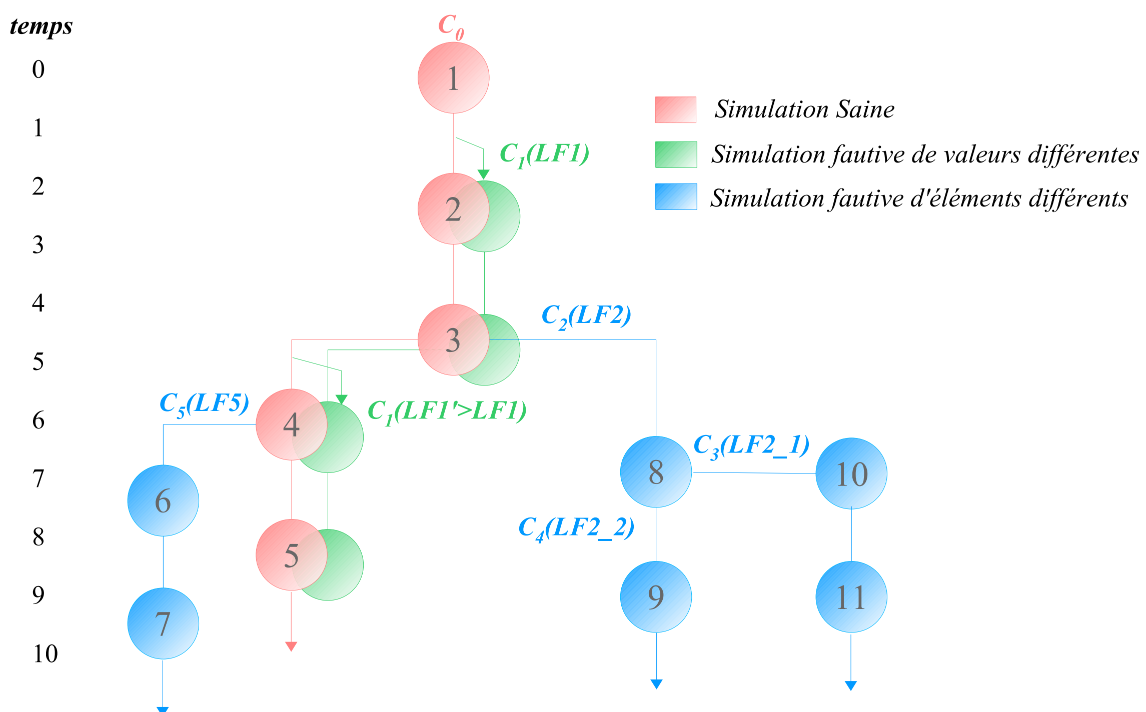


FIG. 3.3*: Technique de propagation de listes de fautes.*

La figure 3.3 montre la génération des $i \in \mathbb{N}^+$ simulations fautives $C_{i\neq0}$ à partir de l'unique simulation saine C_0. Il existe deux manières de caractériser l'origine d'une simulation concurrente :

– soit elle se distingue par des *valeurs différentes* de la simulation de référence tout en gardant la *même activité* que cette dernière *(en vert sur la figure 3.3)*,
– soit elle se distingue par des *éléments différents* de ceux empruntés par la simulation de référence et possède alors une *activité distincte (en bleue sur la figure 3.3)*.

Dans tous les cas, l'origine d'une simulation fautive est toujours dû à la possibilité qu'il apparaisse *une différence (faute)* sur un *élément* ou sur une *donnée* de la simulation saine. Par conséquent, chaque simulation fautive possède sa propre faute qu'elle doit propager. Cependant, il est possible que plusieurs simulations fautives possèdent la même activité. Dans ce cas, ces simulations peuvent être rassemblées en **une unique simulation fautive** contenant **une seule liste de fautes**. Au passage de chaque élément, cette liste peut *augmenter* ou être *découpée* suivant l'activité de cette simulation fautive :

– lorsque la simulation fautive *d'activité commune* diverge de la simulation saine les fautes misent en évidence sont **insérées** dans la liste de faute propagée sur le même chemin que la simulation saine. Comme il est montré sur la figure *3.3*, la première simulation fautive de valeur différente $C_1(LF1)$ à débuté au temps 2 sur l'élément 2 avec la liste de fautes $LF1$. A chaque fois qu'une autre simulation concurrente de valeur différente à C_0 diverge *(au temps 5 sur la figure* 3.3*)*, elle est fusionnée à C_1 et la liste de fautes quelle propage est intégrée à $LF1$ pour donner $LF1'$.
– lorsqu'une simulation fautive *d'activité distincte* diverge de la simulation saine avec une liste de fautes primaire LFi, elle peut impliquer $j \in \mathbb{N}^+$ simulations fautives propageant les listes LFi_j tel que $LFi_j \subset LFi \forall i, j \in \mathbb{N}^+$. Cette propagation des simulations fautives par **découpage et réorientation des listes de fautes** peut s'effectuer jusqu'à l'obtention de liste composée d'une seule faute soit $LFi_j \geq 1$. Comme il est montré sur la figure *3.3*, la simulation fautive d'éléments différents $C2(LF2)$ qui débute au temps 5 à partir de l'élément 3 se scinde en deux simulations fautives $C_3(LF2_1)$ et $C_4(LF2_2)$ à partir de l'élément 8. C'est l'évaluation des fautes de la liste initiale $LF2$ sur l'élément 8 qui permet le découpage et la réorientation de $LF2$ en deux sous listes $LF2_1$ et LF_2.

L'utilisation de la technique de propagation des listes de fautes constitue un avantage car elle

permet le rassemblement des simulations fautives d'activités identiques. La phase d'observabilité des résultats est simplifiée car elle consiste à analyser les signatures des fautes appartenant à une seule simulation fautive. De plus, cette technique permet la propagation des simulations fautives par simple découpage et réorientation des listes de fautes.

Nous allons représenter les éléments de la figure *3.3* par des modèles BFS-DEVS. Nous posons les définitions suivantes :

- **Un message sain** *(resp. fautif)* est un objet permettant la communication et l'exécution des modèles BFS-DEVS pour une simulation saine *(resp. fautive)*. Il permet également la propagation des listes de fautes dans un réseau BFS-DEVS.
- **Un chemin sain** ou *chemin de référence* est la suite des modèles BFS-DEVS *(atomiques et/ou couplés)* exécutés pour une simulation saine du réseau BFS-DEVS. Schématiquement, une simulation saine est représentée par le chemin sain du réseau BFS-DEVS.
- **Un chemin fautif** ou *chemin concurrent* est la suite des modèles BFS-DEVS *(atomique et/ou couplés)* exécutés pour une simulation fautive du réseau BFS-DEVS. Schématiquement, une simulation fautive est un chemin fautif du réseau BFS-DEVS. De la même manière que pour la simulation fautive, un chemin fautif d'un réseau BFS-DEVS peut se ramifier en plusieurs chemins fautifs concurrents.

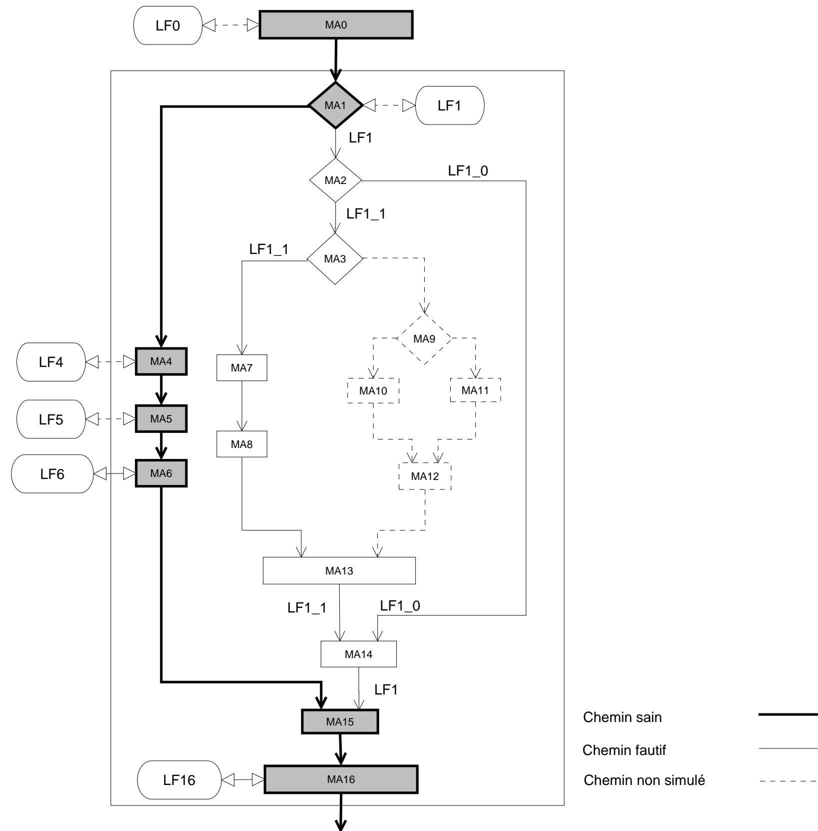


FIG. 3.4*: Simulation de fautes concurrente d'un réseau BFS-DEVS.*

Comme il est montré sur la figure *3.4*, la simulation saine exécute les modèles BFS-DEVS $MA0$, $MA1$, $MA4$, $MA5$, $MA6$, $MA15$ et $MA16$ *(en gris sur la figure 3.4)* et définit ainsi le chemin sain *(en gras sur la figure 3.4)*. Chacun de ces modèles est à l'origine des simulations fautives propageant les listes de fautes : $LF0$ pour $MA0$, $LF1$ pour $MA1$ avec $LF0 \subset LF1$, etc. Ces simulations ayant *la même activité* que la simulation saine, elle sont représentées par une unique simulation fautive propageant la liste de fautes LF dont la taille augmente au cours de la simulation *(*$LF = LF0 \cup LF1 \cup LF4 \cup LF5 \cup LF6 \cup LF16$*)*.

Les simulations fautives exécutent des modèles atomiques BFS-DEVS n'appartenant pas au chemin sain *(en blanc sur la figure 3.4)* . Les chemins fautifs *(en lise sur la figure 3.4)* sont obtenus par dérivation du chemin sain. La liste $LF1$ est propagée, via un message fautif, sur un chemin fautif afin de construire les signatures des fautes pour chacun des modèles BFS-DEVS *(par exemple* $MA7$ *et* $MA8$ *de la figure).* Cependant, cette liste subit des découpages et des réorientations le long de son chemin fautif. En effet, les fautes appartenant à $LF1$ évaluées au niveau des modèles $MA2$

et $MA3$ n'impliquent pas toutes le même comportement fautif pour ces modèles. Par conséquent, la liste de fautes principale $LF1$ est divisée en deux sous-liste $LF1_0$ et $LF1_1$. C'est lorsque les chemins fautifs se rejoignent au niveau des modèles $MA15$ et $MA16$ que les fautes propagées sur ces chemins sont fusionnées et que leurs signatures sont analysées pour construire la liste des fautes détectées. Enfin, la liste de fautes $LF0$ sera analysée à la fin de la simulation saine c.à.d. lorsque les comportements fautifs de tous les modèles BFS-DEVS auront été simulés.

3.5 Conclusion

Ce chapitre nous a permis de définir les algorithmes de modélisation et de simulation du *formalisme BFS-DEVS*. Ce formalisme constitue une *interface idéale* pour accomplir la simulation concurrente des fautes comportementales d'un système appartenant à un domaine d'application. Il donne la possibilité de construire une *librairie de modèles spécifiques* au domaine étudié et il permet également d'*intégrer facilement les algorithmes de la SCC* dans son noyau de simulation. Après avoir défini le modèle de fautes utilisé, il est possible de spécifier le comportement fautif des modèles de la librairie par le biais d'une nouvelle fonction de transition δ_{fault}. L'intégration des algorithmes de la SCC au noyau de simulation BFS-DEVS nous permet de simuler les fautes comportementales de manière concurrente au sein des modèles. Cette simulation s'appuie sur une technique de propagation de listes de fautes au travers du réseau de modèles BFS-DEVS. Cette technique permet le rassemblement des simulations fautives pour une meilleure gestion de la simulation et facilite également l'observabilité des résultats en fin de simulation.

CHAPITRE 4

Le formalisme BFS-DEVS appliqué aux circuits digitaux

Le domaine d'application choisi pour montrer la faisabilité de notre approche est celui de la simulation de fautes comportementales au sein des circuits digitaux décrits en VHDL. Nous débutons ce chapitre par une breve introduction au VHDL et nous présentons le sous ensemble de ce langage choisi pour notre étude : les descriptions comportementales avec instructions séquentielles. Nous abordons ensuite la modélisation des circuits digitaux décrits dans ce sous ensemble vers le formalisme BFS-DEVS. Nous expliquons comment construire la libraire de composants spécifique VHDL BFS-DEVS et nous définissons le modèle de fautes comportementales à partir duquel il est possible de spécifier le comportement fautif de ces composants. Enfin, nous nous appuyons sur l'exemple d'un registre 8 bits pour montrer comment la technique de propagation de liste de fautes permet d'accomplir la simulation BFS-DEVS au sein des circuits digitaux décrits en VHDL.

4.1 La modélisation BFS-DEVS

4.1.1 Le Langage VHDL

4.1.1.1 Bref historique des langages de description du matériel

Les approches modernes de la conception des circuits *(logiques)* électroniques nécessitent que l'on puisse décrire, d'une manière la plus abstraite possible, la fonctionnalité souhaitée pour ces montages. D'un point de vue externe, la syntaxe de ces formalismes ressemble à celle des langages de programmation classiques, mais leur sémantique est liée au comportement des montages électroniques. Après une vingtaine d'années de recherches, de tels formalismes ont été normalisés dans les années 90 *(langages VHDL et Verilog)*. D'un point de vue industriel, ces langages de description de matériel électronique sont une nécessité et ils sont très utilisés. Ils permettent de simuler les circuits avant leur réalisation, d'échanger des descriptions de circuits, de constituer des bibliothèques de modules, de préciser la spécification d'un circuit et d'alimenter des outils automatiques de conception. Il existe un marché *(dit d'Intellectual Property)* pour des descriptions "synthétisables" de blocs internes complexes de circuits intégrés. La description préalable du comportement d'un futur circuit devient une étape obligée dans son processus de conception. Le passage du fer à souder à la description para-informatique des futurs circuits constitue un véritable bouleversement dans les habitudes des électroniciens qui voient leur métier évoluer profondément et adopter des méthodes de raisonnement inspirées de celles des informaticiens. L'existence d'un formalisme précis permet aussi de réaliser des traitements formels sur les descriptions des futurs circuits tels que des transformations, des optimisations et surtout des vérifications. Par exemple, les années 80 et 90 ont vu apparaître des outils capables de vérifier que les comportements de deux descriptions sont formellement équivalents *(outils V-Formal et Chrysalys)*. Le langage VHDL résulte d'un effort conjoint des compagnies Intermetrics, IBM et Texas dans les années 80 sous l'égide du DoD *(Ministère de la défense des USA)*. Le résultat de cet effort a été normalisé en 1987 *(norme IEEE 1076)*. Curieusement, VHDL est surtout utilisé en Europe et c'est l'un de ses concurrents : Verilog qui est le plus utilisé aux USA.

4.1.1.2 Bref notion du langage VHDL

VHDL [Coelho, 1989, Ashenden, 2001] est l'acronyme de VHSIC *(Very high speed integrated circuits* Hardware Description Language) et a été développé dans le cadre du projet VHSIC commandité par le département de la défense U.S. C'est un standard IEEE depuis 1987 et est actuellement largement utilisé. Sa syntaxe reprend celle d'ADA et initialement a été développé pour spécifier de grands systèmes.

Le modèle VHDL d'un circuit est une *entité*. L'entité est constituée de deux composants, la description de l'*interface* et le corps de l'*architecture*. L'interface spécifie les ports du circuits, l'architecture son contenu. Le corps peut être décrit de trois façons : *structurelle*, *comportementale* ou sous forme de *flots de données*. Le modèle structurel décrit une interconnexions de modules. Le modèle comportemental contient des *process* qui contiennent des *instructions séquentielles*. **Tous nos travaux concerneront ce sous-ensemble comportemental du VHDL**. Le modèle flot de données est un style supplémentaire utilisé pour modéliser l'assignation parallèle de signaux. Le langage contient les notions de variables, signaux et constantes :

- **Les variables** ne sont pas directement liées à des notions de matériel ; elles sont utilisées de manière interne dans le modèle comportemental *(à l'instar de variables informatiques)* et leur **affectation est instantanée**.
- **Les signaux** sont une généralisation des variables et leur **affectation n'est pas instantanée**. En effet, l'exécution d'un instruction ne modifie pas instantanément la valeur d'un signal. Cette valeur est modifiée uniquement lorsque tous les processus ont été exécutés.
- **Les constantes** sont des objets qui possèdent une **valeur qui reste identique** tout au long de la simulation.

La notion de *temps symbolique (ou cycle symbolique)* est artificielle est n'a aucune durée physique mais elle permet de simuler l'exécution des processus en parallèle. A contrario, *le temps physique (ou cycle physique)* permet de caractériser l'évolution réelle des systèmes physiques. Le temps physique n'évolue pas pendant la simulation et un cycle physique peut être composé de plusieurs cycles symboliques suivant l'activité des signaux. Par rapport à ces notions de temps, les signaux utilisent des *pilotes de valeurs* pour stocker leurs valeurs courantes *(au cycle symbolique courant)*

et futures *(au cycle symbolique suivant)*. Lorsqu'un signal est affecté au cycle symbolique courant, la nouvelle valeur et le temps d'affectation sont stockées dans son pilote de valeur*s*. Cette valeur ne deviendra effective qu'au cycle symbolique suivant. En début de simulation, les pilotes de valeurs des signaux sont arbitrairement fixés à la valeur inférieur de leur ensemble de définition.

4.1.1.3 Sous ensemble étudié : les descriptions VHDL comportementales

Il s'agit d'une forme très courante de description VHDL. **C'est la forme dans laquelle les compilateurs transforment toutes les autres formes de description pour pouvoir les simuler et c'est pour cette raison que nous travaillons essentiellement sur ce sous-ensemble**. Une telle description consiste à décrire un, ou des, programmes dont l'exécution simule le comportement du circuit. Elle consiste en :

- la liste des signaux internes,
- la liste des programmes appelés *process* qui simulent tout ou une partie du circuit et s'exécutent en *parallèle*s. Ces programmes sont constitués de :
  - la déclaration des variables qu'ils utilisent *(qui ne sont que des intermédiaires de calcul)*,
  - les *instructions séquentielles* de simulation qui peuvent avoir des signaux et des variables comme arguments. Celles ci peuvent être :
    - des instructions d'*affectations*,
    - des instructions *conditionnelles ("if...then...else...endif", "case...when...end case")*,
    - des instructions de *boucles (boucle simple, boucle "for", boucle "while")*

Ces programmes s'exécutent normalement, c'est-à-dire de manière séquentielle. Leur temps d'exécution est supposé nul vis à vis du temps de simulation. Les conditions de démarrage d'un programme permettent de l'assimiler globalement à une super-instruction de connexion d'une description fonctionnelle. Ce démarrage s'effectue : soit parce qu'un signal spécifié dans une *liste de sensibilité* varie, soit parce que le programme était mis en attente par une instruction *wait* et que cette attente est échue.

Voici ci-contre la description VHDL comportementale d'un registre 8 bits. L'entité BUFF_REG d'écrit son interface qui est composée de quatre ports d'entrées portant les signaux DI de type bit_vector, STRB, DS1 et NDS2 de types bit. Il possède également un port de sortie représenté par le signal DO de type bit_vector.
Son architecture THREE_PROC possède trois processus STROBE, ENABLE et OUTPUT. La liste sensitive de STROBE est composée du signal STRB, DS1 et NDS2 font partie de la liste sensitive du processus ENABLE et les signaux internes REG et ENBLD sont chargés de l'activation du processus OUTPUT. Ces trois processus s'exécutent en parallèle.

```
entity BUFF_REG is
  port (DI : in bit_vector(1 to 8);

  STRB, DS1, NDS2 : in bit;
  DO : out bit_vector(1 to 8));
end BUFF_REG;

architecture THREE_PROC of BUFF_REG
is
  signal REG : bit_vector(1 to 8);
  signal ENBLD : bit;
begin
STROBE : process(STRB)
  begin
    if (STRB='1') then
      REG<=DI;
    end if;
end process STROBE;

ENABLE : process(DS1, NDS2)
  begin
    ENBLD<=DS1 and not NDS2;
end process ENABLE;

OUTPUT : process(REG, ENBLD)
  begin
    if (ENBLD='1') then
      DO<=REG;
    else
      DO<="11111111";
    end if;
end process OUTPUT;
end THREE_PROC;
```

4.1.2 Modèle de fautes proposé

De nombreux modèles de fautes comportementales de haut niveau associé à plusieurs techniques de simulation de fautes ont été proposées [Devadas et al., 1996, Buonanno et al., 1997, Thaker et al., 2000]. Cependant, aucune de ces techniques n'établit de manière formelle la relation qu'il existe entre les modèles de fautes à haut niveau et au niveau porte [Goloubeva et al., 2002]. Comme pour [Corno et al., 2000a], le modèle de fautes que nous proposons dans ce papier s'inspire des techniques du *test de logiciel*s. Il reste général et n'a pas été conçu pour valider une métrique particulière au niveau des langages de description du matériel. Il est basé principalement

sur le collage de valeurs des signaux et des variables présents dans une description VHDL comportementale. Afin de considérer les fautes au sein de la structure de contrôle de la description, nous simulons les fautes de collage de branches conditionnelles. Enfin, le saut d'instruction d'affectation est également pris en compte car il permet une analyse du code redondant présent dans les descriptions VHDL. Nous adoptons comme [Corno et al., 2000a] l'hypothèse de faute unique et permanente.

Nous utilisons donc trois types de fautes comportementales agissant sur les instructions présentes dans la description VHDL :

Les fautes de type F_1 :

Ce sont les fautes de collage de valeurs sur les objets *(signaux ou variable)* présents dans une description. Considérons l'instruction d'affectation suivante :

$$E\ :\ S <= (I1\,and\,I2)\,or\,I3$$

où $S, I1, I2$ et $I3$ sont du type T : $bit_vector\,(0\,downto\,1)$. L'évaluation de l'instruction E donne à S une valeur future "saine" notée $v_h(S)$. Toutes les fautes S_i de type F_1 sur ce signal impliquent une valeur "fautive" notée v_f. Ce sont les fautes de collages aux valeurs de l'espace de définition de S exclue de la valeur saine : $\{S_i | v_f \in T - \{v_h\}\}$. Si $v_h(S) =$ "01" ces fautes de collage sur le signal S sont stockées dans une liste $L_S = [S_{00}, S_{10}, S_{11}]$.

De la même façon, les fautes de type F_1 sur les signaux affectant qui impliquent une *évaluation fautive* de E sont déterminés en fonction de leurs valeurs courantes "saines". Si $v_h(I1) =$ "01",$v_h(I2) =$ "01" et $v_h(I3) =$ "00" alors : $L_{I1} = L_{I2} = L_{I3} = L_{Ij} = [Ij_{11}, Ij_{00}, Ij_{10}]\,\forall j \in \{1,2,3\}$. En effet, si une seule faute *(hypothèse de faute unique)* est injectée dans E, elle conduit à une valeur différente de la valeur saine pour S. Par exemple, $E(I1_{11}) = (\text{"11"}\,and\,\text{"01"})\,or\,\text{"00"} = \text{"11"}$ donc $E(I1_{11}) = \text{"11"} \neq \text{"01"} = v_h(S)$.

Les fautes de type F_2 :

Le type F_2 correspond aux fautes de collage de branches des instructions conditionnelles. Lorsqu'une instruction conditionnelle est évaluée, une branche conditionnelle "saine" est choisie parmi un ensemble fini de possibilité. Le collage des branches différentes de la branche "saine" constitue le type F_2. Considérons l'instruction conditionnelle suivante :

$$E\,:\,if\,(S == \text{"01"})\,then$$

où S est un signal de type T : $bit_vector\,(0\,downto\,1)$. Si le signal S possède la valeur courante saine $v_h(S) = \text{"11"}$, l'évaluation de E est fausse. Dans un tel cas, une faute de type F_2 est la faute qui implique l'évaluation de l'instruction de E à vrai quelque soit la valeur de S. Le nombre de possibilités admises par E étant égal à 2, si l'instruction est vraie *(resp. fausse),* elle correspond à la branche numéro 0 *(resp. 1)*. Cette faute est donc notée $F2_E_0$.

Ce type de faute s'applique également dans le cas où l'instruction conditionnelle utilise un "case". Il est une généralisation de l'exemple ci-dessus car il est possible de déterminer une liste de fautes dont la taille est égale au nombre de possibilité admissent par le "case". Considérons l'instruction suivante :

E : case S is
when "00" =>
when "10" =>
when "01" =>
when "11" =>

Le nombre de possibilités admises par le "case" est égal à 4 et les fautes de type F_2 seront numérotés de 0 à 3. Si le signal S possède la valeur courante $v_h(S) = \text{"01"}$, la branche "saine" est la branche numéro 2 et la liste des fautes de type F_2 impliquant une autre branche conditionnelle est la suivante : $[F2_E_0, F2_E_1, F2_E_3]$.

Les fautes de type F_3 :

Toutes les instructions d'affectation présentant une faute de type F_3 ne sont pas évaluées. Considérons l'instruction d'affectation suivante :

$$E : S <= \text{"00"}$$

où S est un signal de type $T : bit_vector(0\, downto\, 1)$ avec le pilote de valeur P_S tel que $P_S\{valeur\, courante = \text{"11"}, valeur\, future = \text{"01"}, cycle = 1\}$ au cycle 0. Si l'instruction d'affectation E est évaluée au cycle 1, la valeur future de S qui est égale à "01" devient égale à "00". Si on considère la faute de type F_3 sur E, cette instruction n'est pas évaluée et la valeur future de S reste inchangée. Comme il a été montré [Bisgambiglia et al., 2001], ce type de fautes permet la suppression d'instruction d'affectations redondantes au sein d'une description VHDL.

4.1.3 Transformation VHDL/BFS-DEVS

La phase de transformation VHDL/BFS-DEVS montrée sur la figure 4.1.

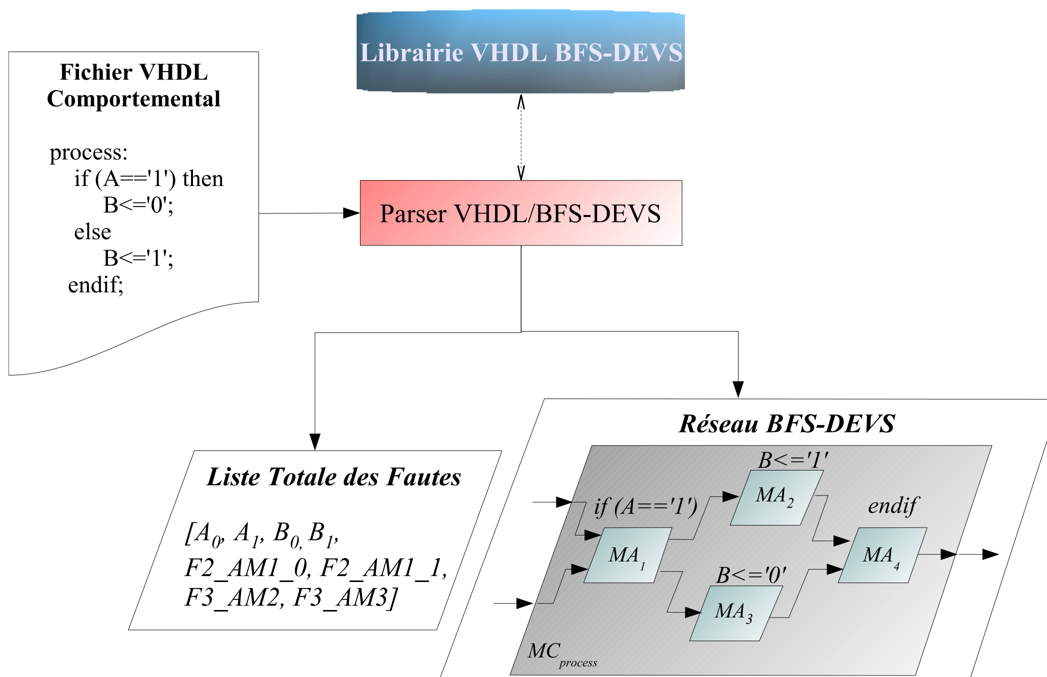


FIG. 4.1: *Schéma du* "parser" *VHDL/BFS-DEVS.*

Cette transformation permet de donner une représentation d'une description VHDL sous la forme d'un réseau de composants BFS-DEVS. Toute l'information sur la structure de contrôle et sur la structure de données du code VHDL est représentée dans l'interconnexion et le comportement des composants du réseau. C'est à partir de *la libraire de composants BFS-DEVS* spécifique au domaine des descriptions comportementales VHDL que le *"parser"* transforme le fichier VHDL en un *réseau de composants BFS-DEVS*.

Dans [Wainer et al., 2001, Wainer, 2004] le professeur Wainer montre que le formalisme DEVS est particulièrement adapté à la modèlisation et la simulation de systèmes digitaux. Il utilise le formalisme DEVS pour modèliser l'architecture d'une partie d'un processeur SPARC afin de rendre celle-ci plus simple du point de vu pédagogique. En ce qui nous concerne, l'utilisation du formalisme DEVS et plus particulièrement de la transformation VHLD/BFS-DEVS présente plusieurs avantages comme :

– **La spécification** des comportements fautifs des modèles,
– **L'adaptation** modulaire des modèles de fautes comportementales et/ou structurelles.
– **L'intégration** de la technique de propagation des listes de fautes,
– **La simplification** de l'analyse et de l'observabilité des résultats au cours de la simulation,

Le réseau BFS-DEVS de la figure 4.1 est obtenu grâce à un *"parser"* qui analyse le fichier VHDL en utilisant une librairie de modèles spécifique au domaine des descriptions VHDL comportementales. Cette librairie est construite par l'utilisateur en considérant que n'importe quelle description VHDL comportementale peut être représentée par l'association des quatre modèles BFS-DEVS de base [Capocchi et al., 2003] suivants :

– **le modèle atomique *"Assignment"*** représentant une instruction d'affectation VHDL *(MA_2 et MA_3, figure 4.1)*.
– **le modèle atomique *"Conditional"*** représentant une instruction conditionnelle VHDL *(MA_1, figure 4.1)*.
– **le modèle atomique *"Junction"*** associé au instruction de fin de code comme *"endif, endcase, etc" (MA_4, figure 4.1)*.
– **le modèle couplé *"Process"*** associé à l'instruction *"process"* VHDL *($MC_{process}$, figure* 4.1*)*.

Deux modèles atomiques BFS-DEVS supplémentaires sont ajoutés à la librairie pour les besoins de la simulation concurrente :

- **le modèle atomique *“Generator”*** : Il permet la génération d'événements destinés à exécuter les composants du réseau BFS-DEVS.
- **le modèle atomique *“ProcessEngine”*** : Il permet la gestion de la synchronisation des modèles couplés *“Process”* présents dans le réseau BFS-DEVS.

La librairie est donc composée de *six composants BFS-DEVS (décrits en détail en annexe)* et est mise à disposition du *“parser”* pour donner le réseau BFS-DEVS de n'importe quelle description VHDL comportementale. La figure 4.2 montre le réseau BFS-DEVS du registre 8 bits dont la description VHDL a été donnée dans la partie 4.1.1.3.

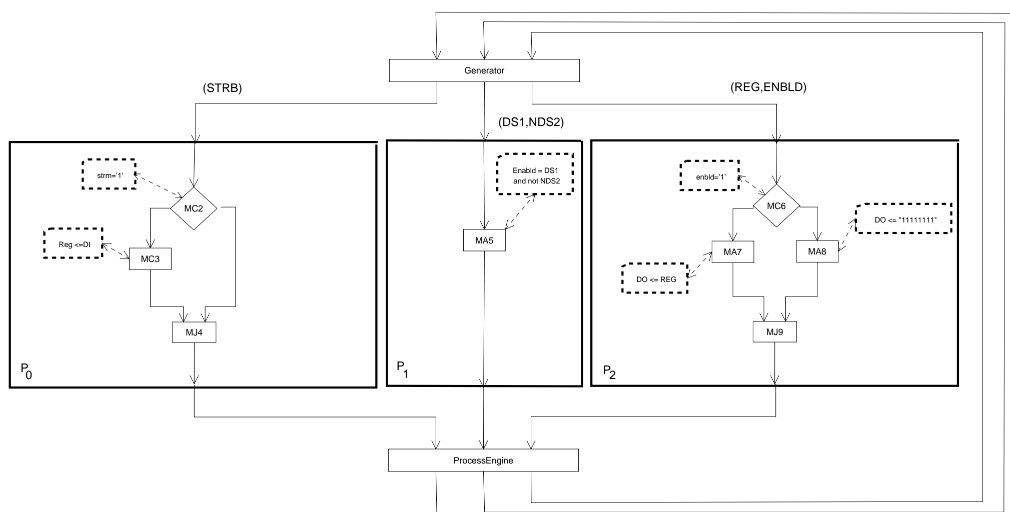


FIG. 4.2*: Réseau BFS-DEVS du registre 8 bits.*

Comme il est montré sur la figure 4.1, le *“parser”* fournit également la liste totale des fautes pouvant intervenir dans le réseau BFS-DEVS. Cette procédure nécessite également une règle de transformation du modèle de fautes proposé *(cf. partie 4.1.2)*. Dans le formalisme BFS-DEVS, la présence d'une faute au sein *d'un modèle atomique BFS-DEVS* représentant une instruction VHDL se traduit par une modification de son comportement *(cf. partie 3.2)*. Donc les trois types de fautes F_1, F_2, et F_3 sont transformées de la manière suivante :

- **La faute du type F1** sur un signal *(ou une variable)* appartenant à une instruction d'affectation conduit à une **transition d'état fautive** du modèle atomique d'affectation.

- **La faute du type F2** sur un modèle conditionnel conduit à l'exécution d'un des **ports de sorties différent** du port de sortie choisi pour la simulation saine.
- **La faute du type F3** sur un modèle d'affectation implique une **non exécution** de celui-ci.

4.2 La simulation concurrente BFS-DEVS

4.2.1 Définitions spécifiques aux descriptions VHDL comportementales

Cette section est consacrée à la redéfinition de quelques concepts déjà introduits dans la partie 2.2 mais également à l'introduction de nouvelles notions comme :

- **La signature d'une faute** correspond à la trace $T_{F_i} = \{((S|V)_j, P_{(S|V)_j})|i,j \in \mathbb{N}\}$ du passage de la faute F_i sur un chemin sain ou fautif. Elle est représentée par l'ensemble des couples $((S|V)_{j \in \mathbb{N}}, P_{(S|V)_j})$ avec :
  - S_j *(ou V_j)*, le signal influencé *(ou la variable influencée)* par la faute F_i,
  - $P_{(S|V)_j} = [v_{cf}, v_{ff}, c]$, le pilote des valeurs courantes fautives v_{cf} et future v_{ff} du signal S_j influencé *(ou de la variable V_j influencée)* par F_j au cycle c.
- **La Liste des fautes localement observables** L_O, est une liste des fautes impliquant un résultat différent de la simulation saine observable sur des signaux et des variables de la description. Le terme *"localement"* ne s'applique pas à un élément du circuit mais bien à une donnée *(signal ou variable)* de la description VHDL.
- **Une faute détectée**, est une faute qui appartient à L_O et qui possède dans sa signature un couple composé d'un signal de sortie et d'un pilote de valeurs fautives différent du pilote de valeurs saines.
- **La liste des fautes détectées** L_D, est la liste des fautes détectées pendant la simulation BFS-DEVS. Les élément appartenant à cette liste sont extraits de L_O de telle sorte qu'à la fin d'un cycle de simulation : $L_O \cap L_D = \emptyset$.
- **Un processus actif** est :
  - au sens VHDL, un processus dont la liste de sensitivité a évolué,
  - au sens BFS-DEVS, *l'exécution du modèle couplé associé au processus pour une simu-*

lation saine.

- **Un processus inactif** est :
  - au sens VHDL, un processus dont la liste de sensitivité est statique,
  - au sens BFS-DEVS, un processus inactif correspond à *l'exécution du modèle couplé associé au processus pour une simulation uniquement fautive.* Cependant, si aucune faute ne doit être propagée dans le processus, le modèle couplé n'est pas exécuté.
- **Une base de données de référence** est un objet qui stocke les pilotes de valeurs VHDL de tous les signaux, variables et constantes appartenant à la description. Cette base de données est sollicitée en lecture et/ou en écriture par les modèles atomique BFS-DEVS suivant *:*

	lecture	**écriture**
Assignment	oui	oui
Conditional	oui	non
Jonction	non	non
ProcessEngine	oui	non
Generator	oui	oui

TAB. 4.1*: Tableau des règles d'accès la base de données.*

La simulation de fautes concurrente d'un réseau BFS-DEVS composé de N modèles couplés correspond à :

- **l'exécution des $x < N$ modèles couplés pour une simulation saine concurrencée par,**
- **l'exécution des $1 \leq i \leq N - x$ autres modèles couplés pour i simulations fautives induites par L_i listes de fautes propagées au sein du réseau.**

Afin de donner une définition plus imagée de la simulation de fautes concurrente, considérons un réseau BFS-DEVS d'une description comportementale multi-processus composé de *PA* (*PI*) processus actif *(resp. inactif)*. Au cours d'un cycle de simulation, $PA_{n>0}$ modèles couplés sont exécutés pour n simulations saines et $PI_{m>0}$ sont exécutés pour m simulations fautives. Les simulations saines s'exécutent au sein des PA_n modèles couplés définissant ainsi les n chemins sains. Afin de mettre en évidence les fautes qui impliquent les chemins fautifs, les simulations fautives sont exécutées en concurrence aux simulations saines dans les $n+m$ modèles couplés. Schématiquement, les chemins fautifs résultant dérivent directement *(resp. indirectement)* des n chemins

sains à l'intérieur des PA_n modèles couplés *(resp. des PA_m modèles couplés)*.

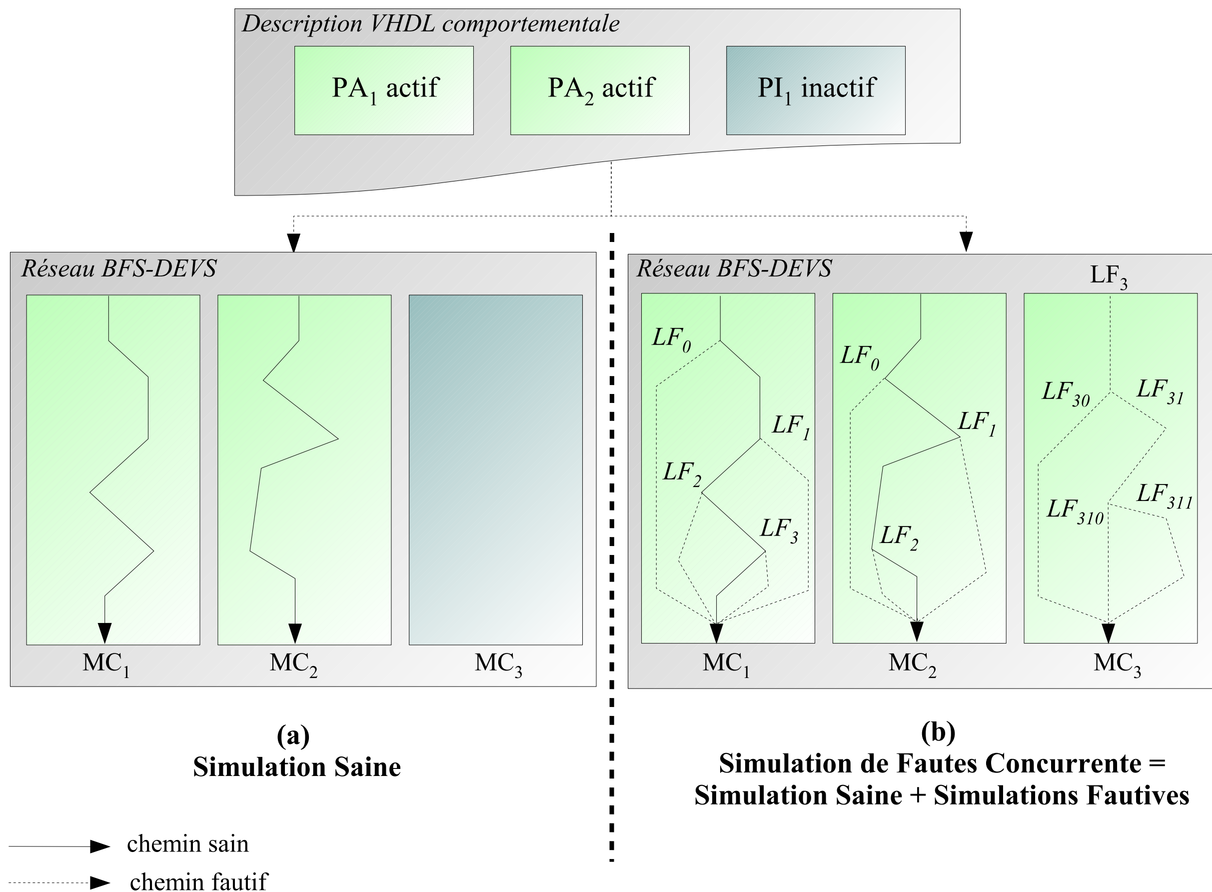


FIG. 4.3: *Vue schématique de la simulation de fautes concurrente BFS-DEVS.*

La figure 4.3 *(a)* montre les chemins sains *(en lisse)* qui résultent des simulations saines des deux modèles couplés MC_1 et MC_2 correspondant aux deux processus actifs PA_1 et PA_2. Le processus PI_1 est inactif et le modèle couplé MC_3 correspondant n'est pas simulé. La figure 4.3 *(b)* montre les chemins fautifs *(en pointillés)* générés par les simulations fautives concurrentes à la simulation saine. Nous voyons que les chemins fautifs à l'intérieur des modèles couplés MC_1 et MC_2 sont *directement concurrents* aux chemins sains. Par contre, les chemins fautifs à l'intérieur du modèle couplé MC_3 sont *indirectement concurrents* aux chemins sains. De plus la propagation par découpage de la liste de faute $LF_3 \geq 4$ du modèle couplé inactif MC_3 permet la génération des simulations des fautes contenues dans les listes $LF_{30} \geq 1$, $LF_{31} \geq 2$, $LF_{310} \geq 1$ et $LF_{311} \geq 1$.

4.2.2 Le cycle de simulation VHDL

Après avoir donné la définition d'une simulation BFS-DEVS, nous allons détailler de quelle manière le cycle de simulation VHDL classique à été modifié pour intégrer la technique de simulation concurrente des listes de fautes.

4.2.2.1 Le cycle de simulation sain

La simulation VHDL est pilotée par l'apparition d'événements sur les signaux de communication entre les processus. Le cycle de simulation appelé *cycle symbolique (ou delta-cycle)* est découpé en trois phase distinctes exécution, mise à jour et analyse représentées sur la figure 4.4.

1. La phase d'**exécution** correspond à l'exécution en parallèle des processus. Au cours de cette phase les variables internes des processus exécutés sont modifiées au moment de leur affectation et les transactions effectuées sur les signaux de communication sont enregistrées.
2. Au cours de la phase de **mise à jour** les pilotes de valeurs et les différents attributs des signaux sont mis à jour.
3. La phase d'**analyse** établit la liste des processus à exécuter en fonction des événements programmés sur les signaux sensitifs.

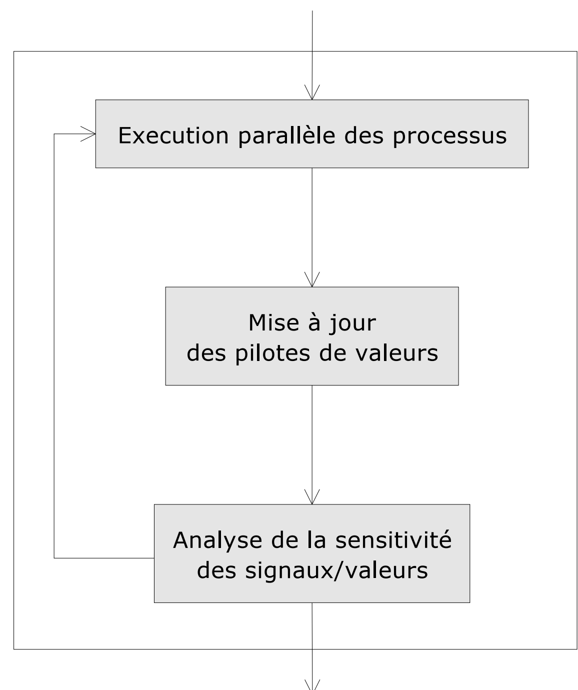


FIG. 4.4: *Schéma d'un cycle de simulation VHDL.*

4.2.2.2 Le cycle de simulation de fautes

Les trois phases d'un cycle de simulation d'un processus sont l'**exécution**, la **mise à jour** et l'**analyse** de l'activité future des processus.

Comme il est montré sur la figure 4.5 la simulation de fautes concurrente s'appuie sur ces trois phases et vient les compléter avec l'adjonction de deux phases supplémentaires relative à la technique de simulation concurrente : la phase d'**observabilité** des fautes destinée à construire la liste L_O et la phase de **calcul de la couverture de fautes**. Nous avons donc dans l'ordre d'exécution d'un description VHDL :

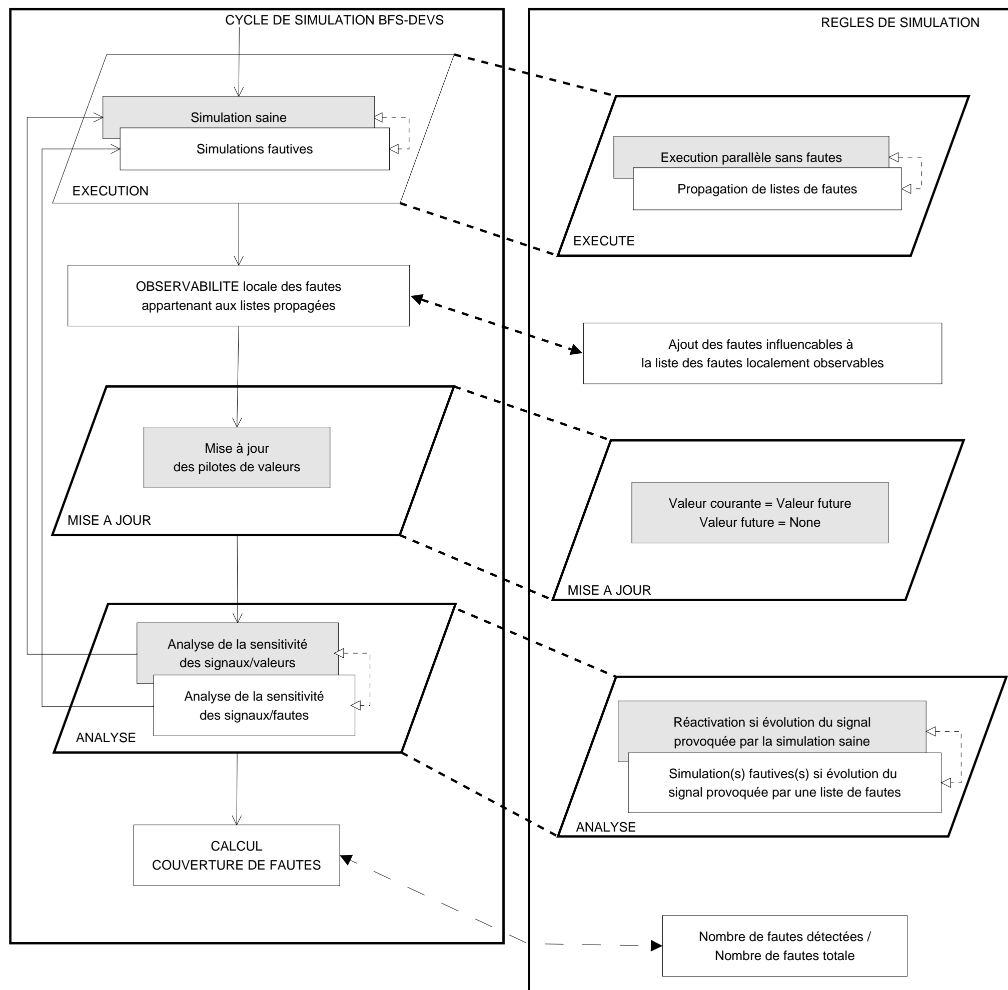


FIG. 4.5*: Schéma d'un cycle de simulation de fautes concurrente.*

1. La phase de **simulation BFS-DEVS** qui consiste en l'exécution concurrente :

   (a) **d'une simulation saine** des modèles couplés correspondant aux processus actifs sans présence de fautes. Les résultats de cette simulation sont stockés dans une base de données de référence appelé *sdb (symbolic data base)*.

   (b) **des simulations fautives** des modèles couplés correspondant aux processus inactifs

s'exécutant en concurrence avec la simulation saine. Chaque simulation fautive emprunte un chemin fautif différent du chemin sain sur lequel sont propagées les listes des fautes responsables de l'orientation fautive dans le réseau BFS-DEVS. *La propagation par découpage et réorientation de ces listes* le long des chemins fautifs initiaux est conduite par des règles spécifiques aux composants BFS-DEVS. C'est durant cette propagation qu'est construite la signature des fautes.

2. La phase d'**observabilité** des fautes $F_{i\in\mathbb{N}}$ appartenant aux listes $LF_P = \{F_i | i \in \mathbb{N}\}$ propagées intervient lorsque tous les processus P de la description ont été simulés :

   (a) **pour un processus ayant été actif**, les fautes $F_i \in LF_P$ sont localement observables sur un signal S si :

      i. il y a eu une variation du pilote de valeurs P_S du signal S dans la base de données de référence *sbd* et,

      ii. les fautes ne sont pas déjà présentes dans la listes des fautes détectées L_D :

$$\begin{aligned} pour\, P_S = [v_c, v_f, c] \in sdb\, si\, v_c \neq v_f \neq None, \\ si\, L_D \cap LF_P = None, \\ alors \{F_i\}\, sont\, localement\, observables\, sur\, S \end{aligned} \tag{4.1}$$

   (b) **pour un processus ayant été inactif**, les signatures $T_i = \{(S_j, [v_{cf}, v_{ff}, c]) | j \in \mathbb{N}, S_j \in sdb\}$ des fautes $F_i^{T_i} \in LF_P$ sont analysées et ces fautes sont localement observables sur S_j si :

      i. elles impliquent des valeurs futures fautives v_{ff} des signaux S_j influencés différentes des valeurs courantes saines v_c dans la *sdb* et,

ii. elles ne sont pas déjà présentes dans la listes des fautes détectées L_D :

$$\begin{aligned} pour\, T_i = \{\{(S_j,[v_{cf},v_{ff},c])|j \in \mathbb{N}, S_j \in sdb\} \in F_i^{T_i}, \\ pour\, P_{S_j} = \{v_c, v_f, c\} \in sdb, \\ si\, v_{ff} \neq v_c\, et\, si\, L_D \cap L_P = None, \\ alors\, \{F_i\}\, sont\, localement\, observables\, sur\, S_j \end{aligned} \tag{4.2}$$

Toutes les fautes localement observables sont ajoutées à la liste L_O mais plusieurs cas sont possibles :

- Les fautes sont insérées dans L_O si elles ne sont pas déjà présentes.
- Si la faute est déjà présente dans L_O sa signature est mise à jour. Pour chaque couple $(S_j, P_{S_j}) \in T_j$ de la faute à insérer dans L_O :
  - si le couple n'est pas présent dans la signature de la faute, on l'insère,
  - si le couple est déjà présente dans la signature de la faute de L_O, plusieurs configurations des pilotes fautifs sont possibles. On pose $P_{F_i}^{L_O}(S_j)$ le pilote fautif du signal influencé S_j par la faute F_i appartenant à L_O et $P_{F_i}^{LF_P}(S_j)$ le pilote fautif du même signal mais de la faute à insérer :
  - si $P_{F_i}^{L_O}(S_j) = [x, None, c_1]$ et $P_{F_i}^{LF_P}(S_j) = [None, y, c_2]$,
    - avec $x \neq y$ et $c_1 (\neq | =) c_2$: on fusionne les deux pilotes pour donner le pilote final $P_{F_i}^{L_O}(S_j) = [x, y, c_2]$,
    - avec $x = y$: la fusion n'est pas nécessaire,
  - le cas $P_{F_i}^{L_O}(S_j) = [x, y, c]$ et $P_{F_i}^{LF_P}(S_j) = [None, z, c]$ avec $x \neq y \neq z$ est impossible car il correspond au cas ou un signal est affecté dans deux processus différents. Autrement dit, deux mêmes fautes sont simulées dans deux processus différents et transportent dans leurs signatures deux valeurs différentes pour un même signal *(conflit)*.
  - Il en est de même pour le cas : $P_{F_i}^{L_O}(S_j) = [None, x, c]$ et $P_{F_i}^{LF_P}(S_j) = [None, y, c]$ avec $x \neq y$ *(conflit)*.

- Pour les variables, les règles restent inchangées mais les pilotes de valeurs fautives sont du type $P_{F_i}(V_j) = [x, c]$. Lorsque la signature est mise à jour, la valeur x est écrasée par la nouvelle valeur $y \neq x$. L'affectation de la valeur fautive est immédiate.

3. La phase de **mise à jour** est appliquée aux pilotes de valeurs de la base de données de référence des signaux qui ont évolués au sein d'un processus ayant été actif. Si aucun processus n'a été exécuté pour une simulation saine, aucune mise à jour n'est effectuée :

$$pour\, P_{S_i} \in sdb, si\, v_f \neq None : \; v_c = v_f = None \tag{4.3}$$

4. La phase d'**analyse de l'activité future** des processus présente une procédure supplémentaire relative à l'influence des fautes sur des signaux sensitifs :

   (a) **Pour les processus actifs :** Un processus P sera actif et sujet à une simulation saine pour un future cycle symbolique si au moins un des signaux $S_{i\in\aleph}$ de sa liste sensitive $L_p = \{S_i | i \in \mathbb{N}\}$ a évolué. Autrement dit, si au moins un signal sensitif S_i présente une différence entre sa valeur courante v_c et sa valeur future v_f dans le pilote $P_{S_i} = [v_c, v_f, c]$ de la base de données de référence :

$$\begin{aligned} &pour\, S_{i\in\aleph} \in L_p\, et\, P_{S_i} = [v_c, v_f, c] \in sdb, \\ &\qquad si\, v_c \neq v_f \neq None, P\, est\, actif\, pour\, une\, simulation\, saine \end{aligned} \tag{4.4}$$

   (b) **Pour les processus inactifs :** Un processus sera inactif et sujet à une simulation fautive pour un future cycle symbolique si :

      i. le processus n'est pas déjà actif et,

      ii. au moins une faute $F^{T_j}_{j\in\aleph}$, de la liste L_O, possédant la signature T_j tel que : $T_j = \{(S_i = [v_{cf}, v_{ff}, time]) | i \in \mathbb{N}\}$, associée à un signal sensitif $S_i \in L_p$ du processus P implique une variation de la valeur du signal alors qu'il est statique dans la base de données de référence sdb :

$$pour\, T_j \in F_j^{T_j} \in L_O,\, pour\, P_{S_i} = [v_c, v_f, c] \in sdb \tag{4.5}$$

$$si\, v_{cf} = None\, et\, v_c \neq v_{ff} \neq None, \tag{4.6}$$

$$ou\, si\, v_{cf} \neq None\, et\, v_{cf} \neq v_{ff} \neq None, \tag{4.7}$$

$$P\, est\, actif\, pour\, une\, simulation\, fautive$$

Toutes les fautes à l'origine de ces variations seront alors *"copiées"* de la liste L_O vers la liste LF_P afin d'être simulées au cours de ce cycle symbolique et leurs signatures sont mises à jour.

S'il n'existe aucune faute respectant ces conditions, aucune faute n'a besoin d'être simulée et le processus n'est sujet à aucune exécution.

Enfin toutes les signatures des fautes de L_O sont mises à jour :

$$pour\, F_j^{T_j} \in L_O,\, pour\, (S_i, [v_{cf}, v_{ff}, c]) \in T_j, \tag{4.8}$$

$$v_{cf} = v_{ff} = None$$

$$si\, v_{cf} = v_c,\, suppression\, du\, couple\, (S_i, P_{S_i})\, de\, F_j^{T_j} \tag{4.9}$$

On remarque que si le pilote $P_{S|!V_i}$ d'une faute F_j implique la même valeur que la simulation sain pour un signal S_i ou une variable V_i *(condition 4.9)*, le couple $(S|V_i, P_{S|V_i})$ est supprimé de la trace à F_j.

La mise à jour des pilotes de valeurs étant effectuée précédemment, les pilotes des signaux statiques sont de la forme $P_{S_i} = [v_c, None, c]$. Les signatures des fautes de la liste L_O ne sont pas mises à jour et elles sont de la forme $T_i = \{(S_j, [(None|v_{cf}), v_{ff}, c])\}$. Nous remarquons

que la valeur courante fautive v_{cf} peut être différente de *"None"*. C'est le cas lorsqu'une faute se propage plusieurs fois dans un processus à des cycles de simulations différents *(processus s'auto-activant par des processus intermédiaires).*

En effet, si une faute est responsable de l'activation fautive d'un processus P, sa signature est mise à jour et elle est insérée dans la liste LF_P. Cependant, lorsque la simulation fautive est terminée, la phase d'analyse de l'activité future des processus est appliquée une seconde fois :

- Si la faute à déjà été simulée dans le processus *(mais pas avec les mêmes conditions)* : les pilotes de valeurs ne font plus office de référence et il faut analyser la sensitivité du signal influencé par rapport aux valeurs de la signature de la faute : v_{cf} et v_{ff}.
- Si la faute n'a jamais été simulée dans le processus : les pilotes de valeurs servent de référence à l'analyse de la future activité des processus.

5. La phase de **calcul de la couverture de fautes** intervient uniquement lorsque plus aucun processus ne doit être réactivé. Une faute F_i de la liste L_O sera détectée si elle implique une valeur courante fautive différente de la valeur courante $(v_{cf} \neq v_c)$ du pilote de valeurs P_{S_i} d'un signal de sortie : S_i

$$pour\,F_i^{T_i} \in L_O,\, pour\,(S_j, [v_{cf}, v_{ff}, c]) \in T_j$$
$$pour\,P_{S_j} = [v_c, v_f, c] \in sdb \quad si\,v_c \neq v_{cf} \quad F_i\,est\,observable \tag{4.10}$$

La couverture de fautes est donnée par :

$$CF(\%) = \frac{\text{Nombre de fautes détectées}}{\text{Nombre totale de fautes}} * 100 \tag{4.11}$$

4.2.3 La technique de propagation des listes de fautes LF_i

La simulation BFS-DEVS peut être composée de plusieurs simulations fautives au cours desquelles les listes de fautes sont propagées et simulées. La propagation de ces listes se fait par *découpage et par réorientation des listes primaires*. Le but de cette section est d'établir les règles de la propagation des listes de fautes au sein d'un réseau BFS-DEVS dérivant une description VHDL multi-processus. Pour ce faire, nous considérons :

- une description possédant N processus : $P_{0 \leq n < N}$,
- les listes des signaux sensitifs *(d'entrée ou internes)* responsables de l'activation des N processus : $L_n = \{S_{k \in \aleph}\}$,
- les listes des fautes propagées au sein des modèles BFS-DEVS : $LF_n = \{F_i | i, n \in \aleph\}$,
- la liste des fautes localement observables L_O,
- les signatures décrivant les traces uniques de la propagation des fautes pendant la simulation concurrente : $\forall F_i \in \aleph, T_{F_i} = \{((S|V)_j, P_{(S|V)_j}) | j \in \aleph\}$.

La propagation des listes de fautes peut être divisée en deux parties :

- la propagation des listes ***au sein*** des modèles couplés : ***propagation intra-processus***,
- la propagation des listes ***entre*** les modèles couplés : ***propagation inter-processus***.

4.2.3.1 Propagation intra-processus

A chaque début d'un cycle de simulation VHDL *(à l'exception du cycle d'initialisation),* chaque processus peut présenter une activité dépendant de la sensitivité des signaux appartenant à L_n. Nous savons que *(cf. partie 4.2.1) :*

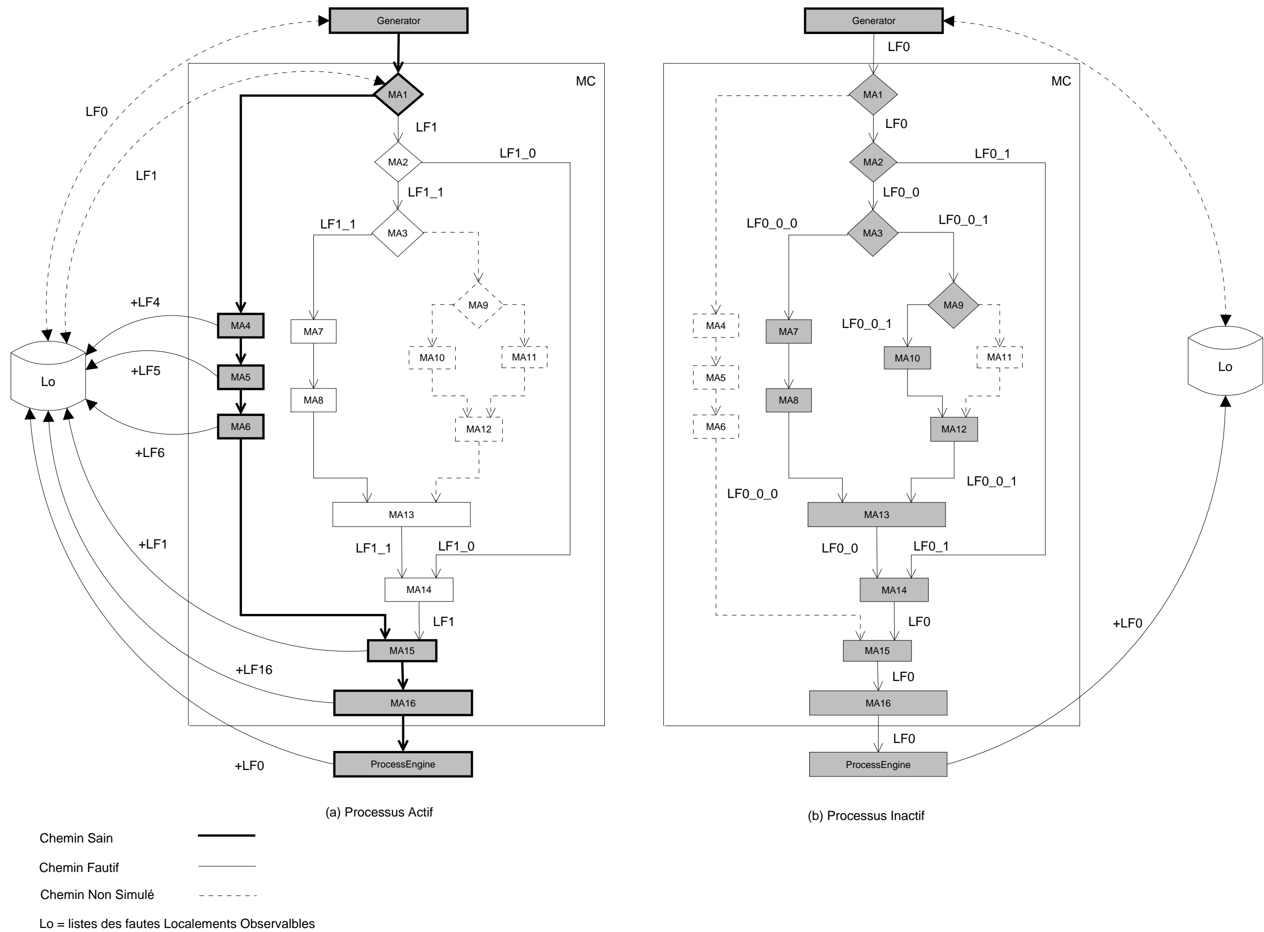


FIG. 4.6*: Propagation intra-processus.*

– **lorsqu'un processus VHDL P_n est actif** *(cf. figure 4.6 (a))*, le modèle couplé BFS-DEVS le représentant est exécuté par un modèle atomique BFS-DEVS *"Generator"* afin d'effectuer la simulation saine ainsi que les simulations :
  – des fautes F_i impliquant une non activation de P_n stockées dans une liste de fautes primaire construite par le modèle *"Generator" (LF_0 dans l'exemple de la figure* 4.6*)*. L'observabilité de ces fautes dépend du résultat de la simulation saine et elles ne seront éventuellement ajoutées à la liste L_O qu'en fin de simulation par le modèle *"ProcessEngine" (cf. figure 4.6 (a))*.
  – des fautes F_i susceptibles d'apparaître au sein des modèles *"Assignment"* et *"Conditional"* se trouvant sur le chemin sain du réseau BFS-DEVS *(en gras sur la figure* 4.6 *(a))*. Ces fautes sont stockées dans des listes LF_n *(LF_1 pour MA_1, LF_4 pour MA_4 etc)*. Dans

le cas où ces listes sont issues d'un modèle *"Assignment" (MA_1,MA_4, MA_5, MA_6)*, elles sont directement ajoutées à la liste des fautes localement observables L_O. Si elles sont issues d'un modèle *"Conditional" (MA_{16} sur la figure 4.6 (a))* elles ne seront ajoutées à la liste L_O par le modèle *"Junction"* correspondant *(MA_{14} sur la figure* 4.6 *(a))* que lorsqu'elles auront été simulées sur le(s) chemin(s) fautif(s) qu'elles impliquent *(en lisse sur la figure* 4.6 *(a))*. En effet, ces listes LF_n peuvent donner naissance à d'autres sous-listes LF_{nm} *($LF_1 = LF_{1_0} + LF_{1_1}$ sur la figure* figure 4.6 (a)*)*.

La liste L_O sert de référence car :
- si une faute F_i est mise en évidence sur un modèle *"Assignment"* et que cette faute est déjà présente dans la liste L_O, une mise à jour de la trace de celle-ci sera effectuée sinon elle est ajoutée entièrement.
- si une faute F_i est mise en évidence sur un modèle *"Conditional"* et que cette faute est déjà présente dans la liste L_O, elle sera copiée et insérée dans la liste LF_n destinée à être propagée sur les chemin(s) fautif(s). L'observabilité du contenu de la liste sera analysée à la réunion de tous les chemins fautifs.

- **lorsqu'un processus VHDL P_n est inactif** *(cf. figure 4.6 (b))*, le modèle couplé BFS-DEVS le représentant est exécuté par le modèle *"Generator"* afin de simuler les fautes $F_i \in LF_n$. Ces fautes sont ciblées sur les signaux appartenant à L_n, mais contrairement au cas précédent, durant la propagation de la listes LF_n, les fautes F_i peuvent posséder des signatures initiales T_{F_i} issues de la liste L_O qui seront mises à jour pendant les simulations fautives du processus P_n. En effet, la construction de cette liste de fautes primaire fait toujours référence à la liste L_O *(voir liste LF_0 sur la figure 4.6 (b))*. Toutes les fautes de LF_n peuvent ne pas emprunter le même chemin. Par conséquent, les listes LF_n peuvent être *découpées et réorientées en plusieurs sous-liste LF_{nm} regroupant les fautes empruntant les mêmes chemins dans le modèle couplé associé au processus P_n*. Dans l'exemple de la figure *4.6* (b) :
  - la liste de faute initiale LF_0 se découpe en deux sous-listes FL_{0_0} et FL_{0_1} *(tel que $LF_0 = LF_{0_0} + LF_{1_1}$)* car les fautes appartenant à LF_{0_0} n'ont pas les mêmes conséquences que

les fautes appartenant à LF_{0_1} au niveau du modèle *"Conditional"* MA_2.

- pour les mêmes raisons, la liste de fautes LF_{0_0} se découpe en deux sous-listes $FL_{0_0_0}$ et $FL_{0_0_1}$.

L'analyse des signatures des fautes F_i s'effectue à la fin des simulations fautives par le modèle *"ProcessEngine"*. Toutes les fautes observables sont ajoutées à la liste L_O. Si une faute est déjà présente dans la liste, une mise à jour de sa signature est effectuée. Après cette mise à jour, si une faute voit sa signature devenir vide, elle n'est plus observable et est supprimée de la liste L_O.

Pour résumer la propagation intra-processus :

- **Si le processus est actif :**
  - Tous les modèles *"Assignment"* et *"Junction"* qui se trouvent sur une chemin sain construisent ou mettent à jour la liste L_O.
  - Les simulations fautives possibles au sein du réseau BFS-DEVS sont générées en concurrence directe à la simulation saine et propagent les listes LF_n issues des modèles *"Conditional"*,
- **Si le processus est inactif :**
  - Seul les modèles *"Generator"* et *"ProcessEngine"* accèdent à la liste L_O afin de construire et d'analyser en fin de simulation les listes de fautes initiales LF_n.
  - Les simulations fautives possibles sont générées en concurrence par *découpage et réorientation* de la liste initiale LF_n issue du modèle*"Generator"*.
  - Les sous-listes LF_{nm} associées aux simulations fautives possibles sont composées des fautes de la liste initiale LF_n.

4.2.3.2 Propagation inter-processus

Nous savons que les signaux sensitifs internes ont été définis pour établir la liaison et la parallélisation entre les processus d'une description VHDL. Par conséquent, les fautes F_i susceptibles d'être présentes sur ces signaux sont également en liaison entre les processus. Que se passe t'il alors lorsqu'une faute localement observable sur un signal sensitif *(c.a.d appartenant à la liste*

L_O) a besoin d'être simulée au sein d'un ou de plusieurs processus ? Autrement dit, de quelle façon les fautes appartenant à L_O sont insérées dans les listes des fautes LF_n activant les n modèles couplés associés aux n processus initialement inactifs ? Afin d'établir ces règles de construction, il est nécessaire de rappeler les propriétés des signaux sensitifs régis par le langage VHDL.

Propriétés des signaux sensitifs :

Les propriétés des signaux sensitifs d'une description VHDL comportementale sont [sta, 2000] :

1. Aucun signal sensitif *(interne ou d'entrée)* ne peut être affecté dans le processus qu'il active.
2. Un signal sensitif ne peut pas être affecté dans plusieurs processus.
3. Un signal sensitif peut activer plusieurs processus.

La propriété n°1 conduit à la structure figure 4.7(a). Cette configuration peut entraîner une activation en boucle du processus. Elle est donc inutile d'autant plus que si l'utilisateur veut exécuter une instruction en boucle le langage VHDL lui fournit l'instruction *"loop"*.

La propriété n°2 conduit à la structure figure 4.7(b) et vient du fait qu'un signal ne possède qu'un seul pilote de valeurs. S'il est affecté dans plusieurs processus, l'utilisateur doit implémenter une fonction de résolution du conflit entre les différentes valeurs possibles du signal.

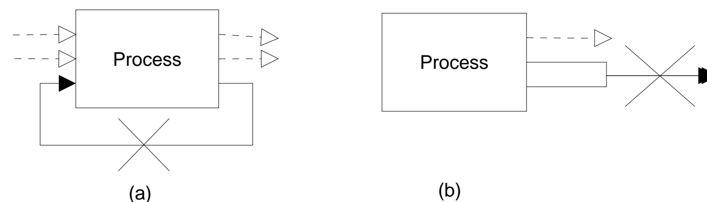


FIG. 4.7*: Auto-activation et conflit d'affectation des signaux d'un processus.*

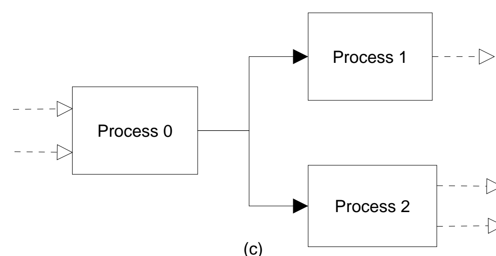


FIG. 4.8*: Multi-activation des processus 1 et 2 à partir du même signal interne.*

La propriété n°3 conduit à la structure figure 4.8 et permet la parallélisation des processus activés par un même signal.

Propriétés des fautes :

1. Une faute peut appartenir à plusieurs listes FL_n des n processus inactifs P_n.
2. Une faute ne peut pas impliquer plusieurs valeurs différentes pour un même signal.

La propriété n°1 découle de la propriété des signaux sensitifs n°3. La propriété n°2 découle de la propriété des signaux sensitifs n°2. Ces deux propriétés impliquent qu'une faute $F_i \in LF_n$ qui doit être simulée au sein du modèle couplé inactif possède forcement dans sa signature au moins un couple (S, P_S) avec S un signal sensitif $S \in L_n$.

Règles de propagation inter-processus des fautes localement observables :

1. Si une faute F_i appartenant à L_O implique la simulation fautive de n processus différents $P_{0 \leq n < N}$, elle sera dupliquée et insérée dans les listes LF_n. A la fin du cycle de simulation la faute redevient unique avec la fusion des signatures de ces fautes dupliquées.
2. L'analyse de la future activité fautive des processus $P_{0 \leq n < N}$ se fait par rapport à la signature des fautes appartenant à L_O en référence à la base de donnée. Toutes les fautes, à l'exception des fautes de type F_1 sur les signaux sensitifs, présentes dans la liste L_O activent les processus associés.

La règle n° 1 découle de la propriété des fautes n°1.

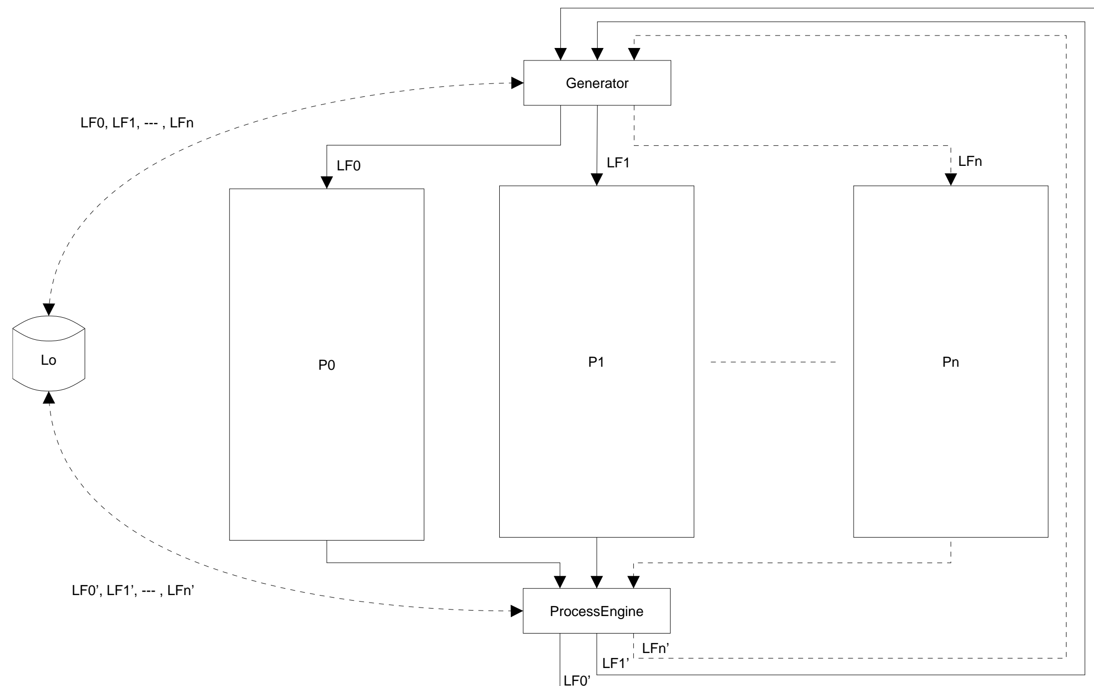


FIG. 4.9*: Propagation inter-processus.*

L'origine des listes de fautes LF_n est double :

– **Si un modèle couplé associé à un processus P_n est actif pour un cycle physique**, les listes de fautes LF_n sont générées par le modèle *"Generator" (LF_0, LF_1, ..., LF_n figure* 4.9*)*. Il construit ses listes avec pour référence la liste L_O. Si une faute devant appartenir à la liste LF_n se trouve déjà dans la liste L_O, elle est copiée et insérée dans la liste LF_n. Elle sera ensuite simulée par propagation inter-processus dans le réseau de composant BFS-DEVS et analysée en fin de simulation au sein du modèle *"ProcessEngine"*. Si une faute F_i appartenant à deux messages différents arrive sur le modèle *"ProcessEngine"*, les signatures sont fusionnées afin de mettre à jour l'implication unique de la faute sur les éléments du réseau *(cf. condition 8.1 et 8.2)*.

– **Si un modèle couplé associé à un processus P_n est actif pour un cycle symbolique**, les listes de fautes LF_n' sont générées par le modèle *"ProcessEngine" (LF_0', LF_1', ..., LF_n' figure* 4.9*)*. En effet si un modèle couplé a besoin d'être reexécuté car une faute présente dans L_O a provoqué une variation d'un signal appartenant à L_n, un message fautif est envoyé sur le port de sortie du modèle *"ProcessEngine"* en direction de P_n *(cf. condition 4.4)*.

4.3 Exemple : Le registre 8 bits

Cette section a pour but d'expliquer en détail le mécanisme d'une simulation de fautes concurrente appliquée à une description VHDL comportementale. Par souci de généralisation, la description VHDL choisie représente le comportement du registre 8 bits dont le réseau BFS-DEVS est montré sur la figure 4.10.

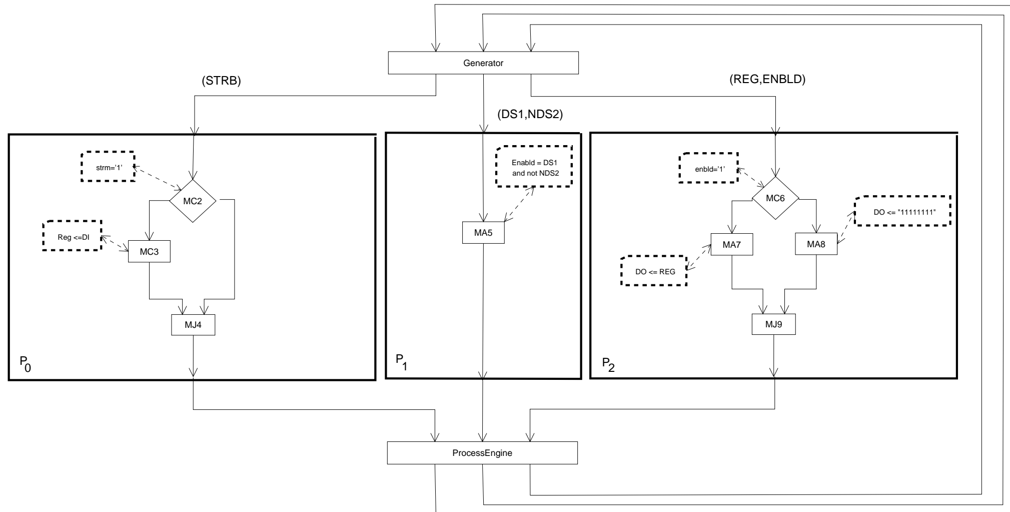


FIG. 4.10*: Réseau BFS-DEVS du registre 8 bits.*

4.3.1 Liste des fautes

Si l'on considère le modèle de fautes $\{F_1, F_2, F_3\}$ *(cf. partie 4.1.2),* la liste L_T des 28 fautes qui peuvent intervenir dans le réseau du registre 8 bits est donnée par le tableau 4.2.

L_T		
$STRB_1$,$STRB_0$	$F2_{MC2}0$,$F2_{MC2}1$	$F3_{MA3}$
$DO_{11111111}$,$DO_{00000000}$	$F2_{MC6}0$,$F2_{MC6}1$	$F3_{MA5}$
$DI_{00000000}$,$DI_{11111111}$	$F2_{P_0}0$,$F2_{P_0}1$	$F3_{MA7}$
$REG_{11111111}$,$REG_{00000000}$	$F2_{P_1}0$,$F2_{P_1}1$	$F3_{MA8}$
$NDS2_1$,$NDS2_0$	$F2_{P_2}0$,$F2_{P_2}1$	
$RESET_1$,$RESET_0$		
$ENBLD_1$,$ENBLD_0$		

TAB. 4.2*: Liste totale des fautes pour le registre 8 bits.*

Le tableau 4.2 est rangé en trois colonnes suivant les trois types de fautes considérées :

- La première colonne rassemble toutes les fautes de type F_1. Par exemple, le signal $STRB$ de type bit, peut être collé aux valeurs '1' ou '0'. Les deux fautes qui correspondent à ces collages sont notées $STRB_1$ et $STRB_0$. Lorsqu'un signal ou une variable admet plus de deux valeurs, nous ne considérons que les fautes de collage aux bornes de l'ensemble de définition. Dans le cas du signal DI, nous choisissons de ne simuler que les fautes de collages aux bornes de son ensemble de définition à savoir : $DI_{00000000}$ et $DI_{11111111}$.
- La deuxième colonne rassemble toutes les fautes de type F_2. Par exemple, pour le modèle *"Conditional" MC*2 qui possède deux branches de sortie, les fautes correspondant au collage de ces branches sont $F2_{MC2}0$ et $F2_{MC2}1$. Les fautes de type F_2 sur les modèles couplés P_0,P_1 et P_2 permettent de représenter une exécution permanente ($F2_{P_{0,1,2}}0$) ou une non exécution permanente ($F2_{P_{0,1,2}}1$) de ces modèles.
- La troisième colonne rassemble toutes les fautes de type F_3. Par exemple, pour le modèle *"Assignment" MA*3, la faute qui implique la non exécution permanente de ce modèle est notée $F3_{MA3}$.

4.3.2 Cycle d'initialisation C_0

Le cycle d'initialisation C_0 implique l'activation de tous les processus. Comme le montre la figure 4.11 les trois processus P_0, P_1 et P_2 sont activés par l'intermédiaire des trois messages sains.

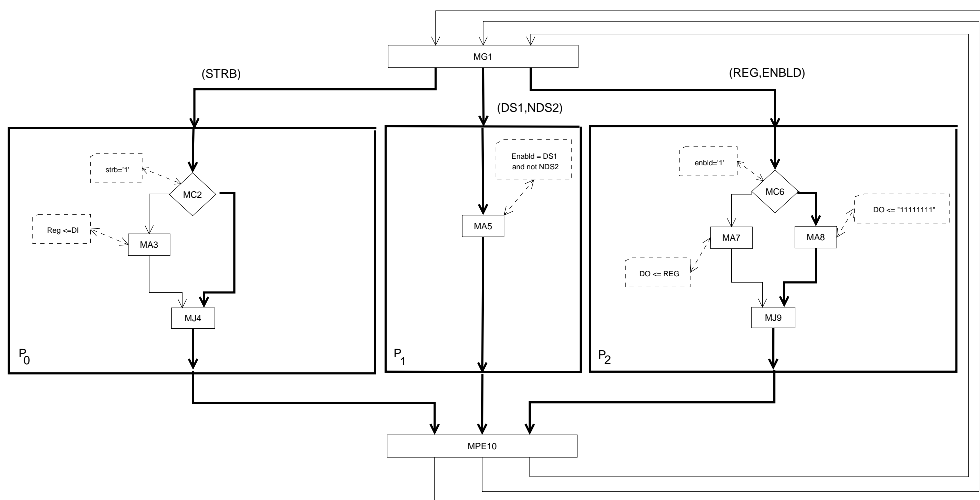


FIG. 4.11*: Réseau BFS-DEVS du registre 8 bits* (cycle d'initialisation).

Les chemins empruntés par ces messages sont inscrits en gras sur la figure 4.11 et correspondent aux chemins sains.

Phase de simulation concurrente des fautes

Le tableau 4.3 regroupe les valeurs initiales des signaux d'entrées du registre 8 bits. Chaque colonne correspond à un signal d'entrée, et les deux lignes correspondent à la valeur initiale par défaut et à la valeur courante du signal. Par exemple, le signal DI possède la valeur "00000000" par défaut *(toujours égale à la valeur de la borne inférieur du domaine de définition)* et la même valeur "00000000" au cycle C_0.

	DI	STRB	DS1	NDS2
Valeur par défaut	"00000000"	'0'	'0'	'0'
C_0	"00000000"	'0'	'0'	'0'

TAB. 4.3*: Valeurs initiales des signaux d'entrée.*

Les résultats de la simulation BFS-DEVS du réseau de la figure *4.11* sont regroupés dans le tableau 4.4.

<table>
<tr><td></td><td colspan="2">Chemin sain</td><td colspan="3">Chemin fautif</td></tr>
<tr><td rowspan="2">P_0</td><td colspan="2">MC2</td><td>MC2</td><td>MA3</td><td>MJ4</td></tr>
<tr><td colspan="2">STRB='0' => Faux</td><td>$LF_{MC2} = [STRB_1,$
$F2_{MC2}0]$</td><td>$LF_{MC2} = [STRB_{1,}^{\{...,(REG,[None,"00000000",0])\}},$
$F2_{MC2}0^{\{REG,[None,"00000000",0]\}}]$</td><td>$LF_O = [$
$STRB_1]$</td></tr>
<tr><td rowspan="2">P_1</td><td colspan="2">MA5</td><td colspan="3">MA5</td></tr>
<tr><td colspan="2">ENBLD =
'0' and not '0'='0'</td><td colspan="3">$L_O = [..., ENBLD_1, DS1_1^{\{(ENBLD,[None,'1',0])\}}, ...]$</td></tr>
<tr><td rowspan="2">P_2</td><td>MC6</td><td>MA8</td><td>MC6</td><td>MA7</td><td>MJ9</td></tr>
<tr><td>ENBLD='0'
=> Faux</td><td>DO="11111111"
$L_O =$
$[DO_{00000000},$
$F3_{MA8}^{\{DO,[None,"00000000",0]\}}]$</td><td>$LF_{MC6} =$
$[ENBLD_1,$
$F2_{MC6}0]$</td><td>$LF_{MC6} =$
$[ENBLD_1^{\{...,(DO,[None,"00000000",0])\}},$
$F2_{MC6}0^{\{DO,[None,"00000000",0]\}}]$</td><td>$L_O = [...,$
$ENBLD_1,$
$F2_{MC6}0,$
$...]$</td></tr>
</table>

TAB. 4.4*: Tableau des résultats du cycle d'initialisation.*

Ce tableau est composé de trois lignes qui représentent les processus simulés et de deux colonnes qui représente le chemin sain et le chemin fautif au sein de ces processus. Par exemple, l'intersection de la première ligne avec la première colonne montre les résultats de la simulation BFS-DEVS du processus P_0 sur le chemin sain. Sur ce chemin, le modèle $MC2$ est exécuté et le

résultat de l'évaluation de l'expression conditionnelle *(STRB=='1')* est Faux car STRB est égal à '1'. De manière similaire, l'intersection de la première ligne avec la deuxième colonne montre les résultats de la simulation BFS-DEVS du processus P_0 sur le chemin fautif. Sur ce chemin, les modèles $MC2$, $MC3$ et $MJ4$ sont exécutés pour donner la liste LF_{MC2}, construire les signatures des fautes de la liste LF_{MC2} et faire l'observabilité de la liste LF_{MC2}.

Le tableau 4.5 rassemble les pilotes des signaux qui ont évolués pendant le cycle C_0 ainsi que la liste L_O.

	Type	**Pilotes de valeurs**	**Liste des fautes localement observables**
REG	INTERNAL	["00000000", None, -1]	$L_O = [$
STRB	IN	['0','0',0]	$STRB_{1,}^{\{(STRB,[None,'1',0])\}}, DS1_1^{\{(ENBLD,[None,'1',0])\}},$
ENBLD	INTERNAL	['0','0',0]	$F2_{MC6}0^{\{(DO,[None,"00000000",0])\}}, ENBLD_1^{\{(DO,[None,"00000000",0])\}},$
DO	OUT	["00000000","11111111",0]	$DO_{00000000}, F3_{MA8}^{\{(DO,[None,"00000000",0])\}}]$

TAB. 4.5*: Tableau des pilotes des signaux après l'exécution du cycle C_0.*

Remarques :

1. La faute $ENBLD_1$ dans la liste L_O est présente car $LF_{P_2} = \emptyset$. En effet si le processus P_2 est activé par la variation du signal ENBLD *(variant de la valeur '0' à '1' au cours d'un cycle autre que le cycle d'initialisation)*, la faute $ENBLD_1$ est présente dans la liste $LF_{P_2} = [..,ENBLD_1,..]$. Lorsque la liste de fautes L_{MC6} est déterminée sur le modèle $MC6$, LF_{P_2} est retranchée de la liste L_{MC6} *(voir la règle de construction des listes de fautes sur le modèle atomique conditionnel).*

2. Le cycle d'initialisation VHDL permet de donner une valeur initiale aux signaux et variables du circuit. Il est censé représenter la mise sous tension du circuit sans application des vecteurs d'entrée. Les valeurs par défaut du tableau 4.3 sont par définition les valeurs des bornes inférieures des intervalles de définition de chaque signal. Comme aucun vecteur d'entrée n'est introduit au sein du circuit pour le cycle C_0, les signaux d'entrée conservent leurs valeurs mais tous les processus sont actifs. De plus, les listes de fautes LF_{P_0}, LF_{P_1} et LF_{P_2} sont vides car par définition, un cycle d'initialisation VHDL exécute toutes les instructions séquentielles entre le mot "BEGIN" et le premier point de suspension rencontré *("WAIT" par exemple)* à l'intérieur d'un processus. Par conséquent l'instruction *"process(...)"* n'est

pas évaluée et nous supposons qu'il n'existe aucune faute susceptible de fausser la phase d'initialisation du circuit sous test.

3. La faute $STRB_1$ impliquait la signature $(REG, [None, "00000000", 0])$ et n'est donc pas localement observable sur le signal REG. On remarque que la suppression de cette influence dans la signature est faite dans le modèle *"Junction"*. En effet dans notre cas un signal ne peut être affecté dans deux processus différent et le modèle *"Junction"* effectue la mise à jour des traces des fautes reçus lorsque tous les messages lui sont parvenus *(donc lorsque la simulation saine à rejoint les simulations fautives éventuelles)*. Par conséquent nous n'avons pas besoin d'attendre la fin du cycle de simulation de tous les processus pour interroger la valeur courante v_c du signal REG dans le pilote de la base de données de référence afin de la comparer à la valeur future fautive v_{cf}. Il en est de même pour la faute $F2_{MC2}0^{\{(REG,[None,"00000000",0])\}}$ qui n'est pas insérée dans L_O.

Phase d'observabilité locale des listes de fautes propagées

La phase d'initialisation implique des listes de fautes propagées nulles. Par conséquent, aucune observabilité locale des listes n'est effectuée *(cf. conditions 4.1 et 4.2 dans la partie 4.2.2.2).*

Phase de mise à jour des pilotes

Le tableau 4.6 montre la mise à jour des pilotes ayant évolué durant le cycle C_0 *(cf. condition 4.3 dans la partie 4.2.2.2).*

	Pilotes de valeurs
REG	["00000000", None, -1]
STRB	['0','0',0]
ENBLD	['0',None,-1]
DO	["11111111",None,-1]

TAB. 4.6*: Mise à jour des pilotes de REG, STRB, ENBLD et DO au cycle C_0.*

On remarque que la mise à jour des signatures des fautes localement observables de la liste L_O n'est pas effectuée. En effet, si nous effectuons cette opération à cet endroit de la simulation,

nous perdons l'information sur la sensitivité des signaux influencés servant à l'analyse de la future activité fautive des processus. Si la signature d'une faute possède le signal influencé S_i tel que son pilote fautif soit $P_{S_i} = [x, y, c]$ avec $x \neq y \neq None$, *(voir phase d'analyse de la future activité des processus pour plus de détails),* une mise à jour nous conduirais à perdre la transition entre les valeurs courantes et futures fautives nécessaires à la phase d'analyse de la future activité fautive des processus.

Phase d'analyse de l'activité future

Il faut à présent déterminer s'il existe des processus à activer pour un éventuel futur cycle symbolique *(cf. conditions 4.4 et 4.5 dans la partie 4.2.2.2).*

Pour les processus actifs : Les signaux sensitifs ENBLD, REG, DS1 et NDS2 n'ayant pas évolué pendant le cycle d'initialisation C_0 *(pilotes de valeurs, tableau 4.5),* aucun processus n'a besoin d'être simulé et il n'existe aucun futur cycle symbolique C_0^{+1}.

Pour les processus inactifs : Si la faute $DS1_1^{\{(ENBLD,[None,'1',0])\}}$ est présente au sein de la description *(cf. tableau 4.6)*, le signal ENBLD passerait de la valeur $v_c =' 0'$ à $v_{ff} =' 1'$. Le processus P_2 est sujet à une simulation de cette faute avec une image de la base de données de référence mise à jour $fsdb$. Il faut donc exécuter le processus P_2 avec le message $M_{fault} =< LF_{P_2} = [DS1_1^{\{(ENBLD,['1',None,0])\}}], fsdb >$.

On remarque que la signature de cette faute issue de L_O est mise à jour afin d'informer au processus P_2 que la valeur fautive courante du signal $DS1$ est '1'.

Mise à jour de L_O :

La phase de mise à jour de la liste L_O consiste à supprimer les fautes qui ne sont plus observables *(condition 4.9)* et à mettre à jour les pilotes des fautes encore observables *(condition 4.8).* Le tableau 4.7 donne la liste L_O après avoir été mise à jour.

L_O
$[STRB_{1,}^{\{(STRB,['1',None,0])\}}, DS1_{1}^{\{(ENBLD,['1',None,0])\}}$ $F2_{MC6}0^{\{(DO,["00000000",None,0])\}}, ENBLD_{1}^{\{(DO,["00000000",None,0])\}},$ $DO_{00000000}, F3_{MA8}^{\{(DO,["00000000",None,0])\}}]$

TAB. 4.7*: Mise à jour des signatures des fautes de L_O au cycle C_0.*

Remarque : les temps d'affectation des pilotes de valeur de la base de données de référence sont mis à jour *(à la valeur -1)* alors que ceux des signatures des fautes de la liste L_O sont conservés *(à la valeur 0)*. En effet le cycle de simulation symbolique étant présent pour la simulation fautive, la *sdb* restera inchangée.

4.3.3 Cycle symbolique C_0^{+1}

Le modèle MPE10 envoie un message fautif sur sa sortie *feedback* en direction du modèle couplé P_2 via le composant MG1 *(cf. figure 4.12)*. Ce message contient une liste de fautes composée de la faute $DS1_1$ et d'une image de la base de donnée de référence *(cf. tableau 4.7)*.

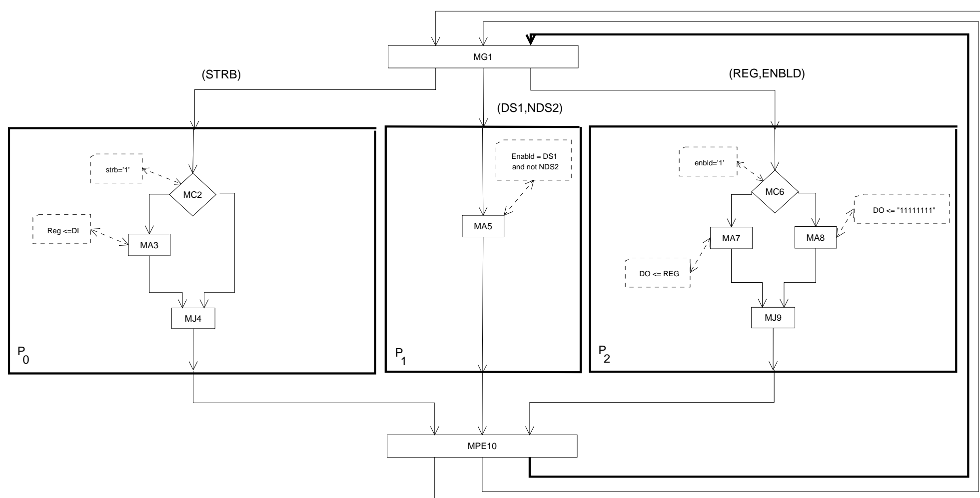


FIG. 4.12*: Réseau BFS-DEVS du registre 8 bits* (cycle symbolique C_0^{+1}).

Phase de simulation de fautes concurrente

Les résultats de la simulation de la liste de fautes LF_{P_2} sont résumés dans le tableau 4.8.

	Chemin fautif	
Liste de fautes propagée	$LF_{P_2} = [DS1_1^{\{ENBLD,['1',None,0]\}}]$	
Composant	MC6	MA7
P_2	$ENBLD_{fault} =' 1'$ => Vrai	$LF_{P_2} = [DS1_1^{\{(ENBLD,['1',None,0]),(\mathbf{DO},[\mathbf{None},\text{"}\mathbf{00000000}\text{"},\mathbf{0}])\}}]$

TAB. 4.8: *Tableau des résultats de la simulation de $DS1_1$.*

Toutes les valeurs fautives impliquées par la faute $DS1_1$ sont insérées dans l'expression conditionnelle MC6 rendant le résultat de son évaluation à vrai. La faute est ensuite évaluée au niveau du modèle atomique d'affectation MA7 donnant la valeur fautive $v_{cf}(DO) =$ "00000000".

Phase d'observabilité locale des listes de fautes propagées

L'observabilité des fautes a lieu au sein du composant MPE10 et la faute $DS1_1$ ayant une influence localement observable sur le signal de sortie DO $(v_c(DO) \neq v_{cf}(DO))$ elle est ajoutée à la liste L_O. Comme elle est déjà présente dans L_O, nous n'effectuons qu'une mise à jour de sa signature.

	Pilotes de valeurs	L_O
REG	["00000000", -1, -1]	$[STRB_1^{\{STRB,['1',None,0]\}},$
STRB	['0','0',0]	$DS1_1^{\{(ENBLD,['1',None,0]),(DO,[None,\text{"00000000"},0])\}},$
ENBLD	['0',-1,-1]	$F2_{MC6}0^{\{DO,[\text{"00000000"},None,0]\}}, ENBLD_1^{\{DO,[\text{"00000000"},None,0]\}},$
DO	["11111111",-1,-1]	$DO_{00000000}, F3_{MA8}^{\{DO,[\text{"00000000"},None,0]\}}]$

TAB. 4.9: *Tableau des pilotes de REG, STRB, ENBLD et DO au cycle C_0^{+1}.*

Phase de mise à jour des pilotes

Aucune mise à jour des pilotes de valeurs n'est effectuée *(aucune simulation saine)*.

Phase d'analyse de l'activité future

Les signatures des fautes de la liste L_O sont analysées afin de déterminer si les fautes localement observables au cycle C_0^{+1} sont susceptibles d'activer des processus pour des simulations

fautives. Seule la faute $DS1_1$ ayant une influence sur le signal ENBLD peut exécuter le processus P_1. Mais les conditions 4.8 ou 4.9 décrites dans la partie *4.2.2.2* de ce chapitre n'étant pas accomplies, aucun processus ne sera actif pour une simulation fautive. Une mise à jour des signatures des fautes de L_O est effectuée. Seule la signature de la faute $DS1_1$ est accomplie *(en gras sur le tableau 4.10).*

L_O
$[STRB_{1,}^{\{STRB,['1',None,0]\}}, DS1_1^{\{(ENBLD,['1',None,0]),(\mathbf{DO},[\text{"}\mathbf{00000000}\text{"},\mathbf{None},\mathbf{0}])\}},$ $F2_{MC6}0^{\{DO,[\text{"}00000000\text{"},None,0]\}}, ENBLD_1^{\{DO,[\text{"}00000000\text{"},None,0]\}}, DO_{00000000}, F3_{MA8}^{\{DO,[\text{"}00000000\text{"},None,0]\}}]$

TAB. 4.10*: Mise à jour des signatures des fautes de L_O au cycle C_0^{+1}.*

Phase de calcul de la couverture de fautes

Les listes de fautes détectées L_D et localement observable L_O sont données par :

L_D	L_O
$DO_{00000000}$	$STRB_1$
$ENBLD_1$	
$F2_{MC6}0$	
$F3_{MA8}$	
$DS1_1$	

TAB. 4.11*: Listes des fautes détectées L_D et localement observables L_O.*

Comme aucun processus n'a besoin d'être simulé, le calcul de la couverture de fautes est effectué par le composant MPE10. Toutes les fautes présentes dans la liste L_O impliquant le signal de sortie DO à une valeur différente de la valeur saine "11111111" sont déplacées dans la liste des fautes détectées L_D *(voir tableau 4.11) (cf. condition 4.10 dans la partie 4.2.2.2)*. En considérant le modèle de fautes $\{F_1, F_2, F_3\}$, il existe 28 fautes susceptibles de se produire au sein de la description et la couverture de fautes est (*cf. formule 4.11 dans la partie 4.2.2.2) :*

$$CF(\%) = \frac{5}{28} \times 100 = 17,85$$

4.4 Conclusion

Ce chapitre montre comment il est possible d'accomplir une simulation BFS-DEVS dans le domaine des circuits digitaux décrits en VHDL comportemental. Il met en évidence le besoin d'implémenter :

- un **modèle de fautes** spécifique à ce domaine d'application,
- une **librairie de six modèles VHDL BFS-DEVS** avec un comportement fautif pour chacun d'eux.

Le modèle de fautes que nous avons choisi prend en considération trois types de fautes : les fautes de collages de valeurs des signaux et des variables *(type F_1)*, les fautes de collage des branches conditionnelles *(type F_2)* et les fautes de saut d'instructions d'affectations *(type F_3)*. Le type F_1 a été choisi car il permet de représenter 80% des défauts pouvant intervenir dans un circuit [Devadas et al., 1996]. Les types F_2 et F_3 sont considérés car ils permettent de localiser les fautes sur les instructions du circuit. Cette localisation peut conduire à une amélioration de l'écriture du code VHDL.

La librairie, construite par l'utilisateur, est indispensable à la transformation de la représentation "*textuelle*" VHDL vers une représentation plus *"schématique"* : le réseau VHDL BFS-DEVS. Chaque modèle qui constitue cette librairie possède un comportement fautif qui est implémenté dans une nouvelle fonction de transition fautive. L'introduction de cette nouvelle fonction permet d'intégrer et de manipuler facilement les fautes au sein des modèles. Les règles de propagation et d'observabilité des listes de fautes sont également intégrées dans chacun des six modèles VHDL BFS-DEVS.

CHAPITRE 5

Modélisation orientée objet

Dans le chapitre précédent, nous avons présenté l'approche formelle de la simulation BFS-DEVS. Dans ce chapitre nous proposons une architecture orientée objet en utilisant le langage *UML (Unified Modeling Language)* [Booch et al., 1998, Oestereich, 2002]. Cette architecture est abordée sous les aspects statiques et dynamiques grâce à des diagrammes de classes, d'états/transitions ainsi qu'un exemple de digramme de séquence. Ceci permet une représentation complémentaire à l'approche formelle présentée précédemment.

5.1 Approche de description statique

Les diagrammes de classes représentent un ensemble de classes/relations permettant une vue statique de la conception de systèmes logiciels. Le diagramme de classes que nous proposons est constitué de trois paquetages nommés "*DEVSModels*", "*Domain*" et "*X BFS-DEVS Library*" liés par des relations de généralisation/spécialisation entre classes contenues. La figure 5.1 de la page suivante présente ce diagramme de classe. Nous n'y rendons visible que les attributs et les méthodes les plus importants.

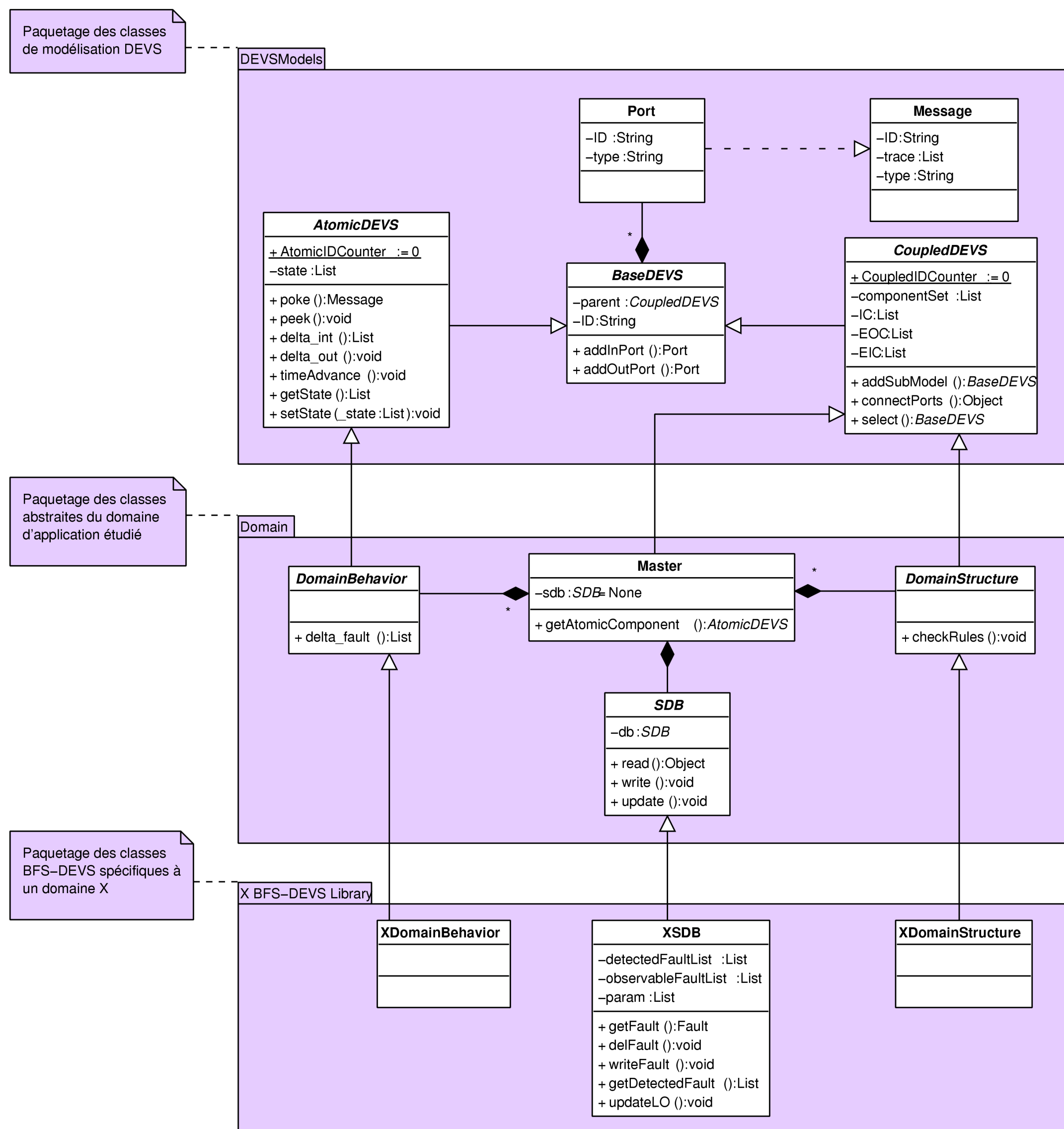


FIG. 5.1: *Diagramme de classes global de l'architecture.*

5.1.1 Le paquetage "*DEVSModels*"

L'arborescence des classes mises en œuvre suit la philosophie DEVS selon laquelle :

- **l'aspect comportemental** d'un système est décrit par l'intermédiaire d'un ensemble de fonctions de transition au sein d'un modèle atomique,
- **l'aspect structurel** est décrit par la définition des interconnexions entre modèles atomiques et/ou couplés au sein d'un modèle couplé.

Le paquetage "*DEVSModels"* de la figure 5.1 [Bolduc et Vangheluwe, 2001] est composé des deux classes abstraites "*AtomicDEVS*" et "*CoupledDEVS*" héritant des propriétés de la classe "*BaseDEVS*". Elles permettent d'implémenter les composant atomiques et couplés DEVS. Les instances de la classe *"Port"* sont utilisées pour représenter les ports de ces composants DEVS. C'est grâce a l'interconnexion des composants via ces ports que la structure du domaine peut se voir définie. La communication entre les composants s'effectue par le biais d'échanges de messages transitant sur les ports. La classe "*Message*" permet donc l'activation et l'échange d'informations entre les composants et est utilisée par la classe "*Port*".

Selon la philosophie de DEVS, il est naturel de considérer l'ensemble des modèles atomiques comme faisant partie d'un domaine comportemental et l'ensemble des modèles couplés comme appartenant à un domaine structurel. Ces deux domaines sont représentés par les deux classes abstraites "*DomainBehavior*" et "*DomainStructure*" constituant le paquetage "*Domain*" de la figure 5.1.

5.1.2 Le paquetage "*Domain*"

Les classes appartenant au paquetage "*Domain*" constituent les bases de notre architecture. La classe principale "*Master*" hérite des propriétés d'un "*CoupledDEVS*" et constitue le modèle couplé principal *(de plus haut niveau de description)* du domaine étudié. Il ne peut exister qu'une seule instance de cette classe "*Master*" à la manière d'un pattern syngleton qui est accédée de manière statique par les classes "*DomainBehavior*", "*DomainStructure*" et "*SDB*". La méthode $getAtomicComponent()$ permet d'accéder à l'instance d'une classe "*AtomicDEVS*" à partir de son ID. C'est à l'intérieur de la classe "*DomainBehavior*" que l'on définit la fonction delta_ fault() permettant de spécifier le comportement fautif d'un modèle atomique.

La classe "*DomainStructure*" permet de décrire les éléments structuraux du domaine qui seront utilisés par l'instance de la classe "*Master*". La méthode $checkRules()$ permet de définir des règles de construction d'un réseau de modèles. La classe "*SDB*" *(Symbolic Data Base)* est introduite afin de faciliter l'accès *(méthodes $read()$ et $write()$)* et la mise à jour *(méthode $update()$)* de la structure de données du domaine étudié qui permet de stocker les objets manipulés par le domaine.

Il n'existera qu'une seule instance de cette classe qui n'est utilisée que par l'instance de la classe "*Master*".

5.1.3 Le paquetage "*X BFS-DEVS Library*"

Comme il est montré sur la figure 5.1, les classes "*XDomainBehavior*", "*XSDB*" et "*XDomain-Structure*" sont propres au domaine X étudié et constituent le paquetage "*X BFS-DEVS Library*". Pour chaque domaine X, il est nécessaire d'implémenter de manière spécifique ces trois classes. Toutes les classes responsables du comportement du système appartenant au domaine X sont des spécialisations de la classe "*XDomainBehavior*" et possèdent un comportement fautif spécifié au travers de la fonction delta_ fault(). La cascade de spécialisation assure que ces classes possèdent les propriétés de la classe "*AtomicDEVS*". De même, les classes responsables de la structure du domaine X sont des spécialisations de la classe "*XDomainStructure*". Comme dans le cas précédent, la cascade de spécialisation assure que ces classes héritent des propriétés de la classe "*Coupled-DEVS*". La structure de donnée propre au domaine X est représentée par la classe "*XSDB*".

En résumé, pour un domaine X particulier, il n'existe qu'une seule instance de la classe "*Master*". Cette instance contient une unique instance de la classe *"XSDB"* comme attribut statique. Les modèles atomiques BFS-DEVS sont représentés par des instances de classes héritant de la classe "*XDomainBehavior*" et les modèles couplés BFS-DEVS par des instances de classes héritant de la classe "*XDomainStructural*". L'ensemble de ces classes constitue la librairie de composants BFS-DEVS du domaine X. Chaque application de X est modélisée grâce l'association des instances de ces classes.

5.1.4 Le paquetage *“VHDL BFS-DEVS Library”*

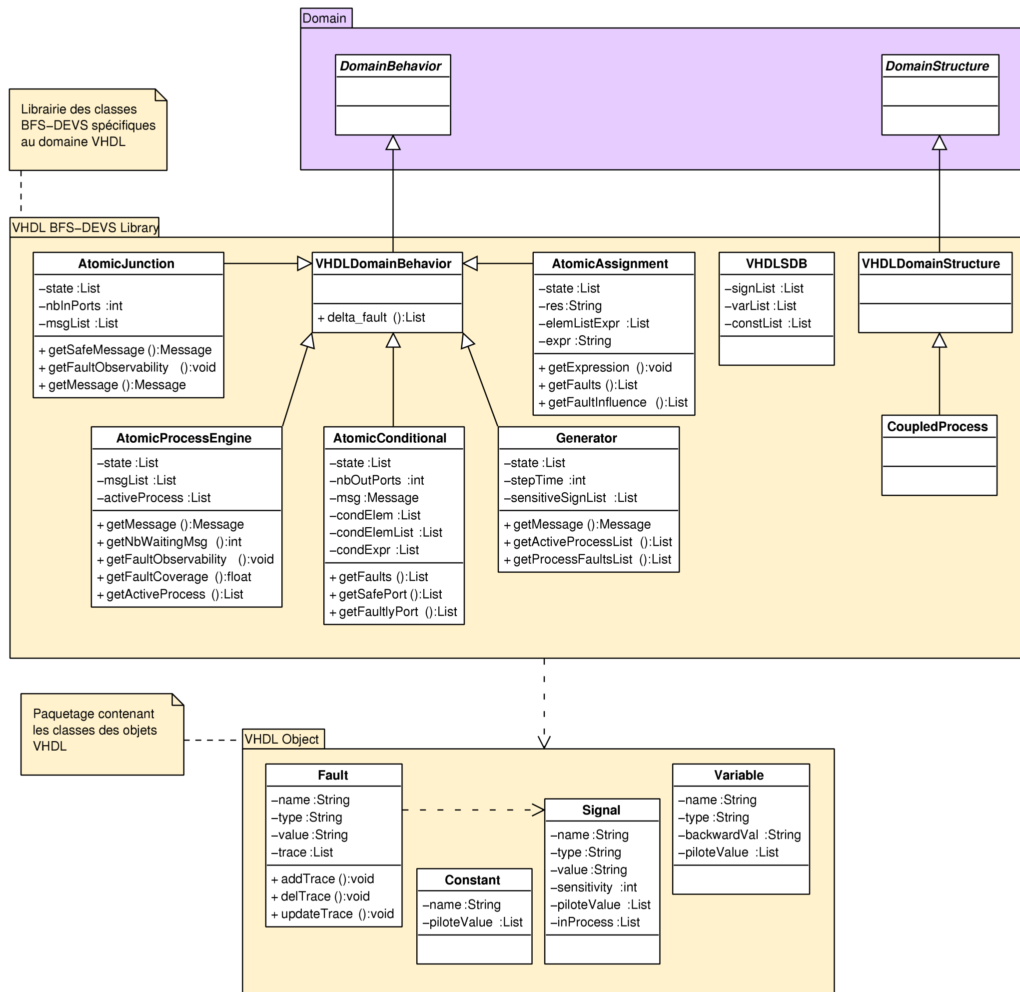


FIG. 5.2*: Diagramme des classes pour le domaine VHDL.*

La figure 5.2 montre le paquetage “*VHDL BFS-DEVS Library*” constitué des quatre classes “*AtomicJunction*”, “*AtomicAssignment*”, “*AtomicProcessEngine*” et “*CoupledProcess*” implémentant les instructions VHDL et dérivant des classes“*VHDLDomainBehavior*” et “*VHDLDomain-Structure*”. Chacune de ces classes spécifie son comportement fautif par l’intermédiaire de sa méthode delta_ fault() surchargée à partir de la classe *“DomainBehavior”*. La classe “*VHDLSDB*” implémente la classe “*SDB*” et représente la structure de données du langage VHDL. Elle stocke les pilotes des objets VHDL, les attributs VHDL et les méthodes de mise à jour.

Le paquetage “*VHDL BFS-DEVS Library*” dépend du paquetage “*VHDL Object*” rassemblant les classes “*Fault*”, “*Signal*”, “*Variable*” et “*Constant*”. Ces classes sont utilisées par l’ensemble

des classes appartenant au paquetage “*VHDL BFS-DEVS library*” . Elles permettent d’implémenter les fautes mais également les objets propres au domaine du VHDL comme les signaux, les variables et les constantes qui sont stockés par l’instance de la classe *“VHDLSDB”* *(méthodes $signList()$,$varList()$ et $constList()$).*

La classe “*AtomicJunction*” possède la méthode $getFaultObservability()$ qui lui permet d’effectuer l’observabilité des fautes contenues dans les messages fautifs obtenus par la méthode $getMessage()$. La classe “*AtomicProcessEngine*” permet l’observabilité des listes de fautes *(obtenus par la méthode $getMessage()$)* et le calcul de la couverture de fautes *(cf. formule 4.11 de la partie 4.2.2.2)* grâce aux méthodes $getFaultObservability()$ et $getFaultCoverage()$. Ces tâches s’effectuent en fin de simulation lorsque tous les messages sont parvenus à l’instance de la classe *(méthode $getNbWaitingMsg()$).*

La classe “*AtomicConditional*” possède la méthode $getFault()$ afin de déterminer la liste des fautes lorsqu’une de ces instances réceptionne un message sain. Les méthodes $getSafePort()$ *(resp. $getFaultlyPort()$)* permet de sélectionner les ports de sorties actifs sains *(resp. fautifs).*

La classe “*Generator*” possède la méthode $getActiveProcessList()$ permettant l’activation des processus avec les listes de fautes obtenues grâce à la méthode $getProcessFaultList()$. La classe “*AtomicAssignment*” évalue l’expression $expr$ grâce à la méthode $getExpression()$ et détermine les fautes pouvant intervenir par la méthode $getFaults()$. L’évaluation d’une liste de fautes par une instance de cette classe est effectuée par la méthode $getFaultInfluence()$.

5.1.5 Le paquetage *"DEVSSimulator"*

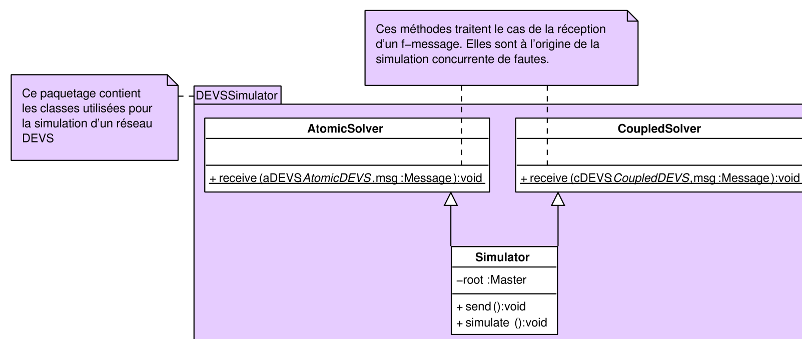


FIG. 5.3*: Diagramme des classes pour le simulateur DEVS.*

Le paquetage "*DEVSSimulator*" montré sur la figure 5.3 regroupe les classes intervenant dans le processus de simulation concurrente des fautes des modèles BFS-DEVS. Ce paquetage contient les classes "*AtomicSolver*", "*CoupledSolver*" et "*Simulator*". La classe "*AtomicSolver*" implémente le simulateur associé à un modèle atomique BFS-DEVS. La classe "*CoupledSolver*" implémente le simulateur *(le coordinateur)* associé à un modèle atomique BFS-DEVS. Les *méthodes statiques receive*() permettent de modifier le comportement des modèles simulés en fonction du type de message passé en paramètre. Cependant, le comportement fautif est simulé grâce à la réception d'un *f-message*. La classe "*Simulator*" est une spécification des deux classes "*AtomicSolver*" et "*CoupledSolver*". La méthode *simulate*() permet d'exécuter la simulation de l'instance du *"Master"*. L'instance de la classe "*Simulator*" peut activer un "*AtomicSolver*" ou un "*CoupledSolver*" grâce à la méthode *end*().

5.2 Approche de description dynamique

Dans cette partie nous abordons l'aspect dynamique des modèles BFS-DEVS spécifiés précédemment, grâce à l'utilisation de diagrammes d'états/transitions et de diagrammes de séquence.

5.2.1 La classe *"Generator"*

La dynamique comportementale de la classe "*Generator*" est représentée par l'automate à états finis de la figure 5.4.

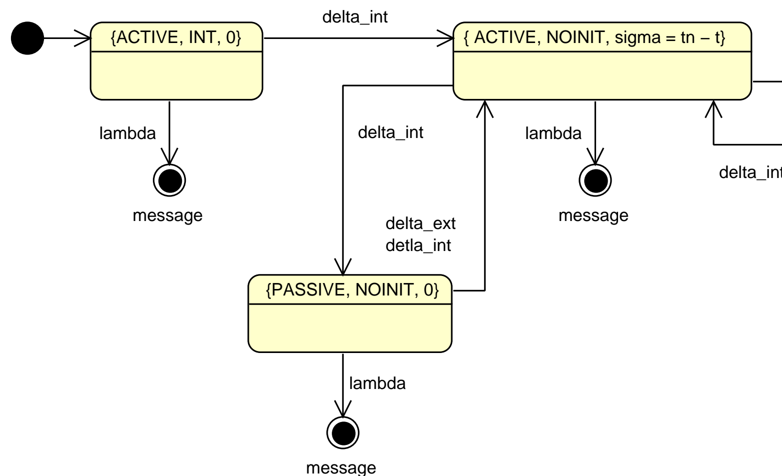


FIG. 5.4*: Graphe d'états du modèle atomique BFS-DEVS* "Generator".

L'automate possède trois états et chacun d'eux est représenté par trois variables :

1. La variable d'état $status \in [{}'ACTIVE', {}'PASSIVE']$ permettant de distinguer le mode de fonctionnement de la classe "*Generator*" :
   - $status = {}'ACTIVE'$, si le *"Generator"* à prévu dans son échéancier la génération de messages de sortie pour un cycle VHDL physique,
   - $status = {}'PASSIVE'$, si le *"Generator"* reçoit des messages pour débuter un cycle VHDL symbolique.
2. La variable d'état $phase \in [{}'INIT', {}'NOINIT']$ permettant de distinguer le mode d'initialisation $(phase = {}'INIT')$ du mode permanent $(phase = {}'NOINIT')$.
3. La variable d'état $sigma \in \mathbb{R}^+_\infty$ représentant la durée de vie d'un état donné du *"Generator"*.

L'automate possède également un état entrée/sortie $\{{}'ACTIVE', {}'INIT', 0\}$ correspondant à l'état d'initialisation des processus *(ou des modèles couplés).* Les deux états de sortie $\{{}'ACTIVE', {}'NOINIT', t_n - t\}$ et $\{{}'PASSIVE', {}'NOINIT', 0\}$ sont accessibles en régime permanent et sont atteints à chaque fin d'un cycle physique et resp. d'un cycle symbolique.

A l'initialisation, le *"Generator"* est dans l'état $\{'ACTIVE','INIT',0\}$, son temps de vie *sigma* étant nul, il génère aussitôt grâce à la fonction de sortie *lambda* des messages d'activation $M_{safe}(\emptyset)$ vers ses modèles couplés et la fonction de transition interne $delta_int$ le fait passer en régime permanent transformant son état en $\{'ACTIVE','NOINIT',t_n - t\}$. Le temps de vie $lambda = t_n - t$ de ce nouvel état est calculé en faisant la différence du temps du prochain événement t_n par le temps de simulation t. Lorsque le temps de simulation devient égal au temps calculé t_n et qu'aucune entrée n'est présente sur le "Generator", l'état courant prend fin. Le *"Generator"* débute un nouveau cycle physique en générant des messages et un nouvel état $\{'ACTIVE','NOINIT',t'_n - t\}$ prend place grâce à la fonction $delta_int$. Cette état diffère du précédent car $t'_n > t_n(= t)$ dans l'échéancier. Si $t_n - t \neq 0$ et qu'un événement externe intervient, le *"Generator"* passe dans l'état $\{'PASSIVE','NOINIT',0\}$ grâce à la fonction de transition externe $delta_ext$. Il transmet alors les messages d'entrées reçus sur ces sorties par la fonction de sortie *lambda* et revient aussitôt $(sigma = 0)$ dans l'état $\{'ACTIVE','NOINIT',t'_n - t\}$.

5.2.2 La classe "*Assignment*"

La dynamique comportementale du composant *"Assignment"* est représentée par l'automate à états finis de la figure 5.5.

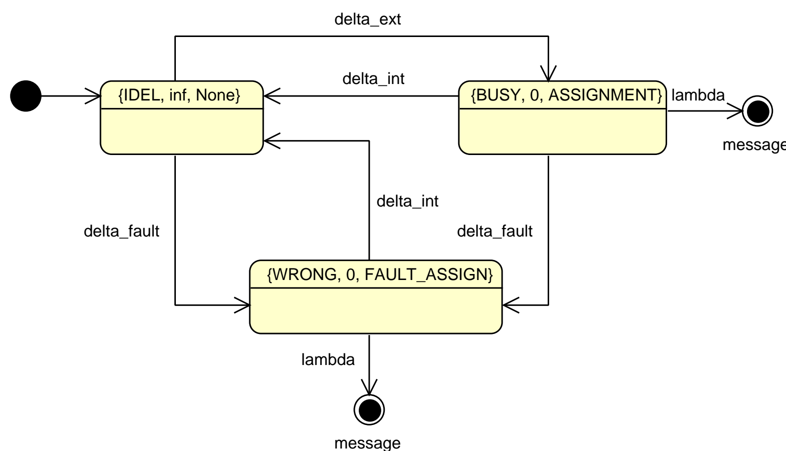


FIG. 5.5*: Graphe d'états du modèle atomique BFS-DEVS* "Assignment" .

L'automate possède trois états et chaque état est représenté par les trois variables :

1. La variable d'état $status \in ['IDLE','BUSY','WRONG']$ permettant de distinguer le mode de fonctionnement du composant :
   - $'IDLE'$ si le composant est au repos en attente d'un message,
   - $'BUSY'$ si le composant reçoit un message du type $M_{safe}(\emptyset)$ ou $M_{safe}(L)$,
   - $'WRONG'$ si le composant reçoit un message du type $M_{safe}(L)$ ou $M_{fault}(L)$.
2. La variable d'état $sigma \in \mathbb{R}^+_\infty$ représentant la durée de vie d'un état donné du composant.
3. La variable d'état $currentTask \in ['ASSIGN','F_ASSIGN','None']$ permettant de savoir si le composant accompli :
   - l'évaluation de l'instruction d'affectation $('ASSIGN')$ dans le cas d'un statut occupé $('BUSY')$,
   - la détermination des fautes sur l'instruction d'affectation $('F_ASSIGN')$ dans le cas d'un statut occupé mais fautif $('WRONG')$,
   - aucune tâche $('None')$ dans le cas d'un composant au repos $('IDLE')$.

Si le composant *"Assignment"* est dans son état initial et qu'un message du type $M_{safe}(L)$ est réceptionné pour une simulation avec propagation de liste de fautes. La fonction de transition externe $delta_ext$ fait passer le composant dans l'état transitoire $\{'BUSY',0,'ASSIGN'\}$ ou il évalue l'expression d'affectation. Puis la fonction de transition externe fautive $delta_fault$ fait transiter le composant dans le nouvel état $\{'WRONG',0,'F_ASSIGN'\}$ ou il détermine les fautes au sein de l'expression d'affectation. La fonction $delta_fault$ intervient après l'exécution de $delta_ext$ car la détermination de la listes des fautes nécessite la connaissance des valeurs saines de chaque signaux et/ou variables intervenants dans l'expression. Si la simulation se déroule sans propagation de la liste de fautes, la fonction de transition interne $delta_int$ remplace la fonction $delta_fault$ pour faire transiter le composant dans son état initial $\{'IDLE',inf,None\}$ après avoir exécuté la fonction de sortie $lambda$.

Si maintenant le composant est dans son état initial et qu'un message du type $M \in \{M_{fault}(L)\}$ est réceptionné. Nous sommes forcement dans le cas d'une simulation avec propagation de liste de fautes et la fonction $delta_fault$ s'exécute à la place de $delta_ext$ pour amener le composant dans l'état $\{'WRONG',0,'F_ASSIGN'\}$. En effet, ici le composant se trouve sur un chemin fautif, donc

l'expression d'affectation n'est pas évaluée. C'est la fonction $delta_fault$ qui se charge de déterminer la liste des fautes observables sur l'expression d'affectation. Enfin, la fonction $delta_int$ ramène l'état du composant dans sa configuration initiale $\{'IDLE', inf, None\}$ après avoir exécuté la fonction de sortie $lambda$.

5.2.3 La classe *“Conditional”*

La dynamique comportementale du composant *"Conditional"* est représentée par l'automate à états finis de la figure 5.6.

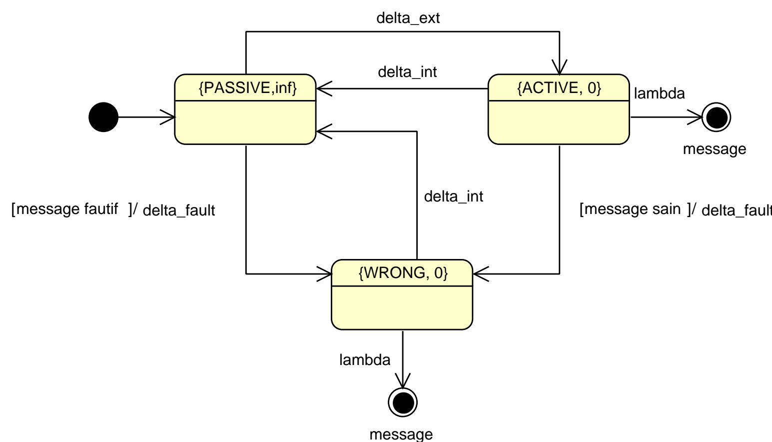


FIG. 5.6*: Graphe d'états du modèle atomique BFS-DEVS* "Conditional".

L'automate possède trois états et chaque état est représenté par les deux variables indépendantes :

1. La variable d'état $status \in ['PASSIVE', 'ACTIVE', 'WRONG']$ permettant de distinguer le mode de fonctionnement du composant *"Conditional"* :
   - $'PASSIVE'$ si le composant est au repos en attente d'un message,
   - $'ACTIVE'$ si le composant est sain et occupé,
   - $'WRONG'$ si le composant est fautif et occupé.
2. La variable d'état $sigma \in \mathbb{R}^{+}_{\infty}$ représentant la durée de vie d'un état donné.

L'état initial du composant $\{'PASSIVE', inf\}$ signifie que celui-ci attend $(sigma = inf)$ dans un configuration passive $(status =' PASSIVE')$ un message d'activation. Trois cas peuvent apparaître

lors de l'arrivée d'un message :

- si la simulation à lieu sans propagation de liste de fautes et le message reçu est $M_{safe}(\emptyset)$: la fonction $delta_ext$ s'active et l'état devient $\{'ACTIVE', 0\}$. La fonction de sortie $lambda$ permet de générer le message de sortie sur l'un des ports de sortie sélectionné en fonction de l'évaluation de l'expression conditionnelle. Enfin la fonction $delta_int$ ramène le composant dans son état sain et passif initial $\{'PASSIVE', inf\}$ dans l'attente d'une nouvelle activation $(sigma = inf)$.
- si la simulation à lieu avec propagation de liste de fautes et que le message est $M_{fault}(L, fsdb)$ *(chemin fautif)* : la fonction $(delta_fault)$ transforme l'état en $\{'WRONG', 0\}$ et permet la réorientation des fautes contenues dans *L*. En effet chaque faute de la liste L est injectée au sein de l'expression conditionnelle et si le résultat de l'évaluation de cette expression infectée diffère de l'évaluation de l'expression dans le cas ou aucune faute est présente, la faute est ajoutée à la liste $L_{0<i\leq N-1}$. A l'issue de cette opération, le composant dispose d'un ensemble de liste L_i associé à chacun de ces ports de sortie *i*. La fonction de sortie $(lambda)$ s'activera pour transmettre les listes L_i non vide sur les port associé *i*. On note que l'évaluation de l'expression utilise les valeurs des signaux ou des variables fautives stocké dans la base de données $fsdb$. Enfin la fonction $delta_int$ ramène le composant dans son état sain et passif initial $\{'PASSIVE', inf\}$ dans l'attente d'une nouvelle activation.
- si la simulation à lieu avec propagation de liste de fautes et le message reçu est $M_{safe}(L)$ *(chemin sain)* : la fonction $delta_ext$ s'active et l'état devient $\{'ACTIVE', 0\}$. L'hypothèse de départ et la parallélisation des simulations saines et fautives au sein du réseau BFS-DEVS implique l'activation de la fonction $delta_fault$. Celle-ci laisse l'état du composant inchangé et permet de déterminer les listes de fautes $L_{0<i<N-1}$ impliquant une fausse évaluation de l'expression conditionnelle *(la vraie valeur étant calculée dans la fonction $delta_ext$ qui précède $delta_fault$)*. La fonction de sortie $lambda$ s'active afin de déterminer le port de sortie sain transmettant le message reçu $M_{safe}(L \cup L_i)$ ainsi que les ports de sortie fautifs acceptants uniquement les messages fautifs $M_{fault}(L_i, fsdb)$ pour lesquels la liste de fautes à propager L_i est non vide. Enfin la fonction $delta_int$ ramène le composant dans son état

sain et passif initial $\{'PASSIVE', inf\}$ dans l'attente d'une nouvelle activation.

5.2.4 La classe "*Junction*"

La dynamique comportementale du composant *"Junction"* est représentée par l'automate à états finis de la figure 5.7.

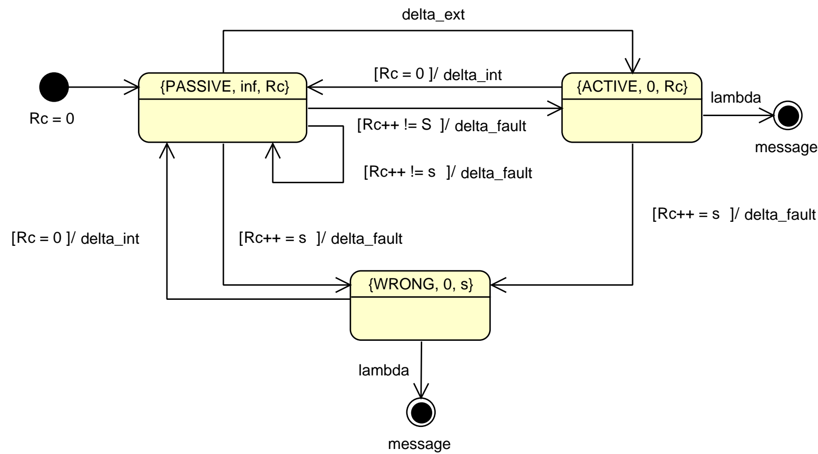


FIG. 5.7*: Graphe d'états du modèle atomique BFS-DEVS* "Junction".

L'automate possède trois états et chacun d'eau est représenté par les trois variables indépendantes suivantes :

1. La variable d'état $status \in ['PASSIVE', 'ACTIVE', 'WRONG']$ permettant de distinguer le mode de fonctionnement du composant :
   - $'PASSIVE'$ si le composant est au repos en attente d'un message,
   - $'ACTIVE'$ si le composant est sain et occupé,
   - $'WRONG'$ si le composant est fautif et occupé.
2. La variable d'état $sigma \in \Re^{+}_{\infty}$ représentant la durée de vie d'un état donné.
3. La variable d'état R_c permet de compter le nombre de messages arrivant sur le composant. Initialement nulle, elle s'incrémentera à chaque réception d'un message jusqu'à atteindre le nombre s de messages imposé et transmis par le premier composant *"Conditional"* rencontré en amont du réseau.

L'état initial du composant $\{'PASSIVE', inf, R_c = 0\}$ signifie que celui-ci attend $(sigma = inf)$ dans un configuration passive $(status =' PASSIVE')$ l'arrivée des messages d'activation $(R_c = 0)$. A chaque arrivée d'un message 5 cas peuvent apparaître suivant le type des messages reçus, le mode de simulation et la valeur de R_c :

1. Si la simulation se déroule sans propagation de listes de fautes, le message reçu est $M_{safe}(\emptyset, s = 1)$ et $R_c = s = 1$: la fonction $delta_ext$ transforme l'état du composant en $\{'ACTIVE', 0, R_c\}$ l'amenant dans une configuration active. Le composant n'est sensé recevoir qu'un seul message du type sain et la fonction de sortie *lambda* permet de transmettre ce message. Enfin la fonction $delta_int$ rétablit une configuration passive $\{'PASSIVE', inf, R_c = 0\}$ au sein du composant.
2. Si la simulation se déroule avec propagation de listes de fautes, le message reçu est $M_{safe}(L, s)$ et $R_c = s$: la fonction $delta_ext$ faisant passer le composant dans l'état $\{'ACTIVE', 0, R_c\}$ est aussitôt $(sigma = 0)$ succédé par la fonction de transition externe fautive $delta_fault$. Celle-ci permet d'incrémenter R_c et fait passer le composant dans une configuration fautive $\{'WRONG', 0, R_c = s\}$ afin d'effectuer l'observabilité des fautes de chaque liste $L_{0<i \leq s-1}$ apportées par les $s-1$ messages fautifs. La fonction de sortie *lambda* permet de générer le message de sortie sain. Le composant ne recevra plus de message pour le cycle courant et la fonction $delta_int$ rétablit aussitôt l'état passif de départ $\{'PASSIVE', inf, R_c = 0\}$.
3. Si la simulation se déroule avec propagation de listes de fautes, le message reçu est $M_{safe}(L, s)$ et $R_c < s$: suite à l'activation de la fonction $delta_ext$ puis de la fonction $delta_fault$ le composant passe dans un état proche de l'état initial $\{'PASSIVE', inf, R_c\}$ ou R_c n'est pas réinitialisé à zéro. Temps que le nombre de message reçu désiré n'est pas atteint, R_c s'incrémente à chaque réception et l'état reste inchangé. Aucune sortie est générée et la fonction $delta_int$ rétablit une configuration passive $\{'PASSIVE', inf, R_c = 0\}$ au sein du composant.
4. Si la simulation se déroule avec propagation de listes de fautes, le message reçu est $M_{fault}(L_i, s)$ et $R_c = s$: l'exécution de la fonction de transition externe fautive $delta_fault$ incrémente R_c et l'état devient $\{'WRONG', 0, s\}$. Comme $R_c = s$ elle effectue également l'analyse de l'observabilité des fautes de chaque liste $L_{0<i \leq s-1}$ apportées par les $s-1$ message fautifs.

Connaissant le nouvel état, la fonction de sortie *lambda* permet la génération du message sain sur l'unique sortie du composant. La fonction *delta_int* rétablit l'état initial $\{'PASSIVE', inf, R_c = 0\}$.

5. Si la simulation se déroule avec propagation de listes de fautes, le message reçu est $M_{fault}(L_i, s)$ et $R_c < s$: l'état reste inchangé.

5.2.5 La classe "*ProcessEngine*"

La dynamique comportementale du *"ProcessEngine"* peut être représentée à l'aide de l'automate à états finis montré sur la figure 5.8 :

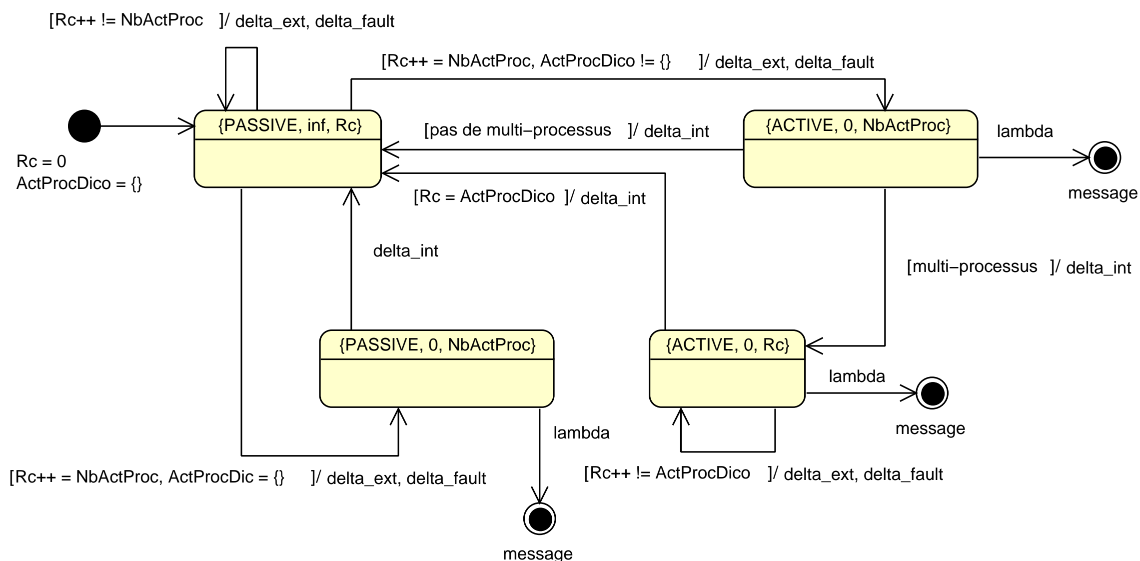


FIG. 5.8: *Graphe d'états du modèle atomique BFS-DEVS* "ProcessEngine"

L'automate est composé de trois places qui sont caractérisées par les trois variables d'états suivantes :

1. La variable d'état *status* $\in$ $['ACTIVE','PASSIVE']$ permettant de distinguer le mode de fonctionnement du composant :
   - $'PASSIVE'$, le composant génère les messages sur les *N* sorties primaires *Out (fin d'un cycle de simulation physique)*,

- $'ACTIVE'$, mise à part l'état initial, le composant génère les messages sur ces $\{1\ldots N\}$ ports de sorties feedbacks en direction des $\{1\ldots N\}$ ports d'entrées du *"Generator"* afin d'activer les $\{1\ldots N\}$ modèles couplés pour de nouveaux cycles de simulations symboliques.

2. La variable d'état *sigma* $\in \mathbb{R}^{+}_{\infty}$ représentant la durée de vie d'un état donné.

3. La variable d'état R_c permet de compter le nombre de messages arrivant sur le composant. Initialement nulle, elle s'incrémentera à chaque réception d'un message jusqu'à atteindre le nombre *NbActProc*. Ce nombre correspond au nombre de modèles couplés actifs au cycle de simulation courant et il est contenu dans chaque message arrivant sur le composant.

L'automate possède un état d'entrée $\{'PASSIVE', inf, R_c = 0\}$ correspondant à l'état initial du *"ProcessEngine"*. Cet état reste inchangé tant que l'arrivée d'un message $(sigma = inf)$ ne provoque la transition vers l'un des deux états de sortie $\{'PASSIVE', 0, NbActProc\}$ ou

$\{'ACTIVE', 0, NbActProc\}$. Cette transition est effectuée grâce à la fonction de transition externe $delta_ext$ et dépendra :

- du nombre de message déjà reçu *(comptabilisé par la variable d'état initialement nulle R_c)*,
- du contenu du dictionnaire des processus actifs au prochain cycle de simulation symbolique *(représenté par la variable ActProcDico)*.

Lorsque un message *(sain ou fautif)* arrive sur le composant *"ProcessEngine"* possédant l'état initial $\{'PASSIVE', inf, R_c = 0\}$ la fonction $delta_ext$ s'exécute et différent cas sont envisageables :

- Si $R_c \neq NbActProc$: l'état reste inchangé et R_c est incrémenté.
- Si $R_c = NbActProc$:
  - Si le dictionnaire des processus actifs pour un prochain cycle symbolique est vide : l'état du composant *"ProcessEngine"* devient $\{'PASSIVE', 0, NbActProc\}$. La fonction de sortie *lambda* permet d'effectuer la mise à jour des pilotes de valeurs de tout les signaux et de toutes les variables avant de générer sur les N sorties "primaire" *Out* les signaux de sortie mettant fin au cycle de simulation physique.
  - Dans le cas contraire ou le dictionnaire des processus actifs n'est pas vide $ActProcDico \neq \emptyset$: l'état du composant *"ProcessEngine"* devient $\{'ACTIVE', 0, NbActProc\}$. La fonc-

tion de sortie *lambda* permet d'effectuer la mise à jour des pilotes de valeurs de tous les signaux et de toutes les variables. Ensuite le composant génère sur ces *M* sorties *FeedBack* des messages d'activations destinés à informer le *"Generator"* du début d'un nouveau cycle symbolique pour les modèles couplés qu'il commande.

– Si au moins un modèle couplé correspondant à un processus doit être activé plus d'une fois au cycle de simulation symbolique courant, nous sommes alors en présence d'un simulation "inter-processus". Ce cas apparaît uniquement lorsque la description sous simulation possède plusieurs processus .En effet avec une telle configuration il peut arriver que plusieurs processus, après avoir été simulés, modifient les signaux sensitifs d'un autre processus. Ce processus demande donc d'être simulé pour une prochaine cycle symbolique. Pour chaque processus devant être activé x fois $(x > 1)$ au cycle de simulation symbolique courant, le composant *"ProcessEngine"* générera sur sa sortie *FeedBack* en direction du processus *(par l'intermédiaire du "Generator")* x messages d'activation toujours au cycle de simulation symbolique courant. Le composant *"ProcessEngine"* passe donc dans l'état $\{'ACTIVE', 0, R_c\}$ et restera dans celui-ci à chaque réception d'un message tant que le compteur R_c n'a pas atteint le nombre de processus actif pour le cycle symbolique *NbActProc*. L'observabilité des fautes contenues dans chaque liste appartenant aux messages reçus est accompli au sein de la fonction de sortie *lambda* lorsque le seuil est atteint *(à la fin du cycle symbolique)*. De plus l'observabilité des fautes contenues dans les x messages *(sain et/ou fautifs)* ayant servis à l'activation d'un processus s'effectue par analyse des signatures et par comparaison des pilotes de valeurs des x *(reps. x-1)* bases de données fautives si les x messages sont tous du type fautifs *(reps. ne sont pas tous fautif)*. Enfin la fonction de transition interne $delta_int$ rétablit l'état initial $\{'PASSIVE', inf, R_c\}$.
– Si aucun des modèles couplés n'a besoin d'être activé au cycle de simulation symbolique courant, alors l'observabilité des fautes est effectuée par analyse des signatures et des bases de données fautives et la fonction $delta_int$ fait revenir le composant dans son état initial $\{'PASSIVE', inf, R_c\}$.

5.2.6 Gestion des messages d'un arbre de simulation

Dans cette partie, nous donnons un exemple de la gestion des messages intervenant dans l'arbre de simulation BFS-DEVS. Nous considérons un modèle BFS-DEVS avec son arbre de simulation montré sur la figure 5.9 (a) et (b). Le modèle couplé *MC* possède deux modèles atomiques MA_1 et MA_2. Le modèle MA_2 peut être connecté à d'autre modèles de *MC (flèche en pointillé sur la sortie de ce modèle)*. L'arbre de simulation associé au modèle global est composé de deux simulateurs, sous le contrôle d'un unique coordinateur géré par le coordinateur de plus haut niveau, le *"Root" (cf. partie 2.3.2)* .

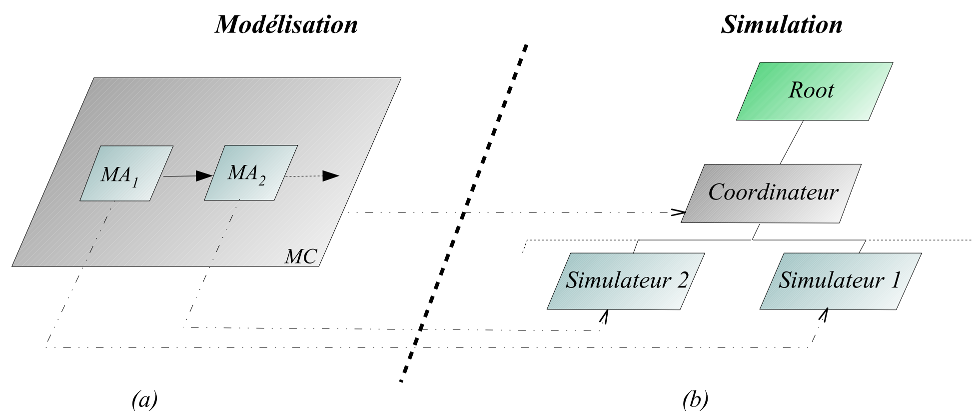


FIG. 5.9*: Exemple d'un modèle BFS-DEVS (a) avec son arbre de simulation (b).*

Le diagramme de séquence de la figure 5.10 *(page suivante)* illustre l'échange de messages dans l'arbre de simulation. Il montre également à quel moment s'effectue l'exécution des fonctions de transitions, des fonctions de sortie et des fonctions d'avancement du temps. Ce diagramme est divisé en deux parties : une première partie illustre l'initialisation des modèles et une deuxième partie montre la génération d'un message sur le modèle MA_2.

Au début de la simulation, une instance de *"Root"* est créée et envoie un *i-message* sur l'instance *"Coordinateur"* . Ce message est généré pour enclencher la phase d'initialisation des modèles. L'instance *"Coordinateur"* initialise les variables temporelles du modèle *MC* et transmet ce message à ses deux fils, les instances *"Simulateur 1"* et *"Simulateur 2",* afin qu'ils puissent également initialiser les variables temporelles des modèles atomiques MA_1 et MA_2. Les deux simulateurs sont par la suite montrés détruits puisque les données sont stockées dans les instances

des modèles atomiques et que ne sont utilisées que leurs methodes statiques. La même remarque s'applique à chaque description d'objet montré sur le diagramme de la figure 5.10.

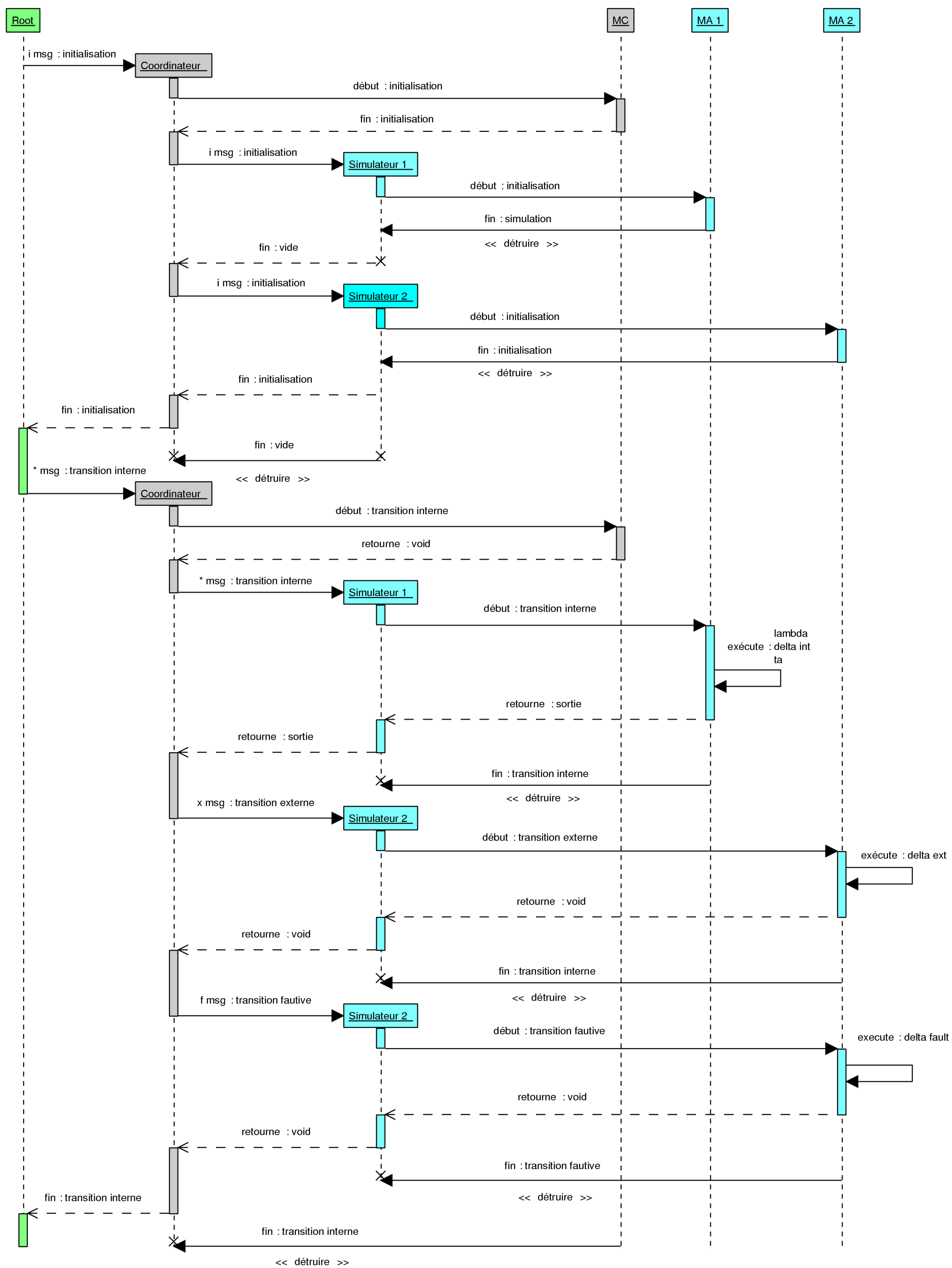


FIG. 5.10: *Diagramme de séquence de l'algorithme de simulation BFS-DEVS.*

Lorsque le *"Coordinateur"* informe le *"Root"* de la fin de la phase d'initialisation, celui-ci

envoie un **-message* à une instance du modèle *MC* qui doit s'exécuter. Lorsque cette instance reçoit le **-message*, elle le transmet au simulateur du fils imminent. Si l'on considère que le modèle MA_1 est le fils imminent, alors le *"Coordinateur"* envoie le **-message* à l'instance *"Simulateur 1"*. Cette instance exécute *dans l'ordre (cf. partie 2.3.1.1)*, la fonction de sortie *lambda*, la fonction de transition interne $delta_int$ et t_a. Lorsque l'exécution est terminée, elle retourne au *"Coordinateur"* la sortie générée par le modèle MA_1 avant d'être détruite. Cette sortie est prise en compte par le *"Coordinateur"* qui va consulter ses couplages internes pour savoir si la sortie générée ne doit pas être transmise à un de ses fils. Le modèle MA_2 est connecté au modèle MA_1 qui vient de générer une sortie. Dans ce cas, le modèle MA_2 doit être exécuté et le *"Coordinateur"* envoie un *x-message* puis un *f-message* en direction de l'instance *"Simulateur 2"*. Cette instance exécute d'abord la fonction de transition externe $delta_ext$ puis la fonction de transition fautive $delta_fault$ du modèle MA_2 *(cf. algorithme 5 dans la partie 3.3.2.3)*.

L'instance *"Root"* est toujours en activité et envoie des **-messages* tant que le temps de simulation final n'est pas atteint et les instances *"Coordinateur"*, *"Simulateur 1"* et *"Simulateur 2"* sont détruites à chaque fin de traitement d'un message.

5.3 Conclusion

Dans ce chapitre, nous nous sommes appuyés sur le langage UML pour décrire les aspects dynamiques et statiques de notre simulateur BFS-DEVS. Nous avons ainsi défini une architecture orientée objet du simulateur pour l'implémentation d'un prototype. Une approche statique a présenté les paquetages de classes de notre achitecture en mettant en évidence la *séparation* entre le domaine d'étude X et le noyau de simulation/modélisation BFS-DEVS. L'aspect dynamique de notre prototype a été présenté sur un exemple d'arbre de simulation BFS-DEVS en utilisant un diagramme de séquence. De plus, le comportement des six modèles VHDL BFS-DEVS de base a été décrit à l'aide des diagrammes d'états/transitions. Ces étapes de description ont défini les bases de l'implémentation du noyau de simulation de notre prototype.

CHAPITRE 6

Expériences et résultats

L'approche générale de simulation BFS-DEVS avec propagation de liste de fautes montrée dans cette thèse à été appliquée sur les systèmes digitaux décrits en VHDL. Nous avons implémenté en langage Python [Beazley, 2000] le simulateur de fautes concurrent BFS-DEVS ainsi qu'une bibliothèque de composants VHDL BFS-DEVS. L'implémentation de ce simulateur dérive du noyau de simulation [Bolduc et Vangheluwe, 2001] conforme aux spécifications DEVS et consiste en 2500 lignes de code Python. La bibliothèque des six composants VHDL BFS-DEVS permettant la construction de réseau BFS-DEVS consiste en 5000 lignes de code Python. Les résultats montrés dans ce chapitre sont obtenus à partir d'un sous ensemble de benchmarks ITC'99 [Corno et al., 2000b, Corno et al., 1997] développés par l'école Polytechnique de Turin.

6.1 Les benchmarks ITC'99

Les benchmarks ITC'99 ont été développés par le groupe CAD de l'école Polytechnique de Turin et sont un ensemble de circuits synthétisables utilisés par plusieurs compagnies d'électronique pour l'expérimentation dans le domaine de la testabilité et de la génération automatique

de séquences de test. Ils ont été construits afin de couvrir une large catégorie de circuits et ils donnent la possibilité de tester des mémoires, des aspects temporelles, des bus interne, un nombre important de portes logiques et des descriptions RTL ou/et comportementales. Les caractéristiques VHDL des benchmarks sont montrées sur le tableau 6.1. Les colonnes résument les informations concernant les descriptions VHDL en termes de lignes, de nombres de processus, de nombres d'instructions d'affectations et conditionnelles, de nombres de signaux et de variables. C'est en exploitant les 10 premiers benchmarks ITC'99 que nous analysons la faisabilité et la rapidité de notre simulateur de fautes concurrent basé sur le formalisme BFS-DEVS.

benchmarks	**#lignes**	**#proc**	**#affectations**	**#conditionnelles**	**#signaux**	**#variables**
B01	110	1	35	12	6	1
B02	70	1	19	7	4	1
B03	141	1	56	14	7	14
B04	102	1	40	12	7	13
B05	310	3	99	46	19	5
B06	128	1	50	11	8	1
B07	92	1	33	9	4	6
B08	89	1	22	8	8	4
B09	103	1	34	8	7	2
B10	167	1	74	19	13	8

TAB. 6.1*: Caractéristiques de quelques benchmarks ITC'99.*

6.2 Critère de couverture de fautes

La couverture de fautes caractérise l'aptitude d'un test à détecter des fautes. A la fin de la simulation de fautes, il est possible de dresser la liste des types de fautes qui ont pu être commises. On définit alors la couverture de fautes d'un ensemble de tests comme le nombre de fautes détectées sur le nombre total de fautes. Le critère de couverture de fautes est le plus utilisé pour valider une séquence test. Il a été observé depuis très longtemps qu'un taux de couverture élevé vis à vis des fautes de collages simple (F_1) s'accompagne d'un bon taux de couverture des défauts réels [WILLIAMS et BROWN, 1981].

La connaissance du modèle de fautes et des caractéristiques du benchmark suffisent à la détermination du *nombre total de fautes* N_T susceptibles de se produire au sein d'une description

VHDL. Dans notre cas, ce nombre $N_T = N_{F_1} + N_{F_2} + N_{F_3}$ est la somme de trois termes :

- N_{F_1} est le nombre de fautes du type F_1. Ce terme est égal au nombre de valeurs possibles du domaine de définition des signaux et des variables VHDL.
- N_{F_2} est le nombre de fautes du type F_2. Ce nombre est égal à la somme du nombre de solution de chaque instruction conditionnelle VHDL.
- N_{F_3} est le nombre de fautes du type F_3. Ce nombre est égal au nombre d'instruction d'affectation présentent dans la description VHDL.

Si N_D est le nombre de fautes détectées à l'issue de la simulation de fautes BFS-DEVS, le *pourcentage de Couverture de Fautes* $CF(\%)$ est donnée par *(cf. formule 4.11 de la partie 4.2.2.2)* :

$$CF(\%) = \frac{N_D}{N_T} = \frac{\text{nombre de fautes détectées}}{\text{nombre total de fautes}} * 100 \qquad (6.1)$$

D'une manière plus générale, la couverture de test est définit dans le domaine du test logiciel comme le rapport du nombre d'objets testés sur le nombre total d'objets. Un objet à tester peut être soit un bloc d'instruction, soit un chemin de décision, soit un saut d'instruction. Dans le premier cas, on parle de *couverture d'instructions*. Nous utilisons ce critère afin de savoir si la séquence de test a bien testée toutes les instructions du code VHDL. Si tel est le cas et si la couverture de fautes est faible, on pourra alors juger le niveau de testabilité de la description sous test [Seth et al., 1990].

6.3 Validation du prototype BFS-DEVS

Le but de cette partie est de montrer la faisabilité de notre approche en implémentant un prototype du simulateur BFS-DEVS. Ce prototype est utilisé pour simuler les fautes du type $\{F_1, F_2, F_3\}$ de manière concurrente sur un sous ensemble des benchmarks ITC'99. Nous analysons l'évolution des couvertures de fautes obtenues après simulation et nous montrons comment il est possible d'améliorer le code VHDL en fonction des résultats de la simulation des fautes de type F_3.

6.3.1 Architecture du prototype

L'architecture du prototype BFS-DEVS est montré sur la figure 6.1.

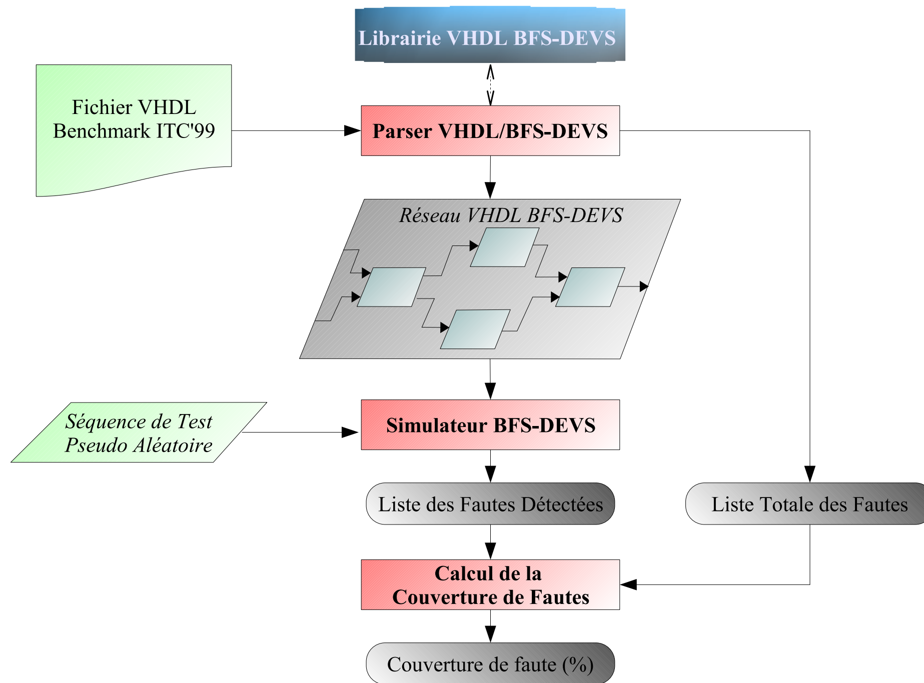


FIG. 6.1*: Architecture générale du prototype BFS-DEVS.*

Un parser VHDL/BFS-DEVS issu d'une adaptation de l'outil *GENESI (GENErateur de StimulI)* [Paoli et al., 2004] permet de donner le fichier représentant le réseau BFS-DEVS. Ce réseau est simulé afin de générer la liste des fautes détectées par une séquence de test pseudo-aléatoire. Cette séquence est générée de manière pseudo-aléatoire car nous voulons uniquement montrer la faisabilité de notre simulateur BFS-DEVS. Elle est composée d'un nombre de vecteurs égal au nombre de signaux d'entrées et chaque vecteur possède 10,000 valeurs. Comme dans [Corno et al., 1997], les configurations des benchmarks nous permettent de considérer les signaux d'entrées CLOCK et RESET comme non aléatoires :

- Le signal CLOCK de type bit est régulièrement alterné de '0' et de '1'.
- Le signal RESET de type bit est égale à '0' sur des plages de 1000 valeurs entrecoupées par un bit de valeur '1'.

Toutes les valeurs des autres signaux d'entrées sont aléatoires.

Nous simulons les dix premiers benchmarks et le calcul du pourcentage de Couverture de

Fautes *CF(%)* est donné en fin de chaque simulation par la formule 6.1. Nous donnons également le pourcentage de Couverture d'Instructions VHDL *CI(%)* car elle permet d'évaluer la testabilité d'une programme [McCabe, 1976]. Les résultats sont obtenus sur un processeur P3 933 MHz avec 512 Mo de RAM et sont reportés sur le tableau 6.2.

benchmarks	N_T	N_D	**CI(%)**	**CF(%)**
B01	79	73	100	92,94
B02	48	42	100	87,50
B03	130	108	100	83,07
B04	105	89	100	84,76
B05	251	61	69,58	24,30
B06	95	78	100	82,10
B07	76	66	91,83	86,84
B08	64	63	100	98,43
B09	68	57	100	83,82
B10	163	143	100	87,67

TAB. 6.2: *Résultats des simulations BFS-DEVS sur les dix premiers benchmarks ITC'99.*

Le but de cette section est avant tout de montrer la faisabilité de la simulation de fautes par BFS-DEVS. L'application du simulateur concurrent BFS-DEVS sur les 10 premiers benchmarks montre qu'il est possible d'obtenir des couvertures de fautes acceptables compte tenu des séquences pseudo-aléatoire appliquées.

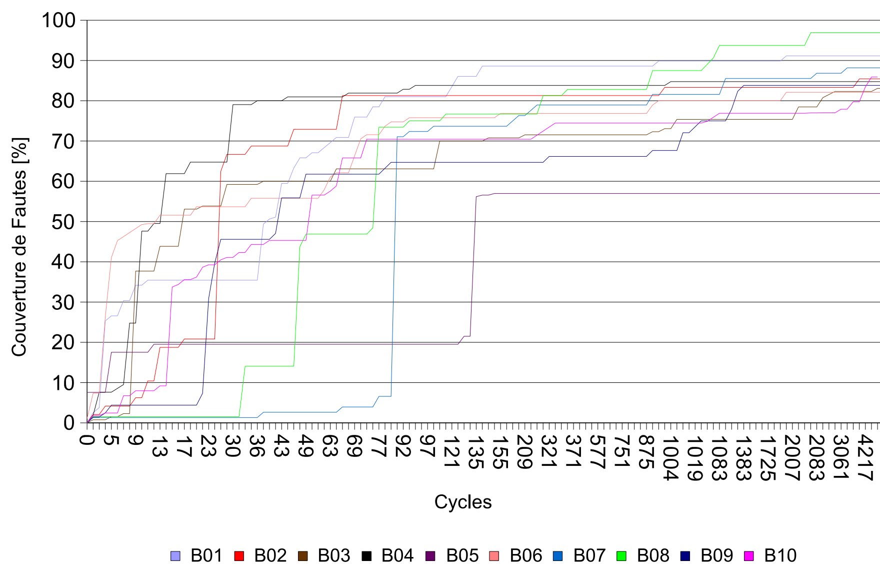


FIG. 6.2: *Évolution de CF(%) en fonction du nombre de cycles physiques VHDL.*

La figure 6.2 montre l'évolution de la couverture de fautes en fonction du nombre de cycles de simulation. D'une manière générale, la méthode concurrente permet d'atteindre une couverture de fautes proche de 60% avec seulement 70 vecteurs de test *(ou cycles)*. On peut remarquer les évolutions brutales entre les cycles 1 et 30 des couvertures de fautes qui sont dûts à la détection simultanée des fautes les plus rapidement détectables. Les courbes des benchmarks B05 et B07 se distinguent en atteignant leur seuil plus tard dans la simulation car le code VHDL de ces circuits présente des particularités faisant qu'il est difficile d'exécuter certaines instructions VHDL.

6.3.2 Modification du code VHDL

La simulation de fautes peut se révéler être un moyen d'amélioration du code VHDL [Bisgambiglia et al., 2001]. En effet compte tenu du modèle de fautes utilisé, toutes les instructions d'affectations *(conditionnelles)* pour lesquelles on ne détecte aucune faute du type F_3 *(resp. de type* F_2*)* sont inutiles et peuvent donc être supprimées du code VHDL. Ces modifications ne changent pas le comportement du circuit et permettent de :

- diminuer la taille du fichier VHDL,
- diminuer la taille de la séquence de test.

La figure 6.3 montre le gain en CPU que l'on obtient en supprimant les instructions redondantes ou inutiles de quelques descriptions simulées. Le gain de 26,99 % obtenu sur le B04 vient du fait qu'il contient beaucoup d'instructions d'affectations redondantes et deux instructions conditionnelles inutiles.

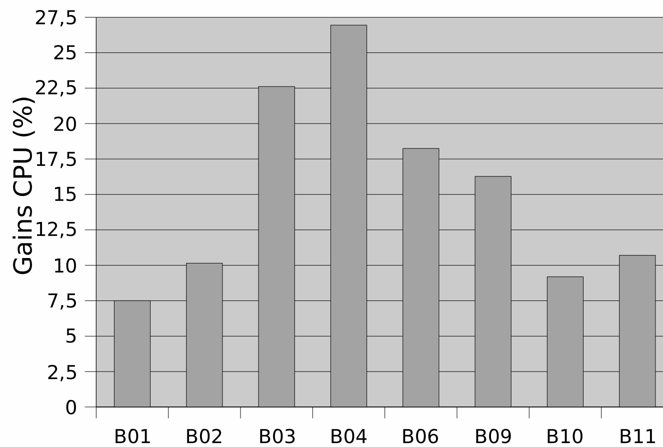


FIG. 6.3*: Gain CPU après suppression des instructions redondantes.*

6.4 Simulation BFS-DEVS des séquences de test RAGE

Les simulateurs de fautes sont principalement destinés à la validation de séquences de test par le calcul de la couverture de fautes *(cf. partie 2.1)*. Si la couverture de fautes obtenue après simulation est acceptable *(supérieure ou égale à un seuil qui dépend du type de circuit testé)*, la séquence est qualifiée de bonne et elle est retenue pour les phases de test de plus bas niveau. Nous allons à présent montrer l'efficacité du simulateur BFS-DEVS en simulant des séquences de test produites par un *ATPG (Automatique Test Pattern Generation)* de haut niveau basé sur l'algorithme *RAGE (RT-Level genetic Algorithm for test pattern GEneration)*.

6.4.1 Le générateur automatique de séquences de test RAGE

L'ATPG [Corno et al., 1997] développé au sein de l'école polytechnique de Torino s'appuie sur un algorithme génétique RAGE et sur un modèle de fautes basé sur les opérations de lecture et d'écriture *("read/write")* des signaux et variables du langage VHDL. Cet ATPG est appliqué sur un sous ensemble des benchmarks ITC'99 et les fichiers au format *.inp* contenant les séquences de test résultantes peuvent être téléchargés à l'adresse `http://www.cad.polito.it/tools/`.

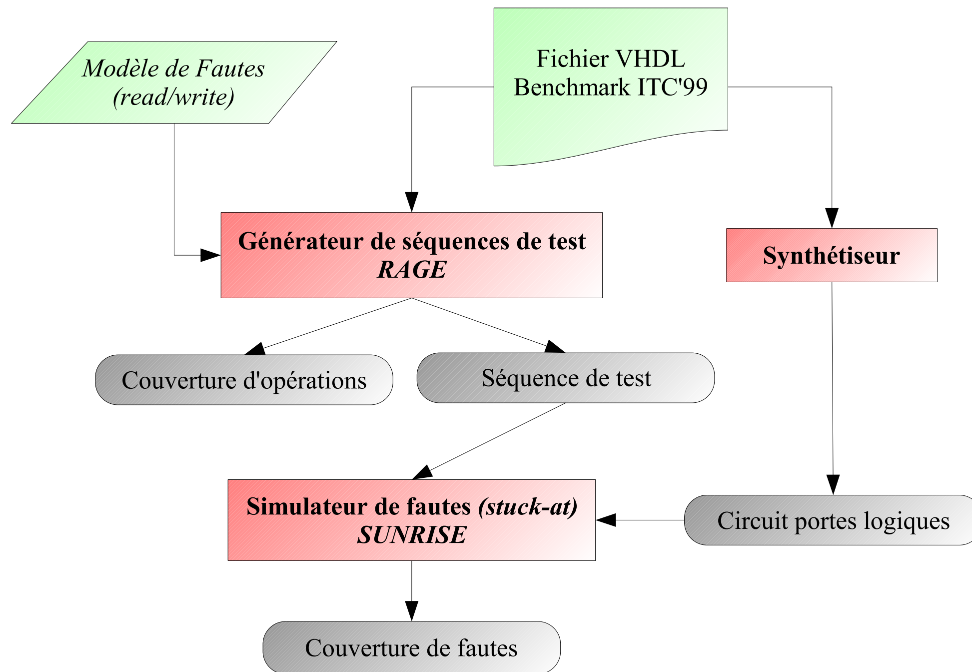


FIG. 6.4: *Architecture du prototype utilisant RAGE.*

La figure 6.4 montre l'architecture du prototype utilisant RAGE. Ce prototype est mis en place pour valider au niveau porte logique les séquences de test générées au niveau RT *(Registre Transfert)* par RAGE. Le générateur de séquences utilise l'algorithme génétique RAGE pour générer les séquences de test de chaque benchmarks ainsi que les pourcentages de Couvertures d'Opérations VHDL *CO(%)* assurées par ces séquences. Comme il est montré sur la figure 6.4, cette génération se fait à partir des fichiers VHDL de chaque benchmark et du modèle de fautes *"read/write"*. Ce modèle de fautes considère toutes les opérations de lecture et d'écriture intervenant dans toutes les opérations du code VHDL. Une faute est représentée par une mauvaise lecture ou une mauvaise écriture d'un signal ou d'une variable dans le code VHDL. Pour être validées, les séquences de test générées par RAGE sont utilisées pour faire de la simulation de fautes à partir des descriptions au niveau porte logique des benchmarks. Comme il est montré sur la figure 6.4, ces descriptions sont obtenus par synthèse des fichiers VHDL. Le simulateur SUNRISE qui utilise un modèle de fautes de type collage de valeur *(stuck-at)*, permet de calculer les pourcentages de Couvertures de Fautes *CF(%)*.

	RAGE			ATPG
benchmark	***#vect***	***CO(%)***	***CF(%)***	***CF(%)***
B01	244	95,65	94,78	100
B02	98	91,67	95,31	99,22
B03	250	75,76	71,90	72,63
B04	32	66,13	82,55	89,58
B06	120	86,44	94,15	94,15
B09	718	65,41	80,84	87,23
B10	1370	57,02	90,23	93,59
B11	1443	81,36	83,01	74,86

TAB. 6.3*: Résultats obtenus par la méthode RAGE.*

Les résultats obtenus sont résumés dans le tableau 6.3 : les pourcentages de couvertures de fautes sont reportées dans la troisième colonne *CF(%)* de la colonne principale RAGE, les nombres de vecteurs nécessaires à l'obtention de ces couvertures de fautes sont regroupés dans la première colonne *#vect* et le pourcentage de couvertures des listes des opérations sont regroupés dans la deuxième colonne *C0(%)*. D'après [Corno et al., 1997], ces résultats montrent qu'il existe une corrélation entre la couverture de fautes et la couverture de liste d'opérations. Cette corrélation justifie le fait que RAGE est un bon outil pour évaluer la testabilité des circuits.

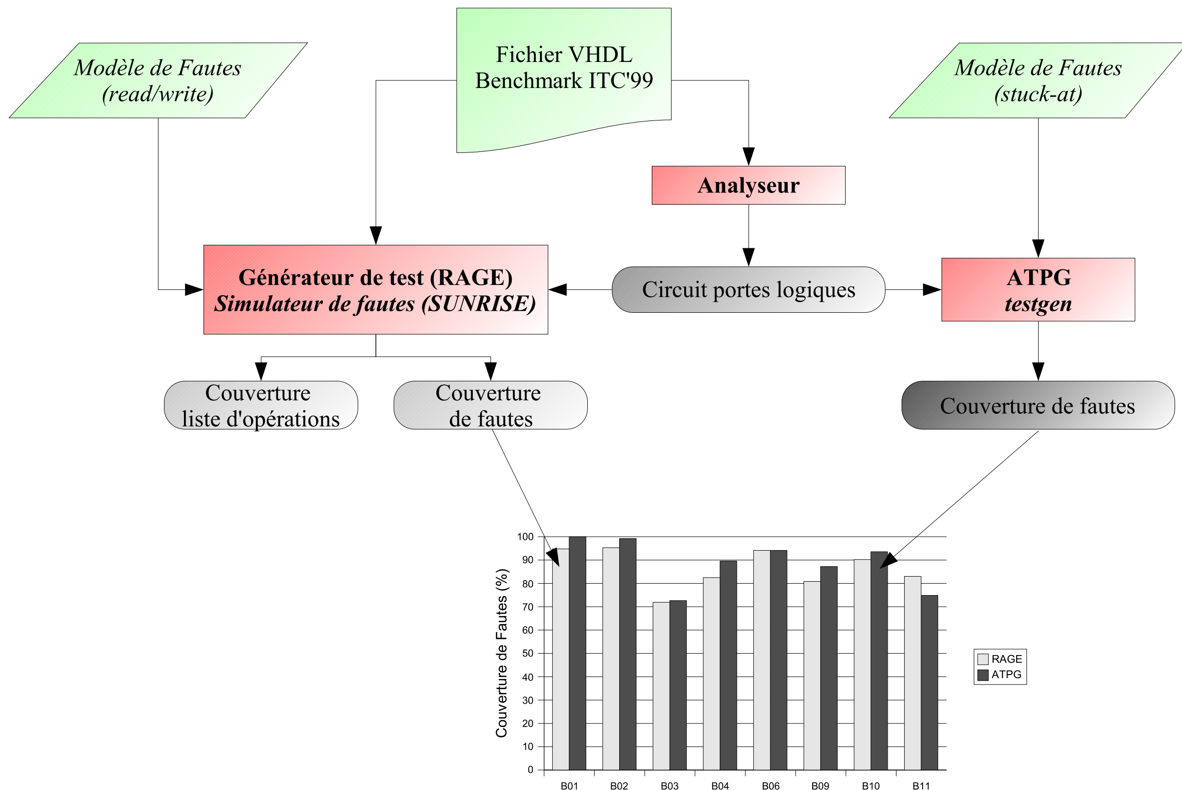


FIG. 6.5: *Comparaison de RAGE avec un ATPG commercial.*

Afin de montrer l'efficacité de la méthode RAGE, les couvertures de fautes de la troisième colonne du tableau 6.3 sont comparées aux couvertures de fautes obtenues à partir de l'ATGP *testgen (dernière colonne du tableau* 6.3*)*. Comme il est montré sur le figure 6.5, cet ATPG travaille au niveau porte logique et il utilise un modèle de fautes du type *"stuck-at"*. D'après [Corno et al., 1997], les faibles différences entre ces couvertures de fautes montrent que RAGE génère des séquences de tests présentant la même efficacité que celles générées au niveau porte logique par un ATPG classique.

6.4.2 Simulation BFS-DEVS

Nous allons à présent utiliser les séquences de test générées par RAGE en entrée de notre simulateur BFS-DEVS. Cette utilisation nous permet non seulement de comparer nos résultats mais également de montrer les avantages de la simulation concurrente BFS-DEVS. La figure 6.6

montre l'architecture mise en place pour les besoins de l'expérience *(en bleue)*. Les régions colorées en vert et en gris clair sont montrées pour une meilleure compréhension. Les séquences de test sont récupérées sous la forme de fichiers *.inp* à l'adresse `http://www.cad.polito.it/tools`. Ils sont utilisés en entrée du simulateur BFS-DEVS avec le modèle de fautes $\{F_1, F_2, F_3\}$ *(cf. partie 4.1.2)* et les fichiers VHDL des benchmarks.

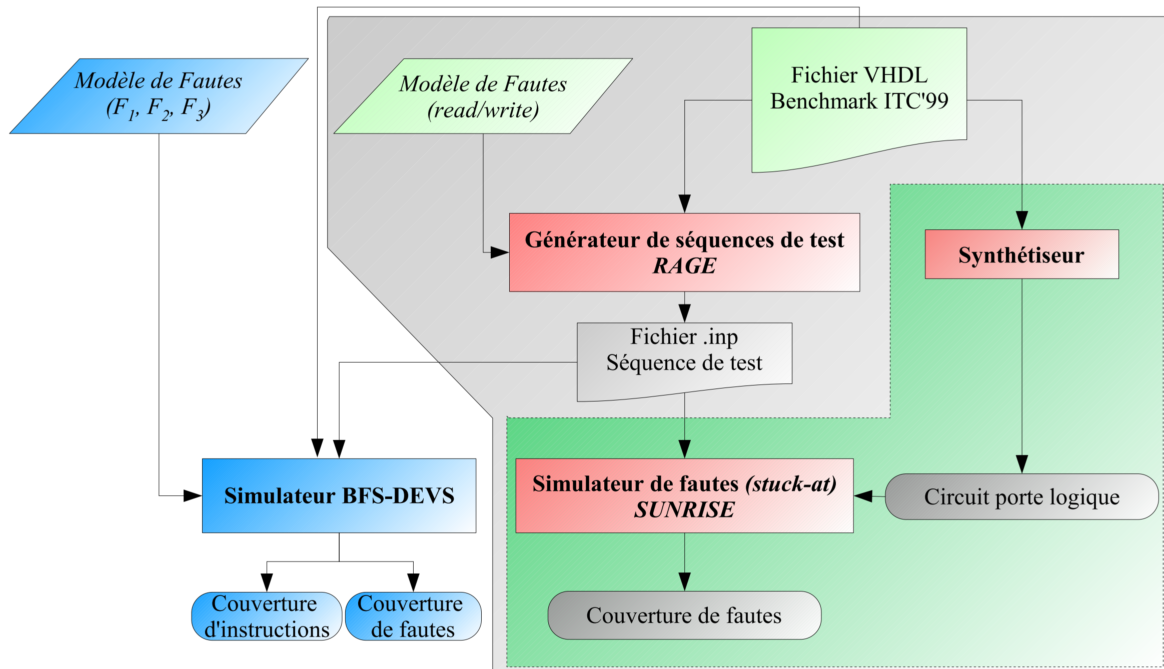


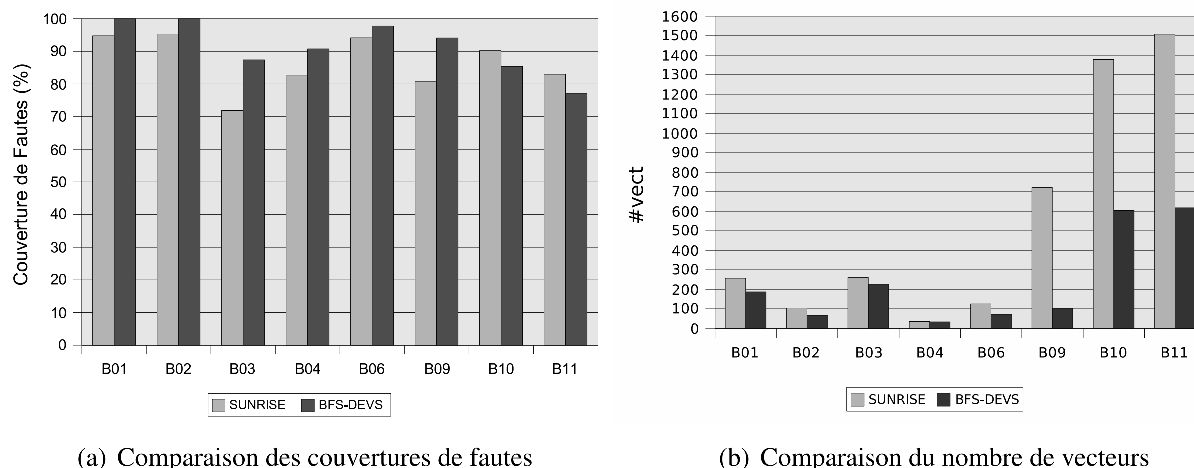
FIG. 6.6: *Utilisation des séquences de test RAGE pour le simulateur BFS-DEVS.*

Les résultats de la simulation par BFS-DEVS des séquences de test RAGE sont résumés dans le tableau 6.4.

	BFS-DEVS		
benchmark	***#vect***	***CI%***	***CF%***
B01	136	100	100
B02	60	100	93,30
B03	206	100	95,94
B04	18	96,77	90,76
B06	88	95,71	97,77
B09	684	93,75	94,11
B10	596	86,36	79,77
B11	638	73,81	78,94

TAB. 6.4: *Résultats de la simulation BFS-DEVS des séquences de test RAGE.*

Pour chaque séquence nous reportons dans la colonne *#vect* les nombres de vecteurs nécessaires à l'obtention des couvertures de fautes *CF%* et des couvertures d'instruction *CI%*.



(a) Comparaison des couvertures de fautes (b) Comparaison du nombre de vecteurs

FIG. 6.7*: Graphiques des résultats obtenus avec BFS-DEVS.*

Comme il est montré sur la figure 6.7 (a) l'algorithme concurrent implémenté dans le simulateur *BFS-DEVS permet d'atteindre des couvertures de fautes proches de celles obtenues par une simulation de fautes SUNRISE* au niveau porte logique. De plus, comme il est montré sur la figure 6.7 (b), les couvertures issues du simulateur BFS-DEVS sont obtenues avec *un nombre de vecteurs de test plus petit* que ne l'avait prévu la méthode RAGE. Cela est dû à la méthode de simulation concurrente des listes de fautes propagées sur plusieurs chemins du réseau en une seule exécution. En effet, supposons qu'une faute F aie besoin d'une séquence de test S_F composée de six vecteurs pour être détectée. Puis, supposons qu'une seconde faute F' aie besoin d'une autre séquence de test $S_{F'}$ composée de quatre vecteurs pour être détectée et que ces quatre vecteurs sont identiques aux quatre premiers vecteurs de la séquence S_F. Pour détecter les deux fautes, le simulateur SUNRISE doit simuler les deux séquences S_F et $S_{F'}$ soit : 6+4=10 vecteurs alors que le simulateur concurrent BFS-DEVS détecte les deux fautes en simulant uniquement la première séquence S_F. Ceci explique pourquoi le simulateur BFS-DEVS utilise moins de vecteurs qu'un simulateur classique qui travaille en série.

Les couvertures d'instructions *CI%* du tableau 6.4 peuvent être rapprochées des couvertures d'opérations *CO%* du tableau 6.3. Une couverture d'instructions maximale assure que la sé-

quence de test couvre le maximum du code pendant la simulation. Comme il a été dit dans [Corno et al., 1997], ces couvertures d'instructions sont fortement corrélées avec les couvertures de fautes assurant ainsi que les vecteurs de test généré par RAGE permettent une bonne analyse de la testabilité des circuits.

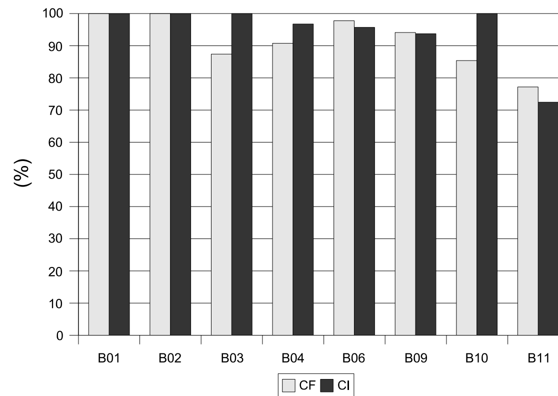


FIG. 6.8*: Corrélation CF/CI pour la simulation BFS-DEVS.*

La figure 6.8 montre la corrélation entre la couverture de fautes issue du simulateur BFS-DEVS à haut niveau avec la couverture d'instructions.

Tous les résultats montrent que les séquences de test RAGE générées au niveau comportemental peuvent être validées par notre simulateur BFS-DEVS. Il n'est donc pas nécessaire de passer au niveau porte logique pour s'assurer que les séquences de test sont de bonne qualité. La région verte sur la figure 6.6 peut donc être remplacée par notre prototype *(en bleue)*.

6.4.3 Équivalence des modèles de fautes

Les couvertures de fautes du tableau 6.4 et du tableau 6.3 sont obtenues avec deux modèles de fautes différents. Notre simulateur de fautes utilise un modèle de fautes du type $\{F_1, F_2, F_3\}$ décrit dans la partie 4.1.2, alors que RAGE considère un modèle de fautes du type *"read/write"* sur les opérations du code VHDL. Les couvertures de fautes obtenues dans les deux cas étant très proches, nous pouvons admettre à postériori que les deux modèles de fautes sont équivalents à haut niveau.

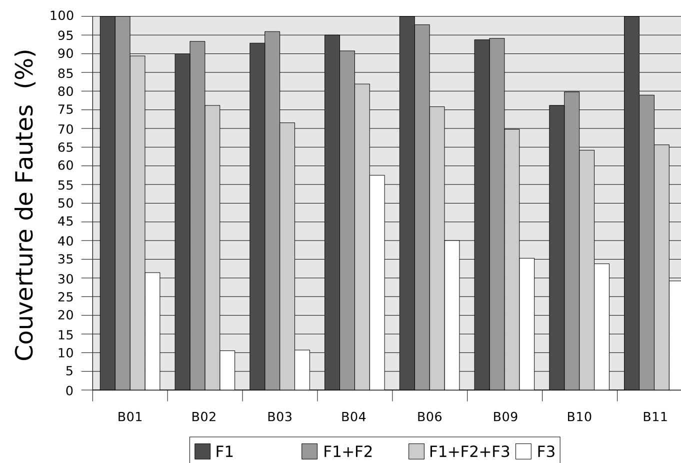


FIG. 6.9: *Couverture de fautes en fonction du modèle de fautes* $\{F_1, F_2, F_3\}$.

La figure 6.9 montre les différentes couvertures de fautes obtenues par la simulation BFS-DEVS en fonction du type de fautes simulées. Nous voyons que la simulation des fautes de type F_1 permet d'atteindre des couvertures de fautes maximales. Si l'on simule les trois types de fautes F_1, F_2 et F_3 la couverture de fautes diminue de manière significative *(-35% pour le B11)*.

Les fautes de type F_3 sont ciblées sur les instructions du code VHDL qui peuvent être difficilement testables car elles dépendent de la structure du code VHDL. Pour détecter une faute de type F_3 sur une instruction d'affectation, il faut *contrôler* et *observer* les effets de la faute sur les signaux de sortie de la description. Plus l'instruction est difficilement atteignable dans le code VHDL, plus il sera difficile de mettre en évidence une faute du type F_3 sur cette instruction. De la même manière, plus la faute du type F_3 a de l'influence sur les signaux de sorties, plus elle sera détectable en fin de simulation. Ces deux conditions dépendent fortement de la manière dont est écrit le code VHDL.

6.5 Conclusion

Ce chapitre montre la faisabilité et les avantages de la simulation BFS-DEVS sur un sous ensemble des benchmarks ITC'99. Nous avons montré que la simulation BFS-DEVS permet d'accélérer la simulation en réduisant le nombre de cycles nécessaires à l'obtention des couvertures de fautes maximales. Ceci est dû à la méthode concurrente avec propagation de listes de fautes implémentée dans le simulateur BFS-DEVS. De plus, nous avons montré que le simulateur BFS-DEVS permet la validation et l'optimisation des séquences de test de haut niveau issu d'un ATPG basé sur un algorithme génétique RAGE développé par [Corno et al., 1997].

CHAPITRE 7

Conclusion et perspectives

L'objectif de nos travaux était de proposer un simulateur de fautes concurrent appliqué au domaine du test de circuits digitaux décrits à haut niveau en s'appuyant sur un formalisme de spécification de systèmes à événements discrets. Pour cela, nous avons définit le *formalisme BFS-DEVS (Behavioral Fault Simulation for DEVS)* qui utilise les algorithmes de la *Simulation de Fautes Concurrente (SFC)* et le formalisme de *Spécification des systèmes à EVénements Discrets (DEVS*). Ce formalisme permet la modélisation et la simulation concurrente des fautes comportementales au sein des systèmes à événements discrets comme les circuits digitaux décrits en VHDL.

Les domaines d'applications du formalisme BFS-DEVS sont nombreux et permettent de dresser plusieurs perspectives de travail. Parmis cells-ci nous pouvons citer les domaines de la *"simulation des feux de forêt"* et de la *"simulation des pannes dans les réseaux électriques"*. Nous donnons également une perspective de travail sur l'amélioration de la rapidité d'exécution de la simulation BFS-DEVS.

7.1 Conclusion générale

La figure 7.1 montre l'utilisation du formalisme BFS-DEVS pour accomplir la simulation concurrente BFS-DEVS d'un système appartenant à un domaine d'application quelconque. Notre domaine d'application étant celui du test des circuits digitaux à haut niveau, le système représenté sur la figure 7.1 est le circuit digital décrit en VHDL comportemental.

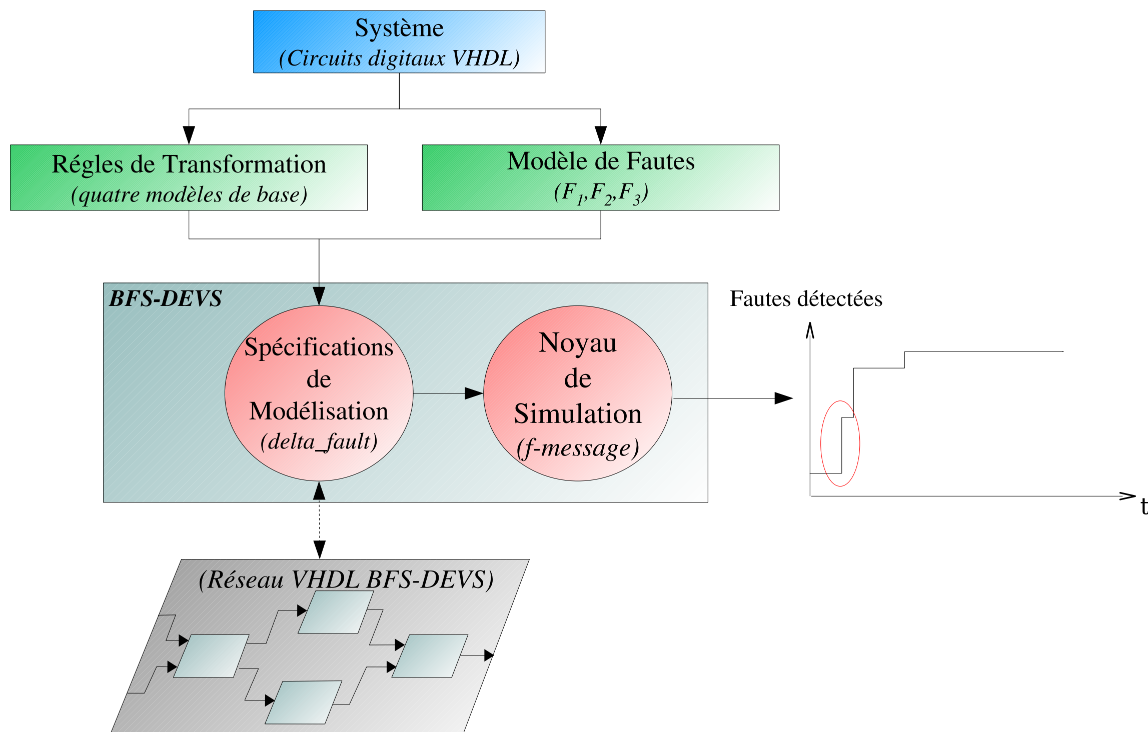


FIG. 7.1: *Simulation concurrente BFS-DEVS d'un système.*

L'utilisateur désireux de simuler des fautes comportementales au sein d'un système grâce au simulateur BFS-DEVS doit déterminer :

- **le modèle de fautes,**
- **les règles de transformation.**

Le modèle de fautes est l'ensemble des défauts comportementaux qui peuvent intervenir au sein du système. Dans le cas des circuits digitaux décrits en VHDL nous avons choisi trois types de fautes comportementales :

- La faute de type F_1 qui correspond au **collage de valeur** des signaux et des variables,
- La faute de type F_2 qui correspond au **collage d'une branche** conditionnelle,

– La faute de type F_3 qui correspond au **saut d'une instruction d'affectation**.

Les règles de transformation sont l'ensemble des **modèles BFS-DEVS de base** qui constituent **la librairie spécifique** au domaine étudié. La transformation entre le système d'origine et le modèle BFS-DEVS global *(réseau BFS-DEVS)* utilise cette librairie. Dans le cas des circuits digitaux décrits en VHDL, cette librairie nommée VHDL BFS-DEVS est composée de quatre modèles :

1. Le modèle atomique **"Assignement"** qui représente une instruction d'affectation,
2. Le modèle atomique **"Conditional"** qui représente une instruction conditionnelle,
3. Le modèle atomique **"Junction"** qui représente la fin d'une instruction conditionnelle,
4. Le modèle couplé **"Process"** qui représente une instruction "process".

Comme il est montré sur la figure 7.1, le formalisme BFS-DEVS est composé de deux noyaux : un **noyau de spécifications de modélisation** et **un noyau de simulation**. Cependant, l'utilisateur n'interagit qu'avec les spécifications de modélisation.

Après avoir défini le modèle de fautes et les règles de transformation du système, les spécifications de modélisation BFS-DEVS donnent la possibilité d'implémenter les modèles de la librairie. Le comportement fautif de ces modèles est spécifié à l'intérieur d'une **nouvelle fonction de transition fautive** nommée $delta_fault$. Dans le cas du VHDL, les quatre modèles de base sont implémentés dans la librairie VHDL BFS-DEVS et ils possèdent leur propre comportement fautif qui dépend du type F_1, F_2 ou F_3 de fautes qu'ils simulent. Un parser s'occupe de transformer la description VHDL en un réseau de modèles BFS-DEVS afin qu'elle puisse être simulée.

Le noyau de simulation est généré de manière automatique à partir des spécifications de modélisation *(ou du réseau BFS-DEVS)*. Le simulateur BFS-DEVS permet de simuler les fautes de manière concurrente grâce à l'introduction d'un nouveau type de message dans son protocole de communication : le **f-message.** Ce message permet l'exécution de la fonction de transition fautive $delta_fault$ d'un modèle atomique BFS-DEVS. De plus, l'algorithme de simulation est basé sur une **technique de propagation de liste de fautes** dans les réseaux BFS-DEVS. Ces listes de fautes qui correspondent à des simulations fautives sont conduites par **découpage** et **réorientation** au sein du réseau. Cette technique permet une meilleure gestion de la simulation et facilite également l'observabilité des résultats en fin de simulation.

La technique de propagation des listes de fautes permet d'accélérer leur détection. Le graphique obtenu en sortie du noyau de simulation sur la figure 7.1 montre l'évolution en escalier du nombre de fautes détectées en fonction du temps de simulation. Les évolutions verticales de ce nombre à unité de temps constante s'explique par **la détection simultanée de plusieurs fautes**. Dans le cas des circuits digitaux, un prototype du simulateur BFS-DEVS a été implémenté en langage Python et les simulations sur les dix premiers benchmarks ITC'99 montrent les mêmes évolutions.

7.2 Perspectives de recherche

La perspective de travail principale dans le domaine du test de circuits digitaux décrits à haut niveau est le développement d'une métrique probabiliste qui permet d'évaluer le degré de détection d'une faute. Cette méthode est basée sur les critères de contrôlabilité et d'observabilité d'un circuit et permettra de dire si une faute est facilement testable. Cette approche est une introduction au développement d'un générateur de séquences de test déterministe qui sera couplé avec le simulateur BFS-DEVS. L'idée principale est d'utiliser un graphe de dépendance entre les chemins d'une représentation VHDL pour déterminer les séquences de test de chaque faute.

Nous avons plusieurs perspectives de recherche concernant les gains en performances et les nouveaux domaines d'applications de la simulation BFS-DEVS. La simulation BFS-DEVS peut être appliquée à d'autre domaine d'application que celui du test de circuits digitaux décrits à haut niveau. L'un des projets porteurs de l'université de Corse est "La propagation des feux de forêt". Notre laboratoire de recherche possède une grande expérience dans ce domaine où le formalisme DEVS est déjà utilisé. La simulation BFS-DEVS peut être appliquée dans ce domaine pour simuler de manière concurrente plusieurs mises à feu à des endroits différents d'une parcelle donnée. Cette configuration peut résulter du phénomène de saut de flamme par exemple. Dans ce cas, la simulation concurrente est avantageuse car les interactions entre ces différentes mises à feu se font en exploitant les signatures de chaque simulation rendant ainsi l'observabilité des résultats automatique.

Le temps de simulation est primordial dans le domaine de la propagation des feux de forêt. Les simulations doivent être rapides afin de prédire à l'avance les directions dans lesquelles le feu va se propager. La simulation distribuée est une solution intéressante car elle accélère le temps de simulation en partageant l'exécution d'un programme en plusieurs processus qui peuvent être traités indépendamment sur les machines d'un réseau. La simulation BFS-DEVS peut générer un grand nombre de simulations fautives d'activités indépendantes. Ces simulations peuvent donc être associées à des processus que l'on exécuterait sur les machines d'un réseau. Cette solution est avantageuse si le nombre de simulations fautives générées est important. C'est souvent le cas dans la simulation de descriptions VHDL de plusieurs processus ou dans la simulation des propagations de plusieurs mises à feu sur une parcelle importante.

Un autre domaine d'application est celui de la simulation des pannes dans des circuits décrits au niveau analogique. Comme pour un circuit logique, un circuit analogique peut, durant son cycle de vie, être soumi à différents défauts : vieillissement, usure, etc. La modélisation et la simulation de ces circuits grâce à des outils comme SPICE ne permettent pas la simulation automatique de ces pannes. Les concepteurs modélisent ces pannes en plaçant des interrupteurs entre les éléments analogiques du modèle. Cependant, lorsque le modèle possède plus d'un millier de mailles, cette méthode devient vite fastidieuse et nécessite beaucoup de travail manuel. La simulation BFS-DEVS permet d'automatiser cette tâche et son aspect concurrent accélère la vitesse de détection des pannes.

CHAPITRE 8

Annexes

8.1 Le composant BFS-DEVS *"Generator"*

Nous savons qu'une instruction VHDL *'process'* est représentée par un modèle couplé BFS-DEVS. Par conséquent, l'activité d'un *'process'* correspond à l'arrivée d'un événement sur l'entrée du modèle couplé associé. Le *"Generator"* est un modèle atomique BFS-DEVS chargé de générer des messages d'activation destinés aux modèles couplés BFS-DEVS simulant ainsi l'activité des 'process' associés.

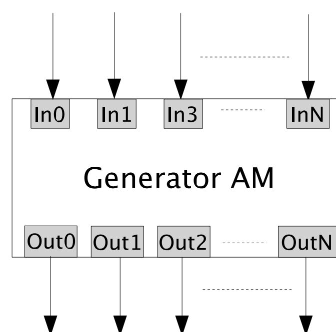


FIG. 8.1*: Modèle atomique "Generator".*

Comme il est montré sur la figure 8.1. le *"Generator"* possède le même nombre de ports d'en-

trée et de sortie N ($N \in \mathbb{R}^+$). Les ports d'entrées permettent la réception des messages parvenant des modèles couplés en fin de *cycle physique* afin de réactiver ceux-ci pour un éventuel *cycle symbolique*. Les ports de sortie permettent la transmission des messages d'activation destinés aux modèles couplés représentants les 'process' VHDL sous contrôles.

La fonction du *“Generator”* ne se résume pas seulement à la génération de messages d'entrée pour des modèles couplés. Serte il rend possible leurs activations mais comme le modèle atomique *“Conditionnal”* il dirige la simulation de fautes concurrente en orientant la propagation des listes de fautes primaire L au sein du réseau BFS-DEVS. La nature x des messages $M_x(L)$ issues du *“Generator”* dépend :

- **du type de simulation :** saine ou fautive avec propagation de listes de fautes,
- **de l'activité des 'process' représentée par les modèles couplés** : quelque soit le type de simulation, si au moins un des signaux de la liste sensitive du 'process' P à varié le modèle couplé associé doit être activé. Si par contre tout les signaux sont invariants le modèle couplé ne sera pas activé.

En considérant ces deux facteurs nous définissons les messages issus du *“Generator”* :

- **Pour une simulation saine (sans propagation de liste de fautes L=$\emptyset$) :**
  - un message $M_{safe}(\emptyset)$ si le modèle couplé doit être activé,
  - aucun message si le modèle couplé doit être inactif.
- **Pour une simulation de fautes concurrente avec propagation de listes de fautes L :**
  - un message $M_{safe}(L)$ si le modèle couplé doit être activé pour une simulation saine,
  - un message fautif $M_{faulty}(L)$ si le modèle couplé doit être activé pour une simulation des fautes présentent dans L.

Remarques :

1. Au cycle d'initialisation VHDL, tout les 'process' sont actifs. Par conséquent, le *“Generator”* enverra à tout ses modèles couplés un message du type $M_{safe}(\emptyset)$ quelque soit le type de simulation.

2. Il faut souligner que la simulation concurrente est effectuée en parallèle de la simulation saine. C'est pourquoi, le message $M_{safe}(L)$ permet d'effectuer la simulation saine servant de référence à la simulation de fautes concurrente dans le même cycle de simulation.

8.1.1 Rôle dans la simulation de fautes

Le rôle principal du *“Generator”* dans la simulation de fautes est la détermination et la distribution des listes de fautes primaires. Le modèle atomique *“Generator”* est l'élément par lequel débute la simulation de fautes concurrente. C'est lui qui détermine les premières listes de fautes propagées au sein des modèles couplés. Elles seront composées de fautes induisant un comportement différent au niveau de l'activation des *'process'* VHDL. En considérant le modèle de fautes utilisé, si on pose :

- $L_p^{C_k}$ la liste des fautes impliquant une fausse évaluation de l’activité du ’process’ P au cycle de simulation k ($k \in \mathbb{R}^+$),
- A_p facteur d’activité du ’process’ P en simulation saine *(ou d’activation de M)*. Si A_p est égale à 1 le ’process’ P doit être actif 0 s’il doit être inactif,
- $F_1^{C_k}$ les fautes du type F1 sur les signaux sensitifs au cycle k pouvant appartenir à L_O et impliquant une non activation du “process” P si celui-ci doit être activé,
- $F_{1,2,3}^{C_{k-1}}$ les fautes du type F1, F2 ou F3 du cycle précédent $k-1$ pouvant appartenir à L_O et impliquant des valeurs différentes pour les signaux sensitifs au cycle courant k,
- F2_P_F (F2_P_V) les fautes du type F2 pouvant appartenir à L_O et impliquant une non activation (resp. une activation) du “process” P si celui-ci doit être actif *(resp. inactif)*.

La liste de fautes $L_p^{C_k}$ du ’process’ P au cycle de simulation k $(k > 0)$ est donnée par la formule suivante :

$$L_p^{C_k} = A_p[\sum^{signaux sensitifs} \left(F_1^{C_k} + F_{1,2,3}^{C_{k-1}}\right) + F2_P_F] + \overline{A_p}[\sum^{signaux sensitifs} \left(F_{1,2,3}^{C_{k-1}}\right) + F2_P_V]$$

Nous allons à présent détailler la méthode de recherche des listes de fautes grâce à l’algorithme 6.

Algorithme 6 Construction des listes de fautes L_p.

```
//initialisation
evalExpr <- {}, faultsDico <- {}

Pour tout les 'process' :
 // évaluation des expressions
 Pour tout les signaux sensitifs du 'process' :
  Si le 'signal' est sensitif :
   evalExpr['process']<- {'signal'=True}
  Sinon
   evalExpr['process']<- {'signal'=False}


Pour tout les 'process' de evalExpr :
 // ajout des fautes du type F2
 Si som(or(evalExpr['process'])) :
  faultsDico['process']<- F2_'process'_F
 Sinon :
  faultsDico['process']<- F2_'process'_V

 // ajout des fautes du type F1
 Pour tout les signaux de evalExpr['process'] :
  // pour les signaux sensitifs
  Si evalExpr['process']['signal'] :
   // injection de la fautes
   evalExpr['process']['signal'] <- not evalExpr['process']['signal']
   Si som(or(evalExpr['process'])) :
    Pour toutes les valeurs de collage valF1 du 'signal' :
     faultsDico['process']<-F1_'signal'_valF1

     // ajout des fautes du cycle précédent
     Pour toutes les fautes F du 'signal' au cycle précédent :
      // si la faute implique le même valeur de collage
      Si valF1 = valF :
        faultsDico['process']<-F


   // rétablissement de la valeur
   evalExpr['process']['signal'] <- not evalExpr['process']['signal']


  //pour les signaux non sensitifs qui sont au sein d'une évaluation fausse
  Sinon Si not som(or(evalExpr['process'])) :
   Pour toutes les fautes F du 'signal' au cycle précédent :
    // si la faute implique une valeur différente du signal
    Si valF <> val_'signal' :
     faultsDico['process']<-F

retourne faultsDico
```

Les deux dictionnaires *faultsDico* et *evalExpr* permettent de stocker : les listes de fautes pour chaque modèle couplé et la sensitivité de chaque signaux appartenant à la liste sensitive de chaque 'process'. Après avoir évalué *evalExpr*, nous recherchons les fautes du type F2 pour tout les 'process' appartenant à *evalExpr*. Les fautes du type F1 sont déterminées en injectant pour chaque signaux sensitifs toutes les valeurs différentes de leurs valeurs courantes *(y compris les valeurs impliquées par les fautes du cycle précédent)* et en évaluant la véracité de l'activité des 'process' infectés. Si l'injection de la valeur fautive implique une évaluation différente de l'activité du 'process', la faute associée sera ajoutée au dictionnaire *faultsDico* si celle-ci ne s'y trouve pas déjà.

Si une faute est localement observable *(appartient à L_O)* et doit être insérée dans une de ces listes primaires, alors nous insérerons une copie de cette faute. Nous récupérons ainsi les signatures des fautes aux cycles précédant et nous les fusionnerons à chaque fin de cycle de simulation *(cf. composant "PorcessEngine")*.

Exemple

Considérons une description VHDL composée de trois *'process'* :

- $P_1(A,B)$: Le *'process'* 1 avec 2 signaux sensitifs A et B du type bit.
- $P_2(C)$: Le *'process'* 2 avec 1 signal sensitif C du type bit.
- $P_3(D)$: Le *'process'* 3 avec 1 signal sensitif D du type bit.

L'activation des 3 modèles couplés BFS-DEVS modélisant les 3 *'process'* VHDL sera assurée par un modèle atomique *"Generator" (cf. figure 8.2)*.

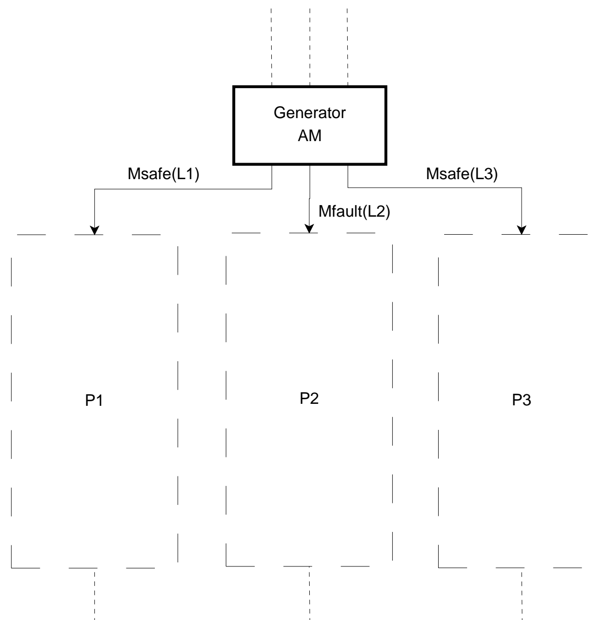


FIG. 8.2*: Modélisation BFS-DEVS d'une description VHDL à 3* 'process'.

Considérons à présent la situation résumée dans le tableau 8.1 :

	A	B	C	D
C_0	’0’	’0’	’0’	’0’
C_1	’0’	’1’	’0’	’1’

TAB. 8.1*: Valeurs des signaux A,B,C,D au cycle C_0 et C_1.*

D’après le tableau 8.1 seuls les *’porcess’* P_1 et P_3 sont actifs au cycle de simulation C_1. Par conséquent le *“Generator”* n’activera que les modèles couplés P_1 et P_3. Il enverra donc deux messages sains $M_{safe}(L_1^{C_1})$, $M_{safe}(L_3^{C_1})$ à ces modèles couplés et un message fautif $M_{fault}(L_2^{C_1})$ au modèle couplé $P2$ *(cf. figure 8.2)*. Voyons à présent comment sont construites les listes $L_1^{C_1}$, $L_2^{C_1}$, $L_3^{C_1}$.

Détermination de $L_1^{C_1}$:

$$L_1^{C_1} = A_{p_1}[F2_P1_F + \sum^{B}(F_1^{C_1} + F_{1,2,3}^{C_0})] = 1 * [F2_P1_F + F1_B_1 + F1_B_0 + \sum^{B}(F_{1,2,3}^{C_0})]$$

P_1 étant actif, $A_{p_1} = 1$. S’il existe une faute de collage sur le signal B, le ’process’ n’est pas actif *(car elle impliquerait un collage de valeur depuis le début de la simulation)*. Donc seule les fautes F1_B_0 et F1_B_1 et toutes les fautes du cycle précédant C_0 impliquant les valeurs de collage pour B sont prises en compte. Si l’on considère $\sum(F_{1,2,3}^{C_0}) = \emptyset$:

$$L_1^{C_1} = F2_P1_F + F1_B_1 + F1_B_0$$

Détermination de $L_2^{C_1}$:

$$L_2^{C_1} = \overline{A_{p_2}}[F2_P2_V + \sum^{C}(F_{1,2,3}^{C_0})] = \overline{0} * [F2_V + \sum^{B}(F_{1,2,3}^{C_0})]$$

P_2 étant inactif $A_{p_2} = 0$. Aucune faute du type F1 car P_2 n’est pas actif pour le cycle C_1. Mais si des fautes présentent au cycle précédant impliquent une valeur différente pour C alors elle seront ajoutées. Par exemple si la faute F1_A_0 implique F1_C_1 au cycle C_0 alors la faute F1_A_0 sera ajoutée à L_2 :

$$L_2^{C_1} = F2_P2_V + F1_A_0$$

Détermination de $L_3^{C_1}$:

$$L_3^{C_1} = A_{p_3}[F2_P3_F + \sum^{D}(F_1^{C_1} + F_{1,2,3}^{C_0})] = 1 * [F2_P3_F + F1_D_1 + F1_D_0 + \sum^{D}(F_{1,2,3}^{C_0})]$$

P_3 étant actif, $A_{p_3} = 1$. S'il existe une fautes de collage sur le signal D, le 'process' n'est pas actif. Donc seule les fautes F1_D_0 et F1_D_1 et toutes les fautes du cycles précédant C_0 impliquant les valeurs de collage pour D sont prises en compte. Si l'on considère $\sum F_{1,2,3}^{C_0} = \emptyset$:

$$L_3^{C_1} = F2_P3_F + F1_D_1 + F1_D_0$$

Les messages émis seront donc :
- $M_{safe}(L_1^{C_1})$ avec $L_1^{C_1} = F2_P1_F + F1_B_1 + F1_B_0$
- $M_{fault}(L_2^{C_1})$ avec $L_2^{C_1} = F2_P2_V + F1_A_0$
- $M_{safe}(L_3^{C_1})$ avec $L_3^{C_1} = F2_P3_F + F1_D_1 + F1_D_0$

8.1.2 Spécifications BFS-DEVS

Nous allons à présent donner les spécifications pseudo-code BFS-DEVS décrivant le modèle atomique *"Generator"*. Nous supposons que celui-ci possède N ports d'entrées $\{'In'_0, ..., 'In'_{N-1}\}$ et N ports de sorties $\{'Out'_0, ..., 'Out'_{N-1}\}$. Les spécifications sont les suivantes :

$$\boxed{MA_{(Generator)} = < X', S', Y', F, \delta'_{ext}, \delta'_{int}, \delta_{fault}, \lambda', t'_a >}$$

avec,

$Ports_entree = \{'In'_0, ..., 'In'_{N-1}\}$ avec $X_{In_0} = ... = X_{In_{N-1}} = V_{In} = \{M_{safe}, M_{fault}, \emptyset\}$
$Ports_sortie = \{'Out'_0, ..., 'Out'_{N-1}\}$ avec $X_{Out_0} = ... = X_{Out_{N-1}} = V_{Out} = \{M_{safe}, M_{fault}, \emptyset\}$
$X' = \{('In'_0, v), ..., ('In'_{N-1}, v) | v \in V_{In}\}$
$Y' = \{('Out'_0, v), ..., ('Out'_{N-1}, v) | v \in V_{Out}\}$
$S' = S \cup S^f = \{'ACTIVE', 'PASSIVE'\} \times \{'INIT', 'NOINIT'\} \times \mathbb{R}^+ \cup \emptyset$
$F = \{'F'_1, 'F'_2, 'F'_3\}$

$\delta'_{ext}(S', F, X)\{$
```
  S' = ('PASSIVE','NOINIT',0)
  retourne S'
```
$\}$

$\delta'_{int}(S', F)\{$

```
  Si S'=('PASSIVE','phase',σ) :
   Si sigmaTransitionList non nulle :
     σ = t_n − sigmaTransitionList[0]
   Sinon :
     σ = ∞
   S'=('ACTIVE','NOINIT',σ)
  Sinon :
   // σ = prochain temps dans sigmaTransitionList
   S'=('ACTIVE','NOINIT',sigmaTransitionList.pop(0))
  retourne S'
}
```

$\delta_{fault}(S',F,X')${retourne S'}

```
λ(S'){
  Si S' == ('PASSIVE',phase,σ) :
   Pour tout les ports de sortie 'Out' :
     // récupération du message ('In' même indice que 'Out')
     msg = peek(IPort('Out'))
     Si type(msg) == 'safe' :
       // la liste des fautes de chaque process dépend du type de message
       updateProcessFaultList(sdb,updateSensitiveSignList(sdb))
     Sinon :
       updateProcessFaultList(msg.sdb,updateSensitiveSignList(msg.sdb))

     // inclusion dans le message de la liste de fautes
     msg.faultList = ProcessFaultList[In]

     // envoi du message
     poke('Out', msg)
   Pour tout les ports de sortie 'Out' :
    Si pour le port d'entrée In correspondant v ≠ ∅ :
     Si simulation de fautes :
      //envoi de M_fault avec modification du nombre de ports de sortie
      poke('Out', M_fault(NbActivePort,processFaultList['Out']))
     Sinon :
      //envoi de M_safe sans aucune modification
      poke('Out', M_safe = peek('In'))
    Sinon si simulation de fautes :
     // envoie de M_fault avec le liste de fautes non nulle et la sdb
```

```
    poke('Out', M_fault(nbActivePort, processFaultList['Out'], sdb))
  Sinon :
   //update de la sdb pour les données d'entrées
   updateSDB()
   Si S'==(status,'NOINIT',σ) :
    //update de la liste des données sensitives
    updateSensitiveSignList()
    // update des listes de fautes
    updateProcessFaultList()
    Pour tout les ports de sortie 'Out' :
     Si 'Out' est actif :
      Si simulation de fautes :
       poke('Out', M_fault(NbActivePort, processFaultList['Out']))
      Sinon :
       poke('Out', M_safe(NbActivePort))
     Sinon si simulation de fautes :
      poke('Out', M_fault(NbActivePort, processFaultList['Out'], sdb))
   Sinon :
    Pour tout les ports de sortie 'Out' :
     // envoi des messages d'initialisation
     poke('Out', M_safe(NbActivePort))
```

$t'_a(S')$ {retourne σ}

On remarque que l'ensemble F est vide. Comme il a été souligné plus haut, le *"Generator"* ne possède pas de comportement fautif. Par conséquent, la fonction δ_{fault} retourne l'état courant sans le modifier.

La fonction de transition externe δ'_{ext} change le status de l'état en *'PASSIVE'* pour une durée de vie nulle afin de transmettre les messages reçus sur les ports d'entrées.

C'est au sein de la fonction de sortie λ' que l'on retrouve toutes les procédures de mise à jours et de gestion des listes de fautes propagées en amont.

La fonction de transition interne δ'_{int} permet le changement d'état avec des temps de vie provenant d'une liste de transition.

La liste *sigmaTransitionList* est composée de toute les durées de vie σ de chaque état prévus dans l'échéancier. Cette liste est construite *grâce* à la connaissance des valeurs de chaque signaux d'entrée.

La fonction *updateSensitiveSignList()* met à jour la liste des signaux sensitifs pour un cycle physique ou symbolique donné. Elle est construite grâce à la connaissance des pilotes de chaque signaux sensitifs dans le base de données.

La fonction *updateProcessFaultList()* utilise l'algorithme 6 et met à jour les listes des fautes qui seront propagées dans chaque 'process' sous commande. Chacune de ces listes est composée

des fautes qui seront transmises sur les ports communiquant avec les modèles couplés.

La fonction *poke()* permet l'envoi de messages sur les ports de sortie.

8.2 Le composant BFS-DEVS *“Conditional”*

Le composant *“Conditional”* est un modèle atomique BFS-DEVS chargé, d'évaluer la véracité de l'expression conditionnelle VHDL à laquelle il est attaché afin de choisir une sortie parmi ces N sorties possible *(cf. partie 8.3)*. Dans le cas d'une simulation de fautes, il est également chargé de déterminer la liste des fautes susceptibles de modifier la véracité de l'expression conditionnelle en faisant référence à la L_O. Comme le reste des composants BFS-DEVS, il a donc un rôle primordial dans le processus de propagation des listes de fautes au sein du réseau BFS-DEVS représentant la description VHDL comportementale du circuit sous simulation.

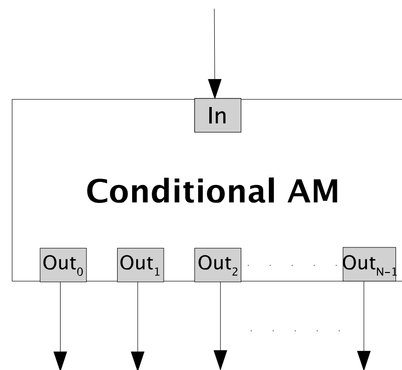


FIG. 8.3*: Le modèle atomique* “Conditional”.

Comme il est montré sur la figure 8.3. le composant *“Conditional”* possède un port d'entrée *In* et N ports de sortie $\{Out_0, ..., Out_{N-1}\}$ ou N dépend uniquement du nombre de solutions admises par l'expression conditionnelle VHDL représentée *(au minimum deux)*. Dans le cas d'une simulation de fautes, la sélection de l'unique port de sortie sain dépend de l'évaluation de l'expression conditionnelle. Les $N-1$ ports de sortie fautifs restants seront actifs uniquement s'ils transmettent un message possèdent une liste de fautes.

Le comportement du composant *“Conditional”* dépend de la nature et du contenu des messages qu'il reçoit mais aussi du type de simulation effectuée :

- **Pour une simulation saine** *(sans propagation de liste de fautes $L = \emptyset$)* :
  - le composant ne recevra qu'un seul message $M_{safe}(\emptyset)$ et sa fonction sera de transmettre celui-ci au reste des composants en aval du réseau via l'un des N ports sortie choisi. Le port de sortie sera sélectionné en fonction de la véracité de l'expression conditionnelle.
- **Pour une simulation de fautes** *(avec propagation de liste de fautes L)* :

- Si un message fautif $M_{fault}(L, fsdb)$ est reçu : la fonction du composant sera d'évaluer les conséquences des fautes appartenant à L lorsque celles-ci sont présentes au sein de l'expression conditionnelle. Il partagera la liste L en N listes contenant les fautes impliquant l'activation des N ports de sortie correspondant.
- Si un message sain $M_{safe}(L)$ est reçu : la fonction du composant sera de déterminer les $N-1$ listes contenants les fautes susceptibles, par leurs présences au sein du circuit, d'engendrer l'activation des $N-1$ ports de sortie du composant *"Conditional"* correspondants. Cette procédure s'effectue en référence à la liste L_O. Si une faute est déterminée et est déjà présente dans L_O, elle sera copiée et ajoutée à la liste à propager. Le port de sortie associé à une liste de fautes non nulle permettra la transmission d'un message de sortie fautif définissant ainsi un nouveau chemin fautif au sein du réseau BFS-DEVS.

8.2.1 Rôle dans la simulation de fautes

Le rôle du composant *"Conditional"* au sein d'une simulation de fautes dépend de la nature du message qu'il reçoit :

1. Si un message fautif $M_{fault}(L, fsdb)$ est reçu : la fonction du composant sera de répartir les fautes contenues dans L parmi les N nouvelles listes $L_{0 \leq i \leq N-1}$ destinées à être envoyées sur les N ports de sortie Out_i. Si l'évaluation de l'expression conditionnelle infectée par la faute F fourni un résultat différent de celui obtenu sans présence de F, alors la faute est ajoutée à la liste L_i. On note que :
$$\forall i \in [0, N-1] \cup L_i = L$$
2. Si un message sain $M_{safe}(L)$ est reçu : la fonction du composant sera de construire les nouvelles listes $L^{C_k}_{0 \leq i \leq N-1}$ constituées des fautes impliquant l'activation des ports fautifs Out_i au cycle de simulation C_k. Les ports de sortie fautifs étant définis comme les ports résultants d'une évaluation de l'expression conditionnelle aboutissant à un résultat différent de celui obtenu dans le cas sain. On note que la construction des listes $L^{C_k}_i$ se fait en référence à L_O. En effet la mise en évidence d'une faute au sein du réseau BFS-DEVS est unique car constante. Une faute déjà localement observable doit être réinjectée dans le réseau si elle redevient observable sans réinitialiser sa signature. Si une faute est présente dans L_O elle a été activée en amont *(ou aux cycles précédent)* du composant *"Conditional"* et apparaîtra dans les listes $L^{C_k}_i$.

1. Répartition des sous listes L_i dans le cas d'une réception d'un message $M_{fault}(L, fsdb)$:

Chaque sous liste $L_{0 \leq i \leq N-1}$ de L étant associée au port de sortie Out_i, nous utilisons un dictionnaire D possédant les couples *('clé',valeur)* :

- 'clé' : numéro i du port de sortie Out_i,

– valeur : liste L_i de fautes impliquant, par leurs présences au sein du réseau BFS-DEVS, la sélection du port de sortie ’clé’.

La procédure de détermination des listes L_i consiste à construire le dictionnaire D.

Soit la liste de fautes L transmise par le biais du message fautif $M_{fault}(L, fsdb)$, on pose :

– N le nombre de port de sortie du composant *“Conditional”*,
– F_j les fautes du type F_1, F_2 ou F_3 appartenant à $L = \{F_j | j \in \mathbb{N}\}$,
– $L_i = \{F_j | j \in \mathbb{N}\}$ la sous liste de fautes impliquant une activation du port de sortie Out_i possédant les propriétés suivantes : $\cup_{0 \leq i \leq N-1} L_i = L$,
– R le résultat de l’évaluation de l’expression conditionnelle sans présence de fautes *(simulation saine)*,
– R_{F_j} le résultat de l’évaluation de l’expression conditionnelle en présence de la par la faute F_j *(simulation de fautes)*,
– $D = \{Out_i : L_i | i \in [0, N-1]\}$ le dictionnaire associant les ports de sortie Out_i aux listes L_i des fautes les rendant actifs pour un simulation de fautes.

le dictionnaire D est donné par :

$$\boxed{D = \{Out_i : L_i | i \in [0, N-1], L_i = \{F_j | j \in \mathbb{N}, R \neq R_{F_j}\}\}} \tag{8.1}$$

Une fois le dictionnaire D construit, le composant *“Conditional”* enverra des messages fautifs sur tous les ports Out_i associés à des listes L_i non vide.

Exemple :

Nous allons donner un exemple illustrant la détermination est la répartition des listes L_i. Considérons l’expression conditionnelle ’case’ associé au modèle conditionnelle $CM0$ suivante :

$$E = case A$$
$$when 0 \Rightarrow B = \text{"00"}$$
$$when 1 \Rightarrow B = \text{"01"}$$
$$when 2 \Rightarrow B = \text{"10"}$$
$$when 3 \Rightarrow B = \text{"11"}$$

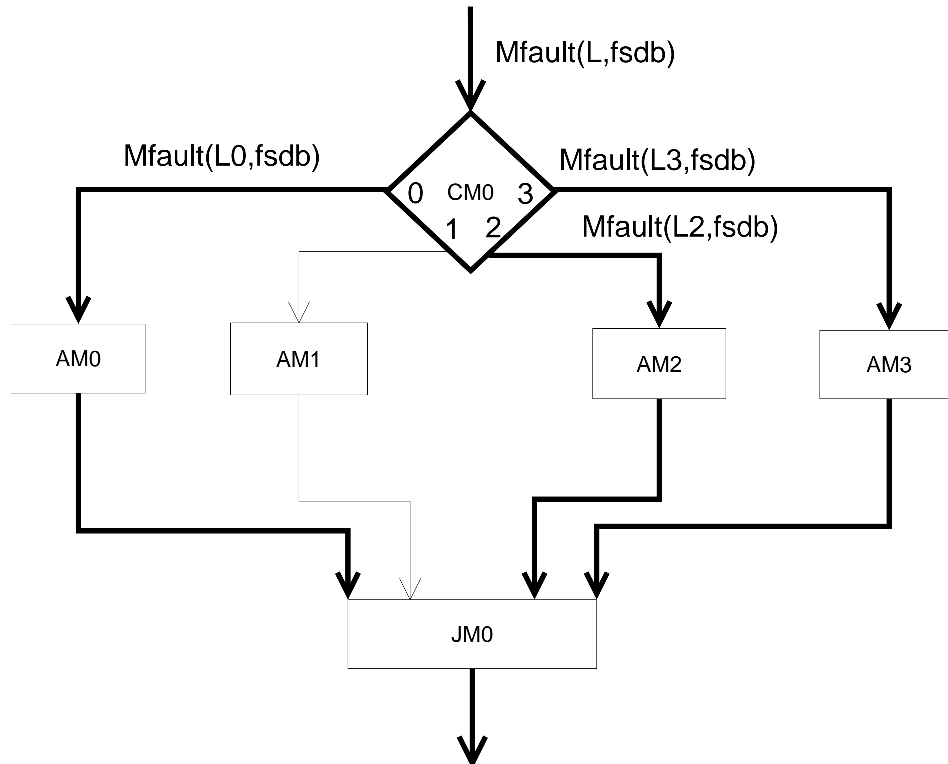


FIG. 8.4*: Représentation BFS-DEVS de l'instruction conditionnelle 'case'*.

L'expression conditionnelle E du composant $CM0$ porte sur le signal A et le résultat de son évaluation influence la valeur du signal B *(composants "Assignment" $AM0$, $AM1$,$AM2$, $AM3$).* Le signal A admettant quatre valeurs possibles, le composant *"Conditional" $CM0$* possède 4 ports de sortie fautifs $\{Out_0, Out_1, Out_2, Out_3\}$ *(cf. figure 8.4).*

Le dictionnaire D possédera donc quatre clés $\{0,1,2,3\}$ associées aux quatre valeurs $\{L_0, L_1, L_2, L_3\}$ tel que :

$$D = \{0 : L_0, 1 : L_1, 2 : L_2, 3 : L_3\}$$

Si l'on considère :

1. la liste de fautes L issue du message en entrée $M_{fault}(L, fsdb)$ de la forme $L = \{F_1(A_0), F_1(A_3), F_1(C_0), F_2CM1_1(A_0), F_2CM8_0()\}$ avec :
   - $F_1(A_0)$ resp. $F_1(A_3)$ les fautes du type F_1 *(cf. partie 4.1.2)* impliquant le collage de A à la valeur 0 resp. 3,
   - $F_1(C_0)$ la faute du type F_1 impliquant le collage d'un signal C à 0,
   - $F_2CM1_1(A_0)$ la faute de collage de le branche n°1 sur un autre composant *"Conditional" $CM1$* mise en évidence en amont du réseau BFS-DEVS impliquant la valeur 0 pour le signal A,
   - $F_2CM8_0()$ la fautes de collage de branche n° 0 sur un autre composant *"Conditional" $CM8$* mise en évidence en amont du réseau impliquant aucune valeur sur les signaux.

2. la valeur du signal A dans $fsdb$ égale à 2.

La seconde considération implique $R = 2$. En effet, comme $CM0$ se trouve sur un chemin fautif *(réception d'un message fautif)* l'évaluation de E s'effectue par rapport à $fsdb$ et implique la troisième solution ($A = 2$). On rappel que le but de la procédure est de construire le dictionnaire D. La formule 8.1 montre qu'il faut à présent déterminer les R_{F_j}.

Détermination de $R_{F_1(A_0)}$ **et de** $R_{F_2CM1_1(A_0)}$ **:** l'évaluation de l'expression 'case' en considérant A collé à la valeur 0 implique la première solution :

$$R_{F_1(A_0)} = R_{F_2CM1_1(A_0)} = 0$$

Détermination de $R_{F_2CM8_0()}$ **:** la faute n'impliquant aucune valeur, l'évaluation de l'expression 'case' se fait en considérant les valeurs de la base de données fautive $fsdb$. Donc $A = 2$ impliquant la troisième solution :

$$R_{F_2CM8_0()} = 2$$

Détermination de $R_{F_1(A_3)}$ **:** l'évaluation de l'expression 'case' en considérant A collé à la valeur 3 implique la dernière solution :

$$R_{F_1(A_3)} = 3$$

Détermination de $R_{F_1(C_0)}$ **:** le signal C n'ayant aucune influence sur l'évaluation de l'expression 'case' celle-ci se fait en considérant les valeurs de la base de données fautive $fsdb$. Donc $A = 2$ impliquant la troisième solution :

$$R_{F_1(C_0)} = 2$$

D'après la formule 8.1. nous en déduisons :

$$L_0 = \{F_1(A_0), F_2CM1_1(A_0)\}$$
$$L_1 = \emptyset$$
$$L_2 = \{F_2CM8_0(), F_1(C_0)\}$$
$$L_3 = \{F_1(A_3)\}$$

La liste de fautes L_1 étant vide, le composant *"Conditional"* $CM0$ envoie *(cf. en gras sur la figure 8.4)* trois messages fautifs $M_{fault}(3, L_0, fsdb)$, $M_{fault}(3, L_2, fsdb)$ et $M_{fault}(3, L_3, fsdb)$ en direction des modèles atomiques $AM0$, $AM2$ et $AM3$. Les messages contiennent en plus des listes de fautes L_i, le nombre total de messages envoyés et la base de données $fsdb$ reçu dans le message d'entrée.

Remarque : la propriété d'inclusion est respectée :

$$
\begin{aligned}
\cup_{0\leq i\leq N-1} L_i &= \{F_1(A_0), F_2(A_0)\} \cup \emptyset \cup \{F_2CM8_0(), F_1(C_0)\} \cup \{F_1(A_3)\} \\
&= \{F_1(A_0), F_1(A_3), F_1(C_0), F_2(A_0), F_2CM8_0()\} \\
&= L
\end{aligned}
$$

2. Détermination des sous listes $L_i^{C_k}$ dans le cas d'une réception d'un message sain $M_{safe}(L)$:

Lorsqu'un message du type sain arrive sur le composant *"Conditional"*, l'expression conditionnelle est évaluée afin de choisir le port de sortie sain Out_s. Les $N-1$ ports de sortie restants Out_i, sont définis comme fautifs et transmettrons les listes $L_i^{C_k}$ des fautes qui par leurs présences au sein du composant imposes l'activation du port Out_i au cycle de simulation C_k. Si l'expression conditionnelle est définit par :

$$E = f(S_l, V_m)$$

avec,

- S_l ($l \in \mathbb{N}$) les signaux intervenant dans E,
- V_m ($m \in \mathbb{N}$) les variables intervenant dans E,
- f une fonction booléen retournant vrai si l'expression conditionnelle est vrai, faux sinon.

On pose :

- $F_1^{C_k}(S_l, V_m)$ les fautes du type F_1 mises en évidence au cycle de simulation C_k sur des signaux S_l ou sur des variables V_m appartenant à E,
- $F_{1,2,3}^{C_{k-1}}(S_l, V_m)$ les fautes du type F_1, F_2 ou F_3 mises en évidence en au cycle de simulation C_{k-1} sur des signaux S_l ou des variables V_m et stockées dans la L_O,
- $F_2^{C_k}CM_i$ les fautes du type F_2 au cycle C_k impliquant une branche fautive i du composant CM, i.e $Out_i \neq Out_s$.

La formule permettant de déterminer la liste de fautes $L_i^{C_k}$ au cycle de simulation C_k pour le composant *"Conditional" CM* est donnée par :

$$
\boxed{L_i^{C_k} = [\sum_l F_1^{C_k}(S_l) + F_{1,2,3}^{C_{k-1}}(S_l) + \sum_m F_1^{C_k}(V_m) + \sum_{0\leq i\leq N-1}^{Out_i\neq Out_s} F_2^{C_k}CM_i]} \tag{8.2}
$$

Exemple :

Considérons la même configuration que l'exemple précédant avec le message $M_{safe}(L)$ arrivant sur l'entrée du composant $CM0$. Si de plus on considère :

- la liste de fautes $L = \{F_1(A_1)\}$,
- les fautes au cycle C_{k-1} pouvant être présentes dans L_O nulles : $\forall S_l$, $F_{1,2,3}^{C_{k-1}}(S_l) = \emptyset$,
- $A = 0$ dans la base de données saine *sdb*.

La seconde considération implique :

– $R = 0$,
– $R = 0 \Leftrightarrow Out_s = Out_0$.

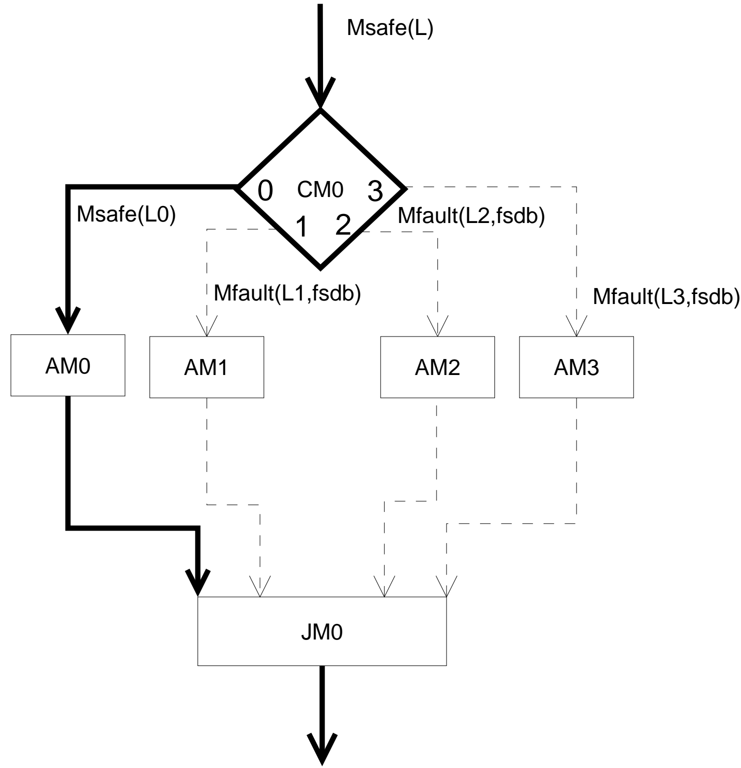


FIG. 8.5*: Représentation BFS-DEVS de l'instruction conditionnelle 'case'.*

Les fautes susceptibles de modifier le résultat R sont :
– les fautes du type F_1 impliquant un collage de A à une valeur différente de 0 : $F_1(A_1)$, $F_1(A_2)$, $F_1(A_3)$,
– les fautes du type F_2 impliquant un collage des branches différentes de la branche saine correspondant au port $0ut_s = Out_0$: F_2CM0_1, F_2CM0_2, F_2CM0_3.

Il faut à présent déterminer les listes de fautes $L_i^{C_k}$ qui seront composées des fautes précédentes à l'aide de la formule 8.2.

Détermination de L_0 : cette liste de fautes contient la totalité des fautes et sera propagée sur le chemin sain *(en gras sur la figure 8.5)*. En effet elle servira en aval de la simulation afin d'informer les composants des fautes déjà actives :

$$
\begin{aligned}
L_0^{C_0} &= [\sum_l F_1^{C_0}(S_l) + \sum_{0 \le i \le N-1}^{Out_i \neq Out_s} F_2^{C_0} CM0_i] - L \\
&= \{F_1(A_1), F_1(A_2), F_1(A_3), F_2CM0_1, F_2CM0_2, F_2CM0_3\} - \{F_1(A_1)\} \\
&= \{F_1(A_2), F_1(A_3), F_2CM0_1, F_2CM0_2, F_2CM0_3\}
\end{aligned}
$$

Détermination de L_1 : cette liste contient les fautes impliquant un chemin fautif sur le port n°1. C'est soit la faute de collage $F_1(A_1)$ imposant la valeur de A à 1, soit le collage de la branche n°1 de $CM0$ qui implique l'activation du port fautif Out_1 :

$$\begin{aligned} L_1^{C_0} &= \{F_1(A_1), F_2CM0_1\} - \{F_1(A_1)\} \\ &= \{F_2CM0_1\} \end{aligned}$$

Détermination de L_2 : cette liste contient les fautes impliquant un chemin fautif au port n°2 :

$$\begin{aligned} L_2^{C_0} &= \{F_1(A_2), F_2CM0_2\} - \{F_1(A_1)\} \\ &= \{F_1(A_2), F_2CM0_2\} \end{aligned}$$

Détermination de L_3 : cette liste contient les fautes impliquant un chemin fautif au port n°3 :

$$\begin{aligned} L_3^{C_0} &= \{F_1(A_3), F_2CM0_3\} - \{F_1(A_1)\} \\ &= \{F_1(A_3), F_2CM0_3\} \end{aligned}$$

Les messages envoyés contiendrons donc ces listes accompagnées de la base de données fautive $fsdb_i$ ainsi que du nombre de messages total envoyés :

$$\begin{array}{ll} M_{safe}(4, L_0^{C_0}) & sur\,le\,port\,sain\,Out_s \\ M_{fault}(4, L_1^{C_0}, fsdb_1) & sur\,le\,port\,fautif\,Out_1 \\ M_{fault}(4, L_2^{C_0}, fsdb_2) & sur\,le\,port\,fautif\,Out_2 \\ M_{fault}(4, L_3^{C_0}, fsdb_3) & sur\,le\,port\,fautif\,Out_3 \end{array}$$

8.2.2 Spécifications BFS-DEVS

Nous allons à présent donner les spécifications pseudo-code BFS-DEVS décrivant le modèle atomique *"Conditional"*. Celui-ci possède un port de d'entrée $'In'$ et N ports de sortie $\{'Out'_0, ..., 'Out'_{N-1}\}$. Les spécifications sont les suivantes :

$$\boxed{MA_{(Conditional)} =< X', S', Y', F, \delta'_{ext}, \delta'_{int}, \delta_{fault}, \lambda', t'_a >}$$

avec,

$Ports_entree = \{'In'\}$ avec $X_{In} = \{M_{safe}, M_{fault}, \emptyset\}$
$Ports_sortie = \{'Out'_0, ..., 'Out'_{N-1}\}$ avec $X_{Out_0} = X_{Out_{N-1}} = X_{Out} = \{M_{safe}, M_{fault}, \emptyset\}$
$X' = \{('In', v) | v \in X_{In}\}$
$Y' = \{('Out'_0, v), ..., ('Out'_{N-1}, v) | v \in X_{Out}\}$
$S' = S \cup S^f = \{'ACTIVE', 'PASSIVE'\} \times \mathbb{R}^+ \times \mathbb{R}^+ \cup \{'WRONG'\} \times \mathbb{R}^+ \times \mathbb{R}^+$
$F = \{'F'_1, 'F'_2, 'F'_3\}$

$\delta_{ext}(S', F, X')\{$

```
// acquisition du message d'entrée
 msg = getMessage()
// exprList : liste des expressions possibles
pour tout les solution de exprList.index()
 Si exprList[solution] est vrai :
   S' = ('ACTIVE',0,solution)
retourne S'}
```

$\delta'_{int}(S',F)\{$

```
// retour à l'état de repo
S' = ('PASSIVE',∞,0)
retourne S'}
```

$\delta_{fault}(S',F,X')\{$

```
Si status <> 'ACTIVE' :
 // acquisition du message d'entrée
 msg = getMessage()
 // on répartie la liste de fautes de L dans le dicoFaultList
 dicoFaultList = getRepFault(L)

Sinon :
 // on détermine la liste de faute L
 L = getFaultList(L)
S' = ('WRONG',0,solution)
retourne S'}
```

$\lambda'(S')\{$

```
Si status == 'ACTIVE' :
  // envoie du message sain
  poke(Out(solution),msg)
Sinon si status == 'WRONG' :
   // si message fautif
   Si msg == 'faulty' :
    // envoie des message fautifs
    Pour tout les liste L_i non vide de dicoFaultList :
      poke(Out_i,msg(len(dicoFaultList),L_i,fsdb_i))
   Sinon :
   // message sain
   Pour tout les liste  L_i^{C_k}  non vide de L :
    Si i <> solution :
```

```
      // envoi des messages fautif
      poke(Out_i ,msg(len(L), L_i , fsdb_i ))
    Sinon :
      // envoi du message sain
      poke(solution ,msg(len(L), L))
```

$t'_a(S')${retourne σ}

8.3 Le composant BFS-DEVS *"Assignment"*

Nous savons qu'une instruction d'affectation VHDL est représentée par un modèle atomique BFS-DEVS *(cf. partie 4.1.3)*. Le composant *"Assignment"* est un modèle atomique modélisant l'exécution d'une affectation VHDL au cours d'une simulation.

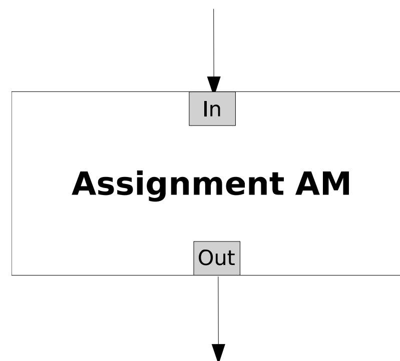


FIG. 8.6*: Modèle atomique*"Assignment".

Comme le montre la figure 8.6. le composant *"Assignment"* possède un port d'entrée *In* et un port de sortie *Out*. Le port *In* est destiné à recevoir le message d'activation dont la nature et le contenu dépendront du type de simulation effectuée. Le port *Out* permettra la transmission du message reçu au modèle atomique suivant afin de poursuivre la simulation. En effet, contrairement aux composants *"ProcessEngine"* et *"Generator"*, le modèle atomique *"Assignment"* ne participe pas directement à l'orientation des messages au sein du réseau BFS-DEVS. Par contre c'est le seul composant possédant le pouvoir de modifier les valeurs des signaux et des variables.

Le comportement du composant *"Assignment"* dépend de la nature du message qu'il reçoit elle même étant fonction du type de simulation :

- **Pour une simulation saine** ***(sans propagation de liste de fautes $L = \emptyset$) :*** si un message sain $M_{safe}(\emptyset)$ est reçu la fonction du composant sera d'évaluer l'affectation VHDL se traduisant par la mise à jour des pilotes des signaux ou des variables affectés.
- **Pour une simulation de fautes avec propagation de liste de fautes L :**

- si un message sain $M_{safe}(L)$ est reçu la fonction du composant sera de déterminer les fautes observables sur les signaux et/ou les variables présentes dans l'expression d'affectation,
- si un message fautif $M_{faulty}(L)$ est reçu la fonction du composant sera d'évaluer l'influence des fautes appartenant à L sur les signaux et/ou les variables appartenant à l'expression.

8.3.1 Rôle dans la simulation de fautes

Le rôle du composant *"Assignment"* au sein de la simulation de fautes concurrente est double. Comme il a été souligné en introduction, c'est la nature du message reçu au cours d'une simulation qui déterminera le rôle du composant :

1. Si le composant reçoit un message $M_{safe}(L)$, son rôle dans la simulation de fautes sera de **mettre à jour la liste** L_O. Cette liste est composée des fautes localement observables sur les signaux *(ou les variables)* impliquant une valeur fautive sur le signal S *(ou la variable* V*)* affecté.
2. Si le composant reçoit un message $M_{fault}(L)$, son rôle sera **d'évaluer les conséquences des fautes contenues dans** L sur l'expression d'affectation.

8.3.1.1 Détermination de la liste L_O

Considérons l'expression d'affectation E suivante :

$$E : S^{C_{k+1}} \Leftarrow f(S_i^{C_k}, V_j, O) \text{ avec } i, j, k \in \mathbb{N}$$

Ou,
- S^{C_k} le signal affecté avec sa valeur au cycle de simulation courant k,
- $S_i^{C_k}$ le signal affectant i avec sa valeur au cycle de simulation courant k,
- V_j la variable affectante j,
- O l'ensemble des opérateurs standards $\{and, or, xor...\}$ du langage VHDL,
- $f : S_i^{C_k} \times V_j \times O \mapsto S^{C_{k+1}}$ la fonction d'affectation.

Considérons à présent le modèle de fautes basé sur les trois types de fautes F_1, F_2 et F_3 définis dans paragraphe 4.1.2. Ces types de fautes sont susceptibles de modifier le résultat de l'exécution de l'instruction E :
- Fautes du type $F_1^{C_k}$ et $F_2^{C_k}$ au cycle k appliquées sur le signal affecté S, sur les signaux affectant S_i et les variables affectantes V_j.
- Faute du type $F_3^{C_k}$ sur l'instruction E.

Si on pose : $S_{F_i}^{C_{k+1}}$ la valeur du signal affecté S au cycle $k+1$ après l'exécution de l'instruction E au cycle de simulation k avec la présence de la faute F_i *(avec* $i \in \{1,2,3\}$*)*. La définition de la liste

L_O déterminée au cycle de simulation k est la suivante :

La liste L_O sera composée des fautes F_i qui par leurs présences lors de l'exécution de l'expression E au cycle k impliquent une valeur différente sur le signal affecté S au cycle $k+1$:

$$\boxed{L_O = \{F_i | S^{C_{k+1}} \neq S_{F_i}^{C_{k+1}}\}}$$

Nous allons définir de manière exhaustive la formule suivante. Pour cela nous posons :

- $F_{1,2,3}^{C_k}(S, S_i, V_j)$ la faute du type F_1, F_2 ou/et F_3 impliquant une valeur différente sur le signal affecté S, sur un des signaux S_i ou sur une des variables V_j au cycle de simulation k,
- $F_3^{C_k}(E)$ la faute du type F_3 susceptible de se produire sur E au cycle de simulation k,
- $\Psi(S)$ la sensitivité du signal affecté S. $\Psi(S) = 1$ si $S^{C_{k+1}} \neq S^{C_k}$, 0 sinon.

La liste L_O des fautes sur le signal S au cycle de simulation k est donnée par :

$$\boxed{L_O = [\sum^{collage\ de\ S} \{F_1^{C_k}(S) + F_{1,2,3}^{C_{k-1}}(S)\} + \sum^{i,j} \{F_1^{C_k}(S_i, V_j) + F_{1,2,3}^{C_{k-1}}(S_i)\}] + \Psi * F_3^{C_k}(E)}$$

Remarque :

1. Il n'existe qu'une seule liste L_O au sein de la simulation et elle est complétée par chaque modèles atomiques *“Assignment”*. Toutes les fautes localement observables et déjà présentent dans la L_O sont mises à jour au travers de leurs signatures. Sinon les nouvelles fautes sont directement ajoutées.
2. Si l'expression E présente une variable V affectée, la formule de détermination de la liste $L_V^{C_k}$ est donnée par :

$$L_O = [\sum^{collage\ de\ V} \{F_1^{C_k}(V)\} + \sum^{i,j} \{F_1^{C_k}(S_i, V_j) + F_{1,2,3}^{C_{k-1}}(S_i)\}] + \Psi * F_3^{C_k}(E)$$

8.3.1.2 Évaluation de l'influence de L

Il s'agit de simuler la totalité des fautes contenues dans la liste L parvenus au composant *“Assignment”* via le message $M_{fault}(L, fsdb)$. La mise en évidence de l'influence des fautes dépend du leurs types :

- Faute du type F_1 ou F_2 : Si la faute est du type F_1 et que sa signature possède le signal *(ou la variable)* affecté, alors on ajoute la valeur fautive de ce signal *(ou de cette variable)* à la signature de cette faute. Sinon, l'expression d'affectation est évaluée avec les valeurs

imposées par la présence de la faute et le résultat fautif est stocké dans la signature de la faute et dans la $fsdb$.
- Faute du type F_3 : Si l'instruction d'affectation rencontrée sur le chemin fautif correspond à l'instruction influencée par la faute du type F_3, l'expression d'affectation n'est pas évaluée. En effet la faute est toujours active au niveau de l'instruction. La base de données fautive et la trace de la faute conserve la valeur du signal ou de la variable affecté. Sinon, l'expression d'affectation est évaluée avec les valeurs imposées par la présence de la faute et le résultat fautif est stocké dans la signature de la faute et dans la $fsdb$.

Remarque : La base de données fautive $fsdb$ est insérée dans le message au niveau du modèle atomique *"Conditional"* en amont du chemin fautif. Le résultat de l'exécution de l'instruction dans une configuration fautive sera stocké dans cette base de données fautive. C'est au bout du chemin fautif, sur un modèle atomique "*Junction*" , que la $fsdb$ est exploité pour l'évaluation de l'observabilité des fautes contenues dans L.

8.3.2 Spécifications BFS-DEVS

Nous allons à présent donner les spécifications pseudo-code BFS-DEVS décrivant le modèle atomique *"Assignment"* :

$$\boxed{MA_{(Assignment)} = < X', S', Y', F, \delta'_{ext}, \delta'_{int}, \delta_{fault}, \lambda', t'_a >}$$

avec,

$Port_entree = \{'In'\}$ avec $X_{In} = \{M_{safe}, M_{fault}, \emptyset\}$
$Port_sortie = \{'Out'\}$ avec $X_{Out} = \{M_{safe}, M_{fault}, \emptyset\}$
$X' = \{('In', v) | v \in X_{In}\}$
$Y' = \{('Out', v) | v \in X_{Out}\}$
$S' = S \cup S^f =$
$\{'IDLE', 'BUSY'\} \times \{'ASSIGN', 'None'\} \times \mathbb{R}^+ \cup \{'WRONG'\} \times \{'F_ASSIGN'\} \times \mathbb{R}^+$
$F = \{'F_1', 'F_2', 'F_3'\}$

$\delta_{ext}(S', F, X')\{$
```
  message = peek('In')            // réception du message sur 'In'
  getExpression()                 // évaluation de l'expression
  S' = ('BUSY','ASSIGN',0)         // passage à l'état de travail
  retourne S'}
```

$\delta'_{int}(S', F)\{$
```
  S' = ('IDLE','None',∞)          // retour à l'état de repos
  retourne S'}
```

$\delta_{fault}(S',F,X')\{$
```
  Si status == 'IDLE' :
   message = peek('In')        // réception du message car pas passé dans δext
   getFaultInfluence()         // évaluation de l'influence des fautes
  Sinon :
   getFaults()           // réception de Msafe(L) et détermination de LS
  S' = ('WRONG','F_ASSIGN',0)       // passage à l'état fautif
  retourne S'}
```

$\lambda(S')\{$`poke('Out',message)`$\}$

$t'_a(S')\{$`retourne` $\sigma\}$

8.4 Le composant *"Junction"*

Nous savons que chaque fin d'instruction conditionnelle VHDL *(comme "end if", "end case", "end loop" ect)* est représentée par un modèle atomique BFS-DEVS *"Junction" (cf. partie 4.1.3)*. Il est donc lié au composant *"Conditionnal"* qui peut lui envoyer indirectement un ou plusieurs messages d'activation. A la réception de la totalité des messages, sa fonction principale est d'orienter la simulation en générant un message de sortie dont la nature résulte d'un filtrage de ces entrées. Si de plus il se trouve au sein d'une simulation de fautes il est également chargé d'évaluer l'observabilité des fautes contenues dans les listes des messages fautifs d'entrée.

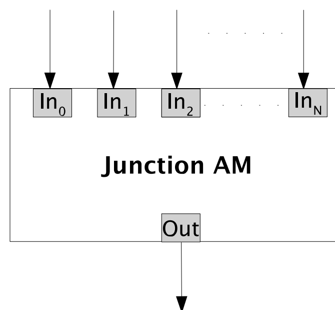


FIG. 8.7*: Modèle atomique* "Junction".

Comme il est montré sur la figure 8.7. le modèle atomique *"Junction"* possède N ports d'entrée $\{In_0,...,In_{N-1}\}$ et un seul port de sortie Out. N dépend du nombre de ports de sortie du premier composant *"Conditional"* auquel le composant *"Junction"* est associé. Autrement dit, N dépend du nombre de solutions admissent par l'expression conditionnelle VHDL associé au composant *"Conditional" (au minimum deux)*. Ces ports d'entrées sont destinés à recevoir des messages d'activation permettant également de définir le comportement que devra adopter le composant.

L’unique port *Out* servira à la transmettre le message de sortie issus du filtrage des N messages d’entrées.

Lorsque le composant est activé pour une simulation de fautes, il est chargé de déterminer l’observabilité du contenu des listes de fautes de chaque message fautif. Cette fonction ne peut donc être accompli qu’à l’arrivée de la totalité de ces messages. C’est pourquoi le composant *“Junction”* possède un compteur R_c s’incrementant à l’arrivée de chaque message d’entrée jusqu’à atteindre un seuil s fixé par le composant *“Conditional”* auquel il est lié. Afin qu’il soit connu du composant *“Junction”*, ce seuil s est transmis au sein de chaque message d’entrée. De plus si la simulation à lieu sans propagation de liste de fautes, seul un message sain sera propagé dans le réseau BFS-DEVS impliquant $s = 1$.

Le comportement du composant *“Junction”* dépend de la nature et du contenu des messages qu’il reçoit mais aussi du type de simulation effectuée :

- **Pour une simulation saine** *(sans propagation de liste de fautes $L = \emptyset$)* : le composant ne recevra qu’un seul message $M_{safe}(\emptyset, s = 1)$ du chemin emprunté par la simulation saine au sein du réseau BFS-DEVS et sa fonction sera de transmettre celui-ci au reste des composants en aval du réseau.
- **Pour une simulation de fautes** *(avec propagation de liste de fautes L)* le composant peu recevoir sur ces N ports d’entrée s messages avec $0 < s \leq N$. Si un message fautif $M_{fault}(L_i, s)$ ou sain $M_{safe}(L, s)$ est reçu le compteur R_c de messages reçu s’incrémente et deux cas sont possibles :
  - si $R_c < s$, le composant restera passif jusqu’à l’arrivée des s messages,
  - si $R_c = s$ le composant n’attend plus de messages et sa fonction dépend de la nature des s messages qu’il a reçus *(ou de la nature du chemin sur lequel il se trouve dans le réseau BFS-DEVS)*,
    - si les s messages reçus contiennent le message $M_{safe}(L, s)$ *(le composant est sur un chemin sain) :* la fonction du composant sera d’évaluer l’observabilité locale des fautes appartenant à toutes les listes $L_{0<i\leq s-1}$ des $s-1$ messages fautifs reçus avant de générer le message sain sur son unique sortie. L’observabilité est possible car nous effectuons la simulation de fautes en parallèle de la simulation saine (cf. paragraphe 3.3).
    - si les s messages reçus sont tous du type $M_{fault}(L_i, s)$ avec $0 < i \leq s$ *(le composant est sur un chemin fautif) :* la fonction du composant sera d’effectuer la fusion des s messages.

8.4.1 Rôle dans la simulation de fautes

Le rôle du composant *“Junction”* au sein de la simulation de fautes avec propagation de liste de fautes est :

- si les s messages reçus contiennent le message sain $M_{safe}(L, s)$: d’évaluer l’observabilité des fautes contenues dans toutes les listes $L_{0\leq i\leq s-1}$ des $s-1$ messages fautifs reçus. Chaque

liste de fautes reçus est stockée dans un dictionnaire D contenant N élément du type *('clé', 'valeur')* ou :
  - la *'clé'* correspond au numéro du port d'entrée,
  - la *'valeur'* correspond à la liste des fautes reçue sur le port 'clé'.
- si les s messages reçus sont tous du type fautifs : de transmettre un message fautif sur le port de sortie.

Détermination de l'observabilité des fautes :

La détermination de l'observabilité des fautes passe par la construction du dictionnaire D. Si par exemple le composant possède $N = 4$ ports d'entrée et s'il attend $s = 3$ messages :
- un message sain $M_{safe}(L_1, s = 3, \emptyset)$ sur le port n°1 avec $L_1 = L_2 \cup L_3 \cup L_4$,
- deux messages fautifs : $M_{fault}(L_2, s = 3, fsdb2)$ sur le port n°2 et $M_{fault}(L_3, s = 3, fsdb3)$ sur le port n° 3,
- aucun message sur le port n°4 car aucune faute n'a besoin d'être simulée sur le chemin fautif n°4 $L_4 = \emptyset$.

le dictionnaire D est construit lorsque tous les messages sont reçus $(R_c = s)$ et sera $D = \{(1 : L_1), (2 : L_2), (3 : L_3), (4 : \emptyset)\}$.

Le dernier champs des messages fautifs reçus correspond à la base de données fautive $fsdb_i$. C'est à dire aux bases de données générées par le passage sur un chemin fautif au sein du réseau BFS-DEVS. En effet, si lors de la propagation des listes de fautes sur un chemin fautif du réseau un composant *"Assignment"* est rencontré avant d'atteindre le composant *"Junction"*, le résultat de l'affectation sera stocké dans la base de données fautive $fsdb_i$.

Après avoir rempli le dictionnaire D, il faut procéder à l'évaluation de l'observabilité des fautes contenues dans chaque listes. C'est à dire analyser l'observabilité du contenu des *'valeur'* de chaque éléments de D. Si la faute est observable, elle est ajoutée à la liste L_O stockée dans la base de données symbolique.

Si on pose :
- sdb la base de données symbolique connue de tous les modèles atomiques BFS-DEVS,
- $fsdb_j$ avec $j \in [0, N-1]$ la base de données fautive contenu dans le message $M_{fault}(L_j, s, fsdb_j)$,
- L_j avec $j \in [0, N-1]$ la liste de fautes construite par le composant *"Conditional"* et contenu dans le message $M_{fault}(L_j, s, fsdb_j)$,
- $D = \{('In'_j, L_j) | j \in [0, N-1]\}$ le dictionnaire des listes de fautes L_j,
- F_i $(i \in \mathbb{N})$ la faute appartenant au modèle de fautes utilisé (cf. paragraphe 3.2) et issue de la liste de fautes L_j,
- $T_{F_i} = \{((S|V)_j, P^{F_i}_{(S|V)_j}) | i, j \in \mathbb{N}\}$ la signature de la faute F_i avec $P^{F_i}_{(S|V_j)}$ le pilote de valeur fautive du signal S_j ou de la variable V_j,

l'observabilité des fautes F_i est définit par :

$$\forall fsdb_j \in M_{fault},$$
$$\forall L_j \in D, \quad \boxed{sdb \neq fsdb_j \, et \, T_{F_i} = \emptyset \Rightarrow F_i \, observable \forall F_i \in L_j} \quad ou$$
$$\forall F_i \in L_j, \forall P^{F_i}_{(S|V)j} \in T_{F_i}, \quad \boxed{P^{F_i}_{(S|V)_j} \neq P_{(S|V)j} \Rightarrow F_i \, localement \, observable \, sur \, S_j \, (ou \, V_j)}$$

Dans le cas ou les signatures des fautes sont vides, c'est en identifiant les signaux dont la valeur diffère dans les deux bases de données *sdb* et $fsdb_j$ que l'on est capable d'établir l'observabilité locale des fautes F_i sur les signaux ayant évolués. Si la signature des fautes contiennent des signaux *(ou des variables)* fautifs, c'est en faisant la différences des valeurs fautives *(dans les pilotes fautifs de la signature)* et des valeurs saines *(dans les pilotes saine de la sdb)* que l'on donne l'observabilité locale des fautes.

Toutes les fautes localement observables sont ajoutées à la liste générale L_O. Si des fautes sont déjà présentent dans cette liste, leurs signatures seront mises à jour.

Exemple

Considérons le réseau de composant BFS-DEVS de la figure 8.8 modélisant l'instruction de contrôle VHDL *"if then else end if"* dans le cas d'une simulation avec propagation de listes de fautes :

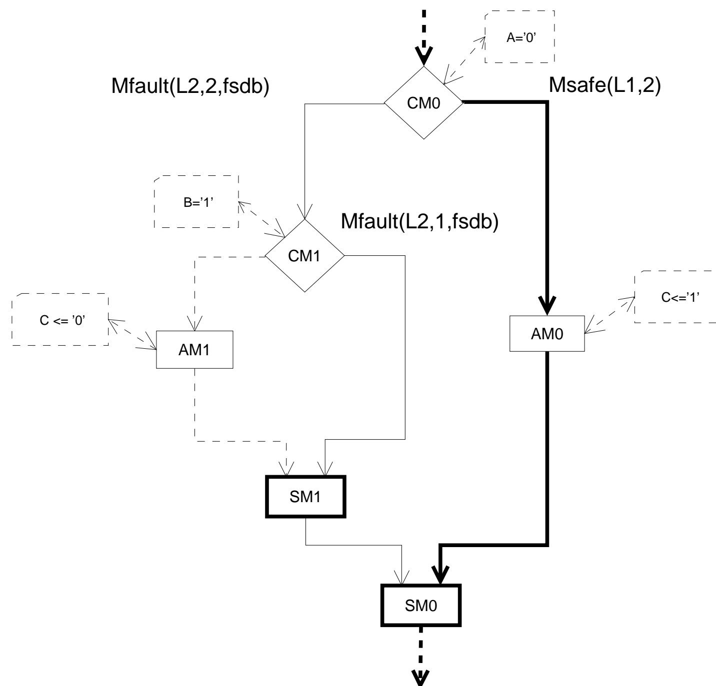


FIG. 8.8*: Représentation BFS-DEVS de l'instruction de contrôle VHDL "if then else endif".*

Si l’on suppose : $A =' 1'$, $B =' 0'$, $C =' 0'$ et CM0 activé par un message sain $M_{safe}(\emptyset)$, le chemin sain représenté en gras sur la figure 8.8 *(fautif en lisse sur la figure 8.8)* sera représenté par la branche droite *(resp. gauche)* du composant CM0. Les deux messages $M_{safe}(L1,2)$ et $M_{fault}(L2,2,fsdb)$ contenant les listes de fautes L1 et L2 sont construites et propagées par le composant CM0. Le composant CM1 reçoit et redirige sur sa branche fautive droite *(car $B =' 0'$)* le message fautif $M_{fault}(L2,1,fsdb)$ avec un seuil $s = 1$. En effet, le composant CM1 ne génère qu’un seul message en direction de JM1.

Concernant les modèles *“Junction”* JM0 et JM1, deux cas de figure sont visibles sur l’exemple ci-dessus :

1. Le composant JM1 ne reçoit qu’un seul message du type fautif. Lorsque son compteur R_c aura atteint le seuil $s = 1$ transmis par le biais du message fautif $M_{fault}(L2,1,fsdb)$ il redirigera le message modifié $M_{fault}(L2,2,fsdb)$ sur sa sortie.
2. Le composant JM0 reçoit deux messages dont le message sain $M_{safe}(L1,2)$. Lorsque son contenu R_c aura atteint le seuil $s = 2$ il effectuera l’observabilité des fautes contenus dans la liste $L2$ transmis par le biais du message $M_{fault}(L2,2,fsdb)$.

   Dans un cas simple ou le pilote des fautes précédentes du signal A est vide on a :

$$L1 = L2 = [F1_A_0, F_CM0_V]$$

   Lorsque le composant JM0 a reçu les deux messages $M_{safe}(L1,2)$ et $M_{fault}(L2,2,fsdb)$:

- il filtre ceux-ci pour générer le message $M_{safe}(\emptyset)$ choisi par rapport au type de simulation,
- il évalue l’observabilité des fautes F1_A_0 et F2_CM0_V de la liste L2.
  - **Observabilté de la faute F1_A_0 :** Cette faute implique un passage sur la branche gauche *(resp. droite)* de CM0 *(resp. de CM1)* et implique aucun changement de valeur sur le signal C. Par conséquent $C =' 0'$ dans la base de données fautive $fsdb$. La simulation saine sur la branche droite de CM0 implique $C =' 1'$ et cette valeur est stockée dans la base de données saine sdb. Les deux bases de données étant différentes pour le signal C la fautes F1_A_0 est observable sur C au niveau du composant JM0.
  - **Observabilté de la faute F2_CM0_V :** Cette faute implique aussi un passage sur le chemin fautif en lisse sur la figure 8.8. et le scénario est identique à celui rencontré pour la faute F1_A_0. Par conséquent la faute F2_CM0_F est observable sur C au niveau du composant JM0.

  Si le signal C est un signal de sortie, les deux fautes précédentes seront détectables en fin de simulation.

Remarque :

Si l’on conserve les mêmes hypothèses de départ sur les signaux A et C mais $B =' 1'$. Le chemin fautif est définit sur les deux branches gauches des composants CM0 et CM1. Cette fois la simulation des deux fautes de la liste L2 implique $C =' 0'$ dans les signatures des deux fautes.

Nous revenons donc à un cas similaire au précèdent puisque C était déjà égale à '0'. La simulation saine est toujours représentée par le passage sur la branche droite du composant CM0 et implique $C='1'$ dans *sdb*. Les valeurs fautives de C stockées dans les pilotes fautifs des signatures étant différentes des valeurs saines stockées dans les pilotes de la sdb, les fautes F1_A_0 et F2_CM0_V sont observables sur C au niveau du composant JM0.

8.4.2 Spécifications BFS-DEVS

Nous allons à présent donner les spécifications pseudo-code BFS-DEVS décrivant le modèle atomique *"Junction"* :

$$\boxed{MA_{(Junction)} = < X', S', Y', F, \delta'_{ext}, \delta'_{int}, \delta'_{fault}, \lambda', t'_a >}$$

avec,

$Ports_entree = \{'In'_0, ..., 'In'_{N-1}\}$ avec $X_{In_0} = ... = X_{In_{N-1}} = X_{In} = \{M_{safe}, M_{fault}, \emptyset\}$
$Ports_sortie = \{'Out'\}$ avec $X_{Out} = \{M_{safe}, M_{fault}, \emptyset\}$
$X' = \{('In'_0, v), ..., ('In_{N-1}, v')| v \in X_{In}\}$
$Y' = \{('Out', v)| v \in X_{Out}\}$
$S' = S \cup S^f = \{'PASSIVE', 'ACTIVE'\} \times \Re \times \mathbb{R}^+ \cup \{'WRONG'\} \times \Re \times \mathbb{R}^+$
$F = \{'F'_1, 'F'_2, 'F'_3\}$

$\delta'_{ext}(S', F, X')\{$
```
  // acquisition du message d'entrée
  msgList+=getMessage()
  // passage à l'état actif
  S' = ('ACTIVE',Rc,0)
  retourne S'}
```

$\delta'_{int}(S', F)\{$
```
  // retour à l'état de repos
  S' = ('PASSIVE',RC = 0,∞)
  retourne S'}
```

$\delta_{fault}(S', F, X')\{$
```
  Si status <> 'ACTIVE' :
   // acquisition du message d'entrée
   msgList+=getMessage()

  // incrémentassions du compteur de messages fautifs
```

R_c++

```
Si Rc == s :
 Si containSafeMessage(msgList) :
  //calcul de l'observabilité des fautes
  getFaultObservability(msgList)

 // passage à l'état fautif
 S' = ('WRONG',Rc,0)
Sinon :
 // passage à l'état passif dans l'attente du message suivant
 S' = ('PASSIVE',Rc,∞)
retourne S'}
```

$\lambda'(S')\{$

```
Si status == 'WRONG' :
 //si le message contient un message sain
 Si containSafeMessage(msgList) :
  // envoie du message sain
  poke('Out',getSafeMessage(msgList))
 Sinon :
  // envoie du message fautif
  poke('Out',getFaultMessage(msgList))
Sinon :
  // envoie du message sain
  poke('Out',getSafeMessage(msgList))
```

$t'_a(S')\{$retourne $\sigma\}$

8.5 Le composant BFS-DEVS *“ProcessEngine”*

Nous savons que chaque processus VHDL est modélisé par un modèle couplé BFS-DEVS *(cf. paragraphe 4.1.3)*. Le modèle atomique BFS-DEVS *“ProcessEngine”* est chargé de synchroniser l’exécution de ces modèles couplés BFS-DEVS. Il est également responsable de la mise à jour de la base de données et du calcul de l’observabilité des fautes actives à chaque fin de cycle. Si l’on fait le parallèle avec le domaine de description VHDL, le *“ProcessEngine”* synchronise l’exécution des instructions *’process’* et fait la mise à jour des pilotes de chaque signaux et de chaque variables appartenant à la description.

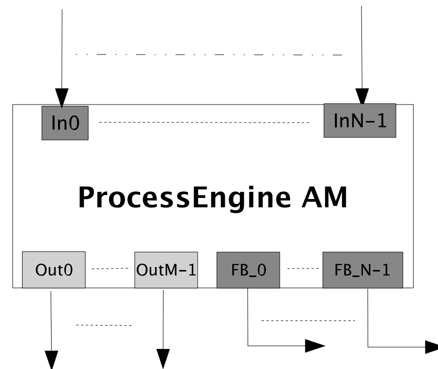


FIG. 8.9*: Modèle atomique* "ProcessEngine"

Comme il est montré sur la figure 8.9. le *"ProcessEngine"* peut admettre N $(N \in \mathbb{N}^+)$ entrées/sorties. Ces sorties sont de deux types :

- M $(M \in \mathbb{N})$ **sorties** *Out* : en direction des sorties primaires du modèle couplé global pour observer les signaux de sortie du circuit sous simulation,
- N **sorties** *FeedBack* : en direction des entrées du *"Generator"* qui peut éventuellement réactiver les modèles couplés pour des cycles symboliques.

Ces N entrées sont connectés aux sorties des modèles couplés qu'il commande. En effet, le composant *"ProcessEngine"* est responsable de leur synchronisation. C'est à dire qu'il ne sera actif que lorsqu'il aura reçu tous les messages provenant des modèles couplés en amont. Il pourra alors faire **la mise à jour** :

- **des pilotes de valeurs** de chaque signaux et variables :
  - de la base de données fautive contenu dans le message d'arrivée,
  - dans la base de donnée saine de référence,
- **de la liste des fautes localement observables** L_O en effectuant l'observabilité des fautes appartenant aux listes provenant des messages envoyés par les modèles couplés,
- dans le cas d'une simulation de fautes en fin de cycle physique :
  - **de la liste des fautes détectées** L_D en déplaçant les fautes localement observables sur les signaux de sortie de la liste L_O vers la liste L_D,
  - **de la couverture de fautes** calculée en faisant le ratio de la liste de fautes détectées L_D par la liste des fautes détectables L_T au sein du réseau BFS-DEVS.

Suite à cette phase de mise à jour, il pourra alors générer :

- les M messages sur ces ports de sortie *Out* si tous les signaux sensitifs du ou des processus simulé(s) sont stables *(fin d'un cycle physique de simulation)*,
- les $\{1 \dots N\}$ messages sur les sorties *FeedBack* connectés, par l'intermédiaire du "Generator", aux modèles couplés dans le cas ou au moins un de leurs signaux sensitifs à évolué

(début d’un nouveau cycle symbolique) ou si une faute de L_O implique l’évolution d’un de ces signaux sensitifs.

8.5.1 Rôle dans la simulation de fautes

Bien que tout comportement fautif du composant *“ProssesEngine”* semble inutile à la simulation de fautes, son rôle au sein de celle-ci est double :

- **l’évaluation de l’observabilité des fautes** contenues dans les messages d’entrées,
- **redirection des messages fautifs** sur les sorties *FeedBack* en direction du composant *“Generator”* afin qu’il active les modèles couplés concernés.

8.5.1.1 Redirection des messages fautifs : propagation inter-processus

Nous allons à présent détailler la procédure employée par le composant *“ProcessEngine”* permettant la redirection des messages de sortie vers le composant *“Generator”* lorsque des cycles symboliques ont besoins d’être simulés. Bien entendu cette procédure n’est pas propre à la simulation de fautes puisqu’elle est également effectuée pour une simulation de la description totalement saine. Cependant, lorsque plusieurs messages de différent type parviennent jusqu’au modèle *“ProcessEngine”*, la répartition des messages du type fautifs constitue une phase importante dans la simulation de fautes.

L’exemple sur lequel nous nous appuyons est identique au précédant. La figure 8.10 illustre la configuration des 3 processus au cours :

- du début d’un cycle physique pour une simulation de fautes *(sur la figure 8.10 (a))*,
- de la fin du cycle physique sain et du début d’une simulation symbolique fautive *(sur la figure 8.10 (b))*,
- de la fin de la simulation symbolique fautive *(sur la figure 8.10 (c))*.

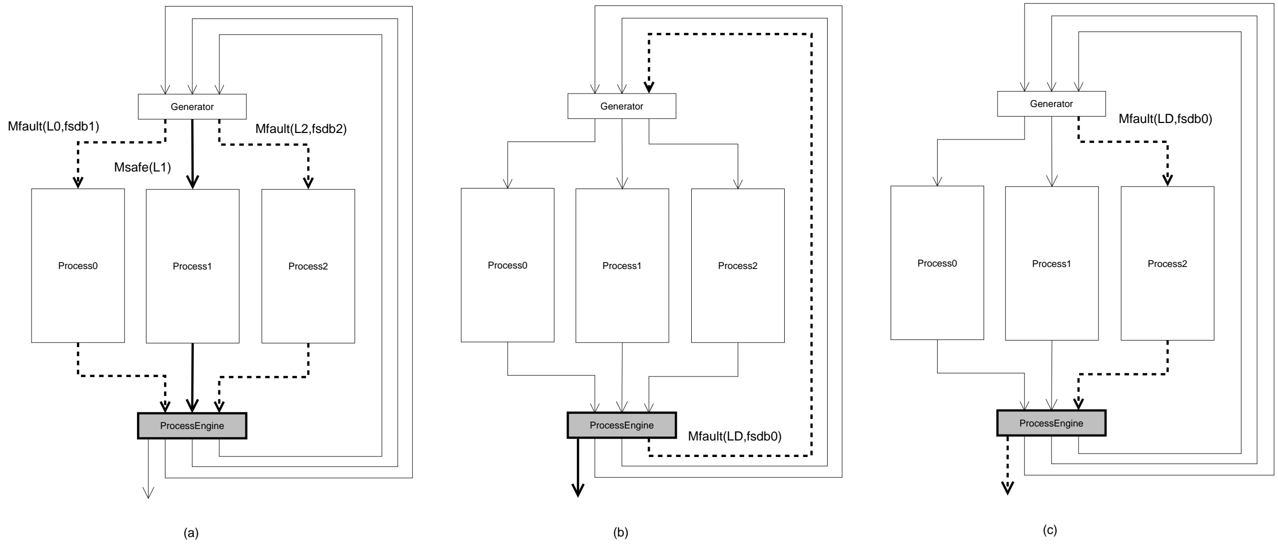


FIG. 8.10*: Simulation de fautes sur une description à 3 processus*

Lorsque le cycle physique se termine, c'est à dire lorsque le composant *"ProcessEngine"* possède tous les messages $M_{safe}(L)$, $M_{fault}(L_0, fsdb_0)$ et $M_{fault}(L_2, fsdb_2)$, la simulation saine du composant *"Process1"* n'implique aucun changement de valeur sur les signaux sensitifs A,B du processus 0 et D du processus 2. Par conséquent aucun des composants *"Process0"* et *"Process2"* n'a besoin d'être réactivé pour un cycle symbolique sain. Donc l'unique sortie *Out* correspondant au signal de sortie S est activé et le cycle physique sain se termine *(en gras sur la figure 8.10 (b))*.

Cependant, la simulation des fautes appartenant à la liste L_0 au sein du composant *"Process0"* implique la variation du signal sensitif D du composant *"Process2"*. Par conséquent celui-ci doit être activé pour un cycle symbolique fautif *(en pointillé sur la figure 8.10 (b))*. Le type de message généré sur la sortie FB_2 est fautif et la liste de fautes *LD* contient toutes les fautes initialement dans L_O appartenant au signal sensitif D qui a varié au sein du process *"Process0" (hypothèse de faute unique)*.

Le composant *"Generator"* reçoit le message fautif $M_{fault}(L_D, fsdb_0)$ et le transmet au composant *"Process2"* pour une simulation des fautes de la liste L_D *(en pointillé sur la figure 8.10 (c))*. Lorsque la simulation symbolique fautive est terminée, le composant *"ProcessEngine"* déterminera l'observabilité des fautes en analysant la base de données fautive $fsdb_0$ ainsi que la signature des fautes de L_D.

Le cas ou un processus est activé pour un cycle symbolique sain et fautif n'est pas montré ici. Dans ce cas précis, seule la simulation saine est effectuée car les fautes qui auraient du être simulées pendant la simulation fautive du processus sont simulées pendant la simulation saine de celui-ci.

8.5.1.2 L’observabilité des fautes

La procédure d’observabilité des fautes appartenant à un message dépend du type de cycle de simulation et de l’inter-processus :

- Si le cycle de simulation est physique : l’observabilité des fautes s’effectue par comparaison des pilotes des signatures avec les pilotes de la base de données saine.
- Si le cycle de simulation est symbolique :
  - Si un processus à besoin d’être activé par x messages $(x > 1)$ nous sommes dans le cas de l’inter-processus :
    - si parmi ces x messages réside le message sain M_{safe} alors la procédure d’observabilité des fautes des x-1 messages fautifs reposent sur la comparaison de leurs bases de données $fsdb_i$ avec la base de données saine sdb et de l’analyse des signatures des fautes.
    - si les x messages sont tous du type fautifs alors la procédure d’observabilité des fautes s’effectue par comparaison des pilotes de valeur des bases de données $fsdb_i$ et de l’analyse des signatures des fautes.
  - Si le processus à besoin d’être simulée une seule fois, alors il n’y a pas de propagation inter-processus et la procédure d’observabilité des fautes repose sur la différence des pilotes de valeurs de la base de données concernés.

Observabilité des fautes sans inter-processus :

Lorsque $R_c = NbActProc$ le composant évalue l’observabilité des fautes contenues dans toutes les listes $L_{0 \leq i \leq N-1}$ non vides des N messages reçus. Chaque liste de fautes reçus est stockée dans un dictionnaire D contenant N élément du type *(’clé’, ’valeur’)* ou :

- la *’clé’* correspond au numéro du port d’entrée,
- la *’valeur’* correspond à la liste des fautes reçue sur le port *’clé’*.

La détermination de l’observabilité des fautes passe par la construction du dictionnaire D. Si par exemple le composant possède $N = 4$ ports d’entrée et s’il attend $s = 3$ messages :

- un message sain $M_{safe}(L_1, s = 3, \emptyset)$ sur le port n°1,
- deux messages fautifs : $M_{fault}(L_2, s = 3, fsdb2)$ sur le port n°2 et $M_{fault}(L_3, s = 3, fsdb3)$ sur le port n° 3,
- aucun message sur le port n°4 car aucune faute n’a besoin d’être simulée sur le processus n°4 $L_4 = \emptyset$.

le dictionnaire D est construit lorsque tous les messages sont reçus $(R_c = s)$ et sera $D = \{(1 : L_1), (2 : L_2), (3 : L_3), (4 : \emptyset)\}$.

Le dernier champs des messages fautifs reçus correspond à la base de données fautive $fsdb_i$. C’est à dire à la base de données générée par la simulation fautive d’un processus au sein du réseau BFS-DEVS. En effet, si lors de la propagation des listes de fautes au sein d’un processus fautif un composant *“Assignment”* est rencontré, le résultat de l’affectation sera stocké dans la base de données fautive $fsdb_i$.

Après avoir rempli le dictionnaire D, il faut procéder à l'évaluation de l'observabilité des fautes contenues dans chaque liste. C'est à dire analyser l'observabilité du contenu des *'valeur'* de chaque éléments de D.

Si on pose :

– L_j avec $j \in [0, N-1]$ la liste de fautes construite par le composant *“Generator”* et contenu dans le message $M_{fault}(L_j, s, fsdb_j)$ ou $M_{safe}(L_j, s)$,

– sdb la base de données symbolique saine connue de tous les modèles atomiques BFS-DEVS,

– $fsdb_j$ avec $j \in [0, N-1]$ la base de données fautive contenu dans $M_{fault}(L_j, s, fsdb_j)$,

– F_i $(i \in \mathbb{N})$ la faute appartenant au modèle de fautes utilisé *(cf. paragraphe 4.1.2)* et issue de la liste de fautes L_j.

– $T_{F_i} = \{((S|V)_j, P^{F_i}) | i, j \in \mathbb{N}\}$ la signature d'une faute F_i avec P^{F_i} le pilote de valeur fautive du signal S_j ou de la variable V_j,

– $P_k = [V_c, V_f, time]$ les pilotes de valeurs d'une base de données contenant une valeur courante V_c et un valeur future V_f ainsi qu'un temps d'affectation $time$,

– $P_k^{F_i} = [V_c, V_f, time]$ les pilotes de valeurs d'une signature de la faute F_i contenant une valeur fautive courante V_c^f et un valeur fautive future V_f^f ainsi qu'un temps d'affectation $time$,

l'observabilité des fautes F_i est définit par :

$$
\begin{aligned}
&\forall fsdb_j \in M_{fault}, \forall sdb, \\
&\quad \forall P_k \in \{fsdb_j, sdb\}, \quad V_c \neq V_f \quad et\, T_{F_i} = \emptyset \Rightarrow \{F_i\} \in L_j\, observables\, ou \\
&\qquad\qquad \forall P_k^{F_i} \in T_{F_i}, \quad V_c \neq V_f^f \quad et\, T_{F_i} \neq \emptyset \Rightarrow \{F_i\} \in L_j\, observables
\end{aligned}
$$

Si les signaux sont des sorties primaires du circuit, toutes les fautes qui leurs sont allouées sont visibles en sortie du circuit. Ces fautes constituent la liste des fautes détectées au sein du circuit servant au calcul de la couverture de fautes.

A chaque fin de simulation *(tout les messages sont reçus par le composant et plus aucun modèle couplé ne doit être activé)*, la liste des fautes détectées L_D est définit comme l'ensemble des fautes appartenant à L_O et dont les signatures portent sur les signaux de sortie S_i :

$$L_D = \sum_{j}^{F_j \in L_O} F_j(T_{S_i})$$

Si la liste de toutes les fautes susceptibles de se produire au sein du circuit est L_T, la couverture de fautes est définit en pourcentage par :

$$C(\%) = \frac{L_D}{L_T}$$

***Remarque** : La méthode d'observabilité des fautes décrite plus haut n'est valable que dans le cas ou il n'existe aucun conflit d'affectation des signaux. En effet on considère qu'un signal ne peut être affecté dans deux processus différents. Dans le cas contraire, le langage VHDL prévoit l'implémentassions d'un fonction de conflit permettant de faire un choix entre les possibilités d'affectation. Si l'on veut déterminer l'observabilité des fautes sur un signal présentant un conflit de valeur, il faudrait alors comparer tous les pilotes des bases de données ou le signal est affecté.*

Exemple

Nous allons à présent donner un exemple illustrant le fonctionnement du composant *"ProcessEngine"* lorsqu'il est actif au sein d'une simulation de fautes avec propagation de listes de fautes.

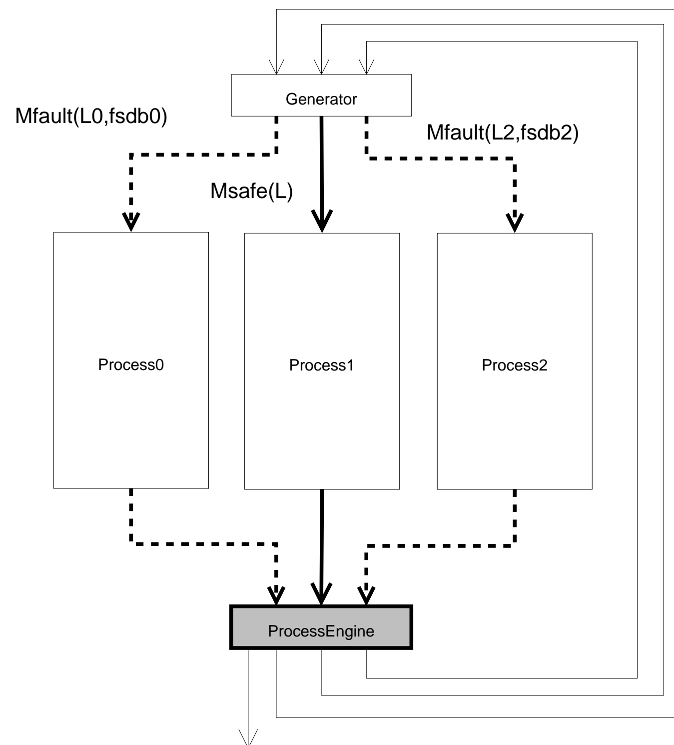


FIG. 8.11*: Simulation de fautes sur une description à 3 processus*

Considérons le cas d'une description à 3 processus illustré par la figure 8.11. Chaque processus possède sa liste de signaux sensitifs de la façon suivante :

Processus	**Process0**	**Process1**	**Process2**
signaux sensitifs	(A,B)	(C)	(D)

TAB. 8.2*: Tableau des signaux sensitifs de chaque processus*

Plaçons nous au début d’une simulation d’un cycle physique avec le signal d’entrée C dans la configuration suivante :

	C_0	C_1
C	’0’	’1’

TAB. 8.3*: Tableau des valeurs du signal C au cycle de simulation C_0 et C_1*

D’après le tableau 8.3, seul le modèle couplé associé au composant *“Process1”* sera activé pour une simulation saine par le message $M_{safe}(L)$ *(chemin en gras sur la figure 8.11)* au cycle de simulation C_1 *(voir composant “Generator”)*. Les deux autres composants *“Process0”* et *“Process2”* seront activés pour une simulation de fautes avec propagation de listes de faute L_0 et L_2 contenues dans les messages $M_{fault}(L_0, fsdb_0)$ et $M_{fault}(L_2, fsdb_2)$ *(chemin en pointillé sur la figure 8.11)*.

Les listes de fautes sont déterminées par le composant *“Generator”* et en partant de l’hypothèse ou aucun signaux ne possède aucunes fautes actives au cycle précédent C_0 *(L_O ne contient aucune fautes possédant les signatures concernant les signaux sensitifs)* :

$$L_0 = (F2_P_0_V)$$

$$L_2 = (F2_P_2_V)$$

$$L = (C_0, F_2_P1_F)$$

Soit S l’unique signal de sortie du circuit sous simulation. Admettons à présent que la simulation saine sur le composant *“Process1”* implique la valeur des signaux dans la base de données symbolique saine suivantes :

	A	**B**	**C**	**D**	**S**
C_0	’1’	’1’	’0’	’0’	’0’
C_1	’1’	’1’	’1’	’0’	’1’

TAB. 8.4*: Base de données symbolique saine : sdb*

Si les deux simulations de fautes sur les deux composants *“Process0”* et *“Process2”* impliques les valeurs suivantes dans les bases de données symboliques fautives :

	A	**B**	**C**	**D**	**S**
C_0	’1’	’1’	’0’	’0’	’0’
C_1	’1’	’1’	’1’	’0’	’0’

	A	**B**	**C**	**D**	**S**
C_0	’1’	’1’	’0’	’0’	’0’
C_1	’0’	’1’	’1’	’0’	’0’

TAB. 8.5*: Bases de données symboliques fautives : $fsdb_0$ et $fsdb_2$*

Lorsque le composant *"ProcessEngine"* reçoit la totalité des messages, il procède à la détermination de l'observabilité des listes de fautes L_0, L_1 et L :

- **Observabilité des fautes appartenant à** L_0 : Si la faute $F2_P_2_V$ possède une signature vide, elle s'effectue en faisant la comparaison des pilotes de données dans la base de données $fsdb_0$ *(cf. tableau 8.5)*. Nous remarquons qu'aucun signal ne change de valeur au cours de la simulation de fautes. Par conséquent les fautes de la liste L_0 qui activeraient le modèle couplé *"process0"* ne sont pas observables sur la signal S au cycle de simulation C_1. Aucune faute ne sera ajoutée à la liste des fautes détectées L_D.
- **Observabilité des fautes appartenant à** L_2 **:** Si la faute $F2_P_2_V$ possède une signature $T = \{A : [None,'0',1]\}$, cette faute a une influence car sa signature possède une valeur future '0' différente de la valeur saine '1' pour le signal A. Elle sera donc localement observable sur A au cycle de simulation C_1. Le signal A n'étant pas un signal de sortie, la liste L_D reste inchangée.
- **Observabilité des fautes appartenant à** L **:** Au cours de la simulation saine, la simulation de fautes est effectuée en parallèle au sein du composant *"Process1"*. En effet le scénario fautif est constamment propagé en parallèle de la simulation saine. Les fautes contenues dans L sont les fautes qui impliqueraient par leur présence une non activation du composant "Process1". Nous considérons que les signatures de ces fautes sont vides. Le tableau 8.4 montre les variations de valeurs entre les signaux du cycle de simulation C_0 au cycle de simulation C_1. Le signal de sortie S passe de la valeur '0' à '1'. Par conséquent si les fautes de la liste L étaient présentes au sein du circuit, cette variation n'aurait pas au lieu. Donc les fautes de la liste L sont observables sur le signal de sortie S et sont ajoutées à la liste $L_D = L = (C_0, F_2_P1_F)$.

Si le nombre de fautes susceptibles de se produire au sein du circuit durant son fonctionnement est de 20, la couverture de fautes au cycle de simulation C_1 est donnée par :

$$C(\%) = \frac{L_D}{L_T} = \frac{2}{20} \simeq 10\%$$

8.5.2 Spécifications BFS-DEVS

Nous allons à présent donner les spécifications pseudo-code BFS-DEVS décrivant le modèle atomique *"ProcessEngine"* :

$$\boxed{MA_{(ProcessEngine)} = < X', S', Y', F, \delta'_{ext}, \delta'_{int}, \delta_{fault}, \lambda', t'_a >}$$

avec,

$Ports_entree = \{'In'_0, ..., 'In'_{N-1}\}$ avec $X_{In_0} = ... = X_{In_{N-1}} = X_{In} = \{M_{safe}, M_{fault}, \emptyset\}$
$Ports_sortie = \{'Out'_0, ..., 'Out'_{M-1}, 'FB'_0, ..., 'FB'_{N-1}\}$ avec $X_{Out} = \{M_{safe}, M_{fault}, \emptyset\}$

$X' = \{('In'_0, v), ..., ('In_{N-1}, v')|v \in X_{In}\}$
$Y' = \{('Out'_0, v), ..., ('Out'_{M-1}, v), ..., ('FB'_0, v), ..., ('FB'_{N-1}, v)|v \in X_{Out}\}$
$S' = S \cup S^f = \{'ACTIVE', 'PASSIVE'\} \times \Re^+ \times \Re^+ \cup \emptyset$
$F = \{'F'_1, 'F'_2, 'F'_3\}$

```
δext(S,F,X){
Si status <> 'ACTIVE' :

  // acquisition du message en entrée
  msg = getMessage
  // incrémentassions du compteur de messages fautifs
  Rc++

  // calcul des process actifs
  pour tout les signaux 'INTERNAL'
    Si signal.sensitivity == 1 :
    // si le signal est sensitif
     activeProcesscomparaison += (msg, signal.process)

  // si le composant à reçu tout les messages
  Si Rc == getNbActiveProcess() :
   Si activeProcesscomparaison <> ∅ :
    // passage à l'état actif (début cycle symbolique)
    S = ('ACTIVE',0,0)
   Sinon :
    // passage à l'état passif (fin cycle physique)
    S = ('PASSIVE',0,0)

  retourne S}

δint(S,F){
  // retour à l'état de repos
  S = ('PASSIVE',RC = 0,∞)
  retourne S}

δfault(S,F,X){
  Si msg <> 'safe' :
   retourne δext

  retourne S}
```

```
λ(S){
  Si simulation de fautes :
    // on effectue l'observabilité des fautes actives
    getFaultObservability()

  // mise à jour des bases de données
  updateDB()

  Si status == 'PASSIVE' :
  // fin d'un cycle physique
    pour tout port Out_i dans [0,M] :
     poke('Out'_i, S_i)
   //début d'un cycle symbolique
   Sinon :
    pour tout (msg, process) dans activeProcesscomparaison :
     Si type(msg) == 'safe' :
       poke(OPort(process), Message())
     Sinon :
       poke(OPort(process), Message(msg.sdb))

t_a(S){retourne σ}
```

Bibliographie

[sta, 2000] (janvier 2000). *IEEE Standard VHDL language Reference Manual*. IEEE standard 1076.

[Abramovici et al., 1990] Abramovici, M., Breuer, M. A., et Friedman, A. (1990). *Digital Systems Testing and Testable Design*. Computer Science Press.

[Agrawal et Agrawal, 1972] Agrawal, V. et Agrawal, P. (septembre 1972). An automatic test generation system for illiac IV logic boards. Dans *IEEE Transactions on Computers*, volume 21, pages 1015–1017.

[Agrawal et al., 1989] Agrawal, V., Cheng, K., et Agrawal, P. (1989). Simulation of high speed computer logic. Dans *IEEE Transactions on CAD*, pages 131–138.

[Agrawal, 1981] Agrawal, V. D. (décembre 1981). Sampling techniques for determining fault coverage in LSI circuits. Dans *Jour. Digital Systems*, volume 5, pages 189–202.

[Armstrong, 1972] Armstrong, D. (mai 1972). A deductive method for simulating faults in logic circuits. Dans *IEEE Transactions on Computers*, volume 21, pages 464–471.

[Ashenden, 2001] Ashenden, P. J. (2001). *The designer guide to VHDL*. Morgan Kaufmann Publishers.

[Beazley, 2000] Beazley, D. M. (2000). *Advanced Python Programming*. Department of Computer Science, University of Chicago.

[Beizer, 1990] Beizer, B. (1990). *Software Testing Techniques (2nd ed.)*. Van Nostrand Rheinold, New York.

[Bisgambiglia et al., 2001] Bisgambiglia, P., Federici, D., et Santucci, J.-F. (février 2001). Fault modeling and simulation at behavioral level. Dans *Proceedings of IEEE 2nd Latin American Test Workshop (LATW'01)*, pages 45–50. Mexico.

[Bolduc et Vangheluwe, 2001] Bolduc, J.-S. et Vangheluwe, H. (juin 2001). pythonDEVS : A modeling and simulation package for classical hierarchal DEVS. Dans *Rapport technique, MSDL, Université de McGill*.

[Booch et al., 1998] Booch, G., Rumbaugh, J., et Jacobson, I. (1998). *The unified modeling language reference manual*. Addison-Wesley.

[Boulding, 1956] Boulding, K. E. (1956). General systems theory. volume 2, pages 197–208. The Skeleton of Science.

[Breuer et Friedman, 1976] Breuer, M. A. et Friedman, A. D. (1976). *Diagnosis and Reliable Design of Digital Systems*. Computer Science Press.

[Buonanno et al., 1997] Buonanno, G., Ferrandi, F., Ferrandi, L., Fummi, F., et Sciuto, D. (mars 1997). How an "evolving" fault model improves the behavioral test generation. Dans *Proceedings of IEEE Seventh Great Lakes Symposium on VLSI (GLS-VLSI)*. Urbana, IL, USA.

[Buonanno et al., 1996] Buonanno, G., Ferrandi, F., Fummi, F., et Sciuto, D. (1996). A testing methodology for vhdl based high-level designs. Dans *IEEE International High Level Design Validation and Test Workshop*.

[Capocchi et al., 2003] Capocchi, L., Bernardi, F., Federici, D., et Bisgambiglia, P. (2003). Transformation of VHDL descriptions into DEVS models for fault modeling and simulation. Dans *Proceedings of IEEE Systems, Man and Cybernetics Conference (SMC'03)*, pages 1205–1211. Washington D.C., USA.

[Chow et Zeigler, 2003] Chow, A. C. et Zeigler, B. P. (2003). *Revised DEVS : A Parallel Hierarchical Modular Modeling Formalism*. ACIMS Laboratory, University of Tucson, Arizona.

[Chow et Zeigler, 1994] Chow, A. C. et Zeigler, B. P. (décembre 1994). Abstract simulator for the parallel DEVS formalism. Dans *Proceedings of Fifth Annual Conference on AI, Simulation and Planning in High Autonomy Systems*, pages 157–163. Gainesville, FL.

[Coelho, 1989] Coelho, D. R. (1989). *The VHDL handbook*. Kluher Academic Publisher.

[Corno et al., 2000a] Corno, F., Cumani, G., Reorda, M. S., et Squillero, G. (novembre 2000a). An RT-level fault model with high gate level correlation. Dans *IEEE International High Level Design Validation Workshop*, page 3. The Claremont Resort & Spa, Berkeley, California.

[Corno et al., 2000b] Corno, F., Manzone, A., Pincetti, A., Reorda, M. S., et Squillero, G. (2000b). Automatic test bench generation for validation of rt-level descriptions : an industrial experience. Dans *Proceedings of Design, Automation and Test in Europe (DATE'00)*, pages 385–389.

[Corno et al., 1997] Corno, F., Prinetto, P., et Reorda, M. S. (novembre 1997). Testability analysis and ATPG on behavioral RT-level VHDL. Dans *Proceedigns of IEEE International Test Conference (ITC'99)*, pages 753–759. Washington D. C., USA.

[Demba et al., 1990] Demba, S., Ulrich, E., Panetta, K., et Giramma, D. (juin 1990). Experiences with concurrent fault simulation of diagnostic programs. Dans *Proceedings of Computer-Aided Design of Integrated Circuits and Systems*, volume 9, pages 621–628. Washington, DC, USA.

[Devadas et al., 1996] Devadas, S., A.Ghosh, et Keutzer, K. (1996). An observability-based code coverage metric for functional simulation. Dans *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD96)*, pages 418–425. San Jose, California, United States.

[Fallah et al., 1998] Fallah, F., Devadas, S., et Keutzer, K. (1998). OCCOM : Efficient computation of observability-based code coverage metrics for functional verification. Dans *Proceedings of 34th Design Automation Conference (DAC'98)*, pages 152–157. San Francisco, California, United States.

[Farzan Fallah et Devadas, 1999] Farzan Fallah, P. A. et Devadas, S. (1999). Simulation vector generation from hdl descriptions for observability-enhanced statement coverage. Dans *Proceedings of 36th ACM/IEEE Design Automation Conference (DAC'99)*, pages 666–671. New Orleans, Louisiana, United States.

[Ferrandi et al., 1998] Ferrandi, F., Fummi, F., et Sciuto, D. (1998). Implicit test generation for behavioral VHDL models. Dans *Proceedings of IEEE International Test Conference (ITC'98)*, pages 436–441.

[Fin et Fummi, 2000] Fin, A. et Fummi, F. (2000). A VHDL error simulator for functional test generation. Dans *Proceedings of Design, Automation and Test in Europe (DATE'00)*, pages 390–395. Paris, France.

[Fulvio Corno, 2000] Fulvio Corno, Matteo Sonza Reorda, G. S. (novembre 2000). RT-level fault simulation techniques based on simulation command scripts. Dans *Proceedings of XV Conference on Design of Circuits and Integrated Systems (DCIS'00)*, pages 825–830. Le Corum, Montpellier.

[Gai et al., 1988b] Gai, S., Montessoro, P., et Somenzi, F. (1988b). The performance of the concurrent fault simulation algorithms in MOZART. Dans *Proceedings of 25th ACM/IEEE conference on Design automation (DAC'88)*, pages 692–697. Atlantic City, New Jersey, United States.

[Gai et al., 1988a] Gai, S., Montessoro, P., et Somenzi, F. (septembre 1988a). MOZART : A concurrent multi level simulator. Dans *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 7, pages 1005–1016.

[Gai et al., 1987] Gai, S., Somenzi, F., et Ulrich, E. (1987). Advance in concurrent multilevel simulation. *IEEE Transactions on CAD*, 6 :1006–10012.

[Ghosh, 2000] Ghosh, S. (2000). *Hardware Description Languages : Concepts and Principles*. Wiley-IEEE Press.

[Goloubeva et al., 2002] Goloubeva, O., Reorda, M. S., et Violante, M. (septembre 2002). Behavioral-level fault models comparison : An experimental approach. Dans *Proceedings of*

Computer-aided Technologies in Applied Mathematics (ICAM'02). Tomsk, Russia.

[Hurst, 1998] Hurst, S. L. (1998). *VLSI Testing : digital and mixed analogue/digital techniques*. IEE Publishing.

[Kearney, 1984] Kearney, M. A. (octobre 1984). DECSIM : A multi-level simulation system for digital design. Dans *Proceedigns of International Conference on Computer Design, New York*, pages 206–209.

[Kofman et al., 2000] Kofman, E., Giambiasi, N., et Junco, S. (2000). FDEVS : A general DEVS-based formalism for fault modeling and simulation. Dans *Proceedings of European Simulation Symposium*, pages 77–82. Hamburg, Germany.

[Landrault, 2004] Landrault, C. (2004). *Test de circuits et de systèmes intégrés (Traité EGEM, série Electronique et micro-électronique)*. Lavoisier.

[Machlin, 1987] Machlin, D. (1987). *A General Purpose Transversal Mechanism for Concurrent Logic Simulation*. Thèse de Doctorat.

[McCabe, 1976] McCabe, T. (1976). A software complexity measure. Dans *IEEE Transactions on Software Engineering*, volume 2, pages 308–320.

[Miczo, 1986] Miczo, A. (1986). *Digital logic testing and simulation*. Harper & Row Publishers, Inc., New York, NY, USA.

[Montessoro et Gai,] Montessoro, P. et Gai, S. CREATOR : General and efficient multilevel concurrent fault simulation. Dans *Proceedings of 28th conference on ACM/IEEE design automation (DAC'91)*. San Francisco, California, United States.

[Oestereich, 2002] Oestereich, B. (2002). *Delveloping software with UML*. Addison-Wesley.

[Paoli et al., 2004] Paoli, C., Nivet, M., Bernardi, F., et Capocchi, L. (2004). Simulation-based validation of VHDL descriptions using constraints logic programming. Dans *Proceedings of 5th IEEE Workshop on RTL and High Level Testing (WRTLT'04)*. Osaka, Japan.

[Patil, 1991] Patil, S. (1991). *Parallel algorithms for test generation and fault simulation*. Thèse de Doctorat, Champaign, IL, USA.

[Phillips et Tellier, 1978] Phillips, N. D. et Tellier, J. G. (octobre 1978). Efficient event manipulation - the key to large scale simulation. Dans *Proceedings of IEEE International Test Conference*, pages 266–273.

[Riesgo et Uceda, 1996] Riesgo, T. et Uceda, J. (1996). A fault model for VHDL descriptions at the register transfer level. Dans *Proceedings of European Design Automation Conference (EURO-DAC/EURO-VHDL)*, pages 462–467.

[Santucci et al., 1993] Santucci, J., Courbis, A., et Giambiasi, N. (1993). Behavioral testing of digital circuits. *Journal of Microelectronic System Integration*, 1(1).

[Seth et al., 1990] Seth, S. C., Agrawal, V. D., et Farhat, H. (1990). A statistical theory of digital circuit testability. Dans *IEEE Trans. Comput.*, volume 39, pages 582–586.

[Thaker et al., 1999] Thaker, P., Agrawal, V., et Zaghloul, M. (1999). Validation vector grade (VVG) : A new coverage metric for validation and test. Dans *Proceedings of 17th IEEE VLSI Test Symposium*, pages 182–188.

[Thaker et al., 2000] Thaker, P. A., Agrawal, V. D., et Zaghloul, M. E. (2000). Register-transfer level fault modeling and test evaluation techniques for VLSI circuits. Dans *Proceedings of IEEE International Test Conference (ITC'00)*, pages 940–949.

[Ulrich, 1985] Ulrich, E. (novembre 1985). Concurrent simulation at the switch, gate and register levels. Dans *Proceedings of IEEE International Test Conference (ITC'85)*, pages 703–709. Philadelphia, USA.

[Ulrich, 1978] Ulrich, E. (septembre 1978). Event manipulation for discret simulation requiring large number of events. Dans *Communications of the ACM*, volume 21, pages 777–785.

[Ulrich et al., 1994] Ulrich, E., Agrawal, V., et Arabian, J. (1994). *Concurrent and Comparative Discrete Event Simulation*. Kluwer Academic publisher.

[Vangheluwe et al., 2002] Vangheluwe, H., de Lara, J., et Mosterman, P. (avril 2002). An introduction to multi-paradigm modelling and simulation. Dans *Proceedings of Conference AI, Simulation and Planning in High Autonomy Systems (AIS'02)*. Lisboa, Portugal.

[Wainer, 2004] Wainer, G. (2004). Using DEVS for modeling and simulation of computer networks. Dans *Symposium on Performance on Computer and Telecommunication Systems*.

[Wainer et al., 2001] Wainer, G., Daicz, S., et Troccoli, A. (2001). Experiences in modeling and simulation of computer architectures in DEVS. Dans *Transactions of the Society for Computer Simulation International*, volume 18, pages 179–202.

[WILLIAMS et BROWN, 1981] WILLIAMS, T. W. et BROWN, N. (decembre 1981). Defect level as a function of fault coverage. Dans *IEEE Transactions on Computers*, volume 30, pages 987–988.

[Zeigler, 1976] Zeigler, B. P. (1976). *Theory of Modeling and Simulation*. Academic Press.

[Zeigler, 1990] Zeigler, B. P. (1990). *Object-Oriented Simulation with Hierarchical, Modular Models*.

[Zeigler et al., 2000] Zeigler, B. P., Praehofer, H., et Kim, T. G. (2000). *Theory of Modeling and Simulation, Second Edition*. Academic Press.

Listes des Publications

Articles dans des conférences internationales avec actes et comité de lecture :

1. F. Bernardi, **L. Capocchi**, E. Innocenti, *Distributed DEVS Simulation using the MOSIX Environment,* Proceedings of the SCS Summer Computer Simulation Conference (SCSC 2005), San Diego, CA, USA.
2. **L. Capocchi**, F. Bernardi, D. Federici, P. Bisgambiglia, *A DEVS-based Concurrent and Comparative Fault Simulation Algorithm,* Proceedings of the SCS Summer Computer Simulation Conference (SCSC 2005), San Diego, CA, USA.
3. E. Innocenti, F. Bernardi, A. Muzy, **L. Capocchi**, J.F. Santucci, *Distributed Fire Spreading Simulation using OpenMOSIX,* Proceedings of the first Open International Conference on Modeling and Simulation (OICMS 2005), Clermont-Ferrand, France.
4. **L. Capocchi**, F. Bernardi, D. Federici, P. Bisgambiglia, *A DEVS-based Modeling Behavioral Fault Simulator for RT-Level Digital Circuits*, Proceedings of the SCS Summer Computer Simulation Conference (SCSC 2004), p. 481-186, San José, CA, USA.
5. C. Paoli, M.L. Nivet, F. Bernardi, **L. Capocchi**, *Simulation-based validation of VHDL descriptions using constraints logic programming,* Proceedings of the 5th IEEE Workshop on RTL and High Level Testing (WRTLT'04), Osaka, Japon.

6. **L. Capocchi**, F. Bernardi, D. Federici, P. Bisgambiglia, *Transformation of VHDL Descriptions into DEVS Models for Fault Modeling and Simulation*, Proceedings of the IEEE Systems, Man and Cybernetics Conference (SMC03), p. 1205–1211, 2003, Washington D.C., USA.

Poster dans une conférence internationale avec comité de lecture :

1. **L. Capocchi**, D. Federici, F. Bernardi, P. Bisgambiglia, *Behavioral Fault Simulation for VHDL Descriptions using the DEVS Formalism*, Fast Abstract at the IEEE Pacific Rim Dependable Computing International Conference (PRDC), 2004, Papeete, Tahiti, French Polynesia.

Articles soumis *(processus de soumission en cours)* :

1. **L. Capocchi**, F. Bernardi, D. Federici, P. Bisgambiglia, *BFS-DEVS : A General DEVS-Based Formalism For Behavioral Fault Simulation*, Elsevier Simulation Pratice and Theory.
2. **L. Capocchi**, F. Bernardi, D. Federici, P. Bisgambiglia, *A High Level Discrete Event-Based Concurrent Fault Simulator*, JETTA, Journal of Electronic Testing.

Liste des figures

1.1 Aperçu de la démarche. 3

2.1 Place du test dans le flôt de conception d'un circuit. 8

2.2 Vecteurs de test. 11

2.3 Méthodologie de la simulation de fautes. 12

2.4 Évolution de la conception numérique. 14

2.5 Évolution et généralisation de la simulation concurrente. 18

2.6 Exemple général d'une simulation comparative et concurrente. 19

2.7 Propagation de l'effet d'une faute au niveau porte logique. 23

2.8 Modèle atomique en action. 27

2.9 Trajectoires d'un modèle atomique. 28

2.10 Hiérarchie de modélisation DEVS. 29

2.11 Arbre hiérarchique de simulation DEVS. 31

3.1 Intégration du formalisme BFS-DEVS. 39

3.2 Construction de la librairie de modèles BFS-DEVS. 41

3.3 Technique de propagation de listes de fautes. 48

3.4 Simulation de fautes concurrente d'un réseau BFS-DEVS. 51

4.1 Schéma du *“parser”* VHDL/BFS-DEVS. 60
4.2 Réseau BFS-DEVS du registre 8 bits. 62
4.3 Vue schématique de la simulation de fautes concurrente BFS-DEVS. 65
4.4 Schéma d’un cycle de simulation VHDL. 66
4.5 Schéma d’un cycle de simulation de fautes concurrente. 67
4.6 Propagation intra-processus. 74
4.7 Auto-activation et conflit d’affectation des signaux d’un processus. 77
4.8 Multi-activation des processus 1 et 2 à partir du même signal interne. 77
4.9 Propagation inter-processus. 79
4.10 Réseau BFS-DEVS du registre 8 bits. 80
4.11 Réseau BFS-DEVS du registre 8 bits *(cycle d’initialisation)*. 81
4.12 Réseau BFS-DEVS du registre 8 bits *(cycle symbolique* C_0^{+1}*)*. 86

5.1 Diagramme de classes global de l’architecture. 91
5.2 Diagramme des classes pour le domaine VHDL. 94
5.3 Diagramme des classes pour le simulateur DEVS. 96
5.4 Graphe d’états du modèle atomique BFS-DEVS *“Generator”*. 97
5.5 Graphe d’états du modèle atomique BFS-DEVS *"Assignment"* 98
5.6 Graphe d’états du modèle atomique BFS-DEVS *"Conditional"*. 100
5.7 Graphe d’états du modèle atomique BFS-DEVS *"Junction"*. 102
5.8 Graphe d’états du modèle atomique BFS-DEVS *“ProcessEngine”* 104
5.9 Exemple d’un modèle BFS-DEVS (a) avec son arbre de simulation (b). 107
5.10 Diagramme de séquence de l’algorithme de simulation BFS-DEVS. 108

6.1 Architecture générale du prototype BFS-DEVS. 113
6.2 Évolution de CF(%) en fonction du nombre de cycles physiques VHDL. 114
6.3 Gain CPU après suppression des instructions redondantes. 116
6.4 Architecture du prototype utilisant RAGE. 117
6.5 Comparaison de RAGE avec un ATPG commercial. 119

6.6 Utilisation des séquences de test RAGE pour le simulateur BFS-DEVS. 120
6.7 Graphiques des résultats obtenus avec BFS-DEVS. 121
6.8 Corrélation CF/CI pour la simulation BFS-DEVS. 122
6.9 Couverture de fautes en fonction du modèle de fautes $\{F_1, F_2, F_3\}$. 123

7.1 Simulation concurrente BFS-DEVS d'un système. 126

8.1 Modèle atomique “Generator”. 130
8.2 Modélisation BFS-DEVS d'une description VHDL à 3 *'process'*. 134
8.3 Le modèle atomique *“Conditional”*. 139
8.4 Représentation BFS-DEVS de l'instruction conditionnelle 'case'. 142
8.5 Représentation BFS-DEVS de l'instruction conditionnelle 'case'. 145
8.6 Modèle atomique*”Assignment”*. 148
8.7 Modèle atomique *“Junction”*. 152
8.8 Représentation BFS-DEVS de l'instruction de contrôle VHDL “if then else endif”. 155
8.9 Modèle atomique *“ProcessEngine”* 159
8.10 Simulation de fautes sur une description à 3 processus 161
8.11 Simulation de fautes sur une description à 3 processus 164

Liste des tableaux

2.1 Loi de Moore. 14

4.1 Tableau des règles d'accès la base de données. 64

4.2 Liste totale des fautes pour le registre 8 bits. 80

4.3 Valeurs initiales des signaux d'entrée. 82

4.4 Tableau des résultats du cycle d'initialisation. 82

4.5 Tableau des pilotes des signaux après l'exécution du cycle C_0. 83

4.6 Mise à jour des pilotes de REG, STRB, ENBLD et DO au cycle C_0. 84

4.7 Mise à jour des signatures des fautes de L_O au cycle C_0. 86

4.8 Tableau des résultats de la simulation de $DS1_1$. 87

4.9 Tableau des pilotes de REG, STRB, ENBLD et DO au cycle C_0^{+1}. 87

4.10 Mise à jour des signatures des fautes de L_O au cycle C_0^{+1}. 88

4.11 Listes des fautes détectées L_D et localement observables L_O. 88

6.1 Caractéristiques de quelques benchmarks ITC'99. 111

6.2 Résultats des simulations BFS-DEVS sur les dix premiers benchmarks ITC'99. . . 114

6.3 Résultats obtenus par la méthode RAGE. 118

6.4 Résultats de la simulation BFS-DEVS des séquences de test RAGE. 120

8.1 Valeurs des signaux A,B,C,D au cycle C_0 et C_1. 135
8.2 Tableau des signaux sensitifs de chaque processus 164
8.3 Tableau des valeurs du signal C au cycle de simulation C_0 et C_1 165
8.4 Base de données symbolique saine : sdb 165
8.5 Bases de données symboliques fautives : $fsdb_0$ et $fsdb_2$ 165

Liste des algorithmes

1 Algorithme du simulateur DEVS. 33
2 Algorithme du coordinateur DEVS. 35
3 Algorithme du coordinateur *"Root"*. 36
4 Compléments algorithmiques d'un simulateur. 46
5 Compléments algorithmiques d'un coordinateur. 47
6 Construction des listes de fautes L_p. 133

Résumé

Simulation concurrente de fautes comportementales pour des systèmes à événements discrets :

Application aux circuits digitaux

La *Simulation Comparative et Concurrente (SCC)* permet d'effectuer plusieurs simulations d'un système en une seule exécution. Une des premières applications de la SCC a été la *Simulation de Fautes Concurrente (SFC)* permettant la simulation de fautes au sein des systèmes digitaux décrits au niveau portes logiques. De nos jours, les concepteurs de circuits évitent de travailler sur ces modèles logiques et préfèrent utiliser des descriptions plus abstraites basées sur des *langages de description de matériel* comme le *VHDL (Very high speed integrated circuits Hardware Description Language)*. Ces langages permettent de modéliser et de simuler le comportement des circuits digitaux mais ils ne sont pas appropriés pour la simulation concurrente des *comportements fautifs ou fautes*. Les barrières au développement d'un simulateur concurrent de fautes comportementales sont le manque de modèles de fautes réalistes et la difficulté à mettre en œuvre les algorithmes concurrents au sein d'un noyau de simulation.

Pour répondre à cette problématique, nous proposons le formalisme *BFS-DEVS (Behavioral Fault Simulator for Discrete EVent system Specification).* Ce formalisme permet de modéliser et de simuler les fautes comportementales sur des systèmes à événements discrets comme les circuits digitaux décrits en VHDL. Il dérive du formalisme *DEVS (Discrete EVent system Specification)* introduit par le professeur B.P. Zeigler à la fin des années 70. Le noyau de simulation BFS-DEVS intègre les algorithmes concurrents de la SFC et il s'appuie sur une *technique de propagation de listes de fautes* au sein des modèles du système. Cette technique améliore la rapidité du processus de simulation car elle permet la détection simultanée de plusieurs fautes et simplifie également *l'observabilité des résultats* en fin de simulation.

Mots clés : DEVS, simulation de fautes, simulation comparative et concurrente, tests de circuits, VHDL.

Abstract

Concurrente behavioral fault simulation on discrete event systems :

Application on digital circuits

The *Concurrent and Comparative Simulation (CCS)* allows several simulations on a system in one single pass. One of the first applications of CCS has been the *Concurrent Fault Simulation (CFS)* for fault simulation in digital systems described at the gate level. However, nowadays digital designers focus on more abstract languages such as VHDL *(Very high speed integrated circuits Hardware Description Language)* rather than on these logical models. Modeling and simulating digital circuits behaviors is possible using these languages, but they do not allow the concurrent simulation of faulty behaviors, also simply called *faults*. Technical barriers for the design of a concurrent fault simulator are on the one hand the lack of realistic fault models and on the other hand the difficulty to integrate the concurrent algorithms into a simulation kernel.

To reach this objective, we propose the BFS-DEVS formalism (*Behavioral Fault Simulator for Discrete EVent system Specification*). This formalism allows to model and simulate behavioral faults on discrete event system such as digital circuits described with VHDL. Its theoretical fundation is the DEVS *(Discrete EVent system Specification)* formalism introduced by Zeigler in the late 70's. The BFS-DEVS simulation kernel integrates the CFS concurrent algorithms and is based on a propagated fault lists technique inside the models of the system. This technique speeds up the simulation processus since it allows the simultaneous detection of several faults and also simplify results observability at the end of the simulation.

Keywords : DEVS, fault simulation, concurrent and comparative simulation, circuits tests, VHDL.

SPE (Systèmes Physiques pour l'Environnement), UMR CNRS 6134
Quartier Grossetti, BP 52, 20250 Corté
`http://spe.univ-corse.fr`