Evaluation de la sûreté des systèmes aéronautiques grâce aux plateformes virtuelles

Julie Roux

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Nano Électronique et Nano Technologies**

Arrêtée ministériel : 25 mai 2016

Présentée par

Julie ROUX

Thèse dirigée par **Vincent BEROULLE (directeur)**
et codirigée par **Katell MORIN-ALLORY (co-directrice) et Lilian BOSSUET (co-encadrant)**

préparée au sein du **Laboratoire de conception et d'intégration des systèmes (LCIS)**
dans **École Doctorale Électronique, Électrotechnique, Automatique et Traitement du Signal (EEATS)**

Evaluation de la sûreté des systèmes aéronautiques grâce aux plateformes virtuelles

Safety Evaluation of Aircraft Systems using Virtual Platforms

Thèse soutenue publiquement le **07 Janvier 2022**,
devant le jury composé de :

Vincent BEROULLE
Professeur des universités, LCIS, Grenoble Alpes Universités,
Directeur de thèse

Katell Morin-Allory
Maître de conférences, TIMA, Grenoble Alpes Universités,
Co-Directrice de thèse

Arnaud Virazel
Professeur des universités, LIRMM, Université de Montpellier,
Rapporteur et Président

Sébastien Pillement
Professeur des universités, IETR, École Polytechnique de l'Université de Nantes,
Rapporteur

Olivier Sentieys
Professeur des universités, INRIA, Université de Rennes,
Examinateur

Régis Leveugle
Professeur des universités, AMfoRS Grenoble Alpes Universités,
Examinateur

Lilian Bossuet
Professeur des universités, LHC, Université de Lyon,
Invité, Co-encadrant de thèse

"À mes parents."

Remerciements

Je tiens à remercier tous ceux qui ont participé, tout au long de ma thèse à me soutenir et m'aider pour avancer.

Les partenaires du projet, Gilles, Frédéric, Régis, François, Lilian, pour leurs retours constructifs et leur aide apportée tout au long de la thèse.

Vincent et Katell pour leur disponibilité, le temps qu'ils m'ont accordé tout au long de la thèse, et leur bienveillance vis à vis des de tous les aléas de la thèse.

Toutes les personnes de l'équipe CDSI, pour leur bonne humeur et leur accueil.

Ma famille et mes proches pour leur soutien inconditionnel.

Table des matières

1 Introduction

Sommaire

1.1 Contexte et motivations

Les systèmes numériques complexes sont aujourd'hui omniprésents dans le monde dans lequel nous évoluons. Énergie, transports, santé, système bancaire sont autant de domaine où des systèmes critiques sont utilisés. L'impact d'une défaillance de l'un de ces systèmes peut être dramatique pour ses utilisateurs, ou pour son environnement. Citons quelques exemples :

— L'un des plus connus, l'accident nucléaire de Tchernobyl [1] de 1986 a eu un impact environnemental, économique, politique et humain très important. Le nombre de mort est difficile à estimer réellement et il varie de 50 à des milliers de morts. Cet accident a été la conséquence de défaillances couplées à des erreurs d'exploitation.

— Plus récemment en 2009, le crash du vol Rio-Paris [2] a entraîné la mort de 228 personnes. Cet accident est encore une fois multi-factoriel : défaillance des sondes Pitots (mesure de vitesse) et réactions inappropriées des pilotes.

— Imai [3] recense plusieurs accidents dans le milieu de l'aéronautique liés à chaque fois à des défaillances des systèmes de capteurs et à des erreurs d'exploitation.

Les sources de défaillances possibles peuvent être multiples : attaques intentionnelles, perturbations dues à l'environnement, erreur de conception, erreur d'exploitation. Pour tous ces systèmes critiques, les concepteurs doivent prouver à des autorités de certification que leur système est robuste.

Différentes études [4–7], basées sur l'expérience acquise et les erreurs passées ont mis en évidence les enjeux majeurs lors du développement et de l'évaluation de la robustesse des systèmes critiques :

— La communication entre les développeurs systèmes, logiciels et matériels : les concepteurs de chaque domaine évoluent dans des environnements et des équipes différentes. Ces concepteurs spécifient leurs besoins sans langage commun. Cela peut conduire à des problèmes de communications inter-domaines conduisant à des erreurs.

— La planification d'une analyse de sûreté détaillée et précise : les analyses sont parfois difficiles à coordonner entre les différents concepteurs lors du développement de systèmes complexes. De plus, l'absence de planification peut entraîner des oublis. Il est nécessaire de planifier de façon détaillée et précise chacune des analyses à effectuer.

— Recherche d'un compromis entre analyses préliminaires et analyses finales. Les analyses en début de flot de conception sont souvent peu précises car les détails d'implémentation ne sont pas connus. À l'inverse, plus le projet avance, plus les analyses peuvent être précises et détaillées. Néanmoins, les analyses finales sont souvent très coûteuses si un problème est relevé. Il est donc nécessaire de trouver le meilleur compromis entre ces différents niveaux d'analyse.

— Automatisation des systèmes d'analyses de sûreté afin d'éviter les erreurs humaines, et d'accélérer les analyses afin d'envisager des analyses plus complexes.

— Prise en compte des effets sur le système à chaque étape du développement (fonctionnel, matériel, logiciel) : afin de prendre en compte toutes les défaillances possibles, et de pouvoir prendre des contre-mesures à tous les niveaux de conception, les effets des défaillances sur le système doivent être pris en compte à chacune des étapes du flot de conception.

Pour répondre à ces enjeux, différentes normes proposent des méthodes à suivre pour les analyses de sûreté et la conception de systèmes critiques. La norme IEC-61508 est un standard d'analyse de sûreté fonctionnelle applicable à toutes les industries. De nombreuses normes proposent des adaptations de la norme IEC61508 à des domaines spécifiques (aéronautique, automobile, nucléaire, *etc.*).

La thèse se déroule dans le cadre du projet SafeAir financé par La Région Auvergne Rhone Alpes. Le projet se déroule avec trois laboratoires et deux partenaires industriels : THALES et AEDVICES Consulting. Nous nous plaçons donc dans le contexte particulier des systèmes aéronautiques.

Dans le domaine de l'aéronautique, comme illustré par la figure 1.1, les conditions de vol à haute altitude font que les systèmes sont soumis à de plus forts flux de neutrons [8]. Ces neutrons peuvent être responsables de perturbations dans le système (appelées bit-flips). De ce fait, les analyses des effets de ces bits-flips dans le système doivent être détaillées.

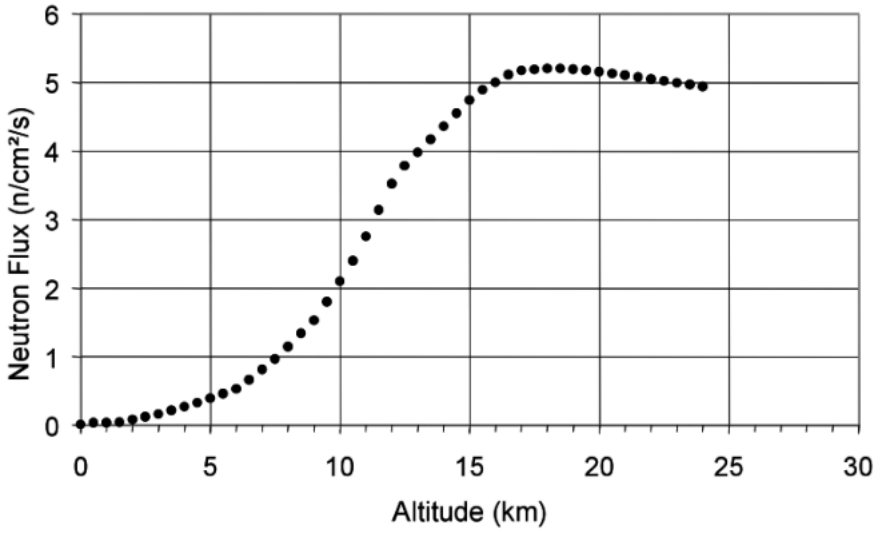

FIGURE 1.1 – *Flux de neutrons dans l'atmosphère en fonction de l'altitude à une latitude de 45° (Modèle NASA [9])*

Les standards de l'aéronautique [10, 11] recommandent des analyses le plus tôt possible dans le flot de conception (dès la spécification fonctionnelle), et à chaque étape du flot de conception top-down.

Actuellement, les analyses de sûreté des systèmes utilisent rarement des outils automatisés. Les analyses actuelles sont faites de manières empiriques, et basées sur l'expérience des ingénieurs. Le manque d'automatisation rend les analyses chronophages et conduit les ingénieurs à prendre plus de contre-mesures que nécessaire. De plus, le nombre de scénarios à simuler est limité, et il est difficile d'étudier l'effet de la propagation d'une faute à travers le système complet.

Des solutions automatisées d'analyse de sûreté existent dans la littérature depuis de nombreuses années, mais elles présentent toutes leurs limitations :

— Les solutions de simulation à bas niveau (niveau portes ou registres) sont précises mais trop chronophages et ne permettent pas de prendre en compte le système complet et sa spécification.

— Les solutions de simulation haut-niveau (niveau système) sont rapides et apparaissent en début de flot de conception mais elles conduisent à des problèmes de réalisme.

La problématique de la thèse est donc de proposer une solution automatisée d'évaluation de la robustesse de systèmes complexes, tôt dans le flot de conception, et réaliste.

1.2 Contributions

Cette thèse propose une méthodologie permettant l'évaluation de robustesse de systèmes complexes. La méthodologie apporte les contributions suivantes :

— Prise en compte de la spécification du système, permettant d'éviter la détection de fausses fautes critiques.

— Analyse multi-abstraction à différentes étapes du flot de conception, comme recommandée par les standards. Pour évaluer les mécanismes de robustesse implémentés à tous les niveaux.

— Analyse détaillée et quantitative des effets des bit-flips sur le système, information utile pour déterminer la probabilité d'occurence d'un évènement critique.

— Évaluation rapide du réalisme du modèle et des fautes de haut-niveau.

La méthodologie proposée exploite un flot d'analyse de sûreté contenant trois étapes :

— une étape de simulation de fautes de haut-niveau, permettant d'obtenir une première évaluation fonctionnelle de la robustesse du système, et d'extraire des propriétés de criticité bloc par bloc. Cette étape prend en compte la spécification du système.

— Une étape de simulation de fautes RTL, réutilisant les propriétés de criticité de l'étape de simulation haut-niveau. Cette étape permet d'obtenir une classification des fautes RTL.

— Une étape de co-simulation, permettant d'obtenir une évaluation de la qualité du modèle de haut-niveau sur lequel les résultats des deux premières étapes reposent.

Une approche itérative permettant d'exploiter ce flot est proposée. Elle permet d'évaluer rapidement le réalisme du modèle de haut-niveau utilisé, puis de faire une dernière évaluation quantitative précise de l'effet des bit-flips sur le système. La méthodologie proposée est appliquée à un cas d'étude réel dans le contexte aéronautique.

1.3 Organisation du manuscrit

Le manuscrit de thèse est divisée en six chapitres. La figure 1.2 nous permet de mettre en évidence les travaux effectués pendant la thèse et de présenter l'organisation du manuscrit. Elle présente en gris les contributions de la thèse : un flot d'analyse de sûreté qui prend en entrée un circuit, et permet d'obtenir en sortie une analyse de sûreté (AMDEC : Analyse des Modes de Défaillances, de leurs Effets, et de leur Criticité). Ce flot

est composé de trois étapes. En noir est représenté l'existant fourni par notre partenaire industriel : le cas d'étude composé du circuit et de son analyse AMDEC traditionnelle. Les différents chapitres sont ensuite encadrés avec leur couleur respective.

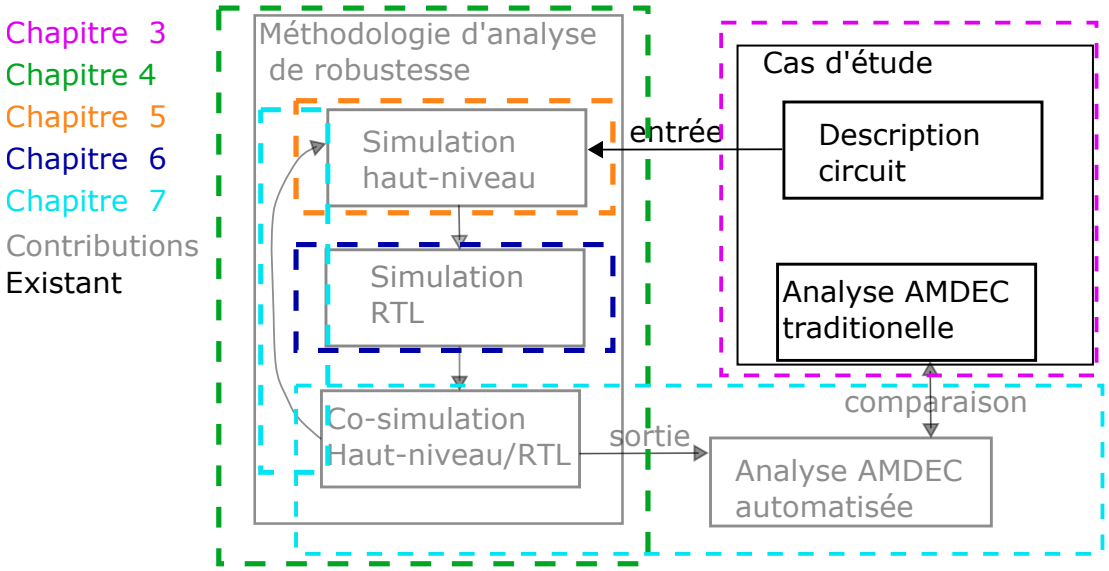

FIGURE 1.2 – *Vue générale des travaux présentés dans le manuscrit.*

Le chapitre 2 présente l'état de l'art. L'état de l'art présente le contexte de sûreté général et ses définitions, puis les généralités sur l'injection de fautes, et plus précisément de la simulation de fautes. Enfin, une partie est consacrée à la présentation de l'analyse de robustesse dans le contexte particulier de l'aéronautique.

Le chapitre 3 (violet) décrit le cas d'étude sur lequel nous avons appliqué la méthodologie proposée. Ce cas d'étude est un système de mesure d'une grandeur physique embarqué dans le contexte aéronautique. Il se compose d'une partie décrivant le circuit, et une seconde partie décrivant les analyses de sûreté effectuées sur ce cas d'étude.

Le chapitre 4 (vert) présente une vue d'ensemble de la méthodologie. Il présente une vue d'ensemble du flot et son approche itérative, puis présente dans trois parties distinctes, les trois étapes composant le flot de la méthodologie, puis montre comment ce flot d'analyse de sûreté proposé s'inscrit dans un flot de conception classique.

Le chapitre 5 (orange) présente la première étape du flot, qui est une étape de simulation de haut-niveau. Elle montre comment modéliser l'environnement de test et les fautes, puis comment extraire les résultats pour obtenir des propriétés utiles à la deuxième étape du flot. Les résultats de cette étape appliquée au cas d'étude sont présentés dans une dernière section.

Le chapitre 6 (bleu foncé) présente la deuxième étape du flot, qui est une étape de simulation RTL. Elle présente d'abord la théorie sur comment modéliser l'environnement de simulation, injecter des fautes grâce à l'injection statistique, et extraire les résultats pour obtenir une classification des fautes RTL. Les résultats de cette étape appliquée au cas d'étude sont présentés dans une dernière partie.

Le chapitre 7 (bleu clair) présente la troisième étape du flot, qui est une étape de co-simulation. Elle montre comment modéliser l'environnement de simulation et les fautes, et comment extraire les résultats pour obtenir une quantification de la qualité du modèle de haut-niveau initialement développé. Les résultats de cette étape appliquée au cas d'étude sont présentés dans une dernière partie. L'approche proposée étant itérative, les résultats

de deux autres itérations du flot appliquées au cas d'étude sont présentés. Ces itérations sont faites jusqu'à obtenir une classification précise des fautes RTL basée sur un modèle de haut-niveau dont le réalisme aura été prouvé.

Enfin, le chapitre 7.4.5 conclu ce manuscrit et propose des perspectives.

2

Etat de l'art

Sommaire

2.1 Contexte de sûreté général

Cette section introduit les différentes notions utiles à la compréhension du contexte de la sûreté de fonctionnement.

Un système se définit comme une entité pouvant interagir avec d'autres entités (autres systèmes, capteurs, humains, phénomènes naturels, etc.). L'ensemble de ces autres entités constitue l'environnement du système. Un système peut être caractérisé par plusieurs attributs :

— Fonction : ce que le système est supposé faire. Elle est donnée par la spécification fonctionnelle qui donne des exigences de fonctionnalités et de performances.

— Comportement : ce que fait le système pour effectuer sa fonction. Il est composé d'une suite d'états : calcul, communication, stockage de données.

— Structure : c'est l'implémentation qui permet au système d'avoir son comportement. Il est composé de composants liés entre eux, où chaque composant est lui-même un système.

— Service : c'est le comportement du système tel qu'il est perçu par son utilisateur.

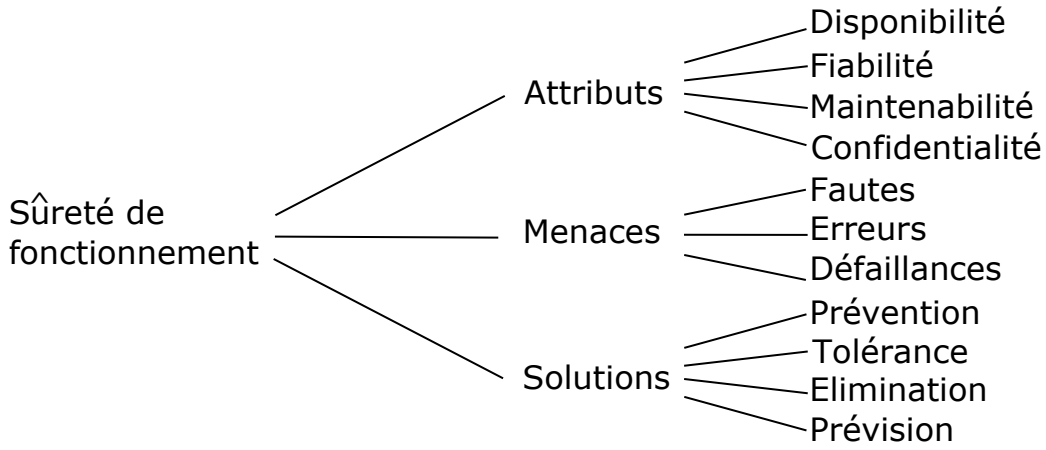

FIGURE 2.1 – *Notions autour de la sûreté de fonctionnement.*

La sûreté de fonctionnement est définie comme « La propriété qui permet de placer une confiance justifiée dans le service qu'il délivre » [12]. C'est la capacité d'un système à répondre à sa spécification. Elle est définie par des attributs, influencée par des causes. Des solutions permettent de l'améliorer. La figure 2.1 illustre les différentes notions utiles à la compréhension du contexte de sûreté. Nous détaillerons chacune de ces notions dans la suite.

Attributs Quatre attributs composent la sûreté de fonctionnement [12] :

— Disponibilité : capacité d'un système à fournir un service donné à un instant donné.

— Fiabilité : capacité d'un système a fournir un service pendant une certaine durée. Un système fiable est un système qui est capable de fournir un service sans défaillance pendant une période de temps relativement longue. Elle s'exprime sur un intervalle de temps.

— Maintenabilité : Capacité du système à détecter une défaillance et à être réparé.

— Confidentialité : capacité à maintenir la confidentialité et l'intégrité des données.

Menaces Un système peut ne pas délivrer la fonction pour laquelle il est fait. On appelle cette déviation une défaillance. Une défaillance résulte de la propagation d'une faute comme définit dans la figure 2.2 :

— Les fautes sont définies comme la modélisation d'un défaut physique, d'une imperfection, d'une perturbation qui a lieu dans un composant matériel ou logiciel. Des exemples de fautes matérielles sont des court-circuits ou des circuits ouverts dans un circuit. Dans du logiciel, une faute se manifeste par une opération qui s'exécute de façon incorrecte (saut d'une instruction par exemple).

— Une erreur est la manifestation d'une faute dans un composant du système. Elle résulte de la propagation d'une faute dans le système. Une mauvaise tension dans un circuit peut être le résultat d'un court-circuit par exemple.

— Enfin, la défaillance est l'impact de l'erreur sur le système, la défaillance a lieu lorsque le système ne répond plus à sa spécification. En reprenant notre exemple, un court-circuit ou circuit ouvert dans un composant peut entraîner une erreur de tension (erreur) dans le système qui empêche le seuil d'un thermostat de se déclencher (défaillance).

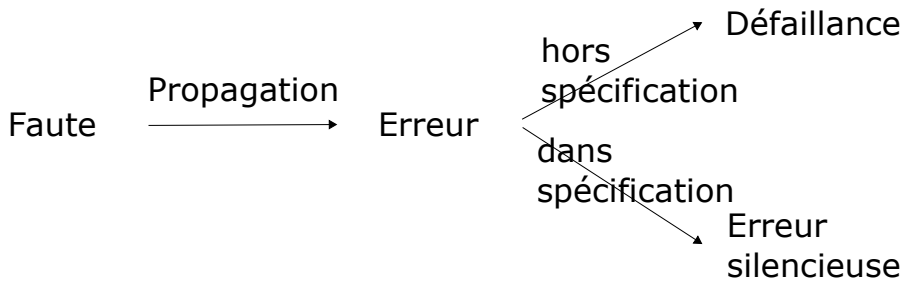

FIGURE 2.2 – *Effet de la propagation d'une faute.*

Solutions On distingue quatre catégories de solutions afin d'améliorer la sûreté de fonctionnement d'un système :

— La prévention de fautes permet d'éviter que des fautes arrivent dans le système grâce à des méthodes de développement et des techniques d'implémentation rigoureuses.

— La tolérance aux fautes permet d'éviter une défaillance en présence de fautes.

— Élimination des fautes permet de réduire le nombre et la gravité des fautes.

— Prévision des fautes permet une estimation qualitative ou quantitative du nombre de fautes et de leurs effets sur le système.

Dans la suite, nous nous concentrerons sur la tolérance aux fautes qui est la technique la plus efficace et répandue pour assurer la sûreté de fonctionnement d'un système.

2.2 Généralités sur l'injection de fautes

2.2.1 Modèles de fautes

Les fautes peuvent être classées en différentes catégories selon leur durée :

— Permanentes : Une faute permanente est due à un défaut physique dans le circuit. Elle est irréversible et a lieu en permanence. Son caractère permanent la rend plus facile à détecter. Les fautes peuvent être dues à une mauvaise fabrication (erreur de process), au vieillissement du circuit, ou à des dégradations irréversibles.

— Intermittentes : Les fautes intermittentes ont lieu de manière cycliques, elles sont généralement dues à des instabilités matérielles et peuvent être activées par des changements environnementaux (températures, champs électriques...).

— Temporaires : les fautes temporaires sont les plus difficiles à détecter en raison de leur caractère aléatoire. Elles ont lieu à cause de perturbations environnementales (particules), mais aussi à cause de l'interférence avec son milieu proche (alimentation, perturbations électromagnétiques, etc.).

Les techniques récentes ont permis de diminuer les fautes permanentes, et intermittentes. Ces erreurs sont plus faciles à détecter car elles ne sont pas aléatoires.

À l'inverse, la réduction des dimensions, la baisse des tensions d'alimentation, ainsi que l'augmentation des fréquences de fonctionnement, ont rendu les circuits de plus en plus vulnérables au bruit dû aux radiations, ou autres phénomènes physiques similaires. Ces phénomènes peuvent conduire à des niveaux d'énergie dans le circuit conduisant à des problèmes inattendus, et donc des fautes temporaires. La source de problème le plus fréquemment observé et le plus gros enjeu dans l'électronique moderne sont les bit-flips dans les cellules mémoires [13]. Les effets des radiations sur les circuits électroniques ont été identifiés depuis 1975 avec le premier rapport sur l'électronique spatiale [14]. En 1978, des résultats similaires ont été identifiés, cette fois ci au niveau de la mer [15]. Après que la mémoire ait été écrite, la valeur de certains bits a changé aléatoirement. Lorsque la lecture de la donnée a lieu, ces fautes peuvent se propager et devenir des erreurs. La mémoire n'est pas endommagée, et lors de la réécriture de la mémoire, des fautes peuvent avoir lieu sur des bits différents. Ces fautes au caractère temporaire et aléatoire sont appelées des « soft errors ». En 1978, il a été prédit que les particules secondaires créées par interactions des rayons cosmiques avec l'atmosphère peuvent entraîner des « soft errors » au niveau de la mer. La même année, des erreurs dues aux neutrons et protons ont été identifiées en laboratoire. Avec la progression des technologies vers des circuits de plus en plus petits, fonctionnant à fréquences de plus en plus hautes, contenant des mémoires de plus en plus grandes, et l'adoption croissante des technologies numériques dans les circuits critiques, la fréquence d'apparition des « soft errors » a augmenté pour en faire un des enjeux majeur aujourd'hui.

Les fautes causées par les radiations se classent en plusieurs catégories. La figure 2.3 illustre les différents types de fautes et leur hiérarchisation. Tout d'abord, les fautes dues aux radiations sont communément appelées « Single Event Effects » (SEE), et peuvent être divisées en deux catégories : les fautes à caractère transitoire appelées « soft errors » et les fautes causant des dommages permanents appelées « hard errors ». Les SEU (Single Event Upset) correspondent à un bit-flip dans une cellule mémoire (ou une flip-flop). Si plusieurs bits sont affectés, on parle de MCU (Multiple Cell Upset ou MBU pour Multiple Bit Upset). Si la particule affecte une partie combinatoire, on parle de SET (Single

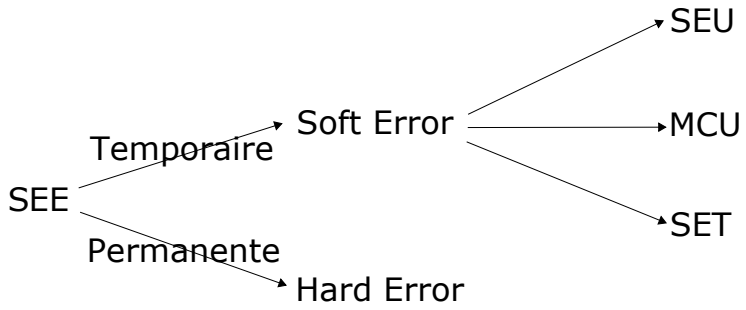

FIGURE 2.3 – *Les différents types de fautes.*

Event Transient). Ces fautes peuvent se propager à travers plusieurs chemins combinatoires jusqu'à des latchs ou des flip-flops. Ces phénomènes sont néanmoins plus rares que ceux induits par les SEUs, mais comme le montre Nicolaidis [16] dans ses travaux, les dimensions de plus en plus réduites et la hausse de la fréquence de fonctionnement en font des phénomènes de plus en plus probables.

2.2.2 Vue générale des techniques d'injections de fautes

Depuis la fin des années 90, l'injection de fautes s'avère être une technique efficace pour évaluer la sûreté des systèmes électroniques. Les approches existantes se classent en trois catégories :

— les techniques d'injection de fautes physiques s'appliquant sur les circuits déjà fabriqués,

— les techniques d'émulations qui consistent à injecter des fautes sur un support matériel reprogrammable,

— les techniques de simulations qui permettent une analyse plus tôt dans le flot de conception, du niveau transfert de registre au niveau transactionnel.

Nous nous concentrerons particulièrement sur les techniques de simulations, car ce sont les techniques qui correspondent le plus à l'analyse de sûreté multi-niveaux. Nous décrirons avant brièvement les techniques d'injection physique et d'émulation.

Injections physique Une approche est d'exposer le circuit une fois fabriqué à une source de perturbation physique comme des radiations ionisantes [17], un laser [18], ou à des interférences électromagnétiques [19]. Les solutions de radiations et d'interférences magnétiques ne permettent pas de cibler une partie du circuit. Le laser permet d'injecter des fautes à un endroit donné, pendant une durée choisie. Ces fautes sont représentatives de fautes transitoires et ne sont ni intrusives, ni destructives. Depuis les années 60, de nombreux travaux tentent de reproduire expérimentalement les effets des SEUs [20–22]. Les méthodes proposées par Samson [23] permettent des précisions spatiales réduites pour des fréquences assez élevées. Néanmoins, Duzellier [24] rapporte deux limitations : des bit-flips parasites et la taille du spot laser qui est de l'ordre de la cellule étudiée (difficile de cibler précisément une zone). Plusieurs autres méthodes ont été étudiées, que nous ne détaillerons pas ici.

L'injection physique est très réaliste, mais présente de nombreuses limitations : elle ne permet pas de cibler une partie précise du circuit, elle est onéreuse, et ne permet qu'un nombre limité d'événements. De plus, elle nécessite d'avoir le circuit fabriqué, ce qui en

fait une méthode utile pour la certification finale du circuit physique, mais pas en début de flot de conception.

Injection par émulation Une autre technique consiste à émuler le circuit sur une autre plateforme matériel, et d'injecter des fautes sur cette plateforme [25, 26]. Les capacités de reconfiguration partielle du FPGA peuvent être utilisées [27]. Les temps d'exécution peuvent être considérablement réduits par rapport aux techniques d'injections physiques et par rapport aux techniques de simulations au niveau porte . Comme le montre Civera [28], les temps sont divisés par 10 à 60, selon le nombre de vecteurs simulés. Néanmoins, l'émulation présente de nombreuses limites : les types de fautes injectées sont limités, il est difficile de cibler une partie du système et d'observer les effets sur le système.

2.3 Simulation de fautes

La simulation consiste à modéliser le système à évaluer, et de le simuler à l'aide d'outils de simulation. Les outils de simulations ont été développés initialement pour permettre de valider le fonctionnement d'un circuit, mais ils peuvent également être utilisés pour évaluer la sûreté de fonctionnement d'un système. Dans cette section nous allons d'abord présenter les langages utiles à la modélisation et à la simulation de SoC, puis nous ferons un état de l'art des techniques de simulation de fautes.

2.3.1 Niveaux de modélisation

On veut vérifier le fonctionnement du système en présence de fautes. Pour cela, la simulation peut être effectuée à plusieurs niveaux de modélisation selon le simulateur et le langage utilisé. Les langages de description de haut-niveau permettent de valider le fonctionnement du circuit au niveau fonctionnel. Les langages de descriptions plus bas niveau permettent de vérifier d'autres paramètres comme le timing, la consommation, *etc.* Le travail de cette thèse porte sur des systèmes embarqués sur FPGA, on cherche à évaluer la sûreté des parties matérielles. Les langages RTL (Verilog, VHDL) sont les langages de description matérielle standards utilisés dans l'industrie. De nombreux outils de simulations d'injection de fautes existent au niveau RTL. Néanmoins, ces langages ne permettent pas de modéliser facilement des systèmes à des haut-niveaux de modélisation. Plusieurs langages ont donc été proposés afin de pouvoir décrire à plus haut-niveau les systèmes. SystemVerilog [29] propose une extension des langages matériels avec des concepts logiciels supplémentaires. D'autres langages, à l'inverse, intègrent des fonctionnalités matérielles à des langages logiciels [30, 31]. SystemC [32] est le langage de description standard utilisé dans l'industrie pour permettre de spécifier des SoC complexes [33]. Les SoC sont des systèmes comprenant une partie matérielle en interaction avec une partie logicielle. Le langage SystemC est en fait une bibliothèque C++ qui permet de modéliser le système à plusieurs niveaux d'abstraction, et permet de co-simuler (modéliser des blocs à différents niveaux). Son intérêt et son efficacité pour explorer les différentes architectures possibles a ont été démontrés depuis de nombreuses années [34, 35]. Lors du

développement d'un SoC, le système est d'abord décrit de manière algorithmique. Le système peut être partitionné en fonctions, qui communiquent entre elles par des canaux de communication.

TABLEAU 2.1 – *Les différents niveaux de modélisation*

	Temps	Communication	Blocs de calcul	Performances simulation
RTL	cycle	bit	Registre	lente, précise
BCA	cycle	Transactionnel	Fonctionnel	lente, assez précise
TLM AT	approximé	Transactionnel	Fonctionnel	rapide, peu précise
TLM	non modélisé	Transactionnel	Fonctionnel	rapide, peu précise

Les niveaux de modélisations sont illustrés dans le tableau 2.1. La première colonne donne les différents niveaux de modélisation. La seconde colonne exprime comment le temps est modélisé. La troisième détaille l'implémentation de la communication, la quatrième des blocs de calculs, et enfin la dernière colonne donne les performances en simulation.

— TLM : C'est un des deux niveaux de modélisation définit par la librairie TLM [36, 37]. Les calculs sont modélisés par des algorithmes, la communication à l'aide de modules de communication.

— TLM AT : C'est un des deux niveaux de modélisation définit par la librairie TLM. Les blocs de calcul et de communication sont modélisés avec du temps approximé. Les blocs de calculs sont modélisés avec des algorithmes comme au niveau BCA, et des mécanismes basés sur les évènements sont utilisés pour la communication (FIFO et interfaces non bloquantes).

— BCA : les interfaces de communications sont implémentés avec des horloges. Les fonctions sont exprimées grâce à des algorithmes et le temps est approximé grâce à l'usage de la fonction *wait()*.

— RTL : c'est le niveau de modélisation classique pour les systèmes matériels. Le calcul et la communication sont communément représentés avec des machines à états. Ils sont synchronisés par une horloge. L'état des éléments mémoire sont connus pour chaque cycle d'horloge.

2.3.2 Techniques de simulation de fautes RTL et TLM

Dans cette section, nous allons décrire les différentes techniques de simulation d'injections de fautes de la littérature effectuées au niveau RTL et TLM. Nous décrirons d'abord les généralités et les différentes techniques développées pour la simulation de fautes RTL [38, 39], puis nous parlerons des techniques d'injections de fautes à plus haut-niveau d'abstraction.

Généralités On peut séparer les techniques d'injections de fautes en deux catégories :

— Avec modification de code : à l'aide de mutants ou saboteurs.

— Avec utilisation des commandes simulateur.

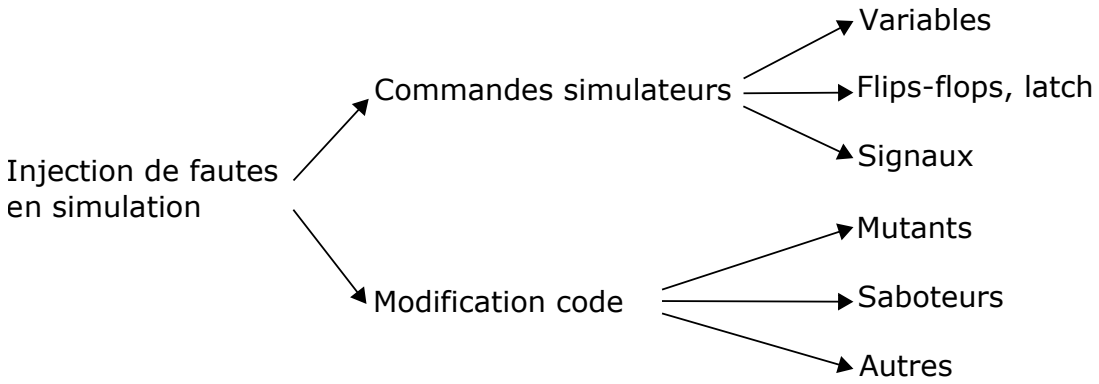

FIGURE 2.4 – *Techniques d'injection de fautes par simulation.*

Commandes simulateur Les commandes simulateur sont des fonctionnalités implémentées dans les outils de simulation. Elles peuvent permettre d'injecter des fautes à différents endroits selon les outils de simulation et leurs commandes disponibles. La faisabilité de cette technique dépend des fonctionnalités offertes par les commandes du simulateur. Par exemple, la figure 2.5donne un exemple du pseudo-code qui peut permettre d'injecter une faute de collage temporaire. Pour des fautes permanentes, il suffit de supprimer l'étape 4, pour une faute intermittente, il faut répéter les étapes 1 à 5.

```
[1] Simuler jusqu'à [moment de l'injection]
[2] Forcer signal [signal1][valeur]
[3] Simuler pour [temps de la faute]
[4] Ne plus forcer signal [signal1]
[5] Simuler pour [temps simulation]
```

FIGURE 2.5 – *Manipulation de signal pour une faute de collage.*

L'outil MEFISTO [40] a été le premier outil d'injection qui a appliqué cette technique. L'utilisation de commandes de simulation est la technique la plus simple à implémenter comme elle ne requiert pas la modification du code. Cela évite donc d'avoir à recompiler le code. Néanmoins, les fautes possibles à injecter sont limitées et cela dépend beaucoup des fonctionnalités du simulateur utilisé.

Modification de code Dans cette partie nous allons décrire les solutions standards d'injection avec modification de code. Les solutions sont infinies, mais on peut distinguer deux principales catégories de modification de code, illustrées par les figures 2.6, 2.7 et 2.8 :

— La première consiste à ajouter un module d'injection de fautes agissant comme un saboteur.

— La seconde solution consiste à modifier le comportement d'un module déjà existant ce qui génère un composant modifié appelé mutant.

Comme illustré dans la figure 2.7, un saboteur est un composant spécial S ajouté au modèle original correspondant à la figure 2.6 [41]. Ce composant doit modifier une valeur, ou le timing d'un signal *sig* lorsque la faute est injectée. On obtient alors un signal

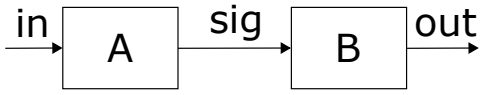

FIGURE 2.6 – *Schéma de référence.*

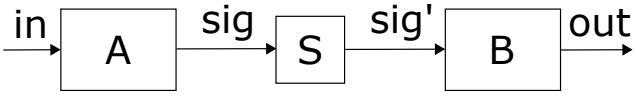

FIGURE 2.7 – *Schéma illustrant un saboteur.*

corrompu sig'. Plusieurs outils, implémentant une extension de l'outil MEFISTO ont implémenté ces techniques : MEFISTO-L [42] qui implémente les saboteurs et de nouvelles fonctionnalités et MEFISTO-C qui permet en plus d'accélérer les temps de simulations.

Comme illustré dans la figure 2.8, un mutant est un composant A' qui remplace un autre composant A. Il se comporte à l'identique en fonctionnement normal. Lorsqu'il est activé par un signal de controle ctl il se comporte comme le composant en présence de faute. Il existe un nombre infini de mutants possibles pour chaque composant, mais plusieurs papiers [43,44] définissent des sets de modèles de fautes correspondants à des comportements réalistes. Le principal problème des mutants est le coût en temps de simulation qui apparaît lorsqu'il faut changer d'architecture. En effet, dans le cas des fautes temporaires, il faut pouvoir commuter entre les différentes architectures possibles. L'avantage est que cela offre de très nombreuses possibilités de fautes modélisables, indépendamment de l'outil de simulation utilisé.

Enfin, pour trouver un compromis entre les différentes techniques d'injection de fautes, d'autres techniques [45, 46] existent proposant d'étendre le langage VHDL avec de nouveaux types de données et de signaux.

Les analyses de sûretés effectuées au niveau porte ou sur des modèles RTL apparaissent tard dans le flot de conception. La découverte de défaillances implique de faire des modifications coûteuses dans le design d'un SoC. Dans la fin des années 2000, les travaux ont commencé à s'intéresser à l'injection de fautes sur des modèles de plus haut-niveau, afin de pallier les problèmes de temps de simulation qui apparaissaient aux niveaux portes et registres. En plus d'accélérer les simulations, cela permet d'évaluer tôt dans le flot de conception la sûreté des circuits. Néanmoins, la modélisation de haut-niveau implique des enjeux de réalisme que nous discuterons dans la prochaine section.

2.3.3 Réalisme des fautes de haut-niveau

La modélisation des systèmes à plus haut-niveau d'abstraction implique d'enlever des niveaux de détails (horloge, types de données, etc.). Même si fonctionnellement et en l'absence de fautes le système se comporte de la même manière, la modélisation de haut-niveau entraîne les questions de réalisme des fautes, et de leur propagation à travers le

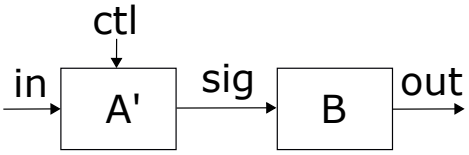

FIGURE 2.8 – *Schéma illustrant un mutant.*

système. Dans cette partie, nous discuterons des différents travaux qui traitent du réalisme de la simulation de fautes à des niveaux d'abstraction plus élevé que le RTL.

Plusieurs travaux se sont intéressés au réalisme des fautes injectées à des niveaux de modélisation plus haut, mais aussi à leur exhaustivité.

Évaluation quantitative des méthodes de simulation de fautes dans un microprocesseur Cho et *al.* [47] propose une évaluation quantitative des méthodes d'injection de fautes dans des microprocesseurs. Pour cela, des fautes sont injectées à plusieurs niveaux : au niveau RTL, dans les registres visibles du microprocesseur, puis dans les variables du logiciel. De plus, ce papier étudie également comment les fautes injectées dans les flip-flops se propagent dans les différentes parties du microprocesseur (registres, mémoire caches, mémoire). Bien que les fautes ne soient pas injectées au même endroit dans les trois cas, l'étude permet de mettre en évidence plusieurs points :

— Une faute injectée sur une flip-flop peut conduire à plusieurs fautes se propageant dans le système sur les registres ou la mémoire.

— Certaines fautes injectées à bas niveau ne se propagent pas dans le système (masquées).

Méthode d'injection de fautes dans un Soc en SystemC et étude du réalisme Des premiers travaux ont proposé des techniques pour la modélisation de fautes TLM [48–50], mais ces approches sont incomplètes et ne vérifient pas le réalisme des fautes injectées. Miele dans ses travaux [51, 52] propose une méthodologie permettant d'injecter des fautes dans des SoCs, modélisés en SystemC TLM. Il utilise une plateforme de simulation *ReSp* [53] avec des fonctionnalités étendues pour l'analyse et la simulation de fautes au niveau TLM. Il étudie ensuite le réalisme des fautes injectées en comparant avec le modèle du circuit au niveau RTL, et en regardant comment les fautes (seulement les soft errors) se propagent à plus haut-niveau. Sur les 1798 fautes RTL possibles (single bits), seules 30 n'ont pas de fautes correspondantes. À l'inverse, sur les 23 fautes définies sur le modèle de haut-niveau, 2 ne correspondent à rien de physique. Miele met donc en évidence des enjeux de réalisme et d'exhaustivité soulevés par l'injection de fautes à haut-niveau d'abstraction.

Correspondance entre fautes RTL et TLM basée sur une approche formelle Herdt [54] étudie la correspondance entre les fautes RTL et TLM grâce à une approche formelle. Pour cela, il injecte des fautes sur un modèle RTL, et compare la sortie obtenue avec un modèle de référence. Lorsque la sortie diffère de celle de référence, un couple entrée/sortie est obtenu et permet ensuite de faire la recherche d'équivalence au niveau TLM. Un modèle du système TLM est développé, avec des extensions permettant d'injecter des fautes TLM dans le système. Une recherche des fautes pouvant créer des comportements fautifs équivalents à la faute injectée au niveau RTL est effectuée, en utilisant une propriété formelle écrite à partir des couples entrée/sortie extraits lors de la simulation RTL. Pour chaque propriété formelle, donc pour chaque couple, une recherche d'équivalence au niveau TLM est effectuée. Cela permet d'avoir un set de « candidats ». Chaque nouveau couple entrée/sortie erroné permet alors de rajouter des candidats ou d'en supprimer.

Ces travaux ont permis de mettre en évidence que certaines fautes RTL ne sont pas modélisables à plus haut-niveau, car elles peuvent entraîner plusieurs fautes TLM (le cas

des fautes multiples n'est pas pris en charge dans l'étude), ou parce qu'aucune combinaison de paramètres n'a permis de modéliser le même comportement fautif.

Résumé Ces derniers travaux permettent de soulever deux problématiques principales quant à la simulation de fautes au niveau TLM :

— Il n'y a pas de réelle correspondance entre une faute RTL et une faute TLM. En effet, certaines fautes RTL se propagent en créant des fautes à plusieurs endroits dans le système. D'autres peuvent ne pas être visibles à plus haut-niveau.

— Il existe une infinité de fautes TLM modélisables, et une grande part des fautes TLM ne représentent ne sont pas physiquement réalistes.

Dans le cas de l'étude de la sûreté de fonctionnement, si des fautes RTL ne sont pas visibles à plus haut-niveau, on peut supposer qu'elles ne créeront pas de comportements critiques (fautes masquées). En revanche, si elles se propagent dans le système en créant plusieurs fautes TLM et que cela n'est pas modélisé, cela peut cacher des comportements fautifs qui sont probables (puisque créés par un seul bit-flip), la part de probabilité dépendant de chaque circuit, et de la granularité (niveau de détail) du modèle TLM. Néanmoins, les cas où cela génère des fautes multiples à plus haut-niveau représentent une faible proportion des fautes, ce qui rend l'analyse de haut-niveau encore pertinente. Pour répondre à ces enjeux de réalisme, il existe plusieurs techniques comme la co-simulation ou la simulation multi-abstraction. Cela permet en plus de pouvoir prendre des contre-mesures à plusieurs niveaux (choix architecturaux, implémentation fonctionnelle, *etc.*).

2.3.4 Solutions multi-abstractions.

Dans cette section nous détaillerons plusieurs techniques d'analyse de sûreté multi-abstractions de la littérature.

Mueller [55] propose une approche de multi-abstraction en injectant des fautes au niveau RTL, et au niveau comportemental. Le principe de l'approche est de simuler des scénarios courts lors de la phase d'injection RTL, et des scénarios plus longs au niveau architectural. L'approche permet de faire des plus grandes campagnes d'injection de fautes sur des systèmes complexes, et de faire des choix d'implémentation à plusieurs niveaux de modélisation du système. Néanmoins, le réalisme des fautes injectées à plus haut-niveau n'est pas vérifié.

Le projet CLERECO [56] propose une évaluation de la robustesse multi-abstraction basée sur l'analyse séparée des trois couches du système composant le microprocesseur afin de simplifier les analyses. Néanmoins, cette méthode ne s'applique que sur des microprocesseurs et n'est pas adaptée à des systèmes complexes.

Perez [57] dans le contexte des systèmes critiques, cherche à concevoir un circuit en accord avec la norme IEC-61508 (norme pour les systèmes critiques). Cette norme suggère de faire des injections de fautes à plusieurs niveaux de modélisation pour évaluer la robustesse du circuit. Sur une étude de cas d'un système de mesure d'odométrie (mesure permettant d'estimer la position d'un véhicule en mouvement) dans un train (contexte critique). Des injections de fautes sont effectuées à trois niveaux de modélisation différents : comportementale, architecturale, et implémentation système. L'étude permet de mettre en évidence l'intérêt d'injecter des fautes à plusieurs niveaux d'abstractions en permettant d'identifier des faiblesses qui peuvent être corrigées à différents niveaux de modélisation. Identifier ces faiblesses tôt permet un gain de temps d'implémentation et

une diminution des coûts. Néanmoins, comme le souligne Perez, cela ne permet pas d'assurer la sûreté de fonctionnement du système devant les autorités de certifications et les tests réels restent indispensables. En effet, les fautes injectées le sont à des niveaux de modélisation abstraits, où la correspondance avec des comportements physiques n'est pas définie.

Mariani [58] propose une approche beaucoup plus bas niveau permettant d'obtenir les données nécessaires pour établir une AMDEC. Sa méthode a été utilisée dans l'industrie pour prouver la sûreté de fonctionnement de systèmes critiques. Néanmoins, l'approche de Mariani est implémentée au niveau porte et RTL, et les temps de simulations peuvent être très longs. Elle ne permet pas de pouvoir évaluer tôt dans le flot de conception la robustesse d'un circuit, ni de modéliser des SoCs pouvant être composés de systèmes mixtes (microcontrôleurs, mémoire, logique combinatoire, etc.).

Plus récemment, Tabacaru [59] propose une méthodologie permettant de faire de l'injection de fautes à haut-niveau d'abstraction, en utilisant des modèles de fautes réalistes. Son approche se base sur la génération d'un modèle SystemC à partir d'une netlist. Une campagne d'injection de faute est effectuée dans les éléments séquentiels de ce modèle, et seules les fautes se propageant dans le système sont retenues. Ces fautes jugées comme réalistes constituent ensuite une bibliothèque de fautes réalistes. Elles sont ensuite réutilisées sur le prototype virtuel et plusieurs itérations sont faites afin d'implémenter les mécanismes nécessaires à la robustesse du circuit. Cette approche est intéressante, mais elle nécessite d'avoir un modèle synthétisable afin de pouvoir dérouler cette méthodologie. De plus, la validation de fonctionnement du modèle s'effectue par comparaison du modèle sans faute et du modèle avec injection de fautes. Cette méthode ne prend donc pas en compte l'environnement du système et sa spécification, ce qui peut conduire à l'identification de faux comportements critiques.

Afin de trouver le meilleur compromis entre temps de simulation et réalisme, et en accord avec les recommandations des normes, plusieurs méthodes proposent des solutions de simulation de fautes multi-abstraction afin de pouvoir évaluer la sûreté des systèmes. Le tableau 2.2 résume les différentes techniques d'analyse de sûreté multi-abstractions présentées précédemment. La plupart des méthodes s'appliquent au cas particulier des microprocesseurs et ne propose pas d'évaluation quantitative utile à la certification des systèmes critiques. Perez propose une analyse quantitative mais l'analyse nécessite un modèle synthétisable. Aucun ne semble prendre en compte les contraintes du système données par la spécification, ce qui peut entraîner l'identification de faux comportements critiques. La méthodologie SafeAir vise alors à proposer une solution de simulation multi-abstraction afin d'évaluer la robustesse d'un système complexe, tout en prenant en compte l'environnement du système et sa spécification.

TABLEAU 2.2 – *Résumé des différents papiers de simulation multi-abstraction*

	Système étudié	Niveau d'analyse	Spécification	Avantages	Inconvénients
Mueller [55]	Hardware	RTL/ comportemental	Non prise en compte	Simulation de plusieurs scénarios différents	Réalisme non vérifié
CLERECO [56]	Microprocesseur	Physique / hardware/ softaware	Non prise en compte	Analyse multi-abstraction	Non adapté à des systèmes complexes
Perez [57]	Mixte	comportemental/ architectural/ transactionnel	Non prise en compte	Analyse tôt dans le flot	Analyse très haut-niveau, réalisme non vérifié
Mariani [58]	Hardware	RTL/portes	Non prise en compte	Analyse réaliste, utiles pour les analyses AMDEC	Nécessite un modèle synthétisable
Tabacaru [59]	Mixte	Transactionnel	Non prise en compte	Modèles faute haut-niveau réaliste	Nécessite un modèle synthétisable

2.4 Présentation générale de l'analyse de robustesse dans le contexte aéronautique

2.4.1 Sûreté, Intégrité et Sécurité dans le contexte aéronautique

Dans le contexte aéronautique, les définitions sont données par la norme DO254 [11]. Elles varient légèrement par rapport à la littérature scientifique :

— La sécurité est l'état dans lequel le risque est inférieur à la limite supérieure du risque acceptable. Elle est spécifique d'un processus technique ou d'un état. Le risque est défini par la probabilité d'occurrence ainsi que les dommages et blessures attendus.

— La sûreté est la prévention d'actes illicites à l'encontre de l'aviation civile.

— L'intégrité indique la capacité d'un article indiquant que l'on peut lui faire confiance pour la réalisation de la fonction attendue.

Sécurité Le niveau de sécurité idéal est lorsqu'on peut garantir aucun accident quel que soit l'utilisation de l'article. En pratique, la sécurité absolue n'existe pas car :

— C'est économiquement impossible : les moyens de développement son nécessairement finis d'un point de vue financier et temporel.

— Il est impossible de connaître la totalité de l'environnement et des évènements possibles.

Il est alors nécessaire de fixer un compromis acceptable entre le service rendu et les risques ressentis. La gestion du risque est liée à son acceptation. Par exemple, il y a un million de plus de chances de mourir prématurément en étant fumeur qu'en effectuant un vol Paris-New-York, pourtant l'acceptabilité est plus grande [60].

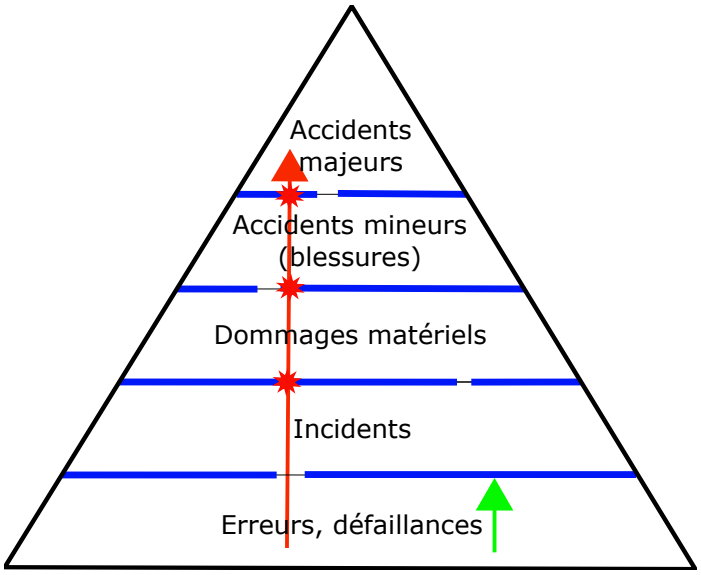

FIGURE 2.9 – *Pyramide des risques.*

La sécurité dans l'aéronautique s'est construite à partir de l'expérience. Les accidents résultent de combinaisons multiples : défaillances, erreurs de conceptions, *etc.* La figure 2.9 illustre la pyramide des risques de Heinrich-Bird et le modèle de Reason. Plus la gravité est grande (vers le haut de la pyramide), plus la probabilité d'occurence est faible. En bleues sont représentées les barrières correspondant au modèle de Reason. Les barrières sont faillibles, et la flèche rouge montre qu'un accident majeur est possible uniquement en cas de défaillances multiples. S'il n'y a pas de faille dans les barrières, une erreur ne se propage pas. Le risque est caractérisé par la probabilité d'occurrence et la gravité. En aéronautique la probabilité d'occurrence est exprimée en probabilité moyenne par heure de vol. La gravité est divisée en classes en fonction de l'impact des évènements. Les principaux moyens d'atteindre un niveau de sécurité requis sont :

— La prévention : conception dans le but d'éviter les fautes et défaillances. Elle se base sur des statistiques prévisionnelles.

— La détection : détecter les défaillances et en informer l'équipage. Des contrôles ont lieu durant toute la vie de l'équipement.

— La limitation des conséquences : éviter la propagation des défaillances et en limiter les effets.

Les trois principales sources de risques sont :

— Les erreurs humaines en conception ou en exploitation.

— Les évènements extrinsèques à l'avion qui sont de la responsabilité de l'avionneur : opérations aériennes, impacts aviaires, environnement *etc.*

— Les défaillances : les méthodes d'analyses quantitatives et qualitatives visent à limiter ces risques.

Afin de maîtriser ces sources de risques, un niveau d'assurance de développement (DAL : Development Assurance Level) est attribué à chaque niveau de gravité. Les normes définissent des processus associés à chaque DAL.

2.4.2 Normes

Les différentes normes de l'aéronautique proposent une classification des évènements redoutés, illustrée par le tableau 2.3. Ce tableau représente les différentes classifications possibles selon les normes, en fonction de la gravité des évènements, et de leur probabilité d'occurrence. Les autorités de certification sont chargées de réglementer et de certifier les équipements de l'aviation civile. La FAA (Federal Aviation Administration) est l'agence chargée des contrôles dans l'aviation civile aux Etats-Unis, l'EASA (European Union Aviation Sagety Agency) est l'homologue dans l'Union Européenne.

TABLEAU 2.3 – *Classification des évènements redoutés*

<table>
<tr><td rowspan="2">Gravité</td><td>État de défaillance</td><td colspan="3">Pas de défaillance</td><td>Mineur</td><td>Majeur</td><td>Dangereux</td><td>Catastrophique</td></tr>
<tr><td>Effets</td><td colspan="2">Normal</td><td>Nuisances</td><td>Opérationnels</td><td>Blessures</td><td>Blessures passagers</td><td>Fatal</td></tr>
<tr><td rowspan="3">Contraintes quantitatives</td><td>Probabilité</td><td>10^{-0}</td><td>10^{-1}</td><td>10^{-2}</td><td>10^{-3}</td><td>10^{-4}</td><td>10^{-5}</td><td>10^{-6}</td></tr>
<tr><td>FAA</td><td colspan="4">Probable</td><td colspan="2">Improbable</td><td>Extrêmement improbable</td></tr>
<tr><td>EASA</td><td colspan="3">Fréquent</td><td>Probable</td><td>Faible</td><td>Très faible</td><td>Extrêmement improbable</td></tr>
<tr><td rowspan="3">Contraintes qualitatives</td><td>ARP4754-A</td><td colspan="3">FDAL E</td><td>FDAL D</td><td>FDAL C</td><td>FDAL B</td><td>FDAL A</td></tr>
<tr><td>DO-178C</td><td colspan="4">IDAL D</td><td>IDAL C</td><td>IDAL B</td><td>IDAL A</td></tr>
<tr><td>DO-254</td><td colspan="3">IDAL E</td><td>IDAL D</td><td>IDAL C</td><td>IDAL B</td><td>IDAL A</td></tr>
</table>

Plusieurs normes définissent les procédures de développement et d'analyses de sécurité à suivre lors du développement d'un avion. Elles sont illustrées dans la figure 2.10 :

— L'ARP4761 [61] définit les méthodes d'analyses de sécurité qui permettent d'obtenir des informations sur les conditions de sûreté et de défaillances.

— L'ARP4754 [10] définit la méthode de développement système, elle s'utilise conjointement avec la norme ARP4761, et est complétée par d'autres normes (décrites dans la suite). Elle consiste à définir des niveaux de criticités (DAL) qui donnent des contraintes de développement et de certifications correspondantes.

— La DO254 [11] définit une méthode de développement matériel pour chaque niveau de criticité la composant. L'accent est mis sur les aspects vérification et validation.

— La DO178 [62] définit une méthode de développement logiciel pour chaque niveau de criticité la composant. C'est l'équivalent au niveau logiciel de la norme DO254.

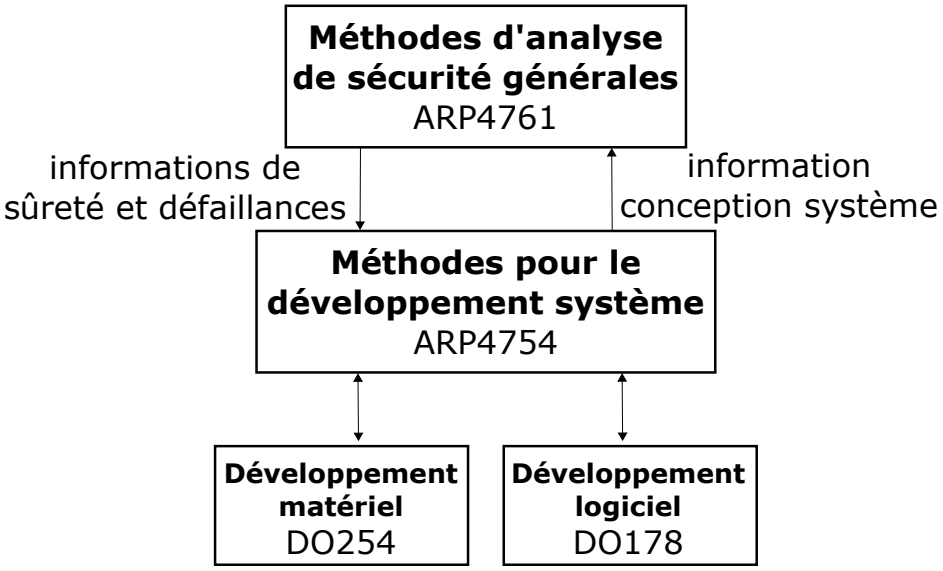

FIGURE 2.10 – *Normes de l'aéronautique.*

2.4.3 Méthodes d'analyse de sûreté

Les méthodes les plus utilisées dans l'aéronautique et recommandées par l'ARP4761 sont :

— L'analyse des modes communs consiste à identifier les défaillances simples entraînant des risques catastrophiques.

— L'arbre de défaillance qui permet de modéliser les différents scénarios de risques.

— L'analyse des modes de défaillances, de leur effet et de leur criticité (AMDEC) qui permet d'analyser les conséquences des défaillances.

— L'analyse des effets des changements d'états logiques (SEU) provoqués par les rayonnements ionisants.

La fiabilité est exprimée en MTBF (Mean Time Before Failure = temps moyen entre défaillances). Afin de la calculer, plusieurs outils sont utilisés :

Arbre de défaillance L'arbre de défaillance est un outil booléen, qui combine tous les scénarios pouvant conduire à une défaillance critique afin d'arriver à une probabilité totale d'occurrence d'un évènement. La figure 2.11 illustre un arbre de défaillance. La probabilité totale d'occurrence d'un évènement critique est calculé à partir des probabilités des événements ($P1$, $P2$, $P3$ et $P4$) qui conduisent au comportement critique redouté. La probabilité de l'évenement critique redouté est $Ptot$. Par exemple, une perte du watchdog apparaissant en même temps qu'un problème dans l'exécution software peut être considérer comme un évènement critique. Dans ce cas, les probabilités de perte du watchdog et d'erreur d'exécution software sont multipliées pour obtenir la probabilité de l'évènement critique.

AMDEC L'AMDEC est une analyse qui permet d'étudier l'effet de la défaillance d'un composant sur le système, elle se déroule en plusieurs étapes :

— Détermination des défaillances possibles, et de leur probabilité d'occurrence

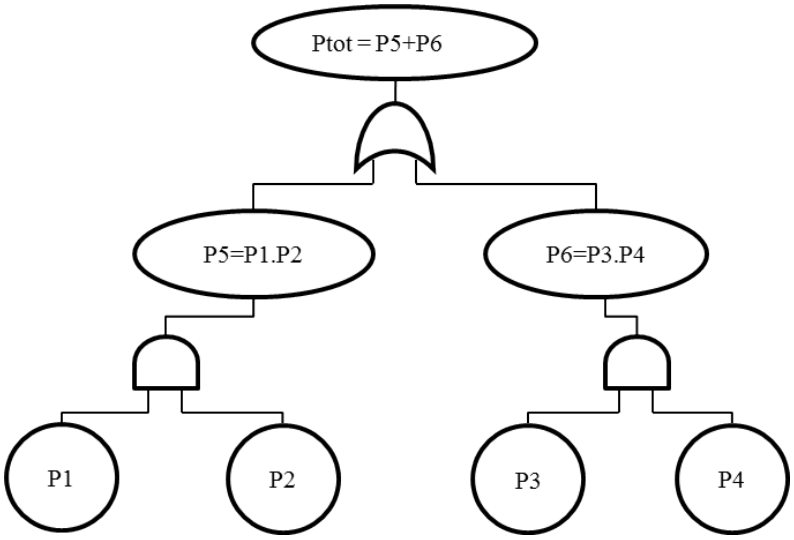

FIGURE 2.11 – *Arbre de défaillance.*

— Étude de l'effet sur le bloc de la défaillance

— Étude de l'effet sur le système de la défaillance, et calcul de la probabilité d'occurrence

Analyse des SEUs Une étude des SEUs est également effectuée. Une première analyse qualitative en identifiant les composants potentiellement sensibles et en mettant en place les protections adaptées (codes correcteurs d'erreurs, redondances, *etc.*). Ensuite une seconde analyse quantitative selon les étapes suivantes :

— Taux de particules par unité de temps [9].

— Nombre de particules par sous-bloc du système, en fonction de la taille de ces sous blocs (mémoire SRAM, registres, éléments logiques, *etc.*)

— Une AMDEC est ensuite effectuée, elle est empirique et pour diminuer les risques elle suppose qu'une grande partie des SEUs conduit à des comportements erronés critiques.

L'ARP4761 ne recommande pas explicitement l'étude des SEUs car très ancienne. En pratique, les autorités demandent qu'ils soient pris en compte dans la conception [63] en raison de l'augmentation de la sensibilité des composants due à la réduction des dimensions.

Outils Afin d'automatiser ces analyses de sûreté, plusieurs outils ont déjà été développés à différents niveaux de modélisation. Les techniques proposant des techniques d'automatisation des analyses AMDEC sont nombreuses [64–66]. La plupart ne font que des analyses très haut-niveau à partir d'informations déjà collectées (taux de défaillances de composants, *etc.*). D'autres [67] proposent des analyses des FPGAs, mais pas une analyse de l'effet des SEUs détaillée. Dans le contexte aéronautique, Altarica est un langage de modélisation de haut-niveau dédié à l'analyse de sûreté d'un système dans son ensemble. Dans le milieu de l'aéronautique ce langage permet de modéliser un système à l'échelle de l'avion. La première version de ce langage a été créée au LaBRI à Bordeaux à la fin des années 90 [68, 69]. Cet outil permet d'aider les ingénieurs de sûreté à établir les modèles

classiques d'études de risques (arbre de défaillances, AMDEC *etc.*). Les modèles AltaRica sont faits de composants hiérarchiques réutilisables. Lorsqu'une partie du système est modifiée, il n'est pas nécessaire de reprendre depuis le début une analyse de sûreté. Cet outil a déjà atteint une maturité industrielle car il a été utilisé pour certifier le système de commande de vol du Falcon 7X [70]. Néanmoins, cet outil d'analyse permet une analyse beaucoup plus haut-niveau que celui permettant d'étudier un SoC. Il permet d'étudier la propagation des défaillances dans un système, mais ne permet pas d'étudier des systèmes matériels. Plus proche de ce que nous cherchons à faire, Mariani [58] propose une solution permettant d'obtenir les données nécessaires à l'analyse de sûreté de fonctionnement en accord avec la norme IEC61508. Néanmoins, cette approche reste au niveau RTL et ne permet pas de modéliser les interactions avec le système. Cet outil a également fait ses preuves dans l'industrie. Nos travaux s'intéressent plus particulièrement à l'analyse des SEUs, pour lesquelles des hypothèses très pessimistes sur leur effet sont habituellement prises. La méthodologie se place au niveau du développement de composants programmables inclus dans un SoC. Elle veut étudier la sûreté de fonctionnement d'un système matériel, et ses interactions avec le système à l'échelle du SoC, c'est-à-dire avec le logiciel du micro-contrôleur, et les autres composants avec lesquels il interagit (mémoires par exemple).

3

Cas d'étude : Système de mesure embarqué dans le contexte aéronautique

Sommaire

Dans la suite de ce manuscrit, nous allons illustrer les différentes étapes de notre méthodologie à l'aide d'un exemple fourni par notre partenaire industriel THALES. Ce cas d'étude se place dans le contexte des systèmes critiques et plus précisément dans le contexte aéronautique. Dans une première partie (3.1), nous présenterons le cas d'étude. Dans la section suivante (3.2), nous détaillerons l'analyse de sûreté telle qu'elle est faite par les ingénieurs de THALES.

3.1 Présentation du circuit

Le circuit que nous présentons dans ce chapitre est confidentiel. Afin de pouvoir tout de même présenter les résultats que nous avons obtenus, une grandeur physique et les noms des signaux associés à cette grandeur ont été obfusqués. Leurs valeurs sont normalisées, et l'unité de mesure de cette grandeur physique est appelée U. Le système que nous étudions, SYS_X, est embarqué à bord des avions (voir figure 3.1). Il est chargé de mesurer numériquement une grandeur physique X_φ. Ce système est tripliqué, et les données des trois systèmes sont envoyées à une unité de contrôle ECU (Electronic Control Unit). Cette unité recueille les différentes données, en déduit des informations de vol et envoie ces informations au tableau de bord.

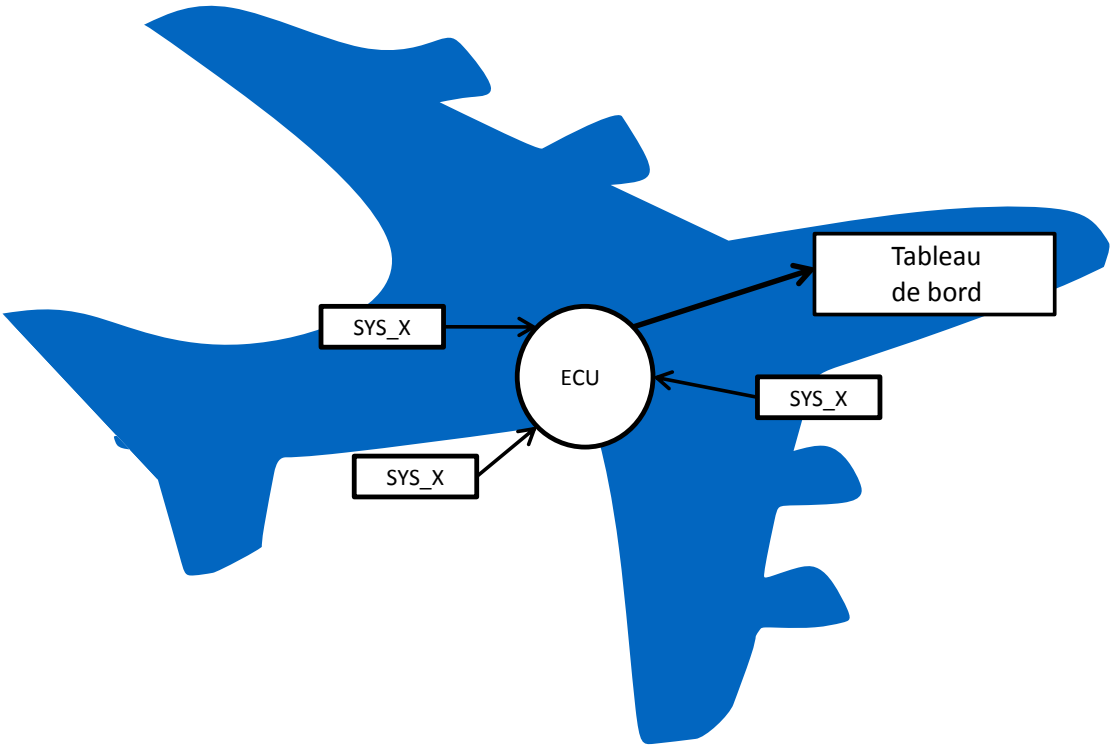

FIGURE 3.1 – *Intégration du système au sein de l'avion.*

3.1.1 Architecture du circuit

Interface du système Le système embarqué SYS_*X* prend en entrée les impulsions provenant de deux capteurs C_X et C_T. Le capteur C_X génère un signal oscillant Sig_X. La fréquence de ce signal Sig_X est F_X. Elle est liée à la grandeur physique X_φ. Ce capteur est sensible à la température. Le second capteur C_T permet donc de mesurer la température T. Il fonctionne de la même manière que C_X et génère une impulsion Sig_T de fréquence F_T.

Architecture globale Le système étudié, illustré dans la figure 3.2, est un système mixte (*i.e.* matériel et logiciel) composé d'un micro-contrôleur, d'une mémoire et d'un FPGA implémentant des fréquencemètres et un bloc de communication :

— Les fréquencemètres contenus dans le FPGA permettent de mesurer numériquement la fréquence des signaux oscillants émis par les capteurs.

— La mémoire contient une table de points, qui pour un couple de fréquences (F_X, F_T) donné, donne la valeur numérique *X* correspondante.

— Le micro-contrôleur calcule la valeur X_{num} en fonction de la fréquence des deux signaux et à partir de la table de points.

— Le bloc de communication met en forme les données calculées par le micro-contrôleur, et transmet la trame X_{tram} à l'ECU.

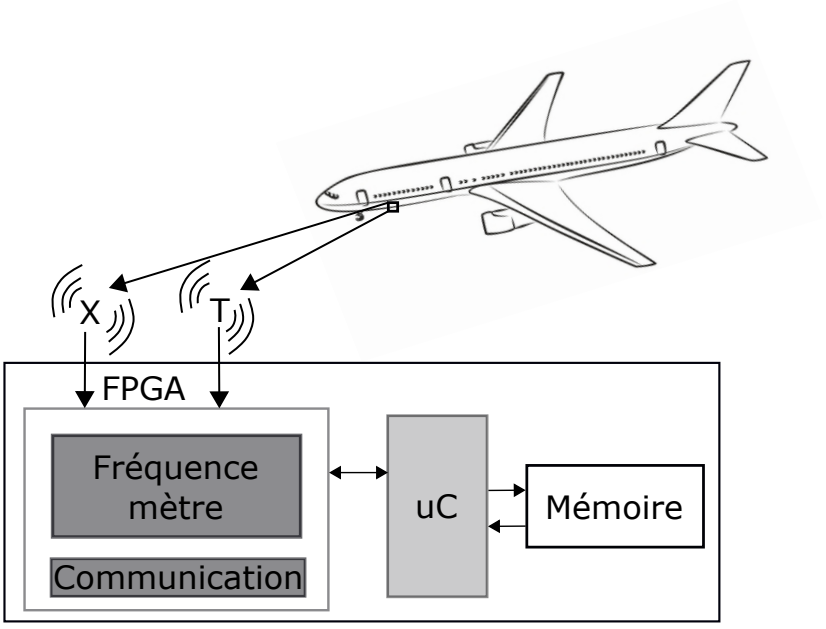

FIGURE 3.2 – *Schéma général du système.*

Architecture d'un fréquencemètre Le fréquencemètre (voir figure 3.3) est composé des blocs DIV_FREQ et COMPTEUR :

— DIV_FREQ est un diviseur de fréquence qui prend en entrée un signal à une certaine fréquence et génère un signal avec une fréquence plus basse.

FIGURE 3.3 – *Architecture d'un fréquencemètre.*

— COMPTEUR prend en entrée trois signaux oscillants à différentes fréquences : le signal Sig (dont on veut mesurer la fréquence), le signal Sig_{ref} généré par un oscillateur interne au système et qui sert de fréquence de référence (F_{ref}) et le signal Sig_{window}, signal obtenu en divisant Sig_{ref}. Le compteur a trois sorties. Les sorties N et N_{ref} renvoient respectivement le nombre d'impulsions des signaux Sig et Sig_{ref} dans une fenêtre définie par deux fronts montants successifs du signal Sig_{window}. La sortie IT indique la fin (ou le début) d'une nouvelle fenêtre.

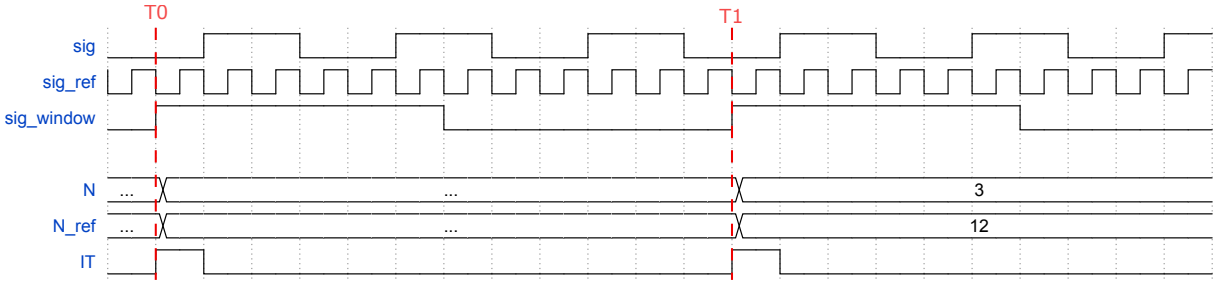

FIGURE 3.4 – *Chronogramme de la mesure de fréquence.*

Le chronogramme 3.4 illustre le fonctionnement du compteur. À l'instant $T0$, il y a un front montant sur le signal Sig_{window}. Le signal IT émet donc une impulsion pour indiquer au micro-contrôleur le début (ou la fin) d'un calcul. Le compteur commence à compter le nombre d'impulsions sur Sig et sur Sig_{ref}. À l'instant $T1$, il y a un nouveau front montant sur le signal Sig_{window}, le compteur met à jour les signaux N et N_{ref} (ici avec les valeurs 3 et 12), et commence un nouveau calcul.

La valeur de la fréquence F du signal Sig est calculée approximativement à l'aide de la fréquence F_{ref} et des signaux N et N_{ref}. Elle est donnée de façon approximative par la formule suivante :

$$F = \frac{N}{N_{ref}} * F_{ref} \tag{3.1}$$

Architecture détaillée du système La figure 3.5 illustre l'architecture détaillée du circuit dans son ensemble. Le FPGA contient deux fréquencemètres. Le premier est utilisé pour calculer N_X. Le compteur est appelé CounterX. Le diviseur est constitué de 3 blocs diviseurs de fréquence ($DIVA, DIVB, DIVC$). Le second fréquencemètre calcule N_T. Son compteur est appelé CounterT. Son diviseur est constitué de 4 blocs diviseurs de fréquence ($DIVA, DIVB, DIVC$, SYNC). Les blocs ($DIVA, DIVB, DIVC$) sont mis en commun dans les deux fréquencemètres. Comme la fenêtre de calcul du second fréquencemètre (pour la température) est plus longue que le premier fréquencemètre (présence d'un quatrième diviseur de fréquence), le calcul de la température prend plus de temps, et moins de calculs sont effectués.

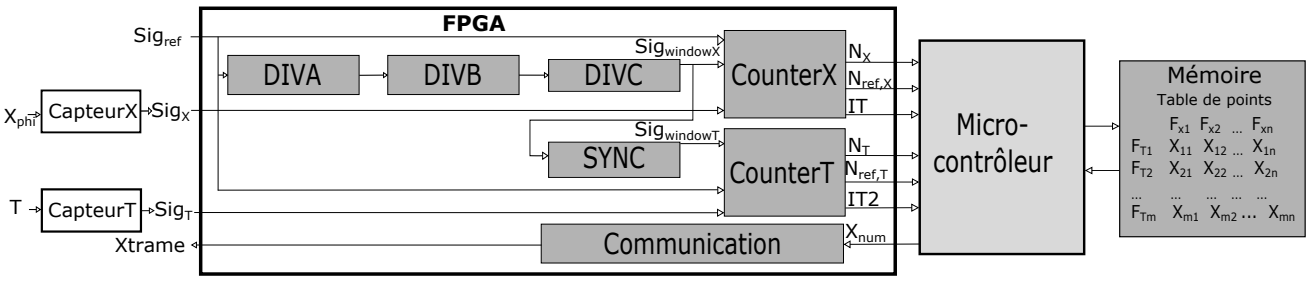

FIGURE 3.5 – *Architecture détaillée du système.*

Le micro-contrôleur utilise le signal d'interruption de CounterX. Ce signal est synchronisé avec celui de CounterT, mais est plus fréquent. Lorsque le micro-contrôleur est interrompu, il vient lire les valeurs N_X, $N_{ref,X}$, N_T et $N_{ref,T}$: les valeurs N_T et $N_{ref,T}$ restent stables pour plusieurs valeurs N_X et $N_{ref,X}$. Le micro-contrôleur implémente l'équation 3.1 et calcule les fréquences correspondantes F_X et F_T.

Le calcul de la valeur de X_{num} s'effectue en utilisant la table de points. Cette table contient les valeurs des $X_{num_{xy}}$ correspondant à des couples (F_{Tx}, F_{Xy}). Elle est spécifique à chaque capteur C_X et est générée grâce à un étalonnage préalable (voir le bloc mémoire de la figure 3.5), où les fréquences sont balayées par pas régulier. Le micro-contrôleur utilise une méthode d'interpolation sur plusieurs couples (F_{Tx}, F_{Xy}), les plus proches du couple (F_T, F_X) mesuré pour calculer une valeur précise de X_{num}. Une fois le calcul effectué, il transmet cette valeur au bloc de communication ainsi que des informations de maintenance (mécanismes de détection d'erreur). Le bloc de communication émet ensuite une trame de 4 mots de 32 bits contenant toutes ces informations (une étiquette, X_{num} et les informations de maintenance).

3.1.2 Contraintes du système et mécanismes de détections

Les contraintes du systèmes sont décrites dans les spécifications de l'avion (*cf.* tableau 3.1). Tous les temps sont illustrés sur le chronogramme 3.6 qui représente le calcul et la transmission de plusieurs trames.

— Le "Flux de donnée" représente le temps entre la transmission de deux trames de données à l'ECU. Sur la figure 3.6, il est représenté par la durée entre le début de la trame Tr1 et le début de Tr2.

— La "Fréquence de transmission" est le temps passé par le bloc de communication pour transmettre une donnée. Ce temps est représenté par la durée de la trame Tr1.

— Le "Temps de calcul" est le temps passé par le micro-contrôleur pour calculer la valeur finale et la transmettre au bloc de communication, après qu'une interruption ait eu lieu. Cette durée correspond au temps entre la réception de N_{ref,X_1} et émission de X_{num1}.

— Le "Délai de sortie d'une donnée" est le temps entre le début d'une fenêtre de mesure et la transmission de la donnée en sortie du système. C'est la durée entre l'instant où l'on commence à compter les impulsions sur Sig_X et l'émission de X_{num1}.

— La propriété "Time to Alarm" correspond au temps minimum avant de notifier une erreur durant la mesure de donnée. Ce délai n'est pas représenté sur la figure. En supposant que toutes les trames indiquent une erreur, il correspondrait au temps

entre l'émission de la première trame erronée (Tr1) et le temps de notification de l'erreur.

Plusieurs mécanismes de détection sont implémentés dans le système : dépassement de la capacité du compteur, détection de période en dehors de la table de point, détection d'une valeur en dehors des plages possibles, *etc.*

Certains d'entre eux sont décrits dans le tableau 3.2. La première colonne correspond au bloc dans lequel le mécanisme est implémenté.

Les valeurs de ces mécanismes sont reportées à l'ECU lors de l'émission de la trame. Pour des raisons de confidentialité, nous ne pouvons pas représenter le contenu exact d'une trame.

TABLEAU 3.1 – *Spécification système*

Propriété	Spécification
Flux de donnée	100 ms +/- 3.3 ms
Time to Alarm	300 ms +/-3.3 ms
Précision	+/- 0.8 U (confidential unit)
Fréquence de transmission	$\leq$ 100 ms
Temps de calcul	$\leq$ 100 ms
Délai de sortie d'une donnée	300 ms +/- 3.3 ms

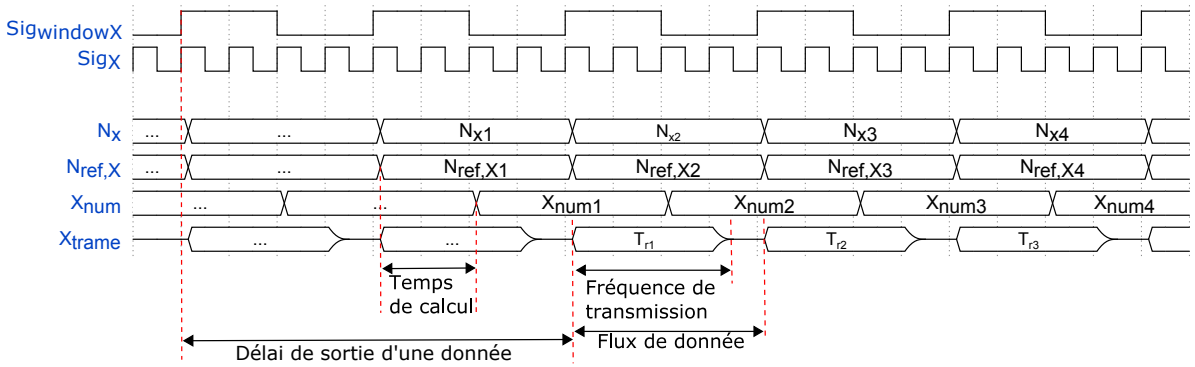

FIGURE 3.6 – *Chronogramme illustrant la transmission de données et trames.*

3.2 Analyse de sûreté dans le cas d'étude

Dans la logique du flot de conception top-down, et en accord avec les recommandations des normes de l'aéronautique, des analyses de sûreté sont effectuées à différents stades de développement. La figure 3.7 illustre les différentes étapes de cette analyse effectuées actuellement par les ingénieurs et comment elles s'inscrivent dans un flot de

TABLEAU 3.2 – *Mécanismes de détection d'erreurs*

	Mécanismes de détection d'erreurs	**Fréquence d'éxecution**	
FPGA	la sortie est rebouclée	100 ms	M1
μC	Reset harware à chaque début de cycle de calcul	100 ms	M2
μC	Vérifie si la période T_X (ou T_T) est dans l'intervalle possible	100 ms	M3
μC	Remet à 0 le μC si le temps est écoulé	100 ms	M4

conception classique. On distingue deux étapes d'analyse : une première étape d'analyse fonctionnelle qui apparaît tôt dans le flot de conception, lors de la définition fonctionnelle du système, puis une analyse plus bas niveau, qui apparaît une fois que la description des blocs est disponible, et que la netlist est générée. Cette analyse plus bas niveau se décompose en deux analyses : une analyse des modes de défaillances des composants, qui sont des fautes permanentes, et une analyse de l'impact des SEUs sur le système. Ces deux dernières analyses permettent enfin de calculer un taux de fautes critiques par heure de vols, qui nous permet de calculer le temps minimum avant une erreur critique (MTBF : Mean Time Before Failure)

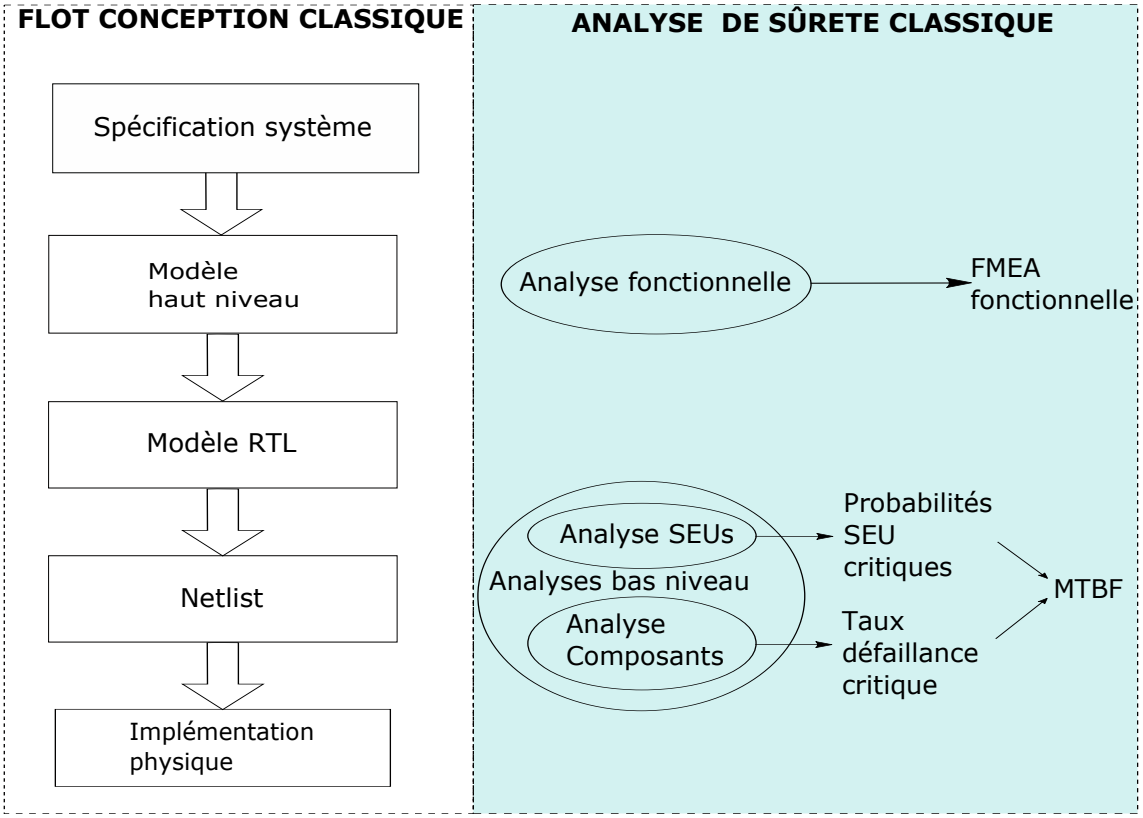

FIGURE 3.7 – *Flot d'analyse de sûreté classique.*

3.2.1 Analyse fonctionnelle

Cette analyse est effectuée au niveau fonctionnel (en accord avec la norme ARP5580). Elle intervient tôt dans le flot de conception, et permet de donner une première estimation de la robustesse du circuit. Elle identifie clairement les chemins critiques du système et évalue la criticité des blocs. Elle n'est pas exhaustive (elle prend uniquement en compte quelques cas de fautes) et les fautes critiques trouvées ne sont souvent pas réalistes.

Le tableau 3.3 illustre un exemple d'AMDEC fonctionnelle. Pour chaque bloc constituant le système, l'impact et la propagation d'une erreur est évaluée manuellement. La première colonne liste les blocs constituant le chemin critique du calcul d'une donnée. La seconde colonne liste des erreurs fonctionnelles pour chacun des blocs. La troisième colonne estime l'impact de cette erreur sur la donnée X en sortie du système. Les erreurs étudiées ne sont pas exhaustives. Cette analyse manuelle repose uniquement sur l'expérience des ingénieurs sûreté. Elle permet de donner une première vision des blocs vulnérables et de spécifier aux concepteurs des blocs des règles afin de garantir la robustesse du système complet.

TABLEAU 3.3 – *AMDEC fonctionnelle*

Bloc	Erreur	Impact
DIVA	Erreur de division	Fréquence de sortie différente, fenêtre de comptage modifiée
DIVB	Erreur de division	Fréquence de sortie différente, fenêtre de comptage modifiée
DIVC	Erreur de division	Fréquence de sortie différente, fenêtre de comptage modifiée
SYNC	Erreur de division	Fréquence de sortie différente, fenêtre de comptage modifiée
COUNTERX	Mauvais fonctionnement du compteur Erreur lors de la lecture des registres	Une erreur de 1 bit entraîne une erreur de 0.1 U sur X
COUNTERT	Mauvais fonctionnement du compteur Erreur lors de la lecture des registres	Une erreur de 1 bit entraîne une erreur de 0.1 U sur X

3.2.2 Analyse bas niveau

La seconde analyse permet de déterminer la probabilité d'événements critiques. Elle intervient à la fin du flot de conception. Elle contient deux parties :

AMDEC au niveau des composants Cette AMDEC étudie l'effet de la défaillance d'un composant sur le système. Ces défaillances composants sont permanentes. Elles sont

très rares. Une analyse globale de leurs effets sur le système est suffisante. Le tableau 3.4 illustre comment cette analyse est effectuée. Pour chaque composant, chaque mode de défaillance est étudié et son impact sur le système est estimé. Le taux de défaillances exprimé en fautes par heure de vol est calculé et permettra à la fin de calculer le taux de défaillance du système global.

TABLEAU 3.4 – *AMDEC composant*

Composant	**Modes de défaillances**	**Impact**	**Taux de défaillance** **Fautes/Heures de vol**
Résistance R1	Court-circuit	Pas de transmission ARINC	1,3E-09
	Valeur trop élevée	Pas d'effet	0,2E-09
	Valeur trop basse	Pas d'effet	0,2E-09

AMDEC sur l'effet des SEUs Cette AMDEC examine l'effet des SEUs et MBUs. Elle nécessite d'être plus précise car la probabilité d'impact de particule sur le système est largement plus élevée. La figure 3.8 illustre le principe de cette analyse.

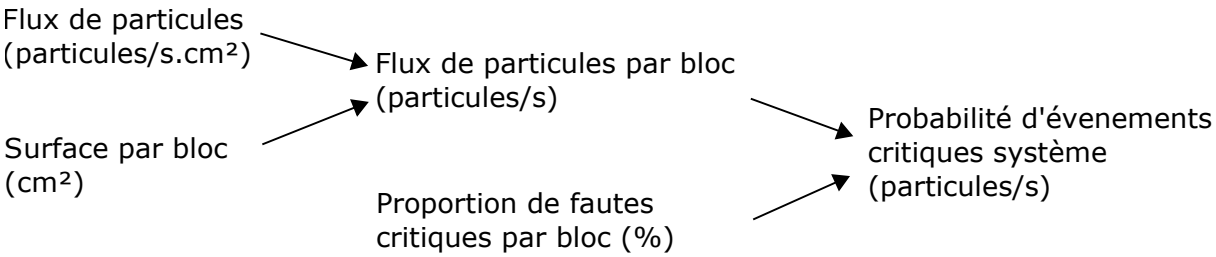

FIGURE 3.8 – *Principe de l'étude de l'AMDEC SEU.*

Le flux de particules et la surface de chaque bloc permet de déduire le flux de particules sur chaque bloc du système. La proportion de fautes critiques (*i.e.* la proportion de SEU/MBU conduisant à une faute critique par bloc du système) permet de calculer la probabilité d'occurrence d'un événement critique. Ces proportions sont déterminées empiriquement, en se basant sur l'expérience des ingénieurs, et sur les précédentes observations. Les ratios utilisés sont pessimistes : en cas de doute, une SEU/MBU est considérée comme critique pour le système.

Le tableau 3.5 montre l'AMDEC SEU effectuée dans notre cas d'étude. La première colonne définit les différents blocs du système. La deuxième colonne donne le flux de particules par bloc. Les colonnes 3 et 4 donnent les modes de défaillance possibles pour le système, et le ratio de fautes conduisant à chaque mode de défaillance. La dernière colonne donne la probabilité d'évènement critique induite par chaque bloc, en fautes par heure.

Cette AMDEC, bien que plus précise, reste peu détaillée. Le pire cas est toujours envisagé. Les ratios sont définis grâce à l'expérience des ingénieurs, mais aucun modèle exécutable ne les vérifie.

TABLEAU 3.5 – *AMDEC SEU*

Bloc	**Flux particules par bloc (f/h)**	**Modes défaillances**	**Ratio**	**Probabilité d'événements critiques système (f/h)**
Micro-contrôleur	0,03E-07	Valeur X erronée	50%	0,015
		Reset	50%	0,015E-07
FPGA	0,15E-07	Valeur X erronée	50%	0,075E-07
		Pas de transmission	50%	0,075E-07
Total	N.A.	N.A.	N.A.	0,18E-07

3.2.3 Conclusion

Les analyses de sûreté sont effectuées à plusieurs niveaux de modélisation, en s'inscrivant dans un flot de conception top-down. Sont effectuées : des analyses fonctionnelles, des analyses des modes de défaillance des composants, et des analyses d'impacts des SEUs. Les analyses sont faites manuellement et se basent sur l'expérience acquise. En l'absence d'outil de simulation, les hypothèses prises sur le taux de fautes critiques sont les plus pessimistes possibles. Dans le cadre de la thèse, le but est de pouvoir proposer une méthode d'aide aux analyses de sûreté. La partie analyse des SEUs est très peu détaillée dans l'analyse de sûreté traditionnellement faite. Elle dispose de très peu d'informations détaillées. Dans la suite, nous proposerons une méthodologie s'appuyant sur de la simulation multi-abstraction et permettant d'effectuer des analyses de sûreté plus réalistes et plus exhaustives que les analyses de sûreté traditionnellement effectuées.

4

Présentation de la méthodologie

Sommaire

Dans ce chapitre, nous allons présenter la méthodologie que nous avons développée au cours de cette thèse.

La méthodologie proposée vise à prendre en compte les différents enjeux des normes ARP5580 et DO254 pour l'analyse de sûreté :

— Fournir une analyse le plus tôt possible dans le flot de conception et ainsi effectuer des analyses à différents niveaux de modélisation.

— Prendre en compte la spécification du système pour éviter la détection de faux évènements critiques.

— Fournir une évaluation de la sûreté à la fois rapide et réaliste.

— Classer les différents blocs selon différents niveaux de criticité (la norme DO254 en définit cinq 2.4).

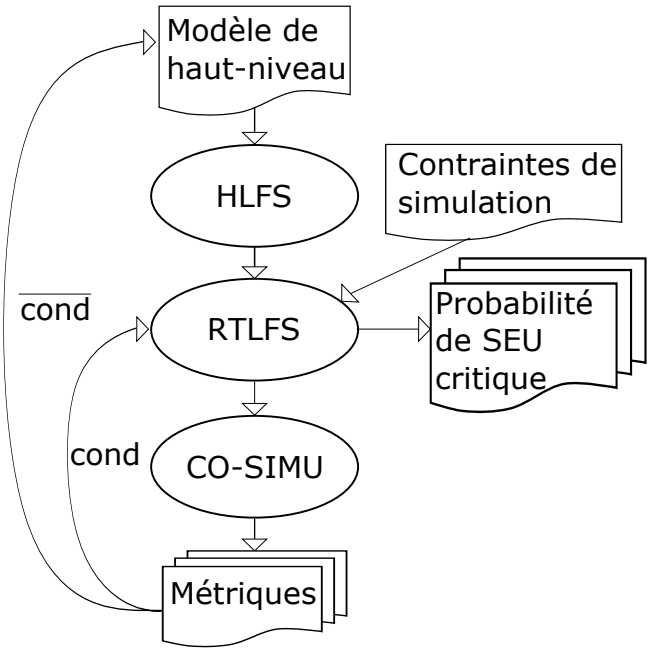

FIGURE 4.1 – *Illustration du flot Quick and Dirty*

La méthodologie illustrée figure 4.1 permet d'évaluer tôt dans le flot de conception et rapidement la sûreté d'un circuit en prenant en compte son environnement. Cette méthodologie est itérative et s'effectue en plusieurs rondes. Chaque ronde effectue des analyses de sûreté (rapides et peu précises) à différents niveaux de modélisation et fournit une métrique permettant d'évaluer la qualité des différents modèles. Tant que la qualité de modélisation n'est pas suffisante, les modèles sont raffinés et de nouvelles rondes sont effectuées. Une fois la qualité atteinte, une évaluation précise de la sûreté est effectuée.

Chaque ronde se décompose en trois étapes, décrites par la figure 4.2. La première étape est une simulation de fautes de haut-niveau (HLFS : High Level Fault Simulation), elle permet d'identifier les fautes de haut-niveau créant des comportements critiques. La deuxième étape est une simulation de fautes RTL (RTLFS : RTL Fault simulation). Elle utilise la description RTL des blocs du circuit. Elle permet de trouver les fautes RTL correspondantes aux fautes de haut-niveau identifiées comme critiques lors de la simulation de haut-niveau. La troisième étape est une étape de co-simulation (CO-SIMU). Elle réutilise la description RTL d'un bloc et le simule dans l'environnement système avec les

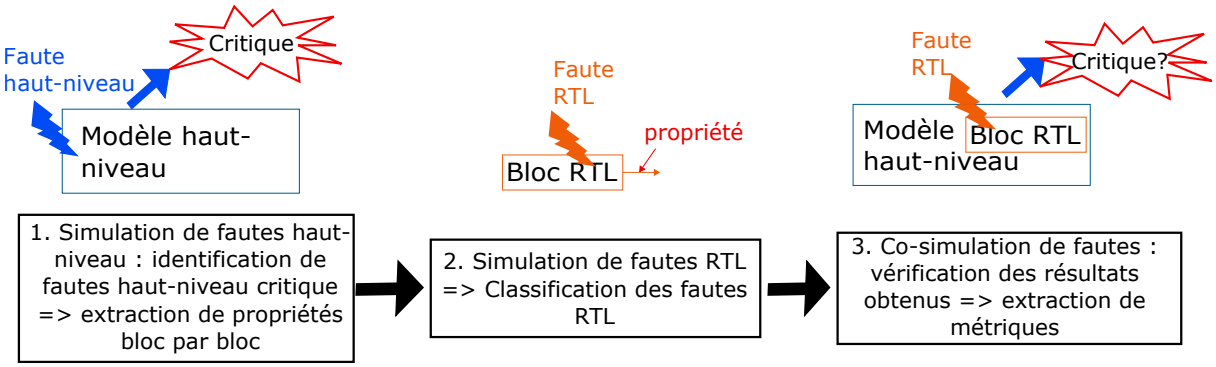

FIGURE 4.2 – *Les trois étapes de la méthodologie*

fautes RTL déjà injectées lors de l'étape de simulation RTL. Cette étape permet ainsi de vérifier le réalisme des résultats obtenus dans lors des étapes précédentes.

Ce chapitre a pour but de montrer une vue générale de la méthodologie en présentant les principes de chacune des étapes et l'intégration du flot d'analyse de sûreté proposé dans un flot de conception classique. Les détails d'implémentation de chacune des étapes et leur application au cas d'étude seront présentés dans les prochains chapitres du manuscrit.

4.1 Etape 1 : Simulation de fautes de haut-niveau

Cette première étape s'effectue avec un modèle du système de haut-niveau. Elle se déroule en trois parties (voir le flot figure 4.3) : la modélisation de haut-niveau du système et du modèle de faute, la simulation de fautes puis l'analyse des résultats de simulation. Cette analyse permet d'obtenir, à la fin de cette étape, des propriétés critiques et de fournir une AMDEC fonctionnelle.

Modélisation : Le modèle du système (figure 4.4) consiste en un modèle de haut-niveau du circuit constitué de plusieurs blocs TLM (voir 2.3.1), un bloc de vérification implémentant la spécification, et un modèle de fautes de haut-niveau.

Simulation : À partir d'un modèle de fautes de haut-niveau, un script permet d'automatiser la simulation de fautes. Les fautes (*Faute_TLM*) sont injectées aux interfaces des blocs. La simulation donne en sortie une classification des fautes de haut-niveau.

Analyse : L'analyse de la classification des fautes nous permet :

— De définir des propriétés de criticité sur chacune des sorties des blocs matériels. Ces propriétés sont définies en deux dimensions : temps et valeurs.

— De fournir des premières informations sur la sûreté du circuit au niveau fonctionnel en identifiant les blocs qui semblent être les plus sensibles. Ces informations sont utiles à l'AMDEC fonctionnelle.

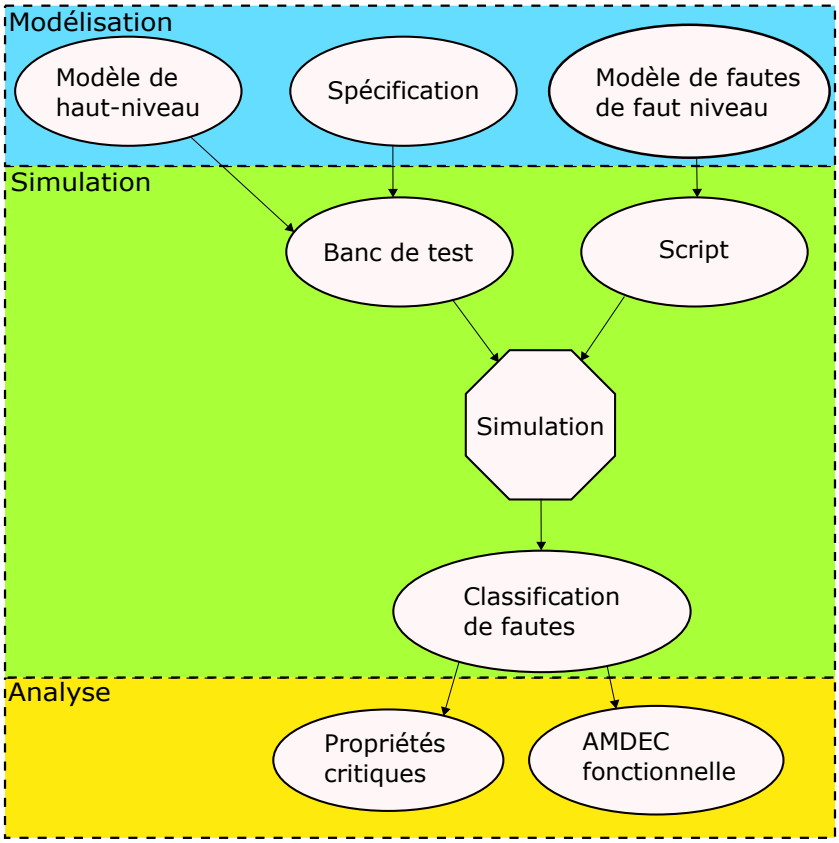

FIGURE 4.3 – *Flot de la simulation de fautes de haut-niveau*

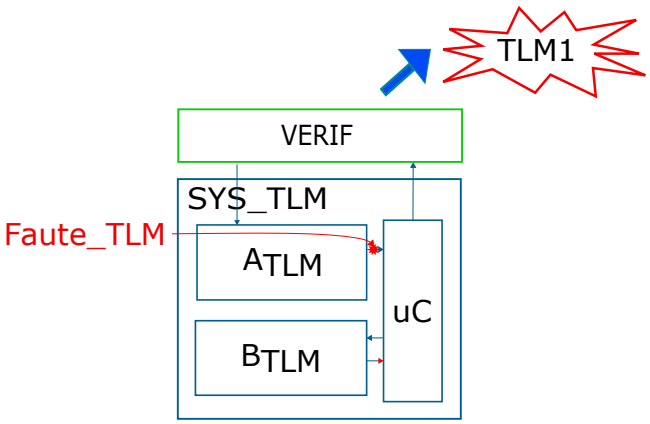

FIGURE 4.4 – *Modélisation de haut-niveau du système et localisation de l'injection de faute.*

Cette première étape permet des temps de simulation rapide. Néanmoins, la simulation de fautes de haut-niveau implique d'enlever des détails sur le modèle, et d'utiliser des modèles de fautes dont le réalisme n'est pas prouvé.

4.2 Etape 2 : Simulation de fautes RTL

La deuxième étape de la méthodologie se passe au niveau RTL. Le but de cette étape est d'utiliser la description RTL des blocs du système et d'injecter des fautes RTL pour étudier leur propagation en sortie du bloc. Nous utilisons les propriétés extraites à l'étape de simulation de haut-niveau. Ces propriétés permettent de vérifier si la sortie du bloc est critique ou pas, et donc de déterminer si la faute RTL injectée est critique.

Le flot de la simulation RTL est illustré figure 4.5. Comme la simulation de haut-niveau, elle se décompose en trois sous étapes : la modélisation, la simulation et l'analyse. La figure 4.6 illustre l'étape de simulation RTL.

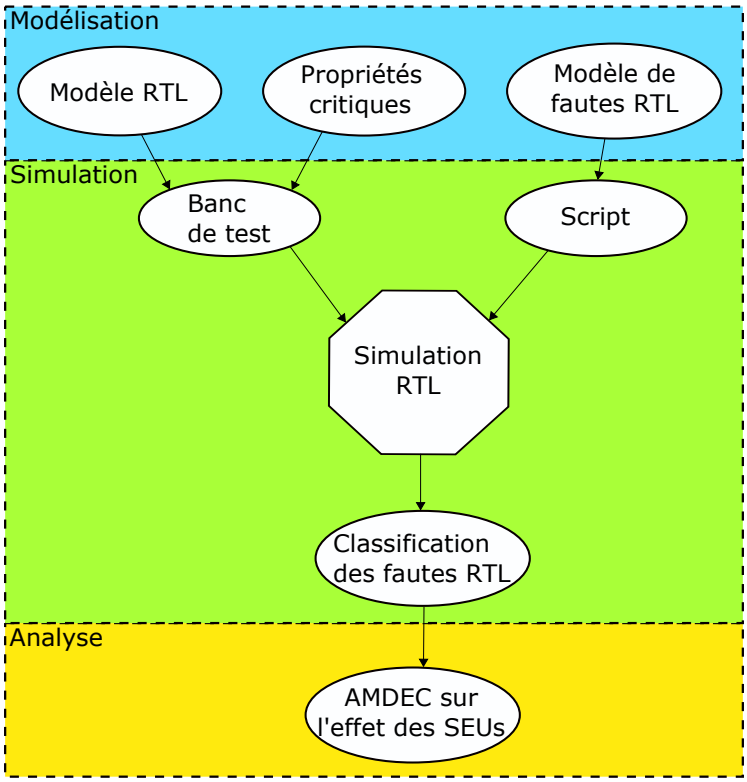

FIGURE 4.5 – *Flot de la simulation de fautes RTL*

Modélisation : Afin de pouvoir effectuer la simulation, cette étape doit prendre en entrée le modèle RTL de chacun des blocs matériels (A_{RTL} dans la figure 4.6), les propriétés critiques générées par la simulation haut-niveau, et un modèle de fautes RTL (*Faute_RTL*).

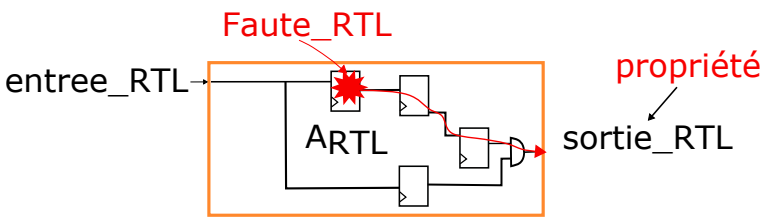

FIGURE 4.6 – *Modélisation RTL d'un bloc du système, localisation de la faute et vérification de la propriété en sortie.*

Simulation : Lors de l'étape de simulation, le banc de test vérifie les sorties du bloc RTL implémenté ($sortie\_RTL$), grâce aux propriétés critiques. Un script automatise l'injection de fautes selon le modèle de fautes RTL défini. Le résultat de la simulation donne une classification des fautes RTL.

Analyse À partir de la classification des fautes RTL, une analyse plus poussée permet d'obtenir une AMDEC sur l'effet des SEUs en calculant la proportion de SEUs critiques pour le système.

Cette AMDEC permet une analyse détaillée quantitative tandis que celle proposée dans la méthodologie industrielle classique proposait des résultats peu détaillés et qualitatifs.

4.3 Etape 3 : Co-simulation

La dernière étape est la co-simulation des blocs RTL critiques dans le modèle haut-niveau. Cette simulation est la référence, mais est très chronophage.

Le but de cette étape est de vérifier la concordance des résultats obtenus pour chacun des blocs dans la simulation de fautes RTL, avec la co-simulation de ces mêmes blocs avec le modèle haut-niveau du système. La description RTL de chaque bloc est co-simulée dans le système grâce au modèle haut-niveau du système. Les fautes RTL injectées dans la simulation de fautes RTL sont réinjectées ici et leur propagation à travers le système est analysée. Dans la figure le bloc A_{RTL} est co-simulé dans le système SYS_TLM. Une faute RTL est injectée et le bloc de vérification permet de vérifier si le comportement $TLM1$ en sortie du système est critique ou non.

Le flot de la CO-SIMU est illustré figure 4.7. Nous retrouvons les trois étapes de modélisation, simulation et d'analyse.

Modélisation Cette étape prend en entrée les descriptions RTL de chaque bloc matériel, le modèle de haut-niveau, la spécification et les fautes classifiées dans la RTLFS. Pour chaque bloc matériel, un modèle de co-simulation (illustré par la figure 4.8) est développé intégrant un à un chaque description RTL (A_{RTL}) dans l'environnement haut-niveau (SYS_TLM et $VERIF$). Cela permet à chaque fois de définir un banc de test. Un script d'injection de fautes est défini à partir des fautes déjà injectées dans l'étape de simulation RTL.

Simulation Lors de l'étape de co-simulation, le banc de test implémentant le modèle RTL de chaque bloc intégré dans le modèle de haut-niveau et la spécification contraignant les sorties de ces blocs sont utilisés. Un script permettant d'injecter les fautes permet de lancer la simulation de fautes. Le résultat de la simulation donne une nouvelle classification des fautes RTL, différente de celle obtenue dans la simulation de fautes RTL.

Analyse À partir de la classification des fautes RTL en co-simulation, une comparaison avec la classification obtenue en sortie de la RTLFS permet d'obtenir des métriques sur la qualité du modèle de haut-niveau utilisé dans l'étape de simulation de haut-niveau.

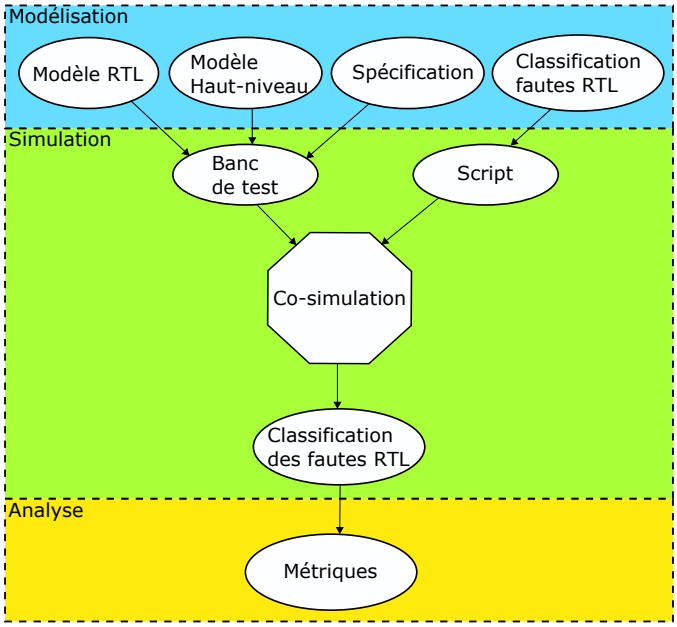

FIGURE 4.7 – *Flot de la co-simulation*

L'étape de co-simulation est extrêmement chronophage, elle n'est donc effectuée que sur des petites campagnes d'injection de fautes. Plus précisément, le nombre de fautes injectées dépendra du temps de co-simulation disponible.

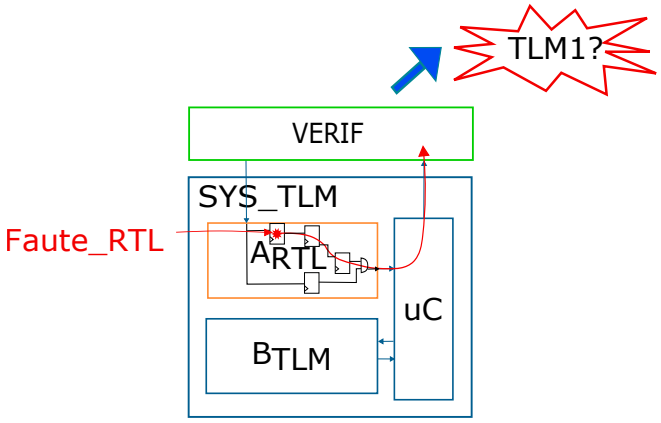

FIGURE 4.8 – *Schéma illustrant l'étape de co-simulation*

4.4 Flot d'analyse de sûreté *vs.* flot de conception classique

La figure 4.9 illustre comment la méthodologie proposée peut s'inscrire dans un flot de conception top-down classique. La simulation de haut-niveau (encadré bleu) nécessite en plus du flot de conception classique de développer un modèle de faute, et permet d'obtenir des propriétés de criticité qui sont utiles à la fois à l'AMDEC fonctionnelle, mais aussi à

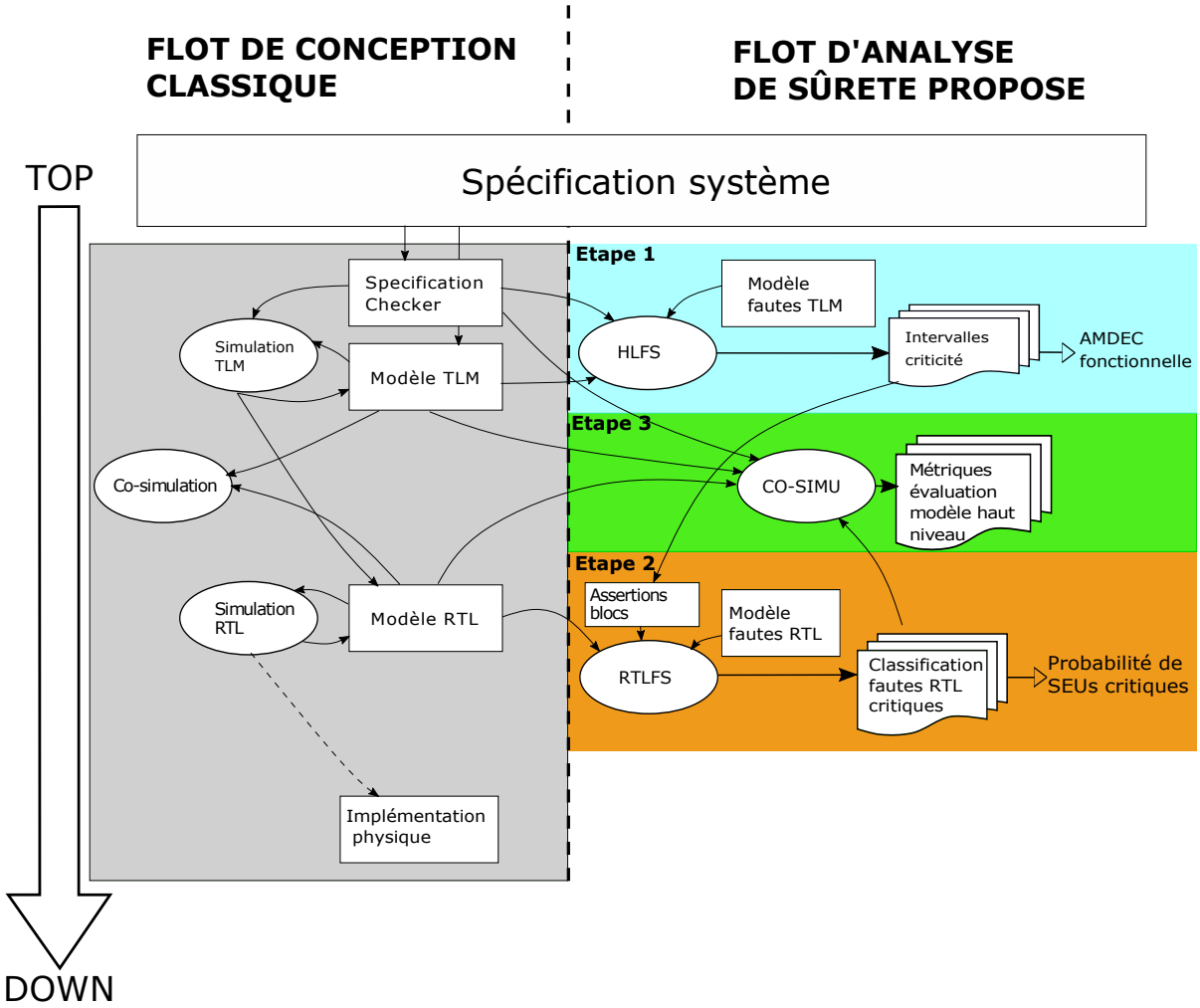

FIGURE 4.9 – *Analogie entre le flot de conception classique et le flot d'analyse de sûreté proposé*

la simulation de fautes RTL. La simulation de fautes RTL (encadré orange) se déroule en même temps que le développement des blocs RTL du système, elle nécessite de définir des assertions à partir des propriétés de criticité, et de définir un modèle de fautes RTL. Elle permet la classification des fautes RTL critiques, qui donnent une première estimation de la probabilité de SEU critique. Enfin, l'étape de co-simulation se déroule en même temps que les étapes de co-simulation du flot de conception classique (*i.e.* avant la finalisation de l'ensemble des blocs RTL), elle permet de déduire des métriques qui donnent une évaluation sur la qualité du modèle de haut-niveau, et de son modèle de fautes. Ces trois étapes interviennent tôt dans le flot de conception, et bien avant les étapes de synthèse, placement, routage qui mènent à l'implémentation physique du circuit. De plus, les standards aéronautiques ARP5580 et DO254 recommandent des analyses à différents niveaux de modélisation. Cette méthodologie permet donc à la fois d'effectuer une analyse fonctionnelle et une simulation de faute de l'architecture matérielle.

La qualité des résultats obtenus dans les étapes de simulation de fautes à haut-niveau et de simulation de fautes RTL dépend principalement du modèle de haut-niveau, et du modèle de fautes de haut-niveau. Néanmoins, l'étape de co-simulation permet de vérifier la qualité de ces modèles haut-niveau, et de les améliorer.

5

1ère étape : simulation de fautes à haut-niveau d'abstraction

Sommaire

Afin d'obtenir une évaluation rapide de la sûreté du système tôt dans le flot de conception, tout en prenant en compte la spécification, la première étape consiste en une simulation de fautes de haut-niveau. Le but de cette étape est d'identifier les fautes de haut-niveau conduisant à des comportements critiques. Cette ensemble de fautes de haut-niveau permettra d'extraire des propriétés critiques sur les sorties de chaque bloc constituant le modèle de haut-niveau.

L'étape de simulation de faute de haut-niveau décrite dans la figure 4.3 du chapitre 4.1 prend en entrée un modèle de haut-niveau, des scénarios de simulation, et une campagne d'injection de fautes. Elle donne en sortie des propriétés critiques. L'étape de simulation de fautes de haut-niveau nécessite donc d'implémenter :

— Un modèle de haut-niveau, qui devra être validé, et des scénarios.

— Un modèle de fautes de haut-niveau.

— Le traitement des résultats pour extraire des propriétés critiques en triant les comportements erronés grâce à la spécification du système.

Cette section contiendra donc trois premières parties correspondant à ces trois étapes d'implémentation, illustrées au fur et à mesure sur le cas d'étude proposé dans le Chapitre 3. La dernière partie présentera les résultats de cette étape appliquée au cas d'étude.

5.1 Modélisation de l'environnement de test

5.1.1 Modélisation du système à haut-niveau et validation

Cette méthodologie s'inscrit dans un flot de conception top-down classique, afin de profiter des efforts de modélisation de haut-niveau et éviter un sur-coût. L'évaluation rapide de la sûreté tôt dans le flot de conception passe par une simulation de fautes sur un modèle de haut-niveau du système. En effet, la rapidité des simulations des modèles de haut-niveau permet de réaliser une évaluation de la sûreté nécessitant des temps de simulation très courts. La modélisation au niveau transactionnel (TLM : Transaction Level Modeling) est une approche de haut-niveau pour modéliser les systèmes (voir section 2.3.1). C'est le niveau que nous avons choisi pour modéliser notre système. À ce niveau de modélisation, les interfaces de communication et les blocs fonctionnels sont séparés. L'attention peut alors porter sur chaque interface de communication et chaque bloc fonctionnel. La modélisation TLM peut être utilisée pour explorer différentes architectures systèmes. Une fois l'architecture définie, chaque fonctionnalité peut être implémentée en utilisant une implémentation matérielle, ou logicielle. Les interfaces étant préservées entre les différents niveaux de modélisation des blocs fonctionnels, les fautes qui sont injectées à haut-niveau pourront être comparées aux fautes injectées à plus bas niveau. Au niveau transactionnel, dans les premiers modèles développés, le temps n'est en général pas modélisé étant donné que l'intérêt est porté sur la modélisation des fonctions du système, on appelle cette façon de modéliser le TLM LT (Loosely Timed). Néanmoins, dans le cas des systèmes critiques temps réels, les spécifications systèmes incluent des propriétés temporelles (*e.g.*, débit de données, ou délai de calcul d'une donnée), et un Time-To-Alarm (TTA) peut être spécifié pour définir la durée minimale avant qu'une erreur soit reportée à l'utilisateur. La prise en compte des temps de calcul est alors primordiale pour une juste

évaluation de la sûreté. La solution la plus appropriée pour la modélisation d'un SoC temps réel est alors la modélisation TLM AT (Approximately Timed). Le langage SystemC est un langage particulièrement adapté à la modélisation TLM : c'est le standard pour la modélisation des SoC. C'est le langage qui sera utilisé.

La modélisation du système à haut-niveau nécessite une phase de validation. Le but de cette étape est de valider que le système remplit bien la spécification initiale. Pour cela, nous utilisons un module supplémentaire appelé "Specification Checker", chargé de vérifier si le comportement est bien dans la spécification du système. Ce "Specification Checker" utilise des assertions pour vérifier que le comportement du système est bien dans la spécification. La section suivante illustre cette phase de validation sur notre cas d'étude.

5.1.2 Illustration sur le cas d'étude : modélisation du cas d'étude au niveau TLM et validation

Modélisation du cas d'étude au niveau TLM Nous avons modélisé le cas d'étude décrit dans le chapitre 3 au niveau TLM. La figure 5.1 illustre le modèle de haut-niveau implémenté en SytemC pour cette modélisation TLM.

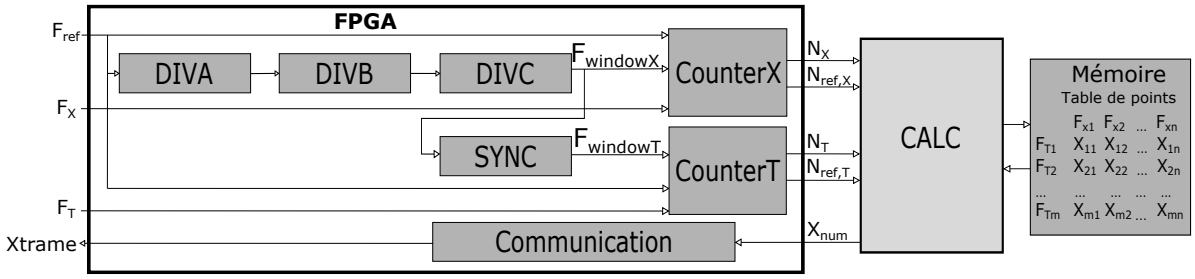

FIGURE 5.1 – *Schéma du système modélisé au niveau TLM.*

Les signaux d'entrées oscillants Sig_{ref}, Sig_X et Sig_T (voir section 3.1) sont représentés dans le modèle TLM par leur valeur de fréquence, soit par les flottants F_{ref}, F_X et F_T.

Les fonctionnalités des diviseurs de fréquences *DIVA*, *DIVB*, *DIVC* et *SYNC* sont implémentées grâce à une simple division en raisonnant sur des flottants, la fonction *wait*() permet de modéliser le temps nécessaire au calcul de chaque résultat.

Les fonctionnalités des deux compteurs *CounterX* et *CounterT* sont aussi implémentées grâce à des divisions et une fonction *wait*(). Les sorties des compteurs sont représentées grâce à des entiers non signés (N_X, $N_{ref,X}$, N_T et $N_{ref,T}$). Le signal d'interruption (voir chapitre 3, section 3.1) n'est pas modélisé car on utilise comme canal de communication pour les sorties des compteurs des *SC_FIFO*, qui permettent de notifier lorsqu'une nouvelle valeur est écrite sur le canal.

Le calcul effectué par le micro-contrôleur est implémenté dans le bloc *CALC*. La mémoire est représentée par le bloc *Mémoire*, qui contient la table de point. Le bloc *Communication* ne fait que lire la valeur transmise par le micro-contrôleur *Xnum* et les informations de simulation (*i.e.*, erreurs détectées dans les différents blocs, format de la donnée transmise, *etc.*), et les retransmet avec un délai modélisant le délai mis par le module de communication pour mettre en forme et transmettre ces données.

Les figures 5.2 et 5.3 donnent un exemple de l'implémentation SystemC pour chaque bloc *DIVA* et *CALC*.

```
1  SC_MODULE {DIVA}{
2  public:
3  sc_in <float> diva_in ;
4  sc_out<float> diva_out;
5  // constructor
6  DIVA (sc_module_name name_) {
7       diva_val = 64;
8       // processes declaration
9       SC_THREAD(DIVA_process);}
10 // process
11 SC_HAS_PROCESS(DIVA);
12 void DIVA_process(){
13      float val;
14      int time_to_wait ;
15      wait(dia1_in.value_changed_event());
16      while (1) {
17          if (dia1_in.read()) {
18              val = diva_in.read()/diva_val ;
19              diva_out.write(val); // ecriture de la valeur de la
    division en sortie
20              time_to_wait = ((1/val) *1000000);
21              wait (time_to_wait, SC_US);
22          }
23      }
24 }
25 private:
26 float diva_val ; };
```

FIGURE 5.2 – *Implémentation SystemC TLM AT du diviseur DIVA*

```
class CALC : public sc_module
{
  public:
    sc_in <unsigned int> N_x, N_t;
    sc_in <unsigned long> Nref_x, Nref_t;
    sc_out<float> Xcomp;
  // constructor
    CALC (sc_module_name name_) {
    Fref = 100000 ;
    // processes declaration
    SC_THREAD(CALC_process);     }
  // process
    SC_HAS_PROCESS(CALC);
    void CALC_process(){
        float X ;
        float F_t ;
        float F_x ;
        while (1) {
            wait( Np.value_changed_event()  ) ;
            F_x = (N_x/Nref_x.read()) * Fref;
            F_t = (N_t/Nref_t.read())* Fref ;
            X = INTER (F_t, F_x) ;
            wait ( 100, SC_MS);
        }
      // fonction pour calculer X a partir des donnees disponibles
        float INTER( float F_t, float F_x) ;
        Xcomp.write(X) ;
        }
    }
private:
    float Fref ;
};
```

FIGURE 5.3 – *Implémentation SystemC TLM AT du microcontrôleur*

Le bloc *DIVA* prend en entrée le signal *diva_in* (qui est en fait F_{ref}) représentant une fréquence sous forme d'un flottant, et ressort un signal *diva_out* égale à la fréquence d'entrée divisée par *diva_val*, une variable interne contenant la valeur de division de *DIVA*. La modélisation du temps se fait à l'aide de la fonction *wait*() (ligne 21). Dans le cas du diviseur le temps attendu est variable et dépend de la valeur de la fréquence d'entrée (ligne 20). Les autres diviseurs ont un fonctionnement similaire, seul la valeur de division change.

Le bloc *CALC*, correspondant au code du micro-contrôleur, prend en entrée les valeurs Nx, $Nref\_x$, Nt, $Nref\_T$ correspondant aux valeurs comptées par les compteurs $CounterX$ et $CounterT$. Le bloc effectue un calcul d'interpolation (fonction $INTER()$ ligne 22) grâce à la table de points contenue dans la mémoire pour calculer la valeur X correspondante. Cette valeur est ensuite écrite sur la sortie $Xcomp$. Le *wait*() qui permet d'approximer le temps attend ici un temps fixe (ligne 23), correspondant au temps nécessaire au micro-contrôleur pour effectuer sa fonction.

Validation du modèle TLM Une fois le modèle implémenté, il faut fonctionnellement faire une première validation du modèle (*i.e.*, vérifier que le modèle calcule la bonne valeur $Xtrame$ en sortie pour un couple en entrée $(Fx;Ft)$). Dans notre cas, nous disposons d'une table de référence générée lors de l'étalonnage de chaque capteur. Les capteurs ont été soumis à 90 couples aléatoirement choisis $(X;T)$ pour lesquels les fréquences $(Fx;Ft)$ ont été mesurées avec des équipement de hautes précisions. Cette table diffère de la table de point utilisée dans le système, qui fournit elle à l'inverse une valeur X pour différents couples $(Fx;Ft)$, balayés régulièrement. La table de référence a donc l'intérêt de pouvoir vérifier la qualité du calcul d'interpolation.

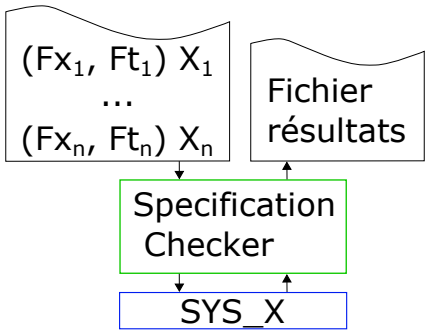

FIGURE 5.4 – *Schéma de l'environnement de validation du système de haut-niveau implémenté.*

Un bloc supplémentaire implémente la spécification décrite dans la section 3.1.2 : il vérifie la cohérence entre les fréquences en entrée du système et la valeur X calculée, et le respect des temps de calculs, et la détection des erreurs. La figure 5.4 illustre l'environnement de vérification du système SYS_X implémenté. Le bloc $Specification\ Checker$ prend en entrée du système un couple de fréquences (Fx_i, Ft_i) et sa valeur correspondante X_i, et sort les informations de simulations dans le fichier contenant les résultats de simulation $Fichier\ resultats$. Il met en entrée du système SYS_X chacun des couples (Fx_i, Ft_i) et vérifie que la sortie du système $Xtrame$ est bien dans la spécification par rapport à X_i attendu. Il implémente des assertions permettant de vérifier la spécification du système.

5.2 Injection de fautes de haut-niveau

5.2.1 Définition du modèle de faute à haut-niveau

Comme mentionné dans l'état de l'art, l'injection de fautes en simulation peut se faire de différentes manières. L'injection de fautes à l'aide de commandes simulateurs est la solution la moins intrusive car elle ne nécessite pas de modification du code. De plus, la simulation avec Modelsim donne des possibilités d'injection de fautes suffisantes pour les modèles de fautes envisagés.

Une faute est représentée par un quadruplé (L,T,I,D) où :

— L est la localisation et représente un signal d'interface (les fautes sont injectées sur toutes les interfaces de communication).

— T est le type de faute. C'est une fonction qui corrompt la valeur originale du signal. Le signal est modifié en pourcentage (signal +/- A), où A est l'amplitude de la faute injectée.

— I est l'instant auquel la faute est injectée.

— D est la durée d'injection. Comme des fautes transitoires sont injectées, la valeur est corrompue soit pour une durée (correspondant à une perturbation), soit jusqu'à ce que le signal soit réécrit (ce qui correspond à une SEU ou une MBU).

```
1 run $I s #simulation de I secondes
2 set $L_temp [examine L] #lecture et stockage de la valeur de L
3 force deposit $L [expr $L_temp * (1 + $A)] # injecte faute sur $L
4 run $D s #simule pour la duree de la faute
5 force deposit $L $L_temp # $L reprend sa valeur initiale
6 run [expr $temps_simu-$I-$D ] s #Lancement jusqu a fin de simulation
```

FIGURE 5.5 – *Extrait du script tcl permettant d'injecter une faute sur un signal*

L'injection de fautes s'effectue à l'aide de commandes du simulateur, car parmi les solutions standards, c'est la solution la moins intrusive et qui offre les meilleurs performances de temps en simulation [71]. La figure 5.5 illustre un script permettant d'injecter une faute dans Modelsim. Le script est écrit en tcl car cela le rend compatible avec le simulateur Modelsim utilisé. La faute injectée est la faute F définie par le quadruplé (L,T,I,D). La durée de simulation totale est *temps_simu*. L'injection d'une faute se passe en plusieurs étapes :

— Simulation jusqu'à l'instant d'injection I (ligne 1).

— Modification de la valeur du signal L en injectant la faute de type T, d'amplitude A (lignes 2 et 3).

— Simulation pour la durée de la faute D, et remise du signal L à sa valeur initiale Cette étape n'a pas lieu dans le cas de l'injection de fautes jusqu'à une nouvelle modification du signal (lignes 4 et 5).

— Lancement de la simulation jusqu'à la fin (ligne 6).

Le but de l'étape est d'extraire des propriétés sur chaque signal. Nous réutiliserons ces propriétés dans l'étape suivante de simulation de fautes RTL. Dans ce but, les fautes injectées peuvent balayer une plage d'amplitude, et de temps, afin de pouvoir définir facilement ensuite des intervalles de criticité. Le pas (en amplitude ou en temps) qui sépare

chaque faute doit être adapté aux capacités de simulation et au temps à allouer à cette première étape de simulation.

Les solutions d'injection de faute à haut-niveau étant infinie, nous faisons le choix, d'injecter des fautes de haut-niveau uniquement sur un seul signal à la fois. Cette hypothèse peut être adaptée à certains systèmes, dans le cas où les signaux de sorties de chacun des blocs ont une faible interdépendance. Une évaluation du modèle de haut-niveau et du modèle de faute de haut-niveau permettra de valider ou non la pertinence de cette hypothèse.

5.2.2 Illustration sur le cas d'étude : Définition d'un modèle de faute de haut-niveau

Pour notre cas d'étude, un relevé de X prend 300ms (cf. spécification système décrite dans 3.1.2). On définit le modèle de faute de manière à ce qu'il couvre un temps supérieur à un relevé de X, soit 300ms. Les instants d'injections choisis sont : 0ms, 50ms, 100ms, 150ms, 200ms, 250ms et 300ms. Les instants d'injections choisis correspondent aux instants où les données transitent. On choisit ces instants par rapport à la durée des fonctions *wait*() des blocs. On veut injecter au moment où les données transitent car c'est le moment où la faute injectée peut se propager dans le système. Si on injecte à d'autres moments, les processus des blocs sont dans des *wait*() et attendent un temps. Les fautes ne sont donc pas prises en compte.

On choisit alors de balayer l'amplitude d'injection sur une plage de -50% à +50% par pas de 10%. Ce choix est fait après une exploration rapide en injectant différentes fautes et afin d'identifier rapidement les intervalles qui semblent poser problème. Le choix du pas peut être raffiné selon les résultats de cette première injection de fautes. La section 5.3 présentant comment exploiter les résultats de l'injection présentera comment un raffinement est possible.

Dans notre cas d'étude, nous injectons des fautes dans les signaux jusqu'à qu'ils soient réécrits. Cela correspond à la durée de l'effet de la faute sur le système. On injecte donc d'abord 10 amplitudes de fautes pour 7 instants d'injections.

5.3 Extraction des résultats

5.3.1 Classification des fautes de haut-niveau et définition de propriétés

Le but de cette partie est de traiter les résultats de la simulation de fautes de haut-niveau, afin de classer les fautes de haut-niveau selon leur effet sur le système. On cherche plus particulièrement à identifier les fautes qui créent un comportement critique pour le système. Cette classification des fautes, portant sur les signaux aux interfaces, permettra alors de déduire des propriétés de criticité portant sur les sorties de chacun des blocs.

Un comportement critique est défini par un comportement erroné (*i.e.*, qui ne respecte pas la spécification du système) qui n'est pas détecté par les mécanismes de détections internes au système (les comportements erronés détectés ne sont pas considérés comme

critiques). Le *Specification Checker* développé lors de la phase de validation du modèle de haut-niveau est réutilisé lors de la simulation de faute, et permet de filtrer parmi les comportements erronés ceux qui sont critiques pour le système.

Lors de notre simulation de fautes, on peut classer les fautes en quatre catégories : Silencieuses (S) lorsque elles n'ont aucun effet sur la sortie du système, Tolérables (T) lorsque l'effet de la faute est tolérée par le système (la sortie reste dans la spécification), Détectées (D) lorsque la sortie n'est pas dans la spécification mais que la faute est détectée par le système ou Critiques (C) lorsque la sortie n'est ni dans la spécification ni détectée.

Algorithm 1: Classification des fautes lors de l'injection de fautes de haut-niveau

```
1  Golden_output ← simulate(Model, input, No_Fault)
2  for F ∈ Fault_injection_location do
3      Faulty_output ← simulate(Model, input, F)
4      if Faulty_output = Golden_output then
5          S ← F
6      else
7          if Faulty_output ∈ specification_requirements then
8              T ← F
9          else
10             if Faulty_output ∈ Detected_Error then
11                 D ← F
12             else
13                 C ← F
```

Algorithm 2: Classification simplifiée des fautes lors de l'injection de fautes de haut-niveau

```
1  for F ∈ Fault_injection_location do
2      else
3          if Faulty_output ∈ specification_requirements then
4              T' ← F
5          else
6              if Faulty_output ∈ Detected_Error then
7                  NC ← F
8              else
9                  C ← F
```

L'algorithme 1 illustre comment les résultats de l'injection de fautes de haut-niveau sont triés. Un fichier `Golden_output` permet de sauvegarder les résultats de simulation sans injection de fautes (ligne 2). C'est le fichier de référence. On parcourt ensuite chacune des fautes composant la campagne d'injection de fautes. Pour chaque faute, on lance une simulation. Les résultats de simulation pour chaque faute injectée sont écrits dans un

fichier temporaire `output` (ligne 6). Un script tcl permet d'implémenter cet algorithme. Cela le rend facilement compatible avec Modelsim. La classification des fautes se fait ensuite de la manière suivante :

— Fautes silencieuses (S) : la sortie du système `output` est égale à `Golden_output` (lignes 7, 8).

— Fautes tolérables (T) : la sortie `output` n'est pas égale à `Golden_output` mais respecte la spécification (lignes 10 et 11).

— Fautes détectées (D) : la sortie `output` ne respecte pas la spécification mais au moins un des mécanismes de détection est activé (lignes 13, 14). Cette catégorie permet de voir l'intérêt des mécanismes de détection, et d'étudier leur redondance.

— Fautes critiques (C) : la sortie `output` est en dehors de la spécification et aucun mécanisme de détection implémenté ne détecte l'erreur (ligne 15).

Les lignes en rouge (lignes 6 à 12) correspondent à la contribution de notre classification proposée. Les approches classiques ne prennent pas en compte la spécification système, et classent donc toute sortie différente du modèle de référence comme critiques. La distinction des catégories Tolérables et Détectées permet de mettre en évidence l'intérêt de la prise en compte de la spécification. Néanmoins, dans le seul but d'obtenir des intervalles de criticité, les catégories Silencieuse, Tolérable et Détecté peuvent constituer une seule catégorie qui correspondrait à toutes les fautes qui ne sont pas considérées comme critiques (Non Critiques : NC). Cette classification-là est illustrée dans l'algorithme 2.

Une fois les résultats de l'injection de fautes de haut-niveau triés dans ces catégories, il est possible d'extraire des propriétés définies sous forme d'intervalles de paramètres critiques, selon les intervalles X ou I qui conduisent à une erreur critique.

Soit e la fonction qui modélise l'erreur d'un signal X en sortie en fonction de l'amplitude de la faute injectée A : $e(A)$ correspond à la différence entre la valeur de X modifiée par une faute d'amplitude A et la valeur attendue.

La valeur de e est partitionnée en trois sous-ensembles distincts : tolérables, critiques et détectés. Pour simplifier et sans perte de généralité, on suppose que ces ensembles sont trois intervalles tels que l'ensemble des erreurs tolérables est défini par la relation :

$$\forall e \in]0; Ec] \Leftrightarrow e \in T \tag{5.1}$$

l'ensemble des erreurs critiques par la relation :

$$\forall e \in]E_C; E_D] \Leftrightarrow e \in C \tag{5.2}$$

et l'ensemble des erreurs détectées parla relation :

$$\forall e \in]E_D; +\infty] \Leftrightarrow e \in D. \tag{5.3}$$

La fonction est illustrée par la figure 5.6. Cette figure illustre l'erreur e en sortie du système en fonction de l'amplitude de la faute injectée A. On distingue les deux seuils E_C et E_D qui correspondent à la valeur de l'erreur en sortie à partir de laquelle le comportement passe de tolérable à critique et de critique à détecté. Partant de ces ensembles, on souhaite en déduire des propriétés à respecter sur l'amplitude de la faute pour que les erreurs soient soit tolérables, soit détectées. Dans le cas où la fonction e est monotone croissante on a :

$$\forall A_0, A_1, A_0 < A_1 \Leftrightarrow e(A_0) < e(A_1) \tag{5.4}$$

Soit A_C et A_D les amplitudes de fautes injectées telles que :

$$e(A_C) = E_C,\ e(A_D) = E_D. \tag{5.5}$$

Soit A_0 une amplitude de faute pour laquelle on obtient un comportement tolérable, alors :

$$e(A_0) \in T \tag{5.6}$$

comme e est une fonction monotone croissante, on en déduit

$$A_O \in]0; A_C] \tag{5.7}$$

De la même manière si A_1 est une amplitude de faute pour laquelle on obtient un comportement détecté, on a alors :

$$A_1 \in]A_D; +\infty] \tag{5.8}$$

La fonction e n'est pas une fonction définie formellement, nous ne connaissons sa valeur que par simulation. Il ne nous est donc pas possible de calculer précisément les valeurs de A_C et A_D. Nous cherchons donc deux approximations A'_C et A'_D de A_C et A_D telles que :

$$A \notin]A'_C; A'_D[\Rightarrow (e(A) \notin C). \tag{5.9}$$

Il existe une infinité d'intervalles $]A'_C; A'_D[$ possibles avec : $A'_C \in]0; A_C[$ et $A'_D \in]A_D; +\infty[$. Nous recherchons l'intervalle le plus proche possible de $]A_C; A_D[$.

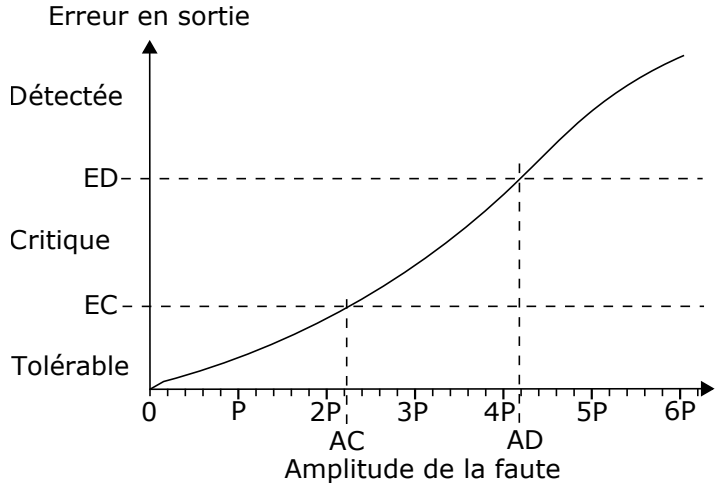

FIGURE 5.6 – *Erreur en sortie du système en fonction de l'amplitude de l'erreur injectée sur un signal sig.*

Pour calculer les deux valeurs A'_C et A'_D, on choisit d'injecter des fautes selon une modification d'amplitude en pourcentage comprise dans l'intervalle $]0; +6P\%]$, par pas de P%, pour chercher les points à partir desquels les comportements en sortie changent. On veut une valeur de A'_C la plus grande possible, et une valeur de A'_D la plus petite possible, pour que l'intervalle soit le plus proche possible de $]A_C; A_D[$. Dans notre exemple, l'injection de faute nous conduira aux résultats reportés dans le tableau 5.1. On sait donc que les seuils A_C et A_D se situent entre $2P$ et $3P$, et entre $4P$ et $5P$. Ces résultats nous permettent de choisir $A'_C = 2P$ et $A'_D = 5P$. On ne sait pas ce qu'il se passe entre ces points, donc l'intervalle de criticité déduit à partir de l'injection de faute est l'intervalle : $]3P\%; 5P\%[$. Pour raffiner cet intervalle pour être plus proche de A_C et A_D, on peut faire une recherche

TABLEAU 5.1 – *Injection de faute sur sig*

signal \ *A*	**0**	**P**	**2P**	**3P**	**4P**	**5P**	**6P**
sig	Silencieuse	Tolérable	Tolérable	Critique	Critique	Détectée	Détectée

par dichotomie, il faudra alors raffiner le pas et refaire une injection de haut-niveau dans les intervalles $]2P;3P[$ et $]4P;5P[$ avec un nouveau pas p, où $p < P$.

Si la fonction $e(A)$ évolue de manière non monotone, le fonctionnement par intervalles deviendra plus complexe. La figure 5.7 illustre une fonction non monotone. La même injection de fautes donnerait les résultats résumés dans le tableau 5.2. Sachant que le concepteur ne connaît pas nécessairement l'allure de la courbe $e(A)$ les résultats obtenus ne permettent pas de déduire des intervalles de criticité simplement. L'extraction de propriétés devient donc plus complexe. Nous nous concentrerons dans la suite sur le cas des fonctions monotones.

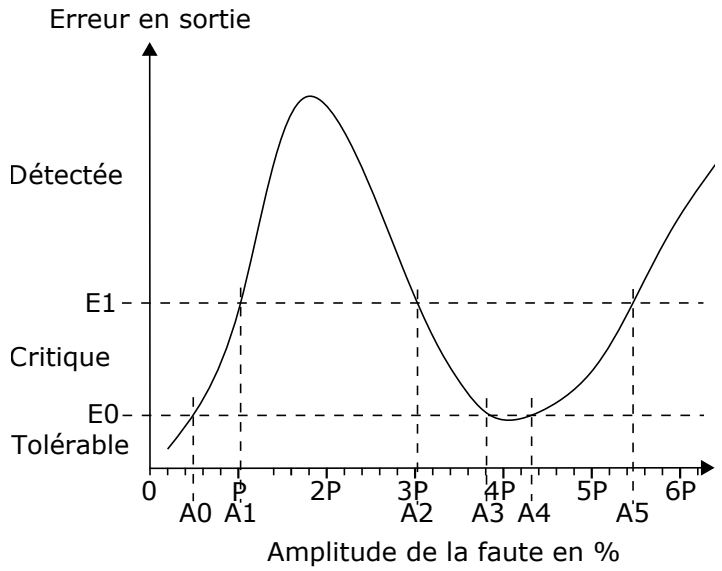

FIGURE 5.7 – *Erreur en sortie du système en fonction de l'amplitude de l'erreur injectée sur un signal sig, pour une fonction non monotone.*

TABLEAU 5.2 – *Injection de faute sur sig*

signal \ *A*	**0**	**P**	**2P**	**3P**	**4P**	**5P**	**6P**
sig	Silencieuse	Critique	Détectée	Détectée	Tolérable	Critique	Détectée

5.3.2 Illustration sur le cas d'étude : Étude de la monotonie, exemple de classification de fautes de haut-niveau, et extraction d'une propriété

Étude de la monotonie de l'erreur en sortie en fonction de l'amplitude de faute injectée La fonction qui définit la valeur $Xnum$ calculée en sortie en fonction de Ft et Fx est $Xnum(Ft, Fx)$. Si Ft est constante, on a $Xnum(Fx)$ fonction monotone, croissante. De même, Si Fx est constante, on a $Xnum(Ft)$ fonction monotone croissante. On distingue alors deux catégories de fautes :

— Les fautes sur les sorties des compteurs, qui ont une influence directe sur Ft et Fx calculées.

— Les fautes sur les sorties des diviseurs, qui ont une influence sur le temps pour calculer une donnée.

En sortie des compteurs, prenons l'exemple du compteur $CounterX$. La sortie Fx est calculée par le bloc $CALC$ avec la formule suivante :

$$Fx = \frac{N_x}{N_{X,ref}} * F_{ref} \tag{5.10}$$

Supposons que Ft soit constante .La monotonie de $Xnum(Fx)$ lorsque Ft est constante donne :

$$F_{x1} < F_{x2} \Rightarrow X_{num}(F_{x1}) < X_{num}(F_{x2}). \tag{5.11}$$

De plus 5.10 donne :

$$N_{x1} < N_{x2} \Rightarrow F(N_{x1}) < F(N_{x2}). \tag{5.12}$$

D'où (5.11 et 5.12) :

$$N_{X1} < N_{X2} \Rightarrow X_{num}(N_{X1}) < X_{num}(N_{X2}). \tag{5.13}$$

Et donc en raisonnant sur l'erreur en sortie sur X_{num} :

$$e_X(N_{Xerr}) = X_{num}(N_{Xerr}) - X_{num}(N_x) \tag{5.14}$$

$$N_{Xerrn} = N_x * (1 + A_n) \tag{5.15}$$

Donc :

$$e_X(N_{Xerr1}) - e_X(N_{Xerr2}) = X_{num}(N_{Xerr1}) - X_{num}(N_{Xerr2}) \tag{5.16}$$

D'où :

$$e_X(N_{Xerr1}) < e_X(N_{Xerr2}) \Leftrightarrow X_{num}(N_{Xerr1}) < X_{num}(N_{Xerr2}) \tag{5.17}$$

Et 5.13 donne :

$$A_1 < A_2 \Leftrightarrow N_{Xerr1} < N_{Xerr2} \Leftrightarrow X_{num}(N_{Xerr1}) < X_{num}(N_{Xerr2}) \Leftrightarrow e_X(N_{Xerr1}) < e_X(N_{Xerr2}) \tag{5.18}$$

Donc en simplifiant :

$$A_{NXerr1} < A_{NXerr2} \Rightarrow e_X(A_{NXerr1}) < e_X(A_{NXerr2}). \tag{5.19}$$

D'où $e_X(A_{Nx})$ fonction monotone croissante. On a donc :

— $e_X(A_{Nx})$ fonction monotone croissante car $Fx(N_x)$ est une fonction monotone croissante et $Xnum(Fx)$ également.

— $e_X(A_{NX,ref})$ fonction monotone décroissante car $Fx(N_{X,ref})$ est une fonction monotone décroissante et $Xnum(Fx)$ une fonction monotone croissante.

Le fonctionnement pour le compteur $CounterT$ est similaire.

En sortie des diviseurs, prenons alors l'exemple de $DIVC$. Une erreur sur $F_{windowX}$ a simplement une influence sur la fenêtre de comptage du compteur $CounterX$. On déduit alors de la même manière $e_{temps}(F_{windowX})$ fonction monotone décroissante.

Pour notre cas d'étude, un relevé de X prend 300ms (cf. spécification système décrite dans 3.1.2). On définit le modèle de faute de manière à ce qu'il couvre un temps supérieur à un relevé de X, soit 300ms. Les instants d'injections choisis sont : 0ms, 50ms, 100ms, 150ms, 200ms, 250ms et 300ms. Les instants d'injections choisis correspondent aux instants où les données transitent, on choisit ces instants par rapport à la durée des fonction $wait()$ des blocs. On veut injecter au moment où les données transitent car c'est le moment où la faute injectée peut se propager dans le système. Si on injecte à d'autre moment, les processus des blocs sont dans des $wait()$ et sont en attente. Les fautes ne sont donc pas prises en compte.

Extraction d'intervalles critiques On a choisi de balayer l'amplitude d'injection sur une plage de -50% à +50% par pas de 10%. Cette première injection de faute nous permet de faire une exploration rapide pour voir où on voudra raffiner le pas. Les résultats de cette première injection de faute Le tableau 5.3 illustre les résultats de cette injection de faute.

TABLEAU 5.3 – *Extrait des résultats de l'injection de fautes sur Nx*

T \ I	0ms	50ms	100ms	150ms	200ms	250ms	300ms
Nx							
-50%	S	S	D	S	D	S	D
... %	...	...	...	...	...	...	...
-10%	S	S	D	S	D	S	D
0%	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
+10%	S	S	C	S	C	S	C
+20%	S	S	D	S	D	S	D
... %	...	...	...	...	...	...	...
+50%	S	S	D	S	D	S	D

On observe que les zones de changements sont entre -10% et 0%, entre 0% et 10% et entre 10% et 20%. On choisira donc de faire une seconde injection de faute sur la plage d'amplitude -10% à +20% par pas de 1%. Les résultats de cette seconde injection seront présentés dans le tableau 5.4

TABLEAU 5.4 – *Extrait des résultats de l'injection de fautes sur Nx avec un pas raffiné*

T \ *I*	**0ms**	**50ms**	**100ms**	**150ms**	**200ms**	**250ms**	**300ms**
Nx							
-10%	S	S	D	S	D	S	D
... %	...	...	...	...	...	...	...
-3%	S	S	D	S	D	S	D
-2%	S	S	C	S	C	S	C
-1%	S	S	C	S	C	S	C
0%	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
+1%	S	S	C	S	C	S	C
...	...	...	...	...	...	...	...
+12%	S	S	C	S	C	S	C
+13%	S	S	D	S	D	S	D
...	...	...	...	...	...	...	...
+20%	S	S	D	S	D	S	D

On a injecté au total 37 amplitudes de fautes pour 7 instants d'injections.

Cette dernière classification nous permet de déduire des propriétés critiques sur le signal *Nx* sous forme d'intervalles. Par exemple, des intervalles de paramètres critiques correspondant au tableau 5.4 sont les suivants :

$$]-3\%;0\%[\cup]0\%;+13\%[\times]50ms;150ms[\tag{5.20}$$

En effet, on peut voir que toutes les fautes injectées d'amplitude comprise entre -2% et +12% sont critiques. On sait que pour les amplitudes -3% et +13% on sort de l'intervalle critique. On choisit donc les intervalles larges]-3% ;0%[et]0% ; +13%[car on ne sait pas ce qu'il se passe entre les amplitudes $]-3\%;-2\%[$, $]-1\%;0\%[$, $]0\%;+1\%[$ et $]+12\%;+13\%[$. Le raisonnement est similaire pour déduire l'intervalle de temps.

5.4 Résultats sur le cas d'étude

5.4.1 Présentation des résultats

À partir de la spécification système, de la description RTL déjà disponible, et du code assembleur du micro-contrôleur, le modèle haut -niveau TLM AT a été implémenté en SystemC. La validation du modèle a été effectuée grâce à la simulation de 90 valeurs de références, la différence entre la valeur X calculée et la valeur obtenue est toujours inférieur à 0,1%. Une campagne d'injection de fautes de haut-niveau a été appliquée sur le modèle TLM AT du cas d'étude : 1120 fautes de haut-niveau ont été injectées. Chaque faute correspond à un quadruplé (L,T,I,D) :

— 8 signaux d'interfaces (L) : tous les signaux d'interfaces dans la partie matérielle du système (5.1 sont visés (*e.g*, $diva\_to\_divb$, $divb\_to\_divc$, $sig_{windowX}$, $Sig_{windowT}$, N_X, $N_{ref,X}$, N_T, $N_{ref,T}$, X_num). Le but est d'analyser l'impact des fautes sur le comportement du FPGA.

— un type de fautes (T) pour 20 amplitudes : chaque signal est modifié pour couvrir un intervalle de déviation compris entre -50% et +50% par pas de 10%. La taille du pas peut être réduite pour affiner l'analyse dans les expérimentations suivantes. Le but est de viser large dans un premier temps, puis de procéder par dichotomie pour affiner.

— 7 instants d'injection (I) : l'intervalle d'injection est de 50 ms Les instants d'injection sont donc 0 ms, 50 ms, 100 ms, ..., 300 ms. Aucune faute n'a été injectée après 300 ms car c'est le temps d'un cycle de calcul complet. Néanmoins, cet intervalle peut aussi être affiné ensuite pour améliorer l'analyse.

— Durée d'injection (D) : Les fautes sur chaque signal ont été injectées jusqu'à que le signal soit réécrit.

Le temps de simulation dure moins de 3 minutes pour la totalité des fautes de haut-niveau à injecter. Le tableau 5.5 présente les résultats de la simulation de faute pour quelques signaux du modèle de haut-niveau du cas d'étude. Les résultats globaux sont les suivants :

— 509 fautes tolérables (T : sortie du système différente de la simulation de référence, mais respectant la spécification),

— 343 erreurs détectées (D),

— 227 erreurs critiques (C).

Comme la probabilité d'occurrence de chaque faute de haut-niveau est inconnue, il est impossible de déduire la probabilité d'un évènement critique à partir de ces résultats. Cette déduction sera possible dans l'étape de simulation RTL. Néanmoins, on peut observer l'intérêt de prendre en compte la spécification pour identifier les blocs à analyser : parmi les 1079 fautes créant une sortie différente de la simulation de référence, seulement 227 sont réellement critiques pour le système.

5.4.2 Analyse des diviseurs

Cette partie présente une analyse des résultats de l'injection de fautes dans les diviseurs. Elle vise à identifier si des intervalles critiques existent, à étudier les mécanismes de détections impliqués, et la vulnérabilité des blocs. La partie supérieure du tableau 5.5

TABLEAU 5.5 – *Injection de fautes en sortie des diviseurs*

T \ *I*	**0ms**	**50ms**	**100ms**	**150ms**	**200ms**	**250ms**	**300ms**
diva_to_divb							
-50%	D	D	D	D	D	D	D
… %	…	…	…	…	…	…	…
-12%	D	D	D	D	D	D	D
-11%	T	T	T	T	T	T	T
… %	…	…	…	…	…	…	…
-1%	T	T	T	T	T	T	T
+1%	S	S	S	S	S	S	S
… %	…	…	…	…	…	…	…
+5%	S	S	S	S	S	S	S
+6%	D	D	D	D	D	D	D
… %	…	…	…	…	…	…	…
+50%	D	D	D	D	D	D	D
divb_to_divc							
-50%	D	D	D	D	D	D	D
… %	…	…	…	…	…	…	…
+50%	D	D	D	D	D	D	D
Injection sur F_windowX							
-50%	D	D	D	D	D	D	D
… %	…	…	…	…	…	…	…
-1%	D	D	D	D	D	D	D
+1%	T	T	T	T	T	T	T
… %	…	…	…	…	…	…	…
+8%	T	T	T	T	T	T	T
+9%	C	C	C	C	C	C	C
… %	…	…	…	…	…	…	…
+50%	C	C	C	C	C	C	C

TABLEAU 5.6 – *F_windowT*

T \ I	0ms	50ms	100ms	150ms	200ms	250ms	300ms
F_windowT							
-50%	S	S	S	S	S	S	S
... %	...	...	...	...	...	...	...
+19%	S	S	S	S	S	S	S
+20%	D	D	D	D	S	S	S
... %	...	...	...	...	...	...	...
+50%	D	D	D	D	S	S	S

illustre les résultats de la simulation de fautes de haut-niveau appliquée au signal d'interface *diva_to_divb*. Pour chaque faute injectée, les résultats de la simulation sont reportés dans le tableau pour identifier des intervalles. Les résultats sont triés en trois catégories comme défini dans la section précédente. Ce tableau montre que l'injection de fautes conduit à :

— Des erreurs détectées (D) pour une amplitude de fautes entre $[-50\%;-12\%]$ et $[+6\%;+50\%]$. Ces intervalles sont indépendants de l'instant d'injection.

— Des fautes silencieuses (S) ou tolérables (T) pour une amplitude de fautes entre $[-11\%;+5\%]$

Différents mécanismes de détections interviennent :

— Pour des amplitudes de fautes entre$[-50\%;-12\%]$, le mécanisme détectant les fautes est $M1$ (voir Chapitre 3.1.2) (le micro-contrôleur rate une valeur sur les registres Nx et $N1$). De plus, le micro-contrôleur détecte également une période calculée en dehors de la table de point ($M3$).

— Pour des amplitudes de fautes entre $[+6\%;+50\%]$, le watchdog du micro-contrôleur ($M4$) indique que le timing n'est pas respecté.

L'injection de fautes sur les signaux *diva_to_divb*, *divb_to_divc*, et $F_{windowT}$ produit uniquement des erreurs conduisant à des erreurs détectées. Il n'y a donc pas d'intervalles critiques pour ces signaux. Néanmoins, le signal $F_{windowX}$ correspondant à la sortie du diviseur *DIVC* a un intervalle critique. Cette analyse au niveau des diviseurs permet donc d'identifier des diviseurs plus vulnérables que d'autres.

5.4.3 Analyse des compteurs

Cette partie présente l'analyse de l'injection de fautes sur les sorties des compteurs $Compteur_X$ et $Compteur_T$. De la même manière que pour les diviseurs, les intervalles critiques sont identifiés, les mécanismes de détections et la vulnérabilité des blocs étudiés. La partie supérieur du tableau 5.6 présente les résultats de l'injection de fautes sur le signal Nx du bloc $Compteur_X$. L'injection de fautes crée des erreurs critiques pour des

TABLEAU 5.7 – *Injection de fautes sur Nx et Nt*

T \ *I*	**0ms**	**50ms**	**100ms**	**150ms**	**200ms**	**250ms**	**300ms**
Nx							
-50%	S	S	D	S	D	S	D
… % …	…	…	…	…	…		
-3%	S	S	D	S	D	S	D
-2%	S	S	C	S	C	S	C
… %	…	…	…	…	…	…	…
+12%	S	S	C	S	C	S	C
+13%	S	S	D	S	D	S	D
… %	…	…	…	…	…	…	…
+50%	S	S	D	S	D	S	D
Nt							
-50%	D	D	D	D	D	D	D
… %	…	…	…	…	…	…	…
-41%	D	D	D	D	D	D	D
-40%	C	C	C	C	C	C	C
… %	…	…	…	…	…	…	…
+7%	C	C	C	C	C	C	C
+8%	D	D	D	D	D	D	D
… %	…	…	…	…	…	…	…
+50%	D	D	D	D	D	D	D

TABLEAU 5.8 – *Injection de fautes sur $N_{ref,X}$ et $N_{ref,T}$*

T \ I	**0ms**	**50ms**	**100ms**	**150ms**	**200ms**	**250ms**	**300ms**
$N_{ref,X}$							
-50%	S	S	D	S	D	S	D
… %	…	…	…	…	…	…	…
-3%	S	S	D	S	D	S	D
-2%	S	S	C	S	D	S	D
-1%	S	S	C	S	C	S	C
+1%	S	S	C	S	C	S	C
+2%	S	S	C	S	D	S	D
+3%	S	S	D	S	D	S	D
… %	…	…	…	…	…	…	
+50%	S	S	D	S	D	S	D
$N_{ref,T}$							
-50%	D	D	D	D	D	D	D
… %	…	…	…	…	…	…	…
-10%	D	D	D	D	D	D	D
-9%	C	C	C	C	C	C	C
… %	…	…	…	…	…	…	…
+8%	C	C	C	C	C	C	C
+9%	D	D	D	D	C	C	C
+20%	D	D	D	D	D	D	D
… %	…	…	…	…	…	…	…
+50%	D	D	D	D	D	D	D

fautes injectées à 100 ms, 200 ms et 300 ms. A ces instants d'injections, les fautes sont critiques pour toutes les amplitudes dans les intervalles $]-3\%;0\%[\cup]0\%;+13\%[$.

Au même instant d'injection, les erreurs en dehors de cet intervalle d'amplitude ont été détectées par les mécanismes de détections implémentés dans le micro-contrôleur, car créant des périodes en dehors de la table de point. Les injections de fautes sur les signaux N_t et $N_{ref,t}$ ont produit des résultats similaires. Seule l'amplitude de l'intervalle change légèrement.

Injecter une faute a un impact uniquement à 100 ms, 200 ms et 300 ms car cela correspond à l'instant pendant lequel une nouvelle valeur est écrite par le bloc $Compteur_X$, mais n'a pas encore été lue par le micro-contrôleur.

5.4.4 Conclusion du chapitre

Les résultats ont montré les avantages suivants de cette première étape :

— La méthode de simulation de fautes de haut-niveau permet une simulation de faute rapide et tôt dans le flot de conception.

— La prise en compte de la spécification peut être utilisée pour filtrer parmi les comportements erronés, ceux qui conduisent à des comportements critiques. Cela permet de filtrer 79% des comportements erronés.

— La méthode de simulation de fautes de haut-niveau permet d'extraire facilement des propriétés critiques sous formes d'intervalles critiques. Ces intervalles sont définis sur deux dimensions que sont le temps et l'amplitude de la faute. Ces intervalles critiques permettent de définir facilement des propriétés blocs par blocs pour filtrer les fautes RTL conduisant à des comportements critiques. En effet, seulement les fautes RTL qui se propageront sur les sorties en tombant dans ces intervalles critiques seront considérées comme telles.

— Les mécanismes de détection redondants peuvent être identifiés. Par exemple pour le diviseur *DIVA*, deux mécanismes de détection détectent l'erreur créée par les fautes d'amplitudes entre -50% et -12%. Des analyses supplémentaires peuvent permettre de valider quel est le mécanisme le plus efficace.

De plus, en comparaison de l'AMDEC traditionnellement réalisée au niveau fonctionnel, la simulation de haut-niveau a permis d'automatiser l'analyse fonctionnelle réalisée manuellement, et donc d'étudier plus de fautes possibles.

Les intervalles critiques extraits dans cette étape seront utiles lors de l'étape de simulation RTL, mais l'extraction de ces intervalles est basée sur un modèle du système de haut-niveau, et un modèle de faute de haut-niveau dont le réalisme n'est pas prouvé. C'est pourquoi il faudra valider ce modèle de haut-niveau pour valider les résultats obtenus.

6

Etape 2 : Simulation de fautes RTL

Sommaire

Cette étape intervient lorsque les choix de partitionnement du circuit sont faits, et que la description des blocs RTL est disponible. Chaque bloc pour lequel des fautes de haut-niveau en sortie conduisent à des comportements critiques lors de la simulation de haut-niveau est simulée avec injection de fautes RTL.

L'intérêt de faire une simulation de fautes RTL est d'obtenir des résultats plus réalistes, et donc d'identifier les fautes RTL (*e.g.,* bit-flips dans des flips-flops ou multiples bit-flips dans des registres) qui sont critiques pour le système. Pour cela, on réutilise les propriétés sur les signaux d'interfaces extraites dans l'étape de simulation de haut-niveau. Ces propriétés permettent de définir des assertions sur les signaux d'interfaces. L'étape de simulation de fautes RTL décrite dans la figure 4.5 du chapitre 4.2 prend en entrée le modèle RTL de chacun des blocs du circuit, un modèle de faute RTL, et les propriétés extraites de l'étape de simulation de haut-niveau. Elle donne en sortie une classification des fautes RTL. L'implémentation de l'étape de simulation RTL se décompose de la façon suivante :

— Modélisation des blocs : tout d'abord, il faut modéliser les blocs au niveau RTL, définir les scénarios à simuler et traduire les propriétés de l'étape de simulation haut-niveau en assertions.

— Injection de fautes : l'injection de faute de cette étape de simulation RTL ne peut être exhaustive car le nombre de fautes possibles est trop grand. Après avoir défini le modèle de faute RTL utilisé, il sera nécessaire de définir le nombre de fautes à injecter pour obtenir un bon compromis entre temps simulation et précision des résultats de la simulation de fautes.

— Extraction des résultats : À l'issue de la simulation, les fautes RTL injectées sont classées en différentes catégories selon leurs effets sur le système. À partir de cette classification, une analyse permet alors d'obtenir des informations utiles à la FMEA.

Une dernière section présentera l'étape de simulation RTL appliquée au cas d'étude.

6.1 Modélisation de l'environnement de simulation

Le banc de test permet de modéliser l'environnement de simulation. La figure 6.1 schématise les différentes entrées nécessaires à la définition du banc de test. Le banc de test permettra de lancer la simulation. La définition du banc de test nécessite donc :

— La déclaration du bloc à simuler : on utilisera deux fois sa déclaration, une servant de référence, la seconde sur laquelle les fautes seront injectées.

— La définition du scénario de test.

— L'implémentation grâce à des assertions PSL des propriétés extraites dans l'étape de simulation de haut-niveau.

L'implémentation de ces trois composantes du banc de test est décrite dans la suite de la section.

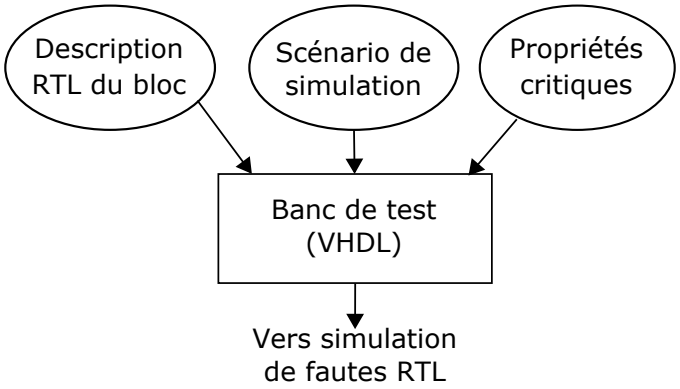

FIGURE 6.1 – *Entrées pour modéliser le banc de test de l'étape 2.*

6.1.1 Modélisation des blocs au niveau RTL et définition de scénarios

Définition des blocs RTL Une fois les décisions de partitionnement prises bloc par bloc (implémentation matérielle ou logicielle), chaque bloc est implémenté au niveau RTL. Chaque bloc RTL doit être équivalent au niveau des entrées/sorties à un bloc du modèle de haut-niveau. De cette façon, il est possible de faire une analogie entre la modélisation RTL et TLM, et d'en extraire des équivalences. Comme les assertions extraites portent sur les signaux du modèle de haut-niveau, qui ont parfois des types différents, il est souvent nécessaire de faire des conversions dans le banc de test afin de pouvoir adapter le type de signal utilisé pour la modélisation TLM et celui utilisé au niveau RTL.

Scénarios simulés Dans le but de réduire le temps de simulation, les scénarios simulés au niveau RTL ne doivent pas forcément être de la même durée que le scénario simulé au niveau système. En effet, certains blocs RTL (compteurs, diviseurs, etc.) ont des fonctionnements cycliques sur des durées largement inférieures. Il est donc inutile de vouloir simuler les mêmes scénarios que pour le modèle haut-niveau complet. Les scénarios sont donc adaptés à l'échelle de temps de chaque bloc, ce qui permet de réduire considérablement les temps de simulation de cette étape.

6.1.2 Définition des propriétés à partir des résultats de l'étape de simulation de haut-niveau

Afin de classifier les fautes RTL selon leurs effets sur le système, les propriétés extraites dans l'étape de simulation haut-niveau sous formes d'intervalles sont traduites en assertions et intégrées au banc de test. Le langage PSL [72] (Property Specification Language) est un langage de description formelle avec lequel on peut spécifier des propriétés temporelles pour les circuits matériels. Il peut être utilisé avec des langages de description matérielle comme le RTL ou le Verilog, ou avec des langages de description de haut-niveau comme le SystemVerilog ou le SystemC. Les assertions sont implémentées en langage PSL.

Soit $[x1;x2] \times [t1;t2]$ définissant un intervalle critique. Où Soit $[x1;x2]$ représente l'intervalle en amplitude de faute, et $[t1;t2]$ l'intervalle de temps. Soit *window* le signal qui définit la fenêtre de temps de validité de la propriété. *window* est défini par la fonction caractéristique du couple $[t1;t2]$ (1 si le temps actuel est dans la fenêtre de temps, 0 sinon).

Ci-dessous, la propriété PSL vérifiant si le comportement fautif du bloc RTL en présence de fautes est équivalente à l'intervalle critique $[x1;x2] \times [t1;t2]$ identifié dans l'étape 1 :

$$\texttt{Assert always } window \texttt{ -> } (s' \bigtriangledown (s*(1+x1))) \texttt{ or } (s' \bigtriangledown (s*(1+x2)))$$

Où s est la valeur du signal l sans corruption, s' sa valeur corrompue, et $\bigtriangledown$ un opérateur relationnel (*e.g.*, $=$, $\geq$ et $\leq$, ...).

Le banc de test dans le modèle RTL est implémenté en se basant sur ces propriétés formelles (équivalentes aux intervalles identifiés à haut-niveau comme critiques). La figure 6.2 illustre un exemple de l'implémentation de l'intervalle.

```
1  -- generation de la fenetre de temps
2
3  gen_window : process
4  begin
5  window <= '0' ;
6  wait for t1 s ;
7  window <= '1' ;
8  wait for (t2-t1) s ;
9  window <= '0' ;
10 end process gen_window ;
11
12 -- assertion pour filtrer les comportements
13
14 --psl assert always window -> ((s <= s'*(1+x1)or (s = s')or(s => s'*(1+
       x2)) report "ERR_ASSERTION";
```

FIGURE 6.2 – *Exemple de l'implémentation d'un intervalle VHDL et PSL*

6.1.3 Illustration sur le cas d'étude : exemple de définition de propriétés PSL pour un bloc RTL

L'étape de simulation de haut-niveau nous a permis d'identifier des intervalles de criticité sur les signaux d'interface. Pour les blocs *DIVA*, *DIVB* et *SYNC*, nous n'avons pas observé de fautes de haut-niveau conduisant à des comportements critiques. Sur les signaux de sorties des blocs *DIVC*, *CounterX* et *CounterT*, plusieurs intervalles critiques ont été identifiés. Ce sont donc les trois blocs que nous étudierons lors de cette étape.

La description de chacun des blocs RTL est déjà disponible dans le cas d'étude fourni. On réutilise ces descriptions. Il reste alors à définir pour ces trois blocs le scénario à simuler, puis à implémenter les assertions PSL correspondantes.

Définition du banc de test de *DIVC* Dans notre modèle de haut-niveau, le diviseur $DIVC_{TLM}$ prend en entrée une valeur de fréquence F_{in} sous forme d'un float, et ressort cette fréquence divisée en sortie F_{out}.

Au niveau RTL, le signal d'entrée de $DIVC_{RTL}$ est un signal logique sig_{in} de fréquence F_{in} et le signal de sortie un signal logique sig_{out} de fréquence F_{out}. Il est alors nécessaire :

- En entrée de faire la conversion entre la valeur de fréquence F_{in} connue que nous voulons traduire en signal oscillant sig_{in}.
- En sortie de déduire la fréquence F_{out} correspondante au signal sig_{out}.

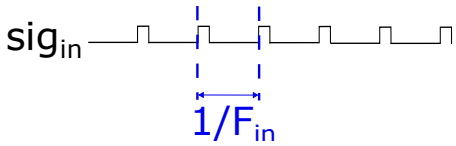

FIGURE 6.3 – *Correspondance entre un signal oscillant et le flottant représentant sa fréquence.*

Le schéma de la figure 6.3 illustre la correspondance entre les deux types de signaux et donc la conversion qu'il y a à faire dans un sens ou dans l'autre. Le signal au niveau RTL est un signal oscillant *sig*, et son équivalent au niveau TLM se modélise sous forme d'un réel F représentant sa fréquence avec :

$$F = \frac{1}{T}$$

où T est le temps entre deux fronts montant sur *sig*.

```
1  --generation de sig_in
2  gen_sig_in : process
3  begin
4  sig_in <='1';
5  loop
6  wait for (T_in/2) ns;
7  sig_in<= not(sig_in);
8  end loop;
9  end process gen_clk;
10 -- generation de T_out
11 gen_T_out :process(sig_out)
12 begin
13     if sig_out'event and sig_out='1' then
14         temp2 <= temp1;
15         temp1 <= now;
16     end if;
17   T_out<=temp1-temp2;
18 end process gen_T_out;
19
20 --psl assert always (window-> (T_out>=(T_out_gold*0.8))) report "
     ERR_ASSERTION";
21 --psl assert always (window-> (T_out<=(T_out_gold*1.1))) report "
     ERR_ASSERTION";
```

FIGURE 6.4 – *Extrait du code VHDL pour l'écriture du banc de test de DIVC*

Le code VHDL 6.4 illustre l'implémentation de ces deux conversions dans le banc de test, ainsi que la définition des propriétés PSL qui vérifient les intervalles critiques. En entrée, un processus *gen_sig_in* (lignes 2 à 9) permet simplement de générer un signal d'impulsion de période T_{in}. La période T_{in} correspondant à la fréquence F_{in}. En sortie, la conversion se fait dans l'autre sens. Il est nécessaire d'extraire la fréquence correspondante au signal sig_{out}. Pour cela, on utilise un processus *gen_T_out* (lignes 11 à 18) sensible aux fronts montants du signal sig_{out}. À chaque front montant (ligne 13) on sauvegarde la valeur du temps actuel dans une variable temporaire *temp*1 (ligne 16), et on fait la différence avec la valeur précédemment stockée (*temp*2), cela nous permet d'obtenir le temps écoulé *T_out* entre deux fronts montants de Sig_{out} (ligne 17).

Les assertions PSL correspondant aux intervalles critiques extraits à l'étape précédente sont ensuite implémentées (lignes 20 et 21). Elles permettent de notifier un message d'erreur "ERR_ASSERTION" si la sortie *T_out* tombe dans l'intervalle critique :

$$[-\infty; -20\%[\cup] + 10\%; -\infty]$$

Définition du banc de test de CounterX À haut-niveau, le bloc $CounterX_{TLM}$ prend en entrée trois flottants représentants des valeurs de fréquences. En sortie sont envoyées deux valeurs entières N_X et $N_{ref,X}$, sur des canaux de communication notifiant automatiquement lorsque une nouvelle valeur est écrite sur le canal.

Au niveau RTL, le bloc $CounterX_{RTL}$ prend en entrée des signaux logiques Sig_{ref}, $Sig_{CounterX}$ et Sig_X oscillant à différentes fréquences, la conversion des signaux oscillants vers les valeurs flottantes est réalisée sur le même principe que pour *DIVC*. En sortie, les valeurs comptées sont stockées dans cinq registres sur huit bits, et une interruption permet de notifier au micro-contrôleur de lire les valeurs des registres. L'entier N_x correspond aux registres *reg*1 et *reg*2 et l'entier $N_{ref,X}$ aux registres *reg*3, *reg*4 et *reg*5. La figure 6.5 illustre la correspondance.

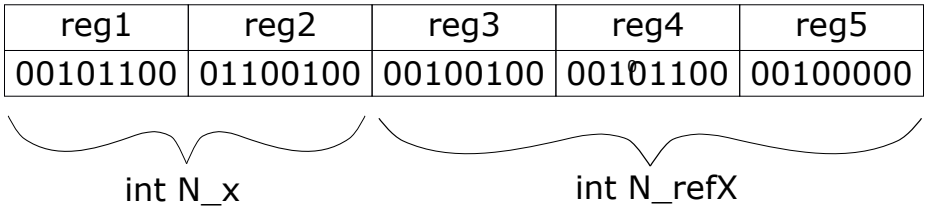

FIGURE 6.5 – *Correspondance entre les registres de* $CounterX_{RTL}$ *et les entiers* N_x *et* $N_{X,ref}$.

Le code VHDL de la figure 6.6 illustre l'implémentation dans le banc de test pour convertir les valeurs des registres en entiers, et les assertions sur les entiers déduites des intervalles obtenus dans l'étape de simulation haut-niveau. Le processus *gen_Nx* fait la conversion des deux registres *reg*1 et *reg*2 concaténés, en entier *N_X* (lignes 1 à 5). Le processus *gen_N_refX* fait la conversion des deux registres *reg*3, *reg*4 et *reg*5 concaténés, en entier *N_refX* (lignes 6 à 9). Enfin, les lignes 11 et 12 implémentent les assertions PSL déduites des intervalles critiques. La première assertion (ligne 11) notifie un message d'erreur lorsque *N_X* est erroné et est dans l'intervalle :

$$[-50\%; 0\%[\cup]0\%; +10\%]$$

La seconde assertion (ligne 12) notifie un message d'erreur lorsque *N_refX* est erroné et est dans l'intervalle :

$$[-10\%; 0\%[\cup]0\%; +10\%]$$

Définition du banc de test de *CounterT* Le développement du banc de test pour *CounterT* suit le même principe que pour *CounterX*.

```
1  gen_Nx: process(reg1_out,reg2_out)
2  begin
3      N_X <= real(conv_integer(reg2_out & reg1_out));
4  end process;
5
6  gen_N_refX: process(reg3_out,reg4_out,reg5_out)
7  begin
8      N_refX <= real(conv_integer(reg5_out & reg4_out& reg3_out));
9  end process;
10
11 --psl assert always (window -> ((Nx<=(Nx_gold*0.5))or(Nx=Nx_gold)or(Nx
       >=(Nx_gold*1.10)))) report "ERR_ASSERTION";
12 --psl assert always (window -> ((N_refX<=(N_refX_gold*0.90))or(N_refX=
       N_refX_gold)or(N_refX>=(N_refX_gold*1.10)))) report "ERR_ASSERTION
       ";
```

FIGURE 6.6 – *Extrait du code VHDL pour l'écriture du banc de test de CounterX*

6.2 Injection de fautes RTL

Cette section définit la méthode d'injection de fautes de l'étape de simulation RTL. Elle comporte deux parties distinctes :

— La définition du modèle de fautes à injecter, ce qui permet de définir toutes les fautes possibles à injecter.

— L'injection statistique de fautes : injecter la totalité des fautes possibles définies dans notre modèle de fautes n'est pas réalisable avec la complexité des systèmes actuels et la puissance de calcul disponible, la solution est alors de définir le nombre de fautes à injecter grâce à l'injection statistique de fautes.

Ces deux parties sont détaillées dans la suite de la section.

6.2.1 Modèle de fautes RTL

Le modèle de faute au niveau RTL est semblable au modèle de faute au niveau système : une faute f est définie par un quadruplé (l,t,i,d) où :

— l est la localisation : les fautes sont injectées dans toutes les flips-flops et registres.

— t est le type de faute : il définit le type de fautes injectées. Nous considérons d'abord des bit flips uniques, puis des injections multiples pourront être effectuées.

— i est l'instant d'injection : les fautes sont injectées à différents instants d'injections.

— d est la durée de la faute : la faute est injectée jusqu'à ce que la flip-flop ou le registre soit réécrit.

L'injection de fautes au niveau RTL est effectuée à l'aide de commandes simulateurs. Les commandes simulateurs permettent d'injecter des fautes transitoires dans les registres et flips-flops des blocs RTL, sans modification de code, c'est pourquoi cette solution-là est retenue.

Le script tcl présenté dans la figure 6.7 illustre l'injection d'un bit-flip dans le simulateur Modelsim. La faute injectée f est définie par le quadruplé (l,t,i,d), la durée du scénario de simulation est *temps_simu* :

— Simulation jusqu'à l'instant d'injection *i* (ligne 2).

— Lecture de la valeur stockée dans la flip-flop *l* et stockage dans une variable temporaire (ligne 4).

— Injection d'un bit-flip dans la flip-flop (ligne 6).

— Simulation jusqu'à la fin du scénario (ligne 8).

```
1 #simulation pour i secondes
2 run $i s
3 #lecture et stockage de la valeur de ff
4 set $ff_temp [examine 1]
5 #injection d'un bit-flip
6 force deposit $ff [expr $ff_temp ^ 1)]
7 #simulation jusqu a la fin
8 run [expr $temps_simu - $i] s
```

FIGURE 6.7 – *Extrait du script tcl permettant d'injecter une faute sur une flip-flop*

6.2.2 Injection statistique de fautes

La complexité des blocs, le nombre de fautes possibles, et les capacités de simulation font qu'il est impossible d'injecter toutes les fautes possibles. L'injection statistique de fautes comme décrite par Leveugle [73] permet de déterminer le nombre de fautes à injecter pour un taux de confiance *c* et une marge d'erreur *e* données. L'injection statistique de fautes considère que les caractéristiques de la population suivent une distribution normale (ici critique ou non critique). En supposant que chaque évènement est équiprobable, une distribution uniforme doit alors être utilisée pour une l'injection aléatoire. Le nombre de fautes à injecter est alors calculé selon les paramètres suivants :

— *N* est la taille initiale de la population, dans notre cas, cela correspond au nombre de fautes possibles, soit le produit du nombre de flips-flops par le nombre de cycles.

— *p* est la proportion estimée qu'un individu dans la population ait la caractéristique donnée. Lorsque p est inconnu, on le fixe à 0.5, car cela correspond à la valeur permettant de maximiser le nombre de fautes n à injecter.

— La marge d'erreur *e* représente la marge d'erreur sur le résultat obtenu. La probabilité exacte qu'une faute ait le comportement voulu se trouve dans l'intervalle [Peval - e ; Peval + e]. Où Peval est le paramètre évalué (*i.e.*, taux d'erreurs critiques, ou silencieuses, *etc.*)

— La valeur *t* correspondante au niveau de confiance *c*. On calcule *t* à partir de *c* avec la formule suivante :

$$\alpha = \frac{1-c}{2} \tag{6.1}$$

La valeur *t* est obtenue grâce à une table de la loi normale centrée réduite, en reportant la valeur de α dans la table. Le niveau de confiance *c* est la probabilité que la valeur calculée soit dans l'intervalle d'erreur. Des valeurs communément choisies sont 90%, 95% ou 99%.

L'équation (6.1) permet de calculer le nombre de fautes à injecter selon les paramètres précédemment décrits.

$$n = \frac{N}{1 + e^2 * \frac{N-1}{t^2 * p * (1-p)}} \tag{6.2}$$

La figure 6.8 nous permet d'illustrer la signification du taux de confiance et de la marge d'erreur à travers un exemple. On effectue une simulation d'injection de fautes. La population est l'ensemble des fautes possibles (bit-flips dans les flip-flops). La caractéristique recherchée est la criticité de chaque faute. Pour un taux de confiance c choisi de 90% et une marge d'erreur e de 5%, le nombre de fautes à injecter est n_i. Un échantillon représente une injection de n_i fautes aléatoires. On effectue dix fois une injections de n_i fautes aléatoires et on regarde la proportion de fautes critiques obtenue pour chacun des échantillon. Le résultat est reporté dans la figure 6.8. Considérons que la proportion exacte de fautes critiques est de 51,5% et est représentée par le trait en rouge. Le taux de confiance de 90% et la marge d'erreur de 5% signifie que dans 90% des cas, la valeur exacte se trouve dans l'intervalle :

$$[Peval_i - 5\%; Peval_i + 5\%]$$

Où $Peval_i$ est le taux d'erreur critique obtenu pour l'échantillon i. Soit ici pour neuf échantillons sur dix. Pour l'échantillon 2, la valeur exacte ne se situe pas dans l'intervalle :

$$[Peval_2 - 5\%; Peval_2 + 5\%]$$

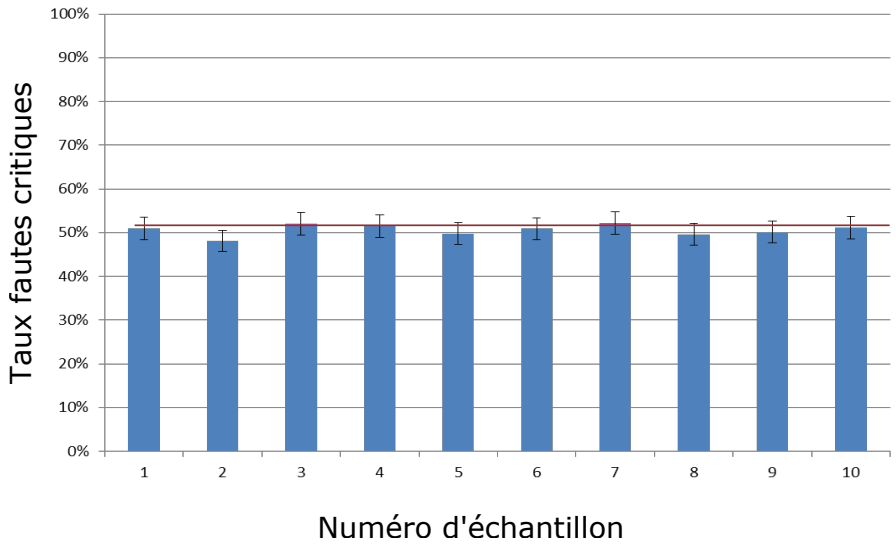

FIGURE 6.8 – *Exemple de résultats d'une injection de fautes pour 10 échantillons différents avec la marge d'erreur représentée pour chaque échantillon.*

6.2.3 Illustration sur le cas d'étude : définition du modèle de faute RTL et injection statistique.

Illustrons l'injection de fautes à travers notre cas d'étude. Nous avons auparavant développé le banc de test comprenant l'implémentation de chaque module RTL, la définition

des scénarios et des propriétés PSL. Afin de lancer la simulation de fautes, il est maintenant nécessaire de définir :

— l'ensemble des fautes possibles selon notre modèle de faute,

— le nombre de fautes à injecter.

Définition des fautes Pour chaque bloc, on définit :

— Toutes les localisations possibles : dans toutes les flips-flops du bloc.

— Tous les instants d'injections possibles : à tous les cycles d'horloge.

— La durée est jusqu'à ce que la flip-flop soit réécrite.

— Le type de faute injecté est un bit-flip.

Cela nous permet de définir un nombre fini N de fautes à injecter. Parmi ces N possibilités, on veut choisir n fautes à injecter aléatoirement.

Injection statistique de fautes Le nombre n de fautes à injecter dépend des contraintes de temps de simulation fixées par le concepteur, ainsi que du temps de simulation d'une faute. Ce temps de simulation d'une faute dépend de la machine sur laquelle est exécutée la simulation, et de la complexité du bloc simulé.

Pour pouvoir définir ce temps de simulation par faute, on lance d'abord une simulation pour chaque bloc RTL à simuler. Cela nous permet de déduire le temps de simulation par faute. Ce temps de simulation par faute nous permet ensuite d'estimer le temps de simulation total, selon le nombre de fautes injectées.

Dans la logique du flot *quick and dirty* présenté dans le chapitre 4, les premières itérations doivent se faire rapidement. Nous avons fixé la contrainte de temps de simulation pour l'étape de simulation RTL à 3h, ce qui permet de faire une première itération de la méthodologie rapidement.

TABLEAU 6.1 – *Calcul des temps de simulation et du nombre de fautes pour la simulation de fautes RTL*

Bloc	Nombre de fautes possibles	Temps simulation RTL par faute	Nombre de fautes pour (c,e) = (95%,5%)	Temps de simulation
DIVA	2.500	1,1 sec	333	6,1 min
DIVB	1.500	1,1 sec	305	5,6 min
DIVC	18.750	1,1 sec	376	6,9 min
SYNC	2,500	1,1 sec	333	6,1 min
CounterX	30.000,000	2 sec	384	12,8 min
CounterT	150.000.000	21 sec	384	134,4 min

Le tableau 6.1 contient les différentes informations qui nous ont permis de fixer le taux de confiance et la marge d'erreur de cette première itération. Le choix de c et e sera expliqué dans la suite.

La 3e colonne du tableau 6.1 donne le temps de simulation d'une faute pour chaque bloc RTL. Elle permet donc de calculer le temps de simulation pour un nombre de fautes à injecter. Le bloc *CounterT* a un temps de simulation élevé. En effet, ce bloc prend en entrée un signal définissant la fenêtre de comptage de fréquence très faible par rapport aux autres blocs du système. L'horloge étant la même pour tous les blocs du système, beaucoup plus de cycles sont simulés, ce qui fait augmenter le temps de simulation.

Dans la Figure 6.9 nous avons représenté le temps de simulation en fonction de la marge d'erreur, pour différents taux de confiance. Pour cela nous avons calculé le nombre de fautes à injecter pour différents couples, et en avons déduit le temps de simulation total pour tous les blocs du système. La ligne en pointillés rouges correspond à la contrainte de temps que nous avons fixé de 3h. Les couples (c,e) sous la ligne en pointillés rouges conviennent. Une valeur classique pour le niveau de confiance est de 95%, cette valeur permettant de réduire le nombre de fautes à injecter tout en gardant une précision élevée : on retiendra alors le couple (95%,5%).

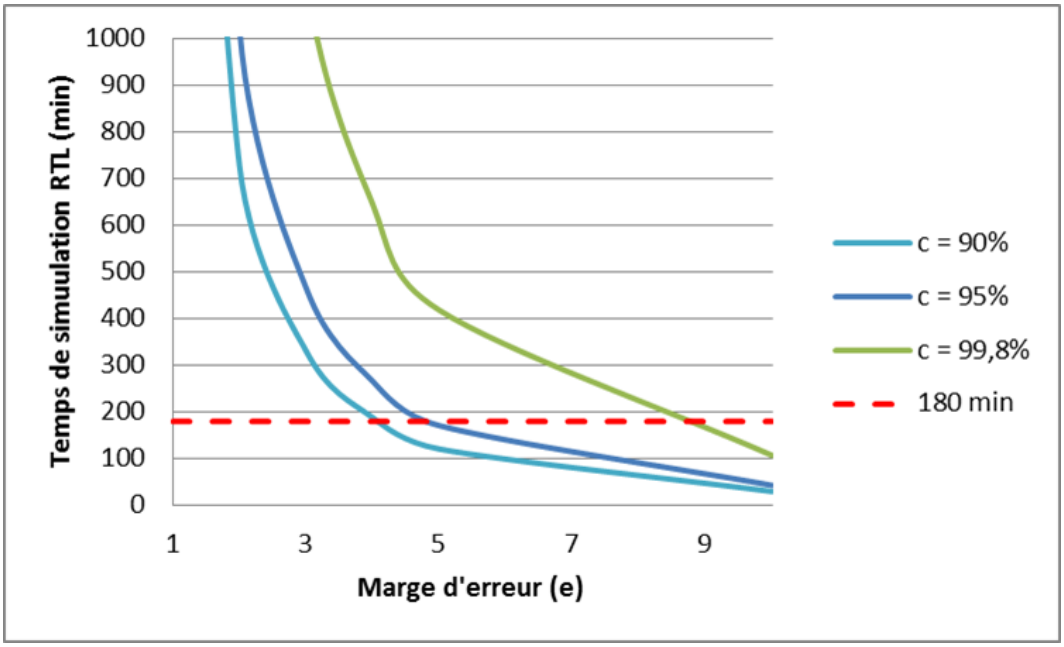

FIGURE 6.9 – *Temps de simulation RTL en fonction de la marge d'erreur, pour différents niveaux de confiance.*

Dans le tableau 6.1, les données calculées à partir de ce couple sont reportées :

— La 4ème colonne présente le nombre *n* de fautes injectées.

— La 5ème colonne présente le temps de simulation par bloc pour simuler les *n* fautes.

6.3 Extraction des résultats

La définition du banc de test et de la campagne d'injection de fautes nous permet de lancer la simulation de fautes pour chacun des blocs. Les résultats de simulations doivent ensuite être traités pour obtenir une classification des fautes RTL. Cette section traite de

la classification des fautes RTL, puis du calcul de taux de SEUs critiques à partir de cette classification. Ce dernier résultat est une information utile à l'AMDEC.

6.3.1 Classification des fautes RTL

Les propriétés PSL nous permettent de classer les fautes.

L'algorithme 3 présente comment les fautes RTL sont classées. L'algorithme propose deux catégories "fautes silencieuses" et "tolérables", afin de pouvoir évaluer le nombre de fautes triées grâce à la prise en compte des intervalles critiques. En pratique, dans le but de l'identification des fautes critiques pour le système, ces deux catégories peuvent être regroupées en une seule catégorie "non-critique". Cela sera expliqué dans la suite. La sortie de la simulation du bloc en l'absence de faute représente la simulation de référence (ligne 1). On simule chaque faute f faisant partie de l'ensemble des n fautes à injecter (lignes 2 et 3) et on effectue la classification des fautes selon le résultat en sortie :

— Si la sortie est identique à celle de la simulation de référence, la faute est silencieuse (lignes 4 et 5).

— Si la sortie est différente de celle de la simulation de référence, mais les propriétés vérifiées, la faute est tolérable.

— Enfin, si la sortie est différente de celle de la simulation de référence, et la propriété non vérifiée, alors la faute est critique.

Algorithm 3: Classification des fautes RTL

```
1  RTL_Golden_output ← simulate(RTL_model, input, No_Fault)
2  for f ∈ RTL_Fault_injection_set do
3      RTL_output ← simulate(RTL_model, input, f)
4      if RTL_output = RTL_Golden_output then
5          silencieuse ← f
6      else
7          if RTL_assertion = verifié then
8              tolerable ← f
9          else
10             if RTL_assertion = non vérifiée then
11                 critique ← f
```

6.3.2 Calcul de la criticité globale

La classification des fautes permet de calculer une estimation du taux de SEUs critiques (FR_SEU), ici défini comme le nombre de fautes critiques par seconde. L'algorithme 4 présente la méthode pour calculer le FR_SEU. On note α le taux de SEU (*i.e.*, le nombre de SEUs par bit et par unité de temps exprimé en SEUs/s). Ce taux est un paramètre déjà utilisé dans l'AMDEC. Cet algorithme estime d'abord le nombre de bit-flips conduisant à un comportement critique pour le circuit entier (ligne 3). Ensuite, le FR_SEU est obtenu en multipliant la proportion de bit-flips critiques pour le circuit par α. Ce chiffre étant obtenu à partir du modèle de haut-niveau non validé, il peut être erroné.

Algorithm 4: Calcul du taux de SEUs critiques

1 $Nb\_fautes\_critiques = 0$
2 **for** $Blocs \in Systeme$ **do**
3 $\quad Nb\_fautes\_critiques = Nb\_fautes\_critiques + (Probabilité\_fautes\_critiques(Block) * Nb\_bits(Bloc))$
4 $\text{FR\_SEU} = \frac{Nb\_fautes\_critiques}{Nombre\ total\ de\ bits} * \alpha$

6.3.3 Illustration sur le cas d'étude : implémentation de la classification des fautes RTL.

Les résultats de chaque simulation sont stockés dans un fichier temporaire *transcript*. Ce fichier contient toutes les informations de la simulation. Comme les assertions, lorsqu'elles ne sont pas respectées, impriment un message "ERR_ASSERTION" dans le *transcript*, il suffit alors de lire ce fichier pour identifier si la faute est critique ou non. La lecture du *transcript* permet alors de classer la faute dans les catégories "critique", ou "non-critique". La figure 6.10 donne un extrait du script tcl permettant la classification de fautes. Le résultat de simulation est stocké dans le fichier *transcript*. Les mots clés "ERR_ASSERTION" correspondant à la violation d'une assertion, et "ERRONEOUS" correspondant à une sortie différente de la simulation de référence sont recherchés dans le *transcript* (lignes 1 et 2). Si la sortie est différente de celle de la simulation de référence (ligne 4), et qu'une assertion est violée (ligne 6), alors on sauvegarde la faute dans le fichier des erreurs critiques (ligne 7). Si aucune assertion n'est violée (ligne 9), on stocke la faute dans le fichier des erreurs tolérables (ligne 10). Enfin si la sortie de simulation est identique à celle de référence (ligne 12), on stocke la faute dans le fichier contenant les fautes silencieuses (ligne 13). Le traitement de la classification des fautes obtenues permet alors de calculer le taux de criticité par bloc.

```
1  set a [catch {grep -c ERR_ASSERTION transcript}]
2  set b [catch {grep -c ERRONEOUS transcript}]
3  # si la sortie est differente par rapport a la simulation de reference
4  if {$b==0} {
5  # si la sortie est critique
6      if {$a==0} {
7          puts $critiques "$faute"}
8  #sinon
9      else {
10         puts $tolerables "$faute" }}
11 #sinon
12 else {
13     puts $silencieuses "$faute"}
```

FIGURE 6.10 – *Extrait du script tcl permettant de classer les fautes RTL*

6.4 Résultats sur le cas d'étude

6.4.1 Présentation des résultats

L'étape de simulation de haut-niveau nous a permis de générer huit assertions. Chacune de ces assertions contraignent les signaux aux interfaces de communications entre les blocs. La seconde étape consiste à faire une injection de fautes RTL sur les blocs pour lesquels des intervalles critiques ont été identifiés dans la première étape. Les blocs simulés sont donc *DIVC*, *CounterX* et *CounterT*. Pour ces trois blocs, nous avons défini les bancs de test contenant le scénario et les assertions PSL, ainsi qu'une campagne d'injection de fautes dépendante des contraintes temporelles de simulation. Les résultats de simulations sont ici présentés en deux étapes :

— Tout d'abord, nous présenterons les résultats de la classification de fautes en trois catégories ; critiques, silencieuses, tolérables. Cette classification permet de mettre en évidence l'intérêt de l'utilisation des assertions.

— Nous présenterons ensuite les résultats de l'injection de fautes avec la classification proposée dans l'intérêt de la méthodologie, et l'extraction du taux de criticité par bloc, utile à l'AMDEC.

Le temps de simulation pour tous les blocs RTL prend 171 minutes, ce qui est conforme au temps maximum de 180 minutes que nous nous étions fixé.

6.4.2 Classification détaillée et intérêt des assertions

Le tableau 6.2 présente les résultats de l'injection de fautes RTL avec la classification détaillée des fautes. La 2ème ligne représente la proportion de fautes critiques. De plus, ce tableau dissocie parmi les fautes non-critiques deux catégories :

— Les fautes tolérables (ligne 3) : lorsque la sortie du bloc simulé est différente de la sortie de référence, mais que les assertions sont respectées.

— Les fautes silencieuses (ligne 4) : lorsque la sortie du bloc simulé est identique à la sortie de référence.

Sans la prise en compte des propriétés extraites de l'étape de simulation haut-niveau, les fautes classées dans la catégorie tolérables auraient été classées comme critiques. Les assertions permettent donc de filtrer 23% des comportements pour *DIVC*, 81,6% pour *CounterX*, et 15% pour *CounterT*. Le cas d'étude confirme donc ici l'intérêt de la méthodologie pour éviter la classification de fausses fautes critiques.

6.4.3 Classification standard et extraction du taux de criticité

Le tableau 6.3 présente la classification des fautes simplifiée, les fautes sont seulement différenciées entre celles qui sont critiques (2ème colonne), ou non (3ème colonne). La 4ème colonne calcule la criticité pour chaque bloc. C'est-à-dire le nombre de fautes potentielles par heure dans chaque bloc. Ce nombre est obtenu en supposant que chaque flip-flop du système a la même probabilité d'être perturbé par une particule. Ce nombre est exprimé en fautes critiques par heure. Cette information permet de voir quel bloc est le plus vulnérable. Comme *DIVA*, *DIVB* et *SYNC* n'ont pas eu d'intervalle critique dans l'étape de simulation haut-niveau, toutes les fautes sont classées comme non critiques et le nombre d'erreurs critiques est nul. On peut observer que les blocs *CounterX* et *CounterT*

TABLEAU 6.2 – *Résultats de la simulation de fautes RTL et classification des fautes avec les propriétés extraites à haut-niveau. (c,e) = (95%, 5%)*

	DIVC	**CounterX**	**CounterT**
Nombre de fautes critiques	37 (9.8%)	182 (47.5%)	157 (40.9%)
Nombre de fautes tolérables	79 (21.1%)	165 (42.9%)	35 (9.1%)
Nombre de fautes silencieuses	260 (69.1%)	37 (9.6%)	192 (50%)

ont un taux de fautes critiques plus élevé (resp. 47.5% et 40.9%). De plus, leur criticité est dix fois plus élevée que pour le bloc *DIVC*. Cela nous permet de déduire que les blocs *CounterX* et *CounterT* semblent être plus vulnérables, selon les résultats obtenus de cette première itération. En additionnant la criticité par bloc, on peut obtenir le taux de criticité de la partie FPGA, qui est alors de $2.87 * 10^{-08}$ fautes critiques par heure. Ce nombre est utile car c'est sur ces valeurs que les normes fixent des seuils à ne pas dépasser.

TABLEAU 6.3 – *Résultats de la simulation d'injection de fautes RTL.*

Bloc	Nombre de critiques	Nombre de non critiques	Taux criticité par bloc (f/h)
DIVA	0	333 (100%)	0
DIVB	0	305 (100%)	0
DIVC	37 (9,8%)	334 (90,2%)	$1,52 * 10^{-09}$
SYNC	0	333 (100%)	0
CounterX	182 (47,5%)	202 (52,5%)	$1,46 * 10^{-08}$
CounterT	157 (40,9%)	227 (59,1%)	$1,26 * 10^{-08}$
Total	N.A.	N.A.	$2.87 * 10^{-08}$

6.4.4 Conclusion

Les résultats de l'étape de simulation RTL appliquée au cas d'étude nous permettent donc :

— D'obtenir une classification des fautes.

— D'obtenir une première estimation des blocs vulnérables du circuit.

— De calculer la criticité totale, et détaillée bloc par bloc.

Ces premiers résultats, dans le cadre d'un flot de conception top-down peuvent permettre d'aider les concepteurs à plusieurs niveaux :

— Au niveau système, si des blocs sont vulnérables, des décisions peuvent être prises au niveau système pour prendre des contremesures (triplication par exemple).

— Au niveau conception RTL, connaître la criticité bloc par bloc rapidement permet d'aider à améliorer la robustesse de chacun des blocs en ciblant les plus vulnérables. La prise en compte des propriétés permet de filtrer les faux critiques et donc éviter du sur-design pour des blocs qui ne sont peut-être pas forcément critiques.

— Pour les analyses de sûreté : on obtient un premier calcul de criticité utile à l'étude de l'effet des SEUs présente dans l'AMDEC.

Néanmoins, les résultats obtenus dans cette partie sont basés sur l'extraction de propriétés issue de l'étape de simulation RTL, elle-même basée sur un modèle de haut-niveau dont la qualité n'est pas connue. Il est donc indispensable de pouvoir mesurer la qualité du modèle afin d'évaluer la qualité des résultats de cette étape de simulation RTL. Nous analyserons la qualité du modèle de haut-niveau dans le chapitre suivant.

7

Etape 3 : Co-simulation de fautes

Sommaire

L'étape de simulation RTL nous a permis d'obtenir une classification des fautes RTL bloc par bloc. Néanmoins, cette classification est basée sur les propriétés extraites de la simulation de haut-niveau. Il est donc nécessaire de valider la qualité du modèle de haut-niveau utilisé, afin de pouvoir connaître la qualité de la classification obtenue. Les mêmes fautes RTL que celles utilisées dans l'étape de simulation RTL sont réutilisées (quand l'injection de fautes a été faite). Cette simulation est considérée comme la simulation de référence car elle combine les détails de chaque bloc RTL (un à un) et la spécification du système. Cette vérification prend beaucoup de temps mais permet de comparer les résultats précédemment obtenus (simulation de haut-niveau et simulation RTL) avec ceux d'une simulation plus détaillée (co-simulation). Les différences entre simulation de haut-niveau et co-simulation peuvent être : fautes RTL se propageant en fautes multiples sur les sorties des blocs, non monotonie de l'erreur en fonction de l'amplitude de la faute injectée, *etc.* L'intérêt d'utiliser un système multi-abstraction pour cette étape est donc de réduire le temps de simulation par rapport à la simulation du système entièrement modélisé au niveau RTL et de prendre la spécification du système en compte.

Cette étape décrite dans la figure 4.7 du chapitre 4.3 prend en entrée les modèles RTL de chacun des blocs, le modèle de haut-niveau développé dans l'étape de simulation haut-niveau, et la classification des fautes RTL obtenues de l'étape de simulation RTL. L'implémentaion de cette étape comprend trois parties :

— Modélisation de l'environnement : Dans cette étape de co-simulation, le modèle de haut-niveau du système ainsi que la description de chaque bloc au niveau RTL est déjà disponible. Il est néanmoins nécessaire d'implémenter pour chaque bloc le modèle à co-simuler en remplaçant dans le modèle de haut-niveau initial, un à un chaque bloc par sa description RTL. Cela nécessite souvent des efforts de modélisation pour adapter les blocs RTL à l'environnement haut-niveau développé.

— Injection des fautes : il est nécessaire de définir la campagne d'injection de fautes à effectuer lors de cette seconde étape. Cela comprend la localisation des fautes, leur type, et l'instant auquel elles sont injectées.

— Extraction des résultats : Après co-simulation du modèle avec injection de fautes, il est nécessaire de classer les fautes, et de calculer les métriques indiquant la qualité du modèle de haut-niveau développé dans l'étape de simulation haut-niveau.

Une dernière partie présentera l'application de l'étape au cas d'étude, et les rondes successives appliquées sur ce cas d'étude.

7.1 Modélisation de l'environnement de co-simulation

7.1.1 Modélisation du système multi-abstraction

Le but de cet étape est de co-simuler le système, on utilisera alors la modélisation de haut-niveau du système, dans laquelle on remplacera un par un chaque bloc par sa description RTL. Cette technique de modélisation s'appelle la co-simulation.

Elle nécessite d'utiliser des connecteurs (wrappers), qui permettent d'adapter le bloc RTL au modèle TLM. La figure 7.1 illustre comment un bloc RTL est intégré pour la co-simulation. A partir du modèle de haut-niveau développé dans l'étape de simulation

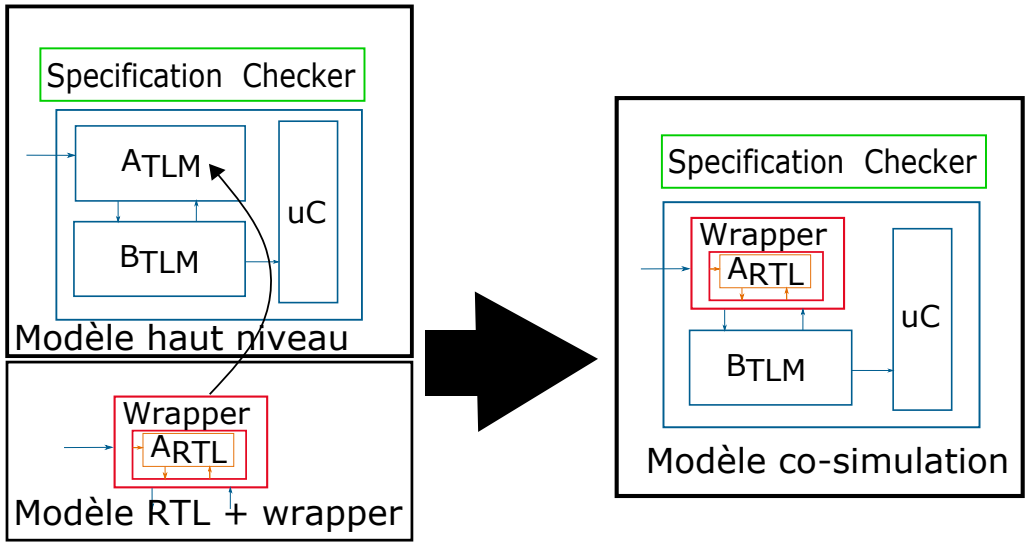

FIGURE 7.1 – *Intégration de la description RTL d'un bloc pour la co-simulation.*

de haut-niveau, on veut remplacer un des blocs par sa description RTL. Sur la figure 7.1 le bloc A_{TLM} est remplacé par sa description RTL A_{RTL}. On intègre la description RTL grâce à un connecteur $Wrapper$ qui permet de rendre compatible les signaux utilisés au niveau RTL et ceux utilisés au niveau TLM.

7.1.2 Scénarios simulés

La simulation étant à l'échelle du système, on va simuler les mêmes scénarios que lors de l'étape de simulation haut-niveau.

7.1.3 Illustration sur le cas d'étude : co-simulation de CounterX et définition du scénario.

Modélisation de l'environnement de simulation Illustrons la modélisation de l'environnement de simulation pour le compteur $CounterX$. La figure 7.2 illustre le rôle et l'importance de l'adptateur $CounterX\_Wrapper$ qui permet de connecter le bloc $CounterX_{RTL}$ à l'environnement système. Le bloc $CounterX_{RTL}$ prend :

— en entrée trois signaux délivrant des impulsions à différentes fréquences,

— en sortie des valeurs entières stockées dans des registres sur 16 bits ou 24 bits, et un signal délivrant des interruptions sous forme d'impulsions.

Le bloc $CounterX_{TLM}$ utilisé dans le modèle de haut-niveau prend :

— en entrée trois signaux flottants représentants les valeurs de fréquence des impulsions,

— en sortie deux entiers représentants les valeurs des registres.

La figure 7.3 présente le code SystemC du connecteur $CounterX\_WRAPPER$. Ce code implémente les différents processus $P1$, $P2$, $P3$, $P4$ illustrés dans la figure 7.2. Le processus $P1$ permet de convertir un signal F_{ref} de type flottant en un signal oscillant sig_{ref} de fréquence F_{ref}. Pour cela, il attend à chaque fois un temps T correspondant à la période de F_{ref} et génère une impulsion (ligne 29) à chaque fois que la période T est

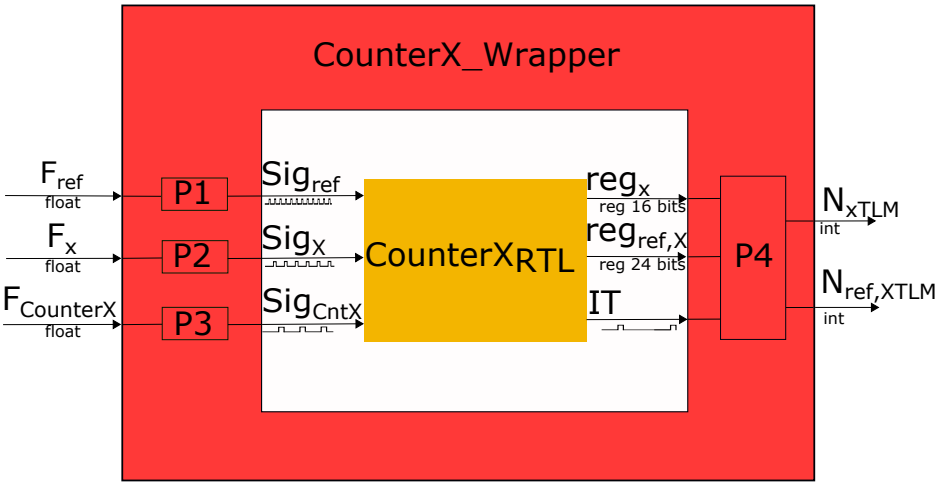

FIGURE 7.2 – *Illustration du wrapper de CounterX.*

écoulée. La valeur T est calculée (ligne 28) lorsqu'une première valeur F_{ref} arrive (ligne 25) puis à chaque fois que le temps T est écoulé ou que la valeur F_{ref} change (ligne 27). Les processus $P2$ et $P3$ fonctionnent sur le même principe.

Le processus $P4$ converti les sorties du bloc RTL $CounterX_{RTL}$ vers le système. On veut convertir les valeurs stockées dans les registres Reg_x et $Reg_{ref,x}$ en entier N_{XTLM} et $N_{ref,XTLM}$. Pour cela, le processus attend une interruption sur le signal IT (ligne 51) qui notifie que des nouvelles valeurs sont écrites dans les registres, puis converti les valeurs des registres en entier (lignes 52 et 53).

7.2 Ré-injection des fautes RTL.

7.2.1 Recherche d'équivalence avec les fautes injectées lors de la simulation RTL.

Les fautes injectées sont les mêmes que celles injectées dans l'étape de simulation RTL. Le but est de vérifier la cohérence entre la classification des fautes de l'étape de simulation RTL et celle de l'étape de co-simulation. Néanmoins, lors de l'étape de simulation RTL les scénarios simulées étaient beaucoup plus courts, ce qui permettait de réduire les temps de simulation. Ré-injecter les mêmes fautes dans l'étape de co-simulation que dans l'étape de simulation RTL nécessite de trouver pour chaque bloc l'instant équivalent à l'instant d'injection T0 de la simulation RTL.

7.2.2 Illustration sur le cas d'étude : définition des fautes à injecter sur DIVC

On veut injecter les mêmes fautes que celles injectées lors de l'étape de simulation RTL appliquée au cas d'étude lors de cette étape de co-simulation. Néanmoins, nous avons vu que, au niveau RTL, les scénarios simulés ont une durée inférieure et sont adaptés à chaque bloc.

```
class CounterX_Wrapper : public sc_core::sc_module {
public:
sc_out<sc_logic> Sig_ref;
sc_out<sc_logic> Sig_X;
sc_out<sc_logic> Sig_CntX;
sc_in<sc_bv<16> > Reg_X;
sc_in<sc_bv<24> > Reg_refX;
sc_in<sc_logic> IT;
sc_in<float> F_ref;
sc_in<float> F_X;
sc_in<float> F_CntX;
sc_out<unsigned int> N_X;
sc_out<unsigned int > N_refX;

SC_HAS_PROCESS( CounterX_Wrapper );
CounterX_Wrapper(sc_module_name nm){
SC_THREAD( P1 );
SC_THREAD( P2 );
SC_THREAD( P3 );
SC_THREAD( P4 ); }
~CounterX_Wrapper(){}
void P1(){
int temp ;
sc_time T( 10000, SC_MS);
wait(F_ref.value_changed_event());
while (1) {
    wait((F_ref.value_changed_event()) | (T));
    sc_time T((1/(F_ref.read()) * 1000000000), SC_MS) ;
    gen_impulsion(Sig_ref);} // genere une impulsion
}
void P2(){
...
}
void P3(){
...
}
void P4(){
    while (1) {
        wait(IT.posedge_event());
        N_X.write( (Reg_X.read()) .to_uint());
        N_refX.write( (Reg_refX.read()) .to_uint());}
} };
```

FIGURE 7.3 – *Code SystemC du wrapper pour CounterX*

La figure 7.4 illustre pour le bloc *DIVC* la longueur d'un processus de *DIVC* par rapport au scénario de simulation du système complet. Un scénario de simulation pour le bloc *DIVC* au niveau RTL dure $D_{RTL\_FS\_DIVC}$. Un scénario au niveau système dure $D_{CO-SIMU}$. On a :

$$D_{CO-SIMU} > D_{RTL\_FS\_DIVC} \tag{7.1}$$

Au niveau système, le système met un temps D_{init} à s'initialiser. Soit une faute injectée au niveau RTL définie par le quadruplé (l,t,i,d). On veut définir le nouveau quadruplé $(l_{CO-SIMU}, t_{CO-SIMU}, i_{CO-SIMU}, d_{CO-SIMU})$ correspondant à la faute équivalente en co-simulation. La localisation $l_{CO-SIMU}$, le type $t_{CO-SIMU}$ et la durée $d_{CO-SIMU}$ sont identiques à l'étape de simulation RTL. L'instant d'injection $i_{CO-SIMU}$ est alors calculé comme suit :

$$i_{CO-SIMU} = i + D_{init} \tag{7.2}$$

l'instant D_{init} correspondant à l'instant où l'état du bloc *DIVC* est dans son état initial.

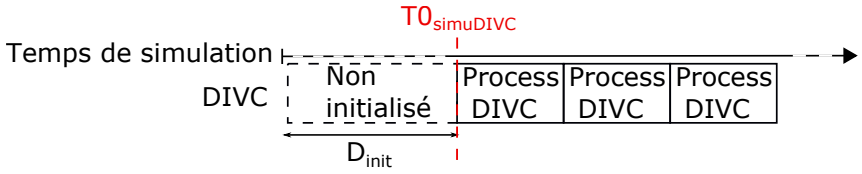

FIGURE 7.4 – *Illustration des différences de scénarios entre l'étape HLFS et RTLFS.*

7.3 Extraction des résultats

7.3.1 Extraction de métriques à partir d'une matrice de confusion.

Le but de cette étape est d'obtenir des métriques permettant d'évaluer le modèle de haut-niveau et son modèle de fautes utilisés dans l'étape de simulation de haut-niveau. Si la spécification du système est vérifiée même après l'injection d'une faute RTL, cela signifie que la faute est silencieuse ou tolérable et que le système global est robuste à cette faute. Néanmoins, comme co-simuler un bloc RTL dans son environnement système nécessite plus de temps que faire une simulation de fautes sur un modèle de haut-niveau, la co-simulation de fautes est utilisée uniquement dans le but de valider les précédents résultats de simulations de fautes basés sur des simulations moins chronophages.

Le tableau 7.1 est une matrice de confusion [74]. Elle permet de définir des métriques pour évaluer les résultats obtenus dans les étapes précédentes. La colonne RTL_FS correspond aux résultats de l'étape de simulation de fautes RTL. La ligne $Co-simulation$ correspond aux résultats de l'étape de co-simulation et est considérée comme le comportement de référence. Le but est donc de classer les fautes dans ces quatre catégories. Les vraies fautes critiques correspondent aux fautes qui sont critiques dans l'étape de simulation RTL, et dans l'étape de co-simulation. Les fausses fautes critiques correspondent aux fautes critiques dans l'étape de simulation RTL, mais pas dans l'étape de co-simulation. Les fausses fautes silencieuces correspondent aux fautes silencieuses dans l'étape de simulation RTL mais qui s'avèrent être critique dans l'étape de co-simulation. Enfin, les

vraies fautes silencieuses correspondent aux fautes silencieuses dans les étapes de simulation RTL et de co-simulation. À partir de ces résultats, plusieurs métriques permettent d'évaluer la qualité du modèle.

La justesse est la proportion de résultats justes par rapport au nombre de fautes injectées. Elle est calculée comme suit :

$$Justesse = \frac{VC + VS}{VC + VS + FC + FS}$$

La précision est définie par la proportion de vrais comportements critiques identifiés dans les deux premières étapes :

$$Precision = \frac{VC}{VC + FC}$$

Enfin, le taux de vrais silencieuses (True Silent Rate en anglais) correspond à la proportion de fautes silencieuses correctement identifiées dans l'étape 2 :

$$TSR = \frac{VS}{VS + FS}$$

TABLEAU 7.1 – *Matrice de confusion*

Co-simulation / Etape RTL_FS	Critiques	Silencieuses
Critiques	Vraies critiques (VC)	Fausses critiques (FC)
Silencieuses	Fausses silencieuses (FS)	Vraies silencieuses (VS)

7.3.2 Illustration sur le cas d'étude : matrice de confusion et extraction de métriques sur CounterX.

Illustrons l'extraction des métriques pour le bloc *CounterX* de notre système. La classification des fautes après simulation nous permet d'obtenir les résultats reportés dans le tableau 7.2.

TABLEAU 7.2 – *Matrice de confusion pour CounterX*

Co-simulation / Etape RTL_FS	Critiques	Silencieuses
Critiques	194	8
Silencieuses	101	81

Un script permet alors d'obtenir les métriques suivantes :

$$Justesse = 71,6\%$$

$$Precision = 96\%$$

$$TSR = 44,5\%$$

Nous illustrons ici simplement comment obtenir les métriques à partir de la matrice de confusion obtenue pour le bloc *CounterX*. Les résultats sur le système complet discutés sont présentés dans la suite.

7.4 Résultats sur le cas d'étude

7.4.1 Présentation des résultats de la première ronde

Nous avons injecté les mêmes fautes que lors de l'étape de simulation RTL. Cette étape de co-simulation a duré 9h, soit 3 fois plus que pour la simulation RTL.

Le tableau 7.3 présente les métriques obtenues à l'issue de l'étape de co-simulation de la première ronde. La première colonne donne le nom de chaque bloc. La seconde colonne présente les temps de simulation correspondant pour chaque bloc. Les trois dernières colonnes présentent les différentes métriques calculées. Ces métriques donnent une indication sur la qualité du modèle de haut-niveau. Le seuil exigé sur ces métriques dépend des contraintes imposées par l'ingénieur sécurité. Dans notre cas, le seuil est fixé à 90% minimum de justesse. Le TSR doit être supérieur à 90% car nous voulons éviter la détection de fausses fautes silencieuses. On ne fixe pas de contrainte sur la précision.

La justesse pour les blocs *DIVA*, *DIVB*, et *SYNC* n'est pas applicable car ces blocs ne sont pas critiques. Le bloc *DIVC* présente une bonne justesse, un TSR à 100%, mais une précision très mauvaise (beaucoup de fautes sont identifiées comme critiques alors qu'elles ne le sont pas réellement). Cela traduit une qualité insuffisante de notre modèle avec ce bloc (on ne filtre finalement pas bien les vrais comportements critiques). Néanmoins, comme la justesse est bien respectée, cela signifie que la précision à une part faible pour ce bloc là (les fautes abusivement considérées comme critiques représentent une proportion faible). De plus, cela correspond à une hypothèse pessimiste, il n'est donc pas prioritaire de modifier le modèle pour ce bloc-là.

La justesse du bloc *CounterX* est en dessous du seuil. Le TSR est très mauvais : près de la moitié des fautes classées silencieuses dans l'étape de simulation RTL ne sont pas correctement identifiées. Il faut redéfinir le modèle de haut-niveau pour être plus proche de la réalité. Les métriques du bloc *CounterT* sont également de mauvaises qualités. Le bloc *CounterT* étant le même module RTL que le bloc *CounterX* avec des entrées différentes, ce comportement est cohérent avec ce qui était attendu.

7.4.2 Interprétation des résultats de la première ronde

Dans cette section, nous allons nous intéresser à comprendre l'origine de la faible qualité du modèle et proposer un modèle amélioré. La méthode permet d'identifier rapidement les blocs problématiques et d'identifier l'origine des problèmes (à partir des loca-

TABLEAU 7.3 – *Resultats de la première ronde "Quick and dirty"*

Block	Co-simulation time (min)	Justesse	Precision	True Silent Rate
DIVA	61	100%	N.A.	100%
DIVB	45,75	100%	N.A.	100%
DIVC	9,4	92%	2,8%	100%
SYNC	6,1	100%	N.A.	100%
CounterX	96	71,6%	96,5%	55%
CounterT	70,4	78,1%	92,4%	69,8%

lisations des fautes RTL mal classifiées). Toutefois cela nécessite alors de rentrer dans les détails de fonctionnement du bloc RTL et du système.

Le bloc *CounterX* (voir figure 7.5) compte le nombre de cycle des signaux $F_{ref,X}$ et F_X pendant une fenêtre de temps définie entre deux fronts montants du signal $F_{windowX}$.

Pour simplifier l'explication, on se focalisera sur le signal F_X. Le signal IT indique qu'une nouvelle valeur est disponible dans le registre N_X. Il correspond à un signal d'interruption du micro-contrôleur. Le comportement de *CounterX* est illustré dans la figure 7.6. Entre deux fronts montants de $F_{windowX}$, on observe trois fronts montants de F_X. Lors du front montant indiquant la fin d'une fenêtre de comptage et le début de la suivante, le signal N_X prend la valeur 3, et le signal IT est activé (niveau logique 1).

FIGURE 7.5 – *Modèle RTL de CounterX*

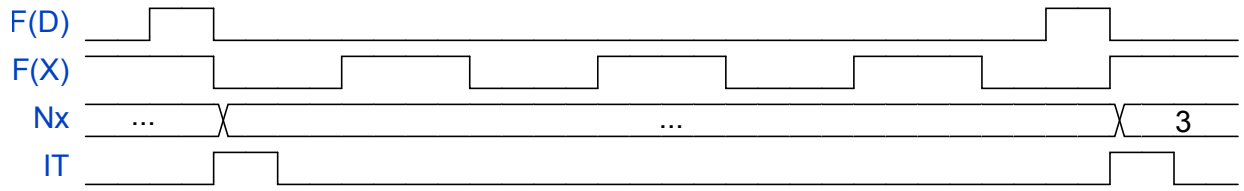

FIGURE 7.6 – *Chronogramme illustrant le fonctionnement de CounterX*

Dans notre modèle de haut-niveau (voir Fig. 7.7), les signaux F_X et $F_{windowX}$ sont des entiers et non des signaux oscillants. La sortie N_X est le résultat de la division de F_X par $F_{windowX}$.

Le signal N_X est envoyé grâce à un canal de communication de type SC_FIFO. Ce type de canal permet au bloc micro-contrôleur d'être sensible à chaque nouvelle donnée

envoyée sur le canal. De ce fait, le signal *IT* n'avait pas été modélisé sur le modèle de haut-niveau, et le fonctionnement en l'absence de faute est respecté.

FIGURE 7.7 – *Modélisation TLM de CounterX*

Pour améliorer la qualité du modèle de *CounterX*, nous avons analysé plus précisément les résultats d'injection de l'étape de co-simulation, en classant par flip-flop les fautes responsables du TSR bas. La figure 7.8 montre la distribution des fausses fautes silencieuses par flip-flop. On remarque que la mauvaise qualité de la métrique est dûe à un nombre limité de flip-flops. Une analyse plus détaillée nous permet d'observer que ces flip-flops sont toutes utiles à la génération du signal *IT*.

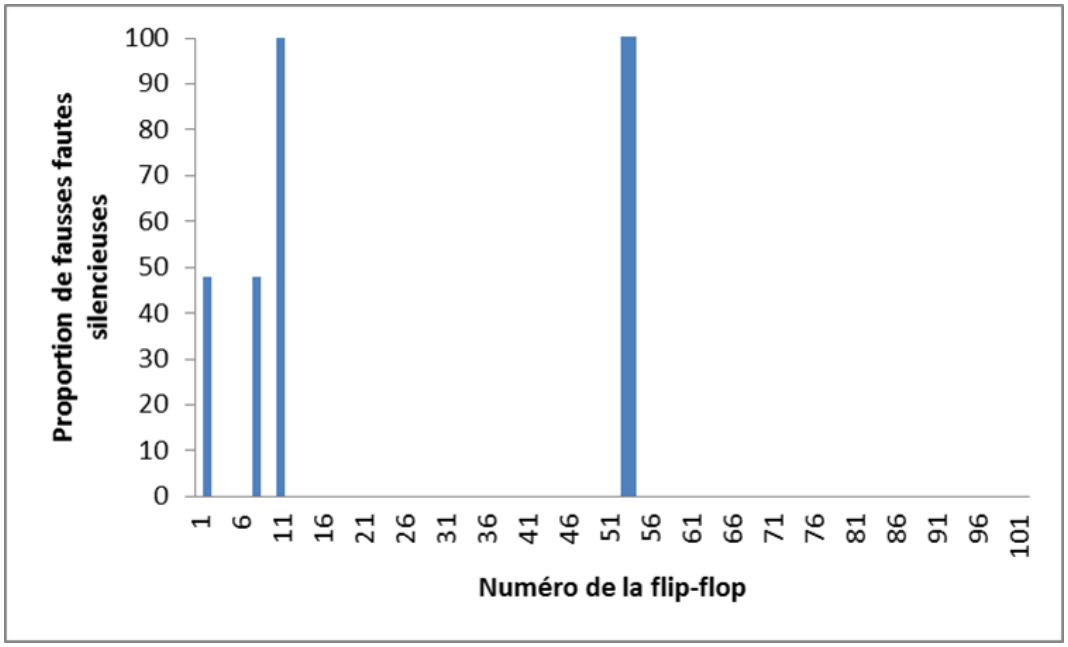

FIGURE 7.8 – *Distribution des fausses fautes silencieuses par flip-flop dans CounterX.*

Pour comprendre ce défaut de modélisation, il suffit alors d'observer ce qu'il se passe lorsqu'une faute se propage sur le signal *IT*. Au niveau système, les fautes qu'on injecte permettent de modifier la valeur N_X, mais ne modifient pas le moment où cette valeur N_X est mise à disposition sur le canal de communication. La figure 7.9 illustre ce comportement : les processus de *CounterX* et du bloc *CALC* (correspondant au micro-contrôleur) sont intrinsèquement synchronisés grâce aux fonctionnalités apportées par le canal de communication de type SC_FIFO.

Au niveau RTL, une faute injectée peut modifier le signal *IT* (alors qu'une nouvelle valeur n'est pas encore disponible sur N_X), et une donnée non actualisée est lue par le microprocesseur. Ce comportement est illustrée dans la figure 7.10 : à l'instant T0, le signal *IT* est correcte et le micro-contrôleur lit une valeur N_X1, mais à l'instant T1, le signal *IT* est corrompu. Le bloc *CounterX* n'a pas fini son calcul, et le micro-contrôleur

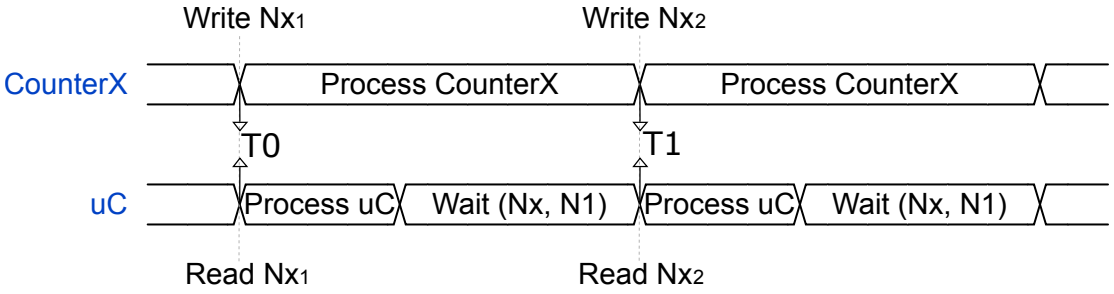

FIGURE 7.9 – *Synchronisation de CounterX avec le micro-contrôleur*

lit à nouveau la valeur N_X1. Ce comportement n'était pas modélisé sur le modèle de haut-niveau car les fonctionnalités du canal de communication de type *SC_FIFO* font que l'écriture d'une nouvelle valeur sur N_X et la notification d'écriture au micro-contrôleur étaient synchronisés.

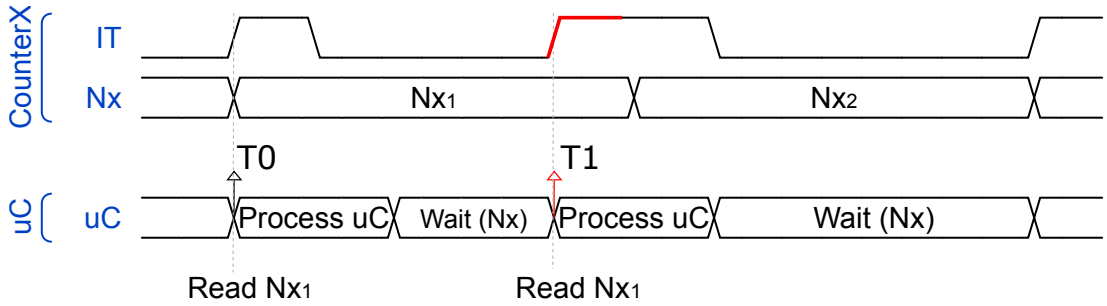

FIGURE 7.10 – *Modèle RTL du bloc CounterX avec propagation de faute sur le signal IT*

Pour résoudre ce problème, le modèle de haut-niveau est modifié en ajoutant un signal d'interruption. Une nouvelle ronde du flot est alors nécessaire pour évaluer ce nouveau modèle.

7.4.3 Réécriture et deuxième ronde

Dans cette partie, nous appliquons une seconde ronde du flot, à partir du nouveau modèle de haut-niveau développé. Cette ronde est de nouveau basée sur le principe du "Quick and dirty". À la fin de cette ronde, les métriques obtenues sont celles reportées dans le tableau 7.4. On observe que dans l'étape de simulation RTL (colonne 2), plus de fautes sont critiques, à la fois dans le bloc *CounterX* et dans *CounterT*. Cela signifie que ces deux blocs sont plus critiques qu'initialement supposé dans la première ronde du flot. Le TSR et la justesse sont améliorés. Les seuils fixés sur la justesse et le TSR sont atteints. Cette deuxième ronde a donc permis de valider le modèle de haut-niveau amélioré. Il reste à effectuer une simulation RTL précise afin d'obtenir une classification de fautes RTL la plus précise possible.

TABLEAU 7.4 – *Métriques obtenues après une seconde ronde "Quick and dirty"*

Bloc	Critical (RTL_FS)	Justesse	Precision	True Silent Rate
CounterX	73,4%	94,4%	93,7%	95,4%
CounterT	70,2%	95,4%	95,3%	95,3%

7.4.4 Calcul du taux de SEU critiques et comparaison avec l'analyse AMDEC

Les deux premières rondes du flot nous ont permis d'obtenir un modèle de haut-niveau de bonne qualité, selon les seuils que nous avons fixé sur les métriques. La dernière ronde consiste à revenir à l'étape de simulation RTL, et d'élargir la contrainte de temps pour obtenir une injection statistique de fautes plus précise (avec un taux de confiance augmenté et une marge d'erreur réduite). Les nouveaux paramètres fixés pour l'injection de fautes statistiques sont un taux de confiance de 99,8% et une marge d'erreur réduite à 2%. Cela correspond à un temps de simulation pour l'étape de simulation RTL de 41 heures, pour 21185 fautes injectées. Le tableau 7.5 présente la classification des fautes de cette dernière ronde. Il présente également la contribution par bloc sur le MTBF. Le MTBF total est alors calculé en dernière ligne.

TABLEAU 7.5 – *Résultats de l'étape de simulation RTL lors de la dernière ronde ($\alpha = 7,2 * 10^{-11}$ neutrons/bit/h).*

Bloc	Critiques	Non-critiques	Taux criticité par bloc (f/h)
DIVA	0%	100%	0
DIVB	0%	100%	0
DIVC	10,2%	89,8%	$1,59 * 10^{-9}$
SYNC	0	100%	0
CounterX	73,4%	26,6%	$2,28 * 10^{-8}$
CounterT	70,2%	29,8%	$2,18 * 10^{-8}$
Total	N.A.	N.A.	$4,62 * 10^{-8}$

Le taux de criticité pour la partie FPGA dans notre cas est de $4,62 * 10^{-8} f/h$. C'est inférieur à l'analyse traditionnelle. De plus, seul les blocs *CounterX* et *CounterT* sont critiques, ils pourraient donc éventuellement être modifiés au niveau RTL pour être plus robustes.

7.4.5 Conclusion du chapitre

Cette dernière étape de co-simulation appliquée au cas d'étude nous a permis de quantifier la qualité du modèle de haut-niveau initialement développé. La première version du modèle présentait un problème de réalisme. L'automatisation et la classification des fautes par bloc et par flip-flop nous a permis de résoudre rapidement ce problème de modélisation et de développer une seconde version du modèle de haut-niveau. L'intérêt de cette dernière étape de co-simulation est donc :

— D'évaluer quantitativement la qualité du modèle de haut-niveau développé dans l'étape de simulation haut-niveau.

— D'identifier les défauts de modélisation grâce à l'extraction des métriques bloc par bloc, puis à la classification des fautes flip-flop par flip-flop.

La deuxième ronde du flot sur la deuxième version du modèle haut-niveau nous a permis de le valider.

Une dernière ronde comprenant seulement l'étape de simulation RTL nous a permis d'obtenir une classification des fautes plus précise et d'en déduire un taux de criticité significativement plus bas que dans l'analyse traditionnelle. De plus, ce taux de criticité est obtenu en se basant sur :

— Un modèle de haut-niveau dont la qualité est quantifiée.

— Une injection statistique qui nous permet d'avoir un taux de confiance et une marge d'erreur connus.

Conclusion, perspectives et publications

Conclusion

Nous avons présenté dans ce manuscrit une méthodologie permettant d'évaluer la sûreté d'un système complexe, tôt dans le flot de conception. La thèse a été effectuée dans un contexte à la fois académique et industriel. La méthodologie proposée répond alors à plusieurs enjeux présents dans les deux contextes.

Le contexte industriel a soulevé la problématique principale de la thèse : les analyses actuelles dans le contexte des systèmes critiques sont faites empiriquement, avec peu de solutions automatisées, et se basent sur l'expérience des ingénieurs. Ces méthodes traditionnelles d'analyse de sûreté présentent les inconvénients d'être chronophages, peu précises, et reposent sur des hypothèses pessimistes afin d'éviter des erreurs.

Dans la littérature scientifique, des solutions d'analyse de sûreté existent, mais aucune ne répond correctement aux problématiques spécifiques du contexte industriel :

— Les solutions de modélisation de bas niveau sont réalistes, mais elles sont chronophages et interviennent tard dans le flot de conception.

— Les solutions de modélisation de haut-niveau permettent de simuler des systèmes complexes rapidement et tôt dans le flot de conception mais sont peu précises, et le réalisme des fautes de haut-niveau utilisées n'est pas vérifié.

Afin de tirer le meilleur compromis entre les différents niveaux de modélisation, nous avons proposé un flot basé sur une simulation de fautes multi-abstraction en trois étapes. Une approche itérative exploitant ce flot a été présentée. Cette méthodologie a été appliquée à un cas d'étude réel utilisé dans l'aéronautique. Ce cas d'étude mesure la fréquence d'un capteur indispensable aux données de vols.

À travers ce manuscrit, nous avons pu mettre en évidence les différents avantages apportés par la méthodologie.

Dans le chapitre 4, nous avons présenté une vue générale de la méthodologie, en présentant un flot d'analyse de sûreté et une approche itérative pour l'exploiter. Nous avons pu montrer que le flot s'inscrit dans un flot de conception top-down traditionnel, ce qui a permis d'éviter des efforts de modélisation supplémentaires. Cela a également permis de montrer que des propriétés permettent entre chaque niveau de simulation de continuer de prendre en compte la spécification et l'environnement du système.

Dans le chapitre 5, nous avons présenté l'étape de simulation de fautes à haut-niveau : Nous avons montré comment créer des modèles de haut-niveau et des modèles de fautes de haut-niveau. Nous avons pu extraire des propriétés de criticité sur les sorties de chacun des blocs, définies sous forme d'intervalles d'amplitude de fautes et de temps. Ces intervalles de criticité extraits nous ont permis :

— D'obtenir une première indication qualitative de la sûreté du système : certains blocs ont des intervalles de criticité, tandis que pour d'autres aucun intervalle de criticité n'a été trouvé.

— De définir des propriétés blocs par bloc. C'est un moyen de communication et de spécification facilement compréhensible par chacun des concepteurs des blocs.

Néanmoins, ces intervalles de criticité ont leurs limites :

— Ils ne donnent pas d'indication quantitative sur la criticité de chacun des blocs, un bloc peut sembler avoir un intervalle de criticité réduit (d'amplitude faible), mais on ne sait pas combien de fautes RTL conduiront à des erreurs dans ces intervalles.

— Ces intervalles de criticité sont basés sur un modèle haut-niveau du système et un modèle de fautes, choisis selon des hypothèses simplificatrice : monotonie de la fonction d'erreur en sortie en fonction de l'amplitude de faute injectée, intervalles non exactes, *etc.*

Il est donc nécessaire à ce stade d'effectuer une étude de la propagation des fautes RTL pour identifier combien mènent à ces intervalles critiques, et de valider le modèle de haut-niveau.

Dans le chapitre 6, nous avons présenté l'étape de simulation RTL, où nous avons réutilisé les propriétés extraites de l'étape de simulation haut-niveau, pour faire une simulation de fautes statistiques sur les blocs matériels du système. L'utilisation de propriétés bloc par bloc a permis de découper la simulation du système par bloc et donc de réduire les temps de simulations par rapport à une simulation du système complet au niveau RTL. Nous avons appliqué cette simulation de fautes RTL sur le cas d'étude. Les paramètres choisis pour l'injection de fautes statistiques ont été définis tels que les temps de simulations soient réduits à quelques heures (3 heures). Cette première simulation nous a permis :

— D'obtenir une première classification des fautes RTL, et donc une première estimation quantitative de la vulnérabilité des blocs.

— D'évaluer la pertinence des mécanismes de détections utilisés.

Néanmoins, cette première simulation RTL est basée sur le modèle de haut-niveau présenté dans le chapitre 5 dont la qualité n'a pas été quantifiée.

Dans le chapitre 7, nous avons présenté l'étape de co-simulation. Elle simule un à un chacun des blocs matériels dans l'environnement système défini dans l'étape de simulation haut-niveau. Cette étape nous a permis d'extraire des métriques qui quantifient la qualité du modèle de haut-niveau utilisé.

Nous avons appliqué cette étape de co-simulation à notre cas d'étude. La première ronde appliquée au cas d'étude a conduit à des métriques qui n'atteignent pas le seuil de qualité que nous avions fixé. Une analyse détaillée nous a permis :

— D'identifier les faiblesses du modèle de haut-niveau grâce au calcul des métriques bloc par bloc.

— De localiser les flip-flops sur lesquelles la classification de fautes est mauvaise.

Un deuxième modèle de haut-niveau a été développé à partir de l'analyse du premier modèle. Nous avons appliqué et présenté une deuxième ronde (à partir du début du flot) à ce deuxième modèle de haut-niveau avec lequel nous avons obtenu des métriques suffisantes pour valider ce deuxième modèle de haut-niveau. Enfin, une troisième ronde plus précise a été appliquée. Cette dernière ronde consistait uniquement à faire une étape de simulation RTL plus précise. Nous avons alors choisi des paramètres pour l'injection statistique plus précis, et donc plus de fautes ont été injectées. Elle nous a permis d'obtenir une nouvelle classification des fautes RTL. Cette dernière classification diffère de la

classification obtenue lors de la 1ère ronde pour les blocs qui avaient des métriques de mauvaise qualité, mais change très peu pour les autres blocs. La classification obtenue lors de la 3ème ronde montre des résultats quasi-identiques à celle de la 2ème ronde, ce qui montre que l'injection de fautes statistiques permet une première évaluation rapide tout en gardant un bon taux de confiance. Les temps de simulation pour cette dernière simulation RTL ont été 20 fois supérieurs à ceux des rondes précédentes.

La classification finale des fautes obtenues est une information utile à l'analyse AMDEC effectuée lors des analyses de sûreté traditionnelle. Cette classification obtenue est plus précise que l'analyse initialement effectuée empiriquement. Elle conduit à un taux de SEUs critiques plus faible que dans l'analyse AMDEC traditionnelle.

L'analyse qualitative de l'étape de simulation haut-niveau est une information très utile pour la première AMDEC fonctionnelle effectuée en début de flot de conception. Elle fournit des propriétés bloc par bloc qui peuvent permettre à l'ingénieur système de spécifier à chacun des ingénieurs matériel et logiciel des propriétés pour le développement de leur bloc. Cette méthodologie propose donc en plus un moyen de communication simple entre les différents ingénieurs (systèmes, matériels, logiciel, sûreté) à travers la définition de propriétés bloc par bloc.

Nous avons donc proposé une méthodologie pour l'évaluation de la sûreté de systèmes complexes. Cette méthodologie a été appliquée à un premier cas d'étude dans un contexte aéronautique, ce qui nous a permis d'imaginer différentes perspectives pour des travaux futurs.

Perspectives

Application à de nombreux cas d'étude La principale limitation de ce travail est que la méthodologie a été appliquée à un seul cas d'étude. La première perspective envisagée est alors d'appliquer l'approche sur différents cas d'études. L'application à différents cas d'études présenterait les intérêts et améliorations suivantes :

— Définir des règles et standard sur le développement de modèle de haut-niveau, afin de pouvoir ensuite converger plus rapidement vers des modèles de haut-niveau de qualité. Dans notre cas d'étude, c'est la prise en compte des interruptions qui a entraîné des erreurs de modélisation.

— Définir des modèles de fautes de haut-niveau plus complexes : dans ce manuscrit nous nous sommes concentrés sur l'injection de fautes par amplitude, sur un seul signal à la fois. Ce modèle était adapté au cas d'étude car il y avait une faible interdépendance entre les signaux de sortie des blocs, et l'injection d'amplitude d'erreur convenait au type de données qui transitaient. De nouveaux modèles de fautes peuvent être envisagés comme des fautes de haut-niveau multiples, ou des injections de délais par exemple.

— Définir des propriétés sur les signaux de sorties des blocs plus complexes : les propriétés définies actuellement sont définies sous formes d'intervalles, et nous avons montrés que ce fonctionnement est adapté lorsque la variation de l'erreur en sortie en fonction de l'amplitude de la faute injectée est monotone. Il serait intéressant de définir des propriétés plus complexes afin de pouvoir appliquer la méthodologie à un spectre plus large de cas d'étude.

Analyse des blocs matériels et logiciels Notre méthodologie actuelle propose une simulation de fautes RTL uniquement sur les descriptions matérielles. Une deuxième perspective serait alors d'étendre l'étape de simulation de fautes sur les descriptions matérielles des microprocesseurs. Dans le cadre du projet SafeAir, une étude préliminaire a déjà été effectuée en modélisant certaines parties du système étudié sur un simulateur de RISCV. L'étude a montré les intérêts suivants :

— Pertinence des propriétés critiques : il a été simple de spécifier à la personne travaillant sur le RISCV les intervalles critiques. Une simulation d'injection de fautes a permis de montrer que les intervalles extraits à haut-niveau continuent d'être pertinents avec une implémentation sur microprocesseur.

— Il a été possible de détecter des étages ou des parties du microprocesseur qui semblaient être plus vulnérables.

Dans ce sens, une étude plus poussée avec des simulations de fautes sur les parties matérielles de microprocesseurs est une perspective qui viendrait compléter la méthodologie.

Injection de fautes RTL multiples L'injection de faute actuellement effectuée au niveau RTL est une injection de fautes simples correspondant à des SEUs. Les dimensions de plus en plus réduites, les conditions de vols en haute altitude, rendent la probabilité de fautes multiples (MBUs) élevée. Il pourrait être intéressant de développer également au niveau RTL des modèles de fautes plus complexes, correspondant à des fautes multiples. Cela pourrait permettre de mettre en évidence des problèmes de modélisation haut-niveau plus complexes, en se propageant sur différents signaux d'interfaces, ou en créant des comportements plus difficiles à modéliser.

Publications

Au cours de la thèse, plusieurs publications ont étés présentées :

— Julie Roux, Vincent Beroulle, Katell Morin-Allory, Régis Leveugle, Lilian Bossuet, Frédéric Cézilly, Frédéric Berthoz, Gilles Genévrier, François Cerisier (2020, April). Cross layer fault simulations for analyzing the robustness of RTL designs in airborne systems. In 2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS) (pp. 1-4). IEEE. Ce papier présente l'étape de simulation de haut-niveau et l'extraction d'intervalle critiques appliqué au cas d'étude aéronautique.

— Julie Roux, Vincent Beroulle, Katell Morin-Allory, Régis Leveugle, Lilian Bossuet, Frédéric Cézilly, Frédéric Berthoz, Gilles Genévrier, François Cerisier (2020, November). High Level Fault Injection Method for Evaluating Critical System Parameter Ranges. In 2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS) (pp. 1-4). IEEE. Ce papier présente une vue d'ensemble des trois étapes composant le flot de la méthodologie.

— Julie Roux, Vincent Beroulle, Katell Morin-Allory, Régis Leveugle, Lilian Bossuet, Frédéric Cézilly, Frédéric Berthoz, Gilles Genévrier, François Cerisier (2021). High-level fault injection to assess FMEA on critical systems. Microelectronics Reliability, 122, 114135. Cet article de journal présente la méthodologie dans son ensemble.

— Julie Roux, Vincent Beroulle, Katell Morin-Allory, Régis Leveugle, Lilian Bossuet, Frédéric Cézilly, Frédéric Berthoz, Gilles Genévrier, François Cerisier (2021, October). Cross-layer Approach to Assess FMEA on Critical Systems and Evaluate High-Level Model Realism. In 29th IFIP/IEEE International Conference on very Large Scale Integration, VLSI-SoC 2021. Cet article présente l'approche itérative proposée pour exploiter le flot.

Bibliographie

[1] *Environmental Consequences of the Chernobyl Accident and their Remediation : Twenty Years of Experience*, ser. Radiological Assessment Reports Series. Vienna : International Atomic Energy Agency, 2006, no. 8.

[2] *Final report on the accident on 1st June 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janeiro–Paris*. Paris : Bureau d'Enquêtes et d'Analyses pour la sécurité de l'aviation civile, 2012.

[3] S. Imai, E. Blasch, A. Galli, W. Zhu, F. Lee, and C. A. Varela, "Airplane flight safety using error-tolerant data stream processing," *IEEE Aerospace and Electronic Systems Magazine*, vol. 32, no. 4, pp. 4–17, 2017.

[4] N. Bidokhti, "How to Close the Gap between Hardware and Software Using FMEA," in *2007 Annual Reliability and Maintainability Symposium*, 2007, pp. 167–172.

[5] N. Ozarin, "Lessons Learned on Five Large-Scale System Developments," in *2007 1st Annual IEEE Systems Conference*, 2007, pp. 1–7.

[6] A. D. Plessis, K. Frank, M. Saglimbene, and N. Ozarin, "The thirty greatest reliability challenges," in *2014 Reliability and Maintainability Symposium*, 2014, pp. 1–6.

[7] N. Ozarin, "Bridging software and hardware FMEA in complex systems," in *2013 Proceedings Annual Reliability and Maintainability Symposium (RAMS)*, 2013, pp. 1–6.

[8] R. Edwards, C. Dyer, and E. Normand, "Technical standard for atmospheric radiation single event effects, (see) on avionics electronics," in *2004 IEEE Radiation Effects Data Workshop (IEEE Cat. No.04TH8774)*, 2004, pp. 1–5.

[9] *IEC 62396-1 : Process management for avionics – Atmospheric radiation effects – Part 1 : Accommodation of atmospheric radiation effects via single event effects within avionics electronic equipment*. International Electrotechnical Commission, 2006.

[10] *Guidelines for development of civil aircraft and systems*. ARP4754A, SAE International, 2010.

[11] R. Fulton and R. Vandermolen, *Airborne Electronic Hardware Design Assurance : A Practitioner's Guide to RTCA/DO-254*. USA : CRC Press, Inc., 2014.

[12] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[13] J. Maiz and N. Seifert, "Introduction to the special issue on soft errors and data integrity in terrestrial computer systems," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 303–304, 2005.

[14] D. Binder, E. C. Smith, and A. B. Holman, “Satellite anomalies from galactic cosmic rays,” *IEEE Transactions on Nuclear Science*, vol. 22, no. 6, pp. 2675–2680, 1975.

[15] T. C. May and M. H. Woods, “A New Physical Mechanism for Soft Errors in Dynamic Memories,” in *16th International Reliability Physics Symposium*, 1978, pp. 33–40.

[16] M. Nicolaidis, “Design for soft error mitigation,” *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 405–418, 2005.

[17] A. Manuzzato, P. Rech, S. Gerardin, A. Paccagnella, L. Sterpone, and M. Violante, “Sensitivity evaluation of TMR-hardened circuits to multiple SEUs induced by alpha particles in commercial SRAM-based FPGAs,” in *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*. IEEE, 2007, pp. 79–86.

[18] R. Velazco, T. Calin, M. Nicolaidis, S. Moss, S. LaLumondiere, V. Tran, and R. Koga, “SEU-hardened storage cell validation using a pulsed laser,” *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2843–2848, 1996.

[19] F. Vargas, D. Cavalcante, E. Gatti, D. Prestes, and D. Lupi, “On the proposition of an EMI-based fault injection approach,” in *11th IEEE International On-Line Testing Symposium*. IEEE, 2005, pp. 207–208.

[20] D. H. Habing, “The Use of Lasers to Simulate Radiation-Induced Transients in Semiconductor Devices and Circuits,” *IEEE Transactions on Nuclear Science*, vol. 12, no. 5, pp. 91–100, 1965.

[21] P. Jaulent, “Etude des effets singuliers transitoires dans les amplificateurs opérationnels linéaires par photogénération impulsionnelle non-linéaire,” Theses, Université Bordeaux 1, Jun. 2009. [Online]. Available : https://tel.archives-ouvertes.fr/tel-00997385

[22] V. Pouget, D. Lewis, H. Lapuyade, R. Briand, P. Fouillat, L. Sarger, and M.-C. Calvet, “Validation of radiation hardened designs by pulsed laser testing and SPICE analysis,” *Microelectronics Reliability*, vol. 39, no. 6, pp. 931–935, 1999, european Symposium on Reliability of Electron Devices, Failure Physics and Analysis. [Online]. Available : https://www.sciencedirect.com/science/article/pii/S0026271499001250

[23] J. Samson, W. Moreno, and F. Falquez, “Validating fault tolerant designs using laser fault injection (LFI),” in *1997 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 1997, pp. 175–183.

[24] S. Duzellier, D. Falguere, L. Guibert, V. Pouget, P. Fouillat, and R. Ecoffet, “Application of laser testing in study of SEE mechanisms in 16-Mbit DRAMs,” *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2392–2399, 2000.

[25] A. Ejlali and S. G. Miremadi, “Error propagation analysis using FPGA-based SEU-fault injection,” *Microelectronics Reliability*, vol. 48, no. 2, pp. 319–328, 2008.

[26] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia, and L. Entrena, “Autonomous fault emulation : A new FPGA-based acceleration system for hardness evaluation,” *IEEE Transactions on Nuclear Science*, vol. 54, no. 1, pp. 252–261, 2007.

[27] D. de Andres, J. C. Ruiz, D. Gil, and P. Gil, “Fault emulation for dependability evaluation of VLSI systems,” *IEEE transactions on very large scale integration (VLSI) systems*, vol. 16, no. 4, pp. 422–431, 2008.

[28] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Exploiting FPGA-based techniques for fault injection campaigns on VLSI circuits," in *Proceedings 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2001, pp. 250–258.

[29] "SystemVerilog 3.1," Standard, 2003. [Online]. Available : //www.eda.org/sv/SystemVerilog3.1final.pdf

[30] T. Stoecklein and J. Bäsig, "Handel-C-an effective method for designing FPGAs (and ASICs)," *Fachbereich Nachrichten-und Feinwerktechnik*, 2003.

[31] D. Rainer, G. Andreas, and G. Daniel, "SpecC Language Reference Manual," Standard, 2002. [Online]. Available : http://www.ics.uci.edu/_specc/reference/SpecCLRM20.pdf

[32] "Open SystemC initiative," Standard. [Online]. Available : www.systemc.org

[33] F. Ghenassia, *Transaction-level modeling with SystemC*. Springer, 2005, vol. 2.

[34] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "SoCRocket - A virtual platform for the European Space Agency's SoC development," in *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2014, pp. 1–7.

[35] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "SystemC cosimulation and emulation of multiprocessor SoC designs," *Computer*, vol. 36, no. 4, pp. 53–59, 2003.

[36] L. Cai and D. Gajski, "Transaction level modeling : an overview," in *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (IEEE Cat. No. 03TH8721)*. IEEE, 2003, pp. 19–24.

[37] OSCI, *OSCI TLM-2.0 Language Reference Manual*, 2009. [Online]. Available : https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf

[38] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[39] J. Gracia, J. C. Baraza, D. Gil, and P. J. Gil, "Comparison and application of different VHDL-based fault injection techniques," in *Proceedings 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE, 2001, pp. 233–241.

[40] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models : the MEFISTO tool," in *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing*, 1994, pp. 66–75.

[41] A. Amendola, A. Benso, F. Corno, L. Impagliazzo, P. Marmo, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda, "Fault behavior observation of a microprocessor system through a VHDL simulation-based fault injection experiment," in *Proceedings EURO-DAC '96. European Design Automation Conference with EURO-VHDL '96 and Exhibition*, 1996, pp. 536–541.

[42] J. Boue, P. Petillon, and Y. Crouzet, "MEFISTO-L : a VHDL-based fault injection tool for the experimental assessment of fault tolerance," in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, 1998, pp. 168–173.

[43] J. Armstrong, F. Lam, and P. Ward, “Test generation and fault simulation for behavioural models,” in *Performance and Fault Modelling with VHDL, Englewood Cliffs, Prentice Hall*, 1992, pp. 240–303.

[44] S. Ghosh and T. J. Chakraborty, “On behavior fault modeling for digital designs,” *Journal of Electronic Testing*, vol. 2, no. 2, pp. 135–151, Jun 1991. [Online]. Available : https://doi.org/10.1007/BF00133499

[45] T. Delong, B. Johnson, and J. Profeta, “A fault injection technique for vhdl behavioral-level models,” *IEEE Design Test of Computers*, vol. 13, no. 4, pp. 24–33, 1996.

[46] V. Sieh, O. Tschache, and F. Balbach, “Verify : evaluation of reliability using vhdl-models with embedded fault descriptions,” in *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, 1997, pp. 32–36.

[47] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, “Quantitative evaluation of soft error injection techniques for robust system design,” in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC ’13. New York, NY, USA : Association for Computing Machinery, 2013. [Online]. Available : https://doi.org/10.1145/2463209.2488859

[48] P. Georgelin and V. Krishnaswamy, “Towards a C++-based design methodology facilitating sequential equivalence checking,” in *2006 43rd ACM/IEEE Design Automation Conference*. IEEE, 2006, pp. 93–96.

[49] N. Bombieri, F. Fummi, G. Pravadelli, and J. Marques-Silva, “Towards equivalence checking between TLM and RTL models,” in *2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*. IEEE, 2007, pp. 113–122.

[50] N. Bombieri, F. Fummi, and G. Pravadelli, “A mutation model for the SystemC TLM 2.0 communication interfaces,” in *2008 Design, Automation and Test in Europe*. IEEE, 2008, pp. 396–401.

[51] A. Miele, “A methodology for the design and the analysis of reliable embedded systems,” Ph.D. dissertation, Politecnico di Milano, 2010.

[52] A. Miele, “A fault-injection methodology for the system-level dependability analysis of multiprocessor embedded systems,” *Microprocess. Microsystems*, vol. 38, no. 6, pp. 567–580, 2014. [Online]. Available : https://doi.org/10.1016/j.micpro.2014.05.008

[53] G. Beltrame, L. Fossati, and D. Sciuto, “Resp : A nonintrusive transaction-level reflective mpsoc simulation platform for design space exploration,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 12, pp. 1857–1869, 2009.

[54] V. Herdt, H. M. Le, D. Grosse, and R. Drechsler, “On the application of formal fault localization to automated RTL-to-TLM fault correspondence analysis for fast and accurate vp-based error effect simulation - a case study,” in *2016 Forum on Specification and Design Languages (FDL)*, 2016, pp. 1–8.

[55] D. Mueller-Gritschneder, P. R. Maier, M. Greim, and U. Schlichtmann, “SystemC-based multi-level error injection for the evaluation of fault-tolerant systems,” in *2014 International Symposium on Integrated Circuits (ISIC)*, 2014, pp. 460–463.

[56] A. Vallero, S. Tselonis, N. Foutris, M. Kaliorakis, M. Kooli, A. Savino, G. Politano, A. Bosio, G. Di Natale, D. Gizopoulos, and S. Di Carlo, "Cross-layer reliability evaluation, moving from the hardware architecture to the system level : A CLERECO EU project overview," *Microprocessors and Microsystems*, vol. 39, no. 8, pp. 1204 – 1214, 2015. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S0141933115000824

[57] J. Perez, M. Azkarate-askasua, and A. Perez, "Codesign and simulated fault injection of safety-critical embedded systems using SystemC," in *2010 European Dependable Computing Conference*, 2010, pp. 221–229.

[58] R. Mariani, G. Boschi, and F. Colucci, "Using an innovative SoC-level FMEA methodology to design in compliance with IEC61508," in *2007 Design, Automation Test in Europe Conference Exhibition*, 2007, pp. 1–6.

[59] B. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, "Fault-effect analysis on system-level hardware modeling using virtual prototypes," in *2016 Forum on Specification and Design Languages (FDL)*, 2016, pp. 1–7.

[60] *Transport safety performance in the EU : a statistical overview*, 2003. [Online]. Available : https://etsc.eu/wp-content/uploads/2003_transport_safety_stats_eu_overview.pdf

[61] S. ARP4761, *Guidelines and methods for conducting the safety assessment process on airborne systems and equipments*, 1996.

[62] V. Hilderman and T. Baghi, *Avionics certification : a complete guide to DO-178 (software), DO-254 (hardware)*. Avionics Communications, 2007.

[63] "Single event effects (SEE) caused by atmospheric radiation," Standard. [Online]. Available : https://www.easa.europa.eu/sites/default/files/dfu/CM-AS-004%20Issue%2001.pdf

[64] Y. Papadopoulos, D. Parker, and C. Grante, "Automating the failure modes and effects analysis of safety critical systems," in *Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings.* IEEE, 2004, pp. 310–311.

[65] L. Grunske, P. Lindsay, N. Yatapanage, and K. Winter, "An automated failure mode and effect analysis based on high-level design specification with behavior trees," in *International Conference on Integrated Formal Methods*. Springer, 2005, pp. 129–149.

[66] J. Elmqvist and S. Nadjm-Tehrani, "Tool support for incremental failure mode and effects analysis of component-based systems," in *Proceedings of the conference on Design, automation and test in Europe*, 2008, pp. 921–927.

[67] Z. Yuan, Y. Chen, and N. Tang, "An integrated method for hardware FMEA of new electronic products," in *2016 Prognostics and System Health Management Conference (PHM-Chengdu)*, 2016, pp. 1–6.

[68] A. Arnold, G. Point, A. Griffault, and A. Rauzy, "The altarica formalism for describing concurrent systems," *Fundam. Informaticae*, vol. 40, no. 2-3, pp. 109–124, 1999.

[69] G. Point and A. Rauzy, "Altarica : Constraint automata as a description language," 1999.

[70] R. Bernard, J.-J. Aubert, P. Bieber, C. Merlini, and S. Metge, "Experiments in model based safety analysis : Flight controls," *IFAC Proceedings Volumes*, vol. 40, no. 6, pp. 43–48, 2007, 1st IFAC Workshop on Dependable Control of Discrete Systems. [Online]. Available : https://www.sciencedirect.com/science/article/pii/S1474667015310934

[71] R. A. Shafik, P. Rosinger, and B. M. Al-Hashimi, "Systemc-based minimum intrusive fault injection technique with improved fault representation," in *2008 14th IEEE International On-Line Testing Symposium*, 2008, pp. 99–104.

[72] "IEEE 1850-2010 - IEEE standard for property specification language (PSL)," Standard. [Online]. Available : https://standards.ieee.org/standard/1850-2010.html

[73] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection : Quantified error and confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*, 2009, pp. 502–506.

[74] T. Fawcett, "An introduction to roc analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

Table des figures

Liste des tableaux

Evaluation de la sûreté des systèmes aéronautiques grâce aux plateformes virtuelles

Safety Evaluation of Aircraft Systems using Virtual Platforms

Résumé

Les systèmes embarqués critiques sont soumis à des standards très stricts. Les analyses de sûretés de ces systèmes sont souvent effectuées empiriquement, et s'appuient sur l'expérience des ingénieurs. Faire des analyses de sûretés non automatisées sur des systèmes complexes est un enjeu majeur qui conduit les ingénieurs à supposer systématiquement le pire cas. De ce fait, les contremesures sont surdimensionnées. Beaucoup de techniques d'évaluation de la robustesse des systèmes matériels fondées sur la simulation d'injection de fautes existent au niveau transfert de registre (RTL). Avec ces techniques, la robustesse est évaluée en comparant le circuit fauté et le circuit de référence. Ces méthodes ne prennent pas en compte la spécification du système, et l'environnement (ex : logiciel). De plus, elles demandent beaucoup de temps de simulation pour des circuits complexes. Par ailleurs, des solutions de simulation de fautes haut-niveau ou multi-abstraction ont été proposées, mais ne prennent pas en compte la spécification et l'environnement système. La simulation est accélérée au détriment du réalisme. Dans cette thèse, nous présentons un flot itératif multi-abstraction pour évaluer la robustesse des circuits complexes. Le flot proposé prend en compte la spécification, l'environnement, et propose une technique d'évaluation du réalisme des modèles de haut-niveau. Ce flot suit la logique des flots de développement top-down traditionnelles, et les recommandations des standards. La première étape du flot est une simulation de haut-niveau conduisant à l'extraction de propriétés critiques pour chaque bloc. Ces propriétés sont réutilisées dans l'étape de simulation RTL afin d'obtenir une classification des fautes. Enfin, une étape de co-simulation nous permet de définir des métriques quantifiant la qualité du modèle. Cette thèse propose une approche itérative pour utiliser ce flot : les premières rondes permettent d'obtenir une évaluation rapide du réalisme du modèle haut-niveau et une première classification des fautes, la dernière ronde permet d'avoir une classification de fautes RTL précise. L'approche a été appliquée à un cas d'étude dans le contexte aéronautique.

Mots-clés : Plateforme virtuelle, Simulation de fautes, Modélisation Système, Tolérance aux fautes, AMDEC, FPGA

Abstract

Embedded systems in critical applications are constrained by very strict standards, but safety analysis of these systems is often empirical and mainly relies on the engineers experience. Performing empirical analyses on complex designs is a major challenge that leads engineers to make very pessimistic assumptions and consequently to over-design multiple countermeasures. Many fault injection techniques have been developed to evaluate the robustness of Register Transfer Level (RTL) hardware designs. With these techniques, robustness is evaluated by comparing the faulty circuit outputs with the circuit specifications or golden RTL fault-free simulations. However, these techniques are too circuit centered. Besides with complex hardware designs, fault simulations become very time-consuming. Some high-level or cross-layer fault injection techniques has been proposed, but none of them propose to take into account the specification. Furthermore, high-level fault simulation is fast to the detriment of the realism. In this thesis, we present a a new iterative cross-layer robustness analysis taking into account the overall critical system specifications andverifying the realism of high-level models. This flow is consistent with the traditional top-down flow and in compliance with standard recommandations. The first step of the flow is a high-level fault simulation that leads to extract critical parameter ranges. Then, these ranges are used to evaluate the robustness of each RTL block in the circuit. In the last step, we compute some metrics reflecting the realism of the high-level models. According to these metrics,we can determine if the high-level models must be improved. The thesis proposes an approach using this flow, based on a first round "quick and dirty" that permits to obtain an early evaluation of the high-level model, and a last round accurate to obtain a classification of RTL faults. We apply this flow to a case study of a real airborne system.

Keywords : Virtual Platform, Fault Simulation, System Modeling, Fault Tolerance, FMEA, FPGA