Systèmes embarqués temps réel fiables et adaptables

Dimitry Solet

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE NANTES

COMUE UNIVERSITÉ BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Electronique*

Par

Dimitry SOLET

Systèmes embarqués temps réel fiables et adaptables

Thèse présentée et soutenue à Nantes, le 9 novembre 2020
Unité de recherche : IETR / LS2N

Rapporteurs avant soutenance :

Katell MORIN-ALLORY	Maître de conférences, INP de Grenoble
Claire PAGETTI	Ingénieur de recherche, ONERA Toulouse

Composition du Jury :

Président :	Frank SINGHOFF	Professeur des universités, Université de Bretagne Occidentale
Examinateurs :	Abdoulaye GAMATIE	Directeur de recherche CNRS, LIRMM Montpellier
	Katell MORIN-ALLORY	Maître de conférences/HDR, INP de Grenoble
	Claire PAGETTI	Ingénieur de recherche/HDR, ONERA Toulouse
Dir. de thèse :	Sébastien PILLEMENT	Professeur des universités, Université de Nantes
Co-enc. de thèse :	Mikaël BRIDAY	Maître de conférences, Ecole Centrales de Nantes

Invité(s)

Jean-Luc BECHENNEC	Chargé de recherche CNRS, Ecole Centrales de Nantes
Sébastien FAUCOU	Maître de conférences, Université de Nantes

Remerciements

Les travaux présentés dans ce manuscrit ont été effectués conjointement à l'Institut d'Électronique et des Technologies du numéRique (IETR) au sein de l'équipe SYSCOM de Nantes, et au Laboratoire des Sciences du Numérique de Nantes (LS2N) au sein de l'équipe STR.

Je tiens tout d'abord à remercier mon équipe encadrante, Sébastien Pillement directeur de thèse, Mikaël Briday co-encadrant de thèse, Jean-Luc Béchennec et Sébastien Faucou, pour m'avoir fourni tous les moyens nécessaires à la mise en œuvre de ces travaux. Leurs conseils, leurs disponibilités et leurs connaissances scientifiques m'ont permis de mener à bien mon projet de thèse.

J'adresse mes remerciements à Mme Katell Morin-Allory et à Mme Claire Pagetti pour avoir accepté d'être rapporteuses de mon mémoire de thèse. Je remercie également M. Abdoulaye Gamatie et M. Frank Singhoff pour avoir examiné mon travail.

Je tiens également à remercier les membres de l'équipe SYSCOM de l'IETR et de l'équipe STR, ainsi que mes collègues du département GEII de l'IUT de Nantes avec qui j'ai eu l'occasion d'enseigner en parallèle de ces travaux.

Mes derniers remerciements sont adressés à ma famille et plus particulièrement à ma compagne Prescillia qui m'a encouragé et soutenu au quotidien, ce qui a fortement contribué à l'accomplissement de mon travail.

SOMMAIRE

INTRODUCTION GÉNÉRALE

Contexte de l'étude

Un système embarqué est un circuit électronique construit autour d'un système informatique qui s'intègre dans un équipement afin de réaliser une ou plusieurs fonctions spécifiques. Ces systèmes sont présents partout dans notre environnement : on les retrouve dans nos appareils électroménagères, dans les transports, dans les machines de production, etc. Selon leurs fonctions et l'environnement dans lequel ils interagissent ces systèmes peuvent être qualifiés de critiques. Un système est considéré comme critique si une défaillance de celui-ci à des conséquences "graves". Par exemple, **2 exemples**. Ces systèmes doivent donc être conçus de façon à être sûr de fonctionnement.

Lors de la conception d'un système embarqué, plusieurs techniques permettent de vérifier que celui-ci est correct vis-à-vis de ses spécifications. On peut notamment utiliser le test, la simulation ou la vérification des modèles. Ces techniques permettent de détecter et corriger les erreurs introduites pendant la phase de conception mais ne prennent pas en compte les erreurs pouvant survenir pendant la phase opérationnelle.

En effet plusieurs phénomènes peuvent impacter le système pendant son exploitation :

— Le vieillissement naturel des composants qui peut avoir un impact sur les temps de propagation des signaux.

— L'impact de particules qui peut entraîner la modification de l'état logique d'une ou plusieurs cellules mémoires.

Alors que le vieillissement naturel des composants peut être pris en compte lors de la conception d'un système (il est possible d'estimer selon les conditions d'utilisation la durée de vie d'un composant), l'impact de particule est un phénomène imprévisible qui requière l'utilisation de composants spécifiques pour s'en prémunir.

Ces phénomènes engendrent des fautes matérielles qui peuvent se propager au niveau du logiciel et ainsi générer une défaillance du système si elles ne sont pas détecté à temps. Il est donc nécessaire d'intégrer des mécanismes de détection et de correction d'erreurs afin qu'un système embarqué soit capable de fonctionner malgré l'apparition de ces erreurs, on parle alors de mécanisme de tolérance aux fautes.

Les systèmes embarqués que l'on considère dans nos travaux doivent être capables de s'intégrer dans un équipement et doivent être peu coûteux. Ainsi ces systèmes disposent de ressources limités que ce soit en terme d'espace mémoire ou de puissance de calcul. Le matériel et le logiciel sont donc conçus au plus proche des besoins. De ce fait les mécanismes de tolérances aux fautes mis en œuvre doivent avoir un impact le plus faible possible sur le système.

Objectif de la thèse

L'objectif de la thèse est de définir une infrastructure logicielle et matérielle permettant de concevoir des systèmes embarqués temps réels sûrs de fonctionnement. Les architectures des systèmes embarqués que l'on considère dans nos travaux sont caractérisés par leurs ressources limités en terme d'espace mémoire et de puissance de calcul. De plus on considère des systèmes temps réels, c'est-à-dire des systèmes qui doivent respecter des contraintes temporelles.

On propose d'implémenter un mécanisme de surveillance permettant de détecter les erreurs au niveau du logicielle avant que celles-ci ne se propagent et fassent défaillir le système. Ce mécanisme de surveillance est basé sur un ensemble de moniteurs qui analyse la trace d'exécution du système afin de donner un verdict le respect de spécification à vérifier.

Généralement ces moniteurs sont implémentés soit de façon logicielle en modifiant le code de l'application afin d'ajouter du code, soit de façon matérielle en utilisant des circuits logiques sur lesquelles les moniteurs et le processeur à surveiller sont synthétisés. Concernant l'implémentation logicielle, le principal inconvénient est le surcoût d'exécution temporel dû à la séquentialité de la vérification. Concernant l'implémentation matérielle, l'inconvénient est due à l'utilisation de processeurs "softcore", ces derniers ne sont pas réaliste du point de vue de l'industrie du fait de leur performance qui est moins adapté que des processeurs "hardcore".

Pour lever ces inconvénients d'implémentation de moniteurs, on considère l'utilisation d'une architecture de type SoPC (System on Programmable Chip) qui intègre sur la même puce un circuit logique programmable sur lequel seront implémenté les moniteurs et un processeur "hardcore" sur lequel s'exécutera le logiciel à vérifier qui sera instrumenté de façon à générer une trace d'exécution. Cette implémentation, que l'on peut qualifier d'hybride vis-à-vis des deux autres, permet de proposer un mécanisme ayant un faible

surcoût d'exécution dû fait que la vérification se fait de façon concurrente.

De plus, pour évaluer le mécanisme que l'on propose, nous considérons le cas où le logiciel à vérifier est le noyau d'un système d'exploitation temps réel. Ce dernier doit, entre autres, garantir qu'une erreur sur l'un des processus de l'application ne se propage pas aux autres processus. Il est donc important qu'il soit fiable afin d'améliorer la fiabilité du système.

Organisation du manuscrit

Ce manuscrit se compose de cinq chapitres.

Le chapitre 1 donne le contexte de la thèse. Nous expliquerons ce qu'est un système embarqué temps réel, puis nous y introduirons les concepts de la sûreté de fonctionnement, ensuite nous présenterons les principaux types de mécanismes de détection et enfin nous parlerons de la sûreté de fonctionnement pour les systèmes d'exploitation temps réel.

Le chapitre 2 introduit le concept de la vérification en ligne, une technique de vérification basée sur l'utilisation de moniteurs pour surveiller l'exécution d'un système. Nous parlerons des différents langages permettant de spécifier les propriétés sur lesquelles sont basés les moniteurs. Puis nous verrons les principales méthodes de synthèse et d'implémentation des moniteurs.

Le chapitre 3 présente l'implémentation du dispositif de vérification en ligne que l'on met en œuvre. Nous expliquerons comment ce dispositif est synthétisé et comment il est implémenté sur un SoPC.

Le chapitre 4 porte sur l'utilisation du dispositif de vérification en ligne pour surveille un système d'exploitation temps réel.

Le chapitre 5 présente l'évaluation réalisée à partir d'une campagne d'injection de fautes.

Chapitre 1

Sûreté de fonctionnement des systèmes embarqués temps réel

1.1 Les systèmes embarqués temps réel

1.1.1 Définition et caractérisation d'un système embarqué

Définition

Un système embarqué est un assemblage complexe de composants matériels et de logiciels qui sont conçus ensembles pour réaliser une ou plusieurs tâches spécifiques. L'adjectif embarqué est utilisé pour désigner des systèmes très différents les uns des autres. Par exemple un smartphone ou une tablette peuvent être considérés comme des systèmes embarqués dans la mesure où ils sont transportables. Cependant du point du vue de leur utilisation ils s'apparentent plus à un ordinateur de bureau dans le sens où ils ne sont pas conçus pour réaliser une tâche spécifique. Nous restreignons ici la définition de « systèmes embarqués » aux systèmes qui sont intégrés à un autre système comme un véhicule ou une machine de production, on définit ces systèmes embarqués comme enfouis.

Comme montré sur la Figure 1.1, un système embarqué est construit autour d'un système informatique qui reçoit des informations provenant de capteurs et qui interagit avec l'environnement à partir d'actionneurs et/ou d'afficheurs. Les capteurs mesurent les grandeurs physiques caractéristiques de l'environnement afin de déterminer son état courant. Ces informations sont converties numériquement pour être traitées par le système informatique qui va produire un résultat en fonction de l'état de l'environnement. Ce résultat est converti et transmis aux actionneurs pour conduire l'environnement dans l'état escompté.

L'environnement d'un système embarqué se compose de deux parties :

— *L'environnement fonctionnel :* il désigne l'environnement qui est en interaction direct avec le système embarqué. Il peut s'agir d'un autre système, d'un procédé

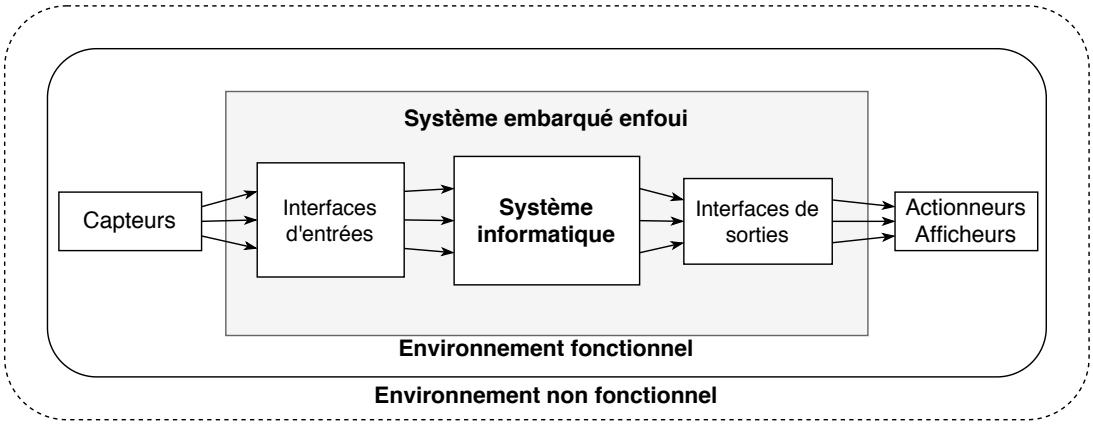

FIGURE 1.1 – Système embarqué

industriel à contrôler ou d'un individu.

— *L'environnement non fonctionnel :* il désigne l'environnement extérieur au système embarqué qui n'est pas contrôlé par celui-ci. Cet environnement va imposer des contraintes et des paramètres d'utilisations (par exemple la température ou le taux d'humidité) qui doivent être pris en compte lors de la conception du système.

Les contraintes

Ces systèmes sont caractérisés par un ensemble de contraintes liées à leur intégration dans un environnement.

Au niveau matériel, on retrouve les contraintes suivantes :

— *Mécaniques* - Ces systèmes doivent pouvoir s'intégrer dans l'environnement fonctionnel pour lequel ils sont prévus, cela implique des contraintes d'encombrement. De plus ces systèmes doivent supporter les contraintes liées à l'environnement comme les vibrations, l'humidité, la chaleur, *etc.*

— *Énergétique* - Ces systèmes sont souvent « autonomes » du point de vue énergétique. Ils doivent fonctionner malgré une source d'énergie limitée comme une batterie.

— *Économique* - Ces systèmes peuvent être fabriqués en grande série, ils doivent donc être rentables.

Ces contraintes ont des conséquences directes sur les choix technologiques mis en œuvre pendant leur conception et imposent une limitation au niveau des ressources disponibles. Cette limitation des ressources va principalement se retrouver au niveau de l'espace mémoire et de la puissance de calcul disponible. Un système embarqué est donc souvent

conçu au plus juste pour répondre aux besoins du service qu'il doit délivrer.

De plus, selon leur application, ces systèmes doivent répondre à des exigences temporelles plus ou moins fortes et avoir un niveau de fiabilité adéquate.

La notion de criticité

Un système embarqué est dit « critique » si une défaillance de celui-ci peut avoir de graves conséquences comme la destruction du système, la mise en danger d'individus ou la dégradation de l'environnement. On retrouve ces systèmes dans différents secteurs comme par exemple :

— le transport : pilotage d'avion, de train, de voiture... ;
— l'énergie : le système de contrôle d'une centrale nucléaire ;
— le médical : le système de pompe à perfusion.

Ces système doivent donc être sûrs de fonctionnement.

La notion de criticité d'un système est souvent évaluée selon la nature de la conséquence possible. Par exemple dans le domaine aéronautique les normes ED-12C et DO-178C définissent 5 niveaux de criticité (de A à E). Le niveau E est attribué aux systèmes dont une défaillance n'a pas d'effet sur la sécurité du vol et le niveau A est attribué aux systèmes qui peuvent provoquer un crash de l'avion.

Le développement des systèmes critiques est soumis à des normes qui dépendent du niveau de criticité. Ces normes permettent entre autres :

— d'assurer une documentation claire des différents composants du système,
— de limiter les pratiques dangereuses de développement comme l'utilisation de fonctions récursives ou l'allocation dynamique de mémoire,
— de valider le système à chaque étape de sa conception avec différentes méthodes de vérification,
— d'utiliser des outils de développement et de vérification eux-même sûrs.

En plus de cela, des mesures doivent être prises pour assurer la fiabilité de ces systèmes pendant leur exécution. Pour cela différentes techniques peuvent être utilisées comme l'utilisation de mécanismes de tolérance aux fautes.

1.1.2 Architecture des systèmes informatiques pour l'embarqué

L'architecture de base d'un système informatique est représentée sur la Figure 1.2. Elle se compose d'un ou plusieurs processeurs qui exécutent le code de du programme, de

mémoires qui stockent le code et les données du programme, et de multiples périphériques. Ces éléments communiquent entres eux à travers des bus de communication.

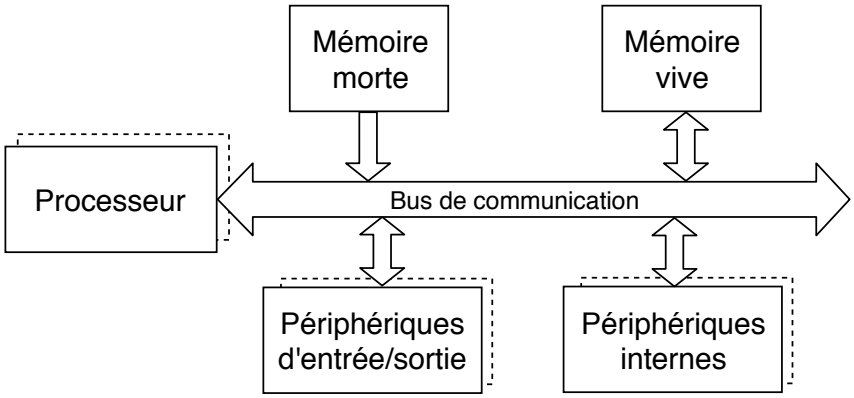

FIGURE 1.2 – Architecture matérielle de base d'un système informatique

La conception d'un système informatique peut se faire de différentes façons [17] :

— Par l'assemblage des différents éléments. De cette façon il possible de concevoir un système informatique au plus proche des besoins réels. Cependant cette approche requiert un temps de développement relativement long.

— Par l'utilisation d'un microcontrôleur qui intègre les ressources de bases (un processeur, de la mémoire et les périphériques de base) nécessaires au développement d'un système informatique.

— Par l'utilisation d'un système sur puce (*SoC* pour *system on a chip*) qui intègre tous les éléments nécessaires à la conception d'un système informatique : un ou plusieurs processeurs, de la mémoire, de multiples périphériques d'interfaces et même des capteurs. L'utilisation de SoCs permet de faciliter grandement le temps de développement d'un système informatique.

1.1.3 Les systèmes temps réel

Un système est qualifié de « temps réel » quand il est capable de contrôler un procédé physique à une vitesse adaptée par la prise en compte d'un ensemble d'exigences temporelles.

Selon l'importance des exigences temporelles à respecter, un système temps réel peut être qualifié de strict ou souple :

— *Le temps réel strict* qualifie les systèmes qui doivent impérativement respecter toutes leurs échéances. Ces systèmes sont généralement intégrés dans en environne-

ment critique où la violation d'une exigence temporelle peut avoir des conséquences dramatiques.

— *Le temps réel souple* qualifie les systèmes qui peuvent se permettre de violer certaines échéances. Ces systèmes sont, par exemple, présents dans le domaine du multimédia où une violation de propriété peut avoir des conséquences sur la qualité du service rendu.

Une application temps réel se compose de tâches, qui sont des séries d'actions à exécuter, qui doivent être ordonnancés par le système de façon à respecter leur priorité et leur date d'échéance. Ces tâches peuvent être :

— *Périodiques*, si elles doivent s'exécuter à intervalle régulier.

— *Sporadiques*, si elles peuvent s'exécuter à n'importe quel moment mais où il est possible de définir un temps minimal entre deux activation. Lors de l'étude de l'ordonnancement, il est souvent commode de se ramener à un modèle de tâche périodique.

— *Apériodiques*, si elles peuvent s'exécuter à n'importe quel moment sans pouvoir définir un temps minimal entre deux activations. Avec ce type de tâche, il impossible de garantir un ordonnancement.

On représente généralement une tâche périodique à travers le tuple r_0, P, C, D où :

— r_0 correspond à la date de réveil de la première instance de la tâche ;

— P correspond à la période de la tâche ;

— C correspond à la durée d'exécution de la tâche (le pire cas) ;

— D correspond au délai critique de la tâche, c'est à dire le temps maximum accepté entre le réveil d'une instance de tâche et sa terminaison.

Sur la Figure 1.3, on retrouve deux instances de tâche où r_i correspond à la date de réveil de la $i$ème instance de la tâche et d_i correspond à la date d'échéance du $i$ème instance. La première instance s'exécute sans être préemptée. La deuxième instance est préemptée et dépasse son échéance, ce qui dans le cas d'une tâche critique est une violation des exigences temporelles.

D'un point de vue logiciel, plusieurs architectures sont envisageables pour construire une application temps réel.

Par la programmation *bare metal* : qui consiste à développer un programme sans le support d'un système d'exploitation, donc sans l'abstraction de la plateforme matérielle du système. Contrairement au développement classique d'un programme dans une boucle

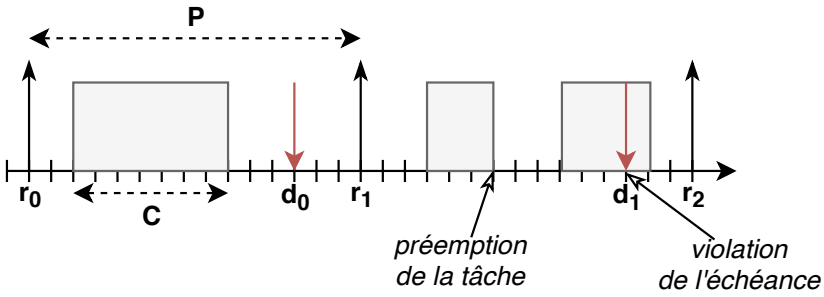

FIGURE 1.3 – Représentation des caractéristiques d'une tâche périodique

infinie, le développement d'une application temps réel se fait à partir de traitements cycliques par l'utilisation de *timers* (pour les tâches periodiques) et/ou événementiels par l'attente d'une interruption matérielle (pour les tâches non-periodiques) :

```
/**********************
*    Boucle infine     *
**********************/

Tant que (TOUJOURS) faire
    Lecture_des_entrees();
    Traitement();
    Ecriture_des_sortie();
Fin Tant que
```

```
/**********************
*  Traitement cyclique *
**********************/

Lorsque (top de l'horloge) faire
    Lecture_des_entrees();
    Traitement();
    Ecriture_des_sortie();
Fin Lorsque
```

```
/**********************
*Traitement événementiel*
**********************/

Lorsque (interruption) faire
    Lecture_des_entrees();
    Traitement();
    Ecriture_des_sortie();
Fin Lorsque
```

Ce type de programmation n'est possible que pour les applications les plus critiques pour lesquelles on n'a pas le choix. En effet lorsque l'application se compose de plusieurs tâches qui s'exécutent à des fréquences différentes, la prise en compte des différentes priorités devient difficile et nuit à la maintenabilité du code.

À partir d'un système d'exploitation temps réel (SETR) : Un SETR est un composant logiciel qui est en charge d'offrir un certain nombre de services afin de faciliter

la conception d'une application temps réel. Parmi ces services on peut citer :

— l'ordonnancement temps réel préemptif de tâches,
— la synchronisation entres plusieurs tâches,
— la mesure du temps,
— la prise en compte d'événements externes par l'intermédiaire d'interruptions.

De plus il permet d'ajouter une couche d'abstraction entre l'application et la plateforme matérielle. De ce fait, cela permet de concevoir des applications qui peuvent s'implémenter sur différentes plateformes matérielles.

À la différence d'un système d'exploitation généraliste, les services comme la gestion des périphériques et la gestion des systèmes de fichiers qui sont généralement pas utiles et ajoutent un surcoût d'utilisation de la mémoire.

1.2 Le principe de la sûreté de fonctionnement

1.2.1 Définition et typologie

La sûreté de fonctionnement est le domaine d'étude qui traite de l'aptitude d'un système à délivrer *un service de confiance justifié* [2]. Le service d'un système désigne son comportement attendu vis-à-vis de son environnement. C'est un domaine d'étude vaste qui regroupe plusieurs aspects comme montré sur la Figure 1.4. Dans cette section nous nous focaliserons principalement sur les concepts requis à la compréhension de nos travaux. On pourra trouver une approche plus générale dans [2, 4, 29, 27].

Les attributs

Les attributs d'un système permettent de caractériser les propriétés attendues d'un système et sa qualité de service. Les attributs de base de la sûreté de fonctionnement sont :

— *la disponibilité :* c'est l'aptitude d'un système à être prêt d'utilisation dans des conditions données ;
— *la fiabilité :* c'est l'aptitude d'un système à accomplir ses fonctions pendant son utilisation ;
— *la sécurité-innocuité :* c'est l'aptitude d'un système à assurer l'absence d'accident pouvant provoquer des conséquences sur des individus, sur l'environnement ou sur lui-même ;

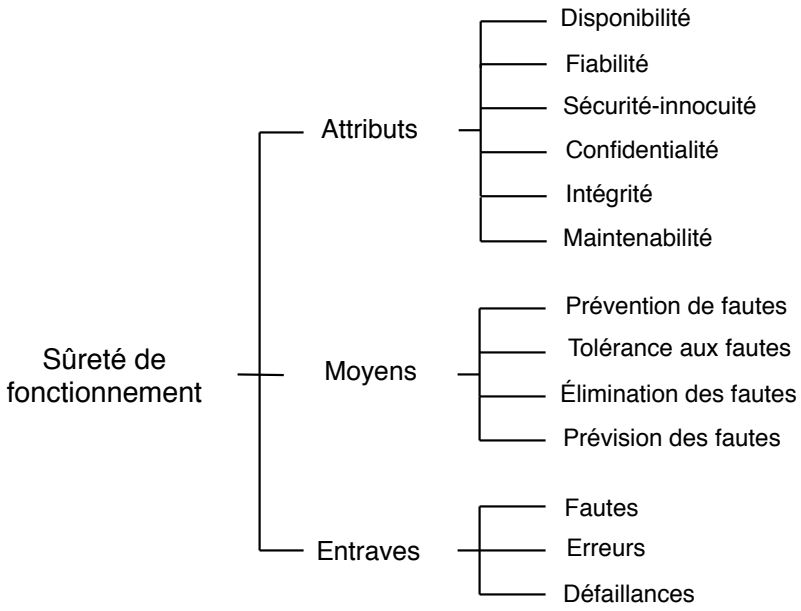

FIGURE 1.4 – Arbre de la sûreté de fonctionnement [2, 4]

— *la confidentialité :* c'est l'aptitude d'un système à ne pas divulguer des données confidentielles ;
— *l'intégrité :* c'est l'aptitude d'un système à détecter et/ou empêcher des altérations non-autorisées de ses données ;
— *la maintenabilité :* c'est l'aptitude d'un système à pouvoir être réparer et/ou évoluer.

Selon le domaine industriel considéré et du service à rendre, les attributs à attacher à un système seront différents. Par exemple, pour les systèmes qui gèrent la conduite dans le transport, la fiabilité et la sécurité-innocuité seront mises en avant. Alors que pour les systèmes qui gèrent des transactions bancaires ce sera la confidentialité qui sera principalement attendue. Cependant les attributs de disponibilité, d'intégrité et de maintenabilité sont forcément requis [2], mais à un niveau plus ou moins important selon l'application considérée.

Les entraves

Les entraves à la sûreté de fonctionnement sont les événements qui peuvent affecter un système et le conduire dans un état non souhaité. On distingue trois types d'entraves : les fautes, les erreurs et les défaillances, qui sont reliés entre elles comme montré sur la

Figure 1.5

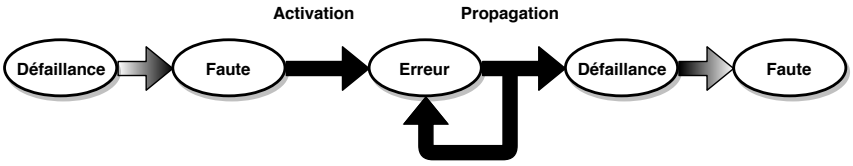

FIGURE 1.5 – Chaîne des entraves de la sûreté de fonctionnement [2]

Une faute est la cause de la non-sûreté de fonctionnement, comme nous le verrons dans la suite, ses origines peuvent être diverses. Tant qu'une faute est située dans une partie du système qui n'est pas utilisée par le processus de traitement, elle est considérée comme *dormante*. Lorsque la faute est activée par le processus de traitement elle produit une erreur, ce qui peut être vu comme le passage du système dans un état non voulu. Cette erreur peut ensuite se propager dans le système et provoquer d'autres erreurs qui peuvent elles-mêmes se propager. Quand une erreur provoque une déviation du comportement du système par rapport aux fonctions attendues de celui-ci, on parle de défaillance du système. Selon le niveau d'observation du système, une déviation peut être vue comme une faute à un autre niveau. Par exemple, la défaillance d'un composant interne d'un système est vue comme une faute dormante par le système tant que celui-ci ne sollicite pas le composant.

Les moyens

Les moyens sont les techniques permettant de développer un système selon les attributs et les entraves considérés afin qu'il puisse être qualifié comme sûr de fonctionnement. Les moyens se classent en quatre catégories qui vont être utilisées dans les différentes phases de vie d'un système :

— *La prévision des fautes*, regroupe les techniques permettant d'évaluer le comportement du système par rapport à l'occurrence de fautes ou de quantifier son niveau de sûreté de fonctionnement. Ces techniques sont donc soit utilisées au début de la phase de conception d'un système afin de déterminer les techniques à appliquer pour se prémunir des fautes, soit à la fin de la phase de conception pour la quantification de son niveau de sûreté de fonctionnement.

— *La prévention des fautes* qui regroupe les techniques visant à empêcher l'introduction de fautes pendant la conception d'un système. Ces techniques sont princi-

palement basées sur l'utilisation d'un environnement de développement (pour le matériel ou le logiciel) qui détecte les comportements suspects.

— *L'élimination des fautes* qui regroupe les techniques de vérification utilisées à chaque étapes de la conception d'un système afin de corriger les fautes qui ont été diagnostiquées. Parmi ces techniques ont retrouve les méthodes de tests ou de vérifications formelles.

— *La tolérance aux fautes* qui regroupe les techniques permettant de concevoir un système qui remplit sa fonction malgré l'occurrence de fautes. Ces techniques consistent généralement à introduire dans le système des mécanismes permettant de détecter en ligne l'occurrence d'erreurs et de rétablir le système en le forçant dans un état exempt d'erreurs. Ces mécanismes seront présentés plus en détail dans la prochaine section.

1.2.2 Classification et origines des fautes

Classification des fautes

Une faute peut être classée selon sa cause (physique ou humaine), sa nature (accidentelle ou intentionnelle), sa phase de création (développement ou opérationnelle), sa source (interne ou externe) et sa persistance (permanente ou temporaire). Les combinaisons pertinentes de ces différents paramètres conduisent à la différenciation de 31 classes de fautes [2].

Ces classes de fautes peuvent être regroupées en trois groupes :

— *Les fautes de développement* qui sont introduites pendant la phase de développement d'un système au niveau matériel et logiciel. On peut par exemple citer les fautes de conception du circuit, les erreurs de codage ou encore des problèmes lors de la compilation du binaire. Les approches comme le test ou la vérification des modèles permettent de détecter à chaque étape du cycle de développement ces fautes.

— *Les fautes physiques* qui se produisent au niveau du matériel pendant la phase opérationnelle. Ces fautes, une fois activées, peuvent se propager au niveau du logiciel et provoquer la défaillance du système.

— *Les fautes d'interaction* qui résultent des interactions du système avec son environnement.

Nous nous intéressons plus particulièrement aux fautes physiques, en effet ce sont celles-ci

que l'on cherche à détecter dans ces travaux.

Origines des fautes physiques

Afin de répondre aux besoins de l'industrie qui demande des circuits plus performants et moins énergivores, la taille des transistors et la tension d'alimentation des circuits sont diminuées. Cela a pour conséquence d'augmenter la sensibilité des circuits faces aux radiations. Ces radiations ont trois principales origines [12, 13] :

- Les impuretés présentes dans les matériaux qui composent les boîtiers des circuits intégrés. Dans ce cas des radiations peuvent être émises sous la forme de particules alpha quand le noyau d'un isotope instable se désintègre.
- L'impact direct de neutrons à haute énergie provenant des radiations cosmiques.
- L'interaction entre des neutrons issues des radiations cosmiques et l'élément chimique Bore 10 qui est utilisé comme dopant de type P, qui peuvent produire des noyaux de lithium ou une particule alpha, pouvant tous deux provoquer une faute physique.

En plus des phénomènes de radiation, il faut aussi prendre en compte la sensibilité des systèmes face aux interférences électromagnétiques et aux perturbations sur l'alimentation (*power glitch*).

Les erreurs issues des fautes physiques

L'impact des phénomènes présentés précédemment sera différent selon sa localisation sur le circuit (combinatoire ou cellule mémoire) et la technologie du circuit utilisé. Par exemple les cellules mémoires basées sur de la technologie FLASH sont beaucoup moins sensibles face à ces phénomènes que de la mémoire vive (RAM).

Les effets causés par ces phénomènes sont généralement appelés *SEE* (pour *single event effect*). Ils peuvent provoquer des erreurs permanentes ou transitoires :

- *Une erreur permanente* est due à la destruction partielle du circuit. Elle peut prendre la forme de *SEL* (*single event latchup*), *SEB* (*single event burnout*), *SEGR* (*single event gate rupture*), etc.
- *Une erreur transitoire* se caractérise par le basculement de l'état d'une cellule mémoire (*SEU*, pour *single event upset*) ou par la propagation d'une impulsion dans un signal (*SET*, pour *single event transient*). Ces erreurs disparaissent d'elles-mêmes au niveau du circuit mais peuvent se propager au niveau du logiciel et engendrer d'autres erreurs.

La propagation d'un SEU se fait lors de la lecture par le logiciel de la cellule mémoire impactée. Donc si un SEU est impact une cellule mémoire qui n'est jamais lue ou qui est écrite avant d'être lue, l'erreur ne se propagera pas.
Pour qu'un SET engendre une erreur, il faut que celui-ci se propage au niveau de l'entrée d'une porte logique qui a une influence à cet instant, ou se propage à l'entrée d'une cellule mémoire au moment de la mémorisation.
Aussi, un SEE peut impacter plusieurs cellules mémoires en même temps, on parle alors de *MBU* (*multiple bit upset*) quand les cellules mémoires impactées se trouvent sur un même mot mémoire, ou de *MCU* (*multiple cell upset*) quand les cellules mémoires impactées se trouvent sur des mots mémoires différents.

1.2.3 Les aspects de la sûreté de fonctionnement abordés dans nos travaux

Les systèmes qui nous intéressent, sont souvent critiques et nécessitent un niveau de fiabilité élevé. Les attributs de la sûreté de fonctionnement qui les caractérisent sont la fiabilité et la sécurité-innocuité.

Les fautes que l'on considère dans nos travaux sont les fautes transitoires qui peuvent provoquer des erreurs logicielles. Pour ce prémunir de ces fautes deux méthodes sont possibles :

1. L'utilisation de composants durcis qui permettent de se prémunir des SEEs.
2. L'utilisation de mécanismes de tolérance aux « fautes » afin de détecter les erreurs issues des SEEs.

La première méthode étant trop coûteuse, nous aborderons dans cette thèse le développement d'un mécanisme de tolérance aux fautes.

1.3 Les mécanismes de tolérance aux fautes

1.3.1 Architecture d'un mécanisme de tolérance aux fautes

Comme présenté sur la Figure 1.6, un mécanisme de tolérance aux fautes se compose de deux composants :

— *Un mécanisme de détection* dont le rôle est de détecter quand une faute ou une erreur apparaît sur le système au plus tôt, c'est-à-dire avant que celle-ci ne se

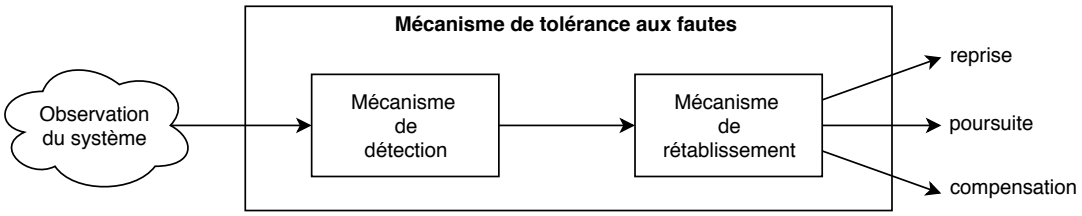

FIGURE 1.6 – Architecture d'un mécanisme de tolérance aux fautes

propage.

— *Un mécanisme de rétablissement* qui, quand une erreur est détectée par un mécanisme de détection, rétablit le système afin qu'il soit toujours opérationnel. Ce mécanisme peut être implémenté selon trois approches possibles :
  — *La reprise* qui consiste à sauvegarder périodiquement l'état du système et de le ramener à l'état de la dernière sauvegarde lors d'une détection d'erreur.
  — *La poursuite* qui consiste à continuer l'exécution du système en le forçant à poursuivre dans un mode dégradé.
  — *La compensation* qui consiste à utiliser la redondance du système quand elle existe pour masquer les erreurs.

La mise en œuvre d'une approche de rétablissement est particulièrement dépendante de l'application visée. Ainsi dans la suite nous nous focaliserons sur les mécanismes de détection.

1.3.2 Panorama des mécanismes de détection

Un mécanisme de détection peut être implémenté dans le système soit au niveau matériel soit au niveau logiciel. Les deux approches sont caractérisées par leur efficacité et leur coût. L'efficacité s'exprime par classes de fautes couvert par le mécanisme et par la latence de détection. Le coût désigne les ressources (surface du silicium, empreinte mémoire, surcoût temporel) utilisés pour l'implémentation du dispositif. Ces mécanismes sont soit basés sur de la redondance, soit sur du contrôle de cohérence.

1.3.3 Mécanismes basés sur la redondance

La duplication

La duplication est l'un des moyens de détection les plus utilisés en raison de sa simplicité. Le principe est de réaliser plusieurs fois la même fonction et de comparer les résultats.

La duplication peut se faire de façon spatiale ou temporelle, et peut être appliquée aux différents niveaux d'un système.

L'approche spatiale consiste à faire réaliser un même calcul simultanément par plusieurs sous-systèmes. Ces sous-systèmes peuvent être identiques si l'on cherche à détecter des erreurs physiques internes, ou différentes si l'objectif est de détecter des fautes de conception. Comme les calculs sont effectués en parallèle, le surcoût temporel est généralement faible. Par contre, en ce qui concerne l'utilisation des ressources, celle-ci est au minimum doublé.

L'approche temporelle consiste à faire exécuter plusieurs fois la même opération par un composant (matériel ou logiciel) et de comparer ensuite le résultat de ces différentes exécutions. Cette approche ne nécessite aucune ressource matérielle supplémentaire (sauf pour le comparateur) mais ajoute un surcoût temporel qui est au minimum doublé. Aussi, cette approche ne permet pas de détecter des changements d'état de la mémoire qui ont eu lieu avant l'exécution de l'opération.

La redondance d'information

La redondance d'information consiste à dupliquer une information ou à lui ajouter des bits de contrôles qui sont calculés par un code détecteur d'erreurs voir un code correcteur d'erreurs. Les approches basées sur la redondance d'information permettent de détecter les erreurs dues à une faute pendant la transmission d'une information ou d'une faute physique au niveau de la mémoire stockant cette information. L'impact de cette approche est l'augmentation de l'empreinte mémoire ainsi qu'une augmentation du surcoût temporel dû au temps de la comparaison et des accès mémoires.

1.3.4 Les mécanismes basés sur du contrôle de cohérence

Le contrôle de cohérence regroupe les mécanismes permettant de vérifier que l'exécution du système est cohérente vis-à-vis d'un certain nombre de spécifications. Ces méthodes permettent généralement de détecter un large spectre de fautes avec un faible coût. Par contre le taux de couverture est généralement faible en comparaison des méthodes précédemment présentées. On peut distinguer 4 types de contrôle de cohérence qui sont : le contrôle de données, le contrôle temporel, le contrôle d'exécution le contrôle de vraisemblance.

Le contrôle de données

Cela consiste à ajouter du code spécifique dans un programme afin de vérifier l'intégrité des données. Cela peut par exemple être utilisé pour vérifier qu'une valeur est bien comprise dans un intervalle de valeurs possibles ou encore que l'écart entre la valeur courante et précédente d'une donnée est cohérente.

Le contrôle temporel

Cela permet d'assurer que les différents composants matériels et logiciels d'un système sont toujours en vie. Cela passe par l'utilisation de *watchdog* qui sont des *timers* devant être rafraîchis soit avant le dépassement d'une échéance, soit à une fréquence fixe. Si le temps est dépassé, un signal d'erreur est généré, celui-ci conduit généralement au redémarrage du système.

Contrôle d'exécution

Le contrôle d'exécution permet de vérifier le flot de contrôle de programmes exécutés, c'est-à-dire que les instructions du programme sont lues correctement et dans le bon ordre. Le contrôle d'exécution peut se faire au niveau des instructions machines ou à des niveaux d'abstractions plus élevés comme au niveau des blocs d'instruction, des fonctions, ou encore en faisant évoluer un modèle abstrait du système.

Contrôle de vraisemblance

Le contrôle de vraisemblance consiste à vérifier la cohérence des différents éléments d'un programme par l'utilisation de matériels ou logiciels spécifiques.

Les exceptions sont des interruptions générées par la détection de conditions exceptionnelles (erreurs) pendant l'exécution d'un programme par des mécanismes matériels présents dans les processeurs. Ces conditions exceptionnelles peuvent porter sur des erreurs au niveau de l'utilisation des bus (accès à une région mémoire invalide, transfert d'une donnée dont la taille n'est pas supportée par le périphérique ciblé, etc.) ou encore de l'exécution d'une instruction (division par zéro, exécution d'une instruction non-définie, etc.).

Les mécanismes de protections mémoires comme les MPU (*memroy protection unit*) ou MMU (*memory management unit*) sont des composants matériels intégrés aux processeurs qui permettent de définir un certain nombre de régions mémoires et d'assigner à chacune de ces régions des droits d'accès. Quand un processus tente d'accéder à une région à laquelle il n'a normalement pas accès, une exception est générée. Ces mécanismes sont par exemple utilisés lors de l'utilisation d'un système d'exploitation pour isoler les différents processus entre-eux.

1.4 Système d'exploitation temps réel robuste

1.4.1 Le rôle d'un SETR dans la sûreté de fonctionnement

Comme on l'a vu dans la section 1.1, un SETR est un élément essentiel à la conception d'une application temps réel, il est le composant logiciel autour duquel est construit l'application. De ce fait, un SETR est considéré comme un composant central de la *RCB* (*Reliability Computing Base*) [16].

De plus un SETR est aussi considéré comme un composant critique. Son exécution se fait quand le processeur est en mode superviseur. Dans ce mode, toutes les instructions sont autorisées. Il est donc possible d'accéder et modifier n'importe quelle partie de la mémoire ou de registres système, ou encore d'activer ou désactiver n'importe quelle interruption.

Des techniques de sûreté de fonctionnement doivent donc être mises en place afin d'assurer un niveau de fiabilité du système.

1.4.2 Classification des erreurs transitoires dans une application multitâche temps réel

Comme montré sur la Figure 1.7, on classifie les erreurs qui peuvent affecter une application multitâche temps réel en quatre catégories selon leur origine et leur propagation dans l'application.

Une erreur qui survient au niveau de l'exécution d'une tâche peut soit rester confinée dans celle-ci (*catégorie 1*), soit se propager au niveau d'une autre tâche (*catégorie 2*), ou encore se propager au niveau du noyau (*catégorie 3*). La *catégorie 4* regroupe les erreurs qui surviennent au niveau du noyau et qui vont potentiellement se propager au niveau applicatif.

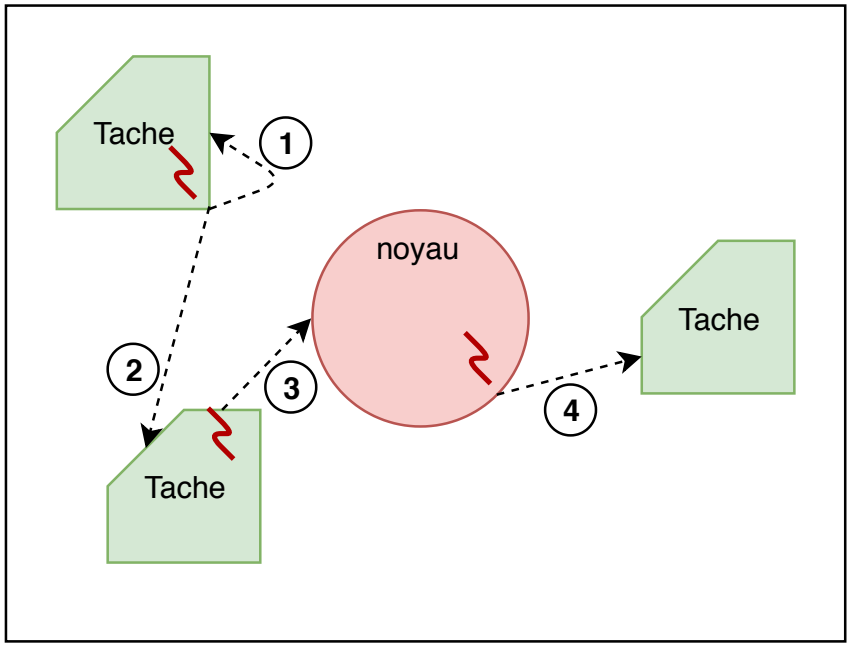

FIGURE 1.7 – Représentation de la propagation d'une erreur transitoire dans une application multitâche temps réel

Il est possible de se prémunir des erreurs des *catégories 2* et *3* en utilisant un mécanisme de protection mémoire pour isoler les différentes tâches de l'application et ainsi éviter la propagation de la majorité des erreurs. Cependant la mise en œuvre de ce type de mécanisme ne permet pas de se prémunir des erreurs qui se propagent via les arguments des appels systèmes.

Pour les erreurs de *catégorie 1*, des techniques basées sur la redondance ou le contrôle de données peuvent être appliqués.

Enfin les erreurs de *catégorie 4* sont considérées comme les plus critiques puisqu'elles peuvent se propager à tout le système. De plus, comme l'exécution du noyau doit être la plus rapide possible, les techniques à mettre en œuvre doivent avoir un impact temporel le plus faible possible tout en détectant le plus d'erreurs possible. C'est ce type d'erreur que l'on cherche à détecter dans nos travaux.

1.4.3 Exemple de SETR robuste

Le concept de "système d'exploitation robuste" est introduit par Rodriguez *et al.* dans [48]. Ils proposent de connecter à un système d'exploitation temps réel un mécanisme qui vérifie, lors du retour d'un appel de service, que les décisions prises par le noyau sont cohérentes. Ce mécanisme est directement généré à partir de formules logiques exprimées dans une logique temporelle dédiée qui spécifient le comportement attendu vu depuis l'application. Ce type de mécanisme est basé sur la vérification en ligne dont nous détaillerons les principes dans le prochain chapitre. Comme les observations sont réalisées au niveau de l'application, les erreurs détectées sont des erreurs de la *catégorie 4* qui se sont propagées au niveau de l'application. Cependant pour faciliter la mise en place de mécanisme de rétablissement du système, il est préférable de pouvoir détecter une erreur aussi tôt que possible, c'est-à-dire avant qu'elle ne se propage dans le système.

La solution que nous proposons est d'intégrer des mécanismes de tolérance aux fautes directement dans le noyau. Pour cela il est possible d'envisager des solutions basées sur de la triplication, mais celles-ci à cause de leur coût ne peuvent pas être considérées comme des solutions universelles.

dOSEK

Le SETR *dOSEK* [23, 24] est un système d'exploitation temps réel conforme au standard industriel AUTOSAR conçu pour détecter les fautes transitoires qui apparaissent pendant l'exécution du code du noyau. Pour cela, deux types de mécanismes sont mis en œuvre.

1. Les mécanismes inhérents au matériel comme le `watchdog` et le MPU qui permet de protéger la propagation des erreurs vers l'espace utilisateur (les tâches).
2. Un encodage arithmétique sur les structures de données du noyau. Celui-ci est basé sur l'AN-code [63] et permet de détecter à la fois les erreurs sur le flot de données et le flot d'exécution.

Leurs résultats montrent qu'en comparaison à un SETR équivalent, le nombre de fautes de type corruption de données silencieuse (`SDC` pour *silent data corruption*) est divisé par 4 mais que le temps d'exécution d'un service peut être multiplié par 0,6 à 6,3. Notons que les `SDC`, dans le cas des SETR se traduisent par une erreur d'ordonnancement.

1.5 Conclusion : Positionnement des travaux de la thèse

Comme on a pu le voir dans ce chapitre, la fiabilité du noyau d'un système d'exploitation temps réel est essentiel à la conception d'un système embarqué temps réel sûr de fonctionnement.

On propose de mettre en œuvre un mécanisme de détection d'erreur permettant de détecter les erreurs de type `SDC` au niveau de l'exécution du code du noyau d'un SETR avant qu'elles ne se propagent dans le système. Les différentes techniques de détection que l'on a présentées ont des coûts, soit en terme de temps d'exécution, soit en terme de ressources matérielles, qui ne sont pas négligeables et pas forcement abordables dans le contexte des systèmes temps réel.

Pour cela, on s'intéressera à la vérification en ligne, une approche permettant de surveiller le comportement de l'exécution d'un système à faible coût.

Chapitre 2

La vérification en ligne

2.1 Introduction à la vérification en ligne

2.1.1 Le principe de fonctionnement

La vérification en ligne [31, 18, 9] est une branche des méthodes formelles dédiée à la synthèse de moniteurs à partir de spécifications. Ces spécifications sont généralement données sous la forme d'un ensemble de propriétés qui portent sur le comportement attendu du système. Elles sont généralement exprimées à partir d'une logique temporelle afin d'être transformées en moniteurs. Ces moniteurs peuvent prendre la forme de composant matériel ou logiciel afin d'être implémentés dans le système.

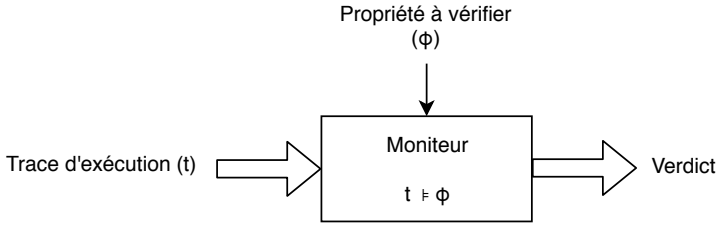

Figure 2.1 – Principe de la vérification en ligne

Comme illustré sur la Figure 2.1, un moniteur prend en entrée la trace d'exécution du système et vérifie qu'elle est conforme à la propriété à vérifier en produisant un verdict.

Cette vérification peut se faire soit après l'exécution du système, soit pendant son exécution. Dans le premier cas, la trace d'exécution est stockée afin d'être transférée à un moniteur *hors-ligne*. Dans le second cas, le moniteur est intégré au système à vérifier, on parle de moniteur *en-ligne*, et vérifie la conformité de la trace au fur et à mesure de sa capture.

Dans le domaine de la vérification en ligne, une trace d'exécution [44] représente la séquence des événements capturés pendant l'observation d'un système. Cette trace est forcément finie : elle débute avec l'observation du système et se termine à l'instant courant

de l'observation. Le moniteur tient à jour un état abstrait du système, souvent représenté sous la forme d'une évaluation d'un ensemble de propositions atomiques. L'observation d'une occurrence d'un événement conduit le moniteur à mettre à jour la valeur d'une ou plusieurs de ces propositions, cela entraîne le changement d'état du système. Dans cette approche, la notion de temps dans une trace est qualitative, c'est-à-dire que l'on connaît l'ordre d'occurrence des événements, mais il est possible d'ajouter une notion de temps quantitatif en datant les différents événements afin de connaître la distance temporelle entre-eux. L'utilisation ou non de la notion de temps quantitatif dépend du langage de spécification utilisé.

2.1.2 Positionnement de la vérification en ligne

La vérification en ligne peut être utilisée comme :

— une technique de vérification pour assurer qu'un système respecte ses spécifications ;

— un mécanisme de détection d'erreurs pour améliorer la fiabilité du système.

En tant que technique de vérification

Les principales techniques de vérification formelle sont la vérification de modèle (*model checking*), le test, et la preuve. La vérification en ligne est une approche complémentaire dont le principe de fonctionnement peut être vue comme une combinaison des deux premières [31].

Le principe de la vérification de modèle [6] repose sur une exploration exhaustive de l'espace d'état du système qui est représenté sous la forme d'un système de transition, afin de vérifier qu'une propriété, exprimée sous la forme d'une formule de logique temporelle, est respectée. Plusieurs algorithmes existent, qui dépendent entre autres, du type de modèle et de la logique considérée.

La vérification de modèle est, comme la vérification en ligne, une technique de vérification formelle. Le principe de cette technique est de vérifier qu'un modèle M d'un système satisfait une propriété ϕ ce qui est noté $M \vDash \phi$. Pour appliquer cette technique, il est nécessaire de (1) modéliser le système à vérifier sous la forme d'un modèle à état, (2) exprimer les propriétés à vérifier sous la forme de formules de logique temporelle et (3) utiliser un algorithme qui à partir du modèle du système et des formules à vérifier, donne un verdict sur celles-ci. Les différences entre la vérification de modèle et la vérification en

ligne sont principalement :

— Dans la vérification de modèle, toutes les exécutions possibles du système sont vérifiées, c'est donc une méthode exhaustive, contrairement à la vérification en ligne où seule l'exécution courante est vérifiée.
— En vérification de modèle on peut traiter des propriétés sur des comportements « infinis ».
— La vérification de modèle est limitée par la précision de l'abstraction du système qui est utilisée pour décrire le modèle. En effet plus le modèle est précis, plus le nombre d'état qui le compose est grand, ce qui augmente le temps de la vérification. En vérification en ligne on utilise le système réel donc tous les aspects du système sont pris en compte.

Le test [1] est une technique de vérification dont le rôle est d'identifier les comportements qui ne respectent pas le cahier des charges d'un système. Selon les objectifs visés (fonctionnalité, performance, etc), des scénarios de test sont conçus afin de placer le système dans un état spécifique pour vérifier que le comportement observé est bien celui attendu. Cela peut être fait, par exemple, en soumettant au système un jeu d'entrées choisi et en comparant le résultat obtenu avec celui attendu. La vérification en ligne peut être vue comme un test dans le sens où il est possible de faire une analogie dans leur fonctionnement :

— la trace d'exécution correspond au scénario de test,
— la propriété à vérifier correspond à un objectif de test,
— le verdict correspond au résultat du test.

De plus, ces deux techniques permettent de vérifier le système à partir du système réel.

Le test et la vérification de modèle sont deux techniques de vérification qui sont utilisées pour détecter les erreurs introduites pendant la phase de conception du système mais qui ne prennent pas en compte les erreurs introduites pendant la phase opérationnelle contrairement à la vérification en ligne.

L'utilisation de la vérification en ligne comme technique de test peut s'envisager de deux façons :

— Soit par l'utilisation de moniteurs « hors-lignes » [60]. Dans ce cas, des sondes (matérielles ou logicielles) sont intégrées au système à surveiller afin de capturer les événements pendant son exécution. Ces événements sont stockés dans une base de donnée afin d'être traités ultérieurement par les moniteurs.
— Soit par l'implémentation de moniteurs sur le système sous test qui seront retirés

sur la version finale du produit.

En tant que mécanisme de détection d'erreurs

L'utilisation de la vérification en ligne pour la détection d'erreurs implique que les moniteurs soient implémentés sur le système. Le principe est que lorsqu'une erreur est détectée par un moniteur, un signal est envoyé vers le mécanisme de recouvrement pour l'informer de la détection d'une erreur. La stratégie de recouvrement à appliquer peut dépendre de la propriété qui est violée.

De façon générale une propriété à vérifier peut être classé en deux types :

Les propriétés de sécurité qui permettent de vérifier que « quelque chose de mauvais » ne se produira jamais. Pour illustrer cela, prenons l'exemple d'un régulateur de vitesse. Une propriété de sécurité pourrait être :

Le régulateur de vitesse ne doit pas être en marche si le conducteur a appuyé sur frein.

Les propriétés de vivacité (ou de garantie) qui permettent de vérifier que "quelque chose de bien" finira par arriver. Par exemple :

Si le conducteur appuie sur le frein, le régulateur de vitesse sera désactivé.

Pour un mécanisme de détection d'erreurs, les propriétés à vérifier sont généralement des propriétés de sécurité.

L'implémentation d'un dispositif de vérification en ligne pour la détection d'erreur peut se faire à plusieurs niveaux dans le système. Dans [53] des moniteurs sont implémentés afin de surveiller la cohérence des signaux d'entrée et de sortie d'un système. Dans [40] les moniteurs permettent de surveiller l'activité d'un bus PCI afin de vérifier les communications entres les périphériques connectés au bus. Dans nos travaux, nous nous focaliserons sur la surveillance de l'exécution d'un programme. De ce fait les propriétés porterons sur la vérification du flot de contrôle et sur la cohérence des données d'un programme.

2.2 Langages formels pour la spécification de propriétés

2.2.1 Les principales catégories de langages de spécifications

Pour exprimer une propriété à vérifier, plusieurs langages peuvent être utilisés. Ces langages peuvent être classés en trois catégories [9] qui sont :

Les logiques temporelles linéaires [42] qui permettent d'exprimer des propriétés à vérifier sous la forme de formules logiques. Ces logiques sont des logiques modales dont les modalités donnent des informations sur l'écoulement du temps.

Les expressions régulières [54] sont des expressions composées de propositions atomiques et d'opérateurs qui permettent de définir les mots, acceptés par une spécification. Une expression régulière décrit donc un langage rationnel qui peut être reconnu par un automate à états finis (théorème de Kleene [28]).

Les machines d'état permettent d'exprimer des propriétés qui sont directement vérifiables. En effet la synthèse d'un moniteur à partir d'une formule de logique temporelle [11] ou d'une expression régulière est souvent réalisée sous la forme d'une machine d'état. Cependant il est généralement difficile d'exprimer directement une propriété sous la forme d'une machine d'état.

Pour manipuler les formalismes mathématiques listés ci-dessus, des langages de spécification ont été définis. Dans le cadre qui nous intéresse dans ce manuscrit, on citera par exemple le langage standardisé PSL (*Property Specification Language*) [43, 14, 37], ou encore la logique EAGLE [8, 20] qui incluent les opérateurs des logiques temporelles et des expressions régulières.

2.2.2 Panorama des logiques temporelles linéaires

Une logique temporelle est linéaire quand elle permet d'exprimer des propriétés portant sur des chemins individuels, donc sur une exécution. Les logiques linéaires s'opposent aux logiques arborescentes qui permettent d'exprimer des propriétés sur des arbres d'exécution. Comme dans le domaine de la vérification en ligne, on s'intéresse à l'évaluation de la satisfaction d'une trace d'exécution par rapport à une propriété, seules les logiques linéaires sont considérées.

On retrouve dans la littérature plusieurs logiques linéaires qui, selon leurs caractéristiques, vont avoir un pouvoir d'expressivité différent. Les deux principales caractéristiques de ces logiques sont :

— **Les types de modalités temporelles** qu'elles utilisent. Celles-ci peuvent faire référence au futur comme la logique LTL (*Linear Temporal Logic*) ou au passé comme la logique ptLTL (*past time Linear Temporal Logic*).

— **La notion du temps** qui peut être simplement qualitatif comme pour les logiques précédemment citées, ou quantitatif comme pour les logiques TLTL [11, 10] et MTL [61] qui étendent LTL, et la logique ptMTL [45] qui étend sur ptLTL.

Dans le cadre de nos travaux, on se restreindra à l'utilisation d'une logique temporelle non-temporisée. Comme on le verra dans la suite, les propriétés que l'on cherche à exprimer porteront sur l'exécution du code d'un SETR où il n'y a pas de notion de temps quantitatif.

2.2.3 Comparaison des logiques LTL et ptLTL

Comme toute logique modale, LTL et ptLTL sont construites à partir d'un ensemble fini de propositions atomiques (noté AP), des opérateurs de la logique propositionnelle (non : $\neg$, et : $\wedge$, ou : $\vee$, implique : $\rightarrow$) et d'opérateurs temporels.

La logique LTL

Une formule LTL φ est définie syntaxiquement de la façon suivant :

$$\varphi := p \in AP \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi_1 U \varphi_2 \mid$$

Où p désigne une proposition atomique et φ, φ_1 et φ_2 des formules LTL. La signification des opérateurs temporels est présentée ci-dessous :

$\bigcirc F$	Suivant	F est vraie à l'instant suivant
$\Box F$	Toujours	F est toujours vraie
$\Diamond F$	Fatalement	F sera vraie
$F_1 U F_2$	Jusqu'à	F_1 est vraie jusqu'à ce que F_2 devienne vraie

La satisfaction d'une formule LTL est évaluée sur l'ensemble $\mathbb{B}_2 = \{\top, \bot\}$ (où $\top$ signifie que la formule est vérifiée et $\bot$ signifie que la formule n'est pas vérifiée) et à partir d'une trace qui est constituée d'une séquence infinie de propositions atomiques. Notons $\sigma = s_0 s_1 \ldots s_i \ldots$ une trace où s_i correspondant à l'état des propositions atomiques à l'instant

i, $\sigma(i) = s_i$ et $\sigma_i = s_i \ldots$ (suffixe de σ). En considérant φ, φ_1 et φ_2 des formules LTL définies sur l'ensemble de propositions atomiques AP, alors la relation de satisfaction $\sigma_i \vDash \varphi$ est définie comme suit :

$$
\begin{array}{lll}
\sigma_i \vDash p & \text{ssi} & p \in \sigma(i) \\
\sigma_i \vDash \neg\varphi & \text{ssi} & \sigma_i \nvDash \varphi \\
\sigma_i \vDash \varphi_1 \vee \varphi_2 & \text{ssi} & \sigma_i \vDash \varphi_1 \text{ ou } \sigma_i \vDash \varphi_2 \\
\sigma_i \vDash \bigcirc\varphi & \text{ssi} & \sigma_{i+1} \vDash \varphi \\
\sigma_i \vDash \Diamond\varphi & \text{ssi} & \exists\ j \geqslant i \text{ tel que } \sigma_j \vDash \varphi \\
\sigma_i \vDash \Box\varphi & \text{ssi} & \forall\ j \geqslant i,\ \sigma_j \vDash \varphi \\
\sigma_i \vDash \varphi_1 U \varphi_2 & \text{ssi} & \exists\ j \geqslant i \text{ tel que } \sigma_j \vDash \varphi_2 \text{ et, } \forall\ i \leqslant k < j,\ \sigma_k \vDash \varphi_1
\end{array}
$$

Comme expliqué dans [10] la principale difficulté à l'utilisation de LTL pour la vérification en ligne est que cette logique permet d'exprimer des propriétés sur des traces infinies alors que la vérification en ligne considère uniquement des traces finies. De ce fait, pour certaines formules LTL, il n'est pas possible de décider de leur satisfaction sur une trace finie. Prenons par exemple le cas de la formule $\phi = \Box a$ qui veut dire que *« a sera toujours vrai »*. Tant que a est vraie, on ne peut pas savoir si la formule est satisfaite, par contre dès que a n'est plus vraie, on sait que la formule n'est plus satisfaite. On remarque pour cet exemple qu'il n'est pas possible, sur une trace finie, de prouver que la formule est satisfaite.

Pour adapter la logique LTL à la vérification en ligne, il est proposé dans [10] d'enrichir le domaine de satisfaction avec l'état « ? » qui permet d'indiquer que le résultat de la satisfaction de la formule est *inconcluant*, on peut alors parler de LTL_3. Ainsi selon le type de propriété temporelle qui est vérifiée, la satisfaction d'une formule évoluera comme présentée sur la Figure 2.2.

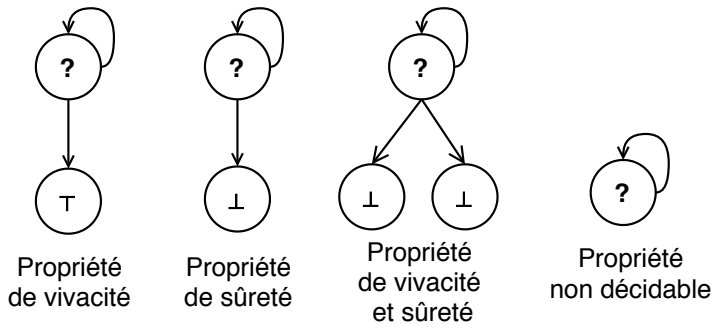

FIGURE 2.2 – Évolution de la satisfaction d'une formule LTL selon LTL_3

Comme on peut le voir, certaines formules LTL ne sont pas décidables, c'est-à-dire qu'elles ne conduisent pas aux verdicts $\top$ ou $\bot$. Les formules qui appartiennent à cette catégorie sont souvent celles qui portent sur des propriétés qui ne sont pas bornées dans le temps. Par exemple, prenons le cas où l'on veut vérifier que l'émission d'une requête est bien suivie par la réception d'une réponse. Si p est une proposition atomique qui est vrai au moment de la requête et q est une proposition atomique qui est vrai au moment de la réponse, alors la formule LTL φ correspondante est :

$$\varphi = \Box(p \rightarrow \Diamond q)$$

La logique ptLTL

De la même façon que pour LTL, une formule ptLTL [22, 21] est définie syntaxiquement de la façon suivante :

$$\varphi := p \in AP \mid \neg\varphi \mid \varphi \vee \varphi \mid \odot\varphi \mid \boxdot\varphi \mid \diamondsuit\varphi \mid \varphi_1 S_s \varphi_2 \mid$$

La signification des opérateurs temporels de ptLTL est présentée ci-dessous :

$\odot F$	(précédemment F)	est vraie si F était vraie à l'instant précédent
$\boxdot F$	(toujours F)	est vraie si F a toujours été vraie dans le passé
$\diamondsuit F$	(fatalement dans le passé F)	est vraie si F a été vraie au moins une fois dans le passé
$F_1 S_s F_2$	(F_1 depuis F_2)	est vraie si F_2 a été vraie au moins une fois dans le passé et F_1 a été vraie depuis
$F_1 S_w F_2$	(F_1 depuis F_2)	est vraie si F_1 a toujours été vraie ou si F_2 a été vraie au moins une fois dans le passé et F_1 a été vraie depuis

En plus, des opérateurs de *monitoring*, présentés dans la Table 2.1, peuvent être utilisés.

Comme montré sur la Table 2.2, ces opérateurs n'ajoutent pas d'expressivité à la logique ptLTL dans le sens où ils peuvent être exprimés par l'utilisation des opérateurs ptLTL classiques.

La sémantique d'une formule ptLTL est définie sur une trace qui est constituée d'une séquence finie de proposition atomique. Notons $\sigma = s_0 s_1 \dots s_n$ une trace où s_n correspond au dernier état des propositions atomiques, et pour $i \leqslant n$, $\sigma_i = s_0 s_1 \dots s_i$ (préfixe de σ).

$\uparrow F$	est vrai si F était faux à l'instant précédent et est vrai à l'instant présent
$\downarrow F$	est vrai si F était vrai à l'instant précédent et est faux à l'instant présent
$[F_1; F_2)_s$	est vrai à l'instant où F_1 est vrai et est faux dès que F_2 devient vrai
$[F_1; F_2)_w$	est vrai initialement et à l'instant où F_1 est vrai, et est faux dès que F_2 devient vrai

TABLE 2.1 – Les opérateurs ptLTL de *monitoring*

start F :	$\uparrow F = F \wedge \neg \odot F$
end F :	$\downarrow F = \neg F \wedge \odot F$
strongly in $[F_1 F_2]$:	$[F_1; F_2)_s = \neg F_2 \wedge ((\odot \neg F_2) S_s F_1)$
weakly in $[F_1 F_2]$:	$[F_1; F_2)_w = \neg F_2 \wedge ((\odot \neg F_2) S_w F_1)$

TABLE 2.2 – Équivalence des opérateurs de *monitoring* avec les opérateurs ptLTL standards

En considérant φ, φ_1 et φ_2 des formules ptLTL définies sur AP, alors la relation de satisfaction $\sigma_i \vDash \varphi$ est définie comme suit :

$$
\begin{array}{lll}
\sigma_n \vDash p & \text{ssi} & p \in s_n \ , \\
\sigma_n \vDash \neg\varphi & \text{ssi} & \sigma_n \nvDash \varphi \ , \\
\sigma_n \vDash \varphi_1 \wedge \varphi_2 & \text{ssi} & \sigma_n \vDash \varphi_1 \wedge \sigma_n \vDash \varphi_2 \ , \\
\sigma_n \vDash \odot\varphi & \text{ssi} & n > 1 \text{ et } \sigma_{n-1} \vDash \varphi \ , \\
\sigma_n \vDash \diamondsuit\varphi & \text{ssi} & \sigma_n \vDash \varphi \vee \left(n > 1 \wedge \sigma_{n-1} \vDash \diamondsuit\varphi\right) \ , \\
\sigma_n \vDash \boxdot\varphi & \text{ssi} & t \vDash \varphi \wedge \left(n > 1 \rightarrow \sigma_{n-1} \vDash \boxdot\varphi\right) \ , \\
\sigma_n \vDash \varphi_1 S_s \varphi_2 & \text{ssi} & \sigma_n \vDash \varphi_2 \vee \left(n > 1 \wedge \sigma_n \vDash \varphi_1 \wedge \sigma_{n-1} \vDash \varphi_1 S_s \varphi_2\right) \ ,
\end{array}
$$

Comme on peut le voir, la sémantique présentée ci-dessus est récursive, c'est-à-dire que la satisfaction d'une formule ptLTL est déduite uniquement de l'observation de l'état courant du système (σ_n) et de l'état précédent (σ_{n-1}). De ce fait ptLTL se prête particulièrement bien à une implémentation matérielle.

Du point de vue de l'expressivité de propriétés, l'utilisation de ptLTL offre les avantages suivants :

— Il est plus naturel d'exprimer des propriétés pour la vérification en ligne à partir des opérateurs qui font référence au passé qu'avec des opérateurs qui font référence au futur [32].

— Les formules ptLTL sont généralement plus concises que les formules LTL [30], ce qui, comme nous le verrons dans la suite à un impact positif sur la taille des moniteurs engendrés.
— Il est possible de définir des opérateurs (syntaxiques) qui facilitent l'expression des propriétés.

Bilan

Du point du vue de l'expressivité, comme LTL permet d'exprimer des propriétés sur des traces infinies son pouvoir d'expressivité est supérieur à celui de ptLTL. Cependant en ce qui concerne les propriétés « monitorables » :

— sur le plan théorique, le problème est ouvert,
— en pratique, nous n'avons jamais rencontré de formule LTL pouvant être synthétisée sous la forme d'un moniteur qui ne peut pas être exprimée sous la forme d'une formule ptLTL.

2.3 Synthèse de moniteur

On trouve dans l'état de l'art deux types d'approches permettant de synthétiser un moniteur à partir d'une formule de logique temporelle.

2.3.1 La synthèse sous la forme d'une machine d'état

La première approche consiste à engendrer une machine d'état à partir d'une formule de logique temporelle.

Dans [10, 11], les auteurs présentent une procédure permettant d'obtenir une machine de Moore à partir d'une formule LTL. Cette procédure est illustrée sur la Figure 2.3. L'idée est de traiter parallèlement une formule φ et sa négation $\neg\varphi$ afin d'en déduire la machine de Moore correspondante. Pour cela il faut :

1. Transformer la formule en un automate de Büchi non déterministe (NBA).
2. Pour chaque état, on analyse l'accessibilité des composants connexes contenant un état acceptant.
3. En déduire un automate fini non déterministe (NFA).
4. Déterminiser l'automate pour obtenir un automate fini déterministe (DFA).

5. Faire le produit cartésien des deux automates afin d'obtenir une machine de Moore.

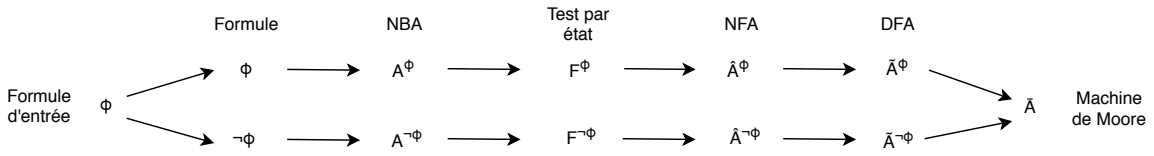

FIGURE 2.3 – Processus de synthèse d'une machine de Moore à partir d'une formule LTL [10, 11]

Les algorithmes réalisant ces différentes étapes introduisent une complexité qui a un impact sur le temps de synthèse des moniteurs, par exemple la complexité de l'algorithme permettant d'obtenir un automate de Büchi non déterministe à partir d'une formule LTL a dans le pire cas une complexité en $O(2^{\phi})$ (où ϕ correspond à la taille de la formule LTL) [38]. Néanmoins comme la synthèse se fait hors ligne, cela n'ajoute pas de contrainte au niveau du système.

La taille des moniteurs engendrés par cette approche peut avoir dans le pire cas une complexité en $O(2^{2^n})$, ce qui est rarement atteint. Cependant il est difficile d'évaluer la taille exacte des moniteurs avant le processus de synthèse.

Le calcul du verdict d'un moniteur se fait rapidement, en effet le verdict dépend directement de l'état courant de la machine d'état.

2.3.2 La synthèse sous la forme d'un circuit

La second approche consiste à engendrer le moniteur sous la forme d'un circuit booléen qui est directement déduit de l'analyse syntaxique de la formule à vérifier. Dans [49, 22] les auteurs présentent des algorithmes permettant de construire un algorithme de satisfaction pour une formule LTL ou ptLTL donnée (l'algorithme correspondant à la formule ptLTL sera présentée dans la suite). De même, dans [36], les auteurs présentent une méthode permettant de synthétiser un moniteur sous la forme d'un circuit logique à partir d'une formule PSL. Notons que pour les logiques LTL et PSL, des contraintes syntaxiques sont imposées au niveau de l'expressivité des formules afin de traiter le problème de non-monitorabilité.

L'idée générale est de définir pour chaque opérateur un circuit booléen puis de connecter les opérateurs selon l'arbre syntaxique de la formule à vérifier et ainsi vérifier chacune des sous-formules.

Pour illustrer cela, prenons l'exemple de la synthèse de moniteurs à partir d'une formule ptLTL. Comme on peut le voir dans la table 2.3, il est possible de réécrire la sémantique de chacun des opérateurs selon que l'on est dans la phase d'initialisation ($n = 0$) ou la phase permanente ($n > 0$).

	Initialisation $n = 0$	**Phase permanente** $n > 0$
$t \vDash \odot F$	$\bot$	ssi $t_{n-1} \vDash F$
$t \vDash \diamondsuit F$	ssi $t \vDash F$	ssi $t \vDash F$ ou $(t_{n-1} \vDash \diamondsuit F)$
$t \vDash \boxdot F$	ssi $t \vDash F$	ssi $t \vDash F$ et ($t_{n-1} \vDash \boxdot F$)
$t \vDash F_1 S_s F_2$	ssi $t \vDash F_2$	ssi $t \vDash F_2$ ou ($t \vDash F_1$ et $t_{n-1} \vDash F_1 S_s F_2$)
$t \vDash F_1 S_w F_2$	ssi $t \vDash F_2$ ou $t \vDash F_1$	ssi $t \vDash F_2$ ou ($t \vDash F_1$ et ($t_{n-1} \vDash F_1 S_s F_2$)
$t \vDash \uparrow F$	$\bot$	ssi $t \vDash F$ et $t_{n-1} \nvDash F$
$t \vDash \downarrow F$	$\bot$	ssi $t \nvDash F$ et $t_{n-1} \vDash F$
$t \vDash [F_1; F_2)_s$	ssi $t \nvDash F_2$ et $t \vDash F_1$	ssi $t \nvDash F_2$ et ($t \vDash F_1$ ou $(t_{n-1} \vDash [F_1; F_2)_s)$)
$t \vDash [F_1; F_2)_w$	ssi $t \nvDash F_2$	ssi $t \nvDash F_2$ et ($t \vDash F_1$ or ($t_{n-1} \vDash [F_1; F_2)_w$))

TABLE 2.3 – Sémantique des opérateurs temporels de ptLTL à l'initialisation et en phase permanente où t correspond à la trace d'exécution et F, F_1 et F_2 sont des formules ptLTL.

Considérons la formule ptLTL utilisée dans l'Annexe A où FOO_{enter}, FOO_{exit} et $X_{>0}$ sont des propositions atomiques :

$$F = \boxdot([FOO_{enter}; FOO_{exit})_s \rightarrow X_{>0})$$

Son arbre syntaxique est représenté sur la Figure 2.4. On définit la profondeur d'un arbre syntaxique comme le nombre maximal de nœud entre une feuille (les propositions atomiques) et la racine (l'opérateur $\boxdot$ dans l'exemple). Dans le cas de notre exemple on peut voir que cette profondeur est de 3.

On peut en déduire la liste des sous-formules à vérifier (où f_0 correspond à la formule initiale) :

$$f_0 = \boxdot([FOO_{enter}; FOO_{exit})_s \rightarrow X_{>0})$$
$$f_1 = [FOO_{enter}; FOO_{exit})_s \rightarrow X_{>0}$$
$$f_2 = [FOO_{enter}; FOO_{exit})_s$$
$$f_3 = FOO_{enter}$$
$$f_4 = FOO_{exit}$$
$$f_5 = X_{>0}$$

Pour évaluer une formule ptLTL, il suffit d'évaluer chacune des sous-formules qui la compose. Le calcul des sous-formules se fait donc « du bas vers le haut » par rapport à

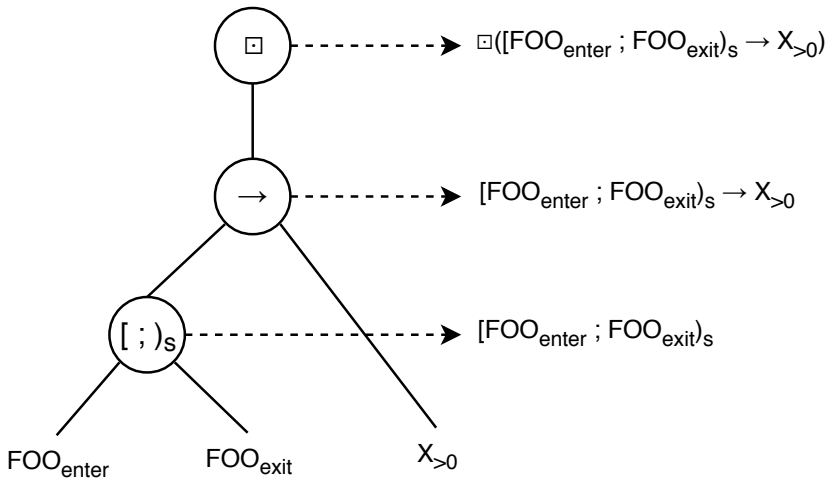

FIGURE 2.4 – Arbre syntaxique de $F = \boxdot([FOO_{enter}; FOO_{exit})_s \rightarrow X_{>0})$

l'arbre syntaxique. On remarque qu'il est possible de calculer en parallèle les formules se trouvant sur des branches différentes de l'arbre syntaxique. Dans notre exemple, les sous-formules f_3 et f_4 doivent être calculées avant la sous-formule f_2, et la sous-formule f_5 peut être calculée en parallèle de celles-ci.

Comme on peut le voir, la taille des moniteurs engendrés par ce type d'approche est linéaire avec la taille de la formule à vérifier. Quant au temps de calcul du verdict d'un moniteur, il est linéaire à la profondeur de la formule à vérifier.

2.4 L'implémentation d'un mécanisme de vérification en ligne

Comme présenté sur la Figure 2.5, les fonctions associées à un mécanisme de vérification en ligne sont :

1. *L'observation du système* par la capture des informations nécessaires à la génération de la trace d'exécution. Cela implique que de pouvoir évaluer à chaque instant l'état des propositions atomiques qui composent les propriétés à vérifier.
2. *La surveillance du système* qui est réalisée par les moniteurs qui vérifient que la trace d'exécution respecte les propriétés à vérifier.

L'implémentation de ces fonctions doit être le moins invasive possible. L'ajout d'un mécanisme de vérification en ligne doit impacter le moins possible les performances du

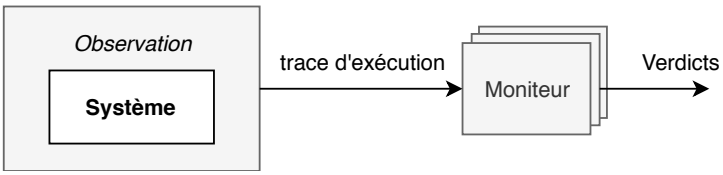

FIGURE 2.5 – Structure d'un mécanisme de vérification en ligne

système à surveiller.

On distingue deux principales approches d'implémentation d'un mécanisme de vérification en ligne [64] :

— L'approche logicielle qui consiste à implémenter les différentes fonctions du mécanisme de vérification en ligne sous la forme de code.
— L'approche matérielle qui consiste à interfacer un dispositif matériel dédié au système.

2.4.1 L'implémentation logicielle

L'implémentation logicielle d'un mécanisme de vérification en ligne est réalisée en modifiant le code source de l'application. L'idée est d'ajouter des instructions dans le code de l'application afin de (1) mettre à jour les valeurs des différentes propositions atomiques et (2) implémenter les moniteurs.

Cette approche présente l'avantage de pouvoir capturer des événements haut niveau, comme l'exécution des fonctions ou la cohérence de variables, permettant ainsi de vérifier des propriétés logicielles.

Cependant, le temps d'exécution de la surveillance, qui doit interférer le moins possible avec l'exécution de l'application, est difficile à maîtriser. Par exemple, pour les mécanismes où la vérification des propriétés se fait à chaque fois qu'une proposition atomique associée au moniteur est mise à jour, le temps de la vérification est proportionnel au nombre de moniteurs qui doivent être activés en réaction à ce changement. Dans [15] les auteurs montrent que dans le pire cas, au-delà de deux moniteurs qui évoluent suite à l'occurrence d'un même événement, le temps d'exécution d'un appel système d'un SETR est doublé. Les auteurs de [41] proposent d'éviter au mieux ces perturbations induites par le mécanisme en faisant la vérification de façon périodique en utilisant un ordonnanceur dédié qui peut être configuré pour ne faire la vérification que pendant les temps inactifs du système.

2.4.2 L'implémentation matérielle

Les premiers travaux concernant l'implémentation matérielle d'un mécanisme de la vérification en ligne pour la surveillance de propriétés portant sur l'exécution du logiciel utilisent une architecture dans laquelle les signaux internes du système sont accessibles [62]. Comme l'observation du système se fait de façon non-intrusive et que la vérification se fait en parallèle de l'exécution, le surcoût de temps induit par le mécanisme est nul. Par contre, l'intégration du composant matériel a un coût en terme de ressources matérielles qui peut ne pas être négligeable et doit être réalisé pendant la phase de conception du système, ce qui n'est pas flexible. De plus, avec l'augmentation de la complexité des architectures matérielles, l'accès aux signaux internes n'est pas forcément possible et l'évaluation des propositions atomiques à partir de ces signaux est difficilement applicable.

Avec l'avènement des processeurs « softcore », qui sont des processeurs synthétisés sur des circuits logiques programmables, plusieurs auteurs proposent d'observer passivement les bus internes des processeurs, en modifiant le design pour ajouter des sondes, afin de surveiller l'exécution du programme :

— P2V [34, 33] est un compilateur qui traduit une spécification exprimée en PSL en un moniteur sous la forme d'un circuit matériel décrit dans le langage de description matériel Verilog. P2V utilise les informations issues de la compilation de l'application afin de générer les circuits permettant d'évaluer les valeurs des propositions atomiques à partir d'informations circulant sur les bus internes.

— De la même façon, dans [47, 46], les bus internes d'un microcontrôleur sont observés passivement afin de calculer le verdict de moniteurs qui sont engendrés à partir de propriétés exprimées en ptLTL afin de vérifier des propriétés sur l'exécution d'un programme.

L'inconvénient des processeurs « softcore » est qu'en terme de puissance de calcul et de consommation d'énergie, ils sont moins efficaces que des processeurs « hardcore » (non synthétisés). Donc dans le contexte industriel ils sont rarement utilisés.

On retrouve de plus en plus des systèmes qui intègrent sur la même puce un processeur « hardcore » et un circuit de logique programmable pour concevoir des systèmes embarqués (Zynq-7000 de Xilinx ou la Smartfusion2 de Microsemi). Ces systèmes offrent la possibilité d'implémenter des moniteurs matériels pour surveiller l'exécution d'un programme sur un processeur "hardcore". Pour l'observation de l'exécution du programme de ces systèmes, il est possible d'utiliser le module de débogage intégré dans les processeurs, comme par exemple le ARM coresight pour les processeurs de type ARM. Dans [35], les auteurs

utilisent ce module pour obtenir des informations bas-niveaux afin de construire la trace d'exécution d'un programme à vérifier. Cependant, le nombre d'événements capturés par le module de débogage est limité. Pour remédier à cela, il est possible de reconfigurer à la volée l'interface de capture des événements [19]. Cela a un coût en terme de ressources logiques qui peut ne pas être négligeable.

2.5 Conclusion : Approche suivie dans la thèse

Dans ces travaux de thèse, on propose d'utiliser une implémentation hybride d'un mécanisme de vérification en ligne : les moniteurs seront implémentés de façon matérielle et l'observation se fera par l'instrumentation du code de l'application à vérifier. L'idée est de tirer parti des avantages des différentes approches. La vérification parallèle des propriétés induite par l'implémentation matérielle des moniteurs permet de maîtriser le surcoût d'exécution temporelle dû à la vérification. L'instrumentation du code offre l'avantage d'avoir accès à des informations de haut niveau.

En ce qui concerne la spécification des propriétés nous utiliserons la logique ptLTL. Cette logique semble particulièrement adaptée à la synthèse de moniteurs matériels et permet de spécifier des propriétés plus naturellement.

Chapitre 3

Implémentation d'un mécanisme de Vérification en Ligne sur un SoPC

3.1 Introduction

La mise en œuvre d'un mécanisme de vérification en ligne est une approche prometteuse pour détecter les erreurs survenant pendant l'exécution d'un programme. Cependant, cela peut avoir une incidence défavorable sur le temps d'exécution du programme à surveiller dans le cas d'une implémentation logicielle ou réduire les performances globales d'un système lors d'une implémentation matérielle. Pour des systèmes embarqués, qui disposent de ressources d'exécution limitées et qui doivent respecter des contraintes temps réel, les approches traditionnelles d'implémentation peuvent ne pas être appropriées.

Dans ce chapitre nous nous intéressons à une implémentation hybride d'un mécanisme de vérification en ligne. L'objectif étant de minimiser les surcoûts induits par le mécanisme. Pour cela, nous proposons d'utiliser un système sur puce de type SoPC (*System-on-Programmable-Chip*) qui intègre un circuit logique programmable de type FPGA (*Field-Programmable Gate Array*). Le programme à vérifier s'exécutera sur le microcontrôleur du SoPC et sera instrumenté de façon à pouvoir générer une « trace d'observation ». La vérification de cette trace se fera à partir d'un circuit matériel synthétisé sur le FPGA.

Dans un premier temps nous nous focaliserons sur la synthèse du composant principal d'un mécanisme de vérification en ligne : le circuit de vérification. Ce composant est constitué d'un ensemble de moniteurs qui sont engendrés à partir des propriétés à vérifier. Celles-ci sont exprimées sous la forme de formule ptLTL (*past-time Linear Temporal Logic*) qui est une logique temporelle qui se prête particulièrement bien à la synthèse de moniteurs matériels.

Dans un second temps, nous présenterons l'architecture du mécanisme de vérification en ligne, qui implémente ce circuit de vérification, sur un SoPC. Pour cela, nous prendrons l'exemple du circuit SmartFusion2 de Microsemi qui est un SoPC conçu pour le

développement de systèmes sûrs de fonctionnement

3.2 Génération d'un circuit logique pour la vérification d'une formule ptLTL

Comme on l'a vu dans le chapitre précédent, il est possible de générer un algorithme permettant de calculer la satisfaction d'une formule ptLTL [22]. Cet algorithme peut être implémenté sous la forme d'un circuit matériel en utilisant un langage de description matériel comme VHDL ou Verilog. Par exemple, dans [47] les auteurs traduisent l'algorithme généré pour une formule ptLTL en un circuit logique avec le langage VHDL en utilisant une description comportementale. Ainsi le code VHDL généré est très proche de celui de l'algorithme.

On propose de générer aussi le circuit logique en VHDL mais en utilisant une description structurelle, c'est-à-dire en utilisant des composants de base que l'on interconnecte les uns aux autres. Les composants de base seront les opérateurs ptLTL et les interconnexions sont réalisées à partir de la structure de l'arbre syntaxique de la décomposition de la formule ptLTL. Les avantages de cette approche sont :

— Il est possible de prouver plus facilement que le circuit est correct en prouvant que chacun des composants est correct par construction [37].
— La mise en œuvre est plus simple et rapide qu'une approche basée sur la génération d'une machine à état.

La génération du circuit est automatisée à partir d'un outil ad hoc.

3.2.1 Architecture du circuit de vérification d'une formule ptLTL

Description VHDL des opérateurs ptLTL

Le circuit logique des opérateurs ptLTL est directement déduit de la sémantique récursive de ptLTL. Par exemple, les circuits logiques correspondant aux opérateurs $\boxdot F$ et $[F1; F2)_s$ sont respectivement représentés sur les Figures 3.1 et 3.2.

Comme on peut le voir dans leur sémantique récursive, les opérateurs ptLTL nécessitent d'avoir la connaissance de l'état précédent de leurs entrées (c'est le cas des opérateurs ptLTL $\odot F$, $\uparrow F$ et $\downarrow F$) ou de leur sortie (c'est le cas des autres opérateurs ptLTL). Cette fonction est réalisée par les bascules D qui sont synchronisées sur un signal d'horloge clk.

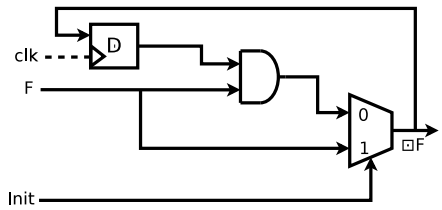

FIGURE 3.1 – Circuit de l'opérateur $\boxdot F$

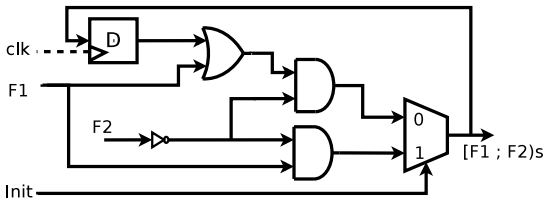

FIGURE 3.2 – Circuit de l'opérateur $[F1; F2)_s$

La sélection du comportement d'un opérateur ptLTL entre sa phase d'initialisation et sa phase permanente se fait à travers un multiplexeur « 2 vers 1 » où le signal de sélection est le signal $Init$. Afin d'initialiser correctement un composant ptLTL, il est nécessaire que ce signal soit à 1 pendant au moins un cycle de l'horloge clk afin d'assurer la cohérence de la valeur mémorisée par la bascule D.

Interconnexion des opérateurs ptLTL

Le nombre de composants à instancier et les connexions à réaliser sont directement données par l'arbre syntaxique de la formule à vérifier. Par exemple, le circuit correspondant à la formule ptLTL $F = \boxdot([FOO_{enter}; FOO_{exit})_s \rightarrow X_{>0})$ est représenté sur la Figure 3.3. Pour simplifier le schéma, on omet le signal d'initialisation $Init$ et le signal d'horloge clk qui sont en entrée de tous les composants correspondant à des opérateurs temporels.

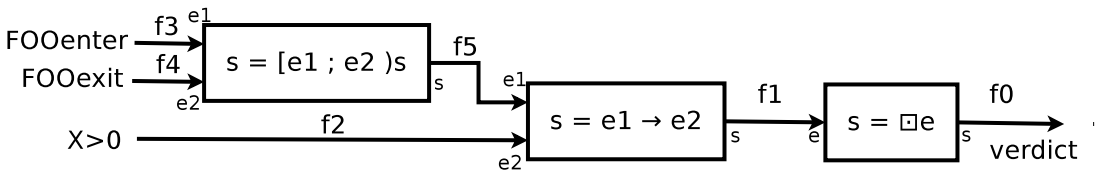

FIGURE 3.3 – Circuit de vérification de F

Toutefois, ce circuit est fonctionnellement correct si l'on suppose que le temps de propagation des signaux dans le circuit est nul, ce qui n'est pas le cas dans la réalité. En effet il faut prendre en compte le temps de propagation entre les différents composants du circuit qui dépend à la fois de la complexité du composant (nombre d'éléments logiques qui le constituent) et le routage de ces éléments logiques les uns par rapport aux autres.

On définit :

— $T_{propagation\_composant}$ comme le pire temps de propagation d'un signal dans un composant.

— $T_{propagation\_circuit}$ comme le pire temps de propagation d'un signal à travers une branche du circuit, ce temps est directement proportionnel à la profondeur de la formule ptLTL.

Dans le cas où $T_{propagation\_circuit}$ est inférieur à un cycle d'horloge ($T_{horloge}$), il faut uniquement assurer la stabilité du signal de sortie. Comme illustré sur la Figure 3.4, cela est réalisé en ajoutant une bascule D en sortie du circuit.

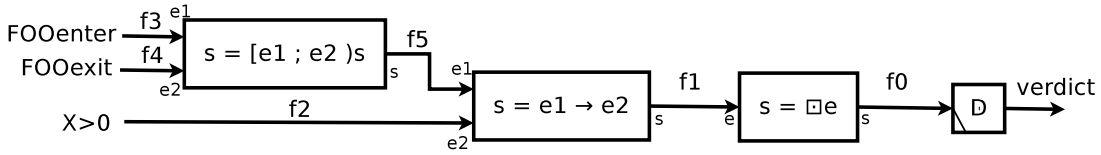

FIGURE 3.4 – Circuit de vérification de F dans le cas où $T_{propagation\_circuit} < T_{horloge}$

Dans le cas où $T_{propagation\_circuit}$ est inférieur à $T_{horloge}$ et que $T_{propagation\_composant}$ est supérieur à $T_{horloge}$, il faut ajouter des étages de pipeline avec des bascules D pour synchroniser les signaux internes de sorte que le temps de propagation entre chaque étage soit inférieur à $T_{horloge}$. Par exemple, la Figure 3.5 illustre le cas où il est nécessaire d'ajouter un étage de pipeline à chaque « niveau de profondeur » de la formule ptLTL.

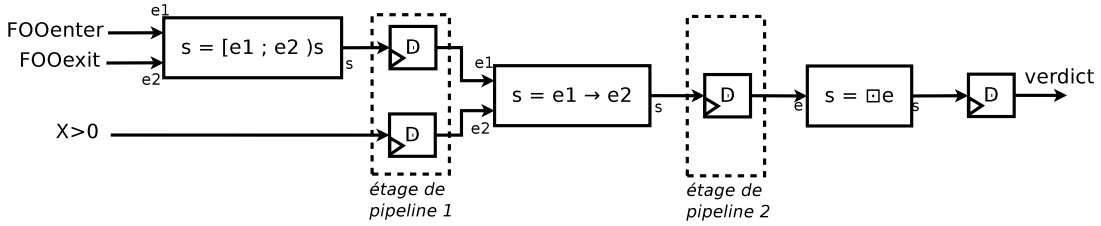

FIGURE 3.5 – Circuit de vérification de F, version avec pipeline dans le cas où $T_{propagation\_circuit} > T_{horloge}$

Enfin dans le cas où $T_{propagation\_composant}$ est inférieur à $T_{horloge}$, il faut diminuer la fréquence du signal d'horloge.

Dans la pratique, toutes les formules que nous sommes amenés à vérifier assurent le respect de la condition $T_{propagation\_circuit} < T_{horloge}$, ainsi le temps de latence de ce circuit, que l'on définit comme le temps écoulé entre la modification d'une ou plusieurs propositions atomiques (les entrées du circuit) et le calcul du verdict correspondant (la sortie du circuit), comme égale à $T_{horloge}$.

Optimisation de l'architecture matérielle du circuit de vérification

Selon la formule ptLTL à vérifier, il est possible que le circuit de vérification possède des bascules D redondantes. Prenons la formule ptLTL $F = \boxdot[F_1; F_2)_s$ qui se traduit littéralement par : « il faut toujours être dans l'intervalle $[F_1; F_2]$ » où le circuit de vérification est représenté sur la Figure 3.6.

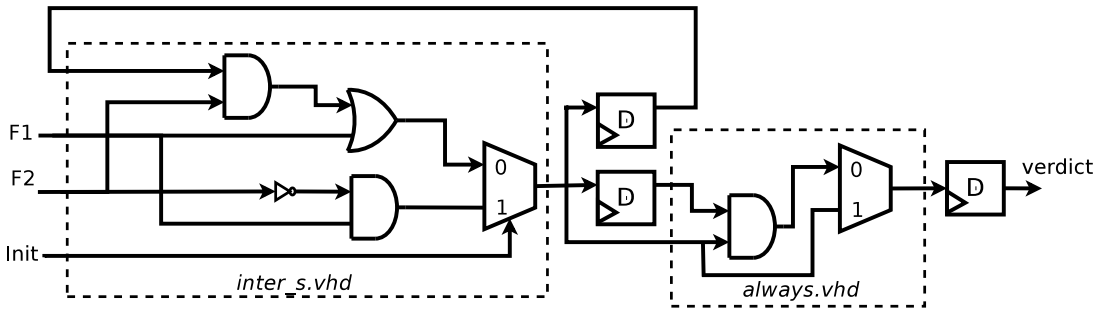

FIGURE 3.6 – Circuit de vérification de la formule $F = \boxdot[F_1; F_2)_s$ sans optimisation

On remarque que deux bascules D sont redondantes. Il est donc possible d'optimiser le circuit en supprimant l'une des deux bascules D.

Cette optimisation est directement identifiée et mise en œuvre par l'outil de génération qui est présenté dans la suite.

3.2.2 Outil de synthèse des circuits de vérification

Présentation de l'outil

La traduction des circuits est réalisée automatiquement par un script Python. La Figure 3.7 décrit le flot de génération de l'architecture matérielle à partir de la liste des formules ptLTL à vérifier.

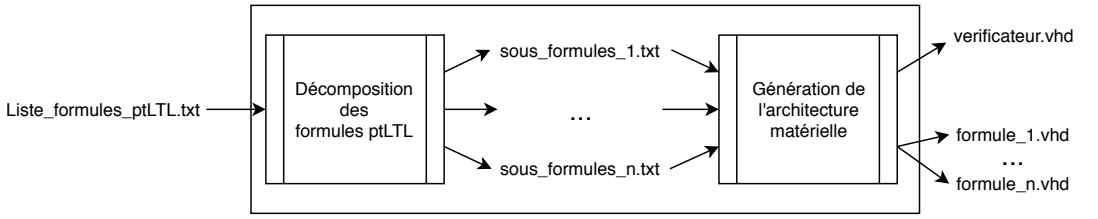

FIGURE 3.7 – Flot de génération du circuit de vérification de formules ptLTL

L'outil prend en entrée un fichier texte qui liste l'ensemble des formules ptLTL afin d'engendrer un ensemble de fichiers VHDL qui contiennent le code de la description

matérielle du circuit de vérification. Le processus de génération est composé de deux étapes.

La première étape est la *décomposition des formules ptLTL* qui consiste à engendrer pour chaque formule ptLTL un fichier $sous\_formule\_x$ (où x correspond à l'identifiant de la formule) listant les sous-formules composant celle-ci. Pour cela, l'outil de génération implémente un algorithme effectuant l'analyse syntaxique d'une formule ptLTL. Par exemple, le fichier engendré par la formule ptLTL $\boxdot([FOO_{enter}; FOO_{exit})_s \rightarrow X_{>0})$ est donné sur la Figure 3.8.

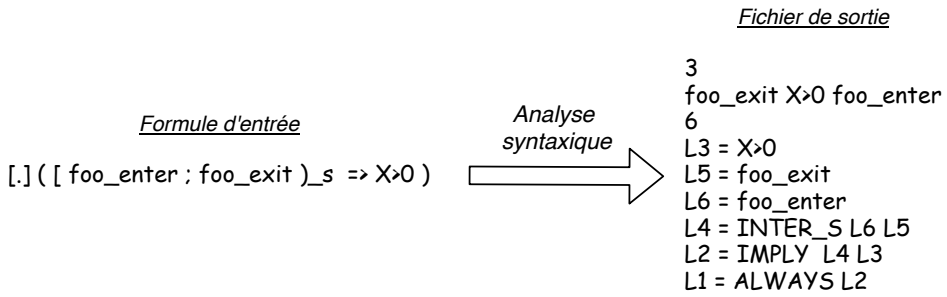

FIGURE 3.8 – Décomposition de la formule $\boxdot([FOO_{enter}; FOO_{exit})_s \rightarrow X_{>0})$

La première ligne indique le nombre de proposition atomique qui composent la formule ptLTL, la seconde ligne liste les propositions atomiques, la troisième ligne indique le nombre de sous-formules et les lignes suivantes décrivent les sous-formules.

La seconde étape consiste à générer l'architecture matérielle du circuit de vérification sous la forme de fichiers VHDL à partir des fichiers de $sous\_formule\_x$.

Hiérarchisation des fichiers de l'architecture matérielle du circuit de vérification

Le langage VHDL permet de décrire une architecture matérielle sous la forme d'un assemblage de composants, on parle alors de description structurelle. Ces composants peuvent eux-mêmes être décrits sous la forme d'un assemblage de composants plus simples ou sous une forme comportementale. La description comportementale consiste à décrire le comportement d'un composant sous la forme d'un algorithme à l'aide d'instructions qui vont s'exécuter de façon concurrente ou séquentielle.

L'architecture du circuit de vérification se compose de composants (un composant par fichier *.vhd*) dont la hiérarchisation est décrite sur la Figure 3.9.

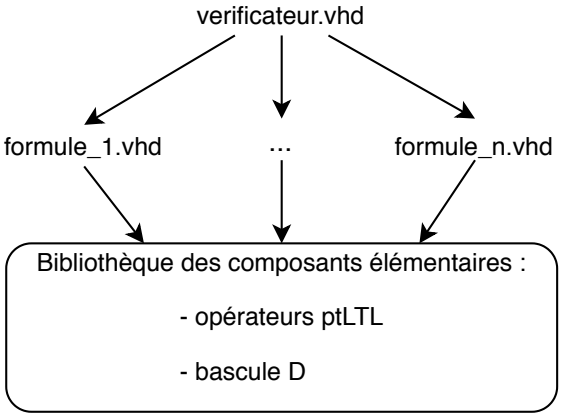

FIGURE 3.9 – Hiérarchisation des fichiers générés

On peut voir que l'architecture du circuit de vérification se compose de trois niveaux :

Le premier niveau : Il correspond au fichier $verificateur.vhd$ qui est engendré par « l'outil de génération » et qui décrit l'architecture générale du circuit de vérification sous la forme du composant *vérificateur* (top level) comme illustré sur la Figure 3.10.

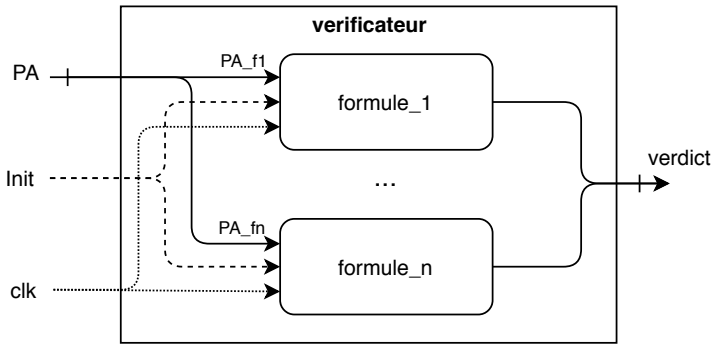

FIGURE 3.10 – Architecture du composant vérificateur

Ce composant prend en entrée l'ensemble des propositions atomiques (PA) utilisées par les formules ptLTL, le signal d'horloge *clk* et le signal d'initialisation $Init$. En sortie les verdicts des formules ptLTL à vérifier sont regroupés dans le bus *verdict*. Il est composé de l'assemblage des composants du deuxième niveau qui correspondent aux circuits de vérification des différentes formules ptLTL. Le fichier $verificateur.vhd$ correspondant à notre exemple est donné sur la Figure 3.11.

```
entity VERIFICATEUR is
port( clk, Init : in std_logic;
      foo_enter, foo_exit, Xsup0 : in std_logic;
      verdicts : out std_logic_vector(0 downto 0) );
end VERIFICATEUR;

architecture ARCHI of VERIFICATEUR is

  component prop1 is
      port(   clk, Init : in std_logic;
              foo_enter, foo_exit, Xsup0 : in std_logic;
              verdict : out std_logic);

  begin

  Instanceprop1 : prop1
      port map (clk => clk, Init => Init,
                foo_enter => foo_enter,
                foo_exit => foo_exit,
                Xsup0 => Xsup0,
                verdict => verdicts(0));

end ARCHI;
```

FIGURE 3.11 – Exemple du fichier *verificateur.vhd*

Le deuxième niveau : Il se compose des fichiers *formule_x.vhd* engendrés par « l'outil de génération ». Chaque fichier décrit un composant correspondant à l'architecture d'un circuit de vérification d'une formule ptLTL. Ces composants sont décrits à partir de l'assemblage de composants élémentaires (troisième niveau) correspondant aux opérateurs ptLTL et au circuit de la bascule D. Pour illustrer ce deuxième niveau, nous allons reprendre la formule ptLTL $\boxdot([FOO_{enter}; FOO_{exit})_s \rightarrow X_{>0})$. La représentation schématique correspondante est donnée sur la Figure 3.12.

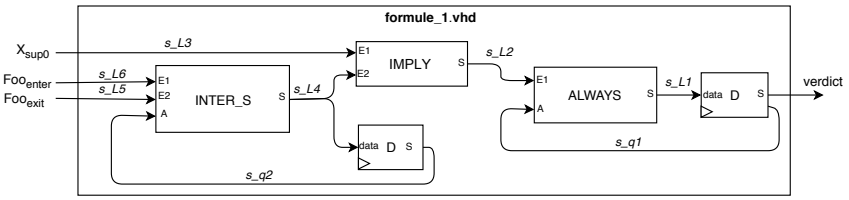

FIGURE 3.12 – Représentation schématique de circuit de vérification de la formule $\boxdot([FOO_{enter}; FOO_{exit})_s \rightarrow X_{>0})$

Le troisième niveau est une bibliothèque de composants décrivant l'architecture de chaque opérateur ptLTL ainsi que de la bascule D. Par exemple les codes VHDL correspondant aux opérateurs *always* et $intervalle_{strong}$ sont donnés sur les Figures 3.13 et

3.14.

```
entity ALWAYS is
port(E1, A, Init : in std_logic;
     S : out std_logic);
end ALWAYS;

architecture ARCH_ALWAYS of ALWAYS is
begin
  process(E1, A, Init)
  begin
    if
Init ='1' then
      S <= E1 ;
    else
      S <= E1 and A ;
    end if;
  end process;
end ARCH_ALWAYS;
```

FIGURE 3.13 – Code VHDL de l'opérateur $\boxdot E1$

```
entity INTER_S is
port(E1, E2, A, Init : in std_logic;
     S : out std_logic );
end INTER_S;

architecture ARCH_INTER_S of INTER_S is
begin
  process (E1, E2, A, Init)
  begin
    if (Init = '1')then
      S <= E1 and not E2;
    else
      S <= (A or E1) and not E2;
    end if;
  end process;
end ARCH_INTER_S;
```

FIGURE 3.14 – Code VHDL de l'opérateur $[E1; E2)_s$

On remarque que l'on ne décrit que la partie combinatoire du circuit des opérateurs ptLTL. Ainsi pour avoir le circuit correspondant à l'opérateur *always* il est nécessaire d'ajouter une bascule D entre le signal $E1$ et A, et pour l'opérateur $intervalle_{strong}$ il est nécessaire d'ajouter une bascule D entre le signal S et A. La séparation entre la partie combinatoire et séquentielle des opérateurs ptLTL est réalisée afin d'optimiser le nombre de bascules D dans le circuit.

3.3 Présentation de l'architecture du mécanisme de vérification en ligne

Dans cette section nous présenterons d'abord l'architecture d'un mécanisme de vérification en ligne implémenté sur un *System-on-Programmable-Chip* (SoPC). Ce type de plateforme est de plus en plus utilisé pour la conception de systèmes embarqués complexes. En effet, cela permet de concevoir des systèmes qui intègrent des accélérateurs matériels dédiés spécifiques sur le FPGA. Nous prendrons comme plateforme d'évaluation le SoC SmartFusion 2 de chez Microsemi car c'est une plateforme industrielle spécialement conçue pour les applications embarquées critiques.

Nous nous intéresserons ensuite à la partie *observation* du programme à vérifier, c'est-à-dire comment faire pour que les informations pertinentes sur l'exécution du programme

soient envoyées sur le FPGA.

Enfin nous nous focaliserons sur l'architecture du dispositif matériel de vérification en ligne (qui est basée sur le circuit de vérification décrit dans la section précédente) et sur les problèmes liés à la synchronisation de notre architecture et à son durcissement.

3.3.1 Architecture générale du système

L'architecture que l'on propose est décrite sur la Figure 3.15.

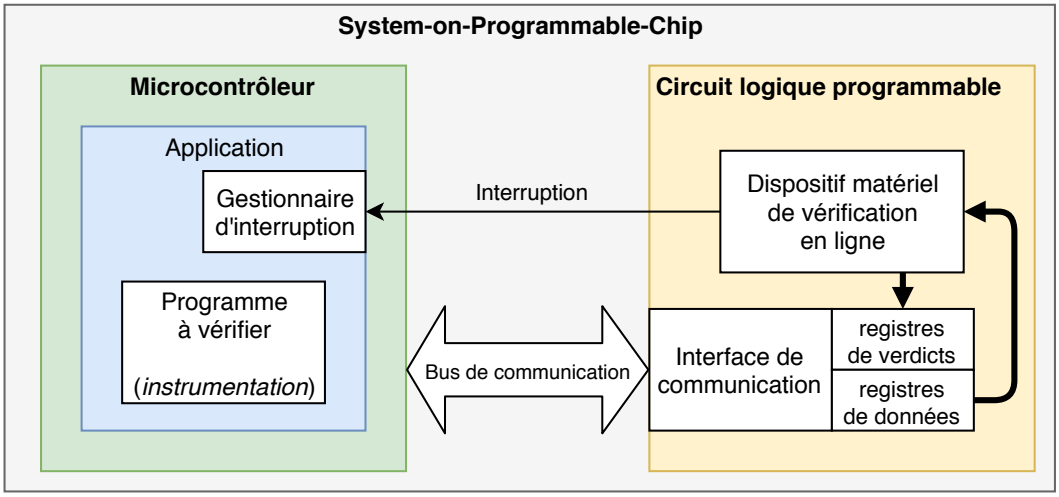

FIGURE 3.15 – Architecture du système

Le principe de fonctionnement de cette architecture est le suivant :

1. Le programme à vérifier s'exécute sur le microcontrôleur. Celui-ci est observé à l'aide d'instructions qui capturent les événements pertinents de l'exécution du programme. Ces instructions sont utilisées pour mettre à jour un ensemble de registres de données implémentées sur le FPGA.
2. Le dispositif matériel de vérification en ligne, implémenté sur le FPGA, utilise les informations contenues dans les registres de données pour calculer le verdict des propriétés à vérifier. Aussi il envoie une interruption vers l'application quand l'une des propriétés est violée.
3. Le gestionnaire d'interruption est responsable de la gestion de l'erreur. Il a accès aux registres de verdict afin d'identifier la propriété qui a été violée et d'appliquer le mécanisme de réaction à l'erreur adéquate. Ce mécanisme de réaction d'erreur

dépend de la politique de sécurité appliquée au système. Cette partie est en dehors du cadre de cette thèse.

Présentation de la Microsemi SmartFusion 2

Le SoC SmartFusion 2 M2S060 (SF2), dont l'architecture est présentée sur la Figure 3.16, est un exemple de SoPC. C'est un SoC qui est conçu pour le développement de systèmes sûrs de fonctionnement, en effet, une partie de ses composants sont immunisés face aux SEU.

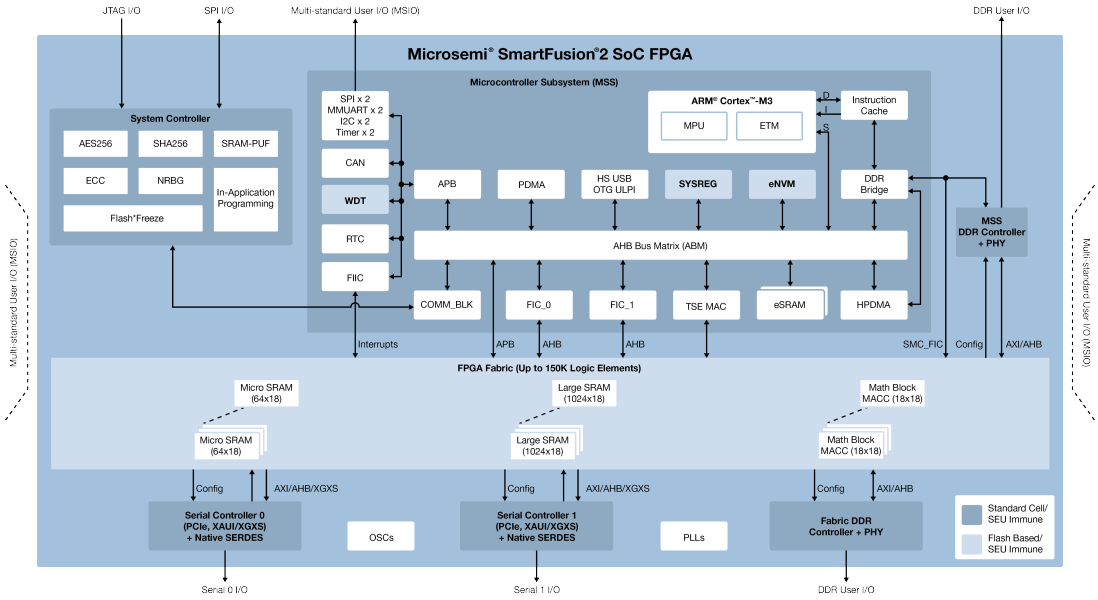

FIGURE 3.16 – Architecture de la SmartFusion2 [58]

Le microcontrôleur est basé sur un cœur Cortex M3 de fréquence maximale 142 MHz. Il dispose de 265 Ko de mémoire de type NVM (*Non-Volatile Memory*) pour le stockage du code et des constantes du programme, et de 64 Ko de mémoire SRAM (*Static Random Access Memory*) pour le stockage des variables du programme et d'un cache d'instruction. La mémoire SRAM intègre un mécanisme de ECC (*Error Correcting Code*) permettant de détecter la corruption de 2 bits d'une donnée et de corriger une corruption de donnée de 1 bit. De plus, le processeur dispose d'une unité de protection de la mémoire (MPU) permettant de protéger les accès à la mémoire.

Le FPGA est constitué de 56520 éléments logiques (4 LUT et 1 bascule D) et est basé sur de la technologie flash. La mémoire de configuration du FPGA est donc protégée contre les SEU.

Au niveau du signal d'horloge, on configure le SoC pour que le microcontrôleur et le FPGA partagent le même signal.

Communication entre le microcontrôleur et le FPGA

Les communications entre le microcontrôleur et le FPGA peuvent se faire par l'intermédiaire de signaux d'interruption, des ports GPIO du microcontrôleur qui peuvent être routés sur les entrées/sorties du FPGA et de l'utilisation d'un bus de communication AMBA (*Advanced Microcontroller Bus Architecture*).

Un signal d'interruption allant du FPGA vers le microcontrôleur est utilisé afin de notifier la détection de la violation de l'une des propriétés à vérifier.

Les ports GPIO sont utilisés par le microcontrôleur pour initialiser le dispositif de vérification et pour l'acquittement de l'interruption.

Le bus de communication AMBA est utilisé pour accéder aux registres implémentés sur le FPGA. On distingue deux types de registres :

— *Les registres de données* qui sont modifiés par le processeur et lus par le dispositif matériel de vérification.

— *Les registres de verdict* qui sont modifiés par le dispositif matériel de vérification et lus par le processeur.

Sur la SF2, les bus spécifiés AMBA que l'on peut utiliser sont :

— AHB *Advanced High-Performance Bus* permet d'effectuer des opérations de hautes performances comme des transferts en rafale (*burst*) ;

— APB *Advanced Peripheral Bus* permet d'effectuer des opérations simples comme l'écriture ou la lecture de registres.

Pour notre architecture, les communications entre le processeur et le FPGA correspondent à la modification de registres, donc l'interface de communication est implémentée selon le protocole APB. Avec ce protocole un transfert prend deux cycles d'horloge.

3.3.2 Observation du programme à vérifier

L'observation du programme à vérifier consiste à capturer la trace d'exécution du système de façon à être capable de calculer l'état des propositions atomiques qui composent les différentes formules à vérifier. Pour cela, il est nécessaire de capturer les transactions mémoires et le flot d'exécution du programme à vérifier. Cependant, sur une plateforme de type SoPC, il n'est pas possible d'observer directement les bus internes du système ou

	SRAM	**FPGA**
Cas d'un accès unique		
en écriture	4 cycles	4 cycles
en lecture	4 cycles	4 cycles
Cas de deux accès consécutifs		
en écriture	5 cycles	5 cycles
en lecture	7 cycles	9 cycles

TABLE 3.1 – Mesure des temps de transfert, en cycle d'horloge du processeur, d'une donnée sur la SRAM et le FPGA

d'accéder aux registres internes de l'architecture, contrairement aux architectures basées sur des processeurs *softcore*.

Nous proposons d'observer le programme en l'instrumentant. Pour cela, nous utilisons les *registres de données* présents sur le FPGA. Chaque registre peut être soit assigné à une variable du programme à vérifier (on parle alors d'une variable monitorée), soit à un ensemble d'événements.

Le mappage d'une variable

Pour mapper une variable du programme à vérifier à l'un des *registres de données*, il faut modifier le code de façon à ce que pendant la phase d'édition de lien, l'adresse allouée à cette variable corresponde à l'adresse d'un registre du FPGA.

Cette approche implique qu'une variable allouée sur le FPGA soit déclarée « volatile » afin d'assurer qu'aucune optimisation de compilation ne lui soit appliquée (*i.e.* toutes les opérations de lecture/écriture sont effectivement réalisées).

Pour évaluer le coût de l'allocation d'une variable sur le FPGA nous avons mesuré et comparé les temps d'accès à une variable selon qu'elle soit allouée sur le FPGA ou sur la SRAM. Comme figuré sur la Table 3.1 les temps d'accès à la SRAM et au FPGA sont identiques sauf dans le cas de deux accès consécutifs en lecture où l'on observe une différence de deux cycles d'horloges.

La génération d'un événement

La génération d'un événement est réalisée par l'ajout d'instructions dans le code du programme à vérifier. Ces instructions sont utilisées afin d'inverser la valeur d'un bit spécifique d'un *registre de données*. Chacun des bits de ces *registres de données* est associé

à un événement particulier. Ainsi le changement d'état d'un bit indique un événement.

Classiquement pour générer un événement (donc inverser un bit d'un registre) il faudrait :

1. Lire la valeur du registre contenant le bit à modifier (*un accès en lecture sur le bus de communication*).
2. Calculer la nouvelle valeur du registre en appliquant l'opération $nouvelle\_valeur = valeur\ xor\ identifiant\_event$.
3. Mettre à jour le *registre de données* avec la nouvelle valeur (*un accès en écriture sur le bus de communication*).

Pour réduire l'impact temporel de cette instrumentation, on propose de transférer l'identifiant de l'événement par le bus de communication et que l'inversion de la valeur du bit soit réalisée par celui-ci en matériel. Ainsi pour générer un événement il est nécessaire d'effectuer un seul accès au bus de communication.

Pour illustrer cela, prenons une fonction *FOO* pour laquelle on veut générer un premier événement quand on entre dans la fonction et un second événement pour quand on en sort. Ces événements sont respectivement associés aux bits 0 et 1 du registre *reg_event*. L'instrumentation de la fonction *FOO* est donc :

```
FOO()
{
    regEvent = 0x00000001;
    ...
    regEvent = 0x00000002;
}
```

La Figure 3.17 illustre l'évolution de l'état des bits 0 et 1 du registre *reg_event* selon l'exécution de la fonction *FOO*.

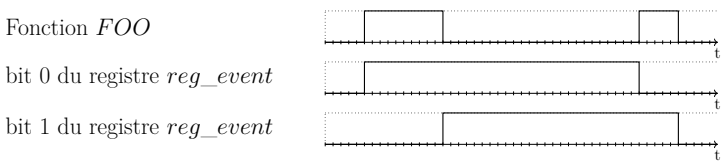

FIGURE 3.17 – Exemple de la génération d'événements

3.3.3 Architecture du dispositif matériel de vérification en ligne

Comme indiqué sur la Figure 3.18 le dispositif matériel de vérification en ligne est composé de trois types de composants.

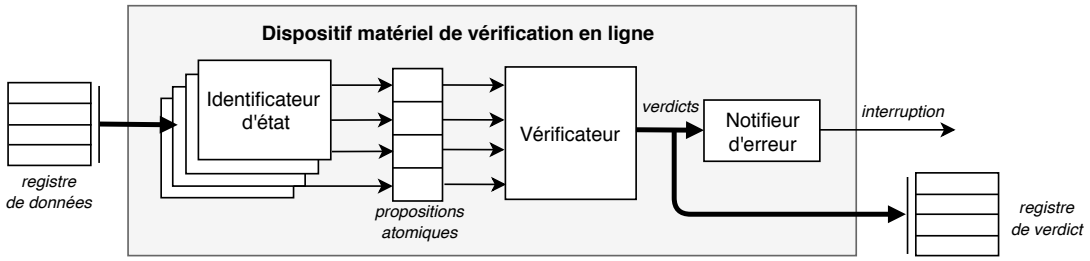

FIGURE 3.18 – Architecture du dispositif matériel de vérification en ligne

Les identificateurs d'état

Le rôle d'un identificateur d'état est d'évaluer l'état d'une proposition atomique à partir des informations contenues dans les registres de données. La conception des identificateurs d'état repose sur la formalisation de propositions atomiques qui peuvent être spécifiées par :

— La comparaison entre deux données où une donnée peut soit correspondre à la valeur d'une variable monitorée, donc allouée dans un des registres de données, soit à une constante. Dans ce cas, la structure de l'identificateur d'état est un circuit combinatoire qui effectue la comparaison entre les deux données.

— L'occurrence d'un événement logiciel qui est notifié par le logiciel par le changement d'un bit d'un registre de donnée qui est associé à l'événement. Comme un événement correspond à un front, il suffit de détecter ce front avec une bascule D (voir Figure 3.19). Le bit associé à l'événement, que l'on note EV, est retardé d'un

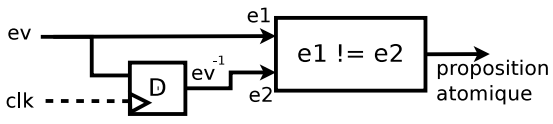

FIGURE 3.19 – Circuit de détection d'un événement (détection de front)

cycle d'horloge (EV^{-1}) par une bascule D. Ensuite, la comparaison entre EV et EV^{-1} permet d'avoir le résultat de l'occurrence d'un événement. La valeur de la

	Moyenne	Min	Max	écart type
4-LUT	2.6	1	10	1.8
bascules D	2.05	1	6	1.1

TABLE 3.2 – Empreinte en ressources logiques des circuits de vérification synthétisés à partir des formules issues de [11]

proposition atomique est donc à 1 pendant un cycle d'horloge lors de l'occurrence de l'événement.

— L'accès à un état spécifique d'un automate fini. Dans ce cas, la structure de l'identificateur d'état est un automate fini où la proposition atomique est définie en fonction de l'état courant de l'automate. Les conditions de déclenchement pour chaque transition sont composées de l'occurrence d'événements logiciels ou la comparaison entre deux données. L'utilisation d'un automate fini permet de spécifier des propositions atomiques plus complexes, ce qui conduit à une simplification de la transcription d'une spécification en une formule logique.

Le vérificateur

Le circuit *vérificateur* évalue l'ensemble des formules ptLTL à partir des propositions atomiques. Les verdicts sont stockés dans les registres de verdicts où un bit est associé au résultat d'une formule ptLTL. L'architecture et la génération de ce circuit sont décrites dans la section précédente.

Pour évaluer le nombre de ressources logiques utilisées en moyenne pour un circuit de vérification, nous avons traduit les 54 formules LTL monitorables issues de l'article [11] en formules ptLTL, puis nous les avons synthétisées sous la forme de circuits de vérification.

L'empreinte en terme de ressources logiques est résumée dans la Table 3.2. Comme on peut le voir la moyenne d'un moniteur est très faible, ce qui est prometteur pour l'évolutivité de l'approche.

Le notifieur d'erreur

La fonction du notifieur d'erreur est de déclencher une interruption dès qu'une erreur est détectée. Une erreur est définie par le passage à 0 de l'un des verdicts et on considère que le signal d'interruption doit être positionné à 1 tant que l'acquittement de l'interruption n'est pas reçu. Ainsi le notifieur d'erreur se compose d'un détecteur de front

descendant et d'un verrou RS comme montré sur la Figure 3.20.

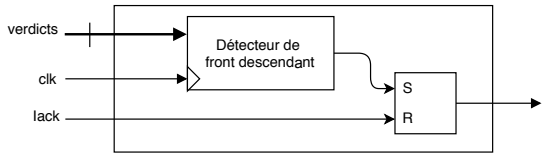

FIGURE 3.20 – Circuit du notifieur d'erreur

Le détecteur de front descendant prend en entrée les signaux des verdicts et génère une impulsion sur sa sortie s'il y a un front descendant sur au moins un des signaux. Quand une impulsion est générée, la sortie de verrouillage du verrou SR (IT) est collée au niveau haut. La sortie reste au niveau haut jusqu'à ce qu'une impulsion soit générée sur le signal d'acquittement IACK. Le signal IACK est généré par le microcontrôleur quand celui-ci à traité l'interruption.

3.3.4 Synchronisation de l'architecture

Entre la génération d'un événement qui conduit à une erreur et la réception du signal d'interruption par le processeur, du temps s'écoule. On le définit comme la latence de détection du mécanisme de vérification que l'on mesure en nombre de cycle d'horloge. Le seul élément de notre architecture pouvant faire varier la latence de détection est le temps de propagation dans le circuit de vérification. Cependant comme nous l'avons vu dans la section précédente, pour les différentes formules que nous sommes amenés à vérifier, ce temps est toujours constant (1 cycle d'horloge). Expérimentalement, en générant un événement qui déclenche une erreur et en comptant le nombre d'instruction *NOP* (1 cycle d'horloge) exécuté avant l'exécution de la routine d'interruption associée, nous avons évalué que ce temps de latence est de 5 cycles d'horloges.

Dans l'architecture présenté jusque-là, le calcul de la satisfaction des verdicts est réalisé en parallèle de l'exécution du programme à vérifier, on est donc dans une approche *non-bloquante*. C'est-à-dire que le programme a exécuté un certain nombre d'instructions avant la détection d'une erreur.

Cependant, pour les systèmes où la fiabilité est plus importante que la performance, il est nécessaire d'assurer qu'aucune instruction n'est exécutée avant l'obtention du résultat du dispositif de vérification en ligne. Il faut donc inclure un mécanisme de synchronisation permettant de bloquer l'exécution du programme à vérifier pendant le calcul des verdicts. On est donc dans une approche *bloquante*.

Ce mécanisme de synchronisation peut être implémenté soit de façon logicielle soit de façon matérielle.

— L'implémentation logicielle consiste à ajouter au niveau du circuit un bit de synchronisation qui est positionné à '1' dès qu'un événement est envoyé vers le dispositif de vérification en ligne et, est positionné à '0' par celui-ci quand la vérification est réalisée. Du point de vue du programme à vérifier il faut ajouter une ligne de code après chaque envoi d'événement qui interroge l'état du bit jusqu'à ce que celui-ci soit positionné à '0'.

— L'implémentation matérielle suppose que le protocole de communication, utilisé entre le microcontrôleur et le FPGA, fournisse un signal permettant d'étendre le transfert. Si c'est le cas il est possible d'utiliser ce signal pour étendre le transfert jusqu'à ce que la vérification soit réalisée. Du point de vue du programme, celui-ci est en attente de signal de fin transfert.

Pour les deux approches, la synchronisation est réalisée en retardant un signal de synchronisation de 5 cycles d'horloges.

3.3.5 Durcissement du mécanisme de détection

La mise en place d'un mécanisme de détection d'erreurs permet d'améliorer la fiabilité du système. Cependant cela est vrai uniquement si ce mécanisme est lui-même considéré comme fiable. Pour cela, il faut que le mécanisme soit robuste face aux fautes qui provoquent les erreurs qu'il doit détecter. Jusqu'ici nous avons considéré des fautes transitoires qui provoquent des erreurs pendant l'exécution d'un programme à vérifier. Mais ces fautes transitoires peuvent aussi se répercuter sur le mécanisme de détection et ainsi provoquer une défaillance de celui-ci. Elles peuvent se traduire par la non-détection d'une erreur logicielle ou par la génération d'une interruption sans raison.

Une faute transitoire peut avoir un impact sur le mécanisme de détection que l'on propose si elle se manifeste au niveau du circuit logique implémenté sur le FPGA ou au niveau des instructions de l'instrumentation du programme à vérifier.

L'instrumentation du programme

Les instructions d'instrumentation ajoutées au programme à vérifier ont pour mission de transmettre un événement sur l'exécution du programme en écrivant une donnée dans un registre implémenté sur le FPGA. Deux types d'erreur peuvent survenir :

— *La corruption de la donnée* à écrire.

— *La corruption de l'adresse du registre* destinataire de la donnée.

Pour se protéger d'une corruption de donnée, on propose de coder la donnée de façon à ce que l'information à transmettre soit tripliquée spatialement dans le registre. Pour rappel, dans le cas où la taille d'un registre est de 32 bits, un événement est assigné à chacun des bits. Donc la valeur contenue dans la donnée à transmettre est de 2^n avec n allant de 0 à 31. Pour tripliquer l'information il suffit de diviser le registre en trois parties de 10 bits et de répliquer l'information dans chacune d'elle. Dans ce cas la valeur contenue dans la donnée à transmettre est de $2^n + 2^{n+10} + 2^{n+20}$ avec n allant de 0 à 9. La détection et la correction d'erreur peuvent ensuite être réalisées au niveau de l'interface de communication implémentée sur le FPGA. Dans ce cas on augmente la fiabilité du système en triplant le nombre de registres nécessaires pour la génération d'événements.

L'adresse d'un registre peut être représenté comme définit sur la Figure 3.21.

FIGURE 3.21 – Représentation de l'adressage d'un registre mémoire sur le circuit logique programmable

Si une faute se manifeste au niveau des bits de destination du périphérique, il n'est pas possible de détecter l'erreur. Si une faute se manifeste au niveau des bits inutilisés, il est possible de corriger l'erreur en masquant ces bits lors de la réception. Si l'erreur se manifeste au niveau des bits d'offset, il est difficile de détecter et corriger l'erreur directement, cependant il est possible de coder l'adresse des registres pendant l'édition de lien avec un code correcteur d'erreur (*i.e.* code de Hamming) et ainsi détecter les erreurs. Ce dernier point n'est pas mis en œuvre dans nos travaux.

Le circuit logique programmable

Pour protéger le circuit synthétisé sur le FPGA il est possible d'utiliser des techniques de TMR (triple modular redundancy) qui consistent à tripliquer une partie du système. La technique de TMR dépend de la technologie du FPGA qui peut être basée sur de la SRAM ou de la flash. En effet ces technologies n'ont pas la même susceptibilité aux rayonnements ionisants [65].

Les FPGAs basés sur de la SRAM sont les plus communs dans l'industrie car ils peuvent être reprogrammés un nombre illimité de fois contrairement à ceux basés sur la

technologie flash. Cependant, ils sont très sensibles aux fautes transitoires. La technique traditionnelle de TMR est appliquée pour cette technologie. Cette approche consiste à tripliquer le circuit et à utiliser un circuit de vote majoritaire pour arbitrer le résultat, comme montré sur la Figure 3.22 (a). La principale faiblesse de cette approche est la vulnérabilité du voteur. Pour y remédier, il peut être possible d'utiliser des voteurs durcis qui sont présents sur certains FPGA ou encore de tripliquer les voteurs eux-mêmes.

Pour les FPGAs basés sur de la flash, ce qui est le cas de la SmartFusion 2, les fautes transitoires affectent uniquement les bascules D. La triplication des bascules D, montrée sur la Figure 3.22 (b), est donc suffisante.

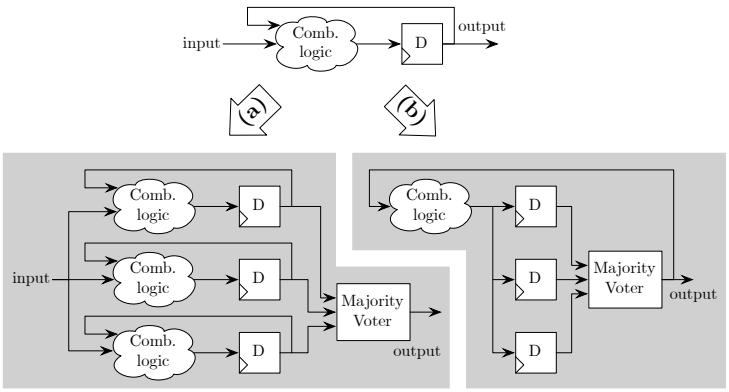

FIGURE 3.22 – Implémentation de la TMR : (a) triplication pour FPGA SRAM et (b) triplication pour des technologies flash

Notons que l'application d'une technique de TMR peut être réalisée de façon automatique par les outils de synthèse des circuits. C'est l'approche qui est mise en œuvre ici.

3.4 Conclusion

Dans ce chapitre, nous avons présenté l'architecture du mécanisme de vérification en ligne et son implémentation sur le circuit SmartFusion 2. Ce mécanisme se compose de deux parties : une partie logicielle et une partie matérielle. La partie logicielle est l'instrumentation du programme qui permet d'écrire dans des registres sur le FPGA et de mapper des variables dans le FPGA. La partie matérielle se compose de trois circuits :

— L'identificateur d'état, qui à partir des registres de données modifiés par le programme, calcule l'état des propositions atomiques.

— Le circuit de vérification qui est généré à partir d'un script qui prend en entrée l'ensemble des formules à vérifier.
— Le notifieur d'erreur qui permet de générer une interruption quand une erreur est détectée.

On a aussi présenté des approches permettant de synchroniser le programme avec le mécanisme de vérification.

Chapitre 4

Cas d'étude : La surveillance du SETR Trampoline

Dans ce chapitre, on s'intéresse à la surveillance du système d'exploitation temps réel Trampoline, au moyen de l'architecture de vérification en ligne présentée dans le chapitre précédent. La surveillance porte sur la cohérence du flot d'exécution et des structures internes lors d'un appel de service. L'idée est donc de surveiller le système quand il est en mode noyau, soit dans sa section la plus critique.

Dans un premier temps nous présenterons le système d'exploitation temps réel Trampoline, les structures internes qui le composent et le flot d'exécution d'un appel de service. Puis nous exprimerons les propriétés à surveiller. Enfin on s'intéressera à la mise en œuvre du mécanisme du surveillance et de son impact sur le système.

4.1 Le système d'exploitation temps réel Trampoline

4.1.1 Présentation de Trampoline

Trampoline est un système d'exploitation temps réel (SETR) à code source libre (licence GPLv2) qui est développé par l'équipe STR (Système Temps Réel) du LS2N (Laboratoire des Sciences du Numérique de Nantes). Il permet de concevoir des applications temps réel pour des architectures matérielles mono ou multi-cœur. Dans notre étude, nous considérons uniquement le cas de l'architecture mono-cœur.

Ce SETR est conforme au standard industriel AUTOSAR OS 4.2 [3] qui vise à développer une méthodologie commune pour la conception des architectures logicielles permettant d'assurer un niveau de fiabilité adéquate dans le domaine automobile. Les SETR développés selon cette norme respectent donc les critères suivants :

— Le système est entièrement statique, les variables et structures de données manipulées par l'exécutif sont allouées et initialisées durant la phase de compilation. Cela permet de : (1) construire des applications temps réel où l'occupation mémoire liée au SETR est connue à la compilation, (2) d'avoir un système plus prédictible et (3) d'améliorer la fiabilité du système puisque les systèmes statiques sont moins sujets aux erreurs issues de fautes transitoires [26, 25].

— Le système est basé sur une politique d'ordonnancement à priorité fixe, ce qui veut dire que la priorité des processus, qui peuvent être des tâches ou des ISRs (*Interrupt Service Routine*), est définie hors-ligne. Cela permet d'avoir un surcoût d'exécution lié à l'ordonnancement très faible en comparaison des ordonnanceurs à priorité dynamique [56]. De plus, cela permet de limiter les accès au code du noyau du système qui est la partie la plus critique.

En plus de l'ordonnancement préemptif des tâches, Trampoline propose d'autres fonctionnalités comme :

— La synchronisation des processus par l'utilisation *d'événements* qui sont des signaux qui peuvent être envoyés par un processus ou une alarme afin de réveiller un autre processus qui est en attente de l'événement en question.

— La protection des sections critiques afin de permettre un accès exclusif aux ressources (logicielles ou matérielles). Pour cela le protocole IPCP (*Immediate Priority Ceiling Protocol*)[55, 7] est utilisé. Ce protocole est basé sur l'utilisation de *ressources* qui sont utilisées dans le code des processus pour définir les sections critiques. Quand un processus prend une ressource alors sa priorité est augmen-

tée afin de le rendre non-préemptable par les autres processus qui utilisent cette même *ressource*, puis le processus reprend sa priorité précédente quand il relâche la *ressource*.

— Réaliser des actions récurrentes au cours du temps avec l'utilisation d'*alarmes* qui permettent de déclencher l'activation d'une tâche ou l'envoi d'un événement à une tâche lors de son expiration. Une alarme est associée à un *compteur* qui s'incrémente selon une source de tic qui peut provenir d'un *timer* ou d'une source externe permettant d'effectuer des actions au cours du temps.

Développement d'une application temps réel avec Trampoline

La chaîne de développement d'un exécutable utilisant Trampoline est présentée sur la Figure 4.1.

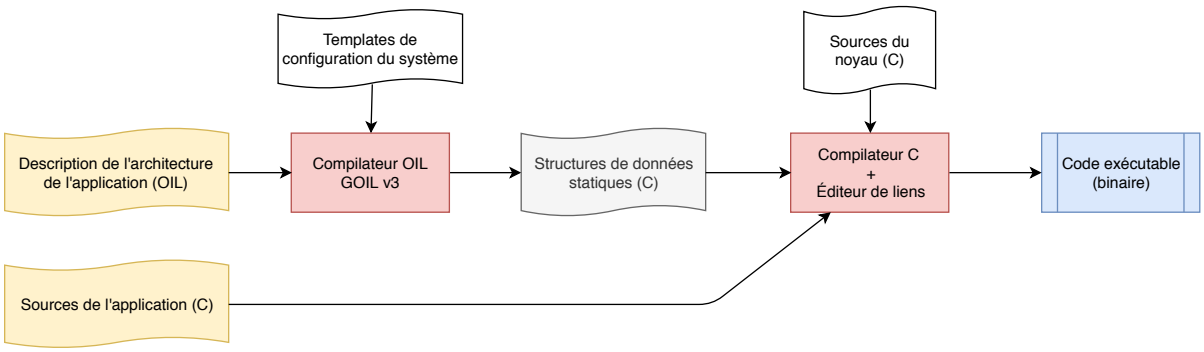

FIGURE 4.1 – Chaîne de développement simplifiée d'un exécutable utilisant Trampoline

La description logicielle de l'application est définie dans un ficher écrit dans le langage OIL (OSEK Implementation Language) ou ARXML. Dans ce fichier on définit tous les objets (tâches, alarmes, événements, ressources, etc.) utilisés par une application et on initialise leurs attributs. Comme exemple, on présente le code OIL pour créer une tâche périodique de période de 50 ticks.

```
TASK a_task {
    PRIORITY = 8;
    AUTOSTART = FALSE;  //La tâche n'est pas activée au démarrage du SETR.
    SCHEDULE = TRUE;    //La tâche est préemptible.
};

ALARM al_a_task {
```

```
        COUNTER = SystemCounter; //Compteur lié à l'alarme qui augmente de 1 tick
                                 //toutes les millisecondes.
        ACTION = ACTIVATETASK {  //Activation de la tâche a_task ...
            TASK = a_task;
        };
        AUTOSTART = TRUE {
            ALARMTIME = 10;  //... après 10 ticks par rapport à l'instant
                             //de démarrage ...
            CYCLETIME = 50;  // ... et tous les 50 ticks.
            APPMODE = std;
        };
    };
```

On définit une tâche (*a_task*) que l'on configure de façon à ne pas être activée lors du démarrage de l'application, avoir une priorité de 8 et d'être préemptible. On définit aussi une alarme (*al_a_task*) qui est configurée pour expirer toutes les 50 ms avec un offset de 10 ms (on suppose que le *SystemCounter* est relié à un timer qui a une période de 1 ms) afin de réveiller la tâche *a_task*.

En plus de la déclaration des objets de l'application, le fichier de description logicielle doit définir l'objet *OS* qui permet de spécifier le comportement du système d'exploitation vis-à-vis de l'application. Les principaux attributs que l'on retrouve dans cet objet permettent de :

- Spécifier le mode de fonctionnement du système qui peut être standard ou étendu. La principale différence entre ces deux modes est que le mode étendu comprend plus de code de détection d'erreur, comme la vérification de la cohérence de l'identifiant des objets lors de leur utilisation. Le mode étendu est donc principalement utilisé lors des phases de développement et de test de l'application.
- Ajouter des *hooks* pour exécuter des routines qui sont appelées lorsqu'un comportement interne au SETR se produit comme la détection d'une erreur lors d'un appel système, avant ou après l'ordonnancement d'une tâche.

Dans l'exemple ci-dessus le système d'exploitation est configuré pour être en mode standard et un *hook* est mis en place pour exécuter une routine à chaque fois qu'une erreur est détectée lors d'un appel système :

```
OS config {
   STATUS = STANDARD;
   ERRORHOOK = TRUE;
```

```
};
```

Les fichiers sources de l'application contiennent le code des différentes tâches de l'application ainsi que le code de la fonction principale (*main*) dans laquelle on retrouve le code d'initialisation des différents périphériques d'entrées/sorties et l'appel à la fonction $StartOS()$ qui permet de démarrer l'OS (initialisation des structures dynamiques et démarrage de l'ordonnanceur). Par exemple, dans le code de l'application présenté ci-dessous, on voit que dans la fonction *main* on appelle la fonction $InitES()$ qui gère l'initialisation des différents périphériques, puis on appelle la fonction $StartOS()$. Aussi on remarque le code de la tâche *a_task* se finit par l'appel du service $TerminateTask()$ qui permet de terminer l'instance de la tâche courante.

```
int main() {
    InitES();
    StartOS();
    return 0;
};
```

```
TASK (a_task) {
    //Code de la tâche

    TerminateTask();
};
```

Le compilateur *GOIL*, qui est un outil intégré dans Trampoline, génère à partir du fichier OIL les structures de données de l'application et le code des fonctions dépendantes de l'architecture ciblée. Les différents fichiers sources générés par GOIL ainsi que les fichiers sources du noyau et de l'application sont ensuite compilés afin de produire le code binaire de l'application.

Organisation de l'architecture de Trampoline

L'architecture du code de Trampoline est organisée selon 3 couches comme présentée sur la Figure 4.2.

Interface de programmation applicative (API) La première couche est l'API (*application programming interface*), elle regroupe les services accessibles pour la conception d'une application. Ces services sont regroupés selon plusieurs modules dont :

- *OS services* qui rassemble les services permettant de démarrer et arrêter l'exécutif.
- *Task services* qui rassemble les services de gestion des tâches comme l'activation ou la terminaison d'un job d'une tâche.
- *Interrupt services* qui rassemble les services de gestion des interruptions.
- *Alarm services* qui rassemble les services de gestion des alarmes.
- *Resource service* qui rassemble les services de gestion des ressources.

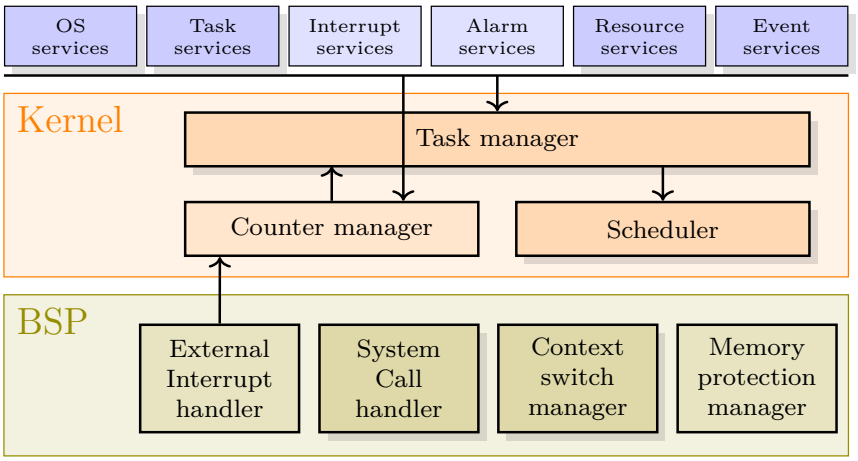

FIGURE 4.2 – Organisation de Trampoline (restriction à la partie OSEK)

— *Event service* qui rassemble les service de gestion des événements.

Un service est instancié à une application selon sa configuration. Par exemple si une application n'utilise pas d'événement alors les services du module *Event service* ne seront pas ajoutés à l'application. Cela permet d'optimiser la taille de l'OS et de diminuer le code mort.

Les fonctions du noyau La deuxième couche regroupe les fonctions qui sont utilisés par les services afin de manipuler les différentes variables et structures de données internes de l'OS. Ces fonctions sont indépendantes de l'architecture cible et sont regroupées en trois modules :

— Le *Task manager* qui rassemble les fonctions permettant de modifier l'état d'une tâche.

— Le *Counter manager* qui gère la mise à jour des différents compteurs de l'application selon l'occurrence d'événements externes.

— Le *Scheduler* (ordonnanceur) qui rassemble les fonctions de bas niveau qui permettent de manipuler la liste des tâches prêtes et la structure de données rendant compte de la tâche en cours d'exécution et des tâches candidates pour s'exécuter.

Le Board Support Package La troisième couche, le *Board Support Package* regroupe le code de bas niveau qui est dépendant de l'architecture de la machine cible. De ce fait, certaines de ces fonctions sont écrites en assembleur. Ces fonctions sont regroupées en quatre modules :

— Le *System Call Handler* est le code qui est exécuté lors du passage du processeur du mode utilisateur au mode noyau suite à l'appel d'une fonction de l'API. Sa fonction est d'effectuer l'appel du service demandé selon un identifiant transmis, puis dans le cas où l'appel du service conduit à un réordonnancement des tâches, il appelle le *Context Switch Manager* pour effectuer le changement de contexte.
— L'*External Interrupt Handler* est le code qui est exécuté lors du passage du processeur du mode utilisateur au mode noyau suite à une interruption. Son fonctionnement est similaire à celui du *System Call Handler* à la différence que ce n'est plus un service qui est exécuté mais une routine d'interruption qui est associée à l'identifiant de la source d'interruption.
— Le *Context Switch Manager* est le code qui réalise la sauvegarde de contexte d'une tâche quand elle est préemptée et le chargement de contexte quand une nouvelle tâche est ordonnancée.
— Le *Memory Protection Manager* est le code qui effectue la configuration de la *memory protection unit* selon la tâche qui est ordonnancée. Ce module n'étant pas implémenté pour notre architecture cible nous n'en tiendrons donc pas compte dans notre étude.

Restriction de l'étude Pour des raisons liées au processus d'injection de fautes qui seront approfondies dans le chapitre suivant, nous ne traiterons pas des modules associés aux interruptions. Ainsi, l'*External Interrupt Handler*, les fonctions du module *Counter manager* et les services des modules *Alarm services* et *Interrupt services* ne seront pas pris en compte.

4.1.2 Description des structures de données internes

Comme Trampoline est un système d'exploitation statique, les structures de données gérées par l'exécutif ne sont pas créées dynamiquement à l'exécution. Parmi ces structures on retrouve :

— Des structures qui contiennent les informations relatives aux objets instanciés par l'application comme les descripteurs de tâches et de ressources.
— La structure *tpl_kern* qui contient les informations relatives à l'état du noyau.
— Deux tableaux *ready_list* et *TailForPrio* où le premier contient la liste des tâches prêtes triées par priorité qui est utilisée pour l'ordonnancement des tâches et le second est utilisé afin de prendre en compte le fait que plusieurs tâches peuvent

avoir la même priorité (dans ce cas l'ordonnanceur utilise une fifo entre ces tâches).

Descripteurs de tâches

Dans Trampoline on retrouve deux descripteurs pour chaque tâche, un statique et un dynamique.

Le descripteur statique contient les informations d'une tâche qui n'évoluent pas au cours du temps comme : le pointeur sur l'adresse de la pile associée à la tâche, le pointeur sur le point d'entrée de la fonction, l'identifiant de la tâche, le nombre maximal de jobs simultanément actifs de la tâche et la valeur de la priorité de base de la tâche. Ce descripteur est donc stocké en mémoire flash.

Le descripteur dynamique contient les informations d'une tâche pouvant évoluer au cours du temps : le pointeur vers la ressource prise, la valeur du compteur d'activation, la valeur de la priorité courante [1] et l'état de la tâche. Ce descripteur est donc stocké en SRAM.

Comme présenté sur la Figure 4.3, une tâche peut être dans quatre états : *Suspended* (suspendue), *Ready* (prête), *Running* (en cours d'exécution) ou *Waiting* (en attente). Initialement une tâche peut être soit dans l'état *Ready* quand elle doit être activée automatiquement lors du démarrage du système d'exploitation, soit dans l'état *Suspended* autrement. Quand une tâche est activée par l'un des appels de service (*activate*), elle passe dans l'état *Ready* et y reste jusqu'à ce qu'elle soit démarrée (*start*). Lors d'un ordonnancement, la tâche qui a la plus haute priorité parmi les tâches prêtes est alors élue afin d'être exécutée et passe donc dans l'état *Running*. Lorsqu'une tâche s'exécute elle peut soit être préemptée par une interruption ou par l'activation d'une tâche plus prioritaire et donc repasser dans l'état *Ready*, soit se terminer et passer dans l'état *Suspended*. Certaines tâches sont susceptibles d'attendre un ou plusieurs événements afin de continuer leur exécution (*wait*), dans ce cas la tâche passe dans l'état *Waiting*. Une fois l'un des événements reçu, la tâche passe dans l'état *Ready* afin de pouvoir être ordonnancée.

En plus de ces quatre états, Trampoline implémente deux états états supplémentaires : *Autostart* et *Ready and new*. Le premier est utilisé pour l'initialisation des tâches qui doivent être prêtes au démarrage. Le second est utilisé pour différentier le cas d'une tâche prête qui a été préemptée d'une qui vient d'être activée.

1. La priorité, bien que définie statiquement, peut évoluer au cours de l'application à cause du mécanisme IPCP

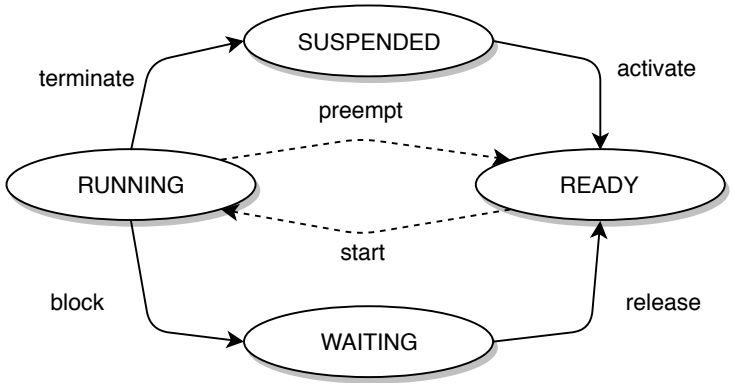

FIGURE 4.3 – États d'une tâche

Descripteurs de ressources

Une ressource est un objet qui permet d'augmenter temporairement la priorité d'une tâche afin de protéger l'accès de ressources partagées (variable ou périphérique) entre plusieurs tâches. Sous Trampoline, on distingue deux types de ressources :

Les ressources standards sont gérées par l'utilisateur via les appels des services correspondants, sont utilisées pour protéger l'accès aux sections critiques. Le descripteur de ressource standard contient les informations suivantes :

- La valeur de la priorité de la ressource qui est calculée de façon à être supérieure à la priorité la plus forte parmi les tâches étant susceptible de prendre la ressource (règle de IPCP).
- La valeur de la priorité de la tâche qui a pris la ressource.
- L'identifiant de la tâche qui a pris la ressource
- Un pointeur permettant de construire une liste chaînée de ressources détenues par tâche dans l'ordre où les ressources ont été prises.

Les ressources internes sont gérées automatiquement par l'exécutif pour rendre des tâches non préemptables avec les autres tâches de l'application ou avec un groupe spécifique de tâches. Le descripteur de ressource standard contient les informations suivantes :

- La valeur de la priorité de la ressource qui est calculée de façon à être supérieure à la priorité la plus forte parmi les tâches de l'application ou d'un groupe spécifique de tâche.
- La valeur de la priorité de la tâche qui a pris la ressource.
- Une valeur booléenne qui indique que la ressource est prise par une tâche.

La structure d'état du noyau

La structure interne *tpl_kern* comporte des informations relatives à la tâche en cours d'exécution et à la tâche élue par l'ordonnanceur (identifiant, descripteur statique et descripteur dynamique), ainsi que des trois bits *need_schedule*, *need_switch* et *need_save* qui donnent l'état du noyau :

— *need_schedule* indique qu'il va falloir tester si un réordonnancement est nécessaire à la fin de l'appel système ;
— *need_switch* indique qu'il est nécessaire de faire un changement de contexte ;
— *need_save* indique qu'il est nécessaire de faire une sauvegarde de contexte.

Par exemple, si une tâche se termine alors il n'est pas nécessaire de faire un réordonnancement et il n'est pas nécessaire de sauvegarder son contexte.

Les structures pour l'ordonnancement des tâches

L'ordonnancement des tâches sous Trampoline se fait à l'aide de deux structures :

— La liste *readyList* qui contient l'identifiant de chaque tâche prête à être exécutée. Cette liste est ordonnée sous la forme d'un tas-max (ou max-heap), la clé de tri étant la priorité de sorte à ce que la tâche la plus prioritaire soit mise en première position.
— Le tableau *tailForPrio*, qui est utilisé pour calculer la priorité de l'instance d'une tâche. Il contient pour chaque priorité statique une valeur qui est décrémentée à chaque fois qu'une tâche avec cette priorité est ajoutée à la *readyList* . Ce tableau est donc utilisé pour prendre en compte l'ordonnancement de deux tâches ayant la même priorité statique de manière à ce que la plus anciennement activée soit la plus prioritaire (fifo).

4.1.3 Flot d'exécution d'un appel de service

Les services se présentent sous la forme de fonctions pouvant être appelées dans le code des tâches ou ISRs de l'application. On peut classer ces services selon qu'ils peuvent entraîner un ré-ordonnancement des tâches ou non. Dans la suite, nous nous intéresserons principalement aux services qui peuvent impliquer un ré-ordonnancement. Ces services sont listés dans la Table 4.1.

Lors de l'appel d'un service au niveau de l'application, le processeur passe du mode utilisateur au mode noyau, ce qui a pour conséquence d'exécuter le code du *System Call*

Service	Description
ActivateTask	Permet d'instancier une tâche spécifique.
TerminateTask	Permet de terminer l'instance de la tâche courante.
ChainTask	Permet d'activer une nouvelle instance d'une tâche spécifique (comme *ActivateTask*) et de terminer l'instance de la tâche courante (comme *TerminateTask*).
Schedule	Réalise un ré-ordonnancement des tâches.
SetEvent	Permet d'envoyer un événement à une tâche spécifique.
WaitEvent	Met la tâche courante en attente d'événements, si au moins l'un des événements est déjà arrivé la tâche continue son exécution sinon elle se met en attente.
GetResource	Permettent de gérer le partage de ressources par l'utilisation de protocole IPCP
ReleaseResource	

TABLE 4.1 – Liste des principaux services de Trampoline

Handler. Ce code, écrit en assembleur et spécifique à l'architecture du processeur, est composé de 4 étapes (voir Figure 4.4) qui sont respectivement en charge de (1) récupérer l'adresse de la fonction correspondant à l'identifiant du service, (2) exécuter le code spécifique du service appelé, (3) sauvegarder le contexte de la tâche appelante si la tâche en cours d'exécution est préemptée et (4) charger le contexte de la tâche élue lors de l'ordonnancement.

Étape 0 : Passage en mode superviseur

Le passage du mode utilisateur au mode superviseur se fait à partir d'une instruction assembleur qui déclenche une interruption logicielle permettant d'exécuter le code correspondant au mode superviseur. Dans le cas d'un processeur à architecture ARM, cette instruction est *svc*.

Du point de vue de l'utilisateur, un appel de service consiste à appeler une fonction. Le code de cette fonction est généré en assembleur par l'outil GOIL et charge l'identifiant du service à appeler dans un registre puis passe en mode superviseur. Par exemple, le code de la fonction *ActivateTask()* est :

```
ActivateTask:
    movs r3,#OSServiceId_ActivateTask ;chargement de l'identifiant
                                      ;du service dans r3
    svc  #OSServiceId_ActivateTask    ;passage en mode superviseur
```

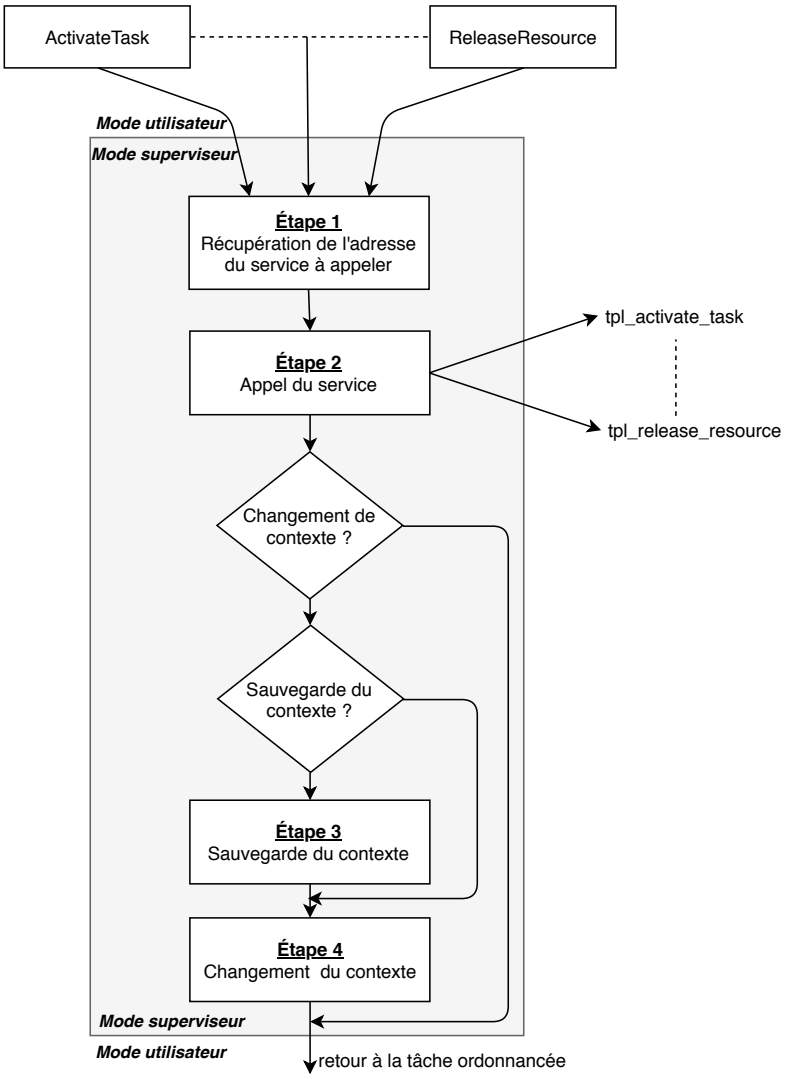

FIGURE 4.4 – Diagramme représentant l'exécution d'un appel système dans Trampoline

Étape 1 : Récupération de l'adresse du service

L'adresse des différents services est disponible dans une table de répartition qui est générée par le compilateur GOIL de telle sorte que seuls les services nécessaires à l'application sont présents. Ceci permet de limiter le code mort et de diminuer l'empreinte mémoire. L'accès à l'adresse d'un service se fait via l'identifiant du service, en effet celui-ci correspond à l'index de l'adresse du service dans la table. Pour une architecture ARM, le code correspondant à la récupération de l'adresse d'un service est :

```
ldr   r4,=tpl_dispatch_table ;Chargement de l'adresse de la table
                             ;de répartition
lsls  r3,r3,#2               ;Calcul de l'offset du service
                             ;(pointeur sur 4 octets)
ldr   r3,[r4,r3]             ;Récupération de l'adresse du service
```

L'adresse de la table de répartition est chargée dans un registre ($r4$ dans l'exemple). Puis l'offset du service est calculé à partir de l'identifiant. Enfin l'adresse du service est récupérée.

Étape 2 : Appel d'un service

L'appel d'un service consiste à exécuter les fonctions internes de l'OS qui vont mettre à jours les différentes structures internes de l'OS.

Pour illustrer les mise à jours effectuées pendant l'appel d'un service, listons les principales modifications réalisées dans le cas où une tâche $T1$ fait un appel au service *ActivateTask* pour activer une tâche suspendue $T2$ de priorité supérieure :

- modification du descripteur dynamique de $T2$ pour indiquer que la tâche passe dans l'état *Ready* ;
- mise à 1 du bit *need_schedule* pour indiquer qu'il va falloir faire un ordonnancement ;
- ajout de $T2$ dans la *ready_list* (cette liste est triée et T2 arrive alors en tête)
- comme *need_schedule* = 1 et que $T1$ n'est plus la tâche la plus prioritaire :
  - modification du descripteur dynamique de $T1$ pour indiquer que la tâche passe dans l'état *READY* (préemption) ;
  - ajout de $T1$ dans la *ready_list* ;
  - mise à 1 des bits *need_save* et *need_switch* ;
  - mise à jour des champs correspondants au processus élu avec les informations de la tâche $T2$.

Étape 3 : Sauvegarde du contexte

La sauvegarde du contexte consiste à copier l'ensemble des registres utilisateurs et le registre d'état dans la zone mémoire dédiée de la tâche qui est en cours d'exécution (la tâche qui à réalisée l'appel d'un service). Cette action est réalisée uniquement si le bit *need_save* de la structure *tpl_kern* a été mis à 1 pendant l'appel du service.

Étape 4 : Changement de contexte

Le changement de contexte, qui se fait uniquement si le bit *need_switch* de la structure *tpl_kern* a été mis à 1 pendant l'appel du service, se fait en deux étapes. La première étape est de copier dans la structure *tpl_kern* les informations relatives au processus élu vers les informations relatives au processus en cours d'exécution, puis de passer l'état du processus en cours d'exécution dans l'état $RUNNING$. La seconde étape est le chargement du contexte qui consiste à charger dans les registres utilisateurs les valeurs contenues de la zone mémoire dédiée au contexte.

4.2 Spécification des propriétés à vérifier

4.2.1 Le principe de la surveillance du noyau de Trampoline

Pour surveiller l'exécution du code du noyau de Trampoline, nous avons besoin d'exprimer un certain nombre de propriétés. Celles-ci portent sur la vérification du flot de contrôle et sur la cohérence des structures internes de l'OS.

Le flot de contrôle

La vérification du flot de contrôle peut se faire selon plusieurs niveaux de granularité :

— Au niveau des instructions machine où l'idée est de surveiller le pointeur d'instruction (PC pour *Program Counter*), c'est-à-dire que lors de l'exécution d'une instruction de branchement la valeur chargée dans le registre PC est cohérente et que pour les autres instructions le registre PC est incrémenté correctement.

— Au niveau des blocs de base.

— Au niveau des appels de fonctions.

Dans le cas de notre architecture, comme nous n'avons pas accès aux registres internes du système (notamment le registre PC), il n'est pas possible de vérifier le flot de contrôle au niveau des instructions machines sans altérer fortement les performances du système. Plus le niveau de la granularité de la vérification de flot de contrôle est fin, meilleur sera la détection des erreurs. Par contre cela impose d'ajouter plus d'instructions dans le code pour générer la trace d'exécution, ce qui ajoute donc un surcoût d'exécution. D'autre part, plus le nombre de propriétés à vérifier est grand, plus l'utilisation des ressources logiques sera élevée. Comme compromis, nous proposons de surveiller l'exécution du code du *System Code Handler* au niveau des blocs de base car c'est un code critique commun

à tous les appels systèmes et de surveiller l'exécution des services internes au niveau des appels de fonctions.

La cohérence des structures internes

La vérification de la cohérence des structures internes de l'OS n'est réalisée que pour les structures qui sont amenées à être modifiées pendant l'exécution du code du noyau (structures dynamiques). En effet les structures statiques (celles qui ne sont pas modifiables) sont stockées dans la mémoire ROM et sont donc naturellement protégées. On peut distinguer deux types de structures dynamiques :

— Les structures locales qui sont allouées sur la pile d'exécution lors de leur déclaration dans une fonction. Pour monitorer ces structures il serait nécessaire de modifier tout le code du noyau de manière à les rendre globales.

— Les structures globales qui sont allouées initialement en mémoire.

Nous nous focaliserons donc uniquement sur la surveillance des structures dynamiques globales.

4.2.2 Surveillance de l'exécution du *System Call Handler*

Pour exprimer les propriétés permettant de vérifier le flot de contrôle du code du *System Call Handler*, nous avons besoin de définir des propositions atomiques pour l'entrée et la sortie de chacune des étapes du code. Ces propositions atomiques sont listées dans la Table 4.2.

Call_Handler_enter	indique que l'on entre dans le code
Call_Handler_exit	indique que l'on sort du code
Service_OS_start	indique le début de l'exécution d'un service
Service_OS_end	indique la fin de l'exécution d'un service
Context_Switch_start	indique qu'un changement de contexte doit être réalisé
Context_Switch_end	indique la fin du changement de contexte
Context_Save_start	indique qu'une sauvegarde du contexte doit être réalisé
Context_Save_end	indique la fin de la sauvegarde du contexte

TABLE 4.2 – Liste des propositions atomiques pour la surveillance du flot de contrôle du *System Call Handler*

Ces propositions atomiques correspondent à l'occurrence d'un événement, elles sont donc vraies pendant un cycle d'horloge. L'algorithme 2 donne une représentation du *System Call Handler* avec la génération des événements.

Algorithm 1 Algorithme du *System Call Handler* avec instrumentation

```
1: Générer l'événement associé à Call_Handler_enter
2: need_switch, need_switch ← 0
3: Obtention de l'identifiant du service à appeler
4: Générer l'événement associé à Service_OS_start
5: Appel du service
6: Générer l'événement associé à Service_OS_end
7: if need_switch == 1 then
8:    Générer l'événement associé à Context_Switch_start
9:    if need_save == 1 then
10:      Générer l'événement associé à Context_Save_start
11:      Sauvegarde du contexte
12:      Générer l'événement associé à Context_Save_end
13:   end if
14:   Changement de contexte
15:   Générer l'événement associé à Context_Switch_end
16: end if
17: Générer l'événement associé à Call_Handler_exit
```

Les descriptions des principales propriétés permettant de vérifier le flot de contrôle du *System Call Handler* sont spécifiées de façon à vérifier (1) les imbrications des différentes étapes et (2) l'ordre d'exécution des étapes. Un exemple pour chacun de ces types de propriété est donné ci-dessous avec leur traduction en formule ptLTL :

Propriété d'imbrication : L'exécution d'une sauvegarde de contexte implique d'avoir un changement de contexte.

$$\boxdot([Context\_Save\_start; Context\_Save\_end)_s \rightarrow [Context\_Switch\_start; Context\_Switch\_end)_s)$$

Propriété d'ordre : Quand on sort du code du *System Call Handler*, on ne peut pas être en train d'exécuter le code de l'appel du service, du changement de contexte ou du changement de contexte.

$$\boxdot(Call\_Handler\_exit \rightarrow (\neg[Service\_OS\_start; Service\_OS\_end)_s \wedge \neg[Context\_Switch\_start; Context\_Switch\_end)_s \wedge \neg[Context\_Save\_start; Context\_Save\_end)_s)$$

Au total, 6 propriétés ont été spécifiées (3 propriétés d'imbrication et 3 propriétés d'ordres). Ces propriétés sont présentées en Annexe B.

4.2.3 Surveillance des appels de fonctions

Le graphe des appels des fonctions internes est présenté sur la Figure 4.5. Un arc d'une première fonction à une seconde indique que la seconde fonction peut être appelée par la première. Par exemple la fonction *terminate* peut appeler la fonction *release_internal_resource* et être appelée par les fonctions *chain_task_service* et *terminate_task_service*.

Pour chaque fonction on veut vérifier que :

1. Il n'y a pas d'appel récursif, c'est-à-dire qu'une fonction ne peut être appelée que si sa précédente exécution est bien terminée (**propriété 1**).
2. Lorsque l'on sort d'une fonction, cela implique que l'on était bien dans la fonction (**propriété 2**).
3. Lorsque que l'on entre dans une fonction, au moins un de ces possibles appelants est en cours d'exécution (**propriété 3**).

On remarque que la troisième propriété ne s'applique pas aux fonctions qui sont directement appelées par le *System Call Handler* comme *chain_task_service* et *terminate_task_service*.

Pour exprimer ces propriétés sous la forme de formules ptLTL, il faut assigner à chaque fonction deux propositions atomiques $nom\_de\_la\_fonction_{enter}$ et $nom\_de\_la\_fonction_{exit}$ qui indiquent respectivement que l'on entre et que l'on sort de la fonction spécifiée.

Si on se focalise sur la fonction *terminate* (voir Figure 4.6) on peut exprimer les trois propriétés suivantes :

$\boxdot(terminate_{enter} \rightarrow [terminate_{exit}; \odot terminate_{enter})_w)$ //propriété 1

$\boxdot(terminate_{exit} \rightarrow [terminate_{enter}; \odot terminate_{exit})_s)$ //propriété 2

$\boxdot([terminate_{enter}; terminate_{exit})_s \rightarrow$
$([chain\_task\_service_{enter}; chain\_task\_service_{exit})_s \vee$
$[term\_task\_service_{enter}; term\_task\_service_{exit})_s))$ //propriété 3

On remarque que pour la première formule, on utilise l'opérateur intervalle dans sa version *weak*. Cela permet d'indiquer qu'à l'initialisation la fonction n'est pas en train de s'exécuter.

Ces propriétés sont à donner pour les 27 fonctions du noyau, ce qui représente 60 formules ptLTL.

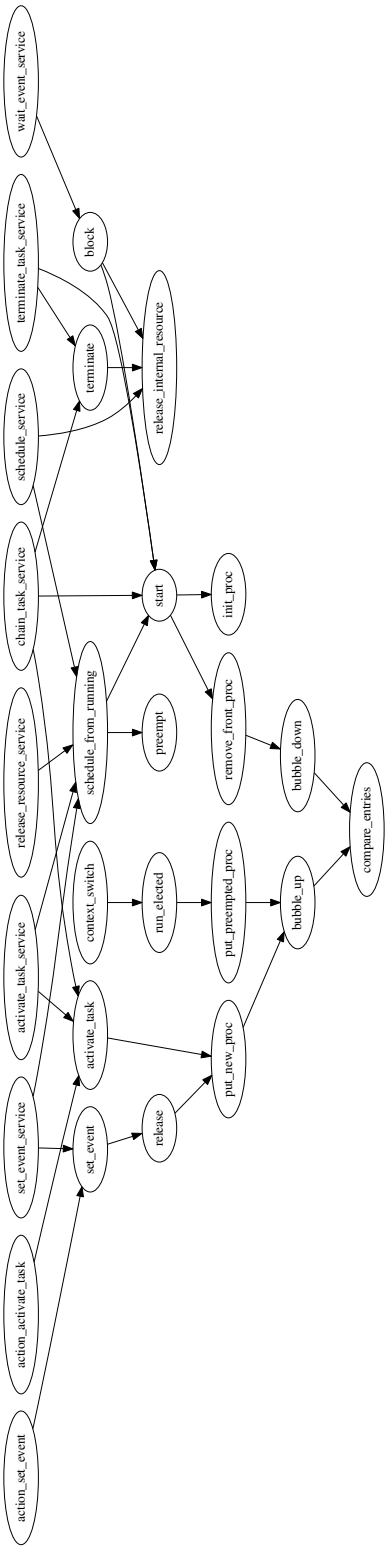

FIGURE 4.5 – Graphe d'appel des fonctions de Trampoline

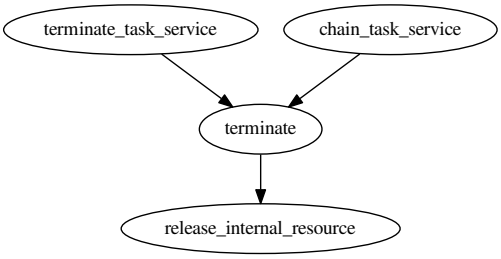

FIGURE 4.6 – Graphe d'appel des fonctions focalisé sur la fonction *terminate*

4.2.4 Surveillance des structures internes

De façon générale, la surveillance des structures internes consiste à vérifier pour chaque champ des structures internes (1) la cohérence de la valeur contenue (intervalle utilisé ou valeurs possibles) et (2) la cohérence de la modification du champ vis-à-vis du flot de contrôle.

Exemple du champ *resource* du descripteur dynamique de tâche

Cohérence de la valeur Le champ *resource* du descripteur dynamique de tâche est un pointeur sur la ressource standard prise par la tâche. Cette valeur peut donc être :

- l'adresse du descripteur d'une ressource standard si dans l'architecture logicielle de l'application celle-ci est associée à la tâche,
- $NULL$ pour indiquer que la tâche n'a pas de ressource.

Pour vérifier la cohérence de ce champ on définit la proposition atomique *Res_Coherency_Task_X* (où X correspond au nom de la tâche) qui est vrai si la valeur contenue dans le champ est bien l'une des valeurs attendues. De ce fait, la formule ptLTL à vérifier est :

$$\boxdot(Res\_Coherency\_Task\_X)$$

Par exemple dans le code de la Figure 4.7 on présente la description de la proposition atomique *Res_Coherency_Task_T1* dans le cas où une tâche $T1$ peut prendre la ressource $Res1$ dont l'adresse du descripteur de la ressource est $0x50000038$.

Cohérence de la modification du champ La modification de ce champ se fait uniquement lors de l'exécution du code des fonctions *ReleaseResourceServie* et *GetResour-*

```
Res_coherancy_Task_T1 <= '1' when ((reg_task_desc_T1_res = X"00000000")
                                or (reg_task_desc_T1_res = X"50000038") ) else '0';
```

FIGURE 4.7 – Exemple du code VHDL pour la description de la proposition atomique

ceService. Les propositions atomiques associées à ces deux fonctions sont respectivement $ReleaseResServ$ et $getResServ$.

La propriété ptLTL correspondant à la modification de ce champ est donc :

$$\boxdot(Modif\_Res\_Task\_TX \rightarrow ReleaseResServ \vee getResServ)$$

La structure *tpl_kern*

Nous allons présenter plus en détail le cas de la structure *tpl_kern*. Comme cette structure évolue régulièrement lors de l'appel d'un service, sa surveillance nécessite de vérifier un grand nombre de propriétés.

Comme présenté sur la Figure 4.8, la structure *tpl_kern* se compose des informations relatives à la tâche en cours d'exécution et de la tâche élue par l'ordonnanceur, ainsi que des trois bits *need_schedule*, *need_switch* et *need_save* qui donnent l'état du noyau (les deux derniers bits sont regroupés dans le même élément *need_save_switch* .

```
typedef struct
{
  tpl_proc_static          s_running;
  tpl_proc_static          s_elected;
  tpl_proc                 running;
  tpl_proc                 elected;
  sint32                   running_id;
  sint32                   elected_id;
  uint8                    need_save_switch;
  tpl_bool                 need_schedule;
} tpl_kern_state;
```

FIGURE 4.8 – Structure *tpl_kern*

On définit l'état du noyau comme la combinaison des trois bits d'état :

$$E = \{need\_schedule, need\_save, need\_switch\}$$

Chacun de ces états correspond à une proposition atomique.

À partir de l'automate, de la Figure B.5, qui modélise l'évolution des états du noyau, il est possible de définir plusieurs propriétés.

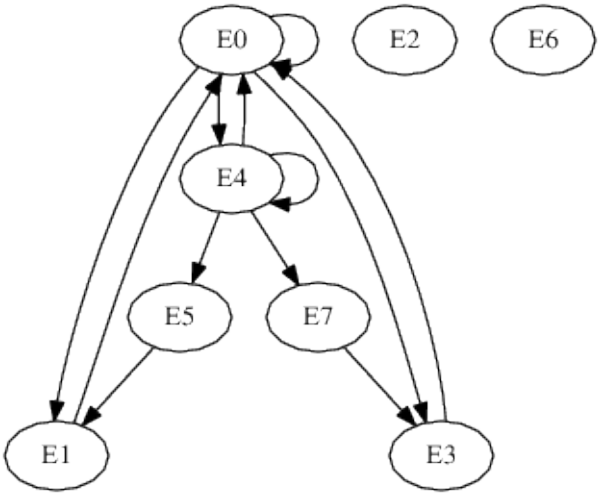

FIGURE 4.9 – Modèle de l'évolution des états du noyau de Trampoline, avec les variables d'états $E = \{need\_schedule, need\_save, need\_switch\}$ Par exemple $E5$ correspond à $\{need\_schedule, \overline{need\_save}, need\_switch\}$

Propriété de sûreté : Les états $E2$ et $E6$ ne sont pas accessibles, ce qui veut dire qu'il n'est pas possible d'avoir besoin de sauvegarder le contexte ($need\_save = 1$) et de ne pas avoir à changer le contexte (($need\_switch = 1$).

$$\boxdot(\neg(E2 \vee E6))$$

Propriétés d'accessibilité : Ces propriétés expriment les conditions nécessaires à l'accès d'un état spécifique. Par exemple, on peut voir que l'accès à l'état $E3$ ne peut se faire qu'à partir des états $E0$ et $E7$.

$$\boxdot(\uparrow E0 \rightarrow \odot\neg(E5 \vee E7))$$
$$\boxdot(\uparrow E1 \rightarrow \odot(E0 \vee E5))$$
$$\boxdot(\uparrow E3 \rightarrow \odot(E0 \vee E7))$$
$$\boxdot(\uparrow E4 \rightarrow \odot E0)$$
$$\boxdot(\uparrow E5 \rightarrow \odot E4)$$
$$\boxdot(\uparrow E7 \rightarrow \odot E4)$$

Quand on entre dans l'état $E4$ ($need\_schedule = 1$), cela veut dire que l'on se trouve juste avant le ré-ordonnancement. Ainsi à cet instant, les champs *elected* et *running* de la structure *tpl_kern* sont identiques. Pour exprimer cette propriété, on définit la proposition atomique *run_elec* qui est à vraie quand ces champs sont identiques. La formule ptLTL correspondante est donc :

$$\boxdot(\uparrow E4 \rightarrow run\_elec)$$

Prise en compte du service : Il est aussi possible d'exprimer des propriétés sur l'évolution des états du noyau en fonction du service qui est appelé. Pour cela il faut définir une proposition atomique pour chaque service qui est vraie pendant l'exécution de son code. On peut exprimer deux propriétés pour chaque service, la première permet de définir les états accessibles par le service et la seconde permet d'indiquer les états finaux possibles. Par exemple, pour le service $ActivateTask$, l'automate qui représente l'évolution des états du noyau est représenté sur la Figure 4.10.

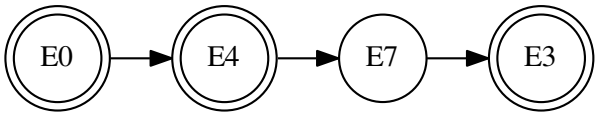

FIGURE 4.10 – Modèle de l'évolution des états du noyau de Trampoline pour l'appel du service $ActivateTask$. Un double cercle indique un état final.

Comme pour tous les services, l'état initial est l'état $E0$. Si le nombre maximal d'activation de la tâche à activer (paramètre statique qui est définie dans le descripteur statique de la tâche) est atteint alors le service s'arrête et finit dans l'état $E0$, sinon le bit *need_schedule* est mis à 1. Comme on atteint l'état $E4$ un ordonnancement des tâches est effectué. L'ordonnanceur compare la priorité de la tâche en cours d'exécution et de la tâche en tête de la *ready_list*. Si la tâche en cours d'exécution a la plus haute priorité, alors il n'y a pas besoin de faire un changement de contexte et service se termine dans l'état $E4$. Autrement, un changement de contexte est requis ($need\_switch = 1$) et le contexte de la tâche en cours doit être sauvegardé ($need\_save = 1$), donc on passe dans l'état $E7$. Enfin le service se termine dans l'état $E3$, le bit *need_schedule* est mis à 0 juste après la mise à jour de la tâche élue. La propriété qui spécifie l'accessibilité de état pendant l'exécution du service $ActivateTask$ est donc :

$$\boxdot(ActivateTaskService \rightarrow \neg(E1 \vee E5))$$

Et la propriété qui spécifie les états finaux possibles pour ce service est :

$$\boxdot(\downarrow ActivateTaskService \rightarrow (E0 \vee E3 \vee E4))$$

Prise en compte du System Call Handler : On peut aussi mettre en relation les propositions atomiques de la structure *tpl_kern* avec celles définies pour vérifier le flot de contrôle du *System Call Handler*. De cette façon, il est possible d'exprimer les propriétés sur les branchements du flot de contrôle. Par exemple :

Les informations contenues dans la structure *tpl_kern* relatives à la tâche en cours d'exécution et à la tâche élue sont mises à jours pendant l'exécution du code du *System Call Handler* et doivent être identiques le reste du temps.

$$\boxdot(Call\_Handler\_enter \rightarrow run\_elec)$$
$$\boxdot(Call\_Handler\_exit \rightarrow run\_elec)$$

Avant de faire le changement de contexte le bit *need_switch* doit être à 1. Après le changement de contexte les informations relatives à la tâche en cours d'exécution et de la tâche élue doivent être identique.

$$\boxdot(Context\_Switch\_start \rightarrow (E1 \vee E3))$$
$$\boxdot(Context\_Switch\_end \rightarrow run\_elec)$$

4.2.5 Synthèse des propriétés spécifiées

Les propriétés qui ont été spécifiées pour la surveillance du noyau de Tampoline sont soit indépendantes, soit dépendantes de l'application.

Les propriétés indépendantes sont celles qui portent sur le flot de contrôle et sur la structure *tpl_kern*. Cela correspond à un total de 100 propriétés.

Les propriétés dépendantes sont celles qui portent sur les structures internes qui dépendent de l'application comme les descripteurs de tâches ou de ressources.

Une spécification plus complète de ces propriétés est présentée dans l'Annexe B de ce document.

4.3 Mise en œuvre de la surveillance de Trampoline

La mise en œuvre de la surveillance de Trampoline consiste à (1) implanter les structures que l'on veut surveiller dans le FPGA et à générer les événements pour la surveillance du flot de contrôle par l'ajout d'instructions, et (2) générer le circuit des différents éléments du dispositif matériel de vérification selon les formules à vérifier.

4.3.1 Instrumentation de Trampoline

L'instrumentation de Trampoline est réalisée par l'ajout de directives de préprocesseur dans le code du noyau et par la modification des templates utilisés par le compilateur GOIL afin de compiler le code de Trampoline avec ou sans instrumentation.

Modification du plan mémoire

La première étape est de modifier le fichier de template *memory_map* relatif à l'éditeur de lien, qui configure les zones mémoires du système afin d'ajouter une nouvelle zone correspondant à la mémoire du FPGA. Comme présenté sur la Figure 4.11, la zone *fabric* est ajoutée au plan mémoire. La zone *fabric* a pour adresse d'origine $0x50000000$, ce qui correspond à la mémoire du FPGA.

```
MEMORY
{
     rom (rx)   : ORIGIN = 0x60000000, LENGTH = 256k
     romMirror (rx) : ORIGIN = 0x00000000, LENGTH = 256k
     ram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
     fabric (rwx) : ORIGIN = 0x50000000, LENGTH = 64k
}
```

FIGURE 4.11 – Code du plan mémoire de la SmartFusion2 pour l'instrumentation

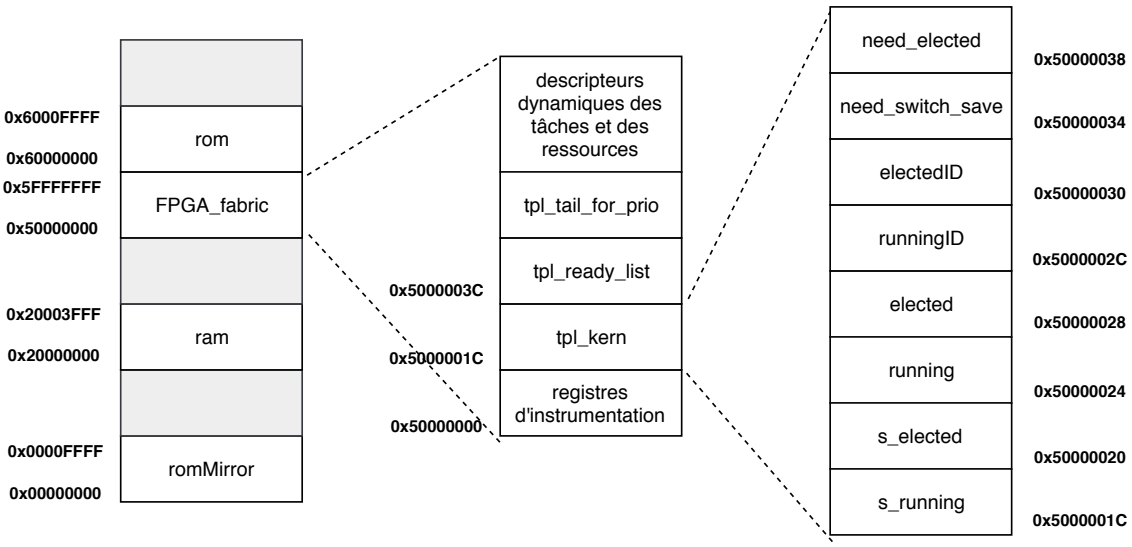

FIGURE 4.12 – Représentation du plan mémoire avec un zoom sur la structure *tpl_kern*

Ensuite il faut modifier les templates et le code du noyau afin que les adresses mémoire des structures de données que l'on veut surveiller soient bien allouées au niveau de la zone mémoire *fabric*. Il faut aussi définir dans la zone mémoire *fabric* les variables qui permettent de recevoir les informations qui traduisent la génération d'un événement et le registre pour recevoir l'identifiant du service en cours. Comme on peut le voir sur la Figure

4.12, les registres correspondant aux structures qui ne dépendent pas de l'application et à la surveillance du flot de contrôle sont alloués au début, ainsi leur adresse est toujours la même. Aussi comme le bus de communication entre le processeur et la mémoire du FPGA impose un alignement mémoire sur 32 bits, il faut modifier le code des différentes structures afin que les adresses des différentes variables soient bien alignées sur 32 bits. Par exemple pour la structure *tpl_kern*, la variable *need_switch_save* est bien codée sur 32 bits.

Ajout des instructions pour la génération des événements

On a besoin d'ajouter des instructions au début et à la fin de chaque fonction, et dans le code du *System Call Handler* pour générer les événements permettant d'évaluer les propositions atomiques pour la vérification du flot de contrôle. Chaque événement est associé à un bit de l'un des registres d'instrumentation, ce qui permet de déduire la valeur à écrire sur le registre d'instrumentation correspondant, comme expliqué dans la section 3.3.5. Pour rappel, cette valeur est codée sur 32 bits de façon à être tripliquée en subdivisant le mot mémoire en trois groupes de 10 bits (les deux bits de poids fort ne sont pas utilisés). Ces valeurs, ainsi que les adresses des registres d'instrumentation, sont définies dans un fichier *fabric_definition.h*.

Par exemple le code à ajouter au début de la fonction *tpl_terminate* est :

```
#if (KERNEL_MONITORING)
        reg_instru_kernel_function_4 = TERMINATE_ENTER;
#endif
```

Où *KERNEL_MONITORING* est une directive préprocesseur qui est vraie quand on veut surveiller l'exécution du noyau, *reg_instru_kernel_function_4* est le quatrième registre d'instrumentation et *TERMINATE_ENTER* est la valeur à écrire pour générer l'événement qui indique que l'on entre dans la fonction *tpl_terminate*.

On retrouve dans le fichier *fabric_definition.h* la valeur correspondant à *TERMINATE_ENTER* dans le cas où celui-ci est associé au 4ème bit de poids faible en partant de 0 :

```
#define TERMINATE_ENTER          0x01004010 //4eme bit
```

Accès à l'identifiant du service

Comme on l'a vu dans le paragraphe 4.2.4, on a besoin d'indiquer dans un registre sur le FPGA l'identifiant du service qui est en train de s'exécuter. On a vu précédemment que l'identifiant d'un service est généré par le compilateur GOIL et dépend donc de l'application. Pour faire en sorte que l'identifiant ne dépende pas de l'application, on génère via le compilateur GOIL un tableau qui fait correspondre à un service un identifiant unique.

```
#if (KERNEL_MONITORING)
ldr r4,=tpl_unique_id_table       ; chargement de l adresse du tableau
                                  ; de correspondance
ldrb r4,[r4,r5]                   ; recuperation de l identifiant unique
ldr r5,=AD_REG_OS_INSTRU_SERVICE  ; chargement de l adresse du registre
                                  ; REG_OS_INSTRU_SERVICE
str r4,[r5]                       ; stockage de l identifiant dans
                                  ; REG_OS_INSTRU_SERVICE
#endif
```

FIGURE 4.13 – Code en assembleur pour la notification de l'identifiant du service où $r5$ contient initialement l'identifiant dynamique du service

La partie du code correspondant à la notification de l'identifiant unique du service dans le *System Call Handler* est présentée sur la Figure 4.13. On charge dans le registre $r4$ l'adresse du tableau de correspondance entre l'identifiant du service « réel » et l'identifiant unique. Ensuite on récupère dans $r4$ la valeur de l'identifiant unique puis on la stocke dans le registre *REG_OS_INSTRU_SERVICE* du FPGA.

4.3.2 Génération du circuit du dispositif de vérification

Le flot de génération du circuit du dispositif de vérification est composé de plusieurs scripts qui automatisent sa génération, comme présenté sur la Figure 4.14.

En entrée du script, deux fichiers sont nécessaires :

— ***info.json*** qui regroupe toutes les informations statiques sur les différents éléments (tâches, ressources, *ready_list*) de l'application à surveiller. Ce fichier est généré par le compilateur GOIL suite à une modification de celui-ci via un template.

— ***appli.map*** est le plan mémoire de l'application qui est automatiquement généré par l'éditeur de liens.

Le fichier *Identificateur_PA.vhd* contient le code vhdl pour l'identification des différentes propositions atomiques qui composent les formules ptLTL à vérifier. Pour générer

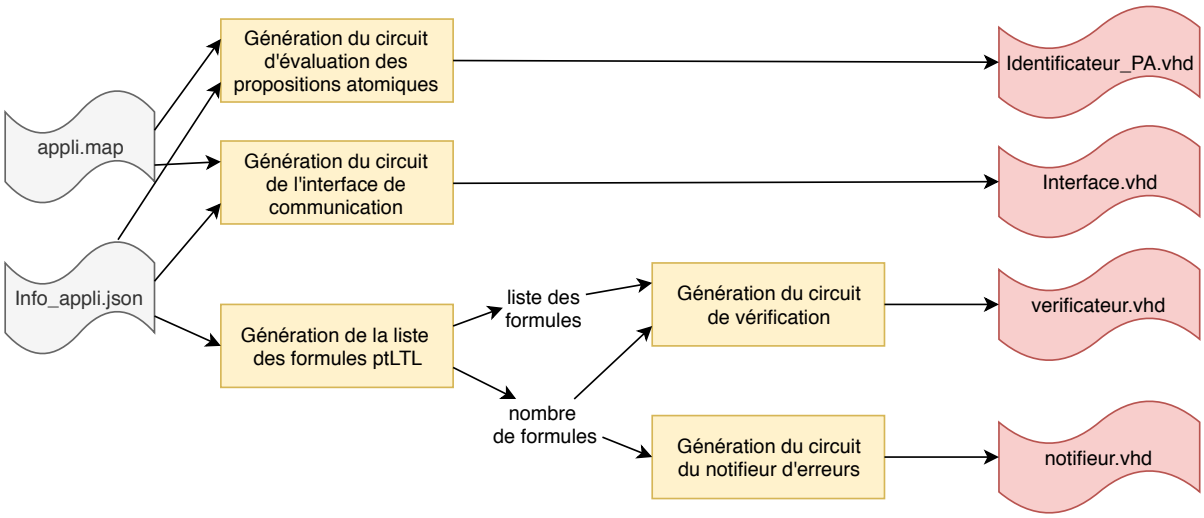

FIGURE 4.14 – Flot de génération du circuit du dispositif de vérification en ligne

ce code il faut définir les propositions atomiques à vérifier, ce qui est fait à partir de l'analyse du fichier *info.json*. Pour pouvoir écrire le code correspondant à certaines propositions atomiques il faut extraire du fichier *appli.map* l'adresse des différents structures.

Le fichier *Interface.vhd* intègre le code du protocole pour les communications avec le bus APB et implémente les différents registres du FPGA. Pour générer ce fichier il est nécessaire d'extraire l'adresse des différentes structures du noyau.

Le fichier *notifieur.vhd* contient le code de notificateur d'erreurs, la seule information à avoir pour le générer est le nombre de formules ptLTL qu'il faut vérifier.

Le fichier *verificateur.vhd* est généré à partir de la liste des formules ptLTL à vérifier. Le fonctionnement de ce script est expliqué dans la section 3.2.1. La liste des formules ptLTL à vérifier, est de la même façon que pour la liste des propositions atomiques, déduite du fichier *info.json*.

4.3.3 Évaluation du coût de mise en œuvre du dispositif de vérification

Évaluation du surcoût temporel

Pour évaluer le surcoût temporel induit par l'instrumentation du code de Trampoline, on mesure le temps d'exécution des principaux services, selon qu'ils entraînent un réordonnancement ou non, avec et sans instrumentation.

Le surcoût temporel correspond au temps supplémentaire ajouté à un service quand on

l'instrumente par rapport à son temps d'exécution de base. On présente dans la Table 4.3 ces surcoûts d'exécution, exprimés en pourcentage, pour les principaux services, selon que le code du noyau est compilé [2] dans sa version standard avec une fréquence de 142MHz.

	Temps d'exécution		**Surcoût d'exécution**
	sans instrumentation	**avec instrumentation**	
ActivateTask sans changement de contexte	4.9 us	6.46 us	32.6 %
ActivateTask avec changement de contexte	17.2 µs	22.6 µs	31.4 %
TerminateTask	5.4 us	7.4 us	37.2 %
ChainTask	12.8 us	16.7 us	30.1 %
Schedule sans changement de contexte	3.4 us	4.6 us	33.5 %
Schedule avec changement de contexte	13.5 us	17.7 us	31.2 %
SetEvent sans changement de contexte	3.3 us	3.8 us	18.2 %
SetEvent avec changement de contexte	11.8 us	15.6 us	32.7 %
WaitEvent sans changement de contexte	2.1 us	2.5 us	16.3 %
WaitEvent avec changement de contexte	6.9 us	9.3	34.4 %
GetResource	2.1 us	2.9 us	34.3 %
ReleaseResource sans changement de contexte	1.9 us	2.6 us	31.8 %
ReleaseResource avec changement de contexte	8.35 us	11.5 us	38 %

TABLE 4.3 – Mesure du surcoût temporel d'exécution des services de Trampoline

Comme on peut le voir le surcoût temporel d'exécution d'un service de Trampoline dû à l'instrumentation au maximum de 38%. Sachant que généralement le temps d'exécution d'un appel service est négligeable en comparaison du temps d'exécution du code de l'application, qui est mesuré en milliseconde, on peut considérer que le surcoût mesuré l'est aussi.

2. Version 4.9.3 de arm-none-eabi-gcc et compilation en $-O0$

Évaluation de l'empreinte mémoire

L'évaluation de l'empreinte mémoire est réalisée par l'analyse du binaire de l'application en mesurant la différence de l'empreinte mémoire du code du noyau de Trampoline stockée en mémoire ROM entre les versions sans et avec le code d'instrumentation.

Pour évaluer l'empreinte mémoire du code de Trampoline due à l'instrumentation, il faut s'intéresser aux parties du code qui sont impactées par celle-ci :

— Le code des fonctions internes du noyau, pour lesquelles du code est ajouté de façon à générer les événements pour tracer les actions d'entrée et de sortie de la fonction.

— Le code du *System Call Handler*, pour lequel du code est ajouté de façon à générer tous les événements pour tracer les actions d'entrée et de sortie des différentes étapes et pour la gestion de l'identifiant unique du service appelé.

— L'ajout du tableau de correspondance des identifiants uniques des services.

Pour chacune de ces parties il est possible d'estimer le surcoût de l'empreinte mémoire sachant que :

— Comme on peut le voir sur le code assembleur de la Figure 4.15, la génération d'un événement nécessite 14 octets :
  — 6 octets pour le code de la génération d'un événement qui se compose de trois instructions de 2 octets qui permettent de : (1) charger l'adresse de destination du registre d'instrumentation, (2) charger la valeur correspondant à l'événement et (3) écrire la valeur à l'adresse du registre d'instrumentation correspondant.
  — 4 octets pour stocker l'adresse du registre d'instrumentation relatif aux événements.
  — 4 octets pour stocker la valeur de l'événement.

— Le code de la gestion de l'identifiant unique du service appelé nécessite 16 octets (voir le code de la Figure 4.13).

— L'identifiant unique d'un service est stocké sur 1 octet.

```
    ldr r3, [pc, #92] ; chargement de l'adresse␣du␣registre␣d'instumentation
    ldr r2, [pc, #92] ; chargement de la valeur de l'evenement
␣␣␣str␣r2,␣[r3,␣#0]␣␣;␣ecriture␣de␣la␣valeur
␣␣
```

FIGURE 4.15 – Code en assembleur pour la génération d'un événement

Au niveau des fonctions internes du noyau, on peut évaluer le coût de l'empreinte mémoire à 24 octets par fonction surveillée : 2 fois 6 octets pour la génération des événements

d'entrée et de sortie de la fonction, 4 octets pour l'adresse du registre d'instrumentation et 2 fois 4 octets pour la valeur des deux événements. Comme on surveille 20 fonctions, le coût est estimé à 480 octets.

Pour le *System Call Handler*, 8 événements sont générés, ce qui correspond à un ajout de 76 octets (comme les événements sont répartis sur deux registres d'instrumentation différents, il n'y a que 6 valeurs d'événements différents). En prenant en compte le code de gestion de l'identifiant unique du service appelé, cela fait un total de 92 octets.

Pour évaluer l'espace mémoire nécessaire au tableau de correspondance des identifiants uniques des services, on se place dans le cas où tous les services que l'on surveille sont ajoutés, ce qui donne un total de 23 services donc une utilisation de 23 octets.

Si on compare cette évaluation aux mesures présentées dans la Table 4.4 on remarque des différences, par exemple pour le code des fonctions internes 516 octets supplémentaires sont nécessaires alors que nous en avons estimé 480. Ces différences sont dues au code qui est généré pour prendre en compte le problème des alignements mémoires du fait que l'on a défini touts les champs des structures internes sur 32 bits pour des raisons de compatibilité avec le protocole APB.

	Taille du code	**Augmentation**
Fonctions internes	2140 octets	516 octets (24 %)
System Call Handler	108 octets	104 octets (96 %)
Tableau de correspondance de l'identifiant unique	-	23 octets
Code du noyau	6436 octets	643 octets (**10 %**)

TABLE 4.4 – Mesure de l'évaluation de l'empreinte mémoire du code d'exécution du noyau

Le surcoût de l'empreinte mémoire est donc de 10 %. La taille du noyau instrumenté reste très faible en comparaison de la taille de la mémoire ROM qui est de 256 Ko.

Évaluation de l'utilisation des ressources matérielles du FPGA

Parmi les ressources matérielles du FPGA qui sont utilisées pour l'implémentation du mécanisme de vérification en ligne, on peut distinguer :

Les ressources statiques qui sont utilisées pour vérifier les propriétés qui ne dépendent pas de l'application, c'est-à-dire les propriétés qui traitent de :

— la vérification du flot de contrôle des fonctions internes et du code du *System Call Handler*,
— la vérification de la cohérence de la structure *tpl_kern*.

Les ressources dépendantes de l'application qui permettent de vérifier les propriétés sur la cohérence des structures internes qui dépendent de l'application comme les descripteurs de tâches ou de ressources.

Pour évaluer les ressources statiques du mécanisme de vérification en ligne, on définit une architecture logicielle qui n'est composée d'aucune tâche ni ressource. Ainsi le circuit de vérification en ligne généré permet de vérifier uniquement les propriétés non-dépendantes de l'application. Dans cette configuration, le circuit de vérification en ligne analyse 100 formules ptLTL qui utilisent 75 propositions atomiques. On présente dans la Table 4.5 les ressources utilisées pour les différents éléments du mécanisme de vérification en ligne.

	Interface APB	**Identificateur d'état**	**Circuit de vérification**	**Notifieur d'erreur**	**Total**
Bascule D	849	738	905	300	2794 (**4.94 %**)
4-LUT	789	484	809	169	2431 (**4.30 %**)

TABLE 4.5 – Mesure de l'évaluation des ressources utilisées sur le FPGA pour la vérification des propriétés non-dépendantes de l'application

Comme on peut le voir le taux d'utilisation des ressources du FPGA de la SmartFusion2 MS060, qui dispose de 56 520 éléments logiques où un élément logique se compose d'une bascule D et d'une 4-LUT, est inférieur à 5 %.

Quand on prend en compte la vérification des propriétés dépendantes de l'application, plusieurs paramètres sont à prendre en compte pour évaluer le taux d'utilisation des ressources. Parmi ces paramètres on peut citer :

— le nombre de tâches et de ressources ajoutées à l'application ;
— la configuration des tâches ;
— les interactions entre les tâches et les ressources.

Ces paramètres vont avoir une incidence sur (1) la taille des différentes variables internes de Trampoline, (2) le nombre de propriétés à vérifier et (3) la complexité des circuits

permettant d'évaluer les différentes propositions atomiques. Pour chacun des ces trois cas nous allons donner un exemple :

1. Selon le nombre de tâches défini dans l'application, la nombre de bits nécessaires pour coder l'identifiant d'une tâche est différent par exemple dans le cas de trois tâches l'identifiant est codé sur 2 bits alors que pour 5 tâches l'identifiant sera codé sur 3 bits.
2. Le nombre de propriétés permettant de vérifier la cohérence de l'état d'une tâche varie selon que la tâche soit prête au démarrage ou encore que la tâche puisse attendre un événement, dans ce cas elle peut être dans l'état *waiting*.
3. Selon le nombre de ressources pouvant être prises par une tâche, la complexité de la proposition atomique qui permet de vérifier la cohérence du champ ressource du descripteur dynamique de la tâche sera différente.

Il est donc difficile de pouvoir évaluer de façon exhaustive l'utilisation des ressources selon la configuration de l'application. De ce fait, on propose d'utiliser une application concrète : l'étude de cas ROSACE [39] qui s'intéresse au modèle d'un contrôleur de vol longitudinal.

D'après la représentation de l'architecture logicielle de l'application de la Figure 4.16, on sait qu'elle se compose de 8 tâches périodiques dont 5 doivent s'activer toutes les 10 ms et 3 toutes les 20 ms. Pour fixer la priorité de ces tâches, on propose d'utiliser la politique d'ordonnancement rate-monotonic qui consiste à attribuer une plus forte priorité à la tâche qui possède la plus petite période. De plus, comme on peut le voir la tâche *altitude_hold* doit précéder la tâche *Vz_control*, de ce fait on fixe une plus forte priorité à la tâche *altitude_hold*. Dans la Table 4.6 on donne les résultats de l'utilisation des ressources du FPGA pour deux variantes de l'application :

— Dans la première version on considère que les variables globales de l'application n'ont pas besoin d'être protégées.
— Dans la seconde version on met en place une ressource de surveillance pour chaque variable globale.

À partir de ces résultats, on peut évaluer approximativement l'utilisation des ressources matérielles du FPGA pour l'ajout d'une tâche dans l'application et pour l'ajout d'une ressource, à partir des relations suivantes :

$$ressources\_fpga_{tache} = (ressources\_fpga_{appliV1} - ressources\_fpga_{base})/nb\_tache$$

$$ressources\_fpga_{ressource} = (ressources\_fpga_{appliV2} - ressources\_fpga_{appliV1})/nb\_ressource$$

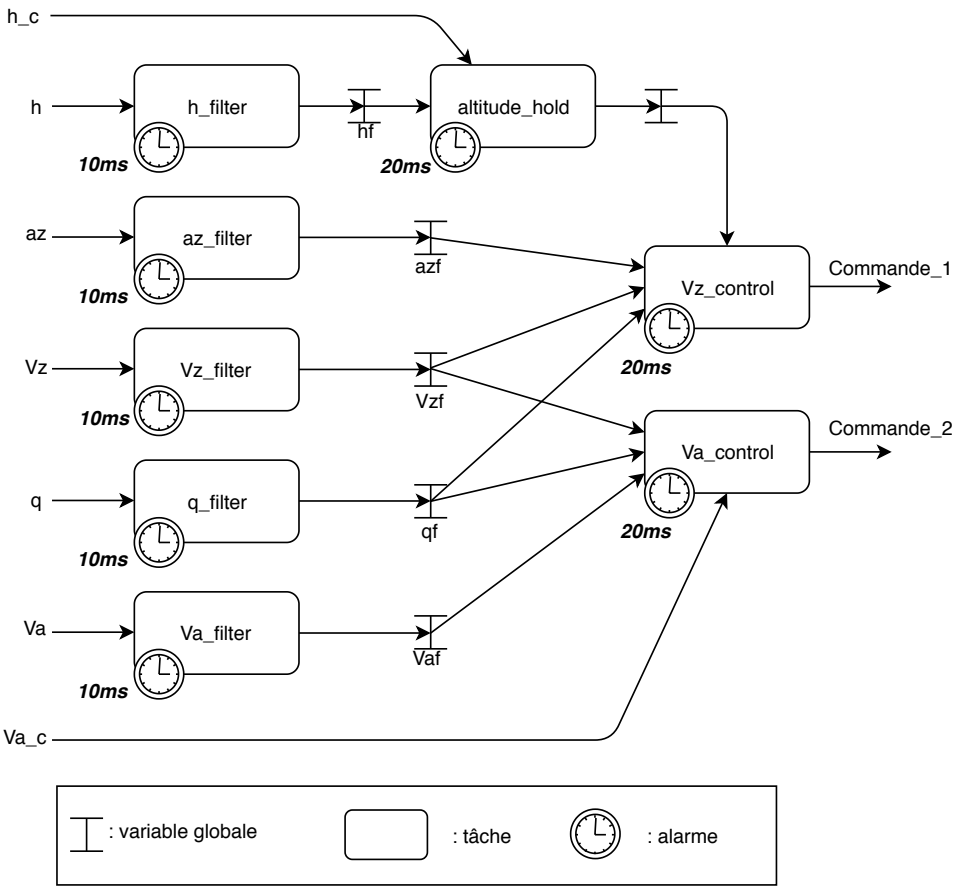

FIGURE 4.16 – Représentation de l'architecture logicielle de l'étude de cas ROSACE

	Bascule D	**4-LUT**	**Utilisation du FPGA**
Version 1 : sans protection des variables	7297	6269	12.9 %
Version 2 : avec protection par ressources	9520	8205	16.8 %

TABLE 4.6 – Mesure de l'évaluation des ressources utilisées sur le FPGA pour l'étude de cas ROSACE

Où $ressources\_fpga_{appliV1}$ et $ressources\_fpga_{appliV2}$ représentent respectivement les ressources utilisées pour la version 1 et 2 de l'étude de cas ROSACE, $ressources\_fpga_{base}$ représente les ressources utilisées pour la partie non dépendante de l'application, nb_tache représente le nombre de tâches et $nb\_ressource$ représente le nombre de ressources. Les résultats sont présentés dans la Table 4.7. Notons que le nombre de tâches est de 9, en

effet dans Trampoline une tâche de fond est automatiquement ajoutée.

	Bascule D	**4-LUT**
Tâche	500.33	426.44
Ressource	370.5	322.66

TABLE 4.7 – Estimation du coût en ressource sur le FPGA de l'ajout d'une tâche ou d'une ressource

4.4 Conclusion

Comme on a pu le constater dans ce chapitre la spécification de propriétés sur l'exécution du code du noyau d'un SETR nécessite une analyse approfondie du code de ce dernier. De la même façon l'instrumentation du code ne peut pas se faire de façon automatique. Cela met en évidence que la mise en œuvre d'un dispositif de vérification en ligne d'un programme doit se faire au plus tôt lors de la conception du programme.

Des compromis sur la vérification ont été pris afin de limiter l'impact de la mise en œuvre de mécanisme de vérification. Dans le prochain chapitre, une évaluation par injection de fautes est présentée.

Chapitre 5

Évaluation par injection de fautes

Dans ce chapitre, on cherche à évaluer le taux de détection d'erreurs du mécanisme de vérification en ligne que l'on propose, dans le cadre de la surveillance de l'exécution du code noyau du SETR Trampoline. Pour cela une campagne d'injection de fautes est mise en place.

Le principe d'une campagne d'injection de fautes est d'exécuter plusieurs fois une application en y injectant une faute à chaque fois afin de classer les effets de celle-ci. Pour l'évaluation d'un mécanisme de détection, il faut comparer les résultats de la campagne d'injection de fautes dans le cas où l'on ne prend pas en compte le mécanisme avec ceux où on le prend en compte.

Pour commencer, on présentera l'environnement d'injection de fautes, c'est-à-dire le modèle de faute que l'on considère, la technique d'injection de fautes choisie et la plateforme d'injection de fautes développée. Puis on expliquera le processus d'injection de fautes. Ensuite on définira l'application qui est utilisée pour réaliser la campagne. Enfin on donnera les résultats de l'injection.

5.1 Présentation du processus d'injection de fautes

5.1.1 Contexte et objectif

Les fautes que l'on considère dans notre étude sont les fautes transitoires, qui se traduisent par le changement de l'état logique d'une ou plusieurs cellules mémoires du système, et qui impactent l'exécution du code noyau du SETR Trampoline. Comme proposé dans les travaux [51, 5], on peut classer ces fautes selon 4 catégories :

Pas d'impact. La faute n'a pas affecté l'exécution du programme : celui-ci se termine normalement.

Exception matérielle. La faute a produit une erreur qui a été détectée comme une exception matérielle par les mécanismes déjà présents dans le processeur.

Hors-délai. La faute a produit une erreur qui empêche le programme de se terminer dans les temps. Un mécanisme de *watchdog* est utilisé afin de détecter ce type d'erreur.

Corruption de donnée (SDC). Le programme se termine mais son résultat n'est pas correct.

Ainsi, pour évaluer les performances d'un dispositif de détection, il faut être capable de déterminer le taux de détection des fautes pour ces différentes classes. Pour cela, on doit :

1. Classifier l'impact des fautes sur le système sans mécanisme de détection.
2. Compter le nombre de fautes détectées avec le dispositif de détection dans les différentes catégories définies lors de la classification.

Les fautes que l'on cherche à détecter en priorité sont les SDCs. Ces dernières sont les seules à ne pas être détectées par ailleurs. Au niveau de l'exécution de l'application, une corruption de donnée va se traduire par l'altération de l'ordonnancement attendu de l'application.

Comme on peut le voir sur la Figure 5.1, notre système se compose de plusieurs types de mémoires. Celles-ci n'ont pas la même sensibilité face aux fautes transitoires :

eNVM et Flash : Les mémoires non volatiles sont généralement peu sensibles aux fautes transitoires. C'est notamment le cas sur la carte d'évaluation considérée qui inclut en plus des mécanismes de protection.

SRAM : La mémoire vive statique est sensible aux fautes transitoires. Néanmoins le circuit de vérification est protégé par la triplication qui est mise en place, et

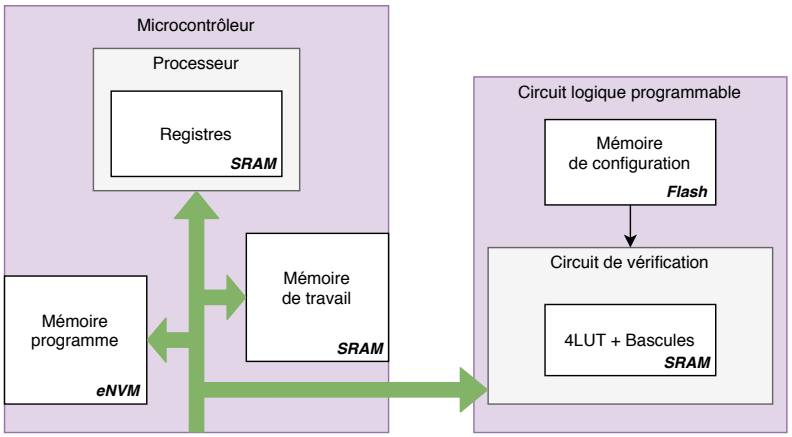

FIGURE 5.1 – Hiérarchie mémoire de la SmartFusion2

la mémoire de travail est protégée par un mécanisme de détection et correction d'erreurs intégré à la carte.

La seule mémoire qui n'est pas protégée est celle des registres du processeur.

L'évaluation des performances de détection est donc réalisée à partir d'une campagne d'injection de fautes au niveau des registres du processeur. Le principe est d'exécuter en boucle une application de référence où pour chaque itération une faute est injectée.

5.1.2 Développement d'une plateforme d'injection de fautes

Il existe plusieurs approches d'injection de fautes permettant de viser les registres du processeur [66] :

— Les approches matérielles où l'injection de fautes se fait directement sur le système. Ces approches requièrent des infrastructures coûteuses qui ne sont pas adaptées à nos travaux.

— Les approches par simulation où l'injection de fautes se fait à partir d'un modèle du système qui est simulé par un simulateur. Ces approches nécessitent l'utilisation d'un modèle HDL du système dont on ne dispose pas.

— Les approches par émulation où l'injection de fautes se fait à partir d'un modèle du système qui est émulé par un émulateur. De la même façon que pour la simulation, on ne dispose pas de modèle adéquat.

La technique d'injection de fautes que l'on utilise se classe dans les approches logicielles, c'est-à-dire que l'injection est réalisée au niveau du logiciel. Cela peut se faire de deux façons :

1. Par l'ajout d'instructions dans le code qui vont modifier aux instants choisis la valeur des registres voulus.
2. Par le contrôle du processus à l'aide d'une sonde de débogage qui va arrêter le programme aux instants choisis afin de modifier la valeur des registres voulus.

Dans la première méthode le comportement temporel du programme est peu affecté par rapport à la seconde approche où le code de l'exécution est stoppé. Pour la première approche, le code de l'application est altéré par rapport à sa version initiale. Donc une même faute peut ne pas avoir la même incidence que dans la réalité. De ce fait, on utilisera l'approche par utilisation d'une sonde de débogage.

Des outils déjà existants comme [57, 52] permettent de réaliser des campagnes d'injection de fautes, cependant ces outils ne sont pas directement compatibles avec notre système. Ainsi nous avons développé notre propre plateforme, cela permet en plus d'avoir une maîtrise complète du processus.

La plateforme d'injection de fautes que nous utilisons est présentée sur la Figure 5.2 Elle se compose d'une machine hôte, du système ciblé, la SmartFusion 2, et d'une sonde de débogage J-Link. La machine hôte contrôle le processus d'injection de fautes à partir d'une interface GDB. L'interfaçage entre la sonde de débogage et la Smartfusion 2 se fait à partir d'un port JTAG.

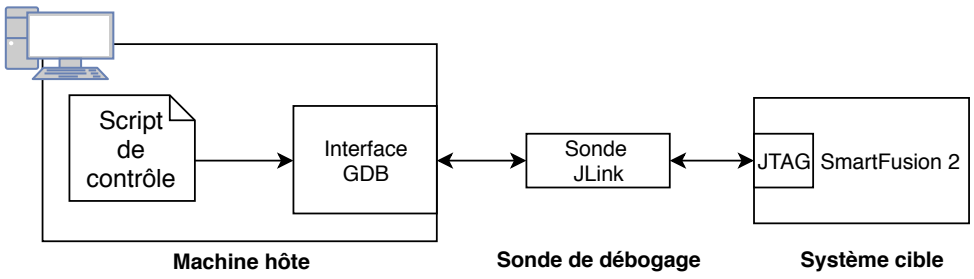

FIGURE 5.2 – Présentation de la plateforme d'injection de fautes

5.1.3 Sélection de la liste des fautes à injecter

Lors de la mise en œuvre d'une campagne d'injection de fautes, il est nécessaire de définir le modèle de faute que l'on utilise. C'est à partir de celui-ci que l'on peut définir la liste des fautes à injecter.

Un modèle de fautes est une abstraction permettant de modéliser l'impact d'un certain type de fautes. Les fautes que l'on cherche à modéliser sont les fautes transitoires. Comme

présenté dans le chapitre 1, il existe plusieurs modèles de fautes permettant de modéliser ce type de fautes : les SEUs (*Single-Event Upset*), les MBUs (*Multiple-Bit Upset*) et les MCUs (*Multiple-Cell Upset*).

Selon le modèle de fautes choisies on peut définir l'espace de fautes qui est une représentation de toutes les fautes pouvant être injectées. L'espace de fautes peut être défini par le couple $< date, localisation >$ [57] où :

— La *date* représente les instants où l'on considère qu'une faute peut se produire.

— La *localisation* représente l'emplacement des ressources où peuvent se produire une faute. Dans notre cas une ressource correspond à un ou plusieurs bits des registres accessibles du processeur.

Le nombre de fautes possibles pouvant affecter un système est défini comme la combinaison des éléments de l'espace de fautes.

Cet espace de fautes est dépendant du modèle de fautes utilisé. Afin de réduire le nombre de fautes possibles pouvant impacter le système, nous considérons le cas du modèle SEU. Pour appuyer ce choix, on peut se référer aux travaux [5, 50] où les auteurs montrent que l'utilisation de bit-flip simple permet de couvrir 98% des SDCs incluant les bit-flip doubles (MBU avec $n = 2$).

Cependant même dans le cas de l'utilisation du modèle SEU, l'espace de faute est généralement trop grand pour réaliser une campagne d'injection de fautes exhaustive. De ce fait, deux approches peuvent être envisagées :

Approche aléatoire (Figure 5.3) qui consiste à « tirer au sort » un certain nombre de fautes de l'espace de faute, puis d'injecter ces fautes. Le nombre de fautes sélectionnées doit être évidement assez grand pour que les résultats obtenus soient significatifs.

Approche par optimisation (Figure 5.4) qui consiste à optimiser l'espace de faute en ne prenant pas en compte les fautes pour lesquelles on sait qu'elles ne produiront pas d'erreurs. Par exemple, il n'est pas nécessaire d'injecter des fautes au niveau des adresses mémoires qui ne sont pas utilisées par le programme. De même à partir du moment où une ressource n'est plus utilisée par le programme, l'injection d'une faute sur celle-ci n'aura pas d'impact.

L'approche aléatoire est particulièrement adaptée lorsqu'il est nécessaire de calculer rapidement le taux de détection d'un composant, par exemple pour comparer plusieurs mécanismes de détection. Elle permet d'obtenir des résultats proches de la réalité d'une point de vue statistique. Dans notre cas, ce qui nous intéresse c'est d'être capable d'ana-

lyser finement les différentes fautes afin d'être capable d'indiquer les limites de notre mécanisme et de proposer des pistes d'améliorations. L'approche utilisée est donc celle par optimisation.

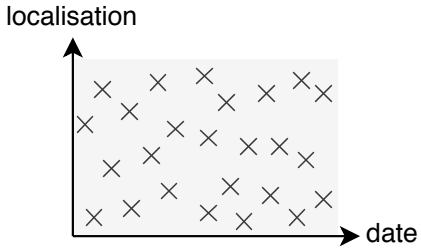

FIGURE 5.3 – Représentation de l'approche aléatoire. Une croix représente une faute tirée au sort.

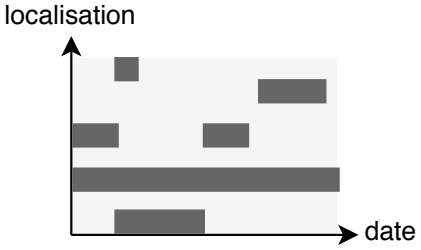

FIGURE 5.4 – Représentation de l'approche par optimisation. Les zones en gris foncées représente l'espace de fautes considéré.

La technique d'optimisation de l'espace de faute que l'on utilise est *Inject-On-Read* [51] qui consiste à cibler uniquement les fautes qui peuvent avoir un impact sur l'exécution. Pour cela, il faut injecter une faute sur une ressource uniquement avant que celle-ci ne soit lue. Ici le temps est approximé à l'instruction machine. Le principe de fonctionnement est le suivant :

- Pour chaque ressource on définit ses intervalles d'expositions qui sont délimités par les accès écriture / lecture et lecture / lecture. En effet le reste du temps la ressource est obsolète, c'est-à-dire que l'injection d'une faute n'aura forcement aucun impact.
- Pour chaque intervalle on injecte les fautes à une seule date. En effet l'impact d'une même faute sur un intervalle d'exposition sera identique pour chacune des dates.
- Afin de prendre en compte la durée d'exposition des fautes, un facteur de poids est associé à chacun des intervalles d'exposition. Ce facteur de poids correspond au nombre d'instructions machines exécutées pendant un intervalle d'exposition.

Pour illustrer cela, prenons le code assembleur de la Figure 5.5 où l'on veut déterminer les dates d'injections de fautes pour la ressource $r0$.

Les accès en écriture et en lecture de la ressource $r0$ sont représentés sur la Figure 5.6. Comme on peut le voir on peut définir trois intervalles d'exposition et pour chacun d'eux on associe un facteur de poids (noté FP), qui correspond au nombre d'instructions machines d'un intervalle d'exposition, et une date d'injection qui correspond à l'instruction

```
1    ...
2    ...
3    ldr r0, [r4]   ; r0 <- mem[r4]  -- écriture de r0
4    ...
5    ...
6    add r2, r0, r1 ; r2 <- r0 + r1  -- lecture de r0
7    ...
8    mov r0, #val1  ; r0 <- val1     -- écriture de r0
9    ...
10   ...
11   ldr r1, [r0]   ; r1 <- mem[r0]  -- lecture de r0
12   ...
13   str r1, [r0]   ; mem[r0] <- r1  -- lecture de r0
```

FIGURE 5.5 – Représentation de l'approche par optimisation. Les zones en gris foncées représente l'espace de fautes considéré.

précédent la fin d'un intervalle (indiqué par un flèche vers le bas). Ici les intervalles d'exposition sont :

- Entre les instructions 4 et 6, où l'intervalle est délimité par un accès écriture/lecture.
- Entre les instructions 9 et 11, où l'intervalle est délimité par un accès écriture/lecture.
- Entre les instructions 12 et 13, où l'intervalle est délimité par un accès lecture/lecture.

Pour cet exemple, on remarque qu'en ciblant le registre $r0$, 96 fautes doivent être injectées (pour les 3 dates d'injection on injecte une faute sur les 32 bits du registre $r0$). Celles-ci permettent de simuler 256 injections de fautes (avec la prise en compte du facteur de poids : (3+3+2) × 32), sachant que l'espace de fautes total est de 448 possibilités (14 instructions × 1 ressource × 32 bits) où la différence entre l'espace de fautes et le nombre de fautes simulées, soit ici 192, correspond à des fautes qui n'ont pas d'impact.

5.1.4 Description du processus d'injection de fautes

Le processus d'injection de fautes est décrit sur la Figure 5.7. Il se compose de 6 étapes regroupées en 3 phases :

La phase de pre-injection qui a pour objectif d'analyser l'exécution de l'applica-

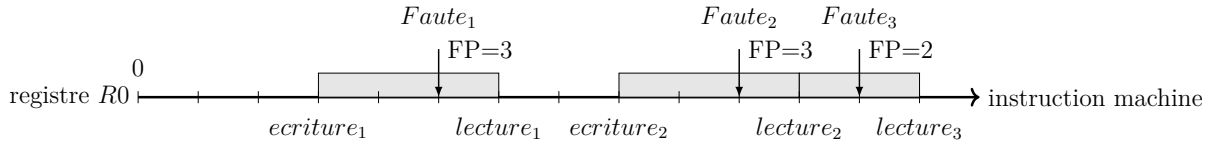

FIGURE 5.6 – Représentation de la stratégie *Inject-on-read* en prenant en compte le temps d'exposition aux fautes d'une ressource (représentée par les rectangles gris).

tion (trace de référence) afin d'en extraire les informations nécessaires à l'injection de fautes.

La phase d'injection qui permet d'injecter les fautes définies en entrée.

La phase de post-injection qui s'occupe de l'analyse des résultats de la campagne d'injection de fautes.

Les différentes étapes du processus sont basées sur l'utilisation de scripts, écrits en python, que nous avons développés.

Phase 1 : L'analyse du code - Pre-Injection

L'objectif de la première phase est de définir la liste des fautes à injecter. On caractérise une faute par sa date, qui est donnée en numéro d'instruction, et par sa localisation, c'est-à-dire le numéro du bit du registre ciblé.

Lors de l'injection d'une faute, pour s'arrêter à un emplacement spécifique du code il faudra placer un point d'arrêt à l'adresse de l'instruction correspondante et s'y arrêter le bon nombre de fois. Le but de l'analyse de la trace (*étape 1 de la Figure 5.7*) est de créer un tableau de correspondance entre le numéro d'une instruction et l'adresse de l'instruction avec son occurrence. Pour cela, un script exécute en pas-à-pas les instructions et pour chacune d'elles il récupère la valeur de *pc* (le pointeur d'instruction) et calcule l'occurrence. Ces informations sont stockées dans un fichier afin d'être utilisées lors de l'injection.

Dans la campagne d'injection de fautes que l'on réalise, on ne cherche pas à injecter des fautes à chaque instant mais uniquement lors du l'exécution du code du SETR Trampoline. Il faut donc extraire la liste des instructions ne s'exécutant pas en mode noyau (*étape 2 de la Figure 5.7*) afin de ne pas les prendre en compte lors de la génération de la liste des fautes. Pour cela, on exécute le code en pas-à-pas et on vérifie si l'adresse de l'instruction correspond à celle de l'entrée ou sortie du mode noyau. Les numéros des instructions du

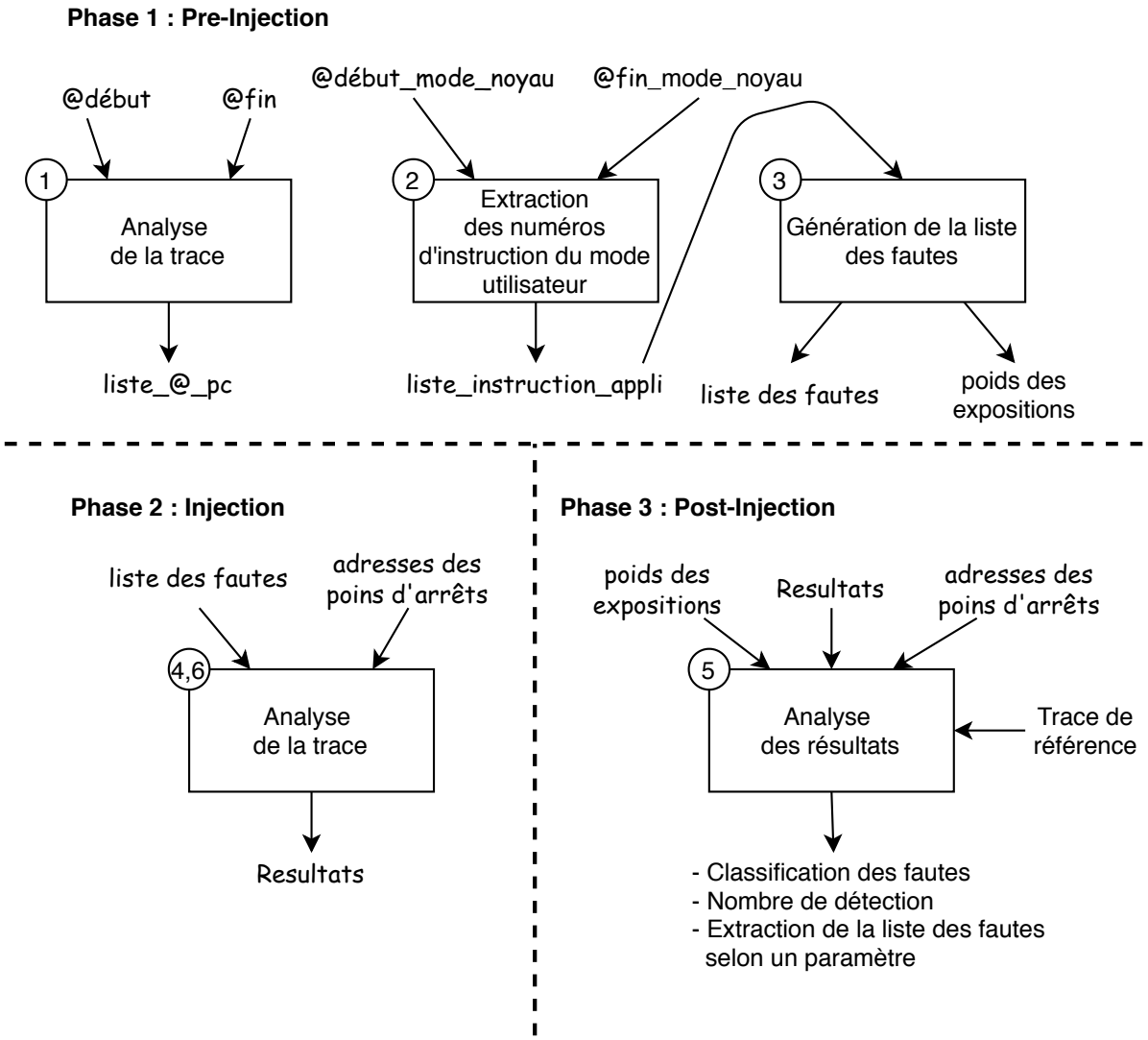

FIGURE 5.7 – Flot du processus d'injection de fautes

mode utilisateur sont ensuite stockés dans un fichier afin d'être pris en compte lors de la génération des fautes.

La génération de la liste des fautes à injecter (*étape 3 de la Figure 5.7*) est réalisée en implémentant l'algorithme correspondant à la stratégie *Inject-On-Read*. Le principe est d'exécuter en mode pas-à-pas le code afin de déterminer pour chacune des instructions les registres qui sont lus et écris. Cela se traduit par la mise à jour d'une première liste qui contient les dates de lecture des différents registres et d'une seconde liste qui contient les dates d'écriture des différents registres. En prenant en compte la liste des instructions du mode utilisateur, il est ensuite possible de générer directement à partir de la liste des dates

de lecture des registres la liste des fautes et à partir des deux listes le poids d'exposition de chacune des fautes.

Phase 2 : L'injection des fautes

Le principe d'injection d'une faute est le suivant :

1. Le programme est exécuté jusqu'au moment où la faute doit être injectée. Pour cela, on associe à une faute l'adresse mémoire de l'instruction pour laquelle elle doit être injectée et son occurrence (une même adresse mémoire peut être utilisée plusieurs fois). Un point d'arrêt est placé à l'adresse mémoire correspondante, puis le programme est exécuté jusqu'à atteindre le bon nombre de fois le point d'arrêt.
2. La faute est injectée. Pour cela il faut :
   — Lire la valeur du registre sur lequel se trouve la faute à injecter.
   — Calculer la valeur que doit avoir le registre afin de simuler la faute voulue.
   — Modifier la valeur du registre avec la valeur calculée.
3. Le programme s'exécute jusqu'à sa terminaison en plaçant des points d'arrêts aux adresses de terminaison possibles. Les adresses de terminaison peuvent être :
   — L'adresse de la fin normale de l'application. Cela implique que l'application est bornée.
   — L'adresse de l'une des exceptions matérielles.

Comme on peut le voir sur la Figure 5.7, la phase d'injection de fautes est réalisée deux fois.

— La première (*étape 4*) correspond à l'injection de l'ensemble des fautes définies précédemment.
— La seconde (*étape 6*) correspond à la re-injection des fautes qui ont été détectées lors de la première phase. La différence ici est que l'adresse de la fin d'exécution correspond à l'adresse de sortie du code en mode noyau et que ce point d'arrêt est placé au moment de l'injection de la faute. Cela permet d'estimer les fautes qui ont été détectées lors de l'appel du service durant lequel la faute se produit, c'est-à-dire les fautes que l'on est capable de confiner afin qu'elles ne se propagent pas au niveau utilisateur.

Pour chaque injection de faute, il faut stocker les informations nécessaires pour la phase d'analyse des résultats. Ces informations correspondent entre autres, à l'état des moniteurs, à l'adresse de la terminaison du programme et à la trace d'exécution du programme.

Plus de détails seront donnés dans la suite sur la façon dont ces informations sont récupérées.

Phase 3 : L'analyse des résultats

Classification de l'impact des fautes La classification est réalisée à partir de l'analyse de la valeur de l'adresse de terminaison du programme et de la trace d'exécution qui sont récupérées pendant la campagne d'injection de fautes comme décrit sur la Figure 5.8.

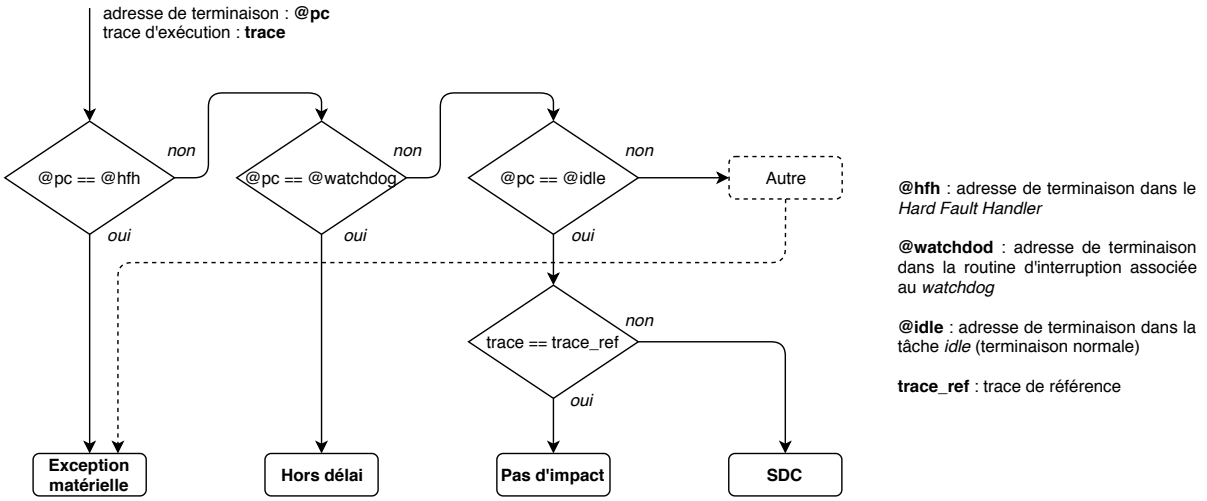

FIGURE 5.8 – Classification des fautes

On retrouve bien les catégories définies précédemment ainsi que la catégorie *Autre* qui permet de prendre en compte les fautes pour lesquelles l'adresse de terminaison du programme ne correspond à aucune adresse attendue.

Aussi la catégorie « pas d'impact » est associée aux fautes qui n'ont pas de répercussions sur l'ordonnancement. Ces fautes peuvent donc avoir un impact sur l'application.

Analyse de la détection des erreurs La détection des erreurs est réalisée à partir de l'analyse de l'état des moniteurs pour les différentes fautes injectées. Cette analyse permettra de déterminer la couverture de détection des fautes pour les différentes classes de faute, ainsi que de déterminer ce que les différents moniteurs permettent de détecter.

Extraction de la liste des fautes selon un paramètre Pour réaliser des analyses plus fines il est possible d'extraire les fautes qui ont un certain impact. Par exemple pour

réaliser la seconde campagne d'injection de fautes, on doit extraire des résultats la liste des fautes qui ont été détectées.

5.2 Développement d'une application de référence

5.2.1 Description de l'application

L'application utilisée pendant la campagne d'injection de fautes doit respecter les exigences suivantes :

1. *Être déterministe :* pour une faute donnée, le résultat de l'injection doit toujours être le même.
2. *Être bornée :* le début et les fins possibles de l'application doivent être clairement identifiés.
3. *Couvrir suffisamment le code du noyau :* l'application doit permettre de passer dans les différentes branches du code du noyau.

Pour respecter la première exigence, il faut que l'application soit indépendante de facteurs externes.

Pour la seconde exigence, on définit le début de l'exécution comme le démarrage de l'exécutif, c'est-à-dire l'appel du service $StartOS()$ dans la fonction $main()$.

Enfin pour assurer que toutes les instructions pertinentes du code du noyau sont exécutées, l'application doit être conçue afin d'appeler tous les services de Trampoline que l'on considère. De plus, certains services doivent être appelés plusieurs fois afin de prendre en compte le cas où l'appel provoque un ré-ordonnancement de l'application ou non. La liste des services et de leurs effets possibles est donnée par la Table 5.1.

La figure 5.9 présente le scénario de l'exécution de l'application de référence que l'on propose. Cette application se compose de 6 tâches ($T1$ à $T6$) avec l'ordre de priorité suivant :

$$P(T3) < P(T1) < P(T2) < P(T4) < P(T5) < P(T6)$$

Toutes les tâches, sauf $T4$, sont préemptables. Et uniquement $T1$ est configurée pour être activée au démarrage de l'exécutif. Une ressource $r1$ est partagée entre $T1$ et $T3$ et deux événements sont associés à $T1$.

Notons que l'application développée présente des biais qu'il faudra considérer lors de la conclusion de l'analyse des résultats :

1. L'application n'est pas réaliste par rapport aux applications que l'on peut retrouver dans un système industriel.

Service	Effet	ref
ActivateTask	pas de préemption préemption	(a) (b)
ChainTask	préemption	(c)
TerminateTask	préemption	(d)
Schedule	pas de préemption préemption	(e) (f)
SetEvent	pas de préemption préemption	(g) (h)
WaitEvent	pas de préemption préemption	(i) (j)
ClearEvent *clear*	pas de préemption	(k)
GetResource	pas de préemption	(l)
ReleaseResource *RelR*	no context change context change	(m) (n)

TABLE 5.1 – Liste des appels de service et de leurs effets. La troisième colonne *ref* indique ces différents cas sur le diagramme de Gantt de la figure 5.9

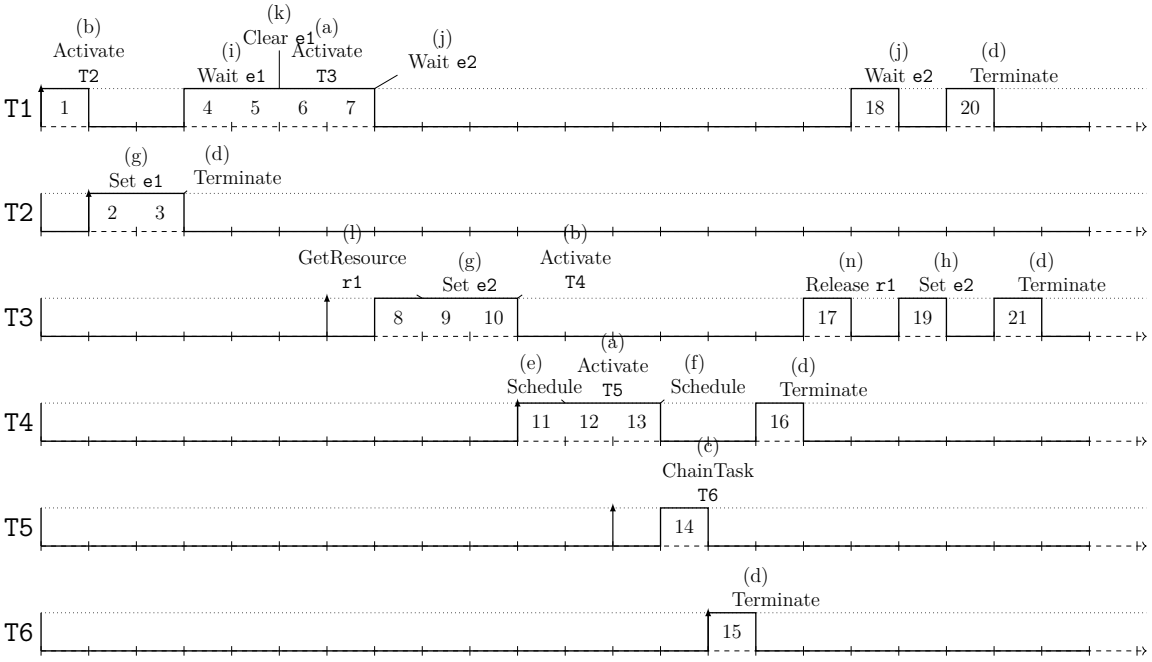

FIGURE 5.9 – Diagramme de Gantt du scénario de l'application de référence qui est utilisé pour la campagne d'injection de faute.

2. L'application n'intègre pas d'appels de services liés à l'utilisation des horloges ou des interruptions. Cela est dû au fait que lors de l'exécution en pas-à-pas du code, qui est réalisée pendant l'analyse du code, les interruptions sont désactivées.

5.2.2 Récupération des résultats

Le protocole d'expérimentation que l'on met en place doit permettre de récupérer les informations nécessaires pour la classification de l'impact des fautes et pour savoir si une erreur a été détectée ou pas par le mécanisme de vérification en ligne.

La classification de l'impact des fautes nécessite de connaître l'adresse de terminaison et la trace d'exécution du programme :

— L'adresse de terminaison est récupérée à partir de la lecture de la valeur de *pc* via la sonde de débogage quand le programme se termine (un point d'arrêt matériel est associé aux différentes terminaisons possibles).
— La trace d'exécution est générée à partir de l'instrumentation du code des tâches de l'application. Avant chaque appel de service, la valeur correspondant à l'ordre d'exécution de référence est ajoutée au tableau *tab_trace[]*. Prenons l'exemple de l'application suivante où l'on définit deux tâches :

```
TASK (T1) {
    ActivateTask(T2);
    TerminateTask();
};
```

```
TASK (T2) {

    TerminateTask();
};
```

— *T1* est une tâche qui est configurée pour être préemptable et démarrée lors du démarrage de l'exécutif.
— *T2* est une tâche qui est configurée avec une plus forte priorité que la tâche *T1*.

Le code de ces tâches est instrumenté de façon à correspondre à la trace d'exécution qui est présentée sur la Figure 5.10.

```
volatile int tab_trace[3] = {0,0,0};
int index = 0;

TASK (T1) {
    tab_trace[index] = 1;
    index ++;
    ActivateTask(T2);
    tab_trace[index] = 3;
    index ++;
    TerminateTask();
};
```

```
TASK (T2) {
    tab_trace[index] = 2;
    index ++;
    TerminateTask();
};
```

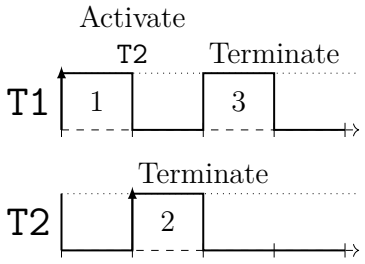

FIGURE 5.10 – Diagramme de Gantt de l'exemple avec la trace d'exécution

Pour savoir si une faute est détectée par le mécanisme de vérification en ligne, on associe le signal d'interruption du circuit de vérification en ligne à une routine d'interruption. Celle-ci stocke dans des variables globales la valeur des registres du FPGA qui représentent l'état des différents moniteurs. Puis l'interruption se désactive, de cette façon on ne garde en mémoire que l'état des moniteurs lors de la première détection.

Il est important de noter que les informations stockées en mémoire peuvent être corrompues. En effet, une faute peut, par effet de bord, modifier les données d'observation. Pour se prémunir de cela, toutes ces données sont triplées, ce qui permet de corriger les erreurs.

Aussi, afin de prendre en compte des fautes qui ont une incidence tardive, ce scénario est exécuté 3 fois de suite (l'injection de fautes ne se fait que lors de la première exécution). Pour cela, une septième tâche $T7$ est ajoutée à l'application de la Figure 5.9. Cette tâche est configurée pour être prête au démarrage et a la priorité la plus faible. Ainsi elle n'est appelée pour la première fois qu'à la fin de l'exécution du scénario. Son rôle est d'activer deux fois la tâche $T1$ avant de se terminer. Cela permet bien d'exécuter le scénario trois fois de suite.

5.2.3 Caractéristiques de l'application

Les caractéristiques de l'application sont résumées dans la Table 5.2.

Nombre d'instructions machines composant la trace d'exécution	14 820
Nombre d'instructions machines en mode noyau	13 782
Espace de fautes total : 16 registres × 32 bits × 13 782 instructions	7 056 384
Espace de fautes ciblé	3 117 824
Nombre d'injection de fautes	892 320

TABLE 5.2 – Caractéristiques de la campagne d'injection de fautes

Comme l'application de référence réalise principalement des appels de service, 93 % des instructions machines qui sont exécutées se font en mode noyau. L'espace de fautes total correspond donc au produit du nombre d'instructions machine exécutées en mode noyau par les 32 bits des registres ciblés par la campagne. L'espace de fautes ciblé correspond à l'espace de fautes calculé en utilisant la stratégie *Inject-On-Read*. De même l'utilisation de la stratégie *Inject-On-Read* permet de déterminer le nombre de fautes à injecter.

L'architecture ARM dispose de 16 registres de 32 bits :

- Les registres $r0$ à $r12$ sont les registres généraux, que l'on notera `GPR` dans la suite du manuscrit. Ils sont principalement utilisés pour effectuer les calculs.
- SP est le pointeur de pile. Il pointe sur l'adresse du sommet de la pile.
- LR est le registre de liens. Il permet de stocker l'adresse du dernier appel de fonction.
- PC est le pointeur d'instruction. Il pointe sur l'adresse de la prochaine instruction à exécuter.

La Table 5.3 regroupe les informations sur les fautes injectées pendant la campagne d'injection de fautes, où pour chaque type de registre, on donne le nombre de fautes à injecter, l'espace de fautes ciblé et les informations statistiques sur les facteurs de poids qui permettent de représenter le temps d'exposition en nombre d'instruction des différentes fautes.

Pour les différents types de registres on peut remarquer que :

Pour le registre PC : le nombre de fautes injectées et l'espace de faute représenté est identique, ce qui est normal puisque la valeur de PC est modifiée à chaque

	Nombre de fautes injectées	**Espace de fautes**	**Informations statistiques du facteur de poids**			
			Moyenne	**min**	**max**	**Écart-type**
GPR	400 288	2 020 992	5.0	1	1627	56.1
SP	40 480	238 304	5.9	1	59	10.4
LR	10 528	417 504	39.7	2	132	33.3
PC	441 024	441 024	1	1	1	0

TABLE 5.3 – Caractéristiques de la campagne d'injection

instruction. De même l'espace de faute correspond bien au nombre d'instruction machine en mode noyau fois les 32 bits du registre.

Le registre LR est utilisé à chaque appel de fonction.

Le registre SP est utilisé en début de fonction pour empiler sur la pile système le registre $R7$ (*frame pointer*) et la valeur du registre LR, et en fin de fonction pour dépiler ces valeurs dans $R7$ et PC.

Pour les registres généraux (GPR) : l'espace de fautes et le nombre de fautes à injecter dépend de leur utilisation, comme on peut le voir sur la Figure 5.11. En analysant le code assembleur et en étudiant l'ABI (Application Binary Interface), on remarque que :

- $r0$ est utilisé pour le passage de paramètres lors des appels de fonctions, ainsi que pour la valeur de retour d'une fonction.
- $r1$ à $r5$ sont utilisés pour la manipulation des données, $r1$, $r2$ et $r3$ sont aussi utilisés pour le passage de paramètres.
- $r7$ est utilisé comme *frame pointer*.
- $r6$ et $r8$ à $r11$ ne sont pas utilisés par le noyau. Leur utilisation est alors limité aux chargement et à la sauvegarde de contexte, ceci explique pourquoi l'espace de fautes associé à ces registres est grand alors que le nombre de fautes injecté est faible.
- $r12$ n'est pas utilisées par le code du noyau et est empilé automatiquement lors du passage en mode noyau.

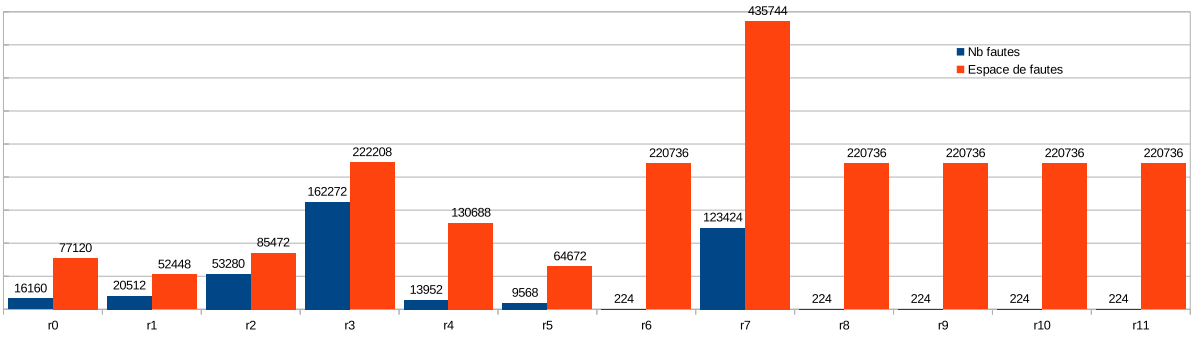

FIGURE 5.11 – Nombre de faute injecté et espace de fautes des registres GPR

5.3 Analyse de la campagne d'injection de fautes

5.3.1 Classification de l'impact des fautes

Présentation des résultats

Les résultats de la classification de l'impact des fautes sont présentés sur la Table 5.4.

	Pas d'impact	**Exception mat.**	**Hors délai**	**SDC**	**Autre**
GPR	1 495 534 74%	456 528 22.6%	3 190 0.2%	65 739 3.2%	1 <0.1%
SP	94 875 39.8%	139 665 58.6%	1 699 0.7%	2 065 0.9%	0 0%
LR	39 677 9.5%	353 638 84.7%	8 545 2%	15 644 3.8%	0 0%
PC	54 096 12.3%	367 363 83.3%	2 220 0.5%	17 231 3.9%	114 0%
Total	1 684 158 54%	1 317 194 42.3%	15 654 0.5%	100 679 3.2%	115 <0.1%

TABLE 5.4 – Classification de l'impact des fautes sur l'espace de fautes ciblé

Comme on peut le voir, on retrouve bien les catégories définies dans la section 5.1.4 En ce qui concerne les fautes de la catégorie *Autre*, en reproduisant manuellement ces fautes, on observe que le programme se termine au niveau du *HardFault_Handler*. On peut supposer que ce phénomène est dû à un bug de la sonde de débogage. Dans la suite, on classera ces fautes comme des exceptions matérielles.

Ces résultats montrent que *46* % des fautes produisent une erreur. Ce fort pourcentage d'erreur vient du fait que l'on fait l'étude sur l'espace de fautes visées. Si on ramène ces résultats à l'espace de fautes total (les fautes qui ne sont pas ciblées par l'injection sont classées dans la catégorie « pas d'impact ») on obtient les résultats de la Table 5.5. On remarque alors que *20.3* % des fautes produisent une erreur et que *1.4*% des fautes ne sont pas détectées par un mécanisme présent de base dans le processeur.

	Pas d'impact	**Exception mat.**	**Hors délai**	**SDC**
Total	5 622 742 79.7%	1 317 309 18.7%	15 654 0.2%	100 679 1.4%

TABLE 5.5 – Classification de l'impact des fautes sur l'espace de fautes total

Analyse des résultats

Afin de mieux comprendre ces résultats, on s'intéresse à l'impact des fautes pour les différentes types de registres.

Les registres PC **et** LR : Ces deux registres contiennent exclusivement des valeurs correspondant aux adresses de la zone mémoire qui correspond au code, soit à la mémoire ROM. Pour la SmartFusion 2, la plage d'adressage de la mémoire ROM est :

$$@ROM = [0x00000; 0x3FFFF]$$

Et la plage d'adressage spécifique à notre application est :

$$@ROM_{application} = [0x00000; 0x3364]$$

Comme montré sur la Figure 5.12, on peut diviser les 32 bits contenus par PC et LR en quatre zones :

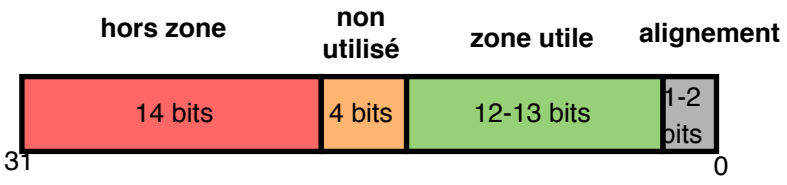

FIGURE 5.12 – Segmentation d'une adresse en Flash

— *Hors mémoire*, ce qui correspond aux 14 bits de poids forts qui doivent être à 0 pour que la valeur corresponde bien à celle d'une adresse en ROM. Si une faute apparaît sur l'un de ces bits, cela conduira le système à accéder à une adresse hors zone mémoire ou à exécuter du code qui ne correspond pas à une instruction, ce qui provoquera une exception matérielle (ce qui représente 43.75% des fautes).
— *Non utilisé*, ce qui correspond aux 4 bits de poids forts de la plage d'adressage de la ROM qui pour notre application ne sont pas utilisés. Si une faute apparaît sur l'un de ces bits, cela conduira le système à exécuter du code qui n'est pas initialisé, ce qui provoquera une exception matérielle (ce qui représente 12.5% des fautes).
— *Alignement*, ce qui correspond au bit de poids faible lors de l'exécution d'une instruction codée sur 16 bits ou aux 2 bits de poids faible lors de l'exécution d'une instruction codée sur 32 bits, qui doivent être à 0 à cause de l'alignement. Ces bits ne sont pas utilisés lors de l'exécution d'une instruction, de ce fait, si une faute apparaît sur l'un de ces bits, cela n'aura pas d'effet (ce qui représente entre 3.12% et 6.25% des fautes).
— *Zone utile*, ce qui correspond aux bits pertinents. Si une faute apparaît sur l'un de ces bits, cela conduira le système à exécuter un saut dans le code du programme. L'impact de cette faute dépend du saut qui a été réalisé (ce qui représente entre 37.5% et 40.62% des fautes)

On remarque que les résultats présentés sur la Figure 5.13 sont cohérent à cette analyse.

Le registre SP : L'apparition d'une faute sur ce registre aura pour effet de modifier l'adresse du pointeur de pile. Cela peut avoir deux principaux impacts :
— Soit la valeur corrompue ne correspond plus à une adresse de la SRAM, dans ce cas une exception matérielle sera générée.
— Soit il y a une modification des données en mémoire et l'erreur pourra apparaître plus tard.

Les registres GPR : Comme présenté sur la Figure 5.14, on observe que les SDCs sont principalement issues des registres qui sont utilisés pour la manipulation des données et pour le passage des paramètres lors des appels de fonctions ($r0$ à $r5$). Aussi on peut voir pour le registre $r7$, qui est utilisé comme alias du pointeur de pile, où une faute provoque une exception matérielle dans 76% des cas. Comme

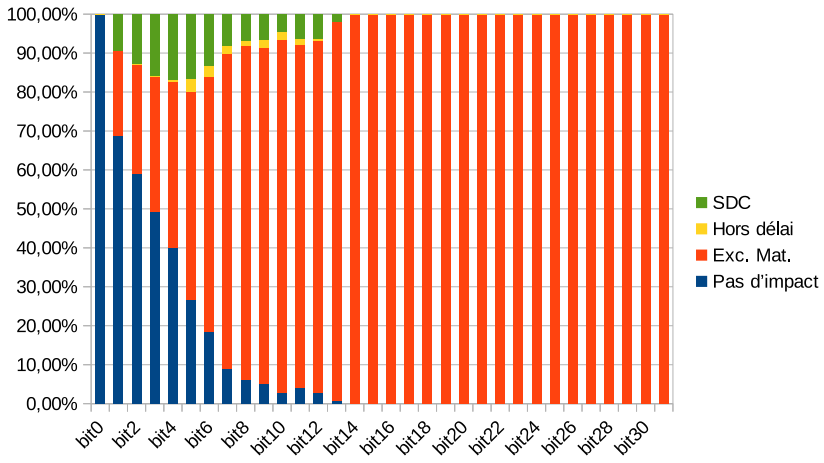

FIGURE 5.13 – Classification de l'impact des fautes pour les différents bits de *PC*

les autres registres ne sont pas utilisés, ils n'ont pas d'impact sur l'exécution du noyau.

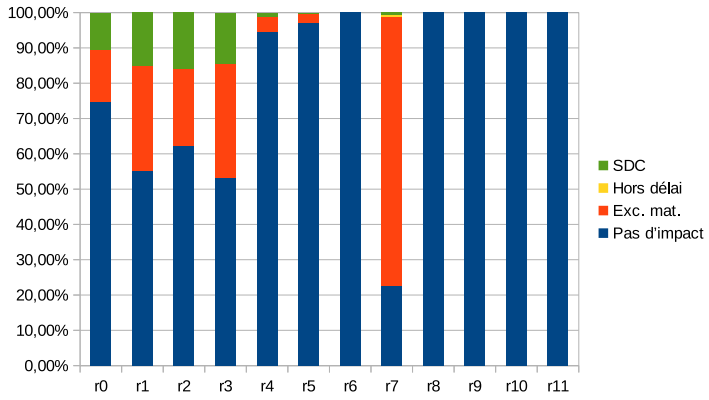

FIGURE 5.14 – Classification de l'impact des fautes pour les différents registres GPR

5.3.2 Évaluation du taux de détection d'erreur par le mécanisme de vérification en ligne

Résultats de la première campagne

Lors de la première campagne d'injection de fautes qui permet d'évaluer les fautes qui ont été détectées avant la fin de l'exécution de l'application, on obtient les résultats de la Table 5.6.

	Pas d'impact	Exception mat.	Hors délai	SDC
GPR	32 679 2.18%	73 564 16.1%	195 6.1%	55 404 84.3%
SP	411 0.43%	11 767 8.4%	1 068 62.9%	1 322 64%
LR	5 842 14.72%	48 657 13.8%	1 704 19.9%	14 282 91.3%
PC	7 010 12.96%	57 377 15.6%	907 40%	14 104 81.9%
Total	45942 2.73%	191365 14.5%	3874 24.7%	85112 84.5%

TABLE 5.6 – Nombre de faute détectées avant la fin de l'exécution de l'application

On remarque que le mécanisme de vérification en ligne est complémentaire aux autres moyens de détection. En effet on détecte 14.5% des fautes classées comme « exception matérielle » et 24.7% des fautes classées comme « hors délai ».

En ce qui concerne les fautes de la catégorie « pas d'impact », 2.73% sont détectées. Pour rappel, ces fautes sont celles qui n'ont pas d'influence sur l'ordonnancement. On peut identifier trois origines pour ces fautes :

— Les fautes qui impactent le mécanisme de vérification en ligne lui-même. Pour se prémunir de cela, on a mis en place de la redondance dans les variables liées au mécanisme. Mais, même avec cela on ne peut pas éviter les erreurs comme les sauts d'instructions au niveau des instructions correspondantes à l'instrumentation et aux fautes qui vont modifier l'adresse des variables stockées sur le FPGA.
— Les fautes qui sont masquées par l'OS lui-même (par exemple si l'appel de service permettant de terminer une tâche échoue, la tâche se terminera d'elle-même).
— Les fautes qui impactent une variable qui n'est plus utilisée.

Pour les fautes que l'on cherche à détecter, les SDCs, le taux de détection est de 84%. Si l'on considère l'impact minimal induit par le dispositif de vérification, ces résultats sont encourageants.

Lorsqu'on analyse le taux de détection des SDCs pour les différents registres on observe que :

— Le registre LR est celui qui a le meilleur taux de détection avec 91.3%. Cela s'explique par le fait que le flot de contrôle est réalisé au début et à la fin de chaque fonction.

— Le registre PC a un taux de détection plus faible avec 81.9%. Comme pour LR, la surveillance de PC est conditionnée par la surveillance du flot de contrôle. Dans le cas où une faute conduit à effectuer un offset suffisamment important (saut dans une autre fonction), celui-ci sera détecté. Dans le cas d'une variation plus faible (saut à une instruction de la fonction) alors la détection ne se fera sûrement pas à cause de la granularité de la détection.
— Les registres GPR ont un taux de détection de 84.3%. Ici la détection provient principalement de la surveillance des structures internes du noyau. Lors d'une campagne d'injection de fautes préliminaire [59], où les structures dépendantes de l'application n'était pas surveillées, le taux de détection était de 38% pour les registres GPR.
— Le registre SP est celui qui a le moins bon taux de détection avec 64%. Cela s'explique par le fait que le mécanisme de vérification en ligne n'a pas d'accès à la pile système, les fautes qui sont détectées sont donc principalement dues à des effets de bord.

Résultats de la seconde campagne

La seconde campagne d'injection de fautes permet d'identifier le nombre de fautes ayant été détectées avant la fin de l'appel de service. Ces résultats sont présentés sur la Table 5.7 où les pourcentages correspondent aux rapports entres les fautes détectées lors de la seconde campagne d'injection et celles de la première.

	Pas d'impact	**Exception mat.**	**Hors délai**	**SDC**
GPR	31 472 96.31%	62 335 84.74%	193 98.97%	46 045 83.11%
SP	198 48.17%	9 940 84.47%	2 0.19%	810 61.27%
LR	5 754 98.49%	47 993 98.64%	1 704 100%	11 862 83.06%
PC	6 744 96.21%	56 318 98.15%	888 97.91%	12 300 87.21%
Total	44 168 96.14%	176 586 92.28%	2787 71.94%	71 017 83.44%

TABLE 5.7 – Nombre de faute détectées avant le retour dans le mode utilisateur

On remarque notamment que pour les SDCs, 83.4% des fautes sont détectées avant le

retour en mode utilisateur. Cette seconde campagne permet de mettre en évidence le fait qu'il serait possible de mettre en œuvre un mécanisme de recouvrement afin de corriger les fautes et de garantir la fiabilité du SETR.

5.3.3 Analyse des fautes non-détectées

Les résultats de la campagne d'injection de fautes sont aussi utilisés pour mettre en évidence les fautes n'ayant pas été détectées afin de proposer des améliorations possibles.

Le principe est d'extraire des résultats la liste des SDCs n'ayant pas conduit à une détection par notre mécanisme afin d'analyser s'il est possible de faire quelque chose. La difficulté de cette analyse réside principalement dans notre approche d'injection de fautes qui se fait au niveau des registres, c'est-à-dire que nous n'avons pas d'information au niveau du programme. Par exemple, on ne sait pas si une faute impacte l'adresse d'une variable ou sa valeur. On doit donc réaliser une analyse manuelle en rejouant l'injection de la faute en pas à pas.

L'analyse présentée ici n'est pas exhaustive, mais elle permet de mettre en avant un certain nombre de cas de figures.

La corruption des variables locales

Les fonctions utilisées par les appels systèmes de Trampoline ont des variables locales. Celles-ci ne sont pas surveillées par le mécanisme de vérification en ligne.

La corruption d'une variable locale peut avoir différentes origines :

— Une erreur au niveau de la pile suite, par exemple, à un bit-flip au niveau de *sp* ou $r7$.

— Un bit-flip au niveau des registres GPR pour la manipulation des données.

— Un bit-flip au niveau des bits de poids faible de *pc* qui provoque un saut en interne de la fonction qui est exécutée.

Quand une faute conduit à l'altération d'une variable locale, trois choses peuvent se passer :

1. Le code se poursuit normalement et il n'y a pas d'erreur. C'est le cas par exemple si la variable n'est pas utilisée dans la suite ou si l'altération de sa valeur n'affecte pas un bit significatif.
2. La faute se propage au niveau des variables globales et dans ce cas elle sera détectée par le mécanisme de détection.

3. La faute a un impact direct sur l'ordonnancement, ce qui fait que la faute n'est pas détectée.

Une solution possible pour détecter ces erreurs serait de surveiller ces variables par le mécanisme de détection en les déclarant comme globales et volatiles. Cependant cela aurait pour conséquence d'augmenter de façon significative le nombre de propriétés à surveiller, donc les ressources utilisées sur le FPGA et le temps d'exécution de l'application.

Une autre solution serait de réduire le temps d'exposition de ces variables au niveau des registres. Pour cela il faudrait que ces variables soient déclarées comme `volatiles`. En contrepartie, à causes de la multiplication des accès mémoires à la variable, le temps d'exécution devrait être augmenté.

La corruption des arguments des fonctions

Pour la majorité des fonctions, l'argument d'entrée correspond à l'identifiant de la tâche. Dans Trampoline, l'identifiant de la tâche est utilisé pour récupérer l'adresse des descripteurs de tâche. Ainsi un bit-flip peut amener un service à exécuter une action pour une autre tâche.

Une solution possible serait de faire en sorte que lors de la compilation par `GOIL`, les identifiants des tâches soient affectés en utilisant un code correcteur d'erreur (code de *Hamming* par exemple) de façon à être tolérant faces aux bit-flips.

Les erreurs sur le flot de contrôle

Ces erreurs peuvent être issues de :

— Un bit-flip sur un registre lors d'une comparaison avant un branchement.

— Un bit-flip sur *pc* au début de l'exécution d'un service qui amène le code à exécuter directement la fin du service ou la fin d'un autre service.

— Un bit-flip au niveau des bits de poids faible de *pc* qui provoque un saut en interne de la fonction qui est exécutée.

La mise en œuvre d'une surveillance plus fine devrait permettre d'améliorer les résultats. Par exemple, la surveillance pourrait se faire au niveau des blocs d'instructions plutôt qu'au niveau des fonctions. Cependant cela augmenterait le surcoût d'exécution des services du fait du code d'instrumentation supplémentaire, et cela augmenterait aussi les ressources matérielles sur le circuit logique programmable du fait des moniteurs en plus.

5.4 Conclusion

La campagne d'injection de fautes présentée dans ce chapitre a consisté à simuler 892 320 fautes de type bit-flip sur une application de référence réalisant l'ensemble des appels systèmes pris en compte dans notre étude. L'injection de faute a été réalisée par une approche logicielle sur les registres du processeur en ciblant uniquement les fautes qui peuvent avoir un impact grâce à la technique *Inject-on-read*. Les résultats ont montré que le dispositif de vérification en ligne à permis de détecter 84.5% des SDCs dont 83.4% avant le retour en mode utilisateur. De plus, cette campagne a permis de mettre en avant les améliorations possibles à mettre en œuvre pour améliorer le taux de détection.

Les résultats de la campagne d'injection de fautes sont à relativiser au regard d'un certain nombre de biais qu'il faut souligner :

— La technique d'injection de fautes utilisée permet de cibler uniquement les registres accessibles du processeur et ne prend pas en compte les autres registres comme le registre d'état par exemple. Aussi on ne considère que des fautes pouvant apparaître au niveau de la mémoire, mais des fautes peuvent aussi apparaître au niveau des différents composants du processeur. Par exemple si une faute survient au niveau de l'unité arithmétique et logique, cela impliquerait un résultat faux et donc une altération du registre résultat qui serait différent d'un bit-flip.

— L'application de référence couvre au mieux le code noyau. Peut-être aurait-il été intéressant d'effectuer l'injection de fautes sur d'autres applications comme des applications issues de l'industrie ou un ensemble d'applications de références reconnus pour l'évaluation des systèmes d'exploitations AUTOSAR. Cependant, la mise en œuvre et l'analyse des résultats qui en découlerai serait trop longue dans le cadre de la thèse.

La campagne d'injection de fautes est un processus itératif qui doit être réalisé conjointement avec le développement du dispositif de détection. En effet, c'est à partir de l'analyse des fautes non-détectées qu'il est possible de déterminer des propriétés à vérifier. Notons que les résultats présentés ici sont ceux de la dernière campagne réalisée.

Conclusion générale et perspectives

Les systèmes embarqués sont omniprésents dans notre quotidien et ont en charge des fonctions de plus en plus complexes et critiques. L'un des principaux facteurs qui a permis à ces systèmes de s'imposer dans notre environnement est la miniaturisation des transistors qui les composent. Cela facilite ainsi leur intégration dans d'autres systèmes et augmente leur puissance de calcul. En contrepartie, cette miniaturisation rend ces systèmes plus sensibles aux phénomènes de radiations qui peuvent provoquer des changements d'états logiques des cellules mémoires pendant leur exécution. Ces fautes peuvent faire dévier le programme du système de son comportement normal. Ces travaux proposent une méthode permettant de détecter ces fautes en surveillant que l'exécution du programme est correcte.

La méthode de détection utilisée est basée sur la vérification en ligne, un ensemble de techniques qui consiste à (1) synthétiser des moniteurs à partir de spécifications formelles sur le comportement du programme, et (2) implémenter ces moniteurs afin de surveiller l'exécution du système. Afin de réduire l'impact temporel induit par les moniteurs sur le programme, leur implémentation est réalisée de façon matérielle. Ainsi la surveillance se fait parallèlement à l'exécution du programme. Pour cela, nous utilisons une architecture matérielle spécifique qui inclut en plus d'un processeur un circuit logique programmable. C'est sur ce circuit que sont synthétisés les moniteurs. L'observation de l'exécution du programme par les moniteurs est effectuée par l'instrumentation du programme à surveiller. Cette instrumentation consiste à allouer les principales structures de l'application dans des registres du circuit de logiques programmables et à ajouter des instructions permettant de suivre le flot d'exécution du programme. Un premier outil a été réalisé afin de générer le circuit des différents moniteurs à implémenter dans le circuit logique programmable à partir de l'ensemble des propriétés à vérifier. Les propriétés à vérifier étant écrites en `ptLTL` et le circuit décrit en `VHDL`.

Cette méthode a ensuite été mise en œuvre pour surveiller l'exécution du code noyau du système d'exploitation temps réel `Trampoline` sur la plateforme SmartFusion 2 MS060 de Microsemi. Du point du vue de la fiabilité, un SETR est un composant central qui doit

être robuste puisque l'exécution de son code se fait en mode superviseur, c'est-à-dire avec un accès total au matériel. Des propriétés, portant sur (1) le flot d'exécution des fonctions internes du noyau, (2) sur les structures statiques du SETR et (3) sur les structures dynamiques (celles qui dépendent de l'architecture logicielle de l'application), ont été spécifiées. Si l'on ne prend pas en compte les propriétés dépendantes de l'application, un total de 75 propriétés ont été spécifiées, ce qui représente une utilisation d'environ 5% des ressources logiques disponibles sur la carte d'évaluation. Notons que le circuit de vérification est synthétisé sur le circuit en utilisant la technique de *triple modular redundancy*. Le code de `Trampoline` a été modifié afin d'allouer les structures à surveiller sur le circuit logique programmable et d'ajouter le code permettant de surveiller le flot d'exécution des fonctions. Ces modifications ajoutent un surcoût d'exécution de 38% maximum lors d'un appel de service. Un second outil, basé sur le premier a été développé afin de générer l'ensemble du circuit de vérification à partir de l'architecture logicielle de l'application temps réel.

Enfin pour évaluer les performances de détection de la méthode, une campagne d'injection de fautes a été réalisée afin de simuler des changements logiques sur les cellules mémoires des registres internes accessibles du processeur. Les fautes injectées ont été sélectionnées selon la technique `inject-on-read` afin de simuler tous les cas d'injections pertinentes et ainsi avoir des résultats sur l'espace de fautes total. Pour l'application de référence utilisée lors de la campagne d'injection de fautes, ont a montré que le système de surveillance mis en œuvre permettait de détecter 83.4% des fautes qui conduisent à des corruptions de données.

Dans l'ensemble, la méthode présentée dans cette thèse permet une détection partielle des corruptions de données pour un coût très faible au niveau du temps d'exécution et des ressources matérielles utilisées. D'autres techniques, que l'on peut retrouver dans la littérature, permettent d'obtenir des meilleurs résultats en terme de détection mais pour des coûts beaucoup plus élevés, ce qui pourrait être prohibitif pour certaines applications. Ainsi dans le cas des systèmes qui intègrent un circuit logique programmable, notre méthode offre une solution très légère. pour améliorer la fiabilité d'un système d'exploitation temps réel.

Plusieurs perspectives peuvent être citées suite à ces travaux.

À court terme, il faudrait explorer les solutions mises en évidences lors de l'analyse des résultat de l'injection de fautes pour améliorer le taux de détection des corruptions de données. Les solutions proposées portent principalement sur l'intensification de la sur-

veillance en augmentant l'instrumentation ou sur la modification du code du noyau de Trampoline. Il serait aussi intéressant d'évaluer les différents compromis possibles, entre le niveau de détection et le coût. Cela permettrait de proposer aux utilisateurs plusieurs niveaux de détection selon les contraintes proposées.

À moyen terme, il serait intéressant de mettre en œuvre un mécanisme de rétablissement du système par reprise en s'appuyant sur une parte des ressources logiques disponibles sur le circuit logique pour faire des sauvegardes périodique de l'état du système. On pourrait par exemple imaginer faire une sauvegarde avant chaque appel système et de ramener le système à cet état lors de la détection d'une erreur.

À plus long terme, il serait possible de mettre en place une surveillance au niveau applicatif à l'aide de moniteurs. Il faudrait pour cela extraire un modèle de l'application à partir de l'analyse de son code et de son architecture logicielle afin d'en déduire des propriétés à vérifier. L'utilisation d'une logique temporelle temporisée serait sûrement nécessaire afin de prendre en compte les contraintes temporelles de l'application.

Annexe A

Utilisation de ptLTL dans le cadre de la thèse

L'objectif de cet Annexe est de fournir aux lecteurs les notions de ptLTL nécessaire à la compréhension de formules exprimées dans le reste du manuscrit.

On présentera dans un premier temps le fonctionnement des opérateurs ptLTL que l'on utilise dans ce manuscrit. Ensuite on donnera un exemple permettant d'illustrer la traduction d'une spécification à vérifier en une formule ptLTL.

A.1 Les opérateurs ptLTL

Les opérateurs que nous sommes amenés à utiliser pour exprimer des formules ptLTL sont rappelés ci-dessous.

Opérateur	Signification	Description
$\odot F$	(précédemment F)	est vraie si F était vraie à l'instant précédent
$\boxdot F$	(toujours F)	est vraie si F a toujours été vraie dans le passé
$[F_1; F_2)_s$	(Intervalle *fort* $[F_1\ ,\ F_2]$)	est vraie à l'instant où F_1 est vrai et est faux dès que F_2*devient* vrai
$[F_1; F_2)_w$	(Intervalle *faible* $[F_1\ ,\ F_2]$)	est vraie initialement et à l'instant où F_1 est vrai, et est faux dès que F_2 devient vrai.
$\uparrow F$	début de F	est vrai si F était faux à l'instant précédent et est vrai à l'instant présent
$\downarrow F$	fin de F	est vrai si F était vrai à l'instant précédent et est faux à l'instant présent

Leur sémantique récursive est définie comme suit où t correspond à la trace d'exécution et n correspond au rang de la trace :

$t \vDash \odot F$ iff $n > 1$ and $t_{n-1} \vDash F$,

$t \vDash \boxdot F$ iff $t \vDash F$ and ($n > 1$ implies $t_{n-1} \vDash \boxdot F$) ,

$\uparrow F$ iff $t \vDash F$ and $n > 1$ and $t_{n-1} \nvDash F$

$\downarrow F$ iff $t \nvDash F$ and $n > 1$ and $t_{n-1} \vDash F$

$t \vDash [F_1; F_2)_s$ iff $t \nvDash F_2$ and ($t \vDash F_1$ or ($n > 1$ and $t_{n-1} \vDash [F_1; F_2)_s$)),

$t \vDash [F_1; F_2)_w$ iff $t \nvDash F_2$ and ($t \vDash F_1$ or ($n > 1$ implies $t_{n-1} \vDash [F_1; F_2)_w$)).

L'opérateur $\boxdot$ est utilisé afin d'exprimer qu'une propriété doit être vérifié tout le temps, cet opérateur permet donc d'exprimer les propriétés de sécurités. Le fonctionnement de cet opérateur est illustré sur la Figure **??** où l'on peut voir que dès l'instant où F devient faux (à t = 6), $\boxdot$ devient faux et le reste même quand F redevient vrai (à t = 9).

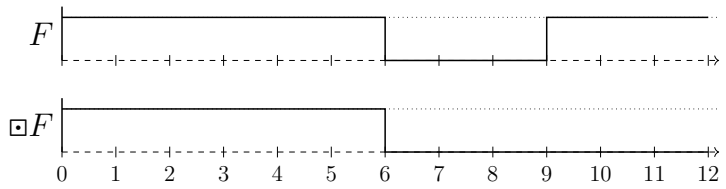

FIGURE A.1 – Chronogramme correspondant à l'opérateur $\boxdot$

L'opérateur $\odot$ est utilisé pour d'écrire l'état d'une formule à l'instant précédent. On considère dans nos travaux qu'un "instant" est défini comme un cycle d'un signal d'horloge. Le fonctionnement de cet opérateur est illustré sur la Figure **??** où l'on peut voir que $\odot F$ correspond au signal F retardé d'un cycle d'horloge.

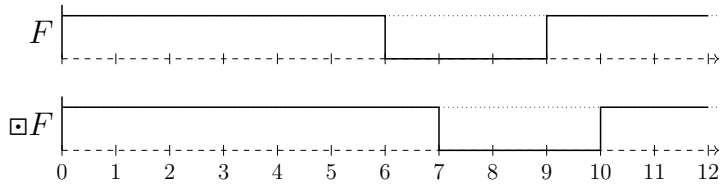

FIGURE A.2 – Chronogramme correspondant à l'opérateur $\odot$

Les opérateurs $\uparrow$ **et** $\downarrow$ sont utilisés pour d'écrire respectivement qu'une formule devient vraie et devient fausse. Le fonctionnement de ces opérateurs sont illustrés sur la Figure A.3.

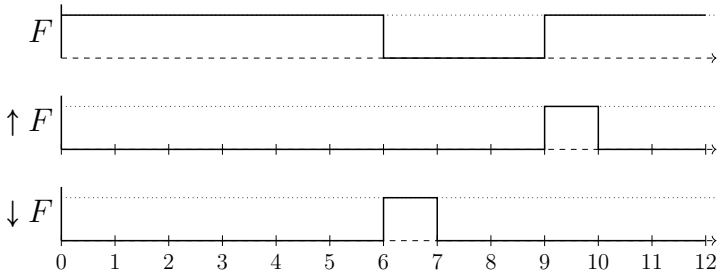

FIGURE A.3 – Chronogramme correspondant aux opérateurs ↑ et ↓

Les opérateurs $[;)_s$ et $[;)_w$ permettent de définir des intervalles. La différence entre les deux opérateurs est au niveau de l'état initiale. Pour l'opérateur $[;)_w$ (version *weak*) on considère que l'on est dans l'intervalle à $n = 0$, donc lors de l'initialisation. Pour l'opérateur $[;)_s$ (version *strong*) c'est l'inverse, il faut que la condition d'entrée dans l'intervalle soit vraie. Le fonctionnement de ces opérateurs sont illustrés sur la Figure A.4.

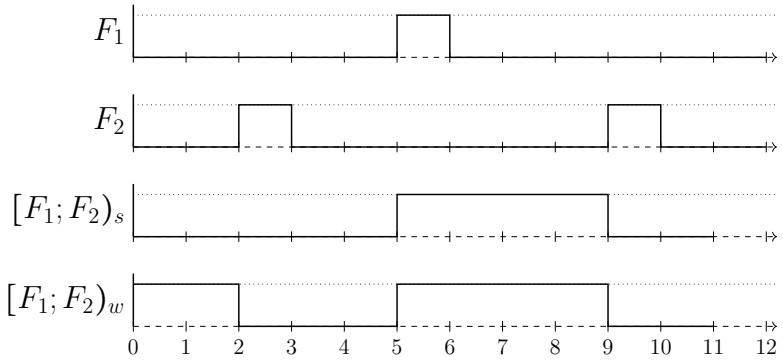

FIGURE A.4 – Chronogramme illustrant l'opérateur *intervalle* $[F_1; F_2)$ pour les versions *strong* et *weak*

A.2 Illustration par un exemple

Pour illustrer l'utilisation de la logique ptLTL pour exprimer des propriétés sur des spécifications décrites dans un langage naturel nous allons présenter un exemple simple.

Soit une fonction foo dont le rôle est de modifier une variable globale x.

Supposons que le cahier des charges impose les spécifications suivantes sur la variable x :

1. *"Pendant l'exécution de la fonction foo, la variable x doit être positive"*
2. *"En-dehors de l'exécution de la fonction foo, la variable x doit être comprise entre -10 et 10"*

On défini les propositions atomiques suivantes :

— FOO_{enter} qui devient vrai lors de l'exécution de la première instruction de la fonction foo et qui devient fausse pour toutes autres instructions.

— FOO_{exit} qui devient vrai lors de l'exécution d'une instruction qui termine la fonction foo et qui devient fausse pour toutes autres instructions.

— $X_{>0}$ qui est vrai quand la variable x est positive.

— $X_{-10<>10}$ qui est vrai quand la variable x est comprise entre -10 et 10.

Pour exprimer le fait que la fonction foo est en cours d'exécution, on peut utiliser la structure ptLTL $[FOO_{enter}; FOO_{exit})_s$. Pour exprimer le fait que la fonction foo n'est pas en cours d'exécution, il est possible d'utiliser la négation de la structure ptLTL précédente ou utiliser la version faible de l'opérateur d'intervalle, ce qui donne : $[FOO_{exit}; FOO_{enter})_w$. Cette dernière version permet de construire une formule ptLTL plus concise.

Les formule ptLTL correspondantes aux spécifications sont :

1. $F_1 = \boxdot([FOO_{enter}; FOO_{exit})_s \rightarrow X_{>0})$
2. $F_2 = \boxdot([FOO_{exit}; FOO_{enter})_s \rightarrow X_{-10<>10})$

Sur le chronogramme (Figure A.5), on présente le résultat de la formule ptLTL F_1 selon un exemple de l'évolution des propositions atomiques qui la compose.

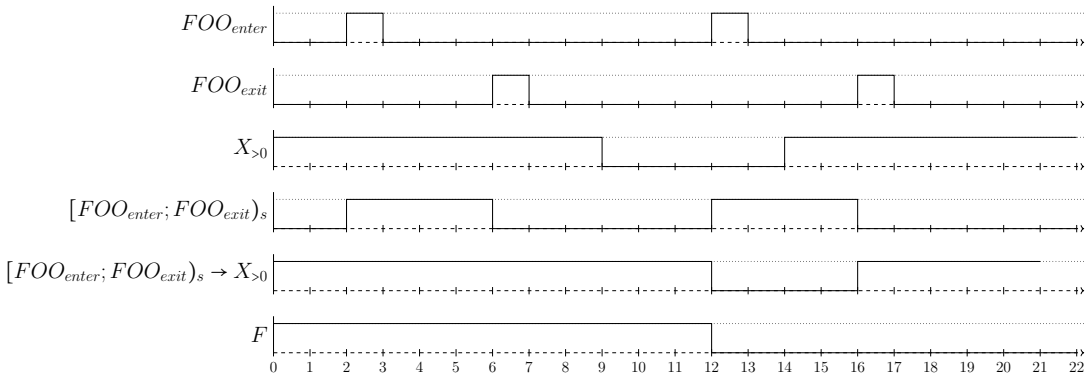

FIGURE A.5 – Exemple de chronogramme pour $F = \boxdot([FOO_{enter}; FOO_{exit})_s \rightarrow X_{>0})$

Annexe B

Spécification des propriétés de Trampoline

Dans cet Annexe nous allons lister et expliquer les différentes propriétés que nous avons défini pour la surveillance du SETR Trampoline comme expliqué dans le Chapitre [**l_chapitre4**] ainsi que définir les propositions atomiques nécessaires à l'écriture des formules ptLTL correspondantes.

Dans un premier temps on rappellera les propriétés que l'on a défini pour la surveillance des appels de fonction. Ensuite on s'intéressera aux propriétés liées à la surveillance de code du *System Call Handler*. Enfin on présentera les structures internes de Trampoline et les propriétés relatives aux différents champs qui les composent.

B.1 Surveillance des appels de fonctions

Pour rappel, on distingue deux types de fonctions internes dans Trampoline :

— Les fonctions appelés directement par le *System Call Handler*.

— Les fonctions appelés par ces fonctions.

Pour chacune de ces fonctions, nous avons assigné deux propositions atomiques :

— $nom\_de\_la\_fonction_{enter}$ qui est vrai à l'instant où l'on entre dans la fonction *nom_de_la_fonction*.

— $nom\_de\_la\_fonction_{exit}$ qui est vrai à l'instant où l'on sort de la fonction *nom_de_la_fonction*.

On veut vérifier pour chaque fonctions que :

— Il n'y a pas d'appel récursif :

$$\boxdot(nom\_de\_la\_fonction_{enter} \rightarrow [nom\_de\_la\_fonction_{exit}; \odot nom\_de\_la\_fonction_{enter})_w)$$

— Lorsque l'on sort d'une fonction cela implique que l'on était bien dans la fonction :

$$\boxdot(nom\_de\_la\_fonction_{exit} \rightarrow [nom\_de\_la\_fonction_{enter}; \odot nom\_de\_la\_fonction_{exit})_s)$$

De plus pour toutes les fonctions qui ne sont pas appelés directement par le *System Call Handler* on veut vérifier que lorsque l'on entre dans une fonction, au moins un de ces possibles appelants est en cours d'exécution. Si l'on considère que la fonction a deux appelants possibles alors la formule ptLTL est la suivante :

$$\boxdot([nom\_de\_la\_fonction_{enter}; nom\_de\_la\_fonction_{exit})_s \rightarrow$$
$$([fonction\_appelante\_1_{enter}; fonction\_appelante\_1_{exit})_s \vee$$
$$[fonction\_appelante\_2_{enter}; fonction\_appelante\_2_{exit})_s))$$

B.2 La surveillance de l'exécution du *System Call Handler*

B.2.1 Rappel sur le code du *System Call Handler*

Le *System Call Handler* est le code qui est exécuté par le passage du système du mode utilisateur au mode superviseur lors d'un appel de service. Ce code est en charge d'exécuter le code spécifique du service appelé, ce qui à pour conséquence de modifier les structures de données internes de l'exécutif. Puis selon l'état des variables de contrôle de la structure *tpl_kern* le code doit réaliser le changement de contexte et la sauvegarde de contexte.

Comme on peut le voir sur l'Algorithme [**SCH_algo_instru**] déjà présenté dans le chapitre, pour surveiller l'exécution de se code nous l'instrumentons afin de générer les propositions atomiques suivantes :

Call_Handler_enter	indique que l'on entre dans le code du *Syste* *ler*
Call_Handler_exit	indique que l'on sort du code du *System*
Service_OS_start	indique le début de l'exécution d'un servi
Service_OS_end	indique la fin de l'exécution d'un service
Context_Switch_start	indique qu'un changement de contexte do
Context_Switch_end	indique la fin du changement de contexte
Context_Save_start	indique qu'une sauvegarde du contexte do
Context_Save_end	indique la fin de la sauvegarde du context

À partir de ces propositions atomiques nous pouvons spécifier des propriétés sur l'imbrication des différentes étapes et sur l'ordre d'exécution de ces étape. De plus, en prenant en compte les propositions atomiques liées à la structure interne *tpl_kern* on peut vérifier des propriétés sur les décisions prises lors des branchements du flot de contrôle.

Algorithm 2 Algorithme du *System Call Handler* avec instrumentation

1: Générer l'événement associé à *Call_Handler_enter*
2: Obtention de l'identifiant du service à appeler
3: Générer l'événement associé à *Service_OS_start*
4: Appel du service
5: Générer l'événement associé à *Service_OS_end*
6: **if** *need_switch* == 1 **then**
7: Générer l'événement associé à *Context_Switch_start*
8: **if** *need_save* == 1 **then**
9: Générer l'événement associé à *Context_Save_start*
10: Sauvegarde du contexte
11: Générer l'événement associé à *Context_Save_end*
12: **end if**
13: Changement de contexte
14: Générer l'événement associé à *Context_Switch_end*
15: **end if**
16: *need_switch*, *need_switch* ← 0
17: Générer l'événement associé à *Call_Handler_exit*

B.2.2 Propriétés d'imbrication

Propriété 1 : L'exécution d'un appel de service implique d'être en train d'exécuter le code du *System Call Handler*

$$\boxdot([Service\_OS\_start; Service\_OS\_end)_s \rightarrow [Call\_Handler\_enter; Call\_Handler\_exit)_s)$$

Propriété 2 : L'exécution d'un changement de contexte implique d'être en train d'exécuter le code du *System Call Handler*

$$\boxdot([Context\_Switch\_start; Context\_Switch\_end)_s \rightarrow [Call\_Handler\_enter; Call\_Handler\_exit)_s)$$

Propriété 3 : L'exécution d'une sauvegarde de contexte implique d'avoir un changement de contexte

$$\boxdot([Context\_Save\_start; Context\_Save\_end)_s \rightarrow [Context\_Switch\_start; Context\_Switch\_end)_s)$$

B.2.3 Propriétés d'ordre

Propriété 1 : Quand on entre dans le code du *System Call Handler*, on ne peut pas être en train d'exécuter le code de l'appel du service, du changement de contexte ou du changement de contexte.

$$\boxdot(Call\_Handler\_enter \rightarrow (\neg[Service\_OS\_start; Service\_OS\_end)_s \wedge \neg[Context\_Switch\_start; Context\_Switch\_end)_s \wedge \neg[Context\_Save\_start; Context\_Save\_end)_s$$

Propriété 2 : Quand on sort du code du *System Call Handler*, on ne peut pas être en train d'exécuter le code de l'appel du service, du changement de contexte ou du changement de contexte.

$$\boxdot(Call\_Handler\_exit \rightarrow (\neg[Service\_OS\_start; Service\_OS\_end)_s \wedge \neg[Context\_Switch\_start; Context\_Switch\_end)_s \wedge \neg[Context\_Save\_start; Context\_Save\_end)_s)$$

Propriété 4 : Un service a forcement été appelé pendant l'exécution du *System Call Handler*

$$\boxdot(Call\_Handler\_exit \rightarrow [Service\_OS\_end; \odot Call\_Handler\_exit)_s)$$

B.2.4 Propriétés relatives à la structure *tpl_kern*

On peut vérifier que pour chacune des étapes du code du *System Call Handler* l'état de la structure *tpl_kern* est cohérente.

Lors de l'entrée et la sortie du *System Call Handler* les 3 bits de contrôles de *tpl_kern* doivent être à 0 (état $E0$).

$$\boxdot(Call\_Handler\_enter \rightarrow E0)$$
$$\boxdot(Call\_Handler\_exit \rightarrow E0)$$

Lors de l'entrée et la sortie du *System Call Handler* les informations relatives à la tâche en cours d'exécution et à la tâche élue, de *tpl_kern*, doivent être identiques (donc la proposition atomique run_elec est vraie).

$$\boxdot(Call\_Handler\_enter \rightarrow run\_elec)$$
$$\boxdot(Call\_Handler\_exit \rightarrow run\_elec)$$

Avant l'appel d'un service les 3 bits de contrôles de *tpl_kern* doivent être à 0 (état $E0$). Après l'appel du service le bit *need_schedule* doit être à 0 s'il y a un changement de contexte donc *need_switch* est à 1 (états $E5$ ou $E7$).

$$\boxdot(Service\_OS\_start \rightarrow E0)$$
$$\boxdot(Service\_OS\_end \rightarrow \neg(E5 \vee E7))$$

Avant de faire le changement de contexte le bit *need_switch* doit être à 1 (états $E1$ ou $E3$). Après le changement de contexte les informations relatives à la tâche en cours d'exécution et de la tâche élue doivent être identique (run_elec est vraie).

$$\boxdot(Context\_Switch\_start \rightarrow (E1 \vee E3))$$
$$\boxdot(Context\_Switch\_end \rightarrow run\_elec)$$

Avant de faire une sauvegarde de contexte les bits *need_save* et *need_switch* doivent être à 1 (état $E3$).

$$\boxdot(Context\_Save\_start \rightarrow E3)$$

On peut aussi vérifier que selon l'état des bits de contrôle *need_save* et *need_switch* les étapes adéquates ont été réaliser.

Si le bit *need_save* est à 1 alors une sauvegarde de contexte doit avoir été réalisée avant la sortie du bloc du changement de contexte.

$$\boxdot((E3 \wedge Context\_Switch\_end) \rightarrow [Context\_Save\_start; Call\_Handler\_exit)_s)$$

Si le bit *need_switch* est à 1 alors un changement de contexte doit avoir été réalisée avant la fin de l'exécution du *System Call Handler*.

$$\boxdot(((E1 \vee E3) \wedge Call\_Handler\_exit) \rightarrow$$
$$[Context\_Switch\_start; \odot Call\_Handler\_exit)_s)$$

B.3 Surveillance des structures internes

Pour chacun des champs de chacune des ressources internes de Trampoline nous allons présenter les propriétés pour vérifier la cohérence de la valeur contenue et de la modification vis-à-vis du flot de contrôle.

Le descripteur dynamique de tâche

Le descripteur dynamique d'une tâche est une structure, dont le code est donné ci-dessous

```
struct TPL_PROC {
  tpl_resource*          resources;        //pointeur vers la ressource prise
  tpl_activate_counter   activate_count;   //valeur du compteur d'activation
  tpl_priority           priority;         //valeur de la priorité courante
  tpl_proc_state         state;            //état de la tâche
};
```

FIGURE B.1 – Code de la structure *tpl_proc* (descripteur dynamique de tâche)

Le pointeur de ressource : Le champ correspondant au pointeur de ressource peut avoir pour valeur :

- l'adresse du descripteur d'une ressource si dans le code de l'architecture logicielle de l'application la ressource est associée à tâche,
- l'adresse du descripteur de la ressource interne (ressource automatiquement implémentée dans l'architecture logicielle, qui permet de rendre une tâche non préemptable),
- $NULL$ pour indiquer que la tâche n'a pas de ressource.

Pour vérifier la cohérence de ce champ on définit la proposition atomique $Res\_Coherency\_Task\_X$ (où X correspond au nom de la tâche) qui est vrai si la valeur contenue dans le champ est bien l'une des valeurs attendues. De ce fait, la formule ptLTL à vérifier est :

$$\boxdot(Res\_Coherency\_Task\_X)$$

Par exemple dans le code de la Figure B.2 on présente la description de la proposition atomique $Res\_Coherency\_Task\_T1$ dans le cas où une tâche $T1$ peut prendre la ressource $Res1$ dont l'adresse du descripteur de la ressource est $0x50000038$ et que l'adresse de la ressource interne est $0x50000028$.

```
Res_coherancy_Task_T1 <= '1' when ((reg_task_desc_T1_res = X"00000000")
                                or (reg_task_desc_T1_res = X"50000038")
                                or (reg_task_desc_T1_res = X"50000028") ) else '0';
```

FIGURE B.2 – Exemple du code VHDL pour la description de la proposition atomique

Le compteur d'activation : Lorsqu'une tâche est activée alors qu'elle n'est pas suspendue, l'activation est enregistrée par l'incrémentation de la valeur du compteur d'activation. Quand une tâche se termine, elle passe dans l'état $SUSPENDED$ et le compteur d'activation est décrémenté, si la valeur du compteur d'activation n'est pas nulle alors la tâche est automatiquement ajouté à la liste des tâches prêtes.

Lors de la description d'une tâche dans le fichier OIL, il est possible de fixer une valeur maximale à ce compteur. Ainsi lors de la demande d'activation d'une tâche, la valeur courante du compteur d'activation est comparé à la valeur maximale et si elle est supérieur l'activation échoue.

Pour vérifier la cohérence de ce champ, on définit la proposition atomique $Act\_Coherency\_Task\_X$ (où X correspond au nom de la tâche) qui est vrai si la valeur contenue dans le champ est inférieure ou égale au nombre d'activation maximum. De ce fait, la formule ptLTL à vérifier est :

$$\boxdot(Act\_Coherency\_Task\_X)$$

La priorité courante : La priorité d'une tâche est initialement égale à sa priorité de base (calculé par le compilateur GOIL et disponible dans le descripteur statique de la tâche), puis elle peut être augmenté quand elle prend une ressource. Dans ce cas la valeur de la priorité correspond à celle définie pour la ressource.

Pour vérifier la cohérence de la priorité courante, on définit la proposition atomique $Prio\_Coherency\_Task\_X$ (où X correspond au nom de la tâche) qui est vrai si la valeur contenue dans le champ est est bien l'une des valeurs attendues. De ce fait, la formule ptLTL à vérifier est :

$$\boxdot(Prio\_Coherency\_Task\_X)$$

L'état de la tâche : Comme nous l'avons vue sur la Figure **??** une tâche peut être dans plusieurs états. Selon leur configuration une tâche ne peut être quand certains états :

— L'état $AUTOSTART$ ne peut être atteint par une tâche uniquement si celle-ci est configurée pour être activé automatiquement au démarrage.

— L'état $WAITING$ ne peut être atteint par une tâche uniquement si celle-ci est configurée pour attendre des événements.

Pour vérifier la cohérence de l'état d'une tâche, l'idée est d'exprimer des propriétés sur l'évolution de l'état d'une tâche. Dans Trampoline l'état d'une tâche est codée par une valeur sur 3 bits, la Table B.1 donne la valeur assigné à chaque état. Puis on définit pour chaque tâche autant de propositions atomiques que d'états accessibles (ces propositions sont vrai quand la tâche est dans l'état correspondant) plus une proposition atomique ($OTHER\_STATE$) qui est vrai quand la valeur correspondant à l'état de la tâche n'est aucun des états accessibles.

État	Valeur en binaire
SUSPENDED	000
READY	001
RUNNING	010
WAITING	011
AUTOSTART	100
READY_AND_NEW	101

TABLE B.1 – Valeur correspondant aux états d'une tâche

Par exemple prenons le cas d'une tâche qui n'attend pas d'événement et qui n'est pas activée au démarrage, les états qu'elle peut prendre sont décrits sur la Figure B.3.

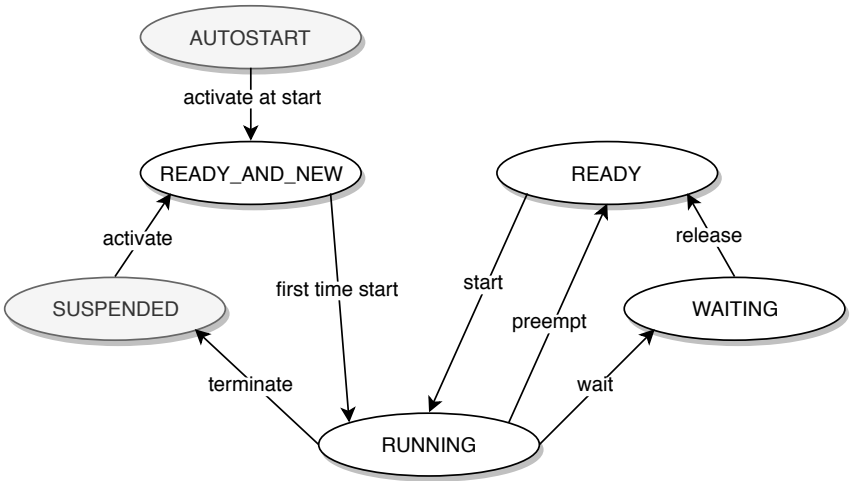

FIGURE B.3 – États d'une tâche qui n'attend pas d'événement et qui n'est pas activée au démarrage

Ici la proposition atomique $OTHER\_STATE$ est vrai quand la valeur de l'état est 011, 100 ou supérieur à 101. On peut donc écrire pour cette tâche les formules suivantes :

$\boxdot(\uparrow RUNNING\_Task\_T1 \rightarrow \odot(READY\_Task\_T1 \vee READYandNEW\_Task\_T1))$

$\boxdot(\uparrow SUSPENDED\_Task\_T1 \rightarrow \odot RUNNING\_Task\_T1)$

$\boxdot(\uparrow READYandNEW\_Task\_T1 \rightarrow \odot SUSPENDED\_Task\_T1)$

$\boxdot(\uparrow READY\_Task\_T1 \rightarrow \odot RUNNING\_Task\_T1)$

$\boxdot(\neg OTHER\_STATE\_Task\_T1)$

La structure *tpl_kern*

Comme présenté sur la Figure B.4, la structure *tpl_kern* se compose des informations relatives à la tâche en cours d'exécution et de la tâche élue par l'ordonnanceur, ainsi qui des trois bits *need_schedule*, *need_switch* et *need_save* qui donnent l'état du noyau.

```
typedef struct
{
  P2CONST(tpl_proc_static, TYPEDEF, OS_CONST) s_running;
  P2CONST(tpl_proc_static, TYPEDEF, OS_CONST) s_elected;
  P2VAR(tpl_proc, TYPEDEF, OS_VAR)            running;
  P2VAR(tpl_proc, TYPEDEF, OS_VAR)            elected;
  VAR(sint32, TYPEDEF)                        running_id;
  VAR(sint32, TYPEDEF)                        elected_id;
  VAR(uint8, TYPEDEF)                         need_switch;
  VAR(tpl_bool, TYPEDEF)                      need_schedule;
} tpl_kern_state;
```

FIGURE B.4 – Code de la structure *tpl_kern*

On définit l'état du noyau comme la combinaison des trois bits d'état :

$$E = \{need\_schedule, need\_save, need\_switch\}$$

Chacun de ces états correspond à une proposition atomique.

À partir de l'automate, de la Figure B.5, qui modélise l'évolution des états du noyau, il est possible de définir plusieurs propriétés.

Propriété de sûreté : Les états $E2$ et $E6$ ne sont pas accessible, ce qui veut dire qu'il n'est pas possible d'avoir besoin de sauvegarder le contexte ($need\_save = 1$) et de ne pas avoir à changer le contexte (($need\_switch = 1$).

$$\boxdot(\neg(E2 \vee E6))$$

Propriétés d'accessibilité : Ces propriétés expriment les conditions nécessaires à l'accès d'un état spécifique. Par exemple, on peut voir que l'accès à l'état $E3$ ne peut se faire qu'à partir des états $E0$ et $E7$.

$$\boxdot(\uparrow E0 \rightarrow \odot\neg(E5 \vee E7))$$
$$\boxdot(\uparrow E1 \rightarrow \odot(E0 \vee E5))$$
$$\boxdot(\uparrow E3 \rightarrow \odot(E0 \vee E7))$$
$$\boxdot(\uparrow E4 \rightarrow \odot E0)$$

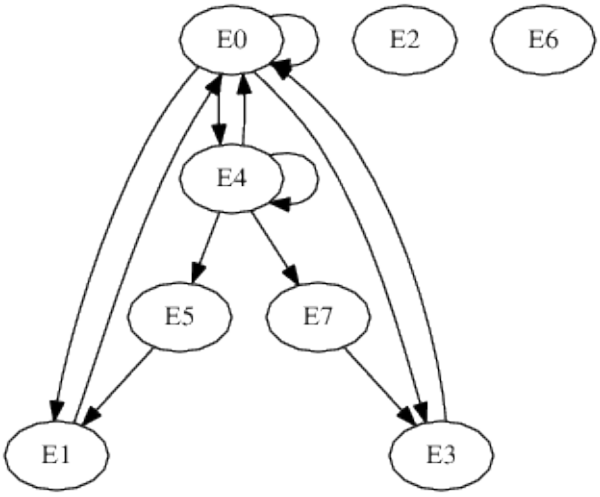

FIGURE B.5 – Modèle de l'évolution des états du noyau de Trampoline

$$\boxdot(\uparrow E5 \rightarrow \odot E4)$$
$$\boxdot(\uparrow E7 \rightarrow \odot E4)$$

Quand on entre dans l'état $E4$ ($need\_schedule = 1$), c'est que un ordonnancement de tâche doit être fait. Donc à cet instant on doit avoir les champs de la structure tpl_kern relatifs à la tâche en cours d'exécution et de la tâche élue qui doivent être identiques. Pour exprimer cette propriété on a définit la proposition atomique run_elec qui est à 1 quand ces champs sont identiques. La formule ptLTL correspondante est donc :

$$\boxdot(\uparrow E4 \rightarrow run\_elec)$$

Prise en compte du service : Il est aussi possible d'exprimer des propriétés sur l'évolution des états du noyau en fonction du service qui est appelé. Pour cela il faut définir une proposition atomique pour chaque service qui est vrai pendant l'exécution du code du service. On peut exprimer deux propriétés pour chaque service, la première permet de définir les états accessible par le service et la seconde permet d'indiquer les états finaux possibles. Par exemple, pour le service $ActivateTask$, l'automate qui représente l'évolution des états du noyau est représenté sur la Figure B.6.

Comme pour tous les services, l'état initial est l'état $E0$. Si le nombre maximal d'activation de la tâche à activer (paramètre statique qui est définie dans le descripteur statique de la tâche) est atteint alors le service s'arrête et finit dans l'état $E0$, sinon le bit $need\_schedule$ est mis à 1. Comme on atteint l'état $E4$ un ordonnancement des tâches est effectué. L'ordonnanceur compare la priorité de la tâche en cours d'exécution et de

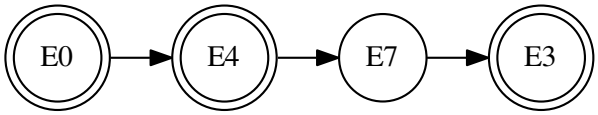

FIGURE B.6 – Modèle de l'évolution des états du noyau de Trampoline pour l'appel du service $ActivateTask$

la tâche en tête de la *ready_list*. Si la tâche en cours d'exécution à la plus haute priorité, alors il n'y a pas besoin de faire un changement de contexte et service se termine dans l'état $E4$. Autrement, un changement de contexte est requis ($need\_switch$ = 1) et le contexte de la tâche en cours doit être sauvegardé ($need\_save$ = 1), donc le service se termine dans l'état $E3$ (le bit *need_schedule* est mis à 0 juste après la mise à jour de la tâche élue. La propriété qui spécifie l'accessibilité de état pendant l'exécution du service $ActivateTask$ est donc :

$$\boxdot(ActivateTaskService \rightarrow \neg(E1 \vee E5))$$

Et la propriété qui spécifie les états finaux possibles pour ce service est :

$$\boxdot(\downarrow ActivateTaskService \rightarrow (E0 \vee E3 \vee E5))$$

Prise en compte du System Call Handler : On peut aussi mettre en relation les propositions atomiques de la structure *tpl_kern* avec celles définies pour vérifier le flot de contrôle du *System Call Handler*. De cette façon, il est possible d'exprimer les propriétés sur les branchements du flot de contrôle :

Les informations contenues dans la structure *tpl_kern* relatives à la tâche en cours d'exécution et à la tâche élue sont mises à jours pendant l'exécution du code du *System Call Handler* et doivent être identiques le reste du temps.

$$\boxdot(Call\_Handler\_enter \rightarrow run\_elec)$$
$$\boxdot(Call\_Handler\_exit \rightarrow run\_elec)$$

Les bits d'état de la structure *tpl_kern* sont remis à 0 juste avant la fin de l'exécution du *System Call Handler* et ne doivent pas avoir été modifié en dehors.

$$\boxdot(Call\_Handler\_enter \rightarrow E0)$$
$$\boxdot(Call\_Handler\_exit \rightarrow E0)$$

Avant l'appel de service la structure *tpl_kern* doit être dans l'état $E0$. À la fin de l'appel de service l'état du bit *need_schedule* doit être à 0 si il y a un changement de contexte ($need\_switch$ = 1).

$$\boxdot(Service\_OS\_start \rightarrow E0)$$
$$\boxdot(Service\_OS\_end \rightarrow \neg(E5 \vee E7))$$

Avant de faire le changement de contexte le bit *need_switch* doit être à 1. Après le changement de contexte les informations relatives à la tâche en cours d'exécution et de la tâche élue doivent être identique.

$$\boxdot(Context\_Switch\_start \rightarrow (E1 \vee E3))$$
$$\boxdot(Context\_Switch\_end \rightarrow run\_elec)$$

Avant de faire une sauvegarde de contexte le bit *need_save* doit être à 1.

$$\boxdot(Context\_Save\_start \rightarrow E3)$$

Si le bit *need_save* est à 1 alors une sauvegarde de contexte doit avoir été réalisée avant la sortie du bloc du changement de contexte.

$$\boxdot((E3 \wedge Context\_Switch\_end) \rightarrow [Context\_Save\_start; Call\_Handler\_exit)_s)$$

Si le bit *need_switch* est à 1 alors un changement de contexte doit avoir été réalisée avant la fin de l'exécution du *System Call Handler*.

$$\boxdot(((E1 \vee E3) \wedge Call\_Handler\_exit) \rightarrow$$
$$[Context\_Switch\_start; \odot Call\_Handler\_exit)_s)$$

Les descripteurs des ressources

Une ressource est un élément qui peut être pris (avec l'appel du service *GetResource*) ou relâché (avec l'appel du service *ReleaseResource*) par une tâche afin de monter sa priorité. Quand la tâche prend une ressource alors sa priorité prend pour valeur celle de la ressource et quand elle relâche une ressource sa priorité reprend sa valeur d'origine. Aussi, pour qu'une tâche puisse utiliser une ressource, il faut que cela soit défini dans le fichier OIL. La valeur de la priorité associé à une ressource est calculé par le compilateur GOIL et correspond à la priorité maximal des tâches pouvant prendre cette ressource.

Sous Trampoline on distingue deux types de ressources :

— Les ressources "classiques", qui sont gérées par l'utilisateur via les appels des services correspondants, dont le code de la structure est donné sur la Figure B.7.

— Une ressource interne, qui est utilisée uniquement par le noyau afin de rendre une tâche non préemptable, dont le code de la structure est donné sur la Figure B.8.

Notons que automatiquement une ressource "classique" est instanciée avec la plus haute priorité possible. Cette ressource peut être utilisée par n'importe quelle tâche.

```
struct TPL_RESOURCE {
  CONST(tpl_priority, TYPEDEF)
ceiling_priority;
  VAR(tpl_priority, TYPEDEF)
owner_prev_priority;
  VAR(tpl_proc_id, TYPEDEF)
owner;
  struct P2VAR(TPL_RESOURCE, TYPEDEF, OS_APPL_DATA)
                                    next_res ALIGNED;
};
```

FIGURE B.7 – Descripteur de ressource

```
typedef struct {
  CONST(tpl_priority, TYPEDEF)
ceiling_priority;
  VAR(tpl_priority, TYPEDEF)
owner_prev_priority;
  VAR(tpl_bool, TYPEDEF)
taken;
} tpl_internal_resource;
```

FIGURE B.8 – Descripteur de la ressource interne

La priorité plafond (ou *ceiling_priority*) est la valeur de la priorité assignée à la ressource. Cette valeur est une constante, pour la vérifier on définit une proposition atomique $Ceiling\_prio\_X$ (où X correspond au nom de la ressource) qui est vrai quand la valeur contenue dans ce champ est bien celle définie lors de la compilation. La formule ptLTL à vérifier est donc :

$$\boxdot(Ceiling\_prio\_X)$$

La priorité précédente du propriétaire (ou *owner_prev_priority*) est l'ancienne valeur de la priorité de la tâche ayant pris la ressource. Cette valeur est nulle quand la ressource n'est pas prise, sinon elle peut prendre pour valeur celle de la priorité d'une tâche qui peut prendre la ressource ou encore la valeur de la priorité d'une ressource dans le cas où la tâche qui prend la ressource tient déjà une autre ressource. Pour vérifier la cohérence de ce champ, on définit la proposition atomique $Owner\_prev\_prio\_X$ (où X correspond au nom de la ressource) qui est vrai quand la valeur contenue dans ce champ est cohérente. La formule ptLTL à vérifier est donc :

$$\boxdot(Owner\_prev\_prio\_X)$$

Propriétaire et ressource suivante (ou *owner* et *next_resource*) sont des champs uniquement présents dans la structure de données d'une ressource "classique" qui contiennent respectivement l'identifiant de la tâche propriétaire de la ressource (valeur nulle si la ressource n'est pas prise) et un pointeur vers la ressource déjà prise par la tâche propriétaire (valeur nulle si aucune autre ressource est prise). Pour vérifier la cohérence de ces champs, on définit les propositions atomiques $Owner\_ID\_X$ et $Next\_Res\_X$ (où X correspond

au nom de la ressource) qui sont vrai quand la valeur contenue dans ces champs est cohérente. Les formules ptLTL à vérifier sont donc :

$$\boxdot(Owner\_ID\_X)$$
$$\boxdot(Next\_Res\_X)$$

Le champ *taken* de la structure de données de la ressource interne est une valeur booléenne qui est à 1 quand la ressource est prise, donc qu'une tâche non préemptable est en train de s'exécuter, et à 0 quand elle n'est pas prise. Pour vérifier la valeur de ce champ il faut vérifier que quand celui-ci est à 1 on a bien l'une des tâches pouvant la prendre qui est dans l'état $RUNNING$, pour cela on a besoin de définir la proposition atomique $Taken_InternalRes$ qui est vrai quand le champ est à 1.

Par exemple si l'on définit deux tâches $T1$ et $T2$ comme les deux seuls tâches non préemptable d'une application alors la formule ptLTL à vérifier est :

$$\boxdot(Taken\_InternalRes \rightarrow (RUNNING\_Task\_T1 \vee RUNNING\_Task\_T2))$$

La liste des tâches prêtes (*ReadyList*)

La *ReadyList* se présente sous la forme d'un tableau de structures de données qui est utilisé afin de mémoriser les tâches qui sont prêtes à être exécutées. La structure de données qui constitue la *ReadyList* se composent de deux champs :

- Le premier champ, *key*, contient une valeur qui prend en compte la priorité de la tâche prête et son ordre d'activation. Par exemple si deux tâches avec la même priorité sont dans l'état $READY$ alors la valeur de la clé de la tâche qui est passée dans l'état $READY$ est supérieur.
- Le second champ, *ID*, correspond à l'identifiant de la tâche mémorisée.

Le tableau est ordonné selon les propriétés d'un tas (*heap* en anglais) où la valeur de la clé correspond au champ *key*.

Aussi le premier élément de la ReadyList est réservé pour contenir la valeur du nombre courant de tâches qui sont mémorisées (donc la taille du tableau).

Vérification de la cohérence de la taille de la *ReadyList* Cela revient à vérifier que la valeur contenue dans le premier élément est bien inférieur à la valeur maximal de la taille de la *ReadyList* (qui correspond au nombre de tâches défini dans l'application) et que cette valeur est bien incrémenté et décrémenté de 1. Pour cela, on définit la proposition

atomique *ReadyList_Size_Coherancy* qui est vrai si les conditions listées ci-dessus sont respectées. La formule ptLTL à vérifier est donc :

$$\boxdot(ReadyList\_Size\_Coherancy)$$

Aussi on vérifie que l'incrémentation et la décrémentation se fait bien quand on est dans les fonctions adéquates. L'incrémentation ne peut se faire que quand on est dans les fonctions *tpl_put_new_proc* (ajout d'un nouveau processus à la *ReadyList*) et *tpl_put_preempted_proc* (ajout d'un processus préempté à la *ReadyList*). De même la décrémentation ne peut se faire que quand on est dans la fonction *tpl_remove_front_proc* (retrait du processus en tête de la *ReadyList*). En définissant les propositions atomiques $Size\_Inc\_ReadyList$ et $Size\_Dec\_ReadyList$, qui sont respectivement vrai au moment de l'incrémentation et de la décrémentation de la taille de la *ReadyList*, on peut donc écrire les formules ptLTL suivantes :

$$\begin{aligned}\boxdot(Size\_Inc\_ReadyList \rightarrow ([tpl\_put\_new\_proc\_start; tpl\_put\_new\_proc\_end)_s \vee \\ [tpl\_put\_preempted\_proc\_start; tpl\_put\_preempted\_proc\_end)_s)) \\ \boxdot(Size\_Dec\_ReadyList \rightarrow \\ [tpl\_remove\_front\_proc\_start; tpl\_remove\_front\_proc\_end)_s)\end{aligned}$$

Vérification de la cohérence de la *ReadyList* Pour vérifier la cohérence des valeurs contenues dans la *ReadyList* on définit deux propositions atomiques :

- $ReadyList\_Modif$ qui est vrai à l'instant où l'une des valeurs de la *ReadyList* est modifiée. Pour cela implémente une copie de la *ReadyList* qui est mise à jour à chaque cycle d'horloge (donc retardée d'un cycle) et que l'on compare avec la *ReadyList*.
- $ReadyList\_coherancy\_father\_child$ qui est vrai si la propriété de tas est vérifiée.

À partir de ces propositions atomiques on peut exprimer le fait que la modification de la *ReadyList* ne peut se faire que à partir des fonctions adéquates :

$$\begin{aligned}\boxdot(ReadyList\_Modif \rightarrow ([tpl\_put\_new\_proc\_start; tpl\_put\_new\_proc\_end)_s \vee \\ [tpl\_put\_preempted\_proc\_start; tpl\_put\_preempted\_proc\_end)_s \vee \\ [tpl\_remove\_front\_proc\_start; tpl\_remove\_front\_proc\_end)_s))\end{aligned}$$

Et que lorsque l'on n'est pas dans l'une de ces fonctions, il faut que la propriété de tas soit respectée :

$$\boxdot(\neg([tpl\_put\_new\_proc\_start; tpl\_put\_new\_proc\_end)_s \vee$$
$$[tpl\_put\_preempted\_proc\_start; tpl\_put\_preempted\_proc\_end)_s \vee$$
$$[tpl\_remove\_front\_proc\_start; tpl\_remove\_front\_proc\_end)_s) \rightarrow$$
$$ReadyList\_coherancy\_father\_child)$$

BIBLIOGRAPHIE

[1] Paul AMMANN et Jeff OFFUTT, *Introduction to Software Testing*, 1re éd., New York, NY, USA : Cambridge University Press, 2008, ISBN : 0521880386, 9780521880381.

[2] J. ARLAT et al., « Tolérance aux fautes », in : *Encyclopédie de l'informatique et des systèmes d'information*, Vulbert, Paris, France, 2006, p. 240-270.

[3] AUTOSAR, *Specification of Operating System*, http ://www.autosar.org/specifications/release 42/software-architecture/system-services/, Part of AUTOSAR Release 4.2.2.

[4] A. AVIZIENIS et al., « Basic concepts and taxonomy of dependable and secure computing », in : *IEEE Transactions on Dependable and Secure Computing* 1.*1* (jan. 2004), p. 11-33.

[5] Fatemeh AYATOLAHI et al., « A Study of the Impact of Single Bit-Flip and Double Bit-Flip Errors on Program Execution », in : *Proceedings of the 32Nd International Conference on Computer Safety, Reliability, and Security - Volume 8153*, SAFECOMP 2013, Toulouse, France, 2013, p. 265-276, ISBN : 978-3-642-40792-5, DOI : 10.1007/978-3-642-40793-2_24.

[6] Christel BAIER et Joost-Pieter KATOEN, *Principles of Model Checking*, t. 26202649, jan. 2008.

[7] T. P. BAKER, « Stack-based scheduling of realtime processes », in : *Real-Time Systems* 3.*1* (mar. 1991), p. 67-99.

[8] Howard BARRINGER et al., « Rule-Based Runtime Verification », in : *Verification, Model Checking, and Abstract Interpretation*, sous la dir. de Bernhard STEFFEN et Giorgio LEVI, Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, p. 44-57.

[9] Ezio BARTOCCI et al., « Introduction to Runtime Verification », in : *Lectures on Runtime Verification : Introductory and Advanced Topics*, sous la dir. d'Ezio BARTOCCI et Yliès FALCONE, Cham : Springer International Publishing, 2018, p. 1-33.

[10] Andreas Bauer, Martin Leucker et Christian Schallhart, « Monitoring of Real-Time Properties », in : *FSTTCS 2006 : Foundations of Software Technology and Theoretical Computer Science*, sous la dir. de S. Arun-Kumar et Naveen Garg, Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, p. 260-272.

[11] Andreas Bauer, Martin Leucker et Christian Schallhart, « Runtime Verification for LTL and TLTL », in : *ACM Trans. Softw. Eng. Methodol.* 20.*4* (sept. 2011).

[12] R. C. Baumann, « Soft errors in advanced semiconductor devices-part I : the three radiation sources », in : *IEEE Transactions on Device and Materials Reliability* 1.*1* (mar. 2001), p. 17-22, issn : 1530-4388, doi : 10.1109/7298.946456.

[13] S. Borkar, « Designing reliable systems from unreliable components : the challenges of transistor variability and degradation », in : *IEEE Micro* 25.*6* (nov. 2005), p. 10-16, issn : 0272-1732, doi : 10.1109/MM.2005.110.

[14] Marc Boulé et Zeljko Zilic, « Automata-based assertion-checker synthesis of PSL properties », in : *ACM Trans. Design Autom. Electr. Syst.* 13.*1* (2008), 4 :1-4 :21.

[15] S. Cotard et al., « A Data Flow Monitoring Service Based on Runtime Verification for AUTOSAR », in : *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, juin 2012, p. 1508-1515.

[16] Michael Engel et Björn Döbel, « The reliable computing base – A paradigm for software-based reliability », in : *INFORMATIK 2012*, sous la dir. d'Ursula Goltz et al., Bonn : Gesellschaft für Informatik e.V., 2012, p. 480-493.

[17] Daniel Etiemble, « Introduction aux systèmes embarqués, enfouis et mobiles », in : (2016).

[18] Yliès Falcone, Klaus Havelund et Giles Reger, « A Tutorial on Runtime Verification », in : *Engineering Dependable Software Systems*, sous la dir. de Manfred Broy, Doron Peled et Georg Kalus, t. 34, NATO Science for Peace and Security Series - D : Information and Communication Security, Summer School Marktoberdorf 2012, IOS Press, 2013, p. 141-175.

[19] Padraig Fogarty, « Runtime verification monitoring for automotive embedded systems using the ISO 26262 Functional Safety Standard as a guide for the definition of the monitored properties », in : *IET Software* 8 (5 oct. 2014), 193-203(10).

[20] Allen Goldberg et Klaus Havelund, « Automated runtime verification with Eagle », in : *Workshop on Verification and Validation of Enterprise Information Systems (VVEIS). INSTICC*, 2005.

[21] Klaus Havelund et Grigore Rosu, « Efficient monitoring of safety properties », in : *STTT* 6.*2* (2004), p. 158-173.

[22] Klaus Havelund et Grigore Roşu, « Synthesizing Monitors for Safety Properties », in : *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, 2002, p. 342-356.

[23] M. Hoffmann, C. Dietrich et D. Lohmann, « dOSEK : A Dependable RTOS for Automotive Applications », in : *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, déc. 2013, p. 120-121.

[24] M. Hoffmann et al., « dOSEK : the design and implementation of a dependability-oriented static embedded kernel », in : *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, avr. 2015, p. 259-270.

[25] M. Hoffmann et al., « Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs », in : *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, juin 2014, p. 230-237.

[26] Martin Hoffmann, Christian Dietrich et Daniel Lohmann, « Failure by Design : Influence of the RTOS Interface on Memory Fault Resilience », in : *in 2nd GI W'shop on Software-Based Methods for Robust Embedded Systems (SOBRES '13), ser. Lecture Notes in Informatics. German Society of Informatics*, 2013.

[27] Jean-Claude-Laprie, « Sûreté de fonctionnement des systèmes : concepts de base et terminologie », in : *Revue de l'Électricité et de l'Électronique* - (jan. 2004), p. 95, doi : 10.3845/ree.2004.109.

[28] SC. Kleene, « Representation of Events in Nerve Nets and Finite Automata », in : (1951).

[29] J.-C Laprie, *Guide de la sûreté de fonctionnement*, Cepadues, 1996.

[30] Timo Latvala et al., « Simple Is Better : Efficient Bounded Model Checking for Past LTL », in : *Verification, Model Checking, and Abstract Interpretation*, sous la dir. de Radhia Cousot, Berlin, Heidelberg : Springer Berlin Heidelberg, 2005, p. 380-395.

[31] Martin Leucker et Christian Schallhart, « A brief account of runtime verification », in : *The Journal of Logic and Algebraic Programming* 78.*5* (2009), The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07), p. 293-303.

[32] Orna Lichtenstein, Amir Pnueli et Lenore Zuck, « The glory of the past », in : *Logics of Programs*, sous la dir. de Rohit Parikh, Berlin, Heidelberg : Springer Berlin Heidelberg, 1985, p. 196-218.

[33] Hong Lu et A. Forin, « Automatic Processor Customization for Zero-Overhead Online Software Verification », in : *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 16.*10* (2008), p. 1346-1357.

[34] Hong Lu et A. Forin, « The Design and Implementation of P2V, An Architecture for Zero-Overhead Online Verification of Software Programs », in : *MSR-TR-2007-99* (août 2007), p. 12.

[35] C. MacNamee et D. Heffernan, « On-Chip Instrumentation for Runtime Verification in Deeply Embedded Processors », in : *2015 IEEE Computer Society Annual Symposium on VLSI*, juil. 2015, p. 374-379.

[36] K. Morin-Allory et D. Borrione, « A Proof of Correctness for the Construction of Property Monitors », in : *Proceedings of the High-Level Design Validation and Test Workshop, 2005. On Tenth IEEE International*, HLDVT '05, Washington, DC, USA : IEEE Computer Society, 2005, p. 237-244, isbn : 0-7803-9571-9.

[37] Katell Morin-Allory et Dominique Borrione, « Proven correct monitors from PSL specifications », in : *Proc. Design Automation & Test in Europe Conference (DATE 2006)*, 2006, p. 1246-1251.

[38] Denis Oddoux, « Utilisation des automates alternants pour un model-checking efficace des logiques temporelles linéaires », thèse de doct., 2003.

[39] C. Pagetti et al., « The ROSACE case study : From Simulink specification to multi/many-core execution », in : *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, avr. 2014, p. 309-318.

[40] R. Pellizzoni et al., « Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems », in : *2008 Real-Time Systems Symposium*, nov. 2008, p. 481-491.

[41] Lee Pike et al., « Copilot : a hard real-time runtime monitor », in : *International Conference on Runtime Verification*, Springer, 2010, p. 345-359.

[42] A. Pnueli, « The temporal logic of programs », in : *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, oct. 1977, p. 46-57.

[43] *Property Specification Language : Reference Manual*, Accellera, 2004.

[44] Giles Reger et Klaus Havelund, « What Is a Trace ? A Runtime Verification Perspective », in : *Leveraging Applications of Formal Methods, Verification and Validation : Discussion, Dissemination, Applications*, sous la dir. de Tiziana Margaria et Bernhard Steffen, Cham : Springer International Publishing, 2016, p. 339-355.

[45] Thomas Reinbacher, Matthias Függer et Jörg Brauer, « Real-Time Runtime Verification on Chip », in : *Runtime Verification*, sous la dir. de Shaz Qadeer et Serdar Tasiran, Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, p. 110-125.

[46] Thomas Reinbacher et al., « Automated Test-Trace Inspection for Microcontroller Binary Code », in : *Runtime Verification : Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, sous la dir. de Sarfraz Khurshid et Koushik Sen, Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 239-244.

[47] Thomas Reinbacher et al., « Past Time LTL Runtime Verification for Microcontroller Binary Code », in : *Formal Methods for Industrial Critical Systems (FMICS)*, 2011, p. 37-51.

[48] M. Rodriguez, J. -. Fabre et J. Arlat, « Formal specification for building robust real-time microkernels », in : *Proceedings 21st IEEE Real-Time Systems Symposium*, nov. 2000, p. 119-128.

[49] Grigore Rosu et Klaus Havelund, *Rewriting-based Techniques for Runtime Verification*.

[50] B. Sangchoolie, K. Pattabiraman et J. Karlsson, « One Bit is (Not) Enough : An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors », in : *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, juin 2017, p. 97-108.

[51] B. Sangchoolie et al., « A Comparison of Inject-on-Read and Inject-on-Write in ISA-Level Fault Injection », in : *2015 11th European Dependable Computing Conference (EDCC)*, sept. 2015, p. 178-189.

[52] H. Schirmeier et al., « FAIL* : An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance », in : *2015 11th European Dependable Computing Conference (EDCC)*, 2015, p. 245-255.

[53] Johann Schumann, Patrick Moosbrugger et Kristin Y. Rozier, « R2U2 : Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems », in : *Proc. of RV 2015, the 6th International Conference on Runtime Verification*, 9333 : Springer, 2015, p. 233-249.

[54] Koushik Sen et Grigore Roşu, « Generating Optimal Monitors for Extended Regular Expressions », in : *Electronic Notes in Theoretical Computer Science* 89.*2* (2003), RV '2003, Run-time Verification (Satellite Workshop of CAV '03), p. 226-245.

[55] L. Sha, R. Rajkumar et J. P. Lehoczky, « Priority Inheritance Protocols : An Approach to Real-Time Synchronization », in : *IEEE Trans. Comput.* 39.*9* (1990), p. 1175-1185.

[56] Vijayshree Shinde et Seema C. Biday, « Comparison of Real Time Task Scheduling Algorithms », in : *International Journal of Computer Applications* 158.*6* (jan. 2017), p. 37-41, issn : 0975-8887.

[57] D. Skarin, R. Barbosa et J. Karlsson, « GOOFI-2 : A tool for experimental dependability assessment », in : *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, juin 2010, p. 557-562, doi : 10.1109/DSN.2010.5544265.

[58] *SmartFusion2 SoC*, https ://www.microsemi.com/product-directory/soc-fpgas/1692-smartfusion2, Accessed : 2020-06-25.

[59] D. Solet et al., « Hardware Runtime Verification of a RTOS Kernel : Evaluation Using Fault Injection », in : *2018 14th European Dependable Computing Conference (EDCC)*, 2018, p. 25-32.

[60] L. Stockmann, S. Laux et E. Bodden, « Architectural Runtime Verification », in : *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, mar. 2019, p. 77-84.

[61] Prasanna Thati et Grigore Roşu, « Monitoring Algorithms for Metric Temporal Logic Specifications », in : *Electronic Notes in Theoretical Computer Science* 113 (2005), Proceedings of the Fourth Workshop on Runtime Verification (RV 2004), p. 145-162.

[62] Jeffrey J. P. Tsai et al., « A noninterference monitoring and replay mechanism for real-time software testing and debugging », in : *IEEE Transactions on Software Engineering* 16.*8* (1990), p. 897-916.

[63] P. Ulbrich et al., « Eliminating Single Points of Failure in Software-Based Redundancy », in : *2012 Ninth European Dependable Computing Conference*, 2012, p. 49-60.

[64] C. Watterson et D. Heffernan, « Runtime verification and monitoring of embedded systems », in : *IET Software* 1.*5* (oct. 2007), p. 172-179.

[65] M. Wirthlin, « High-Reliability FPGA-Based Systems : Space, High-Energy Physics, and Beyond », in : *Proceedings of the IEEE* 103.*3* (mar. 2015), p. 379-389.

[66] Haissam Ziade, Rafic Ayoubi et Raoul Velazco, « A survey on fault injection techniques », in : *Int. Arab J. Inf. Technol.* 1.*2* (2004), p. 171-186.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Titre : Systèmes embarqués temps réel fiables et adaptables

Mots clés : Vérification en ligne, Sûreté de fonctionnement, Système embarqué temps réel

Résumé : Les systèmes embarqués sont en charge de missions de plus en plus critiques qui impliquent qu'ils ne doivent pas avoir de défaillance. Il est donc nécessaire de mettre en œuvre des mécanismes de tolérance aux fautes permettant de détecter les fautes et ainsi pouvoir rétablir le système.
Dans ces travaux, on propose de mettre en œuvre un mécanisme de détection des erreurs qui surviennent au niveau du logiciel. Ce mécanisme est basé sur l'implémentation d'un service de vérification en ligne. L'architecture matérielle du système est un système sur puce qui intègre un microcontrôleur et un circuit logique programmable. Le programme est instrumenté afin de transmettre, vers le circuit logique, les informations adéquates sur son exécution.
Des moniteurs, synthétisés sur le circuit logique à partir de propriétés à vérifier, donnent un verdict sur l'exécution du programme.
Une implémentation de ce mécanisme est réalisée pour la surveillance d'un système d'exploitation temps réel.
Enfin une campagne d'injection de fautes est effectuée afin d'évaluer les performances du mécanisme de détection.

Title : Safe and programmable real-time embedded system

Keywords : Runtime verification, Dependability, Real time embedded system

Abstract : Embedded systems are in charge of critical missions which imply that they should not have any failure. Thus, it is necessary to implement fault-tolerance mechanisms in order to detect faults and restore the system.
In this work, we propose to implement a mechanism to detect errors that occur in the program. This mechanism is based on the implementation of a runtime verification service. The system is a system-on-chip that integrates a microcontroller and a programmable logic circuit. The program is instrumented in order to transmit, to the logic circuit, the adequate information on its execution.
Monitors are synthesized on the circuit logic from properties to verify.
An implementation of this mechanism is realized to monitor a real-time operating system.
Finally, a fault injection campaign is used to evaluate the performance of the detection mechanism.