Méthodologie de conception d'architectures numériques complexes : du formalisme à l'implémentation en passant par l'analyse, préservation de la conformité. Application aux neuroprothèses

Hélène Leroux

Collège Doctoral du Languedoc-Roussillon

THÈSE

Pour obtenir le grade de

Docteur

Délivré par **UNIVERSITE MONTPELLIER 2**

Préparée au sein de l'école doctorale **Information, structures, systèmes (I2S)**
Et de l'unité de recherche **Laboratoire d'Informatique, Robotique et Micro-électronique de Montpellier (LIRMM)**

Spécialité : **Systèmes Automatiques et Microélectroniques (SYAM)**

Présentée par **Hélène LEROUX**

MÉTHODOLOGIE DE CONCEPTION D'ARCHITECTURES NUMÉRIQUES COMPLEXES : DU FORMALISME À L'IMPLÉMENTATION EN PASSANT PAR L'ANALYSE, PRÉSERVATION DE LA CONFORMITÉ.

Application aux neuroprothèses

Soutenue le 28/10/2014 devant le jury composé de

M Alexandre NKETSA, Professeur des Universités, Université de Toulouse	Rapporteur
M Jean-Marc ROUSSEL, Maître de Conférences HDR, ENS Cachan	Rapporteur
M Frédéric BONIOL, Professeur des Universités, INP Toulouse	Examinateur
M David GUIRAUD, Directeur de Recherches, INRIA	Examinateur
M David ANDREU, Maître de Conférences HDR, UM2	Directeur de thèse
Mme Karen GODARY-DEJEAN, Maître de Conférences, UM2	Encadrante de thèse
M Guillaume SOUQUET, Ingénieur R&D, MXM/AXONIC	Invité

UNIVERSITÉ MONTPELLIER 2
SCIENCES ET TECHNIQUES

Remerciements

Ces trois années de thèse ont été particulièrement enrichissantes et épanouissantes pour moi. Ceci est très largement dû aux personnes qui m'ont accompagnée lors de cette aventure.

Je tiens tout d'abord à remercier M. Alexandre Nkesta pour m'avoir fait l'honneur d'accepter de rapporter ma thèse. Merci également à M. Jean-Marc Roussel non seulement pour son rôle de rapporteur mais aussi pour tout ce qu'il a pu m'apporter auparavant afin de me permettre de pouvoir réaliser cette thèse. Un grand merci aussi à M. Frédéric Boniol et à M. David Guiraud pour avoir accepté avec enthousiasme et intérêt d'examiner ma thèse. Je sais le temps des chercheurs très précieux et j'apprécie d'autant plus qu'ils en consacrent tous les quatre à examiner mes travaux.

Mes directeurs de thèse Mme Karen Godary-Dejean et M. David Andreu sont naturellement les personnes que je tiens le plus à remercier. Ce sont en effet, sans aucun doute, eux qui ont eu le rôle le plus important dans ma thèse. Ce n'est pas toujours chose aisée d'avoir des encadrants qui te permettent de progresser et de donner le meilleur de toi-même, merci à eux de m'avoir offert cette opportunité en m'apportant un encadrement de grande qualité tout en restant à mon écoute. Plus particulièrement, merci à Karen, pour le suivi qu'elle a réalisé de mes travaux, elle a été d'une aide précieuse pour parfaire mes raisonnements scientifiques. Elle a aussi su me pousser et me rassurer quand j'en avais besoin. Merci à David, tout d'abord pour m'avoir proposé cette thèse mais aussi pour toute l'aide scientifique, administrative et morale qu'il m'a apportée tout au long de la thèse. Il restera pour moi un modèle à atteindre bien sûr pour l'étendue de ses compétences scientifiques dont je n'ai pas encore trouvé les limites mais surtout pour ses qualités humaines remarquables et rares.

Tout au long de ces trois annéess, j'ai également eu la chance de profiter des précieux conseils et de l'aide de M. Guillaume Souquet, de M. Thierry Gil et de M. Guillaume Coppey. Un grand merci à eux de m'avoir accordé un peu de leur précieux temps.

Je tiens également à remercier les membres de l'équipe DEMAR qui m'ont permis non seulement de découvrir leur domaine de recherche que je trouve aussi intéressant qu'utile mais aussi pour m'avoir permis de travailler dans une bonne mais studieuse ambiance. Au sein de cette équipe, je tiens à remercier plus particulièrement M. David Guiraud pour le soutien matériel qu'il a pu apporté à mes travaux mais aussi pour son soutien moral et ses conseils pour mon avenir. Je tiens également à remercier Mme Annie Alliaga pour le support administratif qu'elle nous apporte à tous, toujours avec le sourire, et qui nous aide grandement. Merci également à M. Nicolas Serrurier pour son aide administrative et la bonne humeur qu'il apportait à l'accueil.

Pour la bonne humeur qui a pu m'entourer durant ma thèse, il faut bien sûr que je remercie également mes collègues thésards ou ingénieurs sans qui mes pauses, mes repas et mes soirées n'auraient pas été les mêmes. Les citer tous serait difficile et je ne m'y risquerai donc pas. J'ai une pensée plus particulière pour Marion, Thomas, Alejandro, Joris, Benjamin, Guillaume (C), Guillaume (M), David et Romain mais je n'oublie pas les autres belles rencontres que j'ai pu faire. Un merci spécial pour François avec qui j'ai partagé mes derniers week-ends de rédaction (même s'il aurait préféré les passer ailleurs qu'au labo...), la fin de ma rédaction aurait été nettement plus pénible sans lui.

Lors d'une thèse, l'occasion se présente aussi (parfois) d'avoir un peu de vie sociale en dehors du laboratoire. Dans ce cas-là, je pouvais compter sur Aurélie, Benoît, Eva, Gaëtan et Séverine. Je les remercie pour les bons moments que j'ai pu passer avec eux à penser à autre chose qu'à ma thèse. Merci également à Guillaume et à François pour le soutien qu'ils m'apportent depuis de nombreuses années maintenant.

En parlant de soutien depuis de nombreuses années, je remercie tout particulièrement mes parents qui m'ont tant apporté aussi bien pour cette thèse qu'en général. Il est évident que je n'en serais pas là aujourd'hui sans eux.

Mes derniers remerciements vont à mon "partenaire", merci de m'avoir soutenue et conseillée tout au long de ma thèse d'un point de vue professionnel et bien sûr plus généralement je le remercie pour tout ce qu'il a pu m'apporter et m'apporte toujours au quotidien.

Résumé

Dans ce mémoire, la conception de systèmes numériques complexes, et notamment de systèmes embarqués critiques, est abordée au travers d'une méthodologie allant de la modélisation formelle à l'implantation sur FPGA : la méthodologie HILECOP. Celle-ci offre au concepteur la possibilité de représenter dans un modèle formel d'une part l'architecture du système selon un assemblage de composants, et d'autre part le comportement de ces composants et leur composition par réseaux de Petri temporels. Le modèle décrit est ensuite transformé automatiquement en un modèle implémentable (en langage VHDL) pour son exécution sur la cible matérielle, mais également en un modèle analysable pour permettre l'analyse formelle des propriétés du système. Les deux objectifs principaux des travaux présentés sont l'étude de la conformité d'un point de vue comportemental entre les différents modèles utilisés dans la méthodologie (modèle conçu, modèle implémentable et modèle analysable), ainsi que l'intégration d'un mécanisme de gestion efficace des exceptions. Ces travaux ont permis de fiabiliser l'implémentation du modèle et d'obtenir un modèle analysable plus pertinent par rapport au modèle conçu, dans le sens où il garantit l'inclusion du comportement du modèle conçu dans celui du modèle analysé et réduit, dans une certaine mesure, le risque d'explosion combinatoire. Les limites de la pertinence des résultats obtenus par analyse formelle sont de plus désormais connues. En ce qui concerne la gestion des exceptions, principalement étudiée au niveau comportemental, le mécanisme de la macro-place a été retenu et adapté aux contraintes fonctionnelles et non-fonctionnelles des systèmes embarqués critiques. L'apport de la macro-place et la conservation de la conformité ont pu être validés sur des modèles industriels relatifs à l'architecture numérique de neuroprothèses.

Abstract

In this thesis, the conception of digital complex systems, and notably of critical embedded systems, is discussed through a methodology which goes from formal modeling to the implementation on a FPGA : the HILECOP methodology. This methodology offers, to a designer, the possibility of representing in a formal model from one hand the digital architecture thanks to some components' assembly, and on the other hand the behavior of these components and their composition, thanks to time Petri nets. The described model is then automatically transformed in an implementable model (in the VHDL language) for its execution on a hardware target, but also in an analyzable model to allow some formal analysis on system properties to be performed. The two main goals of the presented work are the study of the behavioral conformity between the different models used in the methodology (designed model, implementable model and analyzable model) and the integration of an efficient mechanism for handling exception. These works allow to have a more reliable implementation of the model and to obtain a more relevant analyzable model. It is now possible to guarantee that the behavior of the designed model is included in the analyzed one. The risk of combinatorial explosion has also been reduced to some extent. The limits of the relevance of the obtained results thanks to the formal analysis are henceforth known. As for exception handling, it has been mostly studied on the behavioral level. The mechanism of the macroplace has been chosen and adapted to meet the functional and non-functional constraints of critical embedded systems. The benefits given by the use of the macroplace and the preservation of the conformity between the models have been validated on industrial models relative to the digital architecture of neuroprosthetics.

Table des matières

Table des figures

Liste des tableaux

Introduction

Peu connus du grand public, les systèmes embarqués sont néanmoins devenus incontournables car utilisés dans grand nombre de nos objets du quotidien : téléphone portable, voiture, TGV, avion, alarme ... Les systèmes embarqués sont des systèmes électroniques et/ou informatiques complexes et autonomes conçus pour réaliser une tâche spécifique. Dans ce manuscrit, nous nous intéressons aux systèmes numériques électroniques plutôt qu'informatiques. Les systèmes embarqués sont généralement soumis à de fortes contraintes (faible consommation, faible encombrement, capacité mémoire réduite, contraintes temporelles, sécurité, robustesse) ce qui justifie qu'un ordinateur classique (matériel et logiciel) ne puisse pas être utilisé. De plus, un système embarqué ne peut généralement pas être modifié une fois réalisé sous la forme d'un circuit électronique. Sa conception doit donc être fiable. Ceci est d'autant plus vrai lorsque le système embarqué est critique c'est-à-dire utilisé dans des domaines où des vies humaines sont en jeu comme le transport ou le médical.

Or les ingénieurs font face à des systèmes de plus en plus complexes à concevoir. Pour répondre à cette difficulté croissante, l'industrie et la recherche ont développé et développent toujours des méthodologies outillées pour aider les concepteurs et augmenter la fiabilité de leur conception. Ces méthodologies s'appuient autant que possible sur des modèles. Pour les systèmes critiques, les modèles formels sont plus particulièrement intéressants : ils permettent non seulement une structuration plus rigoureuse de la représentation du fonctionnement souhaité du système, limitant ainsi les erreurs humaines, mais aussi l'utilisation de méthodes d'analyse formelle qui fournissent des résultats de validation plus sûrs que les méthodes traditionnelles (test sur prototype et simulation). L'analyse formelle permet en effet de garantir que certaines propriétés seront vérifiées dans tous les scénarios possibles. Cette exhaustivité n'est généralement pas réalisable avec la simulation ou le test sur prototype qui n'offrent qu'une couverture partielle. Cette particularité est évidemment intéressante, voire indispensable, pour la conception des systèmes numériques critiques.

Un des modèles formels le plus utilisé, notamment dans la recherche, pour la conception de systèmes numériques est les réseaux de Petri. L'avantage des réseaux de Petri est d'une part, de se baser sur des fondations mathématiques fortes permettant leur analyse et d'autre part, de permettre d'exprimer facilement la concurrence et le parallélisme à l'aide de sa représentation graphique. Plusieurs méthodologies utilisant les réseaux de Petri ont déjà été proposées et des logiciels existent pour réaliser leur analyse formelle.

Un inconvénient est que l'analyse ne peut être réalisée que sur le modèle du système. Cependant le modèle n'est jamais qu'une représentation du comportement du système réel et non le système lui-même. La technique d'implémentation du modèle sur la cible peut

notamment modifier le comportement du système réel par rapport à celui du modèle. Or l'important n'est pas que le comportement modélisé soit juste mais que le comportement réel du système le soit. Il faut donc pouvoir s'assurer que les résultats d'analyse obtenus apportent une information pertinente par rapport au comportement réel et connaître avec le plus de précision possible les limites de cette pertinence. Il faut alors gérer non seulement l'écart entre le comportement modélisé et le comportement réel mais aussi celui entre le comportement modélisé et le comportement analysé. En effet, le modèle analysé n'est généralement pas le modèle écrit par le concepteur car les échanges avec l'environnement du système (décrits par l'interprétation associée au modèle), nécessaires au bon fonctionnement de la majorité des systèmes dits réactifs (i.e. amenés à réagir en fonction des évolutions de l'environnement), ne peuvent pas être analysés de manière formelle. A notre connaissance aucune des méthodologies existantes ne se préoccupe très précisément de l'écart possible entre le comportement du modèle conçu, le comportement réel au sein du composant électronique programmé et celui du modèle analysé. Nous nous sommes donc intéressés à cet écart et nous avons développé cet aspect dans une méthodologie existante appelée HILECOP [5].

Le principe de cette méthodologie est l'utilisation de composants pour décrire l'architecture du système numérique complexe, et de réseaux de Petri pour décrire le comportement des composants ainsi que leurs interactions. Les avantages offerts par l'utilisation de composants au niveau de la conception sont notamment :

- La possibilité de décomposer la description d'une architecture numérique complexe en plusieurs entités plus simples appelées composants.
- La possibilité de réutiliser le même composant plusieurs fois dans une même architecture voire dans différentes architectures.

La correspondance entre conception, implémentation et analyse est l'une de nos principales préoccupations dans l'amélioration de cette méthodologie. Une autre préoccupation majeure est de respecter les contraintes strictes imposées par le contexte des systèmes embarqués critiques en termes de déterminisme et d'implémentation (surface et consommation). La surface du composant électronique sur lequel le modèle pourra être implémenté et la consommation de ce composant électronique est directement liée au nombre d'éléments du modèle. Au plus le nombre d'éléments sera élevé, au plus la surface sera grande et la consommation élevée, mais cela dépend également de la technique selon laquelle le modèle est implémenté.

Dans la modélisation d'un système, il est possible de distinguer deux comportements : le comportement normal et le comportement en cas d'exception. Le comportement normal est le comportement que le système devra généralement avoir. Le comportement en cas d'exception est l'ensemble des comportements différents du comportement normal. Cela peut-être la description du comportement en cas de problème (arrêt d'urgence par exemple) ou simplement le démarrage ou l'arrêt du système. Présentement, que ce soit au niveau architectural ou au niveau comportemental, aucun mécanisme spécifique pour la modélisation de la gestion des exceptions n'a été développé au sein de la méthodologie HILECOP. De ce fait, la description du comportement en cas d'exceptions est complexe à modéliser et très gourmande en termes de nombre d'éléments du modèle. En effet, le concepteur doit nécessairement décrire la gestion de l'exception pour tous les états dans lequel le modèle peut potentiellement être quand l'exception se produit.

De plus, un modèle complexe est naturellement difficile à concevoir et surtout il est très difficile pour le concepteur d'être certain d'avoir traité tous les cas nécessaires. Cela entraîne un plus grand risque d'erreur humaine et augmente ainsi le risque que le système puisse avoir un comportement non désiré, voire dangereux. Le manque de mécanisme pour la modélisation de la gestion des exceptions nuit donc non seulement au respect des contraintes des systèmes embarqués critiques mais aussi à la "praticité" de la méthodologie. Or cette dernière est importante pour que la méthodologie soit in fine utilisée par les concepteurs. Un tel mécanisme doit donc être développé. Le modèle devra bien sûr toujours pouvoir être implémenté et analysé mais l'écart entre conception, implémentation et analyse devra toujours être maîtrisé. De plus, l'implémentation et l'analyse devront autant que possible être automatisées à la fois pour alléger le travail des concepteurs et pour limiter le risque d'erreurs humaines lors de la transformation des modèles.

Pour illustrer la pertinence des propositions faites, les modifications de la méthodologie HILECOP seront testées sur un exemple de système numérique complexe industriel. Le domaine industriel considéré sera celui du développement de systèmes médicaux implantables actifs : les neuroprothèses qui sont développées par l'équipe-projet DEMAR et l'entreprise Axonic-MXM.

Les travaux décrits dans ce manuscrit ont en effet été réalisés au sein de l'équipe-projet DEMAR (DÉambulation et Mouvement ARtificiel). C'est une équipe de recherche hébergée par le LIRMM qui est commune à l'INRIA (INRIA Sophia Méditerranée), à l'université de Montpellier 2, au CNRS et à l'université de Montpellier 1. Les ingénieurs de l'entreprise Axonic-MXM, qui travaille en partenariat avec l'équipe DEMAR, ont contribué à tester les solutions proposées dans ce manuscrit.

Ce manuscrit est structuré en 4 chapitres.

Le premier chapitre présente le contexte dans lequel les travaux ont été réalisés. Les besoins et les contraintes apportés par le fait de travailler dans le domaine des systèmes embarqués critiques sont présentés. Ainsi la nécessité d'apporter au concepteur une méthodologie pratique et fiable est illustrée. La méthodologie HILECOP est alors introduite et ses limites identifiées. Ceci nous permet de définir les objectifs de ces travaux et les contraintes qui doivent être respectées. Les outils utilisés au cours de ces travaux sont également présentés.

Le deuxième chapitre aborde la problématique de la correspondance des comportements entre un modèle décrit à l'aide de réseaux de Petri, son implémentation et son analyse. Pour faciliter la compréhension, les difficultés de la gestion de cette correspondance entre ces trois "vues" sont présentées au lecteur au fur et à mesure : la correspondance est d'abord assurée pour un modèle réseau de Petri interprété implémenté de manière synchrone. Le problème du déterminisme (gestion des choix) est ensuite abordé et enfin les problèmes amenés par la gestion des contraintes temporelles sont considérés.

Le troisième chapitre s'intéresse à la problématique de la gestion des exceptions. La nécessité de concevoir de nouveaux mécanismes de gestion aussi bien au niveau architectural que comportemental est d'abord expliquée et les solutions existantes dans la littérature sont étudiées. Un nouveau mécanisme pour la gestion des exceptions au niveau compor-

temental est défini à partir du concept de macroplace. Une solution pour implémenter sur FPGA et analyser un modèle contenant des macroplaces est proposée. Le but de cette solution est de conserver la correspondance modèle/implémentation/analyse. Une piste de solution pour la gestion des exceptions au niveau architectural est également fournie au niveau conception, implémentation et analyse.

Le quatrième chapitre concerne l'application des réflexions menées, notamment sur la gestion des exceptions au niveau comportemental, sur un cas industriel : le processeur spécifique embarqué d'un dispositif médical implantable pour la stimulation électro-fonctionnelle. Le principe de la stimulation électro-fonctionnelle est d'abord brièvement présenté. Ensuite deux versions du processeur spécifique, appelé micro-machine, sont étudiées. Sur chaque version, l'impact de la macroplace sur la modélisation de la micro-machine mais aussi sur son implémentation et sur l'analysabilité du modèle obtenu est étudié. Ceci permet de nous assurer que non seulement la macroplace ne pénalise pas la correspondance de comportements établie précédemment entre conception, implémentation et analyse mais aussi que le concepteur a un réel avantage à les utiliser et que les contraintes sont bien respectées.

Pour terminer, une conclusion sur les travaux réalisés est présentée et des pistes de perspectives à ces travaux sont proposées.

Chapitre 1

Contexte et état de l'art

1.1 Contexte des systèmes critiques embarqués

Un système embarqué est généralement défini comme un système électronique et informatique autonome dédié à une tâche bien précise. Nous sommes aujourd'hui entourés par les systèmes embarqués : téléphone mobile, électroménager, automobile,... Parmi ces systèmes embarqués, nous allons nous intéresser plus particulièrement aux systèmes numériques complexes embarqués dit critiques.

Un système embarqué est jugé critique s'il peut mettre des vies humaines en danger ou mettre en péril des investissements importants. Nous les retrouvons ainsi notamment dans les domaines du transport (aéronautique, ferroviaire, spatial) et du médical (pacemakers par exemple). Ces systèmes doivent donc nécessairement avoir un comportement sûr. Garantir la fiabilité est d'autant plus difficile que les architectures numériques et les comportements attendus par ces systèmes sont de plus en plus complexes. Il est alors de plus en plus important de fournir aux concepteurs de ces systèmes un processus de conception performant pour les aider et les guider dans leur travail.

1.1.1 Contraintes des systèmes critiques embarqués

Pour pouvoir étudier comment réaliser la conception d'un système numérique complexe et critique de manière optimale, il faut d'abord considérer les différentes contraintes liées à ce contexte particulier. Naturellement un système embarqué critique devra satisfaire un certain nombre de propriétés fonctionnelles. Il se devra d'être :

- fonctionnel : le système doit être capable de remplir la tâche pour lequel il est conçu.
- robuste : le système doit pouvoir réaliser sa tâche dans des conditions extérieures présentant de grandes variations.
- fiable : la probabilité que le système effectue sa tâche pendant une durée donnée doit être très élevée (et correspondre au cahier des charges).
- sûr : le système doit être déterministe et ne doit pas pouvoir mettre en danger des vies humaines ou endommager la technologie.

Mais les systèmes embarqués doivent aussi être conçus en considérant les contraintes non-fonctionnelles qui sont tout aussi importantes que les fonctionnelles. Les contraintes non-fonctionnelles rencontrées dans tous les systèmes embarqués critiques sont :

- L'encombrement (et le poids) : ce critère est présent pour tous les systèmes embarqués mais peut devenir primordial dans certains contextes comme celui du médical. C'est le cas notamment pour les dispositifs médicaux implantables actifs (DMIA) dont le pacemaker est un exemple connu. En effet pour pouvoir être implanté dans le corps humain un DMIA doit nécessairement avoir des dimensions très réduites.

- La consommation électrique : les systèmes embarqués critiques sont généralement alimentés par batterie ou pile. Une consommation importante de courant nécessiterait soit de grandes batteries, incompatibles avec les contraintes de taille et de poids, soit un changement très régulier des piles (ou rechargement de la batterie) qui n'est pas toujours envisageable. Dans le contexte médical, par exemple, les piles des systèmes embarqués implantés ne peuvent être changées trop régulièrement puisque leur changement nécessite une opération. Pour information, les piles des pacemakers sont remplacées tous les 5 à 10 ans (avec une consommation moyenne de 400 μA).

- Le temps de développement : pour rester concurrentiel dans un marché souvent de niches, il convient d'être capable de développer de nouveaux systèmes opérationnels le plus rapidement possible, notamment en ayant l'opportunité de réutiliser des parties de systèmes existants.

- Le coût : les systèmes embarqués critiques sont des systèmes industriels, ils ont ainsi vocation à être rentables et leur coût doit donc être maîtrisé (notamment en temps de conception et développement).

Pour pouvoir satisfaire au mieux toutes ces contraintes fonctionnelles et non-fonctionnelles et faire face à la complexité inhérente à ces systèmes numériques, les concepteurs doivent alors avoir à leur disposition une méthodologie pratique et efficace proposant des outils de développement adéquats. Fournir des méthodologies outillées pour aider à la conception fiable de systèmes critiques est une problématique de plus en plus abordée dans l'industrie en général et dans la recherche. L'idée a alors été, comme dans d'autres domaines, de s'appuyer sur la modélisation pour essayer de maîtriser cette complexité, tant pour la conception que pour la vérification et la validation des systèmes. Il s'agit alors d'ingénierie système à base de modèles (ISBM) [22]. Pour la partie vérification et validation, trois solutions sont couramment utilisées : la simulation, le test et l'analyse formelle.

La simulation et le test consistent à tester le comportement du système dans le cas de scénarios définis par le concepteur ou créés de manière automatique. Le test est réalisé sur un prototype du système conçu tandis que la simulation est réalisée sur un modèle du système grâce à un logiciel adapté. La validation formelle, quant à elle, permet de vérifier la véracité de propriétés sur le comportement du système dans toutes les situations possibles. Les méthodes sont complémentaires, la simulation puis le test réalisés permettent d'examiner le comportement du système dans les cas prévus par le cahier des charges. Le test valide que le comportement observé en simulation est bien observé sur le prototype du système réel. La validation formelle permet, elle, de garantir notamment que dans tous les scénarios possibles, le comportement décrit par le modèle satisfait toujours un ensemble de contraintes logiques et temporelles spécifiées.

Le fait est que le contrôle de systèmes embarqués implique généralement de réagir suivant les valeurs de signaux internes et/ou externes (appelés signaux d'entrée) en émettant les bons signaux qui peuvent également être des signaux internes ou externes (appelés signaux de sortie). Il est alors impossible de tester ou de simuler le comportement du système embarqué critique dans toutes les évolutions possibles des signaux d'entrée. Tester toutes les situations d'entrée possibles à l'instant initial peut, en effet, déjà se révéler fastidieux mais considérer ensuite toutes les évolutions possibles de chaque signal d'entrée est tout simplement impossible. Dans le contexte des systèmes critiques, il est alors nécessaire de pouvoir utiliser la validation formelle qui considère par essence tous les cas possibles. Réaliser une validation formelle nécessite l'utilisation d'un modèle formel encore appelé modèle mathématique.

Néanmoins garantir le "bon fonctionnement" d'un système d'après l'analyse réalisée sur un modèle ne garantit pas le bon fonctionnement du système réel (cf. figure 1.1). Deux écarts existent en effet entre le comportement du modèle analysé et celui du système réel. Ces deux comportements résultent du comportement souhaité par le concepteur et décrit à l'aide d'un modèle formel. Mais ce modèle formel n'est pas nécessairement analysable notamment à cause de l'interaction avec des entrées (i.e. signaux externes). Le modèle analysé ne sera alors pas exactement le modèle conçu et le comportement souhaité peut alors différer du comportement analysé (écart 1). L'écart 2 provient lui du fait que le modèle n'est jamais qu'une représentation du réel. La méthode d'implémentation va notamment impacter sur le comportement réel alors qu'elle n'est généralement pas prise en compte dans la description du modèle souhaité.

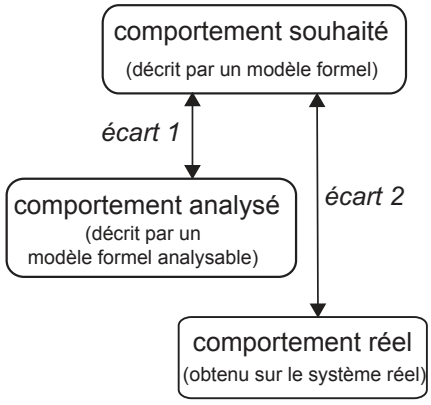

FIGURE 1.1 – Ecarts possibles entre les différents comportements considérés

Dès lors, pour s'assurer que le comportement analysé corresponde au comportement réellement observé sur le système, il faut s'intéresser à la réduction de ces écarts. Il faut notamment prendre en compte les caractéristiques relatives à l'implémentation du modèle. Cette prise en compte est nécessaire aussi bien pour la description du comportement souhaité que celui analysé. Ainsi la technologie utilisée pour implémenter le modèle doit d'abord être définie pour pouvoir réaliser une analyse formelle pertinente, au sens représentative du comportement du système réel. Différentes technologies peuvent être utilisées pour la confection de systèmes numériques complexes. Les plus couramment rencontrés sont les micro-processeurs ou les circuits électroniques programmables ou une combinaison des deux. Nous nous intéresserons dans ce travail à une implémentation sur circuit électronique et plus particulièrement sur FPGA qui est une solution technologique de plus

en plus adoptée dans les systèmes embarqués critiques de nombreux domaines : transport, spatial, médical...

Un FPGA [46] est un circuit intégré programmable. Il peut être utilisé pour divers systèmes où le comportement est décrit à l'aide de logiques combinatoire et/ou séquentielle. Un des avantages des FPGA est qu'ils sont reprogrammables. Ils sont ainsi un compromis entre une réalisation sur microcontrôleur et une réalisation sous forme de circuits spécifiques (ASIC). Comparés aux ASIC, leur temps et leur coût de développement sont plus modérés puisqu'il n'y a pas besoin de fabriquer un composant spécifique. Un FPGA est en revanche plus coûteux à l'unité qu'un ASIC particulièrement en cas de production en grande série. Les performances d'un FPGA sont également limitées en termes de consommation et de rapidité du circuit. Néanmoins elles ont été considérablement améliorées ces dernières années et les FPGA sont ainsi de plus en plus utilisés surtout pour les circuits produits en petite série. Le FPGA fournit par ailleurs un réel parallélisme permettant ainsi une vraie synchronisation entre les opérations réalisées en parallèle mais aussi de réaliser plus d'opérations à fréquence donnée qu'un microcontrôleur. Plusieurs solutions existent pour programmer un FPGA mais les langages les plus couramment employés sont le VHDL (VHSIC Hardware Description Language) [9] et le Verilog [50]. Le VHDL, langage considéré par la suite, est un langage de description matériel. Il est destiné à représenter le comportement ainsi que l'architecture d'un système électronique numérique. Le VHDL permet de réaliser une description fonctionnelle qui est ensuite traduite automatiquement en schéma de portes logiques pour l'implémentation sur FPGA. Ainsi une méthodologie de conception d'un système embarqué critique implémenté sur FPGA peut consister à générer un code VHDL à partir d'un modèle formel.

Pour conclure, il y a donc besoin d'une méthodologie outillée permettant au concepteur de concevoir, à l'aide d'un modèle formel, des systèmes embarqués critiques complexes implémentés sur FPGA mais qui lui permette aussi de vérifier et de valider le comportement réel de son système notamment à l'aide de l'analyse formelle.

1.1.2 Nécessité de développer une nouvelle méthodologie

Comme nous souhaitons travailler avec des modèles, intéressons-nous à quelques méthodes existantes dans l'ISBM. Tout d'abord, depuis plusieurs années, un langage a été développé spécifiquement pour l'ISBM : le SysML. Il permet la spécification, l'analyse, la conception, la vérification et la validation de nombreux systèmes et systèmes-de-systèmes. SysML est une extension d'un sous-ensemble du langage UML 2.0 (Unified Modeling Language), le but étant d'adapter au mieux le langage UML initialement orienté logiciel à la modélisation de systèmes. L'intérêt de SysML est de permettre à des acteurs de corps de métiers différents de collaborer autour d'un modèle commun pour définir un système. La conception de système donne en effet souvent lieu à une accumulation de documentations qui doivent toutes être croisées et mises à jour pour maintenir la cohérence et respecter les spécifications du système.

Avoir un langage pour pouvoir concevoir efficacement un système est peu utile si son utilisation n'est pas outillée. Pour cela, plusieurs projets sont développés notamment en opensource comme le projet TOPCASED [61]. TOPCASED (Toolkit in Open Source for Critical Applications and Systems Development) propose un ensemble d'outils dans le

but d'accompagner le développement de systèmes et applications critiques. L'avantage de TOPCASED est qu'il couvre l'ensemble des besoins de développement logiciel et système (la branche gauche du cycle en V habituel donné figure 1.2), ainsi que les besoins transverses comme la gestion de configuration, la gestion des changements ou l'ingénierie des exigences. La modélisation s'appuie sur les langages standardisés comme SysML mais offre aussi la possibilité d'utiliser d'autres langages comme l'UML.

Cependant les outils développés pour l'instant dans TOPCASED permettent surtout une implémentation logicielle (génération de code C par exemple) et peu une implémentation matérielle. Il n'est à ce stade pas possible, par exemple, de générer du code VHDL (ou un autre code permettant une implémentation sur FPGA). TOPCASED ne peut donc pas être utilisé directement pour la conception de systèmes embarqués critiques implémentés sur FPGA. A notre connaissance, il n'existe pas de méthodologie outillée similaire à TOPCASED, répondant précisément aux besoins caractérisant notre contexte.

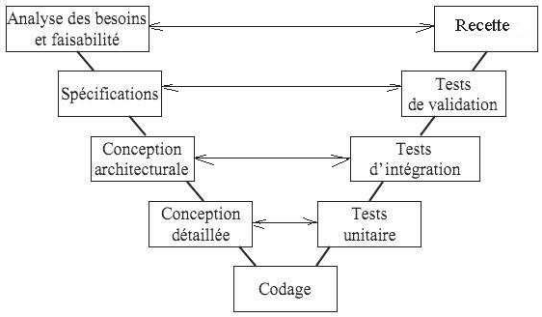

FIGURE 1.2 – Cycle de développement en V

Avoir une méthodologie permettant de gérer toutes les étapes du cycle en V comme TOPCASED est très intéressant et nécessaire mais assez complexe à développer. Nous allons donc, dans le cadre de cette thèse, nous intéresser plus particulièrement à développer une méthodologie permettant de réaliser, dans le cycle en V présenté figure 1.2, la description de la conception architecturale et de la conception détaillée ainsi que l'étape du codage (effectuée automatiquement à partir de la conception). Le but est également de fournir à l'utilisateur un moyen de réaliser l'analyse formelle nécessaire à la validation de son système. Cette méthodologie pourrait alors être par la suite intégrée dans une approche plus large permettant la gestion du cycle en V complet. TOPCASED offre, en effet, l'avantage de pouvoir être facilement complété par de nouveaux outils permettant de rendre cet environnement de plus en plus complet.

Dans le cadre de ses travaux, l'équipe DEMAR a ainsi créé une méthodologie outillée, appelée HILECOP (High Level hardware COmponent Programming). Elle permet la conception d'un système embarqué critique, son analyse formelle et son implémentation sur FPGA [49]. Une autre méthodologie, appelée Hiles [33], a été réalisée s'inspirant de l'approche HILECOP. Cette méthodologie présente l'avantage d'intégrer des possibilités de modélisation de plus haut niveau que HILECOP permettant notamment de décrire les besoins et les spécifications même si elle n'est pas encore compatible SysML ou UML. Cela nous conforte dans l'idée qu'une fois la partie basse du cycle en V maîtrisée, la méthodologie HILECOP pourrait s'inscrire en complément d'environnements existants (comme TOPCASED) ou être complétée.

L'idée est d'avoir une méthodologie permettant de passer d'une description abstraite du comportement souhaité du système, réalisée à l'aide d'un modèle formel, à un code VHDL permettant de programmer la cible. Plusieurs méthodes ont déjà été développées dans ce but. Différents langages de départ ont été choisis, les plus fréquemment utilisés sont le langage C [60] et les réseaux de Petri (RdP) [58] [42]. Or dans le cadre des systèmes critiques, nous avons aussi besoin de pouvoir réaliser une analyse formelle du comportement du système et les méthodes basées RdP offrent naturellement des possibilités d'analyse formelle [44]. Par conséquent nous focaliserons sur les RdP, à la base de la méthodologie HILECOP.

L'utilisation de RdP pour décrire des systèmes numériques complexes est relativement ancienne puisqu'elle a commencé à la fin des années 70 et les travaux académiques étaient déjà importants à la fin des années 90 [42][59]. Les RdP permettent la représentation graphique de comportements complexes comprenant de la concurrence, de la synchronisation et du partage de ressources. De plus, des outils sont déjà disponibles pour exploiter les possibilités d'analyse formelle [62][63]. En outre, plusieurs méthodes ont déjà été développées pour la transformation automatique de RdP en VHDL (cf. section 1.3). Les RdP semblent donc bien être un formalisme, satisfaisant nos besoins, sur lequel baser notre méthodologie car il permettra de réaliser aussi bien une transformation du modèle en code VHDL que d'en réaliser une analyse formelle.

Cependant la méthodologie HILECOP initiale ne permet que l'analyse du modèle conçu, au sens de celui décrit par le concepteur, et ne prend pas en compte l'impact que peut avoir l'implémentation du modèle RdP sur le comportement finalement obtenu sur le FPGA. Or c'est le comportement du système réel qui doit pouvoir être validé. Le même problème se pose dans la méthodologie Hiles. A noter un autre avantage des FPGA est d'assurer un vrai parallélisme, contrairement aux micro-contrôleurs mono-cœur (ou autres processeurs mono-cœur). Ainsi le parallélisme décrit sur le modèle RdP pourra être effectif sur la cible, ce qui permet déjà une certaine cohérence entre le modèle conçu et le comportement réel obtenu (i.e. le phénomène de préemption n'a pas à être pris en compte comme dans les implémentations basées processeur).

La méthodologie de conception se doit de plus d'être la plus pratique possible pour les concepteurs et minimiser le risque d'erreur humaine. En effet, comme indiqué précédemment, les systèmes doivent aussi pouvoir être conçus le plus rapidement possible tout en demeurant fiables. Or les erreurs humaines entraînent, quand elles sont détectées, un retard dans le conception du système et, quand elles ne le sont pas, un comportement potentiellement dangereux ce qui n'est pas acceptable dans le cadre des systèmes embarqués critiques. Le retour des utilisateurs (partenaires industriels notamment) de cette méthodologie HILECOP a mis en exergue la nécessité de pouvoir décrire la gestion des exceptions (i.e. des erreurs, des demandes de stop ...) de manière plus pratique et plus efficace. Ce nouveau mécanisme de gestion des exceptions devra naturellement être conçu en considérant aussi bien la conception que l'implémentation et l'analyse formelle, ainsi que toutes les contraintes liées au contexte des systèmes embarqués critiques, notamment les contraintes non-fonctionnelles telles que la consommation et la surface induite par l'utilisation de ce mécanisme.

Les enjeux des travaux décrits ici sont alors de modifier la méthodologie HILECOP existante pour prendre en compte l'impact de l'implémentation et aussi d'offrir une solution efficace de gestion efficace des exceptions. Nous nous intéresserons, dans un premier temps, à l'impact de l'implémentation sur le modèle conçu et analysé. Le problème de la gestion des exceptions sera ensuite traité à partir du chapitre 3.

1.2 Présentation d'HILECOP

La méthodologie HILECOP (High Level hardware COmponent Programming) [49] a été développée afin de permettre aux ingénieurs de concevoir facilement et de manière fiable une implémentation hardware de systèmes numériques complexes. HILECOP a pour but de traduire automatiquement le comportement désiré décrit à l'aide d'un modèle formel, les RdP, en un modèle implémentable sur FPGA et en un modèle analysable formellement. Avant d'entrer dans le détail de ces transformations, commençons par décrire le formalisme haut-niveau utilisé pour la description de l'architecture et du comportement désiré.

1.2.1 Description du modèle d'HILECOP

Nous avons vu précédemment que le formalisme des RdP sera utilisé pour la conception. Ils permettent la description comportementale du système. Mais les architectures des systèmes numériques sont souvent complexes car ces derniers doivent réaliser de nombreuses fonctions qui peuvent interagir entre elles. Pour pouvoir gérer cette complexité, le concepteur doit pouvoir aussi décrire l'architecture de son système, c'est-à-dire pouvoir organiser le modèle en séparant les différentes fonctions que le système doit réaliser, et décrire leurs interactions. Or les RdP ne sont pas l'outil le plus adapté pour la description architecturale d'un système.

Le choix a alors été fait d'utiliser au sein d'HILECOP une approche à composants pour la description architecturale du système. Cette approche est déjà couramment utilisée dans le domaine de l'informatique ou de la robotique [6]. Pour simplifier, un composant est une entité encapsulant un comportement et des données, offrant et/ou requérant un ensemble de services à travers des ports. Pour être exact, un composant est une entité, instance d'un descripteur de composant. Le descripteur de composant est assimilable à la classe dont le composant est une instance. Nous parlerons, quand la distinction n'est pas nécessaire, de composant pour désigner aussi bien le descripteur que ses instances. L'assemblage (la composition) des instances de composants à l'aide de ports permet de décrire l'architecture du système à concevoir.

Les avantages d'utiliser une approche à composants sont nombreux :

- Le raffinement : Le concepteur décrit l'architecture globale du système numérique puis le comportement de chaque bloc de l'architecture. Ainsi il n'a pas à décrire le comportement complexe du système global mais seulement les comportements des composants et leurs interactions. Un modèle relativement lisible même en cas de système complexe est alors conservé. L'approche à composants permet donc de faciliter le travail du concepteur et de minimiser le risque d'erreur humaine.

- La modularité : Si le concepteur souhaite modifier une partie du système numérique, il n'a pas besoin de concevoir de nouveau le système dans sa globalité mais seulement le (ou les) composant(s) concerné(s) par la modification. Tant que l'interface d'un composant et ses interconnexions avec les autres composants ne sont pas modifiées les autres composants n'ont pas besoin d'être modifiés.

- La réutilisabilité : Un même composant peut être instancié plusieurs fois dans un même projet mais aussi dans différents projets. Ceci permet notamment de diminuer le temps de développement.

Les composants d'HILECOP sont constitués de 2 principales entités (cf. Fig. 1.3) : leur interface contenant les ports permettant les échanges avec l'extérieur et leur comportement interne décrit à l'aide d'un RdP. L'interface d'un composant HILECOP peut être composée de places et de transitions de RdP ainsi que de signaux VHDL. Les places (transitions) de RdP sont nécessairement des places (transitions) appartenant à la description du comportement du composant. Ainsi la place de l'interface du composant C de la figure 1.3 est l'image de la place P_1 interne au composant.

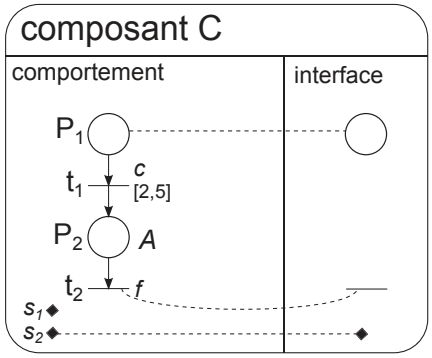

FIGURE 1.3 – Exemple de composant HILECOP

Comportement d'un composant

Nous souhaitons utiliser les RdP pour la description comportementale. Or il existe de multiples classes de RdP pour décrire des systèmes complexes [25]. HILECOP utilise les réseaux de Petri généralisés étendus interprétés T-temporels à priorités (RdP GEITP). La définition formelle de ces réseaux et leurs sémantiques complètes seront présentées dans le chapitre 2. Nous exposons ici les éléments nécessaires à la compréhension du choix de cette classe de RdP pour HILECOP. Le lecteur non familiarisé avec le vocabulaire relatif aux RdP ou leur représentation graphique pourra se référer à l'annexe A pour plus de précisions.

Le RdP est étendu, ce qui signifie que trois types d'arcs peuvent être utilisés pour relier les places aux transitions : les arcs « classiques » de type place-transition ou transition-place, les arcs tests et les arcs inhibiteurs nécessairement de type place-transition. Les arcs tests et inhibiteurs permettent de traduire l'influence d'une place sur une transition sans que la transition n'ait d'influence sur la place. Comme le RdP est aussi généralisé, les poids peuvent être associés à chacun de ces types d'arcs. Avoir toutes ces options permet d'augmenter l'expressivité des RdP. En effet, la figure 1.4 présente un exemple de RdP tiré

de [25] avec arc inhibiteur qui ne peut pas être modélisé sans arc inhibiteur. Cela permet aussi au concepteur d'avoir un modèle plus compact (i.e. contenant moins de places) et plus lisible.

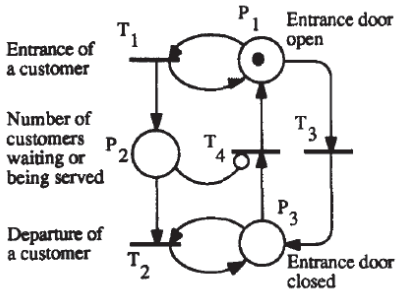

FIGURE 1.4 – Exemple de RdP ne pouvant être décrit sans arc inhibiteur [25]

Le formalisme doit permettre de modéliser des circuits numériques ayant un lien avec le monde extérieur. Il faut donc qu'il puisse non seulement agir sur l'environnement mais aussi réagir en fonction de l'état de l'environnement. Il est donc nécessaire d'utiliser des RdP interprétés. L'interprétation est dans notre cas portée par 3 entités :

- Les actions : elles permettent d'agir de manière continue sur un signal interne ou de sortie. Par exemple, l'exécution d'une action peut maintenir à 1 un signal. Elles sont associées aux places.

- Les fonctions : elles permettent de réaliser des actions impulsionnelles (traitements) sur des signaux internes ou de sortie. Par exemple, une fonction peut ajouter 1 à une variable interne pour réaliser un compteur. Elles sont associées aux transitions.

- Les conditions : une condition permet de faire dépendre l'évolution du modèle des valeurs de signaux d'entrée. Elles sont associées aux transitions.

Le formalisme doit, de plus, permettre de modéliser les propriétés temporelles de la commande, comme les chiens de garde. Pour cela, le RdP est T-temporel : une fenêtre de tir peut-être associée aux transitions devenant ainsi des transitions temporelles. Cette fenêtre de tir est définie par un intervalle $[T_{min}, T_{max}]$. T_{min} et T_{max} définissent respectivement les dates de tir au plus tôt et au plus tard de la transition.

Le contrôle du système décrit par le modèle formel doit être déterministe. Le système étant critique, il est en effet nécessaire de pouvoir toujours prédire le comportement du système. Si jamais il y a un choix dans l'évolution du modèle formel, il est donc nécessaire que dans une même situation, le même choix soit toujours réalisé. Le problème se pose évidemment lorsque le concepteur souhaite exprimer un OU entre plusieurs évolutions possibles. Plusieurs solutions sont envisageables pour s'assurer que le choix sera déterministe : l'utilisation d'arcs inhibiteurs, l'utilisation de conditions mutuellement exclusives ou la définition de priorités entre les transitions. L'utilisation des priorités par rapport aux autres solutions facilite le travail du concepteur et permet d'avoir un modèle plus compact et plus lisible. C'est pourquoi les RdP à priorités sont utilisés.

Le choix d'utiliser des RdP GEITP a ainsi été fait dans le but d'avoir un formalisme qui remplisse toutes les contraintes du contexte et qui permette au concepteur de réaliser des modélisations de manière pratique. Décrivons maintenant comment est gérée l'évolution d'un RdP GEITP.

L'évolution d'un RdP est réalisée par le tir des transitions. Une transition est dite sensibilisée si le marquage de ses places entrantes correspond aux contraintes données par les arcs. Pour les arcs classiques et tests, le marquage de la place entrante doit être supérieur ou égal au poids de l'arc tandis que pour les arcs inhibiteurs le marquage doit être strictement inférieur au poids de l'arc. Le moment où la transition devient sensibilisée est l'origine du temps relatif, auquel se réfèrent les bornes de l'intervalle temporel, si la transition est temporelle. Une transition temporelle ne peut être franchissable qu'après T_{min} et qu'avant T_{max}. La transition n'est effectivement franchissable que lorsque la condition associée à la transition est vraie (l'absence de condition associée correspond à l'association de la condition « toujours vrai »). Une transition tirable doit toujours être tirée ce qui permet d'avoir une évolution déterministe.

Le tir d'une transition entraîne le retrait des jetons des places entrantes liées à la transition par un arc classique et l'ajout de jetons dans les places sortantes de la transition. Le nombre de jetons ajoutés (resp. retirés) est égal au poids de l'arc reliant la transition à la place (resp. la place à la transition). De plus, lors du tir de la transition, la ou les fonctions associées à la transition sont exécutées.

Assemblage et agrégation des composants

L'assemblage et l'agrégation de composants permettent au concepteur de décrire aisément l'architecture de son système. L'assemblage de composants consiste à connecter 2 composants se situant au même niveau hiérarchique. L'agrégation de composants consiste à encapsuler des composants connectés ou non, au sein d'un composant de niveau supérieur. Un composite (composant contenant des composants) est alors obtenu. Un exemple de composite créé avec deux instances du composant donné figure 1.3 est donné figure 1.5. Un composite peut aussi contenir des composites.

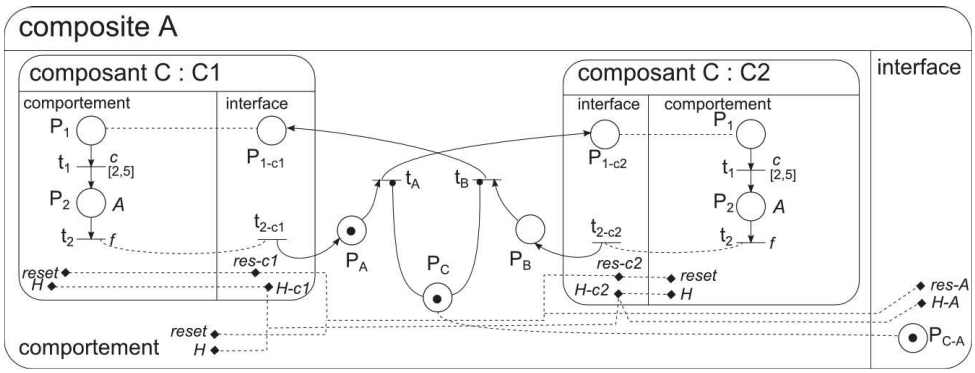

FIGURE 1.5 – Exemple de composite

Les interconnexions entre les composants/composites peuvent être réalisées de trois manières différentes, illustrées figure 1.5 :

- L'interconnexion de signaux ou de variables peut se faire directement et signifie que ces signaux ou variables sont identiques (ou fusionnés). Elle est représentée par un trait pointillé reliant les deux signaux. Il s'agira donc d'un seul et même signal ou d'une seule et même variable qui sera utilisée dans plusieurs composants/composites différents. Le signal $s2$ est ainsi utilisé dans les composants $C1$, $C2$ et A et est utilisable à l'extérieur de A. Il n'est bien sûr possible de fusionner que des variables ou des signaux de même type et de même taille.

- L'interconnexion de places ou de transitions peut se faire par fusion. Le lien de fusion est représenté par un trait pointillé reliant les éléments à fusionner. Comme pour les signaux, tous les éléments de RdP fusionnés sont en fait un seul et même élément utilisé dans plusieurs composants. Tous les arcs entrants et sortants des éléments fusionnés sont donc mis en commun et l'évolution du RdP est réalisée en considérant tous ces arcs. Naturellement des places ne peuvent être fusionnées qu'avec des places et des transitions qu'avec des transitions. Les places et transitions de l'interface d'un composant sont fusionnées avec les places et transitions auxquelles elles sont associées dans le comportement du composant.

- Les éléments de RdP de l'interface peuvent aussi être connectés à des éléments de RdP externes à l'aide d'arcs de RdP. Il s'agit en fait d'utiliser les mêmes arcs que pour un RdP mais pour interconnecter une place à une transition (ou inversement) situés dans des composants différents. Par exemple, la transition t_{2-c1} est interconnectée à la place P_A. Il n'est donc pas possible d'interconnecter des places entre elles ou des transitions entre elles à l'aide de ces arcs.

Pour information, dans un projet de circuit numérique, il y a certains éléments (tels que les mémoires) qui ne sont pas décrits en RdP, ce sont les composants dits "natifs", souvent des composants « non-fonctionnels ». Ces composants sont décrits et assemblés directement au niveau VHDL par l'interconnexion de leurs signaux avec les signaux présents dans l'interface du ou des composants qui les exploitent.

1.2.2 Les différentes synthèses possibles

Nous avons vu au §1.1.1 que la conception de systèmes embarqués critiques nécessite de jongler avec de nombreuses contraintes. Suivant le domaine d'application, certaines contraintes sont plus ou moins importantes. Pour pouvoir s'adapter aux différents besoins des concepteurs, plusieurs possibilités sont offertes pour la synthèse (abus de langage pour désigner simplement le processus de traduction du modèle) en vue de l'implémentation sur FPGA. Elles sont brièvement expliquées ci-dessous. Dans tous les cas, la synthèse du comportement d'un composant est nécessairement synchrone dans la méthodologie HILECOP. Ce choix sera expliqué au chapitre 2.

Synthèse globale La synthèse globale consiste à réaliser la synthèse sans conserver les composants. Il est alors dit que le modèle est « remis à plat », c'est-à-dire qu'il est transformé en un modèle équivalent sans préservation de l'architecture décrite par les composants. Cette synthèse revient alors à l'implémentation d'un RdP GEITP. Cette solution nécessite que tous les composants/composites soient dirigés par la même horloge et que le système soit implémenté sur un seul FPGA. C'est la solu-

tion qui permet d'obtenir la surface la plus faible parmi toutes les possibilités de synthèses proposées.

Synthèse architecturale La synthèse architecturale consiste à réaliser la synthèse en conservant l'architecture apportée par les composants. Les composants sont alors assemblés au niveau VHDL. Cette solution permet d'isoler le code VHDL d'un composant/composite pour le tester séparement ou pour le réutiliser, si besoin, dans un autre projet. Cette solution a aussi pour intérêt de permettre les trois types de synthèse présentées ensuite.

Synthèse avec activité contrôlée La synthèse avec activité contrôlée permet de minimiser la consommation énergétique en désactivant les composants "inutiles" (i.e. n'ayant pas à être actif) à un instant donné. Cette approche est envisageable si certains composants/composites possèdent une fonctionnalité précise et qu'il est possible de savoir à chaque instant si cette fonctionnalité est utile ou non. Un signal est alors utilisé pour piloter l'activité de tous les composants/composites réalisant cette fonctionnalité. L'identification des composants/composites dont l'activité doit être contrôlée et la logique permettant de contrôler cette activité ne peuvent être réalisées de manière automatique. Elles nécessitent l'expertise du concepteur. Suivant ce que la technologie permet de faire, l'activité peut être contrôlée à l'aide du "clock gating", c'est-à-dire que l'horloge n'alimente plus les composants/composites et donc ils n'évoluent plus, ou en pilotant l'alimentation en énergie de chaque composant/composite (nécessite une technologie à base de mémoire FLASH).

Synthèse GALS La synthèse GALS (Globalement Asynchrone Localement Synchrone) permet de définir des fréquences de fonctionnement différentes selon les composants/composites. Dans ce cas, les interactions entre composants doivent se faire de manière asynchrone selon le principe de la "poignée de main" par exemple. L'utilisation de différentes horloges permet notamment d'améliorer la consommation en évitant de devoir contrôler tous les composants avec l'horloge la plus rapide. Cette solution permet aussi de réaliser la synthèse multi-FPGA.

Synthèse multi-FPGA La synthèse multi-FPGA est utilisée comme son nom l'indique quand le concepteur souhaite implémenter son système sur plusieurs FPGA. Dans ce cas, la synthèse est aussi nécessairement de type GALS puisque l'interconnexion entre les différents FPGA sera nécessairement asynchrone. Le concepteur peut vouloir utiliser cette solution pour optimiser la consommation en contrôlant l'activité de chacun des FPGA, ou parce que le comportement qu'il doit implémenter nécessite trop de portes logiques pour pouvoir être implémenté sur un seul FPGA.

Dans le logiciel, ces possibilités sont proposées sous forme de « paramètres » au concepteur qui choisira la plus pertinente pour son application. Les choix du concepteur influenceront non seulement la génération du modèle implémenté mais aussi celle du modèle analysable.

1.2.3 Etapes de la méthodologie

La méthodologie HILECOP comporte plusieurs étapes permettant d'obtenir un modèle implémentable sur FPGA (cf. Fig. 1.6). Ces différentes étapes vont être présentées ici dans le but de présenter la méthodologie dans sa globalité. Précisons qu'il s'agit ici de décrire les différentes étapes que la méthodologie doit suivre pour satisfaire les enjeux présentés dans le §1.1.2 et non celles qui étaient réalisées dans la méthodologie existante. L'impact des propriétés non fonctionnelles sur le modèle et l'analyse est donc par exemple pris en compte. Les étapes ayant été modifiées par rapport à la version existante d'HILECOP seront décrites plus en détails dans le chapitre 2 de ce manuscrit.

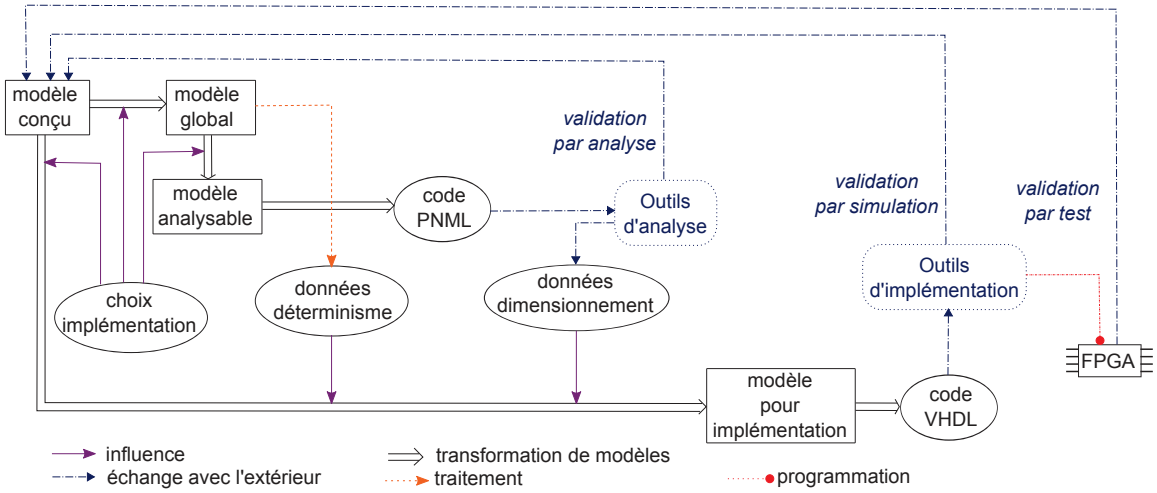

FIGURE 1.6 – Etapes de la méthodologie HILECOP

Conception du modèle

La première étape est la description du système complet par le concepteur à l'aide du langage à composant défini en 1.2.1. Il définit chaque composant ainsi que leurs assemblages et agrégations. Un modèle de l'architecture complexe, appelé modèle conçu, est alors obtenu et doit être implémenté sur le FPGA. Ce modèle contient différentes instances de composants (ou composites) reliées entre elles dont le comportement est décrit à l'aide de RdP.

Transformation en modèle global

Comme les outils d'analyse des RdP ne gèrent pas la hiérarchie apportée par les composants, la deuxième étape est la transformation du modèle conçu en un modèle RdP global. Elle est faite de manière automatique par le logiciel HILECOP et dépend de la synthèse choisie par l'utilisateur. Le concepteur peut, s'il le souhaite, visualiser le RdP obtenu. Dans le cas particulier où le modèle conçu ne contient qu'un seul composant, la transformation est évidente car le modèle RdP global est le modèle RdP décrivant le comportement du seul composant. Dans le cas d'un composite ne contenant pas de composites (mais seulement des composants), les arcs de fusion entre les nœuds du RdP et les connexions entre les signaux VHDL sont résolus afin de transformer le composite en composant (cf. Figure 1.7 dans le cadre d'une synthèse globale). Dans le cas général d'un composite global contenant des composites, les composites internes sont alors progressivement transformés en composants jusqu'à ce que le composite global devienne lui-même

un composant. Le modèle RdP global est ainsi obtenu. Cette étape était déjà traitée par le logiciel HILECOP et n'a pas été modifiée. Nous nous attarderons donc pas plus sur celle-ci dans le cadre de ce manuscrit.

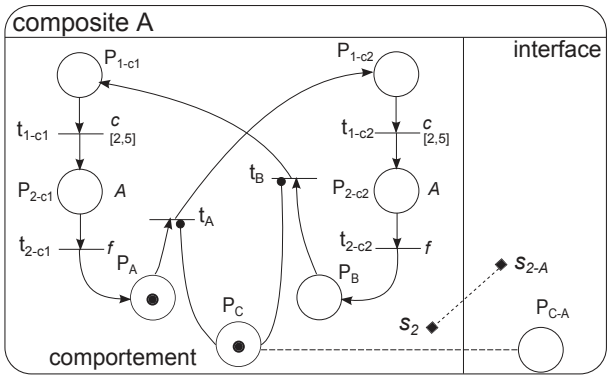

FIGURE 1.7 – Transformation du composite fig. 1.5 en composant dans le cadre d'une synthèse globale

Transformation en modèle analysable

La troisième étape est la transformation automatique du modèle RdP global en modèle analysable. Cette transformation est réalisée avant la transformation en modèle implémentable car cette dernière nécessite certains résultats d'analyse comme détaillé dans le chapitre 2. Plusieurs outils d'analyse existent pour les RdP temporels [62][63], le but ici n'est pas d'en développer un nouveau mais justement de profiter de ceux déjà développés. Tous les outils n'utilisent pas le même langage pour décrire les RdP. Pour ne pas imposer l'outil au concepteur, le choix a été fait de décrire le RdP analysable à l'aide du langage PNML [56] qui est accepté par la plupart des outils d'analyse de RdP. En effet, le langage PNML a justement été conçu pour permettre l'échange de modèles de RdP entre différents outils. La description en PNML d'un modèle RdP compatible avec la syntaxe des modèles décrits en langage PNML est simple. La problématique est de transformer le modèle RdP global décrit à l'aide de RdP GEITP en modèle pouvant être décrit en PNML, appelé ici modèle analysable. En effet, le langage PNML et plus généralement les outils d'analyse, ne gèrent pas l'interprétation contenue dans nos RdP. La classe de RdP, que nous considérons ici comme analysable, est les réseaux de Petri généralisés étendus T-temporels à priorités (GETP) définis dans [15]. C'est pourquoi nous parlerons dans la suite du manuscrit du problème de la transformation en RdP GETP et non de celui de la transformation dite model-to-text en PNML.

Il faut que le comportement analysé corresponde au comportement réel obtenu sur le FPGA. En effet, l'intérêt de l'analyse formelle est la garantie que les résultats obtenus sont valides pour tous les cas possibles. S'il existe des cas du comportement réel non analysés alors cet intérêt est perdu. Il est impossible d'avoir l'équivalence entre le comportement du modèle analysé et le réel notamment à cause de l'interprétation qui ne pourra jamais être analysée formellement. Néanmoins il est possible de s'assurer que le comportement réel est toujours inclus dans le comportement analysé. Pour garantir cette inclusion ou en identifier clairement les limites, la façon dont est implémenté le modèle doit nécessairement être prise en compte. Cette problématique et cette transformation en général seront détaillées dans le chapitre 2.

Génération du code VHDL

La quatrième et dernière étape est la génération du code VHDL en tenant notamment compte de la méthodologie de synthèse choisie par le concepteur (choix d'implémentation) (cf. §1.2.2). Cette génération dans le cas de la synthèse globale sera détaillée dans le chapitre 2. Les autres méthodes de synthèse ne seront pas abordées dans ce manuscrit. Néanmoins la méthodologie de génération du code VHDL prend en considération que d'autres méthodes de synthèse sont possibles. Ainsi une fois la méthodologie définie pour la synthèse globale, elle pourra aussi être utilisée pour les autres synthèses. En effet, dans toutes les méthodes de synthèse, il est nécessaire de générer le code VHDL correspondant au comportement d'un composant ou d'un composite ce qui correspond à la génération présentée ici. Il restera alors à gérer les interconnexions entre les composants VHDL qui seront conservés dans les différentes synthèses possibles. Mais le choix d'implémentation n'est pas le seul facteur qui va influencer sur la transformation du modèle. Les données de dimensionnement obtenues grâce à l'analyse et les données liées au déterminisme seront également utilisées.

Nous avons donc besoin de pouvoir générer un code VHDL à partir d'un modèle RdP et transformer ce modèle en RdP analysable pour pouvoir générer un code PNML. Cette transformation et ces générations doivent être réalisées de manière fiable et automatique. Cette problématique a déjà été très étudiée dans la littérature, nous allons donc commencer par étudier les solutions existantes.

1.3 Transformation du modèle RdP en un modèle implémentable sur FPGA

De très nombreux articles traitent dans la littérature de la problématique de l'implémentation sur FPGA d'un modèle formel. Tout d'abord, bien que le Verilog et le VHDL puissent être utilisés pour la programmation d'un FPGA, la transformation d'un RdP en code VHDL a été beaucoup étudiée alors que la transformation en Verilog ne l'a quasiment pas été. Une approche propose la traduction d'un RdP en code Verilog [57] mais le but de ces travaux est la simulation du code et non le prototypage sur FPGA.

Parmi tous les travaux, peu se concentrent sur un autre formalisme que celui des RdP. Nous pouvons néanmoins citer parmi eux, les travaux de Labiak [41] qui s'intéressent à l'implémentation de modèle conçu à l'aide de Statecharts [34]. Le principe est de transformer le modèle décrit à l'aide de Statecharts en équations booléennes. Ces travaux sont intéressants pour nous car le système HiCoS propose non seulement de générer automatiquement un code VHDL mais aussi le graphe des états accessibles du modèle. Les Statecharts présentent l'avantage, par rapport aux RdP, de permettre au concepteur de traduire facilement les notions de hiérarchie, d'historique et facilite la gestion des exceptions. Néanmoins, la méthode HiCoS ne permet de contrôler que des systèmes non temporels et qui ne manipulent que des signaux binaires. Ces limitations sont trop fortes dans notre contexte. Nous nous intéressons maintenant aux méthodes utilisant comme formalisme les RdP.

Dans l'ensemble des méthodes s'intéressant à l'implémentation d'un modèle RdP, les approches sont parfois très différentes. Nous allons étudier ces différentes approches en

fonction des contraintes que notre système doit remplir, notamment les possibilités de modélisation et les contraintes d'implémentation. Quatre grands critères permettent de différencier les différentes approches : le principe de transformation, les étapes de transformation, les classes de RdP considérées et l'approche de mise en œuvre. Les méthodes seront étudiées suivant le prisme de ces 4 critères. Les articles abordant la problématique de l'implémentation des RdP étant très nombreux, la liste d'articles présentée ici n'a pas la prétention d'être exhaustive mais a pour but de présenter un horizon représentatif des différentes approches existantes dans la littérature.

1.3.1 Etude des approches existantes

Principe de transformation

Deux grands principes de transformation peuvent être imaginés. Le premier, le plus utilisé, consiste à décrire la structure du RdP, tandis que la seconde consiste à implémenter le graphe des marquages accessibles (GMA). L'avantage du graphe des marquages accessibles par rapport au RdP est que le graphe des marquages accessibles est toujours une machine à états finis (MEF), c'est-à-dire un graphe à états ayant toujours un seul état actif. Or des outils industriels permettent déjà de réaliser l'implémentation de MEF sur FPGA comme Synplify [64]. Le problème est que les MEF ne sont pas adaptées pour traduire les modèles exprimant de la concurrence. Or les RdP décrivent naturellement des processus concurrents, ainsi un simple RdP peut amener à un GMA complexe et lorsque la concurrence du modèle RdP augmente, une croissance exponentielle du nombre d'états est constatée. Néanmoins implémenter un GMA complexe entraîne nécessairement l'utilisation de beaucoup de portes logiques et cela entre en conflit avec la contrainte d'optimisation de la surface des FPGA dans le cadre des systèmes embarqués critiques. De plus, le GMA est parfois tellement complexe qu'il est difficile de l'obtenir par les outils d'analyse existants. Pour ces raisons, nous considérerons désormais les solutions qui consistent à implémenter le RdP lui-même et non son GMA.

Étapes de transformation

Dans les méthodes implémentant directement les RdP et non leur graphe des marquages accessibles, 2 possibilités existent :

- l'utilisation d'un langage pivot. Le RdP est alors transformé une première fois pour être décrit à l'aide du langage pivot. Le modèle obtenu est alors traduit en VHDL. Cette transformation peut consister en l'obtention d'équations logiques ou de matrices décrivants le RdP mais aussi à l'utilisation d'un langage créé spécialement pour cela comme le langage CONPAR [31].
- la traduction directe de la structure du RdP en VHDL.

- Une méthode développée à l'université de Zielona Góra [51] [52] [19], appelée méthode de Tkacz, est de décrire le RdP à l'aide de séquents logiques de Gentzen [32]. Un séquent de Gentzen est une expression formelle utilisée pour la déduction et le calcul. Le principe des formules de Gentzen est de décrire à droite les antécédents et à gauche les succédents, les 2 étant séparés par l'opérateur $\vdash$. Par exemple, dans le cadre du RdP donné figure 1.8, le changement du marquage de la place $P1$ peut

être modélisé par le séquent suivant : $\vdash @P1 \Leftarrow P1$ *xor (t5 xor t1)* ; où @ représente l'opérateur suivant. La partie antécédent de la formule est vide car cette formule est vraie sans condition. Un séquent est écrit pour toutes les places du RdP. De même, un séquent est écrit pour chacune des sorties du RdP. Par exemple, il est obtenu pour la sortie $YV1$: $\vdash YV1 \Leftarrow P4$ *or P6* ;.

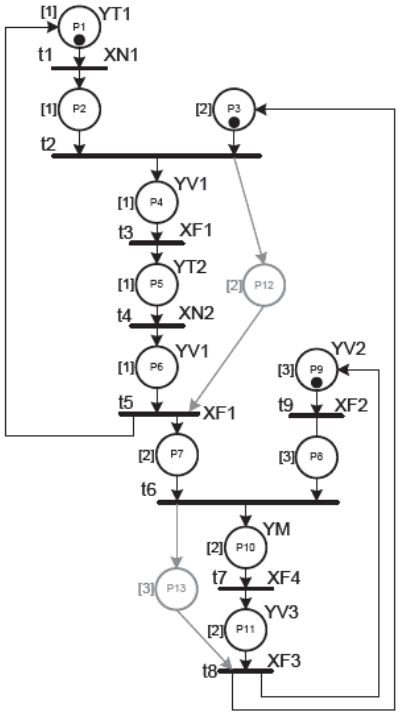

FIGURE 1.8 – Exemple de RdP utilisé dans la méthode de Tkacz [52]

L'avantage d'utiliser des séquents de Gentzen est que des outils permettent de simplifier les séquents sous une forme qui pourra être facilement transformée en VHDL de manière automatique. L'implémentation s'en trouve ainsi réduite en termes de surface. Pour réduire encore plus la surface, le RdP est coloré avant d'être transformé en séquents. La méthode de coloration du RdP est présentée dans [16]. Sur le modèle donné figure 1.8, trois couleurs notées [1],[2] et [3] sur le schéma sont obtenues. Cette coloration sert à intégrer les places dans des macroplaces. Le réseau de Petri hiérarchique obtenu à partir du RdP exemple utilisé est donné figure 1.9. L'utilisation de macroplaces a pour but de diminuer le nombre de variables nécessaires pour décrire l'état du RdP (il n'est plus nécessaire d'avoir une variable pour chaque place). L'implémentation du RdP sur le FPGA est d'autant plus efficace en termes de surface.

- La méthode CONPAR proposée dans [31] utilise, quant à elle, un langage pivot dédié, le CONPAR, pour effectuer l'implémentation d'un RdP sur FPGA. Dans ce langage CONPAR, les RdP interprétés sont traduits en spécifications « rule-based », composées de symboles d'états discrets et de signaux d'entrée et de sortie. Les règles de transitions d'états discrets décrivent un changement d'état local. Une transition est décrite comme une règle conditionnelle : *<label> : <Preconditions>* $\vdash$ *<Post-Conditions>* ; . Les préconditions et les postconditions sont respectivement formées

par les symboles des places d'entrée et de sortie. Quand la précondition d'une règle est satisfaite, les postconditions sont déclarées vraies.

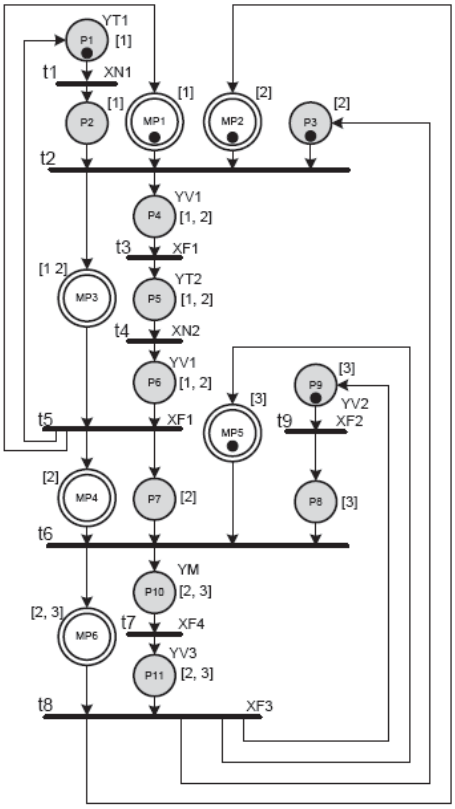

FIGURE 1.9 – RdP hiérarchisé obtenu grâce à la coloration du RdP donné figure 1.8

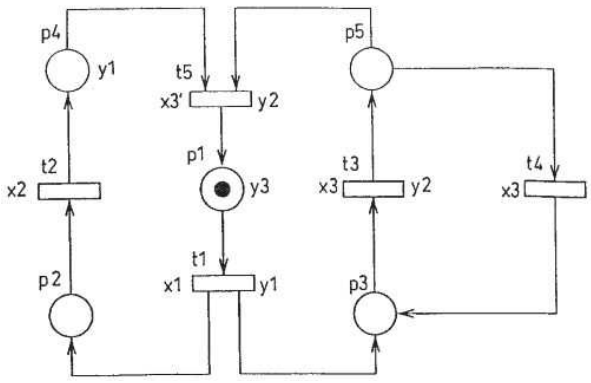

FIGURE 1.10 – Exemple de RdP utilisé pour la méthode CONPAR[31]

Considérons le RdP donné en exemple figure 1.10 : pour la transition t_1 , nous obtenons : *t1 : p1 * x1 ⊢ p2 * p3 * y1 ;* . La traduction en VHDL obtenue est alors : *t1 <= x1 AND p1 AND NOT p2 AND NOT p3 ;* . Les préconditions d'une transition sont directement traduites via une expression booléenne en VHDL. Les postconditions sont distribuées parmi les expressions VHDL *Npi* qui sont des signaux VHDL qui servent au calcul du marquage suivant des places. Par exemple, nous avons *Np1 <= t5 OR (p1 AND NOT t1);*. Pour chaque signal de sortie, un signal concurrent

est utilisé. Ainsi nous obtenons *y1 <= p4 OR t1 ;*. Cette méthode s'approche en fait des méthodes transformant le RdP en système de Gentzen. C'est principalement le formalisme qui est différent entre les séquents de Gentzen et le langage CONPAR, mais la philosophie de traduction est la même.

- L'outil HPetriNets utilisé par Silva dans [47] propose, quant à lui, de transformer le RdP en matrices. Silva utilise ici la représentation matricielle pour en déduire le code VHDL correspondant. 7 matrices sont utilisées pour caractériser le RdP :
  - La matrice *preincidence TI* et la matrice *postincidence TO* qui décrivent respectivement les arcs entrants et sortants des transitions.
  - La matrice *input map E* et la matrice *output map A* qui représentent respectivement les valeurs des variables d'entrées et les valeurs des variables de sorties.
  - La matrice *transition logic F* qui représente les conditions d'entrée, de sortie et internes associées à chaque transition.
  - La matrice *current marking CP* et la matrice *next marking NP* qui indique le marquage actuel du RdP et le marquage au prochain front d'horloge.

  Le principe est alors de calculer la matrice A grâce aux autres matrices. Ceci fournira alors les équations logiques décrivant l'évolution des variables de sorties. Par exemple, si nous considérons le RdP étudié dans l'article [47] et donné figure 1.11, la matrice A obtenue est la suivante :

$$A = \begin{pmatrix} (f_4 \& cp_4 \& cp_5)|(cp_1 \& \bar{f}_1)|(f1 \& cp_1)|(cp_3 \& \bar{f}_3) \\ (f_4 \& cp_4 \& cp_5)|(cp_1 \& \bar{f}_1)|(f3 \& cp_3)|(cp_5 \& \bar{f}_4) \end{pmatrix}$$

  Le détail des opérations à réaliser pour calculer A sont indiqués dans [47].

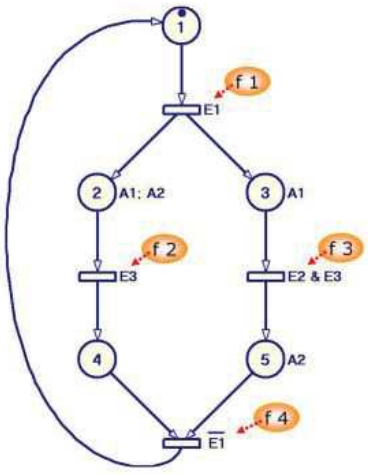

FIGURE 1.11 – Exemple de RdP implémenté à l'aide de l'outil HPetriNets [47]

L'avantage de cette méthode est d'être automatisable sans problème et déjà outillée. Les matrices à manipuler peuvent être importantes dans le cas d'un modèle complexe mais les calculs sont fait hors-ligne et automatiquement. Le temps de calcul ne sera pas problématique même pour de grandes matrices d'autant plus que les opérations à réaliser sur les matrices sont simples (pas d'inverse de matrice à calculer par exemple).

- La méthode proposée par Varshavsky dans [55] propose une traduction en composants des structures élémentaires du RdP telles que la séquence, la synchronisation ou le parallélisme. Les structures élémentaires sont directement traduites en portes logiques, schémas qui peuvent ensuite être implémentés sur FPGA. Des exemples de traduction de structures standards sont donnés figure 1.12.

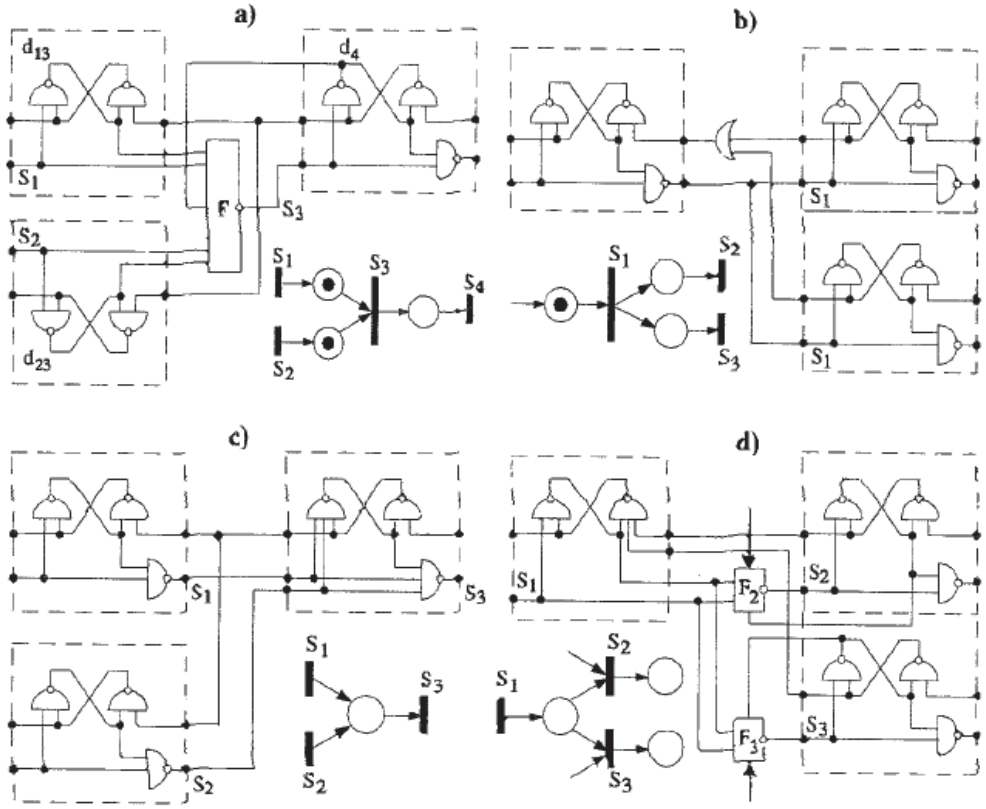

FIGURE 1.12 – Exemples de traduction de structures standards dans la méthode de Varshavsky [55]

La méthode est intéressante parce qu'elle permet de réduire le nombre de portes logiques nécessaires à l'implémentation du RdP. Elle permet aussi d'avoir un fonctionnement rapide mais, de l'aveu des auteurs, le comportement du réseau implémenté peut ne pas être correct dans le sens où il ne correspond pas au comportement décrit à l'aide du RdP ce qui n'est pas acceptable dans notre contexte. Un autre inconvénient de cette méthode est que comme les types de structures sont prédéfinies, elle impose intrinsèquement des contraintes structurelles sur le modèle.

- Certaines méthodes traduisent le RdP en traduisant chacune des places et des transitions qui sont ensuite reliées ensemble en respectant la structure du RdP. Parmi ces méthodes, nous pouvons citer celle présentée par Uzam dans [54]. Cette méthode est basée sur la transformation du RdP directement en circuit logique et non en code VHDL. Le circuit logique peut ensuite être facilement implémenté sur FPGA. Le principe est d'établir une description générique à l'aide d'un circuit logique d'une place et d'une transition (cf. Fig. 1.13 et Fig. 1.14). Il s'agit ensuite d'instancier ces descriptions et de les relier suivant la structure du RdP pour obtenir le circuit logique correspondant au RdP.

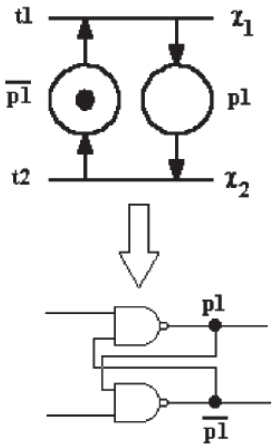

FIGURE 1.13 – Transformation d'une place en circuit logique dans la méthode d'Uzam [54]

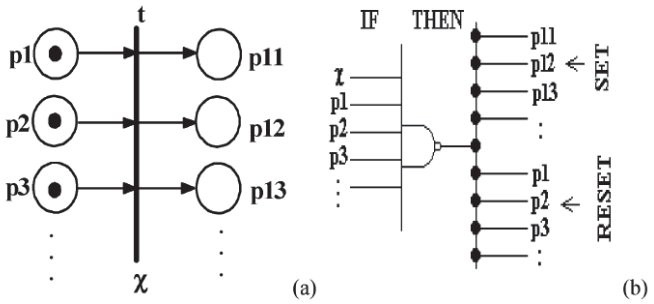

FIGURE 1.14 – Transformation d'une transition en circuit logique dans la méthode d'Uzam [54]

Les transitions sont traduites à l'aide d'une porte NAND ayant en entrée la condition et les signaux provenant des places entrantes. Or les portes NAND implémentables sur FPGA ont un nombre d'entrées limitées (entre 3 et 8 en fonction du FPGA utilisé). Si la transition possède trop de places entrantes, il est toujours possible de découper cette porte NAND en plusieurs portes NAND ayant moins d'entrées mais ce découpage peut entraîner un retard dans les signaux et conduire ainsi à une évolution incorrecte du RdP. Ce décalage est montré dans le cas simplifié présenté figure 1.15 où nous avons supposé les portes NAND limitées à 2 entrées et une transition ayant 3 places entrantes. Le chronogramme illustre le retard de traitement entre les différentes places et l'évolution erronée du RdP qui peut en découler. Il peut exister des solutions pour gérer ce problème de retard mais elles dépendent du circuit à réaliser et sont donc difficilement automatisables. La structure des RdP implémentables grâce à cette solution est donc limitée : une transition ne peut pas avoir trop de places entrantes (la limite étant déterminée par le FPGA utilisé).

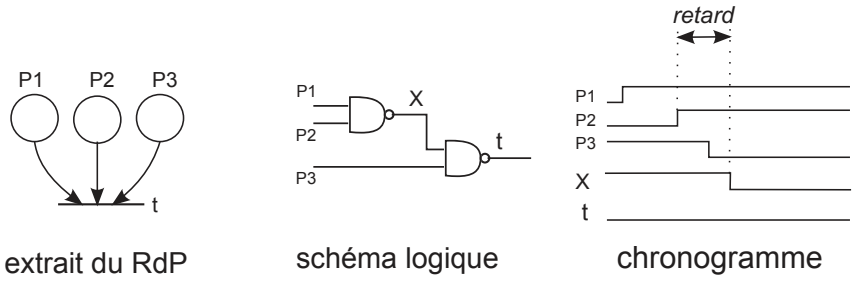

FIGURE 1.15 – Cas simplifié d'une transition nécessitant plusieurs portes NAND

- Nous retrouvons ensuite plusieurs méthodes qui traduisent chaque place et chaque transition mais cette fois-ci à l'aide de composants VHDL : la méthode présentée par Combacau dans [21] ou celle présentée par Doligalski dans [28] mais aussi la méthode existante d'HILECOP [7] ainsi que la méthode d'Hiles [33]. Le principe est qu'un composant VHDL place et un composant VHDL transition sont décrits. Ensuite pour générer le code VHDL correspondant au RdP, les places et les transitions nécessaires qui sont reliées en fonction de la structure du RdP sont instanciées. Pour plus de détails concernant cette méthode de traduction en VHDL, le lecteur est invité à consulter le chapitre 2.

Ainsi, il existe de nombreuses approches permettant d'implémenter un RdP sur FPGA. Nous avons pu remarquer que pour certaines [55] [54], il était nécessaire que la structure des RdP satisfassent certaines hypothèses. Nous ne nous choisirons pas l'approche proposée par Varshavsky dans [55] car le comportement du FPGA peut ne pas être correct ce qui n'est pas compatible avec une utilisation pour des systèmes critiques. Pour la méthode d'Uzam et les autres méthodes, l'un des critères prépondérants qui permet de les différencier par rapport à leur pertinence dans notre contexte, est la classe des RdP pouvant être implémentée grâce à ces méthodes. Néanmoins, à capacité et performances égales, une traduction directe du RdP vers un code VHDL sera privilégiée plutôt qu'une traduction nécessitant un langage pivot pour éviter une étape supplémentaire non nécessaire.

Classe de RdP utilisable

Dans notre contexte, nous avons besoin d'implémenter un RdP généralisé étendu interprété T-temporel contenant potentiellement des choix. Etudions la réponse apportée à ces caractéristiques par les différentes méthodes proposées dans le paragraphe précédent. Un tableau récapitulatif (table 1.1) est fourni dans le §1.3.2.

généralisé : La grande majorité des méthodes ne permettent d'utiliser que des réseaux sauf, c'est-à-dire des réseaux dans lesquels les arcs ne peuvent avoir qu'un poids de 1 et dont le marquage des places ne peut jamais excéder 1. Cela permet effectivement de simplifier l'implémentation du RdP mais limite le pouvoir d'expression du modèle et donc les possibilités de modélisation du concepteur. Les seules méthodes permettant l'utilisation de RdP généralisés sont la méthode proposée par Combacau dans [21] et la méthode HILECOP. Dans ces 2 méthodes, un composant VHDL décrivant la place est utilisé pour réaliser l'implémentation du RdP. Au lieu de définir le marquage d'une place par un booléen comme cela est fait dans les autres méthodes, le marquage est défini comme un entier naturel. Dans ces méthodes, définir le marquage comme un entier naturel est relativement simple car il s'agit d'un type de variables géré par le VHDL. Par exemple cela présenterait plus de complexité dans les méthodes utilisant les séquents de Gentzen [52] ou le langage CONPAR [31] car ces logiques ne manipulent que des booléens.

étendu : Quatres méthodes gèrent les arcs tests et les arcs inhibiteurs : la méthode présentée par Uzam dans [54], la méthode CONPAR [31], la méthode Hiles [33] et la méthode HILECOP. La méthode proposée par Silva [47] déclare pouvoir gérer les arcs inhibiteurs mais n'explique pas comment ces arcs inhibiteurs sont traduits dans les matrices.

La méthode d'Uzam proprose une traduction automatique d'un arc test et d'un arc inhibiteur en portes logiques. Ces traductions sont données respectivement figure 1.16 et figure 1.17.
Dans le langage CONPAR, si une transition est ciblée par un arc test ou inhibiteur, un prédicat est utilisé dans la spécification de la transition. Si les structures de RdP données figure 1.18 sont considérées, les spécifications suivantes sont obtenues : $t4 : p4A \times pred4 \vdash p4B$; et $t5 : p5A \times pred5 \vdash p5B$; avec $pred4 = p4C$ et $pred5 = !p5C$.

Dans la méthode Hiles et celle d'HILECOP, le type de chaque arc est codé dans le composant contenant l'arc (dans notre cas le composant place concerné). Cela permet, suivant le type de chaque arc, de déterminer la sensibilisation de la transition et le nombre de jetons à ajouter ou retirer aux différentes places. Pour plus de détails, le lecteur est amené à lire le chapitre 2.

Pour conclure, la méthode d'Uzam implique que la structure contenant des arcs tests et inhibiteurs corresponde à la structure pouvant être traduite automatiquement, il y a donc une limitation sur la structure du RdP implémentable comme nous l'avions déjà noté précédemment. Il n'y a en revanche aucune limite sur les autres solutions proposées pour traduire les arcs tests et inhibiteurs.

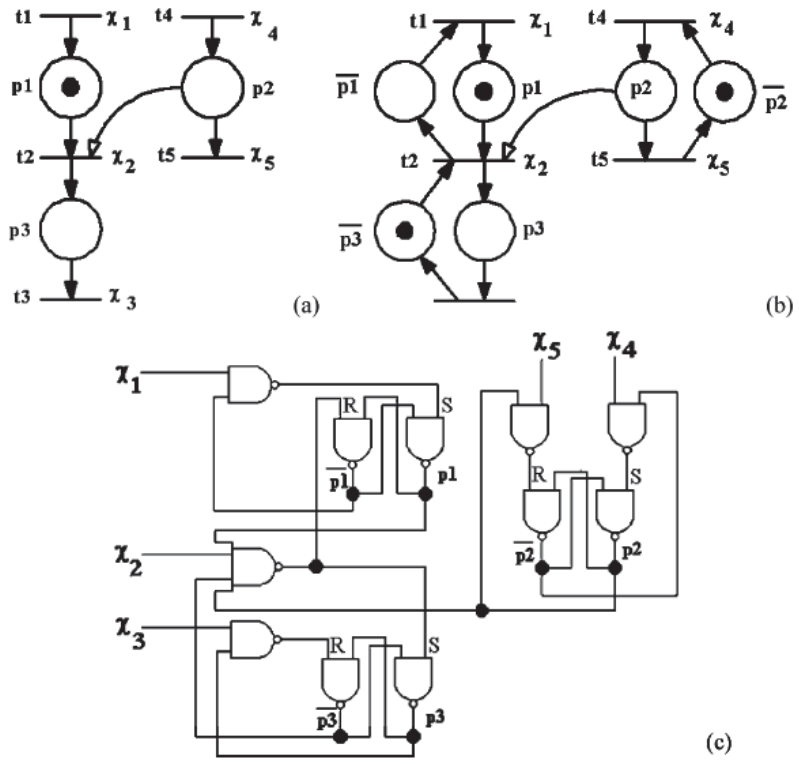

FIGURE 1.16 – Traduction d'un arc test dans la méthode d'Uzam [54]

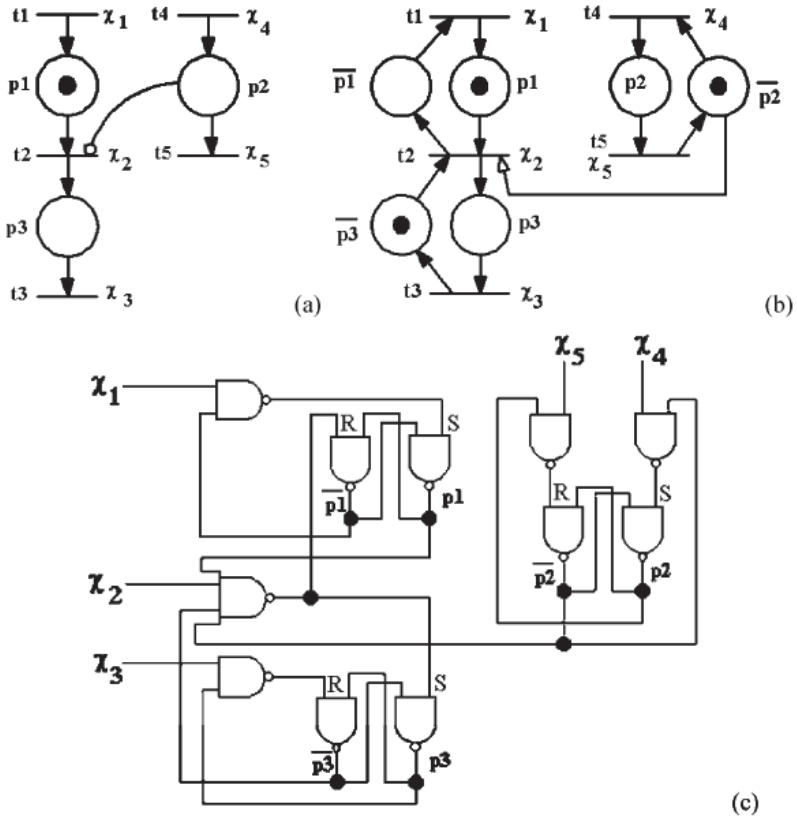

FIGURE 1.17 – Traduction d'un arc inhibiteur dans la méthode d'Uzam [54]

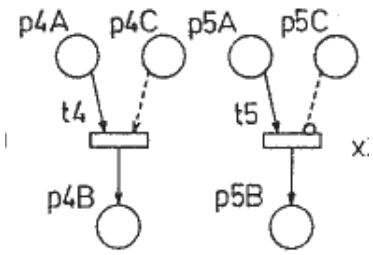

FIGURE 1.18 – Exemples de structures avec un arc test et un arc inhibiteur [31]

interprétation : Toutes les méthodes présentées permettent de gérer une interaction entre le modèle et l'extérieur à l'aide de signaux. Elles permettent toutes de réaliser des actions (i.e. le modèle réalise une action continue sur la variable d'un signal) et d'influer sur l'évolution du modèle à l'aide de conditions associées aux transitions. Par contre, seules certaines permettent d'utiliser des fonctions, c'est-à-dire de modifier des signaux quand une transition est tirée : la méthode de Silva [47], la méthode d'Uzam [54], la méthode CONPAR [31], la méthode Hiles [33] et la méthode HILECOP.

Dans la méthode de Silva, les seules actions impulsionnelles réalisables sur le tir d'une transition sont l'incrémentation ou la réinitialisation d'une variable. Cela permet de réaliser aisément des compteurs qui sont souvent utilisés dans le contrôle de systèmes. Les auteurs n'offrent pas d'autres possibilités pour les fonctions mais comme les compteurs sont gérés dans un process VHDL, des fonctions plus générales pourraient être envisagées.

Dans la méthode d'Uzam, l'implémentation des actions et des fonctions est similaire dans le sens où elles sont toutes deux lancées par un signal venant respectivement d'une place ou du tir d'une transition. Or rien n'est décrit dans l'article pour permettre de déterminer la fin de l'exécution d'une fonction. Mais si les fonctions influent sur la valeur des conditions, il est nécessaire d'attendre qu'une fonction soit terminée pour déterminer l'évolution suivante du RdP pour avoir un comportement correct. Nous en déduisons que l'hypothèse suivante est nécessaire bien que non indiquée dans l'article : les fonctions n'influencent que les signaux externes et non les conditions. Une autre solution est de rendre tous les réseaux T-temporisés pour permettre de retarder l'évolution du RdP suffisamment pour que les fonctions aient le temps de s'exécuter.

Dans la méthode CONPAR, un signal peut être associé à une transition pour modifier la valeur de celui-ci quand une transition est tirée mais ceci est traité comme une action, i.e. comme un signal associé à une place. D'ailleurs un même signal peut être associé soit à une place soit à une transition. Ainsi sur la figure 1.10, nous retrouvons dans le code CONPAR : $t1 : p1 \times x1 \vdash p2 \times p3 \times y1$ pour l'association du signal à la transition $t1$ et dans le code VHDL, nous obtenons $y1 \Leftarrow$ *p4 OR t1 ;*. L'avantage est donc de pouvoir associer la modification de signaux externes soit au marquage d'une place soit au tir d'une transition mais il n'est pas possible dans cette méthode de réaliser des fonctions (traitements).

La méthodologie Hiles et la méthodologie HILECOP fonctionnent de la même manière. Un process est associé aux fonctions et le tir d'une des transitions auxquelles est associée une fonction lance l'exécution de la fonction (cf. chapitre 2).

temps : Peu de méthodes permettent de gérer des contraintes temporelles. Seules les méthodes d'Hiles et d'HILECOP permettent d'associer un intervalle de temps à une transition. Un compteur de temps est alors ajouté au composant transition et une transition temporelle ne peut être tirée que quand la valeur du compteur de temps est incluse dans l'intervalle de temps associé à la transition (cf. chapitre 2).

La méthode présentée par Uzam permet d'implémenter des RdP T-temporisés. Un RdP temporisé est un RdP où il est possible d'associer un délai à une transition. Quand une transition T-temporisée est tirée, les jetons des places entrantes sont dits réservés pendant le délai, c'est-à-dire qu'ils restent dans les places entrantes mais ne peuvent plus être utilisés pour le tir d'autres transitions. Une fois le délai atteint, la transition est effectivement tirée, c'est-à-dire que les jetons sont enlevés des places entrantes et ajoutés aux places sortantes. La méthode ne présente une solution pour les RdP temporisés que dans le cas d'une implémentation sur des FPGA Xilinx car la solution dépend du FPGA choisi. Le problème est que ce type de modélisation du temps ne permet pas de modéliser autant de contraintes que la modélisation du temps dans les RdP T-temporels. Les chiens de garde ne peuvent, par exemple, pas être modélisés dans un RdP T-temporisé.

La méthode présentée par Silva permet, elle, d'ajouter des timers associés aux transitions. En réalité, un même timer va être relié à au moins 2 transitions. Le tir de la première transition déclenche le "chronomètre" qui permettra d'évaluer si le temps du timer est atteint ou non. Le tir de la seconde condition ne sera possible qu'une fois la durée décrite par le timer atteint. Le comportement est donc sensiblement différent du comportement obtenu avec un intervalle de type $[a, a]$ dans un RdP temporel. Ce n'est en effet pas la sensibilisation de la transition qui déclenche le chronomètre mais le tir d'une transition qui peut très bien n'avoir aucun lien avec la transition qui pourra être tirée lorsque le temps imparti se sera écoulé.

conflits : La plupart des méthodes font l'hypothèse que le RdP est sans conflit effectif, c'est-à-dire que le fonctionnement du réseau est nécessairement déterministe, sans proposer de solution pour la gestion des structures de choix. Au contraire, les méthodes d'Uzam [54], CONPAR [31] et de Combacau [21] proposent des solutions pour gérer les choix dans un modèle RdP.

Dans la méthode d'Uzam, deux solutions d'implémentation sont proposées. Dans les deux cas, la transition tirée parmi celles en conflit est prédéterminée. La première solution (Fig. 1.19) est d'ajouter une structure au RdP permettant de donner la priorité à chaque transition alternativement. La seconde (Fig. 1.20) est d'ajouter une condition à chaque transition du conflit telle qu'une seule condition puisse être vraie parmi toutes les conditions associées aux transitions. L'article ne précise pas comment ces variables sont gérées, laissant le choix au concepteur. Le problème est que, dans les 2 cas, la transition tirée est choisie indépendamment du fait qu'elle soit

effectivement tirable. Ainsi si deux transitions sont potentiellement en conflit effectif mais qu'une seule est effectivement franchissable (parce que sa condition est fausse par exemple) alors que ce n'est pas la transition prévue par la résolution du choix, aucune transition ne sera tirée. Dans le pire des cas, un blocage peut même être créé.

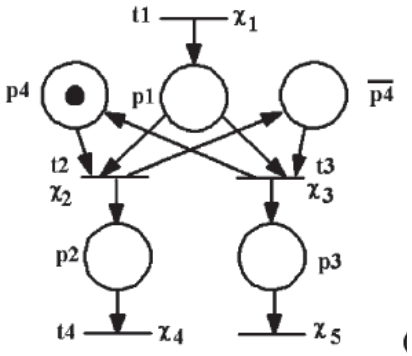

FIGURE 1.19 – Première solution de gestion de conflit [54]

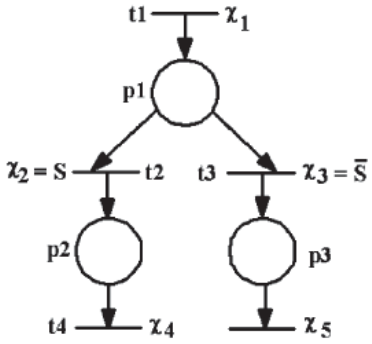

FIGURE 1.20 – Seconde solution de gestion de conflit [54]

Dans la méthode CONPAR, le concepteur est invité à utiliser des arcs inhibiteurs pour gérer les conflits structurels détectés lors de l'analyse. Il est donc toujours de la responsabilité du concepteur de gérer les conflits et il ne peut pas y avoir de conflit effectif dans le modèle final implémenté.

Dans la méthode de Combacau, le code VHDL pour décrire le comportement dans le cas d'un choix ayant des conditions mutuellement exclusives est présenté. Par contre, le cas où les conditions ne sont pas mutuellement exclusives n'est pas géré. Il est donc de la responsabilité du concepteur de s'assurer que ce cas ne se produise pas.

Ainsi, aucune des méthodes étudiées ne présente de solution de gestion des conflits qui satisfasse nos attentes. Nous souhaiterions en effet une solution permettant de gérer les conflits de manière automatique et garantissant qu'une transition franchissable soit toujours franchie.

Approche de mise en œuvre

Le dernier grand critère qui permet de distinguer les différents méthodes existantes est l'approche de mise en œuvre. Deux grandes approches de mise en œuvre peuvent être considérées sur FPGA : la mise en œuvre asynchrone et la mise en œuvre synchrone. Une mise en œuvre synchrone signifie que l'évolution du RdP est cadencée par une horloge au contraire de la mise en œuvre asynchrone où l'évolution se fait dès que les événements permettant l'évolution se produisent. Parmi les articles étudiés précédemment, trois seulement proposent une approche asynchrone : [54] [55] [33].

L'avantage d'une approche asynchrone est que le modèle des RdP étant asynchrone, par définition, la correspondance de comportement entre le modèle conçu et le modèle implémenté est assez directe. Mais, lorsque un RdP est implémenté sur un FPGA, il faut toujours s'assurer que les signaux utilisés pour faire évoluer le modèle sont bien stables lors de leur évaluation. Si ce n'est pas le cas, le comportement observé sur le FPGA ne correspond pas nécessairement au comportement conçu. Cette nécessité est vraie quelle que soit l'approche. Mais, dans l'approche synchrone, il est possible de vérifier, grâce aux environnements de développement (programmation) fournis par les fabricants de FPGA, le temps maximum nécessaire pour que tous les signaux du FPGA se stabilisent. Si la fréquence d'horloge est choisie pour que cette contrainte de temps soit satisfaite, il est alors possible d'assurer la stabilité des signaux. Ce n'est pas aussi simple en asynchrone. C'est pourquoi la majorité des méthodes optent pour une approche synchrone.

Néanmoins l'approche synchrone présente également des inconvénients. En premier lieu, l'utilisation d'une horloge mène à une consommation de courant du FPGA plus importante. D'un point de vue fonctionnement du circuit, un inconvénient est une réactivité du circuit moindre en synchrone qu'en asynchrone. En implémentation synchrone, il est en effet nécessaire de baser l'horloge au minimum sur le temps maximum possible pour attendre la stabilisation des signaux. Or il n'est pas toujours nécessaire d'attendre aussi longtemps pour faire évoluer le système (puiqu'il s'agit du temps maximum). Le circuit synchrone passe donc du temps à attendre sans rien faire alors que le circuit asynchrone va pouvoir réagir immédiatement à un nouvel événement. Un autre inconvénient de l'implémentation synchrone est que les modèles RdP sont définis comme asynchrone. Il est alors plus difficile de s'assurer que le comportement observé sur le FPGA correspondra au comportement décrit à l'aide d'un RdP. Considérons maintenant les méthodes proposant des solutions pour implémenter les RdP en asynchrone.

Comme dit précédemment lors de l'étude de la gestion de l'interprétation, dans la méthode d'Uzam [54], rien ne garantit a priori que les fonctions ont fini de s'exécuter avant de continuer à faire évoluer le modèle. Néanmoins, comme cette méthode propose aussi d'implémenter des RdP temporisés, il est possible de prendre en compte ce temps d'exécution des fonctions en retardant l'évolution du RdP à l'aide d'une temporisation. La temporisation peut alors être adaptée aux fonctions à exécuter. Mais, qui dit temporisation, dit nécessité d'avoir une horloge et alors l'avantage de ne pas avoir d'horloge est perdu. Si les temporisations sont adaptées aux fonctions à exécuter, une réactivité optimale du RdP est conservée mais cela ne peut pas être automatisé et nécessite l'expertise du concepteur. Si nous souhaitons avoir une solution automatisable, il faut alors associer la plus grande temporisation à toutes les transitions et dans ce cas nous perdons l'avantage de l'asynchronisme. En outre, il est à noter que les auteurs indiquent en perspective que bien qu'ils

présentent ici une méthode asynchrone, ils souhaitent orienter leur recherche vers une implémentation synchrone plus adaptée d'après eux aux FPGA actuellement commercialisés.

Dans la méthode présentée dans [55], les auteurs ont modifié la sémantique des RdP pour prendre en compte leur implémentation complètement asynchrone. En effet, l'évolution des RdP est définie comme asynchrone au sens où aucune horloge ne gère le moment de tir des transitions. Par contre, lors du tir des transitions, il est bien supposé que le retrait des jetons des places amonts et l'ajout des jetons dans les places aval sont réalisés de manière instantanée et donc synchrone. Dans cette méthode, il est alors défini que les ajouts des jetons sont réalisés avant les retraits des jetons. Cette approche, si utile pour l'implémentation, nous paraît contre-intuitive et surtout délicate à mettre en place dans le cadre des réseaux de Petri temporels. En effet, pour ces RdP, la gestion des compteurs est définie de telle manière que les retraits sont réalisés par définition avant les ajouts (cf. chapitre 2). De plus, de l'aveu des auteurs, les phénomènes transitoires ne sont pas toujours gérés de manière satisfaisante dans leur implémentation, ce qui peut mener à un comportement non souhaité. Par exemple, le fait de réaliser d'abord les ajouts puis les retraits peut entraîner le tir de transitions qui n'auraient pas dû être tirables. Ainsi, dans le RdP présenté figure 1.21, si la transition S_6 est tirée, la transition S_5 ne devrait plus pouvoir être tirée tout de suite après. Or, puisque l'ajout est d'abord réalisé, un jeton est ajouté dans P_4 alors qu'il y a toujours un jeton dans P_3 et le tir de S_5 peut alors être observé.

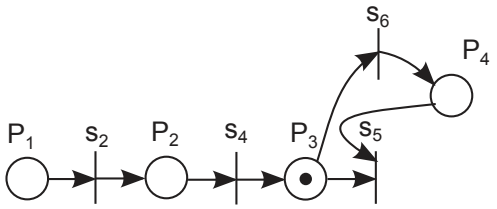

FIGURE 1.21 – Exemple de RdP menant à un comportement non souhaité

Dans la méthodologie Hiles [33], le choix est laissé au concepteur d'implémenter son RdP de manière synchrone ou asynchrone. Dans le cas asynchrone, un temps d'attente est prévu dans les composants places et transitions. Celui-ci doit être configuré par le concepteur pour assurer la stabilité des signaux.

Ainsi il aurait été très intéressant de pouvoir implémenter le RdP de manière asynchrone, mais la gestion des retards dus au temps de propagation des signaux est trop complexe pour parvenir à réaliser une bonne implémentation automatique d'un RdP interprété. Une telle implémentation est toujours possible mais elle doit être réalisée grâce à l'expertise d'un concepteur. C'est pourquoi nous avons choisi de réaliser l'implémentation du RdP de manière synchrone. Le principe de l'implémentation synchrone est similaire dans toutes les méthodes citées et est présenté chapitre 2.

Il sera alors notamment nécessaire dans notre méthode de modifier la sémantique des RdP utilisés pour traduire le synchronisme de l'implémentation du modèle conçu afin de pouvoir conserver une correspondance entre le comportement conçu et le comportement obtenu sur FPGA.

1.3.2 Conclusion

Pour commencer, la possibilité d'analyser le modèle qui sera implémenté fait partie intrinséquemment de nos préoccupations, mais pas des critères utilisés pour étudier les différents méthodes existantes. Ceci est justifié par le fait que ce critère ne permet pas de les distinguer car aucune ne se penche particulièrement sur ce problème. En effet, si beaucoup d'articles mentionnent le côté analysable des modèles conçus à l'aide des RdP pour justifier l'utilisation des RdP, peu parmi eux se posent la question de la correspondance entre modèle conçu, modèle implémenté et modèle analysé. Par exemple, la méthode de Tzacz [51] aborde l'analyse dans le but de déterminer si le RdP est vivant ou non. La solution proposée est d'analyser la structure du RdP sans l'interprétation. C'est la solution couramment utilisée lorsque l'analyse d'un RdP interprété est réalisée. Or l'implémentation est réalisée de manière synchrone alors que les RdP sont définis comme asynchrone. Le problème est qu'il a été démontré dans [24] que, dans le cas général, ce n'est pas parce qu'un RdP asynchrone est vivant que le RdP synchronisé ayant la même structure est vivant.

De plus, il a été démontré que le comportement d'un RdP interprété est inclus dans celui du même réseau sans l'interprétation. Mais cette inclusion n'est pas l'équivalence et il serait intéressant d'essayer de se rapprocher autant que possible de l'équivalence. Cette problématique n'est pas abordée, à notre connaissance, dans les articles traitant de l'implémentation des RdP.

Le tableau 1.1 récapitule les différentes approches mentionnées, issues de la littérature. Cela permet de réaliser que si aucune méthode de traduction ne satisfait complètement les contraintes de notre contexte (y compris la version existante d'HILECOP), certaines ont des particularités néanmoins intéressantes pour notre contexte. Il est donc nécessaire d'adapter les approches existantes, dont celle d'HILECOP, et définir ainsi une "nouvelle" approche d'implémentation des RdP sur FPGA.

méthode	pivot	G/S	arcs	temps	conflit	synchrone
Tzac [52]	✓	S	C	✗	✗	✓
Silva [47]	✓	S	CI	timers	✗	✓
Uzam [54]		S	CTI	T-temporisé	≈	✗
Fernandes [31]	✓	S	CTI	✗	≈	✓
Combacau [21]		G	C	✗	≈	✓
Doligalski [30]		S	C	✗	✗	✓
HILES [33]		S	CTI	T-temporel	✗	2 possibles
HILECOP [49]		G	CTI	T-temporel	✗	✓

TABLE 1.1 – Bilan de l'étude des méthodes existantes
légende : S = sauf G = généralisé C = classique T = test I = inhibiteur
≈ = pris en compte mais pas satisfaisant

1.4 Objectifs et outils

1.4.1 Objectifs et contraintes

Notre principal objectif est d'offrir aux concepteurs un moyen d'assurer que le comportement du FPGA conçu sera conforme à celui souhaité et fiable dans les limites atteignables. Une autre préoccupation importante pour nous est de s'assurer que la méthode soit la plus pratique possible pour les concepteurs et tende à minimiser le risque d'erreur humaine. L'idée est en effet que cette méthode soit effectivement utilisée dans le cadre industriel de la conception et réalisation de systèmes numériques critiques.

Pour atteindre ces objectifs, il faut tout d'abord permettre l'analyse formelle du comportement. L'analyse formelle permettra au concepteur de s'assurer que le comportement du modèle sera sûr quelle que soit l'évolution du système. Ce dernier pourra par exemple vérifier qu'une place ne contiendra jamais de jetons si une transition donnée est tirée grâce au model-checking [14] ou que le marquage d'une place ne dépassera jamais une valeur donnée grâce à l'analyse structurelle [44].

Il faut néanmoins s'assurer que ces résultats aient un sens par rapport au comportement du modèle implémenté dans le FPGA. Le modèle analysé sera en effet un modèle basé RdP donc inspiré du modèle conçu mais le modèle implémenté est décrit à l'aide de VHDL et peut donc avoir un comportement différent du modèle conçu (cf. figure 1.1). Il faut d'une part que le modèle conçu et le modèle implémenté aient bien le même comportement aussi bien d'un point de vue logique que d'un point de vue temporel (les deux influençant l'évolution du modèle). D'autre part, il faut que le modèle analysé soit construit tel que le comportement du modèle conçu (et donc du modèle implémenté) soit toujours inclus dans le comportement du modèle analysé.

Par ailleurs, pour pouvoir gérer l'implémentation sur FPGA dans le contexte des systèmes embarqués, voire implantés, il est nécessaire d'avoir une méthode menant à une implémentation efficace, c'est-à-dire une implémentation demandant le moins de cellules logiques possibles et consommant le moins d'énergie possible.

En outre, il est important d'apporter aux concepteurs un moyen pratique pour décrire le comportement attendu en cas d'exceptions. Cette gestion d'exceptions doit naturellement aussi pouvoir être implémentée de manière efficace et le modèle conçu doit toujours être analysable.

1.4.2 Outils utilisés pour l'implémentation et l'analyse

La méthode doit permettre d'avoir une implémentation efficace et de réaliser une analyse formelle. Il est aussi nécessaire de pouvoir étudier aussi bien le comportement du modèle implémenté que celui du modèle analysé pour pouvoir les comparer. La méthode HILECOP fournit en sortie un code VHDL et un code PNML. Des outils extérieurs sont donc nécessaires aussi bien pour l'implémentation du modèle implémenté que pour l'analyse du modèle analysé (cf. figure 1.6). Le concepteur peut choisir les outils qu'il souhaite car le langage VHDL et le langage PNML sont acceptés respectivement par un grand nombre d'outils d'implémentation et d'analyse. Pour conduire nos travaux, nous avons

néanmoins choisi deux logiciels spécifiques : Libero [65] pour l'implémentation et Tina [62] pour l'analyse. Tous deux permettant également l'étude du comportement du modèle (simulation pour le premier et résultats d'analyse pour le second), il ne sera pas nécessaire d'utiliser d'autres outils. Leurs possibilités au regard de nos besoins sont résumées ici.

Libero Libero est un logiciel développé par la société Microsemi qui vend aussi les FPGA IGLOO. Ce logiciel a été choisi car notre équipe utilise les FPGA IGLOO dans le cadre de leurs travaux. Le choix du type de FPGA utilisé sera expliqué chapitre 4.

Le logiciel Libero permet de programmer un FPGA à partir d'un code VHDL. Pour cela, plusieurs opérations intermédiaires sont nécessaires : la synthèse, la compilation et le placement/routage (cf. Fig. 1.22). Le code est d'abord transformé en un schéma RTL (Register Transfer Level) standard, c'est-à-dire en un schéma constitué de portes logiques. Ce schéma est ensuite transformé en schéma RTL technologique. Le RTL technologique est toujours un schéma comportant des portes logiques mais cette fois seules les portes logiques existantes dans le FPGA cible sont utilisées. La dernière étape avant l'implémentation est le placement /routage qui permet d'obtenir le layout, c'est-à-dire le placement exact de toutes les portes logiques décrites dans le RTL technologique sur une représentation graphique du FPGA.

Lors de ces différentes transformations, le logiciel aura pour but de simplifier au maximum l'architecture obtenue en supprimant par exemple les signaux non utilisés ou en fusionnant les signaux ayant exactement la même évolution. Ceci permet de minimiser le nombre de cellules logiques nécessaires à l'implémentation du code VHDL.

Lors du processus d'implémentation, Libero donne le nombre de cellules logiques utilisées sur le FPGA. A noter que Libero ne réalise pas toujours exactement les mêmes simplifications, une légère variation peut donc être observée sur ce nombre de cellules lors de différentes compilations d'un même code initial. Ainsi Libero nous permet d'avoir une information qualitative sur l'efficacité d'une solution d'implémentation par rapport à une autre en termes de nombre de cellules nécessaires. Dans la suite de ce manuscrit, nous parlerons souvent, par simplification, d'efficacité en termes de surface plutôt qu'en termes de nombre de cellules nécessaires sur le FPGA. Il s'agit d'un abus de langage, puisqu'un FPGA donné fait toujours la même surface quelle que soit la manière dont il est programmé, mais cela permet de simplifier le propos. Par contre, le nombre de cellules logiques nécessaires va influer sur la taille de FPGA qu'il sera possible de choisir.

Libero, en plus de permettre l'implémentation sur FPGA, offre aussi la possibilité de réaliser des simulations. Les simulations peuvent être faites au niveau du code VHDL comme au niveau de l'architecture RTL qui sera effectivement implémentée dans le FPGA. La simulation permet d'observer l'évolution des variables de sortie en fonction des variables d'entrée au cours du temps. Pour effectuer les simulations, un fichier VHDL est utilisé décrivant l'évolution des valeurs des entrées souhaitée, notamment la vitesse de l'horloge. Il est aussi possible de forcer les valeurs des différentes variables d'entrée manuellement lors de la simulation. La simulation réalisée non pas sur le code VHDL mais sur l'architecture après placement et routage permet d'avoir une image réaliste et fidèle du comportement obtenu sur le FPGA programmé. L'avantage de la simulation par rapport à l'observation

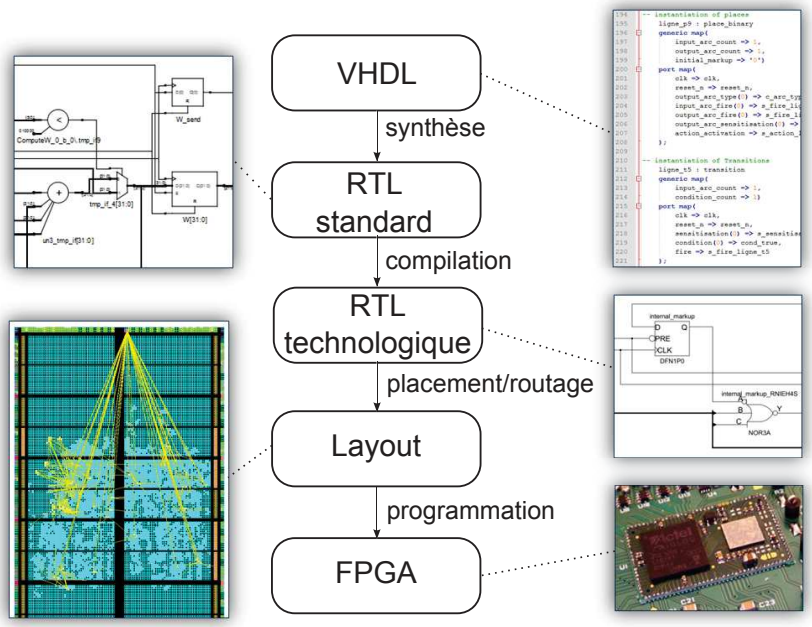

FIGURE 1.22 – Etapes de la transformation du VHDL pour son implémentation sur FPGA

du comportement du FPGA réel est la possibilité d'observer des variables internes sans avoir à modifier le comportement pour transformer ces variables en signaux de sortie. Nous aurons donc recours à cette solution notamment quand nous voudrons inspecter des propriétés temporelles précises.

La simulation permet aussi d'avoir des valeurs indicatives de consommation. La valeur obtenue après simulation du code après placement et routage ne correspond pas toujours aux valeurs mesurées effectivement sur le FPGA. Par contre, les variations de valeurs observées entre deux modèles implémentés différents sont réalistes. Ainsi il est aussi possible d'avoir une information qualitative sur l'efficacité d'une solution d'implémentation par rapport à une autre en termes de consommation. Ceci étant, nous ferons aussi des mesures effectives de consommation via une plateforme dédiée, développée à cette fin au sein de l'équipe. Nous pourrons donc comparer les résultats obtenus par simulation et par mesure effective.

En conclusion, le logiciel Libero nous permettra de vérifier que les contraintes d'efficacité en termes de surface et de consommation sont respectées mais aussi d'avoir une image du comportement du système implémenté afin de le comparer avec celui du modèle analysé.

Tina Le logiciel TINA [12] [62] est un logiciel d'analyse des RdP développé par le laboratoire de recherche LAAS. Il permet notamment de réaliser l'analyse de réseaux de Petri généralisés étendus T-temporels avec priorités. TINA offre deux types d'analyse formelle : l'analyse structurelle et l'analyse d'accessibilité. TINA offre aussi un outil de simulation qui permet de simuler l'évolution du réseau de Petri soit de manière automatique (le logiciel choisit de manière aléatoire la transition tirée en cas de conflit ou de concurrence A) ou de manière manuelle (l'utilisateur choisit la transition tirée).

L'analyse structurelle consiste à étudier la structure du RdP indépendamment de son marquage. Elle détermine, s'ils existent, des invariants de places et de transitions [24]. Les invariants de places (issus des composantes conservatives) sont définis tels que la somme des marquages des places d'un invariant est toujours constante au cours de l'évolution du RdP. Les invariants de places permettent d'affirmer que les places incluses dans un invariant sont bornées. Par extension, si toutes les places du réseau sont couvertes par au moins un invariant, il est possible de conclure que le réseau de Petri est borné. Par exemple, dans le RdP donné figure 1.23, les places P0 et P1 sont dans un invariant ce qui implique que le réseau est borné. La valeur de la borne du marquage de chaque place dépend bien entendu du marquage initial du réseau. Par contre, ce n'est pas parce qu'une place n'est pas dans un invariant de places qu'elle n'est pas bornée. Par exemple, dans le RdP donné figure 1.24, la place P4 n'est couverte par aucun invariant de place et pourtant le réseau est borné et vivant.

Les composantes répétitives stationnaires indiquent, quant à elles, les transitions à franchir pour revenir à un marquage identique. Les invariants de transitions sont les composantes répétitives stationnaires telles que le marquage initial du réseau permette le tir de toutes les transitions de la composante répétitive stationnaire. Si toutes les transitions d'un RdP sont couvertes par au moins un invariant de transition alors le réseau est vivant. Ce n'est pas le cas pour les composantes répétitives stationnaires puisque la vivacité d'un RdP dépend nécessairement de son marquage initial . Par exemple, le RdP donné figure 1.23 est vivant avec le marquage initial donné par contre il ne l'est pas si le marquage initial est nul pour toutes les places.

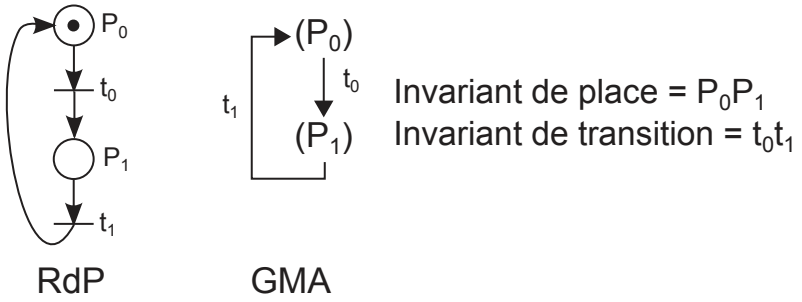

FIGURE 1.23 – Exemple de RdP borné et vivant couvert par un invariant de places et de transitions et son graphe des marquages accessibles

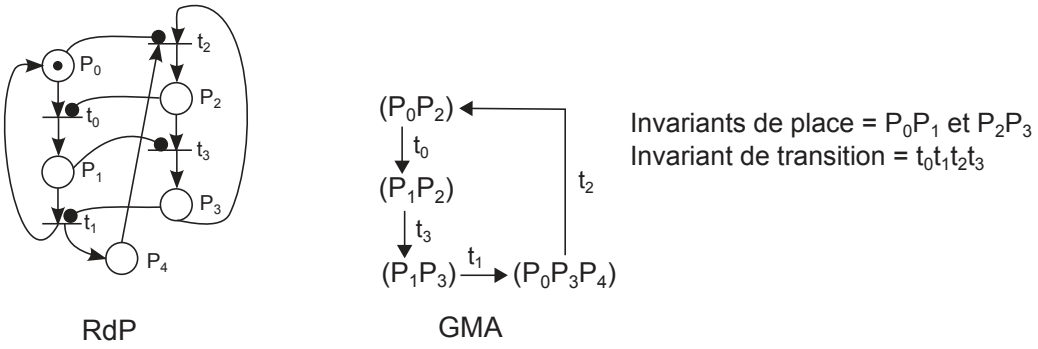

FIGURE 1.24 – Exemple de RdP borné et vivant non couvert par les invariants de place et son graphe des marquages accessibles

L'analyse d'accessibilité consiste à déterminer et à étudier le graphe des marquages accessibles (GMA) (pour les RdP non-temporels) ou graphe des classes d'états (GCE) (pour les RdP temporels) du RdP à partir du marquage initial [15]. Ces graphes représentent l'ensemble des états accessibles (ou espaces d'états) du RdP. Des exemples de GMA sont donnés figure 1.23 et figure 1.24. Ainsi, comme tous les états possibles du RdP sont connus, il est bien sûr possible de déterminer si un RdP est borné. A noter que si un RdP n'est pas borné, il n'est pas possible d'obtenir son GMA (ou GCE) puisqu'il est infini. De même, comme toutes les séquences de tirs possibles sont connues il est possible de déterminer si un RdP est vivant ou non.

L'analyse structurelle n'en reste pas moins utile car le GMA (ou GCE) n'est pas toujours possible à obtenir soit parce qu'il est infini, soit parce qu'il est trop difficile à déterminer avec les outils et ressources que nous utilisons. Il y a en effet un risque d'explosion combinatoire lié à l'exhaustivité intrinsèque de la méthode.

L'analyse d'accessibilité permet, si le GMA (ou GCE) est obtenu, de réaliser du model-checking [14]. Le model-checking est utilisé notamment pour s'assurer que des propriétés comportementales, exprimées en logique temporelle sur l'espace d'états du système, sont vérifiées. Par exemple, il est possible de vérifier la propriété suivante : « un groupe de places donné est toujours vidé (c'est-à-dire l'ensemble des places ont un marquage nul) après le tir d'une transition ».

Les logiques usuellement utilisées sont LTL et CTL, ou TCTL pour les propriétés temporelles. Mais il n'est pas possible sur TINA de vérifier directement des propriétés comportementales temporisées (TCTL) comme : « un groupe de places donné est toujours vidé en moins de 3 unités de temps après le tir d'une transition donnée ». Néanmoins, il est possible de vérifier ce genre de propriétés si le réseau est modifié en lui ajoutant des observateurs A ce qui permet de transformer les propriétés temporisées en propriétés non temporisées.

Ainsi l'analyse par model-checking nous permet aussi de visualiser le comportement du modèle analysé au travers du prisme des propriétés comportementales qu'il vérifie ou non. A noter que TINA nous indique si une propriété est vérifiée ou non mais qu'il fournit également un contre-exemple si la propriété n'est pas vérifiée en fournissant la séquence de tirs ne vérifiant pas la propriété.

Pour conclure, l'étude du contexte nous a permis de préciser nos objectifs et nos contraintes en plaçant au centre de nos préoccupations le concepteur. L'analyse de l'existant ne nous a pas permis de trouver de solutions existantes satisfaisantes dans notre contexte. Ce travail de thèse s'est donc attaché à combler les manques identifiés afin de fournir une méthodologie efficace, fiable et outillée pour la conception de systèmes numériques complexes. Nous allons d'abord nous intéresser dans le chapitre suivant à la transformation automatique et fiable de modèle décrit à l'aide de réseaux de Petri en modèle implémentable et analysable. Une fois ces transformations maîtrisées nous verrons ensuite comment améliorer le formalisme utilisé pour gérer efficacement les exceptions et l'application de ces modifications à un cas industriel.

Chapitre 2

Assurer la conformité entre conception, implémentation et analyse

Dans notre méthodologie, le modèle décrit par le concepteur à l'aide de réseau de Petri doit être transformé en code VHDL implémentable sur FPGA et en code PNML analysable. Réaliser une transformation automatique et fidèle du modèle conçu à l'aide de réseau de Petri généralisé étendu interprété T-temporel synchrone à priorités (RdP GEITSP) en modèle implémentable et en modèle analysable est complexe. Il y a en effet beaucoup de paramètres à prendre en compte pour assurer la correspondance entre les 3 modèles. Les difficultés seront expliquées les unes après les autres dans le but d'être le plus clair possible même si cela risque d'apporter une certaine redondance. Ainsi, au lieu d'aborder directement la transformation d'un RdP GEITSP, nous travaillerons d'abord sur une classe plus simple : les RdP généralisés étendus interprétés synchrones (GEIS) et ajouterons ensuite les conflits puis les intervalles temporels.

Rappelons ici que le modèle analysable et le modèle implémentable ne peuvent pas être traités complètement indépendamment l'un de l'autre. En effet, le comportement du modèle analysable doit correspondre à celui du modèle implémenté et donc la méthode d'implémentation va influer sur la génération du modèle analysable. De même, nous verrons que les résultats d'analyse sont nécessaires pour garantir une implémentation fiable et efficace du modèle conçu.

2.1 Gestion de l'interprétation et du synchronisme

La première problématique présentée est la gestion de l'interprétation. Nous verrons pourquoi l'interprétation nous pousse à implémenter le modèle de manière synchrone. Ce synchronisme est la deuxième problématique à étudier. Nous verrons alors comment un RdP GEIS peut être défini, implémenté et analysé.

2.1.1 Description de l'interprétation et du synchronisme

Présentation de l'interprétation

Pour pouvoir interagir avec l'extérieur, il est nécessaire d'utiliser des RdP interprétés. Dans notre contexte, l'interprétation du RdP consiste en des conditions, des actions et des fonctions. L'interprétation manipule des signaux et cette manipulation doit donc être

décrite à l'aide d'un langage. Puisqu'ensuite le modèle sera transformé en VHDL, l'interprétation est décrite directement en langage VHDL.

Les conditions permettent de faire dépendre l'évolution du RdP des signaux externes ou des variables internes. Une condition est nécessairement une expression logique entre des variables ou signaux VHDL. Quand la condition est nulle, c'est-à-dire si la valeur de son expression logique est nulle, alors la transition ne peut pas être tirée. Par contre, la valeur de la condition n'influe pas sur la sensibilisation de la transition qui ne dépend que du marquage du RdP. Une même condition peut être associée à plusieurs transitions. Il est aussi possible d'associer à une transition la négation d'une condition.

Les actions (continues) permettent de manipuler des signaux externes ou des variables internes et sont associées aux places. Une action est exécutée dès et tant que le marquage de la place à laquelle elle est associée est non nul. Une action est nécessairement booléenne dans le sens où l'action ne peut pas dépendre du marquage de la place. Une même action peut être associée à plusieurs places. Dans ce cas, l'action est exécutée dès et tant qu'au moins une des places auxquelles elle est associée est marquée. Plusieurs actions peuvent aussi être associées à la même place.

Les fonctions permettent aussi de manipuler des signaux externes ou des variables internes mais sont associées aux transitions. Les fonctions sont en fait des actions de type impulsionnel. L'incrémentation d'un compteur est un exemple classique d'actions impulsionnelles. Une même fonction peut être associée à plusieurs transitions ou plusieurs fonctions à une même transition. Une fonction est exécutée dès qu'au moins une des transitions auxquelles elle est associée est tirée. Elle n'est exécutée qu'une seule fois quand plusieurs transitions auxquelles elle est associée sont simultanément tirées.

Principe de l'implémentation synchrone

Nous avons vu au chapitre 1 différentes solutions pour implémenter un RdP de manière asynchrone. Or en ajoutant l'interprétation, pour pouvoir implémenter le RdP de manière asynchrone, il est nécessaire de pouvoir aussi gérer l'interprétation de manière asynchrone. La méthode d'Uzam [54] (cf. chap 1) nous montre qu'il est possible de lancer les actions ou les fonctions de manière asynchrone assez facilement. Mais il n'est jamais vérifié quand les actions ont été lancées ou les fonctions exécutées. Les actions étant associées à l'état du réseau, leurs activations et désactivations sont simples à gérer (affectation d'une valeur aux signaux concernés). Par contre, pour les fonctions, il est important de savoir quand la fonction se termine. En effet, les fonctions manipulant les variables internes, elles peuvent modifier les valeurs des conditions. Or il est nécessaire, pour avoir une évolution fiable et déterministe, de pouvoir assurer que les signaux sont stables quand les conditions sont à évaluer. Pour cela, il faut que l'exécution des fonctions soient toutes terminées quand les conditions sont évaluées. Par exemple, dans le cadre du RdP présenté figure 2.1(a), l'évolution du RdP n'est pas celle souhaitée quand la décision de tir de t_1 est prise avant la fin de l'exécution de la fonction (figure 2.1(b)). Par contre, si l'évaluation de la condition c_1 est réalisée une fois les signaux stabilisés, l'évolution est bien correcte (figure 2.1(c))

Il est tout à fait possible d'assurer qu'une fonction est bien terminée avant d'évaluer une condition sur un circuit asynchrone. Néanmoins cela signifie de connaître les délais de propagation des signaux et ceci ne peut donc être vérifié qu'une fois le placement et

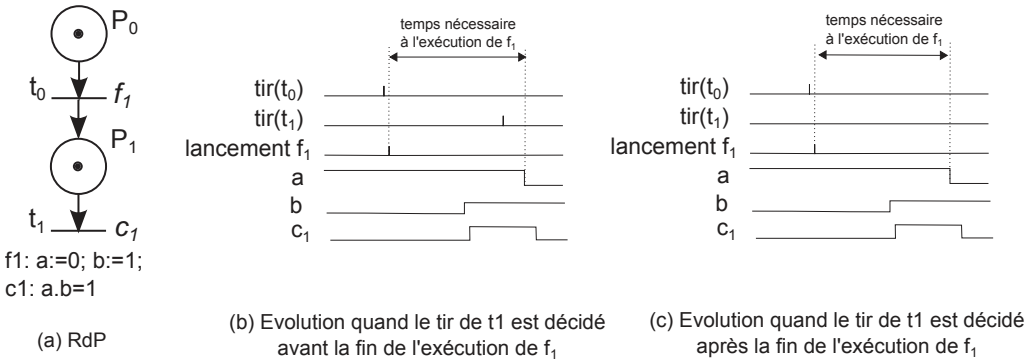

FIGURE 2.1 – Nécessité d'attendre la fin de l'exécution des fonctions

routage du circuit effectué. Si jamais cette condition n'est pas respectée, il est possible de modifier le circuit pour ajuster les délais. Mais comme le placement et routage est modifié, il faut de nouveau s'assurer que la propriété de stabilité est bien vraie sur le nouveau circuit. Si ce n'est pas le cas, il faut de nouveau modifier le circuit jusqu'à obtenir une solution satisfaisante. Ainsi, si avoir une implémentation asynchrone d'un RdP interprété est envisageable, avoir une implémentation automatique ne l'est pas. Une autre solution pour s'assurer que les signaux sont stables dans le cadre d'une implémentation automatisable est d'implémenter le RdP de manière synchrone. Cela a déjà été fréquemment réalisé dans la littérature [52] [47] [31] [21] [30](cf. chapitre 1).

Le principe de l'implémentation synchrone est que l'évolution du RdP est cadencée par une horloge (cf. figure 2.2). Dans notre approche, les 2 fronts du signal d'horloge sont exploités. Sur le front descendant de l'horloge ①, les transitions qui doivent être tirées durant ce cycle d'horloge sont déterminées. Une transition est tirée dès qu'elle est tirable, c'est-à-dire dès qu'elle est sensibilisée et que sa condition est vraie. Les conditions sont évaluées en permanence mais leur valeur est considérée au niveau du front descendant de l'horloge ①. Sur le front montant suivant ③, le marquage des places est mis à jour. Une fois le marquage mis à jour, les prochaines transitions à tirer sont déterminées ① et ainsi de suite.

Les fonctions sont lancées sur le front montant suivant la décision de tir d'une transition ③. Les fonctions s'exécutent pendant ④. Elles doivent nécessairement s'exécuter en moins d'une demi-période d'horloge pour avoir une évolution correcte. En effet, si l'exécution est plus longue, les signaux ne seront pas stables lors de la nouvelle évaluation des conditions en ①. Cette hypothèse peut être vérifiée lors de la synthèse du FPGA. Elle s'exprime en termes de fréquence maximale d'évolution. Les actions sont lancées/stoppées au front descendant ① suivant la modification du marquage qui les a déclenchées/arrêtées(③). Décaler le lancement des actions et des fonctions d'une demi-période d'horloge permet de s'assurer que les signaux de tirs et de marquages seront stables quand ils seront utilisés pour déterminer quelles actions et quelles fonctions exécuter/lancer.

Si la décision d'implémenter le RdP en synchrone est une décision courante, elle n'est néanmoins pas anodine puisqu'elle va impacter directement le comportement du modèle et donc sa sémantique. En effet, dans le cadre d'une évolution asynchrone, l'hypothèse suivante était notamment posée : une seule transition est franchie à la fois. Cette hypothèse n'est plus vraie en synchrone, il n'est donc plus possible de garder exactement la même

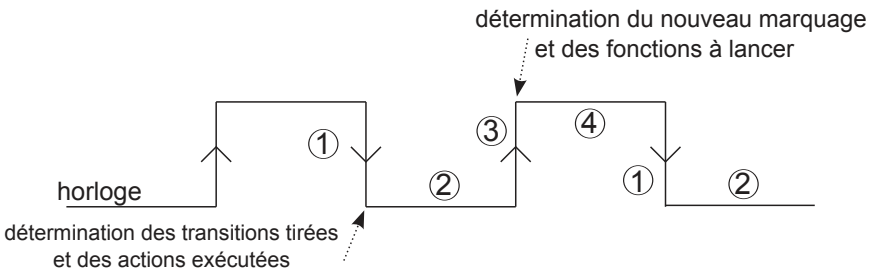

FIGURE 2.2 – Principe de l'implémentation synchrone

sémantique que pour les RdP asynchrones. De plus, dans le cadre des RdP ordinaires, une transition tirable pouvait être tirée ou ne pas l'être. Dans le cadre d'une implémentation synchrone, le RdP étant synchronisé, toute transition tirable est obligatoirement tirée à l'occurrence de l'événement qui lui est associé (ici un front d'horloge)[24]. Les tirs de transitions simultanés posent problème dans le cas d'un conflit. Il est supposé pour l'instant que les RdP étudiés ne contiennent aucun conflit (effectif), c'est-à-dire que le tir d'une transition n'empêche jamais le tir d'une autre transition qui était simultanément tirable. La problématique de la gestion des conflits sera étudiée dans le §2.2. Il faut donc pour l'instant établir une nouvelle définition et sémantique formelle pour les RdP interprétés synchrones sans conflit.

2.1.2 Définition formelle et règles sémantiques

Nous définissons formellement ci-dessous la classe des RdP généralisés étendus interprétés synchrones (RdP GEIS) sans conflit dont un exemple de modèle est donné figure 2.3. Cette définition est inspirée de celles proposées pour d'autres classes de RdP dans [14] et [53].

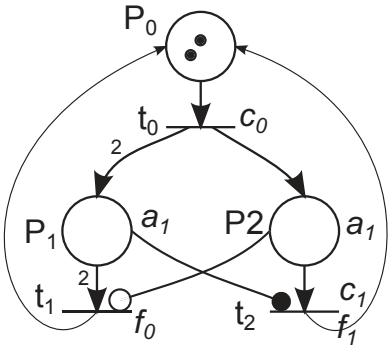

FIGURE 2.3 – Exemple de RdP GEIS

Definition 2.1.1. *Soient $\mathcal{C}$ l'ensemble des conditions, $\mathcal{F}$ l'ensemble des actions impulsionnelles (fonctions de l'interprétation) et $\mathcal{A}$ l'ensemble des actions continues. $\mathcal{C}$, $\mathcal{A}$ et $\mathcal{F}$ interagissent avec des signaux extérieurs (entrées et sorties du système) et/ou manipulent des variables internes du système.*

Un RdP généralisé étendu interprété synchrone sans conflit est un uplet $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, clock >$, *où :*

- $< P, T, Pre, Pre_t, Pre_i, Post, m_0 >$ *est un réseau de Petri généralisé étendu.* P *est l'ensemble des places,* T *l'ensemble des transitions et* m_0 *le marquage initial.* $Pre, Pre_t, Pre_i,\ Post : T \to P \to \mathbb{N}$ *sont respectivement la fonction précondition, la fonction test, la fonction inhibition et la fonction postcondition.*
  La fonction précondition définit les arcs classiques place-transition, la fonction test les arcs tests, la fonction inhibition les arcs inhibiteurs et la fonction postcondition les arcs classiques transition-place. Les fonctions Pre, Pre_t, Pre_i *sont telles que deux arcs ne peuvent pas avoir même source et même destination :*
  – $\forall t \in T, \forall p \in P, Pre(t)(p) \neq 0 \Rightarrow Pre_t(t)(p) = 0 \wedge Pre_i(t)(p) = 0$
  – $\forall t \in T, \forall p \in P, Pre_t(t)(p) \neq 0 \Rightarrow Pre(t)(p) = 0 \wedge Pre_i(t)(p) = 0$
  – $\forall t \in T, \forall p \in P, Pre_i(t)(p) \neq 0 \Rightarrow Pre(t)(p) = 0 \wedge Pre_t(t)(p) = 0$
- $C : T \to \mathcal{C} \to \{-1, 0, 1\}$ *est la fonction condition.* $\forall t \in T, \forall c \in \mathcal{C}, C(t)(c) = 1$ *signifie que la condition* c *est associée à* t. $C(t)(c) = -1$ *signifie que la négation de la condition* c *est associé à* t. $C(t)(c) = 0$ *signifie que la condition* c *n'est pas associée à* t.
- $F : T \to \mathcal{F} \to \mathbb{B}$ *est la fonction des action impulsives.* $\forall t \in T, \forall f \in \mathcal{F}, F(t)(f) = 1$ *signifie que la fonction* f *est associée à* t *sinon on a* $F(t)(f) = 0$.
- $A : P \to \mathcal{A} \to \mathbb{B}$ *est la fontion des actions continues. Elle est définie sur le même principe que* F.
- *clock est le signal d'horloge synchronisant le RdP.* $\uparrow$ *clock, le front montant de l'horloge et* $\downarrow$ *clock son front descendant sont les événements de l'horloge. L'ensemble des événements de l'horloge est noté Clk.*

Le marquage du RdP est défini par la fonction $m : P \to \mathbb{N}$. Une transition t est dite sensibilisée par m, noté $t \in sens(m)$ si et seulement si $\big((m \geq Pre(t) + Pre_t(t)\big) \wedge \big(m < Pre_i(t)\big)$.

La valeur instantanée d'une condition est définie par la fonction $val : \mathcal{C} \to \mathbb{B}$. La valeur d'une condition fixée pour étudier l'évolution du modèle est définie par la fonction $cond : \mathcal{C} \to \mathbb{B}$. L'exécution d'une action ou d'une action impulsive est définie par la fonction $ex : \mathcal{F} \cup \mathcal{A} \to \mathbb{B}$.

L'état d'un RdP généralisé étendu interprété synchrone est défini par $s = (m, cond, ex)$.

Une transition t est tirable depuis l'état s du RdP, notée $t \in tirable(s)$ si et seulement si $\big[t \in sens(m)\big] \wedge \big[\forall c \in \mathcal{C} | C(t)(c) = 1, cond(c) = 1\big] \wedge \big[\forall c \in \mathcal{C} | C(t)(c) = -1, cond(c) = 0\big]$.

La sémantique des RdP généralisés étendus interprétés synchrones sans conflit peut maintenant être définie.

Definition 2.1.2. *La sémantique d'un RdP généralisé étendu interprété synchrone sans conflit* $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, clock >$ *est le système de transition temporisé* $< S, s_0, \rightsquigarrow >$ *où :*

- S *est l'ensemble des états* $(m, cond, ex)$ *du RdP.*
- $s_0 = (m_0, 0, 0)$ *est l'état initial où* 0 *est la fonction nulle, c'est-à-dire la fonction qui associe 0 à tous les éléments d'un ensemble.*

- $\rightsquigarrow \subseteq S \times Clk \times S$ *est la relation de transition d'états, notée* $s \overset{clk}{\rightsquigarrow} s'$ *avec* $clk \in Clk$, *définie comme suit :*
  Soit $Tir(s) \subseteq T$ *l'ensemble des transitions tirées depuis l'état* s. *Cet ensemble ne peut être modifié que sur un front descendant de l'horloge. A l'état initial, on a* $Tir(s_0) = \varnothing$.
  - *On a* $s = (m, cond, ex) \overset{\downarrow clock}{\rightsquigarrow} s' = (m, cond', ex')$ *si et seulement si* $\downarrow clock = 1$ *et :*
    1. $\forall c \in \mathcal{C}, cond'(c) = val(c)$ *(actualisation des valeurs des conditions)*
    2. $\forall a \in \mathcal{A}, \exists p \in P | A(p)(a) = 1 \wedge m(p) \neq 0 \Rightarrow ex'(a) = 1$, *sinon* $ex'(a) = 0$ *(actualisation de la fonction exécution pour les actions continues)*

    Il est alors possible de déterminer $Tir(s')$ *l'ensemble des transitions qui seront tirées.*
    3. $\forall t \in tirable(s'), t \in Tir(s')$ *(les transitions tirables seront tirées)*
    4. $\forall t \notin tirable(s'), t \notin Tir(s')$ *(les transitions non tirables ne seront pas tirées)*
  - *On a* $s = (m, cond, ex) \overset{\uparrow clock}{\rightsquigarrow} s' = (m', cond, ex')$ *si et seulement si* $\uparrow clock = 1$ *et :*
    1. $m' = m - \sum_{t \in Tir(s)} Pre(t) + \sum_{t \in Tir(s)} Post(t)$ *(actualisation du marquage)*
    2. $\forall f \in \mathcal{F}, \exists t \in Tir(s) | F(t)(f) = 1 \Rightarrow ex'(f) = 1$, *sinon* $ex'(f) = 0$ *(actualisation de la fonction exécution pour les actions impulsives)*

2.1.3 Transformation en VHDL

Principe de la transformation

Le comportement attendu du RdP conçu ayant été défini en tenant compte de sa mise en œuvre (exécution), il faut maintenant parvenir à réaliser une transformation automatisée en modèle implémentable qui respecte cette sémantique (cf. figure 2.4). Comme nous l'avons vu au chapitre 1, nous avons choisi le VHDL comme langage de description du modèle implémentable. Pour rappel, nous nous plaçons dans le cadre d'une synthèse globale.

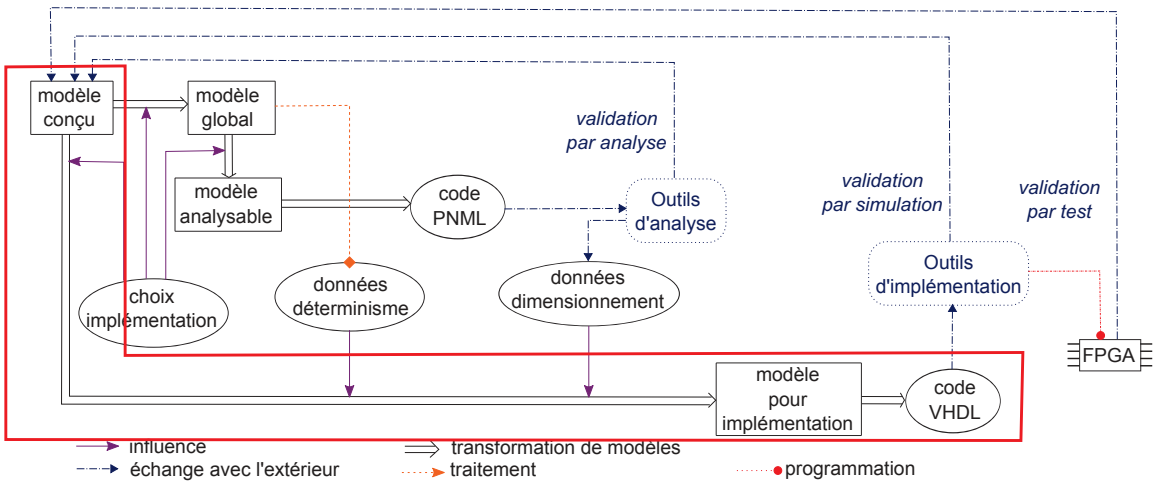

FIGURE 2.4 – Transformation du modèle implémentable en code VHDL dans la méthodologie HILECOP

La proposition faite par la méthodologie HILECOP consiste à exploiter l'orientation composant du langage en définissant deux composants VHDL structurels de base : un composant VHDL place et un composant VHDL transition. Les arcs sont eux intégrés aux places. Les places sont donc le pivot de notre méthode de traduction car elles intègrent les interconnexions du graphe. Ce choix sera justifié une fois les 2 composants définis. La méthode de traduction automatique correspond, dans un premier temps, à la création des instances des composants élémentaires pour chaque place et chaque transition du modèle. Dans un second temps, ces différentes instances sont interconnectées entre elles pour traduire la structure du modèle complet. Enfin, l'interprétation du modèle (qui se trouve déjà sous la forme de code VHDL) est ajoutée à la traduction.

Les règles d'évolution du modèle sont réparties entre les deux types de composants afin de respecter l'approche composant, i.e. chacun assure l'évolution de son propre état. Le composant place met à jour et conserve son marquage. Il détermine de plus si la place sensibilise ou non ses transitions aval. Le composant transition détermine si la transition doit être tirée ou non. Le composant transition et le composant place sont schématiquement représentés figure 2.5.

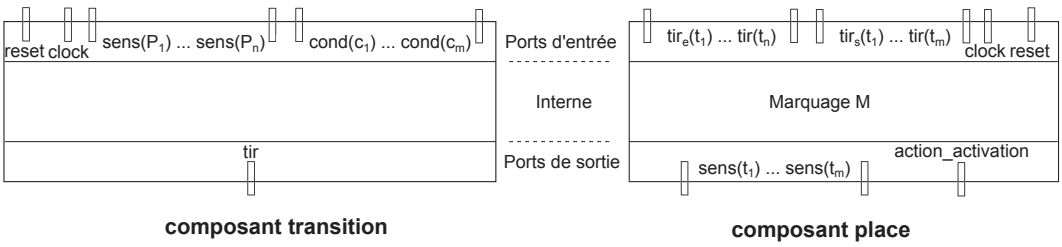

FIGURE 2.5 – Représentation schématique des composants VHDL

Le composant transition est structuré comme suit :

- Un port d'entrée constitué d'un vecteur $sens$ de dimension n, avec n le nombre d'arcs entrants de la transition, d'un vecteur $cond$ de dimension m où m est le nombre de conditions associées à la transition, d'un signal $clock$, horloge du composant et d'un signal $reset$. Chaque élément $sens(P_i)$ du vecteur $sens$ désigne si la place amont P_i sensibilise ou non la transition. Chaque élément $cond(c_i)$ du vecteur $cond$ permet de connaître la valeur de la condition c_i.
- Une zone interne dans laquelle un process détermine si la transition est tirée ou non en fonction des signaux des ports d'entrée (valués au front descendant de l'horloge $\downarrow clock$).
- Un port de sortie constitué d'un signal tir indiquant si la transition est tirée ou non. Ce signal est également utilisé pour déclencher les fonctions associées à la transition.

Le composant place est structuré comme suit :

- Un port d'entrée constitué de deux vecteurs tir_e de dimension n et tir_s de dimension m, avec n le nombre d'arcs entrants de la place et m le nombre d'arcs sortants, d'un signal $clock$, horloge du composant et d'un signal $reset$.
- Une zone interne comportant :
  - le marquage interne de la place.
  - le process permettant l'évaluation du marquage interne en fonction des signaux des ports d'entrée (valués au front montant de l'horloge $\uparrow clock$).

```
entity transition is
generic(nombre_arc_entrants      : natural := 1;
        nombre_conditions        : natural := 1);
port(   clk                  : in std_logic;
        reset_n              : in std_logic;
        sensibilisation      : in std_logic_vector(0 to nombre_arc_entrants-1);
        condition            : in std_logic_vector(0 to nombre_conditions-1);
        tir              : out std_logic);
end;

architecture a_transition of transition is
begin
process(clk, reset_n, sensibilisation, condition)
    variable sens : std_logic;
    variable cond : std_logic;
begin
    cond := '1';
    sens := '1';
    for f in 0 to nombre_conditions-1 loop -- test condition
        cond := cond and condition(f);
    end loop;
    for g in 0 to nombre_arc_entrants-1 loop -- test sensibilisation
        sens := sens and sensibilisation(g);
    end loop;
    if (reset_n = '0') then
        tir <= '0';
    elsif (clk'event and clk='0') then
        -- evaluation du signal tir
        if ((sens = '1') and (cond = '1')) then
            tir <= '1';
        else
            tir <= '0';
        end if;
    end if;
end process;
end a_transition;
```

FIGURE 2.6 – Code VHDL du composant transition

  - l'évaluation du signal de sensibilisation des transitions aval en fonction du marquage interne et du poids des arcs.
- Un port de sortie constitué d'un vecteur *sens* de dimension m indiquant si la place sensibilise ses transitions aval, qui sera transmis aux transitions concernées et un signal binaire *action_activation* qui gère l'activation des actions associées à la place.

Le code VHDL des composants élémentaires est donné figure 2.6 et figure 2.7.

Une fois les instances des différentes places et transitions créées à partir des composants élémentaires définis ci-dessus, elles sont interconnectées. Les signaux de sensibilisation des places (port de sortie de la place) sont reliés aux ports d'entrée des transitions concernées et les signaux de tir des transitions (ports de sortie de la transition) sont reliés aux ports d'entrée des places concernées. Pour exemple, la figure 2.8 représente l'interconnexion entre les différentes instances des composants place et transition dans le cas de la traduction du RdP donné figure 2.3.

Un process gère les actions, un autre gère les fonctions et un dernier gère les conditions. Le principe du process des actions est d'évaluer à chaque front descendant, la valeur des signaux d'activation d'action issus des places et suivant leur valeur les actions sont lancées, arrêtées ou inchangées. Le principe est le même pour les fonctions sauf que l'évaluation est réalisée sur front descendant. Le process condition évalue de manière

```
entity place is

generic(nombre_arc_entrants     : natural := 1;
        nombre_arc_sortants : natural := 1;
        initial_markup      : natural := 0;
        max_markup          : natural := mark_max_net);
port(   clk                 : in std_logic;
        reset_n             : in std_logic;
        type_arcs_sortants      : in vect_at(0 to nombre_arc_sortants-1);
        poids_arcs_entrants : in vect_link(0 to nombre_arc_entrants-1);
        poids_arcs_sortants : in vect_link(0 to nombre_arc_sortants-1);
        tirs_arcs_entrants      : in std_logic_vector(0 to nombre_arc_entrants-1);
        output_arc_tir  : in std_logic_vector(0 to nombre_arc_sortants-1);
        sensibilisations_arcs_sortants : out std_logic_vector(0 to nombre_arc_sortants-1);
        action_activation   : out std_logic);
end;

architecture a_place of place is
signal marquage_interne : natural range 0 to max_markup;
begin
process(clk, reset_n)
    variable marquage_local : natural range 0 to max_markup;
    variable somme_ajout : natural range 0 to max_markup;
    variable somme_retrait : natural range 0 to max_markup;
begin
    if (reset_n = '0') then
        marquage_interne <= initial_markup; -- initialisation marquage
        sensibilisations_arcs_sortants <= (others => '0');
        action_activation <= '0';
    elsif (clk'event and clk='1') then
        marquage_local := marquage_interne;
        somme_ajout := 0;
        somme_retrait := 0;
        for i in 0 to nombre_arc_entrants-1 loop -- ajout jeton
            if (tirs_arcs_entrants(i) = '1') then
                somme_ajout := somme_ajout + poids_arcs_entrants(i);
            end if;
        end loop;
        for j in 0 to nombre_arc_sortants-1 loop -- retrait jeton
            if (output_arc_tir(j) = '1' and type_arcs_sortants(j) = 0) then
                somme_retrait := somme_retrait + poids_arcs_sortants(j);
            end if;
        end loop;
        -- actualisation marquage
        marquage_local := marquage_local + (somme_ajout - somme_retrait);
        -- evaluation des sensibilisations des transitions aval
        for k in 0 to nombre_arc_sortants-1 loop
            -- si arc inhibiteur (2)
            if (type_arcs_sortants(k) = 2) then
                if (marquage_local >= poids_arcs_sortants(k)) then
                    sensibilisations_arcs_sortants(k) <= '0';
                else
                    sensibilisations_arcs_sortants(k) <= '1';
                end if;
            -- si arc test (1) ou arc classique P->T (0)
            elsif (marquage_local >= poids_arcs_sortants(k)) then
                sensibilisations_arcs_sortants(k) <= '1';
            else
                sensibilisations_arcs_sortants(k) <= '0';
            end if;
        end loop;
        -- signaux d'activation des actions
        if (marquage_local >= 1) then
            action_activation <= '1';
        else
            action_activation <= '0';
        end if;
        marquage_interne <= marquage_local;
    end if;
end process;
end a_place;
```

FIGURE 2.7 – Code VHDL du composant place

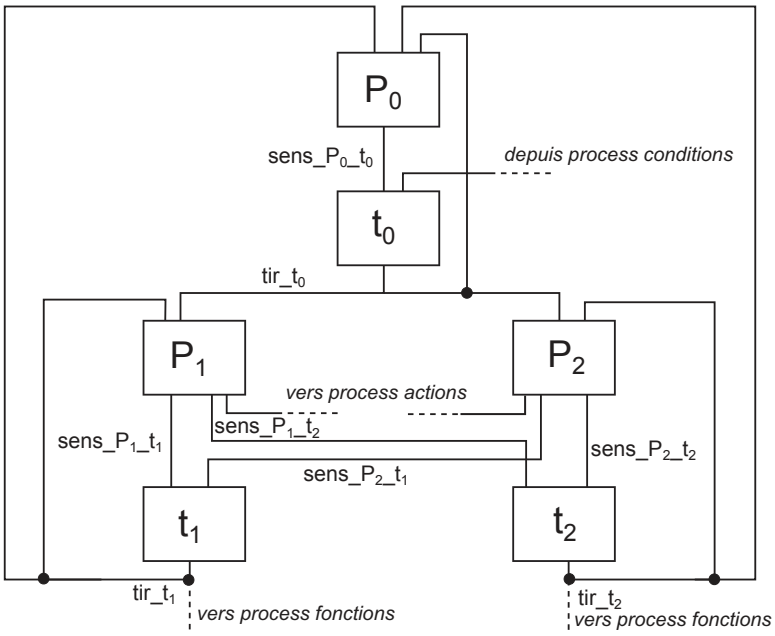

FIGURE 2.8 – Exemple d'interconnexion de composants VHDL

asynchrone (combinatoire) la valeur des différentes conditions. Le code VHDL des process gérant l'interprétation du RdP décrit figure 2.3 est donné figure 2.9.

Place pivot vs Transition pivot

Utiliser la place comme élément pivot plutôt que la transition est plus particulièrement intéressant dans le cadre des RdP généralisés. En effet, dans le cas d'une traduction « place pivot », la place gère le poids et le type des arcs. Il suffit donc qu'elle envoie un signal binaire à la transition aval pour que cette transition puisse décider si elle est sensibilisée ou non. Les échanges entre les composants ne se font alors qu'au travers de 2 signaux binaires : un de la place vers la transition pour la sensibilisation et un de la transition vers la place pour le tir. Au contraire, dans le cadre d'une traduction « transition pivot », la place est obligée de transmettre son marquage aux transitions pour que les transitions puissent décider si elles sont sensibilisées ou non par la place (puisque dans cette traduction les poids et les types des arcs sont stockés dans la transition). Ainsi la place doit envoyer un nombre naturel à la transition : la connexion de la place vers la transition se fait alors à l'aide de plusieurs signaux binaires. Au plus le marquage d'une place est élevé au plus le nombre de signaux binaires nécessaires pour coder l'information sera élevé. Choisir la place comme pivot de la traduction permet ainsi de simplifier la connexion entre le composant place et le composant transition.

Dimensionnement des composants

Le dimensionnement des ports d'entrées et de sorties des instances de composants place et transition est réalisé à l'aide de la structure du RdP. En effet, la taille des vecteurs d'entrée et de sortie des instances des composants est constante et dépend du nombre de nœuds aval et amont dans le RdP. Par contre, il est nécessaire de dimensionner le marquage maximum des instances de place. Il faut en effet décider combien de bits (bascules) seront alloués pour mémoriser le marquage des places. Supposons que le marquage d'une place soit codé sur 2 bits. Si le marquage de la place est tel qu'il devrait passer de 3 à 4, il passerait en fait sur la cible de 3 à 0. Le comportement obtenu sur le FPGA pourrait

```
-- process des actions
actions : process (reset_n, clk)
begin
    if (reset_n = '0') then
        a1 <= '0';
    elsif (clk'event and clk='0') then
        if ((s_action_P1 = '1') or (s_action_P2 = '1')) then
            a1 <='1';
            a1(s_a1); -- exécution de l'action a1 utilisant le signal s_a1
        else
            a1 <= '0';
        end if;
    end if;
end process;
-- process des fonctions
functions : process (reset_n, clk)
begin
    if (reset_n = '0') then
    elsif (clk'event and clk='1') then
        if (tir_t1 /= '0') then
            f0(s_f0); --exécution de la fonction f0 utilisant le signal s_f0
        end if;
        if (tir_t2 /= '0') then
            f1(s_f1); --exécution de la fonction f1 utilisant le signal s_f1
        end if;
    end if;
end process;
-- process des conditions
process(c0) begin
    if (cond_0(c0)) then -- évaluation de la valeur de la condition c0
        s_cond_0 <= '1';
    else
        s_cond_0 <= '0';
    end if;
end process;
```

FIGURE 2.9 – Code VHDL des process gérant l'interprétation

alors être erroné et peut même entraîner un blocage ou une absence complète d'activité dans le RdP. Ceci est bien sûr inacceptable. Par ailleurs, si allouer un très grand nombre de bits à la mémorisation du marquage n'entraîne pas de comportement erroné, cela augmente inutilement le nombre de cellules logiques nécessaires à la description de la place. Il est donc nécessaire de dimensionner avec précision le marquage maximum des instances des composants place. L'analyse formelle contribue alors au dimensionnement du modèle implémenté (cf. figure 2.14).

2.1.4 Construction du modèle analysable

Il s'agit d'être capable d'obtenir un modèle analysable à partir du modèle conçu basé sur un RdP GEIS (cf. figure 2.10). Nous nous plaçons ici dans le cadre de la synthèse globale et supposons le modèle global déjà obtenu, son obtention étant décrite au chapitre 1. Nous nous intéressons donc plus particulièrement à la transformation du modèle global en modèle analysable.

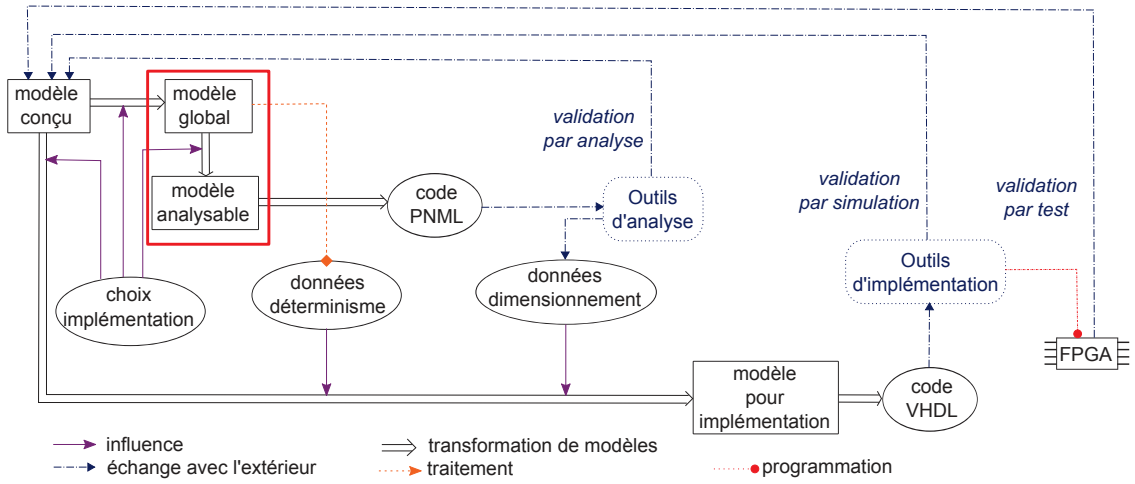

FIGURE 2.10 – Transformation du modèle global en modèle analysable dans la méthodologie HILECOP

Comme expliqué dans le chapitre 1, les outils d'analyse existant ne permettent pas d'analyser l'interprétation d'un modèle RdP. Il n'existe pas non plus, à notre connaissance, d'outils permettant l'analyse de RdP évoluant de manière synchrone. Par contre, l'évolution synchrone et l'impact de l'interprétation surtout des conditions sur l'évolution du RdP pourraient être pris en compte grâce à des intervalles temporels qui, eux, peuvent être analysés. Le modèle analysé sera alors un RdP généralisé étendu T-temporel (RdP GET).

Il n'est pas possible de décrire exactement le comportement d'un RdP GEIS à l'aide d'un RdP GET. Par contre, il est possible d'obtenir un RdP GET dont le comportement est le plus proche possible de celui du RdP GEIS initial. D'une part, pour que les résultats d'analyse obtenus sur le RdP GET soient pertinents pour le RdP GEIS correspondant, il faut que toutes les évolutions possibles du RdP GEIS initial soient possibles dans le RdP GET obtenu après transformation comme expliqué dans le chapitre 1 (cf. figure 2.11). D'autre part, pour que les résultats soient les plus intéressants possibles, il faut minimiser les évolutions possibles dans le RdP GET qui ne le sont pas dans le RdP GEIS, c'est-à-dire la partie grisée de la figure 2.11. Rappelons qu'il ne pourra pas y avoir équivalence stricte

entre les 2 modèles notamment à cause de l'interprétation. La pertinence de la transformation de modèles proposée , du point de vue des résultats d'analyse, sera discutée dans le §2.1.5.

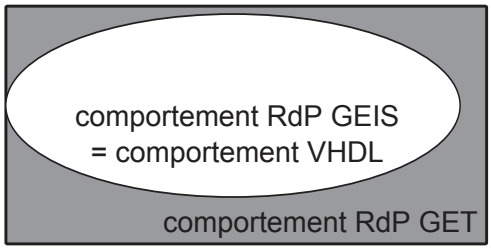

FIGURE 2.11 – Relation souhaitée entre le RdP GEIS et le RdP GET

Il faut d'abord définir la classe de RdP GET et sa sémantique pour pouvoir ensuite être capable d'établir un modèle analysable cohérent avec le modèle conçu décrit à l'aide de RdP GEIS.

Définition et sémantique formelles du modèle analysable

La définition formelle des RdP GET, inspirée de celle donnée pour les RdP temporels non étendus à priorités dans [13] et de celle proposée dans [53] pour les RdP temporels paramétrés avec arcs inhibiteurs, est donnée ci-dessous.

Soit $\mathbb{I}^+$ l'ensemble des intervalles réels non vides avec des bornes non négatives et entières. $\forall i \in \mathbb{I}^+$, $\downarrow i$ est la borne inférieure de l'intervalle et $\uparrow i$ sa borne supérieure qui peut être égale à l'infini. De plus, $\forall \theta \in \mathbb{R}^+$, $i - \theta$ correspond à l'intervalle $[\downarrow i - \theta, \uparrow i - \theta]$.

Definition 2.1.3. *Un réseau de Petri généralisé étendu T-temporel est un uplet* $< P, T, Pre, Pre_t, Pre_i, Post, m_0, Is >$ *où :*

- $< P, T, Pre, Pre_t, Pre_i, Post, m_0 >$ *est un réseau de Petri généralisé étendu avec* P *les places,* T *les transitions,* m_0 *le marquage initial et* $Pre, Pre_t, Pre_i, Post : T \rightarrow P \rightarrow \mathbb{N}^+$ *les fonctions préconditions, tests, inhibitions et postconditions.*
- $Is : T \rightarrow \mathbb{I}^+$ *est la fonction des intervalles statiques*

Le marquage est toujours donné par la fonction $m : P \rightarrow \mathbb{N}$.

Une transition $t \in T$ est toujours sensibilisée par m, noté $t \in sens(m)$ si et seulement si $(m \geq Pre(t) + pre_t(t)) \wedge (m < Pre_i(t))$.

Une transition $k \in T$ est dite nouvellement sensibilisée par le tir de la transition t depuis le marquage m, noté $k \in \uparrow sens(m, t)$, si et seulement si la transition k, différente de t, est sensibilisée par le nouveau marquage $m - Pre(t) + Post(t)$ mais ne l'était pas par $m - Pre(t)$ ou si $k = t$ et t est toujours sensibilisée par le nouveau marquage. Le marquage $m - Pre(t)$ est considéré puisque les jetons dans Pre(t) sont consommés par le tir de t. La définition formelle suivante est alors obtenue :

$$k \in \uparrow sens(m,t) \Leftrightarrow \big(m - Pre(t) + Post(t)\big) \geq Pre(k) + Pre_t(k)$$
$$\wedge \big(m - Pre(t) + Post(t) < Pre_i(k)\big) \wedge \big[\big(Pre_i(k) \leq m - Pre(t)\big)$$
$$\vee (k = t) \vee \big(Pre(k) + Pre_t(k) > m - Pre(t)\big)\big]$$

L'état du RdP est défini par $s = (m, I)$ où I est la fonction intervalle temporel. La fonction $I : T \to I^+$ associe un intervalle de temps à chaque transition sensibilisée par m.

Definition 2.1.4. *La sémantique d'un RdP généralisé étendu T-temporel* $< P, T, Pre, Pre_t, Pre_i, Post, m_0, Is >$ *est le système de transition* $(S, s_0, \rightsquigarrow)$ *où :*

- S *est l'ensemble des états* (m, I) *du RdP.*
- $s_0 = (m_0, I_0)$ *est l'état initial du RdP.*
- $\rightsquigarrow \subseteq S \times (T \cup \mathbb{R}^+) \times S$ *est la relation de transition d'états définie comme suit :*
  - $\forall t \in T$, *on a* $(m, I) \overset{t}{\rightsquigarrow} (m', I')$ *si et seulement si :*
    1. $t \in sens(m)$ *(sensibilisation de t par m)*
    2. $m' = m - Pre(t) + Post(t)$ *(actualisation du marquage)*
    3. $\forall k \in T$, *si* $k \in\uparrow sens(m), I'(k) = I_s(k)$, *sinon* $I'(k) = I(k)$. *(initialisation des intervalles de temps si nécessaire)*
  - $\forall \theta \in \mathbb{R}^+, (m, I) \overset{\theta}{\rightsquigarrow} (m, I')$ *si et seulement si* $\theta \in \mathbb{R}^+$ *et :*
    1. $(\forall t \in T)\ t \in sens(m) \Rightarrow \theta \leq\uparrow I(t)$ *(le temps peut évoluer tant que la borne maximum d'un intervalle de temps n'est pas dépassée)*
    2. $(\forall t \in T)\ t \in sens(m) \Rightarrow I'(t) = I(t) \dot{-} \theta$ *(actualisation des intervalles de temps)*

Méthode de transformation

Le but est donc de transformer un RdP GEIS défini par $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, clock >$ en un RdP GET défini par $< P', T', Pre', Pre'_t, Pre'_i, Post', m'_0, Is' >$.

La structure du RdP GEIS est conservée, nous posons donc :

- $P' = P$
- $T' = T$
- $(Pre' = Pre) \wedge (Pre'_t = Pre_t) \wedge (Pre'_i = Pre_i) \wedge (Post' = Post)$
- $m'_0 = m_0$

Il reste donc à définir Is'. C'est justement dans l'utilisation des intervalles temporels que la mise en œuvre synchrone et l'interprétation seront prises en compte. Nous posons qu'une unité de temps correspond à un cycle complet d'horloge. Comme l'implémentation est synchrone, une transition ne peut pas être tirée en moins d'une unité de temps (cf. §2.1.3). Nous avons donc nécessairement $\forall t \in T', \downarrow Is'(t) = 1$. Par contre l'interprétation peut empêcher le tir d'une transition sensibilisée (condition fausse), ainsi la borne supérieure de l'intervalle statique est définie comme suit :

- $\forall t \in T, \Big(\forall c \in (C), C(t)(c) = 0\Big) \Rightarrow\uparrow Is'(t) = 1$
- $\forall t \in T, \Big(\exists c \in (C) \backslash C(t)(c) \neq 0\Big) \Rightarrow\uparrow Is'(t) = +\infty$

En effet, si la transition n'a pas de condition associée, elle est nécessairement tirée en 1 unité de temps sur le modèle implémenté. Au contraire si au moins une condition est associée à la transition, elle peut être tirée n'importe quand voire jamais puisqu'il n'y a pas moyen de savoir quand la condition deviendra vraie. Les évolutions possibles du RdP GEIS seront donc bien aussi possibles dans le RdP GET.

Les fonctions et les actions n'influent pas sur la construction du RdP GET car elles n'ont aucun effet direct sur l'évolution du RdP. Elles ont par contre un effet indirect en modifiant les valeurs des conditions. Or comme ce n'est pas aux valeurs des conditions que nous nous intéressons (car non prévisible) mais uniquement à leur présence, il n'est donc pas nécessaire de gérer la présence de fonctions et d'actions dans la transformation du modèle.

Nous avons ainsi pu construire un RdP GET analysable à partir d'un RdP GEIS conçu. Il faut maintenant s'assurer que les résultats d'analyse obtenus sur ce RdP GET sont bien pertinents pour le RdP GEIS.

Amélioration possible

Pour l'instant, dès qu'une condition est associée à une transition, l'intervalle temporel associé à cette transition dans le modèle analysable est $[1, +\infty[$ car il n'est pas possible de savoir quand la condition deviendra vraie dans le système réel.

Mais dans le formalisme des RdP GEIS, il est possible d'associer à une transition, soit une condition, soit sa négation. Ainsi, dans le cas du RdP donné figure 2.12, la condition c est associée à t_0 et sa négation à t_1. Dès que t_0 et t_1 sont sensibilisées, une des deux transitions sera nécessairement tirable sur le système réel puisque, soit l'une, soit l'autre, des conditions sera vraie. Ainsi, dans le système réel, la transition tirable sera toujours tirée en 1 période d'horloge. Dans ce cas bien précis, il est alors possible d'associer l'intervalle $[1, 1]$ aux transitions t_0 et t_1. Cette simplification ne peut être réalisée que si les conditions de sensibilisation des transitions concernées sont identiques.

Ceci permet de limiter les évolutions possibles dans le modèle analysable mais impossibles sur le système réel et réduit la complexité de l'analyse. Si le cas peut sembler très particulier, dans la pratique, il est couramment utilisé par les concepteurs.

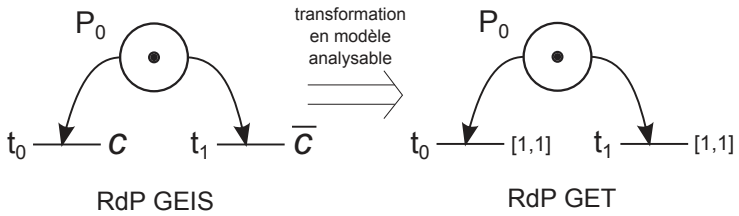

FIGURE 2.12 – Amélioration possible de la transformation en modèle analysable

A noter qu'il ne s'agit pas ici d'analyser les expressions en VHDL des conditions pour établir qu'elles sont mathématiquement mutuellement exclusives ou physiquement mutuellement exclusives (i.e. exclusion intrinsèque au système, comme par exemple un objet ne peut pas être simultanément dans deux endroits différents), mais bien de profiter qu'il soit possible d'associer dans la méthodologie HILECOP la négation d'une condition. Par contre, il serait possible d'étudier les combinaisons de conditions qui peuvent permettre de garantir que parmi plusieurs transitions ayant les mêmes conditions de sensibilisation, une sera nécessairement franchie si elles deviennent sensibilisées (cf. figure 2.13).

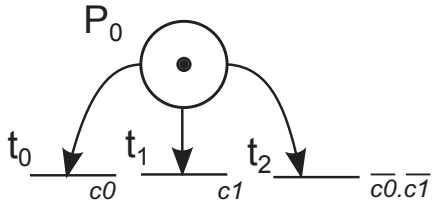

FIGURE 2.13 – Cas de 3 transitions dont les conditions garantissent qu'une sera toujours tirable si elles sont sensibilisées

2.1.5 Pertinence des résultats d'analyse

L'élaboration d'un modèle analysable a pour objectif de pouvoir réaliser des validations formelles sur le comportement du système réel. Or l'analyse formelle sera naturellement réalisée sur le modèle analysable. Nous avons vu chapitre 1 qu'il y avait 2 écarts à prendre en compte entre le comportement du système réel et le comportement du modèle analysable : celui entre le système réel et le modèle conçu et celui entre le modèle conçu et le modèle analysable. Nous posons l'hypothèse, réaliste de par les précautions prises dans la méthodologie HILECOP, que l'écart entre le comportement du système réel et celui du modèle conçu est nul. Nous nous intéressons alors à l'écart entre le modèle conçu et le modèle analysable, pour s'assurer que les résultats d'analyse obtenus seront bien pertinents.

Les résultats d'analyse que nous souhaitons obtenir, du point de vue validation du comportement, concernent deux points :

- les évolutions possibles du système réel ne seront jamais dangereuses et autant que possible seront toujours correctes. Ceci peut notamment être vérifié sur un modèle RdP en utilisant la technique du model-checking qui étudie le graphe des classes d'états. Le graphe des classes d'états représente toutes les évolutions possibles du RdP. Deux types de propriétés sont distinguées : les propriétés d'invariance du type *"Quelle que soit l'évolution du système, on a toujours"* et les propriétés d'accessibilité du type *" Il existe une évolution telle que"* [11].
- le système réel ne sera jamais bloqué dans un état, i.e. il pourra toujours évoluer. Il est dit dans ce cas que le système est vivant. La vivacité d'un modèle RdP peut être étudiée soit en utilisant le graphe des classes d'états soit par analyse structurelle du réseau de Petri [44].

Pour assurer que le modèle sera implémentable, le caractère borné du RdP doit également pouvoir être étudié.

Tout d'abord, nous définissons : une évolution est possible dans un modèle A et dans un modèle B si la succession d'états stables est la même dans les 2 modèles et que les transitions tirées pour passer d'un état stable à un autre sont les mêmes. Il est dit qu'un état est stable quand le modèle ne peut plus évoluer sans que le temps ne s'écoule.

Il est possible de prouver formellement que le comportement du RdP GEIS est inclus dans celui du RdP GET, c'est-à-dire que toutes les évolutions possibles dans le RdP GEIS sont également possibles dans le RdP GET (cf. annexe B). Toutes les évolutions possibles du système réel sont donc bien analysées lors de l'analyse du modèle analysable. Ainsi si

une propriété d'invariance est vérifiée sur le modèle analysable, elle sera nécessairement aussi vraie sur le système réel. De même, si une propriété d'accessibilité est fausse dans le modèle analysable, elle le sera aussi sur le système réel.

De plus, cette inclusion garantit que si le marquage d'une place est borné dans le modèle analysable, il le sera aussi dans le système réel.

Par contre, comme le comportement du modèle analysable est un sur-ensemble du comportement du modèle implémenté (ou conçu) :

- Si une propriété d'invariance est fausse sur le modèle analysable, elle peut être vraie sur le système réel. Il est en effet possible que des évolutions problématiques soient détectées sur le modèle analysable alors qu'elles ne sont pas possibles dans le modèle implémenté (ou conçu). Il y a donc un risque de "fausse alarme". En cas d'erreur détectée par l'analyse, le concepteur devra alors soit s'assurer "manuellement" que le problème détecté peut effectivement se produire sur le système réel, soit systématiquement corriger les erreurs, quitte à corriger un modèle qui était déjà juste.
- Si une propriété d'accessibilité est vraie sur le modèle analysable, elle peut ne pas l'être sur le système réel. Elles peuvent en effet être vérifiées sur des chemins qui ne sont pas possibles sur le système réel. Si le concepteur a néanmoins besoin de vérifier ce type de propriétés, il devra s'assurer manuellement que le chemin trouvé sera bien toujours réalisable sur le système réel.
- Si le modèle analysable n'est pas borné, le système réel peut l'être.

De même, l'inclusion ne garantit pas que si le modèle analysable est vivant alors le système réel le sera (cf §2.2.6). Notamment, si une seule transition est sensibilisée est qu'une condition qui n'est jamais vraie lui est associée, alors il y aura un blocage dans le système réel qui ne sera pas visualisée sur le modèle analysable. En effet, dans le modèle analysable, il est supposé que la condition deviendra nécessairement vraie un jour quitte à ce que ce soit au bout d'un temps infini. Si cette même hypothèse est faite sur le système réel, dans le cas des RdP GEIS sans conflit effectif, si le modèle analysable est vivant alors le système réel l'est aussi.

2.1.6 Utilisation de l'analyse pour le dimensionnement du système réel

Nous avons vu, notamment dans le §2.1.3, que lors de la traduction automatique du modèle conçu en code VHDL, il est nécessaire de connaître un marquage maximum des places pour assurer que l'évolution du modèle implémentable corresponde à celle attendue. Ces données de dimensionnement sont fournies, dans la méthodologie HILECOP, par l'analyse du modèle analysable (cf. figure 2.14).

Pour éviter de surdimensionner inutilement les places, l'idée est de dimensionner chaque instance de composant place avec le marquage maximum de la place. Un marquage maximum pour chaque place du RdP GEIS peut notamment être obtenu grâce à l'analyse comportementale du RdP GET associé par model-checking. Pour ce faire, le graphe des classes d'états, obtenu à l'aide d'un outil d'analyse comme TINA, est automatiquement parcouru et le marquage maximum de chaque place enregistré.

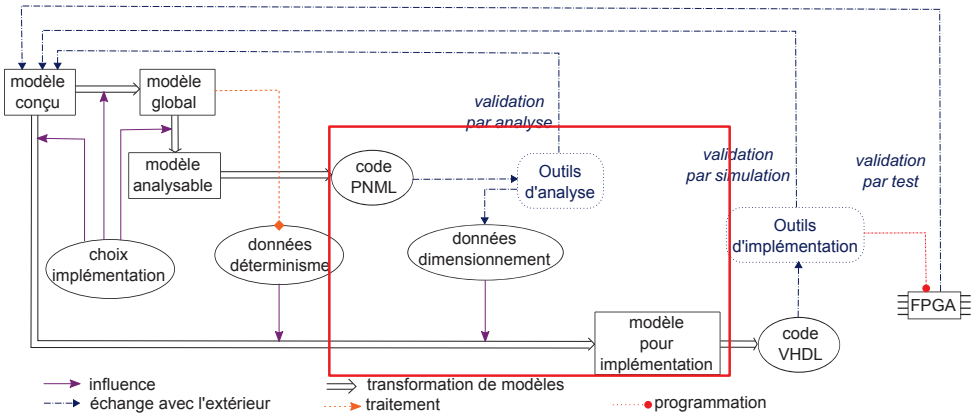

FIGURE 2.14 – Influence de l'analyse sur la génération du code VHDL dans la méthodologie HILECOP

Avec cette méthode, nous pouvons être sûr que le marquage maximum obtenu grâce à l'analyse ne sera jamais dépassé lors de l'exécution du code sur la cible puisque nous avons démontré l'inclusion du comportement du modèle conçu (RdP GEIS) dans celui du modèle analysé (RdP GET) et que la sémantique du modèle conçu intègre les propriétés non fonctionnelles du modèle implémenté. Nous assurons ainsi que le marquage des places ne sera jamais remis à zéro de manière inopportune et donc un comportement plus fiable du système réel.

Néanmoins, il n'est pas possible d'assurer que le marquage maximum obtenu sur le RdP GET est forcément accessible sur le RdP GEIS. Deux raisons peuvent expliquer la différence : la présence de conditions et le tir de transitions en concurrence. La non-équivalence des comportements du modèle conçu et du modèle analysé à cause des conditions a déjà été évoquée dans §2.1.5.

Expliquons pourquoi l'occurrence de tirs de transitions en concurrence peut amener à une sur-estimation du marquage maximum. Par exemple, dans le cadre du RdP donné figure 2.15 (a), les transitions t_0 et t_1 seront toujours simultanément tirées dans le modèle implémenté (synchrone) mais elles ne peuvent pas être tirées simultanément dans le modèle analysé (asynchrone). Ainsi les deux cas possibles (tir de t_0 puis de t_1 ou l'inverse) seront considérés amenant à un marquage maximum égal à 2 (cf. figure 2.15 (b))alors que le marquage maximum du modèle implémenté sera de 1 (cf. figure 2.15 (c)). Le problème vient du fait que l'analyse est réalisée sur tous les états du modèle et non pas uniquement sur les états stables du modèle. L'implémentation ne sera donc pas optimale mais elle reste fiable.

Il est possible que le graphe des classes d'états ne puissent pas être obtenu, soit parce que le modèle analysé n'est pas borné (même si le modèle implémenté l'est), soit parce que le graphe des classes d'états est trop grand pour être calculé. En effet, le calcul du graphe des classes d'états est sensible au problème d'explosion combinatoire. Dans ce cas, il est possible d'utiliser l'analyse structurelle pour obtenir des informations sur le marquage maximum des places [24].

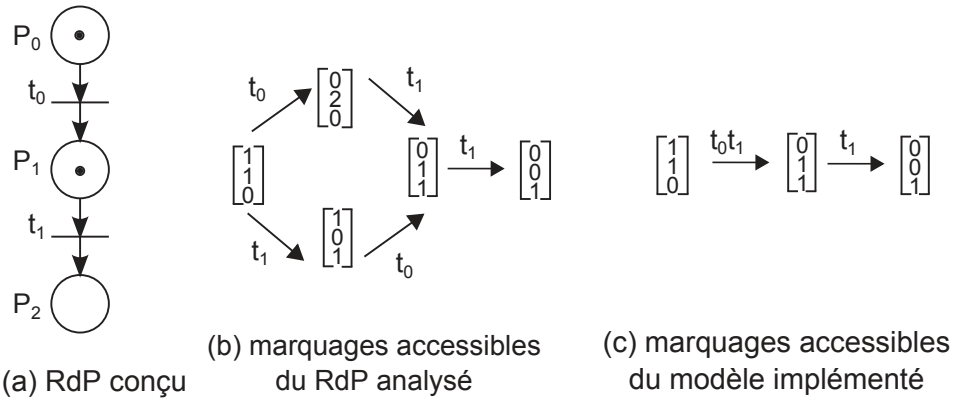

FIGURE 2.15 – Surestimation du marquage maximal en cas de concurrence

L'analyse structurelle permet d'obtenir des invariants de places sous la forme $\sum_{p \in inv} a_p m(p) = k_{inv}$ où inv est un invariant de place, a_p et k sont des entiers naturels. Ainsi, il est possible de garantir pour chaque place $p \in inv$ que $m(p) \leq \frac{k}{a_p}$. Par contre le marquage maximum ne pourra être déterminé avec fiabilité que pour les places appartenant à un invariant. Pour les autres places, l'analyse structurelle n'apportera aucune information.

En conclusion, nous sommes désormais en mesure d'assurer la correspondance modèle/implémentation/analyse pour un modèle conçu à l'aide de RdP GEIS sans conflit et notamment de gérer l'interprétation du modèle et son exécution synchrone. Nous allons donc maintenant aborder la problématique des conflits.

2.2 Gestion des conflits

Nous avons jusqu'à présent supposé les RdP GEIS comme sans conflit effectif. Or faire cette hypothèse peut rendre la modélisation plus complexe pour les concepteurs qui doivent, de plus, s'assurer une fois le modèle conçu que le problème des conflits est géré (par exemple en assurant l'exclusion mutuelle des transitions en conflit). Comme la vérification est manuelle, cela augmente aussi le risque d'erreurs humaines. Nous souhaitons que la méthodologie HILECOP puisse assurer de manière automatique que le comportement en cas de conflit effectif sera correct (cf. figure 2.16).

2.2.1 Nécessité et problématique

Commençons par rappeler la définition communément admise d'un conflit effectif [20] :
Dans le cadre d'un RdP asynchrone, pour un marquage m à un instant donné θ, la transition t_i est dite en conflit effectif avec t_j si et seulement si :
- t_i et t_j sont tirables à θ
- le marquage m', obtenu après tir de t_j, ne sensibilise plus t_i

Par exemple, figure 2.17, les transitions t_1 et t_2 sont en conflit effectif l'une avec l'autre tandis que ce n'est pas le cas des transitions t_3 et t_4 car le marquage de P_2 est suffisant pour pouvoir tirer les deux transitions. La relation "être en conflit effectif avec une transition" n'est pas symétrique. En effet, t_6 est en conflit effectif avec t_5 alors que t_5 ne l'est pas

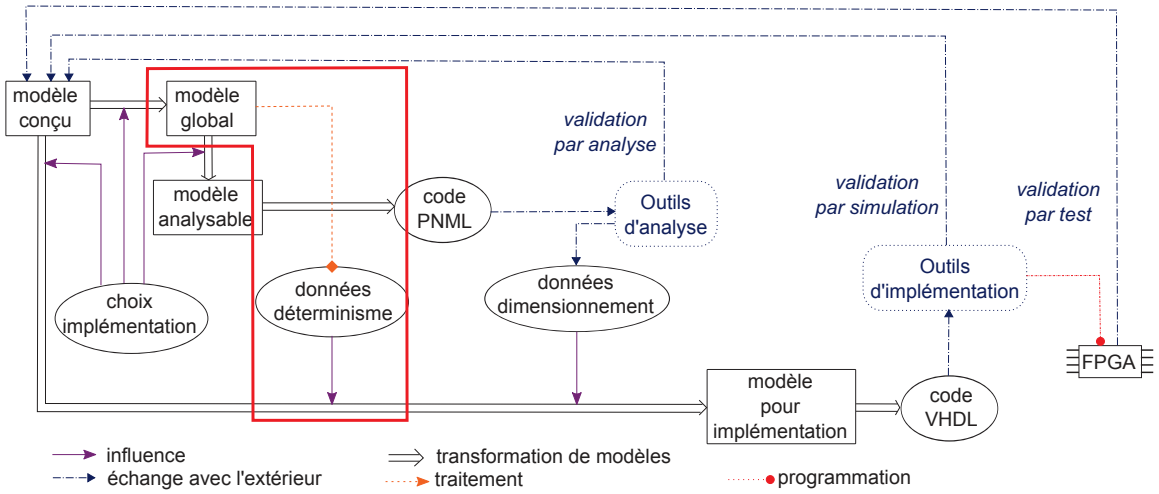

FIGURE 2.16 – Principe de gestion du déterminisme dans la méthodologie HILECOP

avec t_6. De même, t_7 est en conflit effectif avec t_8 alors que l'inverse n'est pas vrai. La non-symétrie du conflit effectif n'est pas due qu'à la présence d'arc test ou inhibiteur puisque t_9 est en conflit effectif avec t_{10} alors que t_{10} ne l'est pas avec t_9.

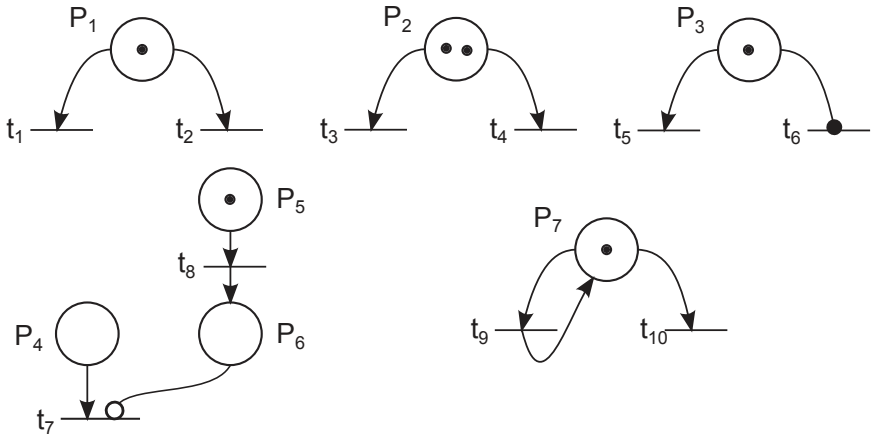

FIGURE 2.17 – Exemples de RdP contenant des conflits

Dans le cadre des systèmes critiques, le premier problème d'un RdP contenant au moins un conflit effectif est que l'évolution de ce RdP n'est plus déterministe puisqu'il n'y a pas de solution pour déterminer la (ou les) transition(s) à tirer quand deux transitions sont tirables. Plusieurs mécanismes ont déjà été proposés et étudiés dans le chapitre 1 (cf. §1.3) pour résoudre ce problème de non-déterminisme [54] [31] [21] mais aucun n'était pleinement satisfaisant.

Dans notre cas, il est d'autant plus nécessaire de gérer les conflits que le problème ne se réduit pas au problème du déterminisme de l'évolution du RdP. En effet, le RdP sera implémenté de manière synchrone. Toutes les transitions simultanément tirables sont donc par défaut simultanément tirées. Ainsi si les conflits effectifs ne sont pas gérés, les transitions en conflit effectif sont toutes tirées menant à un comportement non cohérent avec l'esprit des RdP puisqu'un même jeton est utilisé simultanément par plusieurs transitions. Le comportement souhaité par le concepteur sera alors différent de celui implémenté (un OU devient un ET) ce qui n'est bien sûr pas envisageable.

De plus l'implémentation synchrone nécessite de modifier sensiblement la définition du conflit effectif. En effet, si nous considèrons les exemples de la figure 2.17, d'après la définition habituelle t_9 n'est pas en conflit effectif avec t_{10} puisque le tir de t_9 n'empêche pas celui de t_{10}. Mais le marquage ne permet pas de tirer t_9 et t_{10} simultanément donc, lors de l'exécution, si t_9 est tiré, il faudra soit attendre une unité de temps avant de pouvoir tirer t_{10} soit utiliser plusieurs fois un même jeton ce qui n'est pas souhaitable. Donc dans notre cas il est choisi de considérer que t_9 est en conflit effectif avec t_{10}. Par contre, t_6 n'est toujours pas en conflit avec t_5 puisque dans ce cas le jeton n'est pas consommé. Une nouvelle définition du conflit effectif est alors utilisé pour les RdP GEISP (RdP GEIS à priorités) :

Definition 2.2.1. *Dans le cadre d'un RdP GEISP,* $\forall(t_i, t_j) \in T^2$, t_i *est en conflit effectif avec* t_j *pour un état* $s = (m, cond, ex)$, *noté* $t_i \in eff(t_j, s)$, *si et seulement si, :*

- $t_i \in tirable(s)$
- $t_j \in tirable(s)$
- $t_i \notin tirable\big(m - Pre(t_j), cond, ex\big) \vee t_i \notin tirable\big(m - Pre(t_j) + Post(t_j), cond, ex\big)$

Il nous faut une solution permettant de garantir qu'une fois implémenté le comportement du RdP sera tel que tous les conflits effectifs soient correctement gérés. Néanmoins il est souhaitable que l'évolution du RdP implémenté soit la plus "permissive" possible, c'est-à-dire qu'un tir de transition correct d'un point de vue marquage ne doit pas être empêché à cause de la gestion des conflits. Cela permet de préserver la réactivité du système. Deux sortes d'approches peuvent être utilisées pour la gestion des conflits : s'assurer qu'avoir deux transitions en conflit effectif n'est pas possible ou alors définir quelle transition est tirée quand cette situation arrive. Les méthodes rencontrées jusqu'ici utilisent la première approche.

Une solution déjà évoquée est d'empêcher structurellement qu'il puisse y avoir conflit effectif entre deux transitions. Pour cela, il est par exemple possible d'utiliser un anneau mémoire comme dans la méthode d'Uzam [54] 1.19. L'anneau mémoire tel qu'il est décrit assure de tirer les deux transitions en conflit en alternance mais cela peut modifier la spécification initiale dans le sens où une alternance potentiellement non désirée est imposée. Il est possible d'imaginer d'autres manières de gérer cette alternance à l'aide d'un anneau mémoire comme dans la figure 2.18. Le franchissement de t_{c1} et t_{c2} peut alors être piloté par une condition ou par le marquage d'autres places du réseau. D'autres solutions peuvent bien sûr être imaginées, comme utiliser le marquage de places déjà présentes dans la description du comportement pour déterminer quelles transitions choisir. Mais ce type de solution ne peut pas être automatisée puisqu'elle dépend du modèle.

Une autre solution est de faire en sorte que si les deux transitions peuvent effectivement être sensibilisées en même temps, elles ne puissent pas être simultanément tirables à l'aide des conditions associées aux transitions. Il faut alors que les transitions soient mutuellement exclusives, c'est-à-dire qu'une seule puisse être vraie à un instant donné. Là encore, deux approches sont possibles, l'approche automatisable et celle non-automatisable. L'approche automatisable peut consister à ajouter automatiquement une nouvelle variable à l'une des transitions et son contraire à l'autre transition comme proposé dans la méthode d'Uzam 1.20. Une autre solution est d'automatiquement associer la négation de la condition de la première transition à la condition de la deuxième (ou l'inverse). Une solution

non automatisable est d'exiger que le concepteur assure de lui-même qu'en cas de risque de conflit effectif les conditions des transitions concernées soient toujours mutuellement exclusives.

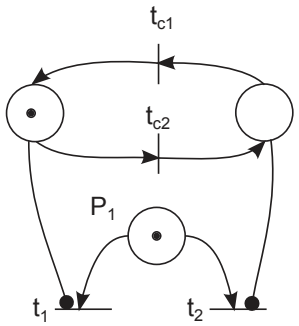

FIGURE 2.18 – Exemple de gestion d'un conflit à l'aide d'un anneau mémoire permettant de "piloter" l'alternance

L'inconvénient des techniques non automatisées est qu'elles introduisent nécessairement un risque d'erreur humaine et sont à la charge du concepteur allongeant ainsi le temps de conception par rapport à une solution automatisée. Nous souhaitons donc utiliser une méthode la plus automatisée possible.

L'inconvénient des solutions automatisées basées sur le principe des anneaux mémoire est que, comme décrit dans le chapitre 1, la transition tirable est préchoisie. Si la transition choisie n'est pas tirable, l'autre transition ne peut pas être tirée même si elle serait tirable sans cette solution de gestion de conflit. De même, en rendant mutuellement exclusives les conditions des transitions qui peuvent être en conflit effectif, il n'est pas garanti que des tirs de transition, qui ne devraient pas être bloqués, ne le seront pas. En effet, avoir des conditions mutuellement exclusives garantit qu'il ne peut y avoir qu'une seule condition vraie, mais pas qu'il y en a nécessairement une de vraie. Des évolutions dans le RdP peuvent ainsi être empéchées alors qu'elles ne l'étaient pas dans le comportement non déterministe et qu'il n'y a pas conflit effectif.

De plus, aucune de ces solutions ne permet de tirer plus d'une transition même quand les transitions ne sont pas en conflit effectif car le marquage des places amont est tel que le conflit n'est pas effectif. Nous préférerions avoir une solution qui ne fasse un choix qu'une fois le conflit effectif avéré, i.e quand il est vraiment nécessaire de faire un choix.

Puisqu'il s'agit de choisir de manière déterministe les transitions à tirer en cas de conflit effectif, une solution, intuitive et déjà utilisée dans la littérature [14] [1], est de définir des priorités entre les transitions, il s'agit alors de RdP à priorités. Ainsi, lors de l'évolution du RdP, si deux transitions sont en conflit effectif, c'est la transition la plus prioritaire qui est tirée. Définir une priorité entre les transitions potentiellement en conflit effectif nous permet donc de savoir de manière déterministe comment le modèle doit évoluer. Néanmoins il ne faut pas oublier que notre définition du conflit est légèrement différente de celle utilisée pour les RdP asynchrones et donc le comportement obtenu d'un RdP à priorités synchrone ne sera pas le même que celui d'un RdP à priorités asynchrone car dans les RdP aysnchrones une seule transition peut être tirée à la fois.

L'utilisation de priorités nous paraît la solution plus adaptée pour nos contraintes notamment car cette solution ne limite le tir des transitions que si elles sont réellement en conflit effectif. De plus, cette solution est très intuitive et donc pratique à utiliser pour le concepteur. La représentation graphique des priorités est un arc partant de la transition la plus prioritaire vers la transition la moins prioritaire. L'arc est représenté en pointillé. Ainsi figure 2.19, la transition t_1 est prioritaire sur la transition t_0.

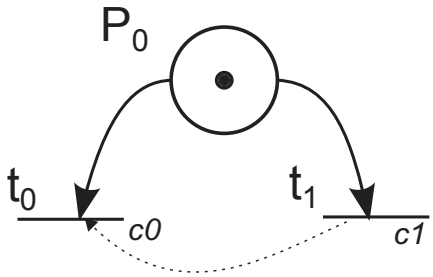

FIGURE 2.19 – Exemple de représentation d'une priorité entre deux transitions en conflit

D'un point de vue formel, la relation priorité est définie comme la relation priorité définie dans [14] et est ajoutée au n-uplet décrivant le RdP GEIS pour définir les RdP GEISP.

Definition 2.2.2. *$\succ$ est la relation priorité supposée irréflexive, asymétrique et transitive.*

La sémantique des RdP GEISP présentée en §2.1.2 doit alors être adaptée pour pouvoir gérer les conflits :

Definition 2.2.3. *La sémantique d'un RdP généralisé étendu interprété synchrone $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, clock, \succ >$ est le système de transition temporisé $< S, s_0, \rightsquigarrow >$ où :*

- *S est l'ensemble des états $(m, cond, ex)$ du RdP.*
- *$s_0 = (m_0, 0, 0)$ est l'état initial où 0 est la fonction nulle, c'est-à-dire la fonction qui associe 0 à tous les éléments d'un ensemble.*
- *$\rightsquigarrow \subseteq S \times Clk \times S$ est la relation de transition d'états, notée $s \overset{clk}{\rightsquigarrow} s'$ avec $clk \in Clk$, définie comme suit : Soit $Tir(s) \subseteq T$ l'ensemble des transitions tirées depuis l'état s. Cet ensemble ne peut être modifié que sur un front descendant de l'horloge. A l'état initial, on a $Tir(s_0) = \varnothing$.*
  - *On a $s = (m, cond, ex) \overset{\downarrow clock}{\rightsquigarrow} s' = (m, cond', ex')$ si et seulement si $\downarrow clock = 1$ et :*
    1. *$\forall c \in \mathcal{C}, cond'(c) = val(c)$*
    2. *$\forall a \in \mathcal{A}, \exists p \in P | A(p)(a) = 1 \wedge m(p) \neq 0 \Rightarrow ex'(a) = 1$, sinon $ex'(a) = 0$*

    Il est alors possible de déterminer $Tir(s')$ l'ensemble des transitions qui seront tirées.

    3. *$\forall t \in tirable(s'), \forall t' \in T | t' \succ t, t' \notin tirable(s') \Rightarrow t \in Tir(s')$* ***(si aucune transition plus prioritaire n'est tirable, une transition tirable est tirée)***

       $\forall t \in tirable(s)$, ***soit*** *$Pr(t)$* ***l'ensemble des transitions*** *t_i* ***telles que*** *$t_i \succ t \wedge t_i \in Tir(s')$.*

4. $\forall t \in tirable(s'), \Big(t \in sens\big(m - \sum_{t_i \in Pr(t)} Pre(t_i)\big) \wedge t \in sens\big(m - \sum_{t_i \in Pr(t)} Pre(t_i) + \sum_{t_i \in Pr(t)} Post(t_i)\big)\Big) \Rightarrow t \in Tir(s')$ ***(une transition tirable est tirée si le marquage est suffisant pour tirer toutes les transitions tirables plus prioritaires et cette transition)***
5. $\forall t \in tirable(s'), \Big(t \notin sens\big(m - \sum_{t_i \in Pr(t)} Pre(t_i)\big) \vee t \notin sens\big(m - \sum_{t_i \in Pr(t)} Pre(t_i) + \sum_{t_i \in Pr(t)} Post(t_i)\big)\Big) \Rightarrow t \notin Tir(s')$ ***(une transition tirable n'est pas tirée si le marquage n'est pas suffisant pour tirer toutes les transitions tirables plus prioritaires et cette transition)***
6. $\forall t \notin tirable(s'), t \notin Tir(s')$

– *On a* $s = (m, cond, ex) \overset{\uparrow clock}{\rightsquigarrow} s' = (m', cond, ex')$ *si et seulement si* $\uparrow clock = 1$ *et :*

1. $m' = m - \sum_{t \in Tir(s)} Pre(t) + \sum_{t \in Tir(s)} Post(t)$
2. $\forall f \in \mathcal{F}, \exists t \in Tir(s) | F(t)(f) = 1 \Rightarrow ex'(f) = 1$, *sinon* $ex'(f) = 0$

Il reste à déterminer comment gérer ces priorités du point de vue de l'implémentation puisqu'il ne s'agit plus d'une solution structurelle utilisant des places et/ou des transitions ou d'une solution utilisant les conditions, éléments que nous savons déjà implémentés.

Contrainte supplémentaire, la solution de gestion des conflits choisie doit s'inscrire dans l'implémentation synchrone définie précédemment (§2.1.1) puisque nous ne souhaitons pas la modifier. La gestion du conflit doit donc être réalisée entre le front descendant et le front montant de l'horloge gérant l'évolution du RdP (phase ② de la figure 2.2). En effet, pour savoir si un conflit est réellement effectif, il faut savoir si les transitions sont tirables et cette information n'est connue qu'à partir du front descendant de l'horloge puisque c'est à ce moment que les conditions sont évaluées. De plus, pour pouvoir réaliser l'actualisation du marquage, il faut connaître les transitions effectivement tirées or cette actualisation est réalisée sur le front montant de l'horloge.

En outre, les contraintes d'efficacité en termes de surface et de consommation propres à notre contexte sont bien sûr toujours d'actualité.

Il s'agit donc d'avoir une solution automatisable de gestion des conflits basée sur la définition de priorités entre les transitions en conflit. Pour pouvoir assurer que tous les conflits sont gérés correctement sans risque d'erreur humaine, il faut tout d'abord une solution automatisable pour détecter tous les conflits potentiels du modèle.

2.2.2 Détection automatique des conflits

Nous souhaitons pouvoir détecter de manière automatique tous les conflits effectifs ou plus exactement tous les groupes de transitions en conflit effectif. Tout d'abord nous avons vu que la relation "être en conflit effectif avec une transition" n'est pas symétrique, il faut donc définir le terme "groupe de transitions en conflit effectif". Derrière cette définition, il s'agit en fait de déterminer quel comportement le modèle doit avoir en cas de conflit effectif non symétrique.

Nous avons indiqué dans le §2.2.1 que nous souhaitons avoir le plus de tirs de transitions possibles autorisés et nous ne gérons donc que les cas où la relation de conflit est symétrique. Dans cet esprit, un groupe de transitions en conflit est défini tel que n'importe quelle transition t_i du groupe est en conflit effectif avec toutes les transitions t_j du groupe. Le fait de ne gérer que le cas où la relation de conflit est symétrique n'est pas problématique car, dans le cas contraire, il est toujours possible de tirer les deux transitions en respectant l'esprit de la spécification des RdP. En effet, l'une des transitions n'étant pas en conflit avec l'autre, nous pouvons conceptuellement considérer que nous tirons toujours cette transition en premier. Il s'agit bien d'une considération conceptuelle puisque les transitions sont tirées de manière synchrone, i.e. simultanément.

De plus, ne gérer que les cas vraiment problématiques permet de minimiser le nombre de cas à considérer ce qui est bien du point de vue des contraintes de notre contexte. En effet, la gestion de chacun des conflits détectés aura nécessairement un coût en surface et en consommation du point de vue FPGA, il faut donc mieux en gérer le moins possible. Nous obtenons ainsi la définition des groupes des transitions en conflit effectif qui peuvent d'ailleurs contenir plus de deux transitions.

Definition 2.2.4. *$\forall T_c \subseteq T$, T_c est un groupe de transitions en conflit effectif pour un état $s = (m, cond, ex)$ si et seulement si $\forall t_i \in T_c, \forall t_j \in T_c \backslash t_i, t_i \in eff(t_j, s) \Leftrightarrow$*

- $\forall t_i \in T_c, t_i \in tirable(s)$
- $\forall t_i \in T_c, t_i \notin tirable\big(m - \sum_{t_j \in T_c - \{t_i\}} Pre(t_j), cond, ex\big) \vee t_i \notin tirable\big(m - \sum_{t_j \in T_c - \{t_i\}} Pre(t_j) + \sum_{t_j \in T_c - \{t_i\}} Post(t_j), cond, ex\big)$

Par définition, le fait qu'une transition soit en conflit effectif dépend non seulement de la valeur du marquage à l'instant θ mais aussi de la valeur des conditions à ce même instant θ. Or il est impossible de connaître par avance l'évolution des variables externes au système et donc la valeur des conditions des transitions dans le cas général. Nous ne pourrons donc pas détecter automatiquement les groupes de transitions qui seront nécessairement un jour en conflit effectif ou ceux qui ne le seront jamais.

Par contre, nous pouvons détecter automatiquement les groupes de transitions qui risquent d'être en conflit effectif. Il s'agira, par la suite, de faire en sorte que si les transitions ne sont pas en conflit effectif, les tirs de transitions ne soient pas empêchés inutilement. Il existe ainsi un type de conflit qui ne dépend ni de la valeur du marquage ni de celles des conditions, appelé conflit structurel.

La définition suivante est proposée pour le conflit structurel dans [20] : une transition t_i est en conflit structurel avec la transition t_j si le tir de t_j entraîne :

- soit la baisse du marquage d'une place entrante p de t_i liée à t_i par un arc classique (cas (a) figure 2.20) ou test (cas (b) figure 2.20), i.e. $\exists\ p \in P | Pre(t_j)(p) \neq 0 \wedge \big(Pre(t_i)(p) \neq 0 \vee Pre_t(t_i)(p) \neq 0\big)$
- soit la hausse du marquage d'une place entrante de p de t_i liée à t_i par un arc inhibiteur (cas (c) figure 2.20),i.e. $\exists\ p \in P | Post(t_j)(p) \neq 0 \wedge Pre_i(t_i)(p) \neq 0$
- soit les deux.

Comme pour les conflits effectifs, la relation "être en conflit structurel" n'est pas symétrique. Il suffit de reprendre les exemples donnés figure 2.17. t_6 est en conflit structurel

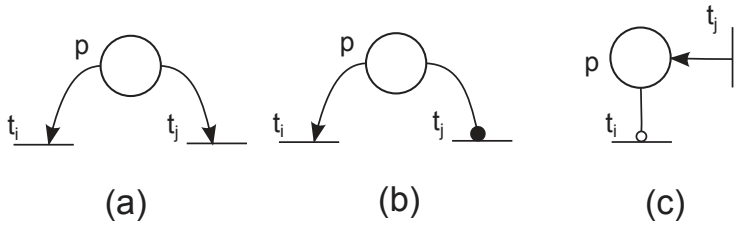

FIGURE 2.20 – Exemples de configurations de transitions en conflit structurel

avec t_5 mais pas t_5 ne l'est pas avec t_6. t_9 est déjà bien considéré comme en conflit structurel avec t_{10}, avec la définition du conflit structurel proposée. La définition peut donc être réutilisée pour notre implémentation synchrone.

Definition 2.2.5. *Dans un RdP GEISP,* $\forall(t_i, t_j) \in T^2, t_i$ *est en conflit structurel avec* t_j *pour un état* $s = (m, cond, ex)$*, noté* $t_i \in str(t_j, s)$*,*
$\Leftrightarrow \Big(\exists p \in (Pre(t_i) \cup Pre_t(t_i)) | Pre(t_j)(p) \neq 0\Big) \vee \Big(\exists p \in Post(t_i) | Pre_i(t_j)(p) \neq 0\Big)$

Une transition en conflit effectif avec une autre transition est nécessairement en conflit structurel avec cette même transition. En effet, pour que le tir d'une transition t_i empêche le tir d'une transition t_j, il faut nécessairement que le tir de t_i, soit retire des jetons à au moins une place liée à t_j par un arc classique ou un arc test, soit ajoute des jetons à au moins une place liée à t_j par un arc inhibiteur, soit les deux. Ainsi détecter automatiquement tous les groupes de conflits structurels nous garantit d'avoir détecté tous les groupes de conflits effectifs. Par contre, la réciproque n'est pas vraie et nous risquons de considérer des conflits structurels qui ne seront jamais des conflits effectifs. Il faudra alors s'assurer que les tirs ne seront empêchés que dans le cas d'un conflit effectif.

Deux transitions en conflit structurel ne seront jamais en conflit effectif si elles ne sont jamais tirables simultanément, cela peut provenir soit de la structure du RdP, soit des conditions, soit du marquage initial. Par exemple, dans les RdP donnés figure 2.21, les transitions t_2 et t_3 sont en conflit structurel mais ne seront jamais en conflit effectif. De même, si deux transitions en conflit structurel ont des conditions mutuellement exclusives comme t_0 et t_1 elles ne seront jamais en conflit effectif.

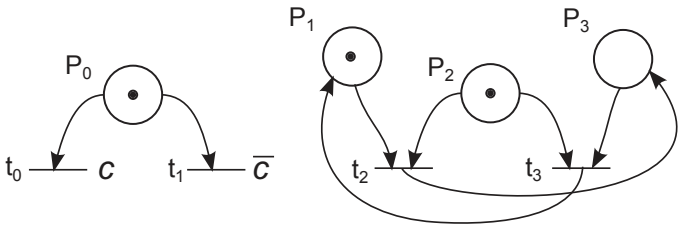

FIGURE 2.21 – Exemples de conflits structurels non effectifs

L'idée est donc d'abord d'être capable d'identifier automatiquement tous les groupes de conflits structurels. Comme pour les conflits effectifs, il est d'abord nécessaire de définir précisément la notion de groupes de transitions en conflit structurel puisque la relation n'est pas non plus symétrique. Comme pour les conflits effectifs, nous considèrons que deux transitions appartiennent au même groupe de transitions en conflit structurel si et

seulement si elles sont mutuellement en conflit structurel l'une avec l'autre. A noter que dans les classes de RdP non étendus, les groupes de transitions en conflit correspondent aux groupes de transitions ayant au moins une place entrante en commun. Ceci n'est pas vérifié dans le cadre des RdP étendus [20]. En effet, dans l'exemple donné figure 2.22, les transitions t_0 et t_1 sont un groupe de transitions en conflit structurel mais elles n'ont aucune place entrante en commun. Nous ne pouvons pas non plus négliger les conflits dus à la présence d'arcs tests puisque, figure 2.22, t_3 et t_4 sont bien mutuellement en conflit structurel.

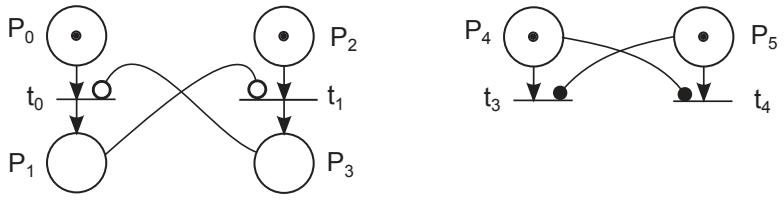

FIGURE 2.22 – Exemples de conflits structurels mutuels avec des arcs tests et inhibiteurs

Definition 2.2.6. *T_{cs} est un groupe de transitions en conflit structurel pour un état s si et seulement si : $\forall t_i \in T_{cs}, \forall t_j \in T_{cs} \backslash t_i, t_i \in str(t_j, s)$.*

L'étude de la structure du RdP permet de détecter automatiquement tous les groupes de transitions en conflit structurel. Il est alors possible de vérifier si les priorités entre les différentes transitions de ces groupes ont bien été définies. Autrement, elles peuvent être ajoutées automatiquement (et arbitrairement). Mais définir les priorités entre les transitions d'un même groupe n'est pas suffisant pour pouvoir gérer les conflits effectifs.

Pour comprendre pourquoi, étudions le RdP donné figure 2.23. Il contient 2 groupes de transitions en conflit structurel : (t_0, t_1) et (t_1, t_2). Une priorité est donc définie entre ces transitions : $t0 \prec t1$ et $t1 \prec t2$. Si nous considèrons (t_0, t_1), t_1 sera tirée puisqu'elle est plus prioritaire. Or si t_1 est tirée, t_2 ne peut plus l'être alors que t_2 était censée être plus prioritaire dans le groupe (t_1, t_2). La même inversion de priorités est observable si nous considérons d'abord le groupe (t_1, t_2). Ainsi introduire seulement des priorités entre les transitions d'un groupe de transitions en conflit structurel n'est pas suffisant, il faut aussi ordonner ces groupes quand ils ont au moins une transition en commun pour pouvoir déterminer exactement quelles transitions doivent être tirées. Ces priorités peuvent être soit définies par l'utilisateur soit arbitrairement lors de la gestion automatique des conflits. Le concepteur pourra être prévenu si les priorités entre les transitions d'un même groupe sont inversées à cause des priorités entre les groupes de transitions en conflit structurel. Une fois les inversions de priorités détectées, la liste des priorités entre transitions est modifiée ce qui permet de prendre en compte les priorités entre les groupes. Pour la suite, nous considérerons les inversions de priorités déjà prises en compte, c'est-à-dire que la liste des transitions plus prioritaires qu'une transition est correcte et respecte les priorités entre groupes de transitions en conflit.

Nous avons ainsi une méthode qui permet de s'assurer que toutes les priorités nécessaires au choix des transitions tirées en cas de conflit ont été définies par l'utilisateur ou déterminées arbitrairement. Cette méthode a été validée en l'implémentant à l'aide d'un code Python.

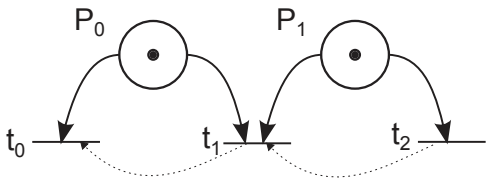

FIGURE 2.23 – Nécessité d'ordonner les groupes de transitions en conflit

Attention, la méthode ne garantit pas que le choix de priorités effectué par le concepteur ou arbitrairement est le plus pertinent au regard du comportement du système. En effet l'évalutaion de la pertinence du choix des priorités ne peut être faite automatiquement et nécessite l'expertise du concepteur. Par contre cela garantit le déterminisme du modèle conçu et donc la possibilité d'avoir un comportement cohérent entre le RdP conçu et le RdP implémenté. Pour assurer cette cohérence, il faut maintenant trouver une solution pour gérer les priorités définies sur le modèle conçu dans le code VHDL.

2.2.3 Solution d'implémentation pour les RdP saufs

Les conflits ayant tous été détectés et les priorités ayant toutes été définies, il est maintenant nécessaire de gérer ces conflits dans l'implémentation du RdP sur FPGA. Nous commençons par nous intéresser au cas particulier des RdP saufs qui est plus simple. Rappelons que la gestion des priorités (i.e. la résolution des conflits) doit nécessairement être réalisée entre le front descendant et le front montant de l'horloge (cf. §2.2.1).

Premier changement dans l'implémentation, le signal s_tir des transitions qui appartiennent à des groupes de transitions en conflit est transformé en un signal $s\_tirable$ qui indique non plus que la transition sera tirée mais qu'elle est tirable (cf. figure 2.24). Il s'agit ensuite justement de calculer les signaux s_tir de ces transitions en fonction des priorités puisque nous en avons besoin notamment pour actualiser le marquage des places mais aussi pour lancer les fonctions potentiellement associées aux transitions.

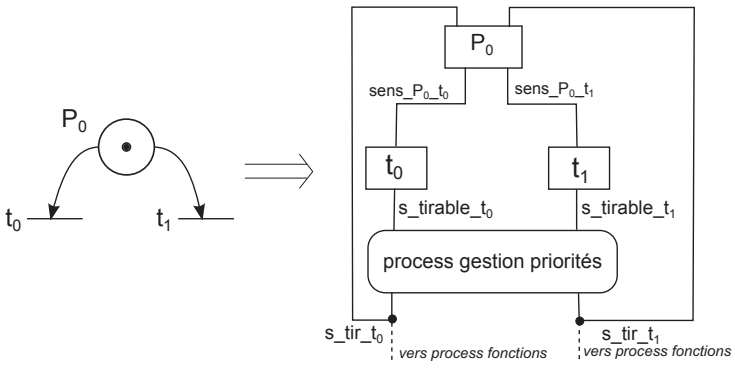

FIGURE 2.24 – Intégration du process de gestion des priorités dans l'interaction entre composants VHDL

Soit $Prio(t_i)$ l'ensemble des transitions qui ont une priorité plus grande que t_i. Comme nous travaillons dans le cas d'un RdP sauf, le marquage maximal des places étant de 1, une seule transition peut être tirée par groupe de transitions en conflit. En effet, le marquage ne peut jamais permettre d'éviter que le conflit structurel ne soit effectif. Ainsi nous savons

qu'une transition ne peut être tirée que si aucune des transitions qui sont plus prioritaires qu'elle n'a été tirée. L'expression combinatoire suivante permet donc de définir le signal tir de chaque transition.

$$s\_tir\_t_i = s\_tirable\_t_i . \prod_{j \in Prio(t_i)} (\overline{s\_tir\_t_j})$$

Par exemple, pour le RdP donné figure 2.25, en ayant (t_2, t_3) plus prioritaire que (t_0, t_1, t_2) les expressions combinatoires décrivant les signaux tir sont les suivantes :

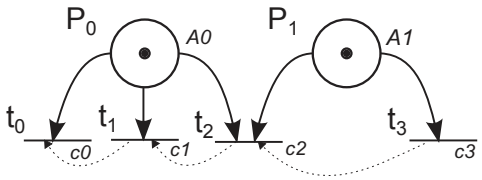

FIGURE 2.25 – Exemple de RdP

$$\begin{aligned}
s_tir_t_3 &= s_tirable_t_3 \\
s_tir_t_2 &= s_tirable_t_2 . \overline{s_tir_t_3} \\
s_tir_t_1 &= s_tirable_t_1 . \overline{s_tir_t_2} \\
s_tir_t_0 &= s_tirable_t_0 . \overline{s_tir_t_1} . \overline{s_tir_t_2}
\end{aligned}$$

Le problème est que nous avons accès aux signaux $s\_tirable$ de chaque transition au front descendant de l'horloge mais pas aux signaux s_tir des transitions plus prioritaires. Cependant il existe nécessairement au moins une transition t_i telle que $Prio(t_i) = \varnothing$, c'est la transition la plus prioritaire du groupe le plus prioritaire (t_3 dans notre exemple). Pour cette transition, on a nécessairement $s\_tir\_t_i = s\_tirable\_t_i$. Il est alors possible de remplacer $s\_tir\_t_i$ par son expression en fonction des signaux $s\_tirable$ dans les expressions combinatoires des transitions moins prioritaires. Il est ainsi possible de calculer séquentiellement hors ligne l'expression des différents signaux $s\_tir\_t_i$ en fonction uniquement des signaux $s\_tirable\_t_j$ des autres transitions plus prioritaires. Par exemple, les expressions combinatoires finales obtenues pour le RdP présenté figure 2.25 sont les suivantes :

$$\begin{aligned}
s_tir_t_3 &= s_tirable_t_3 \\
s_tir_t_2 &= s_tirable_t_2 . \overline{s_tirable_t_3} \\
s_tir_t_1 &= s_tirable_t_1 . \overline{(s_tirable_t_2 . \overline{s_tirable_t_3})} \\
s_tir_t_0 &= s_tirable_t_0 . \overline{\left(s_tirable_t_1 . (\overline{s_tirable_t_2} + s_tirable_t_3)\right)} \\
&\quad . \overline{\left(s_tirable_t_2 . \overline{s_tirable_t_3}\right)}
\end{aligned}$$

Une fois les expressions combinatoires déterminées, elles peuvent être intégrées au code VHDL. Après chaque front descendant, le calcul des nouveaux signaux $s\_tir\_t_i$ est effectué. Ils sont ensuite maintenus jusqu'au front descendant suivant. Il est possible de

voir avec le signal $s\_tir\_t_0$ que les expressions combinatoires pourraient être simplifiées. Cela nécessiterait un traitement supplémentaire des expressions combinatoires. Comme les logiciels d'implémentation de code VHDL sur FPGA réalisent lors de la synthèse une simplification du code VHDL pour limiter la surface et la consommation nécessaire, les expressions combinatoires seront simplifiées au final sur le FPGA. Il n'est donc pas nécessaire que nous réalisions cette simplification dans le code VHDL. La figure 2.26 montre une évolution de l'implémentation du réseau de Petri donné figure 2.25 obtenue en utilisant les formules données. Cette évolution a été obtenue par simulation sur le logiciel Libero. Ci donne la valeur de la condition c_i et $A(Pi)$ donne la valeur des actions associées à la place P_i et donc une image du marquage de la place P_i avec un décalage d'une demi-période d'horloge. Nous pouvons observer que, quand toutes les transitions sont tirables, seules $t3$ et $t1$ sont tirées alors que si $t3$ n'est pas tirable ($c3 = 0$), seule $t2$ est tirée. Les conflits sont donc bien gérés grâce aux équations combinatoires. Pour illustrer la simplification des expressions logiques réalisées par Libero, le schéma des portes logiques utilisées pour implémenter les expressions combinatoires est donné figure 2.27.

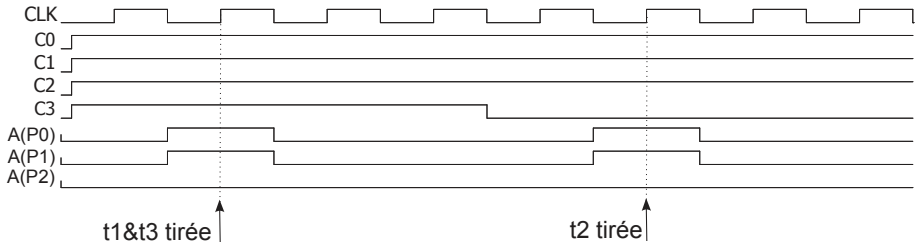

FIGURE 2.26 – Evolution du RdP donné Fig. 2.25 avec $m_0 = (3, 2)$

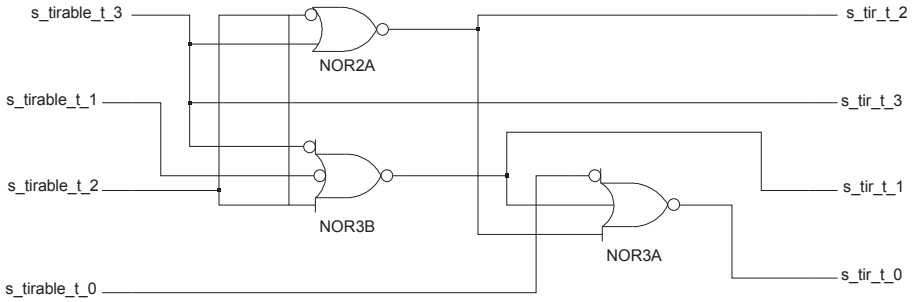

FIGURE 2.27 – Schéma logique implémenté pour le calcul des signaux de tirs des transitions en conflit

A noter que comme les calculs combinatoires sont réalisés de manière asynchrone, il faudra donc s'assurer lors de la synthèse du FPGA, qu'ils seront toujours réalisés en moins d'une demi-période d'horloge pour pouvoir garantir que le comportement sera correct. Ceci peut être vérifié sur les logiciels de synthèse de FPGA. Comme ces calculs sont très simples (logique combinatoire), cette hypothèse est réaliste pour la plupart des fréquences utilisées sur les FPGA dans notre contexte.

Cette méthode permet donc de gérer les conflits dans le cas d'un RdP sauf. Cette solution est intéressante car nécessairement plus simple que celles pour le cas des RdP généralisés. Elle pourra ainsi être utilisée, dans le cadre des RdP généralisés, pour gérer

des groupes de transitions en conflit structurel s'il est possible de démontrer que toutes les places concernées par ce conflit ont toutes un marquage maximum égal à 1.

2.2.4 Solution d'implémentation pour les RdP généralisés

Contrairement aux RdP saufs, dans le cas des RdP généralisés, plus d'une transition peuvent être tirées par groupe de transitions en conflit direct suivant le marquage du RdP. En effet, si dans le RdP donné figure 2.25, la place P_1 contient 2 jetons, les deux transitions t_2 et t_3 ne sont plus en conflit effectif (mais toujours en conflit structurel) et les deux transitions doivent être tirées dans le RdP implémenté pour correspondre au RdP conçu (même s'il peut paraître surprenant de modéliser ainsi un ET). Ainsi le marquage doit désormais être utilisé pour la détermination des transitions qui seront tirées lors de la résolution du conflit. Bien sûr nous avons toujours la contrainte de réaliser cette détermination entre les deux fronts d'horloge. Deux solutions peuvent être envisagées, une solution séquentielle et une solution combinatoire. Les solutions proposées seront illustrées sur la même structure que pour les RdP binaires (cf. figure 2.25).

Solution séquentielle

Le problème est de pouvoir déterminer si le marquage des places est suffisant pour pouvoir tirer des transitions de plus basses priorités. Pour cela, au lieu de gérer le signal s_tir de toutes les transitions simultanément comme dans le cas des RdP saufs, les signaux de tir de chaque transition d'un groupe de conflit structurel seront gérés les uns après les autres en commençant par la transition la plus prioritaire. Un marquage virtuel $M_{virtuel}(P_j)$ est alors défini pour chaque place intervenant dans le conflit. Ce marquage est initialisé à la valeur du marquage de la place du RdP quand le conflit commence à être géré. A chaque décision de tir d'une transition, le marquage virtuel des places évolue en fonction jusqu'à ce que toutes les décisions de tir soient prises (cf. figure 2.28).

Soit $Places(t_i)$ l'ensemble des places liées à t_i par un arc classique qui sont concernées par au moins un conflit structurel incluant t_i. De même, nous définissons $Places_t(t_i)$ l'ensemble des places liées à t_i par un arc test et $Places_i(t_i)$ l'ensemble de celles liées par un arc inhibiteur. La formule pour déterminer si une transition sera tirée est alors :

$$s\_tir\_t_i = s\_tirable\_t_i \cdot \prod_{P_j \in Places(t_i)} \Big(M_{virtuel\_P_j} \geq Pre(t_i)(p_i) \Big) \cdot \prod_{P_j \in Places_t(t_i)} \Big(M_{virtuel\_P_j} \geq Pre_t(t_i)(p_i) \Big) . \prod_{P_j \in Places_i(t_i)} \Big(M_{virtuel\_P_j} < Pre_i(t_i)(p_i) \Big)$$

Plus précisément, les décisions de tirer une transition ne sont pas prises les unes après les autres mais par niveau de priorité. Un niveau de priorité est composé de transitions qui ne sont pas dans le même groupe de conflit et pour lesquelles la décision de tir peut être prise simultanément (i.e. des transitions qui doivent attendre le même nombre de prises de décisions avant de pouvoir être étudiées).

Le problème est d'être sûr que les marquages virtuels seront stables quand ils seront utilisés pour évaluer la valeur des signaux de tirs. C'est le même problème que celui que nous avons rencontré pour les fonctions dans le cadre de l'implémentation asynchrone des RdP interprétés (cf. §2.1.1). Il est alors nécessaire de travailler de manière synchrone en

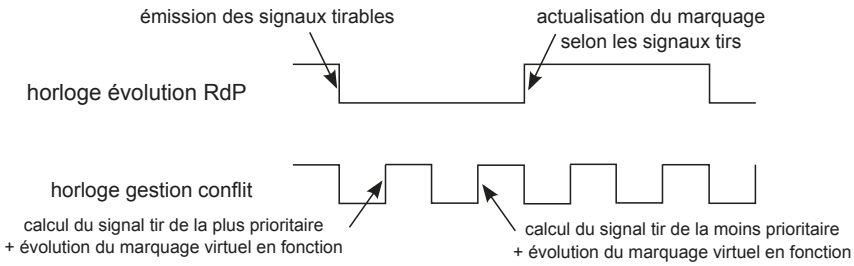

FIGURE 2.28 – Lien entre les deux horloges de la solution séquentielle dans le cas de deux transitions en conflit

réalisant un niveau de priorité par période d'horloge. Pour ne pas ralentir l'évolution du RdP, il faut travailler avec une seconde horloge, appelée horloge conflit, plus rapide que celle utilisée pour faire évoluer le RdP, appelée horloge du RdP (cf. figure 2.28). Pour pouvoir prendre toutes les décisions de tir entre les deux fronts d'horloge RdP, l'horloge conflit doit être $2n$ fois plus rapide que l'horloge RdP où n est le nombre maximal de niveaux de priorités dans tous les conflits du modèle, appelé profondeur du conflit. Un extrait du process VHDL utilisé pour gérer les conflits de l'exemple de la figure 2.25 est donné figure 2.29. Dans cet exemple, clk_prio est 8 fois plus rapide que clk (l'horloge cadençant l'évolution RdP). Une évolution de ce même RdP est donné figure 2.30 dans le cas où toutes les transitions sont tirables et où les places P_0 et P_1 contiennent respectivement 3 et 2 jetons.

Ainsi, la résolution séquentielle permet de résoudre les conflits effectifs en utilisant une seconde horloge plus rapide. Cependant cette solution peut ne pas être utilisable dans le cadre de certains systèmes embarqués ou pour des modèles trop grands. En effet, il peut être impossible d'avoir une horloge assez rapide pour gérer les conflits car soit la profondeur des conflits est trop grande soit la surconsommation qu'entraîne l'utilisation d'une horloge plus rapide ne peut pas être acceptée. A noter que la consommation de courant d'un FPGA est proportionnelle à la fréquence de l'horloge.

Solution combinatoire

Nous recherchons une autre solution qui nous permet de ne pas devoir utiliser une horloge plus rapide. L'idée est alors de s'inspirer de la solution utilisée pour les RdP saufs en déterminant hors-ligne les expressions combinatoires qui calculent en ligne les signaux tirs des transitions. En effet, nous avons vu qu'il était possible de réaliser les traitements combinatoires de manière asynchrone sans problème. Il faut donc être capable d'exprimer qu'une transition doit être tirée en n'utilisant que les signaux de tirabilité (franchissabilité) des autres transitions et le marquage des places.

Une transition impliquée dans un conflit peut être tirée si et seulement si les transitions plus prioritaires ne sont pas tirées ou si le marquage des places qu'elles ont en commun est suffisant pour pouvoir tirer cette transition en plus des transitions plus prioritaires. Le principe est donc, pour chaque transition t_i, de déterminer pour chaque combinaison possible des signaux tirs des transitions plus prioritaires le marquage nécessaire pour permettre le tir de t_i. Nous avons alors besoin de définir une notation pour exprimer les différentes combinaisons possibles de valeurs de variables booléennes dans le cadre d'un produit.

```
-- gestions des priorités
--process priorites
calcul_prio_transition : process (reset_n, clk_prio, clk)
    type enumetat is (choix_0,choix_1,choix_2,choix_3,choix_4);
    variable etat_prio : enumetat;
    variable marqu_virtuel_place_1 : natural range 0 to mark_max_net;
    variable marqu_virtuel_place_0 : natural range 0 to mark_max_net;
    variable traitement_termine : std_logic;

begin
 if (reset_n = '0') then
            etat_prio := choix_4;
            marqu_virtuel_place_1 :=0;
            marqu_virtuel_place_0 :=0;
            s_tir_init_0 <='0';
            s_tir_init_1 <='0';
            s_tir_transition_0 <='0';
            s_tir_transition_1 <='0';
            s_tir_transition_2 <='0';
            s_tir_transition_3 <='0';
elsif (clk='1') then
    traitement_termine:='0';
    marqu_virtuel_place_1 := s_markup_place_1;
    marqu_virtuel_place_0 := s_markup_place_0;
    etat_prio := choix_0;
elsif (clk_prio'event and clk_prio='1') then
    case etat_prio IS
        when choix_0 =>
        if(s_tirable_transition_3 = '1' and marqu_virtuel_place_1 >= 1) then
            s_tir_transition_3 <= '1';
            marqu_virtuel_place_1 := marqu_virtuel_place_1 - 1;
        else
            s_tir_transition_3 <='0';
            marqu_virtuel_place_1 := marqu_virtuel_place_1;
        end if;
        etat_prio := choix_1;

        when choix_1 =>
        if(s_tirable_transition_2 = '1' and marqu_virtuel_place_0 >= 1 and
            marqu_virtuel_place_1 >= 1) then
            s_tir_transition_2 <= '1';
            marqu_virtuel_place_0 := marqu_virtuel_place_0 - 1;
            marqu_virtuel_place_1 := marqu_virtuel_place_1 - 1;
        else
            s_tir_transition_2 <='0';
            marqu_virtuel_place_0 := marqu_virtuel_place_0;
            marqu_virtuel_place_1 := marqu_virtuel_place_1;
        end if;
        etat_prio := choix_2;

        when choix_2 =>
        if(s_tirable_transition_1 = '1' and marqu_virtuel_place_0 >= 1) then
            s_tir_transition_1 <= '1';
            marqu_virtuel_place_0 := marqu_virtuel_place_0 - 1;
        else
            s_tir_transition_1 <='0';
            marqu_virtuel_place_0 := marqu_virtuel_place_0;
        end if;
        etat_prio := choix_3;

        when choix_3 =>
        if(s_tirable_transition_0 = '1' and marqu_virtuel_place_0 >= 1) then
            s_tir_transition_0 <= '1';
            marqu_virtuel_place_0 := marqu_virtuel_place_0 - 1;
        else
            s_tir_transition_0 <='0';
            marqu_virtuel_place_0 := marqu_virtuel_place_0;
        end if;
        etat_prio := choix_4;

        when choix_4 =>
            traitement_termine='1';
    end case;
end if;
end process;
```

FIGURE 2.29 – Process de gestion des priorités (solution séquentielle) pour le RdP Fig.2.25

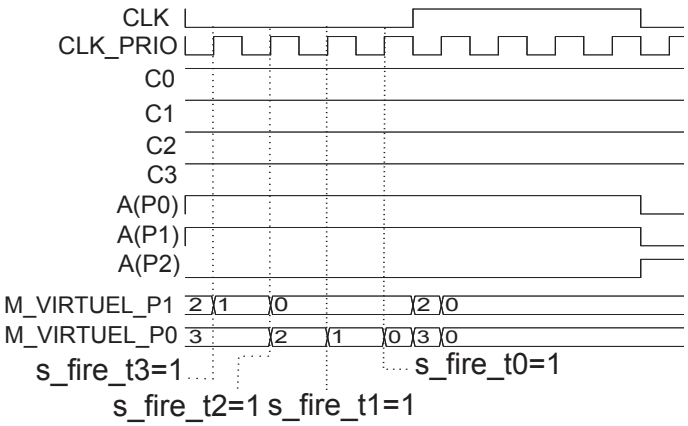

FIGURE 2.30 – Evolution du RdP Fig.2.25(solution séquentielle) avec $m_0 = (3, 2)$

$\forall x \in \mathbb{B}$, nous utilisons la notation suivante x^{b_i} avec $b_i \in \mathbb{B}$ telle que $x^1 = x$ et $x^0 = \bar{x}$, inspirée de [45]. Soit $B_n = b_1, \ldots, b_n$ un vecteur de n booléens. Nous avons alors par exemple $\sum_{B_2 \in \{0,1\}^2} \prod_{i=1}^{i=2} x_i^{b_i} = x_1.x_2 + \bar{x_1}.x_2 + x_1.\bar{x_2} + \bar{x_1}.\bar{x_2}$.

Soit n_i le nombre de transitions de l'ensemble $Prio(t_i)$. Les équations combinatoires pour gérer les conflits dans le cas des RdP généralisés, dont les différents termes sont expliqués par la suite, sont :

$$
\begin{aligned}
s\_tir\_t\_i = & s\_tirable\_t\_i. \sum_{B_{n_i} \in \{0,1\}^{n_i}} \Big[\prod_{t_j \in Prio(t_i)} s\_tir\_t_j^{b_j} \\
& . \prod_{P_j \in Places(t_i)} \Big(m(p_j) \geq Pre(t_i)(p_j) + \sum_{t_j \in Prio(t_i)} (Pre(t_j)(pj) \times b_j) \Big) \\
& . \prod_{P_j \in Places_t(t_i)} \Big(m(p_j) \geq Pre_t(t_i)(p_j) + \sum_{t_j \in Prio(t_i)} (Pre(t_j)(pj) \times b_j) \Big) \\
& . \prod_{P_j \in Places_i(t_i)} \Big(m(p_j) + \sum_{t_j \in Prio(t_i)} (Post(t_j)(pj) \times b_j) < Pre_i(t_i)(p_j) \Big) \Big]
\end{aligned}
$$

Dans cette formule, la somme permet de considérer toutes les combinaisons possibles pour les signaux tirs des transitions plus prioritaires. Le premier produit donne la configuration de signaux tirs considérée : si $b_j = 1$, la transition est t_j est tirée sinon elle ne l'est pas. Le deuxième produit permet de déterminer si le marquage des places entrantes liées à t_i par un arc classique est suffisant pour tirer toutes les transitions plus prioritaires tirées dans la configuration considérée et la transition t_i. Le fait de multiplier par b_k permet de ne tenir compte que des poids liés à des transitions réellement tirées dans la configuration (multiplication par 0 sinon). Le troisième et quatrième produit s'intéressent eux respectivement au marquage des places liées à t_i par des arcs tests et à celui des places liées à t_i par des arcs inhibiteurs.

Dans le premier produit $\left(\prod_{t_j \in Prio(t_i)} s\_tir\_t_j^{b_j} \right)$ décrivant la combinaison de signaux tirs considérés, les signaux tirs qui sont effectivement tirés peuvent être omis. En effet, si

le marquage est suffisant pour que la transition moins prioritaire soit tirée malgré le tir d'une transition plus prioritaire, il est nécessairement aussi suffisant si le transition plus prioritaire n'est pas tirée. Cela permet de simplifier les expressions combinatoires.

Les équations combinatoires obtenues pour l'exemple de la figure 2.25 sont :

$$
\begin{aligned}
s\_tir\_t\_3 =& s\_tirable\_t\_3 \\
s\_tir\_t\_2 =& s\_tirable\_t\_2.(s\_tir\_t\_3.M_{P1} \geq 2 + \overline{s\_tir\_t\_3}.M_{P1} \geq 1) \\
s\_tir\_t\_1 =& s\_tirable\_t\_1.(s\_tir\_t\_2.M_{P0} \geq 2 + \overline{s\_tir\_t\_2}.M_{P0} \geq 1) \\
s\_tir\_t\_0 =& s\_tirable\_t\_0.\,(s\_tir\_t\_1.s\_tir\_t\_2.M_{P0} \geq 3 \\
& + \overline{s\_tir\_t\_1}.s\_tir\_t\_2.M_{P0} \geq 2 \\
& + s\_tir\_t\_1.\overline{s\_tir\_t\_2}.M_{P0} \geq 2 \\
& + \overline{s\_tir\_t\_1}.\overline{s\_tir\_t\_2})
\end{aligned}
$$

Ces équations logiques doivent ensuite être traitées hors-ligne pour remplacer les signaux de tir des transitions plus prioritaires par leur expression combinatoire comme dans le cas des RdP saufs. Les équations combinatoires qui seront implémentées dans le cas de l'exemple figure 2.25 sont alors :

$$
\begin{aligned}
s\_tir\_t\_3 =& s\_tirable\_t\_3 \\
s\_tir\_t\_2 =& s\_tirable\_t\_2.(M_{P1} \geq 2 + \overline{s\_tirable\_t\_3}) \\
s\_tir\_t\_1 =& s\_tirable\_t\_1.(M_{P0} \geq 2 + \overline{s\_tirable\_t\_2.(M_{P1} \geq 2 + \overline{s\_tirable\_t\_3})}) \\
s\_tir\_t\_0 =& s\_tirable\_t\_0.\,\big(M_{P0} \geq 3 + \overline{s\_tirable\_t\_1} \\
& \overline{.(M_{P0} \geq 2 + \overline{s\_tirable\_t\_2.(M_{P1} \geq 2 + \overline{s\_tirable\_t\_3})})}.M_{P0} \geq 2 \\
& + \overline{s\_tirable\_t\_2.(M_{P1} \geq 2 + \overline{s\_tirable\_t\_3})}.M_{P0} \geq 2 \\
& + \overline{s\_tirable\_t\_1.(M_{P0} \geq 2 + \overline{s\_tirable\_t\_2.(M_{P1} \geq 2 + \overline{s\_tirable\_t\_3})})} \\
& \overline{.s\_tirable\_t\_2.(M_{P1} \geq 2 + \overline{s\_tirable\_t\_3})}\big)
\end{aligned}
$$

Il est facile de réaliser que ces expressions combinatoires peuvent rapidement devenir très complexes dans le cas de conflits avec un grand niveau de profondeur. En effet, il est nécessaire de décrire toutes les situations possibles des signaux tirs des transitions de plus haute priorité (il existe 2^n possiblités pour n transitions) mais en plus les expressions des transitions plus prioritaires sont réinjectées dans les expressions des transitions moins prioritaires. Ceci peut mener à une explosion combinatoire. Cette explosion combinatoire est très vite observable : par exemple, la formule complète de $s\_tir\_t\_0$ est déjà composé de 20 éléments alors que celle de $s\_tir\_t\_1$ n'en comprenait que 5. La complexité de la formule n'est pas forcément génante en elle-même pusiqu'elle est générée de manière automatique donc il n'y a pas de risque d'erreur. De plus, la formule sera simplifiée autant que possible lors de l'implémentation sur FPGA. Reste à savoir si cette simplification sera suffisante pour avoir une surface nécessaire acceptable pour gérer les conflits. Cette explosion combinatoire peut par contre être problématique si à cause de la complexité des formules il n'est plus possible d'écrire automatiquement le code VHDL. En effet,

nous arrivons assez rapidement à la limite de la mémoire de l'ordinateur en faisant tourner le code Python qui écrit le code VHDL nécessaire à la gestion des priorités (cf. §2.2.5).

Une fois les expressions combinatoires obtenues, il faut écrire le code VHDL correspondant. Contrairement au cas des RdP saufs, le code ne peut pas être écrit directement sous la forme d'expressions combinatoires à cause des tests à réaliser sur les signaux du marquage. Une structure en $if\ elsif\ else$ est utilisée. Le code VHDL obtenu pour la gestion des priorités dans le cas de l'exemple figure 2.25 est donné figure 2.31. L'évolution observée lors de la simulation du code VHDL est donnée figure 2.32.

La solution combinatoire permet donc aussi de gérer de manière automatique les conflits et présente l'avantage de ne pas nécessiter d'horloge plus rapide. Par contre, cette solution amène à calculer des expressions complexes qui devront être implémentées sur FPGA et qui risquent donc de demander trop de surface. De plus, il faut aussi pouvoir gérer ces équations complexes. Ainsi la solution séquentielle mènera obligatoirement à une augmentation de consommation mais nous craignons que la solution combinatoire mène à une augmentation de la surface. Une comparaison des deux solutions s'avère donc nécessaire.

2.2.5 Comparaison des 2 solutions

Comme les deux solutions, combinatoire et séquentielle, présentent un comportement correct (les conflits sont bien gérés), il s'agit de pouvoir comparer les deux solutions en termes de consommation et de surface. Il existe différentes configurations de conflit possibles et une solution n'est pas nécessairement la meilleure pour toutes ces configurations. Nous étudierons donc les deux solutions sur des exemples "simples" mais ayant des structures caractéristiques : beaucoup de transitions en conflit structurel direct, beaucoup de places communes...

Les différents modèles exemples ont donc été créés grâce au logiciel HILECOP. Une fois le modèle traduit automatiquement en code VHDL (avec la méthode de résolution de conflit choisie) à l'aide du logiciel HILECOP et d'un code Python, il est intégré à un projet Libero puis implémenté sur un FPGA de type IGLOO de Microsemi (ref.AGL1000v2). Un banc dédié de mesure de consommation est utilisé pour étudier la consommation. Cela permet de s'assurer que les mesures estimées par Libero sont fiables (qualitativement). Ce banc alimente le FPGA avec une tension de 1,2 V (Vcore = 1,2V and Vbank = 2,5V). La tension noyau a été mesurée grâce à un multimètre professionnel Fluke 27II. Le comportement du FPGA a été vérifié grâce à l'observation des signaux générés par le RdP sur un oscilloscope. Ces mesures de consommations ont été réalisées par un ingénieur Axonic (notre partenaire industriel).

Nous commençons par comparer les deux solutions sur un exemple simple donné figure 2.34. Cela permet de comparer les deux solutions sur un petit nombre de transitions en conflit et un petit nombre de groupes de transitions en conflit. Les résultats obtenus sont donnés dans le tableau 2.1. Dans la colonne $surface$, le nombre de portes logiques requises pour implémenter le RdP est donné. Ce nombre est une estimation donnée après la synthèse par Libero. Dans la colonne *consommation*, la consommation de courant mesurée est indiquée en μA.

```
-- gestions des priorités
prio_transition : process (clk, s_tirable_transition_0, s_tirable_transition_1,
    s_tirable_transition_2, s_tirable_transition_3)
begin

if (s_tirable_transition_0 = '1') then
    if ( s_markup_place_0 >=3 ) then
        s_tir_transition_0 <='1';
    elsif ( not (s_tirable_transition_1 ='1' and (( s_markup_place_0 >=2 ) or (not (
        s_tirable_transition_2 ='1' and (( s_markup_place_1 >=2 ) or (not (
        s_tirable_transition_3 ='1')))))))) and s_markup_place_0 >=2 ) then
        s_tir_transition_0 <='1';
    elsif ( not (s_tirable_transition_2 ='1' and (( s_markup_place_1 >=2 ) or (not (
        s_tirable_transition_3 ='1')))) and s_markup_place_0 >=2 ) then
        s_tir_transition_0 <='1';
    elsif ( not (s_tirable_transition_1 ='1' and (( s_markup_place_0 >=2 ) or (not (
        s_tirable_transition_2 ='1' and (( s_markup_place_1 >=2 ) or (not (
        s_tirable_transition_3 ='1')))))))
 and not (s_tirable_transition_2 ='1' and (( s_markup_place_1 >=2 ) or (not (
     s_tirable_transition_3 ='1'))))) then
        s_tir_transition_0 <='1';
    else
        s_tir_transition_0 <='0';
    end if;
else
    s_tir_transition_0 <='0';
end if;
if (s_tirable_transition_1 = '1') then
    if ( s_markup_place_0 >=2 ) then
        s_tir_transition_1 <='1';
    elsif ( not (s_tirable_transition_2 ='1' and (( s_markup_place_1 >=2 ) or (not (
        s_tirable_transition_3 ='1'))))) then
        s_tir_transition_1 <='1';
    else
        s_tir_transition_1 <='0';
    end if;
else
    s_tir_transition_1 <='0';
end if;
if (s_tirable_transition_2 = '1') then
    if ( s_markup_place_1 >=2 ) then
        s_tir_transition_2 <='1';
    elsif ( not (s_tirable_transition_3 ='1')) then
        s_tir_transition_2 <='1';
    else
        s_tir_transition_2 <='0';
    end if;
else
    s_tir_transition_2 <='0';
end if;
s_tir_transition_3 <= s_tirable_transition_3 ;
end process;
```

FIGURE 2.31 – Process de gestion des priorités (solution combinatoire) pour le RdP Fig.2.25

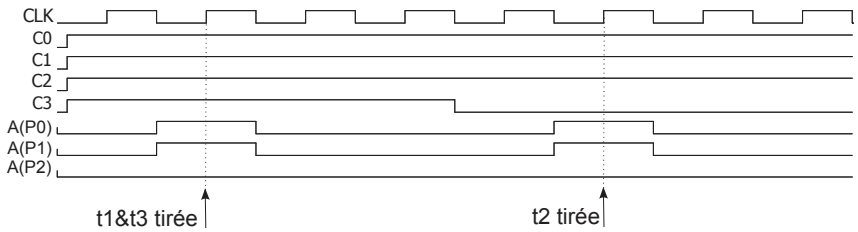

FIGURE 2.32 – Evolution du RdP Fig.2.25 (solution binaire)

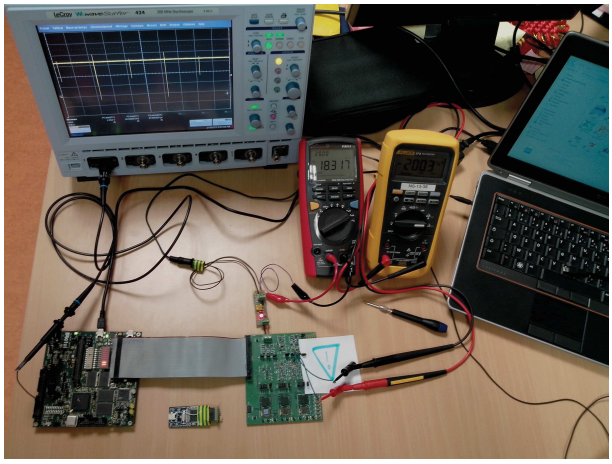

FIGURE 2.33 – Banc dédié à la mesure de consommation

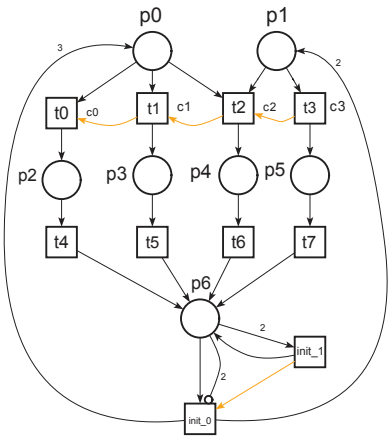

FIGURE 2.34 – Exemple simple utilisé pour la comparaison

	surface	consommation
séquentiel	480	224,0 μA
combinatoire	333	25,4 μA

TABLE 2.1 – Résultats pour le RdP donné Fig.2.34

Ces résultats montrent que pour ce modèle très simple, la solution combinatoire est la meilleure pour les 2 critères. En effet, il n'y a pas d'explosion combinatoire et le nombre de portes logiques nécessaire pour gérer l'évolution du marquage virtuel est donc plus grand que celui nécessaire pour gérer les expressions combinatoires. De plus, comme les expressions combinatoires utilisent plusieurs fois les mêmes termes, les portes logiques peuvent donc être mutualisées ce qui diminue encore le nombre de portes logiques nécessaires. Nous souhaitons maintenant savoir si le poids des arcs a une influence sur ces résultats. Le modèle est ainsi modifié en multipliant par 10 les poids des arcs et le marquage initial. Les résultats obtenus sont donnés dans le tableau 2.2.

	surface	consommation
séquentiel	667	228 μA
combinatoire	473	36,1 μA

TABLE 2.2 – Résultats pour le RdP Fig.2.34 avec des poids multipliés par 10

Ainsi le poids des arcs influe bien sur la surface et la consommation mais il a la même influence pour les 2 solutions. Ce ne semble donc pas un critère pertinent pour déterminer la meilleure solution.

Nous nous intéressons maintenant à la comparaison des deux solutions sur différentes structures. Un groupe de transitions en conflit est noté G_i, le nombre de transitions en conflit $Dim_{G_i^t}$ et le nombre de places impliquées dans le conflit $Dim_{G_i^p}$. Un groupe de groupes de transitions en conflit est noté GC_i et sa dimension (i.e. le nombre de groupes) Dim_{GC_i}. Il y a un risque d'explosion combinatoire dans le cas où il y a trop de transitions dans un même groupe de conflits. Ainsi commençons par étudier le cas où Dim_{G_i} est grand (cf. fig. 2.35). Les résultats sont donnés dans le tableau 2.3.

Comme prédit, la consommation est significativement plus grande avec la solution séquentielle. Par contre, la surface nécessaire pour l'implémentation est du même ordre de grandeur pour les 2 solutions. Néanmoins, il est important de noter que le test a été réalisé avec $Dim_{G_1} = 7$ car avec un nombre plus grand de transitions en conflit, nous ne sommes pas parvenu à obtenir automatiquement le code VHDL sur notre ordinateur (3,1 Ghz - 4Go RAM). La mémoire interne n'a en effet pas été suffisante pour permettre au code Python de s'exécuter complétement. Le problème provient de l'explosion combinatoire décrite dans le §2.2.4.

Pour avoir une idée de l'impact de l'explosion combinatoire sur la taille du code VHDL nous pouvons comparer la taille des fichiers textes le contenant. Pour $Dim_{G_1} = 6$, nous obtenons un fichier de 5 Mo alors que pour $Dim_{G_1} = 7$ nous arrivons déjà à 15 Mo. A titre de comparaison, pour ces deux cas, les fichiers textes décrivant le code VHDL avec la méthode séquentielle font moins de 25 Ko. Ainsi l'explosion combinatoire ne posera

visiblement pas problème d'un point de vue surface nécessaire sur le FPGA car la simplification des expressions et la mutualisation des portes logiques est efficace. Par contre la difficulté sera de réussir à générer automatiquement le code VHDL.

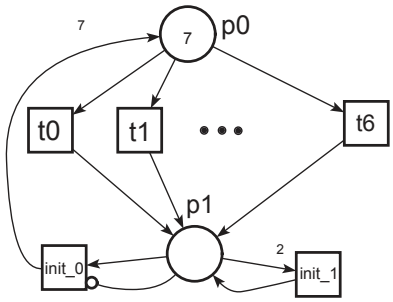

FIGURE 2.35 – Partie du modèle pour $Dim_{G_1^t} = 7$ et $Dim_{G_1^p} = 1$

	surface	consommation
séquentiel	356	343,0 μA
combinatoire	229	24,3 μA

TABLE 2.3 – Résultats pour $Dim_{G_1^t} = 7$ et $Dim_{G_1^p} = 1$

Nous comparons maintenant les solutions sur un RdP (figure 2.36) contenant un conflit avec beaucoup de groupes de transitions en conflit dont les caractéristiques sont $Dim_{G_1^t} = 2$ et $Dim_{G_1^p} = 20$. Les résultats obtenus sont donnés dans le tableau 2.4. Nous observons que dans ce cas la solution combinatoire est la meilleure.

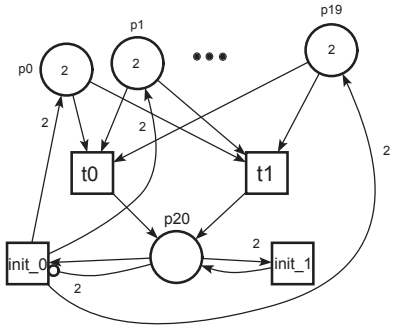

FIGURE 2.36 – Partie du modèle pour $Dim_{G_1^t} = 2$ et $Dim_{G_1^p} = 20$

	surface	consommation
séquentiel	1785	387 μA
combinatoire	923	57,5 μA

TABLE 2.4 – Résultats pour $Dim_{G_1^t} = 2$ et $Dim_{G_1^p} = 20$

Pour terminer, la comparaison est réalisée sur un RdP (figure 2.37) tel que $\forall i, Dim_{G_i^t} = 2$ et $Dim_{GC_1} = 10$, c'est-à-dire avec un conflit dans lequel beaucoup de places sont impliquées. Les résultats (donnés tableau 2.5) montrent que pour cette structure la solution combinatoire est encore une fois une meilleure solution pour nos critères.

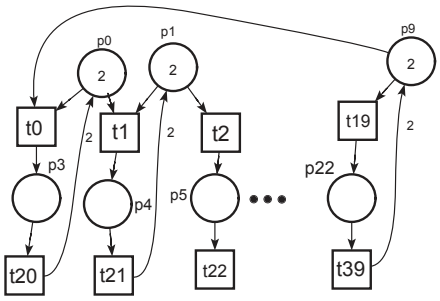

FIGURE 2.37 – Partie du modèle pour $Dim_{G_i^t} = 2$ et $Dim_{GC_1} = 10$

	surface	consommation
séquentiel	2651	2420 μA
combinatoire	1587	74,6 μA

TABLE 2.5 – Résultats pour $Dim_{G_i^t} = 2$ et $Dim_{GC_1} = 10$

D'après les différents résultats obtenus, il apparaît que, quand elle est utilisable, la solution combinatoire est la meilleure aussi bien pour minimiser la surface que la consommation. Nous utiliserons donc toujours cette solution quand il est nécessaire d'avoir une solution pour les RdP généralisés. Le problème est que quand nous avons un groupe avec trop de transitions en conflit, il est parfois impossible d'obtenir le code VHDL permettant de gérer tous les conflits. Ce problème n'est pas à négliger car au niveau industriel il est tout à fait envisageable de rencontrer des modèles où il y a un conflit structurel direct impliquant plus de 7 transitions.

Il est néanmoins intéressant d'avoir les deux solutions. Dans d'autres contextes ou avec d'autres modèles, la solution séquentielle pourrait en effet être plus pertinente. Il est possible de laisser à l'utilisateur le choix de la méthode qu'il souhaite utiliser pour gérer les conflits.

Puisque le problème est la génération du code VHDL, une solution serait de simplifier ces expressions logiques au fur et à mesure qu'elles sont calculées. En effet, pour l'instant, elles n'étaient pas simplifiées du tout au niveau VHDL puisque simplifiées par la suite par le logiciel Libero. La simplification rendra la génération automatique un peu plus complexe puisque pour être capable de simplifier les expressions logiques, le programme doit "comprendre" les expressions au lieu de manier simplement du texte. Néanmoins cette solution est envisageable puisque certains logiciels réalisent déjà des simplifications automatiques d'expressions combinatoires [66] [67]. Elle paraît de plus assez prometteuse car les expressions semblent se simplifier beaucoup dans la plupart des cas. Pour illustrer ce propos, les expressions combinatoires simplifiées pour l'exemple de la figure 2.25 ont été calculées et sont données ci-dessous. Il est clair qu'elles sont plus compactes : le nombre de termes impliqués dans $s\_tir\_t\_0$ est passé de 20 à 10.

$$s\_tir\_t\_3 = s\_tirable\_t\_3$$
$$s\_tir\_t\_2 = s\_tirable\_t\_2.(M_{P1} \geq 2 + \overline{s\_tirable\_t\_3})$$
$$s\_tir\_t\_1 = s\_tirable\_t\_1.(M_{P0} \geq 2 + \overline{s\_tirable\_t\_2.(M_{P1} \geq 2 + \overline{s\_tirable\_t\_3})})$$
$$s\_tir\_t\_0 = s\_tirable\_t\_0.\Big(M_{P0} \geq 3 + \overline{s\_tirable\_t\_1}.(\overline{s\_tirable\_t\_2} + M_{P1} < 2) + M_{P0} \geq 2.(\overline{s\_tirable\_t\_2} + \overline{s\_tirable\_t\_1} + M_{P1} < 2.\overline{s\_tirable\_t\_3})\Big)$$

Nous sommes donc désormais capables non seulement de nous assurer que toutes les priorités en cas de conflit ont bien été définies mais aussi de générer le code VHDL permettant de gérer ces priorités sur le système réel. Il nous reste alors à étudier l'impact de la présence de conflits dans un RdP sur la transformation du modèle en modèle analysable.

2.2.6 Influence des conflits sur l'analyse

Des priorités ont été ajoutées aux modèles conçu et implémenté pour gérer les conflits. Or les outils d'analyse existants permettent de gérer les priorités. Nous pourrions donc envisager de reporter les priorités définies sur le modèle conçu sur le modèle analysable. Mais ce serait une erreur car il faut aussi prendre en considération l'interprétation du modèle. Considérons, par exemple, le cas simple de conflit donné figure 2.38 et sa transformation en modèle analysable. Si nous définissons dans le modèle analysable que t_1 est plus prioritaire que t_0, comme t_0 et t_1 sont toujours simultanément sensibilisées, t_0 ne sera jamais tirée dans le modèle analysable. Or si $val(c_1) = 0$, t_1 n'est pas tirable donc c'est t_0 qui sera tiré et non t_1 dans le modèle implémenté. Ainsi nous n'aurions plus l'inclusion du comportement implémenté dans le comportement analysé. Donc les priorités du modèle conçu sont supprimées dans le modèle analysé et la méthode pour construire le modèle analysable n'est pas modifiée par les conflits.

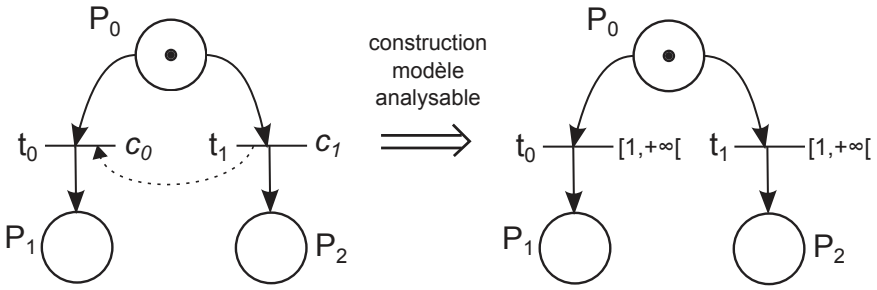

FIGURE 2.38 – Transformation d'une priorité pour l'analyse

Si nous ne définissons aucune priorité sur le modèle analysé, en cas de conflit, comme l'évolution est asynchrone, le conflit sera nécessairement géré. En effet, les transitions étant tirées les unes après les autres, si deux transitions ne peuvent pas être toutes les deux tirées, elles ne le seront pas. Si au contraire le marquage permet de tirer plusieurs transitions du conflit, ces transitions seront aussi potentiellement tirées sur le modèle analysé. Comme tous les cas possibles sont traités, le choix fait sur le modèle implémenté sera nécessairement étudié.

Si l'inclusion du marquage est toujours vraie et peut toujours être prouvée formellement (cf. B), si le réseau analysé est vivant alors le réseau implémenté (ou conçu) peut ne pas être vivant. Le réseau donné figure 2.39 est en effet un contre-exemple fournit par Moalla [43] : le RdP analysé est vivant mais le RdP implémenté ne l'est pas car t_2 n'est jamais tirable. Ceci est dû au fait que le RdP est implémenté de manière synchrone donc synchronisé sur l'événement $\downarrow clock$.

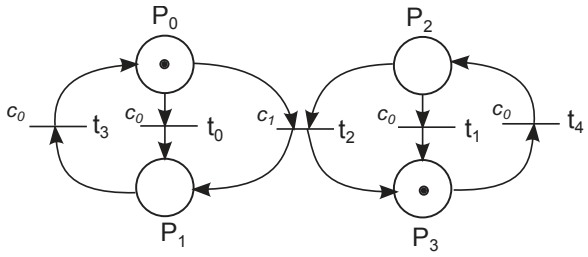

FIGURE 2.39 – Exemple d'un réseau non vivant dont le comportement analysé est vivant

Pour information, il a été démontré dans [43] que, dans le cas où le RdP est ordinaire, simple et totalement synchronisé, si un RdP asynchrone est vivant alors le RdP synchrone correspondant l'est aussi. Un RdP est simple [24] si aucune transition n'est impliquée dans deux conflits structurels différents. C'est pour cela que le problème ne se posait pas pour les réseaux de Petri sans conflit.

A noter que dans notre transformation, nous tenons compte de l'implémentation synchrone, en associant, par exemple, aux transitions sans condition l'intervalle $[1, 1]$. S'il n'y a aucune condition dans le RdP donné figure 2.39, l'analyse du RdP avec les intervalles $[1, 1]$ associés à chaque transition permettra bien de détecter par analyse que la transition t_2 n'est pas vivante. La réduction de l'écart entre système réel et modèle analysable permet donc d'améliorer les résultats d'analyse formelle par rapport à la simple analyse de la structure du RdP.

Mais dès qu'une condition est associée à une transition dans le modèle conçu, c'est l'intervalle $[1, +\infty[$ qui est associé à cette transition dans le modèle analysable. Dans ce cas, l'analyse de l'exemple donné figure 2.39 donne la transition t_2 vivante puisque le tir synchrone des transitions n'est plus exigé. Ce sont les conflits qui posent problème et en général, en cas de conflits, il y a souvent des conditions associées aux transitions pour déterminer quelles transitions franchir. Nous risquons donc d'être confronté à ce problème.

Dans le cas général, nous ne pourrons donc plus garantir la vivacité du modèle implémenté grâce à l'analyse formelle. Il pourrait être intéressant de déterminer de manière automatique si le résultat de vivacité sera pertinent ou pas et de donner cette information au concepteur. Cela est envisageable puisque l'ensemble des conflits structurels sont déjà déterminés.

Nous pouvons désormais implémenter et analyser, au sein de la méthodologie HILECOP, un modèle conçu à l'aide de RdP GEISP. Mais pour remplir les besoins formulés au chapitre 1, il faut de plus être capable d'associer des intervalles temporels aux transitions du modèle conçu.

2.3 Gestion des intervalles temporels

2.3.1 Problématique de l'ajout du temps

Pour satisfaire les contraintes de notre contexte, nous avons besoin de pouvoir modéliser des contraintes temporelles et donc d'utiliser des RdP généralisés étendus interprétés T-temporel synchrone à priorités (RdP GEITSP) (cf. chapitre 1). Il nous faut donc ajouter la possibilité d'associer un intervalle de temps aux transitions dans les RdP GEISP déjà étudiés pour obtenir une méthodologie fonctionnant pour les RdP GEITSP. L'ajout de transitions temporelles nous oblige notamment à considérer deux nouvelles situations au niveau de la correspondance modèle/implémentation/analyse : le risque de blocage des transitions et la gestion des compteurs par rapport aux marquages transitoires.

Risque de blocage d'une transition temporelle avec conditions

Deux sémantiques sont classiquement utilisées en ce qui concerne le tir des transitions dans les réseaux de Petri T-temporels : la sémantique forte et la sémantique faible [18]. La sémantique forte définit que si une transition est franchissable elle est obligatoirement franchie avant que la borne supérieure de l'intervalle temporel de la transition ne soit dépassée. La sémantique faible définit qu'une transition franchissable n'est pas obligatoirement franchie même si son horloge a atteint sa valeur maximale. Si jamais la borne supérieure de l'intervalle de temps est dépassée, la transition ne peut plus être franchie jusqu'à ce qu'elle soit de nouveau sensibilisée (i.e. elle doit être désensibilisée, par le franchissement d'autres transitions par exemple, et alors le compteur est remis à zéro et donc la transition pourra à nouveau devenir sensibilisée). En sémantique faible il est donc possible de désensibiliser une transition par le temps ce qui est impossible en sémantique forte. Par ailleurs, contrairement à la sémantique forte, en sémantique faible, les valeurs temporelles ne modifient pas les résultats d'accessibilité du réseau de Petri non temporel (cf. figure 2.40). En effet, il est possible de réserver des jetons pour pouvoir franchir une transition ou une séquence de transitions.

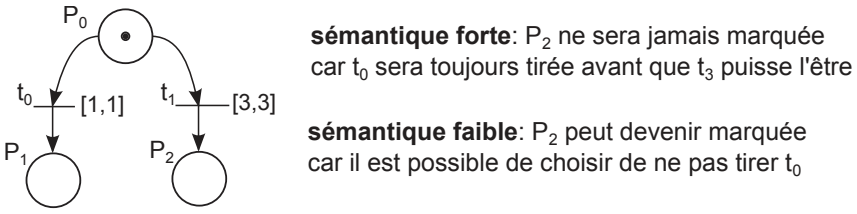

FIGURE 2.40 – Impact de la sémantique sur l'accessibilité des places d'un RdP

La sémantique forte est, malgré cela, la plus utilisée dans la description de systèmes temps-réel car elle permet de modéliser l'urgence de certains événements. Le mécanisme de chien de garde par exemple ne peut être modélisé qu'en sémantique forte. C'est aussi la sémantique utilisée dans les logiciels d'analyse de RdP. Nous souhaitons donc utiliser la sémantique forte. De plus, cela reste cohérent avec le comportement du système réel dans lequel une transition tirable est nécessairement tirée et notre besoin de déterminisme.

Mais la sémantique forte définit que si la limite de l'intervalle de temps d'une transition est atteinte, cette transition est obligatoirement tirée. Or, dans le cadre d'une transition

temporelle à laquelle est associée une condition, le compteur de la transition s'incrémente dès qu'elle est sensibilisée indépendamment du fait que la condition associée soit vraie ou non. Il n'y a donc aucun moyen de garantir que la transition soit tirable quand la limite de l'intervalle de temps est atteinte et donc puisse être tirée. En effet, la condition peut être fausse à ce moment-là et surtout peut rester fausse durant tout l'intervalle de temps et donc la transition n'a jamais eu l'occasion d'être tirée. Par exemple, si dans le RdP donné figure 2.41, la condition c_2 est fausse pendant toute la durée de l'intervalle $[a, b]$, la transition t_2 ne peut pas être tirée avant que la borne b soit atteinte.

La solution envisagée est alors d'adapter la sémantique forte : s'il est possible de tirer la transition dans l'intervalle de temps défini elle doit obligatoirement être tirée sinon la transition ne peut plus être tirée tant qu'elle n'a pas été de nouveau sensibilisée (comme dans le cas de la sémantique faible). Dans notre exemple de RdP figure 2.41, si jamais la transition t_2 est bloquée, le franchissement de t_1 peut permettre de vider P_1 et ainsi de débloquer t_2. Si le concepteur n'a pas prévu d'alternative (si la transition t_1 n'existait pas), la transition t_2 est alors indéfiniment bloquée. A noter que cette sémantique étant différente de la sémantique forte utilisée dans les outils d'analyse, il faudra modifier le modèle analysable pour prendre en compte explicitement ce blocage possible. De même, il faudra s'assurer que le tir de la transition sera bien bloqué dans l'implémentation (cf. §2.3.3).

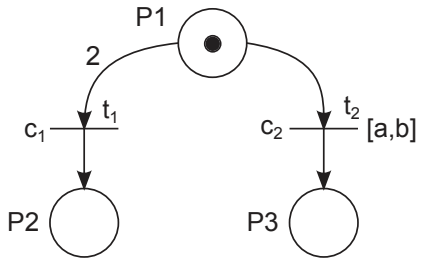

FIGURE 2.41 – Exemple de réseau de Petri GEITSP

Gestion des compteurs

Dans le cas du RdP présenté figure 2.42, quand la transition t_0 est tirée le compteur de la transition t_1 est classiquement réinitialisé même si t_1 reste sensibilisée après le tir. C'est notamment le comportement considéré dans les outils de simulation et d'analyse des RdP T-temporels. La sémantique et l'implémentation des GEITSP respecteront donc la règle suivante : si le marquage d'une place passe de manière transitoire à 0 à cause du nombre de jetons retirés lors du tir d'une transition, les compteurs des transitions aval de la place sont toujours réinitialisés.

Le principe pour pouvoir réaliser cette réinitialisation lors de l'implémentation est de comparer la différence entre le marquage actuel et le nombre de jetons retirés au poids de l'arc sensibilisant la transition temporelle. Par exemple, dans le cas du RdP donné figure 2.42, s'il y a tir de t_0, le marquage était de 1 et le nombre de jetons retiré est 1. La soustraction donne 0 qui est inférieur au poids de l'arc entre P_0 et t_1 (qui est égal à 1) donc le compteur de t_1 doit être réinitialisé. Le détail du principe d'implémentation sera donné §2.3.3. Pour information, le transitoire n'est par contre toujours pas pris en compte

sur les actions comme dans le cas des RdP GEIS, c'est-à-dire que même si le marquage d'une place passe de manière transitoire à 0, le signal activant les actions associées à la place sera maintenu à 1.

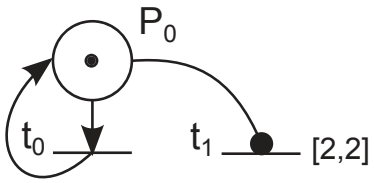

FIGURE 2.42 – Exemple de RdP où le marquage transitoire influe sur le compteur

Dans le cadre d'une implémentation synchrone, une autre situation à considérer est la concurrence entre deux transitions comme dans le cas du RdP présenté figure 2.43. Dans cette configuration, t_0 et t_1 sont simultanément tirables mais ne sont pas en conflit, il est dit qu'elles sont en concurrence. Elles seront donc simultanément tirées, le marquage de la place P_1 sera toujours égal à 1 après le tir des deux transitions (cf. figure 2.43). Mais, lors de l'analyse, l'évolution sera asynchrone, il y aura donc soit tir de t_0 puis tir de t_1 soit l'inverse. Quand t_0 est tirée en premier, une réinitialisation du compteur de t_3 est effectuée tandis que quand t_1 est tirée en premier c'est le compteur de $t2$ qui est réinitialisé. Il n'est jamais possible d'observer, en asynchrone, la réinitialisation des 2 compteurs ou aucune réinitialisation.

Pour pouvoir avoir une évolution synchrone qui sera analysable à l'aide d'outils considérant que les RdP évoluent de manière asynchrone, il est donc nécessaire d'adopter une sémantique telle qu'un seul des compteurs soit réinitialisé. Pour avoir un comportement déterministe et prévisible pour le concepteur, il faut que ce soit toujours le même compteur qui soit réinitialisé. Supposons que nous décidions que dans ce cas, c'est le compteur de t_3 qui est réinitialisé car nous souhaitons donner priorité à l'activation. En effet, dans les modèles synchrones il est généralement défini qu'un état simultanément activé et désactivé reste actif [23].

Mais si nous considérons de nouveau le RdP présenté figure 2.42, il n'y aurait plus réinitialisation du compteur puisqu'il y a désormais priorité à l'activation. Cela n'est pas cohérent dans ce cas puisqu'un jeton ne peut pas être ajouté avant qu'il ne soit retiré. De plus, c'est un comportement qui ne sera jamais observable lors de l'analyse. Ainsi pour rester cohérent avec le cas présenté figure 2.42, le choix est fait de donner, dans tous les cas, priorité aux retraits des jetons plutôt qu'à leurs ajouts. Ainsi dans le RdP de la figure 2.43, c'est toujours le compteur de t_2 qui sera réinitialisé. L'évolution donnée figure 2.43 (b) sera alors observée sur le système réel.

D'un point de vue analyse, nous retrouverons nécessairement le même problème que pour les RdP GEIS, c'est-à-dire que le marquage maximum des places risque d'être surévalué car les deux chemins possibles seront étudiés (cf. figure 2.43 (c) et (d)) mais nous pouvons toujours garantir que le comportement implémenté sera bien analysé. Dans notre exemple de la figure 2.43, l'évolution (d) correspond bien à l'évolution synchrone (b).

L'évolution désirée pour les RdP GEITSP ayant été étudiée et définie, il est désormais possible de les définir et de définir leur sémantique.

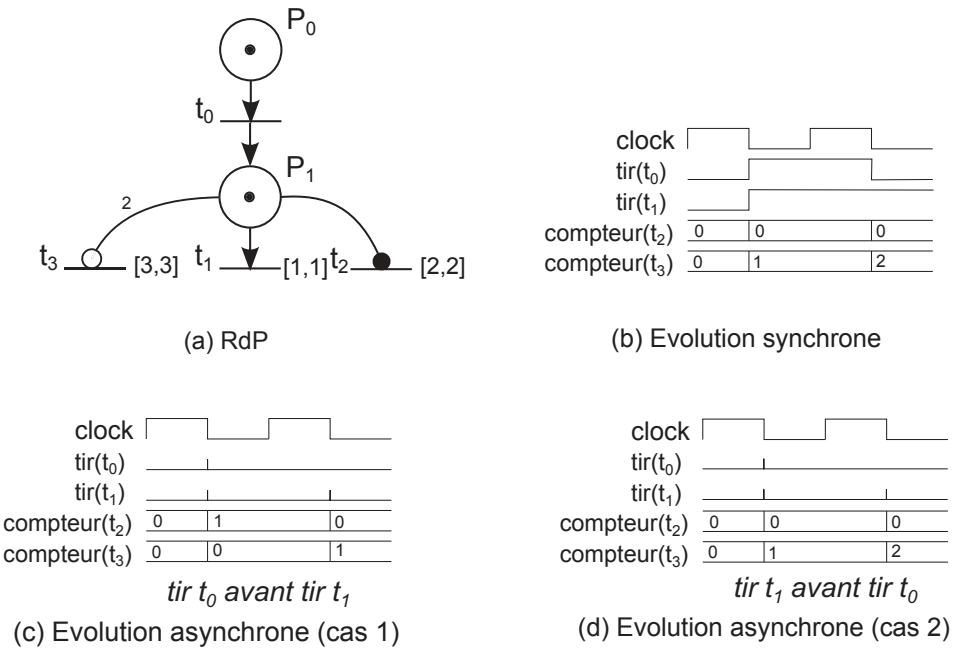

FIGURE 2.43 – Evolution d'un RdP en asynchrone ou en synchrone

2.3.2 Définition et règles sémantiques

Le formalisme des RdP GEITSP étend celui des RdP GEIS donné §2.1.1 avec du temps et des priorités. Les principales problématiques apportées par l'ajout du temps d'un point de vue évolution ont été explicitées dans le §2.3.1. Nous pouvons donc définir de manière formelle les RdP GEITSP et leur sémantique.

Definition 2.3.1. *Soit $\mathcal{C}$ l'ensemble des conditions. Soit $\mathcal{F}$ l'ensemble des actions impulsives. Soit $\mathcal{A}$ l'ensemble des actions continues. Un RdP généralisé étendu interprété T-temporel synchrone à priorités (RdP GEITSP) est un uplet* $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, clock, Is, \succ >$, *où :*

- $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, clock >$ *est un réseau de Petri GEIS.*
- $I_s : T \rightarrow \mathbb{I}^+$ *est la fonction intervalle de temps statique.*
- $\succ : T \times T \rightarrow \mathbb{B}$ *est la relation de priorité supposée irréflexive, asymétrique et transitive.*

Le marquage du RdP est toujours défini par la fonction $m : P \rightarrow \mathbb{N}^+$ et une transition t est toujours dite sensibilisée par m, noté $t \in sens(m)$ si et seulement si $(m \geq Pre(t) + Pre_t(t)) \wedge (m < Pre_i(t))$.

Une transition $k \in T$ est dite nouvellement sensibilisée par le tir d'un groupe de transitions $T_{tir} \subset T$ depuis le marquage m, noté $k \in\uparrow sens(m, T_{tir})$, si et seulement si $k \notin T_{tir}$ est sensibilisée par le nouveau marquage obtenu m' mais ne l'était pas par $m - \sum_{t \in T_{tir}} Pre(t)$ ou si $k \in T_{tir}$ et k est toujours sensibilisée par le nouveau marquage m'.

Formellement on a donc :

$$k \in \uparrow sens(m, Tir) \Leftrightarrow \left(m - \sum_{t \in T_{tir}} Pre(t) + \sum_{t \in T_{tir}} Post(t) \geq Pre(k) + Pre_t(k)\right)$$
$$\wedge \left(m - \sum_{t \in T_{tir}} Pre(t) + \sum_{t \in T_{tir}} Post(t) < Pre_i(k)\right)$$
$$\wedge \left[(k \in T_{tir}) \vee \left(Pre_i(k) \leq m - \sum_{t \in T_{tir}} Pre(t)\right) \vee \left(m - \sum_{t \in T_{tir}} Pre(t) < Pre(k) + Pre_t(k)\right)\right]$$

La valeur instantanée d'une condition est toujours définie par la fonction val, la valeur fixée des conditions pour le calcul de l'évolution par la fonction $cond$ et l'exécution d'une action ou d'une fonction par la fonction ex comme pour les RdP GEIS (cf. §2.1.2).

$I : T \to I^+$ est la fonction, appelée fonction des intervalles de temps, qui associe un intervalle de temps à chaque transition sensibilisée par m comme défini pour les RdP GET (cf. §2.1.4).

$reset_t : T \to \mathbb{B}$ est la fonction de réinitialisation des compteurs. Elle permet de gérer la réinitialisation des intervalles de temps due aux marquages transitoires.

L'état d'un RdP GEITSP est défini par $s = (m, cond, ex, I, reset_t)$.

Une transition est tirable depuis l'état $s = (m, cond, ex, I, reset_t)$ du RdP, notée $t \in tirable(s)$ si et seulement si la transition est sensibilisée par le marquage m, que la valeur des conditions autorise le tir et que la borne minimale de l'intervalle a été atteinte :
$t \in tirable(s) \Leftrightarrow t \in sens(m) \wedge \left(\forall c \in \mathcal{C} | C(t)(c) = 1, val(c) = 1 \wedge \forall c \in \mathcal{C} | C(t)(c) = -1, val(c) = 0\right) \wedge 0 \in I(t)$.

Definition 2.3.2. *La sémantique d'un RdP GEITSP $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, clock, Is, \succ >$ est le système de transition temporisé $< S, s_0, \rightsquigarrow >$ où :*

- *S est l'ensemble des états $(m, cond, ex, I, reset_t)$ du RdP.*
- *$s_0 = (m_0, 0, 0, I_0, 0)$ est l'état initial où 0 est la fonction nulle et I_0 est la restriction de la fonction Is aux transitions sensibilisées par m_0.*
- *$\rightsquigarrow \subseteq S \times Clk \times S$ est la relation de transition d'état, notée $s \overset{clk}{\rightsquigarrow} s'$ avec $clk \in Clk$, définie comme suit : Soit $Tir(s) \subseteq T$ l'ensemble des transitions tirées depuis l'état s. A l'état initial, on a $Tir(s_0) = \varnothing$.*
  - *$s = (m, cond, ex, I, reset_t) \overset{\downarrow clock}{\rightsquigarrow} s' = (m, cond, ex', I', reset_t)$ si et seulement si $\downarrow clock = 1$ et :*
    1. *$\forall c \in \mathcal{C}, cond'(c) = val(c)$ (actualisation des valeurs des conditions)*
    2. *$\forall t \in sens(m), reset_t(t) = 1 \vee t \in \uparrow sens(m, Tir(s)) \Rightarrow I'(t) = Is(t) - 1$ (si l'intervalle de temps d'une transition doit être réinitialisé ou si une transition est nouvellement sensibilisée, l'intervalle de temps de cette transition est réinitialisé à 1 puisque c'est le précédent tir de transitions qui entraîne cette réinitialisation)*
    3. *$\forall t \in sens(m), reset_t(t) = 0 \wedge \left(t \notin \uparrow sens(m, Tir(s)) \wedge I(t) \neq \varnothing\right) \Rightarrow I'(t) = I(t) - 1$ (si la transition est sensibilisée, non bloquée et que son intervalle de*

temps ne doit pas être réinitialisé, son intervalle de temps évolue de manière classique)

4. $\forall t \in T | t \notin \uparrow sens(m, Tir(s)) \wedge I(t) = \varnothing, I'(t) = I(t)$ *(si la transition était bloquée et n'est pas nouvellement sensibilisée, elle reste bloquée)*
5. $\forall a \in \mathcal{A}, \exists p \in P | A(p)(a) = 1 \wedge m(p) \neq 0 \Rightarrow ex'(a) = 1$ *sinon* $ex'(a) = 0$ *(les valeurs de la fonction ex pour les actions sont mises à jour)*

Il est alors possible de déterminer $Tir(s')$ *l'ensemble des transitions qui seront tirées.*

6. $\forall t \in tirable(s'), \forall t' | t' \succ t, t' \notin tirable(s') \Rightarrow t \in Tir(s')$ *(si une transition est tirable et qu'aucune transition plus prioritaire ne l'est alors la transition sera tirée)*
   $\forall t \in tirable(s')$, *soit* $Pr(t)$ *l'ensemble des transitions* t_i *telles que* $t_i \succ t \wedge t_i \in Tir(s')$.
7. $\forall t \in tirable(s'), \Big(t \in sens\big(m - \sum_{t_i \in Pr(t)} Pre(t_i)\big) \wedge t \in sens\big(m - \sum_{t_i \in Pr(t)} Pre(t_i) + \sum_{t_i \in Pr(t)} Post(t_i)\big)\Big) \Rightarrow t \in Tir(s')$ *(une transition tirable est tirée si le marquage est suffisant pour tirer toutes les transitions tirables plus prioritaires et cette transition)*
8. $\forall t \in tirable(s'), \Big(t \notin sens\big(m - \sum_{t_i \in Pr(t)} Pre(t_i)\big) \vee t \notin sens\big(m - \sum_{t_i \in Pr(t)} Pre(t_i) + \sum_{t_i \in Pr(t)} Post(t_i)\big)\Big) \Rightarrow t \notin Tir(s')$ *(une transition tirable n'est pas tirée si le marquage n'est pas suffisant pour tirer toutes les transitions tirables plus prioritaires et cette transition)*
9. $\forall t \notin tirable(s) \Rightarrow t \notin Tir(s')$ *(si la transition n'est pas tirable alors elle ne sera pas tirée)*

– $s = (m, val, ex, I, reset_t) \overset{\uparrow clock}{\rightsquigarrow} s' = (m', val, ex', I', reset'_t)$ *si et seulement si* $\uparrow clock = 1$ *et :*

1. $m' = m - \sum_{t \in Tir(s)} Pre(t) + \sum_{t \in Tir(s)} Post(t)$ *(actualisation du marquage suivant les transitions devant être tirées)*
2. $\forall t \in T, \exists p \in P | m(p) - \sum_{t_i \in Tir(s)} Pre(t_i)(p) < \Big(Pre(t)(p) + Pre_t(t)(p)\Big) \Rightarrow reset'_t(t) = 1$ *sinon* $reset'_t(t) = 0$ *(pour chaque transition si le marquage transitoire d'au moins une place amont est tel qu'il désensibilise cette transition, l'ordre est donné de réinitialiser l'intervalle de temps de la transition)*
3. $\forall f \in (F), \exists t \in Tir(s) | F(t)(f) = 1 \Rightarrow ex'(f) = 1$ *sinon* $ex'(f) = 0$ *(les valeurs de la fonction ex pour les actions impulsives sont mises à jour)*
4. $\forall t \in T, (t \notin Tir(s) \wedge \uparrow I(t) = 0), I'(t) = \varnothing$ *sinon* $I'(t) = I(t)$ *(les transitions non tirées qui ont atteint la valeur maximale autorisée pour leur intervalle de temps sont bloquées, les autres intervalles de temps ne sont pas modifiés)*
5. $Tir(s) = Tir(s')$ *(l'ensemble des transitions tirées n'est pas modifié)*

La sémantique du modèle étant définie, il est alors possible de définir la traduction automatique en VHDL pour avoir un comportement sur la cible correspondant à la sémantique définie.

2.3.3 Transformation en VHDL

Le principe de la traduction automatique en VHDL présenté pour les RdP GEIS (cf. §2.1.3) est conservé. Par contre, cette dernière est bien sûr adaptée pour être capable de gérer les contraintes temporelles. Le composant place est modifié et deux nouveaux composants transitions temporelles sont créés (un pour les intervalles ayant une borne maximum finie et un autre si elle est infinie) (cf. figure 2.44).

Lorsque les instances de composants sont créées, si la transition est non temporelle, nous conservons le composant transition décrit précédemment, si elle est temporelle, nous utilisons un des nouveaux composants. Cela évite de définir inutilement des compteurs. L'ensemble des interfaces des composants utilisés pour la transformation d'un RdP GEITSP en code VHDL sont donnés figure 2.44. Pour les éléments de l'interface non modifiés par rapport au RdP GEIS, le lecteur est invité à se rapporter au §2.1.3 pour plus de détails.

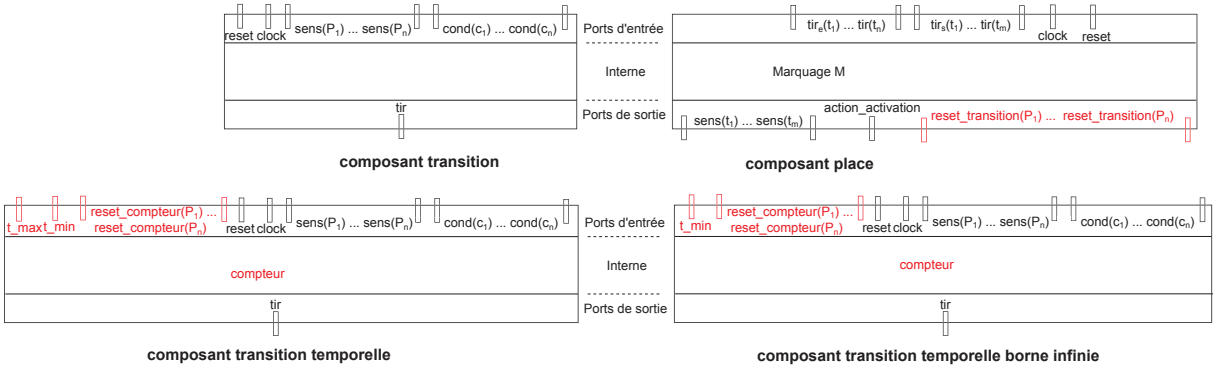

FIGURE 2.44 – Composants VHDL utilisés pour l'implémentation d'un RDP GEITSP (les différences avec ceux utilisés pour les RdP GEIS sont indiqués en rouge)

Le composant place est modifié à cause du problème de la réinitialisation des compteurs en cas de passage temporaire à un marquage inférieur au poids d'un arc test. Pour pouvoir faire la réinitialisation des compteurs, il est nécessaire que la place envoie cette information aux transitions concernées. Un vecteur de signaux binaires $reset\_transition(P_i)$ est alors sorti de chaque place. Nous obtenons alors le composant place dont le code VHDL est donné figure 2.45. Les signaux sont sortis théoriquement pour chaque couple place-transition mais, si certains des signaux sont inutiles parce que les transitions aval de la place sont non temporelles ou parce que la place n'a qu'une seule transition aval, ces signaux ne seront connectés à aucune transition et donc simplifiés lors de la compilation sous Libero.

L'interface du composant transition temporelle avec borne maximum contient, en plus des signaux déjà utilisés dans l'interface du composant transition non temporelle, deux signaux indiquant respectivement le minimum et le maximum de l'intervalle temporel ainsi qu'un vecteur $reset\_compteur$ permettant la réinitialisation du compteur par les places amont. En interne, un compteur est ajouté. Désormais le compteur est incrémenté dès que la transition est sensibilisée et la transition n'est tirée que si le compteur est inclus dans les bornes de l'intervalle. Le signal $reset\_compteur$ venant de la place entraîne la réinitialisation du compteur de même que le tir de la transition. Le code VHDL de ce composant est donné figure 2.46.

```
entity place is

generic(nombre_arc_entrants              : natural := 1;
                nombre_arc_sortants      : natural := 1;
                initial_markup           : natural := 0;
                max_markup                         : natural := mark_max_net);
port(   clk                                        : in std_logic;
                reset_n                            : in std_logic;
                type_arcs_sortants                 : in vect_at(0 to nombre_arc_sortants-1);
                poids_arcs_entrants      : in vect_link(0 to nombre_arc_entrants-1);
                poids_arcs_sortants      : in vect_link(0 to nombre_arc_sortants-1);
                tirs_arcs_entrants                 : in std_logic_vector(0 to nombre_arc_entrants-1);
                output_arc_tir   : in std_logic_vector(0 to nombre_arc_sortants-1);
                sensibilisations_arcs_sortants : out std_logic_vector(0 to nombre_arc_sortants-1);
                reset_timer                        : out std_logic_vector(0 to nombre_arc_sortants-1);
                action_activation        : out std_logic);
end;

architecture a_place of place is
signal marquage_interne : natural range 0 to max_markup;
begin
process(clk, reset_n)
        variable marquage_local : natural range 0 to max_markup;
        variable somme_ajout : natural range 0 to max_markup;
        variable somme_retrait : natural range 0 to max_markup;
begin
        if (reset_n = '0') then
                marquage_interne <= initial_markup; -- initialisation du marquage
                sensibilisations_arcs_sortants <= (others => '0');
                action_activation <= '0';
                reset_timer <= (others => '0');
        elsif (clk'event and clk='1') then
                marquage_local := marquage_interne;
                somme_ajout := 0;
                somme_retrait := 0;
                for i in 0 to nombre_arc_entrants-1 loop -- ajout jeton
                        if (tirs_arcs_entrants(i) = '1') then
                                somme_ajout := somme_ajout + poids_arcs_entrants(i);
                        end if;
                end loop;
                for j in 0 to nombre_arc_sortants-1 loop -- retrait jeton
                        if (output_arc_tir(j) = '1' and type_arcs_sortants(j) = 0) then
                                somme_retrait := somme_retrait + poids_arcs_sortants(j);
                        end if;
                end loop;

                for l in 0 to nombre_arc_sortants-1 loop -- reset timer
                        if ((marquage_interne - somme_retrait < poids_arcs_sortants(l)) and (
                             somme_retrait > 0)) then
                                reset_timer(l) <= '1';
                        else
                                reset_timer(l) <= '0';
                        end if;
                end loop;
                -- actualisation du marquage
                marquage_local := marquage_local + (somme_ajout - somme_retrait);
                -- evaluation de la sensibilisation des transitions sortantes
                for k in 0 to nombre_arc_sortants-1 loop
                        if (type_arcs_sortants(k) = 2) then -- if inhibitor arc P->T
                                if (marquage_local >= poids_arcs_sortants(k)) then
                                        sensibilisations_arcs_sortants(k) <= '0';
                                else
                                        sensibilisations_arcs_sortants(k) <= '1';
                                end if;
                        elsif (marquage_local >= poids_arcs_sortants(k)) then
                                sensibilisations_arcs_sortants(k) <= '1';
                        else
                                sensibilisations_arcs_sortants(k) <= '0';
                        end if;
                end loop;
                -- signal pour l'activation des actions
                if (marquage_local >= 1) then
                        action_activation <= '1';
                else
                        action_activation <= '0';
                end if;
                -- sauvegarde du marquage
                marquage_interne <= marquage_local;
        end if;
end process;
end process;
end a_place;
```

FIGURE 2.45 – Code VHDL du composant place

```
entity transition_time is
generic(time_max                        : natural := 1;
                nombre_arc_entrants             : natural := 1;
                nombre_conditions               : natural := 1);
port(   clk                                     : in std_logic;
                reset_n                         : in std_logic;
                t_min,t_max                     : in natural range 0 to time_max;
                sensibilisation         : in std_logic_vector(0 to nombre_arc_entrants-1);
                condition                       : in std_logic_vector(0 to nombre_conditions-1);
                reset_timer                     : in std_logic_vector(0 to nombre_arc_entrants-1);
                tir                             : out std_logic);
end;

architecture a_transition_time of transition_time is
begin
process(clk, reset_n, sensibilisation, condition, reset_timer)
        variable sens : std_logic;
        variable cond : std_logic;
        variable reset_timer_sum : std_logic;
        variable timer : natural range 0 to time_max + 1;
        variable time_condition : std_logic;
begin
        cond := '1';
        for i in 0 to nombre_conditions-1 loop -- test condition
                cond := cond and condition(i);
        end loop; -- i
        sens := '1';
        for j in 0 to nombre_arc_entrants-1 loop -- test sensibilisation
                sens := sens and sensibilisation(j);
        end loop; -- j
        reset_timer_sum := '0';
        for k in 0 to nombre_arc_entrants-1 loop -- test reset_timer input
                reset_timer_sum := reset_timer_sum or reset_timer(k);
        end loop; -- k
        if (reset_n = '0') then
                timer := 0;
                tir <= '0';
        elsif (clk'event and clk='0') then
                if (reset_timer_sum = '1') then
                        timer := 1;
                elsif (sens = '1') then
                        if (timer <= t_max) then
                                timer := timer + 1; -- incrementation du timer
                        end if;
                else
                        timer := 0; -- reset du timer
                end if;
                -- test de la validite du timer
                if ((timer < t_min) or (timer > t_max)) then
                        time_condition := '0';
                else
                        time_condition := '1';
                end if;
                -- evaluation du tir
                if ((time_condition = '1') and (sens = '1') and (cond = '1')) then
                        tir <= '1';
                        timer := 0 ;
                else
                        tir <= '0';
                end if;
        end if;
end process;
end a_transition_time;
```

FIGURE 2.46 – Code VHDL du composant transition temporelle avec borne maximum

Le composant transition temporelle sans borne maximum est basé sur le même principe que l'autre composant transition temporelle mais est utilisé comme son nom l'indique quand la borne maximale de l'intervalle est infinie. Dans ce cas, seule la borne inférieure de l'intervalle est fournie dans le composant et dès que cette borne est atteinte, le compteur n'est plus incrémenté jusqu'à ce que la transition soit tirée ou désensibilisée. Le code VHDL décrivant son comportement est donné figure 2.47.

Le process pour gérer les conflits n'est pas modifié. Par contre, les transitions temporelles appartenant à un conflit ont besoin de savoir si la transition a effectivement été tirée ou pas pour pouvoir gérer son compteur de temps. Ainsi le signal s_tir des transitions en conflit est toujours envoyé aux places mais il est aussi envoyé aux transitions temporelles. Deux nouveaux composants transitions temporelles doivent alors être définis pour les transitions temporelles en conflit (cf. figure 2.48). Dans le code VHDL, au lieu de faire un OU entre tous les signaux $reset\_compteur$ des places amonts, le OU est réalisé entre le signal s_tir et tous les signaux $reset\_compteur$.

2.3.4 Transformation du modèle global en un modèle analysable

Il est nécessaire de pouvoir construire un RdP GET à partir d'un RdP GEITSP tel que l'évolution du RdP GEITSP est incluse dans celle du RdP GET. En plus de la gestion des intervalles temporels pour traduire l'implémentation synchrone et l'interprétation comme pour les RdP GEIS, il faut aussi illustrer ici le risque de blocage dans le cas d'une transition temporelle à laquelle est associée une condition. Le principe de transformation d'une telle transition pour l'analyse est présenté figure 2.49 et expliquée par la suite.

Soit $< P', T', Pre', Pre'_t, Pre'_i, Post', m'_0, Is' >$ le RdP GET obtenu à partir du RdP GEITSP défini par $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, clock, Is, \succ >$. La transformation consiste d'abord à ajouter les éléments de RdP permettant de modéliser le blocage potentiel des transitions temporelles dont l'intervalle a une borne maximale non infinie et auxquelles est associée au moins une condition.

Soient $P_{blocage}$ et $T_{blocage}$ les ensembles de places et transitions ajoutées au modèle RdP GET pour modéliser le blocage d'une transition. Soit $T_{bloquée} \subseteq T$ l'ensemble des transitions pouvant être bloquées. On a : $\forall t \in T, \exists c \in \mathcal{C} | C(t)(c) \neq 0 \wedge \uparrow Is(t) \neq \infty \Leftrightarrow t \in T_{bloquée}$. Nous supposons que $\forall t \in T_{bloquée}, \uparrow Is(t) \geq 1$. $P_{blocage}$ et $T_{blocage}$ et les arcs associés à ces ensembles sont alors construits comme décrit ci-dessous :

- $\forall t \in T_{bloquée}, p\_blocage\_t \in P_{blocage}$ (l'ajout d'une place est nécessaire pour chaque transition qui risque d'être bloquée)
- $\forall t \in T_{bloquée}, t\_blocage\_t \in T_{blocage}$ et $Is'(t\_blocage\_t) = [\uparrow Is(t), \uparrow Is(t)]$ (l'ajout d'une transition est nécessaire pour chaque transition qui risque d'être bloquée. Cette transition entraînant le blocage de la transition, elle ne peut être tirée qu'une fois la borne maximum de l'intervalle de temps de la transition bloquée atteinte)
- $\forall t \in T_{bloquée}, \forall p \in P | Pre(t)(p) + Pre_t(t)(p) + Pre_i(t)(p) \neq 0, t\_deblocage\_t\_p \in T_{blocage}, Is'(t\_deblocage\_t\_p) = [0, 0]$ et Pre(t_deblocage_t_p)(p_blocage_t)=1(une transition par arc entrant de la transition bloquée doit être ajoutée pour pouvoir modéliser le déblocage de la transition. Ce déblocage consiste à retirer le jeton placé dans la place bloquante de manière instantanée)
- $\forall t \in T_{bloquée}, Pre'_i(t)(p\_blocage\_t) = 1$ (un arc inhibiteur est ajouté entre la place bloquante et la transition bloquée)

```
entity transition_time_inf is
generic(time_max                : natural := 1;
                nombre_arc_entrants     : natural := 1;
                nombre_conditions       : natural := 1);
port(   clk                             : in std_logic;
                reset_n                 : in std_logic;
                t_min               : in natural range 0 to time_max;
                sensibilisation : in std_logic_vector(0 to nombre_arc_entrants-1);
                condition               : in std_logic_vector(0 to nombre_conditions-1);
                reset_timer             : in std_logic_vector(0 to nombre_arc_entrants-1);
                tir                     : out std_logic);
end;
architecture a_transition_time of transition_time_inf is
begin
process(clk, reset_n, sensibilisation, condition, reset_timer)
        variable sens : std_logic;
        variable cond : std_logic;
        variable reset_timer_sum : std_logic;
        variable timer : natural range 0 to time_max + 1;
        variable time_condition : std_logic;
begin
        cond := '1';
        for i in 0 to nombre_conditions-1 loop -- test condition
                cond := cond and condition(i);
        end loop; -- i
        sens := '1';
        for j in 0 to nombre_arc_entrants-1 loop -- test sensibilisation
                sens := sens and sensibilisation(j);
        end loop; -- j
        reset_timer_sum := '0';
        for k in 0 to nombre_arc_entrants-1 loop -- test reset_timer input
                reset_timer_sum := reset_timer_sum or reset_timer(k);
        end loop; -- k
        if (reset_n = '0') then
                timer := 0;
                tir <= '0';
        elsif (clk'event and clk='0') then
                if (reset_timer_sum = '1') then
                        timer := 1;
                elsif (sens = '1') then
                        if (timer < t_min) then
                                timer := timer + 1; -- incrementation du timer
                        end if;
                else
                        timer := 0; -- reset du timer
                end if;
                -- test de la validite du timer
                if ((timer < t_min)) then
                        time_condition := '0';
                else
                        time_condition := '1';
                end if;
                -- evaluation du tir
                if ((time_condition = '1') and (sens = '1') and (cond = '1')) then
                        tir <= '1';
                        timer := 0 ;
                else
                        tir <= '0';
                end if;
        end if;
end process;
end a_transition_time;
```

FIGURE 2.47 – Code VHDL du composant transition temporelle sans borne maximum

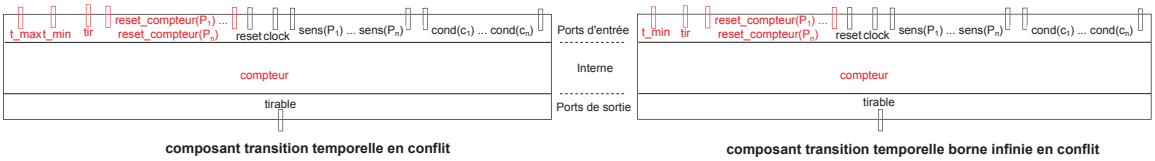

FIGURE 2.48 – Composants transitions temporelles associées à un conflit

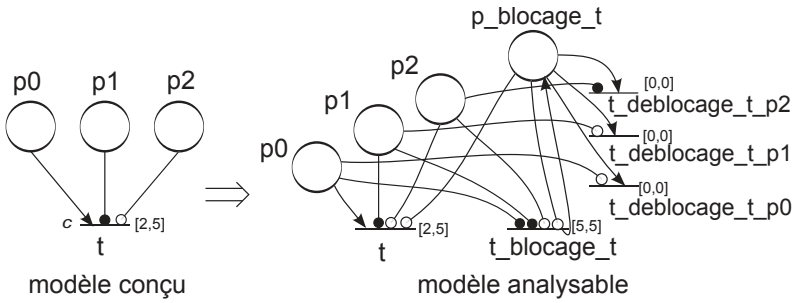

FIGURE 2.49 – Transformation d'une transition qui peut se bloquer

- $\forall t \in T_{bloquée}, Pre'_i(t\_blocage\_t)(p\_blocage\_t) = 1$ (un arc inhibiteur est ajouté entre la place bloquante et la transition bloquante pour éviter de bloquer la transition si elle l'est déjà ce qui n'aurait pas de sens)
- $\forall t \in T_{bloquée}, Post'(t\_blocage\_t)(p\_blocage\_t) = 1$ (quand la transition bloquante est tirée, un jeton est ajouté dans la place bloquante pour bloquer la transition)

Il s'agit ensuite de s'assurer que la transition bloquante a les mêmes conditions de sensibilisation que la transition bloquée mais que la transition bloquante ne puisse pas modifier le marquage du RdP initial.

- $\forall t \in T_{bloquée}, \forall p \in P | Pre(t)(p) \neq 0, Pre'_t(t\_blocage\_t)(p) = Pre(t)(p)$.
- $\forall t \in T_{bloquée}, \forall p \in P | Pre_t(t)(p) \neq 0, Pre'_t(t\_blocage\_t)(p) = Pre_t(t)(p)$
- $\forall t \in T_{bloquée}, \forall p \in P | Pre_i(t)(p) \neq 0, Pre'_i(t\_blocage\_t)(p) = Pre_i(t)(p)$

Il s'agit pour finir de modéliser que la transition sera débloquée dès que l'une des conditions de sensibilisation de la transition bloquée ne sera plus remplie. Le déblocage ne doit bien sûr pas impacter le marquage du RdP initial.

- $\forall t \in T_{bloquée}, \forall p \in P | Pre(t)(p) \neq 0, Pre'_i(t\_deblocage\_t\_p)(p) = Pre(t)(p)$
- $\forall t \in T_{bloquée}, \forall p \in P | Pre_t(t)(p) \neq 0, Pre'_i(t\_deblocage\_t\_p)(p) = Pre_t(t)(p)$
- $\forall t \in T_{bloquée}, \forall p \in P | Pre_i(t)(p) \neq 0, Pre'_t(t\_deblocage\_t\_p)(p) = Pre_i(t)(p)$
- Toutes les valeurs des fonctions Pre, Pre_t, Pre_i et $Post$ non définies ci-dessous et non définies dans le RdP GEITSP sont nulles.

La structure de base du RdP GEITSP est conservée et nous lui ajoutons la gestion du risque de blocage que nous venons de définir, nous obtenons alors :

- $P' = P \cup P_{blocage}$
- $T' = T \cup T_{blocage}$
- $\forall t \in T, \forall p \in P, Pre'(t)(p) = Pre(t)(p) \wedge Pre'_t(t)(p) = Pre_t(t)(p) \wedge Pre'_i(t)(p) = Pre_i(t)(p) \wedge Post'(t)(p) = Post(t)(p)$
- $\forall p \in P, m'_0(p) = m_0(p)$
- $\forall p \in P_{blocage}, m'_0(p) = 0$

Comme expliqué dans §2.2.6, aucune priorité n'est définie sur le RdP GET même si des priorités sont définies sur le RdP GEITSP. Il reste alors à définir Is' pour les transitions appartenant à T. Is' pour les transitions servant à la modélisation du blocage a été défini ci-dessus.

- $\forall t \in T, \downarrow Is'(t) = max(\downarrow Is(t), 1)$ (pour toutes les transitions, la borne inférieure de l'intervalle temporel si elle est supérieure à 1 est conservée sinon elle est définie égale à 1)
- $\forall t \in T, \Big(\forall c \in (C), C(t)(c) = 0\Big) \Rightarrow\uparrow Is'(t) =\downarrow Is(t)$ (si aucune condition n'est associée à une transition, cette transition est nécessairement tirée dès que sa borne inférieure est atteinte)
- $\forall t \in T, \Big(\exists c \in (C) \backslash C(t)(c) \neq 0\Big) \Rightarrow\uparrow Is'(t) =\uparrow Is(t)$ (si une condition est associée à une transition, cette borne maximum est conservée, le blocage ayant été modélisé si besoin par ailleurs)

Nous avons ainsi une solution pour construire un RdP GET à partir d'un RdP GEITSP pour pouvoir analyser ce dernier. Il s'agit maintenant de vérifier, comme pour les RdP GEIS, si les résultats d'analyse obtenus sur ce RdP GET sont fiables par rapport au comportement du RdP GEITSP.

Pertinence des résultats d'analyse

Nous avions vu dans le §2.2.6, que l'inclusion du comportement du RdP GEIS dans celui du modèle analysable était vraie même si le modèle contenait des conflits mais que la vivacité du système réel ne pouvait plus être garantie.

L'introduction d'intervalles temporels dans le modèle conçu nous a amené à modifier la transformation du modèle conçu en modèle analysable, notamment en modélisant le risque de blocage dans le cas de transitions temporelles auxquelles sont associées des conditions.

Il est possible de démontrer formellement que le comportement du RdP GEITSP est inclus dans celui du RdP GET (cf. B) mais cette inclusion n'est évidemment toujours pas une équivalence. Le problème lié au fait que l'analyse est réalisée de manière asynchrone alors que le modèle est synchrone est toujours le même (cf. §2.2.6). La pertinence des résultats d'analyse est donc la même que pour les RdP GEISP :

- si les propriétés d'invariance sont vérifiées sur le modèle analysable, elles le sont sur le système réel.
- si les propriétés d'invariance ne sont pas vérifiées sur le modèle analysable, elles peuvent l'être sur le système réel.
- si les propriétés d'accessibilité sont vérifiées sur le modèle analysable, elles peuvent ne pas l'être sur le système réel.
- si les propriétés d'accessibilité ne sont pas vérifiées sur le modèle analysable, elles ne le sont pas non plus sur le système réel.
- Si le modèle analysable est vivant, le système réel peut ne pas l'être (et réciproquement).
- Si le modèle analysable est borné, le système réel l'est aussi (la réciproque n'est pas vraie).

En conclusion, nous sommes donc désormais capables d'assurer la correspondance entre le modèle conçu à l'aide de RdP GEITSP, le modèle implémentable et le modèle analysable tout en ayant une méthode automatisée pour générer ces deux derniers modèles. Prendre en compte dans le modèle analysable les propriétés non fonctionnelles dues à l'implémentation, nous a permis de connaître précisément les résultats d'analyse formelle que nous pouvions utiliser et ceux qui n'étaient plus utilisables notamment à cause de l'implémentation synchrone. Ainsi il est possible de déterminer de manière fiable si le modèle conçu est borné, si les propriétés d'invariance sont vraies ou si les propriétés d'accessibilité sont fausses. Par contre, les résultats liés à la vivacité ne sont pas fiables et il est possible d'avoir de fausses alarmes ou de faux résultats dans le cas de propriétés d'invariance fausses ou de propriétés d'accessibilité vraies.

Nous avons désormais une méthodologie complète pour gérer le formalisme des RdP utilisé en conservant la conformité entre modèle conçu, modèle implémentable et modèle analysable : gestion de l'interprétation, de l'évolution synchrone, des conflits et des intervalles temporels.

La conception des systèmes numériques complexes est cependant confrontée à un autre problème : la gestion des exceptions. Nous souhaitons donc compléter notre méthodologie et son formalisme pour pouvoir gérer les exceptions de manière pratique et efficace.

Chapitre 3

Gestion des exceptions

Nous avions pour l'instant mis de côté la problématique de la gestion des exceptions. Or les contraintes dans le contexte des systèmes embarqués critiques sont telles qu'il est nécessaire de pouvoir gérer efficacement les exceptions aussi bien d'un point du vue modélisation que d'un point de vue implémentation. Nous commencerons par illustrer ce besoin et proposer une solution au niveau comportemental. Nous nous intéresserons par la suite au niveau architectural.

3.1 Besoin d'une gestion des exceptions efficace et pratique

Lorsqu'un ingénieur conçoit la commande d'un système, il doit non seulement décrire le comportement du système quand il remplit sa tâche principale mais aussi celui attendu dans les situations considérées comme exceptionnelles au sens où elles arrivent moins fréquemment ou sont non souhaitées et non contrôlables (mais observables). La gestion des exceptions ne consiste donc pas uniquement à gérer les réactions lorsqu'une erreur (externe ou interne) se produit durant le fonctionnement du système mais aussi à gérer les situations particulières comme l'extinction du système lorsqu'elle est demandée par l'utilisateur. La description du comportement du système dans les situations exceptionnelles est généralement la partie la plus longue et la plus délicate à concevoir. Il faut en effet s'assurer, d'autant plus dans le contexte des systèmes critiques, que le système ne pourra jamais être dangereux quelle que soit la situation. Or il est souvent difficile de lister de manière exhaustive toutes les situations exceptionnelles pouvant se produire lors de l'utilisation du produit. De même, la réaction du système en cas d'erreur doit être conçue pour garantir la sécurité des utilisateurs. Par exemple, lors de l'utilisation d'un stimulateur dans le domaine médical, les quantités de charge injectées dans un muscle ou dans un nerf peuvent être trop importantes et mettre en danger l'intégrité du muscle ou du nerf. Si un tel cas se produit, il est plus judicieux de réaliser une décharge plutôt que d'arrêter le stimulateur. Autre exemple, dans un autre domaine, lorsqu'un drône volant a un problème en vol, la solution sûre n'est pas de couper tous les moteurs du drône.

La première difficulté pour l'ingénieur dans le cadre de la gestion des exceptions est donc d'abord de décrire les spécifications permettant de garantir que le comportement du système sera toujours sécuritaire. La deuxième est de traduire ces spécifications sur

le modèle décrivant le comportement du système. C'est sur ce deuxième point que nous souhaitons aider le concepteur. Il serait aussi intéressant de travailler sur le premier point mais cela demande de rendre la méthodologie HILECOP plus complète pour qu'elle puisse gérer toutes les étapes du cycle en V (cf. chapitre 1). Commençons par expliquer pourquoi il n'est pas aisé de décrire les réactions en cas d'exceptions à l'aide du modèle HILECOP présenté jusqu'ici.

3.1.1 Nécessité d'un mécanisme de gestion des exceptions au niveau comportemental

Aujourd'hui aucun mécanisme ne permet de gérer simplement les exceptions dans le modèle que ce soit au niveau architectural ou au niveau comportemental. Or l'idéal serait d'avoir une solution pour chacun de ces aspects. Intéressons-nous d'abord au niveau comportemental.

Quand un concepteur souhaite imposer un marquage, comme le marquage initial, il est nécessaire de prendre en compte toutes les situations dans lesquelles le RdP peut se trouver au moment où la réinitialisation doit être réalisée. Cela peut devenir rapidement complexe lorsque le modèle présente un fort parallélisme ou un grand nombre de places. Un principe simple pour forcer un marquage est de vider toutes les places à l'aide d'une transition et de rajouter ensuite les jetons nécessaires. Il serait aussi possible de vérifier que les places qui doivent être marquées ne le sont pas déjà. Mais cela complique d'autant plus la conception puisqu'il faut considérer le cas où elles sont marquées et le cas où elles ne le sont pas. Voyons, sur un exemple simple, les solutions que le concepteur peut utiliser pour modéliser le fait qu'un groupe de places doit être vidé. Nous considérons les méthodes génériques dans le sens où le principe est applicable à n'importe quel modèle. Le RdP donné figure 3.1 sera utilisé comme exemple de référence tout au long de ce chapitre. L'occurence d'une exception est modélisée par le franchissement de la transition t_{exc}. Quand cette transition est tirée, les places P_1 à P_7 doivent être vidées de tous leurs jetons. Une fois les places vidées, un jeton doit être ajouté à la place *init* pour permettre la réinitialisation des places P_1 à P_7.

Supposons pour l'instant que le concepteur souhaite avoir une grande réactivité et donc que les places soient vidées en une période d'horloge. Pour cela une première solution, appelée *tous*, est d'ajouter une transition pour chaque marquage possible du groupe de places P_1 à P_7 (i.e. pour chaque distribution de jetons sur ces places). Mais pour ce cas simple, il existe déjà 59 marquages possibles. Le modèle obtenu est alors non seulement difficile à concevoir sans erreur humaine, mais devient aussi illisible (c'est pourquoi il n'est pas fourni ici). A noter que le concepteur devra, en plus de lister toutes les marquages possibles dans lequel le groupe de places peut se trouver, définir des priorités entre certaines transitions réalisant la purge. En effet, $(P_1, 2P_7)$ et $(2P_1, 2P_7)$ sont deux marquages possibles. Donc si le marquage du réseau est $(2P_1, 2P_7)$, la transition permettant de vider les places dans cette configuration sera sensibilisée, cependant celle permettant de vider la configuration $(P_1, 2P_7)$ le sera aussi. Il y aura donc conflit entre les deux transitions et la priorité ne peut pas être définie arbitrairement. Si ce n'est pas la bonne transition qui est tirée, les places ne seront pas toutes vidées : la place P_1 contiendra, en effet, encore 1 jeton. En outre, pour assurer que la purge sera réalisée en 1 période d'horloge, il faut

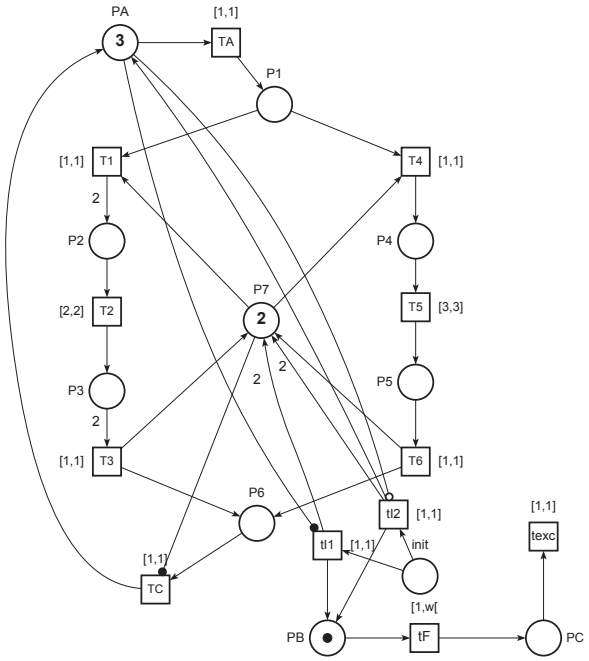

FIGURE 3.1 – Comportement "normal" du RdP : exemple

s'assurer que toutes les transitions liées à la purge sont prioritaires sur les transitions t_1 à t_6 et t_A. En effet, si une des ces transitions est tirée avant une transition de purge, le marquage des places à vider va évoluer. Ce ne sera alors plus la même transition de purge qui sera sensibilisée et il faudra alors attendre une période d'horloge de plus pour que la purge soit réalisée (puisque l'intervalle temporel des transitions de purge est $[1, 1]$). Pour conclure, non seulement cette solution nécessite la connaissance de tous les marquages possibles et présente un fort risque d'erreurs humaines mais l'implémentation de cette solution sera de plus coûteuse en surface puisqu'il y aura de nombreuses transitions et priorités à implémenter.

La contrainte de vider les places en une période d'horloge après le tir de t_{exc} est conservée. Une autre solution, appelée PM, est de considérer tous les marquages possibles de chacune des places indépendamment de celui des autres et de vider chaque place en parallèle. Pour cela, il y a encore deux solutions, soit seuls les marquages vraiment accessibles de la place sont considérés, soit le marquage maximum de chaque place est calculé et tous les marquages entre 0 et le marquage maximum sont pris en compte. La première solution permet de minimiser le nombre de transitions mais nécessite la connaissance a priori des marquages possibles. La deuxième solution ne nécessite que de connaître le marquage maximum des places. Elle permet de minimiser le risque d'erreurs puisque la solution est plus systématique mais elle introduit des transitions potentiellement inutiles. Dans notre cas, les deux solutions sont identiques.

Pour chaque place, une fois son marquage maximum déterminé, une transition est ajoutée pour chacun des marquages possibles. L'ensemble de ces transitions est utilisé pour vider la place concernée. Le modèle obtenu est donné figure 3.2. Il a été dessiné à l'aide du logiciel TINA dans lequel les priorités entre transitions sont représentées par une flèche orange. Il faut de plus s'assurer que les transitions internes et entrantes ne pourront pas être tirées pendant que les places sont purgées (pour éviter que des jetons soient remis

dans des places déjà vidées). De plus, comme pour la solution *tous*, il faudra définir des priorités entre les transitions pour permettre la purge dans le cas où il y a un risque que plusieurs transitions soient sensibilisées pour un même marquage. Si cette solution est réactive, elle n'en demeure pas moins complexe à mettre en place pour un concepteur. Le risque d'erreur est conséquent et évident au vu de la complexité du modèle final obtenu pour le modèle initial très simple donné en exemple (cf. figure 3.2). De plus, si l'analysabilité du modèle est considérée, le fait de vider toutes les places en parallèle aboutira à un graphe des états accessibles plus complexe à cause du phénomène d'entrelacement des transitions (cf. A).

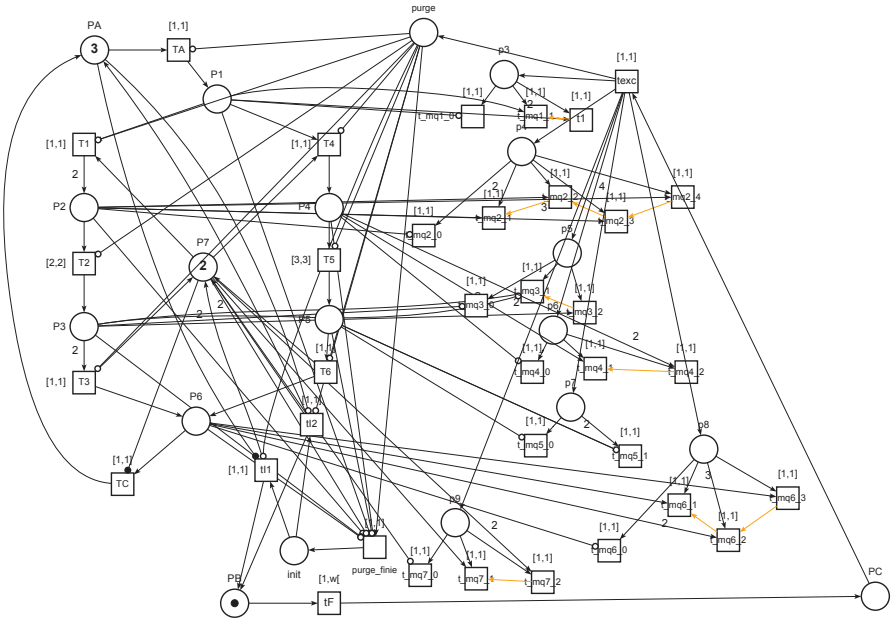

FIGURE 3.2 – Solution PM : purge des places en parallèle en 1 période d'horloge

La contrainte de vider toutes les places en 1 période d'horloge est levée pour voir si, sans cette contrainte, de meilleures solutions peuvent être trouvées. L'un des inconvénients des deux solutions précédemment proposées est qu'elles nécessitent toutes les deux de connaître tous les marquages possibles du modèle (ou au moins le marquage maximum de chaque place). Il serait intéressant de lever cette contrainte car si l'obtention du graphe des marquages accessibles n'est pas possible, garantir que tous les marquages ont bien été pris en compte devient très délicat voire impossible. Pour garder une bonne réactivité, commençons par considérer une autre solution où les places sont vidées en parallèle.

Au lieu de vider systématiquement une place par le tir d'une seule transition, la solution PJ est de vider chaque place jeton par jeton jusqu'à ce qu'elle ne contienne plus de jetons. Le nombre de périodes d'horloge nécessaires pour vider les places sera alors égal au marquage de la place contenant le plus de jetons. Le modèle obtenu est donné figure 3.3. Il est toujours nécessaire de rendre le tir des transitions servant à la purge prioritaire sur celui des transitions normales. Par contre, il n'est plus nécessaire de définir de priorités entre les transitions servant à la purge.

Dans l'exemple choisi, le temps maximum nécessaire pour vider toutes les places est de 4 périodes d'horloge puisque le marquage maximum accessible est 4. Comme pour la

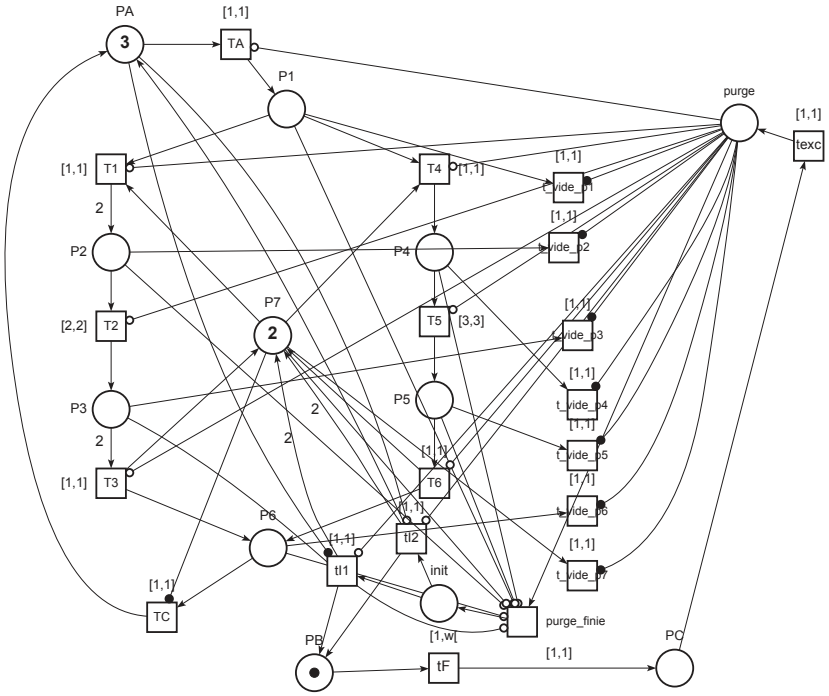

FIGURE 3.3 – Solution PJ : purge des places en parallèle et jeton par jeton

solution en parallèle dépendant des marquages accessibles, l'avantage de cette solution est qu'elle est relativement réactive. Cependant l'analyse du modèle avec cette solution sera plus complexe que dans le cas où les places étaient vidées séquentiellement. Un autre inconvénient de cette solution est qu'elle est, certes, relativement réactive mais son temps de réaction n'est pas fixe. Cela peut être gênant dans certains systèmes où il est plus important de connaître le temps de réaction de manière précise que d'avoir un temps de réaction le plus court possible.

Puisqu'il est possible de vider les places en parallèle, il est tout aussi envisageable de vider les places en séquence c'est-à-dire les unes après les autres. Cela nuira à la réactivité de la solution mais minimisera le risque d'explosion combinatoire dans le calcul du graphe des marquages accessibles. Comme pour la purge des places en parallèle, il est possible d'utiliser soit une solution, appelée SM, dépendant des marquages accessibles où chaque place est vidée en une période d'horloge (cf. figure 3.4), soit une solution, appelée SJ où chaque place est vidée jeton par jeton (cf. figure 3.4).

Les différentes solutions imaginées sont comparées dans le tableau 3.1. La solution avec tous les marquages possibles est appelée *tous*. La lettre S signifie que les différentes places sont vidées séquentiellement et P en parallèle. Si une place est vidée en une période d'horloge la lettre M est utilisée sinon la lettre J est utilisée. Leurs nombres de places et transitions sont comparés pour avoir une idée de la complexité et de la lisibilité du modèle. Le nombre d'états et de transitions du graphe de classes d'états est donné pour apprécier l'impact des différentes solutions sur la complexité de l'analyse. La réactivité est donnée en périodes d'horloge (p.h.) nécessaires pour vider toutes les places.

Pour pouvoir apprécier la complexité des solutions en termes de conception, il est bon de noter que le modèle initial (i.e. sans solution pour vider les places) contenait seulement

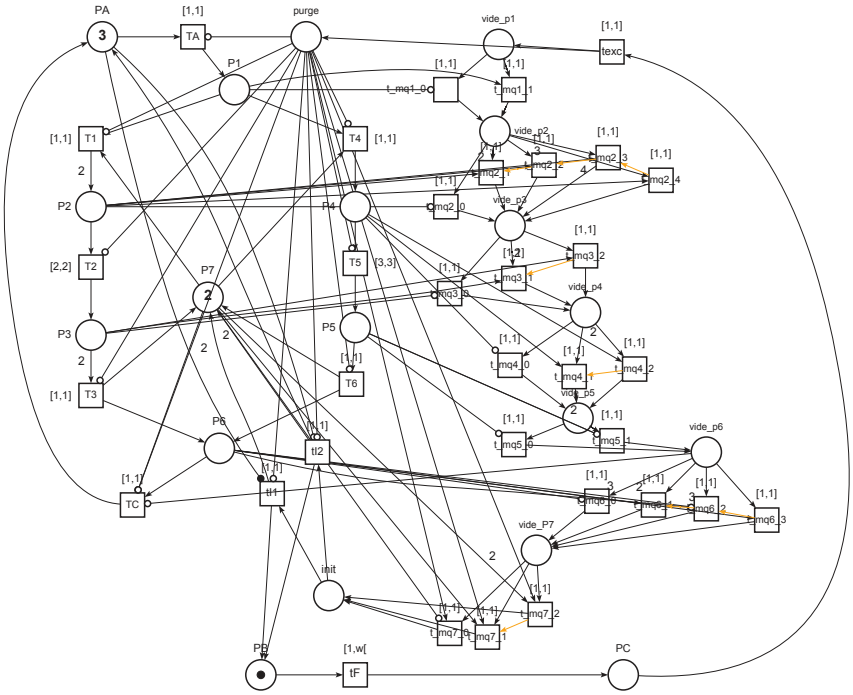

FIGURE 3.4 – Solution SM : purge des places séquentiellement en connaissant les marquages accessibles

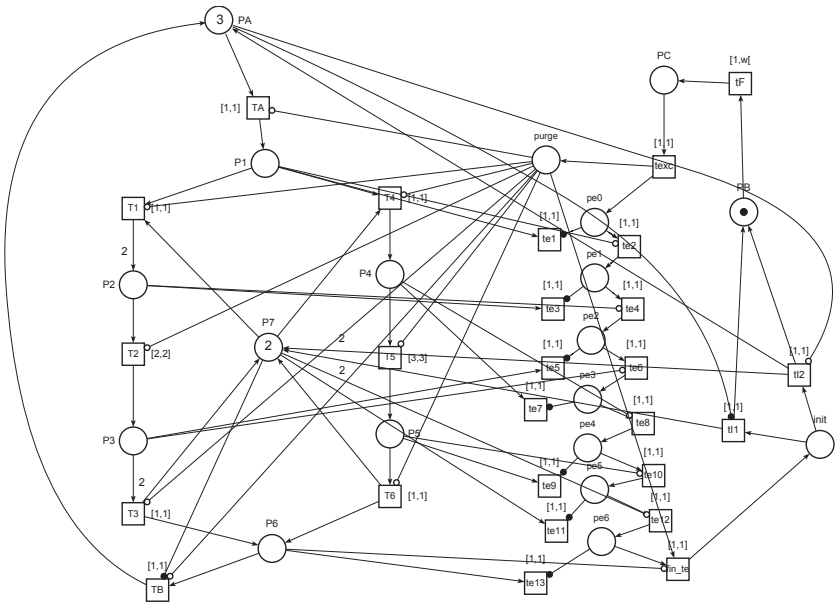

FIGURE 3.5 – Solution SJ : purge des places séquentiellement et jeton par jeton

Solution	Taille RdP		Taille GCE		Réactivité
	places	transitions	états	trans (GC)	(p.h.)
tous	11	72	1 040	2 116	1
PM	18	34	9 489	37 141	1
PJ	12	20	2 287	5 141	de 2 à 5
SM	19	34	2 123	4 279	7
SJ	19	26	2 141	4 275	de 7 à 21

TABLE 3.1 – Comparaison des différentes solutions pour vider les places de l'exemple considéré

11 places et 12 transitions. Il apparaît donc que quelle que soit la solution considérée, le modèle obtenu est beaucoup plus complexe que le modèle initial. La solution *tous* nécessite un modèle particulièrement complexe rendant cette solution inenvisageable dans le cas général bien que ce soit naturellement la meilleure du point de vue complexité de l'analyse. La complexité du modèle entraîne en effet non seulement un risque accru d'erreurs humaines, un manque de lisibilité du modèle et un temps nécessaire pour la conception allongé, mais aussi une surface plus grande pour l'implémentation. En outre, il apparaît que les solutions les plus réactives (solution *tous* mise à part) entraînent un risque plus fort d'explosion combinatoire lors de l'analyse. La différence, d'un point de vue complexité de l'analyse, est faible entre la solution PJ et les solutions séquentielles pour cet exemple. Mais il ne faut pas oublier que l'exemple ici est très simple avec peu de places, cette différence peut être beaucoup plus importante pour des modèles industriels. En outre, la grande différence entre la complexité de l'analyse pour la solution PM et PJ s'explique par le fait que, dans ce modèle, les places contiennent généralement peu de jetons et peu de places sont marquées simultanément. Sur un autre modèle, la solution PJ peut être plus complexe d'un point de vue analyse.

Il est donc nécessaire d'offrir au concepteur un mécanisme pratique permettant de vider un groupe de places. Il doit être d'une part efficace, aussi bien en termes de surface lors de l'implémentation que de réactivité, et d'autre part il ne doit pas nuire à l'analysabilité du modèle. Le modèle doit, de plus, dans la limite du possible, rester lisible et relativement simple à concevoir pour limiter le risque d'erreurs humaines, ce qui est très important dans notre contexte des systèmes embarqués critiques (impact la sûreté et le temps de développement). Pour cela l'idée est de s'appuyer sur le mécanisme de la macroplace, déjà souvent utilisé dans les RdP. Il permettra d'aggréger l'ensemble des places devant être vidées. Il s'agira ensuite de pouvoir vider l'ensemble de ces places.

Ainsi le modèle d'HILECOP décrit jusqu'à présent ne permet pas de gérer de manière pratique et efficace les exceptions au niveau comportemental. Des travaux dans la littérature proposent déjà des pistes de solutions que nous allons étudier au travers du prisme de nos contraintes dans le contexte des systèmes embarqués critiques.

3.1.2 Solutions existantes dans la littérature

Au vu de notre contexte, les contraintes que le mécanisme de gestion des exceptions devra respecter sont :

- Le mécanisme doit pouvoir être décrit graphiquement de façon simple et claire.
- Le mécanisme doit permettre de gérer facilement la purge d'une sous-partie du modèle.
- Le mécanisme doit permettre de gérer le comportement normal comme les exceptions de manière la plus simple possible. Il serait notamment intéressant que la sous-partie du modèle puisse être activée et désactivée dans différentes situations.
- Les conditions pour déclencher une purge peuvent être une (combinaison de) condition(s), un état spécifique du modèle et/ou une contrainte temporelle.
- Le modèle doit être implantable efficacement (en termes de nombre de cellules logiques et de réactivité) sur un FPGA.
- Le modèle doit être analysable avec les outils d'analyse existants.

Il serait souhaitable de pouvoir agréger la sous-partie du RdP qui sera affectée lors de la gestion de l'exception pour conserver un modèle clair et lisible. La solution envisagée est d'utiliser une macroplace, ie une place qui englobe un sous-RdP appelé raffinement. Il faut de plus que la préemption sur la macroplace soit possible pour pouvoir gérer les exceptions. Plusieurs formalismes utilisant la notion de macroplace et/ou la préemption ont déjà été développés dans la littérature.

Nous nous intéressons ici plus particulièrement aux macroplaces utilisables directement pour la conception des modèles. La majeure partie des macroplaces sont utilisées dans ce but. Néanmoins, par exemple, dans la méthodologie de Tzack [52], les macroplaces sont construites à partir du modèle dessiné par le concepteur qui ne contient pas de macroplaces. L'utilité de ces macroplaces n'est alors pas de simplifier la conception mais de minimiser la surface nécessaire à l'implémentation du modèle sur FPGA, elles ne seront pas étudiées plus en détail ici.

Les formalismes non basés RdP

Des formalismes non basés sur les RdP ont été développés comme les Statecharts [34], les SyncCharts [2] et les Grafcharts [37] qui proposent tous des approches intéressantes en termes de gestion des exceptions.

Statecharts : Les Statecharts, déjà présentés au chapitre 1 (§1.3), permettent, notamment, de traduire facilement la notion de hiérarchie. Les états peuvent contenir des états, ils sont alors appelés macroétats. La figure 3.6 présente deux modèles de statecharts ayant le même comportement mais le modèle (i) n'utilise pas de macroétat tandis qu'ils sont utilisés dans le modèle (ii). Cette figure montre qu'il existe deux manières pour entrer dans un macroétat :

- en passant par un arc inter-niveaux (arc e par exemple)
- en utilisant un arc qui cible le macroétat (arc h par exemple). Dans ce cas, l'entrée dans le macroétat entraîne l'activation de l'état par défaut, indiqué par une petite flèche (C dans l'exemple).

Les différentes manières de sortir d'un macroétat sont illustrées sur un second exemple donné figure 3.7. Comme pour entrer, il existe deux manières pour sortir d'un macroétat.

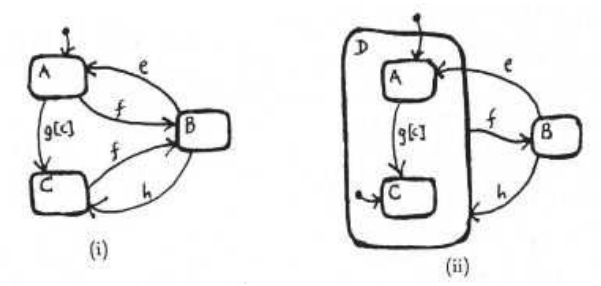

FIGURE 3.6 – Un exemple de Statecharts (i) et son équivalent avec macroétat (i) [34]

- Quand le concepteur souhaite quitter certains états précis du raffinement, il utilise un arc inter-niveaux (arc m ou n).
- Quand le concepteur souhaite quitter tous les états du raffinement quels que soient ses états actifs, il utilise un arc partant du macroétat (arc p). Cet arc permet donc de préempter le raffinement du macroétat.

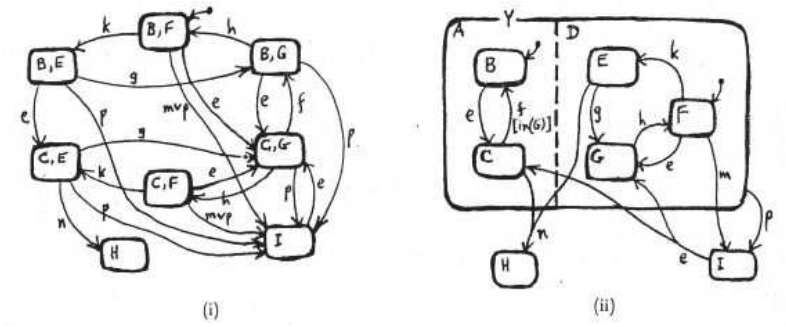

FIGURE 3.7 – Un second exemple de Statecharts (i) et son équivalent avec macroétat (ii) [34]

Vis-à-vis de nos contraintes, il est notamment intéressant qu'il soit possible non seulement d'activer plusieurs états à l'activation d'un macroétat mais aussi de ne pas toujours activer les mêmes états suivant les besoins du concepteur. De même, il est intéressant de pouvoir sortir du macroétat depuis différents états ou de le désactiver complètement. Le gain d'un point de vue lisibilité du modèle offert par les macroétats est déjà observable sur les 2 exemples proposés alors qu'ils sont très simples.

La désactivation du macroétat peut être déclenchée par un événement, soit externe, soit interne. En effet, une action peut être associée à un arc. Cette action peut alors affecter de manière instantanée le comportement des autres composants. Par contre, le modèle n'étant pas temporel, il n'est pas possible de déclencher une désactivation selon une contrainte de temps.

Néanmoins nous préférerions éviter l'utilisation d'arcs inter-niveaux. L'inconvénient des arcs inter-niveaux est que, si le concepteur ne souhaite pas afficher le détail du raffinement des macroétats (pour avoir un modèle plus lisible), il n'est alors plus

possible de distinguer les arcs inter-niveaux des arcs qui ne le sont pas et donc les arcs préemptifs des arcs non-préemptifs.

SyncCharts : Les SyncCharts [2] sont un modèle graphique synchrone. Ils ont été conçus comme une notation graphique du langage Esterel [17]. Les SyncCharts ont donc une sémantique mathématique [3] pleinement compatible avec la sémantique d'Esterel. La syntaxe des SyncCharts a été influencée par les travaux sur les Statecharts, c'est pourquoi il y a une forte ressemblance graphique, mais les comportements décrits sont différents. Un SyncChart peut contenir des états (représentés par des ellipses) et des macroétats (représentés par des rectangles aux coins arrondis) comme sur le modèle donné figure 3.8. Un état ne peut pas être raffiné tandis qu'un macroétat peut contenir des états et/ou des macroétats. Des événements et/ou des actions peuvent être associés aux arcs. Les actions sont précédées du symbole /. Quand un événement est associé à un arc, cet arc ne peut être tiré que si l'événement est vrai. Quand l'arc est tiré, l'action est exécutée, i.e. le signal associé est mis à 1.

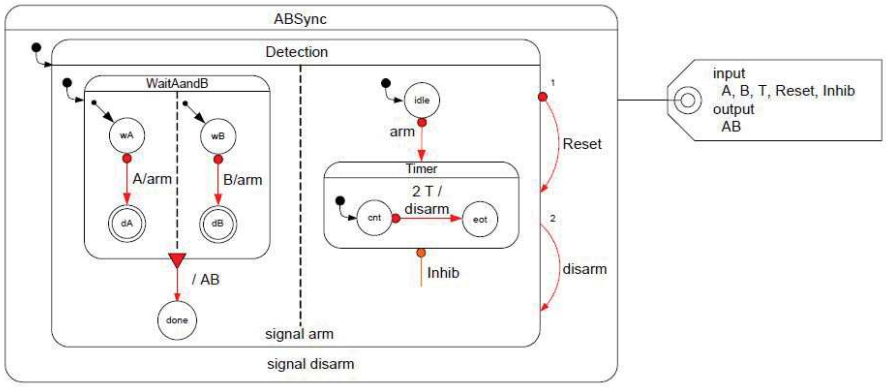

FIGURE 3.8 – Un exemple simple de SyncChart [2]

L'entrée dans un macroétat active l'état initial (ou le macroétat inital), indiqué par une petite flèche. Il n'est pas possible d'entrer dans le macroétat par un autre état (ou macroétat) que l'état (ou macroétat) initial.

Il y a en revanche trois manières de sortir d'un macroétat dans un SyncChart, représentées par trois types d'arcs différents : la préemption forte, la préemption faible et la fin normale. Sur l'exemple de la figure 3.8, l'arc entre l'état wA et l'état dA est une préemption forte, la boucle *disarm* sur l'état *Detection* est une préemption faible et l'arc entre l'état $WaitAandB$ et l'état *done* est une fin normale.

Comme le formalisme des Syncchartes est synchrone, il est nécessaire de définir des priorités entre les arcs. Les préemptions fortes sont les plus prioritaires et les préemptions faibles sont plus prioritaires que les fins normales. Il n'est pas prévu que des priorités soient définies entre arcs de même type.

Quand un macroétat est quitté suite à une préemption forte, aucune évolution interne à ce macroétat n'est autorisée avant d'appliquer la préemption. Par exemple,

figure 3.8, si l'état wA est actif et que les événements A et *Reset* sont vrais, la transition entre wA et dA ne peut pas être tirée car la transition *Reset* est obligatoirement tirée puisque c'est un arc représentant une préemption forte. Le signal *arm* ne sera donc pas émis.

Au contraire, la préemption faible laisse le macroétat évoluer avant de quitter le macroétat. Par exemple, dans la figure 3.8, si l'état wA est actif et les événements A et *disarm* sont vrais, la transition entre wA et dA sera d'abord tirée, émettant donc le signal *arm* puis le macroétat *Detection* sera préempté par la transition *disarm*.

Une fin normale est tirée quand toutes les places finales (représentées par une double ellipse) sont actives. L'arc de fin normale n'augmente pas l'expressivité du modèle car il peut toujours être remplacé par un arc de préemption faible et des signaux locaux, mais cet arc facilite la modélisation pour les concepteurs.

La sémantique des SyncCharts est intéressante pour nous car elle propose une solution, sur un formalisme synchrone, pour gérer tout aussi bien la préemption que la sortie classique sans utiliser d'arcs inter-niveaux. Par contre, les possibilités de modélisation pour entrer dans un macroétat sont limitées puisqu'une entrée dans le macroétat entraîne toujours l'activation des mêmes places. Une préemption (forte ou faible) peut seulement être déclenchée par un signal local ou externe et pas par une contrainte temporelle. Elle ne peut a priori pas être déclenchée automatiquement par le fait que le modèle soit dans une configuration donnée. Il est par contre possible de créer un signal qui indique que le modèle est dans cette configuration et utiliser ce signal pour déclencher une préemption. L'utilisation de ce signal permet une préemption théoriquement instantanée.

Grafcharts : Les Grafcharts sont basés sur la syntaxe graphique du grafcet [36] et ont été développés par Arzen [8] et Johnsson [37][38]. Le principal but des Grafcharts est de transformer le langage Grafcet/SFC, qui est un langage graphique plutôt bas niveau, en un langage graphique haut-niveau orienté objet. Plusieurs classes de Grafcharts existent, seule la version classique sera présentée ici. La syntaxe graphique des éléments de base des Grafcharts est donnée figure 3.9.

Dans les Grafcharts, il existe différents types d'étapes dont trois peuvent contenir un sous-Grafchart : les étapes procédure, les étapes process et les macroétapes. Les étapes process et procédure sont utilisées pour appeler ou démarrer un process ou une procédure donc l'esprit est différent du raffinement que nous souhaitons mettre en place dans notre formalisme. Par contre, les macroétapes sont utilisées pour représenter les étapes ayant une structure interne, ce qui correspond plus à notre approche. Nous ne présenterons donc que les macroétapes.

Les macroétapes ont une étape d'entrée (définie par une flèche en entrée de la place) et une étape de sortie (définie par une flèche en sortie de la place). Le Grafchart du raffinement est contenu dans un sous espace de travail comme présenté figure 3.10. Quand la macroétape est activée, l'étape d'entrée est automatiquement activée. Quand l'étape de sortie est atteinte, la transition de sortie de la macroétape devient sensibilisée. C'est le même fonctionnement que pour les macroétapes dans le Grafcet.

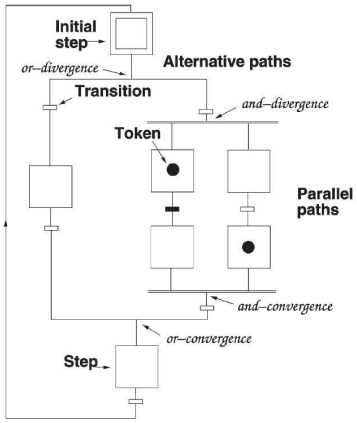

FIGURE 3.9 – Syntaxe graphique du Grafchart [38]

Par contre, la macroétape peut aussi être quittée à l'aide d'un arc exception qui est sensibilisé dès que la macroétape est active, ce qui n'est pas possible dans les Grafcets. Si cette transition exception est tirée, la macroétape et toutes les étapes de son raffinement sont interrompues (c'est-à-dire les étapes actives sont désactivées). Si une action de préemption a été définie, elle est exécutée quand la transition est tirée.

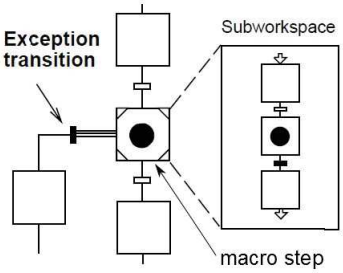

FIGURE 3.10 – Une macroétape avec une transition exception [38]

Une transition exception ne peut avoir en amont qu'une macroétape et il n'est donc pas possible de déclencher une préemption grâce à une autre étape. Le seul moyen de déclencher la préemption est d'associer une condition à la transition exception. Cette condition peut être temporelle.

Les Grafcharts sont synchrones, donc une transition normale et une transition sortie classique peuvent être simultanément tirables. Dans ce cas, l'une des deux transitions est choisie de manière non-déterministe. Ainsi pour avoir un comportement déterministe, il sera nécessaire d'avoir des conditions mutuellement exclusives entre les 2 transitions.

L'inconvénient de la solution proposée pour le Grafchart est que les possibilités pour entrer et sortir d'une macroétape sont assez limitées. En effet, non seulement il est obligatoire de toujours entrer par la même étape mais, en plus, il ne peut y avoir qu'une seule étape d'entrée. De même, il ne peut y avoir qu'une seule étape de sortie

classique. Le principe de modélisation de la préemption par un arc et une transition spécifique est néanmoins intéressant. Elle permet de plus, contrairement aux StateCharts et aux SyncCharts, de définir le déclenchement d'une préemption par une contrainte temporelle.

Si aucun de ces modèles non basés RdP ne satisfait toutes nos contraintes, les idées proposées pour gérer le raffinement et la préemption dans ces modèles permettent d'illustrer comment chacune des contraintes peut être traitée. Étudions maintenant les solutions proposées pour des modèles basés RdP, le formalisme dans lequel notre mécanisme devra s'intégrer.

Les méthodes basées RdP

De nombreuses classes de RdP permettent l'utilisation de macroplace et de préemption. Parmi elles, nous pouvons citer les Place Chart [40], les Petri Chart [35], les PNDS [27] et les HcfgPN [30]. Étudions les solutions apportées par ces classes de RdP.

Place Chart : Les réseaux Place Chart, décrits dans [40] et [39], ont pour but d'apporter aux RdP les points forts des Statecharts. Si ce n'est pas la seule tentative pour coupler les Statecharts aux RdP (les Petri Chart en sont une autre par exemple), de l'avis des auteurs, les réseaux Place Chart fournissent pour la première fois une association concise et cohérente de la préemption et des RdP telle que la hiérarchie du modèle est complètement déterminée par la préemption.

Un réseau Place Chart permet de raffiner les places qui sont alors appelées place chart. Un exemple de réseau Place Chart et le RdP équivalent à ce réseau est donné figure 3.11.

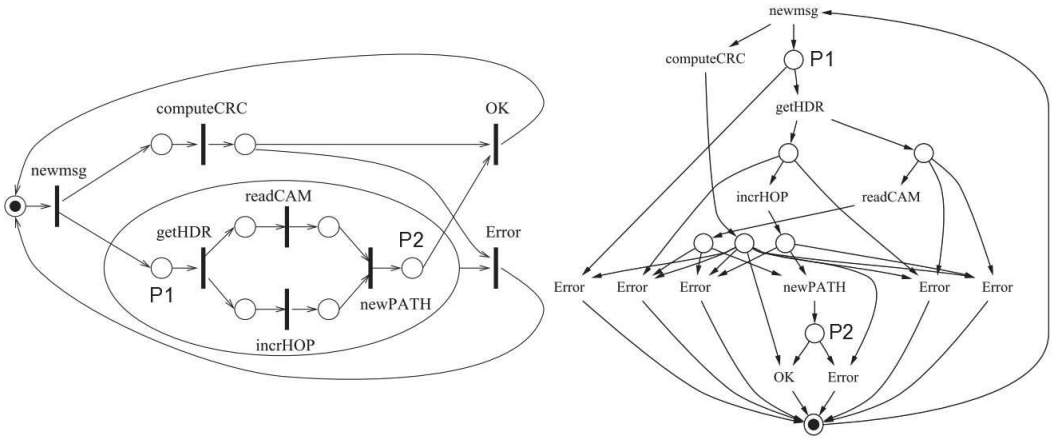

FIGURE 3.11 – Un exemple de réseau Place Chart (gauche) et le RdP au comportement équivalent (les transitions sont remplacées par leur nom) (droite) [40]

Il est possible d'entrer et de sortir d'une place chart grâce à un arc inter-niveau. Par exemple, dans la figure 3.11, l'arc entre *newmsg* et *P1* permet d'entrer dans la place

chart et l'arc entre $P2$ et Ok permet d'en sortir. Dans ce cas, le comportement du modèle sera alors le même que s'il n'y avait pas de place chart. Une autre solution pour entrer dans une place chart est d'utiliser un arc non inter-niveau comme l'arc entre *BeginSearch* et la place chart *SearchDB1* (cf. figure 3.12). Les petites flèches montrent les places qui reçoivent des jetons quand la place chart est activée comme montré figure 3.12.

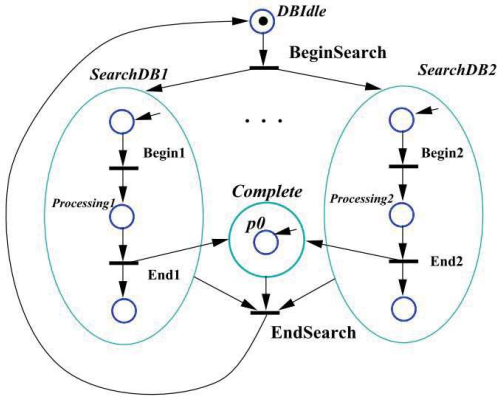

FIGURE 3.12 – Un second exemple de réseau Place Chart utilisant la préemption [40]

La méthode pour gérer la préemption dans les réseaux Place Chart est quasiment la même que pour les Statecharts : des arcs non inter-niveaux sont aussi utilisés pour sortir d'une place chart de manière préemptive (comme l'arc entre *SearchDB1* et *EndSearch*) et des arcs inter-niveaux sont utilisés pour en sortir de manière non préemptive (comme l'arc entre *End1* et *Complete*). Un arc sortant associé à une place chart signifie que la place chart sera préemptée quand la transition associée à cet arc sera tirée. Cette transition est sensibilisée dès qu'au moins une place du raffinement contient au moins un jeton (si le poids de l'arc sortant est de 1). Par exemple, la transition *EndSearch* est sensibilisée par *SearchDB1* dès que l'une des trois places de *SearchDB1* est marquée. En effet, le marquage d'une place chart est défini par le marquage de la place du raffinement contenant le plus de jetons. Ainsi, si le poids de l'arc est de 2, il faut qu'au moins un place contienne 2 jetons ou plus pour que la transition aval soit sensibilisée. Si une préemption est réalisée, par défaut toutes les places sont vidées.

L'avantage est que les transitions préemptives ont la même définition que les transitions normales, c'est seulement la hiérarchie du modèle qui permet de modéliser la préemption. Cela permet d'utiliser une autre place (ou place chart) du réseau comme condition pour déclencher une préemption. Par contre une place du raffinement d'une place chart ne peut pas déclencher la préemption de cette même place chart. Néanmoins, l'utilisation d'une transition intermédiaire est possible. Celle-ci permettra de lancer la préemption en fonction du marquage du raffinement comme montré figure 3.12. Le tir de *End*1 ou *End*2 déclenche le tir de la transition *EndSearch* en utilisant la place chart *complete*. Le modèle n'étant pas temporel, la préemption ne peut pas être déclenchée par une contrainte temporelle.

L'un des avantages du formalisme des réseaux Place Chart est son expressivité. En effet, il permet de décrire différentes solutions pour entrer ou sortir d'une même place chart. La possibilité de réaliser la préemption à l'aide d'un seul arc permet, de plus, de simplifier la complexité du modèle comme montré figure 3.11. Encore une fois le gain en lisibilité offert par l'utilisation d'une macroplace et de la préemption est évident (cf. figure 3.11).

Petri Chart : Les Petri Charts sont présentés dans [35]. Le but des auteurs était d'introduire la hiérarchie dans les RdP en utilisant les principes des statecharts comme pour les réseaux Place Chart. La hiérarchie est introduite en autorisant le raffinement d'une place (ou d'une transition) sans éliminer la place (ou la transition) originale.

Une place (resp. transition) qui est raffinée est alors appelée place chart (resp. transition chart). Un exemple de Petri Chart avec une place chart *cp* et une transition chart *ct* est donné figure 3.13. Le RdP équivalent d'un point de vue comportemental à ce Petri Chart est donné figure 3.14.

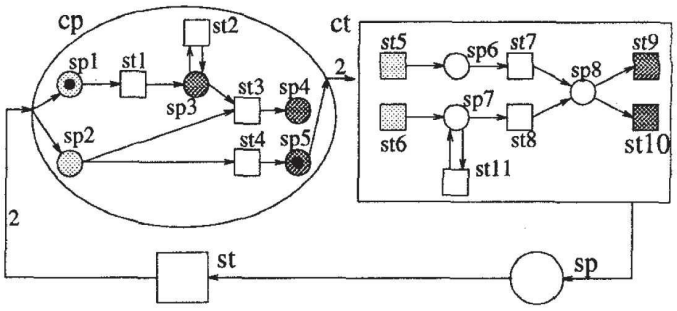

FIGURE 3.13 – Un exemple simple de Petri chart [35]

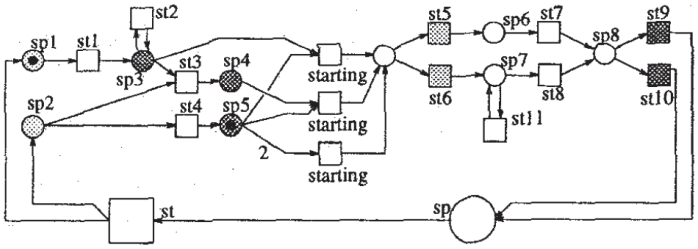

FIGURE 3.14 – Le RdP équivalent au Petri chart donné figure 3.13 [35]

Quand une place chart est activée, les places initiales (places qui sont gris clair) reçoivent des jetons. S'il n'y a pas d'arcs entre la place chart et les places initiales, il s'agit d'un OU entre toutes les places initiales, s'il y en a un c'est un ET. Dans le Petri Chart donné figure 3.13, quand *cp* est activée *sp*1 et *sp*2 reçoivent un jeton (cf. figure 3.14). S'il n'y avait pas eu les deux arcs entrants, seule l'une des places

aurait reçu 2 jetons. Le choix de la place qui reçoit les jetons dans ce cas est réalisé de manière aléatoire.

Les places finales (places qui sont gris foncé) décrivent les jetons qui seront retirés par l'arc sortant. Par défaut, c'est un OU dans le sens où si une transition sortante est tirée et que le poids de l'arc est de 1, un jeton est retiré d'une seule place parmi toutes les places finales marquées. Pour indiquer qu'un jeton doit nécessairement être retiré d'une place finale précise, un arc sortant doit être ajouté entre cette place et l'arc sortant de la place chart. Par exemple, pour sortir de la place chart *cp* de la figure 3.13, il faut au moins 1 jeton dans *sp*5 et 1 autre jeton dans *sp*4 ou dans *sp*5. Le tir de *ct* retirera nécessairement un jeton de *sp*5 et un autre jeton soit de *sp*4 soit de *sp*5 (cf. figure 3.14).

Les transitions charts sont définies dans le même esprit que les places charts. Quand une transition chart est tirée, une transition initiale est tirée (ou plusieurs dans le cadre d'un ET). Si une transition finale (ou plusieurs dans le cadre d'un ET) est tirée, des jetons sont déposés dans la(es) place(s) sortante(s) de la transition chart.

L'avantage de ce formalisme est qu'il est possible de mettre un nombre différent de jetons dans chaque place initiale et que les jetons entrants ou sortants peuvent être décrits à l'aide de OU ou de ET. De plus, le fait que le OU ne nécessite aucun arc permet d'avoir un modèle plus lisible. Le problème est que l'expressivité du modèle est trop limitée par rapport à nos besoins. En effet, il ne peut y avoir qu'un ensemble de places initiales et un ensemble de places finales, et une seule solution pour entrer de la place chart peut être définie tout comme pour en sortir. De plus, la représentation graphique impose qu'une place ne peut pas être initiale et finale. En outre, le modèle devient non déterministe si un OU est utilisé. La question des solutions pour réaliser ce OU dans la pratique se pose également.

PNDS : Selon les auteurs de l'article [27], aucune des classes de RdP utilisées dans la conception de systèmes numériques ne possède toutes les caractéristiques désirées dans un bon langage de modélisation. En effet, le temps pour la communication et le temps pour le calcul n'ont jamais été traités simultanément dans les extensions des RdP. C'est pourquoi ils ont développé une nouvelle classe nommée réseaux de Petri pour les systèmes numériques et plus particulièrement les systèmes informatiques (PNDS : Petri NET for Digital Systems) [27] [26]. Le modèle propose plusieurs types de places et d'arcs, nous ne décrirons ici que ceux entrant dans la gestion des exceptions. Pour la préemption, les PNDS utilisent un arc préemptif. Cet arc est représenté par une ligne en pointillés et a pour source une place et pour destination une transition (cf. figure 3.15). Quand la transition est tirée, toutes les places liées à cette transition par un arc préemptif sont vidées de tous leurs jetons. Cet arc exception peut être lié à deux types de places : les places locales (correspondant aux places classiques des RdP) et les places fonctionnelles (représentée par une triple ellipse). Ces dernières peuvent notamment encapsuler un comportement séquentiel (routine de programme).
Un avantage est que seul l'arc préemptif est différent, donc il est possible d'utiliser d'autres places du modèle comme condition pour déclencher la préemption. De plus, les arcs préemptifs sont facilement différenciables des arcs classiques sans avoir

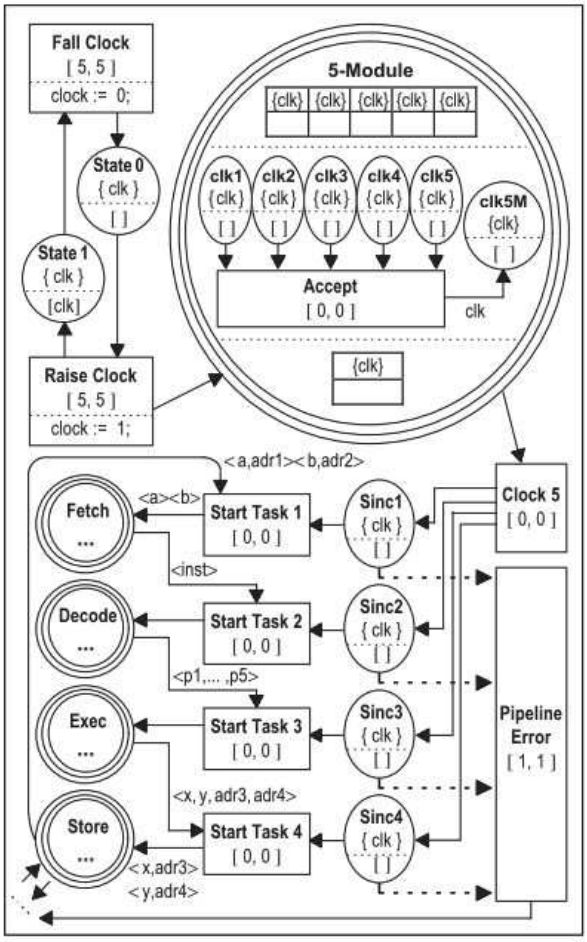

FIGURE 3.15 – Exemple de préemption dans un PNDS

à ajouter de texte ce qui est une bonne chose pour la lisibilité du modèle. De plus, dans les PNDS, la préemption peut être déclenchée par une contrainte temporelle. L'inconvénient des PNDS est que les places fonctionnelles ne peuvent contenir que des comportements séquentiels. Ainsi, pour la description de comportements concurrents, il faudra plusieurs places fonctionnelles qui doivent être chacune vidées par un arc préemptif en cas d'exception ce qui peut amener rapidement à un modèle peu lisible et augmente le risque d'erreur humaine (oubli d'un arc).

Nous ne nous sommes pour l'instant pas intéressés à la manière dont les solutions choisies de macroplace ou de préemption sont implémentées car les auteurs des solutions présentées jusqu'ici ne l'abordent pas dans leurs articles. La méthode de Doligalski [30] propose, elle, une solution pour transformer un modèle hiérarchique avec préemption en code VHDL.

HcfgPN : La transformation d'un RdP non hiérarchique en VHDL avec la méthode de Doligalski a déjà été présentée au chapitre 1. Pour rappel, elle utilise un principe similaire à celui de HILECOP mais en utilisant des process VHDL au lieu de composants. Intéressons-nous ici à la manière dont sont gérées les macroplaces et la préemption.

Dans le formalisme des RdP configurables et hiérachiques (HcfgPN) [29][30][28], le modèle est composé d'un réseau "top level" et de plusieurs sous-réseaux, raffinements du réseau de plus haut niveau. Le raffinement est encapsulé à l'aide d'une macroplace représentée par un double cercle (cf. figure 3.16). Chaque sous-réseau est composé de deux parties : la partie opérationnelle et la partie configuration. Le sous-réseau opérationnel décrit le comportement souhaité du RdP. Il est contrôlé par le sous-réseau de configuration qui gère la gestion des exceptions ou la reprise. La partie configuration est toujours identique quel que soit le sous-réseau. La gestion des exceptions dans cette méthode peut consister, soit en une purge du sous-réseau opérationnel (équivalent d'une réinitialisation du sous-réseau complet), soit à figer le sous-réseau dans son état actuel avant une éventuelle reprise.

Le sous-réseau de configuration consiste en 3 places (cf. figure 3.16) : (P^{init}, P^a, P^i) et 5 transitions : $(T^{init}, T^a, T^i, T^w, T^{fin})$. Quand la place P^a est marquée, le sous-réseau opérationnel associé peut évoluer et émettre des signaux, au contraire des moments où P^i est marquée. Le tir de la transition T^w retire tous les jetons des places du sous-réseau opérationnel et replace le sous-réseau complet dans son marquage initial. Les conditions des transitions T^{init}, T^a, T^i, T^w et T^{fin} sont déterminées en fonction des transitions amont et aval de la macroplace qui contient le sous-réseau comme illustré figure 3.17.

La préemption peut être déclenchée par une condition portant sur des signaux ou par une condition portant sur la situation d'un sous-réseau. Elle ne peut pas être déclenchée par une contrainte temporelle.

Dans le code VHDL, deux process gèrent les places et transitions du sous-réseau opérationnel. Le signal $LocalConfig$ informe les process gérant le sous-réseau opé-

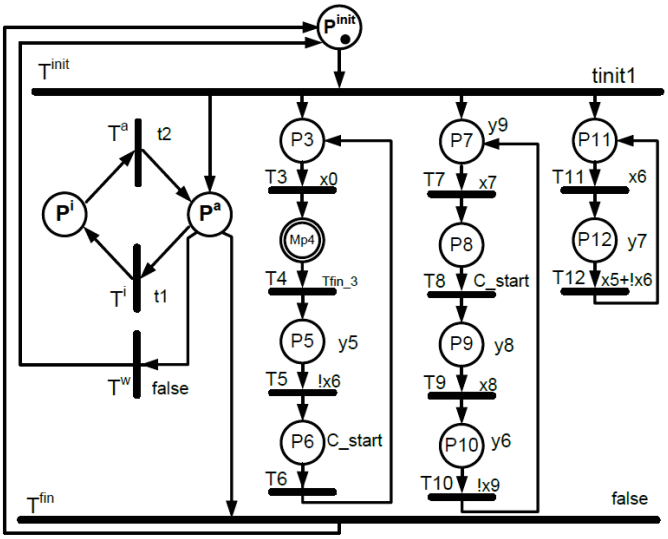

FIGURE 3.16 – Exemple de sous-réseau d'un modèle HcfgPN

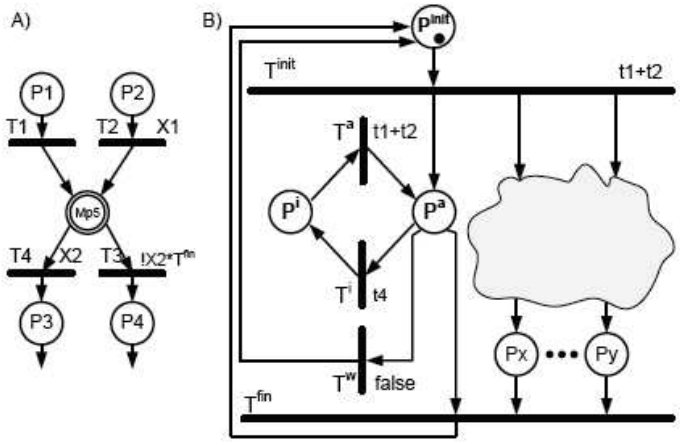

FIGURE 3.17 – Illustration de la communication entre sous-réseaux dans les HcfgPN [30]

rationnel si l'évolution est autorisée ou non et gère aussi la priorité des transitions T^i et T^w sur les autres transitions. Le process *config_ Transitions* décrit toutes les transitions opérationnelles et est responsable de la réinitialisation du marquage dans le cas d'une préemption (tir de T^w) ou du tir de la transition finale. La reprise est gérée par le tir de la transition (T^a). Le marquage des places de configuration est géré par le process *config_ Places*. Les deux process opérationnels et la gestion du signal $LocalConfig$ sont donnés figure 3.18 dans le cas du sous-réseau donné figure 3.16.

D'un point de vue modélisation, la méthode de Doligalski est intéressante car elle permet non seulement la purge d'un sous-résau mais également de figer ce sous-réseau. Mais les RdP gérés dans cette méthode ne sont pas temporels. Le principe de figer un réseau temporel est plus complexe, le problème étant : comment gérer les contraintes temporelles alors que le réseau est figé ? Le temps est-il lui aussi figé ou continue-t-il à évoluer ? Cette méthode ne permet par ailleurs de ne modéliser qu'une situation initiale et qu'une seule transition de fin. De plus, de notre point de vue, il est difficile de distinguer facilement (visuellement) la partie opérationnelle de la partie configuration et il est dommage que la partie configuration soit toujours présente et toujours identique alors que certaines transitions ou places peuvent être inutiles (si le sous-réseau n'est jamais figé par exemple).

Une autre méthode étudiée au chapitre 1 permet l'implémentation en VHDL : la méthode de Silva [47], mais l'implémentation utilise des matrices que nous ne pouvons pas utiliser dans notre contexte. La partie modélisation de la macroplace est très peu détaillée et il ne nous est donc pas possible d'étudier ses apports effectifs par rapport à nos contraintes.

Par ailleurs, il est aussi nécessaire que le mécanisme de gestion d'exceptions n'empêche pas d'analyser le modèle à l'aide des outils d'analyse existants. Aucun des modèles présentés ici n'est directement analysable. Certains des travaux proposent néanmoins de transformer le modèle en RdP analysable équivalent, comme ceux sur les PNDS [26], les Place Chart net[40] ou les Petri Chart [35]. Il s'agit de remettre le modèle à plat.

Conclusion Les différentes possibilités offertes par les formalismes sont synthétisées dans le tableau 3.2 où St représente les Statecharts, Sy les SyncCharts, Gr les GrafCharts, Pl les Place Chart Nets, Pe les Petri Chart et HC les HcfgPN.

Aucune des solutions trouvées dans la littérature ne satisfait toutes nos contraintes mais elles proposent néanmoins des solutions intéressantes pour chacun des critères. Nous nous sommes donc attachés à définir un nouveau mécanisme d'agrégation qui s'inspire des solutions intéressantes trouvées dans les différents formalismes existants.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
Entity Net_2 is
port(CLK, RESET, x0, x5, x6, x7, x8, x9, t1,
    t2, tfin_3, tinit_1 :in  std_logic;
          Ti_out, Ta_out, T3_out, MP4_out, y5,
             y6, y7, y8, y9 : out std_logic) ;
End Net_2_VHD;
Architecture Net_2 of Net_2 is
Signal P3, MP4, P5, ... , P12 : std_logic
    :='0';
Signal T3, ... ,T12: std_logic:='0';
Signal Pinit: std_logic:='1';
Signal Pa, Pi, Tinit, Tfin, Ti, Ta, Tw,
    LocalConfig : std_logic:='0';
Signal C_start :std_logic:='0';
begin
Ti_out <=Ti; --local signals
...
y5              <= Pa and P5 ;--outputs
C_start <= Pa and P6;
...
LocalConfig <= Pa and  not (Ti or Tw);
config_Transitions:process(Pi, Pa, Ti, Tw,
    Pinit, t1,t2, P7, P9, t5, t6, tinit_1)
    begin
    Tinit <= Pinit and tinit_1 ;
    Tfin  <=  Pa and P7 and P9 and not(Ti and
       Tw);
    Ti <= Pa and t1 and  not Tw;
    Tw <= Pa and '0';
    Ta <= Pi and t2;
        end process;

config_places:process (CLK, Reset)
Begin
if RESET='1' then
    Pinit<='1';
    Pi   <='0';
    Pa   <='0';
elsif CLK'event and CLK='1'then
    if Tw='1' or Tfin='1' or (Pinit='1' and
        Tinit='0') then Pinit<='1'; else Pinit
        <='0';end if;
    if Tinit='1' or Ta='1' or (Pa='1' and (Ti
        ='0' and Tw='0' and Tfin='0')) then Pa
        <='1'; else Pa<='0';end if;
    if Ti='1' or (Pi='1' and (Pi='0' or Ta
        ='0')) then Pi<='1'; else Pi<='0';end
        if;
end if;
end process;
operational_transitions:process (P3, MP4, P5,
    ... , P12, x0, ... , x9, t1, t2, tfin_3,
    tinit_1, C_start, tfin_3, LocalConfig)
Begin
...
end process;
operational_places:process (CLK, RESET)
Begin
 ...
end process;
end architecture;
```

FIGURE 3.18 – Extrait du code VHDL décrivant un HcfgPN sur l'exemple donné figure 3.16

Critères	St	Sy	Gr	Pl	Pe	PNDS	HC
plus d'un ensemble de places initiales	✓	✗	✗	✓	✗	✗	✗
plus d'un ensemble de places finales	✓	✓	✗	✓	✗	✗	✗
utilisation d'arcs inter-niveaux	✗	✓	✓	✗	✓	✗	✗
choix entre préemption et sortie	✓	✓	✓	✓	✗	✓	✓
distinction claire entre la préemption et la sortie	✓	✓	✓	✓	✗	✓	✗
préemption déclenchée par un événement interne	✓	✓	✗	✗	✗	✗	✓
préemption déclenchée par un événement externe	✓	✓	✓	✓	✗	✓	✓
préemption déclenchée par une contrainte temporelle	✗	✗	✗	✗	✗	✓	✗
peut être transformé pour être analysé	✗	✗	✗	✓	✓	✓	✓

TABLE 3.2 – Possibilités offertes par les différents modèles étudiés

3.2 Gestion des exceptions au niveau comportemental

Décrivons maintenant comment la macroplace et le principe de préemption sont intégrés au modèle HILECOP, ainsi que leur implémentation et la transformation du modèle avec macroplace pour obtenir un modèle analysable.

3.2.1 Définition de la macroplace

Description générale

Une macroplace (MP) est représentée par un double cercle. Elle contient un réseau de Petri généralisé étendu interprété T-temporel synchrone à priorité (RdP GEITSP) appelé raffinement de la macroplace. Il n'est possible d'entrer ou de sortir du raffinement que grâce à des arcs spécifiques, décrits dans les paragraphes suivants : arcs entrants, arcs sortants classiques et arcs exceptions. Le raffinement ne peut pas contenir de macroplace. Nous appellons marquage d'une macroplace le vecteur contenant le marquage de chacune des places de son raffinement. Un exemple de macroplace est donné figure 3.19.

Une macroplace est considérée comme active si et seulement si son marquage est différent du vecteur nul, i.e. si au moins une de ses places a un marquage non nul. Seul le marquage des places est considéré : le franchissement d'une transition n'entraîne pas une activité de la macroplace. Ainsi ce n'est pas parce qu'une macroplace est inactive qu'il n'y a pas d'activité possible dans son raffinement. En effet, dans le cas d'un unique arc inhibiteur, il peut y avoir franchissement d'une transition même si la macroplace est inactive. Si ce franchissement n'entraîne pas l'apparition de jetons dans le raffinement elle demeurera inactive.

Ce choix de définition a été fait car nous ne voulons pas que l'activité de la macroplace puisse être fugace comme ce serait le cas si le franchissement d'une transition entraînait l'activation de la macroplace. Le concepteur devra donc être attentif au fait que ce n'est pas parce qu'une macroplace a été vidée que des jetons ne peuvent pas apparaître dans son raffinement même sans franchissement d'un arc entrant.

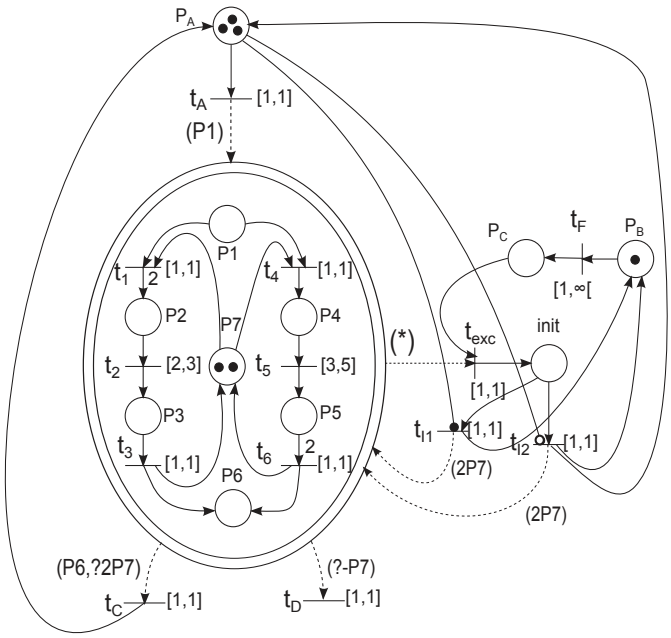

FIGURE 3.19 – Un exemple de macroplace

Arcs entrants d'une macroplace

Pour entrer dans une macroplace un arc entrant différent des arcs classiques est utilisé. Un arc entrant relie nécessairement une transition à une macroplace. Il est représenté par une flèche en pointillé et est caractérisé par une situation s_e appelée situation d'entrée. Une situation d'entrée s_e est l'ensemble non vide des places du raffinement de la MP auxquelles le concepteur souhaite ajouter des jetons lors de l'entrée dans la MP, ainsi que le nombre de jetons à leur ajouter. Le nombre de jetons à ajouter est associé à chaque place, il peut donc différer d'une place à l'autre.

Definition 3.2.1. *Une situation d'entrée s_e est définie telle que $s_e = \{n_i P_i\}$ où $n_i \in \mathbb{N}^+$ et $P_i \in P^{mp}$ avec P^{mp} l'ensemble des places du raffinement de la macroplace mp. Si $n_i = 1$ il peut être omis.*

Lors du tir de la transition amont de l'arc entrant, le nombre de jetons indiqué dans la situation d'entrée s_e est ajouté dans les places concernées. Par exemple dans la figure 3.19, si la transition t_A est tirée, 1 jeton est ajouté dans la place $P1$ et si la transition t_{I1} est tirée, 2 jetons sont ajoutés dans la place $P7$. Une macroplace peut être la cible d'un nombre illimité d'arcs entrants. Par contre, il ne peut pas y avoir deux arcs entrants ayant la même transition amont et la même MP aval. Cette restriction est en cohérence avec la définition des RdP interdisant les arcs multiples de même type ayant la même source et la même destination.

Il convient de bien comprendre qu'un arc entrant d'une MP ajoute seulement des jetons, i.e. il ne s'agit pas du forçage d'un marquage mais de l'ajout de jetons dans certaines places. Pour réaliser un forçage, i.e. imposer un marquage à la MP, il est donc nécessaire de d'abord vider la MP avant d'y entrer avec un arc indiquant la situation dans laquelle la MP doit être "forcée".

Arcs sortants d'une macroplace

Pour sortir d'une MP, un arc spécifique est défini. Il relie nécessairement une MP à une transition. Il est représenté par une flèche en pointillé et est caractérisé par une situation s_s appelée situation de sortie. Deux cas doivent être distingués, les situations de sortie classiques qui indiquent une situation précise, et la situation exception qui indique que le tir de la transition aval entraîne la purge de la MP, quelle que soit sa situation, tant que celle-ci est active. Nous parlerons alors respectivement d'arc sortant classique et d'arc exception. Il peut y avoir autant d'arcs sortants que le concepteur souhaite mais pas deux arcs sortants avec la même MP amont et la même transition aval.

La situation de sortie classique s_s est définie comme étant la description du marquage minimum nécessaire pour sensibiliser la transition aval de la MP.

Definition 3.2.2. *Une situation de sortie classique s_s est définie telle que $s_s = \{Xn_iP_i\}$ où $n_i \in \mathbb{Z}\backslash\{0\}$ et $P_i \in P^{mp}$ avec P^{mp} l'ensemble des places du raffinement de la macroplace et $X \in \{\epsilon, ?\}$.*

Pour modéliser l'effet d'un arc classique, aucun symbole n'est écrit devant le chiffre n_i (i.e. $X = \epsilon$). Pour modéliser celui d'un arc test, on a $X =?$ et $n_i > 0$ et pour modéliser celui d'un arc inhibiteur on a $X =?$ et $n_i < 0$. Le chiffre 1 peut être omis. Ce formalisme est inspiré de celui utilisé dans la description textuelle (format de persistance) d'un RdP dans TINA [62].

Dans le cas d'un arc sortant classique, une transition aval de cet arc n'est sensibilisée que si le marquage de la macroplace contient au moins les jetons indiqués dans la situation de sortie. Si la transition est franchie, les jetons indiqués dans la situation sont retirés de leurs places respectives. Les places non incluses dans la situation et celles qui étaient précédées d'un ? ne seront pas affectées par ce franchissement. La situation de sortie classique ne peut pas être vide. Par exemple, dans la figure 3.19, la transition t_C est sensibilisée si la place $P6$ contient au moins un jeton et la place $P7$ au moins deux quel que soit le marquage des autres places. La transition t_D est sensibilisée si la place $P7$ ne contient pas de jeton. Le tir de t_C entraîne le retrait d'un jeton de la place $P6$ mais pas de $P7$. Le tir de t_D ne modifie pas le marquage de la MP.

Associer la situation $(*)$ à un arc permet de définir un arc exception. Une transition ayant au moins un arc exception en amont sera nommée transition exception. Par exemple, figure 3.19, t_{exc} est une transition d'exception. A noter, c'est bien l'arc sortant exception et seulement celui-ci qui va distinguer la transition exception d'une transition classique de RdP. Les transitions ne sont pas différenciées et une transition exception peut donc avoir tous types d'arcs en amont et en aval. Une transition exception n'est sensibilisée par un arc exception que si la macroplace est active, elle ne peut donc pas être franchie si la macroplace est inactive. Cela permet d'éviter que la transition exception ne se comporte comme une transition source, i.e. indéfiniment tirable. Avoir une transition source est gênant car le RdP obtenu est potentiellement non borné. Quand une transition exception est tirée, le marquage de toutes les places du raffinement devient nul, les ordres de tir des transitions du raffinement sont mis à zéro ainsi que les compteurs des transitions temporelles du raffinement et des transitions sortantes. Le lancement des fonctions ou l'exécution des actions n'est pas directement impacté par le tir d'une transition exception mais indirectement par l'annulation du marquage et des ordres de tir des transitions.

Pour pouvoir réaliser la purge de la macroplace quand la macroplace est dans un état spécifique, il est aussi possible d'avoir un arc exception dont la situation est $(s_{exc}, *)$ où la situation s_{exc} est une situation sortante ne pouvant contenir que des arcs tests ou inhibiteurs. Dans ce cas, la transition exception est sensibilisée par la situation s_{exc} si la macroplace est active. Par contre le tir de la transition exception entraînera la purge de la macroplace comme dans le cas d'un arc exception avec une situation (*). Cette possibilité permet au concepteur de déclencher la gestion d'une exception par un événement interne, ce qui simplifie la modélisation et augmente la réactivité.

Definition 3.2.3. *Une situation de sortie exception s_{s_e} est définie telle que $s_{s_e} = (s_{exc}, *)$ avec $s_{exc} = \{?n_i P_i\}$ où $n_i \in \mathbb{Z}\backslash\{0\}$ et $P_i \in P^{mp}$ avec P^{mp} l'ensemble des places du raffinement de la macroplace.*

Tirs simultanés

Comme les RdP GEITSP sont synchrones, plusieurs transitions peuvent être tirées simultanément. Si une transition entrante et sortante de la macroplace sont simultanément tirées, alors la macroplace reste active. Ce choix a été fait car classiquement (pour les grafcets par exemple) quand il y a activation et désactivation dans un modèle synchrone, le système reste actif.

Une transition exception est nécessairement en conflit avec les autres transitions sortantes. La transition exception est alors prioritaire sur les transitions sortantes.

S'il y a conflit entre des transitions entrantes, sortantes ou exceptions, des priorités sont définies comme entre les transitions en conflit d'un RdP GEITSP sans macroplace.

Définition formelle et sémantique d'un RdP GEITSP avec macroplace

Definition 3.2.4. *Soit $\mathcal{C}$ l'ensemble des conditions. Soit $\mathcal{F}$ l'ensemble des actions impulsives. Soit $\mathcal{A}$ l'ensemble des actions continues. Un RdP généralisé étendu interprété T-temporel synchrone à priorités (RdP GEITSP) avec macroplaces est un uplet $< P, T, M, Pre, Pre_t, Pre_i, Post, Entry, Exist, m_0, C, F, A, clock, Is, \succ >$, où :*

- $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, clock, Is, \succ >$ *est le RdP GEITSP principal.*
- M *est l'ensemble des macroplaces.*
- $mp \in M$ *est un RdP GEITSP défini par* $< P^{mp}, T^{mp}, Pre^{mp}, Pre_t^{mp}, Pre_i^{mp}, Post^{mp}, m_0^{mp}, C^{mp}, F^{mp}, A^{mp}, clock, I_s^{mp}, \succ^{mp} >$
- $P^{all} = P \cup \bigcup_{mp \in M} P^{mp}$, $T^{all} = T \cup \bigcup_{mp \in M} T^{mp}$.
- $Pre_M, Pre_{Mt}, Pre_{Mi}, Post_M : T \to \bigcup_{mp \in M} P^{mp} \to \mathbb{N}$ *sont respectivement la fonction précondition sortante, la fonction test sortante, la fonction inhibition sortante et la fonction postcondition entrante.*
- $Pre_{exc} : T \to M \to \mathbb{B}$ *est la fonction exception. Elle est égale à 1 quand il y a un arc exception entre la MP et la transition, à 0 sinon.*
- $Entry = (Post_M)$ *etExit* $= (Pre_M, Pre_{Mt}, Pre_{Mi}, Pre_{exc})$ *sont les descriptions des situations d'entrée et sortie.*

Le cas particulier d'un arc sortant auquel le concepteur souhaite associer une situation de la forme $(s_{exc}, *)$ est décrit formellement par un arc exception dans la fonction exception et le nombre d'arcs nécessaires dans les fonctions tests et inhibitions sortantes.

Une place ou une transition ne peut pas appartenir au RdP principal et à un raffinement ou à deux raffinements différents : $\forall mp \in M, P \cap P^{mp} = \emptyset$ et $T \cap T^{mp} = \emptyset$ et $\forall (mp, mp') \in M^2, P^{mp} \cap P^{mp'} = \emptyset$ et $T^{mp} \cap T^{mp'} = \emptyset$.

Soit $M_{exc}(t)$ l'ensemble des MP relié à une transition t par un arc exception : $\forall mp \in M, mp \in M_{exc}(t) \Leftrightarrow Post_{exc}(t)(mp) = 1$. La fonction $m : P^{all} \rightarrow \mathbb{N}$ définit le marquage complet du RdP. Le marquage d'une MP mp est défini par la fonction $m^{mp} = m_{|P^{mp}}$.

Dans le RdP principal, il faut désormais prendre en compte la sensibilisation par un arc exception. Ainsi, $\forall t \in T, t \in sens(m) \Leftrightarrow \Big(m \geq Pre(t) + Pre_t(t) + Pre_M(t) + Pre_{Mt}(t)\Big) \wedge \Big(m < Pre_i(t) + Pre_{Mi}(t)\Big) \wedge \Big(\forall mp \in M_{exc}(t), m^{mp} \neq 0\Big)$. Une transition $k \in T$ est nouvellement sensibilisée par le tir d'un groupe de transitions $T_{tir} \subset T^{all}$ depuis le marquage m, noté $k \in \uparrow sens(m, T_{tir})$, si et seulement si :

$$
\begin{aligned}
&\forall k \in T, k \in \uparrow sens(m, T_{tir}) \Leftrightarrow \\
&\quad \Big(m - \sum_{t \in T_{tir}} Pre(t) - \sum_{t \in T_{tir}} Pre_M(t) + \sum_{t \in T_{tir}} Post(t) + \sum_{t \in T_{tir}} Post_M(t) \\
&\quad \geq Pre(k) + Pre_t(k) + Pre_M(k) + Pre_{Mt}(k)\Big) \\
&\quad \wedge \Big(m - \sum_{t \in T_{tir}} Pre(t) - \sum_{t \in T_{tir}} Pre_M(t) + \sum_{t \in T_{tir}} Post(t) + \sum_{t \in T_{tir}} Post_M(t) \\
&\quad < Pre_i(k) + Pre_{Mi}(k)\Big) \\
&\quad \wedge \Big(\forall mp \in M_{exc}(t), m^{mp} - Pre^{mp}(t) + Post^{mp}(t) \neq 0\Big) \\
&\quad \wedge \Bigg[(k \in T_{tir}) \vee \Big(Pre_i(k) + Pre_{Mi}(k) \leq m - \sum_{t \in T_{tir}} Pre(t) - \sum_{t \in T_{tir}} Pre_M(t)\Big) \\
&\quad \vee \Big(Pre(k) + Pre_t(k) + Pre_M(k) + Pre_{Mt}(k) > m - \sum_{t \in T_{tir}} Pre(t) - \sum_{t \in T_{tir}} Pre_M(t)\Big) \\
&\quad \vee \Big(\exists mp \in M_{exc}(t) | m^{mp} - Pre^{mp}(t) = 0\Big)\Bigg]
\end{aligned}
$$

Les raffinements des macroplaces sont des RdP GEITSP donc les définitions des ensembles $sens(m)$ et $\uparrow sens(m, T_{tir})$ avec $T_{tir} \subset T^{all}$ ne sont pas modifiés :
$\forall mp \in M, \forall t \in T^{mp}, t \in sens(m) \Leftrightarrow (m \geq Pre^{mp}(t) + Pre_t^{mp}(t)) \wedge (m < Pre_i^{mp}(t))$ et

$$\forall mp \in M, \forall k \in T^{mp}, k \in\uparrow sens(m, T_{tir}) \Leftrightarrow$$
$$\left(m - \sum_{t \in T_{tir}} Pre^{mp}(t) + \sum_{t \in T_{tir}} Post^{mp}(t) \geq Pre^{mp}(k) + Pre_t^{mp}(k)\right)$$
$$\wedge \left(m - \sum_{t \in T_{tir}} Pre^{mp}(t) + \sum_{t \in T_{tir}} Post^{mp}(t) < Pre_i^{mp}(k)\right)$$
$$\wedge \left[(k \in T_{tir}) \vee \left(Pre_i^{mp}(k) \leq m - \sum_{t \in T_{tir}} Pre^{mp}(t)\right)\right.$$
$$\left. \vee \left(Pre^{mp}(k) + Pre_t^{mp}(k) > m - \sum_{t \in T_{tir}} Pre^{mp}(t)\right)\right]$$

La valeur instantanée d'une condition est toujours définie par la fonction val, la valeur fixée des conditions pour le calcul de l'évolution par la fonction $cond$ et l'exécution d'une action ou d'une fonction par la fonction ex comme pour les RdP GEITSP sans macroplace.

$I : T^{all} \rightarrow I^+$ est la fonction, appelée fonction des intervalles de temps, qui associe un intervalle de temps à chaque transition sensibilisée par m.

$reset_t : T^{all} \rightarrow \mathbb{B}$ est la fonction de réinitialisation des intervalles de temps. Elle permet de gérer la réinitialisation de ces intervalles due aux marquages transitoires.

L'état d'un RdP GEITSP avec macroplace est défini par $s = (m, cond, ex, I, reset_t)$.

Une MP mp est active si et seulement si $m^{mp} \neq 0$.

Une transition est tirable depuis l'état $s = (m, cond, ex, I, reset_t)$ du RdP, notée $t \in tirable(s)$ si et seulement si la transition est sensibilisée par le marquage m, que la valeur des conditions autorise le tir et que la borne minimale de l'intervalle a été atteinte :
$t \in tirable(s) \Leftrightarrow t \in sens(m) \wedge \left(\forall c \in \mathcal{C} | C(t)(c) = 1, cond(c) = 1 \wedge \forall c \in \mathcal{C} | C(t)(c) = -1, cond(c) = 0\right) \wedge 0 \in I(t)$.

Definition 3.2.5. *La sémantique d'un RdP GEITSP avec macroplaces* $< P, T, M, Pre, Pre_t, Pre_i, Post, Entry, Exist, m_0, C, F, A, clock, Is, \succ >$ *est le système de transitions temporisé* $< S, s_0, \rightsquigarrow >$ *où :*

- S *est l'ensemble des états* $(m, cond, ex, I, reset_t)$ *du RdP.*
- $s_0 = (m_0, 0, 0, I_0, 0)$ *est l'état initial où* 0 *est la fonction nulle et* I_0 *est la restriction de la fonction* Is *aux transitions sensibilisées par* m_0.
- $\rightsquigarrow \subseteq S \times Clk, T^n \times S$ *est la relation de transition d'état, notée par exemple* $s \overset{clk}{\rightsquigarrow} s'$, *définie comme suit : Soit* $Tir(s) \subseteq T^{all}$ *l'ensemble des transitions tirées depuis l'état* s. *A l'état initial, on a* $Tir(s_0) = \varnothing$.
  - $s = (m, cond, ex, I, reset_t) \overset{\downarrow clock}{\rightsquigarrow} s' = (m, cond, ex', I', reset_t)$ *si et seulement si* $\downarrow clock = 1$ *et :*
    1. $\forall c \in \mathcal{C}, cond'(c) = val(c)$ *(actualisation des valeurs des conditions)*

2. $\forall t \in T^{all}, t \in\uparrow sens(m, Tir(s) \vee reset_t(t) = 1) \Rightarrow I'(t) = Is(t) - 1$ *(si une transition est nouvellement sensibilisée ou qu'une place ordonne la réinitialisation de son intervalle de temps, l'intervalle de temps de cette transition est réinitialisé)*

3. $\forall t \in T^{all}, \Big(t \in sens(m) \wedge reset_t(t) = 0 \wedge t \notin\uparrow sens(m, Tir(s)) \wedge I'(t) \neq \varnothing\Big) \Rightarrow I'(t) = I(t) - 1$ *(si une transition est sensibilisée mais pas nouvellement sensibilisée, non bloquée et que son intervalle de temps ne doit pas être réinitialisé, son intervalle de temps évolue de manière classique)*

4. $\forall t \in T^{all}, t \notin\uparrow sens(m, Tir(s) \wedge I(t) = \varnothing \Rightarrow I'(t) = I(t)$ *(si la transition était bloquée et n'est pas nouvellement sensibilisée, elle reste bloquée)*

5. $\forall a \in \mathcal{A}, \exists p \in P^{all} | A(p)(a) = 1 \wedge m(p) \neq 0 \Rightarrow ex'(a) = 1$ *sinon* $ex'(a) = 0$ *(les valeurs de la fonction ex pour les actions continues sont mises à jour)*

On détermine alors $Tir(s')$ l'ensemble des transitions qui seront tirées.

6. $\forall t \in tirable(s), \forall t' \in T^{all} | t' \succ t, t' \notin tirable(s) \Rightarrow t \in Tir(s')$ *(si une transition est tirable et qu'aucune transition plus prioritaire ne l'est alors la transition sera tirée)*

7. $\forall t \in T^{all}$, *soit* $Pr(t)$ *l'ensemble des transitions* t_i *telles que* $t_i \succ t \wedge t_i \in Tir(s')$.
   - $\forall t \in T, t \in tirable(s'), \Big(t \in sens\big(m - \sum_{t_i \in Pr(t)} (Pre(t_i) + Pre_M(t_i))\big) \wedge t \in sens\big(m - \sum_{t_i \in Pr(t)} (Pre(t_i) + Pre_M(t_i)) + \sum_{t_i \in Pr(t)} (Post(t_i) + Post_M(t_i))\big)\Big) \Rightarrow t \in Tir(s')$ *(une transition tirable du RdP prioritaire est tirée si le marquage est suffisant pour tirer toutes les transitions tirables plus prioritaires et cette transition)*
   - $\forall mp \in M, \forall t \in T^{mp}, t \in tirable(s'), \Big(t \in sens\big(m - \sum_{t_i \in Pr(t)} Pre^{mp}(t_i)\big) \wedge t \in sens\big(m - \sum_{t_i \in Pr(t)} Pre^{mp}(t_i) + \sum_{t_i \in Pr(t)} Post^{mp}(t_i)\big)\Big) \Rightarrow t \in Tir(s')$ *(une transition tirable du raffinement est tirée si le marquage est suffisant pour tirer toutes les transitions tirables plus prioritaires et cette transition)*

8. - $\forall t \in T, t \in tirable(s'), \Big(t \notin sens\big(m - \sum_{t_i \in Pr(t)} (Pre(t_i) + Pre_M(t_i))\big) \vee t \notin sens\big(m - \sum_{t_i \in Pr(t)} (Pre(t_i) + Pre_M(t_i)) + \sum_{t_i \in Pr(t)} (Post(t_i) + Post_M(t_i))\big)\Big) \Rightarrow t \notin Tir(s')$ *(une transition tirable du RdP prioritaire n'est pas tirée si le marquage n'est pas suffisant pour tirer toutes les transitions tirables plus prioritaires et cette transition)*
   - $\forall mp \in M, \forall t \in T^{mp}, t \in tirable(s'), \Big(t \notin sens\big(m - \sum_{t_i \in Pr(t)} Pre^{mp}(t_i)\big) \vee t \notin sens\big(m - \sum_{t_i \in Pr(t)} Pre^{mp}(t_i) + \sum_{t_i \in Pr(t)} Post^{mp}(t_i)\big)\Big) \Rightarrow t \notin Tir(s')$ *(une transition tirable du raffinement n'est pas tirée si le marquage n'est pas suffisant pour tirer toutes les transitions tirables plus prioritaires et cette transition)*

9. $\forall t \notin tirable(s') \Rightarrow t \notin Tir(s')$ *(si la transition n'est pas tirable alors elle ne sera pas tirée)*

Soit $Tir_{exc}(s)$ *l'ensemble des transitions exceptions tirées depuis l'état* s *:* $t \in Tir_{exc}(s) \Leftrightarrow t \in Tir(s) \wedge \exists mp \in M | Pre_{exc}(t)(mp) = 1$.

– $s = (m, cond, ex, I, reset_t) \overset{Tir_{exc}(s)}{\rightsquigarrow} s' = (m', cond, ex, I', reset_t)$ *si et seulement si* $clock = 0$ *et :*

1. $\forall t \in Tir_{exc}(s), \forall mp \in M | Pre_{exc}(t)(mp) = 1, \forall p \in P^{mp}, m'(p) = 0$ *(annulation du marquage des places appartenant aux macroplaces liées à au moins une transition exception tirée)*
2. $\forall t \in Tir_{exc}(s), \forall mp \in M | Pre_{exc}(t)(mp) = 1, \forall k \in T^{mp}, I'(k) = I_s(k)$ *(réinitialisation des intervalles de temps des transitions appartenant aux macroplaces liées à au moins une transition exception tirée)*
3. $\forall t \in Tir_{exc}(s), \forall k \in T^{all}, \exists mp \in M | \Big(Pre_{exc}(t)(mp) = 1 \wedge \exists p \in P^{mp} | Pre_M(k)(p) + Pre_{Mt}(k)(p) + Pre_{Mi}(k)(p) + Pre_{exc}(k)(mp) \geq 1 \Big) \Rightarrow I'(k) = I_s(k)$ *(réinitialisation des intervalles de temps des transitions sortantes des macroplaces liées à au moins une transition exception tirée)*

   Le marquage et les compteurs de transitions non modifiées par les propositions ci-dessus ne sont pas modifiés (non impactés par le tir des transitions exceptions) :

4. $\forall p \in P, m'(p) = m(p)$
5. $\forall mp \in M | \big(\forall t \in Tir_{exc}(s), Pre_{exc}(t)(mp) = 0 \big), \forall p \in P^{mp}, m'(p) = m(p)$
6. $\forall mp \in M | \big(\forall t \in Tir_{exc}(s), Pre_{exc}(t)(mp) = 0 \big), \forall k \in T^{mp}, I'(k) = I(k)$
7. $\forall t \in Tir_{exc}(s), \forall k \in T, \forall mp \in M | \Big(Pre_{exc}(t)(mp) = 0 \vee \forall p \in P^{mp}, Pre_M(t)(p) + Pre_{Mt}(t)(p) + Pre_{Mi}(t)(p) + Pre_{exc}(k)(mp) = 0 \Big), I'(k) = I(k)$
8. $\forall t \in T, t \in Tir(s) \Leftrightarrow t \in Tir(s')$ *(les ordres de tirs des transitions du RdP principal ne sont pas modifiés)*
9. $\forall t \in Tir_{exc}(s), \forall mp \in M | Pre_{exc}(t)(mp) = 1, \forall k \in T^{mp}, k \notin Tir(s')$ *(les ordres de tir des transitions internes aux macroplaces liées à au moins une transition exception tirée sont annulés)*
10. $\forall mp \in M | (t \in Tir_{exc}(s) \Rightarrow Pre_{exc}(t)(mp) = 0), \forall k \in T^{mp}, k \in Tir(s) \Leftrightarrow k \in Tir(s')$ *(les ordres de tir des transitions internes des macroplaces non liées à une transition exception tirée ne sont pas modifiés)*

– $s = (m, cond, ex, I, reset_t) \overset{\uparrow clock}{\rightsquigarrow} s' = (m', cond, ex', I', reset'_t)$ *si et seulement si* $\uparrow clock = 1$ *et :*

1. – $\forall p \in P, m'(p) = m(p) - \sum_{t \in Tir(s)} \big(Pre(t)(p) + Pre_M(t)(p) \big) + \sum_{t \in Tir(s)} \big(Post(t)(p) + Post_M(t)(p) \big)$ *(actualisation du marquage du RdP principal suivant les transitions devant être tirées)*

- $\forall mp \in M, \forall p \in P^{mp}, m'(p) = m(p) - \sum_{t \in Tir(s)} Pre^{mp}(t)(p) + \sum_{t \in Tir(s)} Post^{mp}(t)(p)$ *(actualisation du marquage des raffinements suivant les transitions devant être tirées)*

2. - $\forall t \in T, \exists p \in P | m(p) - \sum_{t_i \in Tir(s)} Pre(t_i)(p) + Pre_M(t_i)(p) < \Big(Pre(t)(p) + Pre_t(t)(p) + Pre_M(t)(p) + Pre_{Mt}(t)(p)\Big) \Rightarrow reset'_t(t) = 1$ *sinon* $reset'_t(t) = 0$ *(pour chaque transition du RdP principal si le marquage transitoire d'au moins une place amont est tel qu'il désensibilise cette transition, l'ordre est donné de réinitialiser l'intervalle de temps de la transition)*
   - $\forall mp \in M, \forall t \in T, \exists p \in P | m(p) - \sum_{t_i \in Tir(s)} Pre^{mp}(t_i)(p) < \Big(Pre^{mp}(t)(p) + Pre_t^{mp}(t)(p)\Big) \Rightarrow reset'_t(t) = 1$ *sinon* $reset'_t(t) = 0$ *(pour chaque transition interne d'une macroplace si le marquage transitoire d'au moins une place amont est tel qu'il désensibilise cette transition, l'ordre est donné de réinitialiser l'intervalle de temps de la transition)*
3. $\forall f \in \mathcal{F}, \exists t \in Tir(s) | F(t)(f) = 1 \Rightarrow ex'(f) = 1$ *sinon* $ex'(f) = 0$ *(les valeurs de la fonction ex pour les actions impulsives sont mises à jour)*
4. $\forall t \in T^{all}, (t \notin Tir(s) \wedge \uparrow I'(t) = 0) \Rightarrow I'(t) = \varnothing$ *sinon* $I'(t) = I(t)$ *(les transitions non tirées qui ont atteint la valeur maximale autorisée pour leur compteur sont bloquées, les autres compteurs ne sont pas modifiés)*
5. $Tir(s) = Tir(s')$ *(l'ensemble des transitions tirées n'est pas modifié)*

3.2.2 Mise en œuvre de la macroplace

Il a été choisi de ne pas réifier les MP dans le code VHDL, la macroplace est une entité purement descriptive qui facilite la spécification. Il demeure néanmoins nécessaire de trouver un moyen efficace d'implémenter la macroplace car transformer le modèle avec macroplace en RdP GEITSP sans macroplace nous ferait perdre tout l'intérêt de la compacité d'expression de la macroplace notamment pour la gestion de l'exception. Pour implémenter la macroplace, plusieurs points doivent alors être étudiés :

- comment traduire les arcs entrants et arcs sortants classiques,
- comment traduire l'activité de la macroplace pour pouvoir déterminer la sensibilisation par un arc sortant exception,
- comment gérer le tir d'une transition exception.

Traduction des arcs entrants et arcs sortants classiques

Les arcs entrants, les arcs sortants classiques et la partie s_s des arcs exceptions de la forme $(s_s, *)$ peuvent être facilement transformés en arcs classiques de RdP (ie en arcs PT, TP, test ou inhibiteur) (cf. 3.2.3). Ils seront ainsi transformés en arcs classiques automatiquement par une transformation de modèle (endogène) avant la génération du code VHDL. Ils seront donc traités comme des arcs classiques dans la mise en œuvre quel que soit le type d'implémentation choisie.

En général, d'un point de vue implémentation, les arcs exceptions de la forme $(s_s, *)$ sont traités comme un arc sortant classique avec la situation s_s et un arc exception.

Traduction de la sensibilisation par un arc sortant exception

Un arc sortant exception sensibilise sa transition aval si et seulement si au moins une des places de la macroplace est marquée (cf. §3.2.1). Un process VHDL est alors créé pour calculer si les arcs exceptions sensibilisent leurs transitions aval ou non : pour chaque MP, le signal *action_activation* de chaque place de la MP est utilisé pour savoir si le marquage de la place est nul ou non et un OU entre chacun de ces signaux est réalisé. Le résultat de ce OU sera le signal modélisant les arcs exceptions de la MP. Il entre alors dans les instances des composants transitions décrivant les transitions exceptions comme le signal de sensibilisation d'une place classique de RdP (puisqu'une transition exception est identique à une transition classique).

Le calcul de la sensibilisation est réalisé en asynchrone. En effet, la sensibilisation des transitions doit être connue avant le front descendant de l'horloge ① alors que le marquage des places n'est connu qu'après le front montant de l'horloge ③. Il faut donc calculer la sensibilisation entre ces deux instants ④ pour ne pas modifier l'évolution du RdP d'un point de vue temporel (cf. figure 3.20). Il faudra alors s'assurer lors du processus de synthèse du circuit que le calcul combinatoire se stabilise bien avant le front descendant de l'horloge.

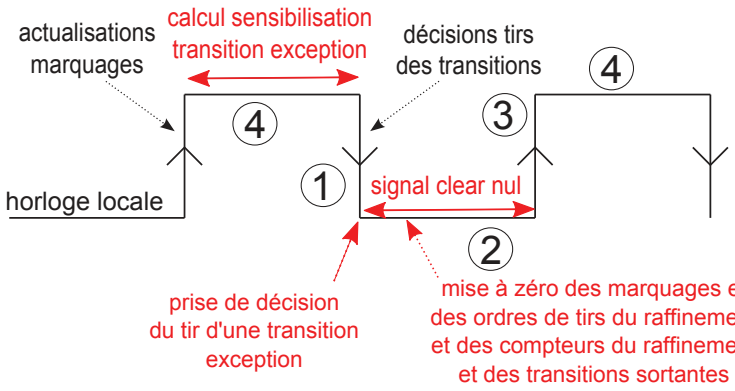

FIGURE 3.20 – Description temporelle de la gestion de l'exception

Le code VHDL pour déterminer l'activité de la MP donnée figure 3.19 (et donc la sensibilisation de la transition exception qui lui est associé) est décrit figure 3.21.

Gestion du tir d'une transition exception Pour implémenter une transition exception d'une macroplace, il faut être capable de vider la macroplace quelle que soit la situation de la macroplace. Vider la macroplace revient à remettre à zéro le marquage de ses places, mais aussi les ordres de tirs internes, les compteurs de temps des transitions internes et sortantes, et faire une réinitialisation des actions liées aux places du raffinement.

Bien que la mise en œuvre du RdP soit synchrone et donc que le tir d'une transition exception soit réalisé de manière synchrone en ①, les effets du tir de cette transition sont gérés de manière asynchrone en ② (cf. figure 3.20) notamment pour éviter que le réseau ne continue d'évoluer sur un pas de temps avant que l'exception ne soit prise en compte.

```
-- calcul signal representant l'activite de la macroplace (=sensibilisation transition
    exception)
process ( s_action_P1, s_action_P2, s_action_P3, s_action_P4, s_action_P5, s_action_P6,
    s_action_P7 )
begin
        if (s_action_P1 or s_action_P2 or s_action_P3 or s_action_P4 or s_action_P5 or
            s_action_P6 or s_action_P7)
                then
                        te_sensibilisation_exception <= '1';
        else
                 te_sensibilisation_exception <= '0';
        end if;
end process;
```

FIGURE 3.21 – Code pour le calcul des signaux relatifs à l'activité de la macroplace pour le modèle donné figure 3.19

Pour gérer le tir de cette exception un signal traité de manière combinatoire, appelé *clear*, est associé à chaque macroplace. La prise de décision du tir d'une transition exception de la macroplace entraîne la mise à 0 immédiate dudit signal. L'ordre de vider la macroplace est alors géré de manière asynchrone (combinatoire) grâce à ce signal.

Pour mettre à zéro les compteurs de temps des transitions du raffinement ainsi que leur ordre de tir, un signal *reset_transition* est créé qui est un OU entre le signal de reset global (signal *reset_n* du système) et le signal *clear*. En effet, cela revient à faire une réinitialisation des composants transition et transition temporelle. Par contre, la mise à zéro du marquage des places du raffinement ne revient pas à faire une réinitialisation d'un composant place. En effet, une réinitialisation rend le marquage égal au marquage initial qui n'est pas forcément nul. Une nouvelle entrée *clear* est alors créée dans le composant place. Elle permet de remettre le marquage à zéro de manière asynchrone. Il n'y a pas besoin de s'occuper des fonctions car l'ordre de lancer les fonctions est donné au front montant d'horloge suivant, or le signal de tir des transitions aura déjà été remis à zéro, si nécessaire. Les actions seront remises à 0, si nécessaire, de manière classique. Ne pas remettre à 0 toutes les actions lors du tir d'une transition exception permet d'éviter d'avoir un ordre d'exécution d'actions fugace (i.e mis à 1 sur ① et annuler pendant ②).

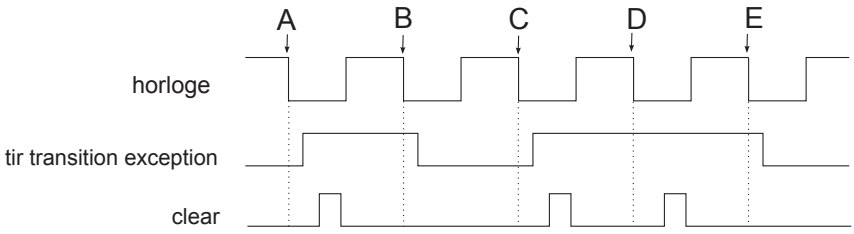

FIGURE 3.22 – Fonctionnement souhaité du signal clear

La gestion du signal *clear* est un peu délicate car aucun front de signaux VHDL ne peut être utilisé pour gérer le début de la purge. La figure 3.22 présente le comportement que le signal *clear* doit avoir, c'est-à-dire l'ordre de réaliser la purge de la macroplace doit être donné à chaque fois que la décision de tirer une transition exception est prise. Ainsi, sur le front descendant A, la décision est prise de tirer la transition, le signal *clear* est émis avant le front montant suivant. Par contre, sur le front descendant B, la transition exception n'est pas tirée donc le signal *clear* n'est pas émis. La décision est prise de tirer à nouveau la transition exception en C et en D donc le signal *clear* est émis deux fois.

L'inconvénient est que la valeur des signaux *horloge* et *tir transition exception* est identique aux instants B et D pourtant le comportement attendu du signal *clear* n'est pas le même. Le problème provient du retard de propagation des signaux tirs. En effet, quand le signal s_tir de la transition exception doit être remis à 0, il n'est pas remis à 0 exactement sur le front descendant de l'horloge mais un peu après, le "un peu après" étant difficilement quantifiable de manière automatique. Comme aucun front de signal ne peut être utilisé, il faut attendre un temps arbitraire mais suffisant avant de relancer si besoin une purge pour être sûr que les signaux de tirs de transitions soient bien actualisés.

Le signal *clear* est alors géré par la logique combinatoire donnée en schéma figure 3.24. Le chronogramme correspondant est donné figure 3.23. Cette logique combinatoire est intégrée dans un composant VHDL. Le signal *clear* est mis à 1 une fois que le signal s_tir de la transition exception passe ou reste à 1 après un front descendant d'horloge. Il est remis à zéro quand le (ou les) signal (signaux) de sensibilisation des macroplaces amont passe(nt) à zéro. Cette solution fonctionne quand il n'y a qu'une transition exception sur la macroplace car le retard instauré par le traitement combinatoire est suffisant dans ce cas.

Ceci n'est plus vrai quand il faut en plus réaliser un OU entre tous les signaux des transitions exceptions associées à une même macroplace pour savoir si la macroplace doit être vidée ou non. Dans ce cas, un buffer est ajouté de sorte que le retard devienne suffisant. Le code VHDL du process utilisé dans ce cas est donné figure 3.25. Les buffers sont utilisables dans tous les FPGA mais leur description en VHDL est non générique. Le code présenté ne fonctionnera donc qu'avec un FPGA de type IGLOO.

Il faudra néanmoins s'assurer une fois le processus de synthèse du circuit effectué que les signaux se stabilisent bien en moins d'un demi-top d'horloge et que le retard instauré par le buffer est bien suffisant.

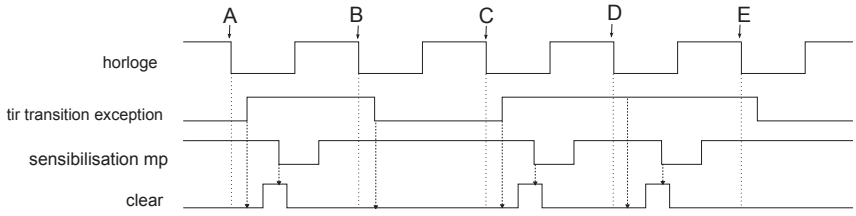

FIGURE 3.23 – Chronogramme des signaux gérant le signal clear

Comme l'implémentation est synchrone, il peut y avoir simultanément le tir d'une transition entrante de la macroplace et celui d'une transition exception de cette même macroplace. Dans ce cas, la macroplace est désactivée mais l'ordre de tir de la transition entrante n'est pas annulé, le marquage de la macroplace sera donc modifié selon ce tir au front montant (suivant) de l'horloge et donc la macroplace sera réactivée. Il y aura donc une désactivation suivie d'une réactivation comme dans le cas du grafcet où le même problème se pose.

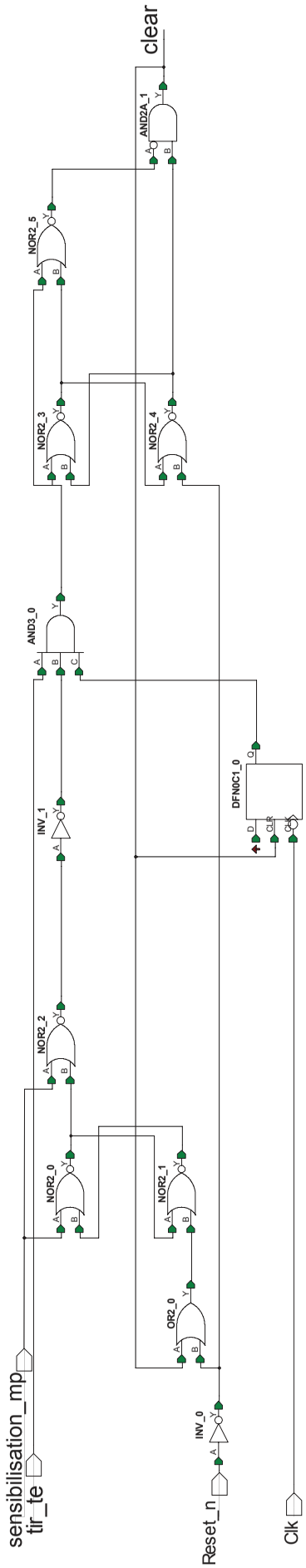

FIGURE 3.24 – Traitement combinatoire pour la gestion du signal *clear* dans le cas d'une unique transition exception

```
entity Handling_clear is
    port(
        Clk                : in  std_logic;
        Reset_n            : in  std_logic;
        sensibilisation_mp : in  std_logic;
        tir_te             : in  std_logic;
        Clear              : out std_logic);
end Handling_clear;

architecture RTL of Handling_clear is

component BUFD
    port(
        A : in  std_logic;
        Y : out std_logic);
end component;

signal Clk_late_1       : std_logic;
signal Clk_late_2       : std_logic;
signal Clk_late_3       : std_logic;
signal Clk_late_4       : std_logic;
signal Clk_late_5       : std_logic;
signal Clk_late_6       : std_logic;
signal Clk_late_7       : std_logic;
signal Clk_late_8       : std_logic;
signal Clk_late_9       : std_logic;
signal Clk_late         : std_logic;
signal NOR2_0       : std_logic;
signal NOR2_1       : std_logic;
signal NOR2_2       : std_logic;
signal NOR2_3       : std_logic;
signal NOR2_4       : std_logic;
signal NOR2_5       : std_logic;
signal AND3_0       : std_logic;
signal AND2A_1      : std_logic;
signal OR2_0        : std_logic;
signal DFF_Q        : std_logic;

begin
-------------------------------------------------------------------------
-- Retard horloge (non générique)
BUF_CLK_0 : BUFD port map(A => Clk, Y => Clk_late_1);
BUF_CLK_1 : BUFD port map(A => Clk_late_1, Y => Clk_late_2);
BUF_CLK_2 : BUFD port map(A => Clk_late_2, Y => Clk_late_3);
BUF_CLK_3 : BUFD port map(A => Clk_late_3, Y => Clk_late_4);
BUF_CLK_4 : BUFD port map(A => Clk_late_4, Y => Clk_late_5);
BUF_CLK_5 : BUFD port map(A => Clk_late_5, Y => Clk_late_6);
BUF_CLK_6 : BUFD port map(A => Clk_late_6, Y => Clk_late_7);
BUF_CLK_7 : BUFD port map(A => Clk_late_7, Y => Clk_late_8);
BUF_CLK_8 : BUFD port map(A => Clk_late_8, Y => Clk_late_9);
BUF_CLK_9 : BUFD port map(A => Clk_late_9, Y => Clk_late);
-------------------------------------------------------------------------

NOR2_0 <= not(sensibilisation_mp or NOR2_1);
NOR2_1 <= not(OR2_0 or NOR2_0);
NOR2_2 <= not(sensibilisation_mp or NOR2_0);
NOR2_3 <= not(AND3_0 or NOR2_4);
NOR2_4 <= not(NOR2_3 or not(reset_n));
NOR2_5 <= not(AND3_0 or NOR2_3);
AND3_0 <= tir_te and not(NOR2_2) and DFF_Q;
AND2A_1 <= not(NOR2_5) and NOR2_4;
OR2_0 <= AND2A_1 or not(reset_n);

process(Clk_late, AND2A_1) begin
        if (AND2A_1 = '1') then
                DFF_Q <= '0';
        elsif (Clk_late'event and Clk_late = '0') then
                DFF_Q <= '1';
        end if;
end process;

Clear <= AND2A_1;

end RTL;
```

FIGURE 3.25 – Code VHDL du process gérant le signal clear

3.2.3 Obtention d'un modèle analysable

Pour pouvoir analyser un modèle contenant des macroplaces (comme celui donné en exemple figure 3.19) grâce à un analyseur déjà existant (TINA par exemple), il faut être capable d'obtenir, en partant du modèle contenant des macroplaces, un modèle équivalent n'utilisant que les éléments d'un RdP GET. Cette transformation de modèles doit pouvoir être automatisée. En dehors des éléments liés à la gestion de la traduction de la macroplace, la transformation du RdP GEITSP en RdP GET est toujours réalisée en utilisant la méthode proposée dans le chapitre 2. Comme dans la littérature, la solution que nous avons adoptée pour obtenir un modèle analysable est la remise à plat du modèle avec macroplace.

Pour cette remise à plat, trois principaux problèmes se posent : traduire les arcs entrants et sortants classiques, traduire les arcs sortants exceptions (i.e. modéliser l'activité de la macroplace et modéliser la purge des places) et gérer les tirs simultanés des transitions entrantes et/ou sortantes. Ces problèmes (et leurs solutions) vont être détaillés dans les paragraphes suivants. Le modèle analysable équivalent à la macroplace décrite figure 3.19 est donné figure 3.33. Les étapes pour obtenir ce modèle sont décrites dans les sous-parties suivantes.

Mise à plat des arcs entrants et sortants classiques de la macroplace

Nous allons ici nous intéresser à la transformation des arcs entrants et sortants classiques de la macroplace en arcs classiques de RdP. Dans ces deux cas, la solution est simple. Il suffit, en effet, d'ajouter au modèle les arcs décrits par la situation associée à chacun des arcs entrants et sortants classiques de la macroplace, comme décrit ci-dessous.

Pour un arc entrant associé à une transition t, pour chaque place P_i dans la liste des places de la situation s_e, un arc pondéré de n_i est ajouté entre t et P_i .

Pour un arc sortant classique associé à une transition t, pour chaque place P_i dans la liste des places de la situation s_s :

- Si $X_i = \epsilon$, un arc pondéré du nombre n_i est ajouté entre P_i et t.
- Si $X_i =?$ et $n_i > 0$, un arc test pondéré du nombre n_i est ajouté entre P_i et t.
- Si $X_i =?$ et $n_i < 0$, un arc inhibiteur pondéré de la valeur absolue du nombre n_i est ajouté entre P_i et t.

Pour les arcs sortants du type $(s_s, *)$, la traduction de la sensibilisation par la situation s_s est traduite comme pour les arcs sortants classiques. Pour la sensibilisation par $*$ et pour la purge, l'arc sera également traduit comme les arcs sortants exceptions.

Après transformation des fonctions d'entrée et de sortie classiques de notre exemple de la figure 3.19, le modèle donné figure 3.26 a été obtenu. En rouge et en gras sont indiqués les arcs qui ont été ajoutés pour la remise à plat des arcs entrants et sortants classiques.

La remise à plat de la fonction de sortie exception, plus complexe, est décrite, par étapes, dans la suite de ce document.

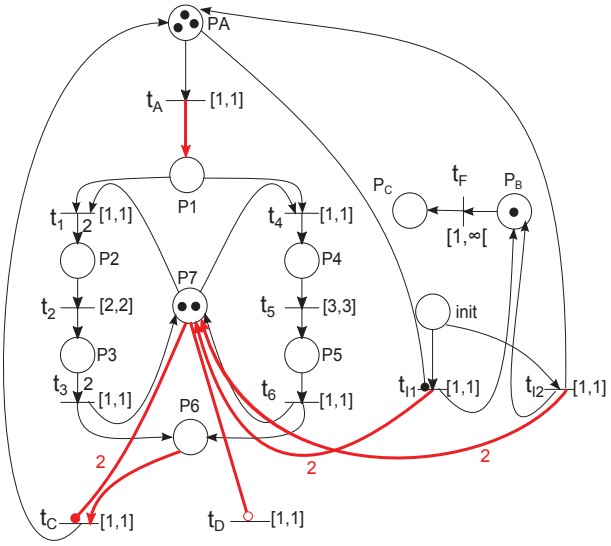

FIGURE 3.26 – Modèle de la figure 3.19 remis à plat en omettant la transition exception

Prise en compte de l'activité de la macroplace dans le tir d'une transition exception

L'arc auquel est associé une situation exception (∗) ne sensibilise sa transition cible que si la macroplace amont de cet arc est active. Rappelons également que selon la définition donnée §3.2.1, une macroplace est active si et seulement si son marquage est non nul.

Comme la macroplace n'existe plus dans le modèle remis à plat, l'idée est de traduire le fait que la MP est active ou non de façon explicite par l'intermédiaire d'une place virtuelle (c'est-à-dire utilisée uniquement pour l'analyse) appelée *MP active*. Le principe est de regarder les cas où la macroplace va être activée, l'information que la MP est active est alors mémorisée dans cette place virtuelle. Il faudra donc gérer le marquage de cette place lors de l'activation, mais aussi lors de la désactivation de la MP.

Activation Pour savoir quand la place *MP active* doit devenir marquée, une solution simple serait de faire partir un arc TP qui se dirige vers la place *MP active* de chaque transition qui peut créer de l'activité dans la macroplace (i.e. les transitions entrantes et les transitions internes pouvant créer des jetons) (cf. figure 3.27).

Mais avec cette solution, la place *MP active* peut contenir plusieurs jetons puisqu'on peut rentrer plusieurs fois dans une MP sans qu'elle ne se désactive entre chaque entrée. Cela est très gênant pour l'analyse car rien ne garantit que le marquage de cette place soit borné. Il faut donc adapter cette solution.

Au lieu de mettre directement un jeton dans la place *MP active*, le franchissement d'une transition créant de l'activité dans la macroplace met un jeton dans une place *demande d'activation* qui n'ajoutera un jeton dans la place *MP active* que si cette dernière n'est pas déjà marquée (cf. figure 3.28). Cette activation doit bien sûr se faire en temps nul donc avec des transitions virtuelles (i.e. utilisée uniquement pour l'analyse) qui ont des fenêtres temporelles [0,0]. Cela permet d'assurer la correspondance du modèle analy-

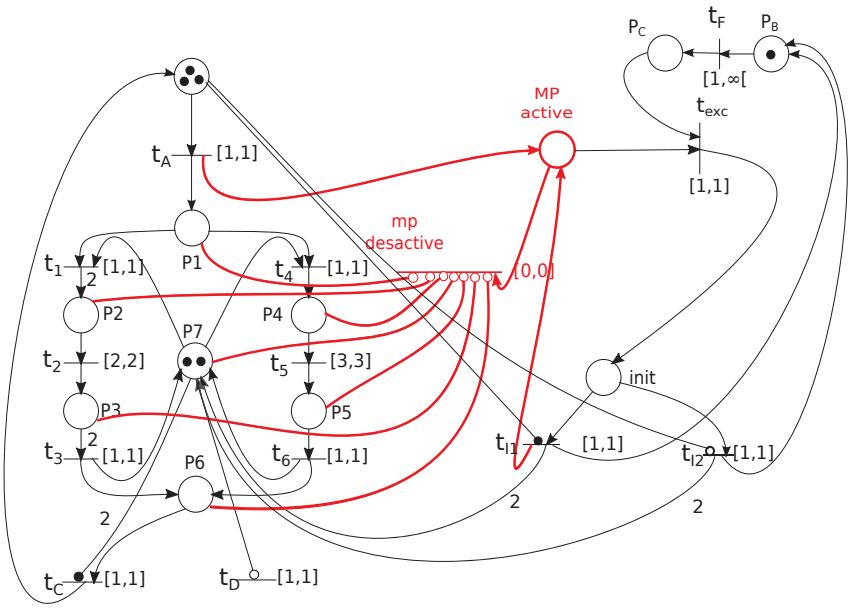

FIGURE 3.27 – Modélisation de l'activité de la MP (version non bornée)

sable avec le modèle implémentable dans lequel l'activation est calculée immédiatement de manière asynchrone (cf. figure 3.20). Sur notre exemple, les éléments à ajouter pour prendre en compte l'activité de la macroplace de façon bornée sont montrés en couleur et en gras dans la figure 3.28.

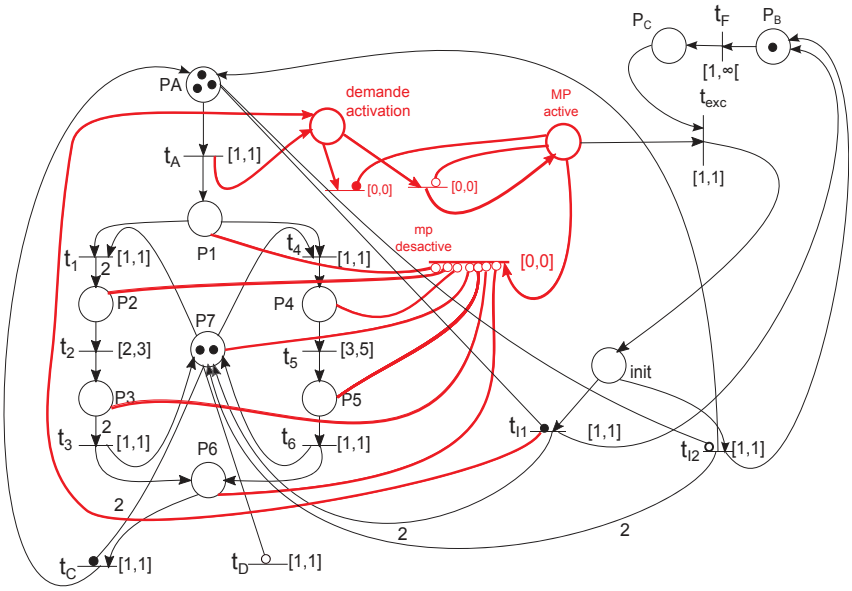

FIGURE 3.28 – Modélisation de l'activité de la MP (version bornée)

Désactivation Pour pouvoir vider la place *MP active* quand c'est nécessaire, une transition puits virtuelle de fenêtre temporelle [0,0] est ajoutée et reliée par un arc PT à la place *MP active*. Un arc inhibiteur est alors ajouté entre chaque place du raffinement

et cette transition, ce qui nous permet de déterminer immédiatement le moment où le marquage de la MP est nul (cf. figure 3.27), et ainsi de la désactiver. La désactivation sera immédiate grâce à l'intervalle $[0, 0]$.

Autre solution pour l'activation Une autre solution pour gérer l'activation de la MP est de faire un test sur le marquage des places. Si au moins l'une des places est marquée, la MP est activée (un jeton est placé dans la place *MP active*) si elle ne l'est pas déjà (cf. figure 3.29). Mais dans ce cas toutes les transitions virtuelles t_{ai} seront en concurrence dès que plusieurs places sont marquées, et cela complexifie inutilement l'analyse. Nous préférerons donc la solution donnée 3.28.

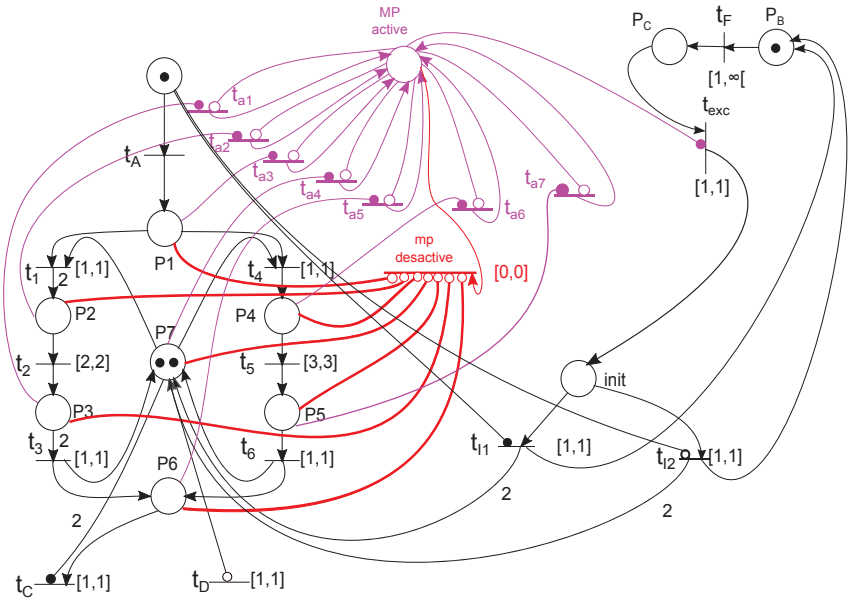

FIGURE 3.29 – Modélisation de l'activité de la MP (deuxième solution)

Concurrence entre l'activation et la désactivation Nous souhaitons que si une macroplace est simultanément activée et désactivée de manière classique (tirs simultanés d'une transition entrante et d'une transition sortante classique), la macroplace reste active d'après la définition donnée en 3.2.1. Pour assurer cela, il faut que l'activité de la MP soit toujours évaluée après le tir des transitions entrantes sinon la MP risque d'être désactivée puis réactivée. Toutes les transitions entrantes sont donc prioritaires sur la transition *mp desactive*.

Purge de la macroplace

Le tir de la transition exception doit entraîner la purge immédiate de la macroplace (i.e. le retrait de tous les jetons pouvant se trouver dans les places de la macroplace en moins d'une unité de temps). Du point de vue de l'implémentation, cette purge se fait sur toutes les places et sur tous les jetons de manière simultanée et asynchrone. Il faut

donc traduire cette caractéristique dans le modèle analysable pour garantir la conformité de l'analyse avec l'implémentation.

L'implémentation étant synchrone, toutes les fenêtres temporelles associées aux transitions implémentées ont une borne inférieure au moins égale à 1. L'idée est donc de vider les places à l'aide de transitions virtuelles spécifiques à l'analyse ayant une fenêtre temporelle [0,0]. Cela permet ainsi de respecter en même temps le caractère simultané du retrait des jetons, puisque tous les jetons seront bien tous retirés avant le prochain pas de temps, mais aussi le caractère asynchrone et immédiat de la prise en compte de l'exception puisqu'elle est traitée dès que possible.

Deux méthodes sont envisageables pour vider automatiquement toutes les places de la macroplace (cf. §3.1.1) sans déterminer l'ensemble des marquages possibles : en parallèle (figure 3.30) ou en séquence (figure 3.31). En parallèle, toutes les places sont vidées simultanément tandis qu'en séquence, on attend que la première place soit vidée pour vider la seconde et ainsi de suite. Pour simplifier la lecture des figures, seul le raffinement de la MP et la transition t_{exc} sont représentés en plus du mécanisme de purge représenté en couleur et en gras.

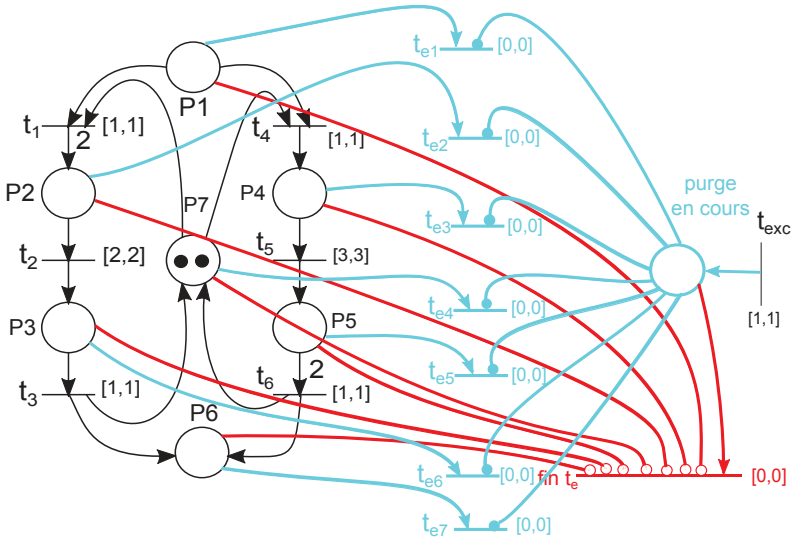

FIGURE 3.30 – Méthode pour purger une macroplace en parallèle

Si la méthode parallèle peut sembler intuitivement plus proche de l'implémentation, puisque toutes les places commencent à se vider simultanément, les deux méthodes reviennent en fait au même du point de vue temporel puisque tous les tirs des transitions de purge se font en "temps nul".

L'inconvénient de la méthode parallèle d'un point de vue analyse est que les transitions t_{e1} à t_{e7} sont en concurrence entre elles et avec toutes les transitions tirables à cet instant : il y a entrelacements des transitions. Ceci entraîne, dans le cadre d'une analyse exhaustive de tous les états possibles, une multiplication des chemins d'exécution et rend ainsi l'analyse plus complexe. Nous préférerons donc la méthode séquentielle.

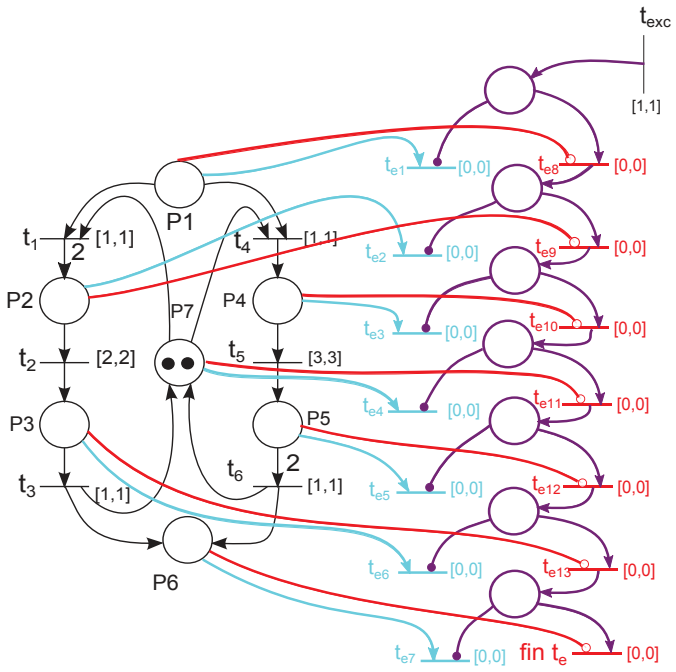

FIGURE 3.31 – Méthode pour purger une macroplace de manière séquentielle

Reste à gérer les autres entrelacements puisque les transitions internes à la MP t_1 à t_7 peuvent être en concurrence avec les transitions de purge t_{e1} à t_{e13} et $fin\ te$ (cf. figure 3.31). La concurrence doit être bien gérée (priorité aux transitions de la purge sur les transitions internes) pour assurer que la purge se fera correctement.

Une solution simple pour éviter cet entrelacement est d'ajouter une place indiquant que le tir de la transition exception est en cours. Des arcs inhibiteurs entre cette place et toutes les transitions internes à la macroplace sont ajoutés, ce qui bloque l'évolution interne de la MP pendant la purge de celle-ci. Cette place sera vidée à la fin de la purge de la dernière place. Cette solution est présentée sur notre exemple figure 3.32.

Cette solution est d'autant plus intéressante que dans le modèle implémentable le tir des transitions du raffinement sont empêchés lors du tir d'une transition exception, certains marquages non réalistes possibles uniquement dans le modèle analysé sont ainsi évités. De plus, l'ajout de cette place et des arcs inhibiteurs permet de garantir la mise à zéro des horloges de toutes les transitions temporelles du raffinement quels que soient leurs arcs amonts (i.e. même si elles n'ont que des arcs inhibiteurs en amont).

Gestion des tirs simultanés

Comme l'implémentation du réseau de Petri est synchrone, il peut y avoir tir simultané de plusieurs transitions si elles ne sont pas en conflits effectifs.

Il faut s'assurer, pour la remise à plat du modèle vers l'analyse, que si une transition exception et une transition entrante de la MP sont tirées simultanément, il y aura, comme dans l'implémentation, purge de la macroplace puis ajout des jetons. La macroplace ne

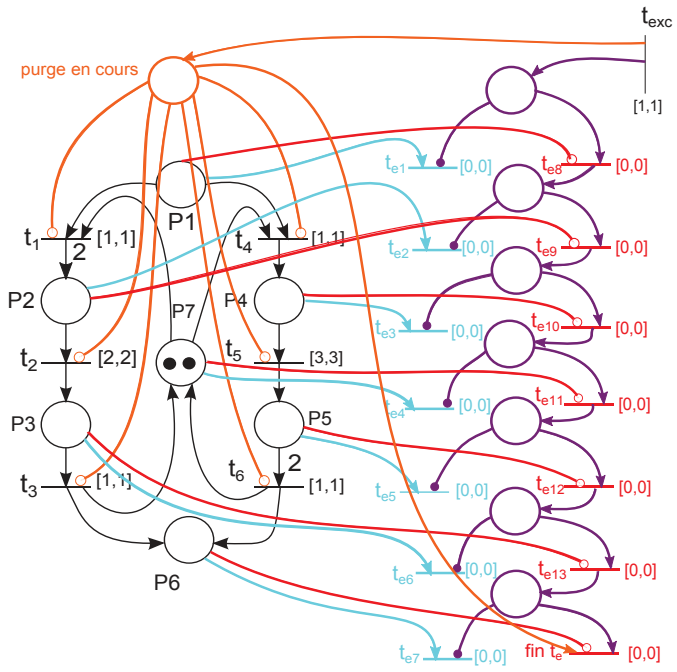

FIGURE 3.32 – Suppression des entrelacements

reste, par contre, pas active comme pour un tir simultané d'une transition entrante et d'une transition sortante. Il est possible d'utiliser des priorités car elles sont gérées par les outils existants d'analyse, il faut donc rendre t_{exc} prioritaire sur toutes les transitions entrantes. Ainsi s'il y a conflit, t_{exc} sera toujours tirée avant une transition entrante et respectera dès lors ce qu'il se passe dans l'implémentation.

Il faut, de plus, rendre toutes les transitions servant à la purge de la MP (i.e. les t_{ei} et *fin* t_e) prioritaires sur les transitions entrantes car si t_{exc} est en conflit avec une transition entrante, ces transitions sont nécessairement, elles aussi, en conflit avec la transition entrante puisqu'elles ont un intervalle temporel égal à [0,0].

Dans notre modélisation de l'activité d'une macroplace (cf. §3.2.3), la place *MP active* est vidée lors du tir de la transition *fin* t_e. Ainsi, en cas de tir simultané, il y a bien désactivation puis réactivation de la MP lorsque la transition entrante est franchie. La transition *mp desactive* ne peut pas être franchie car les transitions entrantes sont prioritaires sur elle (cf. §3.2.3).

Pour être conforme à l'implémentation, il faut aussi rendre prioritaire t_{exc} sur toutes les transitions internes de la MP, ainsi que sur les transitions sorties classiques car la purge l'est par définition.

Il ne peut y avoir tir simultané d'une transition exception et d'une transition sortante classique d'une même MP car une transition exception est nécessairement en conflit avec les autres transitions sortantes.

Modèle analysable pour notre exemple

Ainsi le modèle équivalent au modèle donné figure 3.19 est décrit figure 3.33. Ce modèle est complexe graphiquement mais ce n'est pas gênant car il est obtenu de façon automatique. Ce modèle analysable permet de diminuer la complexité de l'analyse et de garantir la conformité de l'analyse avec l'implémentation. Le graphe des classes d'états de ce modèle contient 2 199 places et 3 449 transitions. La taille de ce graphe peut être comparée à celle des graphes des modèles donnés dans le tableau 3.1. La taille du graphe des classes d'états a une taille du même ordre de grandeur à celle des modèles sans MP permettant d'obtenir les plus petits graphes des classes. Ainsi l'utilisation de MP ne pénalisera pas l'analysabilité du modèle (au sens de sa complexité).

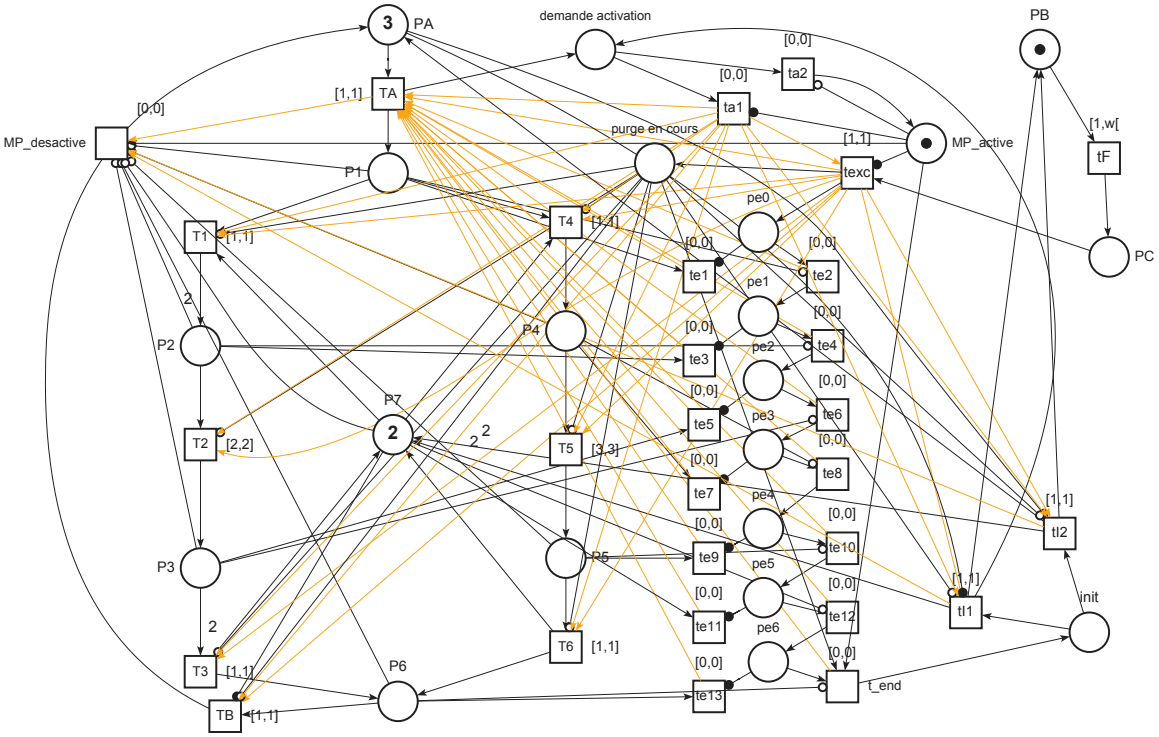

FIGURE 3.33 – Modèle obtenu pour l'analyse

Ainsi le mécanisme de la macroplace associé à celui de l'arc exception permet aux concepteurs de décrire de manière simple et efficace la gestion des exceptions au niveau comportemental. Le modèle peut être implémenté plus efficacement sur FPGA que sans macroplace et il est toujours analysable. L'utilisation de la macroplace permet d'avoir tous ces avantages tout en conservant une réactivité très importante.

3.3 Gestion des exceptions au niveau architectural

Nous pourrions envisager de gérer les exceptions au niveau architectural en utilisant la macroplace comme cela est fait au niveau comportemental pour éviter de multiplier les mécanismes mais ce n'est malheureusement pas possible. En effet, cela ne respecterait

notamment pas le principe du langage à composants. Un composant peut par exemple être réutilisé dans un autre projet ou à un autre endroit dans le modèle. Or si une macroplace est telle qu'elle contient des places de deux composants différents, ils ne peuvent plus être séparés. Un autre problème est que, par exemple, pour déterminer l'activité de la macroplace, il faut pouvoir accéder aux marquages de toutes les places du raffinement. Or les composants n'ont normalement accés qu'aux marquages des places se situant dans l'interface. Il est certes possible de mettre toutes les places de la macroplace dans l'interface, mais une fois de plus l'intérêt d'utiliser un composant est perdu. De plus, si une macroplace englobe deux composants n'évoluant pas sur la même horloge, voire sur des FPGA différents, l'évolution du raffinement devient très délicate. Par exemple, pour l'activité de la macroplace, comme l'évolution du marquage des places n'est pas synchronisée, il est difficile de s'assurer que l'information sera fiable quand elle sera utilisée par les transitions, d'autant plus que le transfert des informations entre les FPGA entraîne nécessairement un retard. Le mécanisme de la macroplace n'est donc pas utilisable au niveau architectural. Néanmoins, un mécanisme de gestion des exceptions au niveau architectural est intéressant. Commençons par étudier pourquoi ce serait intéressant.

3.3.1 Besoin au niveau architectural

Le modèle à composants présenté chapitre 1 permet de décrire l'interaction entre deux composants, d'un point de vue RdP, par fusion des places (ou transitions) ou en reliant directement les places (ou transitions) de l'interface à d'autres places (ou transitions) à l'aide d'arcs. Il est ainsi possible de gérer assez facilement le marquage des places de l'interface et les transitions de l'interface peuvent aussi permettre d'influer sur le marquage des places. Par contre, ces possibilités ne permettent pas forcément de décrire facilement les situations d'exceptions.

Le concepteur peut avoir besoin d'observer de manière pratique et efficace l'état d'un composant pour faire évoluer un autre composant. S'il a besoin d'observer uniquement l'état d'une place, il la placera dans l'interface. Mais s'il souhaite savoir s'il est dans un état spécifique décrit par de nombreuses places, il est dommage de devoir toutes les intégrer à l'interface car l'approche à composants perd alors de son intérêt. En effet, l'un des intérêts d'encapsuler le comportement dans un composant est justement de n'avoir dans l'interface que les informations nécessaires à la description des interactions. Par exemple, dans l'exemple donné figure 3.34, le composant A a besoin pour évoluer de savoir si le composant C est dans son état initial : (P_1, P_3) mais n'a pas besoin de connaître à chaque instant le marquage de chacune de ces places. L'idée serait alors que le composant observé soit capable d'envoyer au composant observateur l'information : "je suis ou non dans l'état souhaité".

Il serait possible de réaliser ces observations à l'aide du modèle HILECOP, comme illustré figure 3.35, en déterminant si la transition doit être sensibilisée dans le composant C à l'aide de la transition t_{obs} et en fusionnant la transition t_{obs} avec la transition t_{exc}. Cela permet d'exprimer la même chose de manière toute aussi réactive en limitant les éléments aux interfaces. Mais cette fusion n'est possible que dans le cadre d'une synthèse globale. Si les deux composants A et C fonctionnent avec deux horloges non synchronisées (cas d'une synthèse multi-FPGA par exemple), il serait alors impossible de réaliser cette fusion.

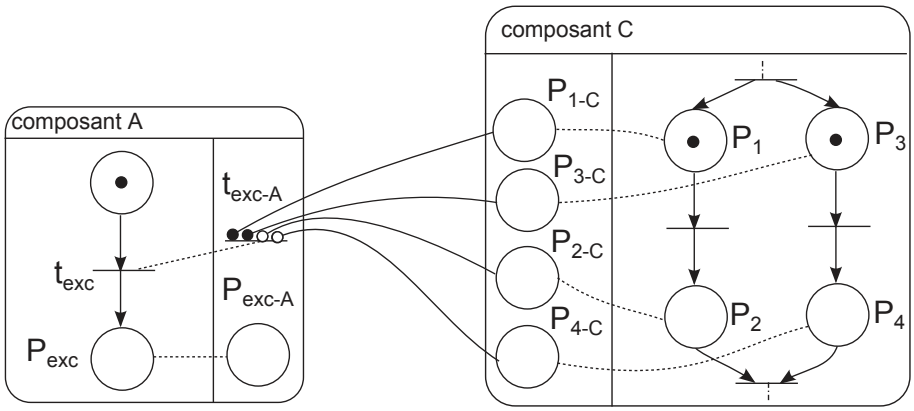

FIGURE 3.34 – Exemple d'un composant devant observer le comportement d'un autre

L'autre inconvénient est que cette solution ne respecte pas complètement l'état d'esprit des composants. En effet, la transition t_{obs} n'est pas nécessaire dans la description du comportement du composant C. Si le concepteur souhaite réutiliser le composant C indépendamment du composant A, cette transition sera en effet potentiellement inutile. Cela nuit donc à la réutilisation efficace des composants. Il est alors souhaitable que cette information sur l'état du composant puisse être fournie aux autres composants sous une autre forme que celle d'une transition (ou d'une place). De plus, en général, un composant C_1 n'est pas censé connaître en détail la description d'un autre comportement C_2. Pour respecter cet esprit, nous choisissons donc que les situations observables par un composant ne peuvent être que des situations ne nécessitant pas la connaissance de la description du comportement observé. Ainsi, les situations facilement observables sont :

- le composant est dans sa situation initiale,
- le composant n'a aucune place marquée.

Il serait aussi possible de définir une situation "spécifique" permettant d'augmenter les possibilités de modélisation. Il serait alors possible, par exemple, de changer le mode de fonctionnement d'un composant. Le composant contrôleur connaîtrait, dans cet exemple, juste la possibilité de mettre le composant dans un mode A ou B. Nous ne considérerons pas dans ces travaux ces situations spécifiques.

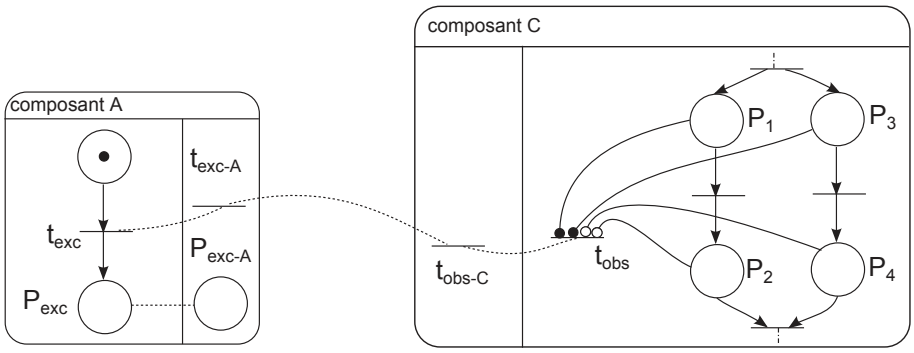

FIGURE 3.35 – Deuxième solution permettant de modéliser l'observation d'un composant

Supposons maintenant que la détection d'une situation dans un composant doive entraîner la réinitialisation du marquage d'un autre composant. Nous considérons bien ici la réinitialisation de l'état donc les variables VHDL ne doivent pas être modifiées (sinon il

est possible de faire un *reset* du composant). Par exemple, sur le modèle donné figure 3.36, quand la transition t_{exc} est tirée, le marquage du composant B doit être réinitialisé. Dans ce cas, il est nécessaire d'intégrer toutes les places du composant B à son interface. Or dans ce cas le marquage exact dans lequel se trouve le composant B a peu d'importance, ce qui importe est qu'il soit réinitialisé. Et même plus, le composant C n'a pas besoin de connaître le marquage initial des composants qu'il souhaite réinitialiser. Il serait donc intéressant d'envoyer seulement l'ordre de réinitialisation. Il serait aussi pratique de pouvoir ordonner à un composant de vider toutes ses places (i.e. que le marquage de toutes les places soit nul). De même que pour l'observation, un composant ne peut pas imposer un marquage à un autre composant qui ne soit pas son marquage initial ou le marquage nul car il ne connaît pas sa description comportementale.

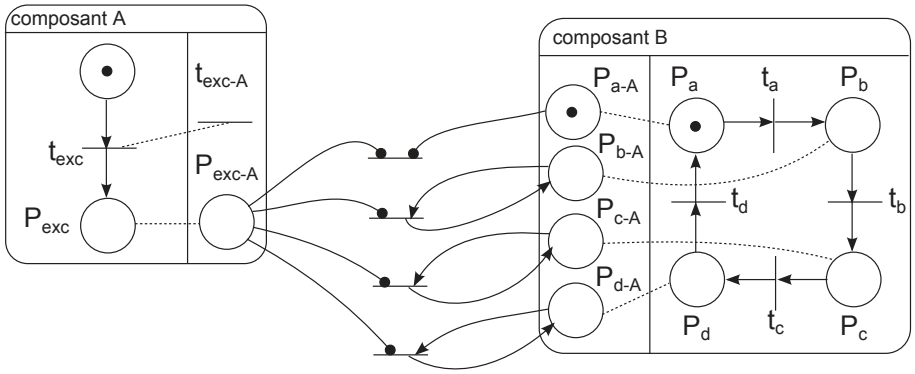

FIGURE 3.36 – Exemple d'un composant contrôlant le marquage d'un autre composant

Pour conclure, d'un point de vue architectural, pour préserver tout l'intérêt de l'approche à composants, il serait donc intéressant d'avoir aussi la possibilité de décrire de manière plus efficace et plus simple une interaction entre les composants de la forme : un composant observe l'état global d'un composant et un composant agit sur l'état global d'un composant (et non seulement le marquage des places de son interface).

3.3.2 Solution proposée pour la synthèse globale

Nous venons d'évoquer qu'il serait souhaitable qu'un composant puisse savoir si un autre composant est dans un état spécifique et/ou forcer le marquage d'un autre composant. Pour cela, il faut donc modifier l'interface des composants. L'idée est alors d'ajouter deux types de port à l'interface : le port d'observation et le port de contrôle.

Modélisation des ports d'observation et de contrôle

Pour modéliser l'observation et le contrôle et pouvoir facilement les distinguer des autres ports de l'interface, les ports d'observation et de contrôle sont placés au-dessus du composant (cf. 3.37). Le port d'observation est représenté par un triangle vide. Le triangle est dirigé vers l'extérieur du composant observé et vers l'intérieur du composant dans le composant observateur. Le port de contrôle utilise le même principe mais est représenté par un triangle plein.

Pour décrire les situations qui doivent être observées, la situation (*init*) ou (*vide*) est associée à l'arc décrivant l'observation.

Au niveau du contrôle, la purge, pour être cohérent et homogène avec le formalisme de la MP, sera décrit par (∗). La réinitialisation sera décrite par (*init*).

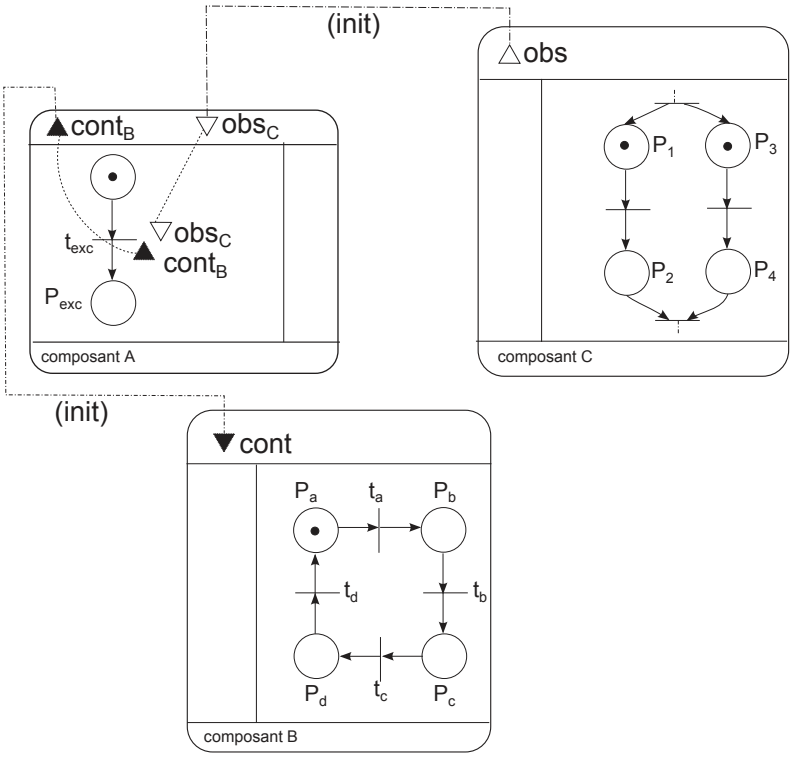

FIGURE 3.37 – Exemple d'utilisation des ports d'observation et de contrôle

L'observation d'un composant peut être utilisée comme condition pour le tir d'une transition interne du composant observateur. Le contrôle d'un composant est déclenché par le tir d'une transition du composant qui le contrôle (cf. figure 3.37).

Solution possible pour l'implémentation

Une solution d'implémentation de l'observation et du contrôle est présentée dans le cadre d'une synthèse globale. Pour l'observation, le composant observateur doit pouvoir savoir si le composant observé est dans un situation spécifique ou non. L'information envoyée par l'observé à l'observateur peut donc être contenue dans un seul signal binaire. Par contre, il faut que le composant observé connaisse la situation qui intéresse le composant observateur (initiale ou vide). Mais cette information est constante, il n'y aura donc pas besoin de l'intégrer à l'interface des composants VHDL au niveau implémentation. Pour déterminer la valeur du signal d'observation, il est possible d'intégrer un process au composant observé qui calculera ce signal. Le marquage de chacune des places du composant observé sera comparé au marquage surveillé par l'observateur et si tous les tests sont vrais, le signal d'observation sera vrai sinon il sera faux.

Dans tous les cas pour l'observation de l'état (*vide*) ou dans le cas des RdP binaire pour l'état (*init*), il est possible d'utiliser les signaux d'activation des places pour réaliser les tests. Par contre, dans le cadre d'un RdP généralisé, pour l'état (*init*), il sera nécessaire de sortir le marquage des places des composants place VHDL. (Le même principe serait retenu pour l'observation de marquages "spécifiques".)

Le process décrivant le signal d'observation dans l'exemple donné figure 3.37 est donné figure 3.38.

```
-- calcul signal d'observation du composant C
process ( markup_P1, markup_P2, markup_P3, markup_P4, reset_n )
begin
        if ((markup_P1 >= 1) and (markup_P2 = 0) and (markup_P3 >= 1) and (markup_P4 = 0)
            )
                then
                        obs_c <= '1';
        else
                 obs_c <= '0';
        end if;
end process;
```

FIGURE 3.38 – Process pour le calcul d'un signal d'observation

Pour le contrôle, l'avantage est qu'il est déjà possible de réinitialiser une place à l'aide du signal de réinitialisation des places ou de vider une place à l'aide du signal de purge (*clear*) des places ajouté pour l'implémentation de la MP. Ainsi comme il est considéré que le contrôle se limite à ces 2 situations, le composant place n'a pas besoin d'être modifié. Il faut alors utiliser un process qui calcule le signal de reset et de purge des places et des transitions du composant. Il s'agit de faire un OU entre tous les signaux pouvant entraîner un reset ou un clear.

Solution possible pour l'obtention d'un modèle analysable

Il est toujours supposé que la synthèse réalisée est une synthèse globale. Pour l'analyse, rappelons que le modèle est nécessairement remis à plat, les composants disparaissent donc. Il faut alors traduire de manière structurelle l'observation et le contrôle des composants.

Pour l'observation d'une situation précise, une place est ajoutée. Elle sera marquée que si la situation observée est vraie (cf. figure 3.39). Ainsi elle a en arcs amonts, les arcs décrits dans la situation observée (situation initiale ou composant vide dans notre cas). Toutes les transitions utilisant ce signal d'observation sont reliées à cette place par un arc test. Cette place est vidée toutes les unités de temps pour pouvoir réévaluer la valeur du signal d'observation. La transition qui vide la place est la moins prioritaire. Le modèle d'analyse obtenu pour l'observation du composant C dans le cadre de la figure 3.37 est donnée figure 3.39.

Pour le contrôle, la purge est modélisée comme la purge réalisée dans le cadre de la macroplace (cf. chapitre 3). Pour modéliser la réinitialisation ou le forçage d'un marquage, les places sont d'abord purgées en temps nul puis les jetons souhaités sont alors ajoutés toujours en temps nul (i.e. avec des transitions dont l'intervalle temporel est égal à $[0, 0]$). Le modèle obtenu pour le contrôle du composant B est donné figure 3.40.

Ainsi il est possible de modéliser l'observation ou le contrôle d'un composant par un autre composant. Cette modélisation peut être implémentée et analysée dans le cadre d'une synthèse globale.

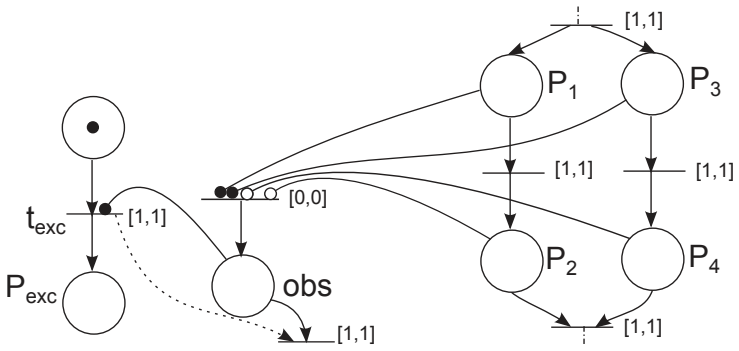

FIGURE 3.39 – Modèle analysable traduisant l'observation du composant C

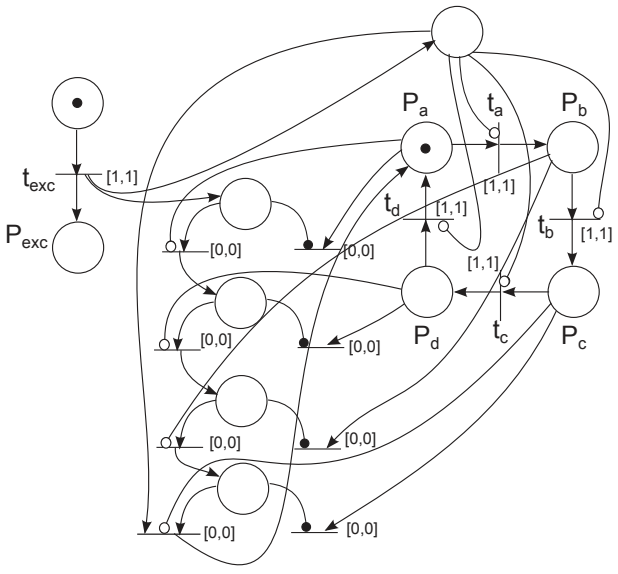

FIGURE 3.40 – Modèle analysable traduisant le contrôle d'un composant B par le composant A

Nous avons donc pu voir qu'il était nécessaire d'avoir un mécanisme efficace et pratique pour la gestion des exceptions. Un mécanisme de gestion des exceptions au niveau comportemental a été proposé. Des pistes pour pouvoir observer et contrôler l'état d'un composant ont été mentionnées et cette réflexion est poursuivie dans les perspectives de ce manuscrit. Le mécanisme de gestion semble permettre de non seulement simplifier le travail du concepteur mais aussi d'augmenter la réactivité du système en cas d'exception tout en conservant une implémentation efficace sur FPGA et la possibilité d'analyser le modèle. Nous allons maintenant utiliser ce mécanisme de gestion comportemental sur des vrais modèles industriels afin de s'assurer des apports réels apportés par ce mécanisme.

Chapitre 4

Application sur un cas industriel critique

Il est important de tester notre mécanisme de gestion des exceptions sur des modèles conçus dans le cadre de systèmes industriels. En effet, seul un test sur des cas industriels permettra de s'assurer que ce mécanisme respecte bien la contrainte d'être adapté au contexte des systèmes numériques embarqués critiques, notamment au regard du facteur d'échelle. Le but est d'observer l'impact de la macroplace sur la conception mais aussi sur l'implémentation du modèle et son analyse dans le cadre d'un exemple de modèle industriel.

Le domaine industriel critique choisi est celui de la conception de neuroprothèses dans le cadre de la stimulation électro-fonctionnelle implantée. Nous avons travaillé sur des modèles conçus en collaboration par l'équipe de recherche DEMAR et l'entreprise Axonic - MXM. Le principe de la stimulation fonctionnelle et le fonctionnement attendu du système étudié sera d'abord présenté. L'impact de la macroplace sera ensuite observé sur deux générations différentes du système étudié. En effet, le système et surtout la gestion des exceptions a évolué de manière notable entre les deux générations d'implants considérées.

4.1 Description du système industriel étudié

De nombreuses déficiences motrices ou sensitives, plus ou moins handicapantes, proviennent d'une atteinte ou déficience du système nerveux. Plusieurs dispositifs médicaux implantables ont été développés par diverses sociétés et/ou laboratoires de recherche pour restaurer ces fonctions motrices ou sensorielles. Le pacemaker, pour les problèmes cardiaques, ou l'implant cochléaire, pour résoudre le problème de la surdité profonde sont des exemples d'implants désormais connus du grand public. Mais ces dispositifs ne permettent généralement de restaurer qu'une unique fonction et sont développés pour une pathologie précise.

L'équipe de recherche DEMAR a notamment pour but de développer une architecture de stimulation neurale implantable générique, au sens exploitable pour différentes applications fonctionnelles. Pour comprendre les contraintes de ce contexte, nous commencerons par présenter brièvement la stimulation électro-fonctionnelle.

4.1.1 Principes fondamentaux de la stimulation électro-fonctionnelle

La stimulation électro-fonctionnelle (SEF) consiste à appliquer un stimulus électrique à un tissu nerveux (stimulation neurale) ou à un muscle (stimulation épimysiale) afin de reproduire artificiellement l'activation de celui-ci [10]. Nous nous intéresserons plus particulièrement à la stimulation neurale. Pour comprendre le principe de la SEF neurale, rappelons d'abord le fonctionnement du système nerveux.

Le système nerveux humain est responsable du contrôle et des actions du corps. Il est composé de deux parties (cf. figure 4.1) : le système nerveux central, composé du cerveau, du cervelet et de la moelle épinière, et le système nerveux périphérique, composé des nerfs.

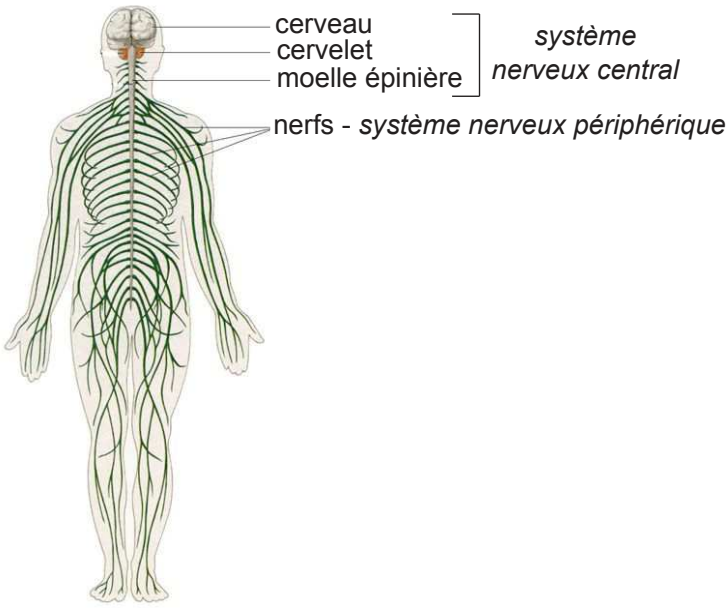

FIGURE 4.1 – Représentation du système nerveux

Deux types de fonctions sont distinguées :

- les fonctions conscientes pour lesquelles une commande volontaire est émise par le cerveau. Cet ordre circule dans la moelle épinière et les nerfs moteurs pour arriver aux muscles. Attraper un objet avec la main est un exemple de fonction consciente.
- les fonctions réflexes pour lesquelles la réaction motrice provient de la réponse des cellules nerveuses à des stimuli sensitifs. Ces fonctions sont innées. Les cellules nerveuses qui les engendrent peuvent être contenues dans la moelle épinière. Le fait de retirer sa main quand un objet brûlant est touché est un exemple de fonction réflexe.

Le système nerveux est essentiellement constitué de neurones. Ils traitent et transmettent l'information aux différents organes du corps. Un neurone est constitué d'un corps cellulaire, de dendrites et d'un ou plusieurs axones (cf. figure 4.2). L'information est collectée par les dendrites et propagée et transmise par les axones. Les dendrites et axones sont appelés fibres. Le long des fibres, l'information se propage sous forme d'un signal électrique appelé potentiel d'action.

La substance grise du système nerveux central est constituée du corps des cellules nerveuses. Au contraire, la substance blanche du système nerveux central et le système

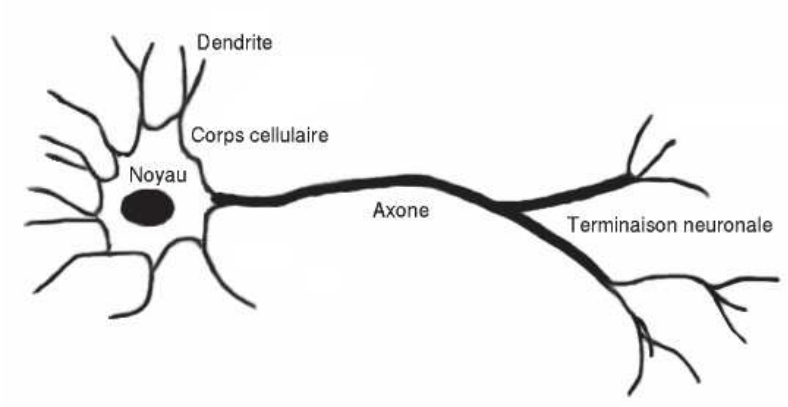

FIGURE 4.2 – Schéma d'un neurone

nerveux périphérique sont constitués de fibres nerveuses qui constitueront les nerfs en périphérie.

Deux sens de propagation de l'information sont observés dans les nerfs. Deux types de fibres sont alors distinguées :

- les fibres efférentes (motrices) qui transmettent les informations venant du système nerveux central vers les muscles et les organes.
- les fibres afférentes (sensitives) qui transmettent les informations sensorielles venant d'une cellule vers le système nerveux central.

Un même nerf peut être constitué de fibres afférentes et efférentes. Il peut, de plus, avoir des fibres motrices qui innervent plusieurs muscles par exemple.

Le système nerveux peut être atteint de différentes pathologies : la paraplégie ou la tétraplégie due à une lésion de la moelle épinière, l'hémiplégie due à un accident vasculaire cérébral, les déficiences sensorielles comme la surdité ou l'amblyopie (malvoyance) ou les maladies dégénératives (maladie de Parkinson). Ces pathologies entraînent l'incapacité pour les patients de réaliser certaines fonctions. Dans le cas de la paraplégie, par exemple, il y a le plus souvent lésion partielle ou totale de la moelle épinière au niveau lombaire ou thoracique. Les ordres du cerveau ne peuvent alors plus atteindre les muscles qui sont contrôlés par les nerfs sous la lésion et le patient est alors notamment paralysé (totalement ou partiellement) des membres inférieurs.

Le principe de la SEF est alors de remplacer les impulsions électriques physiologiques qui ne peuvent plus être émises par les cellules nerveuses par une stimulation électrique artificielle au niveau neural ou épimysial. Le but de cette stimulation est de tenter de suppléer la perte des fonctions liée à la déficience du système nerveux. Il y a trois modalités de réalisation d'une SEF :

- stimulation de surface externe. Cette méthode consiste à positionner une électrode de surface sur la peau au niveau du muscle cible. Le problème d'une utilisation externe est qu'il faut envoyer un fort courant pour obtenir la contraction du muscle. Il faut aussi placer l'électrode très précisément et même en faisant cela, les résultats ne sont pas très reproductibles. L'avantage majeur est que cette méthode ne nécessite pas d'opérer le patient.
- stimulation épimysiale. Cette méthode consiste à positionner l'électrode directement sur les fibres musculaires en intracorporel. Ceci permet de stimuler un muscle avec

environ dix fois moins d'énergie que dans le cas de la stimulation de surface mais il nécessite une partie implantée et donc une opération.
- stimulation neurale. Dans cette méthode, ce n'est plus directement le muscle qui est stimulé mais le nerf qui le contrôle. En stimulant le nerf et en se servant de la réponse naturelle du muscle, la même contraction est obtenue qu'avec la méthode épimysiale mais avec environ dix fois moins d'énergie. La réponse du muscle est aussi plus reproductible. La contrepartie est qu'il faut réussir à stimuler sélectivement les axones activant le(s) muscle(s) visé(s) (et parvenir à recruter différemment leurs fibres lentes, moins fatigables, et rapides).

Pour réaliser la stimulation, il est donc nécessaire de posséder une électrode mais aussi un stimulateur qui va se charger de produire le courant électrique. C'est à ce stimulateur, dans le cadre de la stimulation neurale, que nous allons plus particulièrement nous intéresser. Le stimulateur est donc un dispositif médical implantable (DMI).

Pour être capable de suppléer une fonction qui nécessite plusieurs muscles (contrôler un membre et ses articulations par exemple), il faut prévoir plusieurs sites de stimulation (éventuellement un par muscle devant être sollicité). Deux solutions sont envisageables. La première est de concentrer l'électronique sur un seul implant, auquel cas l'architecture est dite centralisée. Mais alors une multitude de fils doivent partir de l'implant central (un fil pour chaque pôle de chaque électrode) et la chirurgie nécessaire pour implanter le système est lourde et dangereuse, ce qui limite fortement son utilisation. La seconde solution est d'avoir une architecture composée de plusieurs implants. Comme nous souhaitons pouvoir réaliser le contrôle d'électrodes multi-polaires sur plusieurs sites, c'est cette solution qui sera considérée (cf. figure 4.3).

Une autre contrainte forte imposée par la SEF est la précision temporelle des ordres de stimulation qui doivent être donnés, d'autant plus si plusieurs sites de stimulation doivent être synchronisés.

L'architecture de stimulation implantable (cf. figure 4.3) a alors été définie sur la base d'unités de stimulation indépendantes, appelées unités de stimulation réparties (USR) qui embarquent l'électronique nécessaire au contrôle et à la génération du profil de stimulation. Ces unités sont reliées ensemble sur un réseau. Pour plus de détails sur l'architecture du système, le lecteur est invité à se reporter à [5] ou [49]. Nous nous intéresserons ici plus particulièrement au fonctionnement de l'USR.

4.1.2 Description de l'unité de stimulation répartie

L'unité de stimulation répartie (USR) a pour but de générer les profils de stimulation appliqués à travers l'électrode, à partir des ordres fournis par le contrôleur SEF (i.e. pilotant la réalisation des fonctions) (cf. figure 4.3). Elle doit donc communiquer sur le médium, générer et contrôler le profil de stimulation, configurer l'électrode qui lui est associée (i.e. ses pôles actifs) et rester contrôlable à distance. Elle se charge aussi de la surveillance, que ce soit de la génération du stimulus ou des ordres que le contrôleur lui envoie. Elle est composée d'une partie analogique et d'une partie numérique et est constituée des éléments suivants (cf. figure 4.4) [5] :
- Un générateur de stimulus chargé de générer le signal appliqué au nerf via l'électrode.

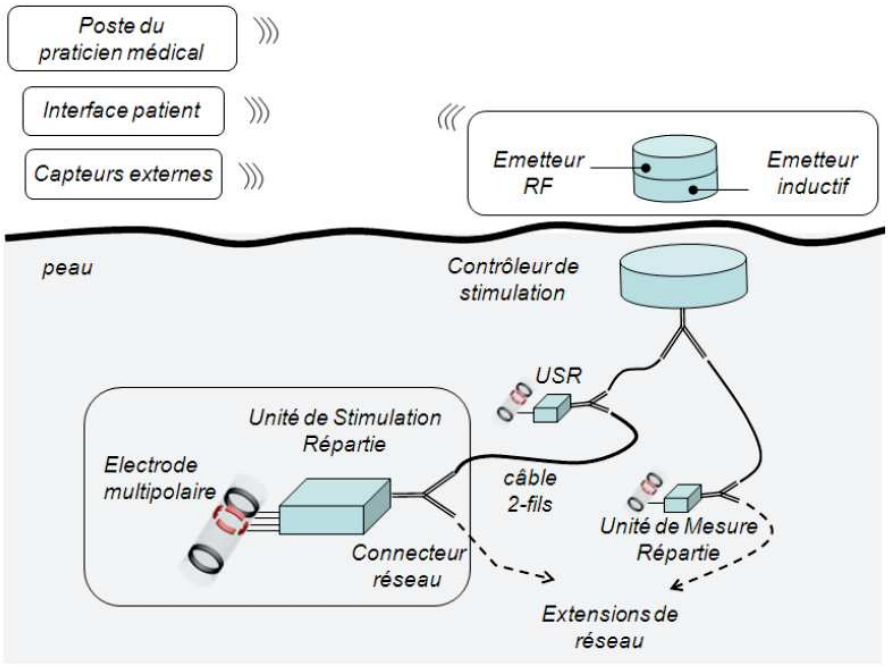

FIGURE 4.3 – Architecture de stimulation neurale [49]

Il est réalisé sur une puce spécifique (ASIC analogique).

- Le reste des éléments sont numériques et implémentés sur un FPGA :
  - Une micro-machine qui est l'exécuteur des profils de stimulation enregistrés sous forme de micro-programmes, ainsi qu'un dispositif de surveillance de la stimulation basé sur des modèles de référence.
  - Une pile protocolaire gérant la communication de l'USR au sein de l'architecture.

Comme la stratégie est de placer les USR au plus près possible de l'électrode (cf. §4.1.1), leur encombrement doit être très faible (pour pouvoir être facilement implantées). Comme elles sont alimentées par le réseau, elles doivent aussi avoir une consommation très faible.

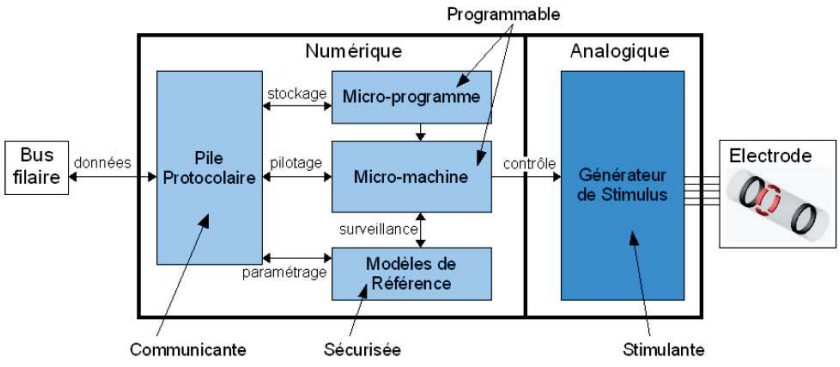

FIGURE 4.4 – Représentation schématique d'une unité de stimulation répartie [49]

Décrivons maintenant, de manière plus détaillée, les différentes surveillances que doit réaliser l'USR pour garantir la sécurité du patient implanté. C'est en effet notamment pour décrire ces surveillances que le mécanisme de gestion des exceptions sera particulièrement intéressant. Quel que soit le type d'erreur surveillée, la surveillance est toujours composée, à ce stade, de trois phases :

- La détection du non-respect de contraintes de sécurité ou d'erreurs.
- La réaction, équivalente à une mise en état de repli sûr.
- La notification et l'attente de réarmement (confirmation et déblocage).

Il n'y a pas de diagnostic au niveau d'une USR. Les erreurs signalées sont simplement transmises au contrôleur via des codes différents pour les erreurs détectables. Le diagnostic est laissé à la charge du contrôleur, voire de l'opérateur, qui a une vision plus globale. C'est plus précisément dans la phase de réaction que la gestion des exceptions est particulièrement pertinente.

Trois types d'erreurs sont à surveiller dans l'USR :

- micro-programme à exécuter non conforme : il faut assurer que lors de l'exécution aucune instruction incorrecte ne se trouve dans le micro-programme (exemple : instruction inconnue, programme qui se termine sans boucle ...). Cette sécurité est assurée par la micro-machine.
- ordres donnés incohérents : il faut assurer que le contrôleur ne puisse pas donner d'ordres incohérents à la micro-machine (exemple : lancer une stimulation alors qu'aucun micro-programme n'a été transmis). Cette surveillance est assurée par la couche application de la pile protocolaire.
- stimulation dangereuse : il faut assurer qu'un profil de stimulation ne puisse pas être dangereux pour le nerf cible (exemple : envoyer un courant trop important ou stimuler pendant trop longtemps). Cette surveillance est assurée par les modèles de référence.

Pour évaluer l'impact du mécanisme de gestion d'exceptions, nous avons choisi de nous intéresser plus particulièrement au modèle de la micro-machine puisqu'elle doit gérer de nombreuses exceptions. La micro-machine est une sorte de micro-processeur avec un jeu d'instructions extrêmement réduit et spécifique à la SEF. C'est l'équivalent d'un micro-contrôleur spécialisé pour la stimulation au sein duquel les opérations sont réalisées en parallèle. La micro-machine se charge d'exécuter localement un profil de stimulation décrit sous la forme d'un micro-programme. Ce dernier est envoyé par le contrôleur de simulation et comprend aussi bien le stimulus à réaliser que la répartition de courant entre les pôles actifs de l'électrode (multi-polaire).

La micro-machine fonctionne selon l'exemple suivant :

- le contrôleur envoie un ordre de début de stimulation (le micro-programme ayant été préalablement enregistré),
- la micro-machine commence à exécuter le micro-programme pour réaliser la stimulation correspondante,
- la micro-machine ne s'arrête que lorsque le micro-programme est terminé, qu'elle reçoit un nouvel ordre en provenance du contrôleur ou qu'une erreur est détectée.

Plusieurs générations de stimulateurs et donc de micro-machines ont été réalisées par l'équipe DEMAR. Nous étudierons l'impact de la gestion des exceptions sur la première

génération de micro-machine et sur la troisième. La gestion des exceptions réalisées par les concepteurs (sans mécanisme spécifique de gestion d'exception) est en effet abordée de manière assez différente dans ces deux modèles. Il est donc intéressant de comparer les résultats obtenus sur ceux-ci, bien que le principe de fonctionnement normal reste relativement proche. Le comportement précis de chacune des micro-machines sera détaillé dans le §4.2.1 et §4.3.1. Commençons par nous intéresser à la micro-machine de première génération.

4.2 Application sur la micro-machine de première génération

La micro-machine de première génération a été utilisée dans le stimulateur appelé Stim3d. Son fonctionnement ainsi que celui du stimulateur en général sont présentés de manière détaillée dans [49].

4.2.1 Comportement attendu

Le principe de la micro-machine est relativement simple, il est similaire au fonctionnement d'un processeur : exécuter le micro-programme, instruction par instruction et envoyer au générateur de stimulus les commandes correspondantes. Pour cela, l'exécution se base sur un compteur ordinal (CO). Les instructions de boucle sont traitées en parallèle (en temps masqué) de l'instruction de stimulation en cours. L'enchaînement des instructions est également effectué en pipe-line, i.e. l'instruction suivante est décodée pendant l'exécution de l'instruction courante.

7 instructions différentes peuvent être utilisées dans les micro-programmes : MT, RT_MT, MIT, RT_MIT, RP_MIT, RT_RP_MIT et LOOP. Ces instructions peuvent être classées suivant 3 types : stimulante (MIT), non-stimulante (aussi dit décharge) (MT) et répétition du profil (LOOP). Deux types de répétition sont possibles, soit un nombre fixe de répétitions (et donc la stimulation s'arrête une fois le nombre de répétitions atteint), soit la stimulation tourne en boucle jusqu'à ce que le contrôleur demande l'arrêt de la stimulation. RT signifie que les paramètres de la stimulation sont modifiés en temps-réel et RP que la stimulation doit être réalisée sous la forme d'une rampe. La manière de coder ces instructions ne sera pas détaillée ici [5].

L'USR possède deux mémoires où elle peut stocker, dans chacune, un micro-programme. Le démarrage de la micro-machine est géré par la couche application lors de la réception d'une requête correspondante venant du contrôleur. Le pilotage de la micro-machine peut se faire par les instructions suivantes :

- Start : démarrage de l'exécution d'un micro-programme en indiquant dans quelle mémoire il se situe.
- Stop : arrêt du micro-programme en cours d'exécution. Notons que l'arrêt peut aussi être réalisé par la micro-machine lors du décodage de l'instruction de fin (dans le cas d'un profil à répétition limitée).
- Commute : ceci correspond à une demande de changement du micro-programme qui est en cours d'exécution. Ce changement doit se faire sans interrompre le profil en

cours avant sa fin, i.e. la commutation interviendra lors de la prochaine instruction de boucle (i.e. de répétition).

La micro-machine intègre également la détection d'erreurs simples au sein du micro-programme. Les erreurs détectées sont :

- La ligne de programme à déchiffrer est vide.
- L'adresse de l'instruction à déchiffrer est incorrecte (hors de la zone mémoire dédiée au micro-programme).

Ces situations sont immédiatement détectées, i.e. lors du traitement de l'instruction concernée, et donnent lieu à une réaction immédiate. La micro-machine est mise dans un état sûr : il s'agit d'un "verrouillage" de la micro-machine après désactivation de tout stimulus (i.e. après la mise en état de sécurité de l'étage analogique). La latence dite "latence de réaction" est comprise entre 4 et 7 μs (pour une horloge de 1MHz). La micro-machine n'évolue alors plus tant que le contrôleur central n'a pas demandé son réarmement. Le réarmement ne signifie pas le redémarrage de la micro-machine, mais simplement la mise dans son état initial à partir duquel elle peut, sur requête du contrôleur, engager à nouveau l'exécution d'un micro-programme. En même temps que le verrouillage, l'USR signale une erreur dont le code précise la nature. Cette erreur est notifiée au contrôleur par le module de communication.

Le modèle de la micro-machine est aussi étroitement lié aux modèles de référence : chaque exécution d'instruction est "propagée" au niveau des modèles de référence. Ces modèles, un par pôle de l'électrode, permet de s'assurer que les contraintes imposées sur la stimulation sont bien respectées. Ces contraintes, principalement temporelles, garantissent que les tissus nerveux des patients stimulés et les contacts (pôles) de l'électrode ne seront pas endommagés. En cas d'erreur, il faut arrêter la stimulation, les modèles de référence préviennent donc la micro-machine. C'est pourquoi les composants décrivant les modèles de référence sont intégrés au modèle de la micro-machine (cf. figure 4.5).

Le modèle RdP permettant de décrire le comportement attendu de la micro-machine est donné figure 4.5. Seule la partie gestion des exceptions sera expliquée en détail (cf. §4.2.2). Définissons néanmoins le rôle de chaque partie identifiée sur le RdP pour faciliter la compréhension globale du modèle. Les modèles de ces différentes parties sont fournis dans l'annexe C. A noter que dans cette version du logiciel HILECOP, les arcs tests sont représentés par des arcs ayant une flèche à chaque extrémité et les arcs inhibiteurs par des arcs avec un rond plein. C'est une autre convention que celle utilisée dans le reste des travaux présentés et qui depuis a été actualisée.

1. exécution des instructions stimulantes (MIT) et non-stimulantes (MT)
2. exécution des instructions de boucle et gestions des requêtes Commute envoyées par le contrôleur
3. décodage des instructions MIT
4. décodage des instructions MT
5. gestion de l'exécution des instructions MT
6. gestion d'erreurs détectées en interne (non-validité du micro-programme) ou par les

modèles de référence (non respect des contraintes de sécurité). Retour de la micro-machine dans un état sûr dans le cas des erreurs détectées en interne

7. gestion des requêtes Start envoyées par le contrôleur
8. gestion des requêtes Stop envoyées par le contrôleur et retour de la micro-machine dans son état initial
9. retour de la micro-machine dans son état sûr après la détection d'une erreur par les modèles de référence
10. modèles de référence (qui sont décrits dans des composants)
11. génération, lors d'une décharge (MT), de deux signaux de commande du générateur de stimuli selon une horloge à 10 kHz
12. dialogue avec le bloc de communication en ce qui concerne les requêtes reçues par la micro-machine

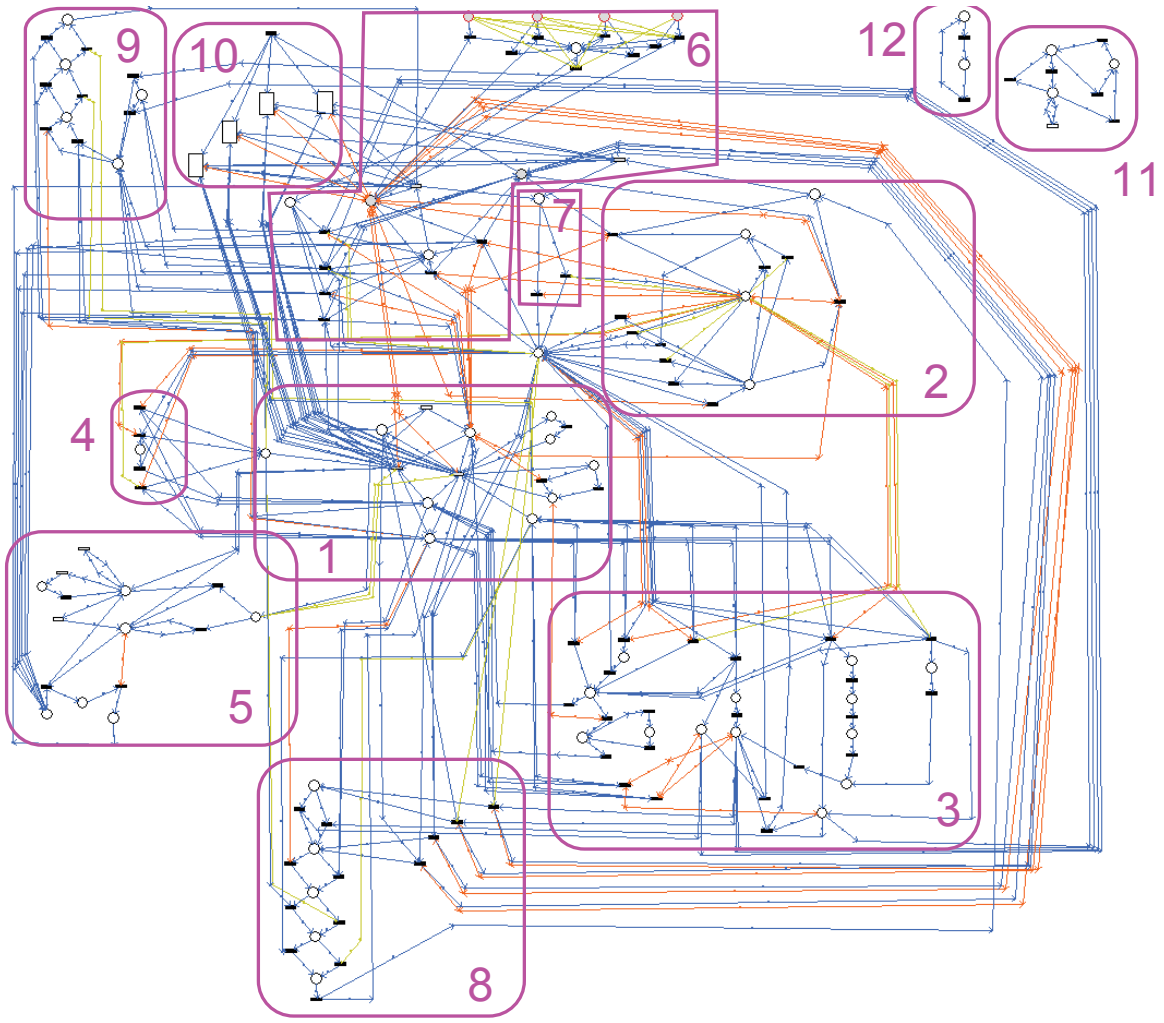

FIGURE 4.5 – RdP complet de la micro-machine 1ère génération [49]

Intéressons-nous maintenant plus spécifiquement à la manière de gérer les exceptions dans ce modèle, c'est-à-dire aux parties 6, 8 et 9.

4.2.2 Gestion des exceptions sans macroplace

Trois types d'exceptions doivent être gérés dans la micro-machine : les erreurs de codes, les erreurs détectées par les modèles de référence, appelée erreurs de stimulation, et les stops. En cas de stop, la micro-machine doit être remise dans son état initial et pourra redémarrer à la prochaine requête de start du contrôleur. En revanche, en cas d'erreur, la micro-machine est remise dans un état sûr proche de l'état initial et une décharge est nécessairement lancée. La différence entre l'état sûr et l'état initial est que la micro-machine située dans un état sûr doit être "réarmée" par un ordre du contrôleur avant de pouvoir redémarrer afin de s'assurer que ce dernier a bien pris en compte le problème. Ainsi, dans l'état sûr de la micro-machine, la place Off et la place $unicite\_erreur$ ne sont pas marquées contrairement à l'état intial. En revanche, la place att_rearm est marquée (cf. figure 4.6). Le tir de la transition $rearm$ permettra de remettre la micro-machine dans son état initial.

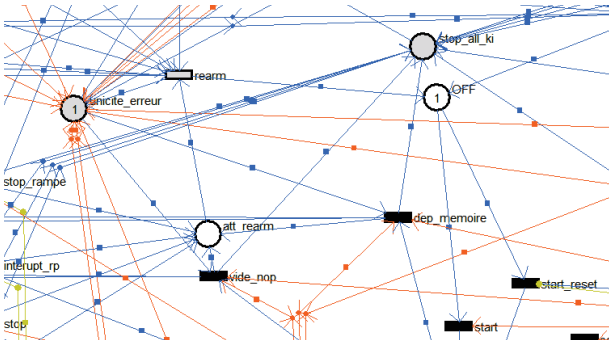

FIGURE 4.6 – Zoom sur le passage de l'état sûr à l'état initial (marquage initial donné)

gestion des erreurs de code : Deux types d'erreurs de code sont détectées dans ce modèle : ligne de programme à décoder vide et dépassement de la zone mémoire allouée. Pour gérer ces deux erreurs deux transitions, situées dans la partie 6, sont utilisées (une pour chaque erreur) : $vide\_nop$ et $dep\_memoire$ (cf. figure 4.7). Deux transitions sont suffisantes car il est possible de connaître assez précisément l'état dans lequel sera la micro-machine quand ces erreurs seront détectées.

La seule incertitude est de savoir si une instruction est en train d'être exécutée ou non. Comme ce ne sont pas des erreurs mettant directement l'intégrité des nerfs du patient en jeu, car elles concernent l'instruction suivante à exécuter, il est possible d'attendre la fin de l'exécution de l'instruction en cours si nécessaire. Cette attente permet de simplifier la modélisation en n'utilisant que 2 transitions pour gérer les erreurs de code au lieu de 4. Pendant cette attente, les modèles de référence peuvent toujours détecter une erreur ainsi le patient ne risque pas d'être mis en danger. Les transitions $vide\_nop$ et $dep\_memoire$, lorsqu'elles seront tirées, retireront et ajouteront les jetons nécessaires pour mettre la micro-machine dans un état sûr.

gestion des erreurs de stimulation : La gestion des erreurs détectées par les modèles de référence est plus complexe car ces erreurs peuvent être détectées n'importe quand durant la stimulation. Et surtout, comme il s'agit là d'erreurs critiques (le nerf du

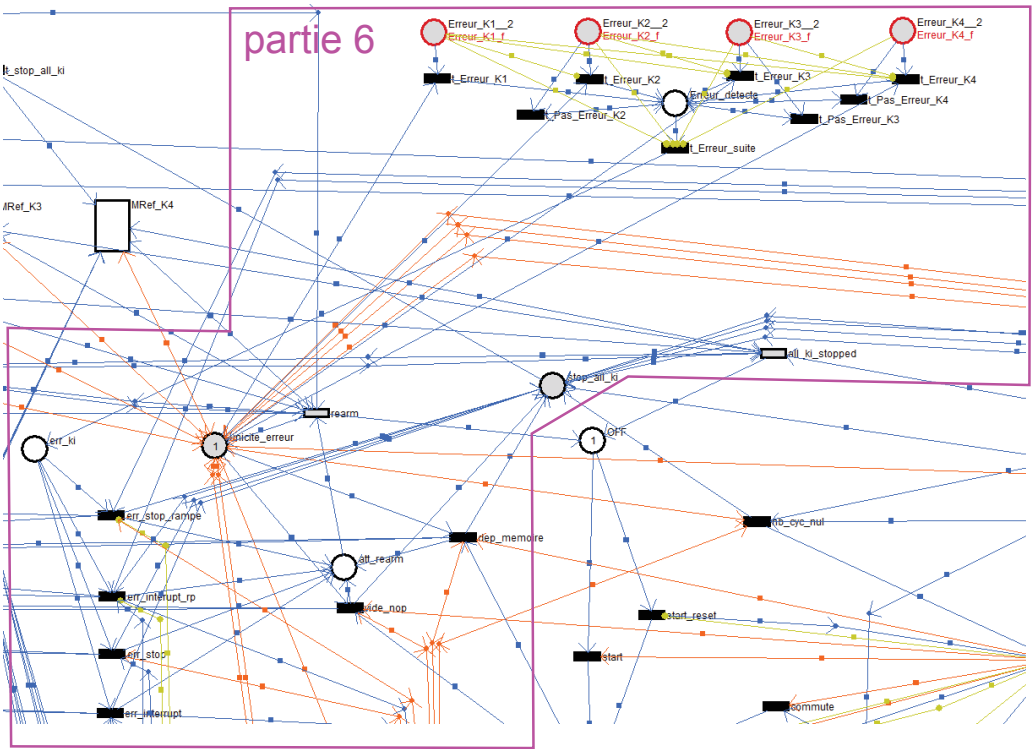

FIGURE 4.7 – Détection des erreurs et gestion des erreurs de code

patient risque d'être endommagé), il n'est pas possible d'attendre que la micro-machine revienne dans un état facilement identifiable pour traiter l'erreur.

Deux problèmes se posent pour gérer ces erreurs : gérer le fait que plusieurs erreurs peuvent être détectées en même temps et remettre la micro-machine dans un état sûr sans connaître son marquage exact. Le premier problème est géré par la partie 6 (cf. figure 4.7) et le second par la partie 9 (cf. figure 4.8).

Les transitions t_Erreur_Ki, $t\_Pas\_Erreur\_Ki$ et t_Erreur_suite (en haut à droite de la figure 4.7) permettent de n'avoir qu'une seule erreur à traiter même si plusieurs sont détectées simultanément. Les priorités nécessaires entre ces transitions d'erreurs sont gérées par des arcs inhibiteurs. En effet, la méthode de gestion des conflits à l'aide de priorités proposée dans le chapitre 2 n'est pas gérée par la version d'HILECOP utilisée pour réaliser cette génération de modèle. Une fois qu'il est certain qu'une seule erreur est prise en compte (tir de t_Erreur_suite), suivant l'état actuel de la micro-machine, une des transitions err_stop_rampe, $err\_interrupt\_rp$, err_stop ou $err\_interrupt$ (en bas à gauche de la figure 4.7) est tirée pour déclencher la mise en état sûr de la micro-machine.

La micro-machine peut, bien sûr, être dans plus que 4 états différents lorsqu'une erreur est détectée, mais il serait impossible pour le concepteur d'utiliser une transition pour chaque marquage possible. Ainsi, le concepteur commence par déterminer quelles fonctions la micro-machine est en train d'exécuter, pour avoir une indication sur son marquage. Deux principales questions se posent :

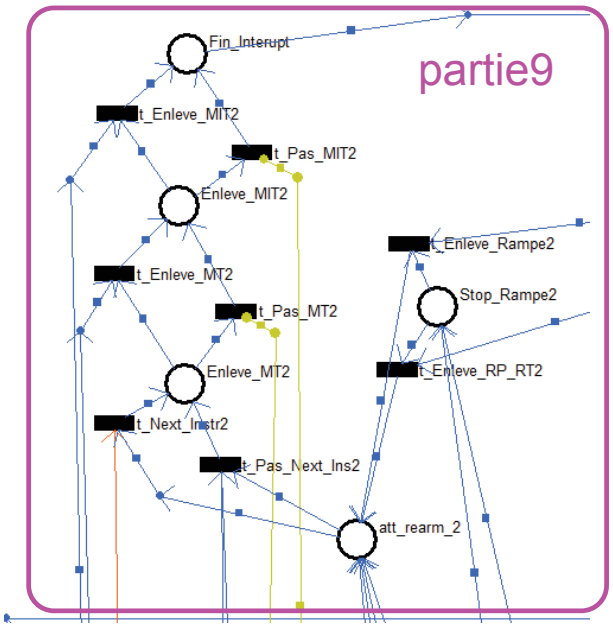

FIGURE 4.8 – Gestion de la remise de la micro-machine dans un état sûr en cas d'erreur [49]

- la micro-machine est-elle en train d'exécuter une instruction ou non ? (place *execute* ou place *mono_inst* marquée)
- la micro-machine est-elle en train de déchiffrer une rampe ou non ? (place *analys_instruct* marquée ou non)

Il y a donc 4 transitions d'erreurs pour gérer ces 4 situations possibles. Si l'instruction qui est en train d'être déchiffrée n'est pas une rampe, l'erreur ne sera traitée qu'une fois le décodage de l'instruction fini (i.e. place *analys_instruct* marquée). Cela permet d'éviter au concepteur de devoir traiter tous les cas possibles lorsqu'une instruction est en train d'être décodée. Il ne peut pas, par contre, attendre qu'une rampe soit terminée, car cela impliquerait d'exécuter de nouvelles stimulations potentiellement dangereuses pour le patient. De même, il ne peut pas attendre la fin de la stimulation en cours.

Une fois l'une des transitions d'erreurs franchie, la partie de RdP contenue dans la partie 9 (cf. figure 4.8) est exécutée. Elle permet de mettre la micro-machine dans son état sûr. Par exemple, si un jeton est présent dans la place MIT, il est retiré par la transition t_enleve_MIT. S'il n'y en a pas, c'est la transition t_pas_MIT qui est tirée. Les places devant être gérées le sont de manière séquentielle.

gestion des stops : Les stops sont gérés exactement de la même manière que les erreurs de stimulation (cf. figure 4.9) car ils doivent être traités immédiatement. 4 transitions traduisent la prise en compte du stop suivant la situation de la micro-machine. La différence est qu'au lieu de mettre la micro-machine dans un état sûr, elle est remise dans son état initial. Ce retour à l'état initial est géré sur le même principe que la mise dans un état sûr pour les erreurs de stimulation.

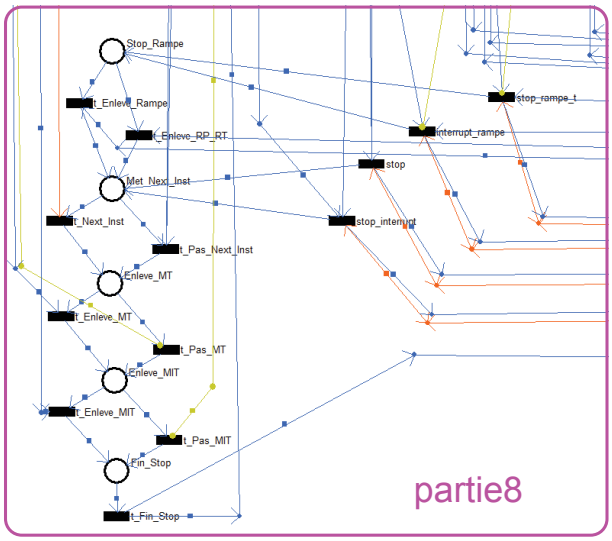

FIGURE 4.9 – Gestion de la remise dans l'état initial de la micro-machine en cas de stop [49]

4.2.3 Gestion des exceptions avec macroplace

Nous souhaitons désormais gérer les exceptions grâce à la macroplace afin de déterminer les gains apportés par celle-ci. Le fonctionnement normal de la micro-machine n'est pas modifié. La gestion du stop et des erreurs le sont mais le comportement attendu reste identique : mise dans un état sûr en cas d'erreur et remise dans l'état initial en cas de stop. Le modèle global alors obtenu en utilisant une macroplace est donné figure 4.10.

Les limites du raffinement de la MP ont été déterminées par la gestion des exceptions à effectuer. Nous savons que les erreurs à gérer se produisent pendant que la micro-machine est en fonctionnement et certaines peuvent arriver n'importe quand pendant ce fonctionnement. Le fonctionnement normal est donc placé dans la MP, c'est-à-dire les parties 1, 2, 3, 4 et une partie de la 5. En effet, les parties 6, 8 et 9 gèrent les erreurs et la partie 7 gère le démarrage de la micro-machine. La partie 10 contient des composants, or les composants ne peuvent pas être intégrés à la MP. De plus, il y a aussi des erreurs de stimulation qui sont gérées par cette partie. Pour les parties 11 et 12, elles évoluent structurellement indépendamment du modèle et ne sont donc pas impactées structurellement par la gestion des exceptions. Elles ne doivent donc pas être intégrées à la MP. Pour la partie 5, les transitions et les places qui permettent de forcer une décharge sont hors de la MP car elles sont utilisées en cas d'erreur et n'appartiennent donc pas au fonctionnement normal. Il aurait été possible de les intégrer à la MP mais cela n'aurait pas respecter l'esprit de la MP. Le reste de la partie 5, qui décrit le fonctionnement d'une décharge, est par contre intégré à la MP.

Les parties 1, 2, 3 et 4 devraient être entièrement placées dans la MP. Mais il n'est possible d'entrer ou de sortir de la MP que par des transitions. Les transitions $inst\_MT$, $inst\_MIT$, fin_1er_cyc et nb_cyc_nul doivent alors être sorties de la MP car elles sont liées à des places nécessairement situées hors de la MP. Par exemple, $inst\_MT$ et $inst\_MIT$ sont liées à des places de l'interface des composants modèles de référence. Ceci est une limite des possibilités d'expression proposées par la MP.

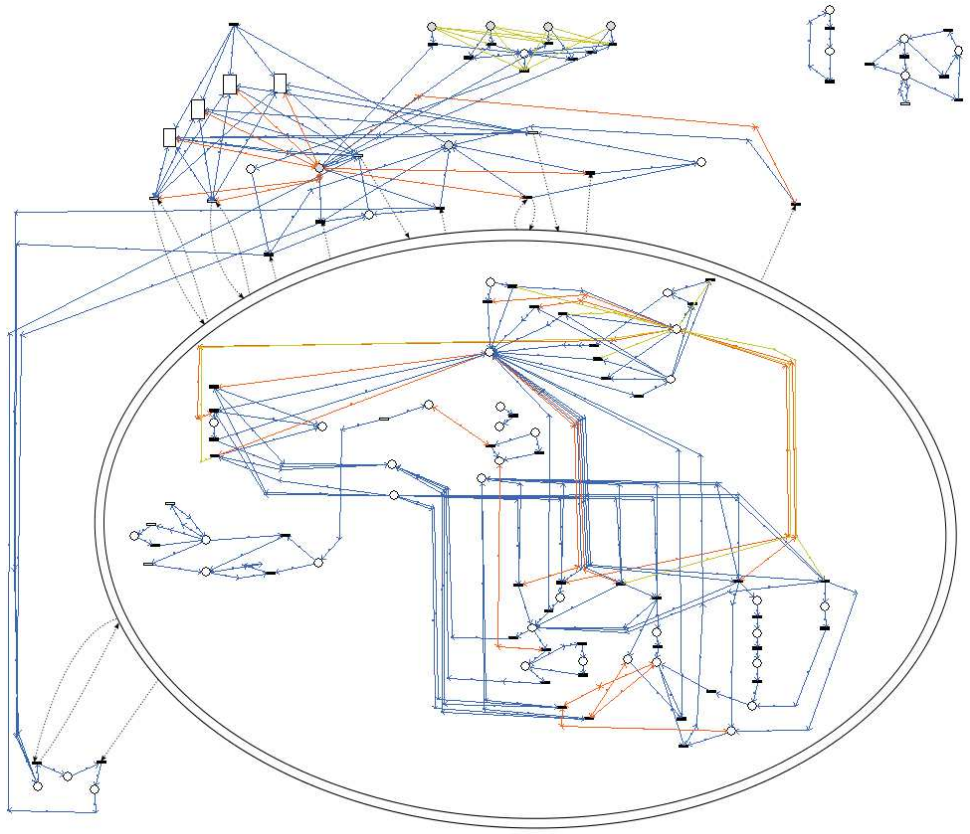

FIGURE 4.10 – Modèle complet de la micro-machine avec une macroplace

Maintenant que les limites du raffinement de la MP ont été posées, il est possible de gérer les différentes exceptions.

Comment les exceptions sont-elles gérées ?

gestion des erreurs de code : Les erreurs de codes étaient déjà gérées par le tir d'une seule transition (soit *vide_nop* soit *dep_memoire*). La gestion de ces erreurs n'a donc pas été modifiée. Les transitions sont situées en dehors de la MP, les jetons sont retirés à l'aide d'un arc sortant classique et ajoutés à l'aide d'un arc entrant (cf. figure 4.11).

gestion des erreurs de stimulation : La gestion des erreurs de stimulation a été modifiée. Elles sont désormais prise en compte dès que l'erreur est détectée. Plus exactement, la partie permettant de ne gérer qu'une seule erreur, même si plusieurs erreurs sont observées simultanément, est conservée. Les erreurs de stimulation sont donc prises en compte dès qu'il est certain que l'erreur détectée est unique (i.e. quand la place *err_ki* est marquée). Une seule transition, appelée *t_interrupt* est alors utilisée (cf. figure 4.11). Elle est donc reliée à la macroplace par un arc exception qui permet la purge de la macroplace et met la micro-machine dans un état sûr à l'aide d'un arc entrant. Ainsi, l'erreur est désormais gérée en 3 périodes d'horloge puisqu'il faut franchir 2 transitions pour marquer la place *err_ki* après qu'une erreur soit détectée par les modèles de référence.

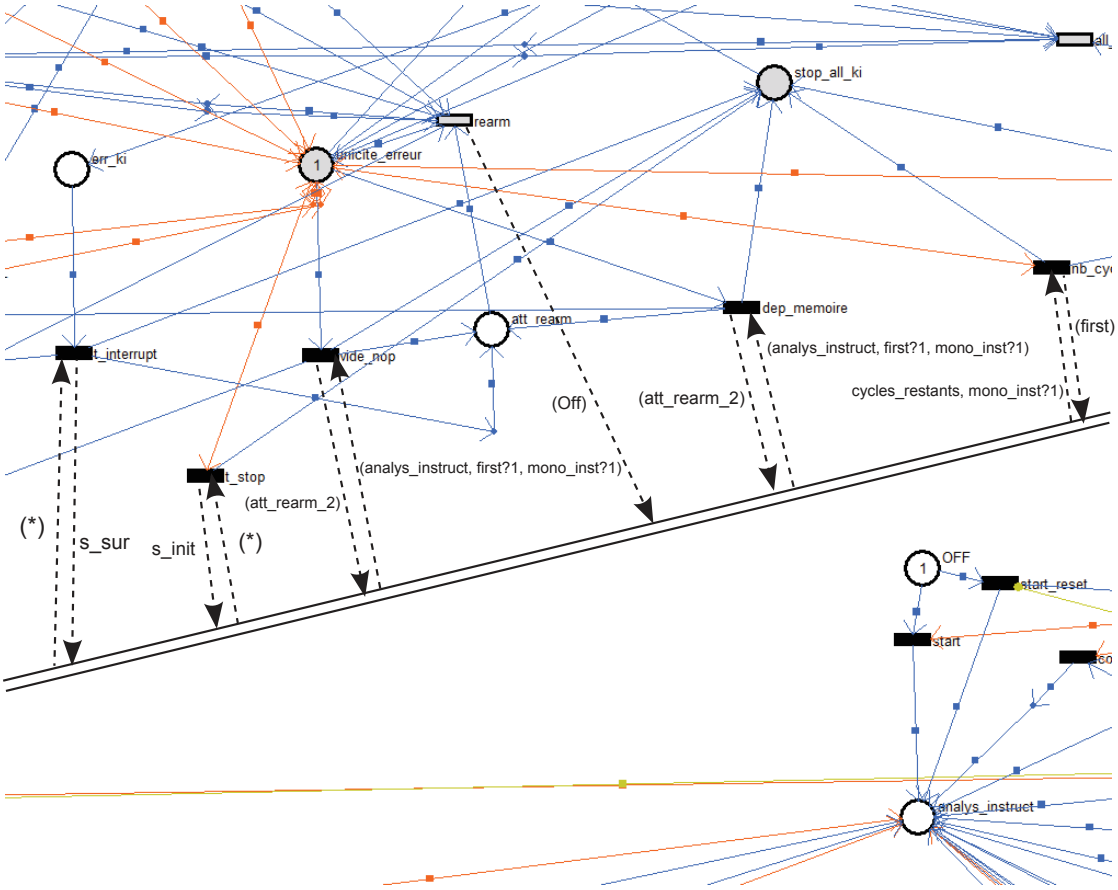

FIGURE 4.11 – Zoom sur le modèle de la micro-machine avec macroplace (transitions erreurs)

gestion des stops : La gestion des stops est effectuée de la même manière que pour les erreurs de stimulation (cf. figure 4.11) avec une transition, cette fois appelée t_stop, reliée à la macroplace par un arc exception. L'arc entrant est différent puisque dans ce cas la macroplace doit être remise dans son état initial. De plus, la transition *stop* ne peut pas être tirée si une erreur est en train d'être gérée ou si les modèles de référence sont en train d'être arrêtés (comme c'est le cas dans le modèle sans macroplace).

4.2.4 Comparaison des résultats obtenus

Nous souhaitons comparer les 2 modèles conçus pour cette génération de micro-machine sur plusieurs points : la conception (complexité du modèle, lisibilité, risque d'erreur humaine et réactivité), l'implémentation (surface et consommation) et l'analysabilité (taille du graphe des classes d'états). Le récapitulatif des différentes données obtenues est fourni dans le tableau 4.2.

Conception

Le modèle de la micro-machine seule (c'est-à-dire sans considérer les modèles de référence) sans macroplace contient : 61 places, 98 transitions et 371 arcs. Le modèle avec macroplace contient : 50 places, 75 transitions et 282 arcs. Ainsi le nombre d'éléments de RdP nécessaire pour modéliser la micro-machine est significativement moins élevé quand une macroplace est utilisée. Ceci permet bien sûr de rendre le modèle plus lisible. La figure 4.5 montre, en effet, que le modèle sans macroplace est assez peu lisible notamment à cause des nombreux arcs et la macroplace a permis de réduire le nombre d'arcs de plus de 30%. De même, la comparaison entre la figure 4.5 et la figure 4.10 rend évident le gain en complexité et la lisibilité du modèle apportés par la macroplace sur cet exemple.

Le risque d'erreurs humaines est aussi fortement réduit. En effet, sans macroplace, le concepteur doit d'abord détecter tous les états où il est nécessaire de traiter les exceptions puisqu'il ne peut pas traiter tous les états possibles comme le permet intrinséquement la macroplace. Ce choix des états où les exceptions sont traitées demande une très bonne connaissance du modèle et une mûre réflexion. Le concepteur doit en effet notamment s'assurer qu'un état où les exceptions seront traitées sera toujours atteint avant qu'une action potentiellement dangereuse pour le patient puisse être effectuée. De plus, une fois ces états définis, le concepteur doit modéliser la remise en état sûr ou initial de la micro-machine. Pour cette étape aussi, le concepteur doit s'assurer d'avoir bien pris en compte tous les cas possibles. En cas d'erreur de conception, la micro-machine pourrait en effet être remise dans un marquage menant à un blocage et/ou à un comportement dangereux. D'ailleurs, lors de l'étude des résultats d'analyse, il sera montré qu'un cas très particulier n'a pas été traité dans ce modèle. Cet oubli mène à un blocage.

L'utilisation de la macroplace permet de considérer très simplement tous les marquages possibles du raffinement et d'imposer facilement un nouveau marquage au raffinement. Le concepteur n'a plus qu'à gérer les places non situées dans la macroplace (ici 18 places sur 50). Cette gestion est beaucoup plus simple car le marquage de ces places, en cas d'exception, est connu puisque ce sont, soit des places qui gèrent le comportement attendu (i.e. la réaction) en cas d'exception, soit des places non concernées par les exceptions : les places des parties 11 et 12.

La réactivité du modèle est définie comme le temps entre le moment d'occurrence d'une exception et le moment où la micro-machine est placée dans son état sûr ou initial. Celle du modèle sans macroplace est différente suivant le type d'exception et la situation de la MM au moment de l'occurrence de l'exception.

Si l'exception est une erreur de stimulation, il faudra 2 périodes d'horloge pour s'assurer que l'erreur traitée est bien unique. Si une rampe est en train d'être décodée, la transition erreur est tirée dès que la place *ramp_en_cours* est marquée soit une attente au maximum d'1 période d'horloge. La mise dans l'état voulu de la micro-machine prendra toujours 5 périodes d'horloge (de par la séquence de RdP à réaliser pour cela). La gestion de l'exception prendra alors au maximum 8 périodes d'horloge. Si une rampe n'est pas en train d'être décodée, il faut attendre que la micro-machine ait fini de décoder l'instruction qu'elle est en train de traiter. Cette attente peut durer au maximum 1 période d'horloge.

Par contre, si jamais des paramètres temps-réel sont en train d'être chargés, la remise en état voulu de la micro-machine devra attendre 3 périodes d'horloge avant de pouvoir être réalisée (temps nécessaire pour charger les paramètres). Ainsi, dans ce cas, la remise du modèle dans l'état souhaité prendra au maximum 9 périodes d'horloge une fois l'exception détectée. Donc la réactivité au pire de la micro-machine sans macroplace est de 9 périodes d'horloge. En dehors de la valeur de la réactivité elle-même (9 périodes d'horloge, donc 9 μs est une réactivité acceptable), il est à noter qu'il n'est pas simple de déterminer "manuellement" la réactivité maximum du modèle et que la réactivité n'est pas constante. Ceci peut rendre la conception plus complexe et être source d'erreurs humaines.

Dans le modèle avec macroplace, le temps pour remettre la micro-machine dans son état initial ou sûr une fois la transition exception tirée est toujours d'1 période d'horloge. Mais, dans le cas des erreurs de stimulation, il faudra toujours attendre 2 périodes d'horloge avant de tirer la transition exception. La réactivité de la macroplace est donc au pire de 3 périodes d'horloge soit 3 μs.

Implémentation

Le modèle sans macroplace a été transformé en code VHDL à l'aide du logiciel HILECOP. Le mécanisme de macroplace n'étant pas encore intégré dans le logiciel HILECOP, le code VHDL a été écrit à la main d'après la méthode proposée dans le chapitre 3 en modifiant le code VHDL généré pour le modèle sans macroplace. L'implémentation virtuelle (car limitée à la simulation) des deux modèles a alors été effectuée sous Libero (cf. chapitre 1).

A noter que les implémentations ont été réalisées sans considérer les priorités entre les transitions en conflit. Le modèle initial a en effet été réalisé avant que les priorités puissent être gérées dans HILECOP, le concepteur les a donc normalement gérées avec des conditions mutuellement exclusives ou des arcs inhibiteurs. De plus, le marquage maximum de chaque place a été défini comme étant égal à 1, d'après les résultats d'analyse.

Le comportement obtenu avec les deux modèles a pu être observé par simulation sous Libero. Le micro-programme utilisé réalise une stimulation en créneau et est constitué de 3 instructions :

- stimulation pendant 100 μs avec une intensité de 500 μA
- décharge pendant 24,5 ms
- fin du programme

Une simulation simple est utilisée car le but ici n'est pas de valider le modèle de la micro-machine mais de comparer les comportements des deux modèles. Pour la comparaison, le comportement du modèle sans macroplace est considéré comme juste.

Plusieurs scénarios ont été testés : le fonctionnement normal, la demande d'un stop, une erreur de code et une erreur de stimulation. Ces simulations ont permis d'observer que le comportement de la micro-machine n'a pas été affecté de manière non désirée par l'introduction de la macroplace.

Dans les figures 4.12 et 4.13, le signal $SYSCLK$ représente l'horloge. $S\_REQUETE$ est la requête envoyée par le contrôleur (le code 4 signifiant une demande de Start et le code 1 une demande de Stop). $EXT\_COMPTEUR\_CARDINAL$ donne le numéro de l'instruction qui est en train d'être décodée. Le signal $EXT\_ASIC\_OUT\_ON$ vaut 1 quand une stimulation est en cours. Le signal $off/action\_activation$ est une image du marquage de la place Off et $stop\_interrupt/fire$ représente le tir de la transition $stop\_interrupt$.

La différence de réactivité entre les 2 implémentations est visible, par exemple, sur le scénario de stop (cf. figure 4.12 et figure 4.13). Un stop est demandé au bout de 53 μs, c'est-à-dire le signal $S\_REQUETE$ devient égal à 1. Le signal $off/action\_activation$ est observé pour pouvoir comparer la réactivité des deux modèles. Ainsi, dans ce scénario, sans la macroplace, la place Off est de nouveau marquée au bout de 7 μs alors qu'elle l'est au bout de 5μs dans le cas avec macroplace (avec une horloge de fréquence 1 MHz). Dans ce cas précis, la différence de temps provient uniquement du temps nécessaire pour réinitialiser la macroplace puisque les deux transitions gérant le stop dans chacun des modèles sont toutes les deux tirées 1 μs après la demande de stop.

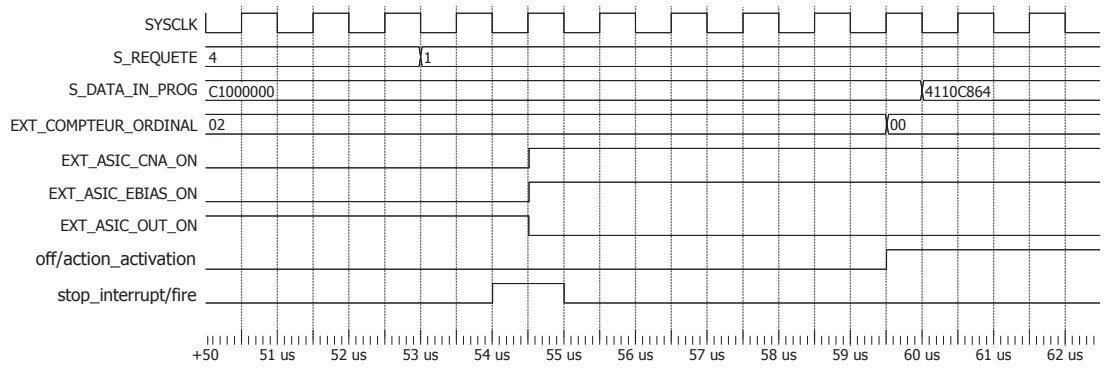

FIGURE 4.12 – Simulation du modèle sans macroplace implémenté, dans le cas d'un stop

Pour le modèle sans macroplace, l'implémentation sur un FPGA nécessite 6 292 cellules logiques tandis que pour le modèle avec macroplace 6 249 cellules logiques sont nécessaires. L'utilisation de la macroplace n'a donc pas d'impact négatif sur le critère important de la surface nécessaire pour l'implémentation du modèle. De même la consommation totale (statique+dynamique), pour une simulation de 30 ms avec une tension de 1,2 V, sont du

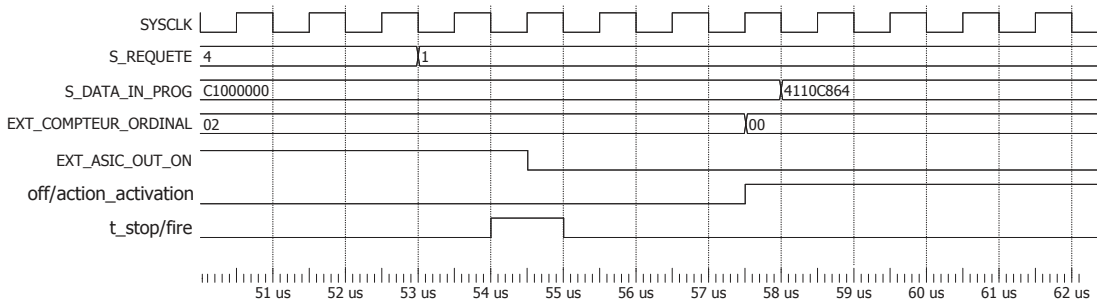

FIGURE 4.13 – Simulation du modèle avec macroplace implémenté, dans le cas d'un stop

même ordre de grandeur : 412,632 μA sans la macroplace et 401,8 μA avec la macroplace. Ainsi, la macroplace permet d'avoir un système plus réactif et plus simple à concevoir pour une surface et une consommation équivalentes.

A noter que les consommations données ici sont issues de l'estimation de consommation de Libero. Les valeurs exactes de consommations peuvent donc être différentes mais nous les exploitons ici de manière qualitative pour pouvoir réaliser une comparaison entre les deux modèles.

Analyse

Certaines hypothèses simplificatrices ont dû être posées pour parvenir à obtenir le graphe des classes accessibles. Ces hypothèses servent à minimiser le risque d'explosion combinatoire. Comme le but est de comparer les résultats obtenus avec et sans macroplace, les mêmes hypothèses sont réalisées sur les 2 modèles. Le but est en fait ici d'étudier l'impact de la macroplace sur la complexité de l'analyse et non la validité du modèle conçu initialement.

Les hypothèses simplificatrices réalisées sont :

- le blocage potentiel des transitions temporelles avec des conditions n'est représenté que sur les transitions où cela est nécessaire, c'est-à-dire les transitions liées à la gestion des instructions de boucle.
- les transitions non temporelles interprétées devraient toutes avoir un intervalle de temps égal à $[1, +\infty[$. Or la majorité des transitions sont dans ce cas. Ainsi le graphe des modèles accessibles devient rapidement trop complexe pour être calculé. Pour éviter ceci, les intervalles temporels associés aux transitions sont égaux $[1, 1]$. Ce n'est pas gênant pour le fonctionnement normal du modèle car, dans le système réel, les conditions sont utilisées pour permettre de tirer la transition adaptée à la situation du système réel et non pas pour attendre qu'un événement se produise. Par contre, cela est problématique dans le cas des transitions traduisant la détection d'une erreur ou une demande de stop. En effet, si seul l'intervalle $[1, 1]$ est conservé, les transitions erreur (ou stop) sont nécessairement tirées et il n'est jamais possible d'observer le fonctionnement normal. Une place est alors utilisée pour bloquer (ou autoriser selon le souhait) le tir des transitions erreur ou stop.
- Le blocage potentiel des transitions $boucle\_inf\_first$, $nb\_cyc\_non\_nul$, nb_cycles_f, $commute$, $commute\_met\_1$ et $inst\_boucle$ est modélisé car ces transitions seront effectivement bloquées sur le système réel. Les autres risques de blocage ne sont pas modélisés.

- Pour la transition *inst_boucle*, modélisant le fait que la prochaine instruction à décoder est une instruction de type boucle, la seule modification du blocage potentiel de la transition n'est pas suffisant. En effet, lors du décodage de cette instruction (instruction de type MT par exemple), le marquage des places sensibilisant la transition *inst_boucle* n'est pas modifié, notamment parce que le jeton de la place *analys_instruct* n'est pas retiré (seul un arc test est utilisé). Ainsi l'instruction suivante ne peut pas être du type instruction boucle, ce qui ne correspond pas à ce qui se passe sur le système réel. Pour éviter ce problème, les transitions *t_mt*, *t_mit* et *rt_mit_nofirst* sont dédoublées pour débloquer, si besoin, la transition *inst_boucle*.
- La génération des demandes de stop et de détection des erreurs par les modèles de référence sont, chacune, gérées par une place et une transition temporelle ayant un intervalle de la forme $[1, +\infty[$. Cela permet de modéliser le fait que ces requêtes peuvent arriver n'importe quand et un nombre infini de fois. Le tir de la transition met un jeton dans la place bloquant le tir des transitions erreur ou stop et permettent ainsi de déclencher la gestion de l'exception.
- Les transitions permettant la détection d'erreurs de code sont bloquées car elles sont traitées de la même manière dans les deux modèles comparés.
- L'enchaînement des phases de décharge est géré par des variables en VHDL. Pour simplifier le modèle, seule la transition *t_mt_phase_3* n'est pas bloquée. Son intervalle temporel est défini à $[200, 200]$ au lieu de $[20000, 20000]$ ce qui réduit la durée de décharge.
- Les parties 11 et 12, qui sont indépendantes du reste du modèle, sont retirées.

Le modèle analysable sans macroplace contient 65 places et 107 transitions tandis que le modèle analysable avec macroplace contient 93 places et 154 transitions. La taille du graphe des classes d'états obtenu pour les 2 modèles a été comparée sur 4 scénarios différents : aucune exception (0), stop possible (S), interruption possible (I) et stop et interruption possibles (SI). Les interruptions correspondent à la détection d'une erreur de stimulation. Les résultats sont donnés dans le tableau 4.1.

	sans MP			avec MP		
scénario	états	transitions	états morts	états	transitions	états morts
0	20 979	65 514	0	21 316	66 005	0
S	61 944	197 546	0	97 975	313 725	0
I	55 266	189 084	6	88 154	289 448	0
SI	152 185	538 912	6	277 950	1 017 514	0

TABLE 4.1 – Taille des graphes des classes accessibles obtenus

Les résultats montrent qu'avec ou sans macroplace la taille du graphe des classes d'états est du même ordre de grandeur. Néanmoins le graphe des classes d'états est systématiquement plus grand pour le modèle avec macroplace que pour celui sans et il peut être jusqu'à 2 fois plus grand. La faible différence dans le cas où aucune exception ne peut se produire provient de l'ajout du mécanisme pour gérer l'activité de la macroplace. Si les demandes d'activation de la macroplace sont bloquées dans le modèle avec macroplace, le nombre d'états du graphe des classes d'états avec macroplace devient bien égal à celui obtenu sans macroplace.

Pour les scénarios avec interruption, sur le modèle sans macroplace, il y a un risque de blocage puisqu'il y a 6 états morts. Cela provient du fait que les cas (rares mais pas impossibles puisque ce cas a été rencontré lors de tests) où une erreur est détectée, alors que la transition *nb_cyc_nul* ou la transition *fin_1er_cyc* doit être tirée, n'ont pas été pris en compte. Ceci permet de démontrer la difficulté pour les concepteurs de penser à tous les cas possibles. Comme avec la macroplace tous les cas sont automatiquement pris en compte, ce problème n'est pas rencontré, c'est pour cela qu'il n'y a pas d'états morts.

La plus grande complexité de l'analyse dans le modèle avec macroplace peut en partie s'expliquer par le fait que ce ne sont pas exactement les mêmes comportements qui sont analysés. En effet, avec la macroplace, les exceptions sont toujours gérées immédiatement (si possible) tandis que sans macroplace, elles ne sont gérées que dans certains états.

Sur tous les scénarios testés, le marquage maximal de chaque place est bien de 1 ce qui permet de valider, dans le cadre de nos hypothèses, la limite maximum utilisée lors de l'implémentation.

	Conception			
modèle	places	transitions	arcs	réactivité
MM	61	98	371	7 μs
MMMP	50	75	282	1 μs
gain	**18%**	**23.5%**	**24 %**	**6** μs
	Implémentation		Analyse (scénario SI)	
modèle	surface	consommation	états	transitions
MM	6 292	413 μA	152 185	538 912
MMMP	6 249	402 μA	277 950	1 017 514
gain	**0,6%**	**2,6%**	**- 82%**	**- 88%**

TABLE 4.2 – Comparaison des résultats obtenus sur les 2 modèles

Bilan

Sur le modèle de micro-machine de première génération, l'utilisation de la macroplace a donc eu un impact très positif d'un point de vue conception, neutre d'un point implémentation et négatif sur la complexité de l'analyse. L'utilisation de l'analyse a, de plus, permis de détecter des cas que le concepteur n'avait pas pris en compte. Cela permet d'illustrer la difficulté pour le concepteur d'être exhaustif dans les cas qu'il traite pour la gestion d'exceptions. L'analyse formelle peut être d'une aide précieuse pour détecter les oublis. Cela demande néanmoins du temps et l'utilisation de la macroplace, toujours couplée avec l'analyse bien sûr, reste donc une solution préférable pour traiter les exceptions dans ce modèle. Etudions si l'impact de la macroplace est le même sur le modèle de la micro-machine de troisième génération.

4.3 Application sur la micro-machine de troisième génération

La micro-machine de troisième génération, appelée plus simplement dans ce paragraphe micro-machine, a été utilisée dans le stimulateur appelé stimulateur générique. Elle a toujours le même rôle : la micro-machine décode les instructions du micro-programme pour ensuite envoyer les ordres permettant d'exécuter la stimulation prévue sur l'électrode. Le décodage est réalisé en parallèle de l'exécution de l'instruction précédente. Par contre, le fonctionnement précis de la micro-machine a été fortement modifié, notamment car le jeu d'instructions est différent [4] ainsi que la gestion des exceptions.

Une des modifications importantes dans cette troisième génération vient de la dissociation entre le profil de stimulation (micro-programme) et la configuration de l'électrode (appelée électrode virtuelle). Une électrode virtuelle décrit les pôles actifs et la répartition de courant entre ces pôles. Une simulation est donc désormais décrite par un couple micro-programme, électrode virtuelle.

A noter que les deux modèles de micro-machine ont été conçus par le même ingénieur. Ainsi le changement observé dans la gestion des exceptions provient de l'expérience gagnée par le concepteur et non d'une différence de vision entre 2 personnes. C'est pour cela notamment qu'il est intéressant d'étudier l'impact de la macroplace sur ces modèles. En effet, l'ingénieur, de par son expérience de la modélisation à l'aide d'HILECOP, a amélioré la modélisation de la gestion des exceptions. L'impact de la macroplace sur la gestion des exceptions sera peut-être alors différent que sur le précédent modèle.

4.3.1 Comportement attendu

Le modèle décrivant le comportement attendu de la micro-machine est donné figure 4.14. Le détail des différentes parties est donné en annexe D. Les principales fonctions remplies par la micro-machine sont exposées ci-dessous. L'attention du lecteur est attirée sur le fait que le logiciel permet de créer des renvois (dits raccourcis) de places et de transitions permettant de faire une copie d'une place ou d'une transition pour éviter d'avoir des arcs traversant tout le modèle. Ainsi, le modèle n'est pas constitué de différents sous-RdP indépendants, comme la figure en donne l'impression. Les places et transitions raccourcies sont indiquées par une petite flèche dans le coin en bas à droite. Cette fonctionnalité graphique facilite le travail du concepteur car son modèle est plus lisible et il peut plus facilement travailler par "bloc fonctionnel" mais elle ne permet pas de se rendre compte de la complexité du modèle et cache les liens entre les différentes parties.

Comportement normal

La micro-machine possède trois modes de fonctionnement : la stimulation (S), la décharge globale (D) et la mesure d'impédance (M). La stimulation permet d'exécuter un micro-programme. Lors de cette exécution, seuls les pôles de l'électrode ayant été utilisés lors de la stimulation sont déchargés (il s'agit donc d'une décharge partielle). Par mesure de sécurité, il est aussi possible de réaliser après une ou plusieurs stimulations, une décharge globale qui déchargera cette fois tous les pôles de l'électrode. La mesure d'impédance permet, comme son nom l'indique, de mesurer l'impédance de l'électrode. Cela permet de s'assurer que l'électrode (ou le lien avec l'électrode) n'est pas endommagée.

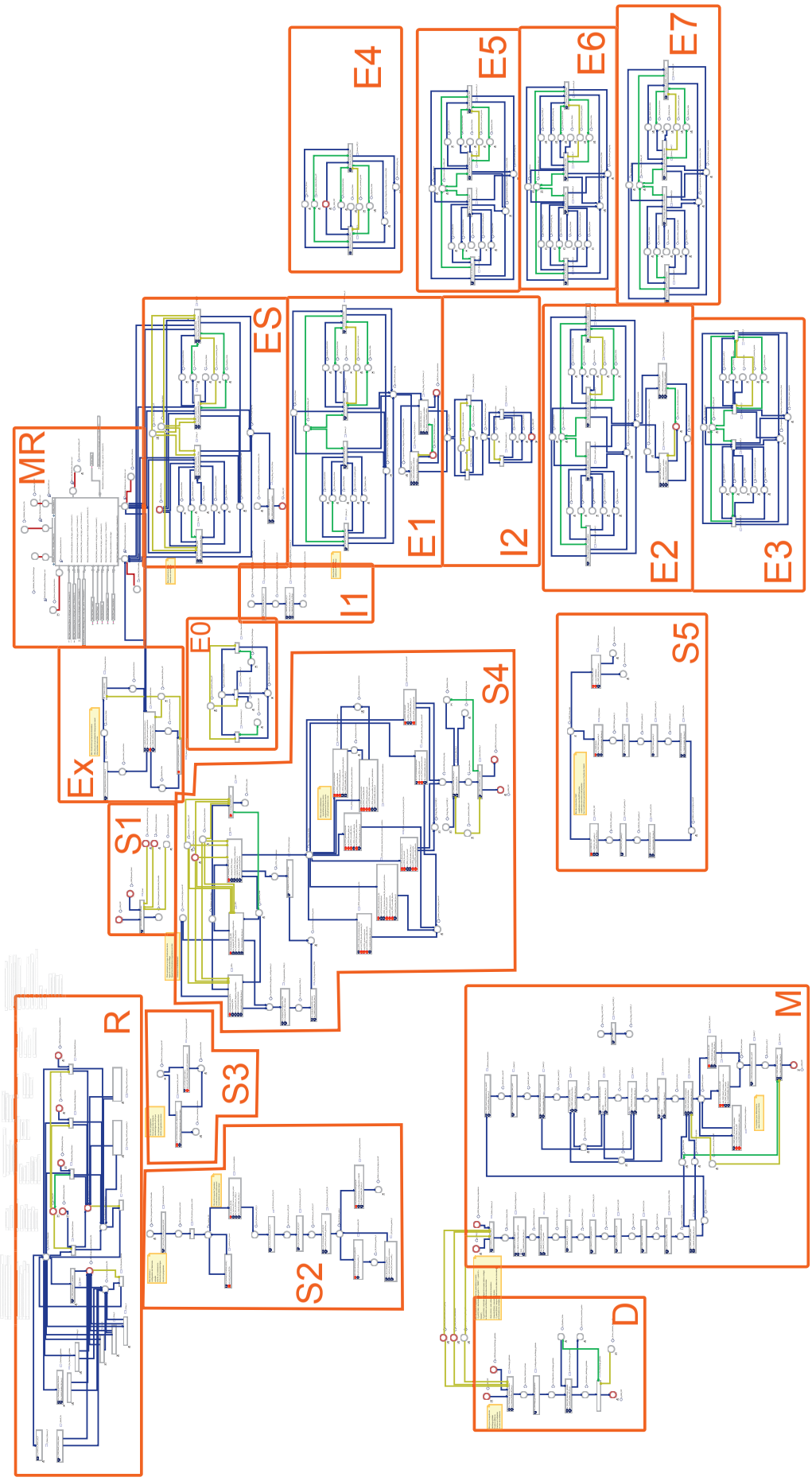

FIGURE 4.14 – Modèle complet de la micro-machine de troisième génération

Le choix entre les différents modes de fonctionnement ainsi que la validité des requêtes envoyées par le contrôleur sont gérés par la partie R (cf. figure 4.14). Le mode décharge globale est géré par la partie D et le mode mesure d'impédance par la partie M.

Pour le mode stimulation, l'électrode virtuelle doit d'abord être configurée (choix des pôles de l'électrode qui seront utilisés pendant la stimulation et répartition du courant entre les pôles). Cette configuration est gérée par la partie S2. Une fois cette opération réalisée, les instructions du micro-programme sont décodées puis exécutées (respectivement partie S4 et parties Ex ou S5). Les instructions du micro-programme ont été modifiées par rapport à la première génération de micro-machine et sont désormais au nombre de 4 : STIa, STIr, DT2L et CMZ. Les instructions STIa et STIr sont des instructions stimulantes. La première permet de programmer une stimulation absolue (i.e. les valeurs d'intensité et de temps sont indiquées de manière absolue). La seconde permet de programmer une stimulation relative (i.e. les valeurs d'intensité et de temps indiquées définissent un incrément, potentiellement négatif, par rapport aux valeurs précédentes). L'instruction DT2L permet de réaliser différentes actions :

- des phases de non stimulation (décharges passives ou phases inter-stimulation qui sont paramétrées en temps)
- des boucles locales (avec adresse du saut arrière et nombre de boucles à effectuer)
- indication de la fin du micro-programme

L'instruction CMZ permet de configurer l'électrode virtuelle, c'est à dire les pôles actifs et la répartition de courant, et donc la configuration du générateur de stimulus. Ce changement de configuration est géré par la partie S5.

La partie S1 garantit que le lancement d'une stimulation ne peut avoir lieu que si aucune exception n'a pas été détectée ou est en train d'être analysée.

Détections des exceptions

En ce qui concerne les exceptions, la micro-machine doit gérer les requêtes de stop envoyées par le micro-contrôleur, les erreurs de code et les erreurs de stimulation. Commençons par étudier comment les erreurs sont détectées.

- La détection d'une requête de stop est gérée par la partie R puisque c'est une requête du contrôleur.
- Les erreurs de code détectées sont : dépassement de la mémoire allouée pour le micro-programme (partie S3), les erreurs sur les instructions de boucle (i.e. boucles imbriquées ou adresse de ligne inexacte (partie S4)) et les erreurs sur les instructions CMZ (modification ni des pôles ni des ratios de courant) (partie S5). Toutes les erreurs de code marquent la même place appelée $p\_Erreur\_detecte\_Execution$.
- Les erreurs de stimulation sont détectées par les modèles de référence (MR) qui marquent la place $p\_Erreur\_detecte\_Mise\_Off$ en cas d'erreur.

Les parties E0 et I2 gèrent la possibilité d'avoir simultanément une erreur de code et une erreur de stimulation. Dans ce cas, l'erreur de stimulation est gérée et l'erreur de code est ignorée (il n'est possible de gérer qu'une seule erreur à la fois dans la micro-machine). Si seule une erreur de code est détectée (partie E0), les modèles de référence n'analysent plus la commande en cours et un jeton est placé dans la place $p\_Erreur\_detecte\_Mise\_Off$.

Traitement des exceptions

Une fois que le système est capable de détecter les exceptions, il faut modéliser les réactions attendues quand celles-ci se produisent. Trois comportements différents doivent être adoptés en cas d'exception :

- si un stop est demandé, la micro-machine doit être remise dans son état initial.
- si une erreur se produit alors qu'aucune stimulation n'est en cours (micro-machine en off ou dans un autre mode de fonctionnement), la micro-machine doit être remise dans son état initial et le contrôleur doit seulement être prévenu qu'il y a eu erreur.
- si une erreur se produit pendant une stimulation, la micro-machine doit être remise dans son état initial et le contrôleur doit être prévenu qu'il y a eu erreur mais aussi que l'erreur a eu lieu pendant la stimulation.

Dans tous les cas, une réinitialisation des registres de configuration doit être réalisée. Elle est gérée par la partie I1.

Les parties E1 à E7 et ES sont toutes dédiées à la remise en état initial de la micro-machine en cas d'exception et seront étudiées dans le §4.3.2. En effet, ce sont logiquement ces parties qui seront modifiées par l'utilisation de la macroplace.

4.3.2 Gestion des exceptions sans macroplace

Comment les exceptions sont-elles gérées ?

Le principe de la remise à l'état initial de la micro-machine est le même quelle que soit l'exception gérée. Il a été conçu de la manière suivante :

- Le concepteur détermine les états de la micro-machine dans lesquels il souhaite gérer les erreurs. Il n'est en effet pas possible de gérer les erreurs dans tous les états, cela est quasi-impossible à réaliser manuellement vu la complexité du modèle. Des arcs inhibiteurs sont alors placés entre la place qui indique qu'une exception doit être gérée et les arcs sortants des places marquées dans cet état. Par exemple, dans le cas d'une décharge globale, la gestion d'erreur n'est réalisée qu'une fois la place $p\_activation\_decharge\_globale$ marquée (cf. figure 4.15). C'est-à-dire que quand une erreur est détectée (place $p\_Erreur\_detecte\_Mise\_Off$ marquée), le RdP de la partie D continuera à évoluer comme si aucune erreur n'avait été détectée jusqu'à ce que la place $p\_activation\_decharge\_globale$ soit marquée. Par contre, la transition $t\_Fin\_Decharge\_Globale$ ne peut pas être tirée si une erreur a été détectée. Cela permet de s'assurer que l'erreur est gérée avant de revenir dans l'état initial de la micro-machine.

- Les états choisis sont regroupés. En effet, comme les instructions de stimulation sont réalisées en parallèle des autres actions de la micro-machine (décodage, mise en off,...), il n'est pas possible de connaître l'état de la partie Ex quand le traitement de l'erreur sera demandé dans les autres parties du modèle. La partie Ex peut se situer dans 4 marquages différents. Ainsi, pour la décharge globale par exemple, il existe 4 états qui permettent de traiter l'erreur dans lesquels la place $p\_activation\_decharge\_globale$ est marquée (cf. figure 4.16). Ces 4 états sont regroupés dans ce que nous appelerons un groupe d'états. Un groupe d'états est défini comme l'ensemble des états ayant le même marquage si les places de la partie Ex ne sont pas considérées.

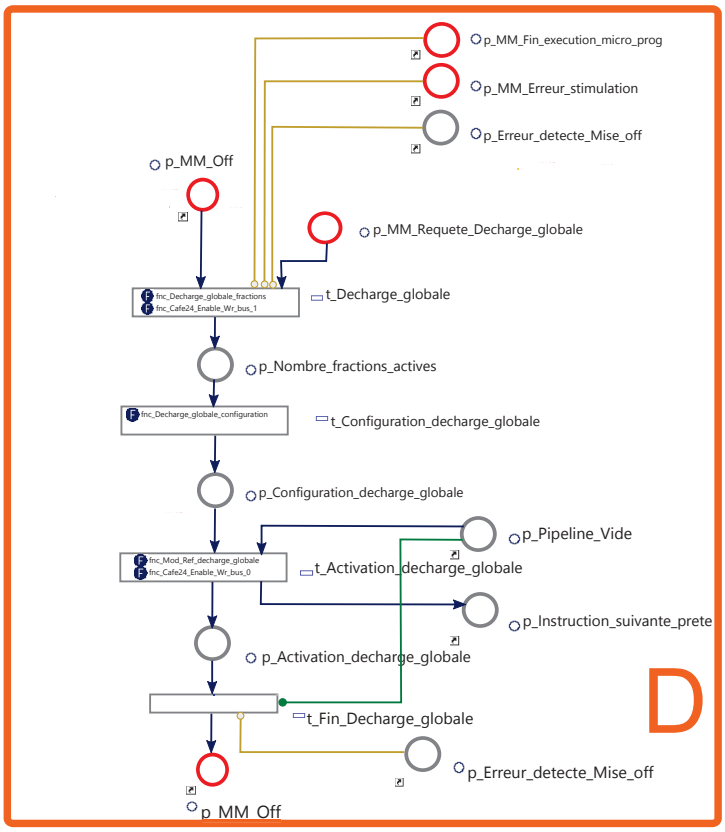

FIGURE 4.15 – Sous-partie du modèle gérant la décharge globale (partie D)

- Le concepteur dessine une partie Ei du modèle pour chaque groupe d'états qui permettra de gérer la réinitialisation de la micro-machine. La partie E2 donnée figure 4.16 concerne par exemple la gestion d'une erreur en cas de décharge globale. Dans chaque schéma, nous retrouverons 4 transitions ayant les mêmes arcs entrants et sortants avec les places de la partie Ex. La place entrante $p\_activation\_decharge\_globale$ commune aux 4 transitions est modifiée quand ce n'est pas le même groupe d'états qui est traité. Le seul cas différent est la gestion d'une erreur quand la micro-machine est déjà en mode Off (place p_MM_Off marquée). Dans ce cas seuls 2 états sont possibles dans la partie Ex (il n'y a forcément aucune instruction en attente d'être traitée) (cf. partie E4). Le tir de ces 4 (ou 2) transitions, appelées transitions exceptions, permet la réinitialisation directe de la micro-machine à part pour la place p_MM_Off qui sera marquée seulement quand l'exception aura été complètement gérée (messages erreurs envoyés et traités par le contrôleur par exemple).

- Les transitions permettant la réinitialisation ayant été modélisées, il reste à modéliser l'envoi des erreurs et, si besoin, la réinitialisation des registres de configuration et la réinitialisation des modèles de référence. Toutes les transitions exceptions ont ainsi pour place aval, la place $p\_Programmation\_registre\_configuration\_commun\_Deb$ qui permet de lancer cette réinitialisation (gérée par la partie I1). Sinon 3 places et 3 transitions sont utilisées pour décrire les 3 comportements différents attendus en cas d'exception en termes de message d'erreur. C'est le tir des transitions ainsi créées qui enverra, si besoin, les informations d'erreurs au contrô-

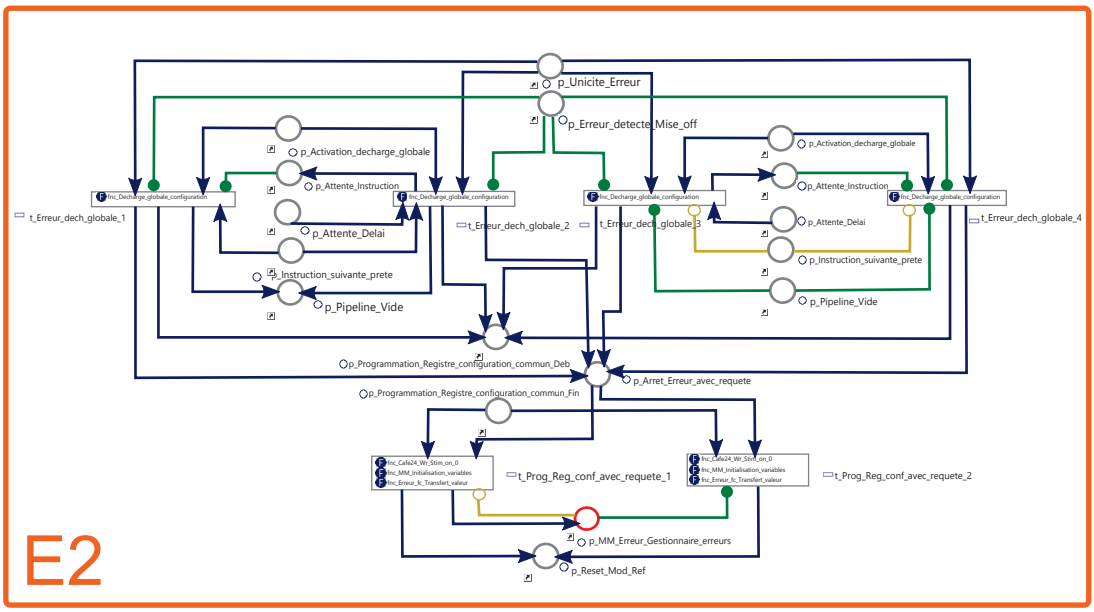

FIGURE 4.16 – Sous-partie du modèle gérant la réinitialisation de la micro-machine en cas d'erreur durant une décharge globale (partie E2)

leur et remettra un jeton dans la place p_MM_Off. Dans les 3 cas, la place $p\_Reset\_Mod\_Ref$ est marquée ce qui entraîne la réinitialisation des modèles de référence (partie I2). Les transitions d'exceptions ont toutes en place aval une des ces 3 places, suivant l'exception considérée. Pour la décharge globale, par exemple, un jeton est mis dans la place $p\_Arret\_Erreur\_avec\_requete$ ce qui placera ensuite un jeton dans la place $p\_MM\_Erreur\_Gestionnaire\_erreurs$ (cf. figure 4.16). Cette dernière place est placée à l'interface du composant de la micro-machine pour que l'information puisse être remontée au contrôleur.

Seul un groupe d'états doit être considéré deux fois : celui où la place $p\_Analyse\_instruction$ est marquée. En effet, cette place peut être marquée en cas de stop et en cas d'erreur pendant une stimulation (partie ES et E1). Il faut donc gérer les 2 cas avec des transitions différentes puisque ce n'est pas la même réaction qui est attendue.

La modélisation des exceptions est intéressante d'un point de vue réactivité car le marquage de la micro-machine est réinitialisé en ne tirant qu'une seule transition (sans considérer la place p_MM_Off qui ne sera à nouveau marquée que lorsque l'erreur sera totalement traitée). De plus, le concepteur a été assez astucieux pour limiter le nombre de groupes d'états à traiter (et ainsi limiter le nombre de transitions exceptions nécessaires). Par exemple, en étudiant précisément le modèle, nous remarquons qu'en cas d'une erreur de code détectée sur les boucles, un jeton est replacé dans la place $p\_Analyse\_instruction$ (cf. figure 4.17). Cet ajout est, a priori, contre-intuitif puisque justement, comme il y a erreur, il n'est pas souhaitable qu'une autre instruction soit analysée. Mais cela permet au concepteur d'utiliser la gestion d'erreur offerte par la partie E1 plutôt que de devoir ajouter un groupe d'états à traiter. Et le comportement n'est pas dangereux pour le patient puisque dans tous les cas il n'est pas possible de lancer une nouvelle analyse d'instruction en cas d'erreur.

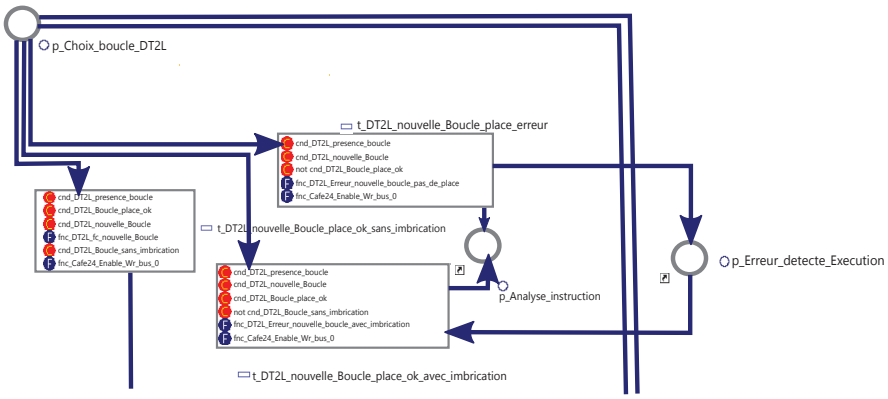

FIGURE 4.17 – Zoom de la partie S4

Cette modélisation permet d'illustrer à nouveau le fait que la modélisation de la gestion d'exceptions est plus complexe dans le cas où il y a des "sous-réseaux" qui évoluent en parallèle. En effet, seul 7 groupes d'états sont considérés mais avec le parallélisme entre la partie Ex et les autres parties, 26 états ont besoin d'être considérés.

L'avantage de cette modélisation des erreurs, en dehors de la réactivité, est qu'elle est assez systématique, ce qui limite donc les risques d'erreurs humaines. Cependant la réflexion évidente réalisée sur l'optimisation de la modélisation des exceptions, 30 transitions exceptions sont quand même nécessaires, soit environ le tiers des transitions du modèle complet. Ainsi, lors de la première version fournie par l'ingénieur, nous avons réalisé grâce à l'analyse que le cas où une erreur de code et une erreur de stimulation se produisaient simultanément n'avait pas été géré. La version présentée ici est la version corrigée par l'ingénieur une fois l'oubli détecté.

De plus, si la réinitialisation de la micro-machine est réactive une fois que l'exception est traitée, comme elle n'est traitée que dans certains états, la réactivité réelle de la gestion des exceptions (c'est-à-dire le temps entre l'arrivée de l'erreur et la réinitialisation de la micro-machine) reste nettement perfectible.

4.3.3 Gestion des exceptions avec macroplace

Comment les exceptions sont-elles gérées ?

Nous souhaitons modéliser à l'aide de la macroplace le même comportement global en cas d'exception que celui modélisé par le modèle présenté précédemment. La différence est que les exceptions seront gérées dès leur apparition au lieu de n'être gérées que dans certains états bien choisis. Le modèle global obtenu avec la macroplace est donné figure 4.18.

La macroplace contiendra autant que possible les parties D, M, S1, S2, S3, S4, S5 et Ex puisqu'elles décrivent le fonctionnement normal de la micro-machine et n'émettent pas d'erreurs. La partie R n'est pas placée dans la macroplace car elle gère la communication avec les autres composants du modèle. Évidemment, le composant décrivant le modèle de référence (Partie MR) n'est pas inclus dans la macroplace. La transition $t\_Execution\_Instruction$ est reliée à une place de l'interface de ce composant, elle n'est donc pas non plus incluse dans la macroplace.

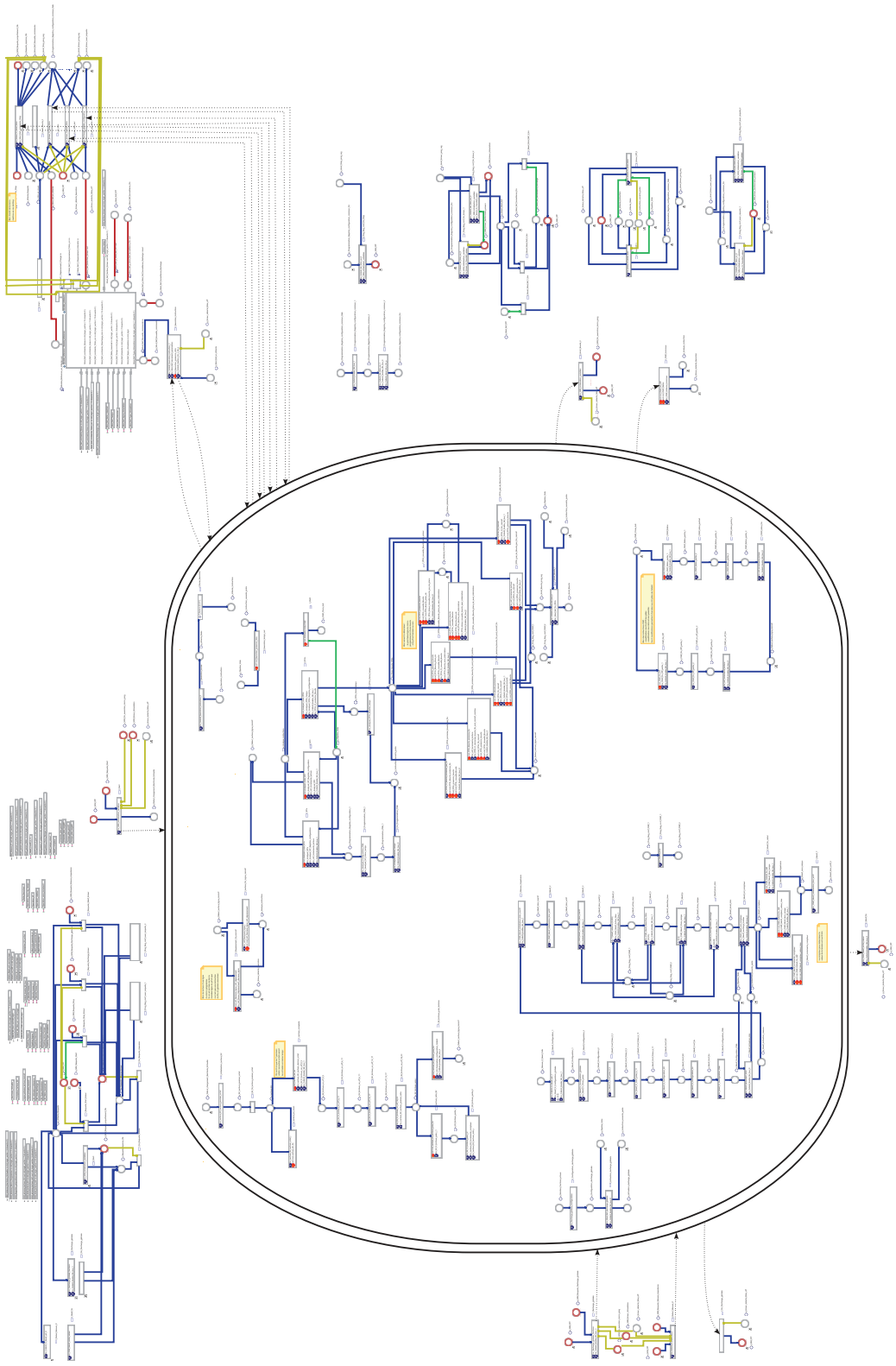

FIGURE 4.18 – Modèle complet de la micro-machine troisième génération avec macroplace

Les transitions permettant de remettre la micro-machine en Off dans le cas du fonctionnement normal sont définies comme transitions sortantes de la macroplace puisque la place p_MM_Off est située en dehors de la macroplace.

La partie I1 est conservée puisque la réinitialisation des registres communs est toujours nécessaire. Il est placé hors de la macroplace puisqu'utilisé pour le traitement des erreurs.

Toutes les transitions exceptions du modèle sans macroplace sont supprimées. Les 3 places et 3 transitions permettant de gérer les 3 comportements attendus suivant le type d'exception sont conservées. Les transitions exceptions du modèle sans macroplace sont remplacées par 4 transitions exceptions : t_stop, t_err_stim, $t\_err\_stim\_exc$ et $t\_erreur\_pas\_stim$ (cf. figure 4.19).

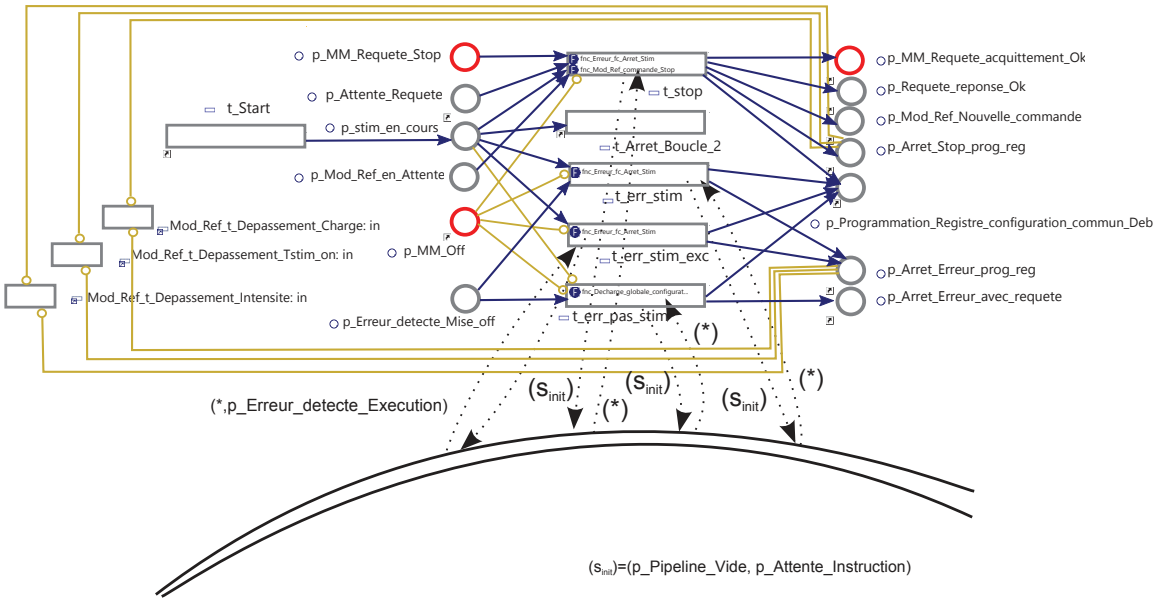

FIGURE 4.19 – Zoom sur les transitions exceptions ajoutées

4 transitions sont nécessaires bien qu'il n'y ait que 3 comportements différents. Dans notre modélisation, nous avons en effet choisi de séparer la détection d'une erreur de code ($t\_err\_stim\_exc$) et la détection d'une erreur de stimulation (t_err_stim). En effet, les erreurs de code sont toujours détectées en marquant la même place. Cette place est située dans la macroplace et le choix a été fait de sensibiliser directement une transition exception plutôt que de sortir l'information de l'erreur pour marquer la même place erreur que celle utilisée pour les erreurs de stimulation et donc perdre en réactivité.

Une place $p\_stim\_en\_cours$ est ajoutée pour permettre de modéliser si une stimulation est en cours ou non. Elle est marquée quand une stimulation démarre (tir de t_Start) et démarquée, soit en cas d'une fin normale de la stimulation (tir de $t\_Arret\_Boucle\_2$), soit si une transition exception est tirée. Cette place est de plus liée à la transition $t\_erreur\_pas\_stim$ par un arc inhibiteur.

Les 4 transitions exceptions ont bien sûr un arc exception qui les lie à la macroplace et les arcs sortants qui permettent de réinitialiser la macroplace. Chacune des transitions cible la place qui permettra d'envoyer les bons signaux d'erreurs au contrôleur.

Le cas où une erreur de code et une erreur de stimulation se produisent simultanément est géré par une priorité entre les transitions $t\_err\_stim\_exc$ et t_err_stim. Le jeton situé dans la place $p\_Erreur\_detecte\_Execution$ sera automatiquement retiré lors de la purge de la macroplace (puisqu'elle appartient au raffinement de la macroplace). Par contre, les modèles de référence doivent toujours être réinitialisés, ainsi la partie I2 est conservée mais modifiée pour procéder immédiatement à la réinitialisation des modèles de référence(cf. figure 4.20). La partie E0 est, elle, supprimée. Pour s'assurer qu'une erreur ne soit pas émise par les modèles de référence pendant qu'une erreur de code ou qu'un stop est géré, des arcs inhibiteurs sont ajoutés entre les places $p\_Arret\_Erreur\_prog\_reg$ et $p\_Arret\_Stop\_prog\_reg$ et les 3 transitions du modèle de référence servant à détecter les erreurs de stimulation.

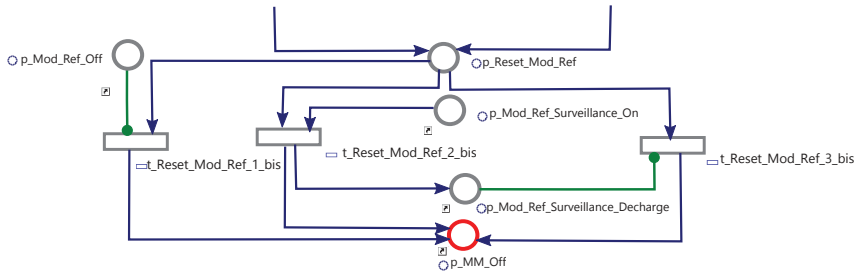

FIGURE 4.20 – Partie I2 du modèle avec macroplace

4.3.4 Comparaison des résultats obtenus

Comme pour la micro-machine de première génération, les résultats obtenus sur les 2 modèles sont étudiés au travers du prisme de la conception, de l'implémentation et de l'analyse. Le récapitulatif des différents résultats obtenus est donné dans le tableau 4.4.

Conception

Nous considérons ici le modèle complet remis à plat, c'est-à-dire contenant le modèle de référence. Le modèle sans macroplace contient 80 places, 133 transitions et 649 arcs. Le même modèle avec macroplace contient 79 places, 105 transitions et 384 arcs. Cette fois encore la macroplace a permis de simplifier significativement le modèle. Par contre, les exceptions gérées, sans la macroplace, étant principalement portées par des transitions, le nombre de places demeure quasi inchangé. La complexité du modèle et sa lisibilité a par contre bien diminué, le nombre d'arcs ayant chuté de 42%. Ainsi même quand la gestion des exceptions est réalisée de manière optimale, l'apport de la macroplace est toujours très important sur la facilité de conception.

Le risque d'erreur humaine est toujours assez conséquent sans macroplace car le concepteur doit toujours déterminer les états de la macroplace dans lesquels il va considérer les erreurs.

La réactivité sera considérée ici aussi comme le temps nécessaire entre le moment où l'exception apparaît et le moment où le fonctionnement de la micro-machine est réinitialisé. Cela signifie que nous ne considérons pas le temps nécessaire pour remettre la micro-machine en off une fois son fonctionnement arrêté (i.e le temps nécessaire pour

réinitialiser des modèles de référence, envoyer les messages d'erreur au contrôleur et réinitialiser les registres communs). Avec la macroplace, la réactivité est toujours d'1 période d'horloge. Sans macroplace, quand l'erreur est traitée, le fonctionnement de la micro-machine est réinitialisé en une période d'horloge. Le manque de réactivité vient du fait qu'il faut attendre qu'un état, où l'exception peut être gérée, soit atteint. Lorsque l'exception intervient pendant la mesure d'impédance, l'attente peut être de plus de 350 périodes d'horloge. En stimulation, par contre, cette attente est limitée à 36 périodes d'horloge et ce temps maximum ne peut être obtenu que dans le cas où l'exception provient au tout début du mode stimulation (pendant la configuration de l'électrode virtuelle) donc à un moment où le patient n'est normalement pas stimulé. Pendant que le patient est effectivement stimulé, l'attente sera toujours inférieure à 10 périodes d'horloge. Néanmoins, la macroplace permet une réaction toujours plus rapide (en 1 période d'horloge) et le temps de réaction est toujours fixe, ce qui est plus simple à gérer et plus sûr.

Pour conclure sur la partie conception, sur les 2 modèles (avec et sans macroplace), il a été nécessaire de mettre dans l'interface du composant modèle de référence toutes les places du comportement de son composant pour pouvoir le réinitialiser en cas d'exception. Ceci illustre le problème exposé au chapitre 3 et montre la nécessité d'avoir un autre mécanisme de gestion d'exception au niveau architectural.

Implémentation

Comme pour la micro-machine de première génération, le modèle sans macroplace a été transformé en code VHDL à l'aide du logiciel HILECOP et ce code VHDL a été modifié à la main pour le modèle avec macroplace. L'implémentation des deux modèles a alors été effectuée sous Libero à la fois pour simulation et pour test sur FPGA.

Les priorités n'ont été utilisées que lorsque c'était nécessaire au bon fonctionnement du modèle : des priorités ont ainsi été définies entre les transitions exceptions du modèle avec macroplace. Elles n'ont pas été utilisées systématiquement car le modèle de départ ne contenait pas de conflits effectifs : les conflits étaient gérés avec des conditions mutuellement exclusive ou des arcs inhibiteurs. Le marquage maximum de chaque place a été défini à 1.

Le comportement obtenu avec les deux modèles a pu être observé par simulation sous Libero et testé sur un FPGA. Le micro-programme utilisé pour les simulations est le suivant :

- une instruction de stimulation (SITa) avec I = 1950 μA et T = 100 μs
- une instruction de décharge active (SITa) avec I = -975 μA et T = 100 μs
- une instruction de stimulation (SITb) avec I = 2340 μA et T = 150 μs
- une instruction de décharge active (SITb) avec I = -585 μA et T = 150 μs
- une instruction de boucle (DT2L) indiquant qu'il y a aura 5 répétitions des instructions ci-dessus avec une décharge de 250 μs entre chaque répétition

Plusieurs scénarios ont été testés : le fonctionnement normal, la demande d'un stop, une erreur de stimulation et une demande de stop simultanée avec une erreur de stimulation. Ces simulations et tests sur prototype ont permis d'observer que le comportement attendu de la micro-machine n'a pas été affecté par l'introduction de la macroplace. Des simulations ont été réalisées en plus du test, car les tests sont réalisés sur l'USR complète.

Il n'est alors pas possible d'accéder directement aux signaux internes de la micro-machine et il n'est donc pas possible, par exemple, de savoir exactement quand les requêtes sont prises en compte.

La différence de réactivité entre les 2 implémentations est visible, par exemple sur le scénario de stop. Deux cas différents ont été testés (cf. figure 4.21 et figure 4.22). Sur ces figures, les signaux de type $MM\_...$ sont les signaux actions associés aux places du modèle portant le même nom. Les signaux $Visu\_Off$ et $Visu\_Analyse\_instruction$ permettent de visualiser respectivement le marquage de la place p_MM_Off et $p\_Analyse\_instruction$. Dans le premier cas (cf. figure 4.21), le stop est demandé $1\mu s$ après le start. La différence de réactivité avec et sans macroplace est alors de $36\mu s$. Mais ce cas n'est pas réaliste car une demande de stop ne peut pas être effectuée aussi rapidement par le contrôleur. Dans le deuxième cas, plus réaliste, le stop est demandé $192\mu s$ après le start (cf. figure 4.22). La différence de réactivité n'est alors plus que d' $1\mu s$ dans ce cas. Tous les cas n'ont pas été testés et surtout nous n'avons utilisé qu'un seul micro-programme, il est donc possible que cette différence soit plus grande même en phase de stimulation.

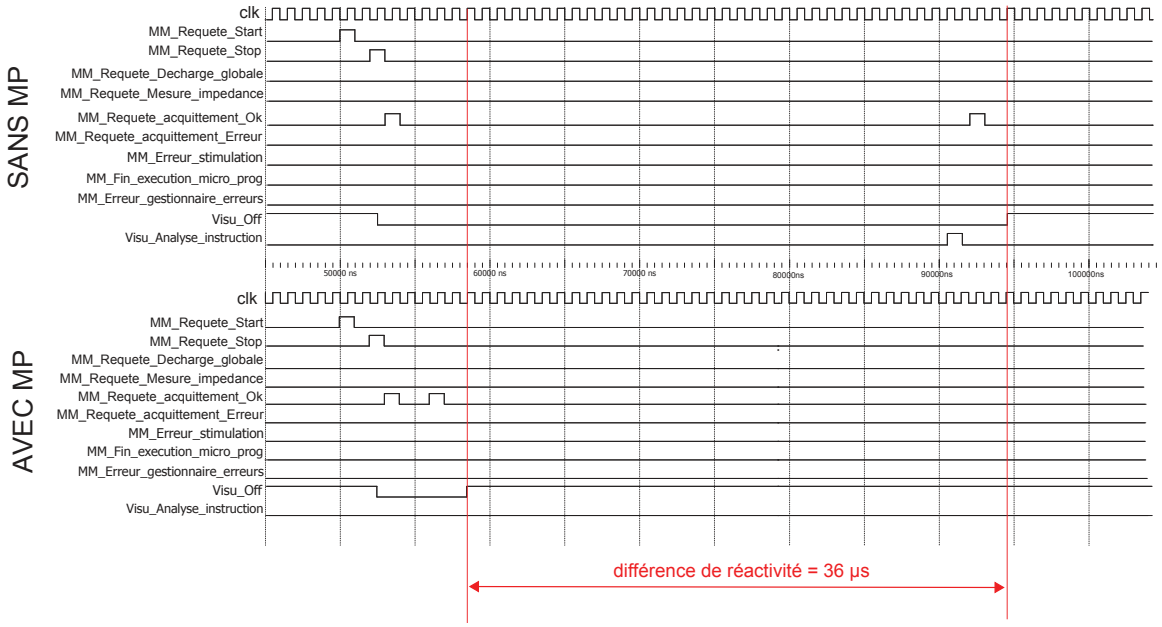

FIGURE 4.21 – Simulation dans le cas d'un stop demandé 1 μs après le start

Pour le modèle sans macroplace, l'implémentation sur un FPGA nécessite 3 340 cellules logiques tandis que pour le modèle avec macroplace 3 093 cellules logiques sont nécessaires. L'utilisation de la macroplace n'a donc dans ce cas aucun impact négatif sur le critère de la surface nécessaire pour l'implémentation du modèle.

Des mesures de consommation ont été réalisées par un ingénieur Axonic - MXM sur une carte accueillant un FPGA Igloo AGL1000v2 en boîtier 281CS. L'ASIC de génération du courant de stimulation n'est pas installé sur la carte pour cette série de tests. Le courant du "Core" du FPGA est mesuré à l'aide d'un multimètre professionnel Fluke 27II. Afin d'envoyer des requêtes particulières à la micro machine, un programme de test permet d'envoyer des trames d'un PC vers l'USR au travers d'un lien série (d'où l'utilisation d'une petite carte séparée accueillant un CP2102).

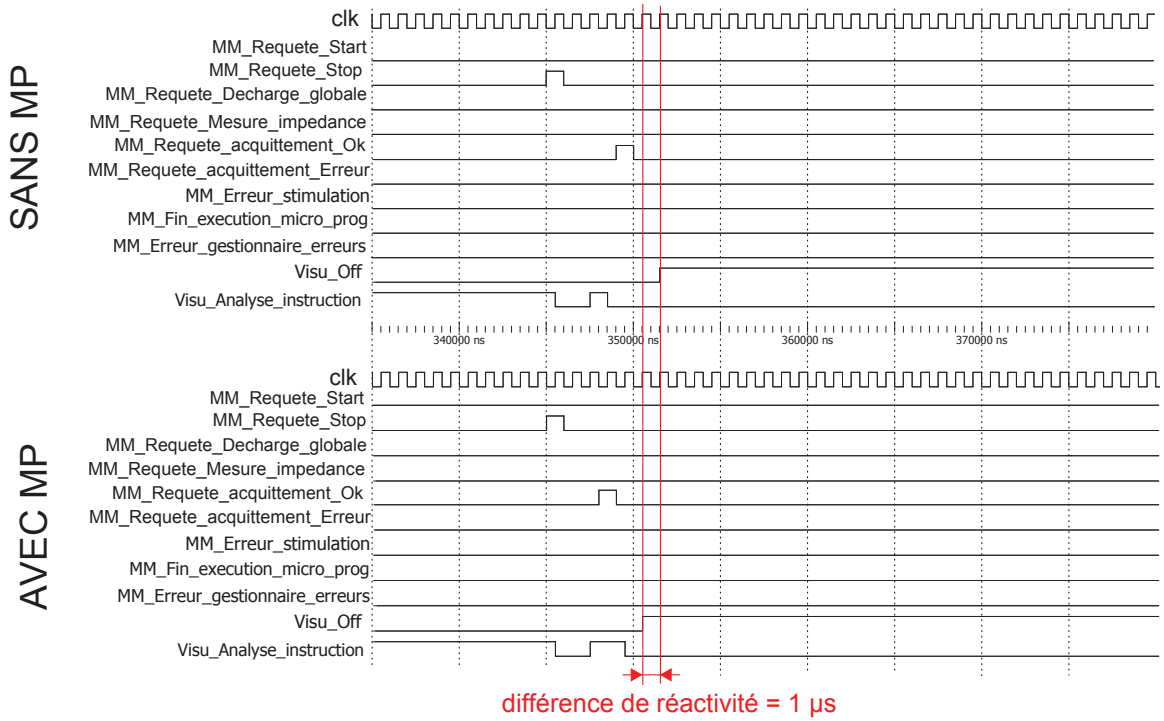

FIGURE 4.22 – Simulation dans le cas d'un stop demandé 192 μs après le start

Les consommations mesurées sont du même ordre de grandeur pour les deux modèles : 244,4 μA sans la macroplace et 247,0 μA avec la macroplace.

Analyse

Pour pouvoir analyser les modèles avec et sans macroplace de la micro-machine, il est nécessaire de modéliser sur les deux modèles la communication de la micro-machine avec le réseau. Les places modélisant l'arrivée des requêtes sont marquées par d'autres composants. L'arrivée des requêtes de start, stop, de mesure d'impédance ou de décharge globale est alors modélisée, pour chaque requête, par une place et une transition temporelle avec un intervalle temporel de la forme $[1, +\infty[$ liés par un arc P-T classique. Un arc entre les transitions temporelles ajoutées et les places modélisant la demande d'une requête est ajouté. Il est ainsi possible de modéliser l'arrivée d'une requête n'importe quand.

De même, les places indiquant qu'une requête est effectuée ou qu'une erreur s'est produite sont vidées par d'autres composants. Une transition avec un intervalle temporel $[1, 1]$ est alors ajoutée pour chaque place et un arc classique P-T est ajouté pour lier les places concernées aux transitions ajoutées. Il est donc supposé que les requêtes sont tout de suite traitées par les autres composants.

En ce qui concerne la transformation du modèle analysable, la majorité des transitions non temporelles, auxquelles été associées des transitions, étaient dans des conflits où les conditions garantissaient qu'une transition serait toujours tirable. Ainsi l'intervalle $[1, +\infty[$ a été associé à seulement 4 transitions : t_SITa, t_SITr, t_DT2L et t_CMZ, transitions qui déterminent le type d'instruction à déchiffrer. Si nous étudions le code VHDL, les conditions de ces transitions garantissent que l'une des transitions sera toujours tirable mais il est impossible de le garantir sans regarder le code VHDL. C'est pourquoi le choix a été fait de garder l'intervalle $[1, +\infty[$.

Aucune transition ne peut être bloquée car aucune transition temporelle n'a de condition associée.

Les scénarios testés pour l'analyse sont : une requête de décharge globale (D), une requête de mesure d'impédance (M), une requête de Start (Sta) et une requête de Start et une requête de Stop (Sta-Sto). Les résultats obtenus sont donnés dans le tableau 4.3.

	sans MP			avec MP		
scénario	états	transitions	états morts	états	transitions	états morts
D	615	2 773	1	2 176	6 931	1
M	69 339	309 483	1	181 035	560 368	1
Sta	220 387	906 949	1	324 432	1 191 182	1
Sta-Sto	797 042	3 503 627	1	1 476 218	5 816 909	1

TABLE 4.3 – Taille des graphes des classes accessibles obtenus pour la micro-machine de 3ème génération

Il y a toujours un état mort dans les scénarios étudiés car une seule requête est considérée. Une fois celle-ci traitée, le modèle n'évolue plus. Dans chacun des scénarios, le graphe des classes d'états est environ 2 fois plus important pour le modèle avec macroplace que pour le modèle sans. Comme précédemment, cette différence dans la complexité s'explique notamment par l'ajout de la macroplace. Mais aussi par le fait que la gestion des exceptions peut être réalisée n'importe quand dans le modèle avec macroplace alors qu'elle n'est gérée que dans certains états sans macroplace.

Bilan

Ainsi avec un traitement des exceptions sans macroplace très différent les résultats obtenus sont équivalents à ceux obtenus sur la micro-machine de 1ère génération (cf tableau 4.4) : la macroplace a un impact très positif sur la conception, neutre sur l'implémentation et négatif pour la complexité de l'analyse.

	Conception			
modèle	places	transitions	arcs	réactivité
MM	80	133	649	>350
MMMP	79	105	384	1
gain	**18%**	**21%**	**41 %**	–
	Implémentation		Analyse	
modèle	surface	consommation	états	transitions (Sta-Sto)
MM	3 340	244,4 μA	797 042	3 503 627
MMMP	3 093	247,0 μA	1 476 218	5 816 909
gain	**7,4%**	**-1%**	**-85%**	**-66%**

TABLE 4.4 – Comparaison des résultats obtenus sur les 2 modèles pour la micro-machine de 3ème génération

4.4 Conclusion

Le mécanisme de la macroplace a ainsi pu être utilisé pour décrire le comportement de la micro-machine de deux stimulateurs de différentes générations. Le gain apporté par la macroplace du point de vue de la complexité de la conception et du modèle est indéniable sur les deux modèles. La difficulté d'être exhaustif en décrivant la gestion des exceptions a pu être observée puisque dans les deux exemples considérés, des cas avaient été oubliés. Il aurait été intéressant de pouvoir comparer le temps nécessaire à l'élaboration des modèles avec et sans macroplace mais malheureusement il est difficile d'avoir une donnée fiable. En effet, pour la pertinence de la comparaison, il faudrait que ce soit la même personne qui réalise le modèle avec macroplace et celui sans. Dans ce cas quand cette personne réalisera le second modèle, elle bénéficiera de la réflexion réalisée pour élaborer le premier modèle et sera donc plus rapide.

Un des inconvénients de la macroplace, rencontré lors de l'élaboration des modèles avec macroplace, est qu'il n'est possible d'entrer ou de sortir de la macroplace qu'à l'aide de transitions. Nous avons donc du sortir de la macroplace des transitions alors qu'il aurait été logique de les mettre dans la macroplace. Le problème est peut-être accentué par le fait que nous souhaitons incorporer une macroplace dans un modèle qui n'a pas été pensé au départ pour l'utilisation de ce mécanisme.

Le comportement général des modèles avec et sans macroplace sont identiques. Néanmoins, dans les modèles avec macroplace, les exceptions sont gérées dans tous les états tandis que, dans ceux sans macroplace, elles ne sont gérées qu'une fois un état prédéfini atteint. Il aurait été impossible de gérer les exceptions dans tous les cas, sans macroplace. Il est nécessaire de garder cette différence en tête pour étudier les résultats obtenus d'un point de vue implémentation et analyse.

En ce qui concerne l'implémentation, les modèles avec macroplace nécessitent la même surface et consomment autant que les modèles sans macroplace. La macroplace nous permet donc d'avoir un modèle plus réactif sans augmenter son coût en surface ou en consommation.

Les modèles avec macroplace ont pu être analysés mais ils sont plus complexes à analyser que les modèles sans macroplace. La taille du graphe des classes d'états reste néanmoins dans le même ordre de grandeur (environ 2 fois plus grand). L'analyse de modèles de taille industrielle reste néanmoins difficile et il a été nécessaire de faire des hypothèses pour pouvoir obtenir des résultats. En outre, tout ce qui est modélisé à l'aide des conditions en VHDL ne peut pas être analysé. Or le choix est parfois offert au concepteur de décrire des éléments de l'interprétation influant sur le comportement à l'aide de l'interprétation ou à l'aide d'une structure de RdP. Par exemple, les compteurs peuvent être modélisés, soit par des fonctions décrites en VHDL, soit par une place de RdP. Pour obtenir le plus d'informations sur le comportement du modèle à l'aide de l'analyse formelle, il est plus judicieux d'exprimer à l'aide du RdP tout ce qui peut l'être mais au prix d'un modèle plus complexe.

Conclusion et perspectives

Conclusion

Nous avons abordé dans ce mémoire la conception de systèmes numériques complexes, à travers la méthodologie HILECOP. Les principaux objectifs des travaux présentés étaient la correspondance d'un point de vue comportemental entre les différents modèles utilisés dans la méthodologie et l'intégration d'un mécanisme de gestion efficace des exceptions.

Avoir une correspondance d'un point de vue comportemental signifie minimiser l'écart entre le comportement conçu, le comportement du système réel et celui analysé. Le but est d'assurer, dans la mesure du possible, que le comportement du système réel soit celui spécifié par le concepteur et que les résultats d'analyse obtenus soient pertinents par rapport au comportement du système réel.

Quant au mécanisme de gestion efficace des exceptions, le but est non seulement de faciliter le travail de conception et de limiter ainsi le risque d'erreurs humaines mais aussi de permettre une implémentation efficace du modèle conçu à l'aide de ce mécanisme et d'avoir un modèle toujours analysable. Ceci est nécessaire pour garantir que la méthode soit utilisable dans le contexte des systèmes embarqués critiques où les contraintes sont très fortes notamment sur la fiabilité de la conception et sur les critères de surface et de consommation du système réel.

Se préoccuper de la correspondance entre modèle conçu, modèle implémentable et modèle analysable nous a amené à modifier dans la méthodologie HILECOP, aussi bien la définition du formalisme que les transformations automatiques du modèle conçu en un modèle implémentable et un modèle analysable :

- La définition formelle et les règles sémantiques des RdP GEITSP ont ainsi été modifiées pour prendre en compte le fait que le modèle sera implémenté de manière synchrone.
- Le déterminisme du comportement est désormais assuré de manière automatique grâce à la mise en place de priorités en cas de conflits.
- La garantie d'un fonctionnement correct sur le FPGA est obtenue notamment grâce au dimensionnement par l'analyse formelle du marquage maximal des places, quand cela est possible.
- Les intervalles temporels du modèle analysable sont déterminés pour prendre en compte non seulement le retrait de l'interprétation par rapport au modèle conçu mais aussi l'implémentation synchrone du modèle.
- Le risque de blocage d'une transition temporelle à laquelle une condition est associée est désormais intégré dans le modèle analysable.

Le modèle analysable est tel qu'il est possible désormais de garantir que le comportement du modèle conçu est inclus dans celui du modèle analysable. Ainsi si les propriétés d'invariance sont vérifiées dans le modèle analysable, elles le seront aussi sur le modèle conçu. L'implémentation étant réalisée pour traduire fidèlement le comportement du modèle conçu, les propriétés seront aussi vérifiées sur le comportement du système réel. Ainsi l'analyse formelle du modèle analysable nous permet d'avoir des informations pertinentes sur le comportement du modèle réel.

Par contre, il n'y a pas équivalence stricte entre le comportement du modèle analysable et celui du système réel. Cette équivalence ne peut pas être obtenue à cause de l'interprétation. Certains comportements seront ainsi observables sur le modèle analysable alors qu'ils ne pourront jamais se produire dans le système réel. Ainsi, par exemple, si une propriété d'accessibilité est vraie sur le modèle analysable, elle ne le sera pas nécessairement sur le système réel. De même, une propriété d'invariance peut être fausse sur le modèle analysable mais vraie sur le système réel. Par contre, si une propriété d'accessibilité est fausse ou une propriété d'invariance vraie sur le modèle analysable, elle le sera nécessairement également sur le système réel.

En outre, l'analyse est réalisée sur un réseau de Petri exécuté de manière asynchrone alors que le système réel fonctionne de manière synchrone. Or, dans le cas général, la vivacité sur un réseau de Petri asynchrone ne permet pas de garantir la vivacité du même réseau de Petri exécuté de manière synchrone. L'étude de la vivacité du modèle analysable devra donc aussi être réalisée avec précaution pour s'assurer de la validité des résultats sur le système réel.

Pour conclure sur le premier objectif, les travaux présentés ont permis de fiabiliser l'implémentation du modèle et d'obtenir un modèle analysable plus pertinent par rapport au modèle conçu, dans le sens où il garantit la conservation des propriétés d'invariance et réduit, dans une certaine mesure, le risque d'explosion combinatoire. Les limites de la pertinence des résultats obtenus par analyse formelle sont de plus désormais connues.

En ce qui concerne la gestion des exceptions, il a été montré qu'elle devait être gérée de manière plus pratique et efficace aussi bien au niveau architectural qu'au niveau comportemental pour permettre de satisfaire les contraintes des systèmes embarqués critiques. Au niveau architectural, une piste de solution a été proposée et développée sur le cas d'une architecture numérique dirigée par une seule horloge. Au niveau comportemental, pour gérer les exceptions, le mécanisme de la macroplace a été retenu et adapté aux contraintes fonctionnelles et non-fonctionnelles des systèmes embarqués critiques.

Ce mécanisme permet non seulement de spécifier le comportement en cas d'exceptions de manière plus simple, mais permet aussi d'avoir un modèle conçu plus lisible et plus compact. Ceci permet notamment de réduire le risque d'erreurs humaines. Une solution pour implémenter de manière efficace, c'est-à-dire en conservant l'apport en termes de compacité de la macroplace, a été proposée et cette dernière prend en compte les contraintes de surface et de consommation rencontrées dans le cadre des systèmes embarqués. En outre, une transformation automatique du modèle avec macroplace a été établie pour pouvoir obtenir un modèle analysable par les outils existants. La correspondance comportementale entre modèle conçu, modèle implémentable et modèle analysable a bien entendu

été conservée lors de l'introduction de ce nouveau mécanisme dans la méthodologie. Les transformations du modèle conçu en modèle implémentable et en modèle analysable sont toujours automatiques non seulement pour éviter tout risque d'erreur humaine lors de la transformation mais aussi pour diminuer le temps nécessaire pour concevoir un système.

L'apport de la macroplace a pu être validé sur des modèles industriels. L'apport de la macroplace sur ces modèles en termes de compacité, de facilité de conception et de gain en réactivité est avéré sur les exemples considérés. De plus, sur ces modèles, l'utilisation de la macroplace n'a pas eu d'impact négatif sur les critères de surface et de consommation, ces critères étant primordiaux pour déterminer l'efficacité de l'implémentation d'un modèle. Les modèles avec macroplace ont aussi pu être analysés à l'aide d'un outil d'analyse existant (TINA), par contre cette analyse s'avère souvent plus complexe que pour les modèles sans macroplace dans le sens où le graphe des classes d'états obtenu est généralement plus important avec la macroplace que sans. Cette augmentation est notamment due au fait que le modèle avec macroplace gère les exceptions dans plus de cas que le modèle sans macroplace. Il existe donc plus de comportements possibles et l'analyse est donc plus complexe. En outre, même sans macroplace, il est difficile d'obtenir des résultats d'analyse formelle sur les modèles de taille industrielle et des simplifications doivent être réalisées au niveau du modèle.

Perspectives

Les travaux présentés ont permis d'améliorer la méthodologie HILECOP. Plusieurs points peuvent néanmoins être encore développés que ce soit au niveau de chacune des étapes de la méthodologie ou de la méthodologie en général pour rendre la méthodologie plus pratique, plus efficace ou plus complète.

Intégrer et améliorer la macroplace

Tout d'abord, la suite logique des travaux présentés dans ce manuscrit consiste à intégrer les modifications proposées pour la méthodologie dans l'outil HILECOP, et notamment l'ajout du mécanisme de la macroplace. Cet ajout permettra notamment d'avoir des retours des ingénieurs utilisant le logiciel et de confirmer la "praticité" de la macroplace pour des ingénieurs qui maîtrisent la méthodologie HILECOP mais qui ne sont pas familiers avec ce mécanisme. Utiliser la macroplace dans de nombreux modèles permettrait de s'assurer que les ingénieurs parviennent à exprimer facilement tout ce dont ils ont besoin avec ce mécanisme. Proposer des solutions pratiques à l'usage a, en effet, été une préoccupation permanente dans ces travaux et l'avis des utilisateurs est donc indispensable pour valider définitivement nos solutions.

Une amélioration possible de la macroplace est déjà envisageable après son utilisation sur les deux modèles industriels étudiés dans ce manuscrit. Dans ces travaux, une transition exception est tirée, du point de vue de la macroplace, dès que la macroplace est active et quel que soit son marquage. Pouvoir vider la macroplace quel que soit son marquage exact était le but recherché. Mais parfois certaines parties du modèle ne peuvent pas être arrêtées à n'importe quel instant. Par exemple, si des données sont en train d'être écrites en mémoire comme ce peut être le cas dans la micro-machine de première génération, il est inopportun d'interrompre l'écriture et il serait donc nécessaire d'attendre que l'écriture soit finie.

Aujourd'hui deux solutions s'offrent au concepteur :

- La transition exception est reliée à la macroplace par un arc exception dont la situation serait $(*, \text{situation assurant que la macroplace peut être vidée})$ au lieu d'utiliser simplement $(*)$. La transition exception ne sera alors plus sensibilisée quand, par exemple, l'écriture sera en cours. Elle ne pourra donc pas être tirée et ainsi interrompre l'écriture. Cette solution ne peut par contre pas être utilisée s'il est important de connaître le temps depuis lequel la macroplace est active (cas de la modélisation d'un watchdog par exemple) ou dans le cas particulier où au moins deux situations différentes mais incompatibles (qui ne peuvent pas être décrites toutes les deux par une même situation de sortie) doivent empêcher le tir d'une transition exception.
- La sous-partie du réseau qui ne doit pas être stoppée sans précaution n'est pas placée dans la macroplace.

L'idée serait de pouvoir différer la gestion de l'exception quand le système se trouve dans certains états sans modifier la sensibilisation de la transition exception.

Dimensionner le nombre de compteurs nécessaires pour les transitions temporelles

Pour l'instant, l'analyse formelle, d'un point de vue dimensionnement, n'est utilisée que pour déterminer le marquage maximum des places. L'analyse formelle pourrait aussi être utilisée pour déterminer le nombre de compteurs nécessaires pour les transitions temporelles. En effet, pour l'instant, un compteur est associé à chaque transition temporelle. Or les compteurs peuvent nécessiter une surface non négligeable lors de l'implémentation si les bornes des intervalles temporels sont grands. L'idée serait de réduire le nombre de compteurs, et donc la surface nécessaire pour l'implémentation, en établissant le nombre de compteurs nécessaires. En effet, si deux transitions temporelles ne sont jamais sensibilisées en même temps, leurs compteurs ne sont jamais utilisés en même temps et il est alors possible de n'avoir qu'un seul compteur pour les deux transitions. L'analyse structurelle, et plus particulièrement l'analyse des invariants de transitions, peut nous permettre de déterminer les transitions qui ne seront jamais sensibilisées simultanément.

Etendre les travaux aux architectures GALS et multi-FPGA

Dans ces travaux, seul le cas d'une architecture implémentée sur un même FPGA et avec une même horloge pour tous les composants constituant l'architecture a été considéré. Or la méthodologie HILECOP vise également à permettre au concepteur de réaliser des architectures de type GALS (Globalement Asynchrone Localement Synchrone) qu'il s'agisse de composants fonctionnant à des horloges différentes au sein d'un même FPGA ou de composants déployés sur des FPGA différents (mais interconnectés). La conception de ce type d'architecture est déjà possible sur le modèle conçu puisque chaque composant a sa propre entrée horloge.

Une perspective importante pour la méthodologie HILECOP est d'adapter la transformation automatique du modèle conçu en modèle implémentable et en modèle analysable pour pouvoir prendre en compte les architectures de type GALS (mono-FPGA ou multi-FPGA). La correspondance entre les 3 modèles devra bien entendu être conservée ce qui sera la principale difficulté. Deux cas doivent être considérés : le cas où les différentes horloges des composants sont basées sur une même horloge circuit (et donc synchronisées entre elles) et le cas où elles sont basées sur des horloges circuits différentes et ne sont

donc pas synchronisées ensemble (et il y a donc notamment un risque de dérive entre ces horloges). La solution proposée doit permettre de gérer ces deux cas.

L'interconnexion entre les composants, dans le cadre d'une architecture GALS, doit être réalisée avec plus de précautions que dans les travaux présentés. Il n'est, par exemple, pas possible de fusionner deux transitions (ou deux places) appartenant à deux composants ne fonctionnant pas sur la même horloge. En effet, les deux comportements du modèle n'évoluent pas à la même vitesse. Or l'arc de fusion exprime pour l'instant que les deux nœuds du RdP reliés par cet arc sont les mêmes et une même transition (ou place) ne peut pas évoluer à deux vitesses différentes. Il serait imaginable de réaliser un arc de connexion entre deux places ou deux transitions mais il faudrait alors définir avec précision le comportement asynchrone correspondant si les deux nœuds ne sont pas dirigés par la même horloge.

Pour ce qui est de l'interconnexion entre nœuds de RdP par arcs classiques, cela demande aussi certaines précautions. Par exemple, lorsqu'une transition de l'interface doit vérifier si elle est tirable, elle considère les signaux des sensibilisations de toutes les places amont (qu'elles soient internes ou externes au composant). Si elle est tirée, l'information de tir sera donnée à toutes les places amont et aval. Les places amont et aval actualisent leur marquage suivant l'horloge de leur composant. Or une transition dirigée par une horloge plus rapide que celles de ses places amont ou aval peut être tirée plusieurs fois avant que ses places amont ou aval puissent mettre à jour leur marquage. Il faudra alors non seulement définir le comportement attendu dans ce cas, mais aussi être capable de l'obtenir sur la cible et de l'analyser.

L'idée serait de s'appuyer sur la notion de connecteur des langages à composants. Le connecteur spécifiera l'interaction entre les composants et pourra être traduit, à la fois pour l'implémentation ou pour l'analyse, différemment suivant le type d'architecture réalisée (i.e. selon une relation synchrone ou asynchrone).

Développer la gestion des exceptions au niveau architectural

Dans ces travaux, l'observation et le contrôle des composants n'ont été proposés que pour les situations : composant dans son état intial ou composant vide. Pouvoir observer ou imposer un marquage quelconque est une possibilité à considérer car intéressante dans certains cas. En effet, ne pouvoir observer ou imposer que la situation initiale ou la situation vide est assez limité. Le concepteur peut vouloir imposer un marquage spécifique pour, par exemple, gérer le mode dans lequel un composant est utilisé. Le problème est alors que le composant contrôleur doit nécessairement connaître les places du composant contrôlé. Il faut alors distinguer deux cas : pouvoir imposer le marquage d'une ou plusieurs places d'un composant ou pouvoir imposer le marquage de toutes les places du composant. Si le concepteur souhaite pouvoir imposer le marquage de toutes les places (et que ce marquage n'est pas le marquage vide ou le marquage initial), il faudrait alors définir dans le composant contrôlé (ou observé) le marquage spécifique et le composant contrôleur (ou observateur) ne connaîtrait que ces différentes possibilités de marquages spécifiques qui peuvent être imposés (ou observés) sans connaître bien sûr le modèle interne du composant, conformément au principe des approches à composants. Cela permettrait notamment de contrôler ou d'observer le "mode" de fonctionnement courant du composant concerné. Si seules quelques places du composant doivent être contrôlées (ou observées), il serait

alors envisageable de placer ces places dans l'interface de contrôle comme cela est fait pour l'interface classique. Le choix entre les deux solutions serait laisser à l'appréciation du concepteur.

De même, il pourrait être intéressant de pouvoir imposer un marquage dynamique à un composant. Cela permet de pouvoir reconfigurer les composants en cas de problème lors de l'utilisation du système réel. Le problème est que, si le concepteur peut imposer n'importe quel marquage, il peut imposer un marquage qui crée un blocage, ne respecte pas les contraintes de sécurité du système ou ne respecte pas les contraintes non-fonctionnelles du système (comme le marquage maximum des places). Si la possibilité d'imposer des marquages dynamiques est laissée au concepteur, il faudra que le concepteur s'assure que le marquage imposé fait partie des marquages analysés et garantit le bon fonctionnement du système.

La possibilité d'imposer un marquage statique ou dynamique peut faciliter le travail des concepteurs mais présente aussi des inconvénients du point de vue de l'approche à composants et peuvent nuire à la fiabilité du système. Il faudra donc étudier si le bénéfice apporté en vaut la peine, notamment suivant le contexte dans lequel la méthodologie HILECOP est utilisée.

Il sera naturellement nécessaire de développer les solutions proposées pour pouvoir implémenter et analyser des interactions de type observation et contrôle d'un composant dans le cadre des architectures de type GALS et multi-FPGA.

Améliorer l'analyse formelle Nous avons vu que l'analyse formelle des modèles de taille industrielle s'avère souvent trop complexe pour obtenir les résultats voulus. Une première solution serait d'utiliser, ou d'adapter si besoin, les méthodes de réduction déjà développées dans le cadre des RdP T-temporel asynchrone [48] qui permettent de diminuer la complexité de l'analyse tout en garantissant la conservation de certaines propriétés de l'analyse formelle (comme la caractère borné ou vivant d'un modèle). De plus, les méthodes de réduction consistent à supprimer les places ou transitions du modèle qui sont "inutiles" du point de vue de l'analyse formelle. Comme nous avons notamment besoin d'informations sur chacune des places du modèle (marquage maximum), l'idée serait de préserver la partie du modèle sur laquelle nous souhaitons récolter des informations et de réduire le reste du modèle.

Par ailleurs, la méthodologie HILECOP se base pour l'instant sur les outils d'analyse existants. L'inconvénient est que les outils existants ne permettent d'analyser que des réseaux de Petri ne comprenant pas de macroplaces et asynchrones.

Dans ce mémoire, une solution est proposée pour pouvoir remettre à plat un modèle contenant des macroplaces. L'avantage est de pouvoir utiliser les outils existants mais cela crée un graphe des classes d'états plus important que nécessaire. Si les outils d'analyse existants étaient modifiés pour pouvoir intégrer la macroplace, le graphe des classes d'états obtenu serait significativement réduit. En effet, la purge d'une macroplace nécessite aujourd'hui au moins autant d'états que le nombre de places à vider. En intégrant la macroplace à l'outil d'analyse, un seul état serait nécessaire.

Le fait qu'aucun outil ne permette d'analyser les réseaux de Petri de manière synchrone est assez problématique. En effet, il a déjà été montré que la propriété de vivacité n'est pas conservée quand un modèle synchrone est analysé de manière asynchrone. Or l'information de vivacité est particulièrement intéressante et nécessaire puisqu'elle permet de s'assurer que le modèle ne sera jamais dans un état où il ne pourra plus évoluer. L'importance d'analyser les réseaux de Petri synchrones de manière synchrone est connue depuis longtemps puisque elle est déjà expliquée par Moalla dans [43]. Pourtant, à notre connaissance, aucun outil n'a pour l'instant été développé dans ce sens.

Une autre piste de solution, pour obtenir de meilleurs résultats au niveau analyse comportementale du système implémenté, serait de transformer notre réseau de Petri en un langage synchrone (comme ESTEREL [17]) qui pourrait être analysé, ou de décrire directement le modèle à l'aide d'un langage synchrone. La méthodologie HILECOP pourrait alors proposer différentes générations de codes en sortie suivant les besoins du concepteur.

Rendre la méthodologie plus complète

Comme évoqué au chapitre 1, la méthodologie HILECOP ne permet pas pour l'instant de réaliser complètement le cycle en V mais seulement les 2 dernières étapes. Il serait notamment intéressant d'avoir une méthodologie qui puisse prendre en compte la description des spécifications et la vérification du respect de celles-ci (en termes de propriétés étudiées en model-checking). La méthodologie Hiles [33] peut apporter des pistes intéressantes de réflexion, d'autant plus que dans cette méthodologie, la réalisation de la partie basse du cycle en V est inspirée de la méthodologie d'HILECOP. Une alternative serait d'intégrer la méthodologie HILECOP à un environnement de conception de systèmes critiques plus large tel que TOPCASED.

Annexe A

Glossaire sur les RdP

Un réseau de Petri (RdP) est un réseau composé de places et de transitions reliées par des arcs orientés et pondérés. Un arc est dit pondéré quand un poids, c'est-à-dire un entier naturel non nul, est associé à l'arc.

Une **place** est représentée par un cercle et peut contenir des jetons. Le **marquage** d'une place est défini par le nombre de jetons que contient la place. Le marquage du RdP est le vecteur contenant le marquage de chacune des places du RdP.

Une **transition** est représentée par un trait ou par un rectangle.

Il existe 3 types d'arcs :

- les **arcs classiques** qui peuvent soit relier une place à une transition (PT) soit une transition à une place (TP). Ils sont représentés par une flèche. Quand une place et une transition sont reliées par un arc classique (PT) de poids n, la transition n'est sensibilisée par la place que si le marquage de la place est supérieur ou égal à n. Le tir de la transition entraîne le retrait de n jetons de la place. Quand une transition est reliée à une place par un arc classique TP de poids n, le tir de la transition ajoute n jetons à la place.
- les **arcs tests** qui relient nécessairement une place à une transition. Ils sont représentés par un trait terminé par un rond plein. Quand une place et une transition sont reliées par un arc test de poids n, la transition n'est sensibilisée par la place que si le marquage de la place est supérieur ou égal à n. Le tir de la transition n'a aucun impact sur le marquage de la place.
- les **arcs inhibiteurs** qui relient nécessairement une place à une transition. Ils sont représentés par un trait terminé par un rond vide. Quand une place et une transition sont reliées par un arc inhibiteur de poids n, la transition n'est sensibilisée par la place que si marquage de la place est strictement inférieur à n. Le tir de la transition n'a aucun impact sur le marquage de la place.

Un RdP est **généralisé** lorsque les poids associés aux arcs sont des entiers naturels non nuls. Il est dit ordinaire quand les poids ne peuvent être égaux qu'à 1. Un RdP est **sauf** si le marquage de chaque place est toujours au maximum égal à 1. Un RdP sauf est nécessairement ordinaire.

Un RdP est dit **étendu** quand les trois types d'arcs sont utilisés.

Un RdP est dit **interprété** quand les interactions entre le modèle et son environnement, portées par l'interprétation, sont décrites sur le RdP. Dans notre cas, l'interprétation consiste en des conditions, des actions et des fonctions (cf. chapitre 1).

Un RdP est dit **synchrone** quand l'évolution du RdP est cadencée par une horloge. Le tir des transitions n'est dans ce cas possible que sur un événement de l'horloge. Dans notre cas, une transition en peut être tirée que sur le front descendant de l'horloge.

Un RdP est dit **T-temporel** quand un intervalle temporel peut être associé aux transitions. Associer un intervalle de type $[t_{min}, t_{max}]$ signifie qu'une transition ne peut être tirée que quand son compteur de temps local est compris entre t_{min} et t_{max}. Le compteur d'une transition commence à s'incrémenter quand une transition est sensibilisée par toutes ses places entrantes. Il s'incrémente de 1 à chaque unité de temps. Le compteur est réinitialisé si la transition est tirée ou si elle n'est plus sensibilisée. La borne maximum de l'intervalle temporel peut-être infinie.

Un RdP est dit **à priorités** quand des priorités peuvent être définies entre les transitions. Si deux transitions sont simultanément tirables et qu'une priorité est définie entre les transitions, c'est la plus prioritaire qui est tirée.

Un RdP est **vivant** si et seulement si toutes les transitions du RdP peuvent toujours être ultérieurement franchies à partir d'un état accessible quelconque du RdP.

Un RdP est **réversible** si l'état initial peut toujours être de nouveau atteint à partir de n'importe quel état accessible du RdP.

Un RdP est **borné** si toutes les places sont bornées. Une place est **bornée** si et seulement si le nombre de jetons contenu dans la place est toujours fini quelle que soit l'évolution du RdP.

Une transition t_i est en **conflit effectif** avec une transition t_j si et seulement si les transitions t_i et t_j sont simultanément tirables et que le tir de t_j désensibilise la transition t_i.

Deux transitions sont dites **en concurrence** si elles sont simulaténement tirables mais qu'elles ne sont pas en conflit effectif.

L'**analyse structurelle** consiste, comme son nom l'indique, à analyser la structure du RdP. Elle peut permettre de déterminer si le RdP est vivant, réversible ou borné.

L'**analyse d'accessibilité** consiste à établir l'ensemble des marquages accessibles pour les RdP non-temporels ou l'ensemble des classes d'états accessibles pour les RdP temporels. Une classe d'état est définie par un marquage du RdP et des valeurs de compteurs pour les transitions temporelles. A partir de ces graphes, il est possible, comme pour l'analyse structurelle, d'établir si le RdP est vivant, réversible ou borné. Il est par contre aussi possible de réaliser du model-checking.

Le **model-checking** consiste à vérifier des propriétés comportementales. Deux types de propriétés sont distinguées :

- les propriétés d'invariance qui vérifient si une propriété est vraie sur tous les états du graphe ou sur tous les chemins du graphe (donc que la propriété sera toujours vraie).
- les propriétés d'accessibilité qui vérifient s'il existe au moins un état du graphe ou au moins un chemin du graphe sur lequel une propriété est vraie.

Il existe différents langages pour décrire les propriétés. Les plus courants sont le LTL (Logique Temporelle Linéaire), CTL (Computation Tree Logic ou logique du temps arborescent) et TCTL (Timed Computation Tree Logic ou logique temporisée du temps arborescent). Seul le TCTL permet de décrire des propriétés temporisées, c'est-à-dire des propriétés incluant des données temporelles quantitatives du type : "La transition t_i est toujours tirée au plus tard 3 unités de temps après la transition t_j". Néanmoins l'outil d'analyse TINA ne permet de vérifier que des propriétés décrites en LTL. Pour pouvoir étudier des propriétés temporisées, une solution est alors d'utiliser des observateurs. Les **observateurs** sont des places et des transitions ajoutées au RdP qui ne servent qu'à l'analyse formelle. Elles sont liées au RdP de sorte qu'elles ne modifient pas l'évolution du RdP mais qu'elles puissent observer le temps depuis lequel un événement a eu lieu ou le temps entre l'apparition de deux événements. Pour vérifier la propriété donnée en exemple précédemment, l'observateur à ajouter est donné figure A.1. Une fois cet observateur ajouté, pour vérifier si la propriété est vraie, il suffit alors de vérifier que la place $p_{observateur2}$ n'est jamais marquée.

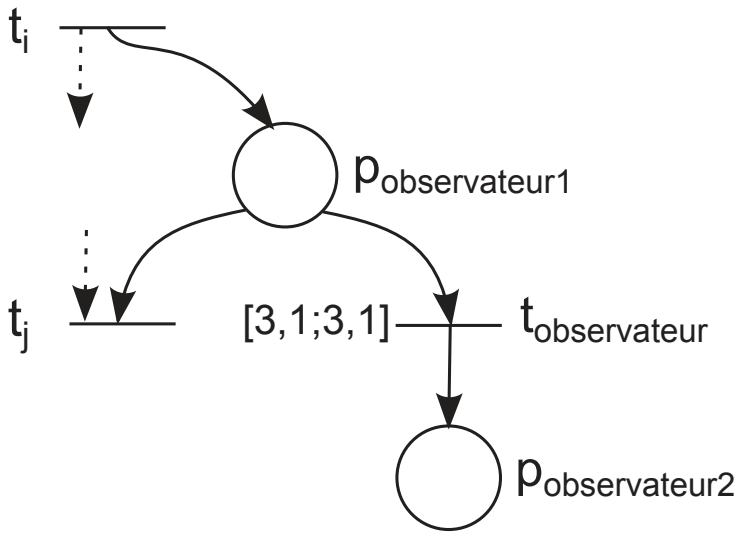

FIGURE A.1 – Exemple d'observateur

Annexe B

Preuve de l'inclusion du comportement du modèle conçu dans celui du modèle analysable

B.1 Cas d'un RdP GEIS sans conflit effectif

Nous souhaitons démontrer que l'évolution du RdP GEIS est incluse dans celle du RdP GET.

Soit $\omega = \omega_0 \dots \omega_n$ une séquence de tirs de transitions sous la forme $\omega_i = (T_i, \theta_i)$ où $T_i \subseteq T$ est l'ensemble des transitions tirées à l'instant θ_i. Comme le RdP GEIS est synchrone tandis que le RdP GET est asynchrone, les marquages accessibles ne sont pas exactement les mêmes (cf. figure B.1). Par contre nous pouvons démontrer que l'exécution de la même séquence de tirs dans les deux RdP mène aux mêmes marquages stables. Un marquage $m(\theta)$ d'un RdP est dit stable pour l'instant θ si et seulement si le marquage du RdP ne peut plus évoluer sans faire évoluer le temps. Dans les deux modèles, les marquages ne se stabilisent pas au même moment, nous les comparerons donc quand ils seront tous les deux stabilisés (cf B.2). Sur le système réel, seuls les marquages stables seront visualisés.

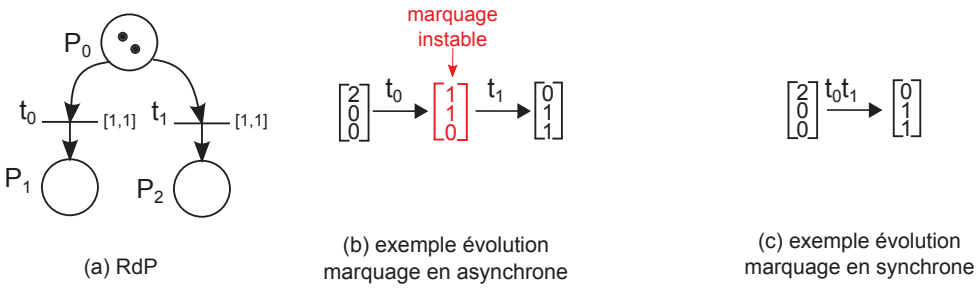

FIGURE B.1 – Différences entre l'évolution dans un RdP asynchrone et un RdP synchrone

Pour faciliter la comparaison des séquences de tirs, même dans les RdP asynchrones, les séquences de tirs sont décrites sous la forme $\omega = \omega_0 \dots \omega_n$ avec $\omega_i = (T_i, \theta_i)$ où $T_i \subseteq T$ est l'ensemble des transitions tirées à l'instant θ_i. Dans le cas des RdP asynchrones, les transitions du groupe T_i sont quand même tirées les unes après les autres. Nous n'aurons jamais besoin de l'ordre de tir exact des transitions, car nous ne nous intéressons qu'aux

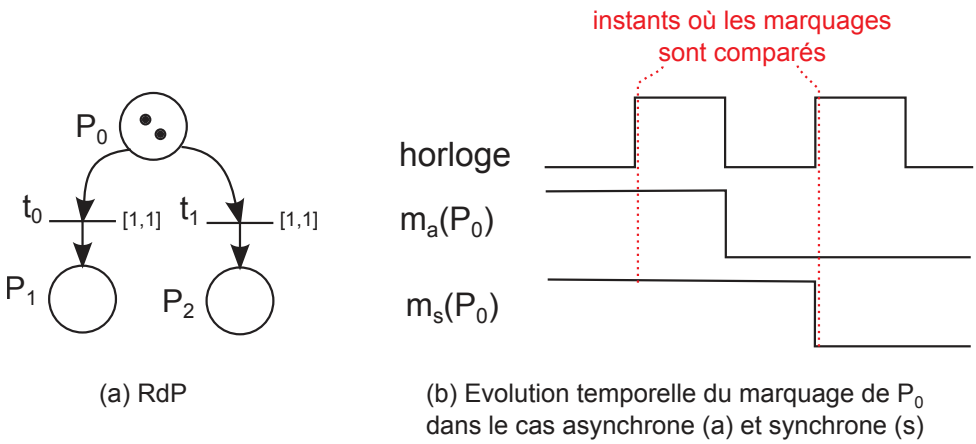

FIGURE B.2 – Instants où les marquages stables sont comparés

marquages stables et l'absence de conflits effectifs garantissent que toutes les transitions tirables à un instant θ le resteront même si d'autres transitions tirables sont tirées. C'est pourquoi nous pouvons nous permettre de regrouper les transitions.

B.1.1 Démonstration de l'équivalence des marquages stables pour une même séquence de tirs

Soit N_s le RdP GEIS conçu et N_a le RdP GET construit à partir de N_s. Soit m_a le marquage de N_a et m_s celui de N_s. Nous souhaitons donc démontrer, par récurrence sur θ, que si ω est une séquence de tirs possible dans N_s et dans N_a, alors on a $\forall\theta \in [\![0,+\infty[\![, m_s(\theta+\frac{1}{2}) = m_a(\theta+\frac{1}{2})$ avec l'hypothèse suivante : à $\theta = 0$, on a $\downarrow(clock) = 1$. Pour qu'une séquence de tirs soit possible dans N_s et dans N_a, les instants où sont tirées les transitions sont nécessairement de la forme θ avec $\theta \in \mathbb{N}$ puisqu'il ne peut y avoir de tir que sur les fronts descendants dans N_s.

Par définition de N_a, on a $m'_0 = m_0$. Par définition de N_s, aucune transition ne peut être tirée à $\theta = 0$ et pour qu'une séquence de tir soit possible dans N_s, il ne peut pas y avoir tir de transition pendant $\theta = \left[0, 0+\frac{1}{2}\right]$. On a donc bien $m'(0+\frac{1}{2}) = m(0+\frac{1}{2})$. La propriété est bien vérifiée au rang 0.

Nous supposons que la propriété est vraie au rang n. Nous supposons donc que $m'(n+\frac{1}{2}) = m(n+\frac{1}{2})$ et nous voulons montrer que $m'(n+1+\frac{1}{2}) = m(n+1+\frac{1}{2})$.

- Dans N_s : suivant la définition de la sémantique formelle, la prochaine évolution sera faite au prochain front montant de l'horloge donc à l'instant $n+1$. Les transitions tirées sont définies dans la séquence de tir ω_s. On a $\omega_{sn} = (T_{sn}, n)$. Par définition des RdP GEIS, on a donc $m_s(n+1+\frac{1}{2}) = m_s(n+\frac{1}{2}) - \sum_{t\in T_n} Pre(t) + \sum_{t\in T_n} Post(t)$.
- Dans N_a : comme ω est nécessairement tel qu'il n'y a des tirs de transitions que sur les instants de la forme k avec $k \in \mathbb{N}$ pour pouvoir être exécuté dans N_s, on sait que le marquage n'évoluera qu'à l'instant n. A cet instant, les transitions incluses dans T_n seront tirées. L'ordre de tir n'importe pas puisque nous avons posé l'hypothèse qu'il n'y a pas de conflit effectif dans le RdP. Ainsi le tir d'une transition ne pourra

pas empêcher le tir des transitions suivantes. A chaque tir de transition, le marquage évolue depuis m_i vers m_j avec $\forall t \in T_n, m_j = m_i - Pre(t) + Post(t)$. On a alors $m_a(n+1) = m_a(n+\frac{1}{2}) - \sum_{t\in T_n} Pre(t) + \sum_{t\in T_n} Post(t)$ et comme il n'y aura pas de nouveau tir avant $n+\frac{3}{2}$ par définition de ω, on a $m_a(n+1+\frac{1}{2}) = m_a(n+1)$

On a donc bien $m_a(n+1+\frac{1}{2}) = m_s(n+1+\frac{1}{2})$, la propriété est donc vraie au rang n+1. Ainsi nous avons pu montrer que si nous trouvons la même séquence de tirs dans les deux réseaux alors le comportement au sens de l'évolution du marquage sera le même pour les marquages stables. Il reste maintenant à démontrer que nous pouvons toujours trouver une telle séquence.

B.1.2 Démonstration qu'une séquence de tirs possible dans le modèle conçu est toujours possible dans le modèle analysable

Soit Ω_s l'ensemble des séquences de tir ω possibles dans N_s et Ω_a l'ensemble de celles possibles dans N_a. Pour prouver l'inclusion de l'évolution de N_s dans celle de N_a, il faut montrer que $\forall \omega_s \in \Omega_s, \exists \omega_a \in \Omega_a | \omega_s = \omega_a$. Nous supposons toujours qu'à $\theta = 0$, on a $\downarrow clock = 1$.

$\forall \omega_s \in \Omega_s$, on a $\omega_s = \omega_{s0} \dots \omega_{sn}$ où $\omega_{si} = (T_i, i)$ avec $T_i \subseteq T$. On peut potentiellement avoir $T_i = \varnothing$. On a $\omega_a = \omega_s \Leftrightarrow \forall i \in [\![0, n]\!], \omega_{ai} = \omega_{si}$. Démontrons alors la propriété voulue par récurrence sur les composantes ω_{si} d'une séquence de tirs possible de N_s.

Pour alléger les notations, nous noterons $cond(C(t)) = 1$ quand $(\forall c \in \mathcal{C} | C(t)(c) = 1, cond(c) = 1) \wedge (\forall c \in \mathcal{C} | C(t)(c) = -1, cond(c) = 0)$. Nous noterons de même $cond(C(t)) = 0$ quand $(\exists c \in \mathcal{C} | C(t)(c) = 1 \wedge cond(c) = 0) \vee (\exists c \in \mathcal{C} | C(t)(c) = -1 \wedge cond(c) = 1)$.

Montrons d'abord la propriété au rang 0, c'est-à-dire $\forall \omega_s \in \Omega_s, \exists \omega_a | \omega_{a0} = \omega_{s0}$.

A $\theta = 0$, dans N_s, on a, par hypothèse, $T_{s0} = \varnothing$. Dans N_a, comme $\forall t \in T_a, \downarrow Is(t) = 1$, on a aussi nécessairement $T_{a0} = \varnothing$. Ainsi on a bien $T_{s0} = T_{a0}$ et donc $\forall \omega_s \in \Omega_s, \exists \omega_a | \omega_{a0} = \omega_{s0}$. La propriété est donc vraie au rang 0.

Supposons maintenant la propriété vraie au rang N_s et prouvons qu'elle est alors vraie au rang $n+1$. Nous supposons donc que $\forall \omega_s \in \Omega_s, \exists \omega_a | \omega_{a0} \dots \omega_{an} = \omega_{s0} \dots \omega_{sn}$. Montrons qu'il existe alors $\omega_{a(n+1)} | \omega_{a0} \dots \omega_{a(n+1)} = \omega_{s0} \dots \omega_{s(n+1)}$.

Soient $sens_s(n+1)$ et $sens_a(n+1)$ les groupes de transitions toujours sensibilisées après les tirs de transitions réalisés à l'instant $\theta = n+1$ et qui l'étaient déjà à l'instant $\theta = n$. Soient $\uparrow sens_s(n+1)$ et $\uparrow sens_a(n+1)$ les groupes des transitions nouvellement sensibilisées par les tirs de transitions réalisés à l'instant $\theta = n+1$.

Dans N_s, les nouveaux tirs se déroulent au plus tôt à $\theta = n+1$. A $\theta = n+1$, on a :
- $\forall t \in \uparrow sens_s(n+1), C(t) = 0 \Rightarrow t \in T_{s(n+1)}$
- $\forall t \in (sens_s(n+1) \cup \uparrow sens_s(n+1)) | C(t) \neq 0, cond(C(t)) = 1 \Rightarrow t \in T_{s(n+1)}$
- $\forall t \in (sens_s(n+1) \cup \uparrow sens_s(n+1)) | C(t) \neq 0, cond(C(t)) = 0 \Rightarrow t \notin T_{s(n+1)}$

Considérons maintenant le RdP N_a.

Nous avons montré précédemment que si la même séquence de tirs était exécutée dans N_s et dans N_a, il y avait égalité des marquages stables. Ainsi, en exécutant $\omega_s \in \Omega_s$, on a $m_a(n+\frac{1}{2}) = m_s(n+\frac{1}{2})$ mais on a aussi $\forall k \in [\![0,n]\!], m_a(k+\frac{1}{2}) = m_s(k+\frac{1}{2})$. Donc on a nécessairement $sens_a(n+1) = sens_s(n+1)$ et $\uparrow sens_a(n+1) = \uparrow sens_s(n+1)$ puisque la structure du réseau a été conservée et donc les conditions de sensibilisation sont identiques dans les 2 réseaux.

Considérons d'abord l'ensemble des transitions nouvellement sensibilisées.

$\forall t \in \uparrow sens_s(n+1), \exists \omega_a \in \Omega_a | t \in \uparrow sens_a(n+1)$.

- $C(t) = 0$ dans $N_s \Rightarrow Is_a(t) = [1,1]$. Donc $\theta = n+1 \Rightarrow I_a(t) = [0,0] \Rightarrow t \in T_{a(n+1)}$.
- $C(t) \neq 0$ dans $N_s \Rightarrow Is_a(t) = [\,1, +\infty\,[$. Donc $\theta = n+1 \Rightarrow I_a(t) = [\,0, +\infty\,[$. Donc $\exists \omega_a^1 = \omega_{a0} \dots \omega_{an} \omega^1_{a(n+1)} \in \Omega_a | t \in T_{a(n+1)} \wedge \exists \omega_a^2 = \omega_{a0} \dots \omega_{an} \omega^2_{a(n+1)} \in \Omega_a | t \notin T_{a(n+1)}$.

Considérons maintenant l'ensemble des transitions sensibilisées qui ne sont pas nouvellement sensibilisées.

$\forall t \in sens_s(n+1), \exists \omega_a \in \Omega_a | t \in sens_a(n+1)$. On a nécessairement $C(t) \neq 0$ (sinon la transition aurait été tirée précédemment) donc $Is_a(t) = [\,1, +\infty\,[$. Comme t n'est pas nouvellement sensibilisée à $\theta = n$, on a nécessairement $\theta = n+1 \Rightarrow I_a(t) = [\,0, +\infty\,[$. Donc $\exists \omega_a^1 = \omega_{a0} \dots \omega_{an} \omega^1_{a(n+1)} \in \Omega_a | t \in T_{a(n+1)} \wedge \exists \omega^2 = \omega_{a0} \dots \omega_{an} \omega^2_{a(n+1)} \in \Omega_a | t \notin T_{a(n+1)}$.

Comme il n'y a pas de conflit effectif, on peut choisir, pour chaque transition qui n'est pas nécessairement tirée, si elle est tirée ou pas. Ce choix n'impacte pas sur le tir des autres transitions et ne peut pas entraîner le tir de nouvelles transitions non inclues dans $T_{a(n+1)}$ à l'instant $\theta = n+1$. En effet, si une transition est nouvellement sensibilisée par le tir d'une transition, comme on a $\forall t \in T_a, \downarrow Is_a(t) = 1$, elle ne pourra pas être tirée à l'instant $\theta = n+1$. De plus, il n'est pas possible d'être obligé de tirer une transition anciennement sensibilisée qui n'appartiendrait pas à $T_{a(n+1)}$ car, si une transition est sensibilisée et qu'elle n'appartient pas à $T_{a(n+1)}$, alors son intervalle temporel est nécessairement de la forme $[\,1, +\infty\,[$. Elle n'est donc jamais obligatoirement tirée.

Par conséquent, $\exists \omega_a \in \Omega_a | T_{a(n+1)} = T_{s(n+1)}$. Donc, $\forall \omega_s \in \Omega_s, \exists \omega_a \in \Omega_a | \omega_{a0} \dots \omega_{a(n+1)} = \omega_{s0} \dots \omega_{s(n+1)}$. La propriété est donc vraie au rang $n+1$.

Nous avons ainsi montré que $\forall \omega_s \in \Omega_s, \exists \omega_s | \omega_a = \omega_s$.

Nous avons donc montré que toute séquence de tirs possible dans N_s sera aussi possible dans N_a. De plus, si la même séquence de tirs est exécutée sur les deux RdP, les mêmes marquages stables seront observés. Nous pouvons donc en déduire que le comportement du modèle conçu (et donc du modèle implémenté) est bien inclus dans le modèle analysable.

B.2 Cas d'un RdP GEISP

Dans la preuve précédente de l'inclusion du comportement du modèle conçu dans celui du modèle analysable, nous avions utilisé l'hypothèse qu'il n'y avait pas de conflit effectif aussi bien pour l'égalité des marquages stables pour une même séquence de tirs que pour l'inclusion des séquences de tirs du modèle conçu dans celles du modèle analysable.

Cette hypothèse a cependant toujours été utilisée pour justifier que le tir d'une transition n'empêcherait jamais le tir d'autres transitions. Or dans le cadre d'un RdP GEISP, si l'hypothèse n'est plus vraie, l'argument est toujours vrai. En effet, la définition formelle des RdP GEISP garantit que des transitions sont tirées simultanément que si elles ne sont pas en conflit effectif. Le tir d'une transition du groupe de transitions tirées ne pourra donc toujours pas empêcher le tir d'une autre transition.

La preuve de l'inclusion du comportement pour les RdP GEISP est donc exactement la même que pour les RdP GEIS en modifiant le point évoqué ci-dessus.

B.3 Cas d'un RdP GEITSP

Dans le cas des RdP GEITSP, nous n'avons plus exactement la même structure entre le modèle conçu et le modèle analysable puisque des places, des transitions et des arcs sont ajoutés au modèle analysable pour modéliser le risque de blocage des transitions temporelles dans le modèle implémenté (pris en compte dans le modèle conçu). Il n'est donc plus possible de prouver l'égalité des marquages stables ou des séquences de tirs entre les 2 RdP. Il est néanmoins toujours possible de montrer l'égalité entre le marquage du modèle conçu et la restriction du marquage du modèle analysable aux places appartenant aussi au modèle conçu (donc toutes les places sauf celles modélisant le blocage). Il est possible de même de ne comparer que la partie comparable des séquences de tirs des deux réseaux (i.e. de restreindre les séquences de tirs du modèle analysable aux transitions ne modélisant pas le blocage).

La démonstration est alors réalisée sur le même modèle que pour le cas des RdP GEISP (ou RdP GEIS).

Pour les marquages stables, comme le modèle analysable est défini tel que $\forall t \in T_{blocage}, \forall p \in P, Pre(t)(p) = 0 \wedge Post(t)(p) = 0$, le marquage des places ne modélisant pas le blocage ne sera jamais affecté directement par le tir d'une transition modélisant le blocage. Il est alors facile de démontrer l'égalité des marquages stables entre les deux modèles si une même séquence de tirs (sans considérer les transitions modélisant le blocage) est appliquée sur les deux réseaux.

Pour l'inclusion des séquences de tirs, la démonstration est réalisée sur le même principe que précédemment sauf que l'on montre également que l'ensemble des transitions bloquées dans le modèle conçu peuvent également être bloquées dans le modèle analysable.

Annexe C

Modèle de la micro-machine de première génération sans macroplace

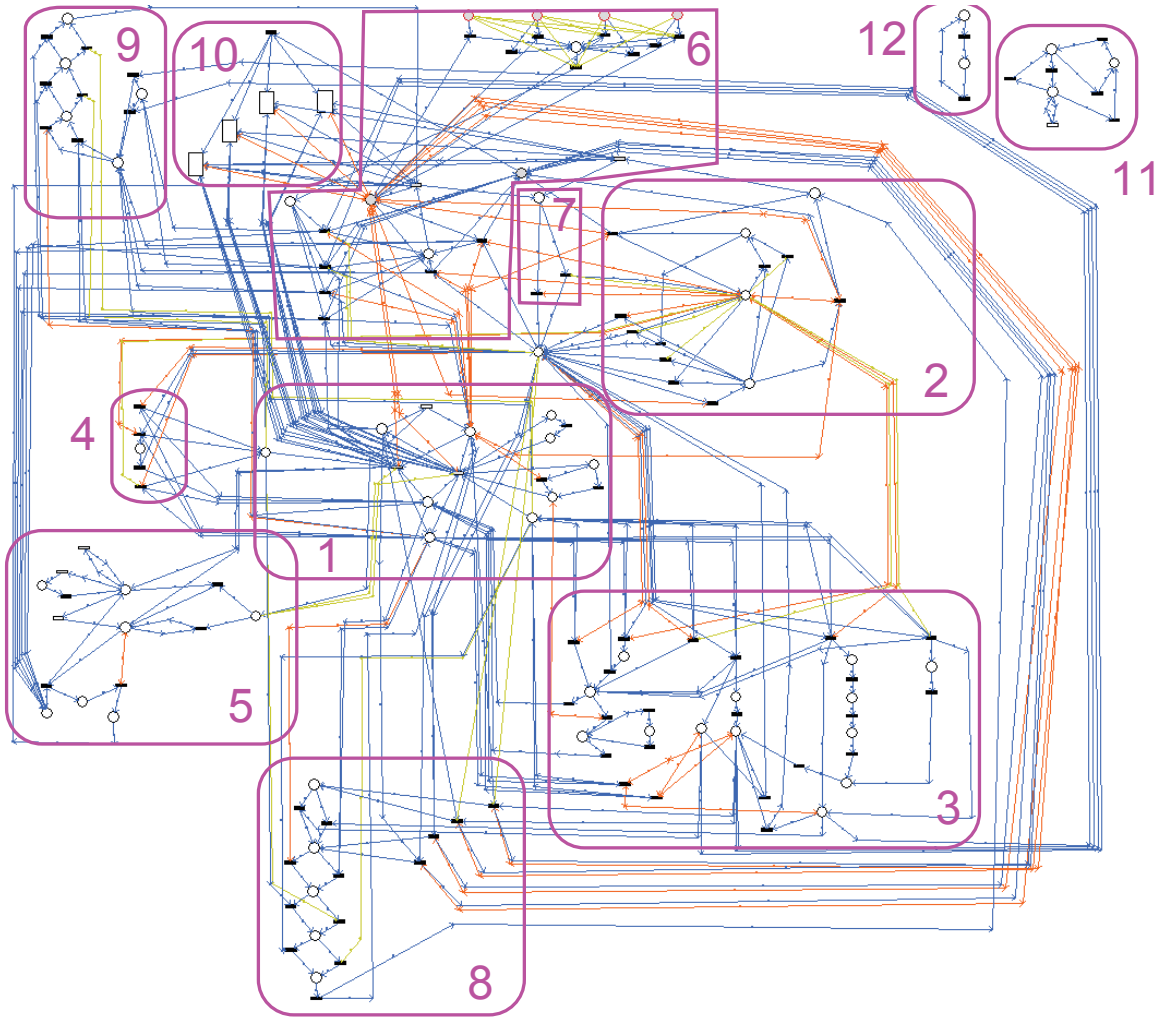

FIGURE C.1 – Modèle complet de la micro-machine de première génération

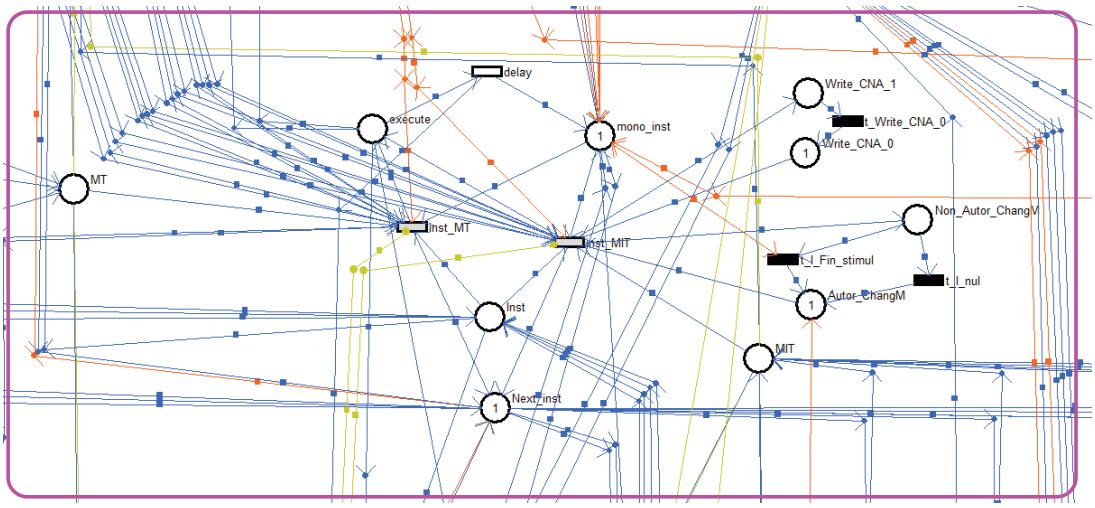

FIGURE C.2 – Partie 1 de la micro-machine de première génération

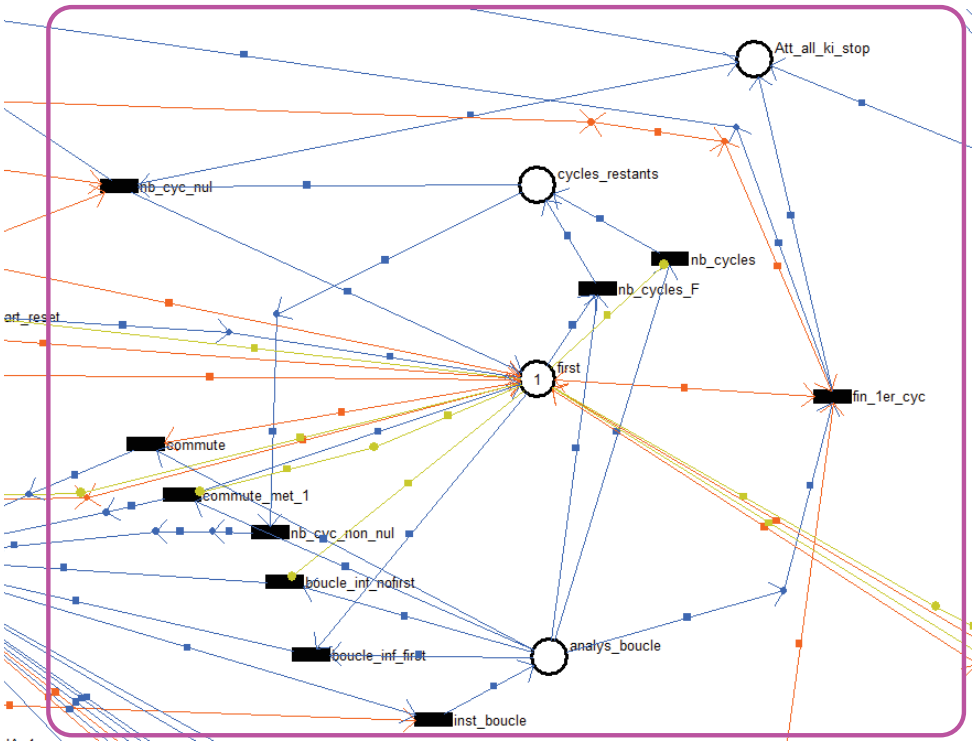

FIGURE C.3 – Partie 2 de la micro-machine de première génération

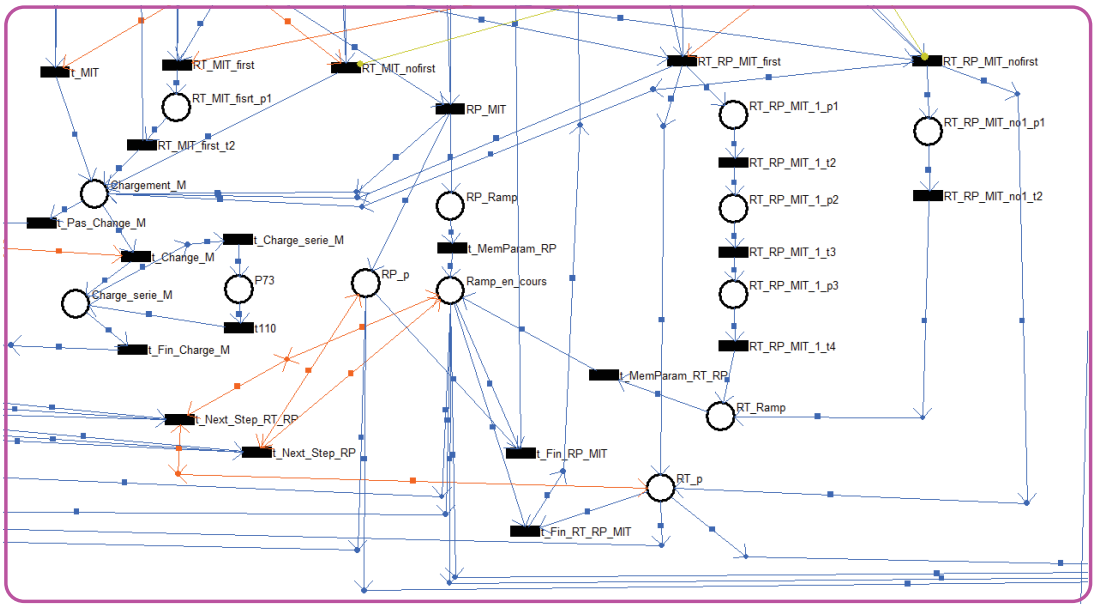

FIGURE C.4 – Partie 3 de la micro-machine de première génération

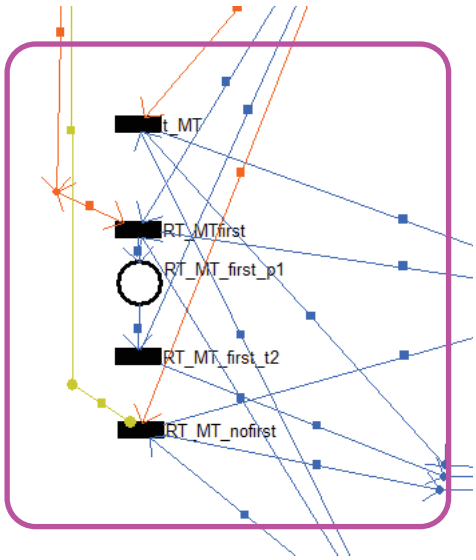

FIGURE C.5 – Partie 4 de la micro-machine de première génération

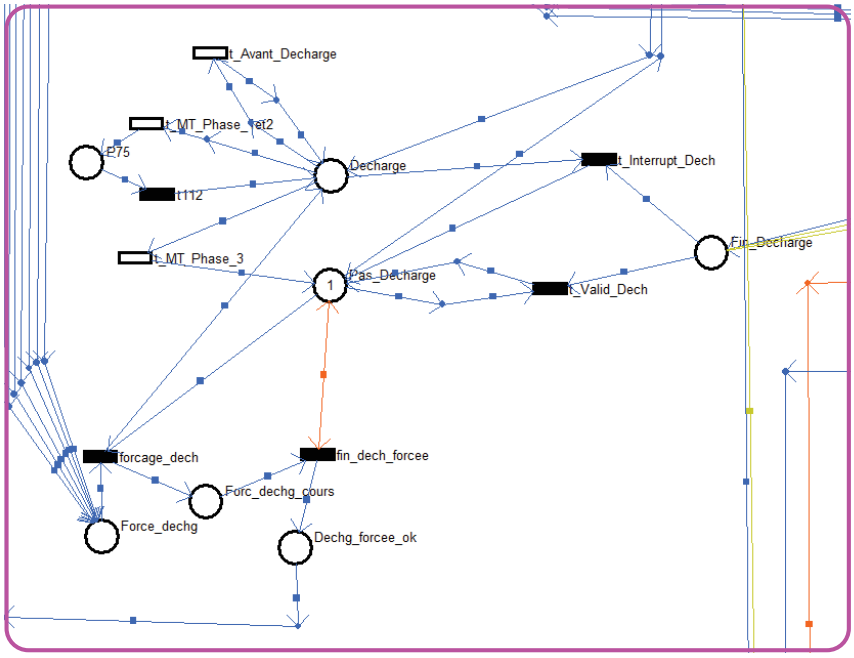

FIGURE C.6 – Partie 5 de la micro-machine de première génération

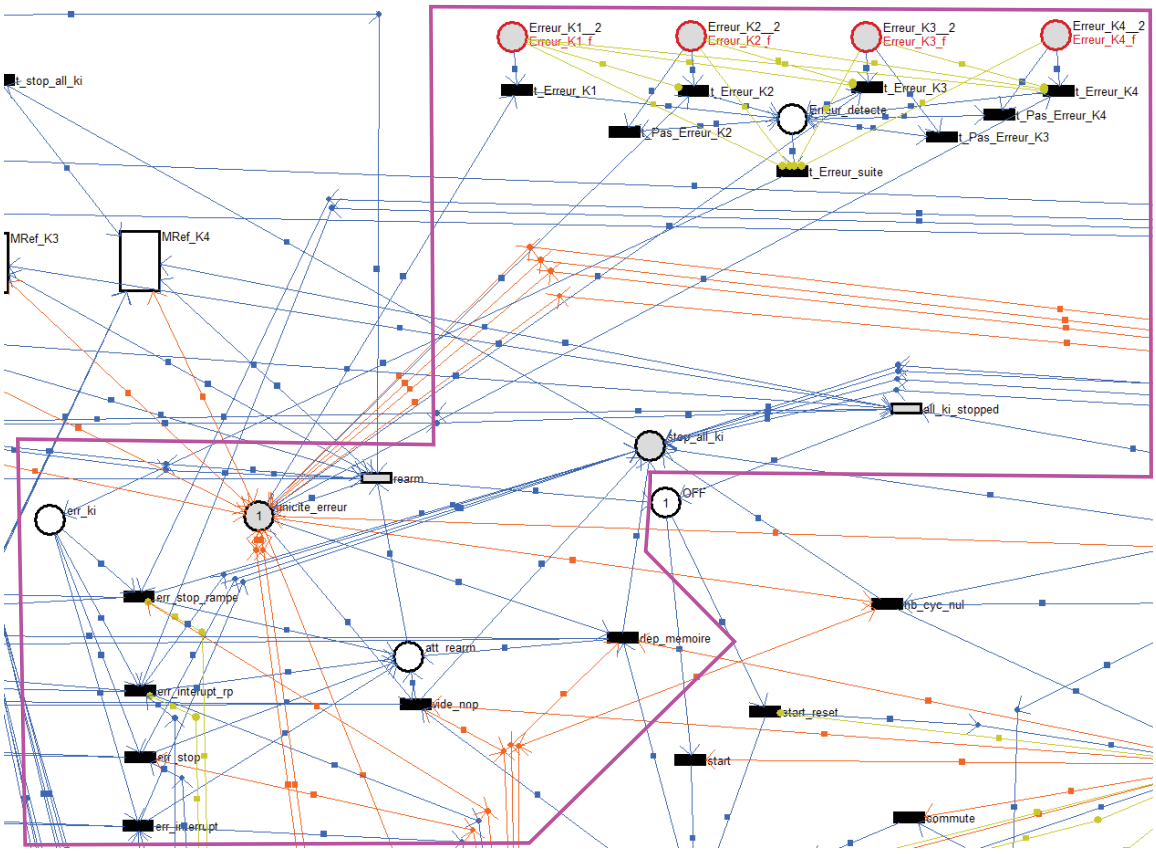

FIGURE C.7 – Partie 6 de la micro-machine de première génération

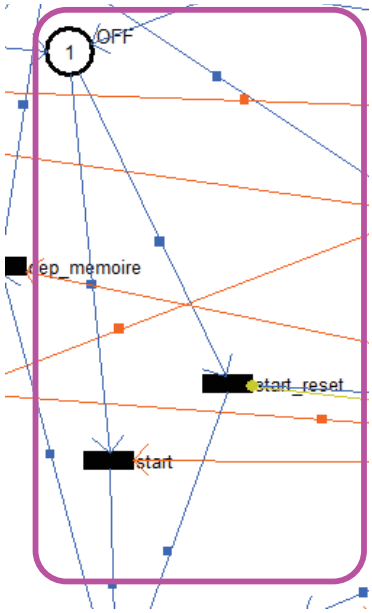

FIGURE C.8 – Partie 7 de la micro-machine de première génération

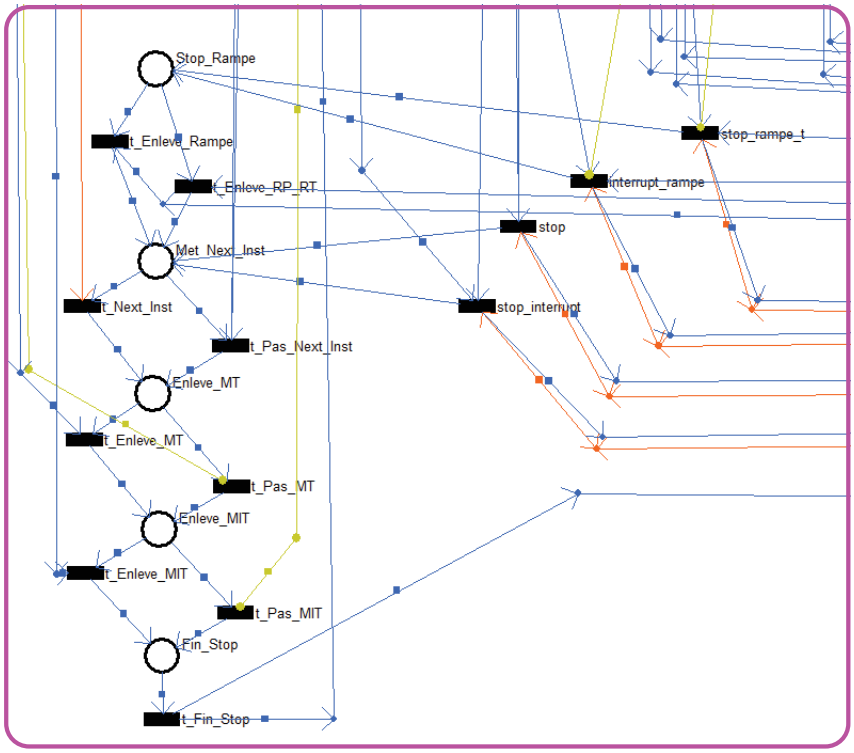

FIGURE C.9 – Partie 8 de la micro-machine de première génération

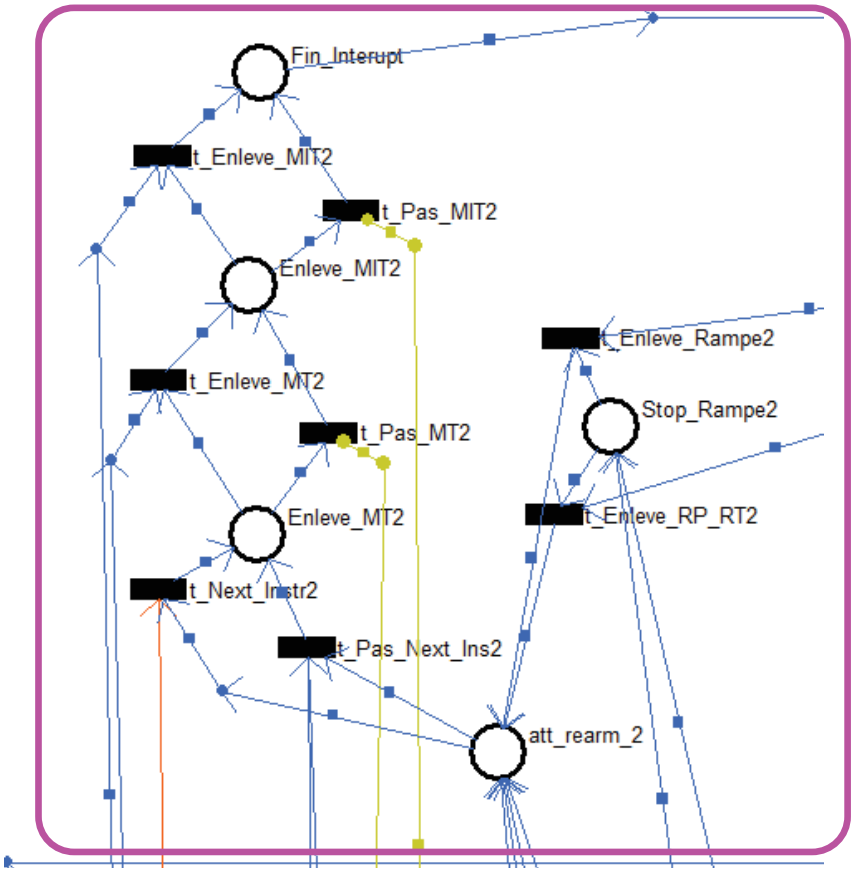

FIGURE C.10 – Partie 9 de la micro-machine de première génération

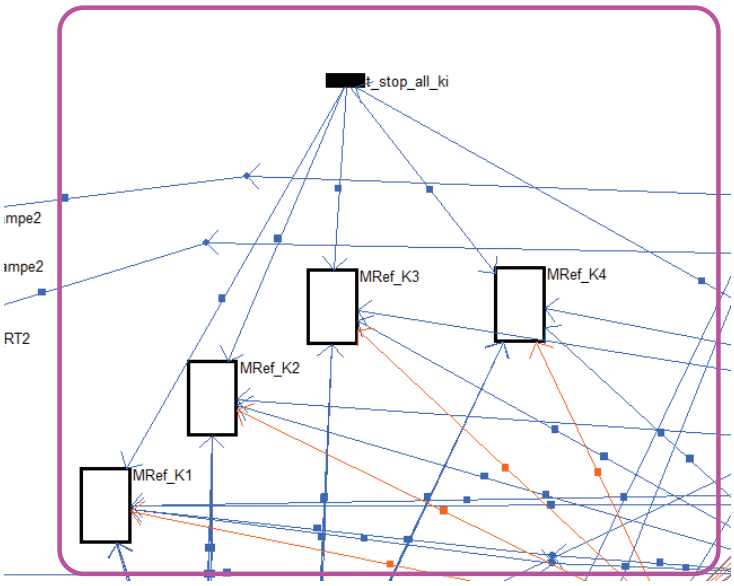

FIGURE C.11 – Partie 10 de la micro-machine de première génération

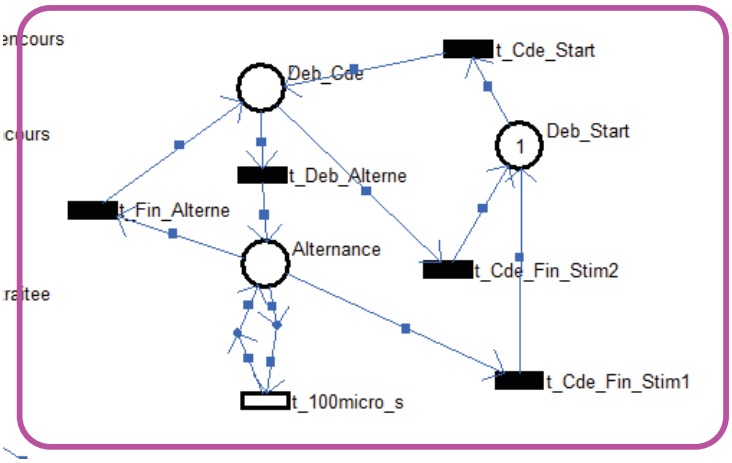

FIGURE C.12 – Partie 11 de la micro-machine de première génération

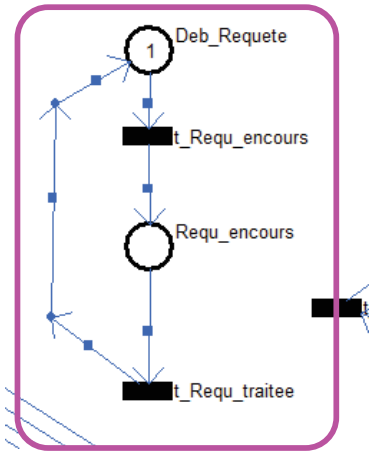

FIGURE C.13 – Partie 12 de la micro-machine de première génération

Annexe D

Modèle de la micro-machine de troisième génération sans macroplace

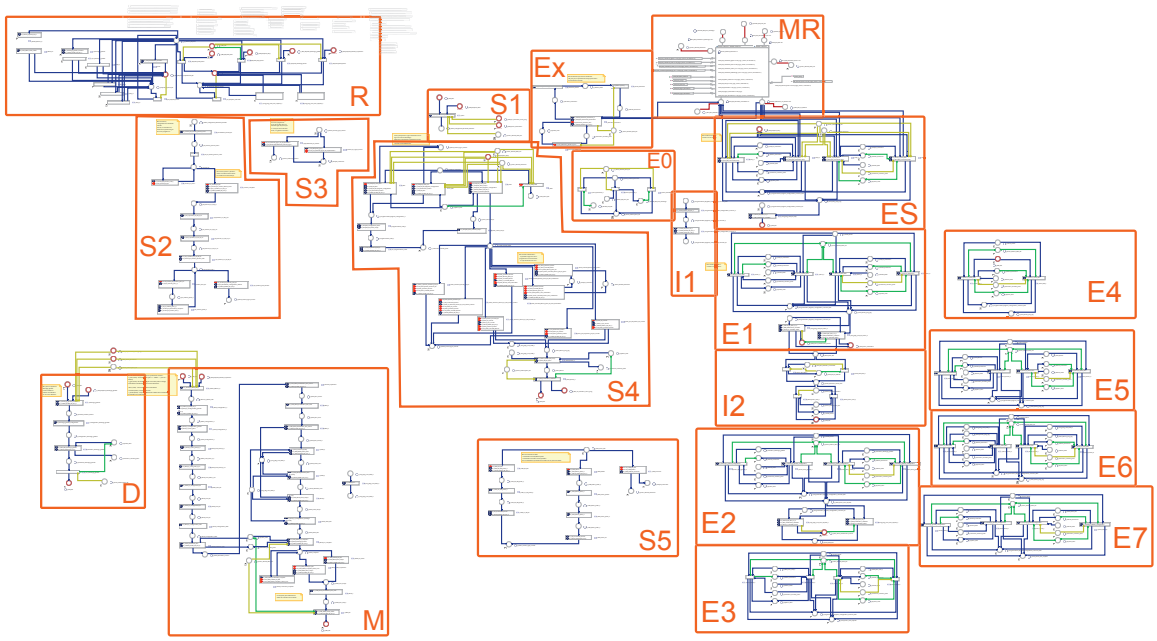

FIGURE D.1 – Modèle complet de la micro-machine de troisième génération

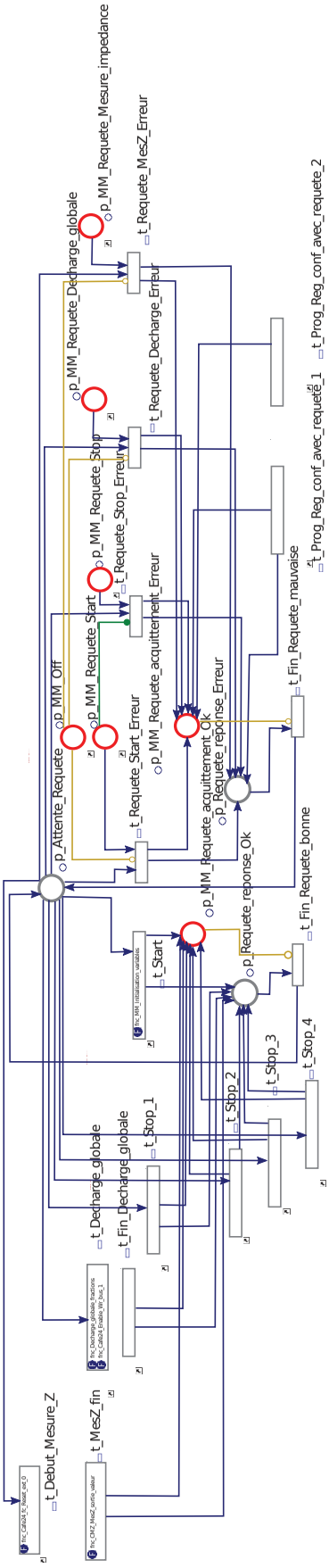

FIGURE D.2 – Partie R de la micro-machine de troisième génération

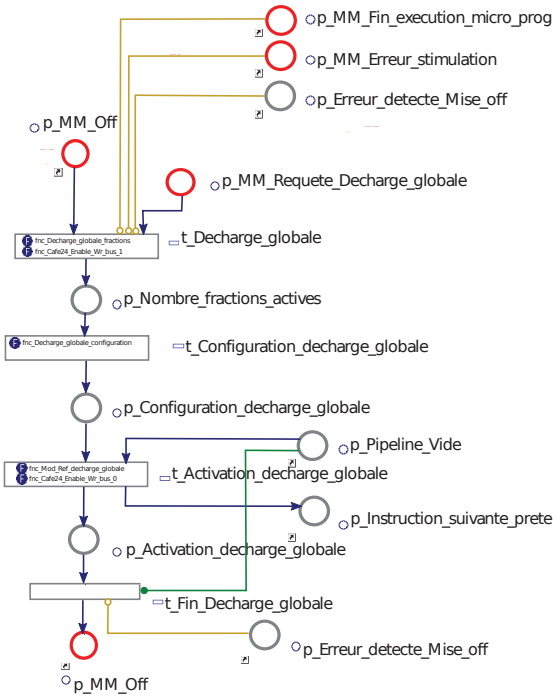

FIGURE D.3 – Partie D de la micro-machine de troisième génération

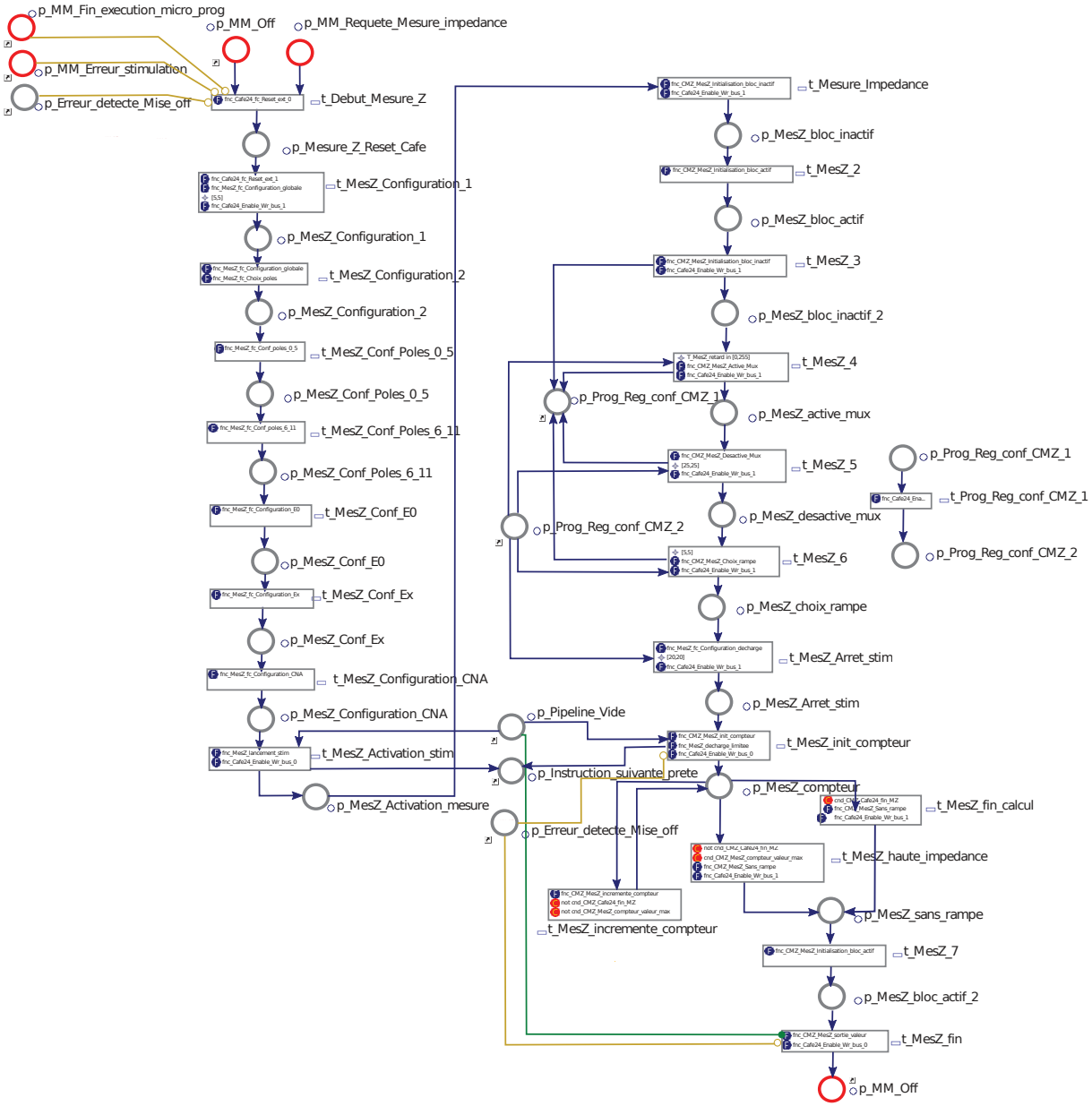

FIGURE D.4 – Partie M de la micro-machine de troisième génération

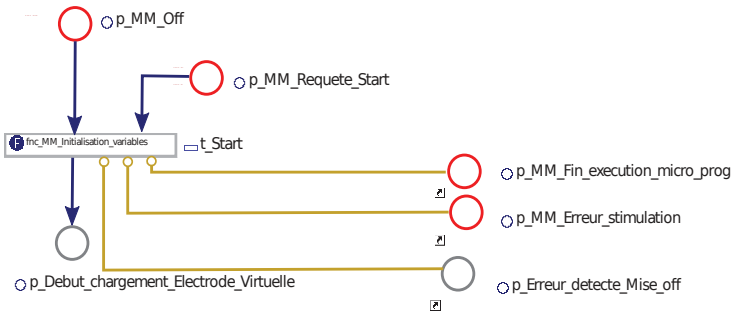

FIGURE D.5 – Partie S1 de la micro-machine de troisième génération

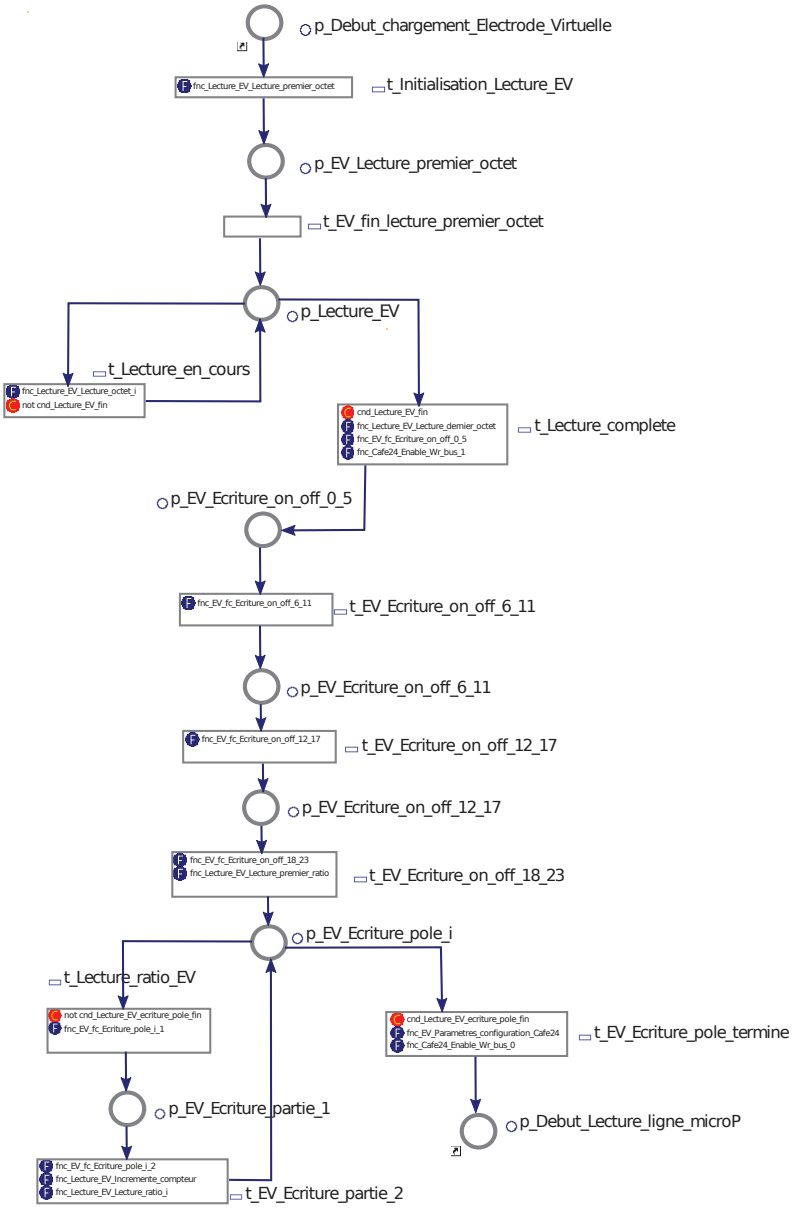

FIGURE D.6 – Partie S2 de la micro-machine de troisième génération

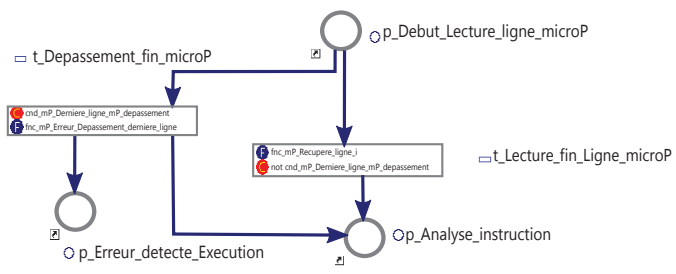

FIGURE D.7 – Partie S3 de la micro-machine de troisième génération

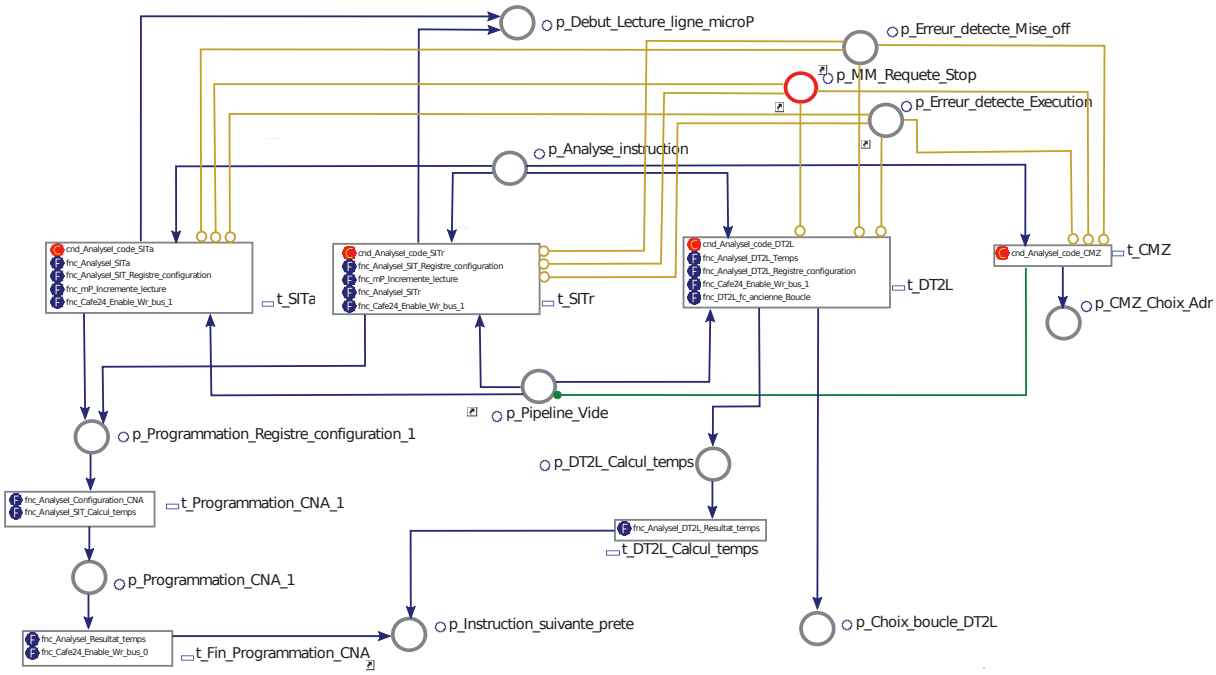

FIGURE D.8 – Partie S4a de la micro-machine de troisième génération

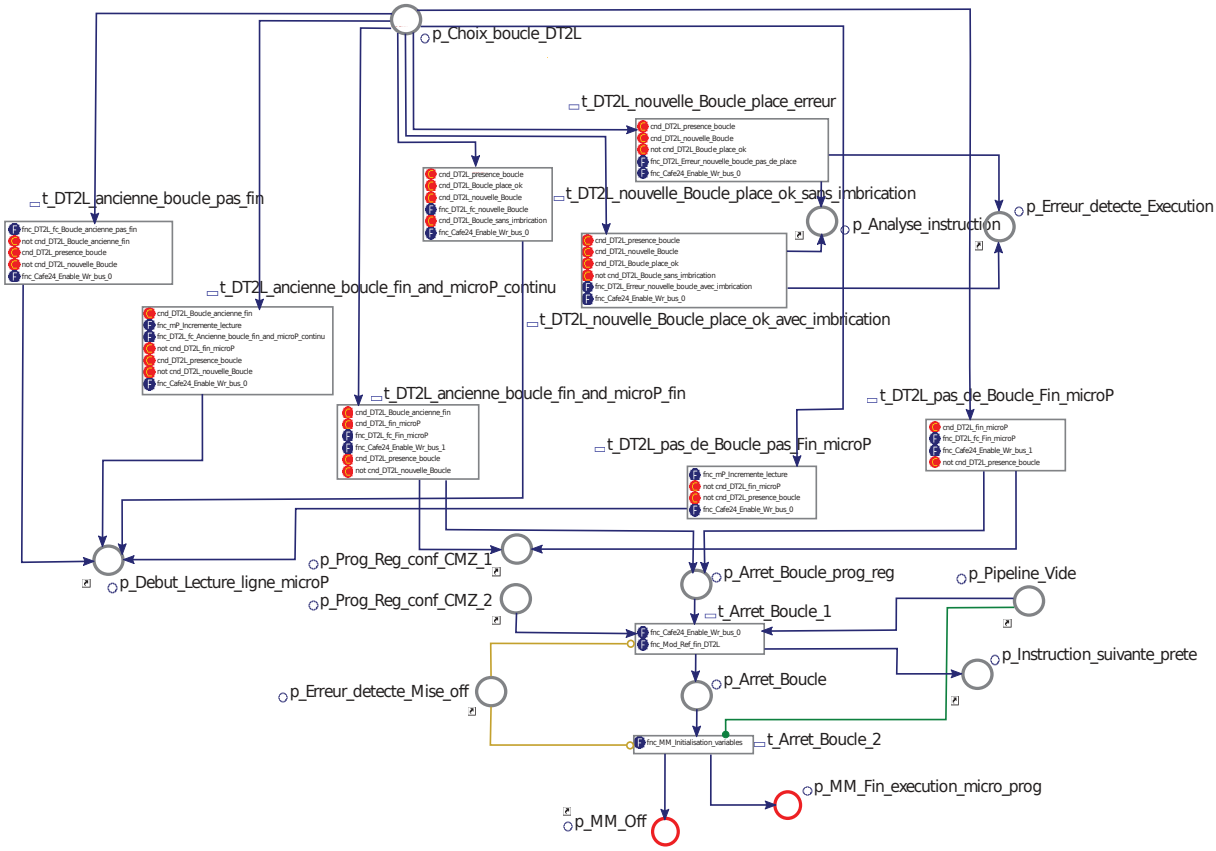

FIGURE D.9 – Partie S4b de la micro-machine de troisième génération

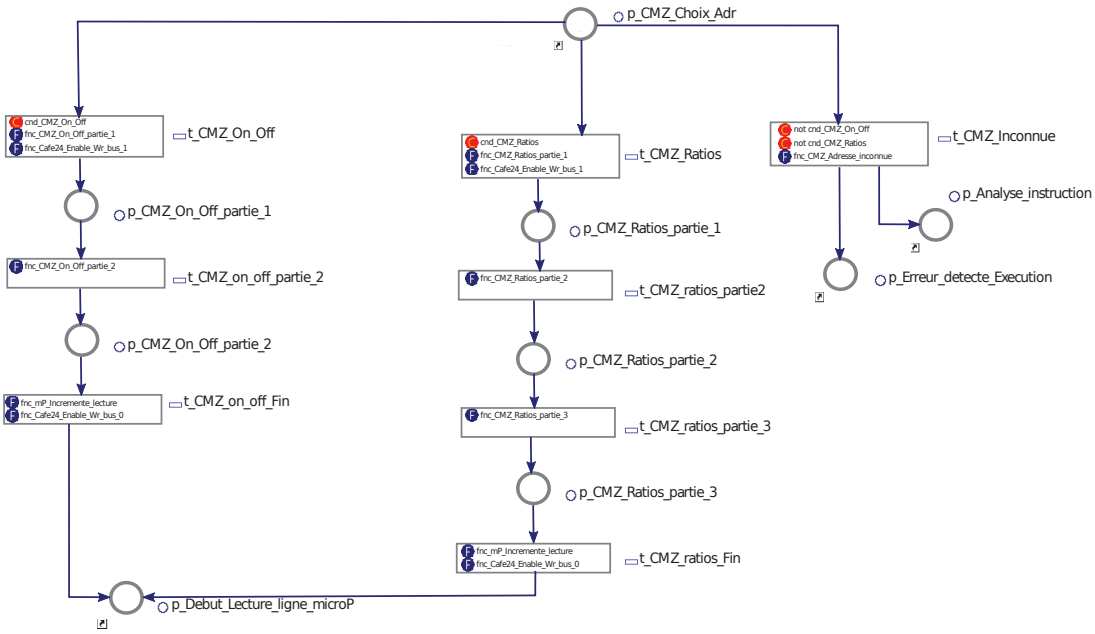

FIGURE D.10 – Partie S5 de la micro-machine de troisième génération

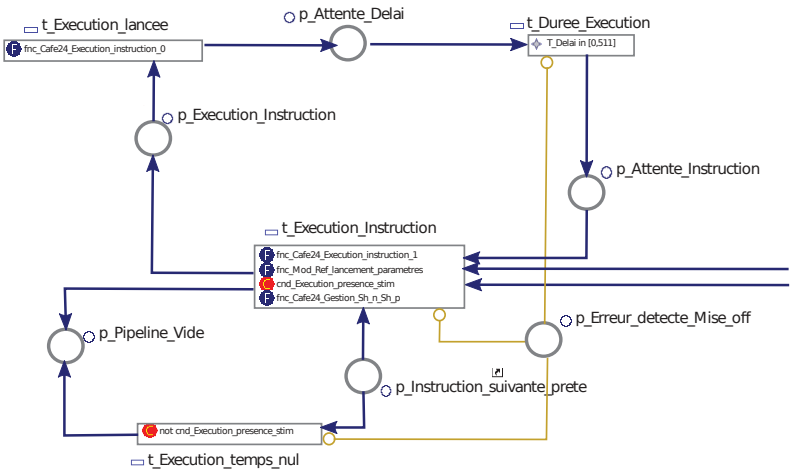

FIGURE D.11 – Partie Ex de la micro-machine de troisième génération

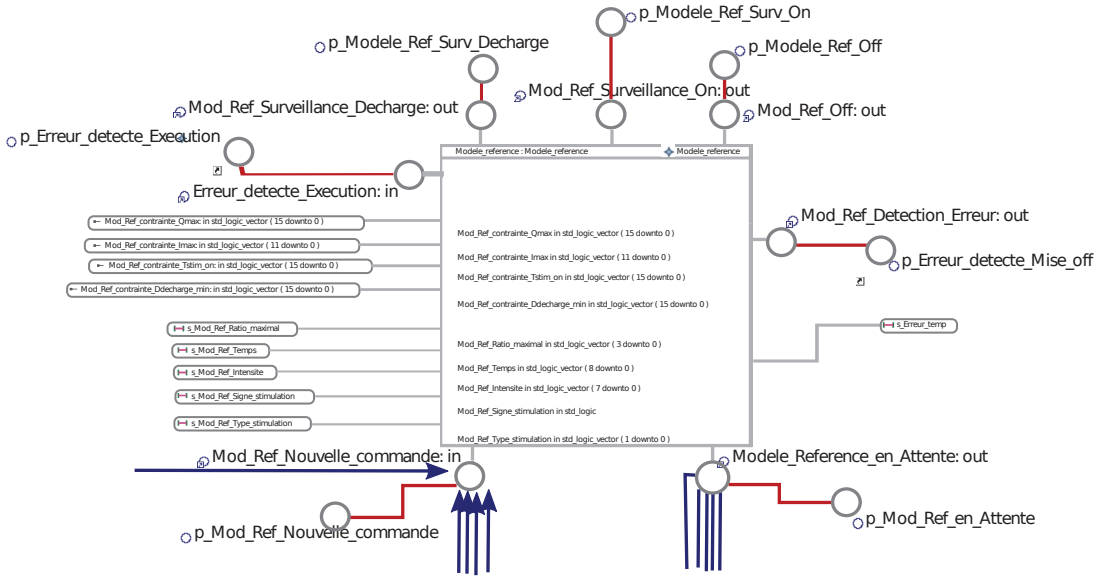

FIGURE D.12 – Partie MR de la micro-machine de troisième génération

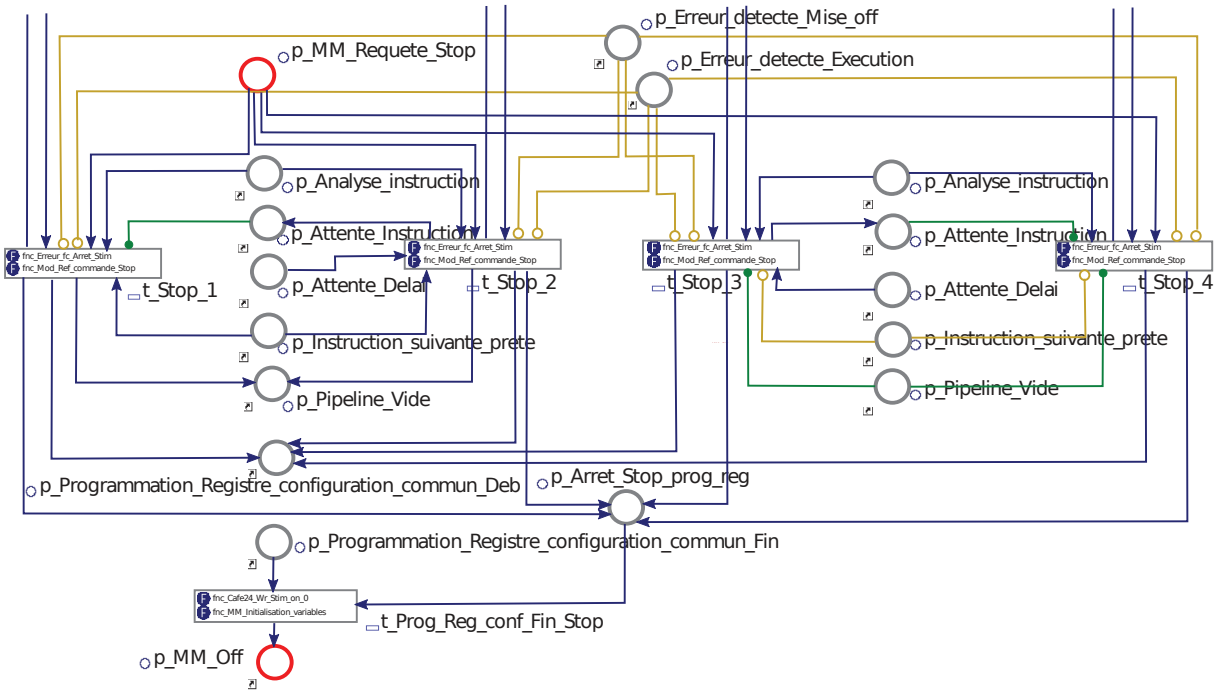

FIGURE D.13 – Partie ES de la micro-machine de troisième génération

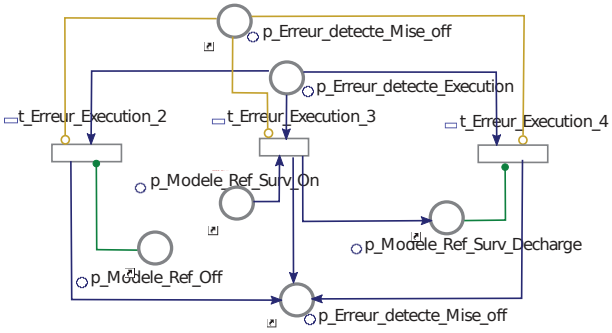

FIGURE D.14 – Partie E0 de la micro-machine de troisième génération

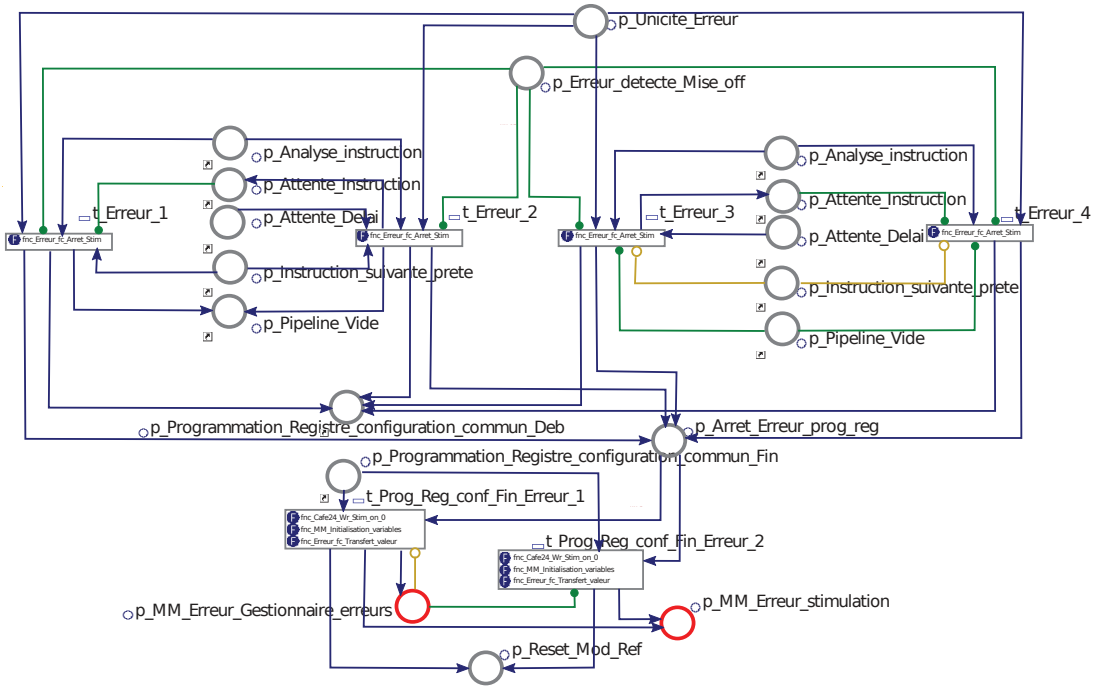

FIGURE D.15 – Partie E1 de la micro-machine de troisième génération

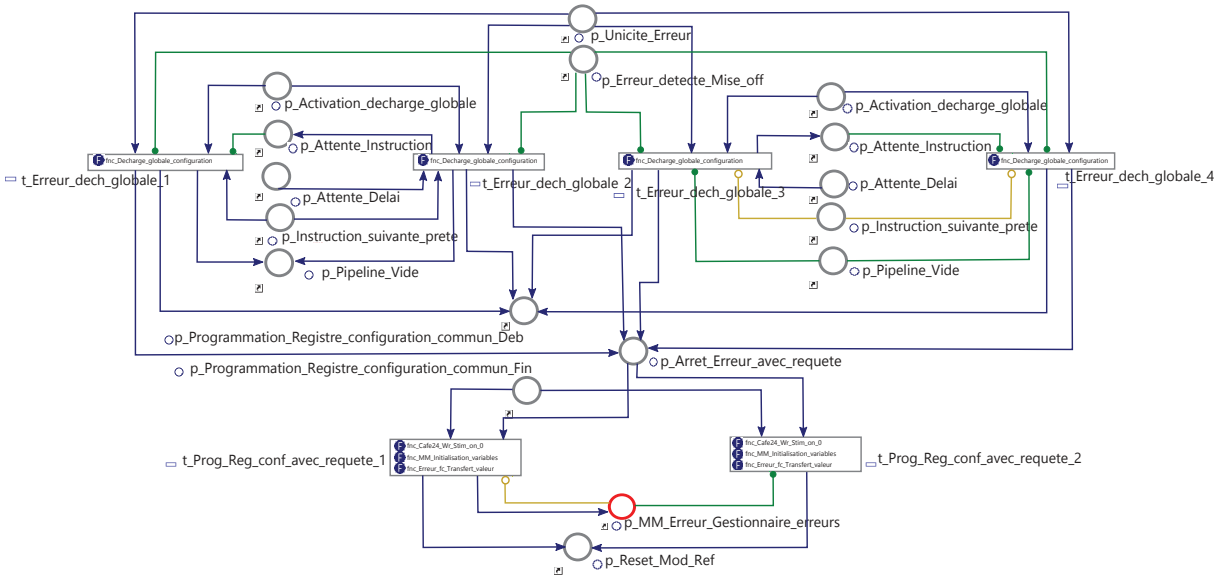

FIGURE D.16 – Partie E2 de la micro-machine de troisième génération

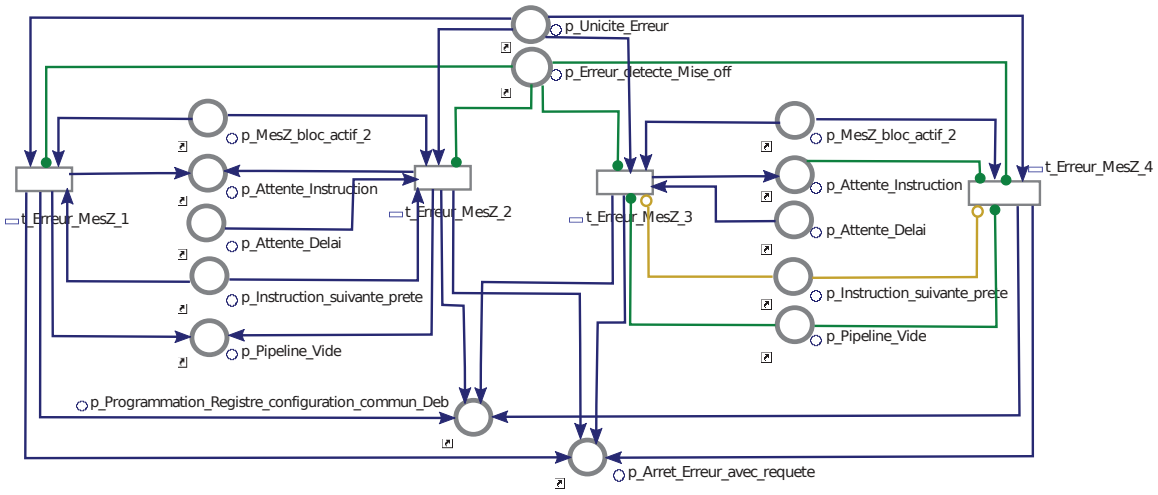

FIGURE D.17 – Partie E3 de la micro-machine de troisième génération

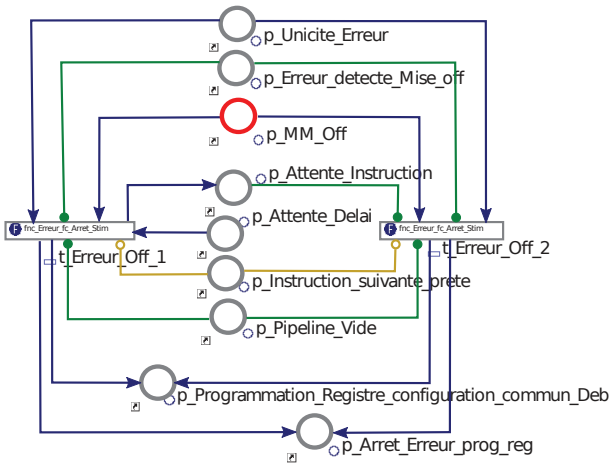

FIGURE D.18 – Partie E4 de la micro-machine de troisième génération

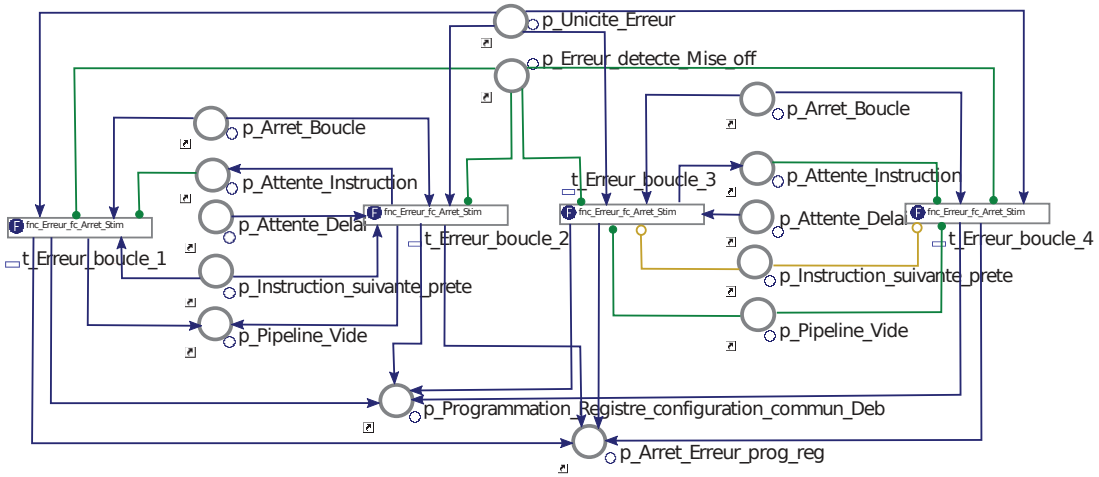

FIGURE D.19 – Partie E5 de la micro-machine de troisième génération

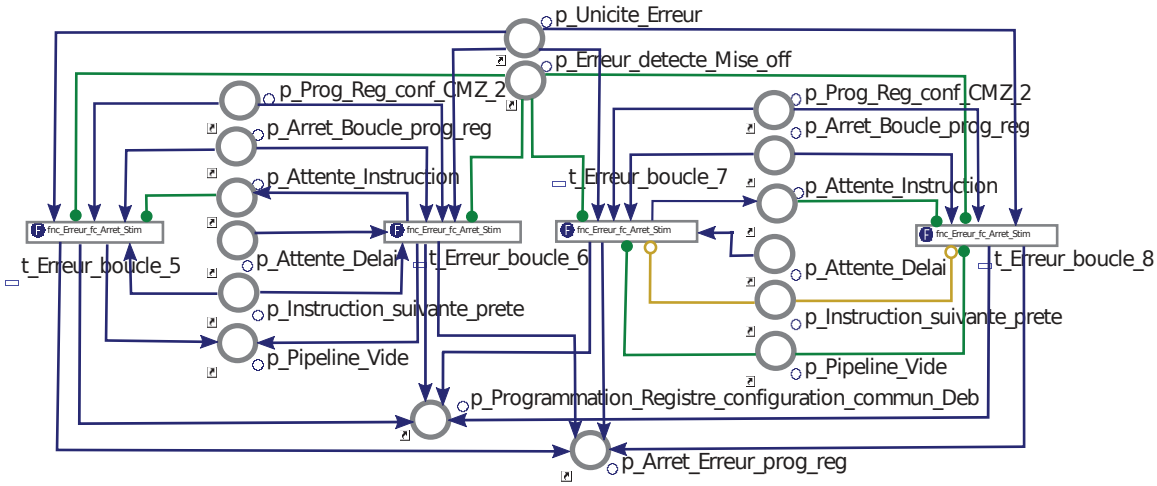

FIGURE D.20 – Partie E6 de la micro-machine de troisième génération

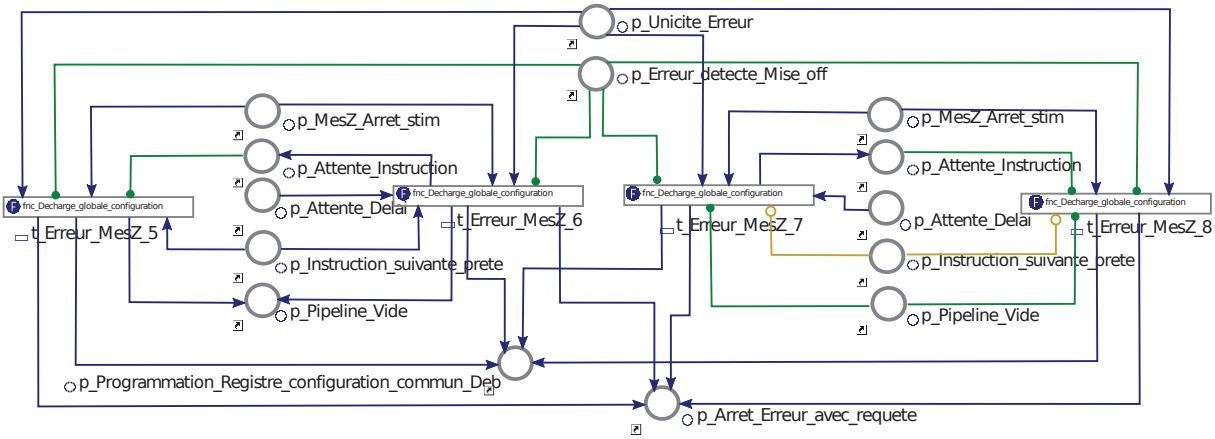

FIGURE D.21 – Partie E7 de la micro-machine de troisième génération

FIGURE D.22 – Partie I1 de la micro-machine de troisième génération

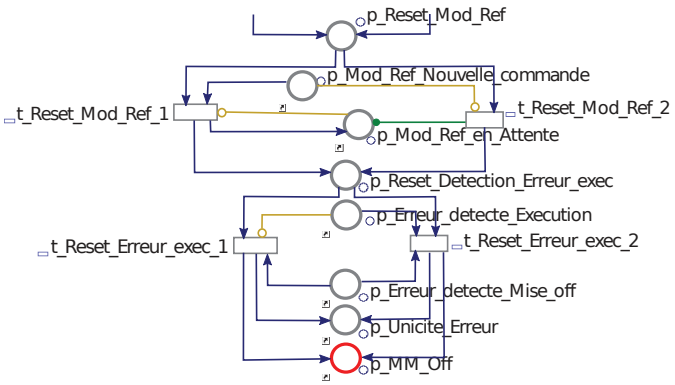

FIGURE D.23 – Partie I2 de la micro-machine de troisième génération

Annexe E

Articles publiés dans le cadre du doctorat

Hélène Leroux, Karen Godary-Dejean and David Andreu. Gestion des exceptions sur un réseau de Petri généralisé interprété T-temporel. In *5èmes Journées Doctorales / Journées Nationales MACS*. Strasbourg, France, July 2013.

Hélène Leroux, Karen Godary-Dejean and David Andreu. Complex digital system design : A methodology and its application to medical implants. In *18th International Workshop on Formal Methods for Industrial Critical Systems*. Madrid, Spain, 2013.

Hélène Leroux, Karen Godary-Dejean, Guillaume Coppey and David Andreu. Integrating implementation properties in analysis of petri nets handling exceptions. In *IEEE Computer Society Annual Symposium on VLSI*. Cachan, France, May 2014.

Hélène Leroux, Karen Godary-Dejean and David Andreu. Automatic handling of conflicts in synchronous Interpreted Time Petri nets implementation. In *12th IFAC - IEEE International Workshop on Discrete Event Systems (WODES)*. Tampa, Florida, USA, July 2014.

Hélène Leroux, David Andreu and Karen Godary-Dejean. Handling exceptions in Petri nets based digital architecture : from formalism to implementation on FPGAs. *IEEE Transactions on Industrial Informatics*. Revue soumise, 2e soumission après correction, en attente de l'avis final.

Bibliographie

[1] A Abdelli and M Daoudi. Un nouveau mécanisme pour la résolution du non déterminisme dans les réseaux de petri temporels. In *proc of IEEE Inter symposium SETIT*, 2005.

[2] C. André. Representation and analysis of reactive behaviors : A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC.

[3] C. André. Syncharts : a visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, April 1996.

[4] C. André. Semantics of synccharts. Technical Report RR-2003-24-FR, I3S Laboratory, University of Nice-Sophia Antipolis, 2003.

[5] D. Andreu, D. Guiraud, and G. Souquet. A distributed architecture for activating the peripheral nervous system. *Journal of Neural Engineering*, 6(2) :026001, 2009.

[6] David Andreu. *Architectures de contrôle en Robotique et en Stimulation Electro-Fonctionnelle : une contribution à la croisée des disciplines*. Habilitation à diriger des recherches, Université Montpellier 2, Décembre 2009.

[7] David Andreu, N. Bruchon, and Thierry Gil. Du modèle à l'exécution : traduction automatique d'un réseau de petri interprété en langage vhdl. Technical report, LIRMM, 2004.

[8] K.E. Årzén. Sequential function charts for knowledge-based, real-time applications. *Annual Review in Automatic Programming*, 16, Part 1(0) :91 – 96, 1991.

[9] Peter J Ashenden. *The designer's guide to VHDL*, volume 3. Morgan Kaufmann, 2010.

[10] Christine AZEVEDO-COSTE, David GUIRAUD, David ANDREU, and Serge BERNARD. Stimulation électrique pour la rééducation et la suppléance fonctionnelles. applications. *Techniques de l'ingénieur Imagerie médicale, électronique et TIC pour la santé*, base documentaire : TIB607DUO.(ref. article : re127), 2014.

[11] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[12] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool tina – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14) :2741–2756, 2004.

[13] Bernard Berthomieu, Florent Peres, and François Vernadat. Bridging the gap between timed automata and bounded time petri nets. In *Formal Modeling and Analysis of Timed Systems*, pages 82–97. Springer, 2006.

[14] Bernard Berthomieu, Florent Peres, and François Vernadat. Model checking bounded prioritized time petri nets. In *Automated Technology for Verification and Analysis*, pages 523–532. Springer, 2007.

[15] Bernard Berthomieu and François Vernadat. Réseaux de petri temporels : méthodes d'analyse et vérification avec tina. *Traité IC2*, page 101, 2006.

[16] K Bilinski, M Adamski, JM Saul, and EL Dagless. Petri-net-based algorithms for parallel-controller synthesis. In *Computers and Digital Techniques, IEE Proceedings*, volume 141, pages 405–412. IET, 1994.

[17] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79 :1293–1304, 1991.

[18] Marc Boyer and Olivier H Roux. Comparison of the expressiveness of arc, place and transition time petri nets. In *Petri Nets and Other Models of Concurrency–ICATPN 2007*, pages 63–82. Springer, 2007.

[19] A. Bukowiec and M. Adamski. Synthesis of petri nets into fpga with operation flexible memories. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2012 IEEE 15th International Symposium on*, pages 16–21, April 2012.

[20] Giovanni Chiola, Marco Ajmone Marsan, Gianfranco Balbo, and Gianni Conte. Generalized stochastic Petri nets : A definition at the net level and its implications. *IEEE Transactions on Software Engineering*, 19(2) :89–107, 1993.

[21] Michel COMBACAU, Philippe ESTEBAN, and Alexandre NKETSA. *Commandes à réseaux de Petri : Mise en oeuvre et application*, volume 2. Techniques de l'ingénieur, 2005.

[22] Robin Cressent. *Valorisation de l'Ingénierie Système à Base de Modèles, pour l'Analyse de Sûreté de Fonctionnement des Systèmes Complexes Critiques intégrant des COTS*. PhD thesis, Université d'Orléans, 2012.

[23] R. David. Grafcet : a powerful tool for specification of logic controllers. *IEEE Transactions on Control Systems Technology*, 3(3) :253–268, Sep 1995.

[24] R. David and H Alla. Discrete, continuous and hybrid petri nets. *Control Systems, IEEE*, 28(3) :81 –84, june 2005.

[25] René David and Hassane Alla. Petri nets for modeling of dynamic systems : A survey. *Automatica*, 30(2) :175–202, 1994.

[26] Wagner Luiz Alves de Oliveira, Norian Marranghello, and Furio Damiani. Modeling a processor with a petri net extension for digital systems. In *Proceedings of the Conference on Design, Analysis, and Simulation of Distributed Systems*, 2004.

[27] W.L.A. de Oliveira, N. Marranghello, and F. Damiani. Exception handling with petri net for digital systems. In *Proceedings of the 15th symposium on Integrated circuits and systems design*, pages 229–235. IEEE Computer Society, 2002.

[28] M. Doligalski and M. Adamski. Exceptions handling in hierarchical petri net based specification for logic controllers. In *Systems Engineering (ICSEng)*, pages 459–460. IEEE, 2011.

[29] MichałDoligalski. Behavioral specification diversification for logic controllers implemented in fpga devices. In *Proceedings of the Annual FPGA Conference*, volume 6 of *FPGAworld '12*, pages 1–5, New York, NY, USA, 2012. ACM.

[30] Michał Doligalski and Marian Adamski. Hierarchical configurable petri net modeling in vhdl. *International Journal of Electronics and Telecommunications*, 58(4) :397–402, 2012.

[31] Joao M Fernandes, Marian Adamski, and Alberto J Proenca. VHDL generation from hierarchical Petri net specifications of parallel controllers. In *IEE Proceedings of Computers and Digital Techniques*, volume 144, pages 127–137. IET, 1997.

[32] Jean H. Gallier. *Logic for Computer Science : Foundations of Automatic Theorem Proving*. Harper & Row Publishers, Inc., New York, NY, USA, 1985.

[33] Juan-Carlos HAMONT. *Méthodes et outils de la conception amont pour les systèmes et les microsystèmes*. PhD thesis, Institut National Polytechnique de Toulouse, 2005.

[34] D. Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231 – 274, 1987.

[35] T. Holvoet and P. Verbaeten. Petri charts : an alternative technique for hierarchical net construction. In *Proceedings of the 1995 IEEE Conference on Systems, Man and Cybernetics (IEEE-SMC95*, pages 22–25. IEEE Press, 1995.

[36] IEC 60848 ed.2, specification language grafcet for sequential function charts. *International Electrotechnical Commission*, 2001.

[37] C. Johnsson. Recipe-based batch control using high-level grafchart. Licentiate Thesis 3217, June 1997.

[38] C. Johnsson and K.E. Arzen. Grafchart and grafcet : A comparison between two graphical languages aimes for sequential control applications. In *Preprints 14th Word Congress of IFAC*, pages 19–24, 1999.

[39] M. Kishinevsky, J. Cortadella, A. Kondratyev, L. Lavagno, A. Taubin, and A. Yakovlev. Place chart nets. Technical report, University of Aizu, Japan, 1996.

[40] M. Kishinevsky, J. Cortadella, A. Kondratyev, L. Lavagno, A. Taubin, and A. Yakovlev. Coupling asynchrony and interrupts : Place chart nets. In *ICATPN*, volume 1248 of *Lecture Notes in Computer Science*, pages 328–347. Springer, 1997.

[41] Grzegorz Łabiak. Symbolic state exploration of uml statecharts for hardware description. In *Design of Embedded Control Systems*, pages 73–83. Springer US, 2005.

[42] Norian Marranghello. Digital systems synthesis from petri net descriptions. *DAIMI Report Series*, 27(530), 1998.

[43] M Moalla, Jacques Pulou, and Joseph Sifakis. Synchronized petri nets : A model for the description of non-autonomous sytems. In *Mathematical Foundations of Computer Science 1978*, pages 374–384. Springer, 1978.

[44] Tadao Murata. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989.

[45] Jean-Marc Roussel and Jean-Jacques Lesage. Design of Logic Controllers Thanks to Symbolic Computation of Simultaneously Asserted Boolean Equations. *Mathematical Problems in Engineering*, 2014 :Article ID 726246, May 2014.

[46] Olivier Sentieys, Arnaud Tisserand, et al. Architectures reconfigurables fpga. *Technologies logicielles Architectures des systèmes*, pages 1–22, 2012.

[47] C.F. Silva, C. Quintans, A. Colmenar, M.A. Castro, and E. Mandado. A method based on Petri nets and a matrix model to implement reconfigurable logic controllers. *IEEE Transactions on Industrial Electronics*, 57(10) :3544–3556, 2010.

[48] Robert H Sloan and Ugo Buy. Reduction rules for time petri nets. *Acta Informatica*, 33(7) :687–706, 1996.

[49] Guillaume Souquet. *Architecture de stimulation électro-fonctionnelle implantable : des concepts aux applications*. PhD thesis, Université Montpellier 2, 2009.

[50] Donald E Thomas and Philip R Moorby. *The Verilog hardware description language*, volume 2. Springer, 2002.

[51] Jacek Tkacz and Marian Adamski. Logic design of structured configurable controllers. In *Networked Embedded Systems for Every Application (NESEA), 2012 IEEE 3rd International Conference on*, pages 1–6. IEEE, 2012.

[52] Jacek Tkacz and Marian Adamski. Design of structured configurable controllers using gentzen reasoning. In *International Journal of Design, Analysis and Tools for Integrated Circuits and Systems (IJDATICS)*, volume 4-2, page 10, 2013.

[53] Louis-Marie Traonouez, Didier Lime, Olivier H. Roux, et al. Parametric model-checking of stopwatch petri nets. *J. UCS*, 15(17) :3273–3304, 2009.

[54] Murat Uzam, I. Burak Koc, Gokhan Gelen, and B. Hakan Aksebzeci. Asynchronous implementation of discrete event controllers based on safe automation petri nets. *The International Journal of Advanced Manufacturing Technology*, 41(5-6) :595–612, 2009.

[55] VI Varshavsky and VB Marakhovsky. Asynchronous control device design by net model behavior simulation. In *Application and Theory of Petri Nets 1996*, pages 497–515. Springer, 1996.

[56] Michael Weber and Ekkart Kindler. The petri net markup language. In *Petri Net Technology for Communication-Based Systems*, pages 124–144. Springer, 2003.

[57] Agnieszka Węgrzyn and Marek Węgrzyn. A new approach to simulation of concurrent controllers. In *Design of embedded control systems*, pages 95–108. Springer, 2005.

[58] Alexandre V. Yakovlev and Albert M. Koelmans. Petri nets and digital hardware design. In *Lectures on Petri Nets II : Applications*, volume 1492 of *Lecture Notes in Computer Science*, pages 154–236. Springer Berlin Heidelberg, 1998.

[59] Alexandre V Yakovlev and Albert M Koelmans. Petri nets and digital hardware design. In *Lectures on Petri Nets II : Applications*, pages 154–236. Springer, 1998.

[60] Yana Yankova, Koen Bertels, Stamatis Vassiliadis, Roel Meeuws, and Arcilio Virginia. Automated hdl generation : Comparative evaluation. In *Circuits and Systems. IEEE International Symposium on ISCAS 2007*, pages 2750–2753. IEEE, 2007.

[61] http://www.topcased.org/.

[62] http://projects.laas.fr/tina.

[63] http://romeo.rts-software.org/.

[64] http://www.synopsys.com/Tools/Implementation/FPGAImplementation/FPGASynthesis/Pages/SynplifyPro.aspx.

[65] http://www.microsemi.com/products/fpga-soc/design-resources/design-software/libero-soc.

[66] http://simplesolverlogic.com/.

[67] http://boolengine.com/.