Validation formelle des systèmes numériques critiques : génération de l'espace d'états de réseaux de Petri exécutés en synchrone

Ibrahim Merzoug

THESE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITE DE MONTPELLIER

En informatique industrielle

École doctorale : Information, Structure, Systèmes (I2S)

Unité de recherche : Laboratoire d'Informatique, Robotique et Micro-électronique de Montpellier (LIRMM)

Titre de la thèse

VALIDATION FORMELLE DES SYSTEMES NUMERIQUES CRITIQUES : GENERATION D'ESPACE D'ETATS DE RESEAUX DE PETRI EXECUTES EN SYNCHRONE

Présentée par Ibrahim Merzoug

Le 15 Janvier 2018

Sous la direction de David Andreu

Devant le jury composé de

Frédéric Boniol, Professeur, Institut National Polytechnique de Toulouse	Rapporteur
Thomas Chatain, Maitre de conférences HDR, ENS Paris-Saclay	Rapporteur
David Guiraud, directeur de recherches, INRIA, Sophia-Antipolis Méditerranée	Examinateur
François Vernadat, Professeur, Institut national des sciences appliquées de Toulouse	Président du jury
David Andreu, Maitre de conférences HDR, Université Montpellier	Directeur de thèse
Karen Godary-dejean, Maitre de conférences, Université Montpellier	Co-Encadrante de thèse

Résumé

La méthodologie HILECOP a été élaborée pour la conception formelle de systèmes numériques complexes critiques ; elle couvre donc l'intégralité du processus, allant de la modélisation à la génération de code pour l'implantation sur la cible matérielle (composant électronique de type FPGA), en passant par la validation formelle. Or, si le modèle formel, les réseaux de Petri en l'occurrence, est par essence asynchrone, il est néanmoins exécuté de manière synchrone sur la cible. De fait, les approches d'analyse usuelles ne sont pas adaptées au sens où elles construisent des graphes d'états non conformes à l'évolution d'états réelle au sein de la cible. Dans l'objectif de gagner en confiance quant à la validité des résultats de l'analyse formelle, ces travaux visent à capturer les caractéristiques dites non-fonctionnelles, à les réifier sur le modèle et enfin à considérer leur impact à travers l'analyse.

En d'autres termes, l'objectif est d'améliorer l'expressivité du modèle et la pertinence de l'analyse, en considérant des aspects comme la synchronisation d'horloge, le parallélisme effectif, le risque de blocage induit par l'expression conjointe d'un événement (condition) et d'une fenêtre temporelle d'occurrence, sans omettre la gestion des exceptions.

Pour traiter tous ces aspects, nous avons proposé une nouvelle méthode d'analyse pour les réseaux de Petri temporels généralisés étendus interprétés exécutés en synchrone, en les transformant vers un formalisme équivalent analysable. Ce formalisme est associé avec une sémantique formelle intégrant tous les aspects particuliers de l'exécution et un algorithme de construction d'un graphe d'états spécifique : le Graphe de Comportement Synchrone.

Nos travaux ont été appliqués à un cas industriel, plus précisément à la validation du comportement de la partie numérique d'un neuro-stimulateur.

Titre en Anglais : Formal verification of digital critical systems : state space generation for synchronously executed Petri nets.

Abstract

The HILECOP methodology has been developed for the formal design of critical complex digital systems ; it therefore covers the entire design process, ranging from modeling to code generation for implementation on the hardware target (FPGA type electronic component), via formal validation. However, if the formal model, the Petri nets in this case, is inherently asynchronous, it is nevertheless executed synchronously on the target. In fact, the usual analysis approaches are not adapted in the sense that they construct state graphs that do not conform to the real state evolution within the target. In order to gain confidence in the validity of the results of the formal analysis, this work aims to capture the so-called non-functional characteristics, to reify them on the model and finally to consider their impact through the analysis.

In other words, the aim is to improve the expressiveness of the model and the relevance of the analysis, considering aspects such as clock synchronization, effective parallelism, the risk of blocking induced by the expression of an event (condition) and a time window of occurrence, without omitting the management of exceptions.

To deal with all these aspects, we have proposed a new method of analysis for extended generalized synchronously executed time Petri nets, transforming them into an analysable equivalent formalism. This formalism is associated with a formal semantics integrating all the particular aspects of the execution and a dedicated state space construction algorithm : the Synchronous Behavior Graph.

Our work has been applied to an industrial case, more precisely to the validation of the behavior of the digital part of a neuro-stimulator.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Les systèmes embarqués sont présents dans de multiples domaines tels que la téléphonie mobile, les transports, l'énergie, l'avionique, le spatial... et, pour ce qui nous concerne, les dispositifs médicaux. Si l'on s'en tient à la définition générale, un système embarqué est un système électronique piloté par un logiciel informatique autonome. Ceci étant, selon le domaine d'application considéré, la complexité du système embarqué est différente et surtout sa qualification est différente. En effet, dans des domaines comme l'avionique et les dispositifs médicaux, pour ne citer que ceux-là, le système embarqué est un système qualifié de critique puisqu'il peut, au delà de causer des pertes matérielles et/ou financières importantes, fondamentalement mettre en danger des vies humaines. Il existe malheureusement de multiples exemples de comportements erronés de ces systèmes, ayant entraîné des conséquences sévères (automobile, avionique, systèmes médicaux, etc.). En évoquant le comportement du système nous précisons le cadre de nos préoccupations scientifiques. Le bon fonctionnement d'un système critique est évidemment primordial, si ce n'est que cette généralité ouvre bien des considérations telles que l'approche de décomposition face à la complexité sans cesse croissante de ces systèmes, l'approche de représentation de leurs comportements, l'approche de mise en oeuvre... sans omettre l'expression même du "bon" fonctionnement, au sens de l'expression du comportement désiré et des contraintes à respecter, que l'on peut plus généralement appeler exigences. Ces exigences peuvent être de nature fonctionnelle et réglementaire ; le cahier des charges exige que le système exhibe un comportement précis dans des situations précises, qu'il soit notamment déterministe et fiable, et le domaine apporte son lot d'exigences sur des contraintes normatives à respecter. De la représentation du comportement à la vérification du respect des exigences, de nombreux aspects sont à prendre en considération. Tout d'abord s'assurer que le formalisme de représentation permet de vérifier des exigences, communément appelées propriétés. Il est bien connu que le test est une méthode incontournable mais aussi non exhaustive, au sens de s'assurer que toutes les situations sont couvertes par ces tests ; dès lors le recours à une approche formelle s'impose. La recherche et l'industrie ont proposé différentes méthodologies, plus ou moins outillées, offrant la possibilité

d'une conception formelle de tels systèmes avec différentes approches de *Model Checking* permettant d'effectuer des vérifications.

Représenter le comportement d'un système de manière formelle permet déjà une "traduction" du cahier des charges, potentiellement ambigu car décrit dans un langage naturel, en spécifications plus rigoureuses. Cela permet bien sur d'aller plus loin, en offrant la possibilité de vérifier formellement des propriétés (exigences) et c'est le but. Néanmoins, vérifier des propriétés sur un modèle du système, au stade de la conception, n'est pas vérifier les propriétés du système réel, à savoir une fois mis en oeuvre sur sa cible. Or, l'attente est clairement de garantir que le "produit" est conforme et pas seulement que sa conception l'est, et la nuance est conséquente. D'ailleurs, le contenu de certaines normes relatives au cycle de développement d'applications critiques laisse apparaître cette difficulté : les normes vont imposer un maximum de rigueur dans le processus de conception et de développement, des recommandations principalement de nature méthodologique, et vont exiger de réaliser des tests unitaires et d'intégration les plus couvrants (à défaut de pouvoir être exhaustifs)... c'est indirectement souligner que la technique de mise en oeuvre, sans omettre les possibles erreurs induites par l'intervention humaine (e.g., lors de la programmation), a une influence significative sur le comportement résultant.

Cependant, la prise en compte de la technique de mise en oeuvre, et surtout de son impact, est souvent omise, voire négligée, dans les approches de conception formelle. Prenons comme exemple évident, le cas d'une application multitâche temps-réel exécutée sur un processeur (mono-coeur), selon une approche asynchrone : modéliser l'ensemble des tâches et leurs interactions est nécessaire, mais pas suffisant pour garantir le respect à l'exécution de priorités logiques et temporelles puisque l'effet de la préemption au sein du système d'exploitation temps-réel ne saurait être négligé. Ce propos ne se veut pas général puisqu'il existe, dans le domaine de l'avionique notamment (un domaine précurseur sur ce sujet), des environnements (méthodologies outillées) visant explicitement à produire du code certifié au sens d'un code dont le comportement correct a été prouvé (e.g., SCADE) [36].

Le cadre de nos préoccupations scientifiques s'affine ; il s'agit donc d'étudier la prise en compte de caractéristiques d'implémentation, que nous qualifierons par la suite de propriétés non-fonctionnelles, au sein d'un processus existant

de conception et réalisation de systèmes numériques complexes critiques. Le contexte, que nous détaillerons dans le manuscrit, est plus particulièrement celui de la mise en oeuvre sur circuits électroniques de type composants logiques programmables (e.g., FPGA) ou circuit spécifiques (e.g., ASIC). Le terme d'implémentation peut donc être entendu ici comme l'implantation sur circuit électronique. Dans ce contexte, l'équipe CAMIN a créé une méthodologie outillée, appelée HILECOP (High Level hardware COmponent Programming). Initiée à travers la thèse de G. Souquet [**?**], elle permet la conception d'un système embarqué critique, son analyse formelle et son implémentation sur FPGA. Elle a été en particuliers exploitée pour la conception de la partie numérique de dispositifs médicaux implantables actifs.

Considérant cette méthodologie de conception formelle HILECOP, l'objectif est d'améliorer cette approche formelle, du formalisme à l'analyse formelle elle-même, afin de capturer des caractéristiques non-fonctionnelles et d'exprimer leur impact à travers l'analyse. En d'autres termes, l'objectif est d'améliorer l'expressivité du modèle en termes de conformité au comportement se produisant sur la cible, et par voie de conséquence de gagner en confiance quant à la validité des résultats de l'analyse formelle ; tous les états du graphe d'états sont effectifs, tous les chemins sont effectifs, aucun état n'est "électroniquement" inatteignable (i.e., sur la cible) ni n'est omis, etc. Les travaux décrits dans ce manuscrit ont été réalisés au sein de l'équipe projet DEMAR (nouvellement CAMIN), une équipe de recherche hébergée par le LIRMM qui est commune à INRIA (INRIA Sophia Méditerranée), à l'Université de Montpellier et au CNRS.

Ce manuscrit est structuré en 6 chapitres.

Le chapitre 1 introduit le contexte dans lequel s'inscrivent les travaux. Ainsi, un bref survol de la méthodologie HILECOP est présenté. Ensuite, nous rappelons les définitions formelles des formalismes utilisés pour la modélisation et la mise en oeuvre sur la cible matérielle. Les contraintes relevant de ces formalismes ainsi que celles issues de l'implémentation, sont explicitées selon un regard orienté analyse. Le cadre de nos contributions est alors défini, à savoir la nécessité de proposer une nouvelle approche pour l'analyse des systèmes numériques complexes critiques conçus selon la méthodologie HILECOP. Le chapitre 2 aborde ensuite en détails les contraintes "non-fonctionnelles" qui doivent être prises en compte dans l'objectif d'atteindre une analyse non pas

du modèle de conception mais du modèle exécuté. Ceci nous permet de définir un nouveau formalisme, basé sur les réseaux de Petri, issu des formalismes initiaux utilisés dans HILECOP mais intégrant toutes ces contraintes. Le chapitre 3 est alors consacré à la génération du graphe de comportement du nouveau modèle analysable. La sémantique de ce graphe ainsi que son algorithme de construction sont présentés, complété par un deuxième algorithme afin de gérer les situations de conflits. La prise en compte et l'analyse des situations de blocage potentiel sont ensuite traités dans le chapitre 4. Ces situations résultent de la sémantique "impérative" que nous utilisons dans nos formalismes, qui peut conduire à des blocages potentiels au sein du modèle (exécuté) dès lors que conditions et d'intervalles temporels sont conjointement associés à des transitions du modèle. La détection de telles situations est bien sur primordiale ; notre solution consiste à intégrer explicitement le risque de blocage dans la sémantique du modèle analysable et ainsi à l'exprimer sur son graphe de comportement. Toujours dans un objectif d'une analyse "complète" de l'évolution du modèle (exécuté), le chapitre 5 traite de la gestion d'exception. En effet, la méthodologie HILECOP permet d'exprimer la gestion d'exception à l'aide d'un mécanisme basé sur les réseaux de Petri, combinant macroplace et transition d'exception. Diverses solutions possibles pour l'analyse de ce mécanisme sont étudiées et une solution rigoureusement conforme à la réalité, au sens évolution au sein de la cible, est présentée. Enfin, le dernier chapitre du manuscrit expose, en deux parties, d'une part la mise en oeuvre, la validation et l'intégration de nos algorithmes dans l'environnement HILECOP et d'autre part leur application à un exemple industriel. Ce cas industriel est celui du système numérique d'un dispositif médical implantable dédié à la stimulation électrique-fonctionnelle (un stimulateur neural). Le manuscrit est clôturé par une conclusion sur ces travaux ainsi qu'un ensemble des pistes scientifiques de perspectives qui s'en sont dégagées.

Chapitre 1

Contexte et état de l'art

Dans ce chapitre, nous présentons tout d'abord les principes de HILECOP (High Level Hardware Programming), une méthodologie de conception formelle que notre équipe a conçu dans le cadre de la réalisation de dispositifs médicaux implantables actifs (DMIA), dont la partie numérique est un cas de système numérique complexe critique. Cette méthodologie s'appuie entre autres sur les réseaux de Petri afin de représenter formellement le comportement du système. Nous rappellerons succinctement les bases de ce formalisme ainsi que les apports de quelques extensions, dans le cadre de nos préoccupations. Nous nous intéresserons alors plus particulièrement à l'analyse formelle, au coeur de cette thèse, avec un état de l'art sur les méthodes d'analyse et leurs limitations.

Les objectifs de ce chapitre sont donc d'une part de préciser le contexte dans lequel s'inscrivent nos travaux et d'autre part de décrire l'état de l'art sur notre préoccupation précise, celle de l'analyse formelle d'un modèle asynchrone, en l'occurrence les réseaux de Petri, exécuté de manière synchrone.

1.1 La méthodologie de conception HILECOP

A ce jour, les DMIA sont principalement basés sur une approche logicielle au sens d'un programme informatique exécuté par un microprocesseur, qui pilote un étage électronique (souvent analogique) dédié à stimuler ou à mesurer une activité, sur un nerf, un muscle ou un organe. Cette approche basée logiciel implique un processus de cycle de développement très long qui doit se conformer au cycle de vie imposé par la norme IEC 62304 [56], et qui vise à assurer une conception et une mise en œuvre rigoureuses. Cependant, le recours à une approche basée processeur pose de multiples limitations dont celle relative à la capacité de contrôler une stimulation à la microseconde sans induire une consommation excessive, et celle relative à la capacité de prouver formellement un comportement correct et fiable. En effet, la gestion du parallélisme dans les applications multi-tâches temps réel est toujours délicate, alors que les tests sont inévitablement limités, au sens non-exhaustifs.
Ces deux considérations ont amené l'équipe DEMAR à concevoir des DMIA basés sur des circuits électroniques, soit des composants électroniques program-

mables (type FPGA [1]), soit des circuits électroniques dédiés (type ASIC [2]). La mise en oeuvre de systèmes numériques complexes directement sur circuits électroniques a pour avantage de permettre un parallélisme effectif, et ce à faible consommation. C'est donc dans le cadre de la conception formelle de ces systèmes que la méthodologie HILECOP a été développée au sein de l'équipe INRIA DEMAR.

1.1.1 Principes de HILECOP

HILECOP vise à permettre de gérer la complexité et les besoins de fiabilité de tels systèmes. Pour ce faire, la méthodologie sous-jacente se base sur les concepts d'ingénierie dirigée par les modèles [95, 62]. Il est essentiel de fournir aux ingénieurs des "outils" (modèles, mécanismes, méthodologies, etc.) facilitant certes la conception et sa validation, mais également permettant fondamentalement de visualiser la conception et de pouvoir exploiter les représentations graphiques associées comme support de discussions dans le cadre d'une réalisation en équipe. Ces représentations doivent bien sur être non ambiguës, pour éviter toute erreur d'interprétation. Cette dimension, plus relative à l'usage concret des "outils", ne doit pas être négligée ; elle a d'ailleurs été prise en considération dans les choix des formalismes sur lesquels s'appuie HILECOP. Ainsi, un système numérique complexe critique est vu comme :

— Une architecture : pour faire face à la complexité du système, la description de l'architecture est basée sur un modèle à composants. Un composant est une entité manipulable et déployable, possédant une interface (ensemble de ports) et un comportement. Les composants peuvent être assemblés par interconnexion des ports de leurs interfaces. Ils peuvent également être "hiérarchisés" au sens de composants incluant des composants, dès lors appelés composites (figure 1.1). L'approche à composants favorise donc la conception par raffinement et/ou composition, la modularité et la réutilisabilité.

— Un comportement : pour répondre tant au besoin de modélisation du comportement qu'à son analyse formelle, le comportement de chaque composant est basé sur les réseaux de Petri (figure 1.2). L'assemblage

1. Field Programmable Gate Array
2. Application Specific Integrated Circuit

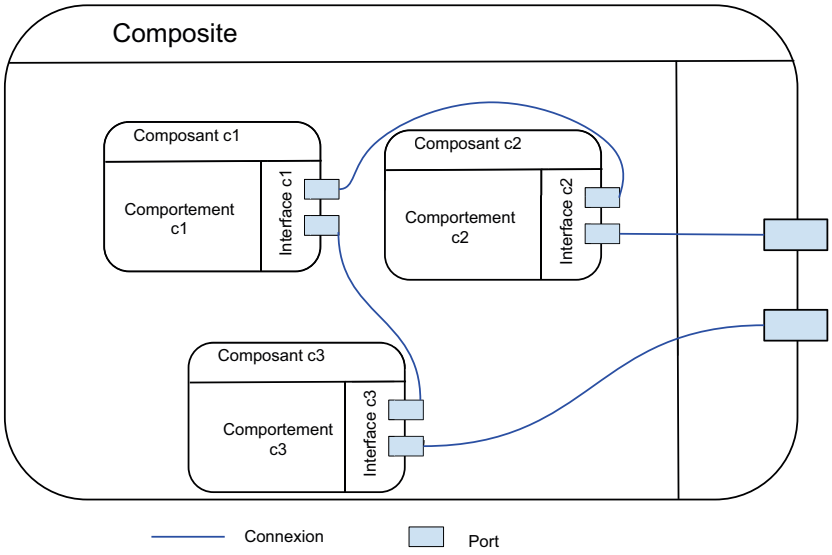


FIGURE 1.1 – Exemple de composants interconnectés dans un composite

de composants lors de la construction de l'architecture induit donc la composition des comportements de ces composants, donnant lieu alors à un modèle global au sens d'un modèle décrivant le comportement de l'intégralité de l'architecture. Il est alors possible d'effectuer une analyse formelle tant du comportement de chaque composant (ou composite) que du comportement du système numérique complet. Parmi les multiples formalismes existants (Machines à états finis [27], Grafcet [32], Statecharts [51], SyncCharts [2], etc.), le recours aux réseaux de Petri est, en plus de son côté formel, aussi motivé par son côté graphique. Ils permettent en effet de représenter explicitement la séquentialité, le parallélisme, la synchronisation, la concurrence et le partage de ressources, le raffinement, la composition, la hiérarchisation, etc. et s'avèrent être un vrai langage graphique et formel de conception pour les ingénieurs, qui peuvent aisément composer (assembler) leurs modèles et visualiser l'évolution de leur modèle même dans le cas d'une structure fortement parallèle.

— Un déploiement : la projection de l'architecture à composants sur une architecture matérielle, comprenant un ou plusieurs FPGA interconnectés, conduit à répartir les composants sur des domaines d'horloge potentiellement différents (figure 1.3). Soit les composants sont tous

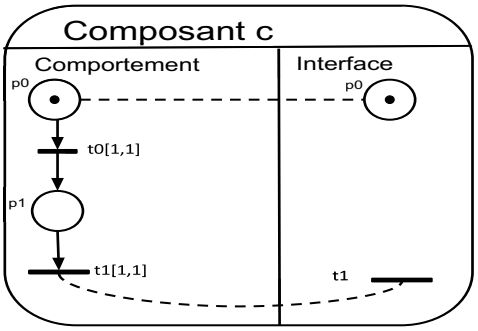


FIGURE 1.2 – Exemple de composant dont le comportement est formalisé par RdP

sur le même FPGA et fonctionnent à la même horloge, soit ils sont sur un même FPGA mais ont des horloges différentes, soit ils sont sur des FPGA différents et donc nécessairement leurs horloges sont considérées différentes. Cette précision vise uniquement à souligner que le comportement global est dans certains cas assimilable à celui d'un système GALS : Globalement Asynchrone Localement Synchrone [75]. Il s'agit là aussi d'une caractéristique non-fonctionnelle, visant en particulier à permettre la minimisation de la consommation en faisant fonctionner chaque composant ou composite juste à la fréquence requise.

— Un contrôle d'activité : tous les composants ne doivent pas nécessairement être actifs tout le temps. Dès lors il peut être intéressant de contrôler l'activité des composants pour économiser de l'énergie. Ce contrôle s'exerce par le biais de l'horloge, selon le principe du "clock gating". Ainsi, l'horloge d'un composant ou d'un composite, voire d'un ensemble de composants ou de composites, peut être contrôlée par un autre composant via un port d'interface dédié ; si l'horloge est "verrouillée" leur comportement n'évolue donc pas. C'est basé sur le principe de propagation d'activité au niveau composant : un composant peut activer ou non l'horloge d'un autre composant pour que son comportement soit exécuté. Cette représentation simplifiée (figure 1.4) montre schématiquement les dépendances en termes d'activation et de désactivation, voire d'auto-désactivation (au sens où le composant s'arrête une fois qu'il a terminé son traitement). Par exemple, dès que le coupleur physique reçoit une trame, il active le composant interpréteur qui comprend le reste de la pile protocolaire, qui active à son tour le composant

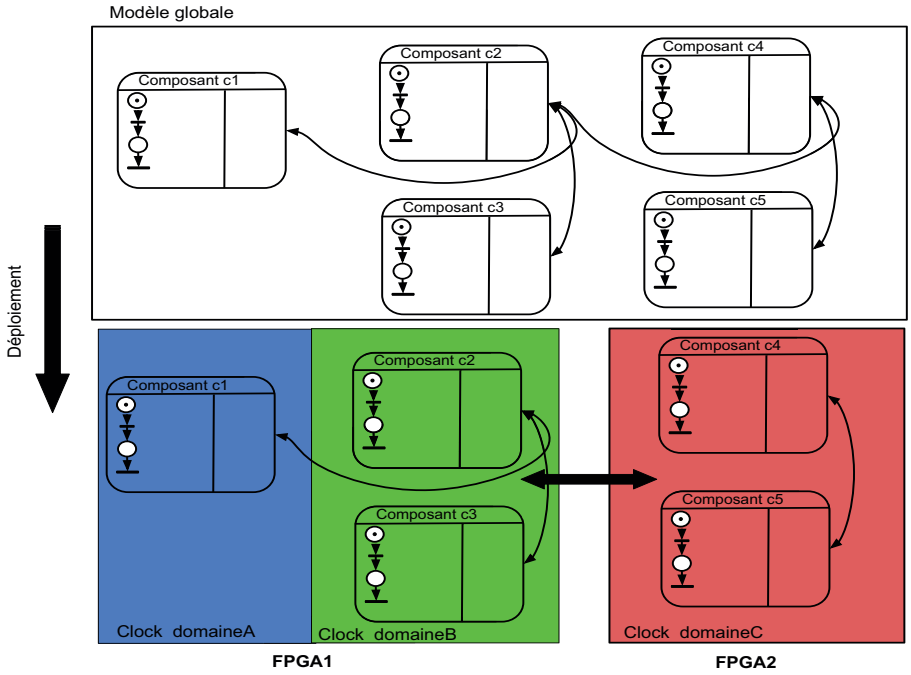


FIGURE 1.3 – Exemple de déploiement de composants sur différents domaines d'horloge

micro-machine (l'exécuteur des profils de stimulation, cf. chapitre 6) si la requête reçue correspond au lancement de la stimulation. La micro-machine désactivera son propre fonctionnement une fois la stimulation terminée.

La méthodologie HILECOP ne saurait se limiter à la conception. Nous avons en effet mentionné que :

— Le modèle comportemental, en l'occurrence le réseau de Petri, sera mis en oeuvre au sein d'un circuit électronique. Qu'il s'agisse d'un FPGA ou d'un ASIC, le langage de description retenu est le langage standardisé VHDL[3] [91]. Au regard de la complexité du système et de la nécessité de minimiser l'intervention humaine dans la phase de programmation, le code VHDL équivalent au modèle est automatiquement généré par HILECOP.

— Le modèle comportemental, qu'il soit partiel (un ou un sous-ensemble de composants) ou global (tous les composants assemblés), doit pouvoir être analysé. De fait, HILECOP génère automatiquement une descrip-

3. VHSIC Hardware Description Language

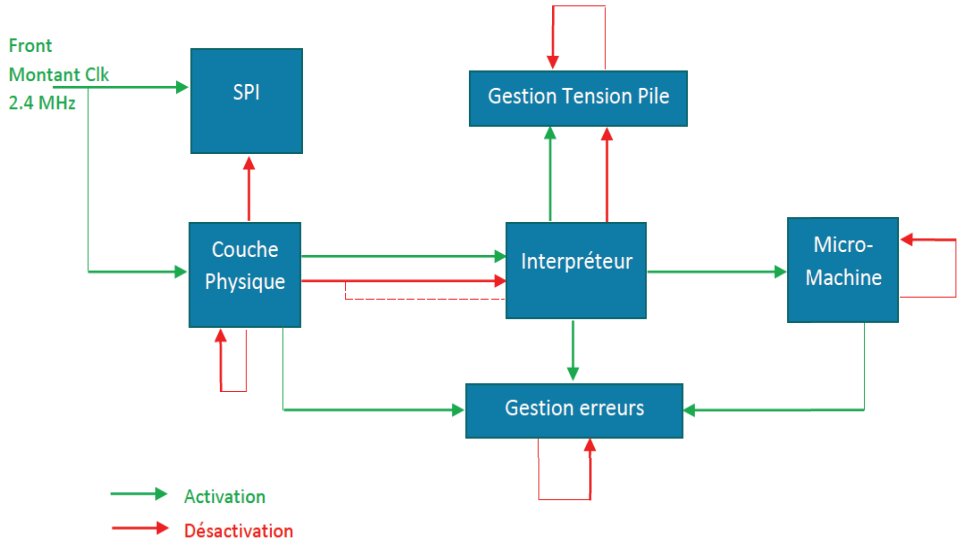


FIGURE 1.4 – Représentation schématique du contrôle d'activité au sein de l'architecture numérique d'un DMIA

tion équivalente dans le langage PNML [4] [52] afin de pouvoir exploiter tout analyseur acceptant ce langage normalisé en entrée.

— Le modèle comportemental doit, en d'autres termes, constituer un langage de programmation abstrait facilitant le travail des ingénieurs (à l'image du Grafcet par exemple).

Nos travaux portent sur cette relation entre le modèle initial et le code généré (issu du modèle implémenté, voir figure 1.5), et comment cela doit être pris en compte dans l'analyse formelle. Un code VHDL décrit certes des "relations logiques" mais aussi la manière selon laquelle ces relations logiques sont exécutées au sein du circuit. C'est donc cette génération de code que nous allons approfondir pour comprendre comment le modèle sera exécuté sur la cible (FPGA), et faire émerger les caractéristiques non-fonctionnelles qui doivent impérativement être prises en considération dès l'analyse formelle. En d'autres termes, il est nécessaire de : 1/ capturer tout ce que "cache" cette transforma-

4. Petri Net Markup Language

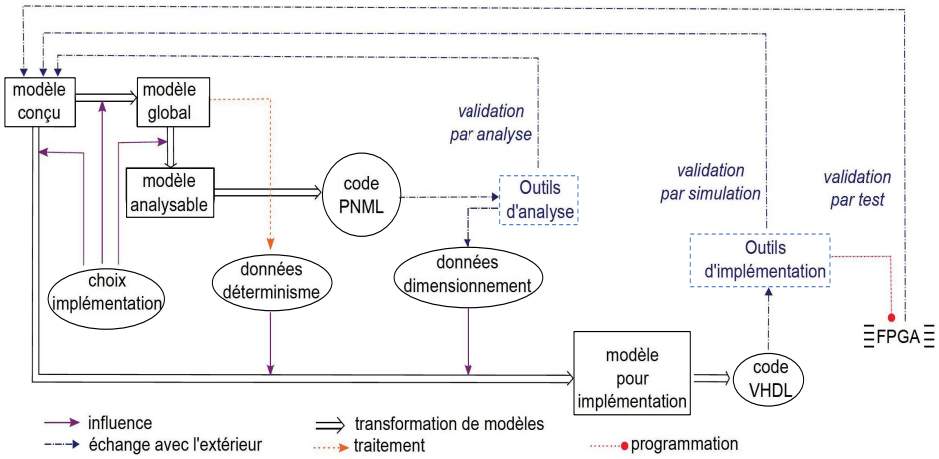


FIGURE 1.5 – Schéma général d'HILECOP

tion de modèles (en l'occurrence une transformation de type "model-to-text", du modèle RdP initial au code VHDL), 2/ faire évoluer le formalisme du modèle pour exprimer ces informations significatives et 3/ proposer une technique d'analyse formelle adéquate, i.e. prenant en considération les caractéristiques non-fonctionnelles retranscrites sur le modèle.

Nous allons décrire en détails le formalisme RdP à partir duquel le système initial est modélisé, mais avant cela rappelons les fondamentaux sur les réseaux de Petri.

1.1.2 Les Réseaux de Petri

Le réseau de Petri (RdP) est un modèle mathématique proposé initialement par Carl Adam Petri dans sa thèse de doctorat [81]. Ils servent à modéliser et vérifier divers systèmes, initialement à événements discrets. Une représentation graphique est associée au formalisme. Les RdP permettent en effet la représentation graphique de comportements complexes comprenant du parallélisme, de la synchronisation, de la concurrence, du partage de ressources, etc. Comme nous l'avons mentionné, la représentation graphique est très utile pour visualiser le modèle, qui sert alors de support aux ingénieurs lors de la conception. Un modèle est constitué de places et transitions, interconnectées par des arcs (cf. exemple figure 1.6) pour former un graphe orienté :

— une place est représentée par un cercle, elle peut contenir des jetons. La

distribution des jetons dans les places, appelée le marquage, représente l'état courant du système.

— une transition est représentée par un rectangle ou un trait. Elle est liée à des places amont et aval par des arcs. Une transition est sensibilisée s'il y a suffisamment de jetons dans les places amont. Le tir (ou franchissement) d'une transition représente un changement d'états. Lors du tir, des jetons sont retirés des places amont et des jetons sont ajoutés dans les places aval, selon les poids associés aux arcs reliant la transition à ses places amont et aval.

— un arc est représenté par une flèche (lien orienté). Il relie les places et les transitions, aucun arc ne relie deux éléments de même type. Les arcs sont porteurs d'un poids (1 par défaut) indiquant le nombre de jetons nécessaires pour sensibiliser la transition (lien T-P) ou produits suite au tir (lien P-T).

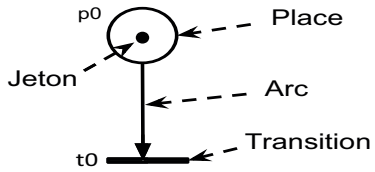


FIGURE 1.6 – Exemple d'un RdP

Définition formelle

Formellement, un RdP est décrit par un n-uplet $< P, T, Pre, Post, m_0 >$ tel que :

1. P est l'ensemble des places, T l'ensemble des transitions,
2. m_0 est le marquage initial,
3. $Pre : P \times T \to \mathbb{N}$ est la fonction de précondition et $Post : P \times T \to \mathbb{N}$ celle de postcondition. Ces fonctions définissent les relations d'incidence (les arcs) existant entre les places et les transitions.Pour une transition t, on note $Pre(t)$ et $Post(t)$ les ensembles des places en amonts et en avales, respectivement.

extensions des RdP

De nombreuses classes de RdP ont été introduites au fil du temps. Nous ne passerons pas en revue toutes ces classes mais précisons succinctement quelques extensions que nous exploitons. Chaque classe est caractérisée de façon unique selon les concepts supportés pour la modélisation et la sémantique d'exécution. Les différentes classes de RdP peuvent être classées comme autonomes ou non-autonomes, principalement au regard des définitions de sensibilisation et de tir d'une transition [33].

1. RdP autonome : L'évolution d'un RdP autonome n'est pas affectée par son environnent externe. L'instant de tir d'une transition n'est pas défini ni indiqué concrètement. Cette classe de RdP est utilisé principalement pour modéliser les systèmes distribués. Nous pouvons distinguer les extensions suivantes :
   — RdP généralisé [65] : Il accepte un nombre entier de jetons dans les places. Le flux de jetons dans le modèle est décrit par l'intermédiaire de poids (strictement positifs) associés aux arcs. Une transition n'est sensibilisée que si le nombre de jetons dans une place amont est supérieur au poids de l'arc les reliant. La transition peut alors être tirée. Chaque arc indique le nombre de jetons qui doivent être retirés ou ajoutés dans les places (respectivement amont et aval) lors d'un tir de transition. La pondération est utilisée par exemple pour modéliser des assemblages/décompositions d'entités, des ressources, des compteurs, etc. Un RdP est dit ordinaire si tous ses arcs ont un poids de 1. Un RdP ordinaire est un graphe d'événements si toutes ses places n'ont qu'une seule transition en entrée et en sortie, et il est une machine à états si toutes ses transitions n'ont qu'une place en entrée et en sortie.
   — RdP étendu [99] : Cette classe de RdP contient deux types d'arcs particuliers, en plus des arcs classiques, qui permettent une amélioration du pouvoir d'expression du modèle :
      (a) Arc de test : Cet arc connecte toujours une place à une transition. Il permet de tester si le marquage d'une place sensibilise ou non une transition. Dans le cas de RdP généralisés, l'arc test vérifie que la place amont possède au moins n jetons, n étant le poids

de l'arc test. Le tir de la transition est neutre pour la place concernée, i.e. il n'enlève pas de jeton à la place liée à cette transition par l'arc de test. Cet arc est habituellement représenté par une flèche avec un cercle plein à l'extrémité.

(b) Arc inhibiteur : Comme l'arc de test, l'arc inhibiteur connecte toujours une place à une transition. Il permet de tester si le marquage d'une place est nul. Une transition est donc sensibilisée, via un arc inhibiteur, seulement si la place amont est vide. Dans le cas de RdP généralisés, l'arc inhibiteur vérifie si la place amont possède moins de n jetons, n étant le poids de cet arc. L'arc inhibiteur est représenté par une flèche avec un cercle vide à l'extrémité (à l'image d'un inverseur en électronique). Le tir de la transition est également neutre pour la place concernée, i.e. il n'enlève pas de jeton de la place amont.

2. RdP non-autonome : un RdP non-autonome est dépendant de son environnement, en termes d'état ou de temps. Le tir de transition d'un RdP non-autonome est synchronisé par un événement (externe). Cette classe de RdP est utilisée pour modéliser les systèmes qui sont dépendants de leur environnement comme par exemple les systèmes de contrôle.
   — RdP synchronisé [74, 33] : La synchronisation est une sorte d'interprétation ; le tir des transitions peut être conditionné par l'occurrence d'événements externes. Un événement peut être associé avec plusieurs transitions, qui seront toutes tirées simultanément (si sensibilisées). Par contre, les RdP synchronisés ne permettent pas l'occurrence de plusieurs événements au même instant, et considèrent que le tir d'une transition est instantané. Il n'y a pas de représentation quantitative du temps.
   — RdP interprété [86, 33, 46] : Le concept global dans les RdP interprétés est l'association de transitions et de places du modèle respectivement avec les entrées / sorties du système. Une transition sera tirée si et seulement si l'entrée associée (état d'un capteur, d'une variable) est activée. Les RdP interprétés ont en général les mêmes hypothèses, précédemment citées, que les RdP synchronisés.
   — RdP temporisé [109] : Un RdP temporisé est obtenu en associant une durée soit aux transitions (RdP T-temporisé), soit aux places

(RdP P-temporisé). La transformation d'un modèle à un autre est possible [33]. Dans un RdP T-temporisé, la durée de temps (continu ou discret) associée à une transition reflète le temps requis par un processus ou une attente. Les RdP temporisés sont utilisés principalement pour l'évaluation de performances des systèmes.

— RdP temporel [109] : Les RdP temporels les plus courants, les RdP t-temporels, associent un intervalle de temps (continu ou discret) à chaque transition afin de conditionner ou planifier son tir. Ce formalisme est utilisé pour modéliser des propriétés temporelles, par exemple issues d'un chien de garde (watchdog) ou d'une temporisation. L'association du temps aux transitions rend la modélisation de la concurrence et du parallélisme plus explicite, car la dimension temporelle précise l'évolution d'états dans le temps.

Définition des conflits

Quelle que soit la classe de RdP, il est nécessaire de considérer les possibles conflits dans l'évolution d'états du modèle, a fortiori dans notre contexte de réalisation de systèmes numériques critiques. De fait, explicitons les situations de conflits au sein d'un RdP.

La définition classique du conflit structurel est que des transitions ont des places amont en commun. La figure 2.8 montre un conflit structurel entre les deux transitions t_0 et t_1. Formellement, les deux transitions t_0 et t_1 sont en conflit structurel ssi : $Pre(t_0) \cap Pre(t_1) \neq \emptyset$.

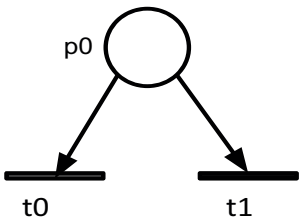


FIGURE 1.7 – Illustration d'un conflit structurel

Un conflit structurel ne conduit pas nécessairement à un conflit effectif. La notion de conflit effectif prend également en considération le marquage. Pour un marquage m, deux transitions sont en conflit effectif si elles sont en conflit

structurel et que le nombre de jetons n'est pas suffisant pour tirer les deux transitions. Dans le cas des réseaux de Petri temporels, la définition usuelle de conflit effectif devra être complétée pour prendre en compte le temps. Globalement, la définition d'un conflit effectif se compose de deux conditions i) les deux transitions sont tirables à un même instant ; ii) le tir d'une transition empêche le tir de l'autre.

Les éléments principaux d'un réseau de Petri étant posés, nous allons maintenant présenter le formalisme RdP utilisé pour modéliser le système initial dans la méthodologie HILECOP.

1.1.3 Le formalisme ITPN

Le modèle initial, au sens de celui établi par l'ingénieur (cf. figure 1.5), du comportement du système est modélisé à l'aide de réseaux de Petri temporels généralisés étendus interprétés, que nous nommerons pour simplifier *Interpreted Time Petri Nets* - ITPN. Ce sont les réseaux de Petri temporels classiques [73] (c'est à dire avec un intervalle temporel associé aux transitions), enrichis de poids sur les arcs et de l'utilisation des arcs tests et inhibiteurs. A cela, s'ajoute l'interprétation qui permet de représenter l'interaction du système avec son environnement. L'utilisation de réseaux de Petri associés à une interprétation est relativement courante, en particulier dans les domaines des systèmes à événements discrets et des systèmes logiques de contrôle (par exemple les *Signal Interpreted Petri Nets* - SIPN [107] ou les *Control Interpreted Petri Nets* - CIPN [46]).

Par contre, l'interprétation dans notre cas diffère un peu des RdP interprétés de la littérature : elle n'est pas réduite à la forme traditionnelle d'entrées/sorties, mais est constituée de trois éléments différents : les actions, les fonctions et les conditions qui expriment, comme précédemment indiqué, les interactions du système avec son environnement à des instants spécifiques de son évolution. L'interprétation est associée au modèle RdP comme suit :

— Les conditions sont des expressions logiques, associées aux transitions, qui influencent le comportement du système en autorisant ou non (suivant leur valeur `true` ou `false`) le tir des transitions concernées.
— Les fonctions (aussi appelées actions impulsives) sont également asso-

ciées aux transitions, et sont exécutées lors du tir de ces transitions.

— Les actions (aussi appelées actions continues) sont associées aux places et exécutées de façon continue tant que les places concernées sont marquées.

Un exemple de modèle ITPN est donné sur la figure 1.8.

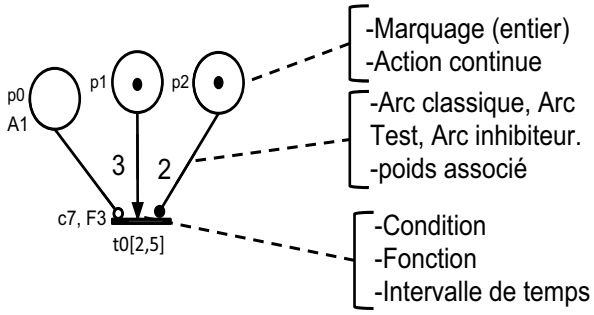


FIGURE 1.8 – Exemple de modèle ITPN

Définition formelle du modèle ITPN

La définition formelle du formalisme ITPN que nous utilisons a été décrite dans [62]. Nous en présentons ici une adaptation, avec comme principale différence que nous n'exprimons pas ici les priorités. Ces dernières, nécessaires pour garantir le déterminisme de l'exécution du modèle sur la cible matérielle dans le cas de conflits effectifs, seront introduites dans la section 1.1.5.

Soit $\mathbb{I}^*$ l'ensemble des intervalles non vides d'entiers positifs. Pour un intervalle $I \in \mathbb{I}^*$, $\downarrow I \in \mathbb{N}^*$ et $\uparrow I \in \mathbb{N}^* \cup \{+\infty\}$ représentent respectivement ses bornes minimale et maximale. Soit $\mathcal{C}$ l'ensemble des conditions, une condition étant une expression booléenne, et $\mathbb{B}$ l'ensemble des booléens $\{0, 1\}$. Soit $\mathcal{F}$ l'ensemble des fonctions (actions impulsives), et $\mathcal{A}$ l'ensemble des actions (actions continues), décrites en VHDL.

Un modèle ITPN est un n-uplet $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, I_s >$, tel que :

1. $< P, T, Pre, Pre_t, Pre_i, Post, m_0 >$ est un réseau de Petri généralisé étendu. P est l'ensemble des places, T l'ensemble des transitions, m_0 le marquage initial. $Pre, Pre_t, Pre_i, Post : T \times P \rightarrow \mathbb{N}$ sont respectivement les fonctions pré-condition, test, inhibiteur et post-condition.

2. $C : T \times \mathcal{C} \to \mathbb{B}$ est la fonction condition telle que $C(t, c) = 1$ signifie que la condition $c \in \mathcal{C}$ est associée à une transition $t \in T$. $\forall c \in C$, Si $C(t, c) = 0$, alors aucune condition n'est associée avec la transition t.
3. $F : T \times \mathcal{F} \to \mathbb{B}$ est la fonction des actions impulsives : si une fonction $f \in \mathcal{F}$ est associée avec une transition $t \in T$ alors $F(t, f) = 1$, sinon $F(t, f) = 0$.
4. $A : P \times \mathcal{A} \to \mathbb{B}$ est la fonction des actions continues. Elle est définie de la même façon que F.
5. $I_s : T \to \mathbb{I}^* \cup \emptyset$ est la fonction associant les intervalles temporels statiques aux transitions.

Un marquage d'un modèle ITPN est défini par une fonction $m : P \to \mathbb{N}$. Une transition $t \in T$ est sensibilisée par un marquage m ssi toutes ses places amont ont le nombre de jetons exigé pour le tir, i.e. $(m \geq Pre(t) + Pre_t(t)) \wedge (m < Pre_i(t))$. Nous notons l'ensemble des transitions sensibilisées par un marquage m par $enabled(m)$.

Une transition $k \in T$ est dite nouvellement sensibilisée par le tir d'une transition $t \in T$ depuis un marquage m ssi :

— Si $k \neq t$ est sensibilisée par le nouveau marquage $m' = m - Pre(t) + Post(t)$ mais ne l'était pas par le marquage intermédiaire $m - Pre(t)$,
— Si $k = t$ et qu'elle reste sensibilisée par le nouveau marquage m'.

L'ensemble des transitions nouvellement sensibilisées par le tir d'une transition $t \in T$ depuis un marquage m est noté $newEnabled(m, t)$.

Formellement, $k \in newEnabled(m, t)$ ssi :

$$\begin{aligned}(m - Pre(t) + Post(t) \geq Pre(k) + Pre_t(k)) \\ \wedge (m - Pre(t) + Post(t) < Pre_i(k)) \\ \wedge [(k = t) \vee (Pre_i(k) \leq m - Pre(t)) \\ \vee (Pre(k) + Pre_t(k) > m - Pre(t))]\end{aligned}$$

L'état d'un modèle ITPN est défini par $s = (m, I)$, avec :

— m est le marquage.
— $I : T \to \mathbb{I}^* \cup \Phi$ est la fonction qui associe, à chaque transition sensibilisée par m, soit un intervalle dynamique de temps soit Φ dans le cas où la

transition est bloquée. L'intervalle dynamique représente l'écoulement du temps θ par rapport à l'intervalle statique (dès lors que la transition est sensibilisée) ; il représente donc le temps restant pour le tir à travers un intervalle temporel dont les bornes sont respectivement $\downarrow I_s - \theta$ et $\uparrow I_s - \theta$.

Définition de la sémantique du modèle ITPN

La sémantique du modèle ITPN que nous donnons ici est une adaptation de celle donnée dans [62]. C'est une sémantique qui regroupe la sémantique classique des TPN [109] (étendue avec les arcs inhibiteurs et tests) avec une sémantique d'interprétation adaptée à notre contexte. Nous devons donc considérer ces deux aspects pour définir notre sémantique ITPN.

Les sémantiques classiques liées aux réseaux de Petri interprétés sont majoritairement issues de la programmation des PLC (Programmable Logic Controller) [108, 47, 33]. Dans notre cas, nous retrouvons la représentation des conditions sous forme d'équations logiques booléennes. Cela se traduit pas l'utilisation d'une fonction $val : \mathbb{C} \rightarrow \mathbb{B}$ qui représente la valeur instantanée d'une condition. L'exécution des fonctions (actions impulsives) et des actions (continues) n'est pas directement exprimée dans notre sémantique. En effet, ces actions n'influencent le comportement du système que par l'intermédiaire de la modification des valeurs de signaux ou de variables internes, ce qui a pour conséquence la modification de la valeur des conditions, qui elles sont intégrées à la sémantique d'exécution de notre formalisme.

Ainsi, une transition est tirable à partir d'un état $s = (m, I)$ ssi : elle sensibilisée par m, la valeur des conditions qui lui sont associées autorise son tir, la borne minimale de l'intervalle de tir est atteinte et la borne maximale n'est pas dépassée (formalisée par $0 \in I(t)$). L'ensemble des transitions tirables à partir de s est noté $Firable(s)$. Formellement, $t \in Firable(s)$ ssi :

$$(t \in enabled(m)) \wedge (0 \in I(t)) \wedge \big(\forall c \in C, C(t,c) = 1 \wedge val(c) = 1\big)$$

Pour les RdP temporels, il existe deux sémantiques de tir classiques : la sémantique forte et la sémantique faible [21]. Dans le cas de la sémantique forte, le tir d'une transition peut se produire à tout moment pendant son

intervalle temporel, et doit obligatoirement se produire à la date de tir au plus tard. La sémantique faible définit qu'une transition sensibilisée n'est pas obligatoirement tirée même si la date au plus tard de son intervalle de tir est dépassée. Dans ce cas, cette transition n'est plus tirable jusqu'à ce qu'elle soit de nouveau sensibilisée.

Dans nos travaux nous considérons une sémantique forte adaptée, telle que définie dans [63], que nous qualifierons de sémantique "impérative". A savoir, une transition tirable doit être tirée au plus tôt. Dans le cas d'une transition temporelle ayant également une condition associée, si la valeur de la condition n'est toujours pas vraie à la date de tir au plus tard, cette transition sera bloquée (elle ne peut plus être tirée jusqu'à ce qu'elle soit de nouveau sensibilisée). Pour le déblocage, le concepteur devra prévoir une structure de déblocage explicite dans le modèle ; d'où l'importance de prendre en considération ces situations dans l'analyse, dans la mesure où il est impossible parfois de garantir qu'un capteur, par exemple, sera bien activé dans l'intervalle temporel prévu.

Ainsi, la sémantique du formalisme ITPN est un système de transitions temporisé $(S, s_0, \rightsquigarrow)$, où S est l'ensemble des états, $s_0 = (m_0, I_0)$ est l'état initial et $\rightsquigarrow \subseteq S \times (T \cup \mathbb{R}^+) \times S$ est la relation de changement d'états (aussi appelée transition d'états) notée :

— Pour $t \in T$, $s \overset{t}{\rightsquigarrow} s'$.

— Pour $\theta \in \mathbb{R}^+$, $s \overset{\theta}{\rightsquigarrow} s'$.

La relation de changement d'états est définie comme suit :

— Changement d'état discret : $s = (m, I) \overset{t}{\rightsquigarrow} s' = (m', I')$, ssi $t \in T$ et :

1. $t \in Firable(s)$
2. $m' = m - Pre(t) + Post(t)$
3. $\forall k \in T$, if $k \in newEnabled(m, t), I'(k) = I_s(k)$, sinon $I'(k) = I(k)$.

— Changement d'état continu : $s = (m, I) \overset{\theta}{\rightsquigarrow} s' = (m, I')$, ssi $\theta \in \mathbb{R}^+$ et :

1. $\forall t \in T$, $I_s(t) \neq \emptyset \wedge I(t) \neq \Phi \wedge t \in enabled(m) \Rightarrow \theta \leq \uparrow I(t)$.
2. $\forall t \in T$, $I_s(t) \neq \emptyset \wedge I(t) \neq \Phi \wedge t \in enabled(m) \Rightarrow I'(t) = I(t) - \theta$.
3. $\forall t \in T$, $I(t) = \Phi \Rightarrow I'(t) = I(t)$.

— Changement d'état de blocage : $s = (m, I) \overset{t}{\rightsquigarrow} s' = (m, I')$, ssi $t \in T$ et :

1. $t \in enabled(m) \wedge \forall c \in C \mid C(t, c) = 1, val(c) = 0 \wedge \uparrow I(t) = 0 \Rightarrow I'(t) = \Phi$

2. $\forall k \in T \setminus t, I'(k) = I(k)$

Le formalisme et la sémantique initiaux du modèle ITPN étant décrits, nous allons considérer l'exécution du modèle sur la cible. Pour cela, il faut se pencher sur l'exécution du code généré à partir du modèle initial. Nous allons donc maintenant décrire brièvement la génération du code VHDL, permettant la mise en oeuvre sur cible FPGA, et faire émerger quelques-unes des caractéristiques non-fonctionnelles selon lesquelles nous devrons faire évoluer le formalisme afin de refléter les contraintes réelles d'exécution.

1.1.4 Génération de code et principe d'exécution sur la cible

La transformation de modèle RdP en code VHDL exploite l'approche composants du langage VHDL. Ainsi, une place et une transition sont décrits par deux descripteurs de composants VHDL dédiés (pour simplifier l'explication ces descripteurs seront appelés composants). Chaque place du RdP donne lieu à une instance de composant VHDL Place, et chaque transition du RdP donne lieu à une instance de composant VHDL Transition (Figure 1.9). Les arcs du RdP sont traduits par l'interconnexion des instances des composants VHDL Place et Transition (Figure 1.1.4). Les arcs sont pris en compte dans les composants places, c'est à dire que les calculs impliquant le poids des arcs (sensibilisation d'une transition et mise à jour du marquage) sont effectués dans le composant Place.

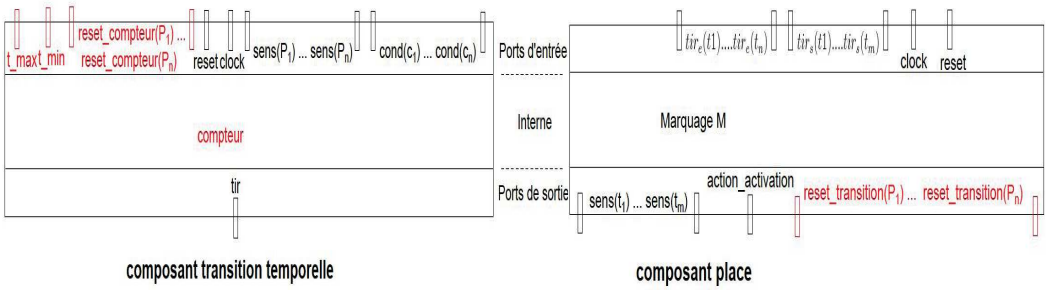


FIGURE 1.9 – Représentation simplifiée des composants VHDL Place et Transition [60]

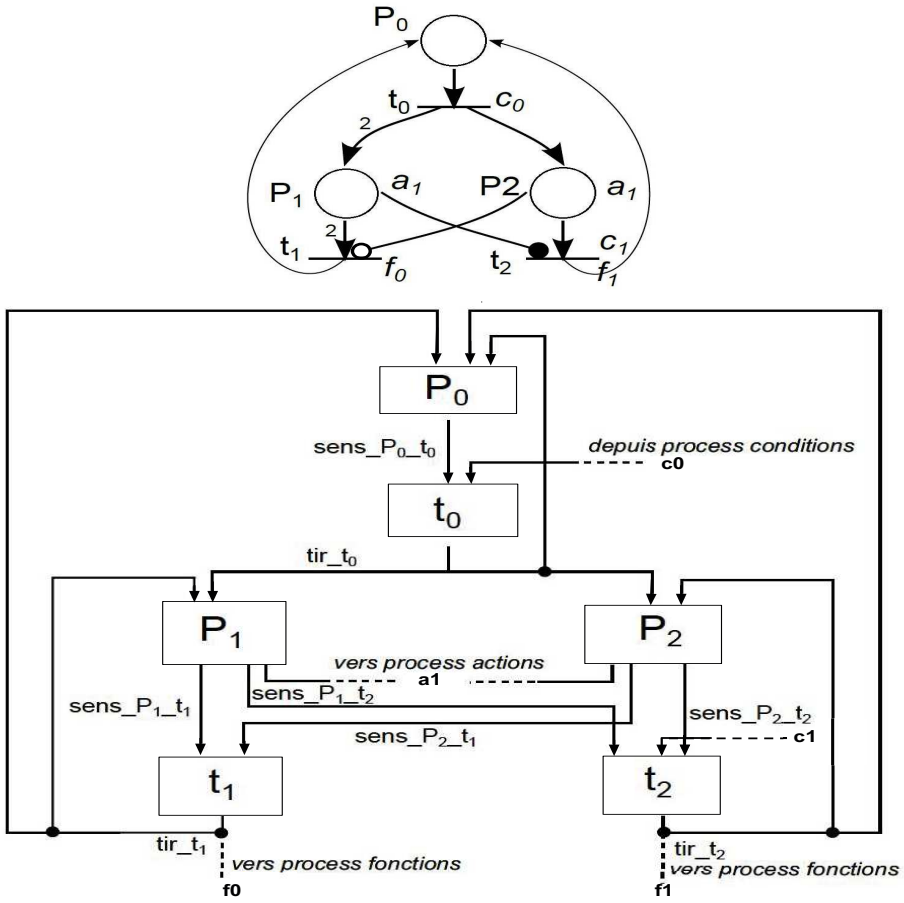


FIGURE 1.10 – Exemple de transformation d'un RdP en instances VHDL interconnectées

Les règles d'évolutions du RdP sont directement traduites dans les composants VHDL Place et Transition :

— Transition : elle reçoit de chaque place amont un signal lui indiquant si cette place la sensibilise (cf. Place). Si toutes ses places amont la sensibilisent, alors la transition est sensibilisée. Dès lors, elle vérifie si sa condition associée est vraie. De plus, si la transition est temporelle, alors l'intervalle de temps associé est considéré : elle vérifie si le temps courant relatif (son propre compteur) est compris dans l'intervalle statique défini. Si tout cela est vérifié la transition est alors tirable. Le tir consiste à réinitialiser le compteur de temps, à envoyer un signal aux places amont et aval leur indiquant le tir de la transition et à lancer l'exécution de sa fonction associée lorsqu'elle en a une. Une transition temporelle sensibilisée mais non franchissable fait évoluer son comp-

teur de temps, tant que la borne supérieure n'est pas dépassée. Sinon, dans le cas contraire, le compteur de temps n'évolue plus mais la transition n'est plus franchissable tant qu'elle n'est pas désensibilisée puis re-sensibilisée. Il s'agit d'un cas de blocage que nous traiterons plus en détails par la suite.

— Place : une place compare les poids de ses arcs sortants et son marquage courant et indique aux transitions concernées si elles sont sensibilisées par cette place ou non. Son marquage évolue selon les signaux de tir qui lui sont envoyés par ses transitions amont et aval. Pour chaque transition en sortie lui signifiant un tir, elle retire le nombre de jetons indiqué par l'arc la reliant à la transition tirée. Et pour chaque transition en entrée, la place ajoute à son marquage le nombre de jetons indiqué par l'arc la reliant à la transition tirée. Une fois le marquage actualisé, la place contrôle l'activation des actions.

Si l'évolution d'état peut être instantanée, et donc asynchrone sur le circuit, ce n'est pas le cas de l'exécution des fonctions lors du tir des transitions. Ces fonctions, pouvant potentiellement porter sur des variables du système qui sont impliquées sur des conditions de tir, doivent impérativement être terminées avant de pouvoir poursuivre l'évolution d'état. Or, il est impossible de savoir quand une fonction (code VHDL) a terminé de s'exécuter, i.e. que les signaux de sortie sont stables. Par conséquent, seule une exécution synchrone sur FPGA permet d'obtenir un comportement déterministe du système. En effet, baser l'évolution d'état sur une horloge permet de garantir que les fonctions exécutées lors du tir des transitions ont le temps de terminer leur exécution avant l'évaluation des conditions (la synthèse indiquant la fréquence maximale à laquelle le circuit peut fonctionner en respectant la stabilité et la propagation des signaux). C'est un élément important pour la cohérence d'état du système.

L'exécution est donc régie par un signal horloge dont les deux fronts sont exploités. Le principe d'exécution du modèle est schématisé Figure 1.11. Il s'agit d'une évolution synchrone du modèle, selon 4 étapes synchronisées par l'horloge. A chaque cycle d'horloge :

— Sur le front montant (1) : Mise à jour du marquage en fonction des transitions qui viennent d'être tirées. La modification du marquage peut durer la demi-période (2).

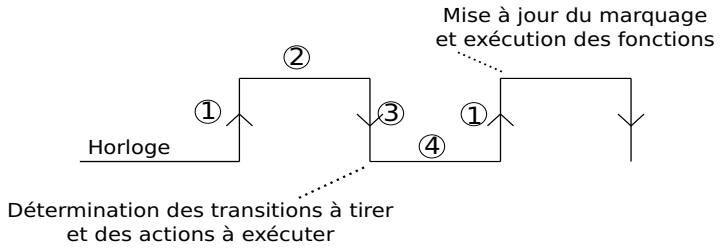


FIGURE 1.11 – Principe d'exécution synchrone [61]

— Sur le front descendant (3) : Évaluation, à partir de l'état courant (c'est à dire le marquage des places, la valeur des conditions et des compteurs temporels), des transitions devant être tirées lors de la demi-période (4).
— Les conditions sont calculées dans une logique combinatoire (i.e. en permanence) mais les valeurs prises en compte sont celles capturées au front descendant pour évaluer si les transitions sont tirables.
— Les fonctions et les actions sont exécutées sur des états stables : l'exécution des fonctions est démarrée lorsque le tir des transitions est terminé, c'est à dire sur le front montant suivant le tir des transitions, et les actions sont actualisées lorsque la mise à jour du marquage est terminée, c'est à dire sur le front descendant suivant.

Une exécution synchrone sur FPGA permet d'obtenir un comportement déterministe du système. Comme nous l'avons précisé, ce type d'implémentation garantit que les fonctions exécutées lors du tir des transitions ont le temps de terminer leur exécution avant l'évaluation des conditions. C'est un élément important pour la cohérence du système, car les fonctions peuvent influer sur la valeur de variables internes et donc sur la valeur des conditions, variables et conditions étant fondamentalement des signaux. Plus généralement, il s'agit donc, comme dans les langages synchrones, de s'assurer que le statut (état) des signaux soit défini (stable) avant toute utilisation.

Ce principe d'exécution du modèle sur la cible doit nécessairement être pris en compte dans le formalisme et la sémantique. Fondamentalement, dans la mesure où l'objectif est de décrire, pour l'analyser, le comportement effectif (réel sur la cible) il faut étudier et intégrer le "modèle de calcul" sous-jacent,

au sens général des données manipulées et des opérations réalisées sur ces données selon un ordre et un temps logiques définis (données et opérations étant ici relatives à l'exécution de l'évolution d'états).

1.1.5 Prise en compte de l'exécution synchrone dans le formalisme et la sémantique ITPN : SITPN

Dans le contexte de l'exécution sur FPGA, l'évolution du modèle ITPN est donc basée sur une horloge (les deux fronts d'horloge), puisque la mise en oeuvre est synchrone. Le cycle d'horloge est notre unité de temps logique, unité de temps au sein de laquelle plusieurs "traitements" ont lieu dans le cadre de l'évolution d'états du modèle comportemental du système numérique. De plus, l'exécution sur une cible avec du vrai parallélisme implique du "multi-firing" : lorsque plusieurs transitions sont tirables au même moment, elles sont effectivement tirées en même temps. Cela pose un problème en présence de conflits effectifs : deux transitions tirables en conflit vont être tirées simultanément en utilisant le même jeton, ce qui est contraire aux règles fondamentales des réseaux de Petri. La gestion des conflits peut être traitée par l'utilisation de priorités, comme montré dans [64]. Il est donc nécessaire d'intégrer les contraintes liées à l'exécution synchrone dans la définition et la sémantique du formalisme ITPN, afin de définir un formalisme proche du comportement effectif sur la cible FPGA, que l'on appellera SITPN (Synchronous Interpreted TPN). Ce formalisme a été initialement décrit dans [60].

Définition du modèle SITPN

A la définition précédente du modèle ITPN nous rajoutons donc une horloge et les priorités : un modèle SITPN est un n-uplet $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, I_s, clock, \succ >$ avec :

- *clock* le signal d'horloge qui synchronise l'évolution d'états du modèle, au sens de la synchronisation des opérations réalisées dans le cadre de cette évolution. $\uparrow clock$ et $\downarrow clock$ représentent les événements de front montant et de front descendant de l'horloge, respectivement. Cet ensemble est noté $CLK = \{\uparrow clock, \downarrow clock\}$.
- $\succ$: $T \times T \rightarrow \mathbb{B}$ la relation de priorité entre les transitions, supposée irréflexive, asymétrique et transitive.

Les fonctions de marquage et la sensibilisation des transitions sont les mêmes que pour le modèle ITPN. Par contre, la définition de l'ensemble des transitions nouvellement sensibilisées change, étant donné que pour le modèle SITPN plusieurs transitions peuvent être tirées en même temps. Ainsi, une transition $k \in T$ est dite nouvellement sensibilisée par le tir d'un ensemble de transitions $Fired \subset T$ depuis un marquage m ssi :

— Si $k \notin Fired$ est sensibilisée par le nouveau marquage m' mais ne l'était pas par $m - \sum_{t \in Fired} Pre(t)$;

— Si $k \in Fired$ reste sensibilisée par le nouveau marquage m'.

Nous notons l'ensemble de transitions nouvellement sensibilisées par le tir d'une ou d'un ensemble de transitions $Fired$ depuis un marquage m par $newEnabled(m, Fired)$. Formellement, on a $k \in newEnabled(m, Fired)$ ssi :

$$\begin{aligned}
&\left(m - \sum_{t \in Fired} Pre(t) + \sum_{t \in Fired} Post(t) \geq Pre(k) + Pre_t(k) \right) \\
&\quad \wedge \left(m - \sum_{t \in Fired} Pre(t) + \sum_{t \in Fired} Post(t) < Pre_i(k) \right) \\
&\quad \wedge \Bigg[(k \in Fired) \vee \left(Pre_i(k) \leq m - \sum_{t \in Fired} Pre(t) \right) \\
&\quad \vee \left(m - \sum_{t \in Fired} Pre(t) < Pre(k) + Pre_t(k) \right) \Bigg]
\end{aligned}$$

Un état d'un SITPN est défini par $s = (m, cond, ex, I, reset_t)$, où :

— m est le marquage.

— $cond : \mathcal{C} \to \mathbb{B}$ est la fonction qui retourne la valeur $val(c)$ de la condition $c \in \mathcal{C}$. A ce stade, il est en effet nécessaire de faire la distinction entre la valeur d'une condition, que l'on représente par la fonction val, et la valeur de cette même condition lue à un instant précis (dans notre cas sur le front descendant), que l'on représente par la fonction $cond$. En effet, sur la cible matérielle, les équations logiques des conditions sont codées, comme les actions impulsives ou continues, par une logique combinatoire, c'est à dire qu'elle est exécutée en asynchrone et que la valeur des conditions peut évoluer à chaque instant, dès que l'un des signaux im-

pliqués dans l'équation change. La fonction *cond* au contraire représente une image instantanée et stable de la valeur de toutes les conditions sur le front descendant, au moment de l'estimation des conditions de tir des transitions.

— $I : T \to I^* \cup \Phi$ est la fonction qui soit associe un intervalle dynamique de temps, soit Φ, à chaque transition sensibilisée par m.

— $ex : \mathcal{F} \cup \mathcal{A} \to \mathbb{B}$ est la fonction d'exécution des fonctions $f \in \mathcal{F}$ et des actions $a \in \mathcal{A}$.

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

Les conditions de tir d'une transition SITPN, et donc la définition de l'ensemble $Firable(s)$, sont les mêmes que pour les ITPN, à la différence près que la valeur des conditions utilisée sera $cond(c)$.

Sémantique du modèle SITPN

La sémantique d'un modèle SITPN est un système de transitions temporisé $(S, s_0, \rightsquigarrow)$, où [5] :

— S est l'ensemble des états,

— $s_0 = (m_0, 0, 0, I_0, 0)$ est l'état initial,

— $\rightsquigarrow \subseteq S \times CLK \times S$ est la relation de changement d'états, notée $s \overset{clk}{\rightsquigarrow} s'$, avec $clk \in CLK$. Soit l'ensemble de transitions $Fired \subseteq T$ tirées depuis l'état s (à l'état initial, $Fired = \emptyset$). Cette relation de changement d'états est définie comme suit :

  — $s = (m, cond, ex, I, reset_t) \overset{\downarrow clock}{\rightsquigarrow} s' = (m, cond, ex', I', reset_t)$, ssi l'événement $\downarrow clock$ a eu lieu, on a alors :

    1. $\forall c \in \mathcal{C}, cond'(c) = val(c)$.
    2. $\forall t \in enabled(m), reset_t(t) = 1 \ \vee \ t \in newEnabled(m, Fired) \Rightarrow I'(t) = I_s(t) - 1$.
    3. $\forall t \in enabled(m), reset_t(t) = 0 \ \wedge \ (t \notin newEnabled(m, Fired) \wedge I(t) \neq \Phi) \Rightarrow I'(t) = I(t) - 1$.
    4. $\forall t \in T \mid t \notin newEnabled(m, Fired) \wedge I(t) = \Phi \Rightarrow I'(t) = I(t)$

5. Cette définition formelle sera suivie d'une explication en "français".

5. $\forall a \in \mathcal{A}, \exists p \in P \mid A(p,a) = 1 \wedge m(p) \neq 0 \Rightarrow ex'(a) = 1$ sinon $ex'(a) = 0$.

   Il est alors possible de définir $Fired$, l'ensemble des transitions tirables qui seront effectivement tirées :

6. $\forall t \in Firable(s'), \forall t' \mid t' \succ t, t' \notin Firable(s') \Rightarrow t \in Fired$. $\forall t \in Firable(s')$, soit $Pr(t)$ l'ensemble des transitions t_i telles que $t_i \succ t \wedge t_i \in Fired$.
7. $\forall t \in Firable(s'), \Big(t \in enabled(m - \sum_{t_i \in Pr(t)} Pre(t_i)) \wedge t \in enabled(m - \sum_{t_i \in Pr(t)} Pre(t_i) + \sum_{t_i \in Pr(t)} Post(t_i))\Big) \Rightarrow t \in Fired$.
8. $\forall t \in Firable(s'), \Big(t \notin enabled(m - \sum_{t_i \in Pr(t)} Pre(t_i)) \vee t \notin enabled(m - \sum_{t_i \in Pr(t)} Pre(t_i) + \sum_{t_i \in Pr(t)} Post(t_i))\Big) \Rightarrow t \notin Fired$.
9. $\forall t \notin Firable(s') \Rightarrow t \notin Fired$.

— $s = (m, cond, ex, I, reset_t) \overset{\uparrow clock}{\rightsquigarrow} s' = (m', cond, ex', I', reset'_t)$, ssi l'évènement $\uparrow clock$ a eu lieu, on a alors :

1. $m' = m - \sum_{t \in Fired} Pre(t) + \sum_{t \in Fired} Post(t)$.
2. $\forall t \in T, \exists p \in P \mid m(p) - \sum_{t_i \in Fired} Pre(t_i, p) < Pre(t,p) + Pre_t(t,p) \Rightarrow reset'_t = 1$, sinon $reset'_t = 0$.
3. $\forall f \in \mathcal{F}, \exists t \in Fired \mid F(t,f) = 1 \Rightarrow ex'(f) = 1$, sinon $ex'(f) = 0$.
4. $\forall t \in T \mid t \notin Fired \wedge \uparrow I(t) = 0 \Rightarrow I'(t) = \Phi$, sinon $I'(t) = I(t)$.
5. $Fired' = Fired$.

Donc, cette sémantique, incluant la relation entre le modèle (qui s'apparente à un langage) et son modèle de calcul (sur la cible), peut être résumée selon les fronts de *clock* comme suit :

— Sur le front descendant $\downarrow clock$:

1. Les valeurs des conditions sont mises à jour.
2. L'intervalle de temps d'une transition sensibilisée t est réinitialisé si cet intervalle doit être réinitialisé (indiqué par le reset) ou bien

si cette transition est nouvellement sensibilisée par le tir d'une ou d'un ensemble de transitions. Comme le tir est effectué sur le front montant d'horloge précédent, alors l'intervalle de cette transition est réinitialisé à $I_s(t) - 1$.

3. Pour toutes les transitions sensibilisées qui ne sont ni nouvellement sensibilisées ni bloquées, et dont l'intervalle ne doit pas être réinitialisé, leurs intervalles de temps évoluent de manière classique.
4. Si une transition était bloquée et n'est pas nouvellement sensibilisée alors elle reste bloquée.
5. Les valeurs de la fonction ex pour les actions sont mises à jour.
6. A partir de ce nouvel état, l'ensemble des transitions qui seront effectivement tirées est alors déterminé. La différence entre tirables et tirées est basée sur la gestion des conflits : une transition ne sera tirée que s'il n'existe pas de transitions tirables plus prioritaires, ou s'il y a suffisamment de jetons pour la tirer ainsi que toutes les transitions plus prioritaires qu'elle.
7. Toutes les transitions tirables en situation de conflit sont tirées si le marquage est suffisant.
8. Si le marquage n'est pas suffisant pour tirer toutes les transitions tirables, alors les transitions tirables les moins prioritaire ne seront pas tirées.
9. Les transitions qui ne sont pas tirables, ne sont pas tirées.

— Sur le front montant $\uparrow clock$:

1. Mise à jour des marquages en fonction des transitions tirées.
2. Pour les transitions qui ont un marquage intermédiaire (transitoire) qui les désensibilise, les signaux $reset$ de réinitialisation des intervalles de temps sont mis à jour.
3. Mise à jour des valeurs de la fonction ex pour les actions impulsives.
4. Les transitions non tirées qui ont atteint la borne maximale de leur intervalles de temps sont bloquées. Les autres intervalles de temps ne sont pas modifiés.
5. L'ensemble des transitions tirées n'est pas modifié.

Jusqu'à présent, nous avons présenté ce qui permet de modéliser le fonctionnement normal d'un système. Cependant dans les systèmes complexes et critiques, il est également nécessaire de gérer les situations dites "anormales", au sens d'exceptions auxquelles il faut réagir. Nous allons brièvement exposer comment est assurée la gestion d'exception, avant de la prendre en compte dans le formalisme et la sémantique.

1.1.6 Gestion des exceptions

Une exception est une situation particulière (e.g., une erreur interne ou externe) qui se produit durant le fonctionnement du système. La gestion d'exception consiste à détecter, prendre en compte et réagir au plus vite selon une réaction définie. Modéliser la gestion des exceptions est souvent très compliqué pour un concepteur. En effet, il doit par définition prendre en compte les exceptions pour tous les états dans lequel le modèle peut potentiellement être lors de son occurrence. Dès que le modèle est un peu complexe, il est difficile, si ce n'est impossible, de garantir d'avoir traité tous les cas possibles ou nécessaires. De plus, modéliser cette prise en compte des exceptions par l'intermédiaire de places et transitions ajoute de nombreux éléments au modèle et donc le complexifie encore. Il faut impérativement faciliter l'expression (modélisation) de la gestion des exceptions afin de limiter ce risque d'erreur humaine et bien sûr ses conséquences en termes de comportements non désirés, voire dangereux.

Donc, dans un double but de renforcer la fiabilité et la "praticité" de l'approche, le formalisme de modélisation a été modifié par l'ajout d'un mécanisme spécifique pour la modélisation de la gestion des exceptions en s'assurant que le modèle sera toujours analysable et implémentable de manière efficace (i.e., maîtrise de l'impact sur la "surface" et la consommation du circuit). Ce mécanisme est basé sur la macroplace, avec transitions d'exception.

Macroplace

Le recours à la macroplace (notée MP) a été proposé par Hélène Leroux afin de représenter l'agrégation d'états [63]. Elle est représentée par un double cercle et elle contient un modèle SITPN (i.e., un modèle ITPN exécuté en syn-

chrone), appelé le raffinement. Un exemple de MP est donné sur la figure 1.12. Les "entrées/sorties" de la MP se font seulement à travers des arcs spécifiques, représentés par une flèche en pointillés. Une situation est associée à ces arcs pour décrire l'interaction entre la ou les transitions externes à la MP et le raffinement de celle-ci ; une situation exprime un marquage sur un ensemble de places internes à la MP. Selon le type d'arc et la situation associée, nous distinguons :

1. Arcs entrants de la MP : Ils relient nécessairement une transition externe et une MP. Ils sont caractérisés par une situation d'entrée qui représente un ensemble non vide de places du raffinement ainsi que le nombre de jetons qui seront ajoutés à chaque place de cet ensemble lors du tir de la transition. Dans l'exemple de la figure 1.12, le seul arc entrant relie la transition $t_{entrée}$ et la MP. La situation d'entrée qui lui est associée est $(p0, p1)$, c'est à dire que le tir de la transition $t_{entrée}$ ajoute un jeton dans la place $p0$ et un jeton dans la place $p1$ de la MP.
   Formellement, 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.

2. Arcs sortants de la MP : Ils relient nécessairement la MP avec une transition externe. Il existe deux types d'arcs sortants :
   — Arcs sortants classiques : Ils sont associés avec une situation de sortie classique qui est définie par le marquage minimum nécessaire pour sensibiliser la transition aval de la MP. Par exemple, sur la figure 1.12, la transition aval t_{sortie} est associée avec la situation de sortie classique $(p3)$. Donc, la transition t_{sortie} sera sensibilisée ssi la place $p3$ de la MP est marquée d'au moins 1 jeton.
   Formellement, une situation de sortie classique s_s est définie telle que $s_s = \{X n_i P_i\}$ où $n_i \in \mathbb{Z}\{0\}$ 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.

— Arcs sortants exception : Ce sont les arcs sortant qui sont associés avec des situations d'exception, notée $(*)$. Si une transition aval a au moins un arc d'exception alors elle est nommée "transition d'exception". La transition d'exception est sensibilisée par un arc d'exception ssi la MP est active, c'est à dire si son raffinement a au moins une place marquée. Par exemple sur la figure 1.12, la transition t_{exc} est une transition d'exception. Il est également possible de gérer une exception provoquée par une situation particulière interne à la MP. Par exemple, si un sous-ensemble particulier de places du raffinement est marqué alors une exception doit être déclenchée. Il est ainsi possible de combiner une situation de sortie avec une situation d'exception, l'arc d'exception possède alors la situation $(s_{exc}, *)$. Dans ce cas, s_{exc} ne peut contenir que des arcs tests ou inhibiteurs. L'exemple de MP dans la figure 1.13 montre ce cas. La transition d'exception t_{exc} est sensibilisé la place du raffinement p_5 est marqué. Une transition exception est nécessairement en conflit avec les autres transitions sortantes, et peut l'être avec d'autres transitions. Dans ce cas, des priorités sont définies entre les transitions, et la transition exception est toujours la plus prioritaire par rapport à toutes les transitions du raffinement.
Formellement, une situation de sortie exception s_{se} est définie telle que $s_{se} = (s_{exc}, *)$ avec $s_{exc} = ?n_i P_i$ où $n_i \in \mathbb{Z} \setminus \{0\}$ et $P_i \in P^{mp}$ avec P^{mp} l'ensemble des places du raffinement de la macroplace.
Le tir d'une transition d'exception provoque la purge du raffinement de la MP :
  — Mise à zéro de l'ordre de tir des transitions internes au raffinement.
  — Mise à zéro des compteurs des transitions temporelles internes au raffinement ainsi que des transitions sortantes à la MP.
  — Mise à zéro des compteurs des transitions temporelles sortantes.
  — La mise à zéro du marquage des places du raffinement.

Les éléments liés à la gestion d'exception, à savoir la macroplace et les transitions d'exception ayant été exposés, considérons leur mise en oeuvre sur la cible.

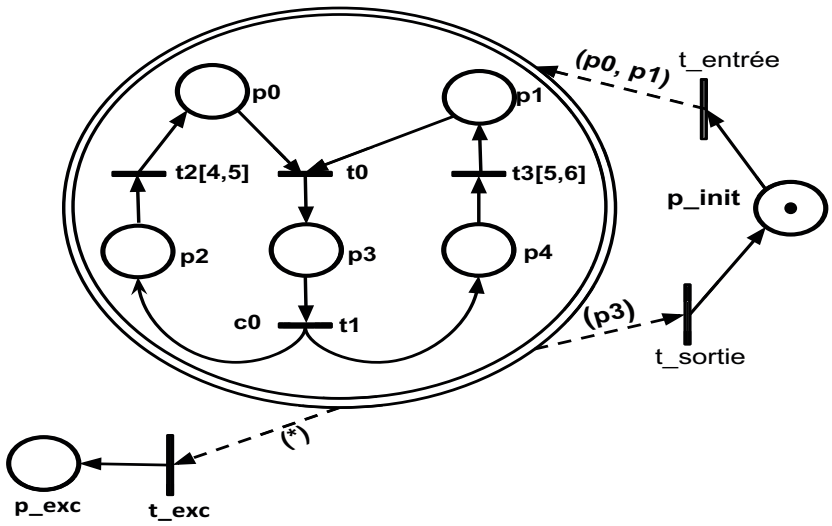


FIGURE 1.12 – Un exemple de macroplace

Génération de code et principe d'exécution sur la cible pour la Macroplace

Dans le même esprit que notre étude de la génération de code précédente, nous devons considérer la manière dont la MP et la gestion d'exception sont mises en oeuvre sur la cible. En effet, la MP n'est qu'une entité de modélisation, au sens description, qui facilite la spécification du modèle. Elle n'est pas réifiée sur la cible. De fait, un modèle équivalent est automatiquement construit à l'aide de places et transitions classiques. Il en est de même pour les arcs d'entrée/sortie de la MP, hors arc décrivant la situation d'exception ($*$). Ils sont remplacés par des arcs (normaux, tests, inhibiteurs) reliant directement les places impliquées dans la situation aux transitions concernées, et vice-versa. L'arc relatif à la situation d'exception ($*$) fait l'objet d'une transformation permettant d'exprimer la sensibilisation de la transition concernée par la détermination de l'activation de la MP (à savoir un OU entre le marquage de toutes les places du raffinement). De plus, en cas de tir de cette transition, toutes les places du raffinement sont purgées via un signal de mise à zéro de l'état des bascules codant le marquage des places concernées.

En termes d'exécution, l'évaluation de la franchissabilité des transitions d'exception est synchrone, mais les effets de leur tir sont réalisés immédiatement, de façon asynchrone (i.e., sans attendre le front montant suivant). En

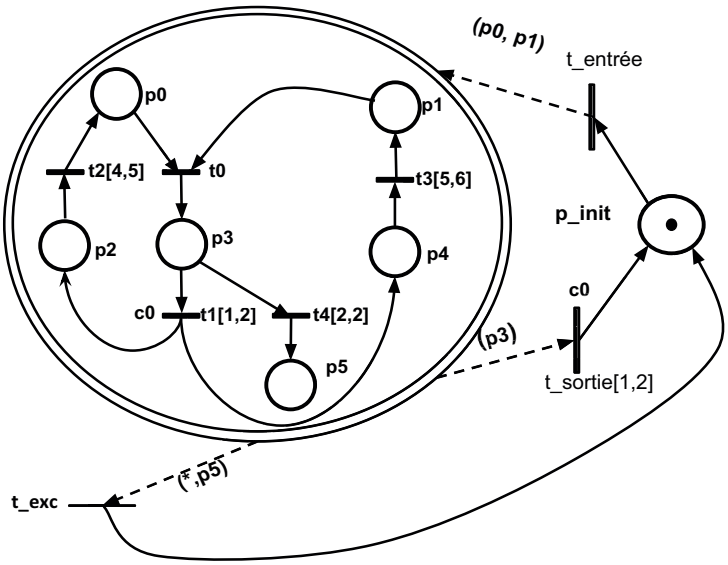


FIGURE 1.13 – Un deuxième exemple de macroplace

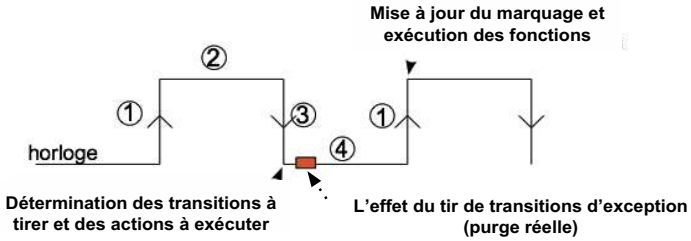


FIGURE 1.14 – Principe d'exécution synchrone (la purge)

effet, les transitions d'exception agissent (au sens font évoluer le modèle) immédiatement après le front descendant et avant le front montant suivant (cf. la figure 1.14), ce qui entraîne que son traitement s'effectue avant celui des transitions devant être tirées sur ce front d'horloge.

Modifions en conséquence le formalisme SITPN pour intégrer la macroplace.

Formalisme SITPN avec Macroplace

Un modèle SITPN avec macroplace est composé de deux parties : le modèle SITPN principal et l'ensemble des macroplaces M. Une place ou une transition ne peut pas appartenir à la fois au modèle SITPN principal et au raffinement

d'une MP, ou à deux raffinements différents.

Formellement, un réseau de Petri généralisé étendu interprété T-temporel synchrone à priorités avec macroplaces, noté SITPN avec MP, est un n-uplet $< P, T, M, Pre, Pre_t, Pre_i, Post, Entry, Exit, m_0, C, F, A, clock, I_s >$, tel que :

- $< P, T, Pre, Pre_t, Pre_i, Post, m_0, C, F, A, clock, I_s >$ est un SITPN.
- M est l'ensemble des macroplaces.
- $mp \in M$ est un SITPN 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 les fonctions de pré-condition sortante, la fonction test sortante, la fonction inhibiteur sortante et la fonction post-condition sortante.
- $Pre_{exc} : T \to M \to \mathbb{B}$, est la fonction d'exception. Elle est égale à 1 quand il y a un arc exception entre la MP et la transition, à 0 sinon.
- $Entry = (Post_M)$ et $exit = (Pre_M, Pre_{Mt}, Pre_{Mt}, Pre_{exc})$ sont les descriptions des situations d'entrée et de sortie.

Le marquage d'une MP mp est défini par $m^{mp} = m_{|P^{mp}}$. La MP mp est dite active ssi $m^{mp} \neq 0$. Les fonctions de marquage m, d'intervalle dynamique I et de réinitialisation $reset_t$ du modèle SITPN complet (le SITPN principal plus l'ensemble de MP) sont appliquées sur les ensembles P^{all} et T^{all}.

Comme le raffinement d'une MP est un modèle SITPN, les définitions des ensembles $enabled(m^{mp})$ et $newEnabled(m^{mp}, Fired)$ sont définis comme pour un modèle SITPN sans macroplace (cf. section 1.1.3). Dans le modèle SITPN principal, par contre, la sensibilisation d'une transition par un arc d'exception doit être prise en compte. Soit $M_{exc}(t)$ l'ensemble des MP relié à une transition t par un arc exception : $\forall mp \in M, mp \in M_{exc}(t) \Leftrightarrow Post_{exc}(t, mp) = 1$. Une transition $t \in T$ est sensibilisée ssi :

$m \geq \Big(Pre(t) + Pre_t(t) + Pre_M(t) + Pre_{Mt}(t)\Big) \wedge \Big(m < Pre_i(t) + Pre_{Mi}(t)\Big) \wedge \Big(\forall mp \in M_{exc}(t), m^{mp} \neq 0\Big)$.

Une transition k est nouvellement sensibilisée par un ensemble $Fired$ de transitions tirées à partir d'un marquage m, noté $k \in newEnabled(m, Fired)$, ssi :

$$\Big(m - \sum_{t\in Fired} Pre(t) - \sum_{t\in Fired} Pre_M(t) + \sum_{t\in Fired} Post(t) + \sum_{t\in Fired} Post_M(t)$$

$$\geq Pre(k) + Pre_t(k) + Pre_M(k) + Pre_{Mt}(k)\Big)$$

$$\wedge\Big(m - \sum_{t\in Fired} Pre(t) - \sum_{t\in Fired} Pre_M(t) + \sum_{t\in Fired} Post(t) + \sum_{t\in Fired} Post_M(t) <$$

$$Pre_i(k) + Pre_{Mi}(k)\Big) \wedge \Big(\forall mp \in M_{exc}(t), m^{mp} - Pre^{mp}(t) + Post^{mp}(t) \neq 0\Big)$$

$$\wedge\Big[\big(k \in Fired\big) \vee \big(m - \sum_{t\in Fired} Pre(t) - \sum_{t\in Fired} Pre_M(t) > Pre_i(k) + Pre_{Mi}(k)\big)$$

$$\vee\big(m - \sum_{t\in Fired} Pre(t) - \sum_{t\in Fired} Pre_M(t) < Pre(k) + Pre_t(k) + Pre_M(k) +$$

$$Pre_{Mt}(k)\big) \vee \big(\exists mp \in M_{exc}(t) \mid m^{mp} - Pre^{mp}(t) = 0\big)\Big]$$

Un état d'un modèle SITPN avec macroplace est défini, comme pour les SITPN sans MP, par $s = (m, cond, ex, I, reset_t)$. Les fonctions de la valeur instantanée d'une condition *val* et de sa valeur *cond* fixée pour le calcul de l'évolution du modèle (donc sa valeur capturée au front descendant), ainsi que la fonction d'exécution des fonctions/actions *ex* sont définies comme pour le modèle SITPN sans MP (cf. section 1.1.3).

Une transition est tirable à partir d'un état s ssi elle est sensibilisée par m, que la valeur des conditions qui lui sont associées autorise son tir et la borne minimale de l'intervalle de tir est atteinte et la borne maximale n'est pas dépassée. L'ensemble des transitions tirables à partir d'un état s est noté $Firable(s)$, et est défini de la même façon qu'un SITPN sans MP (sauf qu'il utilise la définition de *enabled* adaptée à la MP).

Sémantique du modèle SITPN avec Macroplace

La sémantique des SITPN avec MP est extraite de la thèse de Hélène Leroux [60]. Elle est donnée ci-dessous de façon formelle, et expliquée en français à la suite. La sémantique initiale des SITPN avec MP inclue la gestion des priorités.

Nous ne présentons pas ici ces détails, le lecteur intéressé pourra se référer à la référence précitée.

La sémantique du modèle SITPN avec MP est un système de transitions temporisé $(S, s_0, \rightsquigarrow)$, où S est l'ensemble des états, $s_0 = (m_0, 0, 0, I_s, 0)$ est l'état initial et $\rightsquigarrow \subseteq S \times (CLK \times T^n) \times S$ est la relation de changement d'états, notée $s \overset{clk}{\rightsquigarrow} s'$, avec $clk \in CLK$. La relation de changement d'états est définie comme suit :

— $s = (m, cond, ex, I, reset_t) \overset{\downarrow clock}{\rightsquigarrow} s' = (m, cond, ex', I', reset_t)$, ssi l'évènement $\downarrow clock$ se produit et si :

1. $\forall c \in \mathcal{C}, cond'(c) = val(c)$
2. $\forall t \in T^{all}, (t \in newEnabled(m, Fired) \vee reset_t(t) = 1) \Rightarrow I'(t) = I_s(t) - 1$
3. $\forall t \in T^{all}, (t \in enabled(m) \wedge reset_t(t) = 0 \wedge t \notin newEnabled(m, Fired) \wedge I(t) \neq \Phi) \Rightarrow I'(t) = I(t) - 1$
4. $\forall t \in T^{all}, (t \notin newEnabled(m, Fired) \wedge I(t) = \Phi) \Rightarrow I'(t) = I(t)$
5. $\forall a \in \mathcal{A}, (\exists p \in P^{all} \mid A(p, a) = 1 \wedge m(p) \neq 0) \Rightarrow ex'(a) = 1$ sinon $ex'(a) = 0$

   On détermine alors $Fired$, l'ensemble des transitions qui seront tirées depuis l'état s', et $Fired_{exc}$, l'ensemble des transitions exceptions qui seront tirées depuis ce même état :

6. $\forall t \in Firable(s') \Rightarrow t \in Fired$.
7. $\forall t \in Fired \mid \exists mp \in M \mid Pre_{exc}(t, mp) = 1 \Rightarrow t \in Fired_{exc}$.

— $s = (m, cond, ex, I, reset_t) \overset{Fired_{exc}(s)}{\rightsquigarrow} s' = (m', cond, ex, I', reset_t)$, ssi $clock = 0$ et :

1. $\forall t \in Fired_{exc}, \forall mp \in M \mid Pre_{exc}(t, mp) = 1, \forall p \in P^{mp}, m'(p) = 0$
2. $\forall t \in Fired_{exc}, \forall mp \in M \mid Pre_{exc}(t, mp) = 1, \forall k \in T^{mp}, I'(k) = I_s(k)$
3. $\forall t \in Fired_{exc}, \forall k \in T^{all}, \exists mp \in M \mid \Big(Pre_{exc}(t, mp) = 1 \wedge \exists p \in P^{mp} \mid Pre_M(k, p) + Pre_{Mt}(k, p) + Pre_{Mi}(k, p) + Pre_{exc}(k, mp) \geq 1\Big) \Rightarrow I'(k) = I_s(k)$.

4. $\forall t \in Fired_{exc}, \forall mp \in M \mid Pre_{exc}(t, mp) = 1, \forall k \in T^{mp}, k \notin Fired$
5. Pour les places et les transitions non concernées par les propositions ci-dessus, leurs marquages, intervalles de temps et ordres de tir ne sont pas modifiés ($m'(p) = m(p)$, $I'(t) = I(t)$ et $Fired' = Fired$).

— $s = (m, cond, ex, I, reset_t) \overset{\uparrow clock}{\rightsquigarrow} s' = (m', cond, ex', I', reset'_t)$, ssi l'évènement $\uparrow clock$ a lieu et si :

1. $\forall p \in P, m'(p) = m(p) - \sum_{t \in Fired} (Pre(t,p) + Pre_M(t,p)) + \sum_{t \in Fired} (Post(t,p) + Post_M(t,p))$
2. $\forall mp \in M, \forall p \in P^{mp}, m'(p) = m(p) - \sum_{t \in Fired} Pre^{mp}(t,p) + \sum_{t \in Fired} Post^{mp}(t,p)$
3. $\forall t \in T, \exists p \in P \mid \Big(m(p) - \sum_{t_i \in Fired} (Pre(t_i,p) + Pre_M(t_i,p)) < Pre(t,p) + Pre_t(t,p) + Pre_M(t,p) + Pre_{Mt}(t,p)\Big) \Rightarrow reset'_t = 1$, Sinon $reset'_t = 0$
4. $\forall mp \in M, \forall t \in T, \exists p \in P \mid \Big(m(p) - \sum_{t_i \in Fired} Pre^{mp}(t_i,p) < Pre^{mp}(t,p) + Pre^{mp}_t(t,p)\Big) \Rightarrow reset'_t = 1$, sinon $reset'_t = 0$
5. $\forall f \in \mathcal{F}, \exists t \in Fired \mid F(t,f) = 1 \Rightarrow ex'(f) = 1$, sinon $ex'(f) = 0$
6. $\forall t \in T^{all}, (t \notin Fired \wedge \uparrow I'(t) = 0) \Rightarrow I'(t) = \Phi$ sinon $I'(t) = I(t)$
7. $Fired' = Fired$

La sémantique du modèle SITPN avec MP fonctionne comme suit :

— Sur le front descendant $\downarrow clock$:

1. Toutes les valeurs des conditions sont mises à jour.
2. L'intervalle de temps d'une transition est réinitialisé s'il doit l'être ($reset_t(t) = 1$) ou si la transition est nouvellement sensibilisée. L'intervalle est réinitialisé à $I_s(t) - 1$, car le tir des transitions dure $1ut$.
3. Mise à jour des intervalles de temps des transitions sensibilisées mais qui ne sont ni nouvellement sensibilisées, ni bloquées et dont les intervalles ne doivent pas être réinitialisés.

4. Si une transition était bloquée et n'est pas nouvellement sensibilisée alors elle reste bloquée.
5. Mise à jour des valeurs de la fonction ex pour les actions continues.

6. Après toutes ces mises à jours, on peut déterminer $Fired$, l'ensemble des transitions qui seront tirées à partir de l'état s'.
7. On détermine ensuite $Fired_{exc}$, qui est l'ensemble des transitions devant être tirées (donc $\in Fired$) mais qui sont des transitions exception.

— Si le signal de l'horloge est entre le front descendant et le front montant ($clock = 0$) et si au moins une transition d'exception appartient aux transitions sélectionnées pour le tir sur le front descendant précédent, alors l'effet de son tir est immédiat (i.e. l'évaluation de leur franchissabilité est synchrone, mais les effets de leur tir sont réalisés immédiatement de façon asynchrone, sans attendre le front montant suivant).
Pour chaque MP liée à au moins une transition d'exception tirée :

1. Annulation du marquage des places de son raffinement.
2. Réinitialisation des intervalles de temps des transitions de son raffinement.
3. Réinitialisation des intervalles de temps de ses transitions sortantes.
4. Annulation des ordres de tir de toutes les transitions de son raffinement.
5. Comme dit précédemment, pour les places et les transitions non concernées par les situations ci-dessus, leurs marquages, intervalles de temps et ordres de tir ne sont pas modifiés.

— Sur le front montant $\uparrow clock$:

1. Mise à jour du marquage du modèle SITPN principal suivant les transitions tirées ($Fired$).
2. Mise à jour du marquage des raffinements des MP suivant les transitions tirées.

3. Réinitialisation des intervalles de temps pour les transitions du modèle SITPN principal qui sont désensibilisées par le marquage intermédiaire lors du tir des transitions de $Fired$.
4. Réinitialisation des intervalles de temps pour les transitions des raffinements des MP qui sont désensibilisées par le marquage intermédiaire lors du tir des transitions de $Fired$.
5. Mise à jour des valeurs de la fonction ex pour les actions impulsives.
6. Toutes les transitions non tirées, et qui ont atteint la borne maximale de leur intervalle de temps, sont bloquées. Les autres intervalles de temps ne sont pas modifiés.
7. L'ensemble des transitions tirées n'est pas modifié.

Nous avons expliqué le formalise utilisé dans notre contexte (SITPN avec MP) et le principe de sa mise en oeuvre sur la cible. Nous allons maintenant expliciter la problématique abordée dans ces travaux de thèse.

1.2 Problématique

Nous avons vu que notre méthodologie de conception, rappelée figure 1.15, va de la modélisation à la mise en oeuvre sur composants électroniques (FPGA, ASIC). L'architecture du système numérique complexe est décrite à l'aide de composants dont le comportement est modélisé selon le formalisme ITPN, avec la possibilité d'utiliser des macroplaces pour la gestion d'exception. Ce modèle initial est ensuite mis à plat puis transformé en SITPN pour être traduit en VHDL et implémenté sur un FPGA. Il est exécuté sur la cible de façon synchrone : l'évolution du modèle SITPN est principalement dirigée par un signal d'horloge avec, pour le cas très particulier des transitions d'exception, une évolution d'état asynchrone.

Les systèmes numériques considérés étant critiques et temps réel, notre méthodologie comprend également une étape d'analyse formelle afin d'obtenir des résultats de validation fiables, au plus tôt dans le processus de conception. Ainsi, initialement décrite dans [95], la méthodologie a été complétée par un processus d'analyse asynchrone [61, 62, 60] permettant d'exploiter les possibilités des méthodes et outils d'analyse existants des RdP temporels classiques

[13]. L'ensemble de la méthodologie existante est résumée en noir sur la Figure 1.15.

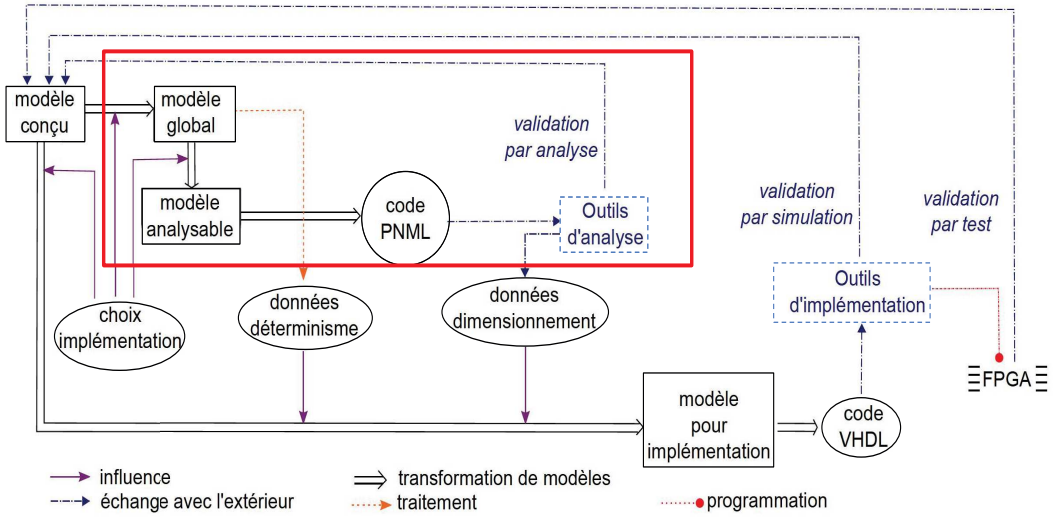


FIGURE 1.15 – Positionnement de la contribution dans la méthodologie HILECOP

Cependant, le choix d'implémentation induit des contraintes très spécifiques liées à l'exécution, en particulier : la simultanéité de tir des transitions (vrai parallélisme) et l'évolution discrète et synchrone du temps, sans omettre la prise en compte effective des blocages et la gestion des exceptions. De fait, il est nécessaire de faire un état de l'art des approches existantes dans le cadre de l'analyse fine de nos modèles, de façon efficace et réaliste, en considérant les contraintes d'exécution (appelées propriétés non-fonctionnelles). L'objectif est de trouver une méthode d'analyse permettant d'énumérer aussi bien que possible l'espace d'états et l'évolution réelle de notre système. La contribution à l'amélioration de la méthodologie est indiquée en rouge sur la figure 1.15.

Nous allons donc nous intéresser aux techniques d'analyse des réseaux de Petri.

1.3 Analyse des RdP

L'étape de validation formelle à partir d'un modèle du système consiste principalement à vérifier des propriétés sur le modèle et au cours de son exé-

cution, ou au contraire à vérifier que certaines situations ne peuvent jamais se produire.

Les propriétés "qualitatives" classiques d'un RdP, comportementales (dépendantes du marquage) et structurelles (indépendantes du marquage), sont [48] :

La vivacité : Un RdP est vivant ssi toutes ses transitions sont vivantes, i.e. chaque transition t du RdP est toujours franchissable à partir de n'importe quel marquage m accessible de l'état initial. C'est-à-dire qu'il existe une séquence de franchissements qui, partant de m, permet de franchir t. Si une transition est franchissable un nombre limité de fois (au moins une fois) la transition est dite quasi-vivante. C'est le cas par exemple d'une transition représentant l'initialisation d'un système, initialisation qui n'est faite qu'une seule fois au lancement. Le cycle d'évolution ne repassera plus par cette transition, sans pour autant induire un blocage du graphe.

Le blocage (ou l'absence de blocage) : Un RdP présente un blocage à partir du moment où il est possible d'atteindre un état à partir duquel plus aucune évolution n'est possible (i.e. aucune transition franchissable). Le blocage est alors qualifié de mortel si aucune transition n'est franchissable à partir de ce marquage. Un blocage peut aussi être partiel, dans le sens où seulement une partie du modèle n'évolue plus. Le réseau n'est alors pas bloquant (puisqu'il existe des transitions franchissables) mais pas vivant non plus. Le blocage est bien évidemment une propriété dont la vérification est essentielle ; ceci étant, il est possible que ce soit une situation désirée dans des cas très particuliers comme par exemple le fait que le système se mette, après une exception particulière, dans un état sûr bloquant dans l'attente d'une ré-initialisation. Quoiqu'il en soit, tout blocage potentiel, partiel ou mortel, doit être examiné avec attention.

La finitude (ou bornitude) : Un RdP est dit borné si toutes ses places sont bornées. Une place P est dite k-bornée pour un marquage m, si le nombre de jetons dans cette place ne dépasse jamais k ($k \in \mathbb{N}$). Un RdP est k-borné si toutes ses places sont k-bornées, pour tout marquage atteignable à partir de son marquage initial. L'étude de la finitude, au delà de simplement borné / non-borné (même si c'est important dans

la mesure où aucun état du système, a fortiori embarqué, ne saurait être infini), donne des informations importantes à travers les bornes du marquage des places. Dans notre contexte par exemple, la valeur des bornes est nécessaire pour assurer un dimensionnement correct et optimal des compteurs (bascules) sur la cible matérielle, dont celui servant à stocker le marquage d'une place.

La persistance : Un RdP est dit persistant si, quelles que soient les transitions franchissables à tout instant, leurs franchissements sont sans conflit.

La réversibilité : Un RdP est dit réversible (propre) s'il est toujours possible de revenir au marquage initial, à partir de tout marquage atteignable. Comme nous l'avons mentionné pour la vivacité, la réversibilité n'est pas forcément une propriété désirée (cas de la séquence d'initialisation qui ne doit pas être répétable).

L'atteignabilité (ou accessibilité) : Il s'agit d'une propriété importante considérant la dynamique du modèle. Le franchissement de transitions modifiant la distribution des jetons dans le modèle, une série de tirs conduit à une série de marquages. Un marquage est dès lors dit atteignable s'il existe une séquence de franchissements y conduisant.

Les invariants : Il s'agit de relations indépendantes inhérentes à la structure du modèle mais dont la "validation" nécessite la prise en compte du marquage initial. Par exemple, les invariants de transitions (issus des composantes répétitives stationnaires) permettent de mettre en évidence des séquences de tirs de transitions répétitives ; ces séquences ne sont valides que si le marquage initial les rend effectives (un marquage initial nul rendrait impossible tout franchissement). Le modèle est dit **consistent** s'il existe au moins une telle séquence impliquant un nombre quelconque de fois toutes les transitions du modèle. Les invariants de transitions permettent par exemple de mettre en évidence des processus cycliques. Les invariants de places (issus des composantes conservatives) donnent quant à eux des informations sur des relations de conservation (de jetons) entre les états (marquages) ; pareillement, ces relations ne sont valides que si le marquage initial les rend effectives. Les invariants de places permettent par exemple de mettre en évidence

le respect de contraintes de capacité d'une ressource ou l'exclusion mutuelle. Ces invariants de place et de transitions permettent également parfois de conclure quant à la finitude (toutes places du modèle sont impliquées dans au moins une relation de conservation) et la vivacité (toute transition du modèle est impliquée dans au moins une séquence répétitive), en prenant en compte le marquage initial.

Pour étudier les propriétés d'un RdP classique, deux types principaux d'analyse de propriétés sont possibles sur un modèle RdP [28] :

— L'analyse comportementale (par énumération) : elle permet d'étudier l'évolution du modèle à travers l'évolution de ses marquages accessibles. Le comportement du RdP est ainsi représenté par un graphe. Les sommets de ce graphe représentent les marquages accessibles (les états du système), les arcs représentent les changements d'états possibles lors des franchissements des transitions. L'analyse de propriétés s'effectue sur le graphe des marquages accessibles. Elle est indépendante du modèle formel à partir duquel le graphe est généré. L'inconvénient majeur de l'analyse énumérative est sa limite en termes de complexité du modèle, i.e. le risque d'explosion combinatoire dans la génération du graphe de marquages (ou de classes).

— L'analyse structurelle : C'est l'analyse de propriétés à partir de la structure (topologie) du RdP, où l'état initial est considéré comme un paramètre. Exploitant l'algèbre linéaire [48], l'analyse est basée sur la matrice d'incidence et l'équation fondamentale (et donc pas énumérative). D'autres approches relevant du « raisonnement logique », s'appuyant directement sur la structure du modèle et répondant plus à une approche approche orientée événements (liens de causalité), permettent également d'examiner certaines questions d'accessibilité ; elles ont recours à logique linéaire [34]. Ces différentes méthodes permettent de vérifier (parfois indirectement, et après considération du marquage pour filtrer des états inatteignables ou des séquences de franchissement inexistantes) des propriétés comme la vivacité, la finitude, la réversibilité, la persistance et les invariants.

— La transformation de modèles à des fins d'analyse : les réseaux de Petri temporels sont un formalisme adapté à la modélisation des sys-

tèmes temps réel distribués, qui permet une représentation simple et efficace de la synchronisation et du parallélisme. De nombreuses approches permettent de construire son espace d'états (graphe de classes d'états, graphes de région et graphe de zone), avec différentes abstractions permettant de conserver les propriétés de marquages, les propriétés logiques LTL et/ou CTL [12, 10, 79, 112, 14, 43] . Cependant, la vérification de propriétés exprimant une contrainte de temps quantitative, typiquement exprimées en logiques temporelle temporisée de type TCTL, est difficile pour les TPN. Des approches ont été développées pour résoudre ce problème [105, 49, 18, 39], mais souvent en étant peu efficace, en modifiant le modèle initial et en ne permettant la vérification que d'un sous-ensemble de propriétés TCTL. Une autre solution est alors de traduire le modèle TPN initial dans un autre formalisme, tel que les réseaux d'automates temporisés (NTA, Networks of Timed Automata) pour lesquels la vérification de propriétés TCTL a été prouvée décidable [1], et qui possèdent des outils de modélisation et validation performants (par exemple l'outil UPPAAL [59]). Ces deux formalismes sont proches en termes d'expressivité et d'équivalence [8], et plusieurs méthodes de traduction des TPN en (N)TA ont été proposées dans la littérature [25, 31, 5]. Cependant, la transformation de modèles dans un autre formalisme est une étape intermédiaire qui complexifie le processus d'analyse, peut limiter les résultats d'analyse (en fonctions des hypothèses et des abstractions effectuées) et qui ne se révèle nécessaire que dans certains cas extrêmes (vérification de certaines propriétés TCTL). Nous ne partirons donc pas sur ce type de méthode.

— Précisons par ailleurs qu'il existe des méthodes de réduction qui permettent de simplifier le modèle (PN) tout en préservant la validité des résultats d'analyse [20, 94]. Bien sûr le modèle n'est réduit qu'à des fins d'analyse (i.e. le système mis en oeuvre est basé sur le modèle initial).

La vérification de modèles (*Model Checking*) [30] est une technique de vérification de propriétés plus poussée par rapport aux méthodes classiques (e.g. analyse structurelle des RdPs [48, 34]). Elle se fait à travers le parcours de l'espace d'état (graphe des marquages, graphe de recouvrement), si ce dernier est fini. La limite principale de ces méthodes est également l'explosion

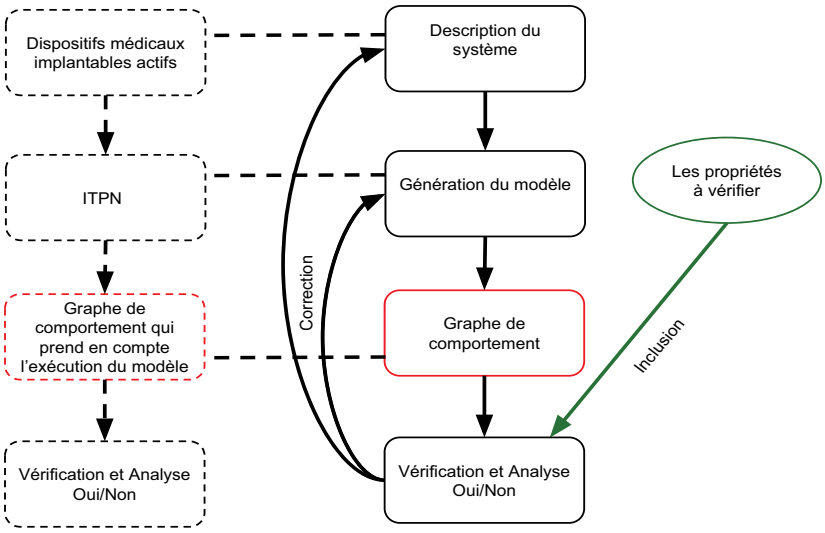


FIGURE 1.16 – Schéma général de la vérification formelle

combinatoire de l'espace d'états, induit par une énumération exhaustive de ce dernier.

Les propriétés sont formalisées en expressions logiques telles que *Linear Time Logic* LTL et *Computation Tree Logic* CTL [97, 30, 89]. La dernière étape est la vérification des propriétés spécifiées, à l'aide d'un *model checker* [72, 30]. Dans le cas d'un résultat négatif (i.e. propriété non vérifiée), une intervention humaine (Ingénieur/Testeur) est nécessaire pour analyser la trace de l'erreur. Une erreur peut provenir d'une modélisation incorrecte du système, comme elle peut provenir d'une spécification erronée de la propriété. Pour les propriétés qui ne peuvent pas être directement extraites du graphe de comportement, il est parfois possible de les modéliser et de les ajouter au modèle à analyser afin de les vérifier sur le comportement global du modèle. Par exemple, dans le cas de l'étude du partage d'un médium par plusieurs entités communicantes, il est possible de modéliser le médium par un état ; si cet état comprend plusieurs jetons alors l'accès unique n'est pas garanti car la présence de plusieurs jetons témoigne que plusieurs entités émettent potentiellement sur le médium en même temps.

Les différentes étapes de la vérification du modèle par *Model checking* sont présentées dans la figure 1.16.

La vérification de propriétés est donc en grande partie basée sur le graphe

de comportement. Nous allons donc présenter les approches existantes de génération de graphe de comportement, en considérant plus particulièrement le temps et l'interprétation associés au modèle.

1.3.1 Les graphes de comportement de RdP autonomes

RdP classique : Le RdP classique [80] est défini dans la section 1.1.2. Le comportement d'un RdP classique est défini par l'ensemble des marquages accessibles depuis son état initial. Ce graphe est appelé graphe des marquages [77]. Il est composé de noeuds et arcs. Chaque marquage accessible est représenté par un noeud et le tir de transition depuis un marquage m (noeud n_1) qui conduit vers un marquage m' (noeud n_2) est représenté par un arc orienté de n_1 vers n_2, mentionnant la transition tirée. Un exemple de RdP classique et son graphe de marquage sont présentés sur la figure 1.17.

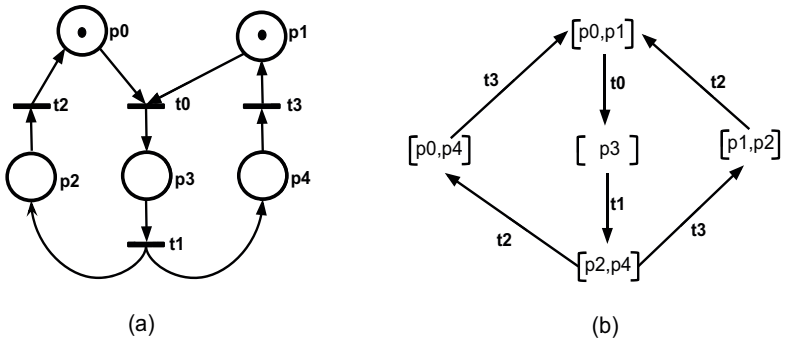


FIGURE 1.17 – Exemple de RdP et son graphe de marquage

Les extensions des RdP autonomes comme les RdP généralisés [65] et les RdP étendus [99] (cf. définitions section 1.1.2) ne changent pas la méthode de construction du graphe de comportement même si elles impactent le tir des transitions.

1.3.2 Les graphes de comportement de RdP non-autonomes

Synchronisation et Interprétation

RdP synchronisé - Un RdP synchronisé [74, 33] est un RdP classique où le tir de transitions est conditionné par l'occurrence d'un événement externe. Un RdP synchronisé est une paire (N, Syn), où N est un RdP classique et Syn est la fonction qui associe un évènement e à une (des) transition(s).

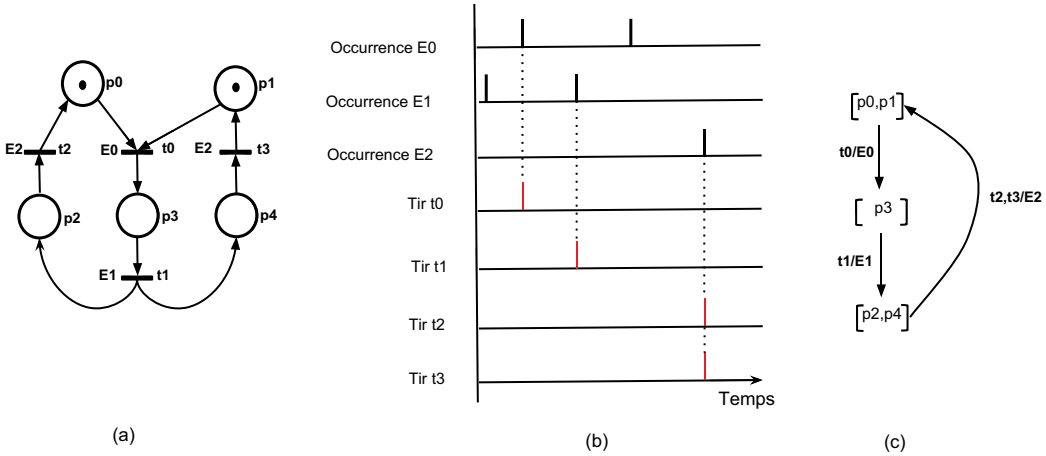


FIGURE 1.18 – Le concept du RdP synchronisé et son graphe de comportement

Les noeuds du graphe de comportement d'un RdP synchronisé représentent les marquages accessibles. Les arcs représentent la relation d'évolution du marquage. Les arcs sont étiquetés par l'ensemble des transitions tirées et l'ensemble des événements qui ont déclenché les tirs éventuellement simultanés. Les figures 1.18(a) et (b) illustrent le principe d'un RdP synchronisé. La figure 1.18(c) donne le graphe de comportement du RdP de la figure 1.18(a).

RdP interprétés - Les RdP interprétés, comme les Control Interpreted Petri Nets CIPN [46] et les Signal Interpreted Petri Nets - SIPN [40] sont basés sur les RdP synchronisés [86]. L'interprétation associée est souvent une représentation des entrées/sorties du système, avec les entrées qui conditionnent le tir des transitions et les sorties qui sont activées lors du marquage des places auxquelles elles sont associées.

Un graphe de comportement d'un RdP interprété [33] est représenté par l'ensemble de ses marquages accessibles depuis l'état initial. Les noeuds contiennent les marquages accessibles et les arcs représentent la relation d'évolution du marquage suite au tir d'une (ou plusieurs) transition(s) éventuellement simultanés. Les figures 1.19 (a) et (c) illustrent le principe de RdP interprété, dont la figure 1.19 (c) donne le graphe de comportement.

Une autre approche pour générer le graphe de comportement d'un RdP interprété a été proposée dans [58]. Elle consiste à supprimer toutes les informations de l'interprétation et ne garder que la structure du RdP et le marquage initial. À partir du modèle obtenu (appelé RdP sous-jacent) le graphe de mar-

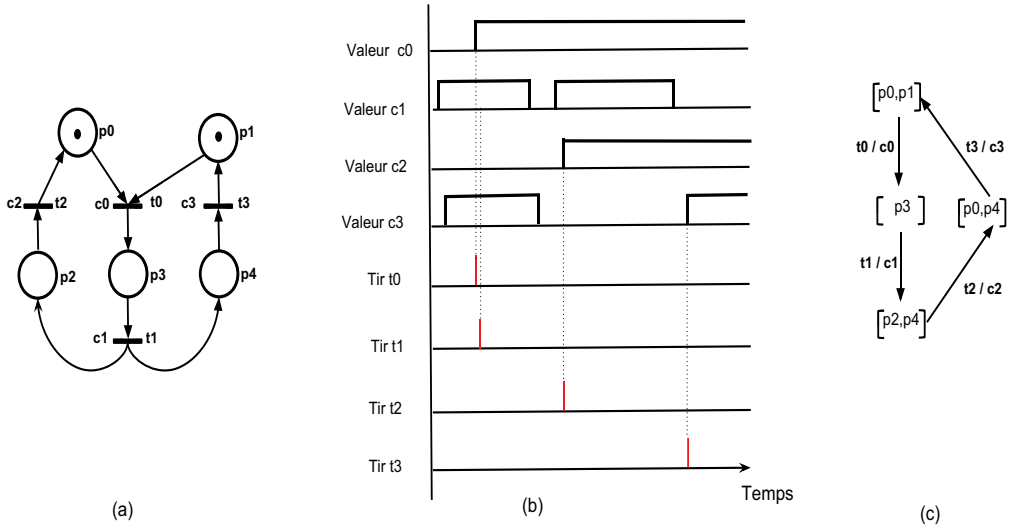


FIGURE 1.19 – Le concept du RdP interprété et son graphe de comportement

quage est généré afin d'être analysé. Cette approche est basée sur l'idée que le graphe de marquage du RdP sous-jacent est un sur-ensemble du RdP interprété. L'ajout de l'interprétation réduit toujours les possibilités d'évolution du marquage et jamais l'inverse. Nous retombons ainsi dans la problématique abordée section 1.2 : l'analyse d'un sur-ensemble des comportements conduit, en plus de l'augmentation de la taille de l'espace d'états, à une limitation des résultats de l'analyse liée à l'écart entre le comportement réel et le comportement analysé.

Pour vérifier la finitude, une méthode a été développée afin de l'appliquer directement sur le RdP interprété. Les auteurs proposent un algorithme qui consiste à transformer le RdP interprété en un automate séquentiel simple [58]. La preuve d'équivalence et la méthode de transformation sont présentées dans [57]. Après la transformation, la deuxième étape consiste à supprimer les informations de l'interprétation et à générer le graphe réduit. Pour cela, une adaptation de la méthode *Stubborn*[6] [101] a été proposée dans [57]. Cette méthode est limitée à la vérification d'états de blocage pour des RdP interprétés saufs (un jeton par place).

6. Stubborn est une approche de réduction de graphe de comportement, proposée initialement pour les RdP classiques.

Prise en compte du temps

Le temps est pris en compte dans deux classes de RdP : les RdP temporisés et les RdP temporels.

RdP temporisés - Nous nous intéressons aux RdP T-temporisés avec une sémantique de temps discret telle que définie dans [33] parce qu'ils sont plus proches de notre modèle initial en termes d'évolution du temps et d'association du temps aux transitions. Un RdP T-temporisé est une paire $< N, tempo >$, où N est un RdP classique et $tempo : T \rightarrow \mathbb{N}$ est la fonction qui associe une durée constante avec chaque transition t, $tempo(t) = d_t$. Une transition sensibilisée t peut être tirée si seulement si le temps écoulé θ (en temps discret donc $\theta \in \mathbb{N}$) depuis sa sensibilisation est égal à d_t. Pour une transition sensibilisée t et un temps écoulé depuis sa sensibilisation égal à θ (avec $\theta < d_t$), le temps $R = d_t - \theta$ est le temps résiduel de tir.

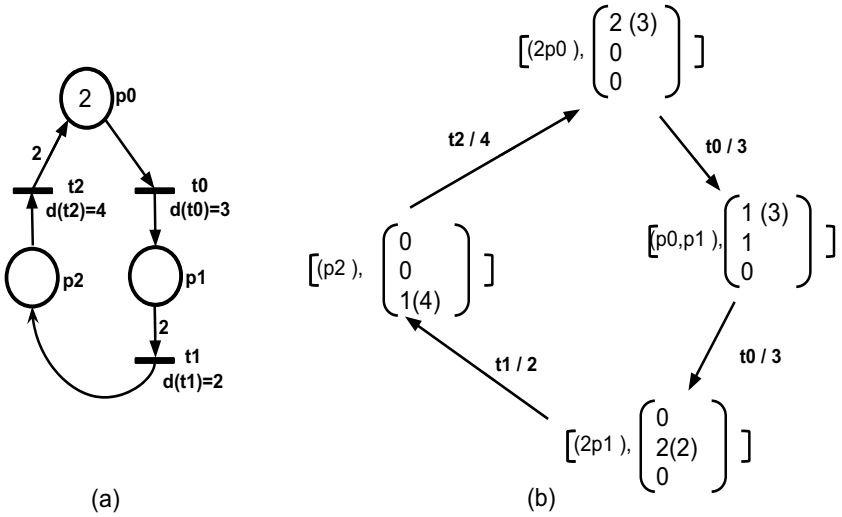


FIGURE 1.20 – Exemple de RdP t-temporisé et son graphe de comportement

Le graphe de comportement d'un RdP T-temporisé [33] est également un ensemble de sommets et d'arcs orientés. Mais les sommets sont des états représentant le marquage, le vecteur de transitions sensibilisées et, pour chaque transition sensibilisée ou k-sensibilisée, la liste du (des) temps résiduel(s) de tir. Les arcs représentent la relation d'évolution des états : évolution du marquage par rapport aux tirs des transitions, potentiellement simultané, ainsi qu'au temps (relatif) écoulé ; le tir d'une transition t à partir d'un marquage

m après un temps d_t (conduisant à un marquage m_2) est exprimé par un arc étiqueté par t/d_t. Les figures 1.20(a) et (b) illustrent respectivement un RdP T-temporisé et son graphe de comportement.

RdP Temporels - Un RdP temporel [84, 10, 109, 41] est un RdP classique avec association d'un intervalle de temps à chaque transition ; une transition sensibilisée peut être tirée durant un intervalle de temps spécifié. Cet intervalle associé aux transitions est défini par deux valeurs $[v_{min}, v_{max}]$. Pour une transition t sensibilisée (temps relatif exprimé par la variable θ), v_{min} et v_{max} représentent la date de tir au plus tôt et la date de tir au plus tard de t, respectivement. La transition t ne peut pas être tirée tant que θ n'est pas compris entre v_{min} et v_{max}, et ne pourra plus être tirée après v_{max}. Le tir est considéré instantané et donc ne prend pas de temps. Deux extensions principales de RdP temporels sont proposées dans la littérature : soit avec une évolution de temps continue ($\theta \in \mathbb{R}^+$) [73], soit avec une évolution en temps discret ($\theta \in \mathbb{N}^+$) [84].

Plusieurs approches ont été proposées pour générer le graphe de comportement pour les RdP temporels avec un temps continu [12, 14, 19], [43], et pour les RdP avec un temps discret [82, 83, 88, 98].

Les deux approches classiques sont :

1. L'approche par Graphe de Classes d'États [12] (*State Class Graph*, noté SCG) est adaptée aux RdP temporels à temps continu. Dans la sémantique des RdP temporels à temps continu, un état peut avoir une infinité de successeurs par le tir d'une transition tirable à tous les instants d'un interval continu. La méthode SCG regroupe donc tous ces successeurs en une seule classe d'équivalence. Une classe c est définie par un couple (m, D), avec m le marquage et D le domaine de tir des transitions, défini par l'ensemble des contraintes temporelles de tir des transitions sensibilisées par ce marquage. Chaque noeud de SCG représente une classe et chaque arc représente un tir d'une transition (Figure 1.20).

2. L'approche *Integer-State Graph*, notée ISG, a été proposée par Louchka Popova-Zeugmann [82] pour les RdP temporels asynchrones à temps discret. Un état est un couple $z = (m, h)$, avec m le marquage et h le

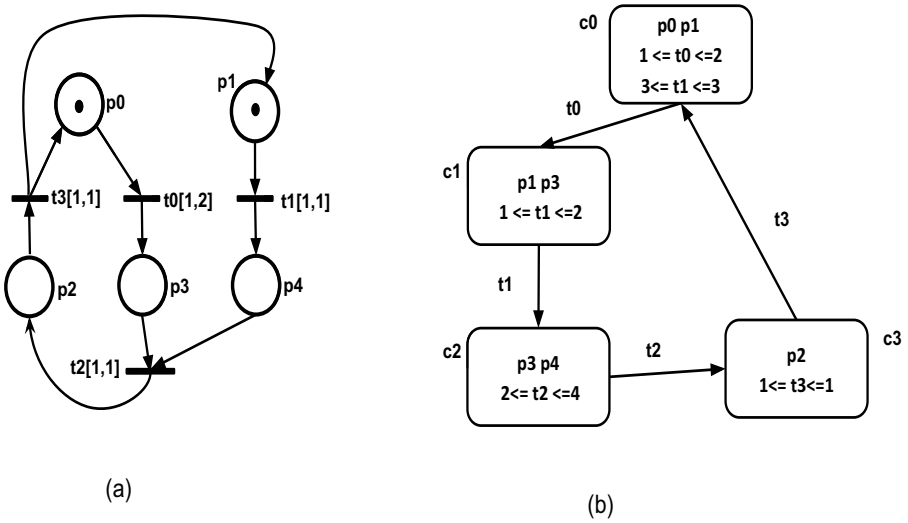


FIGURE 1.21 – Exemple de RdP temporel à temps continu et son SCG

vecteur des horloges (relatives) des transitions. L'horloge d'une transition sensibilisée est définie par le temps écoulé depuis son instant de sensibilisation. Si la transition est désensibilisée, son horloge est désignée par #. Cette approche se limite aux intervalles de temps non infinis. Cette limite a été abordée dans [83], où les auteurs proposent d'utiliser la notion *"integer-pseudo-state"* ; dans le cas où la borne supérieure de l'intervalle de temps est l'infini, alors l'évolution de l'horloge s'arrête à la borne inférieure de l'intervalle et une boucle sur le même état est ajoutée afin de représenter l'évolution infinie/indéterminée du temps. La figure 1.22 représente un RdP temporel et son ISG.

Précisons enfin que plusieurs travaux (articles) couplent les notions de temps et d'interprétation pour différentes classes de RdP, comme par exemple [93, 6, 87, 92]. Il ne s'agit pas de l'interprétation au sens d'associer au RdP des conditions, des actions et/ou des fonctions, mais plutôt d'étudier la signification du temps porté par le modèle (*time-interpreted*). Par exemple, si le temps est associé aux jetons, alors il sera interprété comme l' "âge des jetons". Selon l'interprétation donnée au temps sur le modèle, ce dernier aura un pouvoir de modélisation spécifique. Ce n'est pas l'objet de nos travaux.

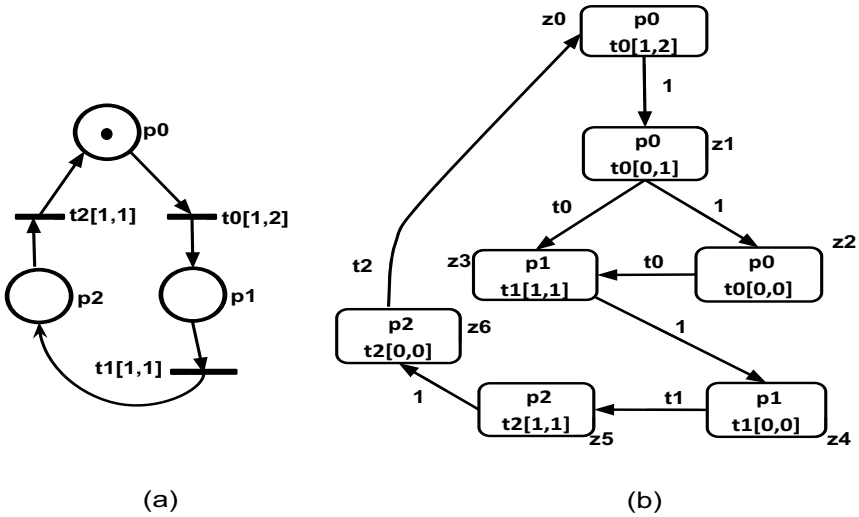


FIGURE 1.22 – Exemple de RdP temporel à temps discret et son ISG

1.4 Bilan

Après avoir explicité notre contexte et donc les origines de notre formalisme et de notre sémantique, nous avons pu constater que les approches d'analyse existantes ne permettent pas d'atteindre notre objectif : en effet, la synthèse donnée dans le tableau ci-dessous met en évidence qu'aucune des méthodes existantes ne couvre toutes les caractéristiques de notre modèle (rappelées dans la suite de cette section). La grille de lecture du tableau 1.1 est la suivante : O (Oui, caractéristique prise en compte), N (Non, prise en compte impossible de la caractéristique) et T (la caractéristique pourrait être prise en compte sous réserve de transformation de modèles).

Nous comparerons plus en détails ces approches et notre proposition dans les chapitres 3 et 4, mais à ce stade nous avons pu constater que l'analyse asynchrone du comportement d'un modèle exécuté en synchrone introduit potentiellement un écart entre le comportement réel et le comportement analysé. Par exemple, lors de l'analyse asynchrone, le tir des transitions parallèles (i.e. tirables au même instant) sont nécessairement représentées par un entrelacement ; qu'en serait-il dans le cas d'une analyse synchrone ? Le tir simultané, i.e. en une seule étape (unité de temps logique), ne conduirait-il pas à la disparition de certains états intermédiaires ? Le comportement réel, i.e. résultant d'une exécution synchrone, est certes inclus dans l'espace d'états des comportements asynchrones [60], néanmoins ceci réduit significativement l'efficacité

TABLE 1.1 – Comparaison synthétique des méthodes existantes

Formalisme et graphe de comportement	Caractéristiques					
	Temps qualitatif	Temps discret	Tirs multiples	Tir au plus tôt	Sémantique de blocage	Macro Place
Classique	N	N	N	T	N	-
Synchronisé	-	-	O	O	N	-
Interprété	N	N	O	O	N	-
t-Temporisé	O	O	O	O	N	-
t-Temporel, (SCG)	O	N	N	T	T	T
t-Temporel, (ISG)	N	O	N	T	T	T

de l'analyse et la confiance dans les résultats, ces derniers comprenant sans distinction des états non atteignables sur la cible. De fait, une propriété vérifiée sur le modèle analysable ne l'est pas forcément dans le comportement réel, et réciproquement une propriété non vérifiée sur le modèle analysé peut provenir d'un état non atteignable (non réel).

La technique d'analyse utilisée jusque là dans la méthodologie HILECOP était, après transformation du modèle ITPN vers un RdP Généralisé étendu temporel, basée sur l'approche asynchrone du SCG [13] [41]. Nous venons d'en souligner les limitations, donc cette thèse est centrée sur l'analyse et la validation du modèle SITPN que nous avons exposé, dont nous rappelons les particularités :

1. Relevant du modèle ITPN :
   — Le modèle ITPN combine le temps (quantitatif, au sens intervalle de tir) et l'interprétation, ce qui n'a jamais été traité à notre connaissance, dans la littérature. Représenter le graphe de comportement d'un modèle ITPN seulement avec l'ensemble des marquages accessibles est impossible. Plusieurs approches ont été proposées pour gérer l'aspect temporel comme le SCG et l'ISG, et pour gérer l'interprétation comme dans [58], mais aucune d'entre-elles ne gère conjointement les deux aspects.

2. Relevant de l'implémentation et de la sémantique d'exécution :
   — Le modèle ITPN est implémenté sur FPGA de façon synchrone, il devient donc un SITPN. Cela implique une évolution discrète du temps et surtout un vrai parallélisme (évolution synchrone). Cela doit bien évidemment être explicite sur le graphe de comportement. Par exemple, toutes les transitions [7] non temporelles (non associées à un intervalle de temps) sont tirées immédiatement ce qui en synchrone signifie qu'elles sont tirées après une unité de temps logique (équivalente au cycle d'horloge).
   — Notre sémantique "impérative" est différente des sémantiques utilisées dans les méthodes d'analyse que nous avons évoquées. Donc il faut que le graphe de comportement du modèle SITPN montre explicitement le tir au plus tôt de transitions sans condition et le blocage potentiel de transitions.
3. Relevant de la gestion des exceptions :
   — Comme nous l'avons expliqué précédemment, le tir d'une transition d'exception induit une purge de la macroplace. Or, cette purge est exécutée immédiatement sur la cible, i.e. de façon asynchrone. L'exécution asynchrone de ce type de transitions doit être prise en considération et logiquement reflétée sur le graphe.

Ce constat nous a conduit à étudier et à proposer une nouvelle approche d'analyse, au sens d'une nouvelle approche de génération du graphe de comportement qui soit conforme aux caractéristiques de notre modèle et de sa technique d'implémentation.

Nous allons donc décrire en détails notre proposition en commençant, dès le prochain chapitre, par la prise en compte de l'exécution synchrone sur la cible. Nous nous intéresserons de fait à la transformation du modèle SITPN vers un modèle analysable intégrant cette caractéristique. Une fois cette transformation présentée, nous montrerons dans les chapitres suivants la méthode de génération de graphe de comportement pour le modèle obtenu.

7. Les transitions d'exception sont traitées séparément.

Chapitre 2

Transformation du modèle à analyser pour refléter l'exécution synchrone

Comme nous l'avons expliqué dans le chapitre précédent, la conception du système est décrite à l'aide de réseaux de Petri temporels interprétés (ITPN). Ce modèle est implémenté et exécuté sur FPGA de manière synchrone ; dès lors nous avons défini formellement ce modèle à travers le formalisme SITPN (ITPN Synchrones). Cependant, toutes les particularités relatives au modèle, à sa sémantique et à son exécution synchrone sur la cible constituent un défi pour l'analyse. A ce jour, il n'existe pas de méthode d'analyse qui permette de considérer conjointement toutes ces particularités. Dans ce chapitre nous allons donc proposer une transformation du modèle initial ITPN vers un réseau de Petri temporel synchrone, noté STPN *Synchronous Time Petri Net*, en intégrant ces particularités dont l'influence de l'interprétation (la relation entre le processus et les formalismes est schématiquement représentée sur la figure 2.1).

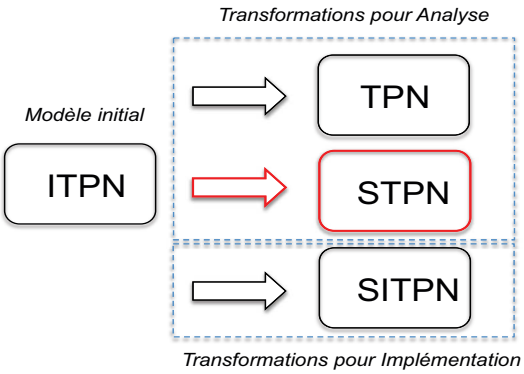


FIGURE 2.1 – Processus et formalismes

Concernant l'interprétation en particulier (i.e., son impact potentiel), elle sera prise en considération à travers les intervalles temporels, ce qui nous conduira à définir le formalisme puis la sémantique associée à ce nouveau modèle STPN. L'interprétation n'est pas le seul aspect traité ; nous allons donc expliquer tous les aspects considérés dans cette transformation qui, fondamentalement, va s'articuler autour de la dimension temporelle du modèle (ie., les intervalles temporels).

2.1 Transformation du modèle initial vers un modèle analysable

La transformation du modèle doit considérer les propriétés non fonctionnelles, liées à la stratégie d'implémentation, et les contraintes imposées par la cible matérielle. Le principe d'exécution de notre modèle est donné section 1.1.4. Si nous désirons que le comportement du modèle analysable (STPN) soit le plus proche possible du comportement réel, la transformation vers le modèle analysable doit alors considérer très finement ces impacts. En effet, l'interprétation et l'exécution synchrone ne peuvent pas être analysées directement, cependant nous allons voir qu'il est possible d'exprimer leur impact à l'aide des intervalles temporels. Ces derniers sont choisis tout simplement parce qu'ils constituent une composante essentielle du modèle, affectant sa dynamique, à travers lesquels l'influence sur la dynamique peut être analysée. Certaines des transformations présentées dans ce chapitre sont d'ailleurs proches de celles introduites dans [60], mais elles ont été revisitées dans la mesure où cette transformation des ITPN en TPN classiques (ie., non synchrone) reste notamment sur un temps continu, même si cette approche garantit l'inclusion des comportements réels dans ceux analysés. Dès lors, nous allons plutôt transformer le modèle ITPN initial en un modèle STPN (S pour Synchrone) qui considérera explicitement le temps discret et le synchronisme, sous une forme nécessairement analysable.

Précisons par ailleurs que nous ne traiterons pas ici des priorités (dédiées à la résolution de conflit) dans la mesure où notre préoccupation sera l'analyse exhaustive des comportements, i.e. nous désirons au contraire pouvoir analyser tous les comportements possibles. Ces priorités sont présentes dans le formalisme ITPN et peuvent être prises en compte dans l'analyse si besoin, mais ne font fondamentalement pas partie des caractéristiques essentielles à ce stade.

2.1.1 Définition du formalisme STPN

Un STPN est défini par $\langle P, T, Pre, Pre_t, Pre_i, Post, m_0, R_s \rangle$, avec P l'ensemble des places et T l'ensemble des transitions, Pre, Pre_t, Pre_i $Post$: $T \times P \rightarrow \mathbb{N}$ sont respectivement la fonction pré-condition, la fonction test, la fonction inhibition et la fonction post-condition, m_0 le marquage initial et

$R_s : T \to \mathbb{I}^*$ la fonction d'intervalle de temps résiduel statique.

Les définitions suivantes seront également utiles :

— Le marquage d'un STPN est défini par la fonction $m : P \to \mathbb{N}$.

— Une transition $t \in T$ est sensibilisée par un marquage m si ses places d'entrée possèdent tous les jetons permettant le tir de t, c'est à dire si $\big((m \geq Pre(t) + Pre_t(t)) \wedge (m < Pre_i(t))\big)$. On note $Enabled(m)$ l'ensemble des transitions sensibilisées par m.

— Une transition t est nouvellement sensibilisée par le tir d'un ensemble de transitions $Fired$ depuis un marquage m, noté $t \in newEnabled(m, Fired)$, si l'évolution du marquage induite par le tir de ces transitions entraîne la désensibilisation de cette transition, y compris par le marquage intermédiaire. Cette définition est la même que pour les SITPN, voir section 1.1.5.

— Un état d'un STPN est une paire $s = (m, R)$, avec m le marquage et R la fonction qui associe à chaque transition sensibilisée par m un intervalle de tir résiduel. On note respectivement $\downarrow R(t)$ et $\uparrow R(t)$ les bornes minimale et maximale de l'intervalle $R(t)$. L'état initial s_0 est (m_0, R_0), ou $R_0(t) = R_s(t)$, $\forall t \in Enabled(m_0)$.

— On appelle $Firable(s)$ l'ensemble des transitions tirables à partir de l'état $s = (m, R)$. Une transition $t \in Firable(s)$ est tirable à un instant discret θ ($\theta \in \mathbb{N}^*$) ssi : $t \in Enabled(m) \ \wedge \ \theta \in R(t)$.

2.1.2 Règles de transformation

A partir d'un modèle initial ITPN $\langle P^{ITPN}, T^{ITPN}, Pre^{ITPN}, Pre_t^{ITPN}, Pre_i^{ITPN}, Post^{ITPN}, m_0^{ITPN}, I_s, C, \succ \rangle$, le modèle analysable STPN est construit ainsi :

1. Structure du modèle :

   Le modèle STPN conserve la même structure que le modèle ITPN initial : il conserve les ensembles de places et transitions, les fonctions Pre (précondition, test, inhibiteur) et $Post$, et le marquage initial :

   — $P = P^{ITPN}$

   — $T = T^{ITPN}$

   — $(Pre = Pre^{ITPN}) \wedge (Pre_t = Pre_t^{ITPN}) \wedge (Pre_i = Pre_i^{ITPN}) \wedge (Post =$

$Post^{ITPN}$)

— $m_0 = m_0^{ITPN}$

2. Les intervalles résiduels

Les intervalles résiduels statiques R_s du modèle STPN sont définis par des règles de transformation spécifiques, basées sur l'impact des particularités précédemment évoquées, et décrites comme suit :

(a) *Impact de l'exécution synchrone* :

i. Etant donné que les transitions ne sont tirées que sur les fronts du cycle d'horloge (unité de temps logique), alors le modèle doit évoluer en temps discret. Pour un marquage m, $R : T \rightarrow \mathbb{I}^*$ est la fonction qui associe un intervalle résiduel dynamique de temps à chaque transition sensibilisée par m. Cet intervalle dynamique représente l'écoulement du temps $\theta \in \mathbb{N}^*$ par rapport à l'intervalle résiduel de temps statique R_s (dès lors que la transition est sensibilisée) ; il représente donc le temps restant pour le tir à travers un intervalle résiduel dont les bornes sont respectivement $max(1, \downarrow R_s - \theta)$ et $\uparrow R_s - \theta$, sachant que les tirs se produisent au rythme de l'unité de temps logique (i.e., $1ut$). La figure 2.2 montre les dates de tir potentielles (en rouge) pour la transition t_0 par rapport à une échelle de temps continu.

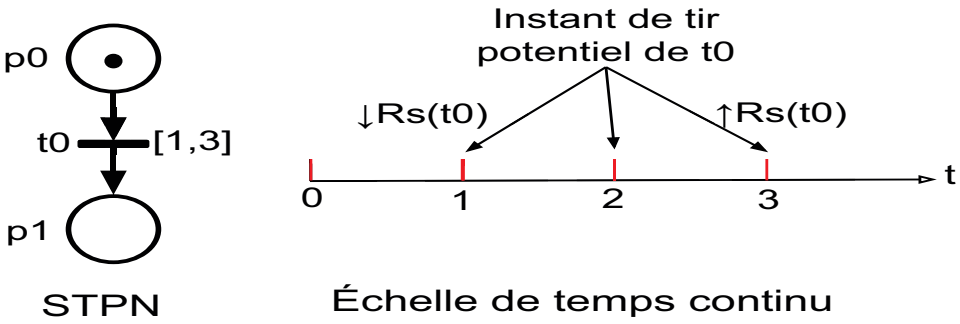


FIGURE 2.2 – Impact de l'exécution synchrone : évolution discrète du temps

ii. Nous rappelons qu'en plus d'être synchrone, notre modèle doit respecter une sémantique de tir "impérative". Ainsi, si une transition n'est associée ni à un intervalle temporel ni à une condi-

tion, cette transition sera alors tirée immédiatement, en 1 unité de temps ($1ut$), ce qui représente un cycle d'horloge ; son intervalle temporel résiduel statique est alors égal à $[1, 1]$. Prenons par exemple le modèle ITPN de la figure 2.3, la transition t_0 n'est liée ni avec un intervalle temporel ni avec une condition. Ainsi, dans notre STPN, cette transition est associée avec un intervalle $R_s(t_0) = [1, 1]$.

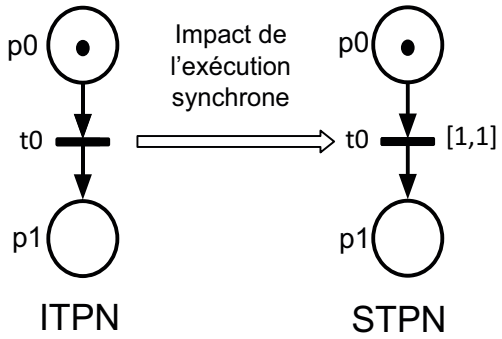


FIGURE 2.3 – Exemple de transformation : impact de l'exécution synchrone

iii. L'exécution sur une cible matérielle avec du vrai parallélisme impose une sémantique de tirs simultanés : toutes les transitions tirables à un même instant doivent être effectivement tirées en même temps. Par exemple, les transitions t_0 et t_1 du modèle STPN de la figure 2.4 sont tirées simultanément depuis le marquage initial $(p0, p1)$ après une unité de temps.

(b) *Impact de l'interprétation* :

L'impact de l'interprétation peut être représenté en ne considérant que les conditions. Certes, les actions impulsives et continues influencent l'évolution du système, notamment en modifiant les valeurs des signaux et des variables internes, elles-mêmes impliquées dans des conditions. Ceci étant, plus généralement, les évolutions du système "contrôlé", qui n'est évidemment pas modélisé (le modèle exprimant le contrôle et pas le système contrôlé) sont indirectement toutes prises en considération dans les conditions ; en effet, la valeur des conditions suffit à représenter les différentes évolutions possibles

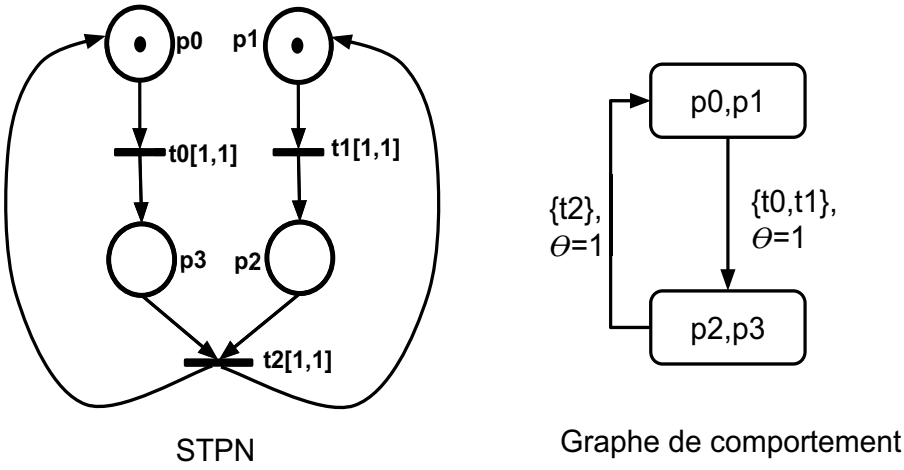


FIGURE 2.4 – Impact de l'exécution synchrone : tirs simultanés de transitions

du système liées à l'interprétation (pour toute condition, les valeurs "True" et "False" sont prises en compte).

Si une transition est associée avec une condition, elle sera tirée dès que sa condition devient vraie : soit immédiatement (en $1ut$) si la condition est vraie dès la sensibilisation de la condition, soit dans le pire des cas elle ne sera jamais tirée si cette condition reste fausse. Ainsi, l'intervalle de tir que nous devons analyser sera $[1, +\infty[$. Cette transformation permet de considérer toutes les valeurs potentielles des conditions.

Considérons le modèle ITPN de la figure 2.5 : la transition t_0 est liée à la condition c_0, c.à.d, $C(t_0, c_0) = 1$. Alors, cette transition sera tirable avec un temps résiduel $R_s(t_0) = [1, +\infty[$ sur notre STPN.

Dans le cas où deux transitions en conflit sont associées avec des conditions complémentaires (cf. modèle ITPN dans la figure 2.6), l'une de ces deux transitions est nécessairement tirée après une unité (i.e. soit la condition c est vraie, soit l'inverse de la condition c est vrai). Ainsi, ces deux transitions auront un intervalle de $[1, 1]$ dans notre modèle STPN. Cette situation est applicable aussi dans le cas de plusieurs transitions.

(c) *Impact de l'association de l'interprétation et des intervalles temporels* :

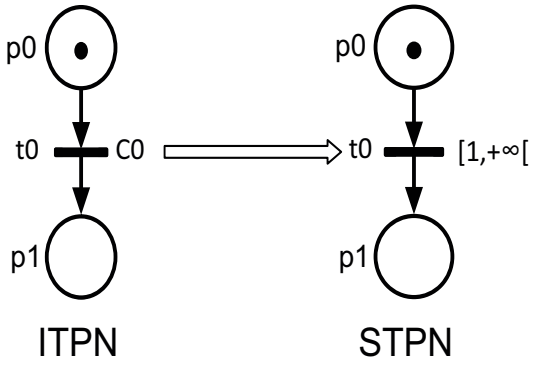


FIGURE 2.5 – Exemple de transformation : impact de l'interprétation

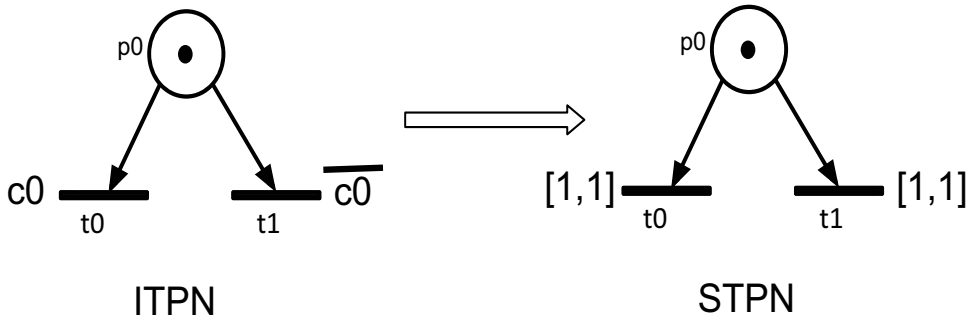


FIGURE 2.6 – Exemple de transformation : impact de l'interprétation, avec conditions complémentaires

Si un intervalle temporel et une condition sont associés à une même transition, cela peut mener à une situation de blocage : si la valeur de la condition est toujours fausse alors que la borne supérieure de l'intervalle est atteinte, cette transition ne peut pas être tirée mais le temps ne peut pas non plus continuer à s'écouler. Cette situation est particulière (pour l'analyse) et elle sera traitée en détails dans le chapitre 5.

(d) *Impact de notre sémantique d'exécution* :

Afin de garantir une exécution déterministe, nous imposons une sémantique de tir "impératif" : une transition doit être tirée dès qu'elle est tirable. Ainsi, si un intervalle temporel $[a, b]$ est associé à une transition (sans condition) sur le modèle ITPN, alors il deviendra un intervalle résiduel de la forme $[a, a]$ sur le modèle analysable STPN. L'exemple 2.7 illustre ce cas.

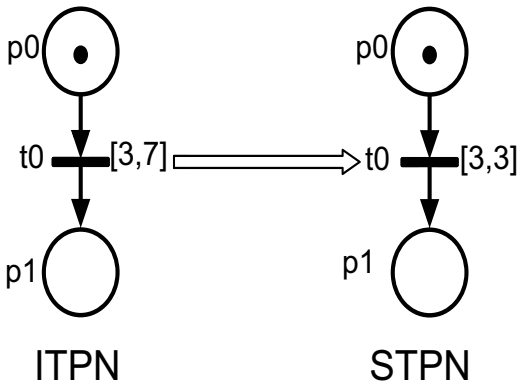


FIGURE 2.7 – Exemple de transformation : impact de la sémantique d'exécution

2.1.3 Gestion des conflits

Les règles de la transformation d'un modèle ITPN vers un modèle analysable STPN, ainsi que les définitions liées à ce formalisme, ont été présentées. Avant de définir la sémantique du modèle STPN, dans cette section, nous approfondissons la notion de conflit pour l'adapter à notre contexte. Comme notre modèle STPN est un RdP temporel généralisé (i.e., le nombre maximum de jetons est un entier) étendu (i.e., avec des arcs tests et inhibiteurs) synchrone, nous allons étudier les définitions de conflits existantes pour les

RdP temporels et étendus. Il s'agira alors de les adapter afin d'obtenir celle correspondant à notre formalisme et respectant le synchronisme. Une fois la relation de conflit définie, elle sera ensuite utilisée dans notre sémantique afin de garantir un comportement déterministe du modèle STPN et conforme à son exécution sur la cible matérielle.

Conflit Structurel

La définition classique du conflit structurel, qui consiste en la simple condition de ne pas avoir de places amont communes, peut être modifiée dès lors que l'on considère la possibilité d'avoir des arcs tests et surtout inhibiteurs. La prise en compte de ces arcs peut modifier la définition de conflit structurel. Par exemple, sur la figure 2.8 (a), les deux transitions t_0 et t_1 pourraient être considérées en conflit structurel ou pas étant donné que les jetons ne sont pas consommés. Cela introduit également une non-symétrie potentielle de la relation de conflit. Par exemple sur la figure 2.8 (b), les transitions t_0 et t_1 partagent une place amont mais avec 2 types d'arcs différents. Le tir de t_0 empêche celui de t_1, mais l'inverse n'est pas vrai. La notion de non-symétrie peut d'ailleurs également exister avec des arcs normaux, comme illustré sur la figure 2.10 (c), dans laquelle le tir de t_0 n'empêche pas le tir de t_1 puisque le jeton est remis. Enfin, la figure 2.8 (d) montre une situation dans laquelle l'ordre des tirs est influent (ce qui est une évidence) : le tir de t_0 met un jeton dans la place p_1 ce qui empêche alors le tir de t_1. Nous évoquons ce cas pour clairement préciser qu'une telle situation n'est pas considérée comme un conflit car d'une part ces deux transitions ne sont pas structurellement en conflit, et d'autre part cela reviendrait à considérer, et donc mélanger, l'influence d'un tir au cycle courant (instant présent) sur les tirs possibles au cycle suivant (instant futur).

Nous approfondirons ces notions sur la prise en compte des arcs tests/inhibiteurs et de la non-symétrie par la suite, en prenant en compte le contexte synchrone.

Conflit Effectif

Les conflits structurels ne s'adaptent parfaitement aux situations de conflits effectifs que si on se limite aux RdP saufs (un jeton au maximum par place), comme par exemple le RdP (a) dans la figure 2.9 où les transitions t_0 et t_1

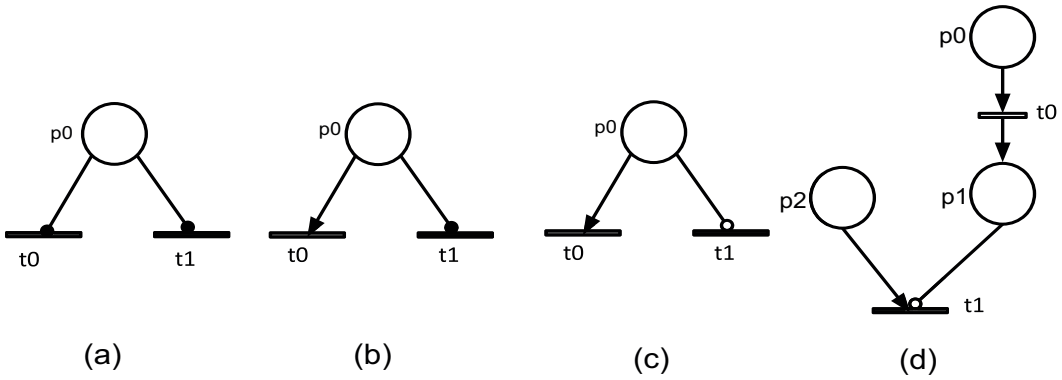


FIGURE 2.8 – Illustration des conflits structurels sur des RdP généralisés étendus

sont effectivement en conflit. Mais si l'on considère les RdP généralisés et/ou temporels, la notion de conflit effectif prend tout son sens. Par exemple sur la figure 2.9 (b), le nombre de jetons permet de tirer les deux transitions t_0 et t_1 ; de fait, elles ne sont pas en conflit (pour un tel marquage). De même, sur l'exemple 2.9 (c) les transitions t_0 et t_1 ne sont pas tirables à la même date, et donc pas en conflit effectif.

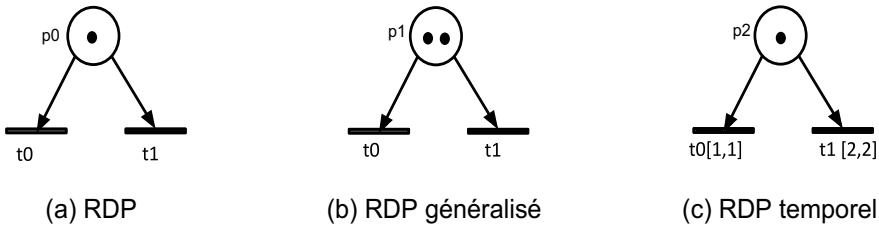


FIGURE 2.9 – Illustration des conflits structurels vs. effectifs

Il est ainsi utile d'approfondir la notion de conflit effectif afin de définir plus précisément les situations de conflits. De façon générale, la définition d'un conflit effectif entre deux transitions se compose de deux conditions :

— les deux transitions sont tirables à un même instant,

— le tir d'une transition empêche le tir de l'autre.

D'un point de vue formel, une première définition générale est la suivante : $\forall t_i, \forall t_j \in T$, t_i est en conflit effectif avec t_j pour un état s de marquage m si et seulement si : $t_i, t_j \in Firable(s)$ et, soit s' de marquage m' l'état obtenu par le tir de t_i, alors $t_j \notin Enabled(m')$. Dans cette définition, l'évolution du

marquage, mais également celle de l'évolution du temps dans le cas de réseaux de Petri temporels, sont prises en compte par l'utilisation des fonctions $Firable$ et $Enabled$.

Les situations de conflits en synchrone

Cependant cette définition ne prend pas en compte les cas spécifiques, en particulier les cas de re-sensiblisation, les arcs spécifiques, et les cas de non-symétrie. Nous allons donc approfondir cette définition pour définir précisément les situations de conflits dans notre contexte d'implémentation synchrone. En effet en synchrone, les gestion des conflits est d'autant plus importante que deux transitions en conflit risquent d'être tirées en même temps, utilisant un même jeton pour le tir de deux transitions différentes. Cette situation, contraire à la sémantique de base des RdP, peut mener à des comportements non souhaités par le concepteur (un OU devient un ET).

Re-sensibilisation La particularité de la sémantique synchrone réside dans le fait que toutes les transitions d'un modèle STPN doivent attendre au minimum une unité de temps après leur instant de sensibilisation pour être tirées. Ceci est vrai y compris dans le cas d'une re-sensibilisation. Considérons les modèles (a) et (b) de la figure 2.10 : le tir de t_1 empêche le tir de t_0 alors que le tir de t_0 désensibilise et re-sensibilise la transition t_1. Dans le modèle (a), qui est un réseau de Petri classique, le tir de t_0 n'empêche pas le tir de t_1 car le marquage de p_0 ne change pas. Par contre, si un intervalle temporel est associé à la condition t_1, cet intervalle sera ré-initialisé car t_1 est désensibilisée par le marquage intermédiaire $m - Pre(t_0)$. Or dans un STPN, ce qui est le cas du modèle (b), toutes les transitions ont obligatoirement un intervalle temporel de borne minimal $1ut$. Donc, en synchrone, le tir de t_0 empêche également le tir de t_1, car la désensibilisation induit que le prochain tir de t_1 ne sera possible qu'après 1 unité de temps. Ainsi, les deux transitions seront considérées comme en conflit l'une avec l'autre. Pour détecter ces situations, nous utiliserons l'adaptation de la définition du conflit effectif proposée dans [60], en utilisant le marquage intermédiaire $m - Pre(t)$, plutôt que le nouveau marquage "complet" $m' = m - Pre(t) + Post(t)$, pour détecter une désensibilisation inter-

médiaire.

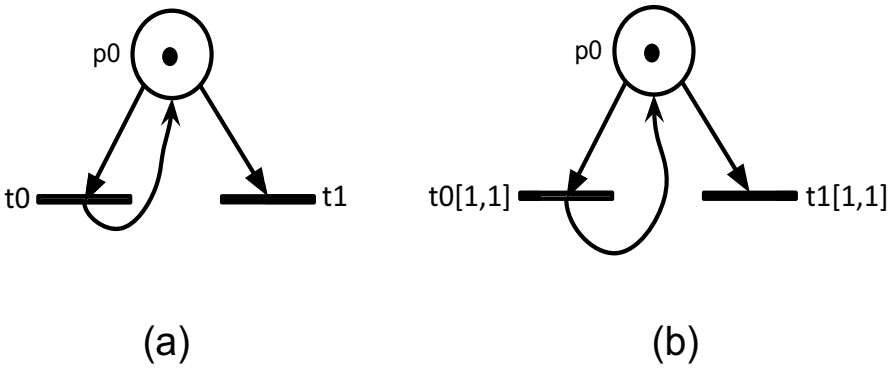


FIGURE 2.10 – Illustration du problème de la resensibilisation

Arc test Les arcs tests sont un cas particulier étant donné qu'ils ne consomment pas le(s) jeton(s), ils n'empêcheront pas le tir d'une autre transition (au sein d'un conflit effectif). De fait, nous considérons que ces arcs génèrent pas de situations de conflits, dans des cas tels que ceux décrits figures 2.8 (a) et (b). Pour la structure décrite figure 2.8 (b), nous aurons donc, en synchrone, t_0 et t_1 qui seront tirées en même temps. Afin d'exclure ces situations structurelles des conflits détectés, nous rajoutons comme condition que deux transitions ne peuvent être en conflit que si elles partagent des places amonts à travers des arcs "normaux", c'est à dire uniquement par la fonction Pre (donc sans considérer Pre_t, ni Pre_i cf. arc inhibiteur). Nous utilisons dans ce cas la définition des conflits structurels classiques.

Arc inhibiteur La détection des conflits basée uniquement sur la fonction de Pre-condition Pre (donc sans considérer Pre_i, ni Pre_t, cf. arc test) écarte également la possibilité que des transitions soient considérées en conflit lors de liens avec une (des) place(s) amont à travers des arcs inhibiteurs, comme sur l'exemple figure 2.8 (c). En effet, le tir de t_1 n'affecte nullement la transition t_0, qu'elle soit temporelle ou non.

La vérification de la désensibilisation d'une transition en se basant uniquement sur le marquage intermédiaire $m - Pre(t)$ contribue également à écarter les situations de conflits basées sur les arcs inhibiteurs.

Non symétrie Enfin, la dernière situation que l'on pourrait considérer comme particulière est la prise en compte de la non symétrie d'une

situation de conflit (cf. figure 2.11). Cependant, cette notion est entièrement liée aux situations que l'on vient d'étudier, et est donc résolue par les décisions précédemment prises.

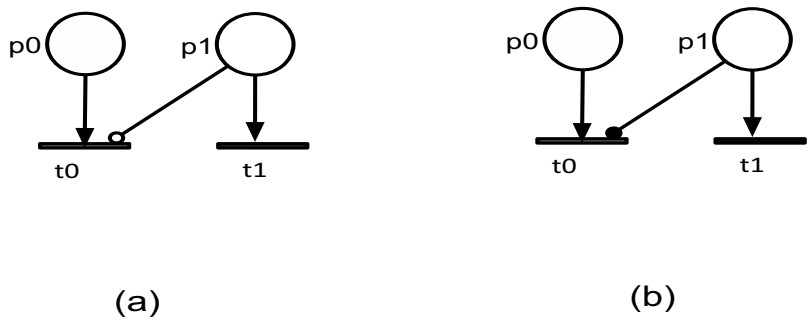


FIGURE 2.11 – Exemples de conflits non symétriques

Tous ces éléments peuvent être représentés par la définition formelle suivante, adaptée de [60] notamment car nos hypothèses sur les cas particuliers des arcs tests et inhibiteurs sont différentes.

Une transition t est en conflit effectif avec une transition t' dans l'état $s = (m, R)$, que l'on note $t \in conflict(s, t')$ si et seulement si :

- Les deux transitions sont en conflit structurel en partageant des places amont par des arcs "normaux" : $Pre(t) \cap Pre(t')$
- Les deux transitions sont tirables : $t, t' \in Firable(s)$
- Le tir de t' désensibilise (éventuellement de façon transitoire) t : $t \notin Enabled(m - Pre(t'))$

Les règles de transformation du modèle initial ITPN vers un modèle analysable STPN étant posées, ainsi que les règles de gestion de conflits, définissons la sémantique du modèle STPN, sans considérer les situations de blocage à ce stade.

2.2 La sémantique du formalisme STPN (sans blocage)

Dans cette section, nous définissons la sémantique du modèle STPN en prenant en compte la gestion des conflits telle que précédemment spécifiée. Par

contre, pour simplifier l'approche, cette section ne s'intéresse pour l'instant pas à la sémantique particulière du blocage (transition avec intervalle temporel et condition associés), que nous introduirons dans le chapitre 4.

Soit T^n l'ensemble des ensembles de transitions d'un modèle STPN. On définit la sémantique du modèle STPN comme un système de transitions temporisé $(S, s_0, \twoheadrightarrow)$, avec S l'ensemble des états, $s_0 = (m_0, R_0) \in S$ l'état initial et $\twoheadrightarrow \subseteq S \times (T^n \times \mathbb{N}^*) \times S$ la relation de changement d'états. Soit $Fired$ l'ensemble des transitions tirées d'un état s à un instant discret non nul θ. La relation de changement d'états est définie, $\forall Fired \in T^n$ et $\forall \theta \in \mathbb{N}^*$, comme suit :

1. Relation de changement d'états $s = (m, R) \overset{Fired,\theta}{\twoheadrightarrow} s' = (m', R')$, i.e. avec tir de transitions, ssi :
   (a) $\forall t \in Fired,\ t \in Firable(s) \wedge (Fired \cap Conflict(m, t) = \emptyset)$
   (l'ensemble $Fired$ contient toutes les transitions tirées, i.e. chaque transition qui est tirable et qui n'est pas en conflit avec une autre transition de cet ensemble, et donc qui sera tirée.)
   (b) $\forall t \in Enabled(m),\ \theta \leq\uparrow R(t)$
   (il n'existe pas de transition devant être tirée à une date antérieure.)
   (c) $m' = m - \sum_{t \in Fired} Pre(t) + \sum_{t \in Fired} Post(t)$
   (Mise à jour du marquage.)
   (d) $\forall t' \in Enabled(m'),\ R'(t') = R_s(t')$ ssi $t' \in newEnabled(m, Fired)$, sinon $R'(t') = [max(1, \downarrow R(t') - \theta), \uparrow R(t') - \theta]$
   (mise à jour des intervalles résiduels de tir : pour toute transition nouvellement sensibilisée par le tir de $Fired$, son intervalle résiduel est initialisé par son intervalle résiduel statique. Sinon, dans le cas où elle reste sensibilisée, alors son intervalle résiduel est décrémenté par θ (la borne minimale restant ≥ 1).)
2. Si aucune transition n'est tirée, c.à.d. $Fired = \emptyset$, alors seule une évolution du temps est possible : $s = (m, R) \overset{\emptyset,\theta}{\twoheadrightarrow} s' = (m, R')$ ssi :
   (a) $\forall t \in Enabled(m), \theta <\uparrow R(t)$
   (le temps peut évoluer tant qu'il n'existe pas de transition devant être tirée à une date antérieure.)

(b) $\forall t \in Enabled(m), R'(t) = R(t) - \theta$

(mise à jour des intervalles résiduels : tous les intervalles résiduels des transitions sensibilisées sont décrémentés par θ.)

Dans ce chapitre nous avons expliqué la transformation du modèle ITPN vers un modèle analysable STPN. Cette transformation réifie l'ensemble des impacts des particularités issues de l'implémentation synchrone et de la sémantique"impérative", à travers les intervalles temporels du modèle analysable (STPN). Le modèle analysable (STPN) est dès lors défini ainsi que sa sémantique. Au regard de ce nouveau formalisme, une re-considération de la définition des conflits a été présentée et prise en compte dans sa sémantique. Désormais, afin d'analyser ce modèle, il faut générer son graphe de comportement. Ce dernier, fait l'objet du prochain chapitre.

Chapitre 3

Analyse du modèle synchrone

Le chapitre précédent a exposé la transformation de modèles selon laquelle nous produisons un modèle STPN qui réifie son "modèle de calcul", au sens des particularités de son exécution sur la cible. Ces particularités ont principalement été intégrées dans la dimension temporelle du modèle et sa sémantique. Ce modèle STPN est évidemment analysable, et l'objet du présent chapitre est de proposer la construction d'un graphe de comportement synchrone, sa sémantique ainsi que l'algorithme de construction du graphe. La méthode de construction sera par la suite illustrée sur un exemple de modèle STPN simple.

3.1 Graphe de comportement synchrone

Le comportement des réseaux de Petri temporels est en général défini par l'ensemble des états accessibles à partir de son état initial. Comme discuté dans les sections 1.1.3 et 1.1.4, il n'existe actuellement pas de méthode permettant la représentation du graphe de comportement réunissant toutes les spécificités de notre formalisme. Cette section propose donc une approche de génération par énumération du graphe de comportement d'un modèle STPN : le Graphe de Comportement Synchrone, ou SBG (*Synchronous Behavior Graph*).

Un SBG est un graphe d'états défini par l'ensemble des marquages d'accessibles depuis l'état initial. Un état du SBG est noté $e = (m, R)$, avec m le marquage et $R : enabled(m) \rightarrow \mathbb{I}^*$ la fonction des intervalles de temps résiduel pour le tir des transitions sensibilisées par m. L'état initial est $e_0 = (m_0, R_0)$ avec m_0 le marquage initial et $R_0(t) = R_s(t)$, $\forall t \in enabled(m_0)$.

3.1.1 Sémantique du SBG

Nous définissons la sémantique du SBG comme un système de transitions temporisé $(E, e_0, \twoheadrightarrow)$, avec E l'ensemble des états, $e_0 = (m_0, R_0) \in E$ l'état initial et $\twoheadrightarrow \subseteq E \times (T^n \times \mathbb{N}^*) \times E$ la relation de changement d'états. La sémantique du SBG est très proche de celle du STPN (cf. section 2.2). Tous les changements d'états présentés dans la sémantique du STPN sont possibles dans le SBG. Cependant, l'espace d'état du STPN est potentiellement infini, lorsque les transitions peuvent être tirées à un temps discret compris entre 1 et $+\infty$, ce qui conduit à un nombre infini d'états successeurs. De plus, les approches par énumération sont connues pour être sources d'explosion com-

binatoire, même dans des cas finis (cf. chapitre 1). Ainsi, pour éviter dans certains cas la représentation explicite de la discrétisation du temps, la sémantique du SBG comporte deux cas particuliers qui dérivent de la relation de changement d'états définie dans la sémantique du STPN. Ces deux cas particuliers définissent deux "sauts" de temps qui permettent de réduire la taille du graphe SBG en préservant tous les marquages accessibles. Ils sont appelés relations de changement d'états par saut de temps.

Soit $Fired$ l'ensemble des transitions tirées à partir d'un état e à un instant discret non nul θ ($\theta \in \mathbb{N}^*$). La relation de changement d'états est définie, $\forall Fired \in T^n$ et $\theta \in \mathbb{N}^*$, comme dans la sémantique du STPN (cf. section 2.2). Dans cette section nous ne présentons que les différences d'avec cette sémantique, c'est-à-dire les changements d'états par saut de temps.

Le premier saut de temps utilisé concerne la représentation de la possibilité d'une progression infinie du temps dans le cas d'un intervalle dont la borne supérieure est infinie. Afin d'éviter l'énumération infinie de tous les instants de tirs possibles, nous représentons une boucle sur un même état qui représente un saut de temps d'une durée $\theta = 1$. Considérons par exemple l'état e_1 du SBG décrit figure 3.2 : la seule transition sensibilisée est t_1, avec un intervalle résiduel $[1, +\infty[$. Il y a donc une boucle sur e_1 permettant de représenter toutes les évolutions temporelles discrètes possibles avant le tir de t_1. Cette boucle sur un même état n'est évidemment possible que dans le cas ou toutes les transitions de cet état ont un intervalle résiduel en $[1, +\infty[$. Dans le cas contraire, l'évolution de $1ut$ fera évoluer les intervalles résiduels des autres transitions, ce qui mènera vers un état différent.

Formellement, ce changement d'états est décrit pour un état $e = (m, R)$ ainsi :

— Changement d'états avec évolution indéfinie/infinie du temps : $e \overset{\theta=1}{\twoheadrightarrow} e$ ssi :

1. $\forall t \in T, t \in enabled(m) \mid R(t) = [1, +\infty[$
   (cette relation n'est possible que si toutes les transitions sensibilisées dans l'état e ont un intervalle résiduel de tir en $[1, +\infty[$)

Comme le temps passé avant le tir de cette transition est indéfini (donc potentiellement infini), boucler sur le même état avec un temps $\theta = 1$ représente

bien cette indétermination. Il s'agit en fait d'une classe d'équivalence dans la mesure où les états issus de la discrétisation du temps sont réunis dans un seul état. Cette abstraction est utilisée dans la méthode ISG décrite dans [82] (cf. section 1.3.2). Dans la méthode ISG, l'auteur introduit en effet des états appelés *integer-pseudo-state*, selon une approche consistant à limiter l'horloge de chaque transition sensibilisée à sa borne inférieure et boucler sur le même état si la borne supérieure tend vers l'infini. Sinon, l'horloge continue jusqu'à la borne supérieure. Cette abstraction d'états a été prouvé dans [83]. Nous nous sommes inspirés de cette méthode afin de propose notre changement d'états avec évolution indéfinie/infinie du temps. Ainsi, la preuve proposé dans [83] est applicable dans notre cas.

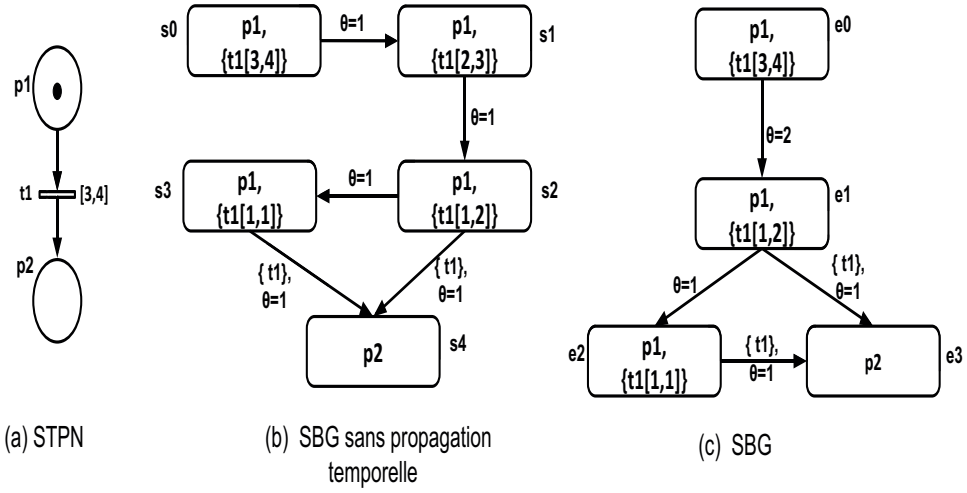


FIGURE 3.1 – Exemple d'un modèle STPN et son SBG sans et avec propagation temporelle

Le deuxième saut de temps que nous proposons afin de réduire la taille du SBG est de réduire les états qui sont similaires en termes de marquage mais différents en termes d'intervalles résiduels de tir ; il s'agit des états obtenus à partir des changements d'états basés sur l'évolution de temps seulement, i.e. sans tir de transitions. Par exemple, la transition t_1 du modèle STPN 3.1(a) n'est pas tirable avant la date $\theta = 3$. Sans réduction, il y a énumération de tous les changements d'états temporels de $1ut$ avant que le tir de la transition ne soit possible, comme représenté dans le SBG sans réduction de la figure 3.1(b). Pour réduire cette énumération, une réduction en termes de propaga-

tion temporelle est introduite par le changement d'états suivant :

— Changement d'états avec propagation temporelle, $e = (m, R) \overset{\theta}{\twoheadrightarrow} e' = (m, R')$ ssi :

1. $\forall t \in T,\ t \in enabled(m) \mid (R(t) = [c, d] \wedge c \neq d \wedge c \neq 1) \wedge (\nexists t' \in T, t' \in enabled(m) \mid R(t') = [j, k]) \wedge j \neq k \wedge j < c.$ (toutes les transitions sensibilisées sont tirables avec un intervalle de temps où la borne inférieure est la plus petite supérieure à 1, sans avoir atteint la borne supérieure)
2. $\theta =\downarrow R(t) - 1$ (la propagation temporelle permise avant qu'un tir soit possible)
3. $\forall t \in T,\ t \in enabled(m) \Rightarrow \downarrow R(t) > 1$ (la borne inférieure de toutes les transitions sensibilisées doit être strictement supérieures à 1).
4. $\forall t \in enabled(m), R'(t) = R(t) - \theta$ (mise à jour des intervalles résiduels de tir)

Le graphe basé sur cette réduction est présenté en 3.1(c). Il est évident que les deux graphes sont bi-similaires temporellement puisque cette réduction consiste seulement en une propagation temporelle, sans modification du marquage.

3.1.2 Préservation des propriétés du modèle STPN sur le SBG

Le comportement d'un RdP est défini par son graphe d'états. Comme nous l'avons expliqué, dans le cas du modèle STPN le graphe d'états est potentiellement infini (discrétisation de l'intervalle $[1, +\infty[$). Dans ce cas, faire des abstractions d'états est nécessaire afin d'éviter l'énumération infinie des états. Une abstraction d'états est un regroupement d'états, basé sur un critère d'équivalence prédéfini. Pour vérifier qu'une abstraction est correcte, cette dernière doit préserver les propriétés des états regroupés et plus généralement du graphe d'états complet. Les propriétés à préserver dépendent de l'utilisation prévue du graphe d'états. Classiquement, dans le domaine de la vérification formelle, on s'intéresse aux problèmes d'accessibilité, de vivacité et au caractère borné de l'espace d'états. On peut également utiliser les graphes d'états pour vérifier

des propriétés plus spécifiques comme les propriétés linéaires ou les propriétés de branchements (classiquement exprimées en logiques temporelles LTL et CTL*, respectivement), voir les propriétés temporisées (logiques TCTL par exemple).

Ainsi, lorsqu'on fait une abstraction ou une transformation de modèle, il est nécessaire de conserver la possibilité d'effectuer ces vérifications, ce qui passe par la préservation de l'ensemble des marquages accessibles et/ou des séquences de tirs, des traces complètes ou temporelles, etc. Plus la transformation préserve le comportement du modèle initial, plus les analyses effectuées sur le modèle transformé sont intéressantes.

Il existe différentes méthodes pour comparer deux sémantiques liées à des modèles temporels [10, 102, 14, 24, 21, 9], souvent basées sur l'inclusion des langages temporisés générés par le modèle ou sur la bisimulation temporisée. Cette dernière méthode permet de garantir la préservation de toutes les propriétés classiques des RdP, y compris les propriétés de branchement. Supposons que l'on désire comparer les sémantiques S_1 et S_2. Ces sémantiques sont temporellement bisimilaire s'il existe une relation d'équivalence telle que S_1 simule S_2 et vice-versa. On dit que S_2 simule S_1 ssi : à partir de 2 états équivalents e_1 et e_2 (un dans chaque sémantique), s'il existe une trace d'exécution τ_1 dans S_1 partant de e_1 et menant à un état e'_1, alors il existe dans S_2 une trace d'exécution équivalente τ_2 qui permet, à partir de e_2, d'atteindre un état e'_2 équivalent à e'_1.

Dans notre cas, nos abstractions sont très simples et la preuve de bisimilarité des sémantiques STPN et SBG est évidente. Nous avons donc une préservation du comportement des modèles, et toutes les propriétés de bases pourront donc être vérifiées sur notre graphe SBG.

De plus, sans passer par une preuve formelle, la réflexion suivante montre intuitivement que notre SBG préserve les propriétés du graphe de comportement du modèle STPN. En effet, la sémantique du SBG est la même que celle du modèle STPN, la seule différence repose sur les deux changements d'états introduits dans le SBG, qui servent seulement à une réduction du graphe dans deux cas :

— Changement d'états basé sur l'évolution indéfinie/infinie du temps : Comme nous l'avons mentionné ce changement d'états est effectué dans

le cas où toutes les transitions sensibilisées d'un état donné ont des intervalles de la forme $[1, +\infty[$, pour éviter une énumération infinie. Cette relation est inspirée de [83, 84], qui de plus présente une preuve de préservation de propriétés du modèle RdP temporel discret. Cette preuve est applicable dans notre cas.

— Changement d'états basé sur une propagation du temps : cette relation consiste à faire un saut de temps (une abstraction d'états) dans le cas où les transitions tirables ont un intervalle résiduel de tir de la forme $[c, d]$. De fait, pour une transition dans cette situation, au lieu d'énumérer tous les états obtenus suite à une évolution de temps de 1 ut, un saut de $\theta = c - 1$ est effectué. Autrement dit, cette réduction consiste à faire une abstraction de l'ensemble des états qui représentent la discrétisation du temps entre 1 et $c - 1$. Il n'y a aucun tir de transition, i.e. pas de changement de marquage : cette abstraction a le même marquage que les états regroupés, et son intervalle résiduel de tir est donc décrémenté d'une valeur égale à l'addition des instants de temps issus de la discrétisation $c - 1$. Le comportement du modèle est donc préservé par cette abstraction.

3.2 Algorithme de construction du SBG

La définition et la sémantique du SBG ayant été présentées, nous décrivons ici l'algorithme qui en permet sa construction. Nous commençons par définir différents ensembles de transitions tirables, qui permettent de classer les transitions sensibilisés selon leurs intervalles résiduels de tirs. Une fois ces ensembles définis, l'algorithme de construction du graphe est expliqué et donné en pseudo code (en 2 parties pour raison de lisibilité). Ensuite, il sera illustré sur un exemple simple de modèle STPN.

3.2.1 Les ensembles de transitions tirables

Le graphe de comportement d'un modèle STPN doit énumérer tous les changements d'états possibles. Pour cela, à partir d'un état du SBG, divers ensembles de transitions potentiellement tirables sont définis en fonction de leurs intervalles de temps résiduels. Ces ensembles sont utilisés afin de classer

les transitions sensibilisées selon leur "urgence" de tir. Les 4 ensembles de transitions (potentiellement) tirables à partir d'un état $e = (m, R)$ sont les suivants :

1. Transitions Nécessairement Tirées (*Necessarily Firable Transitions - NFT*) :
   — $1.1F(e)$ est l'ensemble des transitions qui doivent être immédiatement tirées après 1 unité de temps (1 *ut*), $t \in 1.1F(e)$ ssi :

$$t \in enabled(m) \wedge R(t) = [1, 1]$$

   .
2. Transitions Probablement Tirables (*Probably Firable Transitions - PFT*) [1] :
   — $1.bF(e)$ est l'ensemble des transitions qui peuvent être tirées après au moins 1 *ut* et au plus tard après b *ut*, $t \in 1.bF(e)$ ssi :

$$t \in enabled(m) \wedge R(t) = [1, b], b \in \mathbb{N}^* \setminus \{+\infty\}, b \neq 1$$

   .
   — $1.\infty F(e)$ est l'ensemble des transitions qui peuvent être tirées après au moins 1 *ut* mais qui, au pire, peuvent ne jamais être tirées, $t \in 1.\infty F(e)$ ssi :

$$t \in enabled(m) \wedge R(t) = [1, +\infty[$$

   .
3. Transitions Tirables Plus Tard (*Later Firable Transitions - LFT*) :
   — $a.aF(e)$ est l'ensemble des transitions qui devront être tirées après a *ut*, avec a le plus petit temps résiduel supérieur à 1 des transitions de cet ensemble, $t \in a.aF(e)$ ssi :

$$t \in enabled(m), R(t) = [a, a] \wedge$$

$$\nexists t' \in enabled(m) \mid R(t') = [b, b], b \neq 1 \wedge b < a$$

1. On rappelle que d'après les règles de transformation de la section 2.1, les intervalles $[a, +\infty[$ avec $a \neq 1$ sont impossibles.

4. Transitions Tirables dans un Intervalle (*Interval Firable Transitions - IFT*) :
   — $c.dF(e)$ est l'ensemble des transitions qui pourront être tirées après au moins c *ut* et au plus tard après d *ut*, avec c le plus petit temps résiduel supérieur à 1 des transitions de cet ensemble, $t \in c.dF(e)$ ssi : $t \in enabled(m), R(t) = [c, d] \wedge \nexists t' \in enabled(m) \mid R(t') = [j, k], c \neq 1 \wedge j < c \wedge j \neq k$

3.2.2 Description détaillée de l'algorithme

L'algorithme de construction du SBG est présenté dans les pseudo-codes Parties 1 et 2. Dans un soucis de simplification, cette version ne détaille pas la gestion des conflits, qui est présentée séparément dans la section 3.2.4.

Ainsi, à partir d'un état $e = (m, R)$, il existe un ou plusieurs ensembles de transitions tirables comme décrit ci-dessus. Ces ensembles sont calculés au premier pas de l'algorithme. En fonction de la combinaison de ces ensembles, il est ensuite possible de définir l'ensemble des changements d'états possibles, l'ensemble $Fired$ des transitions qui seront simultanément tirées lors de chacun de ces changement d'états, ainsi que le temps θ associé. Les différents cas sont tous représentés dans l'algorithme, et décrits ci-dessous en prenant comme exemple d'illustration le modèle STPN et son graphe SBG présentés sur la figure 3.2 .

Définition 1 *Pour un ensemble de transitions $K \in T^n$, on appelle $PowerSet(K)$ l'ensemble non vide de tous les sous-ensembles de K : $PowerSet(K) = \{B \in T^n | \ B \neq \emptyset, B \subset K\}$. Par exemple, le powerset de $K = \{t_1, t_2\}$ est $PowerSet(K) = \{\{t_1\}, \{t_2\}, \{t_1, t_2\}\}$.*

Les *powerset* seront utilisés lors de la construction du SBG, pour construire les ensembles de transitions tirables à partir d'un état donné.

L'algorithme de construction du SBG a comme entrée un modèle STPN et l'état initial du graphe e_0. À partir d'un état, s'il existe des transitions sensibilisées, les ensembles de transitions tirables (ETT) sont identifiés. Selon les combinaisons possibles des ETT, l'ensemble des changements d'états possibles, avec leur ensemble de transitions tirées et leur l'évolution temporelle, sont calculés. Les changements d'états identifiés serviront alors à construire

Algorithme 1 : Algorithme de construction du SBG - Partie 1

Data : $e_0 = (m_0, R_0)$; STPN
Result : SBG

Algorithm $SBG(e)$
  $ETT(e)$;
  /* Calcule des ensembles de transitions tirables */
  if $1.1F(e) \neq \emptyset$ **then** // cas (1)
    $e \overset{1.1F(e);\theta=1}{\twoheadrightarrow} e'$;
    $SBG(e')$;
    if $1.bF(e) \neq \emptyset \vee 1.\infty F(e) \neq \emptyset$ **then** // cas (2)
      for $(W \in PowerSet(1.bF(e) \cup 1.\infty F(e)))$ **do**
        $e \overset{1.1F(e)\cup W,\theta=1}{\twoheadrightarrow} e'$;
        $SBG(e')$;
      end
    end
  else
    if $1.bF(e) \neq \emptyset \vee 1.\infty F(e) \neq \emptyset$ **then** // cas (3)
      for $(W \in PowerSet(1.bF(e) \cup 1.\infty F(e)))$ **do**
        $e \overset{W,\theta=1}{\twoheadrightarrow} e'$;
        $SBG(e')$;
      end
      if $1.bF(e) = \emptyset \wedge a.aF(e) = \emptyset \wedge c.dF(e) = \emptyset$ **then** // sous-cas (3.1)
        /* boucle sur le même état */
        $e \overset{\theta=1}{\twoheadrightarrow} e$;
      else // sous-cas (3.2)
        $e \overset{\theta=1}{\twoheadrightarrow} e'$;
        $SBG(e')$;
      end
    else
      /* cf. partie 2 */
    end
  end

Algorithme 2 : Algorithme de construction du SBG - Partie 2

```
1  if a.aF(e) ≠ ∅ ∧ c.dF(e) ≠ ∅ then                          // cas (4)
2      if a < c then                                          // sous-cas (4.1)
3          e -a.aF(e),θ=a->> e' ;
4          SBG(e') ;
5      else
6          if a > c then                                      // sous-cas (4.2)
7              e -θ=c-1->> e' ;
8              SBG(e') ;
9          else                                               // sous-cas (4.3)
10             e -a.aF(e),θ=a->> e' ;
11             SBG(e') ;
12             for (W ∈ PowerSet(c.dF(e))) do
13                 e -a.aF(e)∪W,θ=a->> e' ;
14                 SBG(e') ;
15             end
16         end
17     end
18 else
19     if a.aF(e) ≠ ∅ then                                    // cas (5)
20         e -a.aF(e),θ=a->> e' ;
21         SBG(e') ;
22     end
23     if c.dF(e) ≠ ∅ then                                    // cas (6)
24         e -θ=c-1->> e' ;
25         SBG(e') ;
26     end
27 end
```

des successeurs à l'état en cours. Un appel récursif de l'algorithme est alors effectué pour chaque nouvel état obtenu. Ainsi l'algorithme boucle jusqu'à ce que plus aucune transition ne soit sensibilisée ou que tous les nouveaux états successeurs trouvés appartiennent déjà au SBG.

Nous allons expliquer l'algorithme à partir d'un exemple de modèle STPN et de son SBG (donnés figure 3.2) afin d'illustrer toutes les situations possibles issues des ETT.

Gestion de NFT seul

Les transitions appartenant à NFT doivent être tirées obligatoirement après $1ut$. Si NFT est non vide et qu'il n'y a pas de transition probablement tirable ($PFT = \emptyset$, que LFT et IFT soient vides ou non), alors il existe un changement d'états comportant toutes les transitions de NFT tirées simultanément en 1 ut (cas (1) de l'algorithme).

Exemple : La seule transition sensibilisée dans l'état e_0 (t_0) appartient à NFT. Elle est donc tirée en $1ut$, ce qui conduit au nouvel état e_1.

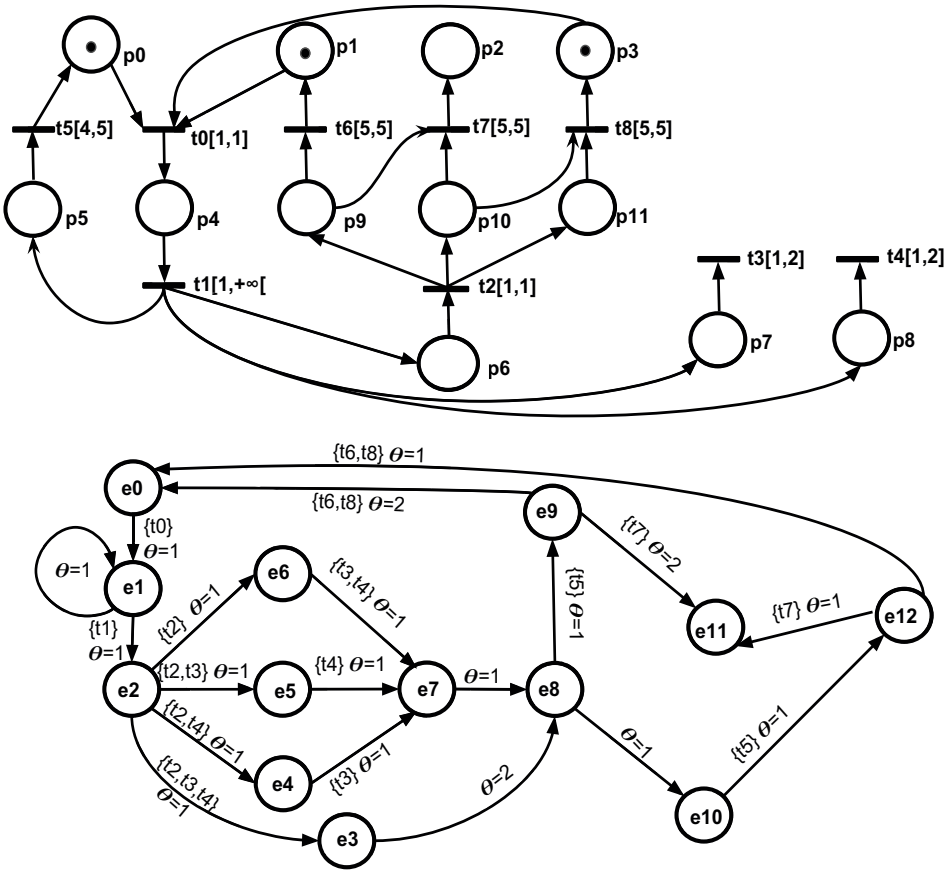
Gestion de PFT *(1.bF et 1.∞F)* seuls

Les transitions appartenant à PFT peuvent, ou non, être tirées. Dans le cas où les ensembles PFT sont non vides mais NFT est vide (que LTF et ITF soient vides ou non), le nombre de changements d'états possibles est alors calculé en utilisant le *powerset*. Pour chaque sous-ensemble de transitions du *powerset*, les transitions sont tirées simultanément après $1ut$ (cas (3)).

Il est également possible qu'aucune transition ne soit tirée :

— Dans le cas où $1.\infty F(e)$ est le seul non vide (NFT, LFT, IFT et $1.bF(e)$ étant donc vides), nous tombons dans la situation de l'évolution indéfinie/infinie du temps, pour laquelle nous avons définie dans la section 3.1.1 un changement d'états spécifique bouclant sur l'état lui-même (sous-cas (3.1)) pour représenter l'évolution temporelle. Ce changement d'états spécifique s'ajoute aux changements d'états résultant des tirs possibles des transitions de PFT (cas (3) de l'algorithme).

— Par contre, si l'un des intervalles LFT, IFT ou $1.bF(e)$ est non vide, alors le changement d'états représentant l'absence de tir des transitions



e0 p0,p1,p3, {t0[1,1]}	**e3** p5,p9,p10,p11, {t5[3,4],t6[5,5], t7[5,5],t8[5,5}]	**e6** p5,p7,p8,p9,p10,p11 {t3[1,1],t4[1,1],t5[3,4] t6[5,5],t7[5,5],t8[5,5]}	**e9** p0,p9,p10,p11, {t6[2,2],t7[2,2], t8[2,2]}	**e12** p0,p9,p10,p11, {t6[1,1],t7[1,1], t8[1,1]}
e1 p4, {t1[1,+∞[}	**e4** p5,p7,p9,p10,p11, {t3[1,1],t5[3,4], t6[5,5],t7[5,5],t8[5,5]}	**e7** p5,p9,p10,p11, {t5[2,3],t6[4,4], t7[4,4],t8[4,4]}	**e10** p5,p9,p10,p11, {t5[1,1],t6[2,2], t7[2,2],t8[2,2]}	
e2 p5,p6,p7,p8, {t2[1,1],t3[1,2] t4[1,2],t5[4,5]}	**e5** p5,p8,p9,p10,p11, {t4[1,1],t5[3,4], t6[5,5],t7[5,5],t8[5,5]}	**e8** p5,p9,p10,p11, {t5[1,2],t6[3,3], t7[3,3],t8[3,3]}	**e11** p0,p2,p11	

FIGURE 3.2 – Exemple d'un STPN et son SBG

PFT avec évolution du temps ($1ut$) entraînera un nouvel état successeur avec évolution des intervalles temporels (sous-cas (3.2)).

Exemple : Dans l'état e_1, seul $t_1 \in 1.\infty F(e)$ est sensibilisée. Donc, à partir de cet état, deux changements d'états sont possibles : un tir de transition, lorsque t_1 est tirée avec $\theta = 1$, ce qui mène à e_2 ; ou une évolution temporelle de $\theta = 1$ sans changement d'états pour représenter l'évolution indéfinie / infinie du temps.

Gestion de NFT et PFT combinés

La combinaison des deux cas précédents, c'est à dire lorsque que les deux ensembles NFT et PFT sont tous les deux non vides, est présentée dans le cas (2) de l'algorithme. Dans ce cas, les transitions de NFT seront tirées simultanément avec celles de chacun des sous-ensembles de PFT, menant à chaque fois à un nouvel état. Il existe également le cas ou aucune transition de PFT n'est tirée, alors seules celles de NFT sont simultanément tirées.

Exemple : Les transitions sensibilisées dans e_2 sont t_2, t_3, t_4 et t_5. La transition t_2 appartient à NFT, t_3 et t_4 appartiennent à PFT, et t_5 à ITF. Nous verrons plus tard comment sont gérées les transitions ITF, il suffit juste ici de savoir que t_5 n'est pas tirable. Ainsi, quatre changements d'états sont possibles, chacun menant à un nouvel état :

- *Toutes les transitions sont tirées avec $\theta = 1$ et mènent à l'état e_3.*
- *Une seule transition entre t_3 et t_4 est tirée simultanément avec t_2, avec $\theta = 1$, chaque instance menant à nouvel état (e_5 et e_4, respectivement).*
- *t_2 est tirée seule avec $\theta = 1$, ce qui mène à l'état e_6.*

Autres cas (propagation temporelle)

Les cas restants sont lorsque seuls LFT et IFT sont non vides. Nous tombons ici dans la situation de propagation temporelle définie dans la section 3.1.1. Ainsi, les transitions qui ont la plus petite borne inférieure des intervalles de temps résiduels seront tirées (cas (4.1), (4.3) et (5)). Si seulement IFT est non vide, ou si ce sont des transitions de IFT qui ont la plus petite borne inférieure c, alors seul le temps évolue de $c - 1$ ut (cas (4.2) et (6)) ce qui mène à un nouvel état e'.

Exemple 1 : Cette situation peut être illustrée en étudiant l'état e_3. A partir

de cet état, la transition sensibilisée possédant la plus petite borne inférieure est $t_5[3,4]$*, qui appartient à* IFT*. Les autres transitions* $t_6[5,5]$*,* $t_7[5,5]$ *et* $t_8[5,5]$ *appartiennent à* LFT*. Ainsi, aucune transition n'est tirée à partir de l'état* e_3*, et le temps évolue de* $\theta = 2$ *(i.e.,* $c-1$ *avec* $c = 3$*) menant vers l'état* e_8*. En* e_8*, l'intervalle de la transition* t_5 *devient* $[1,2]$ *et* t_5 *devient donc* PFT*.*

3.2.3 Étude de l'algorithme

Trois questions se posent pour la vérification d'un algorithme : sa terminaison (i.e, il ne boucle pas à l'infini), sa correction (i.e. il donne le bon résultat) et sa complexité algorithmique (i.e, pour une donnée d'entrée combien d'opérations sont nécessaires pour finir). En considérant ces trois questions, nous allons analyser notre algorithme.

Nous considérons dans ce qui va suivre la génération d'un nouvel état comme une opération atomique, c.à.d que les opérations élémentaires réalisées pour la génération d'un état sont agrégées (calcul de l'ensemble des transitions sensibilisées, calcul des ensembles de transitions tirables et les structures conditionnelles (if-then-else), gestion des conflits, etc). Ces opérations élémentaires sont vérifiés séparément (cf. chapitre 6). Une génération d'état est donc considérée comme une opération correcte. Le but de cette preuve est de montrer que notre algorithme énumère tout les états du SBG pour n'importe quel STPN d'entrée.

Terminaison

L'algorithme a comme entrée un état e. A partir de cet état, il calcule les états successeurs en se basant sur la structure du STPN, la sémantique du SBG $(E, e_0, \twoheadrightarrow)$ et la répartition des transitions sensibilisées par rapport aux ensembles des transitions tirables ETT. Une récursion de l'algorithme est alors lancée pour chaque nouvel état successeur e', si cet état n'existe pas déjà ($e' \notin E$). La récursion s'arrête s'il n'y a aucune transition sensibilisée dans l'état d'entrée, ou si tous les états successeurs générés existent déjà dans le SBG.

La terminaison de l'algorithme dépend également du nombre de successeurs générés à partir de l'état d'entrée. Or le nombre de successeurs est dépendant du nombre de transitions sensibilisées dans cet état, et des ensembles ETT dans

lesquelles ils se trouvent. Nous allons étudier en particulier les cas conduisant à utiliser la fonction PowerSet. Comme l'ensemble de transitions sensibilisées à chaque état e est un ensemble fini, la fonction PowerSet retourne un nombre nécessairement fini d'ensembles de transitions. Par conséquent, les variables i des boucles *for* utilisées dans l'algorithme sont comprises entre 1 et le cardinal du *PowerSet()*. Ainsi, ces boucles "for" se terminent, au sens d'un nombre fini d'itérations.

Dans le cas où le STPN n'est pas borné, l'algorithme boucle certes à l'infini, mais il suffit juste de fixer une condition de terminaison pour limiter le marquage "atteint" dans les places (i.e. imposer une borne k de sortie) ; c.à.d, si un nombre k ($k \in \mathbb{N}$) de jetons est présent dans un marquage m, alors l'algorithme s'arrête. Donc, nous considérons que les données d'entrée (le modèle STPN) sont bornées (nombre de transitions, marquage, etc).

Correction

Nous pouvons démontrer, par récurrence, que notre algorithme produit un résultat correct. Une preuve par récurrence consiste à montrer que l'algorithme donne le bon résultat dans deux étapes :

1. Initialisation $P(0)$: pour le cas de base, s'assurer que le résultat est correct ;
2. Transmission $P(n+1)$: supposer que le résultat est correct pour n itérations de l'algorithme $P(n)$ et prouver qu'il reste correct pour l'itération $n+1$.

$P(n)$ **:** Supposons que le SBG généré après n itérations est correct.

$P(0)$ **:** A partir de l'état initial e_0, si aucune transition n'est sensibilisée alors l'algorithme retourne un SBG qui a seulement e_0 comme état. Dans le cas inverse (l'ensemble de transitions sensibilisées n'est pas vide), il y a toujours un nombre fini de transitions sensibilisées d'entrée, puisque le modèle conçu comporte toujours un nombre fini de transitions. Les différents types d'intervalles temporelles possibles des transitions sont : $[1,1]$, $[a,a], a \neq 1$, $[1,b]$, $[1,+\infty[$ et $[a,b], a \neq 1$. Comme le montre la table 3.1, chacun de ces cas est assimilés à un des ensembles ETT et est traité par un des cas de l'algorithme. La grille de lecture de la

table 3.1 est la suivante : les colonnes de la première ligne correspond aux différents types de transitions possibles, et la seconde ligne sont les ensembles de transitions tirables correspondant. Une ligne correspond donc à une combinaison possible, avec O (Oui, l'ensemble n'est pas vide) et N (Non : l'ensemble est vide). La première colonne indique par quel cas l'algorithme traite cette combinaison. Selon la combinaison activée par les transitions sensibilisées dans un état donné, l'algorithme détermine, en respectant la sémantique du SBG, les tirs possibles de ces transitions. Les marquages et les intervalles résiduels de tirs des nouveaux états sont calculés comme défini dans la sémantique du SBG.

TABLE 3.1 – Les différentes combinaisons de transitions traitées par l'algorithme

	$[1,1]$	$[1,b]$	$[1,+\infty[$	$[a,a]$	$[a,b]$
	NFT	PFT		LFT	IFT
cas 1	O	N		O/N	O/N
cas 2	O	O		O/N	O/N
cas 3	N	O		O/N	O/N
cas 4	N	N		O	O
cas 5	N	N		O	N
cas 6	N	N		N	O
Arrêt d'algorithme	N	N		N	N

$P(n+1)$ **:** Quel que soit le nombre d'itérations n, à un état e, s'il existe une (plusieurs) transition(s) sensibilisée(s), alors toutes les transitions sensibilisées dans cet état sont classées forcément dans l'un des ensembles de transitions tirables. L'algorithme traite toutes les combinaisons possibles de ces ensembles (cf. table 3.1) et conduit vers les états successeurs e'. Pour chaque état successeur e', s'il n'existe pas de transition sensibilisée ou si tous les états e' existent déjà dans le SBG alors l'algorithme s'arrête et retourne le SBG résultant. Sinon, une nouvelle récursion de l'algorithme avec les nouveaux états est faite. Et ainsi de suite.

Ainsi, l'algorithme traite toutes les combinaisons possibles d'ETT pour chaque état traité, et génère un successeur pour chaque évolution d'états possibles. Puis une récursion est lancée sur chacun de ces successeurs. L'algorithme énumère donc tous les états du SBG pour n'importe quel modèle STPN d'en-

trée. Par supposition (même si cela a été vérifié lors de la validation des algorithmes, cf. chapitre 6), la génération d'un nouvel état est toujours correcte. Par conséquent, cette preuve permet d'assurer que notre algorithme donne toujours un bon SBG pour n'importe quel STPN d'entrée.

Complexité

En général, la complexité de la génération des graphes de comportement pour les RdP classiques dépend directement du parallélisme entre les transitions du modèle (i.e., parallélisme traité par entrelacement). Dans notre cas, cela n'est plus vrai, i.e. toutes les transitions en parallèle sont tirées simultanément par un seul changement d'états. Par contre, la complexité de notre algorithme est liée à l'énumération discrète du temps et les tirs potentiels des transitions (parallélisme effectif). Le calcul de la complexité de notre algorithme est lié à la configuration des intervalles temporels des transitions sensibilisées (les combinaisons ETT). Ainsi, c'est impossible de calculer la complexité exacte ; en l'occurrence nous pouvons donner le pire cas.

La définition classique de la complexité algorithmique est le nombre d'opérations effectuées par l'algorithme afin de rendre le résultat attendu. Le nombre d'opérations est exprimé en fonction de la taille des données d'entrées. Dans le cas de notre algorithme, les opérations sont les conditions, des boucles "for", la boucle de récursion de l'algorithme et la génération des nouveaux états. Comme nous l'avons expliqué, nous considérons ici la génération d'un nouvel état comme une opération atomique "constante". La table 3.2 indique le nombre d'états successeurs générés pour chaque cas de l'algorithme. Les lignes de la table 3.2 sont les cas présents dans l'algorithme et les colonnes sont les ensembles de transitions tirables. Les notations O et N correspondent respectivement aux cas où les ensembles issus de ETT sont non vides ou vides. La dernière colonne de la table 3.2 représente le nombre d'états successeurs générés. Il faut préciser que pour chaque itération de l'algorithme, un seul cas (combinaison de ETT) est possible.

Dans la majorité des cas (les combinaisons ETT) d'algorithme, un seul état est généré par itération. Par contre, le nombre d'états générés peut varier entre 1 et m ($m \in \mathbb{N}^+$) dès qu'il y a présence de transitions probablement tirables. Comme nous l'avons préalablement expliqué, si $PFT \neq \emptyset$ ou $IFT \neq \emptyset$ (cas 4.3), une boucle "for" est utilisée pour énumérer les tirs probables, avec comme

borne maximale d'itération la cardinalité de l'ensemble $PowerSet(PFT)$ (resp. $PowerSet(IFT)$). Dès lors, dans cette situation le nombre d'états générés m est égal à $2^{|PFT|}$ (resp. $2^{|IFT|}$), ce qui correspond à la complexité de calcul de la fonction $PowerSet$. Ainsi, la pire complexité de l'algorithme, si on suppose que l'algorithme génère chaque pour itération $2^{|PFT|}$ états, alors sera pour n itérations, est $n * 2^{|PFT|}$ (resp. $n * 2^{|IFT|}$).

TABLE 3.2 – Nombre d'états générés pour une itération de l'algorithme, à partir d'un état donné

Cas /ETT	NFT	PFT	LFT	IFT	Nombre d'états générés (m)
cas 1	O	N	N	N	1
cas 2	O	O	N	N	|PowerSet(PFT)|+1
cas 3	N	O	O/N	O/N	|PowerSet(PFT)|+1
cas 4 ($a \neq c$)	N	N	O	O	1
sous-cas 4.3 ($a = c$)	N	N	O	O	|PowerSet(IFT)|+1
cas 5	N	N	O	N	1
cas 6	N	N	N	O	1

La pire complexité estimée de l'algorithme est exponentielle. Elle dépend directement du nombre de transitions potentiellement tirées et du parallélisme entre de telles transitions (dont l'intervalle est, rappelons-le, infini). Il s'agit bien évidemment d'une limitation classique des méthodes de construction de graphe, induite ou accrue par un fort parallélisme. Néanmoins les systèmes que nous considérons, à savoir les dispositifs implantables ou plus généralement petits systèmes embarqués, sont nécessairement contraints (limités en ressources) et le parallélisme effectif (i.e., à l'instant t) reste limité en réalité. Si on veut traiter cette complexité théorique, notre méthode pourrait tout de même être améliorée en s'inspirant des approches symboliques [69] pour la limitation liée à l'énumération, et des approches par ordre partiel [45, 16] pour l'élimination des entrelacements liés aux tirs probables.

3.2.4 Algorithme de gestion des conflits

Cette section explique notre méthode de gestion des conflits lors de la construction du SBG et en donne l'algorithme détaillé. Il intervient dans l'algorithme à chaque fois qu'un changement d'états basé sur des tirs de transi-

tions est construit. L'algorithme 3 de gestion des conflits prend alors en entrée l'ensemble des transitions tirables à partir d'un état e (que nous appellerons $Fired$). La première étape est de diviser cet ensemble en sous-ensembles de transitions ne contenant aucun conflit. Pour chaque couple de transitions en conflit[2], $Fired$ est divisé en sous-ensembles afin de séparer ces transitions. Ceci est appliqué récursivement sur chacun des sous-ensembles jusqu'à avoir séparé tous les conflits. Après suppression des doublons et des partitions, nous obtenons alors un ensemble WC de sous-ensembles de transitions tirables simultanément, car libres de conflit. Chacun des sous-ensembles $Fired_i$ de WC mène alors à un nouvel état du graphe SBG : $\forall Fired_i \in WC, e \overset{Fired_i,\theta}{\twoheadrightarrow} e'_i$.

Exemple : L'état e_{12} de la figure 3.2 illustre la gestion des conflits : les transitions sensibilisées $t_6[1,1]$, $t_7[1,1]$ et $t_8[1,1]$ appartiennent à NFT et sont immédiatement tirables, mais sont en conflit (t_7 est en conflit avec t_6 et t_8, mais attention t_6 et t_8 ne sont pas en conflit). En appliquant l'algorithme 3 sur $Fired = \{t_6, t_7, t_8\}$, nous obtenons $WC = \{\{t_7\}, \{t_6, t_8\}\}$. Chacun des sous-ensembles de WC est alors tiré avec $\theta = 1$, et ils mènent respectivement aux états e_{11} et e_0.

3.2.5 Comparaison du SBG avec d'autres approches

Dans cette sous-section nous désirons comparer les différences conceptuelles entre notre graphe SBG et les graphes de comportement pour RdP temporel les plus proches de notre modèle STPN, en terme de sémantique : le SCG (State Class Graph) [12] pour une sémantique continu de temps et l' ISG (Integer-State Graph) [82, 83] pour la sémantique en temps discret (cf. section 1.3). Les différences entre les trois graphes seront illustrées à travers deux exemples de modèles STPN, décrits sur la figure 3.3, avec leur SCG, ISG et SBG.

Les tirs simultanés de transitions

Les deux approches ISG et SCG suivent une sémantique d'entrelacement pour représenter le tir de transitions en parallèle. Dans ces graphes, il y a une différentiation explicite de l'ordre de tir des transitions en parallèle, ce qui génère des états avec des marquages inexistants dans le contexte d'une exécution synchrone. Considérons le modèle STPN (1) et ses graphes, donnés

2. les conflits sont détectés suivant la relation définie section 2.1.3

Algorithme 3 : Algorithme de gestion des conflits

Data : $Fired \in T^n$, $e = (m, R)$
Result : $WC = \{L \neq \emptyset | L \subset Fired\}$

```
1  WC ← ∅ ;
   /* WC est l'ensemble des ensembles de transitions tirables sans
      conflit.                                                    */
2  Algorithm algo(Fired)
3      if t, t' ∈ Fired ∧ t' ∈ conflict(s, t) then
4          Fired' ← Fired − {t} ;
5          Fired'' ← Fired − {t'} ;
6          algo(Fired') ;
7          algo(Fired'') ;
8      else
9          if ∄L ∈ WC ∧ Fired ⊄ L then
10             WC ← Fired ;
               /* si l'ensemble de transitions tirables Fired n'est inclus
                  dans aucun ensemble de WC, alors il est affecté à
                  l'ensemble des ensembles de transitions sans conflit.  */
11         else
12             if L ⊂ Fired then
13                 WC − {L} ;
14                 WC ← Fired ;
15             end
16         end
17     end
18     return WC
```

dans la figure 3.3. Nous pouvons remarquer que certains marquages existent dans le SCG (dans les classes d'états c_1 et c_2) et dans l'ISG (dans z_2 et z_3) alors qu'ils ne sont pas présents dans le SBG. En effet, les deux premières méthodes traitant les transitions de façon asynchrone, cela génère des marquages qui n'existent pas dans notre contexte d'exécution synchrone pour lequel les transitions en parallèle sont tirées simultanément.

Cette particularité limite les possibilités d'analyse d'atteignabilité des graphes SCG et ISG dans notre contexte. Si un état est atteignable dans le SCG ou l'ISG, en termes de marquage, nous ne pouvons pas savoir a priori si c'est un état réel ou non. Par contre, si un marquage est inatteignable dans le SCG ou l'ISG, nous pouvons être sûr qu'il ne le sera pas non plus dans l'espace d'états réel. Au contraire, notre SBG respecte exactement les contraintes de

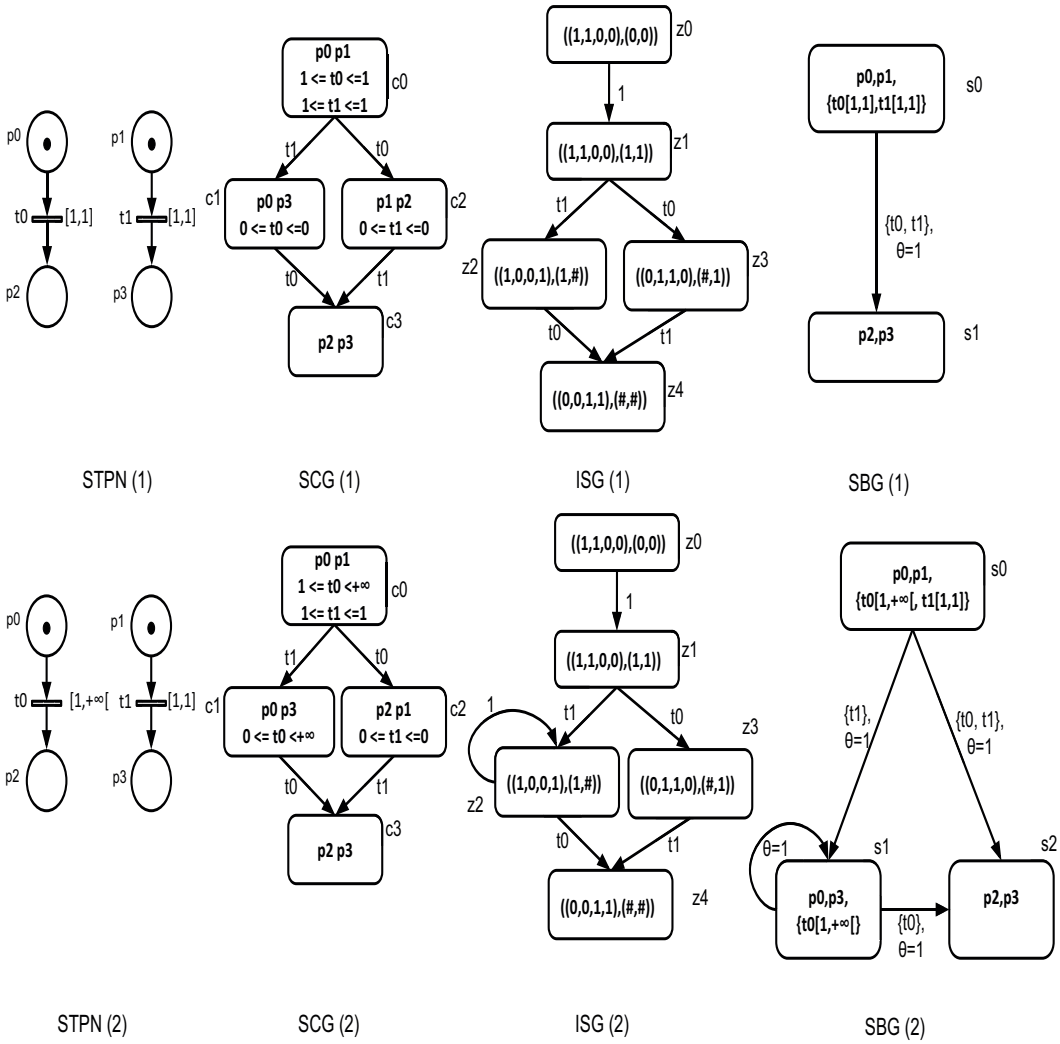


FIGURE 3.3 – Un exemple de STPN avec son SCG, ISG et SBG

tir synchrone, et donc par conséquence préserve la réalité des marquages.

Le tir potentiel de transitions

Dans le cas du modèle STPN (2) de la figure 3.3, la transition t_0 peut être tirée à n'importe quel moment entre 1 et $+\infty$, ce qui conduit à un espace d'état infini. Dans ce cas, le SCG fait une abstraction d'espace d'états (classes c_0 et c_1), alors que la méthode sous-jacente à l'ISG utilise la notion de d'*integer-pseudo-state* [83] (état $z2$), qui consiste à boucler sur le même état si la borne supérieure tend vers l'infini. Cela permet de représenter la possibi-

lité de progression du temps infini/indéfini, méthode dont nous nous sommes inspirés dans notre méthode de construction du SBG.

Précision des traces temporelles

Dans les systèmes temps réel, l'analyse des traces (quelles transitions sont tirées et dans quel ordre) est importante, principalement en considérant la dimension temporelle (à quel moment). Cependant, les propriétés temporelles ne peuvent pas être vérifiées précisément dans le SCG, puisque le graphe est une abstraction de l'espace d'états (qui est infini) en effectuant un sur-ensemble des intervalles temporels. De plus, la valeur absolue des horloges n'est pas stockée dans le graphe, ce qui rend difficile une analyse portant sur la dimension temporelle des traces. Ces différentes affirmations sont présentées dans [68, 42], même si plusieurs méthodes ont été proposées pour remédier à ce problème [14]. Clairement, les méthodes SBG et ISG étant énumératives sur le temps discret, elles conservent de façon plus précise les valeurs des horloges ainsi que les traces temporelles.

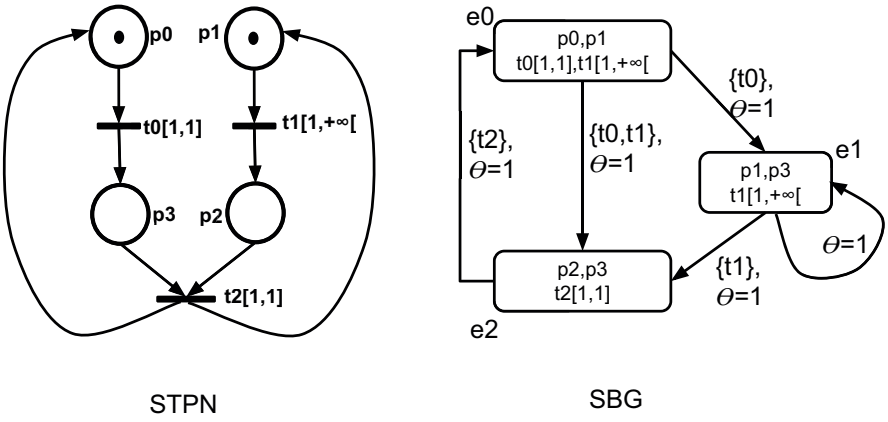


FIGURE 3.4 – Les traces temporelles d'exécution du modèle STPN, extraites du SBG

La figure 3.4 montre un modèle STPN et son SBG, et la table 3.3 indique les traces temporelles minimales et maximales entre les états de ce SBG. La première colonne représente les états sources (depuis lesquels le tir de transitions est considéré) et la première ligne représente les états cibles. S'il existe un chemin entre les deux états source-cible, alors la séquence de tir de transitions,

TABLE 3.3 – Traces temporelles du modèle STPN de la figure 3.4

Source \ Cible	e0	e1	e2
e0	$\{t0, t1, t2\}$ 2, $+\infty$	$\{t0\}$ 1, 1	$\{t0, t1\}$ 1, $+\infty$
e1	$\{t1, t2\}$ 2,$+\infty$	$\{\}$, $\{t1, t2, t0\}$ 1, $+\infty$	$\{t0, t1\}$ 1, $+\infty$
e2	$\{t2\}$ 2, $+\infty$	$\{t2, t0\}$ 2, 2	$\{t2, t0, t1\}$ 2, $+\infty$

la durée minimale et la durée maximale sont données dans la cellule concernée. Dans le cas inverse, s'il n'existe pas de chemin, cela est noté par le caractère "-".

Exemple : Prenons l'exemple de la deuxième colonne : la séquence de tir de transitions (t_0, t_1, t_2) commence à l'état e_0 et conduit à l'état e_0, en passant soit directement par l'état e_2, soit en passant d'abord par l'état e_1. Cette séquence de tir a besoin de 2 unités de temps au minimum (si chemin direct) et a une durée infinie au maximum (si boucle infinie sur e_1). De l'état e_1 vers l'état e_0, une seul chemin est possible par le tir des transitions t_1, t_2 en séquence entre une durée 2, $+\infty$. Et, depuis l'état e_2 vers l'état de e_1, un seul chemin est possible par le tir de la transition t_2 qui prend une unité de temps.

Dans ce chapitre nous avons présenté le graphe de comportement SBG (State Behaviour Graph) pour le modèle STPN, sa sémantique, son algorithme de construction ainsi que l'algorithme de gestion des situations de conflits. Nous avons souligné que notre graphe SBG préserve toutes les propriétés comportementales du modèle STPN, de par le fait que la sémantique du graphe SBG ne diffère de la sémantique du modèle STPN que par des considérations de propagations temporelles. Nous avons également étudié la correction de l'algorithme de construction, qui a été illustrée sur un modèle STPN simple. Donc, les bases de construction de notre graphe SBG ont été introduites. Cependant, le graphe SBG n'est à ce stade pas complet. En effet, la gestion des transitions probablement bloquées (les transitions du modèle ITPN associées avec un intervalle de temps et une condition) n'est pas encore prise en compte. Nous allons donc, dans le prochain chapitre, compléter notre formalisme STPN et

son SBG afin de pouvoir introduire explicitement (sur le graphe) le blocage potentiel de ce type de transition.

Chapitre 4

Prise en compte des situations de blocage

Nous avons mentionné dans les chapitres précédents que l'association conjointe d'un intervalle temporel et de conditions à une même transition pouvait induire un blocage, sans pour autant avoir considéré ce phénomène dans la sémantique du modèle ni dans la construction du graphe de comportement. C'est l'objet de ce chapitre que de prendre en compte cette sémantique de blocage à la fois dans le modèle analysable STPN et dans son SBG. Les transitions potentiellement bloquées doivent être au préalable identifiées lors de la transformation du modèle ITPN vers le modèle STPN.

4.1 Formalisme STPN avec transitions potentiellement bloquées

Les transitions potentiellement bloquées doivent être traitées différemment des autres transitions. En effet, si une de ces transitions est sensibilisée, elle peut être tirée, si la condition devient vraie, entre la date de tir au plus tôt et la date de tir au plus tard de son intervalle temporel. Une fois la date de tir au plus tard atteinte, deux cas sont possibles : soit la condition devient vraie et la transition est tirée, soit la condition reste fausse et la transition passe dans un état dit de blocage. Dans le cas de l'analyse formelle, nous devons être exhaustif et donc énumérer tous les cas possibles, y compris, voire même surtout, le risque de blocage.

Pour différencier le formalisme STPN présenté jusque-là du formalisme STPN avec transitions potentiellement bloquées, ce dernier sera notée STPNB. La définition du formalisme STPNB et sa sémantique sont une extension de celles du STPN (cf. chapitre 2).

4.1.1 Définition du formalisme STPNB

Un modèle STPNB est un n-uplet $< P, T, T_{PB}, Pre, Pre_i, Pre_t, Post, m_0, R_s >$, avec T_{PB} l'ensemble des transitions potentiellement bloquées de T ($T_{PB} \subset T$), et $R_s : T \rightarrow \mathbb{I}^*$ est la fonction d'intervalle résiduel de tir statique incluant le blocage. Ce modèle analysable est obtenu à partir du modèle initial ITPN $< P^{ITPN}, T^{ITPN}, Pre^{ITPN}, Pre_i^{ITPN}, Pre_t^{ITPN}, Post^{ITPN}, m_0^{ITPN}, C,$ $F, A, I_s, Clock, \succ >$ comme suit :

— Comme pour la transformation en STPN la structure du modèle est préservée : $P = P^{ITPN}$, $T = T^{ITPN}$, $Pre = Pre^{ITPN}$, $Pre_i = Pre_i^{ITPN}$, $Pre_t = Pre_t^{ITPN}$, $Post = Post^{ITPN}$ et $m_0 = m_0^{ITPN}$.
— L'ensemble T_{PB} contient toutes les transitions potentiellement bloquées. Cet ensemble est constitué des transitions du ITPN possédant un intervalle temporel et une condition associée. Formellement, $t \in T_{PB}$ ssi : $t \in T^{ITPN} \wedge I_s(t) \neq \emptyset \wedge \exists c \in C \mid C(t,c) = 1$.
— Les intervalles résiduels R_s du modèle STPNB sont définis par les mêmes règles de transformation que pour la génération du modèle STPN (cf. section 2.1).

Les fonctions de marquage et de sensibilisation sont les mêmes que pour un modèle STPN. Un état du modèle STPNB est un triplet (m, B, R), avec m le marquage, B l'ensemble des transitions bloquées dans cet état et R la fonction qui associe à chaque transition sensibilisée par m un intervalle de tir résiduel. L'état initial s_0 est (m_0, B_0, R_0), où m_0 est le marquage initial, $B_0 = \emptyset$ et $R_0(t) = R_s(t)$, $\forall t \in enabled(m_0)$.

On appelle $Firable(s)$ l'ensemble des transitions tirables à partir de l'état $s = (m, B, R)$. Une transition $t \in Firable(s)$ est tirable à un instant θ ($\theta \in \mathbb{N}^*$) si elle est sensibilisée, tirable d'un point de vue temporel et non bloquée : $t \in Firable(s), s = (m, B, R)$ ssi : $t \in enabled(m) \ \wedge \ \theta \in R(t) \wedge t \notin B$.

On appelle $PBlocked(s)$ l'ensemble des transitions qui pourraient être bloquées à l'état $s = (m, B, R)$. Une transition $t \in PBlocked(s)$ ssi : à un instant discret non nul θ, $t \in T_{PB} \wedge t \in enabled(m) \wedge \theta =\uparrow R(t)$.

Une transition bloquée à l'état $s = (m, B, R)$ sera débloquée si le tir d'une ou d'un ensemble de transitions $Fired \in T^n$ à partir de cet état la désensibilise. On note $Unblocked(s, Fired)$ l'ensemble des transitions désensibilisées par le tir de $Fired$ à partir de l'état s. Formellement, $t \in Unblocked(s, Fired)$ ssi : $t \in B \wedge t \notin enabled(m - \sum_{t' \in Fired} Pre(t'))$.

La définition, et donc la gestion, des conflits est la même que pour le modèle STPN. Précisons juste que les transitions bloquées ne pourront jamais être considérées en conflit dans la mesure où une transition bloquée n'est plus tirable jusqu'à ce qu'elle soit débloquée.

4.1.2 Sémantique du formalisme STPNB

La sémantique du formalisme STPNB est très proche de celle des STPN. Nous en donnons ici la définition formelle, suivi par des explications "en francais" et par un exemple d'illustration.

Soit T^n l'ensemble des ensembles de transitions d'un STPNB, et T^n_{PB} l'ensemble des ensembles de ses transitions potentiellement bloquées (i.e. transitions appartenant à T_{PB}). La sémantique d'un STPNB est le système de transitions étiquetées $(S, s_0, \twoheadrightarrow)$, avec S l'ensemble des états, $s_0 = (m_0, B_0, R_0) \in S$ l'état initial et $\twoheadrightarrow \subseteq S \times (T^n \times T^n_{PB} \times \mathbb{N}^*) \times S$ est la relation de changement d'états basée sur le tir/blocage des transitions. Soit $Fired$ et $Blocked$ respectivement l'ensemble des transitions tirées et l'ensemble des transitions bloquées à partir d'un état s à un instant discret non nul θ. Cette relation est définie, $\forall Fired \in T^n$, $\forall Blocked \in T^n_{PB}$ et $\forall \theta \in \mathbb{N}^*$, comme suit :

1. Changement d'états basé sur le tir/blocage de transitions :
   $s = (m, B, R) \overset{Fired, Blocked, \theta}{\twoheadrightarrow} s' = (m', B', R')$ ssi :
   (a) $Fired \cap Blocked = \emptyset$ (il n'y a pas de transition tirée et bloquée au même instant).
   (b) $\forall t \in Fired$, $t \in Firable(s) \wedge (Fired \cap Conflict(s, t) = \emptyset)$ (les transitions en conflit ne sont jamais tirées simultanément).
   (c) $\forall t \in enabled(m), \theta \leq\uparrow R(t)$ (il n'existe pas de transition qui doive être tirée à une date antérieure).
   (d) $\forall t \in Blocked$, $t \in PBlocked(s)$ (l'ensemble $Blocked$ contient toutes les transitions bloquées, i.e. des transitions potentiellement bloquées $PBlocked(s)$ à cet état)..
   (e) $m' = m - \sum_{t \in Fired} Pre(t) + \sum_{t \in Fired} Post(t)$ (mise à jour du marquage)
   (f) $\forall t' \in enabled(m')$, $R'(t') = R_s(t')$ ssi $t' \in newEnabled(m)$, sinon $R'(t') = [max(1, \downarrow R(t') - \theta), \uparrow R(t') - \theta]$ (mise à jour des intervalles résiduels de tir)
   (g) $B' = B \cup Blocked$ (mise à jour de l'ensemble des transitions bloquées : ajout de toutes les transitions nouvellement bloquées)
   (h) $B' = B - Unblocked(s, Fired)$ (déblocage de toutes les transitions désensibilisées par le tir de $Fired$)

Cette relation comporte trois cas particuliers, selon que les ensembles $Blocked$, $Fired$ et $Blocked$ sont vides ou non, que l'on peut ajouter à la sémantique pour plus de clareté :

2. Si aucune transition n'est bloquée ($Blocked = \emptyset$) : Changement d'états basé sur le tir de transitions, $s = (m, B, R) \overset{Fired,\theta}{\twoheadrightarrow} s' = (m', B', R')$, ssi :
   (a) $\forall t \in Fired$, $t \in Firable(s) \wedge (Fired \cap Conflict(s,t) = \emptyset)$
   (b) $\forall t \in enabled(m), \theta \leq\uparrow R(t)$
   (c) $m' = m - \sum_{t \in Fired} Pre(t) + \sum_{t \in Fired} Post(t)$
   (d) $\forall t \in enabled(m')$, $R'(t) = R_s(t)$ ssi $t \in newEnabled(m)$, sinon $R'(t) = [max(1, \downarrow R(t) - \theta), \uparrow R(t) - \theta]$
   (e) $B' = B - Unblocked(s, Fired)$
3. Si aucune transition n'est tirée alors seulement le blocage d'un ensemble de transition est possible ($Fired = \emptyset$) : Changement d'états basé sur le blocage de transitions, $s = (m, B, R) \overset{Blocked,\theta}{\twoheadrightarrow} s' = (m, B', R')$, ssi :
   (a) $\forall t \in Blocked$, $t \in PBlocked(s) \wedge \theta =\uparrow R(t)$.
   (b) $B' = B \cup Blocked$
   (c) $\forall t \in enabled(m) \wedge t \notin Blocked, R'(t) = R(t) - \theta$
4. Si aucune transition n'est tirée et aucune transition n'est bloquée ($Fired = \emptyset \wedge Blocked = \emptyset$) alors seulement une évolution temporelle est possible : Changement d'états basé sur l'évolution de temps, $s = (m, B, R) \overset{\theta}{\twoheadrightarrow} s' = (m, B, R')$, ssi :
   (a) $\forall t \in enabled(m), \theta <\uparrow R(t)$
   (b) $\forall t \in enabled(m), R'(t) = R(t) - \theta$

De façon moins formelle, on peut expliquer la relation de changement d'états de cette sémantique ainsi :

1. L'évolution d'états basée sur le tir/blocage : ce changement d'état définit la relation de changement d'état de la sémantique de STPNB. Le tir et le blocage simultanés de deux ensembles de transitions à partir d'un état s conduit vers un nouvel état s', où le marquage et les intervalles résiduels de tir sont mis à jour. Les nouvelles transitions bloquées sont ajoutées à B et les transitions débloquées sont ôtées de B.

2. Le changement d'états basé sur le tir seul : ce cas particulier est possible si aucune transition n'est bloquée ($Blocked = \emptyset$) à l'état courant après un temps θ à partir d'un état s. Ce tir conduit à un nouvel état s', où le marquage et les intervalles résiduels de tir sont mis à jour. Les transitions bloquées dans s et désensibilisées par ce tir sont débloquées dans s'.
3. Le changement d'états basé sur le blocage : ce changement d'états représente le cas du blocage d'un ensemble de transition alors qu'aucune transition n'est tirée à partir d'un état s. Ce blocage conduit à un nouvel état s', où juste les intervalles résiduels de tir sont mis à jour et où toutes les nouvelles transitions bloquées sont ajoutées à B.
4. Le changement d'états basé sur l'évolution de temps : si aucune transition n'est obligatoirement tirable ni potentiellement bloquée à partir d'un état, alors un changement d'états possible est une évolution de temps θ. L'évolution de temps à partir d'un état s conduit à un nouvel état s' où seuls les intervalles résiduels de tir sont mis à jour.

4.2 Le graphe de comportement synchrone avec blocage

Le graphe de comportement synchrone avec blocage (*Synchronous Behavior Graph with Blocking*) d'un modèle STPNB est noté SBGB. C'est une extension du SBG afin de représenter les évolutions liées au blocage de certaines transitions.

Un état du SBGB est noté $e = (m, B, R)$, avec m le marquage, B l'ensemble de transitions bloquées et $R : enabled(m) \rightarrow I^*$ la fonction des intervalles de temps résiduels pour le tir des transitions sensibilisées par m. L'état initial est $e_0 = (m_0, B_0, R_0)$ avec m_0 le marquage initial, $B_0 = \emptyset$ et $R_0(t) = R_s(t)$, $\forall t \in enabled(m_0)$.

4.2.1 Les ensembles de transitions tirables / potentiellement bloquées

Dans le cas du modèle STPNB le changement d'états basé sur le blocage doit être pris en compte dans la génération du graphe. Ainsi, les ensembles de transitions tirables (ETT) pour le STPN sont adaptées pour les transitions à blocage potentiel. Comme les transitions potentiellement bloquées ne peuvent être bloquées qu'à leur date de tir au plus tard, les seuls ensembles qui doivent être modifiés par rapport à leur définition sans blocage (section 3.2.1) sont NFT et LFT, qui deviennent respectivement $NFBT$ et $LFBT$. Ainsi, pour un état $e = (m, B, R)$, les ensembles de transitions tirables / potentiellement bloquées (ETTB) sont définis comme suit :

1. Transitions Nécessairement Tirées/Bloquées (*Necessarily Fired/Blocked Transitions* - $NFBT$) :
   — $1.1FB(e)$ est l'ensemble des transitions qui doivent être immédiatement tirées ou bloquées après 1 unité de temps, $t \in 1.1FB(e)$ ssi :

$$t \in enabled(m) \wedge R(t) = [1, 1]$$

2. Transitions Potentiellement tirées (*Probably Firable Transitions* - PFT) :
   — $1.bF(e)$ est l'ensemble des transitions qui peuvent être tirées au moins après 1 *ut* et au plus tard après b unités de temps, $t \in 1.bF(e)$ ssi :

$$t \in enabled(m) \wedge R(t) = [1, b[, b \in \mathbb{N}^* - \{+\infty\}, b \neq 1$$

   — $1.\infty F(e)$ est l'ensemble des transitions qui peuvent être tirées au moins après 1 *ut*, mais qui au pire peuvent ne jamais être tirées, $t \in 1.\infty F(e)$ ssi :

$$t \in enabled(m) \wedge R(t) = [1, +\infty[$$

3. Transitions Tirables/ Potentiellement bloquées Plus Tard (*Later Firable/Blocked Transitions* - $LFBT$) :
   — $a.aFB(e)$ est l'ensemble des transitions qui doivent être tirées ou bloquées après a *ut*, avec a le plus petit temps résiduel supérieur à

1 des transitions de cet ensemble, $t \in a.aFB(e)$ ssi :

$$t \in enabled(m), R(t) = [a, a] \wedge$$

$$\nexists t' \in enabled(m) \mid R(t') = [b, b], b \neq 1 \wedge b < a$$

4. Transitions Tirables dans un Intervalle (*Interval Firable Transitions - IFT*) :
   — $c.dF(e)$ est l'ensemble des transitions qui peuvent être tirées au moins après c ut et au plus tard après d ut, avec c le plus petit temps résiduel supérieur à 1 des transitions de cet ensemble, $t \in c.dF(e)$ ssi :

$$t \in enabled(m), R(t) = [c, d] \wedge$$

$$\nexists t' \in enabled(m) \mid R(t') = [j, k], c \neq 1 \wedge j < c$$

4.2.2 La sémantique du SBGB

La sémantique du SBGB est similaire à celle des STPNB, dans laquelle les transitions de blocage sont gérées. Comme pour le SBG, afin d'éviter l'évolution de temps indéfini/infini et de réduire son nombre d'états, le changement d'états basé sur l'évolution indéfinie/infinie du temps et le changement d'états basé sur une propagation du temps, définis dans la section 3.1.1, sont aussi considérés pour la sémantique SBGB. La différence entre les états de SBG et SBGB est la présence de l'ensemble de transitions bloquées B. Or, comme ces changements d'états n'ont aucun impact sur l'ensemble des transitions bloquées B, alors aucune adaptation n'est nécessaire (i.e., repris tels quels pour le SBGB).

4.2.3 Algorithme de construction du SBGB

L'algorithme de construction du graphe SBGB (algorithme 4) est une extension de l'algorithme de construction du graphe SBG (algorithme 1). L'entrée est l'état initial e_0. Tous les ensembles de transitions tirées/bloquées sont calculés (ETTB(s)) à partir des transitions sensibilisées à cet état. Les combinaisons des ETTB permettent de sélectionner les transitions tirées et/ou bloquées pour chaque changement d'états possible à partir de l'état d'entrée, générant ainsi à de nouveaux états. Puis, l'algorithme boucle récursivement sur les nouveaux

états jusqu'à ce qu'il n'existe plus aucune transition sensibilisée ou que tous les nouveaux états existent déjà dans le SBGB.

Plus précisément, la gestion du blocage dans l'algorithme consiste premièrement à vérifier s'il existe des transitions qui peuvent être bloquées (ensemble $PBlocked(s)$). Les transitions appartenant à cet ensemble seront soit bloquées, soit tirées. Ainsi, si cet ensemble n'est pas vide, alors les blocages probables sont exprimés avec la fonction $PowerSet$. Par exemple, prenons le sous-cas (2.1) de l'algorithme 4. Ce cas correspond à la combinaison ETTB où $NTFB \neq \emptyset$, $PFT \neq \emptyset$ et $PBlocked(s) \neq \emptyset$. Pour exprimer les tirs probables et les blocages probables, deux boucles imbriquées sont utilisées afin de parcourir toutes les possibilités de $PowerSet(PFT)$ et $PowerSet(PBlocked(s))$. À chaque itération, les transitions qui vont être tirées et celles qui vont être bloquées sont séparées (lignes 9 et 10), aucune transition ne pouvant être tirée et bloquée au même instant ($Fired \cap Blocked = \emptyset$). Ainsi, toutes les combinaisons des transitions tirées $Fired$, des transitions bloquées $Blocked$ et du temps θ sont calculés, chacune conduisant vers un nouvel état. Le marquage, l'ensemble des transitions bloquées et l'intervalle résiduel des états successeurs sont alors calculés en respectant la sémantique du STPNB.

La possibilité de blocage est exprimée par $PowerSet$ sur l'ensemble des transitions potentiellement bloquées ($PBlocked(e)$). Pour chaque sous-ensemble, les transitions qui vont être tirées et celles qui vont être bloquées sont séparées. Ces transitions sont tirées/bloquées simultanément après une unité de temps, conduisant vers un nouvel état.

La preuve de correction de l'algorithme de construction SBGB est presque la même que celle présentée dans le chapitre 3, dans la mesure où la seule différence est la présence d'une structure de boucles imbriquées qui sert à énumérer les blocages potentiels (e.g. algorithme 4 ligne 8). Or, cette boucle est finie, son compteur étant compris entre 0 et la cardinalité de l'ensemble $PowerSet(PBlocked(e))$. Intuitivement, on peut dire qu'elle augmente la complexité de l'algorithme, mais cela n'affecte pas sa correction. Ainsi, notre algorithme traite toutes les combinaisons de l'ensemble des transitions tirables/potentiellement bloquées (cf. tableau 4.2.3).

Algorithme 4 : Algorithme de construction du SBGB - Partie 1

Data : $e_0 = (m_0, B_0, R_0)$
Result : SBG

```
1  Algorithm SBGB(e)
       ETTB(e);
       /* Calcule des ensembles de transitions tirables/ potentiellement
          bloquées                                                    */
2      if 1.1FB(e) ≠ ∅ then                                  // cas (1)
3          e -1.1FB(e),θ=1->> e' ;   SBGB(e') ;
4          if 1.bF(e) ≠ ∅ ∨ 1.∞F(e) ≠ ∅ then                 // cas (2)
5              if PBlocked(e) ≠ ∅ then                       // sous-cas (2.1)
6                  for (W ∈ PowerSet(1.bF(e) ∪ 1.∞F(e))) do
7                      for (Blocked ∈ PowerSet(PBlocked(e))) do
8                          Fired ← 1.1FB(e) \ Blocked ;
9                          e -Fired∪W,Blocked,θ=1->> e' ;
10                         SBGB(e') ;
11                     end
12                 end
13             else                                          // sous-cas (2.2)
14                 for (W ∈ PowerSet(1.bF(e) ∪ 1.∞F(e))) do
15                     e -1.1FB(e)∪W,θ=1->> e' ;   SBGB(e') ;
16                 end
17             end
18         else                                              // cas (3)
19             if PBlocked(e) ≠ ∅ then
20                 for (Blocked ∈ PowerSet(PBlocked(e))) do
21                     Fired ← 1.1FB(e) \ Blocked ;
22                     e -Fired,Blocked,θ=1->> e' ; SBGB(e') ;
23                 end
24             end
25         end
26     else
           /* Voir Algorithme 5                                  */
27     end
```

Algorithme 5 : Algorithme de construction du SBGB - Partie 2

```
1  if 1.bF(e) ≠ ∅ ∨ 1.∞F(e) ≠ ∅ then                                  // cas (4)
2      for ((W ∈ PowerSet(1.bF(e) ∪ 1.∞F(e))) do
3          e ⇝^{W,θ=1} e' ;        SBGB(e')
4      end
5      if 1.bF(e) = ∅ ∧ a.aFB(e) = ∅ ∧ c.dF(e) = ∅ then // sous-cas (4.1)
6          e ⇝^{θ=1} e
7      else
8          e ⇝^{θ=1} e' ;        SBGB(e') ;
9      end
10 else
11     if a.aFB(e) ≠ ∅ ∧ c.dF(e) ≠ ∅ then                              // cas (5)
           /* Voir Algorithme 6                                              */
12     else
13         if a.aFB(e) ≠ ∅ then                                        // cas (6)
14             e ⇝^{a.aFB(e),θ=a} e' ;        SBGB(e') ;
15             if PBlocked(e) ≠ ∅ then                  // sous-cas (6.1)
16                 for (Blocked ∈ PowerSet(PBlocked(e))) do
17                     Fired ← a.aFB(e) \ Blocked ;
18                     e ⇝^{Fired,Blocked,θ=a} e' ;
19                     SBGB(e') ;
20                 end
21             end
22         end
23         if c.dF(e) ≠ ∅ then                                         // cas (7)
24             e ⇝^{θ=c−1} e' ;        SBGB(e') ;
25         end
26     end
27 end
```

Algorithme 6 : Algorithme de construction du SBGB - Partie 3

```
1  if a < c then                                                    // sous-cas (5.1)
2      e -a.aFB(e),θ=a->> e' ; SBGB(e') ;
3      if PBlocked(e) ≠ ∅ then
4          for (Blocked ∈ PowerSet(PBlocked(e))) do
5              Fired ← a.aFB(e) \ Blocked ;
6              e -Fired,Blocked,θ=a->> e' ;
7              SBGB(e') ;
8          end
9      end
10 else
11     if a > c then                                                // sous-cas (5.2)
12         e -θ=c-1->> e' ; SBGB(e') ;
13     else                                                         // sous-cas (5.3)
14         e -a.aFB(e),θ=a->> e' ; SBGB(e')
15         if PBlocked(e) ≠ ∅ then
16             for (Blocked ∈ PowerSet(PBlocked(e))) do
17                 Fired ← a.aFB(e) \ Blocked ;
18                 e -Fired,Blocked,θ=a->> e' ;
19                 SBGB(e') ;
20             end
21         end
22         for (W ∈ PowerSet(c.dF(e))) do
23             if PBlocked(e) ≠ ∅ then
24                 for (Blocked ∈ PowerSet(PBlocked(e))) do
25                     Fired ← a.aFB(e) \ Blocked ;
26                     e -Fired∪W,Blocked,θ=a->> e' ;
27                     SBGB(e') ;
28                 end
29             else
30                 e -a.aFB(e)∪W,θ=a->> e' ; SBGB(e') ;
31             end
32         end
33     end
34 end
```

TABLE 4.1 – Les différentes combinaisons de transitions traitées par l'algorithme de construction du SBGB

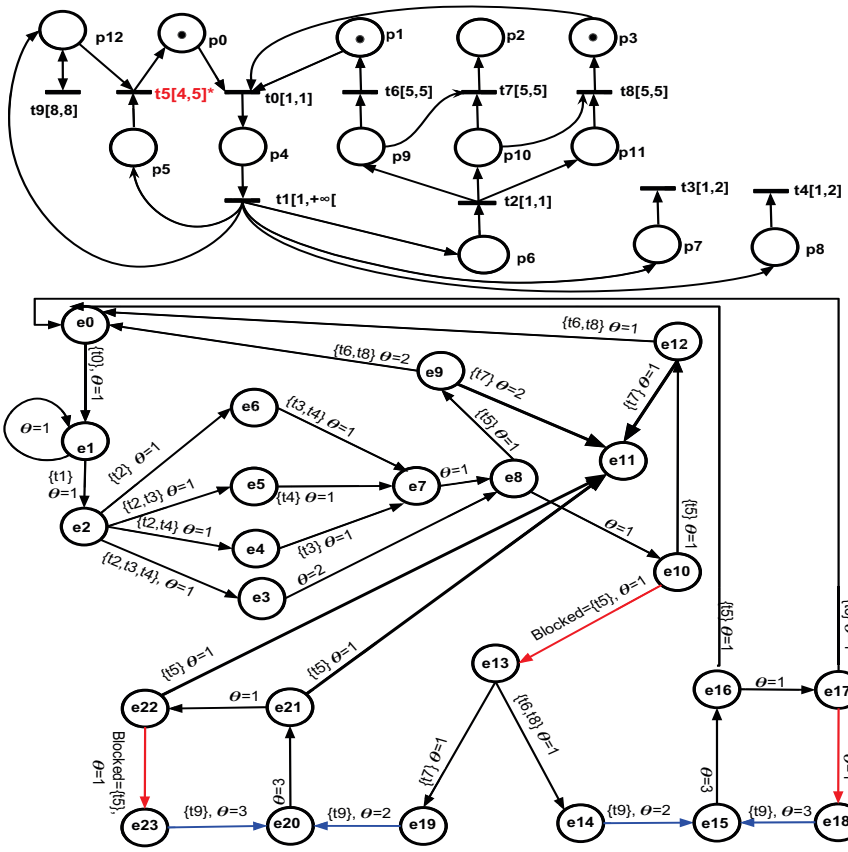
	$[1,1]$	$[1,b]$	$[1,\infty]$	Pblocked	$[a,a]$	$[c,d]$	a vs c
cas 1	O	-	-	-	-	-	
cas 2	O	O	-	-	-	-	
cas 2.1	O	l'un des 2 = O		O	-	-	
cas 2.2	O	l'un des 2 = O		N	-	-	
cas 3	O	N	N	O	-	-	
cas 4	N	l'un des 2 = O		-	-	-	
cas 4.1	N	l'un des 2 = O		N	N	-	
cas 5	N	N	N	-	O	O	-
cas 5.1	N	N	N	O	O	O	a < c
cas 5.2	N	N	N	-	O	O	c < a
cas 5.3	N	N	N	O/N	O	O	a = c
cas 6	N	N	N	-	O	N	
cas 6.1	N	N	N	O	O	N	
cas 7	N	N	N	-	N	O	
arrêt	N	N	N	N	N	N	

Illustration du blocage de transitions

Pour illustrer la construction d'un SBGB, considérons le STPNB de la figure 4.1. La seule transition potentiellement bloquée est distinguée en rouge ($T_{PB} = \{t_5\}$). Nous nous focaliserons sur le blocage et donc nous n'allons pas décrire la construction de tous les états, puisque l'évolution du graphe à partir l'état initial e_0 jusqu'à l'état e_{10} est similaire à l'évolution décrite dans l'exemple 3.2. Dans l'état e_{10}, la transition t_5 appartient à $NFBT$ et elle peut être soit tirée, soit bloquée à cet état ($t_5 \in PBlocked(e_{10})$). Dès lors, deux changements d'états sont possibles : le tir de t_5 qui conduit à l'état e_{12}, comme pour le SBG, ou bien le blocage de t_5 qui conduit à l'état e_{13}, ces deux cas étant en 1 *ut* ($\theta = 1$). Ce cas est géré dans l'algorithme 4 cas (1) et (3).

Illustration du déblocage de transition

Après le blocage de t_5 dans l'état e_{13}, le tir des transitions t_6 et t_8 simultanément avec $\theta = 1$ mène à l'état e_{14}. Le tir de la seule transition sensibilisée à cet état, i.e. t_9, permet alors le déblocage de t_5 (on a donc $Unblocked(e_{14}, \{t_9\}) = t_5$), ce qui conduit à l'état e_{15}. A partir de ce nouvel état e_{15}, la transition t_5 est de nouveau sensibilisée.



e0: p0,p1,p3, **B{}**,{t0[1,1]}	**e5**:p5,p8,p9,p10,p11, p12,**B{}**,{t4[1,1],t5[3,4] ,t6[5,5],t7[5,5],t8[5,5], t9[7,7]}	**e10**:p5,p9,p10, p11,p12,B{}, {t5[1,1],t6[2,2] ,t7[2,2],t8[2,2], t9[4,4]}	**e15**:p1,p3,p5 ,p12,**B{},** {t5[4,5],t9[8,8]}	**e20**:p2,p5,p11, p12,**B{}**, {t5[4,5],t9[8,8]}
e1:p4,B{},{t1[1,+∞[}	**e6**:p5,p7,p8,p9,p10,p11, p12,**B{}**,{t3[1,1],t4[1,1],t 5[3,4],t6[5,5],t7[5,5],t8[5, 5],t9[7,7]}	**e11**: p0,p2,p11	**e16**:p1,p3,p5,p 12,**B{}**,{t5[1,2], t9[5,5]}	**e21**:p2,p5,p11, p12,**B{},** {t5[1,2],t9[5,5]}
e2:p5,p6,p7,p8,p12, **B{}**,{t2[1,1],t3[1,2], t4[1,2],t5[4,5],t9[8,8]}	**e7**:p5,p9,p10,p11,p12, B{},{t5[2,3],t6[4,4],t7[4,4] ,t8[4,4],t9[6,6]}	**e12**:p0,p9,p10,p1 1,**B{},**{t6[1,1], t7[1,1],t8[1,1]}	**e17**:p1,p3,p5,p 12,**B{},**{t5[1,1], t9[4,4]}	**e22**:p2,p5,p11, p12,**B{}**,{t5[1,1],t9[4,4]}
e3:p5,p9,p10,p11,p12 ,**B{}**,{t5[3,4],t6[5,5], t7[5,5],t8[5,5],t9[7,7]}	**e8**: p5,p9,p10,p11,p12, **B{}**,{t5[1,2],t6[3,3], t7[3,3],t8[3,3],t9[5,5]}	**e13**:p5,p9,p10, p11,p12,**B{t5}**, {t6[1,1],t7[1,1], t8[1,1],**t9[3,3]**}	**e18**:p1,p3,p5,p 12,**B{t5}**, {**t9[3,3]**}	**e23**:p2,p5,p11, p12,**B{t5}**, {**t9[3,3]**}
e4:p5,p7,p9,p10,p11, p12,**B{},**{t3[1,1],t5[3,4] ,t6[5,5],t7[5,5],t8[5,5] ,t9[7,7]}	**e9**:p0,p9,p10,p11,**B{}**, {t6[2,2],t7[2,2],t8[2,2]}	**e14**:p1,p3,p5,p12, **B{t5}**, {**t9[2,2]**}	**e19**:p2,p5,p11, p12,**B{t5}**, {**t9[2,2]**}	

FIGURE 4.1 – Exemple d'un modèle STPNB et son SBGB

4.2.4 Comparaison avec le SCG et l'ISG, pour les situations de blocage

Comme nous l'avons mentionné dans le chapitre 2, nous adoptons une sémantique de blocage dans le formalisme STPNB. Cette sémantique est l'impact de l'interprétation (condition) couplée à des intervalles temporels ; le principe repose sur la création d'un ensemble spécifique de transitions, les transitions potentiellement bloquées (elles sont identifiées et définies dans l'ensemble T_{PB} de notre STPNB). Cet ensemble est ensuite exploité lors de la construction du graphe SBGB qui permet de représenter tous les états possibles, y compris ceux incluant des transitions bloquées. Par contre, les approches ISG et SCG ne relèvent pas du même formalisme, mais des réseaux de Petri temporels classiques qui ne permettent pas le représentation du blocage des transitions. Donc si l'on veut étudier le blocage potentiel de transitions via ces approches, la seule solution consiste à introduire (représenter) structurellement la situation tel que cela est décrit sur la figure 4.2. Cette approche a été adoptée dans [63] qui propose une méthode de validation asynchrone des modèles STPNB. Ceci étant, cette transformation consistant à ajouter des éléments de RdP (places, transitions et arcs) tant pour traiter du blocage que du déblocage, d'une part augmente la complexité du modèle à analyser et d'autre part induit l'existence de places et d'états non réels sur la cible d'exécution, ce qui est un problème dans notre contexte.

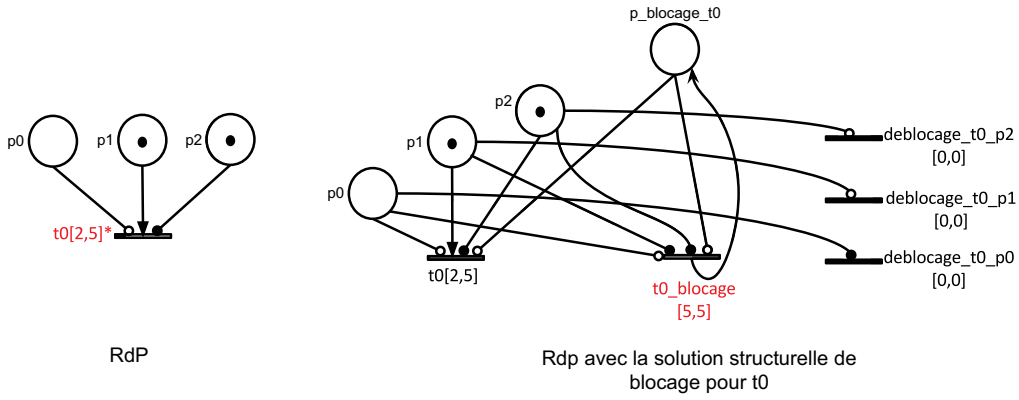


FIGURE 4.2 – Représentation du blocage en RdP temporels classiques

Par conséquent, notre approche évite la solution qualifiable de "structu-

relle" mais ne permet plus une comparaison directe avec les SCG et ISG.

De fait, quand on analyse le modèle STPNB décrit sur la figure 4.3 la situation de blocage n'apparaît pas sur les graphes de comportements SCG et ISG mais seulement sur le graphe SBGB. En effet, sur le graphe SBGB, la transition t_0 est potentiellement bloquée à partir de l'état initial, c.à.d, $PBlocked(s_0) = \{t_0\}$. Ainsi, deux changements d'états sont possibles : le tir de t_0 qui conduit à s_1 ou le blocage de t_0 qui conduit à s_2. Ceci n'apparait pas sur les deux autres graphes.

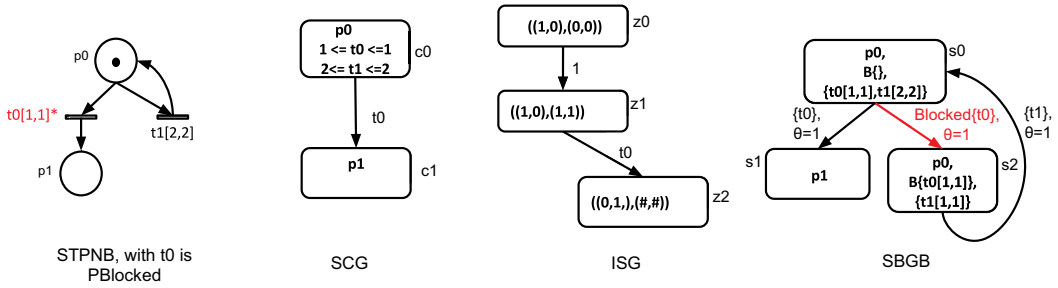


FIGURE 4.3 – Comparaison de SBGB avec SCG et ISG

Les transitions bloquées peuvent être débloquées si elles sont désensibilisées par un tir d'une ou plusieurs autre(s) transition(s) et elles seront à nouveau tirables si elles sont nouvellement sensibilisées. Considérons le même exemple (modèle STPNB de la figure 4.3) : à partir de l'état s_2 le tir de t_1 débloque la transition t_0, c.à.d, $Unblock(s_2, t_1) = \{t_0\}$, et re-sensibilise de nouveau cette transition, ce qui conduit à l'état initial. Notons d'ailleurs, que la transition t_1 n'est jamais tirée sur les graphes de comportement ISG et SCG (t_0 étant toujours tirée avant, selon la sémantique "forte" de ces approches).

En conclusion, nous pouvons désormais détecter (expliciter) les situations de blocage potentiels de transitions, induites par notre sémantique "impérative", sur le modèle analysable STPNB. Pour ce faire, nous avons associé une sémantique qui prend en considération ces blocages potentiels, au sein du modèle STPNB. Puis, nous avons complété le graphe de comportement SBGB, sa sémantique ainsi que son algorithme de construction afin de représenter explicitement un blocage potentiel d'une transition sur le graphe.

Le dernier aspect que nous devons encore prendre en considération dans

notre méthode d'analyse concerne la macroplace ainsi que les transitions d'exception. Cela fait l'objet du prochain chapitre.

Chapitre 5

Prise en compte des situations d'exception

La méthodologie HILECOP, et son outil associé, permettent d'exprimer la gestion d'exception à l'aide de macroplaces et de transitions d'exception. Rappelons aussi que ces entités sont uniquement destinées à la spécification, au sens où elles ne sont pas réifiées telles que sur la cible. En effet, une macroplace et une transition d'exception sont mises à plat avant la génération du code VHDL à implanter sur la cible, i.e. elles sont d'abord traduites en places et transitions classiques pour reconstruire un modèle équivalent.

Pour la validation formelle, la traduction en un modèle analysable ne consiste pas seulement à traduire l'ensemble des arcs d'entrée et de sortie, et les situations qui leur sont associées, mais aussi à traduire l'activité de la MP ainsi que l'impact du tir des transitions d'exception. Ce dernier point est particulièrement important dans notre contexte puisque la transition exception est traitée en partie de façon asynchrone, au contraire des autres transitions qui sont purement synchrones. Ainsi, nous souhaitons étudier l'impact des évolutions asynchrones induites par le tir des transitions d'exception telles qu'elles sont mises en oeuvre sur la cible d'exécution.

Une solution a été proposée initialement dans les travaux de Hèlene Leroux & al. [63, 60]. Elle consiste à transformer la MP vers un modèle RdP temporel avec priorités, où l'activité de la MP et le fonctionnement des transitions d'exception ont été modélisés structurellement sur le modèle obtenu (c.à.d, des transitions, places et arcs sont ajoutés au modèle). Ce modèle a ensuite été analysé avec l'outil TINA de façon asynchrone. D'un point de vue implémentation et exécution synchrone, cette solution n'est pas fidèle à la mise en oeuvre sur la cible car elle ajoute des éléments structurels de RdP (places et transitions) qui ne sont pas implémentés et qui conduisent à des états intermédiaires asynchrones inexistants ; autant de "non-conformités" entre analyse et exécution réelle.

Dans ce chapitre, nous proposons une nouvelle sémantique pour les STPN avec blocage, en intégrant l'activité de la MP et l'impact du tir des transitions d'exception, sans passer par une solution structurelle. Cette sémantique est conforme au fonctionnement sur la cible. L'algorithme de construction du comportement du nouveau modèle analysable est donné. Enfin, une illustra-

tion sur un exemple de modèle ITPN avec MP est présentée.

5.1 Mise à plat de la macroplace pour obtention du modèle analysable

Avant de présenter les transformations, rappelons que le comportement global du modèle ainsi que celui des transitions d'entrée et de sortie de la MP sont régis selon le même principe d'exécution synchrone que précédemment présenté. Dans un modèle avec macroplace, la seule différence se situe au niveau de la transition d'exception. En effet, si le tir d'une transition d'exception est évalué en synchrone, comme pour toutes les transitions, les effets de son tir sur les noeuds du raffinement de la MP sont immédiats, i.e. réalisés en asynchrone (par ex. la purge des places internes).

5.1.1 Transformation des arcs entrants et sortants de la macroplace

Pour analyser un modèle avec macroplace (MP), il est donc nécessaire de mettre à plat la MP, i.e. la transformer dans le respect de ce que fait HILECOP pour obtenir le modèle analysable équivalent au modèle implémentable.

Les arcs entrants et sortants sont mis à plat de la même façon que proposé dans [63, 60]. Nous allons résumer ici cette transformation.

1. Arcs entrants classiques : les arcs entrant de la MP sont transformés en plusieurs arcs classiques, un par élément de la situation qui lui est associée. Donc, pour une transition $t_{entrée}$ reliée à un arc entrant associée à une situation ($\{n_i P_i\}$), cet arc entrant est remplacé par un ensemble d'arcs classiques entre la transition $t_{entrée}$ et chaque place P_i indiquée dans la situation, ces arcs étant respectivement pondérés par n_i. Sur l'exemple de la figure 5.2 [1], qui montre la mise en plat de la MP du

1. Le modèle de la figure 5.2 est également le résultat des transformations du modèle ITPN de la figure 5.1 vers son modèle analysable STPN (c.f. section 2.1), d'où l'ajout des intervalles temporels $[1, 1]$ et $[1, +\infty[$.

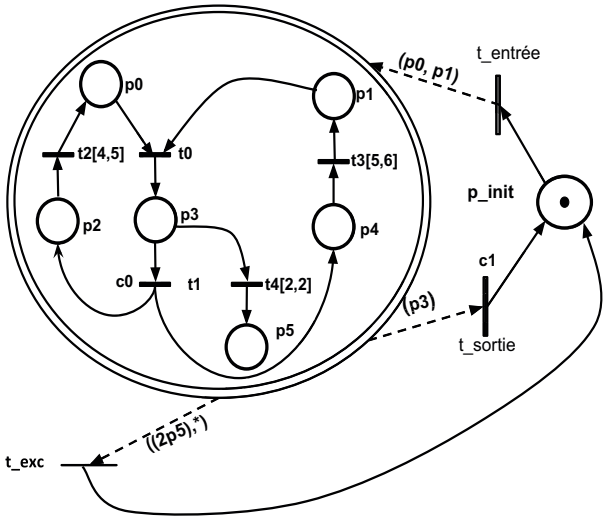


FIGURE 5.1 – Un exemple de macroplace

modèle initial de la figure 5.1, l'arc entrant de situation (p_0, p_1) est donc remplacé par 2 arcs normaux (en rouge) de poids 1 vers les places p_0 et p_1.

2. Arcs sortants classiques : Pour une transition t_{sortie} associée avec un arc sortant classique, selon la situation de sortie (Xn_iP_i), la transformation introduit un ensemble d'arcs reliant chacune des places P_i de la situation à la transition aval t_{sortie}. Le type de l'arc généré (classique, inhibiteur ou test) et le poids de l'arc sont définis à l'aide des paramètres X et n_i de la situation de l'arc sortant, comme suit :
   — si $X = \epsilon$, un arc classique est généré, pondéré par n_i,
   — si $X =? \wedge n_i > 0$, un arc test est généré, pondéré par n_i,
   — si $X =? \wedge n_i < 0$, un arc inhibiteur est généré, pondéré par n_i.
   Sur l'exemple de la figure 5.2, l'arc sortant classique de situation (p_3) est donc remplacé par un arc classique (en rouge) de poids 1, entre la place p_3 et la transition de sortie t_{sortie}.
3. Arcs d'exception de la forme $(*)$: Cet arc (sortant) sensibilise sa transition aval (la transition d'exception) ssi sa MP amont est active, c.à.d qu'il existe au moins une place du raffinement de cette MP qui est marquée. Au stade de la modélisation (graphique), la présence d'un arc d'exception de ce type sert uniquement à identifier qu'une transition d'exception est liée une MP (i.e. cela permet d'identifier cette transi-

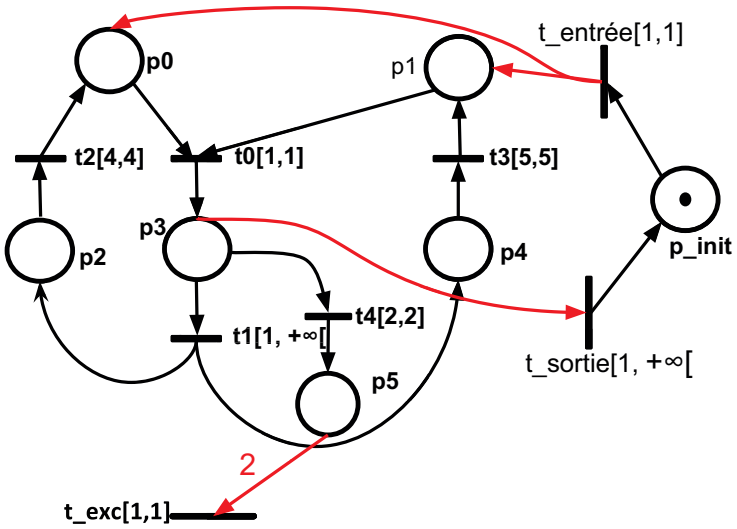


FIGURE 5.2 – Transformation des arcs entrants et sortants

tion comme une transition d'exception). De fait, cette transition n'est directement liée à aucune place du raffinement de la MP de façon structurelle, mais à l'ensemble de la MP.

La gestion d'une transition exception peut se décomposer en plusieurs éléments : la sensibilisation, la décision de tir et l'effet du tir (ici, principalement la purge des places du raffinement de la MP concernée). Dans le modèle implémenté, cette gestion est effectuée à l'aide de signaux et de variables internes, sans ajout de place ou de transition supplémentaires. Nous ne faisons donc pas de mise à plat correspondant aux arcs exception : les éléments liés au comportement de cette transition (sensibilisation et tir) seront considérés plus globalement au sein de la nouvelle sémantique des STPNB avec exception.

4. Arcs d'exception sortant de la forme $(s_{exc}, *)$: Un arc associé avec une telle situation est traité comme un arc sortant classique plus un arc d'exception. En effet, la situation $(s_{exc}, *)$ est composée de deux parties : 1/ la partie s_{exc} est considérée comme un arc sortant classique, 2/ la partie $(*)$ est traitée comme un arc d'exception. Dans l'exemple des figures 5.1 et 5.2, la situation $(2p_5, *)$ est remplacée par un arc sortant classique pondéré de poids 2 (en rouge) entre la place p_5 et la transition t_{exc}. L'arc d'exception n'est quant à lui pas traduit structurellement comme nous l'avons souligné, mais sera considéré dans la sémantique

de tir des transitions d'exception.

Nous allons maintenant aborder en détails la gestion du tir des transitions d'exception.

5.1.2 Analyse des transitions d'exception et de leur impact

La sensibilisation d'une transition d'exception dépend de l'activité des MP auxquelles elle est liée. Or la MP n'est pas réifiée sur la cible d'exécution : elle n'existe pas dans le modèle implémenté. Il est donc nécessaire d'analyser en détails l'évolution du modèle, au sens notamment de l'activation / désactivation de la MP.

Activité de la MP

L'activité de la MP était initialement mise à plat dans [63]. L'idée était de rajouter une structure RdP au modèle mis à plat afin de mémoriser l'état de la MP (active/inactive). Cette méthode est résumée dans cette section.

— Activation : L'activation de la MP est représentée à l'aide de deux places ajoutées au modèle initial : MP *active* et *demande activation*. Pour chaque transition qui peut générer un jeton dans le raffinement (transition entrante ou transition interne créatrice de jeton), un arc classique est ajouté de cette transition vers la place *demande activation*. L'ajout de deux transitions ta_1, ta_2 permet alors la gestion de la demande d'activation, respectivement si la MP est déjà active ou si elle ne l'est pas encore. Ces deux places et ces deux transitions ne sont ajoutées que pour l'analyse : elles ne sont pas implémentées sur la cible d'exécution. Aussi, afin d'assurer la cohérence du modèle analysé avec l'implémentation sur la cible pour laquelle l'activation est calculée immédiatement de manière asynchrone, ces transitions sont tirées immédiatement en temps nul (leurs intervalles temporels sont à $[0, 0]$).
— Désactivation : Si toutes les places du raffinement sont non marquées, alors la MP est désactivée. Il faut alors, et en temps nul pour la même

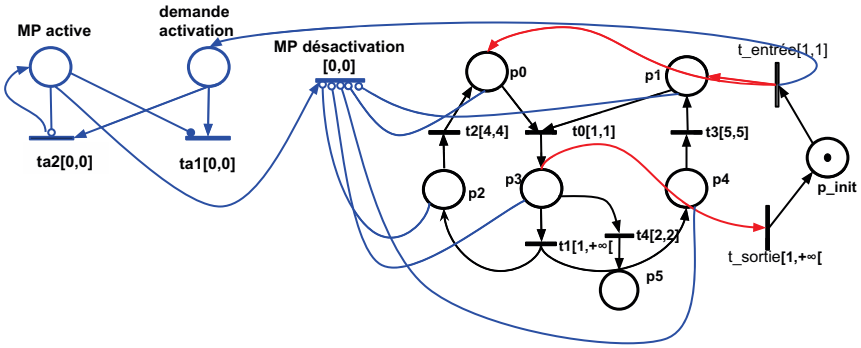


FIGURE 5.3 – Modélisation de l'activité de la macroplace

raison, supprimer le jeton de la place *MP active*. Pour cela la transition *MP désactivation* est ajoutée au modèle. Cette transition est reliée avec un arc inhibiteur à chacune des places du raffinement, et avec un arc classique à la place *MP active*.

La structure rajoutée pour gérer l'activation et la désactivation de la macroplace est présentée en bleu sur la figure 5.3. Cette structure est donc rajoutée au modèle analysable, pour chaque MP, afin de simuler son activité, dans le but de déterminer la sensibilisation des transitions d'exception liées à la MP.

Purge de la MP

Le tir d'une transition d'exception entraîne la purge des places de la MP (toutes les places du raffinement sont vidées). Pour être conforme à l'implémentation, la purge étant traitée en asynchrone et donc instantanée, sa mise à plat dans le modèle analysable doit correspondre à une purge en temps nul. Selon l'approche d'analyse proposée dans [63], la purge est traduite par une structure RdP qui vide toutes les places du raffinement. Deux solutions structurelles sont possibles pour faire une purge : en parallèle ou en séquence.

— Purge en parallèle : La structure représentée en bleu dans la figure 5.4 est ajoutée au modèle afin de vider l'ensemble des places du raffinement en parallèle. Si la transition d'exception est tirée, alors ce tir va générer un jeton dans la place *Purge en cours*, qui est elle-même reliée à 5 transitions tp_i servant à vider respectivement les places p_i du raffinement. Lorsque tous les jetons ont été vidés, la transition *t_ end* devient

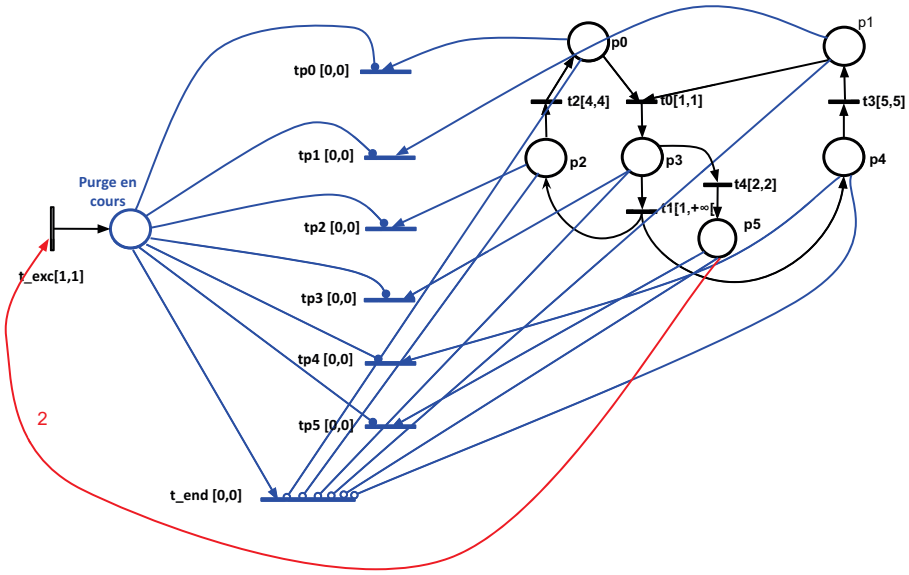


FIGURE 5.4 – Structure de la purge en parallèle

tirable. Ainsi toutes les places de la MP seront vidées "simultanément" (les tirs étant en temps nul) et "en parallèle", les transitions tp_i étant tirables en parallèle.

— purge en séquence : La purge en séquence est représentée en bleu dans la figure 5.5. Après le tir de la transition exception, toutes les places sont vidées une par une en séquence. Cette purge s'effectue également en temps nul (l'ensemble des transitions étant tirées en temps nul)

Le modèle final obtenu par la mise à plat de l'activité et de la purge de la MP, pour le cas parallèle, est représenté dans la figure 5.6. Une fois que la transition d'exception est tirée, alors la MP est indiquée inactive en enlevant le jeton de la place $MP\ active$.

Cependant ces deux solutions (en parallèle ou en séquence) introduisent un écart entre l'implémentation et l'analyse. En effet, comme nous l'avons déjà souligné, ces solutions ajoutent des transitions, des places et des arcs qui ne sont pas existants sur la cible. Cela induit nécessairement, au niveau du modèle analysable, l'existence d'états et de changements d'états non existants dans la réalité. De plus, nous devons prendre en compte à la fois l'évolution synchrone du modèle et l'effet du tir des transitions d'exception qui sur la

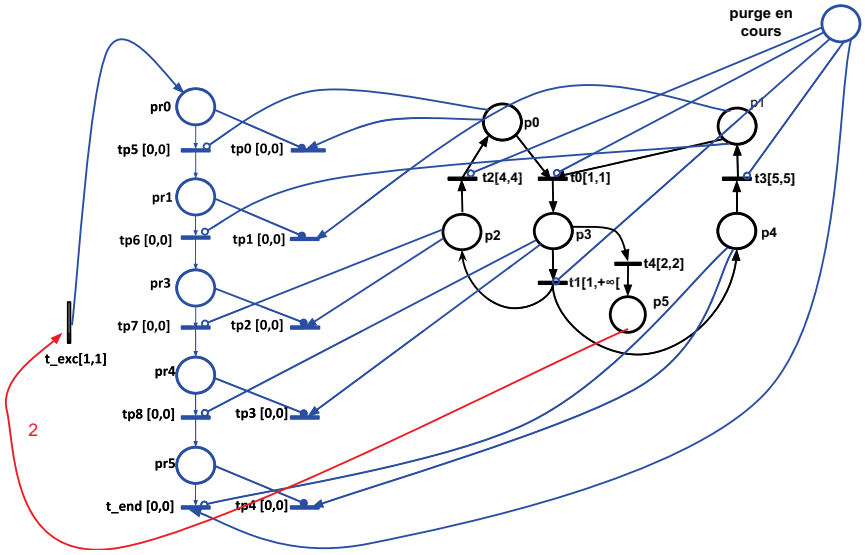


FIGURE 5.5 – Structure de la purge en séquence

cible, rappelons-le, est réalisé en asynchrone immédiatement lors du tir de la transition d'exception tel que cela est représenté sur la figure 5.7. Certes, les transitions ajoutées ont un intervalle nul ($[0,0]$) mais chaque tir suppose une actualisation du marquage ; or, ce marquage évolue normalement selon le principe d'exécution synchrone. Donc un tel fonctionnement ne serait pas conforme à l'évolution réelle sur la cible, induisant encore une fois un "décalage" entre le modèle analysé et le modèle implémenté (i.e., l'évolution synchrone du marquage des places du raffinement telle que définie dans notre sémantique et la purge réelle sur la cible).

A titre d'exemple, il existe dans le SCG des états atteignables dans lesquels seule une partie des places du raffinement d'une MP a été vidée après le tir d'une transition d'exception, alors que sur la cible ce tir induit une purge simultanée de toutes les places de la MP.

La solution consistant à faire des ajouts structurels (P et T) ne convenant pas, il est nécessaire de chercher une autre solution. Nous pourrions avoir recours à un autre mécanisme pour traiter le comportement particulier de la purge lors du tir d'une transition d'exception : exploiter un type d'arc spécifique des réseaux de Petri qu'est l'arc de vidange [37].

Arc de vidange

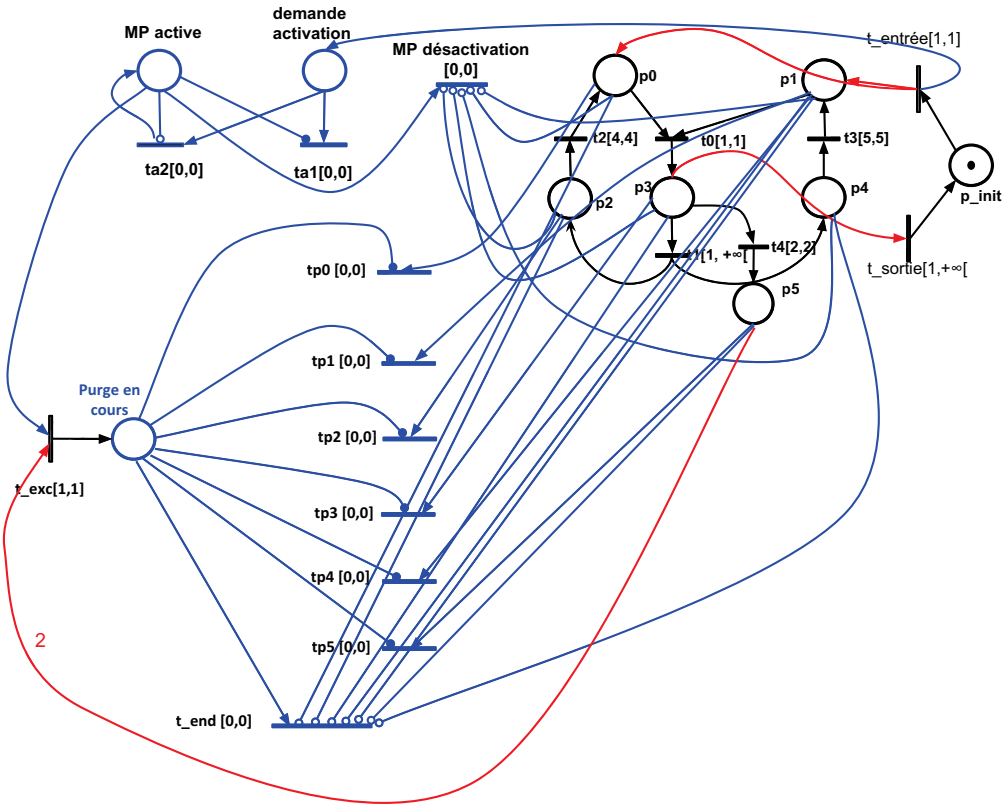


FIGURE 5.6 – Modèle analysable obtenu selon [63]

L'arc de vidange consiste à supprimer tous les jetons de sa place amont. Il relie toujours une place avec une transition. Si une transition t est liée à une place par un arc de vidange, les conditions de sensibilisation de la transition t ne sont pas modifiées par ce type d'arc. Si t est tirée, alors les jetons des places amont liées avec cette transition par un arc vidange sont supprimés. Un arc de vidange est représenté graphiquement par une flèche avec un losange à l'extrémité, tel qu'illustré sur la figure 5.8.

A priori, nous pourrions modéliser la purge de la MP en liant toutes les places du raffinement à la transition exception par un arc de vidange. Une fois la transition d'exception tirée, toutes les places du raffinement seraient alors vidées (cf. figure 5.9). Cependant, il faudrait d'une part accepter qu'une place et une transition aient plusieurs arcs les reliant dans le même sens (cf. les liens entre p_8 et t_{exc} sur la figure 5.9) et d'autre part il est tout de même nécessaire

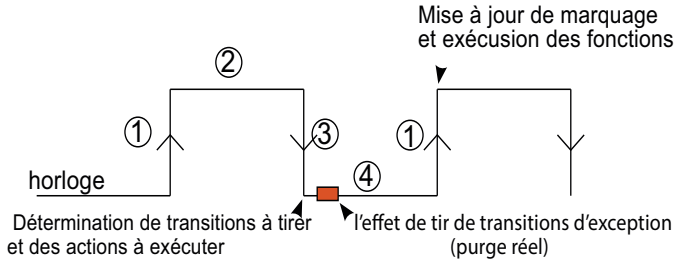


FIGURE 5.7 – Principe d'exécution de la purge

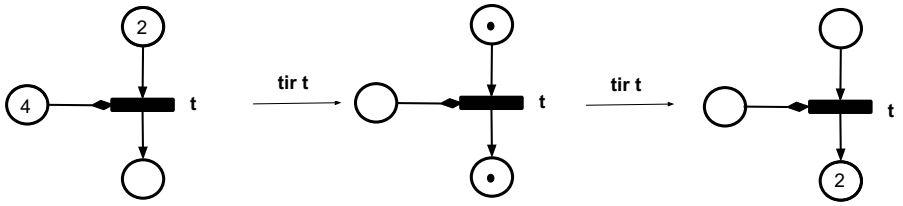


FIGURE 5.8 – Le principe de l'arc de vidange

de conserver les ajouts structurels relatifs à l'activité de la MP.

Notons par ailleurs, que ce type d'arc n'a pas été retenu pour une autre raison : pour être conforme à la réalité de l'exécution sur la cible, il aurait fallu que l'effet (la purge) du tir d'une transition exception soit immédiat afin d'obtenir la purge désirée. Or, en associant un arc de vidange à la transition exception comme montrée sur la figure 5.9, la vidange se fera de la même façon que l'actualisation du marquage de tous les autres arcs, sur le front montant suivant, et donc après au moins $1ut$.

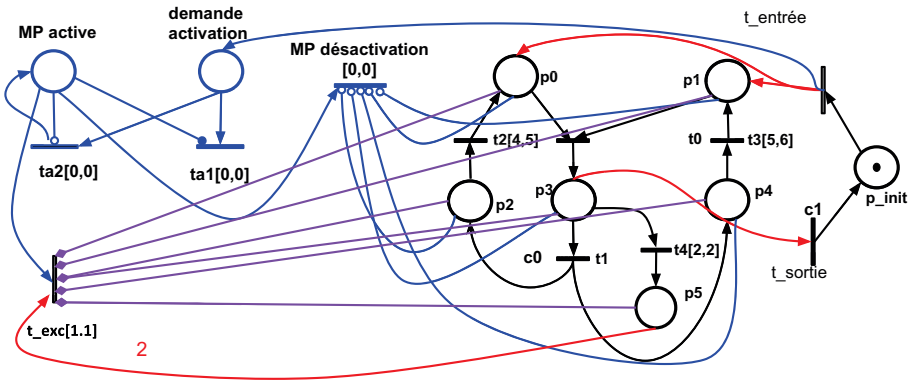


FIGURE 5.9 – Modélisation de la purge par arc vidange

Par conséquent, nous proposons un nouveau formalisme STPNB pour la gestion d'exception, avec une mise à plat des arcs entrants/sortants de la MP et une gestion intégrée du tir de transitions d'exception. Cette proposition repose sur l'intégration de l'activité de la MP et de la purge des places des raffinements, directement au sein de la sémantique du modèle analysable plutôt que d'essayer de reconstruire une équivalence structurelle.

Gestion des tirs simultanés :

— transition exception / transitions entrantes : dans ce cas notre sémantique doit, pour être conforme à la réalité, désactiver la macroplace puis la réactiver. C'est effectivement le cas sur la cible (cf. figure 5.7) : i/ sur le front descendant du signal d'horloge (3) , les transitions à tirer sont toutes sélectionnées (exception comme entrantes) ii/ sur la demi-période de signal bas (2) la purge est effectuée puis iii/ sur le front montant (1), la mise à jour du marquage liée au tir des transitions entrantes est effectuée.

— transition exception / transitions internes : dans ce cas les transitions internes ne doivent pas être tirées, car elles seront désactivées par la purge. Sur la cible, les signaux de tir des transitions internes de la macroplace sont annulés sur la demi-période de signal bas, avant que la mise à jour du marquage ait pu être faite sur le front montant suivant. Les transitions entrantes ne seront donc pas tirées.

— transition exception / transitions sortantes : de même, dans ce cas, notre sémantique doit, pour être conforme, supprimer les ordres de tir des transitions sortantes. Seules celles qui étaient tirables et qui sont reliées avec des places de la MP par un arc inhibiteur seront tirées.

5.2 Le modèle Analysable

Le modèle complet obtenu à partir de la mise à plat des arcs entrants et sortants de la MP et de la transformation du modèle ITPN principal est un réseau de Petri temporel synchrone avec transitions potentiellement bloquées et transitions d'exception (*Synchronous Time Petri Nets with Blocking and Exception Transitions*, STPNBE). La définition du formalisme STPNBE et sa sémantique sont une extension de celles du formalisme STPNB (cf. Section 4),

auquel la prise en considération de la gestion des transitions d'exception a été ajoutée.

5.2.1 Définition du formalisme STPNBE

Un STPNBE est un n-uplet $< P, T, T_{EXP}, T_{PB}, M, Pre_{exc}, Pre, Pre_i, Pre_t, Post, m_0, R_s >$, où :

- $< P, T, T_{PB}, Pre, Pre_i, Pre_t, Post, m_0, R_s >$ est un STPNB.
- $T_{EXP} \subset T$ est l'ensemble des transitions d'exception. Il contient toutes les transitions reliées aux MPs via un arc d'exception.
- M est l'ensemble des raffinements des macroplaces du modèle initial. Cet ensemble permet de préserver les ensembles des éléments (places et transitions) contenus dans les raffinements des MPs, nécessaires à la gestion du tir des exceptions. Ils ne conservent pas la structure des arcs ni le marquage, l'évolution des raffinements des MP se faisant directement grâce aux fonctions Pre et $Post$ et avec marquage du STPNBE global. Ainsi, un raffinement d'une MP, $raf \in M$ est un couple défini par $< P^{raf}, T^{raf} >$, où P^{raf} est l'ensemble des places et T^{raf} est l'ensemble de transitions qui étaient contenus dans la MP correspondant au raffinement raf.
- $Pre_{exc} : T_{EXP} \times M \to \mathbb{B}$ est la relation reliant une transition d'exception à un raffinement $raf \in M$. Cette relation est le reflet des liens entre les transitions d'exception et les macroplaces dans le modèle ITPN avec MP initial.

Le modèle analysable STPNBE est obtenu à partir du modèle initial ITPN avec MP [2] $< P^{ITPN}, T^{ITPN}, M^{ITPN}, Pre_{exc}^{ITPN}, Pre^{ITPN}, Pre_t^{ITPN}, Pre_i^{ITPN}, Post^{ITPN}, Entry^{ITPN}, Exit^{ITPN}, m_0^{ITPN}, C, F, A, Clock, I_s >$

comme suit :

- Construction de l'ensemble des transitions d'exception T_{EXP}, $t \in T_{EXP}$

2. La définition initiale des ITPN avec MP donnée dans [60] n'intègre pas directement la fonction $Pre_{exc}^{ITPN} : M^{ITPN} \times T^{ITPN} \to \mathbb{B}$ dans le n-uplet. Cependant, les auteurs l'utilisent comme tel. Nous pensons que c'est une erreur, nous l'intégrons donc pour simplifier l'écriture de notre transformation.

ssi :
$\forall t \in T^{ITPN}, \forall mp \in M^{ITPN} \mid Pre_{exc}^{ITPN}(mp, t) = 1$.

— Conservation des éléments constituant les raffinements des MP :
$\forall mp \in M^{ITPN}$, on a $raf \in M$ tel que :
$P^{raf} = P^{mp}$ et $T^{raf} = T^{mp}$, avec P^{mp} et T^{mp} respectivement l'ensemble des places et des transitions de mp.

— Conservation de la relation entre les transitions d'exception et les raffinements des MP : s'il existe un lien entre une transition d'exception et une MP, alors ce lien est conservé entre cette transition et le raffinement correspondant à cette MP. Pour simplifier, nous noterons : $Pre_{exc} = Pre_{exc}^{ITPN}$.

— Les arcs entrants et sortants de chaque macroplace mp de l'ensemble M^{ITPN} sont mis à plat en utilisant la méthode proposée dans la section 5.1.

— Le reste de la transformation du modèle ITPN complet vers un modèle STPNBE est faite selon la méthode proposée au chapitre 2.

Pour une transition d'exception $t \in T_{EXP}$, on appelle $enabled_{exc}(t) : T_{EXP} \rightarrow \mathbb{B}$, la fonction qui permet de vérifier si tous les raffinements $raf \in M$, qui correspondent aux MPs liées avec cette transition d'exception, sont marqués ou non. La fonction $enabled_{exc}(t) = 1$ si ces raffinements ont au moins une place marquée, 0 sinon. Formellement, elle est définie comme suit :

$\forall t \in T_{EXP}, \forall raf \in M \mid Pre_{exc}(t, raf) = 1, \exists p \in P^{raf}, m(p) > 0$
$\Rightarrow enabled_{exc}(t) = 1$, sinon $enabled_{exc}(t) = 0$.

Une transition d'exception t est donc sensibilisée ssi la condition classique de sensibilisation de transition est vraie et si la fonction $enabled_{exc}(t) = 1$. Ainsi, la définition de la sensibilisation de transition est adaptée comme suit : on a $t \in enabled(m)$ ssi :

$$\big[(m \geq Pre(t) + Pre_t(t)) \wedge (m < Pre_i(m))\big]$$
$$\wedge \big[(t \notin T_{EXP}) \vee (t \in T_{EXP} \wedge enabled_{exc}(t) = 1)\big]$$

Les définitions d'un état, de la fonction Firable(s) et des ensembles de transitions $Fired$, $PBlocked(s)$ et $Unblocked(s, Fired)$ sont les mêmes que pour les STPNB sans exception (cf. section 4.1.1). La définition du conflit est également la même que précédemment (cf. section 2.1.3), nous précisons juste que les transitions bloquées ne seront jamais considérées en conflit (puisqu'une transition bloquée n'est plus sensibilisée).

Afin de gérer la priorité des transitions d'exception sur les transitions internes et/ou sortantes des MPs, ainsi que pour gérer la purge, nous introduisons deux nouvelles définitions : les ensembles *Prio(t)* et *Purge(t)*.

Définition 2 *Pour une transition d'exception $t \in T_{EXP}$, $Prio(t)$ est l'ensemble de toutes les transitions qui ne pourront plus être tirées après le tir de t. Cet ensemble regroupe les transitions internes des raffinements, ainsi que les transitions sortantes non liées par un arc inhibiteur seul, pour toutes les MPs liées avec cette transition. Formellement, $\forall t \in T_{EXP}, \forall raf \in M \mid Pre_{exc}(t, raf) = 1$:*

$k \in Prio(t) \iff k \in T^{raf} \vee (\exists p \in P^{raf} \mid Pre(k,p) \neq 0 \vee Pre_t(k,p) \neq 0)$

Définition 3 *On appelle $Purge(t)$ l'ensemble des places qui seront vidées lors du tir de t. Cela correspond à toutes les places des raffinements de toutes les MPs liées avec cette transition. Formellement, $\forall t \in T_{EXP}, \forall raf \in M \mid Pre_{exc}(t, raf) = 1$:*

$p \in Purge(t) \iff p \in P^{raf}$.

5.2.2 La sémantique du modèle STPNBE

La sémantique du modèle STPNBE est présentée à travers un système de transitions étiquetées comme suit : Soit T^n l'ensemble des ensembles de transitions d'un STPNBE, et T^n_{PB} l'ensemble des ensembles de ses transitions potentiellement bloquées (i.e. transitions appartenant à T_{PB}). La sémantique des STPNBE est définie par $(S, s_0, \twoheadrightarrow)$, avec S l'ensemble des états, $s_0 = (m_0, B_0, R_0) \in S$ l'état initial et $\twoheadrightarrow \subseteq S \times (T^n \times T^n_{PB} \times \mathbb{N}^*) \times S$ est la relation de changement d'états basée sur le tir/blocage des transitions.

Lorsqu'aucune transition d'exception n'est tirable, cette relation d'état est la même que pour la sémantique du formalisme STPNB (section 4.1.2). Il

s'agit simplement de rajouter formellement la vérification qu'aucune transition d'exception n'est tirée. Ainsi, cette relation est définie comme suit : Soient $Fired$ et $Blocked$ l'ensemble des transitions tirées et l'ensemble des transitions bloquées à partir d'un état s à un instant discret non nul θ ($\theta \in \mathbb{N}^*$). $\forall Fired \in T^n$, $\forall Blocked \in T^n_{PB}$ et $\theta \in \mathbb{N}^*$, nous avons :

1. Changement d'états basé sur le tir/blocage de transitions (sans tir de transition d'exception) :
   $s = (m, B, R) \overset{Fired,Blocked,\theta}{\twoheadrightarrow} s' = (m', B', R')$ ssi :
   (a) $\forall t \in Fired$, $t \notin T_{EXP}$ (aucune transition d'exception n'est tirée).
   (b) cf. changement d'état n°1 section 4.1.2.

2. Changement d'états basé sur le tir de transitions, sans blocage et sans tir de transition d'exception :
   $s = (m, B, R) \overset{Fired,\theta}{\twoheadrightarrow} s' = (m', B, R')$ ssi :
   (a) $\forall t \in Fired$, $t \notin T_{EXP}$ (aucune transition d'exception n'est tirée).
   (b) cf. changement d'état n°2 section 4.1.2.

3. Changement d'états basé sur le blocage seul de transitions :
   $s = (m, B, R) \overset{Blocked,\theta}{\twoheadrightarrow} s' = (m, B', R')$ ssi :
   (a) cf. changement d'état n°3 section 4.1.2.

4. Changement d'états basé seulement sur l'évolution du temps :
   $s = (m, B, R) \overset{\theta}{\twoheadrightarrow} s' = (m, B, R')$ ssi :
   (a) cf. changement d'état n°4 section 4.1.2.

Nous allons maintenant définir les changements d'états contenant des tirs de transitions d'exception. Ces changements d'états sont spécifiques car ils représentent en même temps le tir des transitions exception et celui des transitions classiques. Il est important de souligner ici que pour être conforme à la réalité sur la cible, ces changements d'états doivent prendre en compte l'effet asynchrone du tir des transitions exception, c'est-à-dire que l'effet de leur tir doit être considéré avant celui des autres transitions. Concrètement, cela revient à considérer la purge, et donc la désensibilisation potentielle des transitions internes et sortantes de la macroplace. Ainsi, les transitions tirables

mais appartenant à l'ensemble $Prio(t)$ ne seront pas tirées si la transition d'exception t est tirée. De plus, si une transition d'exception est tirée, alors elle annule le marquage antérieur de tous les raffinements des MPs liées avec cette transition.

5. Changement d'états basé sur le tir/blocage de transitions avec tirs de transitions d'exception :
   $s = (m, B, R) \overset{Fired, Blocked, \theta}{\twoheadrightarrow} s' = (m', B', R')$ ssi :
   (a) $\forall t \in Fired,\ t \in Firable(s) \wedge (Fired \cap Conflict(s,t) = \emptyset)$ (les transitions en conflit ne sont jamais tirées simultanément).
   (b) $\exists t \in Fired \wedge t \in T_{EXP}$ (il existe au moins une transition d'exception dans l'ensemble $Fired$).
   (c) $\forall t \in Fired \wedge t \in T_{EXP} \mid \forall k \in Prio(t), \nexists k \in Fired$ (effet de la purge : toutes les transitions désensibilisées par la purge exécutée par le tir de la transition d'exception t, ne sont pas tirées).
   (d) $Fired \cap Blocked = \emptyset$ (il n'y a pas de transition tirée et bloquée au même instant).
   (e) $\forall t \in enabled(m), \theta \leq \uparrow R(t)$ (il n'existe pas une transition qui doit être tirée à une date antérieure).
   (f) $\forall t \in Blocked,\ t \in PBlocked(s)$ (les transitions bloquées (i.e. appartenant à $Blocked$) sont forcément des transitions potentiellement bloquées de cet état (i.e. appartenant à $PBlocked(s)$).
   (g) $\forall t \in Fired \wedge t \in T_{EXP} \mid \forall p \in Purge(t), m'(p) = 0 + \sum_{k \in Fired} Post(k)$ (mise à jour du marquage des places internes des raffinements : purge, i.e $m'(p) = 0$, plus la mise à jour du marquage liée aux tirs des transitions entrantes[3], i.e. $+ \sum_{k \in Fired} Post(k)$).
   (h) $\forall t \in Fired \wedge t \in T_{EXP} \mid \forall p \in P \backslash Purge(t), m'(p) = m - \sum_{k \in Fired} Pre(k) + \sum_{k \in Fired} Post(k)$ (mise à jour du marquage des autres places).
   (i) $\forall t' \in enabled(m'),\ R'(t') = R_s(t')$ ssi $t' \in newEnabled(m)$, sinon $R'(t') = [max(1, \downarrow R(t') - \theta), \uparrow R(t') - \theta]$ (mise à jour des intervalles résiduels de tir).

3. Seules les transitions entrantes peuvent ajouter des jetons dans une MP.

(j) $B' = B \cup Blocked$ (mise à jour de l'ensemble des transitions bloquées : ajout de toutes les transitions nouvellement bloquées).

(k) $B' = B - Unblocked(s, Fired)$ (déblocage de toutes les transitions désensibilisées par le tir de $Fired$).

6. Changement d'états basé sur le tir de transitions, sans blocage, avec tirs de transitions d'exception :
$s = (m, B, R) \overset{Fired,\theta}{\twoheadrightarrow} s' = (m', B, R')$ ssi :

(a) $\forall t \in Fired,\ t \in Firable(s) \wedge (Fired \cap Conflict(s,t) = \emptyset)$ (les transitions en conflit ne sont jamais tirées simultanément).

(b) $\exists t \in Fired \wedge t \in T_{EXP}$ (il existe au moins une transition d'exception dans l'ensemble $Fired$).

(c) $\forall t \in Fired \wedge t \in T_{EXP} \mid \forall k \in Prio(t), \nexists k \in Fired$ (effet de la purge : toutes les transitions désensibilisées par la purge exécutée par le tir de la transition d'exception t, ne sont pas tirées).

(d) $\forall t \in enabled(m), \theta \leq\uparrow R(t)$ (il n'existe pas une transition qui doit être tirée à une date antérieure).

(e) $\forall t \in Fired \wedge t \in T_{EXP}, \forall p \in Purge(t) : m'(p) = 0 + \sum_{k \in Fired} Post(k)$ (mise à jour du marquage des places internes des raffinements : purge, i.e $m'(p) = 0$, et mise à jour du marquage liée aux tirs des transitions entrantes, i.e. $+ \sum_{k \in Fired} Post(k)$).

(f) $\forall t \in Fired \wedge t \in T_{EXP}, \forall p \in P \backslash Purge(t) : m'(p) = m - \sum_{k \in Fired} Pre(k) + \sum_{k \in Fired} Post(k)$ (mise à jour du marquage des autres places).

(g) $\forall t' \in enabled(m')$, $R'(t') = R_s(t')$ ssi $t' \in newEnabled(m)$, sinon $R'(t') = [max(1, \downarrow R(t') - \theta), \uparrow R(t') - \theta]$ (mise à jour des intervalles résiduels de tir).

(h) $B' = B - Unblocked(s, Fired)$ (déblocage de toutes les transitions désensibilisées par le tir de $Fired$).

5.2.3 Le graphe de comportement d'un STPNBE

La prise en compte des transitions d'exception ainsi que des effets de leur tir sont gérés dans la sémantique du formalisme STPNBE. Par conséquent,

la sémantique du graphe de comportement synchrone avec blocage et transitions d'exception (*Synchronous Behavior Graph with Blocking and Exception transitions*) d'un STPNBE, noté SBGBE, est la même que la sémantique du STPNBE. Et comme pour la sémantique du SBG et celle du SBGB, afin d'éviter l'évolution de temps indéfini/infini et pour réduire le nombre d'états du graphe, les deux changements d'états par saut de temps présentés dans le chapitre 4 sont également considérés.

Un état du SBGBE est représenté sous la forme $e = (m, B, R)$, avec m le marquage, B l'ensemble des transitions bloquées et $R : enabled(m) \rightarrow I^*$ la fonction des intervalles résiduels de tir des transitions sensibilisées par m. L'état initial est $e_0 = (m_0, B_0, R_0)$ avec m_0 le marquage initial, $B_0 = \emptyset$ et $R_0(t) = R_s(t), \forall t \in enabled(m_0)$.

5.2.4 Algorithme de construction du SBGBE

Principe de l'algorithme de construction du SBGBE

Le principe de l'algorithme de construction de SBGBE est le même que celui de graphe SBGB. Dans un état donné, la première étape est d'identifier les ensembles de transitions tirables/potentiellement bloquées à partir de l'ensemble des transitions sensibilisées. Ensuite, tous les changements d'états possibles sont identifiés en faisant toutes les combinaisons des ETTB. Les états successeurs qui correspondent à ces changements d'états sont alors construits, et ajoutés au graphe s'ils n'existent pas déjà. Les changements d'états ne contenant pas de tir de transition d'exception sont les mêmes que pour la construction d'un SBGBE, les algorithmes traitant de la construction de cette partie du graphe ont donc été présentés dans le chapitre 4. Les changements d'états avec tirs de transitions d'exception sont présentés dans les algorithmes 8, 9 et 10, qui sont des extensions des algorithmes précédents.

Les ensembles des transitions tirables/potentiellement bloquées ETTB et la fonction *Powerset* sont définis exactement comme dans le chapitre 4.

Pour chaque état, la grande différence entre les algorithmes SBGB et SBGBE consiste à empêcher le tir des transitions tirables appartenant à l'ensemble $Prio(t)$, pour chacune des transitions d'exception t appartenant aux transi-

tions tirées $t \in Fired \wedge t \in T_{EXP}$. La détermination de ces transitions est faite à l'aide de la fonction 7.

Algorithme de construction de la fonction Prio(t)

La fonction 7 sert à construire, pour toute transition d'exception t tirée à partir d'un état (i.e. $t \in Fired$), l'ensemble des transitions appartenant à l'ensemble $Prio(t)$ et qui sont effectivement tirables dans cet état. Cet ensemble est appelé EXP_{prio}, et est construit à partir de l'ensemble $Fired$ donné en entrée.

Algorithme 7 : Fonction *Prio()*

```
Data : Fired
Result : EXP_prio
1  EXP_prio ← ∅; /* EXP_prio est l'ensemble de toutes les transitions
     tirables appartenant à l'ensemble Prio(t) pour chaque transition
     d'exception t ∈ Fired.                                          */
2  fonction EXP_prio(Fired)
3      for (t ∈ Fired) do
4          if t ∈ T_EXP then
5              for (k ∈ Prio(t)) do
6                  if k ∈ Fired then
7                      EXP_prio ← k;
8                  end
9              end
10         end
11     end
```

Illustration sur un exemple

Nous allons expliquer l'algorithme 8 et la fonction 7 à partir d'un exemple de modèle STPNBE (cf. figure 5.10) afin d'illustrer les situations de tirs de transitions d'exception.

La figure 5.10 présente un modèle ITPN avec une MP, liée à deux transitions d'exception t_{exc} et t_{exc2}. La première (t_{exc}) sera tirée après une unité de temps si la place p_5 du raffinement de la MP contient deux jetons. La deuxième

Algorithme 8 : Algorithme de construction du SBGB - Partie 1

Data : $e_0 = (m_0, B_0, R_0)$
Result : SBG

```
1  Algorithm ConstructB(e)
       ETTB(e); /* déterminer les ensembles ETTB                    */
2      if 1.1FB(e) ≠ ∅ then                                  // case (1)
3          Fired ← 1.1FB(e) \ EXP_prio(1.1FB(e));
4          e -Fired,θ=1->> e' ;        ConstructB(e') ;
5          if 1.bF(e) ≠ ∅ ∨ 1.∞F(e) ≠ ∅ then                 // case (2)
6              if PBlocked(e) ≠ ∅ then                       // sous-case (2.1)
7                  for (w ∈ PowerSet(1.bF(e) ∪ 1.∞F(e)) do
8                      W ← w \ EXP_prio(1.bF(e) ∪ 1.∞F(e));
9                      for (Blocked ∈ PowerSet(PBlocked(e))) do
10                         Fired ← Fired \ Blocked;
11                         e -Fired∪W,Blocked,θ=1->> e' ;
12                         ConstructB(e') ;
13                     end
14                 end
15             else                                          // sous-case (2.2)
16                 for (w ∈ PowerSet(1.bF(e) ∪ 1.∞F(e))) do
17                     W ← w \ EXP_prio(1.bF(e) ∪ 1.∞F(e));
18                     e -Fired∪W,θ=1->> e' ;        ConstructB(e') ;
19                 end
20             end
21         else                                              // case (3)
22             if PBlocked(e) ≠ ∅ then
23                 for (Blocked ∈ PowerSet(PBlocked(e))) do
24                     Fired ← Fired \ Blocked ;
25                     e -Fired,Blocked,θ=1->> e' ; ConstructB(e') ;
26                 end
27             end
28         end
29     else
           /* See part 9                                       */
30     end
```

Algorithme 9 : Algorithme de construction du SBGB - Partie 2

```
1  if 1.bF(e) ≠ ∅ ∨ 1.∞F(e) ≠ ∅ then                                  // case (4)
2      for ((w ∈ PowerSet(1.bF(e) ∪ 1.∞F(e))) do
3          W ← w \ EXP_prio(1.1FB(e));
4          e -W,θ=1->> e' ;        ConstructB(e')
5      end
6      if 1.bF(e) = ∅ ∧ a.aFB(e) = ∅ ∧ c.dFB(e) = ∅ then
7          e -θ=1->> e
8      else
9          e -θ=1->> e' ;        ConstructB(e') ;
10     end
11 else
12     if a.aFB(e) ≠ ∅ ∧ c.dFB(e) ≠ ∅ then                            // case (5)
           /* PART 3                                                    */
13     else
14         if a.aFB(e) ≠ ∅ then                                       // case (6)
15             Fired ← a.aFB(e) \ EXP_prio(a.aFB(e));
16             e -Fired,θ=a->> e' ;        ConstructB(e') ;
17             if PBlocked(e) ≠ ∅ then
18                 for (Blocked ∈ PowerSet(PBlocked(e))) do
19                     Fired ← Fired \ Blocked ;
20                     e -Fired,Blocked,θ=a->> e' ;
21                     ConstructB(e') ;
22                 end
23             end
24         end
25         if c.dF(e) ≠ ∅ then                                        // case (7)
26             e -θ=c−1->> e' ;        ConstructB(e') ;
27         end
28     end
29 end
```

Algorithme 10 : Algorithme de construction du SBGB - Partie 3

```
   /* ∀t ∈ a.aFB(e), R(t) = [a, a] ∀t ∈ c.dFB(d), R(t) = [c, d]                    */
 1 if a < c then                                                  // sub-case (5.1)
 2     Fired ← a.aFB(e) \ EXP_prio(a.aFB(e));
 3     e -Fired,θ=a↠ e' ;        ConstructB(e') ;
 4     if PBlocked(e) ≠ ∅ then
 5         for (Blocked ∈ PowerSet(PBlocked(e))) do
 6             Fired ← Fired \ Blocked ;
 7             e -Fired,Blocked,θ=a↠ e' ; ConstructB(e') ;
 8         end
 9     end
10 else
11     if a > c then                                              // sub-case (5.2)
12         e -θ=c−1↠ e' ;        ConstructB(e') ;
13     else                                                       // sub-case (5.3)
14         Fired ← a.aFB(e) \ EXP_prio(a.aFB(e));
15         e -Fired,θ=a↠ e' ;        ConstructB(e')
16         if PBlocked(e) ≠ ∅ then
17             for (Blocked ∈ PowerSet(PBlocked(e))) do
18                 Fired ← Fired \ Blocked ;
19                 e -Fired,Blocked,θ=a↠ e' ; ConstructB(e') ;
20             end
21         end
22         for (w ∈ PowerSet(c.dFB(e))) do
23             W ← w \ EXP_prio(c.dFB(e));
24             if PBlocked(e) ≠ ∅ then
25                 for (Blocked ∈ PowerSet(PBlocked(e))) do
26                     Fired ← Fired \ Blocked ;
27                     e -Fired∪W,Blocked,θ=a↠ e' ; ConstructB(e') ;
28                 end
29             else
30                 e -a.aFB(e)∪W,θ=a↠ e' ;        ConstructB(e') ;
31             end
32         end
33     end
34 end
```

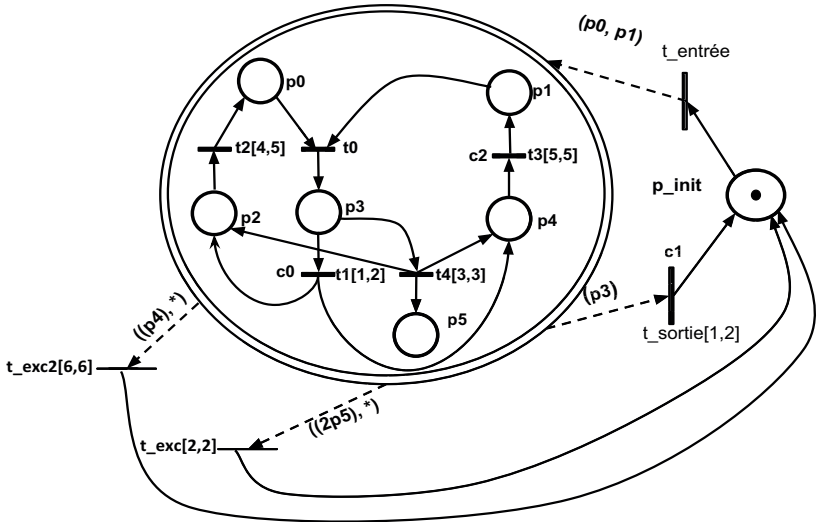


FIGURE 5.10 – Exemple d'illustration : ITPN avec macroplace et 2 transitions d'exception

transition (t_{exc2}) est sensibilisée si la place du raffinement p_4 est marquée. Elle sera tirée si le temps écoulé depuis sa sensibilisation est égal à 6 unités de temps. Le tir de la transition t_{exc2} sert à gérer le blocage/déblocage de la transition t_3 (qui peut se bloquer à $[5, 5]$).

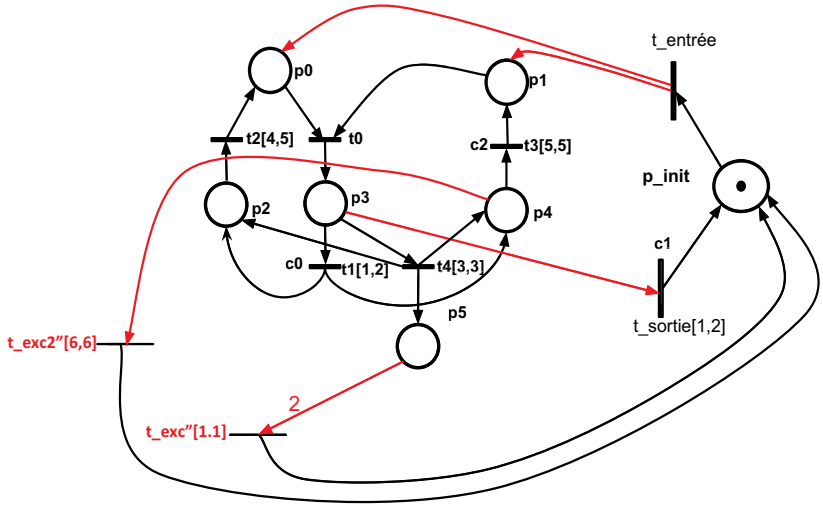


FIGURE 5.11 – Mise à plat du ITPN avec macroplace de la figure 5.10

La figure 5.11 montre la mise à plat de la MP du modèle ITPN présenté dans la figure 5.10, en conservant pour l'instant l'interprétation. La transformation

de ce modèle pour considérer l'interprétation et l'exécution synchrone donne le modèle STPNBE présenté dans la figure 5.12.

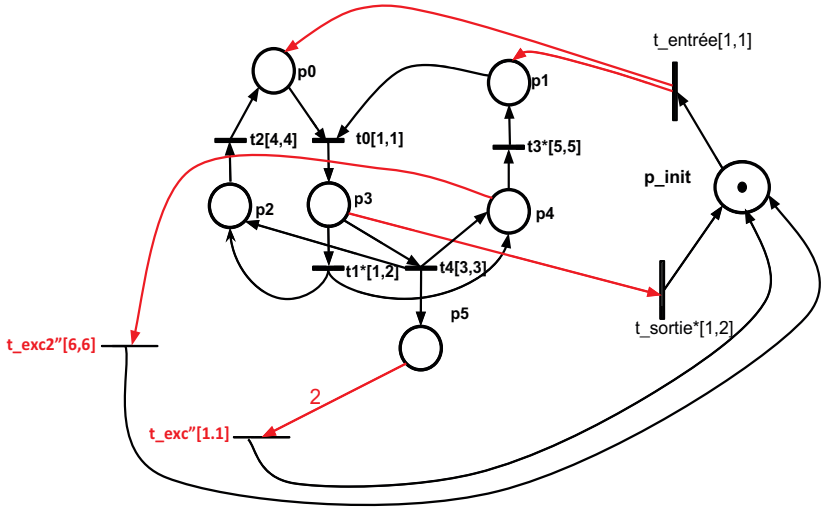


FIGURE 5.12 – Transformation du modèle ITPN avec macroplace mise à plat de la figure 5.11 vers un modèle STPNBE

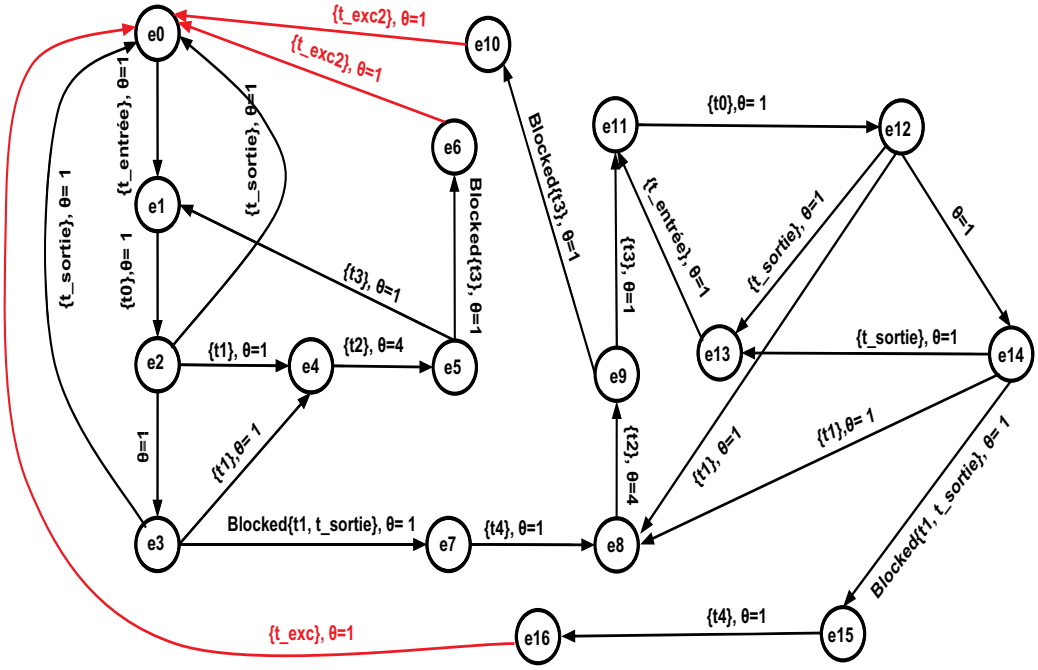
À partir du modèle STPNBE obtenu, nous allons illustrer la construction du graphe de comportement SBGBE.

Seuls les quelques états et changements d'états qui illustrent les tirs de transitions d'exception sont présentés. La construction du reste des états est similaire aux illustrations données dans les chapitres 3 et 4. Les tirs des transitions $t_{entrée}$ puis t_0 depuis l'état initial conduisent à l'état e_2. À partir de cet état, deux cas sont possibles : soit tirer la transition t_1 qui conduit à l'état e_4, soit une évolution de temps d'une unité de temps (ut) vers l'état e_3. De l'état e_4, la transition d'exception t_{exc2} est sensibilisée. Elle sera tirée si elle n'est pas désensibilisée et que le temps passé à partir de sa sensibilisation est égal à 6 ut. Cela devient possible dans l'état e_6, qui est obtenu à partir de e_4 par le tir de la transition t_2 puis du blocage de la transition t_3. Le tir de la transition d'exception vide toutes les places du raffinement, ainsi les jetons des places p_0 et p_4 sont supprimés. Cette transition rejoute 1 jeton dans la palce p_{init}, ce tir conduit donc à l'état initial.

À partir de l'état e_3, si les deux transitions t_1 et t_{sortie} sont bloquées (menant à l'état e_7), alors la transition t_4 est tirée après 1 ut et un premier jeton est

mis dans la place p_5. Le tir de la transition t_4 conduit à l'état e_8. La séquence de tir t_2, t_3, t_0 conduit vers l'état e_{12}. La transition t_1 n'est pas nécessairement tirable à partir de cet état. En effet, une évolution de temps de 1 ut mène à l'état e_{14}. Un deuxième blocage des transitions t_1 et t_{sortie} conduit à un nouveau tir de la transition t_4, et ainsi un deuxième jeton dans la place p_5. Si deux jetons sont présents dans la place p_5, alors la transition d'exception t_{exc} est tirée après une 1 ut. Ce tir annule les marquage $p3, p4, p_5(2)$ et conduit vers l'état initial.

Dans ce chapitre nous avons redéfini la sémantique du formalisme STPNB afin d'introduire la gestion des macroplaces et des transitions d'exception. Le formalisme obtenu, appelé STPNBE, est maintenant suffisamment complet pour pouvoir traiter un exemple réel.



e0: p_init, **B**{}, {t_entrée[1,1]}	**e3:** p3, **B**{}, {t1[1,1], t4[2,2], t_sortie[1,1]}	**e6:** p0,p4, **B**{t3}, {t_exc2[1,1]}	**e9:** p0,p4,p5, **B**{}, {t3[1,1], t_exc2[2,2]}	**e12:** p3,p5,**B**{}, {t1[1,2], t4[3,3], t_sortie[1,2]}	**p15:** p3, p5 **B**{t1, t_sortie}, {t4[1,1]}
e1: p0,p1,**B**{}, {t0[1,1]}	**e4:** p2,p4, **B**{}, {t2[4,4], t3[5,5], t_exc2[6,6]}	**p7:** p3, **B**{t1,t_sortie}, {t4[1,1]}	**e10:** p0,p4, p5 **B**{t3}, {t_exc2[1,1]}	**e13:** p_init,p5 **B**{}, {t_entrée[1,1]}	**p16:** p3, p4, p5(2) **B**{}, {t_exc[1,1]}
e2: p3, **B**{}, {t1[1,2], t4[3,3], t_sortie[1,2]}	**e5:** p0,p4, **B**{}, {t3[1,1], t_exc2[2,2]}	**e8:** p2,p4,p5 **B**{}, {t2[4,4], t3[5,5], t_exc2[6,6]}	**e11:**p0,p1,p5, **B**{}, {t0[1,1]}	**e14:** p3,p5, **B**{}, {t1[1,1], t4[2,2], t_sortie[1,1]}	

FIGURE 5.13 – Graphe de comportement SBGBE du modèle STPNBE de la figure 5.12

Chapitre 6

Mise en oeuvre et Application

6.1 Mise en oeuvre des algorithmes d'analyse

HILECOP est un logiciel permettant la conception des systèmes numériques critiques, de la modélisation à la génération de code. Il est donc nécessaire que le processus de validation soit directement relié à cet outil, afin de pouvoir exploiter les résultats d'analyse au sein même de l'outil de modélisation. Nous avons donc décidé d'intégrer nos algorithmes dans l'outil HILECOP. Ce travail a été effectué en collaboration avec Baptiste Colombani, ingénieur en charge du développement d'HILECOP.

Le logiciel HILECOP a été développé en Java avec la technologie Eclipse RCP *Rich Client Platform* [106], par plusieurs ingénieurs de l'équipe CAMIN. Le développement du logiciel est guidé par différentes techniques d'ingénierie logicielle, dont l'ingénierie dirigée par les modèles [54]. HILECOP repose donc sur une modélisation des concepts de : réseau de Petri (RdP), composants, architecture, transformations de modèles dont la génération de code, etc. A chaque concept est associé une ou plusieurs classes Java qui décrivent le comportement attendu, le rendu visuel, les contraintes associées au RdP, etc.

L'architecture du logiciel HILECOP est complexe. Les classes Java utilisées sont en grande partie générées automatiquement par la technologie utilisée dans le logiciel EMF *Eclipse Modeling Framework* [96]. Cette technologie permet de modéliser graphiquement des classes puis d'en générer le code java correspondant. Les avantages sont nombreux mais un des désavantages est la complexité à reprendre manuellement le code généré automatiquement en vue de le modifier. J'ai donc intégré mes contributions, dont les algorithmes de génération du graphe de comportement, par ajout de classes Java (sans passer par la génération de code d'Eclipse).

6.1.1 Structure du code

La figure 6.1 montre le diagramme de classes UML [76] de notre projet, qui représente une vue statique abstraite (sans prendre en compte le facteur temporel) ainsi que les relations entre les classes (héritage, agrégation, dépendance entre les classes, etc). Ce diagramme de classes a été généré à l'aide de l'extension java ObjectAid [78].

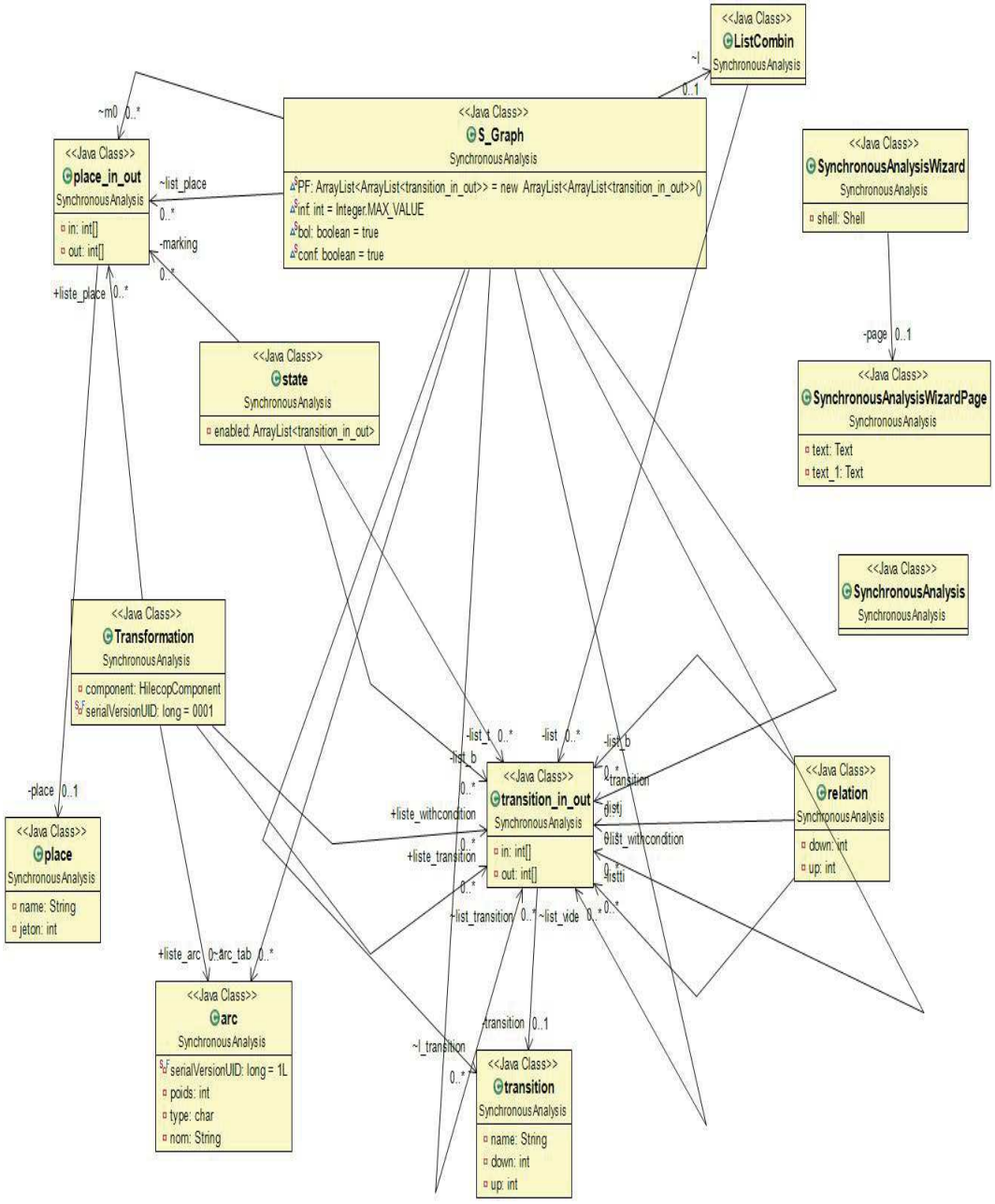


FIGURE 6.1 – Diagramme des classes

Le diagramme de séquences UML [90] du projet est représenté sur la figure 6.2. Ce diagramme montre les interactions entres les classes (objets) principales du projet d'un point de vue temporel, et précise l'ordre d'envoi de messages entre ces classes.

Ces deux diagrammes aideront à la compréhension des sections suivantes.

6.1.2 Implémentation des algorithmes

Entités de base : Chaque entité du RdP (place, transition, arc) est implémentée dans une classe afin de les utiliser comme des types pour une structure de données. Cette structure est constituée de trois listes : la liste des arcs, la liste des transitions et la liste des places. Chaque élément de la liste des arcs contient le numéro de l'arc (les arcs n'étant pas étiquetés sur le RdP, ils sont numérotés dans le code), le type de l'arc et le poids qui lui est associé. Chaque élément de la liste de transitions contient le nom de la transition, l'intervalle temporel associé, la liste des arcs entrants de cette transition et la liste des arcs sortants de cette transition. Pour les places, chaque élément contient le nom de la place, son marquage (les jetons), la liste des arcs entrants et la liste des arcs sortants. Deux autres listes sont utilisées pour déclarer les transitions possiblement bloquées et les transitions d'exception.

Transformation du ITPN en STPNBE : La première étape avant l'analyse consiste à générer un modèle global « mis à plat », en remplaçant, si le composant est un composite, les instances de composants le constituant par leurs RdP. Par analogie, on peut comparer cette étape à celle de compilation lors de laquelle un appel de fonction est remplacé par le code de cette fonction. Cette étape était déjà implémentée dans HILECOP car elle est également utilisée pour la génération du code VHDL à implémenter.

Puis ce modèle intermédiaire résultant (i.e., après mise à plat), qui est un ITPN, est utilisé afin de le transformer en un modèle analysable STPNBE. La structure du STPNBE, représentée par les listes de places, d'arcs et de transitions, est construite automatiquement à partir des éléments existants dans HILECOP. Il s'agit ensuite d'appliquer les règles transformation nécessaires pour refléter l'impact de l'exécution synchrone et de l'interprétation (cf. chapitre 2),

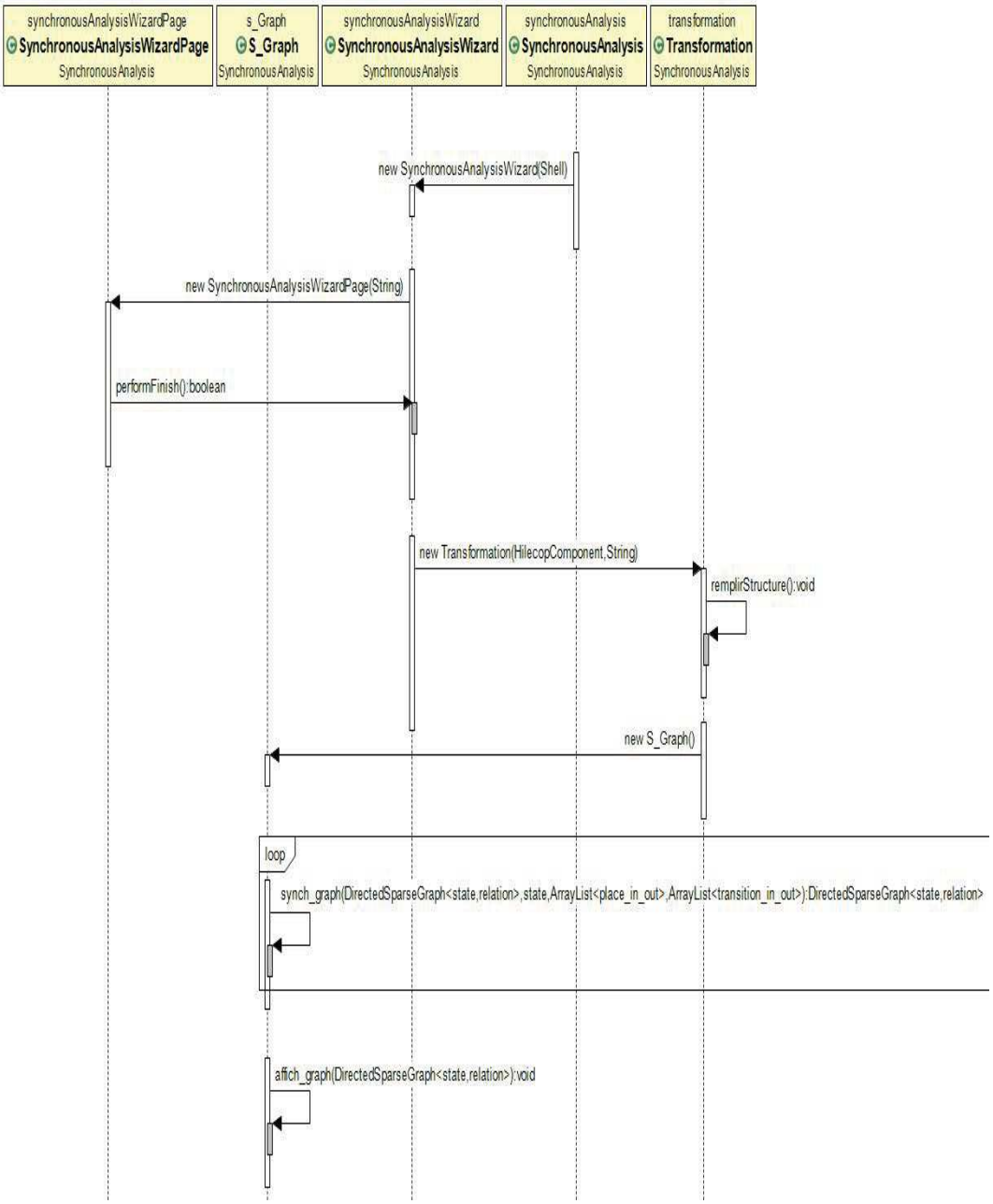


FIGURE 6.2 – Diagramme de séquences

ainsi que d'identifier les transitions potentiellement bloquées et les transitions d'exception en se basant sur les critères présentés dans les chapitres 4 et 5.

Ces transformations s'effectuent à l'aide des classes suivantes :

— $SynchronousAnalysisWizard$ et $SynchronousAnalysisWizardPage$: technologie utilisée, Eclipse E4 : Wizard/Wizard Page [35]. Elles consistent à communiquer avec d'autres Projet HILECOP, afin de récupérer les entités de bases d'un ITPN.

— $SynchronousAnalysis$: Elle consiste à faire un intermédiaire entre les classes $SynchronousAnalysisWizard$, $SynchronousAnalysisWizardPage$ et la classe *Transformation*.

— $Transformation$: Les éléments du modèle ITPN sont parcourus afin d'être transformés et générer les listes de données utilisées.

Fonctions de bases de l'algorithme : La classe principale S_Graph contient les méthodes de pré-traitement suivantes :

— $return\_place\_pre$, $return\_place\_post$: Ce sont les méthodes Pre et $Post$ d'un RdP classique.

— $enable\_t$: C'est la fonction qui détermine la liste des transitions sensibilisées.

— nFT, un_bF,pFT, lFT, p_a_bF : ce sont les méthodes qui déterminent les ensembles de transitions tirées/bloquées tels que définis dans la section 4.2.1.

— $update\_m$: Ce sont les méthodes qui font la mise à jour du marquage après le(s) tir(s) de transition(s).

— $reinit\_b$: C'est la méthode qui réinitialise les intervalles temporels des transitions tirées.

— $conflit$: Il s'agit de la méthode qui gère les conflits. Elle partage la liste des transitions sensibilisées en conflit en sous-listes de transitions sensibilisées qui ne contiennent pas de conflit. Pour cela, cette méthode fait appel à d'autres méthodes comme $conflict\_2\_t\_d$ pour vérifier si deux transitions sont en conflit et $existe\_conflict$ pour voir s'il reste des transitions en conflit dans la liste d'entrée.

— $residual\_time$: Cette méthode fait la mise à jour des intervalles résiduels de tir pour les transitions sensibilisées après chaque tir de transition(s).

— *unblocked* : La méthode qui débloque les transitions.

Algorithme de construction du SBG : L'algorithme de construction du graphe SBG (cf. algorithmes 4 et 8) est implémenté par la méthode $Synch\_graph$. Cette méthode a comme entrée un état du graphe SBGBE. Une classe *State* est définie pour déclarer le type état du graphe (marquage, liste de transitions bloquées et fonction d'intervalle résiduel de tir). Pour chaque état, tous les états successeurs de cet état sont générés et liés avec cet état via une relation de changement d'états, telles que définies dans la sémantique du STPNBE/SBGBE (cf. 5). La relation entre les états du graphe (liste de transitions tirées, liste de transitions bloquées et le temps écoulé) est définie par la classe *relation*. La méthode est implémentée de manière récursive et donc elle boucle tant qu'il y a des transitions sensibilisées, donnant de nouveaux successeurs (non existants dans le graphe).

L'affichage graphique du graphe (états, relations) est généré et implémenté dans la méthode $affich\_graph$. Pour cela, nous utilisons la bibliothèque JUNG [1].

Le graphe SBGBE est également généré dans format XML, plus précisément GraphML [22]. Cela permet de sauvegarder le graphe généré dans un format de persistance semi-structuré et de permettre la reconstruction de tout ou partie du graphe sans (re)passer par l'algorithme de construction. Un exemple de SBGBE en format GraphML est donné dans la figure 6.3. Les états sont décrits par la balise $< node >$, les changements d'états par la balise $< edge >$ qui contient l'état source *source*, l'état cible *target* et la description $< desc >$ représentant le label sur chaque changement d'états.

6.1.3 Intégration des algorithmes dans l'environnement HILECOP

Comme précédemment mentionné, nous avons intégré nos contributions à HILECOP par l'ajout de classes Java et de fenêtres d'affichage. Nos contributions représentent environ 4000 lignes de code.

1. http ://jung.sourceforge.net/

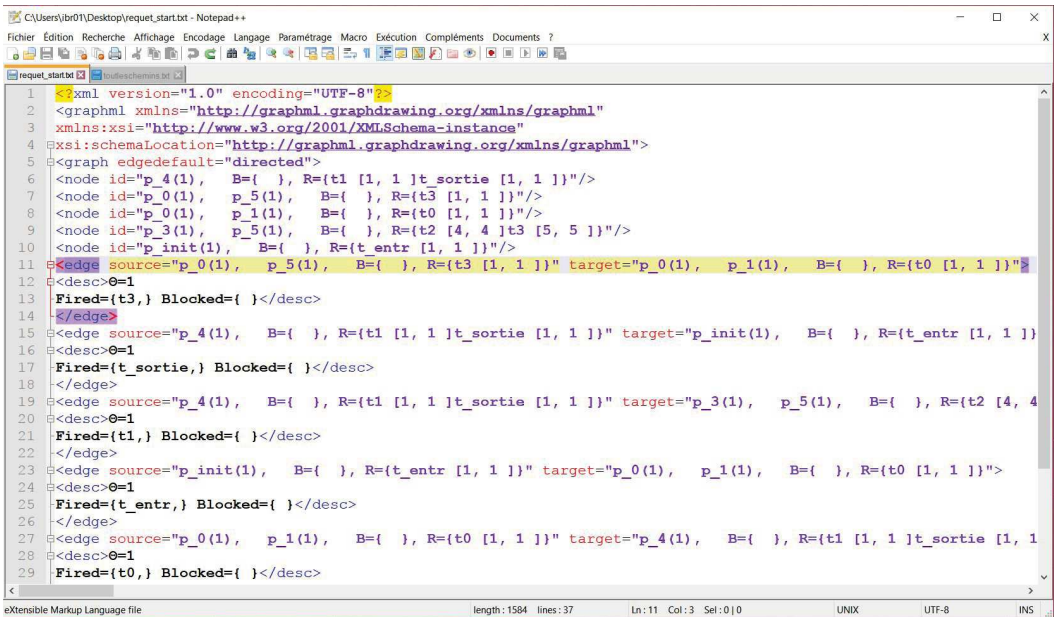


FIGURE 6.3 – SBGBE en format GraphML

Interface d'affichage de HILECOP : Avant de passer à l'étape d'analyse, plus précisément de génération du graphe, il est nécessaire d'introduire l'interface d'HILECOP. Cette interface est présentée dans la figure 6.4. Elle est composée de trois parties :

- *Partie (1) - explorateur de projet* : Cette partie affiche les projets HILECOP. Plusieurs opérations sont possibles sur les projets telles que la modification, l'analyse ou la création d'un nouveau projet. Un projet est composé d'un ou plusieurs composants. Ces derniers sont modélisés à l'aide de RdP qui peuvent être visualisés dans la partie (2).
- *Partie (2) - affichage* : il s'agit de l'affichage graphique des composants (potentiellement interconnectés), du comportement d'un composant, du code VHDL généré depuis ce composant et du graphe de comportement SBGBE du projet (l'assemblage de l'ensemble des composants du projet).
- *Partie (3) - palette d'outils* : Cette partie est composée de trois sous-parties :
  1. Les outils de modélisation par RdP : c'est l'ensemble des éléments graphiques d'un RdP (arcs, transitions, places, etc.), utilisés pour la modélisation du comportement des composants (ITPN).

2. Les outils VHDL : ce sont les outils d'édition du code VHDL. Ils servent à spécifier le code VHDL qui décrit une condition, une fonction ou une action, associés avec une (des) transition(s) ou une (des) place(s) du modèle SITPN.
3. Les outils de la macroplace : ces éléments permettent de modéliser la macroplace ainsi que ses entités attenantes (arcs entrants, arcs sortants, arcs d'exception) et de les associer avec un RdP (le raffinement de la MP).

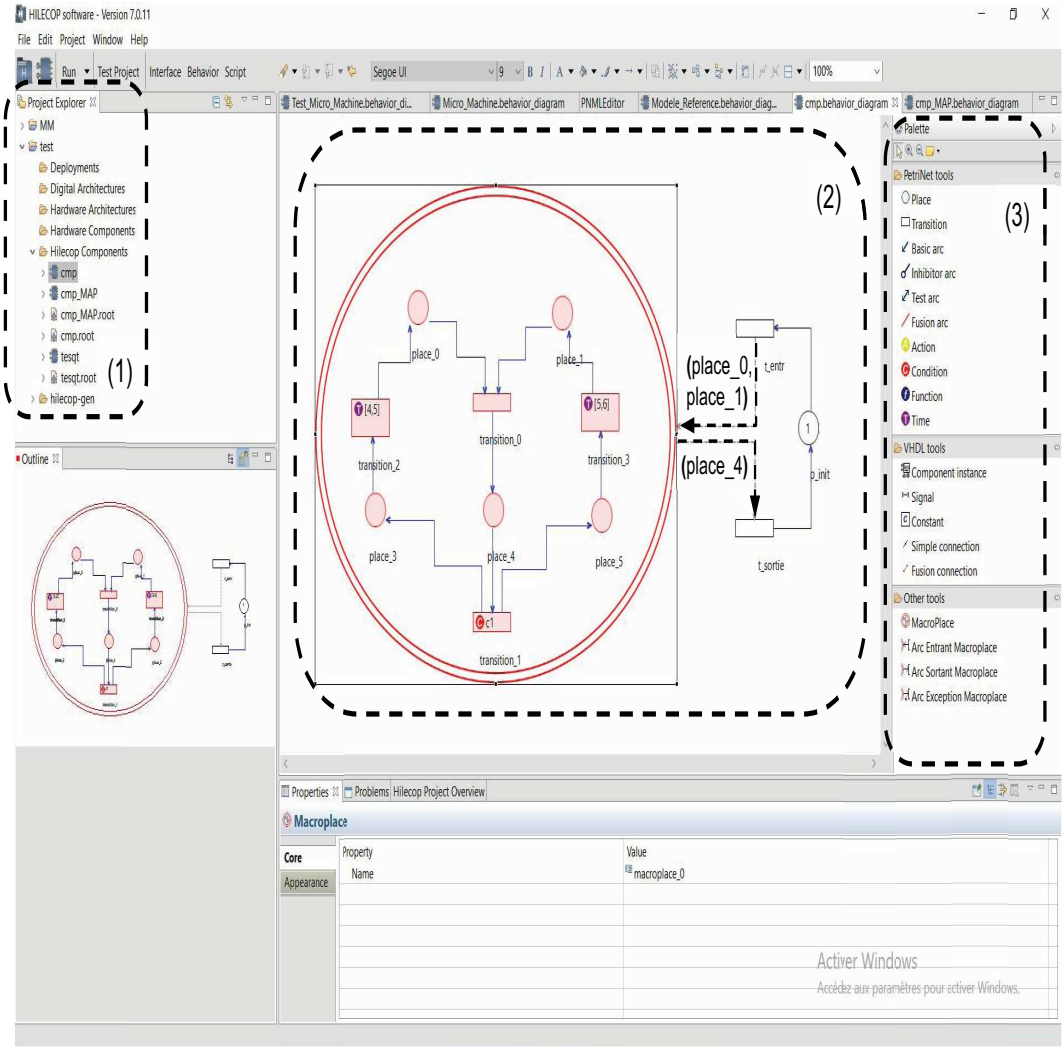


FIGURE 6.4 – Interface d'HILECOP

Lancement de la construction du graphe / l'analyse formelle : Pour effectuer l'analyse formelle d'un projet, la première étape consiste à générer son graphe de comportement. Pour cela, il est nécessaire dans un premier temps de désigner (sélectionner) le projet HILECOP concerné. Le composant de plus haut-niveau (qu'il soit un simple composant ou un composite) est l'architecture numérique de ce projet, et c'est sur ce composant que portera l'analyse ; i.e. elle sera réalisée sur le comportement global de cette architecture numérique. Une fois le composant sélectionné, l'analyse peut être lancée, tout au moins la génération du graphe. La figure 6.5 représente la fenêtre de lancement.

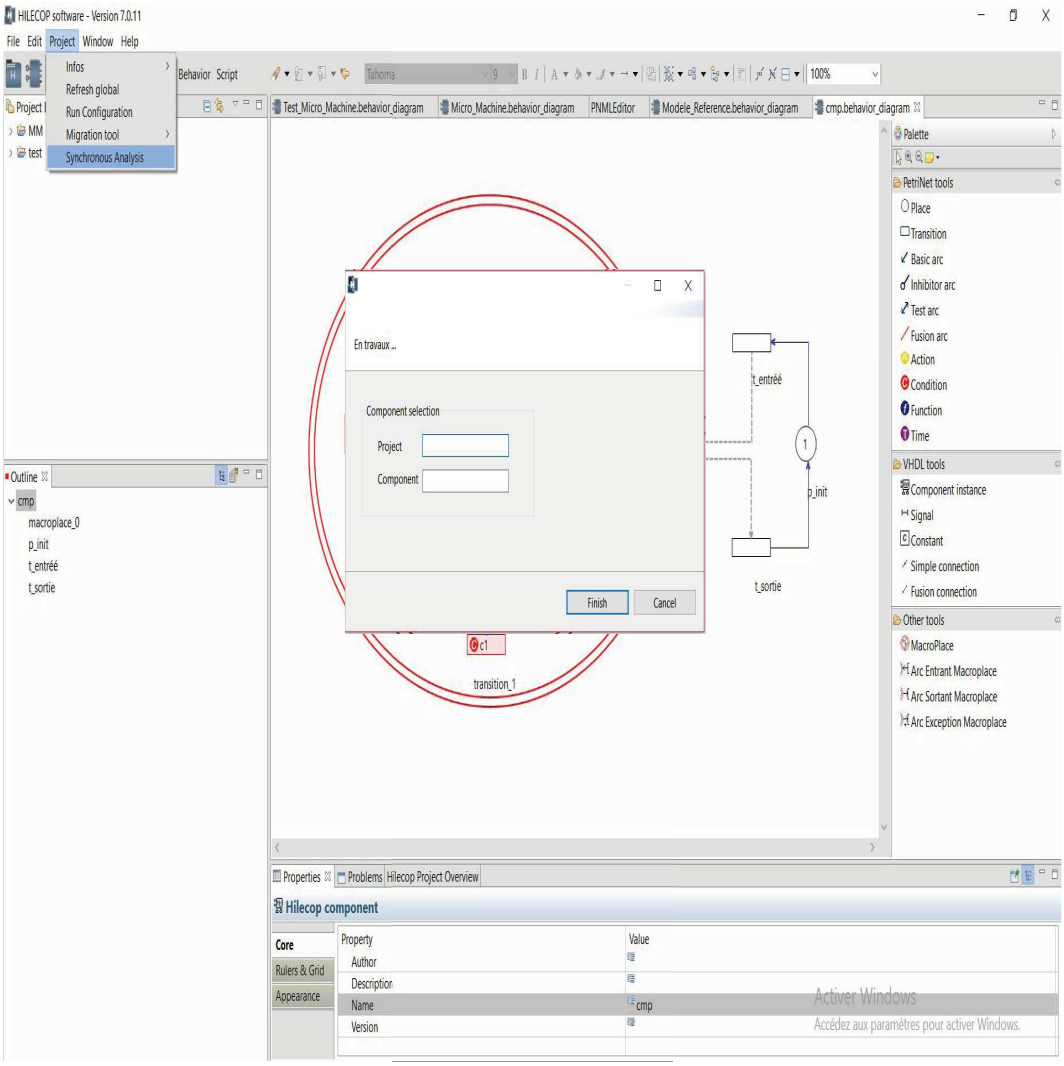


FIGURE 6.5 – Fenêtre de sélection de projet

Comme expliqué précédemment, la première étape du processus d'analyse est celle de transformation du ITPN en STPNBE, afin de remplir les structures de données à partir desquelles l'algorithme travaillera pour construire le graphe SBG. Cette étape est automatique et invisible en termes d'affichage graphique.

Une fois la génération terminée, une fenêtre s'ouvre dans HILECOP pour présenter les résultats. L'affichage est en l'occurrence le graphe de comportement du modèle analysé.

6.1.4 Exemple sous HILECOP

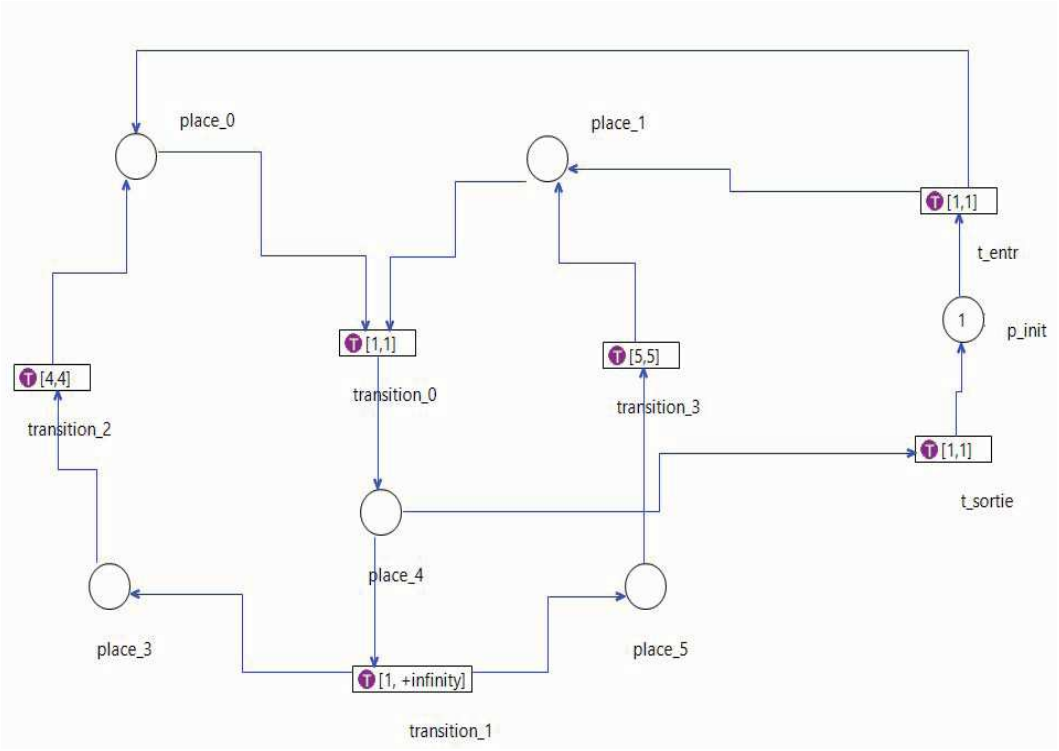
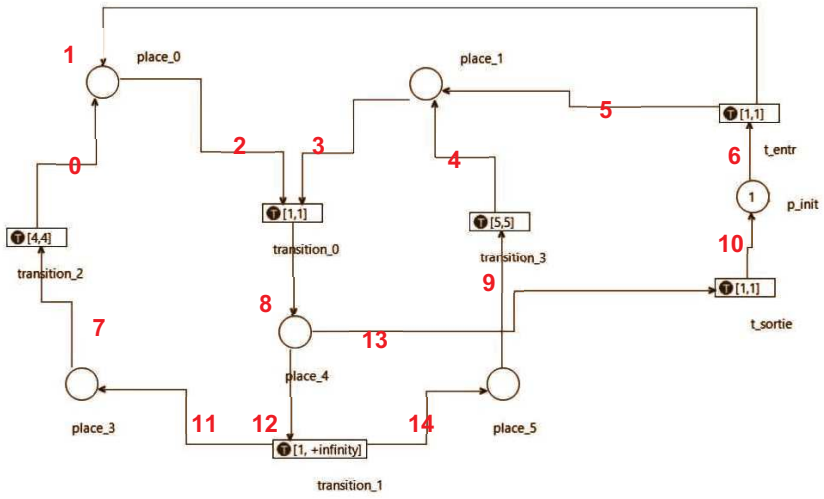


FIGURE 6.6 – Représentation du STPNBE obtenu de l'exemple donné figure 6.4

Nous prenons l'exemple du modèle ITPN donné figure 6.4. Une fois que l'analyse est lancée, le modèle est dans un premier temps mis à plat, et les règles de transformations pour la prise en compte des contraintes synchrones

et de l'interprétation sont appliquées. La figure 6.6 montre une représentation du modèle analysable STPNBE qui en résulte. Les structures des données remplies automatiquement à partir de ce modèle sont présentées dans la figure 6.7.

Le graphe de comportement SBGBE obtenu est donné dans la figure 6.8. Les cercles (en rouge) désignent les états du graphe. Dans chaque état nous trouvons le marquage, la liste des transitions avec leurs intervalles résiduels de tir mis à jour et la liste des transitions bloquées à cet état. Les arcs orientés représentent les changements d'états. Les arcs sont étiquetés par l'ensemble des transitions tirées, l'ensemble des transitions bloquées et le temps écoulé θ.



(a) STPNBE avec arcs numérotés (en rouge)

(transition_0 ,[1,1], {2,3}, {8})	(transition_1, [1, +∞[, {12}, {11, 14})	(transition_2, [2,2], {7}, {0})	(transition_3, [5,5], {9}, {4})	(t_entr, [1,1], {6}, {1,5})	(t_sortie, [1,1], {13}, {10})

(b) La liste des transitions

(place_0, 0, {0,1}, {2})	(place_1,0, {4, 5}, {3})	(place_3, 0, {11}, {7})	(place_4, 0, {8}, {12, 13})	(place_5, 0, {14}, {9}	(p_init, 1, {10}, {6})

(C) La liste des places

FIGURE 6.7 – Les listes arcs / transitions / places générées à partir du modèle STPNBE donné figure 6.6

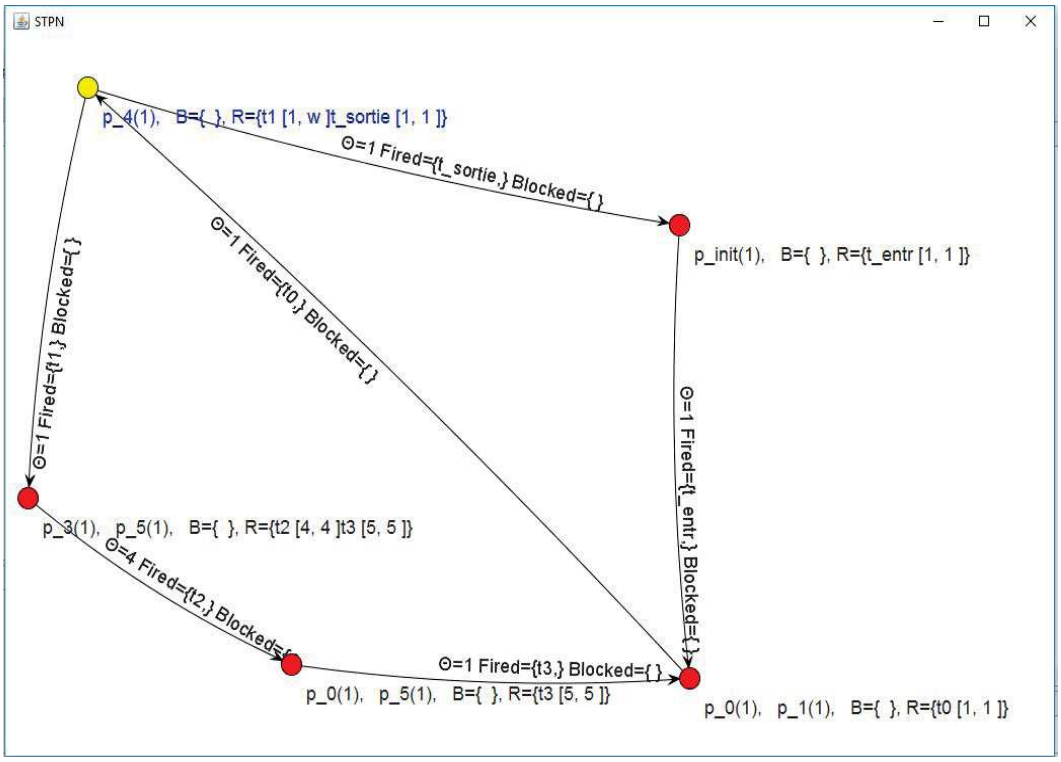


FIGURE 6.8 – Fenêtre d'affichage du graphe SBGBE

6.1.5 Validation de l'implémentation des algorithmes

Les algorithmes de génération de graphe SBGBE ont été testés sur plusieurs exemples-tests de modèles ITPN. Ces exemples présentent toutes les combinaisons possibles des aspects suivants : les types d'arcs, les intervalles temporels et les conditions associées aux transitions, et les situations de conflits. Tester ces différents aspects conduit à établir différentes combinaisons des types d'ensembles de transitions tirées / bloquées (cf. section 4.2.1). Pour chaque exemple testé, la vérification de conformité entre le graphe généré et le graphe SBGBE (théorique) attendu a été faite manuellement, à partir de l'affichage graphique du graphe.

Nous avons défini cinq ensembles de transitions tirables, ce qui donne théoriquement 32 combinaisons possibles de ces ensembles. En ajoutant les possibilités de blocages sur les ensembles $NFBT$ et $LFBT$, nous avons au total 128 combinaisons théoriquement possibles. Cependant, ces 128 combinaisons ne produisent pas toutes un changement d'états. Par exemple, si l'ensemble NFT

(en $[1, 1]$) est non vide, les transitions de cet ensemble seront obligatoirement tirées, indépendamment des valeurs des ensembles LFT ou IFT (cf. chapitre 3 table 3.1) ; ces cas se ramènent à un même changement d'états. Selon ce raisonnement, il existe en réalité 9 changements d'états différents dans notre sémantique de graphe, 13 lorsqu'on rajoute le blocage. Ces 13 changements d'états doivent évidemment être validés, et idéalement les 128 combinaisons devraient l'être pour garantir la validation complète de nos algorithmes. A ce jour, la validation des algorithmes a été faite sur un ensemble de modèles ITPN de test, couvrant une cinquantaine de combinaisons possibles (les plus pertinentes), en associant à chaque fois différents intervalles temporels et/ou une (ou plusieurs) condition(s) aux transitions de ces exemples. La validation se déroule donc en deux étapes :

1. La validation de la transformation : Pour chaque modèle de l'ensemble de test, la validation consiste premièrement à vérifier que la structure du modèle ITPN est préservée dans le modèle analysable généré dans le formalisme STPNBE. Pour cela, la conformité entre les structures des modèles de départ et de sortie est vérifiée ; à savoir, les ensembles de transitions, les ensembles d'arcs, les ensembles de places, le marquage initial, les interconnexions des noeuds (topologie), etc. Ensuite, pour chaque transition du modèle STPNBE, nous vérifions que l'intervalle associé respecte bien les règles de transformation décrites dans le chapitre 2.
2. La validation du graphe : Après la validation de la transformation, nous effectuons la validation de l'algorithme de génération du graphe. Pour chaque modèle ITPN, nous générons deux graphes : le premier à l'aide de l'outil développé et le deuxième manuellement (le graphe attendu de notre approche). Une vérification de conformité entre les deux graphes est faite. Lors de cette étape nous vérifions que toutes les séquences de tirs de transitions (traces) sont correctes, en considérant notamment si besoin le comportement des transitions en conflit, ainsi que les blocages potentiels et les tirs des transitions d'exception.

La figure 6.9 présente deux des exemples de modèles ITPN que nous avons utilisés pour la validation de nos algorithmes. Cette figure ne montre que la structure des modèles. Ensuite, en fonction des scénarios désirés, nous réparti-

TABLE 6.1 – Exemples de scénarios utilisés pour la validation des algorithmes

	NFBT	PFT		LFBT	IFT	
	$1.1FB$	$1.bF$	$1.\infty F$	$a.aFB$	$c.dF$	Blocage
scénario (1)	*					
scénario (2)	*		*			
scénario (3)	*			*	*	*
scénario (4)		*		*	*	*

rons les conditions et les intervalles temporels différemment sur les transitions. Nous allons montrer 4 scénarios pour chacun de ces modèles, afin d'illustrer la phase de validation. Les scénarios sont présentés dans la table 6.1. Les colonnes représentent les ensembles de transitions tirées / bloquées, ainsi que la présence du blocage. Les lignes représentent les scénarios. Pour un scénario, si au moins une transition appartient à un ensemble de transitions, alors la case de cet ensemble est cochée. De même pour le blocage, si au moins une transition est potentiellement bloquée alors la case blocage est cochée. Par exemple, toutes les transitions du scénario (1) appartiennent à l'ensemble $NFBT$ (1.1FB) et aucune transition n'est potentiellement bloquée.

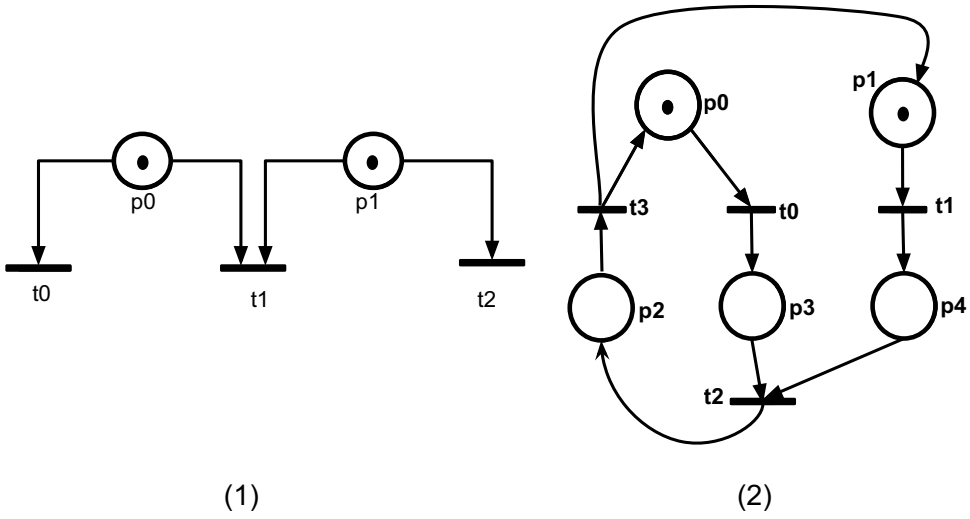


FIGURE 6.9 – Deux exemples de modèles simples, utilisés pour la validation de la mise en oeuvre des algorithmes

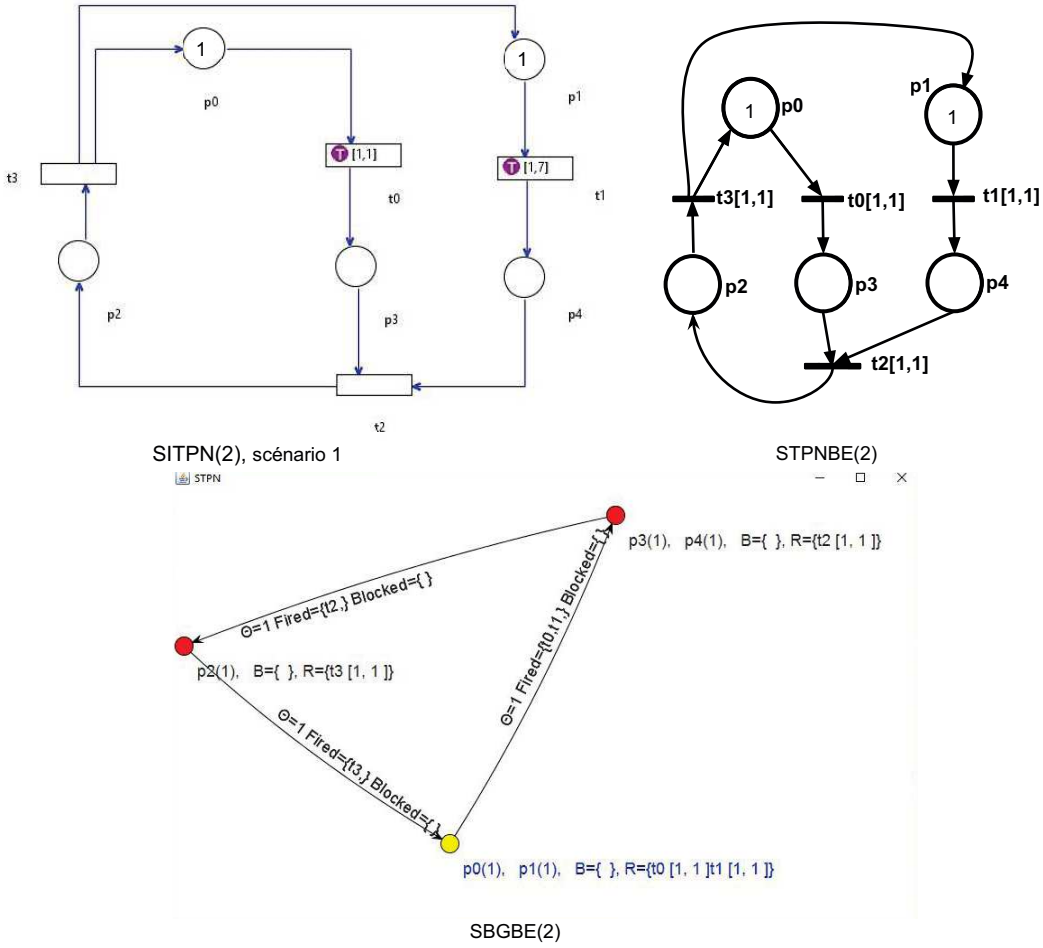


FIGURE 6.10 – Exemple de validation des algorithmes

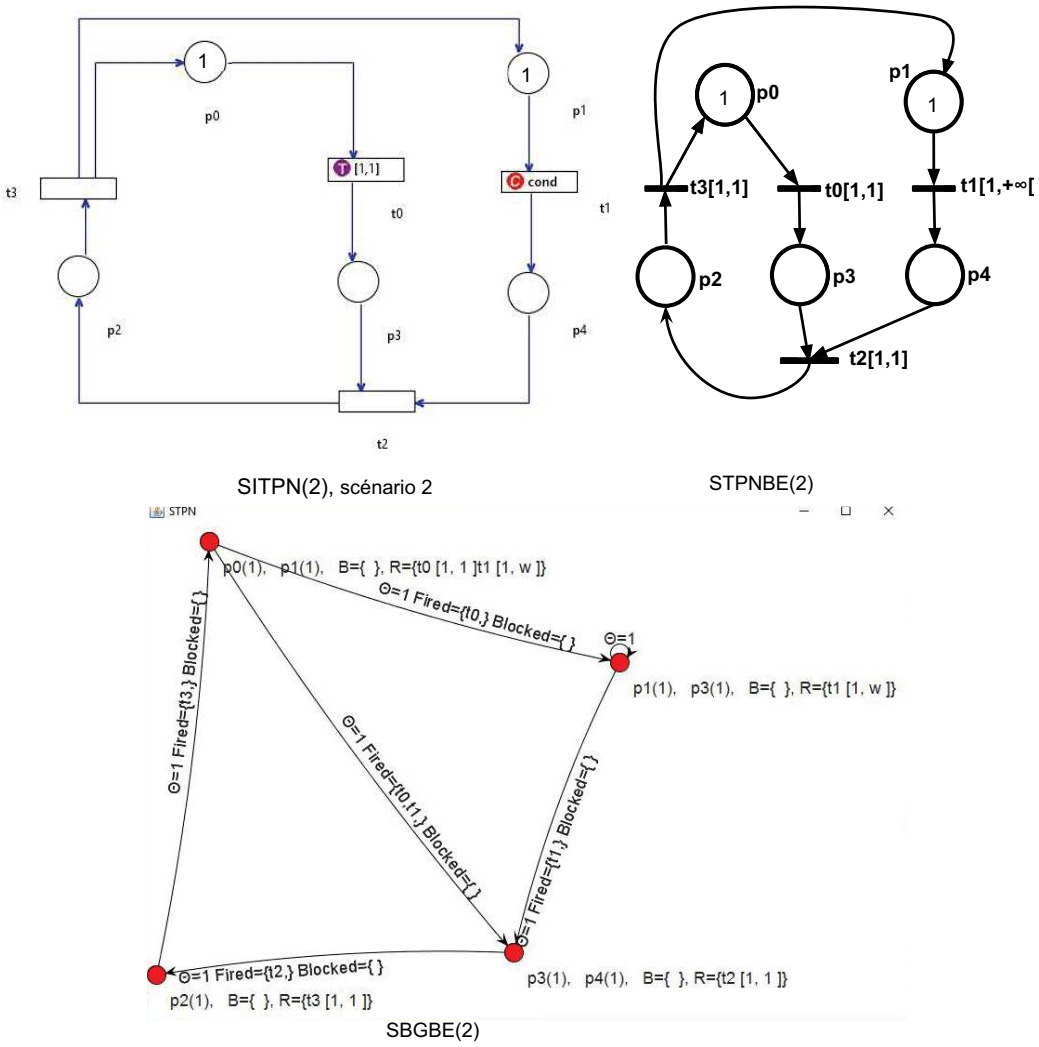


FIGURE 6.11 – Exemple de validation des algorithmes

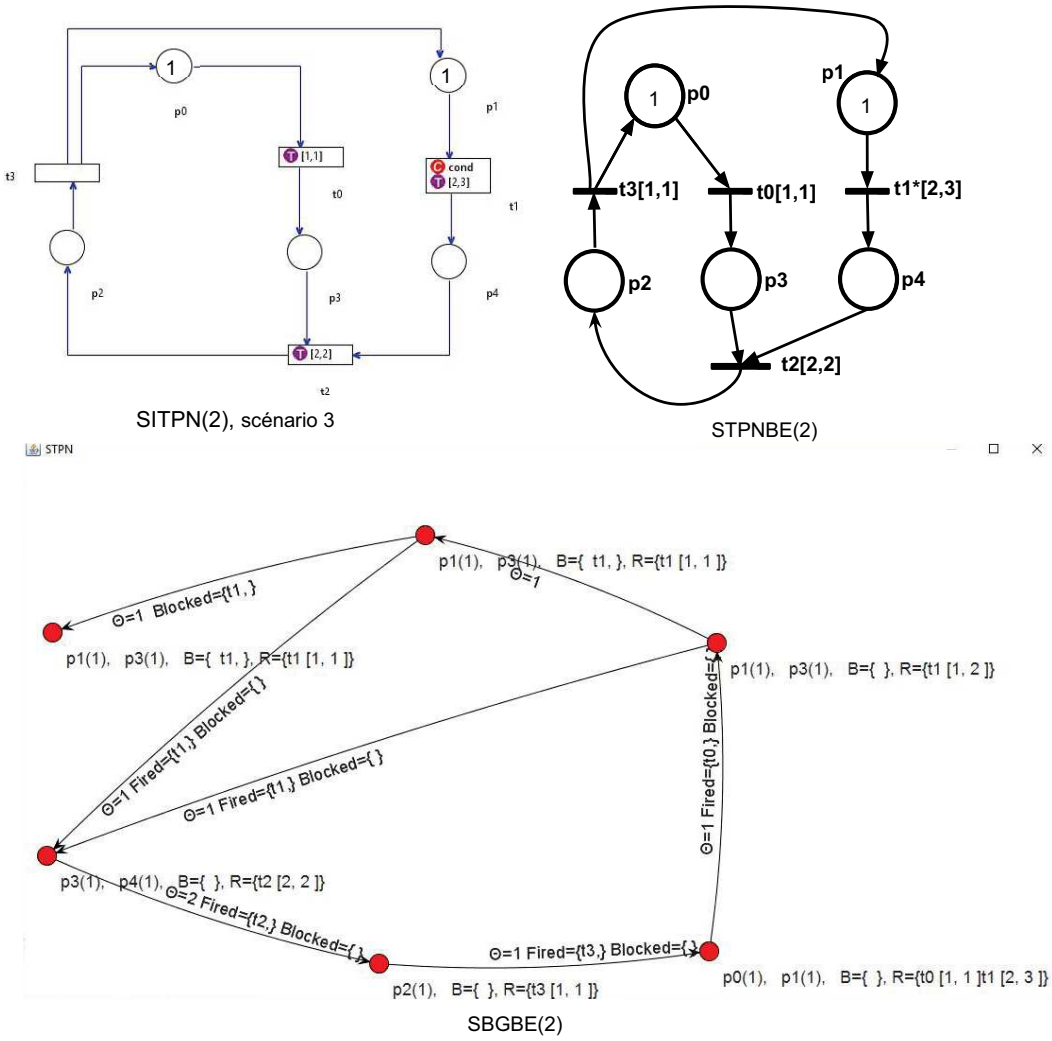


FIGURE 6.12 – Exemple de validation des algorithmes

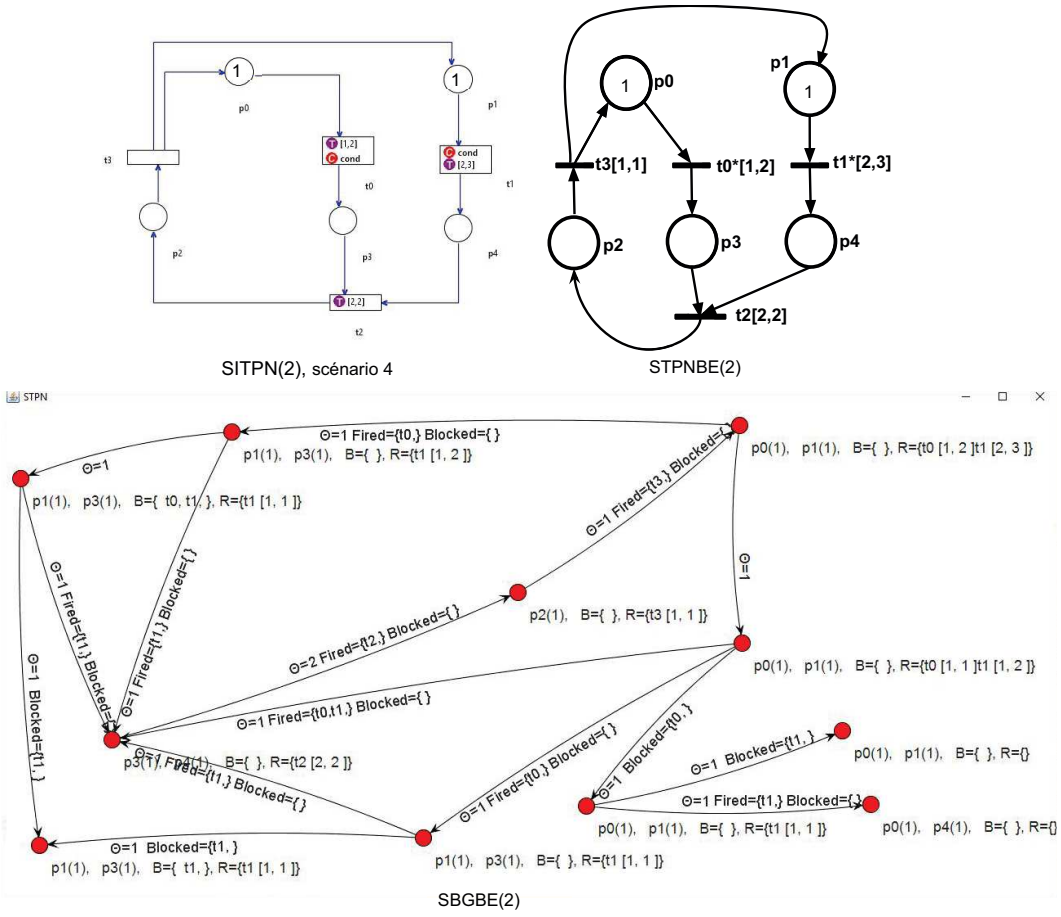
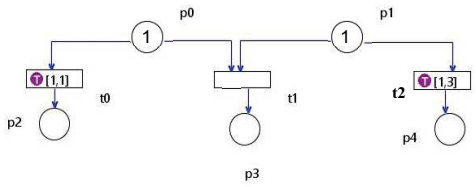
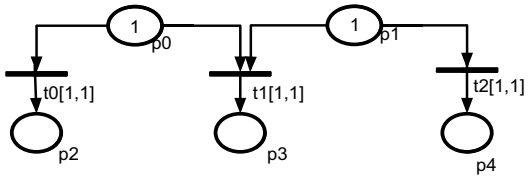


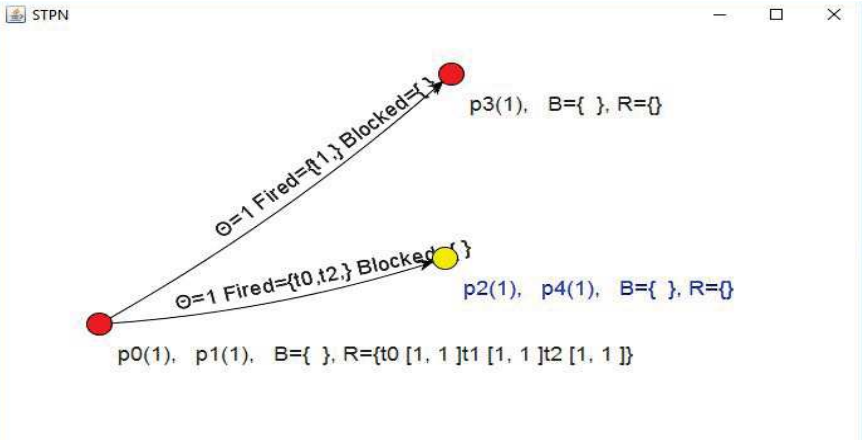
FIGURE 6.13 – Exemple de validation des algorithmes



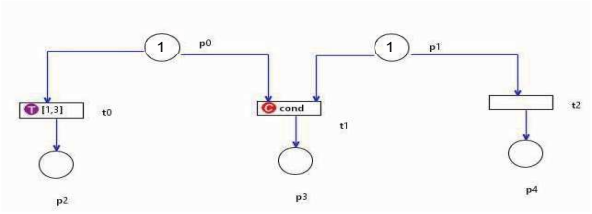
SITPN(1), scénario 1



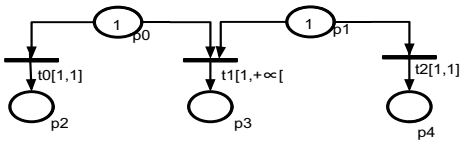
STPNBE(1)



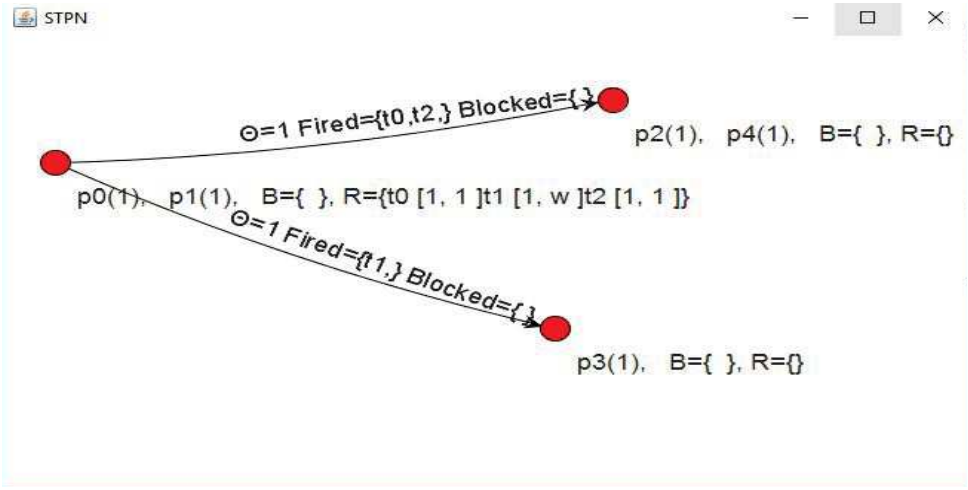
SBGBE(1)



SITPN(2), scénario 2



STPNBE(1)



SBGBE(2)

FIGURE 6.14 – Exemple de validation des algorithmes

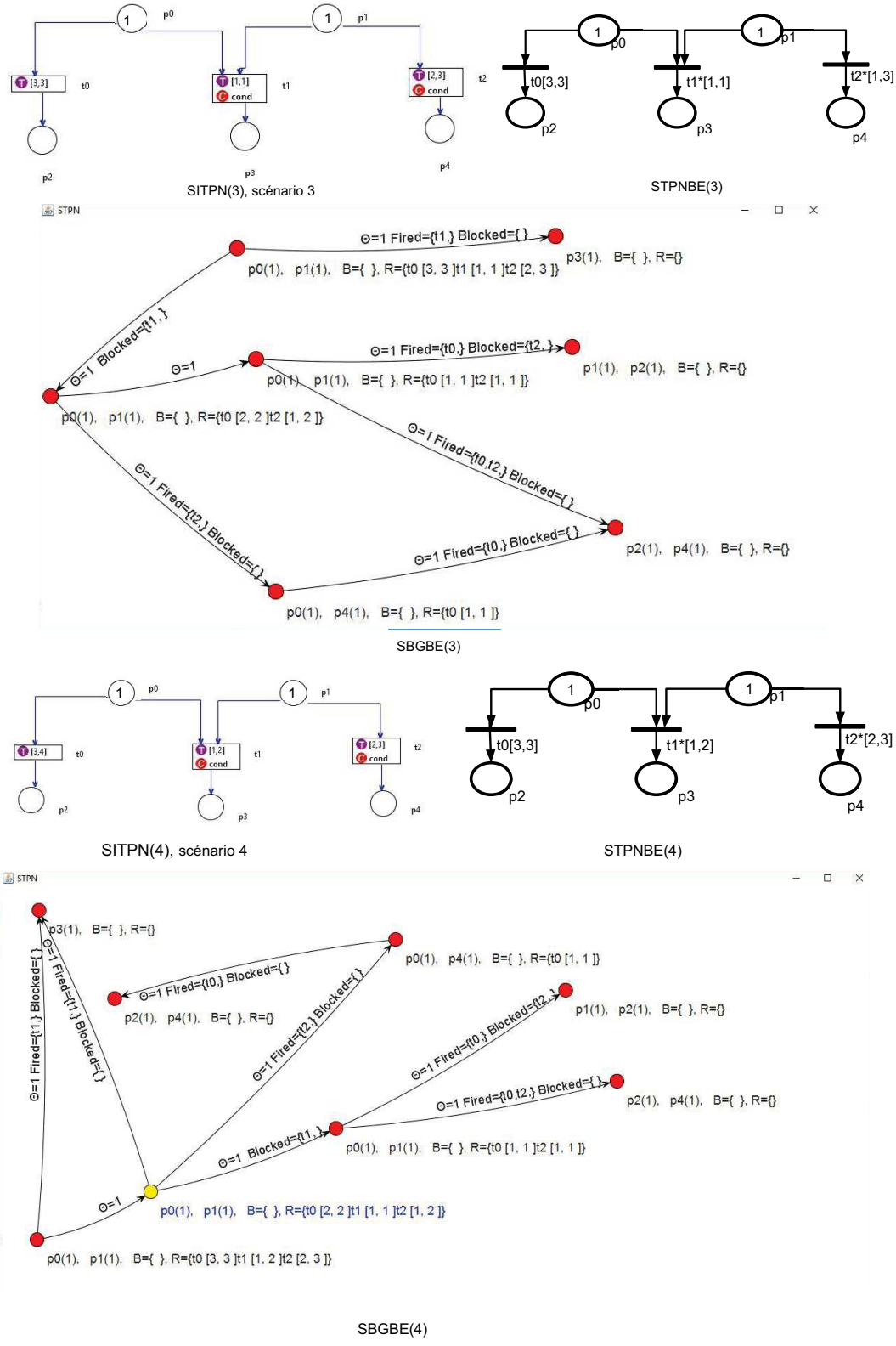


FIGURE 6.15 – Exemple de validation des algorithmes

6.2 Application à la conception d'un dispositif médical implantable actif

Nous avons appliqué notre processus d'analyse aux systèmes complexes et critiques pour lesquels la méthodologie HILECOP a été initialement développée, à savoir les dispositifs médicaux implantables actifs (DMIA). Il s'agit ici de stimulateurs implantables dans le corps humain, dédiés à la restauration de fonctions déficientes par stimulation électrique fonctionnelle (SEF). Ces déficiences peuvent avoir diverses origines dont une lésion de la moelle épinière - induisant une paraplégie ou une tétraplégie -, ou un accident vasculaire cérébral - induisant une hémiplégie -, et entraînent l'incapacité à réaliser certaines fonctions, selon la pathologie.

Plus généralement la SEF consiste à appliquer une stimulation électrique à un tissu nerveux (stimulation neurale) ou directement à un muscle (stimulation épimysiale) à l'aide d'un DMIA, afin d'induire artificiellement son activation. Un exemple bien connu de DMIA est le pacemaker dédié à la régulation du rythme cardiaque [110, 55].

Le stimulateur retenu pour l'étude est un stimulateur neural, autrement appelé neuro-stimulateur, que nous allons présenter.

6.2.1 Le neuro-stimulateur

Le DMIA considéré dans notre étude est un stimulateur implantable, visant la stimulation neurale à travers une électrode multipolaire [3]. Pour suppléer une fonction qui nécessite plusieurs muscles (contrôler un membre et ses articulations par exemple), il faut pouvoir stimuler sur plusieurs sites (éventuellement un par muscle devant être sollicité) de façon coordonnée. Ce stimulateur s'inscrit dans une architecture de stimulation distribuée composée de stimulateurs dont les activités sont coordonnées par un contrôleur : les stimulateurs sont en charge de la stimulation neurale pour induire les contractions musculaires et le contrôleur coordonne ces stimulations dans le cadre de la fonction à réaliser (contrôle du préhenseur de la main, contrôle des membres inférieurs pour le lever et la station debout, etc.). L'architecture de SEF distribuée repose donc sur des unités communicant sur un bus implanté (Figure 6.16), selon une solution protocolaire (pile de protocoles) fiable et déterministe.

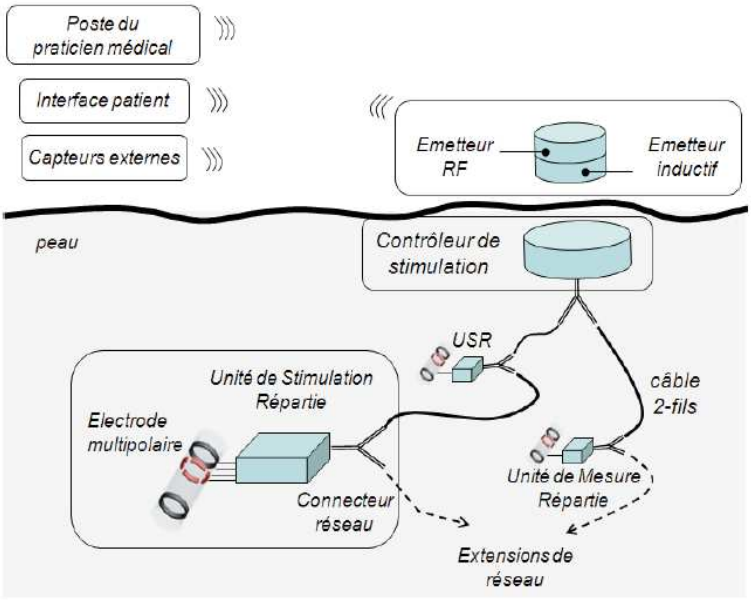


FIGURE 6.16 – L'architecture de stimulation distribuée

Nous allons nous intéresser uniquement au stimulateur neural (neuro-stimulateur), sachant que ce dernier est en charge d'exécuter un profil de stimulation précis (profil d'injection des charges sur le nerf, sur une base de temps à la microseconde et une précision de courant de quelques micro-ampères) et sur une configuration d'électrode précise (ensemble de contacts de l'électrode utilisés pour recruter un sous-ensemble de fibres nerveuses dans le nerf). Une autre contrainte forte imposée par l'architecture distribuée de SEF est la réactivité dans le cadre de la synchronisation des activités des stimulateurs dès lors que plusieurs sites de stimulation doivent être synchronisés.

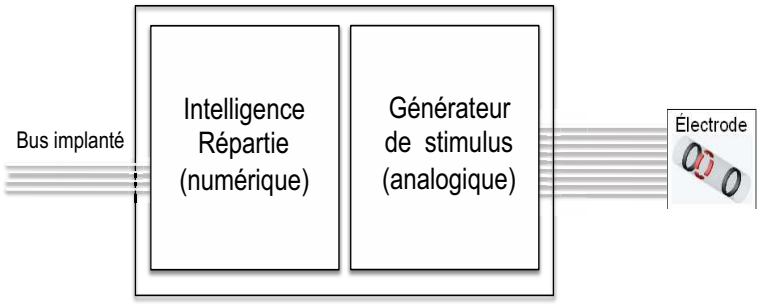


FIGURE 6.17 – L'architecture du neuro-stimulateur

Le neuro-stimulateur est composé de deux parties (voir Figure 6.17 et [3]) :

— Une partie analogique comprenant un générateur de stimulus chargé de générer le signal appliqué au nerf via l'électrode, et d'assurer la répartition de courant selon la configuration retenue. Cette partie est réalisée sous la forme d'une circuit spécifique (ASIC analogique).
— Une partie numérique comprenant l'ensemble des fonctionnalités du neuro-stimulateur dont l'exécution des profils de stimulation, l'ordonnancement des profils, la surveillance du respect des contraintes d'innocuité (limite sur la quantité de charges injectées) vis-à-vis du nerf et de l'électrode, la communication sur l'architecture répartie, etc.

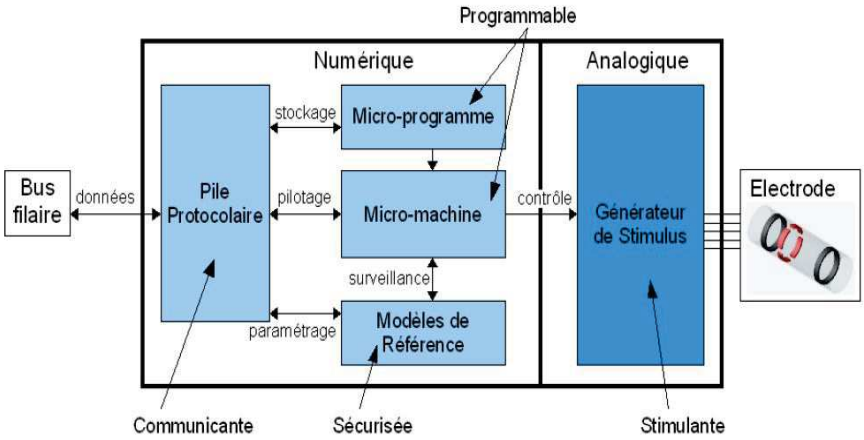


FIGURE 6.18 – L'architecture numérique du neuro-stimulateur

La partie numérique est complexe, bien évidemment critique puisqu'elle contrôle l'application d'un courant électrique sur un nerf et elle comprend de nombreuses fonctions réalisées en parallèle. Toutes ces fonctions sont implémentées sur un circuit programmable de type FPGA, sous forme de composants HILECOP interconnectés (Figure 6.18) et qui partagent des données, à travers des zones mémoires dédiées :

— Une micro-machine qui exécute des profils de stimulation décrits sous forme de micro-programmes et pilote en conséquence le générateur de stimulus,
— Un module de surveillance de la stimulation via des modèles de référence qui assurent la détection de toute violation de contrainte en parallèle de la stimulation, telles que la quantité de charges injectées sur le nerf via l'électrode,

— Un séquenceur en charge du séquencement de l'exécution des microprogrammes,
— Une pile protocolaire gérant la communication de l'unité au sein de l'architecture, principalement des échanges avec le contrôleur qui lui transmet des microprogrammes, des requêtes de start/stop, des requêtes de modulation des paramètres de stimulation (amplitude, largeur d'impulsion), des requêtes de modification de la fréquence, des requêtes de test de présence ou d'erreurs, etc. sachant que le stimulateur doit acquitter ou répondre à certaines de ces requêtes.

Nous allons nous intéresser plus particulièrement à la micro-machine, sur laquelle nous avons appliqué nos travaux.

6.2.2 La micro-machine

La micro-machine (MM) est similaire à un processeur à jeu d'instructions 32-bits réduit (et spécifique SEF) qui exécute des profils de stimulation décrits à l'aide de ces instructions sous forme de micro-programmes. Le principe de la MM est d'exécuter les instructions séquentiellement. Une des instructions (Loop) est dédiée aux sauts arrières, pour répéter tout ou partie d'un profil de stimulation un nombre fini de fois. Cette instruction, également utilisée comme instruction de fin du micro-programme, est exécutée en parallèle de la stimulation. En fait, pour éviter toute latence dans l'enchaînement des instructions (et donc respecter très précisément les paramètres temporels du profil de stimulation), la MM gère leur exécution selon un pipeline, i.e. l'instruction suivante est préparée durant l'exécution de l'instruction courante. La MM intègre également la détection d'erreurs dans le micro-programme et réagit aux éventuelles violations de contraintes qui lui sont notifiées par le module de surveillance (modèles de référence).

Le modèle de la MM (présenté Figure 6.19) comprend 100 places and 154 transitions, avec 143 transitions classiques (i.e. ayant un intervalle temporel $[1, 1]$, pour une exécution en 1 ut), 10 transitions temporelles dédiées à des délais (i.e., ayant un intervalle temporel de type $[a, a]$, $a \neq 1$) et 1 transition ayant à la fois un intervalle temporel et une condition associée (i.e. de type $[a, b], a, b \in \mathbb{N}^*$ et donc potentiellement bloquée).

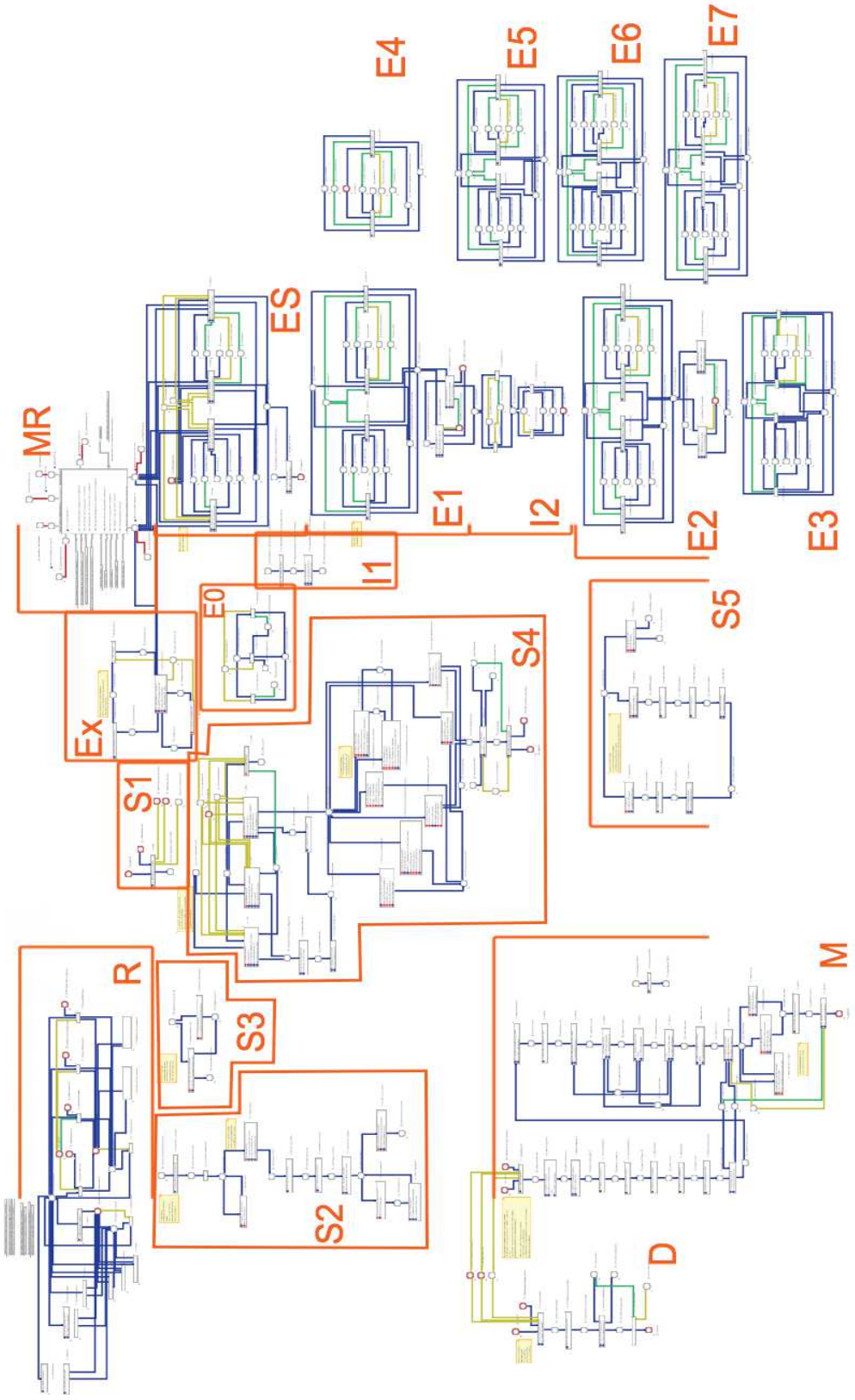


FIGURE 6.19 – Modèle de la micro-machine

6.3 Analyse de la partie numérique

6.3.1 Scénarios et graphes de comportements

La MM peut recevoir différentes requêtes envoyées par le neuro-stimulateur, qui génèrent des comportements différents et indépendants, de la MM. Nous allons donc générer les graphes de comportement synchrones SBGBE pour chacun de ces trois scénarios qui correspondent aux requêtes d'entrée suivantes :

— Start : Le début de stimulation, qui permet de lancer un micro-programme et de l'exécuter.

— Décharge globale : Après plusieurs stimulations, par mesure de sécurité, il est possible de faire une décharge globale de tous les pôles de l'électrode. Cette requête est générée par la partie D du modèle de la figure 6.19.

— Mesure d'impédance : Ce scénario est géré par la partie M du modèle de la MM (cf. figure 6.19). Il permet de mesurer l'impédance de l'électrode afin d'assurer que l'électrode ou le lien avec l'électrode n'est pas endommagé.

Les différents scénarios sont gérés par la partie R du modèle de la MM (cf. figure 6.19), qui active la partie adéquate de la MM, en fonction de la requête reçue.

Les tailles des graphes de comportement générés pour chacun des différents scénarios de la MM sont donnés dans la table 6.2. Tel que mentionné dans les sections précédentes, nous générons les graphes de comportement sous une forme graphique, mais aussi sous une forme textuelle à des fins d'analyse.

TABLE 6.2 – Résultats de la génération de graphes de comportement de MM

	Décharge globale	Mesure d'impédance	Start
SCG Classes/ Changements de classe	577/ 1790	22453/ 52442	22662/ 65654
ISG États/ Changements d'état	508/ 1185	35074/ 64394	26417/ 65389
SBGB États/ Changements d'état	295/ 1107	19552/ 32902	6551/ 15363

6.3.2 Vérification des propriétés

Nous n'allons pas ici détailler la totalité des propriétés que nous avons analysées, mais nous ferons le focus sur 6 propriétés pertinentes du point de vue des exigences fonctionnelles (ce que doit faire le système numérique), et qui illustrent différents cas d'analyse.

Détection des erreurs sur la partie "mesure impédance"

La partie du graphe qui doit détecter les erreurs lors d'une mesure impédance est séparée en 8 cas possibles, représentée sur le modèle par 8 transitions. Un zoom sur cette partie du modèle de la MM est donné dans la figure 6.20. Chaque transition est tirable à partir de la même place *p_ Unicite_ Erreur*, lorsque la place *p_ Erreur_ detectee_ Mise_ off* est marquée. Les 8 transitions représentent 8 situations différentes du stimulateur (en fonction du marquage de place reflétant son état interne, par exemple *p_ Attente_ Instruction*, *p_ Instruction_ suivante_ prete* ou *p_ Pipeline_ Vide*). La propriété vérifiée ici est une "simple" propriété de vivacité. Attention, l'analyse par scénario nous impose une analyse fine des résultats de vivacité : on veut vérifier que toutes les transitions devant être tirées dans ce scénario sont effectivement tirées. L'analyse de vivacité peut (devra) également être faite de manière globale, en construisant le graphe de l'ensemble des scénarios possible de la MM et en vérifiant la vivacité de chacune des transitions. Mais les analyses étant effectuées pour l'instant à la main sur le fichier texte du graphe, nous n'avons pour l'instant pas effectué d'analyse globale.

L'analyse de vivacité de ce scénario de mesure d'impédance a détecté que 4 de ces 8 transitions ne sont pas franchissables. Cela s'explique par la raison suivante. Dans un souci de prise en compte exhaustive des situations lors desquelles l'erreur peut se produire, l'ingénieur qui a conçu le modèle a structurellement représenté tous ces cas, sachant qu'il n'avait pas à sa disposition le mécanisme de macroplace. Dès lors, cette exhaustivité comprend des situations impossibles, même si non bloquantes au sens deadlock (disons que ces situations sont inutiles). La mise en évidence de ces situations impossibles (inutiles) a été rendue possible grâce à notre approche qui prend en considération certes le temps, mais aussi l'interprétation associée. Nous sommes donc en mesure de prouver et expliquer à l'ingénieur que sur les 8 cas qu'il a prévus seulement

4 sont pertinents et les 4 autres peuvent être enlevés (puisque inutiles). Nous pouvons d'ailleurs souligner l'apport de la macroplace qui, si elle avait été disponible à l'ingénieur au moment de la conception de ce modèle, aurait évité ce problème par une gestion plus rigoureuse des exceptions (erreurs dans le cas présent).

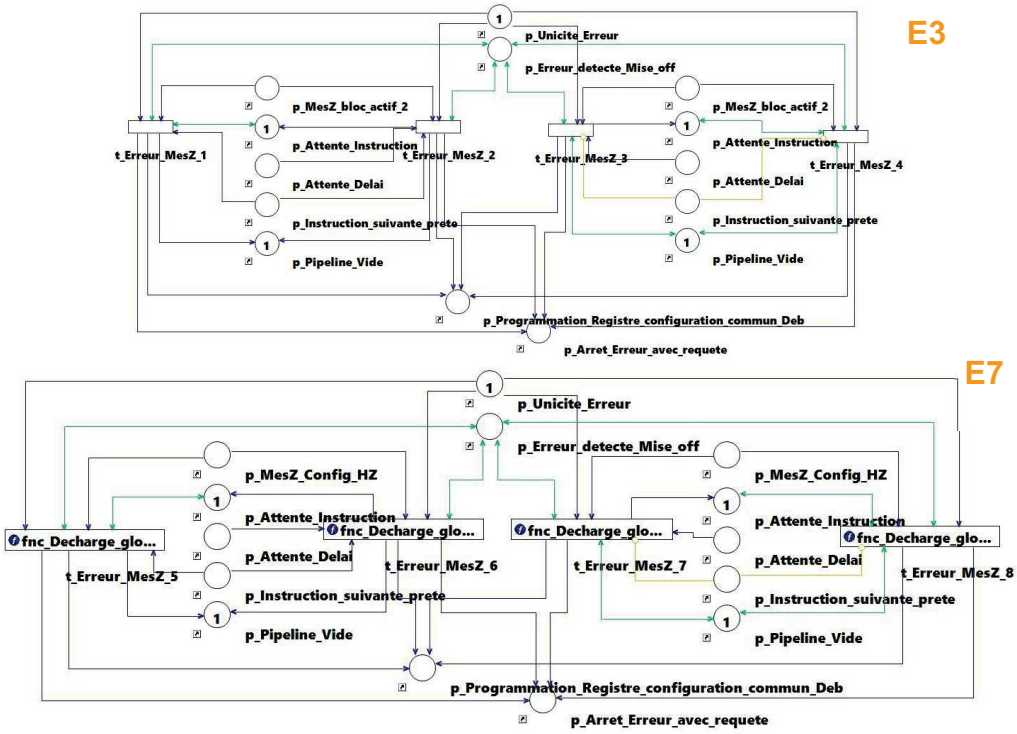


FIGURE 6.20 – Zoom sur la Partie E3 et E7 de MM

Concernant ce scénario de mesure d'impédance, une autre propriété doit être également vérifiée dans la gestion des erreurs. Il faut s'assurer que l'apparition d'un jeton dans la place $p\_Erreur\_detectee\_Mise\_off$ entraîne bien dans tous les cas l'arrêt de la mesure d'impédance (ce qui correspond à la présence d'un jeton dans la place p_MM_Off). Pour cela, il faut qu'à chaque fois qu'un jeton arrive dans $p\_Erreur\_detecte\_Mise\_off$, l'enchaînement des transitions (la séquence) conduit dans tous les cas à un jeton dans p_MM_Off. Cette propriété pourrait s'exprimer aisément en logique temporelle LTL ou CTL*. Dans notre cas malheureusement, elle nécessite l'analyse fine "à la main" de toutes les traces existantes à partir d'un état dont le mar-

quage contient $p\_Erreur\_detectee\_Mise\_off$, pour vérifier pour chacune l'atteignabilité d'un état dont le marquage contient p_MM_Off. L'analyse effectuée a montré que cette propriété était bien respectée.

Fonctionnement exclusif du pipeline :

Le pipeline est une partie du graphe de la micro-machine qui permet de pré-charger une instruction pendant que la précédente est en exécution. Cela permet d'accélérer le décodage et d'éviter une latence dans l'exécution séquentielle des instructions (voir la figure **??**). Dans le modèle de ce pipeline les places représentent les différentes phases de décodage des instructions. Un jeton circulant d'une place à l'autre représente donc une instruction en train d'être décodée. Lorsque le pipeline est inoccupé, la place $p\_Pipeline\_Vide$ est marquée. Comme le pipeline ne gère qu'une seule instruction à la fois, il faut vérifier qu'il doit y avoir obligatoirement un jeton et un seul dans l'ensemble des places représentant le pipeline. Cette propriété pourrait facilement être vérifiée par une propriété logique sur les marquages des places du pipeline. Dans notre cas, l'analyse a été faite par une analyse couplée de l'accessibilité du marquage et des traces (relativement simple ici) du pipeline ; cela a confirmé que cette propriété est bien vérifiée.

Fonctionnement exclusif de l'exécuteur

L'exécuteur est une partie du graphe de la micro-machine qui permet d'exécuter une instruction, et de piloter en conséquence l'ASIC de stimulation (cf. figure 6.18). De même que pour le pipeline, cette exécution doit être unique, au sens d'une instruction à la fois (i.e. séquence de pilotage de signaux des l'ASIC). Il faut donc vérifier qu'il doit y avoir un seul jeton (jeton obligatoirement présent et un seul jeton maximum) dans l'ensemble des places impliquées dans cet exécuteur. Cette propriété a été vérifiée comme pour la propriété précédente.

Enchaînement de l'exécution d'instructions

L'exécution de la micro-machine va correspondre au décodage d'instructions successives. Il est donc nécessaire que, lorsque le décodage d'une instruction se termine, la micro-machine retourne dans un état lui permettant de dé-

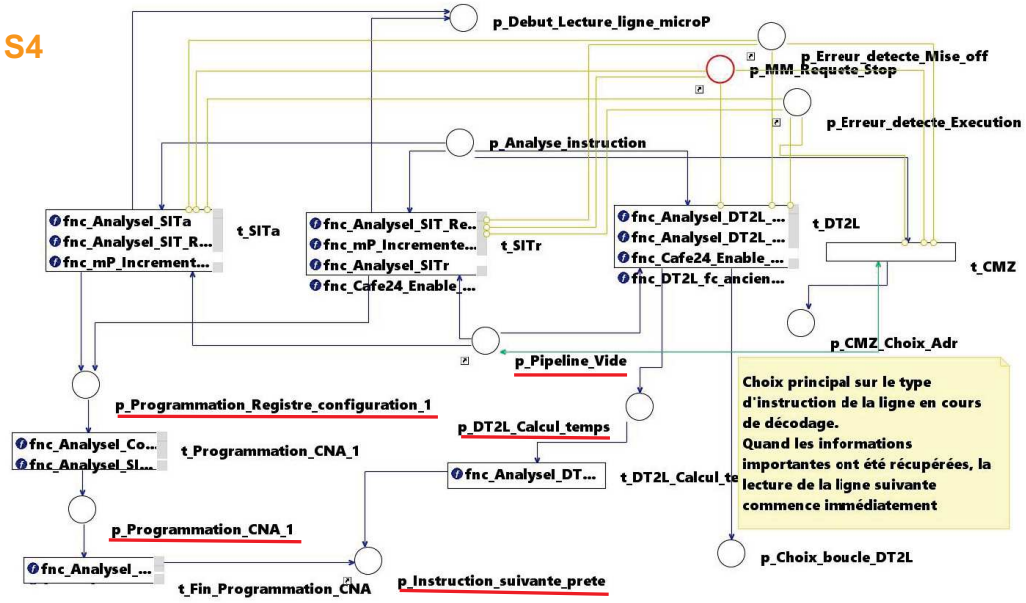


FIGURE 6.21 – Zoom sur le pipeline (partie S4) du modèle de la MM

coder la suivante. Il est difficile de faire un zoom sur la partie du modèle gérant le cycle complet de décodage des instructions, car il est dispersé sur différentes parties du modèle. Nous avons cependant le début sur le zoom de la figure 6.21 : la place *p_Analyse_Instruction* est l'élément central de la gestion du cycle. Suivant le type d'instruction à décoder, plusieurs transitions sont tirables (4 sont montrées figure 6.21 : *t_SITa*, *t_SITr*, *t_DT2L* et *t_CMZ*), chacune lançant un cycle de décodage puis d'exécution de l'instruction correspondante. La propriété que l'on désire vérifier est qu'à chaque début de cycle correspond une fin de cycle faisant revenir dans l'état de début de cycle. Nous avons partiellement vérifié cette propriété sachant que la fin d'un cycle est marquée par le tir de la transition *t_lecture_fin_Ligne_microP*. Nous avons alors vérifié que cette transition menait dans tous les cas à l'état correspondant au début du décodage (i.e., marquage de *p_Analyse_Instruction*). Pour ce faire, tous les changements d'état pour lesquels la transition *t_lecture_fin_Ligne_microP* est tirée sont considérés, en vérifiant que la place *p_Analyse_Instruction* est marquée dans l'état atteint suite à ce tir. Cette analyse a confirmé la propriété.

Réactivité à l'occurrence d'une erreur

Cette propriété concerne le scénario `start` : on veut vérifier qu'à partir du moment où une erreur est détectée (un jeton arrive dans la place $p\_Erreur\_detecte\_Mise\_off$), il ne s'écoule pas plus de 5 cycles d'horloge (5 ut) avant que le système ne s'arrête (donc qu'un jeton arrive dans la place p_MM_Off). A savoir que la fréquence de l'horloge sur notre circuit est égale à $1MHz$, ce qui génère un pas de temps (unité de temps) de notre modèle égal à 1 microseconde. Cette propriété permet donc de vérifier que la réactivité de notre système est de l'ordre de $5s$ [2]. Cette propriété est une propriété impliquant une mesure du temps quantitatif, ce qui est beaucoup plus complexe que les propriétés plus classiques. Dans notre cas, cela implique : 1) de détecter l'occurrence du tir de la transition de détection de l'erreur ; 2) d'analyser toutes les traces commençant par ces transitions pour vérifier qu'elles mènent toutes à un état contenant la place p_MM_Off marquée ; 3) et de calculer le pire temps d'exécution de toutes ces traces afin de vérifier qu'il n'existe pas un cas plus long que la borne imposée de $5ut$. Ainsi, cette propriété n'a pas pu être vérifiée sur notre modèle en raison du nombre important d'états du SBGB où cette place est marquée (plus de 300 états), chacun menant à plus d'un dizaine de traces différentes.

Ce chapitre conclut notre contribution qui porte sur la génération du graphe de comportement pour les réseaux de Petri exécutés de manière synchrone. Nous avons présenté l'intégration de nos travaux dans l'outil HILECOP et nous avons généré le SBGB d'un modèle industriel, en occurrence la micro-machine d'un dispositif médical implantable actif, un neuro-stimulateur. Même si cet exemple ne comprenait pas de macroplace et d'exception à gérer, un ensemble de propriétés pertinentes du point de vue de la "réalité" du modèle (exigences fonctionnelles) est analysé avec une confiance renforcée dans les résultats grâce à la conformité de notre graphe avec l'exécution synchrone. Nous avons également souligné que, évidemment, une vérification de propriétés impose de disposer d'un *model checker* pouvant exploiter toutes les caractéristiques de notre graphe. Les résultats obtenus par une analyse "à la main" confirment

2. Pour comparaison, une purge asynchrone des places d'un raffinement de macroplace prend 27 nanosecondes

néanmoins la pertinence de notre approche. Ceci étant, pour pousser plus loin la prise en compte des propriétés dites non-fonctionnelles, et donc plus de conformité avec la réalité, il faut aller plus loin dans la prise en compte des spécificités issues de l'affectation des horloges, de la projection et du déploiement de l'architecture numérique sur l'architecture matérielle. Autant de points que nous allons évoquer dans les perspectives.

Chapitre 7

Conclusion & Perspectives

7.1 Conclusion

Ces travaux se sont situés dans le cadre de la méthodologie HILECOP qui, basée sur les concepts d'ingénierie dirigée par les modèles, est dédiée à la conception formelle de système numériques critiques. HILECOP permet de décrire un système numérique complexe critique à travers une architecture et un comportement. Pour faire face à la complexité du système, l'architecture est basée sur un modèle à composants ; elle est dès lors décrite par un assemblage de composants (interconnexion des composants via les ports de leurs interfaces). Les composants ont un comportement formalisé par réseaux de Petri. L'assemblage des composants donne naissance au modèle comportemental de l'architecture. Il est alors possible d'effectuer une analyse formelle, via des outils mathématiques d'analyse liés aux réseaux de Petri, tant du comportement de chaque composant que du comportement du système numérique complet. Dans notre contexte, ces systèmes numériques sont mis en oeuvre sur des circuits électroniques de type ASIC ou FPGA. Pour ce faire, HILECOP intègre également la génération du code VHDL pour implémentation (implantation) sur le circuit. Ce code, décrivant l'architecture électronique numérique, donne lieu à une exécution synchrone sur le circuit.

Nous avons donc d'une part un modèle de comportement par essence asynchrone (les réseaux de Petri) et d'autre part une exécution synchrone sur la cible. Là est notre préoccupation : étudier la prise en compte des caractéristiques d'implémentation et d'exécution (qualifiées parfois de propriétés non-fonctionnelles) dans l'analyse formelle du modèle. Et ce, dans le cadre de la méthodologie HILECOP.

Jusque là, l'analyse formelle du modèle était réalisée avec des approches (et outils) asynchrones via des transformations de modèles [60]. Une analyse

asynchrone de modèles à l'évolution synchrone induit un écart entre l'analyse et la réalité (analyse d'un sur-ensemble des comportements). Cet effet est engendré principalement par le fait que ces approches asynchrones représentent le parallélisme par un entrelacement, ce qui génère des états "intermédiaires" inexistants sur la cible. De plus, pour parvenir à une analyse complète, au sens couvrant l'ensemble des spécificités du modèle exécuté (blocage potentiel, gestion d'exception, etc), la transformation de modèles introduisait des modifications structurelles du modèle initial qui par conséquence donnait un graphe de comportement résultant non rigoureusement conforme à la réalité. De fait, la façon de traiter le parallélisme et ces modifications structurelles affectaient la fiabilité de l'analyse, ou tout du moins sont utilité. Par exemple, si un état est indiqué atteignable par l'analyse asynchrone, il n'est pas sûr que cet état le soit dans la réalité (i.e., sur la cible). Suite à ce constat, nous avons proposé une approche d'analyse synchrone. Ainsi, nous avons développé une approche formelle, du formalisme à la génération du graphe de comportement, qui capture et considère l'impact des caractéristiques non-fonctionnelles engendrées par notre contexte d'exécution.

Notre approche consiste à générer un graphe de comportement fidèle à l'évolution d'état synchrone, au sens où nous éliminons tous les états inexistants sur la cible, même si nous passons tout de même par une transformation du modèle initial vers un modèle analysable. Cette transformation ne repose pas seulement sur la prise en compte des caractéristiques de la mise en oeuvre, mais elle considère aussi toutes les particularités relevant de la sémantique utilisée dans les modèles de conception et d'implémentation. Par exemple, les blocages potentiels des transitions et les transitions d'exception.

Une nouvelle sémantique est définie pour le nouveau modèle analysable que nous qualifions de réseau de Petri synchrone (STPN) (i.e., obtenu à partir du modèle initial transformé), ainsi que son graphe de comportement, appelé SBG (Sychronous Behavior Graph). Cette sémantique inclut toutes les particularités déjà évoquées afin de mieux représenter les états et évolutions d'états du système. Par exemple, les transitions tirées simultanément (en parallèle) ne sont plus représentées par un entrelacement, mais par un seul changement d'état qui regroupe toutes ces transitions. Il n'y a donc plus d'états avec des

marquages inexistants sur la cible. De plus, les traces temporelles sont préservées sur le graphe de comportement. Cela signifie que nous pouvons a priori calculer, à partir de ce graphe, le temps pris par des scénarios d'exécution. En conclusion, tous les états du graphe d'états sont effectifs, tous les chemins sont effectifs, aucun état n'est "électroniquement" inatteignable (i.e., sur la cible) ni n'est omis.

Nous sommes donc parvenus à gagner en confiance quant à la validité des résultats de l'analyse formelle. Ceci étant, ces travaux doivent être poursuivis pour permettre d'aller plus loin dans la prise en compte des caractéristiques issues de la projection et du déploiement de l'architecture numérique sur l'architecture matérielle.

7.2 Perspectives

Les travaux présentés ont visé à améliorer la méthodologie HILECOP en termes d'analyse formelle. Or, pour la rendre plus efficace et/ou complète, plusieurs points peuvent encore être considérés :

La complexité

Comme pour toute approche énumérative, notre méthode permet de par son exhaustivité la vérification de propriétés sur l'ensemble de l'espace d'états du système. Elle est dont soumise aux inconvénients classiques de ce type de méthode, c'est à dire la complexité de la construction de cet espace. La complexité estimée de notre algorithme est exponentielle, dans le pire cas. Mais la complexité réelle, comme pour toutes les approches de construction de graphes d'états (e.g SCG, ISG, etc), dépend du parallélisme réel du système, c'est à dire du nombre de transitions tirables en parallèles dans chaque état. Dans notre cas, certaines situations de parallélisme sont supprimées de par l'implémentation synchrone, et les incertitudes de tirs sont limitées par une sémantique impérative (tirable = immédiatement tirée). Cependant, des situations de parallélisme provoquant des entrelacements subsistent notamment à cause de la présence de l'interprétation. Lorsque les transitions ont un intervalle temporel associé à une condition, ou une condition seule (i.e. les transitions dont l'intervalle résiduel est de type [1,b], [a,b] ou [1,+∞[), l'incertitude liée à la valeur

de la condition oblige à considérer tous les cas dans l'analyse. Un exemple simple d'entrelacements dans notre sémantique est donné Figure 7.1 : pour 3 transitions en parallèle avec un intervalle de tir $[1, +\infty[$, le SBG représente toutes les combinaisons de tir de ces 3 transitions, ce qui engendre 8 états et 27 changements d'états.

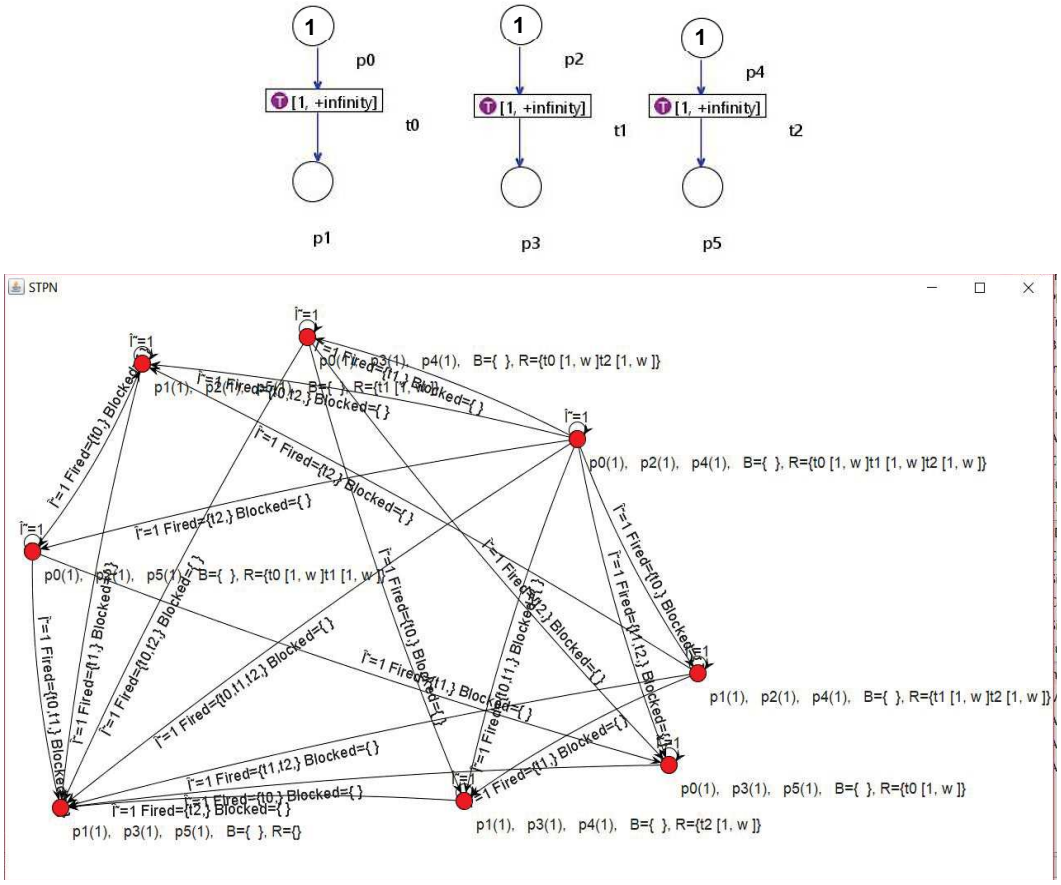


FIGURE 7.1 – Exemple d'entrelacement dans un SBG

Même si dans les systèmes que nous considérons, à savoir les dispositifs implantables, le parallélisme effectif (i.e., à un instant t) reste limité car le système est nécessairement contraints (limité en ressources), cette limitation reste un élément intéressant à étudier. Pour ce faire, les approches symboliques [69] pour la limitation liée à l'énumération temporelle et les approches par ordre partiel [45, 16] pour l'élimination des entrelacements liés aux tirs probables, pourraient être considérées. Cependant, ces techniques sont majoritairement dédiés aux formalismes non temporels. Nous donnons en annexe nos premières

réflexions basées sur l'adaptation des techniques ordres partiels au contexte temporel.

Le multi-horloge

Dans le cadre de nos travaux nous n'avons considéré qu'une seule et même horloge pour l'ensemble des composants, et donc l'ensemble du modèle, supposant que l'intégralité de l'architecture numérique était implémentée sur un même FPGA et que ses composants relevaient tous de la même horloge. Or, sur un système réel, les différents composants peuvent être implémentés sur différents FPGA ou sur différents domaines d'horloge au sein d'un même FGPA. Cela permet par exemple d'optimiser la consommation globale du système si différentes fonctionnalités s'exécutent à des fréquences différentes. Une synchronisation de ces sous-systèmes peut de fait conduire à un système GALS (Globalement Asynchrone Localement Synchrone). Cela a bien sûr un impact sur le comportement, au sens logico-temporel, et impose de traiter simultanément des phénomènes synchrones dont le temps logique peut être différent, et des phénomènes asynchrones.

Cet impact doit logiquement être pris en compte dans l'analyse. Peu de travaux en analyse formelle, a fortiori en réseaux de Petri, s'intéressent à la fois aux systèmes temps réel et distribués, au sens du nombre d'horloges. Dans [26] par exemple, les auteurs se penchent sur le problème de l'utilisation d'horloges partagées, mais au sein d'un réseau d'automates temporisés. Ce problème pourrait être abordé selon deux sous-problèmes : 1/ gérer les horloges multiples ; 2/ gérer les interactions (transitions) asynchrones. Une première piste pour les horloges multiples serait de s'inspirer de l'approche proposée dans [75]. Cette approche est basée sur les réseaux de Petri étendus avec entrées/sorties, fenêtres de temps, priorités et canaux asynchrones. Les entrées/sorties expriment les interactions avec l'environnement externe qui peuvent être vue comme des interprétations dans notre cas. La dimension temporelle est représentée par une fonction associée aux places et aux transitions. Les relations asynchrones sont exprimées via des canaux asynchrones (*asynchronous channels*) de différents types : les canaux asynchrones simples (*simple asynchronous channels*), les canaux asynchrones acquittés (*acknowledged asynchronous channels*) et les canaux asynchrones de non-sensibilisation (*not-enabled asynchronous channels*). Ces trois canaux servent à établir une communication entre les sous-modèles

d'un réseau de Petri afin de déterminer leur sensibilisation. Un canal, assimilé à une place, est dédié à l' "écoute" d'une interaction provenant d'un autre sous-modèle. L'interaction, dépendante du type de canal, met en jeu des transitions, entre sous-modèles, ayant des fenêtres de temps différentes et relevant potentiellement d'horloges différentes. Via ce concept le auteurs modélisent les systèmes de type GALS (sachant que le temps de transmission n'est pas spécifié, i.e. inclus entre 0 et l'infini).

Ces travaux sont une première piste, cependant la considération des horloges multiples est très différente si les horloges des différents sous-systèmes sont des signaux émanant d'une même horloge physique, et donc en phase (synchrones-synchronisées), ou si ce sont deux sources physiques différentes (synchrones-non-synchronisées). Le premier cas garantit des horloges, donc des compteurs, certes multiples (par exemple, une des horloges est deux fois plus rapide que l'autre) mais avant tout synchrones au sens en phase. De fait la synchronisation des sous-systèmes dans ce cas est simplifiée, cependant le rapport des fréquences est potentiellement source de calculs conséquents. Par contre, si les horloges proviennent de générateurs différents, cette non-synchronisation et le phénomène intrinsèque de dérive des horloges sont deux caractéristiques pouvant poser de vrais problèmes dans la construction de l'espace d'état : évolutions locales de composants ou composites selon des horloges déphasées avec plusieurs combinaisons possibles de positionnement relatif de ces horloges. Ceci doit être globalement considéré, i.e. en prenant bien sûr en compte les relations asynchrones des interactions entre ces "compteurs discrets locaux". La considération couplée de temps asynchrone et synchrone dans un même processus d'analyse est très complexe. Ces interactions de nature asynchrone sont peut-être gérables en temps discret sachant que l'émetteur d'une interaction l'induit nécessairement à un instant discret (en rapport avec sa propre horloge) et que le récepteur la considérera à un autre instant discret (exprimé dans une seconde horloge). L'incertitude entre les positionnements relatifs de ces instants discrets peut éventuellement être exprimée via un intervalle entre ces deux instants, mais cela reste une problématique non triviale en termes de sémantique de modélisation et d'analyse.

Le contrôle des composants

Sur le plan comportemental, il serait intéressant de prendre en compte le contrôle d'activité des composants. En effet, les composants ne doivent pas nécessairement être actifs tout le temps, ce qui permet de minimiser la consommation énergétique. Cette action de contrôle d'activité peut être faite par un autre composant, voire par le composant lui-même ("auto-activation/auto-désactivation"). En termes d'analyse, ce contrôle d'activité d'un ou plusieurs composant(s) / composite(s) doit être reflété sur le graphe de comportement. Leur activité est contrôlable par le biais de l'horloge, selon l'approche classique du *clock gating*. L'impact consiste, au niveau de notre formalisme, à arrêter, voire à réinitialiser, les compteurs de temps (discrets) des transitions qui décrivent le comportement de ce composant et ainsi, leur sensibilisation. Nous pourrions pour cela considérer les réseaux de Petri temporels à *stopwatch* [11], dont une sémantique à temps discret est étudiée dans [70]. Une fois de plus, cela requiert de modifier la sémantique du modèle analysable et d'adapter la génération du graphe de comportement.

Un autre aspect intéressant dans l'approche à composants de HILECOP est la possibilité d'ajouter des ports de contrôle et d'observation (non présents à ce stade). Les ports de contrôle et d'observation d'un composant permettent respectivement de forcer l'état d'un composant dans une situation particulière (pré-spécifiée) ou de signaler (notifier) une situation particulière (pré-spécifiée). Nous pouvons faire l'analogie avec les situations d'entrée et de sortie d'une MP mais dans ce cas, c'est tout le comportement du composant qui est considéré comme une MP, au sens où tout le comportement du composant est considéré et traité comme le raffinement d'une MP. Donc la solution serait de transformer ce type de "fonctionnalité" des composants selon le même type de transformation de modèle que nous effectuons pour traiter une MP, puis de le gérer de la même manière dans le processus d'analyse.

Un point qui mériterait d'être étudié et qui est étroitement lié au précédent, est la profondeur de la hiérarchie. En effet, si l'on admet qu'un composant contienne des composants eux-même comprenant des composants, et que les ports de contrôle soient utilisés, alors cela impose de considérer que les MP peuvent être également "composées", imbriquées au sens hiérarchisées ; ce qui n'est pas le cas actuellement puisque le formalisme ne couvre qu'un seul niveau de hiérarchie.

Le model checking

Nos travaux ont porté sur la génération du graphe de comportement, avec certes pour objectif de pouvoir vérifier et valider formellement les propriétés du système réel. Afin d'exploiter ce graphe pour valider des propriétés, il est nécessaire de développer des outils spécifiques qui pourront extraire des informations d'analyse à partir de ce graphe. Un premier outil pourrait faire l'analyse des propriétés de bases, telle que la vivacité du graphe, l'absence de deadlock, etc.. Pour les propriétés plus complexes, il est nécessaire d'utiliser un *model checker* [29, 85], à savoir un outil qui permet de s'assurer que les propriétés spécifiées sont vérifiées sur le graphe de comportement. Les propriétés à vérifier sont alors formalisées à l'aide de la logique temporelle [38]. La vérification de modèle est une technique totalement automatique et exhaustive (couvrant l'ensemble de tous les états et transitions du graphe). Parmi les vérificateurs de modèles existants, nous pouvons citer l'outil Romeo [67] utilisé pour les automates temporisés et l'outil TINA [15] pour les RdP temporels.

A ce jour, notre graphe et sa sémantique étant différents des autres graphes de comportements utilisés dans la littérature, il n'est pas possible d'exploiter directement des outils existants, comme TINA [15] par exemple. La solution ne consiste pas seulement à transformer notre graphe vers un langage d'entrée d'un outil existant (par exemple, au format .KTZ *Kripke transition systems* pour TINA) puisque les algorithmes de ces outils ne traitent pas toutes les informations portées par notre graphe de comportement. En effet, une adaptation des algorithmes de vérification doit être réalisée afin de prendre en considération quelques particularités de notre graphe dont, par exemple, la sémantique de blocage / déblocage de transitions (qui n'est à notre connaissance pas traitée dans ces algorithmes).

De fait, nous envisageons d'implémenter par la suite, directement dans HILECOP, des algorithmes de vérification de modèles basés sur la logique LTL et/ou CTL, et d'introduire des nouvelles règles logiques pour vérifier les propriétés générales ou spécifiques de notre graphe et sa sémantique.

De la vérification de propriétés à la certification du code généré

Enfin, une perspective plus globale sur la méthodologie HILECOP concerne la dernière transformation de modèles. HILECOP permet de générer automatiquement le code VHDL implémenté sur la cible, à partir du modèle de conception. L'objectif global et le but de ces travaux étant d'assurer que l'analyse est conforme avec la mise en oeuvre sur la cible et d'en garantir le bon fonctionnement, tout le processus de transformation de modèles devrait être couvert par la méthodologie. Notre contribution a amélioré l'analyse formelle (côté abstrait) donc logiquement il faudrait prouver l'équivalence entre le code VHDL généré et les modèles utilisés dans la méthodologie HILECOP (modèle de conception, modèle implémenté et modèle analysable). Cela permettrait de "certifier" le code VHDL et ainsi de valider les étapes du processus de la méthodologie HILECOP.

Bibliographie

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[2] Charles André. Syncharts : A visual representation of reactive behaviors. *Rapport de recherche tr95-52, Université de Nice-Sophia Antipolis*, 1995.

[3] David Andreu, David Guiraud, and Guillaume Souquet. A distributed architecture for activating the peripheral nervous system. *Journal of neural engineering*, 6(2) :026001, 2009.

[4] Tuomas Aura and Johan Lilius. *Time processes for time Petri nets*. Springer, 1997.

[5] Sandie Balaguer, Thomas Chatain, and Stefan Haar. A concurrency-preserving translation from time Petri nets to networks of timed automata. *Formal Methods in System Design*, 40(3) :330–355, 2012.

[6] Darlam Bender, Benoît Combemale, Xavier Crégut, Jean Farines, Bernard Berthomieu, and François Vernadat. Ladder metamodeling and plc program validation through time Petri nets. In *Model Driven Architecture–Foundations and Applications*, pages 121–136. Springer, 2008.

[7] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In *CONCUR'98 Concurrency Theory*, pages 485–500. Springer, 1998.

[8] Beatrice Bérard, Franck Cassez, Serge Haddad, Didier Lime, and Olivier H. Roux. Comparison of the expressiveness of timed automata and time Petri nets. In *Proc. of the Third International Conference in Formal Modeling and Analysis of Timed Systems, FORMATS 2005*, pages 211–225, Uppsala, Sweden, September 26-28 2005. Springer Berlin Heidelberg.

[9] Béatrice Berard, Franck Cassez, Serge Haddad, Didier Lime, and Olivier Henri Roux. The Expressive Power of Time Petri Nets. *Theoretical Computer Science*, 474 :1–20, 2013.

[10] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE transactions on software engineering*, 17(3) :259–273, 1991.

[11] Bernard Berthomieu, Didier Lime, Olivier H. Roux, and François Vernadat. Reachability problems and abstract state spaces for time Petri nets with stopwatches. *Discrete Event Dynamic Systems*, 17(2) :133–158, 2007.

[12] Bernard Berthomieu and Miguel Menasche. An enumerative approach for analyzing time Petri nets. In *Proceedings IFIP*. Citeseer, 1983.

[13] Bernard Berthomieu, F Vernadat, and SD Zilio. Tina : Time Petri net analyzer.

[14] Bernard Berthomieu and François Vernadat. State class constructions for branching analysis of time Petri nets. In *TACAS*, volume 2619, pages 442–457. Springer, 2003.

[15] Bernard Berthomieu and Francois Vernadat. Time Petri nets analysis with tina. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 123–124. IEEE, 2006.

[16] Hanifa Boucheneb and Kamel Barkaoui. Covering steps graphs of time Petri nets. *Electronic Notes in Theoretical Computer Science*, 239 :155–165, 2009.

[17] Hanifa Boucheneb, Kamel Barkaoui, and Karim Weslati. Delay-dependent partial order reduction technique for time Petri nets. In *Formal Modeling and Analysis of Timed Systems*, pages 53–68. Springer, 2014.

[18] Hanifa Boucheneb, Guillaume Gardey, and Olivier H. Roux. Tctl model checking of time Petri nets. *J. Log. and Comput.*, 19(6) :1509–1540, 2009.

[19] Hanifa Boucheneb and Hind Rakkay. A more efficient time Petri net state space abstraction useful to model checking timed linear properties. *Fundamenta Informaticae*, 88(4) :469–495, 2008.

[20] JL Boussin. Synthesis and analysis of logic automation systems. *IFAC Proceedings Volumes*, 11(1) :1527–1535, 1978.

[21] Marc Boyer and Olivier H. Roux. On the compared expressiveness of arc, place and transition time Petri nets. *Fundam. Inform.*, 88(3) :225–249, 2008.

[22] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M Scott Marshall. Graphml progress report structural layer proposal. In *International Symposium on Graph Drawing*, pages 501–512. Springer, 2001.

[23] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8) :677–691, 1986.

[24] Joakim Byg, Morten Jacobsen, Lasse Jacobsen, Kenneth Yrke Jørgensen, Mikael Harkjær Møller, and Jiří Srba. Tctl-preserving translations from timed-arc Petri nets to networks of timed automata. *Theoretical Computer Science*, 537 :3–28, 2014.

[25] Franck Cassez and Olivier H. Roux. Structural translation from time Petri nets to timed automata. *The Journal of Systems and Software*, 79 :1456–1468, 2006.

[26] Thomas Chatain and Sandie Balaguer. Avoiding shared clocks in networks of timed automata. *Logical Methods in Computer Science*, 9, 2013.

[27] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3) :178–187, 1978.

[28] Feng Chu. *Conception des systèmes de production à l'aide des réseaux de Petri : vérification incrémentale des propriétés qualitatives*. PhD thesis, Metz, 1995.

[29] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[30] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[31] Davide D'Aprile, Susanna Donatelli, Arnaud Sangnier, and Jeremy Sproston. From time Petri nets to timed automata : An untimed approach. In *Proc. of the 13th International Conference of Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007*, pages 216–230, Braga, Portugal, March 24 - April 1 2007. Springer Berlin Heidelberg.

[32] René David and Hassane Alla. Petri nets and grafcet. *Tools for modelling discrete event systems*, 1992.

[33] René David and Hassane Alla. *Discrete, continuous, and hybrid Petri nets*. Springer Science & Business Media, 2010.

[34] Michel Diaz. *Vérification et mise en oeuvre des réseaux de Petri*. Hermès Science, 2003.

[35] Daniel P Dolata and Robert E Carter. Wizard : applications of expert system techniques to conformational analysis. 1. the basic algorithms exemplified on simple hydrocarbons. *Journal of chemical information and computer sciences*, 27(1) :36–47, 1987.

[36] Francois-Xavier Dormoy. Scade 6 : a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08)*, pages 1–9, 2008.

[37] Catherine Dufourd, Alain Finkel, and Ph Schnoebelen. Reset nets between decidability and undecidability. *Automata, Languages and Programming*, pages 103–115, 1998.

[38] E Allen Emerson and Chin-Laung Lei. Modalities for model checking (extended abstract) : branching time strikes back. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1985.

[39] Mohammad Esmail Esmaili, Reza Entezari-Maleki, and Ali Movaghar-Rahimabadi. Improved region-based tctl model checking of time Petri nets. *Journal of Computing Science and Engineering, JCSE*, 9 :9–19, 2015.

[40] Georg Frey and Lothar Litz. Verification and validation of control algorithms by coupling of interpreted Petri nets. In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, volume 1, pages 7–12. IEEE, 1998.

[41] Guillaume Gardey, Didier Lime, Morgan Magnin, et al. Romeo : A tool for analyzing time Petri nets. In *International Conference on Computer Aided Verification*, pages 418–423. Springer, 2005.

[42] Guillaume Gardey, Olivier Roux, and Olivier Roux. Using zone graph method for computing the state space of a time Petri net. *Formal modeling and analysis of timed systems*, pages 246–259, 2004.

[43] Guillaume Gardey, Olivier H Roux, and Olivier F Roux. Using zone graph method for computing the state space of a time Petri net. In

Formal modeling and analysis of timed systems, pages 246–259. Springer, 2003.

[44] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Computer-Aided Verification*, pages 176–185. Springer, 1990.

[45] Patrice Godefroid, J Van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems : an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.

[46] Iwona Grobelna and Marian Adamski. Model checking of control interpreted Petri nets. In *Proc. of the 18th Int. Conf. Mixed Design of Integrated Circuits and Systems (MIXDES 2011)*, Gliwice, Poland, June 2011.

[47] Iwona Grobelna and Marian Adamski. Model checking of control interpreted Petri nets. In *18th International Conference Mixed Design of Integrated Circuits and Systems, (MIXDES 2011)*, Gliwice, Poland, 2011.

[48] Serge Haddad, Fabrice Kordon, and Laure Petrucci. Méthodes formelles pour les systèmes répartis et coopératifs, 2006.

[49] Rachid Hadjidj and Hanifa Boucheneb. On-the-fly tctl model checking for time Petri nets. *Theor. Comput. Sci.*, 410(42) :4241–4261, 2009.

[50] John Håkansson and Paul Pettersson. *Partial order reduction for verification of real-time components*. Springer, 2007.

[51] David Harel and Michal Politi. *Modeling reactive systems with statecharts : the STATEMATE approach*. McGraw-Hill, Inc., 1998.

[52] Lom-Messan Hillah, Fabrice Kordon, Laure Petrucci, and Nicolas Treves. Pnml framework : An extendable reference implementation of the Petri net markup language. In *Petri Nets*, volume 6128, pages 318–327. Springer, 2010.

[53] Peter Huber, Arne M. Jensen, Leif O. Jepsen, and Kurt Jensen. Reachability trees for high-level Petri nets. In *High-Level Petri Nets*, pages 319–350. Springer, 1986.

[54] Jean-Marc Jézéquel, Benoit Combemale, and Didier Vojtisek. *Ingénierie Dirigée par les Modèles : des concepts à la pratique...* Ellipses, 2012.

[55] Zhihao Jiang, Miroslav Pajic, Rajeev Alur, and Rahul Mangharam. Closed-loop verification of medical devices with model abstraction and refinement. *International Journal on Software Tools for Technology Transfer*, 16(2) :191–213, 2014.

[56] Peter Jordan. Standard iec 62304-medical device software-software life-cycle processes. 2006.

[57] Andrei Karatkevich. Stubborn set method for interpreted Petri nets. *IFAC Proceedings Volumes*, 39(17) :227–232, 2006.

[58] Andrei Karatkevich. *Dynamic analysis of Petri net-based discrete systems*, volume 356. Springer Science & Business Media, 2007.

[59] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1 :134–152, 1997.

[60] 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.* PhD thesis, Université Montpellier 2, 2014.

[61] Hélène Leroux, David Andreu, and Karen Godary-Dejean. Handling exceptions in Petri net-based digital architecture : from formalism to implementation on fpgas. *IEEE Transactions on Industrial Informatics*, 11(4) :897–906, 2015.

[62] Helene Leroux, Karen Godary-Dejean, and David Andreu. Complex digital system design : A methodology and its application to medical implants. In *Proc. of the International Workshop on Formal Methods for Industrial Critical Systems, FMICS 2013*, pages 94–107. Springer, 2013.

[63] Hélène Leroux, Karen Godary-Dejean, and David Andreu. Integrating implementation properties in analysis of Petri nets handling exceptions. *IFAC Proceedings Volumes*, 47(2) :406–411, 2014.

[64] Helene Leroux, Karen Godary-Dejean, Guillaume Coppey, and David Andreu. Automatic handling of conflicts in synchronous interpreted time Petri nets implementation. In *VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on*, pages 100–105. IEEE, 2014.

[65] Y Edmund Lien. Termination properties of generalized Petri nets. *SIAM Journal on Computing*, 5(2) :251–265, 1976.

[66] Johan Lilius. Efficient state space search for time Petri nets. *Electronic Notes in Theoretical Computer Science*, 18 :113–133, 1998.

[67] Didier Lime, Olivier H Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo : A parametric model-checker for Petri nets with stopwatches. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 54–57. Springer, 2009.

[68] Didier Lime and Olivier Henri Roux. State class timed automaton of a time Petri net. In *The 10th International Workshop on Petri Nets and Performance Models,(PNPM'03)*, pages 124–133, 2003.

[69] Morgan Magnin, Didier Lime, et al. Symbolic state space of stopwatch Petri nets with discrete-time semantics (theory paper). In *International Conference on Applications and Theory of Petri Nets*, pages 307–326. Springer, 2008.

[70] Morgan Magnin, Pierre Molinaro, and Olivier (H.) Roux. Expressiveness of Petri nets with stopwatches. discrete-time part. *Fundam. Inf.*, 97(1-2) :139–176, 2009.

[71] Antoni Mazurkiewicz. Trace theory. In *Petri nets : applications and relationships to other models of concurrency*, pages 278–324. Springer, 1986.

[72] Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[73] Philip Merlin and David Farber. Recoverability of communication protocols–implications of a theoretical study. *IEEE transactions on Communications*, 24(9) :1036–1043, 1976.

[74] M Moalla, Jacques Pulou, and Joseph Sifakis. Synchronized Petri nets : A model for the description of non-autonomous sytems. *Mathematical Foundations of Computer Science 1978*, pages 374–384, 1978.

[75] Filipe Moutinho and Luís Gomes. *Distributed embedded controller development with Petri nets : application to globally-asynchronous locally-synchronous systems*, volume 150. Springer, 2015.

[76] Pierre-Alain Muller and Nathalie Gaertner. *Modélisation objet avec UML*, volume 514. Eyrolles Paris, 2000.

[77] Tadao Murata. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989.

[78] UML ObjectAid. Explorer for eclipse, 2014.

[79] Yasukichi Okawa and Tomohiro Yoneda. Symbolic computation tree logic model checking of time Petri nets. *Electronics and Communications in Japan (Part III : Fundamental Electronic Science)*, 80(4) :11–20, 1997.

[80] James L Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3) :223–252, 1977.

[81] Carl Adam Petri. Communication with automata. 1966.

[82] Louchka Popova-Zeugmann. On time Petri nets. *Elektronische Informationsverarbeitung und Kybernetik*, 27(4) :227–244, 1991.

[83] Louchka Popova-Zeugmann. *Essential states in time Petri nets*. Humboldt-Univ. zu Berlin, 1998.

[84] Louchka Popova-Zeugmann. Time Petri nets. In *Time and Petri Nets*, pages 31–137. Springer, 2013.

[85] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.

[86] Antonio Ramírez-Treviño, Israel Rivera-Rangel, and Ernesto López-Mellado. Observability of discrete event systems modeled by interpreted Petri nets. *IEEE Transactions on Robotics and Automation*, 19(4) :557–565, 2003.

[87] Laura Recalde and Manuel Silva. Pn fluidification revisited : Semantics and steady state. *J. Zaytoon S. Engell, Automation of Mixed Processes : Hybrid Dynamics Systems*, pages 279–286, 2000.

[88] Olivier H Roux, David Delfieu, and Pierre Molinaro. Discrete time approach of time Petri nets for real-time systems analysis. In *Emerging Technologies and Factory Automation, 2001. Proceedings. 2001 8th IEEE International Conference on*, volume 2, pages 197–204. IEEE, 2001.

[89] Olivier Henri Roux and Claude Jard. Approches formelles des systèmes embarqués communicants, 2008.

[90] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[91] Moe Shahdad, Roger Lipsett, Erich Marschner, Kellye Sheehan, and Howard Cohen. Vhsic hardware description language. *Computer*, 2(18) :94–103, 1985.

[92] Manuel Silva, José Manuel Colom, Jorgé Julvez, Cristian Mahulea, Jan H van Schuppen, Rong Su, Jan Komenda, Jörg Raisch, Stephanie Geist, and Philippe Darondeau. On modelling of hierarchical and distributed discrete-event systems. *Manuel Silva, José Manuel Colom, Jorgé Julvez, Cristian Mahulea, Jan H. van Schuppen, Rong Su, Jan Komenda, Jörg Raisch8, Stephanie Geist, Philippe Darondeau//The DISC Project Perspective*, page 85, 2007.

[93] Manuel Silva and Laura Recalde. On fluidification of Petri nets : from discrete to hybrid and continuous models. *Annual Reviews in Control*, 28(2) :253–266, 2004.

[94] Robert H Sloan and Ugo Buy. Reduction rules for time Petri nets. *Acta Informatica*, 33(7) :687–706, 1996.

[95] Guillaume Souquet, David Andreu, and David Guiraud. Petri nets based methodology for communicating neuroprosthesis design and prototyping. In *ISABEL'08 : 1st International Symposium on Applied Sciences in Biomedical and Communication Technologies*, 2008.

[96] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF : eclipse modeling framework*. Pearson Education, 2008.

[97] Paulo Tabuada and George J Pappas. Linear time logic control of discrete-time linear systems. *IEEE Transactions on Automatic Control*, 51(12) :1862–1877, 2006.

[98] Yann Thierry-Mieg, Béatrice Bérard, Fabrice Kordon, Didier Lime, and Olivier H Roux. Compositional analysis of discrete time Petri nets. In *1st workshop on Petri nets compositions (CompoNet 2011)*, volume 726, pages 17–31. CEUR, 2011.

[99] Kimon P Valavanis. On the hierarchical modeling analysis and simulation of flexible manufacturing systems with extended Petri nets. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(1) :94–110, 1990.

[100] Antti Valmari. Error detection by reduced reachability graph generation. In *Proceedings of the 9th European Workshop on Application and Theory of Petri Nets*, pages 95–112, 1988.

[101] Antti Valmari. Stubborn sets for reduced state space generation. In *International Conference on Application and Theory of Petri Nets*, pages 491–515. Springer, 1989.

[102] Moshe Y Vardi. Branching vs. linear time : Final showdown. In *TACAS*, volume 1, pages 1–22. Springer, 2001.

[103] François Vernadat, Pierre Azéma, and François Michel. Covering step graph. In *Application and theory of Petri nets 1996*, pages 516–535. Springer, 1996.

[104] François Vernadat and François Michel. Covering step graph preserving failure semantics. In *Application and Theory of Petri Nets 1997*, pages 253–270. Springer, 1997.

[105] I. Virbitskaite and E. Pokozy. A partial order method for the verification of time Petri nets. In *Proc. of the 12th International Symposium in Fundamentals of Computation Theory, FCT'99*, pages 547–558, Iaşi, Romania, August 30 - September 3 1999. Springer Berlin Heidelberg.

[106] Lars Vogel. Eclipse rcp tutorial. *Retrieved May*, 20 :2012, 2012.

[107] Florian Wagner, Philipp Münch, Steven Liu, and Georg Frey. Development process for dependable high-performance controllers using Petri nets and FPGA technology. In *1st IFAC Workshop on Dependable Control of Discrete Systems, (DCDS 2007)*, volume 40, pages 139–144, 2007.

[108] Florian Wagner, Philipp Münch, Steven Liu, and Georg Frey. Development process for dependable high-performance controllers using Petri nets and FPGA technology. In *1st IFAC Workshop on Dependable Control of Discrete Systems, (DCDS 2007)*, volume 40, pages 139–144, 2007.

[109] Jiacun Wang. Time Petri nets. In *Timed Petri Nets*, pages 63–123. Springer, 1998.

[110] Jens Weber and Isabelle Perseil. *Foundations of Health Information Engineering and Systems : Second International Symposium, FHIES 2012, Paris, France, August 27-28, 2012. Revised Selected Papers*, volume 7789. Springer, 2013.

[111] Pierre Wolper and Patrice Godefroid. Partial-order methods for temporal verification. In *CONCUR'93*, pages 233–246. Springer, 1993.

[112] T. Yoneda and H. Ryuba. Ctl model checking of time Petri nets using geometric regions. *IEICE Transactions on Information and Systems*, 81(3) :297–306, 1998.

Annexe A

State Graph Reduction using Partial Order Semantics for Time Petri Nets (NOT PUBLISHED)

Time Petri Nets are traditionally used for the specification of real time systems, notably for analysis and formal validation. Model checking is a technique which achieves properties verification through an exhaustive analysis of the state graph of the system model. The main limitation of this technique is the state space explosion. In this article, we propose a compact state graph, called the Reduced Graph, which preserves all sequences of transitions firing as well as minimal and maximal elapsed duration of each sequence. To do so, we extend the partial order semantics to define several temporal parallelism relations. According to covering step approach, we compute our reduced graph reducing transitions interleaving, while keeping potential parallelism information.

A.1 Introduction

The formal validation of complex systems that operate in domains like health, nuclear, avionics, etc. is a mandatory phase. It aims at ensuring dependability of the system behaviour. Among formal methods, model checking allows an exhaustive analysis on the set of possible states of the system model. After system modeling, using state machine formalisms (example : Petri Nets (PN), automata, etc.), the first step is to construct the state graph, then it will be analyzed to guaranty that all the properties of the system are satisfied. Formal methods are even more important when considered systems are critical from a real-time point of view. Both modeling formalisms and techniques of state graph construction must be adapted to consider time constraints. In

general, the set of time Petri Nets [73] (TPN for short) states is infinite. Several approaches are proposed in literature [14] [19] [43] to generate finite state space. Among the techniques of State Space Abstraction the main technique is the State Class Graph (SCG) introduced in [12] by Berthomieu and Menasche. It consists in grouping in one equivalent class, all the states obtained by the firing of the same transition from a given state at different firing moments. Even with these methods that could ensure the finiteness, the state graph often increases in an exponential way called *combinatorial explosion*, which makes the analysis very difficult, even impossible. This problem is linked to the intrinsic exhaustive enumeration of the states space.

A.1.1 Related Work

Many approaches are proposed to deal with this problem like the compression techniques [23], the reduction by symmetry [53] and the partial order reduction [111] [103]. These last methods deal with the problem of combinatorial explosion linked to asynchronous treatment of parallelism. i.e, interleaving (the differentiation of the execution order of parallel actions).

Partial order techniques aim at reducing the combinatorial explosion of the state graph size using an independence relation [71], which is based on the following idea *if two actions can be executed in any order and the system arrives to the same final state then these two actions are independent*. According to the way of exploiting this relation, these techniques can be classified into two approaches : (1) The approaches called *persistent set* that aim to remove the interleaving by the exploration of one single path among all paths that have the same equivalent trace [111] [100] [44] ; (2) The approaches called *covering steps*, which consist in regrouping the independent transitions in one single transition step. Every transition step is considered like an atomic action [103] [104].

The application of partial order methods on untimed models can be efficient. On the contrary, with time models, these methods are not perfectly suitable. The main problem comes from the temporal dependency between transitions firing. For example, it is possible that two actions are structurally independent while being linked because of time constraints addition, which drives to different clock values by interleaving. Thus, the final state for the

two interleaving paths will not be the same. In this case, classical partial order methods cannot be applied directly.

To solve these limitations, many approaches have been proposed, which are grouped into two categories : (1) Approaches based on internal temporal semantics [7] [50] ; they eliminate the implicit synchronization by only ordering the actions that are causally related in the network. Re-synchronization is established when needed, by including an additional clock. The differences between clocks may lead to infinite state space. (2) Approaches based on POSETs (partial ordered sets) [66] [17], to reduce the state graph. Timing information are represented in the form of partially ordered sets. The idea is ignoring some firing orders and exploring only one firing sequence. To highlight the limitations of these approaches, let us introduce the idea of POSETs with the following example.

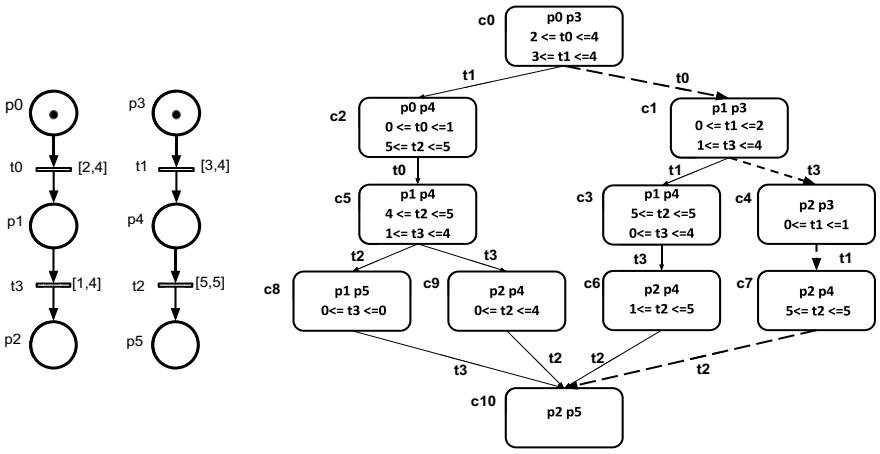


FIGURE A.1 – Example of a TPN and its State Class Graph.

Let's consider the TPN in Figure 1(a) and its SCG in Figure 1(b) , transitions t_0 and t_1 are firable in different orders (different interleavings) from the initial state class c_0. The idea is to explore only one sequence of transitions firing. For that, we choose an arbitrarily one of c_0 output transitions to compute the successor class. Let $Succ(c_0, t_0)$ be the successor class of c_0 by firing t_0. Transitions t_1 and t_3 are firable in both orders from the class $Succ(c_0, t_0)$. Only one transition is chosen arbitrary (t_1 or t_3) to compute the new successor class of $Succ(c_0, t_0)$, and so on. The resulting graph will be in form of a sequence, for example $t_0t_3t_1t_2$ (dotted lines on Figure 1(b)).

The main limit of the POSETs approaches is the consideration of only one scenario of transitions firing, which does not necessarily represent all possible sequences nor firing orders of the complete graph. For example, for the sequence $(t_0t_1t_3t_2)$ that started by firing t_0 before t_1, the transition t_3 must necessarily be fired before t_2. In contrary, for the sequences that start by the firing of t_1 before t_0 ($t_1t_0t_3t_2$ and $t_1t_0t_2t_3$), we will have transitions t_2 and t_3 firable in different orders. In our context where transitions are associated to events significant for the system functioning (shared data), this loss of information can not be acceptable.

A.1.2 Context

In our application context, we aim in active implantable medical devices (AIMD) design and prototyping. More precisely, we design the digital controller part of an implantable neural simulator, detailed in [3]. Its digital architecture is composed of a set of interconnected components (see Figure 2).

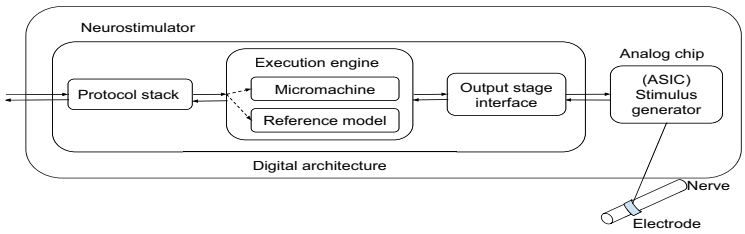


FIGURE A.2 – Schematic Representation of Neurostimulator.

Protocol Stack ensures the communication of the neurostimulator through the distributed stimulation architecture. It receives for example start and stop requests as well as remote modulation ones (online modification of the stimulus amplitude, pulse width, etc.). When a start request is received it launches the execution engine i.e., it activates simultaneously *Micromachine* and *Reference Model* functional blocks. *Micromachine* is in charge of executing the stimulation profile and thus of controlling *Stimulus Generator* (through *output stage interface*). As soon as *Micromachine* is running *Reference Model* monitors the respect of specific constraints like the maximum quantity of injected charges (to preserve both the nerve and the electrode integrity). To do so, *Micromachine* and *Reference Model* functional blocks must share data.

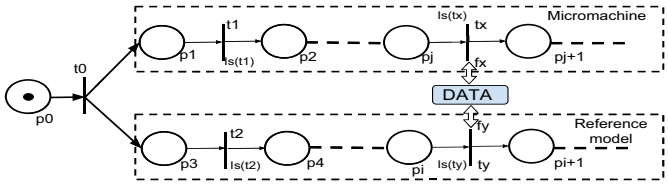


FIGURE A.3 – Schematic Representation of Simultaneously Lunched TPN Sequences.

The behavior of these components is specified by means of interpreted TPN [62]. Schematically, we could represent the behavior of the whole stimulator, as shown Figure 3, by two sequences of transitions (one for *Micromachine* and one for the *Reference Model*) launched in parallel at the instant of a request reception. The parallel execution of these two sequences causes multiple interleavings leading to combinatory explosion of the analysis process. Also, functions could be associated with transitions, manipulating shared data between the two sequences, which requires keeping the information on their firing orders. We thus focus on such a parallel sequences structure.

A.1.3 This Work

Our work is classed as an extension of covering step approaches used for Petri Nets, to parallel sequences of TPNs (see Section A.2.2). The extension of this approach must consider : (1) Time constraints, the direct application of independent relation is less appropriate for TPN due to time constraints addition. These constraints lead to different state space abstraction by interleaving even if transitions are independent. (2) The immediate firing semantics, these approaches consider that two firable transitions are immediately fired in one step. But the introduction of delayed firing leads to others potential parallelisms. In the example of Figure 1(a), we see that t_1 and t_3 could also be in interleaving.

In this paper we propose a new Reduced Graph (RG for short) for TPN with parallel sequences (see Section A.2.2) that preserves all possible sequences of transitions firing while reducing the number of interleavings by representing parallel transitions with one atomic step. Also, we compute the minimum and the maximum time during each step. We construct the reduced graph taking

inspiration from partial order semantics. More precisely, we adapt covering step approach. The rest of the paper is organized as follows : Section A.2 introduces our modeling context, recalls the basic concepts of Time Petri Nets, and partial order foundation. Section A.3 introduces the formal definition of temporal parallelism and the foundation of our approach. Section A.4 devotes to the semantics and the construction of our reduced graph. Section A.6 reports experiment results. Finally, the conclusion and further works are presented in Section A.7.

A.2 Preliminaries

A.2.1 Time Petri Nets

Definitions

Time Petri Nets [73] are Petri Nets with a time interval associated with each transition defining its firing instant.

Let $\mathbb{Q}_+$ and $\mathbb{N}$ be the sets of rational positive and natural numbers, respectively. A Time Petri Net is a tuple $(P, T, Pre, Post, m_0, I_s)$, where P is the set of places, T is the set of transitions $(T \cap P = \emptyset)$, Pre and $Post$ are the functions of pre-conditions and post-conditions $(Pre, Post : P \times T \to \mathbb{N})$, M_0 is the initial marking and $I_s : T \to Q_+$ is a function called static interval associating a static interval $[\downarrow I_s(t), \uparrow I_s(t)]$ to each transition $t \in T$. The boundaries $\downarrow I_s(t)$ and $\uparrow I_s(t)$ are the *static earliest date* and the *static latest firing date*, respectively. For convenience, for a transition t, we use the input set (preset) and the output set (postset) denoted ${}^\bullet t = \{p \in P \mid Pre(p,t) > 0\}$ and $t^\bullet = \{p \in P \mid Post(p,t) > 0\}$.

The marking of a TPN is defined by the function $m : P \to \mathbb{N}$. The transition $t_i \in T$ is enabled by m if its input places have all the required tokens to fire t_i, i.e.,$\forall p \in P, m(p) \geq Pre(p, t_i)$. We denote by $enable(m)$ the set of transitions enabled by m.

TPN Evolution

A TPN state is a pair $e = (m, I)$, where m is the marking and I is the interval function that associates a dynamic firing interval with every transition

enabled by m. We denote by $\downarrow I(t), \uparrow I(t)$ the lower and the upper bounds of the interval $I(t)$, respectively. The initial state e_0 is (m_0, I_0), where $I_0(t) = I_s(t)$, $\forall t \in enable(m_0)$. An enabled transition could be fired only if the time elapse since its enable instant is within its firing interval. The set of firable transitions at state e is denoted by $firable(e)$. A transition $t \in T$ is firable from state $e = (m, I)$ at time $\theta \in Q^+$ iff transition $t \in enable(m)$ and t can be fired and there is no transition that must be fired at an earliest date, i.e. :

$$\downarrow I(t) \leq \theta \leq min(\uparrow I(t), \uparrow I(t')), \forall t' \in enable(m) \tag{A.1}$$

The states of a TPN evolve if the time progresses or if a transition is fired. The state $e' = (m', I')$ is reachable from $e = (m, I)$ by the time progression θ, iff $m = m'$ and $\forall t \in enable(m)$:

$$I'(t) = [max(0, \downarrow I(t) - \theta), \uparrow I(t) - \theta] \tag{A.2}$$

The firing of transition t from state $e = (m, I)$ drives to a new state $e' = (m', I')$ where the new marking is obtained by : $\forall p \in P m'(p) = m(p) - Pre(p, t) + Post(p, t)$, and the new interval function I' by : $\forall t' \in enable(m')$, $I'(t') = I_s(t')$, if t' is newly enabled by the firing of t (i.e. t' was not enabled by m), otherwise $I'(t') = I(t')$.

A.2.2 Parallel Sequences of Transitions

From a modeling point of view, we choose to deal with Safe-TPN represented as a set of two parallel sequences of transitions (denoted SP). Such a model can be formalized by $SP = \{SP_1, SP_2\}$, where $SP_i = (P_i, T_i, Pre, Post, mi_0, I_s)$, $i \in \{1, 2\}$ is a Safe-TPN representing a transitions sequence. Each sequence is an elementary sequential path (no branching), i.e. $\forall p \in P_i, \forall t \in T_i,\ Pre(p, t) \leq 1, Post(p, t) \leq 1$). The assembly of the two sequences give us the global model $SP(P, T, Pre, Post, m_0, I_s)$, where $P = P_1 \cup P_2$, $T = T_1 \cup T_2$ and $m_0 = m1_0 + m2_0$. An example is given on Figure 1(a). Thus, to the previous definitions, for a transition t_i, we add the nth predecessor (denoted t_{i-n}) and the nth successor (denoted t_{i+n}) functions ($T \rightarrow T$). These functions define the relation of the sequential structure of our parallel sequences. Transitions t_{i-1} and t_{i+1} are the direct predecessor and the direct successor of transition

t_i iff $(t^{\bullet}_{i-1} = {}^{\bullet}t_i)$ and $(t^{\bullet}_i = {}^{\bullet}t_{i+1})$, respectively. The nth predecessor and the nth successor are obtained by applying the direct predecessor and the direct successor n time, respectively.

Initial Transitions

The initial transitions (denoted $Init_{SP}$) are the set of transitions that initiate the set of parallel sequences SP. The specific structure of our model guarantees that there is only one initial transition for each parallel sequence. For example, for a set of parallel sequences $SP = (P, T, Pre, Post, m_0, I_s) = \{SP_1, SP_2\}$, transitions $t_i \in T_1$ and $t_j \in T_2$ are initial if they are both enabled by the initial marking : $t_i, t_j \in enable(m_0)$. In all cases, the set of the initial transitions of a set of parallel sequences is supposed to be known.

A.2.3 Partial Order Technique Foundation

Independence Relation[71]

For a system, if two actions are executed in parallel and if their execution (regardless of the order) leads to the same final state, then these two actions are independent. Formally, two actions a_1 and a_2, $a_1 \neq a_2$, are independent iff :

$$\forall s, s_{a_1}, s_{a_2}\ states : [s \xrightarrow{a_1} s_{a_1}, s \xrightarrow{a_2} s_{a_2}] \Rightarrow \exists s'\ state : s_{a_2} \xrightarrow{a_1} s', s_{a_1} \xrightarrow{a_2} s' \quad \text{(A.3)}$$

Parallelism Relation for Petri Nets

The parallelism relation of Petri Nets is an application of the independence relation on transitions. For that, two transitions t_i and t_j are independent iff the firing of each one does not affect the firing of the other and both sequences of firing (t_i, t_j) and (t_j, t_i) lead to the same state. Formally, the transitions t_i and t_j are parallel, iff they are enabled at the same time and they are not in structural conflict : ${}^{\bullet}t_i \cap {}^{\bullet}t_j = \emptyset$.

A.3 Extension of Parallelism Relation to Time Petri Nets

The basic idea is to extend the previous parallelism relation of Petri Nets for TPN to be able to consider the time constraints. Therefore, we will define the notion of temporal parallelism (TP for short) that we will use to identify the independence of firing between two transitions. To define TP, we maintain the basic theoretical concept of parallelism relation : *the firing of one transition does not affect the firing of the others*. Therefore, two temporal transitions t_i and t_j are in TP, if they are parallel by ignoring time constraints (i.e., both enabled and without structural conflict) and if they have a common period of firing. For that, the intersection of transition intervals must be a non-empty sub-interval in which the firing of these transitions is possible.

Temporal Parallelism (TP)

For a state $e = (m, I)$, transitions t_i, $t_j \in enable(m)$ are in TP (denoted $TP(t_i, t_j)$), iff they are structurally parallel and their intervals are not disjoint. Formally :

$$TP(t_i, t_j) \Leftrightarrow (\bullet t_i \cap \bullet t_j = \emptyset) \wedge (I(t_i) \cap I(t_j) \neq \emptyset) \tag{A.4}$$

The TP relation can be translated into an equation on interval bounds of the concerned transitions as follows :

$$TP(t_i, t_j) \Leftrightarrow (\bullet t_i \cap \bullet t_j = \emptyset) \wedge (Max(\downarrow I(t_i), \downarrow I(t_j)) \leq Min(\uparrow I(t_i), \uparrow I(t_j))) \tag{A.5}$$

Exemple 1 *Let's consider the TPN in Figure 1(a). Initial transitions* t_0 *and* t_1 *are in TP because* $\bullet t_0 \cap \bullet t_1 = \emptyset$ *and they are both firable in the sub-interval* $I(t_0) \cap I(t_1) = [3, 4]$.

Temporal Sequentiality (TS)

Two transitions which are parallel without considering their temporal intervals, can be sequential if their interval are disjoint. For example, let's consider the TPN in Figure 1(a). If the interval of t_0 *remains* $[2, 4]$ *and the one of* t_1 *becomes* $[5, 5]$ *then transition* t_0 *must always be fired before* t_1 *(denoted*

$TS(t_0)$). Formally, for a state $e = (m, I)$, transition $t_i \in enable(m)$ is in TS iff $\forall t_j \in enable(m)$:

$$TS(t_i) \Leftrightarrow (\bullet t_i \cap \bullet t_j = \emptyset) \wedge (I(t_i) \cap I(t_j) = \emptyset) \wedge (\uparrow I(t_i) <\downarrow I(t_j))) \quad \text{(A.6)}$$

A.3.1 Temporal Parallelism by Order

The SCG of Figure 1(b) shows that the interleaving between t_0 and t_1 does not create a classic diamond interleaving structure. We call this structure triangular. It is directly related to the standard semantics of SCG, which stores transitions firing order into explicit timing constraints. These timing constraints could be different depending on the firing order, which leads to different class. Thus, we also consider parallelism relation taking into account the influence of the firing order as follows : (1) In the simple case where transitions are initial, to determinate the relation between transitions (TP or TS) we just use static intervals. (2) For more complex cases, to compute temporal parallelism considering this influence, we use transitions dynamic intervals (computed as in dynamic intervals the Section A.2.1) i.e, intervals taking into account the past evolution. We then can determinate if transitions are always in TP or only for specific firing order (denoted TPO). This also allows to compute the time elapsed since the moment at which each state has been reached until a transition from this state is fired (denoted θ), and all the possible firing orders of transitions. In the same way, we can define temporal sequentiality by order (TSO).

Temporal Parallelism by Order

For two transitions enabled by $e = (m, I)$, to compute TPO (resp. TSO) for a specific firing order, we must use dynamic intervals. We note TPO_{trace}, where trace represents this firing order starting from the initial state e_0 to this state e. Formally, for a set of parallel sequences $SP = \{Sp_1, Sp_2\}$, where $T_1 = \{t_i, t_{i+1}, ..., t_{i+n}\}$ and $T_2 = \{t_j, t_{j+1}, ..., t_{j+k}\}$ $(n, k \in \mathbb{N})$ and $t_i, t_j \in Init_{SP}$; let consider trace being a trace of transitions firing that leads to state $e = (m, I)$ from e_0. Thus, transitions t_{i+n} and t_{j+k} (both enabled by m) are in TPO_{trace},

or in TSO_{trace}*, iff :*

$$TPO_{trace}(t_{i+n}, t_{j+k}) \Leftrightarrow I(t_{i+n}) \cap I(t_{j+k}) \neq \emptyset \quad \text{(A.7)}$$

$$TSO_{trace}(t_{i+n}) \Leftrightarrow (I(t_{i+n}) \cap I(t_{j+k}) = \emptyset) \wedge \ (\uparrow I(t_{i+n}) <\downarrow I(t_{j+k})) \quad \text{(A.8)}$$

If all different firing orders between e_0 *and* e *lead to the same parallel relation, i.e.* t_i *and* t_j *are in TPO (resp. TSO) for all the traces, then the firing order will not be specified and the relation of parallelism will be TP instead of TPO (resp. TS instead of TSO).*

Exemple 2 *: Let's consider the TPN and its SCG in Figure 1, transitions* t_0 *and* t_1 *are in TP and their successors are* t_3 *and* t_2*, respectively. These successors are in TPO if* t_1 *is fired before* t_0*, denoted* $t_1 < t_0$*. In the reverse firing order, transition* t_2 *must always be fired in last position. To compute the parallelism relation for* t_2 *and* t_3 *by the firing order* $t_1 < t_0$*, we compute the dynamic intervals of* t_2 *and* t_3 *for this order. For that, we have* $\theta_1 \in [\downarrow I_s(t_1), min(\uparrow I_s(t_0), \uparrow I_s(t_1))] = [3, 4]$*. The new class reached by the firing of* t_1 *before* t_0 *is* c_2 *with* $I_2(t_0) = [max\big(0, \downarrow I_s(t_0) - max\theta_1\big) \ , \uparrow I_s(t_0) - min\theta_1] = [0, 1]$ *and* $I'(t_2) = I_s(t_2) = [5, 5]$*. After the firing of* t_0 *from* c_2 *(leading to class* c_5*), we will have* $\theta_0 \in [0, 1]$ *and the new dynamic intervals* $I_5(t_3) = I_s(t_3) = [1, 4]$ *and* $I_5(t_2) = [max\big(0, \downarrow I_2(t_2) - max\theta_0\big) \ , \uparrow I_5(t_2) - min\theta_0] = [4, 5]$*. As the dynamic intervals of* t_2 *and* t_3*, obtained by the firing order* $t1 < t0$*, intersect, then* t_2 *and* t_3 *are in TPO by this order :* $TPO_{t_1<t_0}(t_2, t_3)$*. The reverse order* $t_0 < t_1$ *leads to* $I_3(t_2) = [0, 4]$ *and* $I_3(t_3) = [5, 5]$*, which means temporal sequentiality by this order :* $TSO_{t_0<t_1}(t_2)$*.*

A.3.2 Transitions Newly in Temporal Parallelism

To satisfy the second condition that we have putted to adapt covering step approach into our context, we introduce the concept of newly in temporal parallelism (denoted NTP). Intuitively, even if two transitions are in TP, it is possible that one transition is fired before the other. This firing enable a new transition that can be potentially parallel with the still enabled transition, we say that these transitions are newly in temporal parallelism (in NTP). The NTP relation is not restricted to the direct successor of the fired transition, but it can go up to the nth successor that can be potentially in temporal paralle-

lism with the still enabled transition. For example, in the model of Figure 1(a), transitions t_1 and t_3 are newly in temporal parallelism.

Potential Interval

Dynamic intervals represent the realistic values according to the firing order of transitions that are fired. Using them to compute NTP relation is not possible because of the computing complexity (considering all the possible future traces in a dynamic way). To solve that, we use potential intervals (denoted I_P) which are inspired from the remark in [4] the possible occurrence time of a transition is fully determined by the occurrence times of its predecessors, to compute the potential firing instant of the successor transitions, considering the current state as the origin of computing.

Formally, for $n \in \mathbb{N}$ and SP_i a parallel sequence, the potential interval of transition t_{i+n} (the nth successor of t_i) is computed as follows :

$$I_p(t_{i+n}) = I(t_i) + \sum_{r=i+1}^{n} I_s(t_{i+r}) \tag{A.9}$$

Newly in Temporal Parallelism

The NTP and NTPO relations are computed with the same principles as the TP and TPO relations (equations A.4 and A.7), but using potential intervals instead of dynamic ones.

Exemple 3 *Let's consider the TPN in Figure 1(a). At the initial state there is no trace consideration. We have $I(t_0) = I_s(t_0) = [2, 4]$ and $I(t_1) = I_s(t_1) = [3, 4]$. The potential intervals of their successors are : $I_p(t_2) = I(t_1) + I_s(t_2) = [8, 9]$ and $I_p(t_3) = I(t_0) + I_s(t_3) = [3, 8]$. As $I(t_1) \cap I_p(t_3) \neq \emptyset$ transition t_3 is newly in temporal parallelism with transition t_1, $NTP(t_1, t_3)$.*

A.4 Reduced Graph Semantics

To limit the exhaustive enumeration of the state graph, we have defined a reduced graph (RG for short) which allows us to represent the parallelism of two transitions in a unique step. A step can be a relation of parallelism

between two transitions (TP, NTP, TPO) or a relation of sequentiallity for simple transition (TS, TSO).

A RG state is a tuple (m, δ) *where* m *is the marking and* $\delta = [\downarrow \delta, \uparrow \delta]$ *is the possible durations of each step* s *leading to this state. The boundaries* $\downarrow \delta$*, and* $\uparrow \delta$ *are the minimum and maximum time elapsed during these steps. The initial state is* $r_0 = (m_0, \delta_0)$*, where* m_0 *is the initial marking,* $\delta_0 = 0$ *and* $enable(m_0) = Init$*.*

The RG graph is defined by the labeled transition system such that $RG = (R, r_0, \twoheadrightarrow)$*, where :*

- $R = \mathbb{N} \times \mathbb{Q}_+$ *is the set of graph states,*
- r_0 *is the initial state,*
- $\overset{s}{\twoheadrightarrow} \in R \times S \times R$ *is the step relation, where* S *is the set of steps.*
  Let consider a state $r = (m, \delta)$ *with* $t_i, t_j \in enable(m)$*. Computation of the output steps from* r *is carried as follows :*

  1. *We identify all the possible traces from the initial state leading to* r*. For each trace :*
     (a) *We compute the dynamic intervals* $I(t_i)$ *and* $I(t_j)$*.*
     (b) *With equations A.7 and A.8, we verify if* t_i *an* t_j *are in TPO or TSO.*
     (c) *If they are in TPO, we compute the potential interval of their successors (equation A.9), then verifying if theses successors are in NTPO (see section A.3.2). And so on, until there is no more intersection between the potential intervals of the nth successor of* t_i *(resp.* t_j*) and the dynamic interval of* t_j *(resp.* t_i*).*
  2. *We then compare all the found parallelism relations for each trace : if the relations are all the same, then the order will be not specified : TPO (resp. TSO, NTPO) relation becomes TP (resp. TS, NTP).*
  3. *For each parallelism relation, we will have an output step* s*. We then can compute the new states of the RG as follows :*

- *If* $s \in \{TP, TPO\}$ *then :* $(m, \delta) \overset{s(t_i,t_j)}{\twoheadrightarrow} (m', \delta')$ *iff*

$$\begin{cases} m' = m + t_i \bullet + t_j \bullet - \bullet t_i - \bullet t_j \\ \delta' = [max(\downarrow I(t_i), \downarrow I(t_j)), max(\uparrow I(t_i), \uparrow I(t_j))] \end{cases}$$

— *If* $s \in \{NTP, NTPO\}$ *then* : $(m, \delta) \overset{s(t_i, t_{j+n})}{\twoheadrightarrow} (m', \delta')$ *iff*

$$\begin{cases} m' = m + t_i \bullet + \sum_{k=0}^{n} t_{j+k} \bullet - \bullet t_i - \sum_{k=0}^{n} \bullet t_{j+k} \\ \delta' = [max(\downarrow I(t_i), \downarrow I_p(t_{j+n})), max(\uparrow I(t_i), \uparrow I_p(t_j + n)] \end{cases}$$

As a NTP step represents the firing of several transitions, δ *is computed using potential and dynamic interval to take into account the considered firing sequence of this step.*

— *If* $s \in \{TS, TSO\}$ *then* : $(m, \delta) \overset{s(t_i)}{\twoheadrightarrow} (m', \delta')$ *iff*

$$\begin{cases} m' = m - \bullet t_i + t_i \bullet \\ \delta' = [\downarrow I(t_i), \uparrow I(t_i)] \end{cases}$$

States are merged if they have the same output parallelism relation and the same marking. In this case, the resulting time δ *is computed taking the maximum of the durations of all the merged states.*

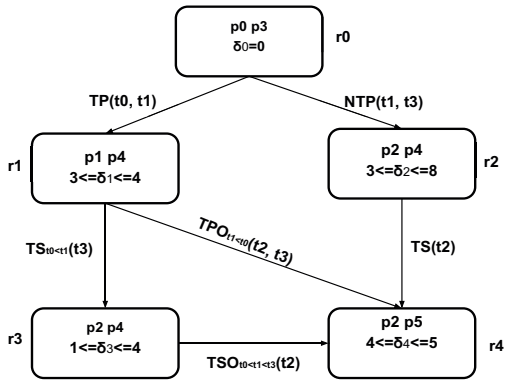


FIGURE A.4 – Reduced graph of the TPN Given in Figure 1(a)

Exemple 4 *To illustrate the construction of RG, let consider the TPN in Figure 1(a) and its RG in Figure A.4 : from the initial state* $r_0 = (m_0, \delta_0)$*, we have* $enable(m_0) = \{t_0, t_1\} = Init$ *and* $\delta_0 = 0$*.*

— ***State*** r_0 ***:*** *Transitions* t_0 *and* t_1 *are in TP. Thus, the step* $TP(t_0, t_1)$ *leads to a new state* $r1 = (m_1, \delta_1)$ *with* $m_1 : \{p_1, p_4\}$ *and* $\delta_1 = max(\downarrow$

$I(t_0), \downarrow I(t_1)), max(\uparrow I(t_0), \uparrow I(t_1))] = [3, 4]$. *We compute the potential intervals of the successors of* t_0 *and* t_1 *:* $I_p(t_2) = [8, 9]$ *and* $I_p(t_3) = [3, 8]$. *Hence, transitions* t_1 *and* t_3 *are in NTP, representing the firing of* t_0 *(implicitly) first then a temporal parallelism between* t_1 *and* t_3. *This step leads to a state* $r_2 = (m_2, \delta_2$ *with* $m_2 : \{p_2, p_4\}$ *and* $\delta_2 = [max(\downarrow I(t_1), \downarrow I_p(t_3)), max(\uparrow I(t_1), \uparrow I_p(t_3))] = [3, 8]$.

— ***State*** r_1 ***:*** *Transitions* t_2, t_3 *are enabled by* m_1. *Their dynamic intervals obtained by the order* $t_0 < t_1$ *are* $[5, 5]$ *and* $[0, 4]$, *respectively. As they are disjoint, then we have* $TSO_{t0<t_1}(t_3)$. *In the inverse order* ($t_1 < t_0$), *transitions* t_2, t_3 *dynamic intervals are* $[4, 5]$ *and* $[1, 4]$, *respectively. We then have* $TPO_{t1<t_0}(t_2, t_3)$. *As the parallelism relations from state* r_1 *are different for both traces, then there is two different output steps from* r_1. *The steps* $TSO_{t0<t_1}(t_3)$ *that leads to* $r_3 = (\{p_2, p_4\}, [1, 4])$ *and* $TPO_{t1<t_0}(t_2, t_3)$ *that leads to* $r_4 = (\{p_2, p_5\}, [1, 5])$ *and so on...*

A.5 Proof

Our method preserves the behavior of the SCG, while optimizing the storage memory. The RG preserves all complete firing sequences of the SCG without explicit enumeration. However, it does not preserve all possible markings (intermediary markings of an interleaving are not stored). Finally, the RG gives the minimum and maximum time elapsed during each step of the graph. We will prove these statements by using simulation in this section.

A.5.1 Marking Inclusion

State Class Graph

The SCG of a TPN is a labeled transition system (LTS) $\Sigma =< C, c_0, \rightarrow >$ *where* C *is the set of reachable state classes ;* $c_0 = (m_0, D_0)$ *is the initial state ; and* $\rightarrow \subseteq C \times (T \cup R_{\geq 0}) \times C$ *is the transition relation between state classes.*

For this section and the next one, let $RG = (R, r_0, \twoheadrightarrow)$ *and* $SCG = (C, c_0, \rightarrow)$ *be the RG and the SCG of the TPN* $(P, T, Pre, Post, m_0, I_S)$ *which is composed of two parallel sequences, respectively.*

Let $\mathcal{R}$ *be a simulation relation between a state of RG and a class of SCG. It is defined by* $\forall r = (m_r, \delta) \in RG$ *and* $\forall c = (m_c, D) \in SCG$, r *simulates* c

(denoted $(r,c) \in \mathcal{R}$*) iff* $m_r = m_c$.

All markings of RG are included in SCG.

$\forall r = (m_r, \delta), \in RG$ *and* $c = (c_m, D)$

[label=()]$\forall\ s \in \{TP, TPO\},\ if (r,c) \in \mathcal{R}$ *and* $r \overset{s(t_i,t_j)}{\rightarrow} r_1,$ *then*
$(\exists$ *path* $p = \ c \overset{t_j}{\rightarrow} c_1 \overset{t_i}{\rightarrow} c_2,$ *or* $p = \ c \overset{t_i}{\rightarrow} c_1 \overset{t_j}{\rightarrow} c_2)$ *and* $((r_1, c_2) \in \mathcal{R})$.
$\forall\ s \in \{TS, TSO\},\ if (r,c) \in \mathcal{R}$ *and* $r \overset{s(t_i,t_j)}{\rightarrow} r_1,$ *then*
$\exists\ c_1,\ c \overset{t_i}{\rightarrow} c_1$ *and* $(r_1, c_1) \in \mathcal{R}$. $\forall s \in \{NTP\},\ if (r,c) \in \mathcal{R}$ *and* $\forall\ r \overset{s(t_i,t_j)}{\rightarrow} r_1,$ *then*
$(\exists$ *path* $p = \ c \overset{t_{j-n}}{\rightarrow} c_1 ... \overset{t_i}{\rightarrow} c_{n-1} \overset{t_j}{\rightarrow} c_n,$ *or* $p = \ c \overset{t_{j-n}}{\rightarrow} c_1 ... \overset{t_j}{\rightarrow} c_{n-1} \overset{t_i}{\rightarrow} c_n)$ *and* $((r_1, c_n) \in \mathcal{R})$

The initial markings of the SCG and RG are the same by definition $((r_0, c_0) \in \mathcal{R})$ *(i) The proof is evident : if* t_i *and* t_j *are in TP (or TPO), by definition they are both enabled, firable in a common sub-interval, and not in conflict. Such traces (interleavings) always exist. The markings are computed in the same way. For the SCG, it is obtained by firing transitions* t_i *and* t_j *(interleavings) and for the RG, it is the firing of the same transitions in parallel. (ii) As the step is a single transition firing, the evolution of the marking is the same. (iii)* $s = NTP(t_i, t_j)$ *represents the firing* t_{j-n} *predecessors of* t_j $(n > 1)$ *before the interleaving of* t_i, t_j*. In this case, the markings of both graphs are obtained with the sum of all the pre and post functions of the concerned transitions.*

A.5.2 Complete Traces Preservation

A sequence of transitions $\sigma \in T^*$ *is a complete trace of SCG iff : from the initial class* c_0 *of SCG,* $\exists$ *class* c_1 *such as* $c_0 \overset{\sigma}{\rightarrow} c_1$ *and* $succ_c(c_1) = \emptyset$ *(with* $succ_c(c) = \{t \in T \mid \exists c', c \overset{t}{\rightarrow} c'\}$*).*

A sequence of steps $\psi \in S^*$ *is a complete trace of RG iff : for the initial state* r_0 *of RG,* $\exists$ *state* r_1 *such as* $r_0 \overset{\psi}{\rightarrow} r_1$ *and* $succ_r(r_1) = \emptyset$ *(with* $succ_r(r) = \{s \in S \mid \exists r',\ r \overset{s}{\rightarrow} r'\}$*).*

Let $\cong$ *be a relation over complete traces of SCG and RG.* $\cong$ *is relation of simulation* $\forall c = (m_c, D),\ \forall r = (m_r, \delta),\ \forall t_i \in enable(m_c)$ *iff :*

$if (c,r) \in \mathcal{R}$ *and* $c \overset{t_i}{\rightarrow} c_1,$ *then* $\exists s, r \overset{s(t_i)}{\rightarrow} r_1$ *and* $(c_1, r_1) \in \mathcal{R},\ \forall s \in \{TS,\ TSO\}$ *Or*
$\exists c_2, c_1 \overset{t_j}{\rightarrow} c_2$ *and* $\exists s,\ r \overset{s(t_i,t_j)}{\rightarrow} r_1$ *and* $(c_2, r_1) \in \mathcal{R},\ \forall s \in \{TP,\ TPO\}$

Or
$\exists$ *path*($p = c_1 \overset{t_{i+1}}{\rightarrow} c_2 ... \overset{t_{i+n}}{\rightarrow} c_{n-1} \overset{t_j}{\rightarrow} c_n$ *or* $p = c_1 \overset{t_{i+1}}{\rightarrow} c_2 ... \overset{t_j}{\rightarrow} c_{n-1} \overset{t_{i+n}}{\rightarrow} c_n$) *and* $\exists s, r \overset{s(t_j,t_{i+n})}{\rightarrow} r_1$ *and* $(c_n, r_1) \in \mathcal{R}$, $\forall s \in \{NTP\}$.

All complete traces of SCG are preserved in RG.

for all σ *complete trace of SCG :*

$\forall \sigma, c_0 \overset{\sigma}{\rightarrow} c_f, \exists \psi,\ r_0 \overset{\psi}{\rightarrow} r_f$ *and* $(\sigma, \psi\ ) in \cong$.

A complete trace of an SCG is a sequence of transitions firing starting from the initial class leading to last class. It can be composed of : (1) A single transition firing which is represented in our graph by the steps TS and TSO. In both graphs the same transition will be fired in the same order. Necessarily, it will lead to an equivalent class/state ; (2) An interleaving between two parallel transitions which is represented by the steps $s \in \{TP,\ TPO\}$*, both lead to an equivalent class/state ; (3) A sequence of transitions firing followed by an interleaving of two transitions which is represented by the step* $s \in \{NTP\}$*. As a single transition firing leads to an equivalent class/state, this is true also for a sequence of transitions firing. Also, the interleaving of two transitions leads to equivalent class/state. Thus, the combination leads to an equivalent class/state.*

A.6 Experimental Results

The approach proposed above was implemented and a set of experiments was conducted in order to evaluate the efficiency of our RG approach. For this purpose, we compute the Reduced Graph of several sets of two parallel sequences. The experiments performed on Macbook Pro with 2.7GHz Intel Core i7 processor and 16GB of RAM. The results are depicted in Table A.1 where the first column represents the lengths of the two parallel sequences in terms of the number of successive transitions. The second column represents the number of the SCG[1] *classes, which directly gives an idea of the used memory to store this graph, and the time necessary for its computation. The third column represents the same parameters for RG (the number of states and the computation times). The last column reports the rate of reduction in the number of states using RG, compared to the SCG classes.*

1. The SCG has been built with the tool Tina, http://projects.laas.fr/tina//

Length	SCG/CPU	RG/CPU	Memory saving
All transitions in $[0, +\infty]$			
4,4	25/0s	17/0s	32.0%
15,15	256/1s	226/5m	11.7%
20,20	441/1s	401/3d	9.0%
All transitions in $[1, 3]$			
10,10	544/0s	77/2s	85.8%
15,15	1269/0s	166/3m	86.9%
20,20	2292/1s	287/7d	87.4%
All transitions in $[1, 1]$			
20,20	61/0s	21/0s	65.5%
50,50	151/0s	51/	66.2%
random intervals with $[1, 1]$ or $[1, 3]$			
10,10	101/0s	21/1s	79.2 %
20,20	379/0s	69/3s	81.7 %

TABLE A.1 – Comparison of SCG and RG

These parameters of performance have been studied on several types of models : the first line represents the worst case of parallelism as all the transitions could be fired at anytime ($[0, +\infty[$*). For this type of model, our method is not really more efficient than the SCG. Indeed, our RG has almost the same number of states than the SCG, but with a higher computation cost due the huge number of potential parallelism on each state. For the second kind of model (*$[1, 1]$*), our methods is much more efficient where our RG is in the form of a sequential line of states while in the SCG, each step is represented with an interleaving. In the next two type of models (*$[1, 3]$ *and random intervals with* $[1, 1]$ *or in the form of* $[a, b]$, $a, b \in \mathbb{N}$*), there are complex parallelism situations. The SCG deals with this by enumerating all different time domains even if they have the same marking and that increases the number of classes. Our method handles this problem from a memory space point of view (states number), but at the cost of higher computational load.*

According to the results reported in Table A.1 we see that our RG is much more significant in states number and representation of parallelism for the sets of parallel sequences compared to the SCG. These results are logical as the SCG stores more information. Nevertheless, our approach is less appropriate when the goal is reachability analysis (a solution will be proposed in a future work). As in our context we are interested only in possible parallelism and firing traces,

our method is compact to provide this information. To handle computational cost of RG a parallel implementation will be developed.

A.7 Conclusion

In this paper, we propose an approach that allows the application of partial order on a subset of TPN. The targeted TPN is composed of two parallel sequences of transitions. The basic idea is to represent into one atomic step the relation of parallelism between two transitions. This approach computes a reduced graph (RG) comparing to State Class Graph (SCG) which enumeration complexity leads to state space explosion. This later is moved to computing complexity in our approach.

The continuation of this work is the extension of our approach to more generalized cases : extension to n parallel sequences, to closed-loop sequences, where the last transition is the predecessor of the initial one (This will be possible just by adding some sufficient conditions to fix a stop point of RG computing), and to sequences with conflicts/branchings structures. Also, we want to integrate our approach in the SCG method (modifying thus the way to deal with parallel sequences) in order to analyze complete TPN models.

The main perspective of our work is to adapt our RG method to generate state space of Synchronous Interpreted TPN [62]. Indeed, in the context of AIMD design and prototyping, our methodology automatically generates VHDL code from this class of TPN, to implement it in synchronous way into FPGA. So, if we want to use the powerful analysis capacities for properties verification of the model, we firstly have to guarantee that this non-functional property does not prevent the relevance of the analysis results. As a consequence, we want to directly construct an adapted state graph for synchronous interpreted TPN.