THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DES SCIENCES ET DE LA TECHNOLOGIE D 'ORAN, ET DE l'ECOLE ENSTA BRETAGNE

Délivré par : Laboratoire SIR : Systèmes Informatiques et Réseaux

Définir le nom de l'établissement avec l'option 'Ets' du paquet tlsflyleaf.sty

---

Présentée et soutenue le *09.07.2015* **par :**

Nadia Menad
Université des Sciences et de la Technologie d'Oran (MB), Algérie

Rapport de thèse

Modélisation des Systèmes embarqués Temps-Réel : Vers une ingénierie dirigée par les méthodes formelles

---

JURY

Philippe Dhaussy	Professeur	Président
Mekki Rachida	Directeur de recherche	Membre
Nom	Directeur de recherche	Membre
Nom	Directeur de recherche	Membre
	... (Préciser la qualité de chacun des menbres)	

École doctorale et spécialité : ENSTA Bretagne , LabSticc

Définir l'école doctorale avec l'option 'ED' du paquet tlsflyleaf.sty

Unité de Recherche :

Nom de l'Unité de recherche

Directeur de Thèse : Philippe Dhaussy

Nom(s) du/des directeur(s) de thèse

Rapporteur :

Noms des rapporteurs (s'ils ne font pas partie des membres du jury)

Rapport de thèse

Modélisation des Systèmes embarqués Temps-Réel : Vers une ingénierie dirigée par les méthodes formelles

Nadia Menad
Université des Sciences et de la Technologie d'Oran (MB), Algérie

March 17, 2016

To Nadia Menad (Me) and all our other colleagues from the ENSTA Bretagne research community

Résumé

Les défis que pose le développement des systèmes distribués et des systèmes électroniques multi-horloges modernes s'accentuent, du fait qu'ils sont caractérisés par une compléxité croissante. En effet, ces systèmes passent à l'échelle et rendent plus de services qu'auparavant, mais ils sont en même temps soumis à de plus fortes exigences en matière de leur bon fonctionnement logique ou temporel. Dans cet esprit, les méthodes, les langages ainsi que les outils de développement mis en oeuvre pour spécifier et développer ces systèmes doivent s'élever à un niveau de qualité, tout en permettant de répondre à la demande croissante de services et ainsi de satisfaire les pertinentes exigences. Cependant, la pratique montre qu'ils ne sont souvent pas utilisés de manière appropriée/adaptée.

Dans une activité classique d'Ingénierie Dirigée par les Modèles (IDM), les systèmes sont modélisés à l'aide d'une notation semi-formelle et sont par la suite validés puis implantés. L'étape de validation, basée sur ces modèles, est particulièrement cruciale pour les Systèmes Embarqués Temps-Réel (SETR) [1], afin de s'assurer de leur bon fonctionnement. Cependant, une démarche IDM reste insuffisante dans le sens ou elle n'indique pas comment utiliser les modèles pour appliquer l'analyse. Face à cette situation, l'intégration de méthodes formelles dans les cycles de développement de ces systèmes est devenue primordiale. Ces méthodes sont depuis longtemps reconnues afin d'aider au développement de systèmes fiables, en raison de leurs fondements mathématiques, réputés rigoureux sur l'exhaustivité de la vérification formelle qu'ils permettent d'activer.

La modélisation des systèmes distribués et des systèmes électroniques multi-horloges impliquent l'association implicite d'un ensemble d'horloges à un modèle. Pour la modélisation de tels sytèmes, un modèle de temps logique a été proposé à l'OMG pour enrichir le formalisme UML MARTE et permettre la description et l'analyse de contraintes temporelles. Ce modèle de temps est associé au langage de spécification de contraintes d'horloge CCSL. Une fois le logiciel modélisé, la difficulté est de pouvoir exprimer les propriétés pertinentes et les vérifier formellement. Dans ce contexte, nous rendons compte, dans cette thèse, d'un travail portant sur une technique de vérification de propriétés par model-checking exploitant le langage CDL (Context Description Language) et l'outil OBP (Observer-based Prover). La technique s'appuie sur une traduction des modèles MARTE et des contraintes CCSL en code Fiacre. CDL permet d'exprimer des prédicats et des observateurs. Ceux-ci sont vérifiés lors de l'exploration exhaustif du modèle complet par OBP.

[1] On qualifie de temps réel tout système qui doit réagir dans des intervalles de temps bornés à des sollicitations émises par un environnement extérieur.

Remerciements

Je voudrais d'abord témoigner mon attachement et ma profonde gratitude à mes parents. Je voudrais également exprimer ma gratitude à mes frères, pour leur immense soutien, aussi bien dans les périodes sombres que plus lumineuses, et toute ma grande famille.
Je tiens à remercier chaleureusement mon encadrant Mme Mekki Rachida pour son encadrement et son aide pendant ce parcours. Un éternel merci à mon co-encadrant Pr Philippe Dhausssy, mon infatigable directeur de thèse qui a trouvé le temps, parmi les connaissances et les méthodes de travail qu'il m'a transmises. Merci pour ta confiance, pour ta générosité et pour avoir accepté d'encadrer mes travaux de thèse. Ton écoute, ton soutient et ta disponibilité ainsi que ton amitié de ces années de thèse. Une éxpérience inoubliable et si sympathique. Par ailleurs, ta rigueur et tes inestimables conseils qui m'ont toujours éclairé et m'ont aidé à garder les objectifs de mes travaux. Pour tout cela, je suis infiniment redevable. Je voudrais saluer Annick et Michele, pour leur disponibilité et efficace assistance administrative. Je voudrais également saluer tous mes collègues et amies, particulièrement Hannene, Meriem et Asmaa , pour leur gentillesse et leur aide, ainsi que pour les agréables moments de déjeuner, et ces discussions vériées et marrantes au moments des pauses, et avec qui j'ai partagé de bons moments de travail.

Merci à Julien Deantonie, Loïc Lagadec, Joël Champeau, Jean-Charles Roger, Jean-Philippe Schneider, Bruno Aizier, Lucas Le Roux, Ciprian Teodorov , Stephen Creff, Papa Issa Diallo, Zoé Drey, Sébastian Tleye pour tous leurs aides et encouragements. Il m'est impossible d'oublier Nadia, compagnon de tous les combats, pour son aide précieuse. Ma gratitude va également aux rapporteurs, de m'avoir fait l'honneur d'être mes examinateurs et de m'avoir fait l'honneur de rapporter ma thèse. Je tiens également à adresser un grand merci à eux pour leurs remarques et leurs conseils qui m'ont permis d'améliorer de façon significative ce document.

Je n'oublirai pas mes amis stagiaires à l'ENSTA, avec qui j ai partagé de très beaux souvenir à Brest. Il me sera très difficile de remercier tout le monde car c'est grâce à l'aide de nombreuses personnes que j'ai pu mener cette thèse à son terme. À vous tous, merci pour votre aide et vos précieux conseils. J'en oublie certainement encore et je m'en excuse. Encore un grand merci à tous pour m'avoir conduit à ce jour mémorable. A toutes les personnes qui ont rendu cette thèse possible par leur aide et leurs contributions.

Sommaire

Liste des figures

Part I

Introduction générale

Chapitre 1

Introduction

"If we knew what it was we were doing, it would not be called research, would it?"

Albert Einstein

1.1 Contexte et problématique de recherche

1.1.1 Besoins et contexte pour la modélisation temporisée

Dans le domaine de la modélisation d'architectures logicielles de systèmes distribués, de systèmes de contrôle-commande ou de systèmes électroniques multi-horloges, la spécification des parties fonctionnelles des systèmes est souvent associée à une spécification de contraintes temporelles. En effet, ces systèmes sont souvent critiques et les exigences à respecter lors de la modélisation concernent non seulement le déterminisme sur le plan fonctionnel mais aussi sur le plan temporel. Dans le processus de développement des systèmes, les concepteurs recherchent des méthodes et des langages leurs permettant de décrire leurs architectures, tout au long du cycle et à différents niveaux d'abstraction.

1.1.2 Les formalismes

A chaque niveau, la modélisation de tels systèmes doit permettre de manipuler, au sein de modèles, des exigences temporelles et d'évaluer l'exactitude et l'efficacité des applications en fonction de critères temporels et mesurables. Dans ce but, un concept de modélisation abstraite des horloges logiques a été introduit avec le langage CCSL (*Clock Constraint Specification Language*) [And09] au sein de MARTE [MAS08], adopté par l'OMG [OMG10]. Cette introduction vise non seulement à rendre complémentaires des formalismes existants mais aussi à doter les modèles de capacité d'analyse en vue d'évaluer leur correction au regard d'exigences exprimées par le concepteur.

1.1.3 Les besoins de simulation et de vérification

Il est donc primordial d'adopter des approches d'analyse temporelle par l'intégration des processus de vérification et de validation robustes basés sur des notions formelles afin de répondre aux exigences de qualité actuelles de ces systèmes. L'ingénierie des architectures logicielles basée sur le prototypage rapide conduit à des itérations nombreuses dont le coût en simulation ne peut être négligé. Ceci se révèle particulièrement crucial pour les applications multimédia (encodeurs, décodeurs vidéo, etc), qui cumulent des spécificités de flux de données, de contrôle et de traitements complexes, découpés en automates à grains fins. Les outils de modélisation HW/SW de telles applications souffrent donc d'une forme d'incompatibilité entre les ambitions en terme de prototypage et les besoins en tests incessants, consommateurs de temps.

1.1.4 MARTE, CCSL

De nombreux travaux ont proposé une approche d'ingénierie, basée sur les modèles, et l'utilisation de notations semi-formelles, comme UML, enrichies par des notations formelles. Nous avons évoqué le profil UML-MARTE qui intègre des notations pour exprimer des

contraintes temporelles. Dans ce domaine également, un travail récent étudie une approche de formalisation pour capturer la sémantique de MOCs (Models of Computation). Une extension du profil MARTE nommé COMETA [KCS09, DCL11, DCL13] a pour objectif de proposer au concepteur de manipuler explicitement une définition des communications et des traitements concurrents. Notre objectif est de définir une méthodologie afin de pouvoir appliquer la vérification sur des modèles décrits avec le formalisme MARTE enrichi de contraintes CCSL.

Le profil UML pour la modélisation et l'analyse des systèmes embarqués temps réel [Gro09b] MARTE, a introduit un modèle de temps [AMdS07] qui étend le temps simple informel d'UML 2.x. Ce modèle de temps est suffisamment général pour supporter différents formes de temps (discret ou dense, chronométrique ou logique); nommées horloges, permettant de mettre en vigueur l'observation des occurences d'évènements ainsi que le comportement des éléments UML annotés. MARTE promet un cadre de modélisation générale pour concevoir et analyser les systèmes. Plusieurs travaux ont été publiés concernant les capacités de modélisation offertes par MARTE, beaucoup moins vis à vis des techniques de vérifications supportées. Le modèle de temps de MARTE est accompagné d'un langage dit (*Clock Cosntraints Specification Language*) (CCSL) [AM08] défini dans l'annexe des spécifications de MARTE. Initialement conçu comme un langage simple pour l'expression de contraintes entre les horloges du modèle MARTE, le langage CCSL a évolué et a été développé indépendement d'UML. Ce langage est maintenant doté d'une sémantique formelle [And09].

1.1.5 Techniques de Model-checking

Les modèles ainsi construits doivent pouvoir être non seulement simulés mais également exploités lors d'analyses formelles pour vérifier les exigences temporelles spécifiées par le concepteur. Dans notre travail, nous nous focalisons sur les techniques de vérification de type $model - checking$. Celles-ci [QS82, CES86] ont été fortement popularisées grâce à leur capacité d'exécuter automatiquement des preuves de propriétés sur des modèles logiciels. De nombreux outils (model-checkers) ont été développés dans ce but [Hol97a, LPY97a, BRV04, FGK$^+$96, CCGR00a]. Prenant en compte ces techniques, nous cherchons à étudier leurs apports et leurs limites lors du développement des modèles et les conditions dans lesquelles le concepteur peut raisonnablement les exploiter.

Dans le but de faciliter l'application des techniques de vérification par model-checking sur les modèles de conceptions, une technique a été proposée dans [RDb11, RDA10], pour la génération des modèles comportementaux des acteurs de l'environement. Dans cette approche une formalisation des exigences des cahiers de charge a été introduite. En effet, une spécification des exigences à un niveau d'abstraction plus élévé a été proposée, ainsi que la génération automatique des propriétés décrites dans un langage spécifique nommé

CDL (*Context Specific Language*).

1.1.6 Plateforme de vérification OBP

Les travaux de [Rog06], [DAB+08], [eFB07], [DR11b] ont été conçus dans le but de faciliter l'utilisation des outils de Model-checking. L'objectif principale était d'étudier les techniques ainsi que les conditions qui permettent à des ingénieurs cherchant à vérifier et valider des exigences dans un contexte spécifique, sur des modèles logiciels. Ces travaux ont ainsi contribué à permettre d'exprimer facilement des exigences, et cela sous une forme compréhensible vis à vis des non experts des logiques temporelles. Ces travaux voulaient être un apport pour pouvoir mener des vérifications par model-checking sur des modèles de grande taille.

D'autre part, l'idée qui a suivi pour ces approches concerne les explorations des modèles, afin de contourner l'explosion combinatoire. L'objectif est de chercher à réduire le comportement des modèles lors de leur exploration, et ceci en prenant en consideration leur environnement (avec lequel le système intéragit), qu'ils nomment *Contexte*. Cependant, l'objectif visé était de guider l'exploration à ne pas focaliser ses efforts sur l'exploration d'un automate global mais sur une restriction supposée pertinente du comportement du modèle à vérifier. La description des contextes et des propriétés est associée dans un même langage nommé CDL (Contexte Description Language)[DPC+09]. Ces travaux prennent un point de vue complémentaire, par rapport aux contributions citées précédement, en définissant un langage qui est à base de patrons de définitions de propriétés, qui permet de faciliter l'expression des exigences.

1.2 Contribution

Nous présentons dans cette thèse un travail exploratoire qui se focalise sur l'association de la spécification d'un sous ensemble de MARTE enrichi par des spécifications de contraintes CCSL et d'un outillage de vérification de propriétés formelles, nommé OBP (Observer-Based Prover)[1] [DPC+09]. Les vérifications opérées par OBP sont basées sur l'exploration de programmes Fiacre [FGP+08] et l'exploitation d'observateurs. Notre objectif est de tirer parti des avantages de la vérification basée sur les contextes tout en partant de modèles temporisés en MARTE, considérés plus facile à définir que des modèles formels (e.g Fiacre). Par conséquent, l'approche que nous avons retenue consiste à transformer les modèles MARTE définis par les utilisateurs en modèles formels exploitables par les outils existants tels qu'OBP.

La contribution de la thèse est de : (1) Identifier du périmètre d'UML-Marte étendu avec les opérateur CCSL, (2) Transformer des contraintes CCSL en code Fiacre, (3) Pro-

[1] http://www.obpcdl.org

poser une méthodologie d'expression des propriétés CDL pour prouver la correction des transformations CCSL vers Fiacre, (4) Illustrer sur un modèle exemple.

Dans cette approche, nous cherchons à vérifier des contraintes CCSL ainsi que les propriétés fonctionnelles de l'architecture logicielle modélisée. Pour cela, nous avons, d'une part, générer des programmes Fiacre, à partir de modèles UML-MARTE. D'autre part, nous exploitons les spécifications CCSL pour enrichir ces programmes par l'ajout de processus implantant les contraintes CCSL en nous inspirant de l'approche décrite dans [YM11]. L'automatisation de la génération de code Fiacre à partir des parties fonctionnelles de l'application décrite en UML-MARTE, fait ainsi partie de l'objectif de notre approche. Nous illustrons notre contribution par un exemple et nous décrivons des résultats sur les preuves d'exigences menées.

1.3 Plan de la thèse

Ce rapport de thèse est organisé comme suit : Une première partie I consiste à une intoduction générale de la thèse. On présentera la problématique de recherche ainsi qu'une synthèse des contributions. Nous présentons un état de l'art dans la seconde partie du mémoire II. Cette dernière est subdivisée en trois chapitres. Le chapitre 2 présente des définitions et préliminaires, ainsi que les principaux formalismes de MARTE. Un survol sur les techniques de transformation des modèles et l'analyse formelle des exigences temporelles est présenté dans le chapitre 3. Un état de l'art du domaine incluant les travaux connexes concernant la modélisation temporisée est présenté dans le chapitre 4.

La partie III présente notre approche pour la modélisation temporisée : cette partie décrit notre contribution à la thèse dans le chapître 5. Nous y présentons les principales notations semi-formelles du domaine permettant de satisfaire les contraintes de modélisation évoquées précédemment. Les principes de transformation sont décrits dans le chapitre 6. Le chapitre 7 présente les principes de vérification des exigences sur les modèles temporisés. Les expérimentations sur des cas d'etude sont ainsi exposées dans ce chapitre. Nous concluons la thèse par une discussion générale dans la partie IV des résultats obtenus, tout en présentant les limites et les perspectives de recherche.

Part II

Etat de l'art

Chapitre 2

Définitions et préliminaires

"*Failure is the opportunity to begin again, more intelligently.*"

Henry Ford

2.1 Introduction

Dans le domaine des SETR et les systèmes réactifs, les architectures logicielles doivent être conçues pour assurer des fonctions très critiques soumises à de très fortes exigences en termes de performances temps réel et de faisabilité. En effet, avec la croissance de la capacité des calculateurs embarqués, la taille de ces systèmes accroît ainsi les risques d'erreurs. Parmi ces systèmes, les systèmes asynchrones, composés de sous-systèmes communiquant par échange de messages via des file d'attentes (par exemple des protocoles, des systèmes de communication. etc), apportant une complexité supérieure. De nos jours, les industries consacrent tous leurs efforts dans le processus de tests et de simulations à des fins de certification. Toutefois, si les tests s'indiquent souvent efficaces pour détecter les erreurs, ils ne permettent généralement pas de démontrer exhaustivement l'absence d'erreurs. Dans ce cas, il est alors primordial d'utiliser d'autres méthodes afin de garantir la fiabilité des logiciels.

Face à ce constat, de nombreuses techniques ont été explorées, parmi lesquelles celles de la famille des méthodes formelles qui ont contribué, depuis de nombreuse années, à l'apport de solutions précises et rigoureuses afin d'aider les concepteurs à produire des systèmes non défaillants. Dans ce domaine, les techniques de model-checking [QS08] [ECS86] ont été fortement popularisées grâce à leur capacité d'exécuter automatiquement des preuves de propriétés sur des modèles logiciels. Plusieurs outils ont été développés dans ce but [Hol97b] [LPY97b][BBV04] [JCFS96], [CCGR00b] tels que SPIN, UPPAAL, TINA, CADP, etc. L'idée générale de fonctionnement de ces outils consiste à tenter de modéliser de façon abstraite et compacte l'ensemble des comportements possibles du système à valider ainsi que son environnement, et de les parcourir exhaustivement afin de décider si l'ensemble des exécutions possibles est exempt d'erreurs. La rapidité du parcours dépend du degré de compactage de l'ensemble des comportements.

Plusieurs travaux ont été menés dans ce sens [MP95] [EJP97] [Val91] [RAR97] [Pel98][PGS96] [BH05]. Néanmoins, compte tenu des tailles gigantesques des ensembles considérés, les progrès réels des outils de vérification ne permettent pas encore de traiter la vérification des systèmes réels de taille industrielle. Malgré l'efficacité progressive de ces outils, leur utilisation reste toujours difficile en contexte industriel. Leur intégration dans un processus d'ingénierie industriel est encore trop insuffisant comparativement à l'énorme besoin de faisabilité dans les systèmes critiques. Cette contradiction trouve en partie ses raisons dans la compléxité réelle de mettre en oeuvre des notions théoriques dans un contexte industriel.

Une première difficulté liée à l'utilisation de ces techniques de vérification provient du problème bien identifié de l'explosion combinatoire du nombre de comportements des modèles, provoqué par la complexité interne du logiciel à vérifié. Cela est particulièrement vrai dans le cas des SETR, qui interagissent avec des environnements possèdant un grand nombre d'entités. Ainsi, parmi les barrières d'utilisation d'adoption de processus de model-

checking est l'existance du fossé sémantique entre les langages utilisés dans l'industrie et ceux pris en compte par les outils formels, ce qui nécéssite des méthodes de transformation robustes. Une autre difficulté est liée à l'expression formelle des propriétés nécessaire à leur vérification. Habituellement, cette expression se réalise à l'aide de formalismes tels que les logiques temporelles. Bien que ces logiques aient une grande expressivité, elles ne sont pas faciles à utiliser par des ingénieurs dans des contextes industriels.

L'approche développée dans nos travaux a pour vocation d'appréhender les difficultés mentionnées précédemment. Elle repose sur deux idées conjointes : D'une part, réduire l'écart entre les modèles semi-formelles exprimés en MARTE-CCSL pour décrire les comportements du système à valider et les langages formels utilisés commme entrée pour le processus de model-checking de manière à exploiter ce que ces modèles permettent d'exprimer. D'autre part, décrire des propriétés temporelles réutilisables et exploitables lors de la vérification automatique. Cet axe de recherche a donné lieu au développement de travaux dans deux directions complémentaires :

- Une première direction s'est intéréssée à une approche de transformation des modèles UML MARTE/CCSL en code formel. Ce langage est associé à l'outil OBP qui a été développé pour sa mise en oeuvre.
- Ainsi, ce travail a nécessité des réflexions dans une deuxième direction d'ordre méthodo-logique. On choisis d'exploiter l'expresivité du langage CDL, afin de coder des propriétés qui sert à vérifier des contraintes temporelles bien spécifiques. L'utilisation de ce langage est associée à une méthode qui permette son intégration dans un processus plus large de développement industriel de logiciels et plus particulièrement qui permette une aide à la mise en oeuvre des techniques de vérification d'exigences.

2.2 Vers un plus haut niveau d'abstraction

Un système temps réel est un système mutli-taches destiné particulièrement à l'execution d'application temps-réel. Ce type de système est capable d'executer plusieurs taches d'une manière concurrente, alternant entre les différentes taches. Dans ce type de systèmes, une ou plusieurs politiques d'ordonnoncement est fournie par un algorithme général permettant de déterminer l'ordre dans lequel les taches doivent s'executer en fonction de leurs caractéristiques temps-réel (Echéance, periode. etc).

Parmis les outils permettant de fournir une interface adequate aux specifications temporelles. On peut citer VxWorks par Wind River Systems [Riv95] et OSE [OSE04] pour

les systèmes commerciaux. RTLinux [1] et RTAI [2] pour les systèmes open source, ou encore ERIKA [PGB00], SHARK [PGB01] et MARTE [Gro09b], considérés comme des noyaux de recherche visant à fournir un niveau d'abstraction plus élevé que les systèmes classsiques.

Développer des SETR à l'aide des langages bas niveau permet assurèment concevoir une large classe de systèmes, mais cela présente de majeurs désavantages. D'une part, le faible niveau d'abstraction rend la correction du programme difficile à démontrer. Et d'une autre part, il est difficile de selectionner/distinguer dans un programme les élements liés à de forte exigences de conception de ceux liés à des soucis d'implémentation.

Dans ce qui suit, nous présentons dans ce qui suit un résumé des langages existant qui concernent la programmation des SETR. Parmis les caractéristiques souhaitables de tels langages sont :

- Le déterminisme au niveau fonctionnel et temporel du code généré,
- La possibilité de spécifier un ensemble d'opérations, des échanges de données entre opérations,
- La possiblité de spécifier des contraintes temporelles (e.g. les contraintes sur la périodicité et d'échéance),
- Une définition formelle du langage, dans le but d'assurer que la sémantique d'un programme donné est définie d'une manière non ambigue.
- Une génération de code automatique, pour couvrir le procesus de développement à partir de la spécification jusqu'à l'implémentation.

2.2.1 Langages de programmation

La conception d'un langage de programmation est le résultat d'un compromis qui détermine une balance entre les différentes qualités du langage telles que l'expressivité, la prédiction des performances, l'efficacité ou bien la simplicité de la sémantique. Ces propriétés diffèrent considérablement selon les choix effectués.

Les langages synchrones L'approche synchrone [] propose un haut-niveau d'abstraction, basé sur des fondements mathématiques à la fois simples et solides, permettant de gérer la vérification et la compilation d'un programme de manière formelle. Le concept de synchronisation simplifie le développement des systèmes temporels afin de le remplacer par du *temps logique*. Ce dernier est défini comme une suite d'instants, ou chaque instant représente l'execution d'une réaction du système. Ce principe a été implémenté dans

[1] www.rtlinuxtree.com
[2] www.rtai.org

plusieurs langages, tel que LUSTRE [CPHP87] ou SIGNAL [BGJ91a], ou même ESTEREL [BdS91].

Les langages synchrones impératifs ESTEREL [BdS91] est considéré comme le principal représentant des langages synchrones impératifs, qui s'interéssent essentiellement à la description du flot de contrôl d'un système. Il s'agit d'un ensemble de processus concurrents qui communiquent par le biais de *signaux* structurés en *modules*. Les processus executés par un module sont décrits comme une constitution d'instructions.

Les langages globalement asynchrones et localement synchrones (GALS) (multihorloges, Polyshynchrounous...) Dans un tel système, plusieurs systèmes localement synchrones sont assemblés à l'aide de mécanisme de communication asynchrone. La classe générale de ce système est étudiée par exemple dans *Esterel* multi-horloge et *Polychrony* [PLG03]. *CCSL est inspiré des langages synchrones mais se destine à cette classe considéré plus générale.*

Notons qu'il y'a une difference entre un langage synchrone et un langage utilisant des communications synchrones.

2.2.2 Langages de modélisation

L'émergence de langages de modélisation se fait de plus en plus dans le milieu industriel, dont les objectifs essentiels sont d'une part d'assister les formateurs et ingénieurs dans la conception de systèmes soumis à de forte contraintes dans des représenation plus abstraites; et d'autre part, fournir un formalisme de description des architectures afin de l'intégrer dans une démarche de génération automatique pour la vérification. L'approche MDA (Model Driven Architecture) peut être instanciée en utilisant différents langages. C'est pourquoi un certain nombre de langages permettant cette description sont apparus et offrent à ce jour un certain nombre de fonctionnalités. La plupart des approches guidées par des modèles se basent soit sur l'un des langages suivants :

- le langage de modélisation unifié (Unified Modeling Language, UML) [ACD],
- des langages de description d'architecture (Architecture Description Language, ADL), qui sont des langages propres à des domaines particuliers. Un ADL est un langage qui permet la modélisation d'une architecture conceptuelle d'un système logiciel et/ou matériel. Il fourni une syntaxe concrète et une structure générique (framework) conceptuelle pour caractériser les architectures.
- des langages spécifiques, conçus pour répondre à des besoins particuliers de modélisation et d'analyse (e.g. Behavior Interaction Priority (BIP) BBS06, Fractal [EBS06]).

Nous citons dans ce qui suit quelques langages de modélisation des systèmes répartis.

AADL Langage d'analyse et de description d'architectures (Architecture Analysis and Design Language, AADL) [SAE], a fait l'objet d'un intérêt croissant dans l'industrie de la modélisation des SETR (comme Honeywell, Rockwell Collins, l'Agence Spatiale Européenne, Astrium, Airbus). Il a été standardisé par la SAE (International Society of Automotive Engineers) en 2004, pour faciliter la conception et l'analyse de systèmes complexes, critiques, temps réel dans divers domaines comme l'avionique et l'automobile. Ce langage fournit une notation textuelle et graphique standardisée pour décrire des architectures matérielles et logicielles.

Le langage AADL, pour sa part, considéré comme langage très prometteur pour offrir une description rigoureuse et assez fiable des systèmes embarqués. De plus, les outils utilisant ce langage sont encore pour la plupart au stade du développement et n'implémentent pas tous les éléments spécifiés dans la norme AADL. En outre, AADL n'offre pas une sémantique précise et manque d'outils de vérification robuste.

SysML En abrégé - est un langage de modélisation spécifique au domaine de l'ingénierie système. Il permet la spécification, l'analyse, la conception, la vérification et la validation de nombreux systèmes. À l'origine, SysML a été développé dans le cadre d'un projet de spécification open source, et inclut une licence open source pour sa distribution et son utilisation. SysML se définit comme une extension d'un sous-ensemble d'UML via l'utilisation du mécanisme de profil défini par UML. On peut citer quelques apports de SysML par rapport à UML :

- SysML, spécialisé dans la modélisation de systèmes, offre aux ingénieurs systèmes plusieurs améliorations notables par rapport à UML, qui est plus centré sur le logiciel,
- La sémantique de SysML est plus riche et flexible,
- SysML impose moins de restrictions liées à la vision d'UML centrée sur le logiciel, et ajoute deux nouveaux types de diagrammes: Le diagramme des exigences peut être utilisé pour la gestion des exigences alors que le diagramme paramétrique peut être utilisé pour l'analyse des performances et l'analyse quantitative,
- SysML est capable de modéliser une large gamme de systèmes, incluant tant du matériel, que du logiciel, de l'information, des processus, du personnel, ou des équipe--ments (au sens large),
- C'est un langage plus réduit qu'UML ce qui facilite son apprentissage et son utilisation,
- SysML supprime beaucoup de concepts d'UML trop liés à sa vision centrée sur le logiciel. L'ensemble du langage SysML est plus petit, tant en nombre de types de diagrammes qu'en nombre de concepts totaux,

- SysML gère mieux les notations tabulaires. Il fournit des tableaux d'allocations flexibles qui supportent l'allocation des exigences, l'allocation fonctionnelle, et l'allocation structurelle, ce qui facilite l'automatisation de la vérification et de la validation.

Les concepts propres à SysML étendent les possibilités d'UML et sont architecturalement alignées avec le standard IEEE-Std-1471-2000 (IEEE Recommended Practice for Architectural Description of Software Intensive Systems).

UML UML est l'accomplissement de la fusion de précédents langages de modélisation objet : Booch, OMT, OOSE. Principalement issu des travaux de Grady Booch, James Rumbaugh et Ivar Jacobson. UML est à présent un standard défini par l'OMG, considéré comme un langage de modélisation à usage général, utilisant des notations graphiques permettant de créer des modèles abstraits. Si UML permet une grande expressivité, il ne transporte pas en lui-même une sémantique précise pour décrire les architectures [DGK02]. Ce langage a une sémantique pas précise, qui peut être raffinée par la définition de *Profil*. Nous définissons dans ce qui suit la notion de profil.

UML et l'Aspect Temporel Le profil UML d'ordonnoncement de performance et de temps, *UML Profile for Schedulability, Performance, and Time*, *SPT* [OMG05] a été proposé pour les concepteurs et developpeurs d'applications temps-réel. SPT introduit des notions facilitant la manipulation du temps et les ressources en UML. Ce profil permet d'annoter les élements de modèle par des informations quantitatives relatives au temps qui sont utilisées pour des analyses de performance, d'ordonnancabilite ou de vérification de contraintes temporelles. SPT ne considère qu'un temps métrique qui fait implicitement référence au temps physique, en introduisant des concepts d'instants et de durées, ainsi que ceux d'évènements et de stimuli liés au temps.

MARTE remplace le profil UML SPT existant, qui devait étre aligné sur UML 2 [Gro06]. Ce dernier avait pour but de combler les lacunes d'UML 1.4. Les constructions de SPT ont été condidérées trés abstraites et difficiles à appliquer, d'ou la proposition d'un nouveau profil qui a fait issue de MARTE. Son extention pour la modélisation des SETR répond ainsi au besoin de modéliser des contraintes temporelles, en introduisant un modèle de temps, concept absent dans les versions d'UML. Nous citons dans ce qui suit les opportunités qu'apporte le profil MARTE. Les retombées attendues de l'usage de ce profil sont de :

- Faciliter la construction de modèle sur lequels des prévisons quantitatives peuvent être faites, tout en tenant compte des caractéristiques du matériel et du logiciel;
- Fournir de la modélisation unifiée pour les parties logicielles et materielles des sytèmes;
- Permettre l'interopérabilité entre les outils de développement utilisés pour la spécifica-tion, la vérification, ainsi que la génération du code.

2.3 Le Formalisme UML MARTE

La conception des systèmes à un niveau élevé (system-level) a été largement recommandée comme une façon pour diminuer la complexité croissante de la conception des technologies embarqués ainsi que les facteurs aggravants leurs contraintes. Des langages de modélisation de haut niveau, des outils, ainsi que des intergiciels ont été proposés afin de concevoir, simuler et valider les systèmes embarqués, dans le but (with the claim of allowing) de permettre au ingénieurs de bénéficier d'une meilleurs compréhension ainsi d'une meilleurs productivité grâce au niveau d'abstraction élevé offert par ces modèles et outils.

Dans ce contexte, le langage de spécification UML (Uniform Modeling language) [Gro09a] a été utilisé comme un moyen général de modélisation (general-purpose modeling language). Ce langage fournit une interface graphique pour le modèle, et possède une sémantique large —pas précise —. Les points de variation sémantique ouverts par cette spécification peuvent étre exploités par ce qu'on appel des *Profils*, afin de raffiner et définir une sémantique adaptée à un domaine précis.

2.3.1 La Notion de Profil

UML a pour but de modéliser un large éventail de differents systèmes relevant de divers domaines d'application. Cet outil va bien au-delà de la modélisation d'application logicielles selon *Bran Selic* dans [Sel04] sur les bases sémantiques d'UML 2, " *In essence, standard UML is the foundation for a family of related modeling languages"*. Cette diversité sous-jacente a une conséquence sur la sémantique de langage afin d'être adaptable à différents domaines. UML introduit plusieurs points de variation sémantique, et par conséquent de diverse interprétations peuvent être proposées pour le même élément de modèle. En effet, le mécanisme de profil permet de lever les ambiguités dues à ces points de variation sémantiques. Par conséquent, la cohérence entre les différentes vues d'un modèle UML n'est pas garantie, ce qui est néttement génant, à cause de l'absence de sémantique formellement exprimées.

Le Unified Modeling Language (UML) [Gro06] considéré comme langage de modélisation à but général, constitué d'un ensemble de notations graphiques permettant de créer des modèles abstraits d'un système. Sa sémantique est imprécise et peut être raffinée par l'introduction de profils spécifiques à un domaine. C'est le cas du profil UML pour la modélisation et l'analyse de systèmes temps réel embarqués (MARTE) [Gro09b]. Ce profil est enrichie en particulier par langage de spécification de contraintes d'horloges (CCSL) [AM08] inspiré des langages synchrones, qui permet de spécifier les propriétés temporelles d'un système. Ce langage se destine à la classe plus générale des *Globally Asynchronous Locally Synchronous Systems* (GALS) citée auparavant (cf 2.2.1) .

Un profil passe par la définition d'extentions. Son intéret est de réutiliser et d'étendre des outils existants. En d'autre termes, repérer les concepts de base du modèle et de

dire *****qu'on a besoin de quelque chose de plus / quelque chose de moins****** pour modéliser tel type de système. Cette extension concerne l'ajout de *Stéréotypes*, permettant aux concepteurs d'étendre le vocabulaire d'UML en incluant de nouveaux éléments ayant des propriétés particulieres adaptées à un domaine spécifique. Ces mécanismes d'extensibilité sont insérés d'une manière strictement addictive avec la sémantique initiale d'UML [AGD11].

2.3.2 Les Stéréotypes

Un stéréotype sert à affiner la signification d'un élément de modèle. Par exemple, on peut appliquer les stéréotypes "*call*", "*create*", "*instantiate*", et "*receive*" aux relations d'utilisation afin d'indiquer d'une manière précise comment un Elément/Composant de modèle utilise l'autre. On peut ainsi employer un stéréotype afin de décrire un élément de modèle qu'on veut distinguer d'un autre élément de modèle, quant à sa signification ou bien son utilisation. Les stéréotypes peuvent avoir des propriétés, appelées définitions étiquetées. Lors de l'application d'un stéréotype à un élément de modèle, les valeurs des propriétés sont appelées *valeurs marquées*.

A partir de ces stéréotypes des terminologies, une syntaxe, des notations spécifiques sont inclus afin d'ajouter de la sémantique. Prenons l'exemple de la modélisation des composants spécifiques tels que les ports, en UML on peut modéliser un composant, mais il n'existe aucun moyen pour préciser si c'est un port de sortie ou d'entrée. Cela est possible en UML MARTE ou on peut étendre le port par un stérérotype possèdant des propriétés spécifiques, afin de spécifier si c'est un port d'entrée ou de sortie.

2.3.3 Le Profil UML MARTE

Le profil UML MARTE (Modeling and Analysis of Real Time and Embedded systems) [Gro09a] adopté par l'OMG (Object Management Group) a été proposé comme un nouveau profil. Ce standard est une contribution à l'approche de développement dirigée par les modèles pour les applications temps-réel embarqués pour fournir une base, qui permet d'utiliser tout ce qui est autour de l'ingénierie des modèles pour la modélisation basée sur les descriptions (model-based descriptions) des applications. Ces concepts de base sont par la suite raffinés afin d'éliminer les ambiguités sémantiques et de fournir des supports pour les étapes de spécification, de conception. Ainsi de modéliser des concepts qui lui permettent d'être utilisable pour la spécification formelle des SETR. Ce profil s'appuye sur des outils existants (UML) qui utilisent des formalismes et des interfaces graphiques (appelé aussi langages graphiques). Ces interfaces ont été customisés, en injectant ponctuellement des concepts spécifiques afin d'étendre UML, et d'introduire des notions pour la vérification formelle (CCSL, Timesquare).

2.3.4 L'Intéret du Profil MARTE

L'un des désavantages d'utilisation du profil, est qu'on est contraint par UML, même si la sémantique de ce dernier n'est pas précise. D'autre part, UML est déja trop grand. L'autre inconvenient d'utilisation de profils, est qu'on ne peut faire que des extension légères : Comme par exemple l'ajout des stéréotypes, qui doivent respecter la sémantique d'UML. Cependant, UML est vue comme une boite à outils qui nous permet de modéliser, le travail qui reste à faire est une contribution de fournir des moyens pour ajouter de la sémantique de ces interfaces et par la suite faire de la vérification. Même si MARTE a quelques lacune pour modéliser des systèmes et leurs éxigences, on peut constater plusieurs intéréts :

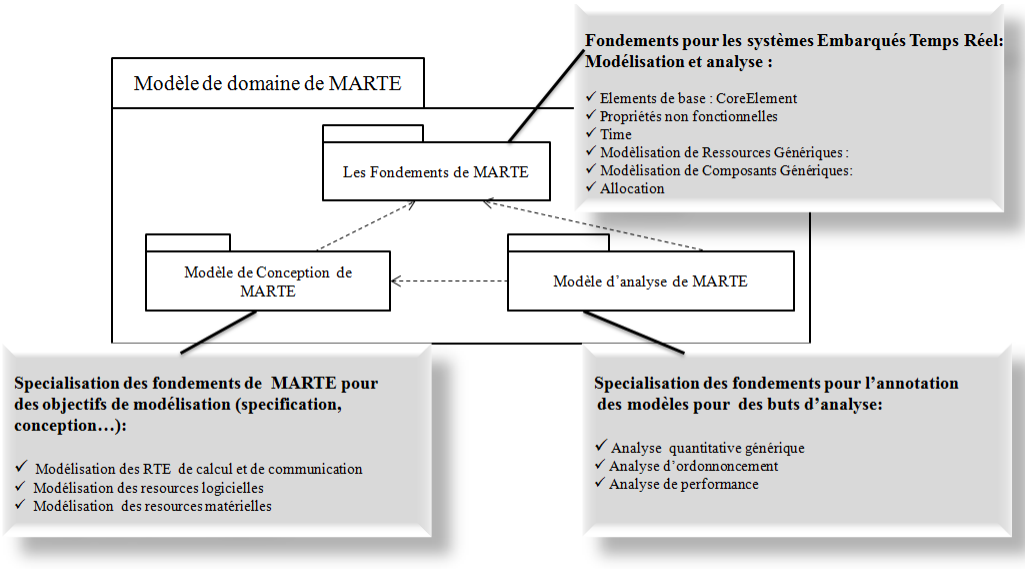

Figure 2.1: Les différents packetage de spécifications du profil MARTE

a. Le profil MARTE fournit un modèle formel de temps, un aspect absent dans UML SPT, pour l'annotation exacte des objets UML avec des informations temporelles, s'inspirant des langages synchronnes [Benv03] et des modèles des signaux marqués (tagged signal model) [LSV98]. Il fournit une sémantique au diagramme d'activités d'UML avec un modèle de temps (Time modeling et CCSL): Ce qui permet de bénéficier d'une sémantique par rapport à ce qu'on perd dans UML, qui devient un inconvénient pour la génération du code ainsi pour les aspects de simulations.

b. Les spécifications de MARTE permettent de fournir une modélisation des élements pour représenter l'application (plateforme logicielle), l'architecture (plateforme d'exé--cution) et ainsi de définir le mappage (the mapping) des fonctionnalités de l'application vis à vis des ressources et des services des architectures (Un support pour spécifier des aspects structuelles, comportementaux, fonctionnels et non-fontionnels).

c. Ce profil offre un ensemble riche et suffisant de concepts pour la conception des SETR à haut niveau, enrichit pour étre adapter à des domaines et des plateformes spécifiques, avec une méthode de modélisation unifiée des aspects logiciels et matériels afin d'amélioer la communication entre les développeurs. Il permet également de faciliter la construction de modèles sur lesquels on peut faire des prévisions quantitatives en tenant compte des caractéristiques du matériel et logiciel.

d. MARTE permet la distinction entre un composant matériel et logiciel -(cf. Figure 2.2 dans les packages DRM , SRM et HRM), cette distinction est crucial dans certain systèmes embarqué tel : *Syteme On Chip*: SOC. Cette distinction facilite le partage des taches entres les differents équipes de developpement. Tout en permettant aussi la performance et l'intéroperabilité entre les outils de développement utilisés pour la spécification, la conception, la vérification, ainsi que la génération du code, etc.

e. MARTE favorise la construction des modèles qui peuvent étre utilisés pour la prédi--ctions quantitatives concernant les fontionnalités des SETR en tenons en compte les caractéristques logicielles et matérielles.

f. UML/MARTE couvre une large gamme (range) de modélisation, avec plusieurs couches d'abstractions, avec la capacité de modéliser à partir des exigences (modeling capabilities from requirements), la conception de haut niveau, la conception détaillée, jusqu'à la génération de code.

2.3.5 L'Architecture du Profil MARTE

La structure du profil MARTE englobe deux préoccupations : l'une est pour modéliser les fonctionnalités des SETR, et l'autre pour annoter les modéles d'applications d'une façon à supporter et faciliter l'analyse des propriétées de ces systémes. Cela est illustré dans le packetage du modèle de conception de MARTE : *MARTE Design Model*, (Cf. Figure 2.1 et 2.2). Ainsi que le *cluster* des trois packetages *MARTE Analysis Model*, respectivement. Ces deux parties partagent des préoccupations en commun, avec la description du temps ainsi que l'utilisation des ressources concurrentes, qui sont incluses dans le package de fondement de MARTE nommé : *MARTE Foundation*.

Les fonctionnalités de la modélisation d'analyse (*Analysis Modeling*) sont subdivisées dans la partie du fondement générique (*Foundational Generic Part*) dans le package : *Modélisation générale d'analyse quantitative* (*General Quantitative Analysis Modeling*), et

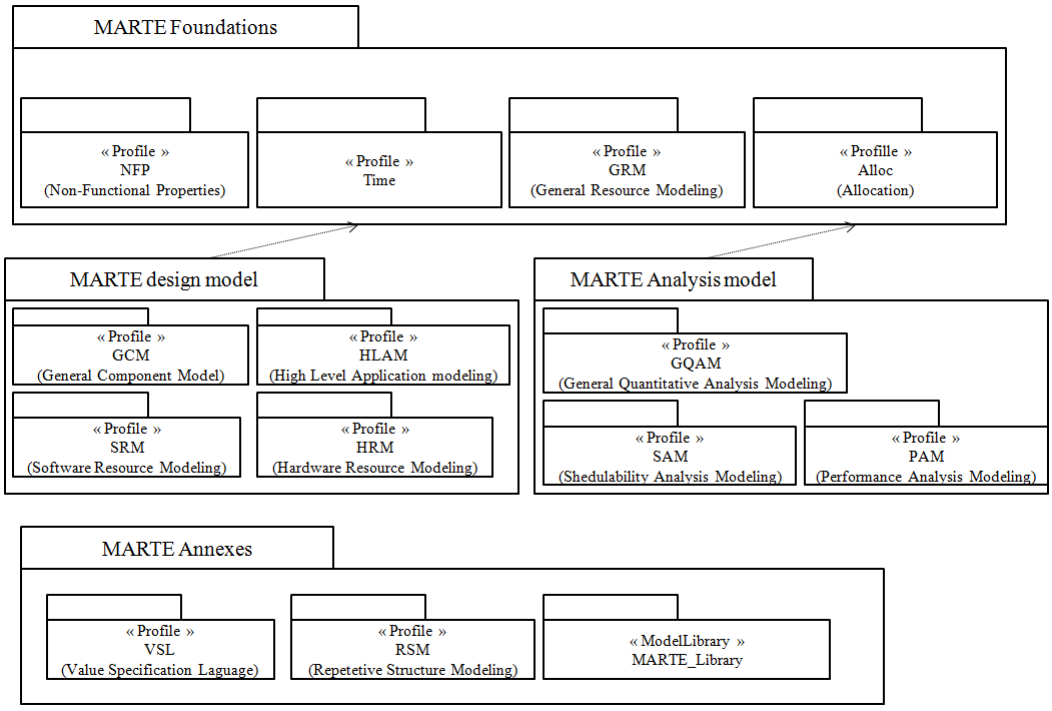

Figure 2.2: Architecture globale du profil MARTE

dans deux autres packages pour les domaines de l'analyse sépcifique. Ces deux premiers domaines d'analyse spécifique sont entiérement liés au temps. De plus, la structure du profil permet d'inclure des domaines d'analyses additionnels, comme par exemple : la mémoire utilisée, la consomation d'énérgie ainsi que la fiabilité.

2.3.6 Fondement pour les Techniques Dirigées par les Modèles : *A Foundation for Model Driven Techniques*

Le profil est destiné à fournir une base pour l'application de transformations de modèles UML dans une grande variété de modèles d'analyse. En effet, l'environnement d'exploitation du profil doit étre composé d'un ensemble d'outils, y compris les transformateurs de modèle, comme le montre la Figure 2.3. La chaîne de transformation illustre la façon dont le modèle est prévu pour être transformé par l'intermédiaire de la sortie XMI *Extensible Markup Language*[OMG11], en un format lisible par un outil d'analyse. La ligne en pointillés indique une voie de retour potentiel de réimporter les résultats de l'analyse dans les diagrammes UML. Une autre voie de retour existe de l'analyse jusqu'au modeleur.

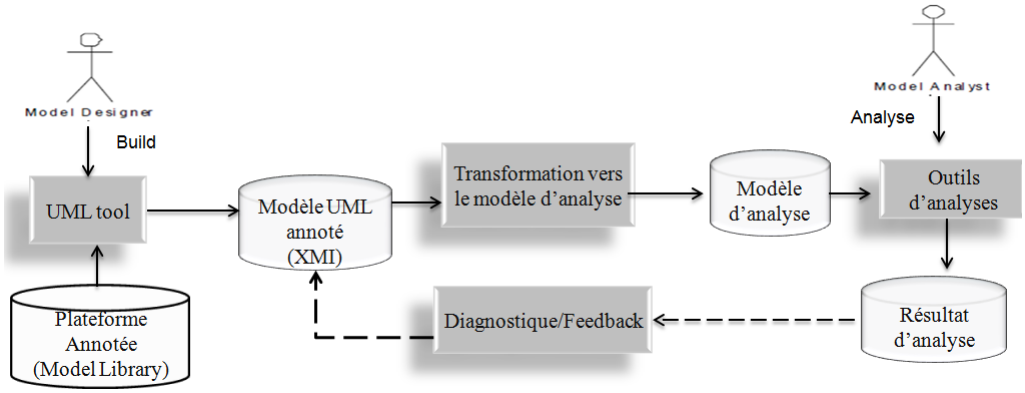

Figure 2.3: La chaîne d'outils pour l'analyse d'un modèle

MARTE est organisé en différents packages, classés en trois parties. Ce classement permet simplement de mieux se repérer à l'utilisation du standard. Cependant, MARTE est vu comme un catalogue de concepts qui sont mis à disposition, et dans lequel l'utilisation consiste à appliquer certains stéréotypes au modèle UML, en fonction des besoins. Les parties de modélisation fournies des supports exigés de la spécification à la conception detaillée (*Detailed Design*) des caractéristiques des systémes SETR. Ces concepts sont organisés d'une façon héarchique en diffèrents sous packages. Par exemple, la conception des SETR nécéssite la modélisation des propriétés qui mesure les services offerts et requis à l'aide des ressources de l'application.

2.3.7 Le Fondement de MARTE : *MARTE Foundation*

Le noyau de MARTE est décrit dans la spécification MARTE du packetage *Foundation*. Les notions fondamentales sont définies dans cette partie.

Les Elements de Base de MARTE: *Core Elements*

Les élèments de bases de MARTE décrits dans la partie *Core Elements*, définissent le fondement de base pour l'approche basée sur le modèle (*Model-Based Approach*) spécialement pour le domaine des SETR, comme le modèle causale. Nous constatons l'existance du package dédié à décrire les propriétés non fonctionnelles "*Non Functional Properties*". Les concepts présentés dans cette clause servent à une base générale de description de la plupart des éléménts du reste de la spécification. C'est un ensemble globale de concepts reliés, divisées en deux packetages:

a. Le packetage *Foundations* qui contient les éléments de base utilisés pour représenter la nature de la dualité des descriptions des instances (dual descriptor instance) des entités du modèle. Ce concept peut servir à différents besoins de modélisation et d'analyse, servant généralement à la modélisation de la structure.

b. Le package *Causality* décrit les éléments de base nécéssaires pour modéliser les comportements et leurs sématiques d'exécution temporelle (run-time semantics).

La Modélisation des Propriétés Non Fonctionnelles en MARTE

Dans le contexte des approches de dévelopement dirigé par les modèles pour les SETR, la modélisation des propriétés non fontionnelles est d'une importance fondamentale, impliquant un certain nombre de décisions de conception. Ces propriétés fournissent des informations sur différentes caractéristiques. Comme par exemple : les délais, les politiques d'ordonnoncements, les deadlines, ou méme sur l'usage de la mémoire.

MARTE permet d'utiliser certains sous-paquetages, en particulier pour l'expression des contraintes non-fonctionnelles. Cette partie du profil est vue comme un modèle de construction pour les propriétés non fontionnelles. Cette partie définit un intergiciel pour l'annotation des modèles avec des informations non-fonctionnelles quantitatives et qualitatives (Figure 2.4).

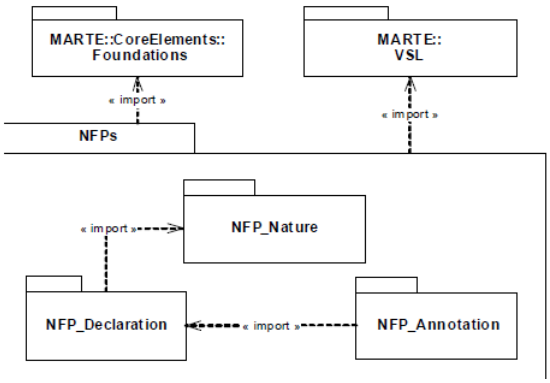

Figure 2.4: La stucture et les dépendances du packetage de modélisation des propriétés non fonctionnelles [OMG10].

L'intergiciel de la modélisation des propriétés non fonctionnelles en MARTE, traite un ensemble d'exigences. Parmi celle-ci, est la capacité de préciser quelles propriétés non fontionnelles doivent étre prise en compte et comment sont elles decrites. Cette partie de spécification permet ainsi de préciser comment les instances particuliéres de ces propriétés

sont reliées aux éléments du modèle UML. Elle permet également de définir la relation entre ces propriétés, et pemrettre l'expression des contraintes sur ces propriétés afin d'exprimer les exigences sur le modèle du systéme. L'utilisabilité des annotations peut permettre à son tour de minimiser des efforts de conceptions.

Prenons l'exemple des modèles de système de calcul (computing system), ces types de système décrivent leur architectures et leur comportement par l'intermédiaire des éléments de modèle (e.g., les resources, les services de resources, les fonctionnalités du comportements, les opérations logiques, les modes de configurations, etc), et les propriétés de ces élèments des modèles. Il est donc pratique de regrouper les propriétés des applications en deux catégories : les propriétés fonctionnelles, qui initialement liées au but de l'application, et les propriétés non fontionnelles, qui sont plus préoccupées par leur aptitude à l'usage (i.e.,Comment çà se fait ou comment çà doit étre fait ?). Les propriétés fonctionnelles et non fontionnelles sont une spécialisation d'un concept plus général de la propriété de la valeur, reliées à une quantité spécifique.

Le Modèle Générique des Ressources : *Generic Resource Modeling*

Afin de décrire des resources abstraites nécessaires à la modélisation de plateformes générales dédiées à l'execution des SETR. MARTE permet de spécifier des modèles de ressources à un niveau bien spécifique du système, dans un packetage appelé "*Generic Resource Modeling*". Le modéle des ressources générale (GRM) inclue des fontionnalités qui sont requises pour :

- La modélisation des plateformes d'execution à différents niveaux de détails. Le niveau de granualité utlisés pour une modéisation d'une plateforme dépend des préoccupations motivant la description de la plateforme par exemple le type de la plateforme, de l'application, ou bien le type de l'analyse effectuer à un modèle.
- La modélisation des plateformes matérielles (Les unités de la mémoire, les cannaux physique de communication ainsi que logicielles). Et permettre ainsi de fournir des constructions de modélisation fondamentales qui sont raffinés pour supporter des conceptions des modèles d'analyse.

Les packetages "*Software Resource Modeling (SRM)*" et "*Hardware Resource Modeling (HRM)*" fournissent une spécialisation du modèle général des resources, reliés respectivement, à des plateformes logicielles et matérielles. La Clause de la modélisation des allocations en MARTE dite "*Allocation Modeling*", définit les concepts nécessaires pour la description des allocations lorsque'il s'agit d'allouer des applications fonctionnelles à des ressources physiques. La deuxième partie de MARTE concerne à définir les concepts de model de base de conception "*MARTE Model-Based Design*" afin de spécifier des plateformes d'execution matéielles et logicielles des modèles de calcul et de communication :

Contenant "SRM" et "HRM" inclues dans '*Detailed Ressource Modeling : DRM*, considéré comme un raffinemment du packetage "GRM".

Le Modèle Détaillé des Resources : *Detailed Resource Modeling*

La Clause "*Detailed Resource Modeling*" est divisée en deux sous-clauses respectivemment dédiées à la modelisation détaillée. Considérée comme un raffinement du packetage GRM des ressources logicielles (sous clause 14.1, SRM, Software Resource Modeling) et materielles (la sous clause 14.2, HRM, Hardware Resource Modeling), cette partie prend en compte les concepts nécessaires pour spécifier des modeles de calcul et de communication, ainsi que les plateformes d'excecution logicielles et materielles. Ce niveau de modélisation détaillé vise à fournir un modèle détaillé de la plateforme par un raffinement des modèles. Cependant, Le modèle alloué de ce niveau doit contenir assez d'informations pour permettre la génération du code d'implémentation pour les parties matérielles et/ou logicielles. Ce package est composé de deux sous-packages :

a. La modélisation des resources logicielles "*Software ressource Modeling SRM*": fournit des concepts de base qui concerne les resource logicielles . Nous allons détailler seulement les stéréotypes utilisés.

b. La modélisation des resources matérielles "*Hardware ressource Modeling HRM*": fournit des concepts de base qui concernent les resource matérielles.

On peut cité un exemple de stéréotype définit dans ce packetage, appelé $MessageCom--Resource$, qui définit des resources de communication pour échanger des messages. Ce stéréotype contient plusieurs attributs qui servent à ajouter des details supplémentaires de la communication.

2.3.8 Le Modèle Général des Composants : *General Component Model*

General Component Model, introduit un modèle général de composant , compatible avec les modèles de composants connus comme : SysML, CCM, AADL et EAST-ADL. Ce modèle rassemble des concepts nécéssaires pour une modélisation à base de composants.

2.3.9 La Modélisation des Applications de Haut Niveau :*High-Level Application Modeling, HLAM*

La clause de MARTE concernée pour la modélisation des applications de haut niveau, définit des concepts de modélisation de haut niveau pour la conception des fontionnalités qualitative et quantitative des SETR (e.g., la synchronisation et la concurrence), liées

à la modélisation à base de composants. Ce packetage dépend des spécifications de "*GRM*" et des élements de base de MARTE (*CoreElements*). Si on compare les domaines d'applications, le développement des systémes temp-réel nécéssite des possibilités de modéliser d'un coté les fonctionnalités quantitatives (deadline, periode...), et d'un autre coté, les fonctionnalités qualitatives reliés au comportements et la conccurence.

Cette clause fournit des concepts d'un niveau élevé pour la modélisation quantitatives des fonctionnalités temps-réel en utilisant des objets actifs et passifs : *Real-Time Unit* (RTUnit) et *Protected Passive Unit* (PPUnit).

Les Unités Actives Temps-réel en MARTE

La notion de l'unité temps réel "*RTUnit*", fournit par le profil MARTE dans le paquetage "*HLAM*" supporte la notion d'entité concurrente. Un "RTUnit" est définit comme un bloc qui fournit et/ou requit un ensemble de services temps réel (*RTService*). Les comportements temps-réel peuvent étre décomposés en actions temps-réel (*RTAction*). Une unité temps-réel est une unité de concurrence qui encapsule dans une seule entité l'objet et les paradigmes du processus, qui veut dire que le controle de la concurrence est encapsulé à l'intérieur d'une unité.

Chaque unité temporelle peut invoquer des services ou bien d'autre unités temporelles, comme par exemple l'envoi des sigaux ou bien des données. Les unités temporelles sont vues comme des tâches qui peuvent satisfaire différentes requêtes à partir de différentes unités temporelles, permettant ainsi le parallelisme si necéssaire. Ces unités possèdent des controlleurs de concurrence et de comportement pour gérer les contraintes reliés à des messages par rapport à son état concurrent ainsi que les contraintes attachées au contraintes d'execution du message. Une unité temporelle est similaire d'un objet actif en UML, mais avec une sémantique de description plus detaillée. Ces unités possédent une ou plusieurs ressources odonnonçables (*GRM::Scheduling::SchedulableResource*). Si les attributs dynamiques sont misent à vrai, les resources ordonnonçables sont crées dynamiquement si nécéssaire. Cependant, une unité temporelle peut étre vu comme une ressource d'execution autonome, capable de gérer en méme temps différents messages. Comme elle peut gérer la concurrence et des contraintes temporelles attachées à des méssages reçus.

Une application doit posséder au moins une *RtUnit*. Chaque unité temporelle principale (indiquée par mettre la valeur $isMain$ à vrai) commence à invoquer le pricipale sevice temps-réel.

Une unité temps-réel doit avoir un ou plusieurs comportements (GCM::Structured--Component et CoreElements::Causality::CommonBehavior::BehavioredClassifier). Cette dernière peut avoir aussi une seule file d'attente de messages pour sauvegarder les messages reçus une fois l'execution est débutée.

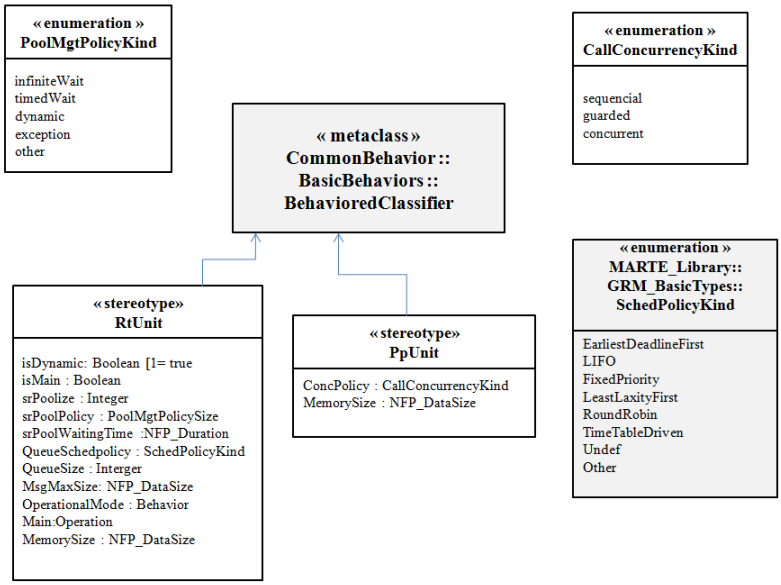

Figure 2.5: Les stérérotypes RTUnit et PPUnit de MARTE :: HLAM

A l'exeption du point de variation de la sémantique reliée à la selection de l'évènement est résolue par la possiblité de spécifier une politique d'odonnencement de la file. Les messages contenus dans la file, peuvent représenter un appel d'opération, des occurences d'un signal ou bien une réception de données. Chaque message peut étre utilisé pour déclancher l'execution d'un comportement propre à l'unité (i.e., décrit par son propre service). La taille du méssage peut étre infinie ou bien limitée. Dans le cas ou elle est limitée, la taille de la file est spécifée par son attribut *maxSize*. De plus, une *RtUnit* possède son propre comportement bien spécifique, appelé le mode opérationnel. Ce comportement prend souvent la forme de comportement à base d'état ou les états représente une configuration de l'unité et les transitions dénotent les reconfigurations de cet unité.

Les Unités Passives Temps Réel en MARTE

Dans la modélisation de la conccurrence, il est primordiale d'étre capable de partager des informations. Pour cela, en MARTE la notion d'unité passive est introduite avec le stéréotype (*PpUnit*), dénotée dans la Figure 2.6. Les unités passives protégées, spécifient les politiques de concurrence pour tout les services fournis par l'attribut (*concPolicy*). Ou bien localement, par l'intermédiare de l'attribut (concPolicy) du service temps réel (*RtService*).

Les Actions Temps-réel en MARTE

Parmi les fontionnalités quantitatives pour manipuler le domaine du temps-réel concerne les aspects de communication. Dans UML, les communications sont initiées par l'execution d'actions spécifiques dites *Action*. Dans MARTE la notion d'action temps-réel est introduite par le stéréotype "*RtAction*"(une spécialisation d'une invocation d'une action est introduite par concept de "*InvocationAction*" dans MARTE::GCM package). Une action temps-réel peut spécifier des fontionalités temps-réel comme le deadline ainsi que la période. Elle peut aussi décrire la taille du message généré à l'execution ou bien le type de synchronisation par l'attribut *synchKind*.

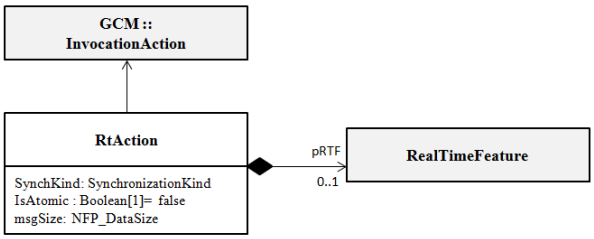

Figure 2.6: Les stérérotypes des actions temps-réel "*RtAction*"dans MARTE

2.3.10 L'Analyse Basée sur le Modèle : *Model-Based Analysis*

MARTE inclut également la modélisation basée sur l'analyse (*model-based analysis*). Dans ce contexte, le but est non pas de définir de nouvelles techniques pour l'analyse des SETR, mais plutôt d'offrir la possibilité d'annoter les modèles pour supporter l'analyse. En effet, cela permet de faciliter l'annotation des modèles avec des informations exigées pour la performance de l'analyse (perform specific analysis). Plus specifiquement, MARTE vise à la performance et à l'analyse de l'ordonnocement. Cependant, il définit également un intergiciel d'analyse géneral qui sert à raffiner et spécialiser n'importe quel type d'analyse.

La Clause 15 de MARTE, concerne la modélisation d'analyse quantitative générale (*Generic Quantitative Analysis Modeling*), dont lequel ils sont définis les concepts de base pour des techniques d'analyse spécifiques. La Clause 16, concerne la modélisation d'analyse d'ordonnonçabilité (*Schedulability Analysis Modeling*, qui specialise un intergiciel générique pour l'analyse et la performance d'ordonnonçabilité, quant à la Clause 17, est condidérée comme une modèlisation des performances, vue comme une spécialisation d'un modèle basé sur l'analyse des performances.

Le profil MARTE fournit également un langage de spécification de valeurs nommée

VSL (*Value Specification Language*) sous forme d'un méta-modèle et d'une grammaire. Ce langage permet d'annoter les modèles par des contraintes portant sur des grandeurs physiques, tel que le temps et les puissances.

2.4 La Modélisation du temps

Le temps joue un rôle important dans les systèmes embarqués (réactifs). En effet, chaque domaine peut avoir sa propre représentation (modélisation et interprétation) du temps. Dnas [Sch94] un article de synthèse est décrit plusieurs aspects de temps en définissant des ontologies pour le temps des divers domaines de l'informatique.

Le temps physique est l'un de type de temps utilisé dans les lois de physique et de mécanique. Ce dernier est supporté dans le modèle de MARTE comme une l'horloge idéal -*IdealClock*. Dans les systèmes numériques, le temps est perçu au travers des systèmes de circuits spéciaux, appelés *horloges*, engendrants des signaux periodiques, souvent appelées *tics d'horloges*. Ce type de systèmes nécessitent plusieurs horloges, cela peut mener à des problemes de synchronisations [Mes90].

2.4.1 Le temps et le concept d'évènement : Le temps logique

Le concept d'*horloge logique* a été introduit par L. Lamport [L.L78], permettant de calculer une relation d'ordre totale sur les occurrences d'évènement dans le système. Des améliorations ont été apportées à ce type d'horloges dans le but de mieux caractériser des relations de causalité entre des évènements [SM94]. Les travaux de Carl Adam Petri [BRR87] présentent une caractéristique plus basique liant le concept de temps et la concurence. Ces travaux modélisent les évolutions des systèmes des réseaux représentant des structures d'évènements.

La notion de temps avec un ordre partiel d'un ensemble d'instants est une idée reprise dans le modèle sémantiquée du temps de MARTE. Afin de permettre des évaluations de performance temporellles, ou même plus pour des objectifs de vérification de propriétés temporelles, un modèle de temps est réduit à un *poset* [3] d'un ensemble d'instants, ce qui est considéré pas suffisant. Ce qui appel à introduire des synchronisations d'horloges sur le temps physique, voir par exemple le standard proposé dans [Gro04] (*EVoT*) *Enhanced View of Time Specification*. Le livre [Kop11] de H.Kopetz traite la modélisation du temps dans les systèmes répartis, en introduisant la notion de *(TTP) Time Triggered Protocols*, permetttant de concevoir des applications temporelles.

2.4.2 Le Modèle de Temps dans MARTE

Chaque domaine peut avoir sa propre modélisation et interprétation du temps. Dans

[3] Poset : partially ordered set

l'article [Sch94], F. Schreiber décrit plusieurs aspects du temps en modélisation et définit ainsi des ontologies pour le temps dans différents domaines. Une partie importante de MARTE concerne la modélisation du temps. L'expression des contraintes temporelles utilise le paquetage "*Time*" fournissant un modèle de temps plus fin par rapport à UML. Les concepts de temps sont décrits dans la Clause 9, qui définit le modèle de temps tel qu'il est utilisé dans MARTE.

Ce qui n'allait pas dans le méta-modéle UML, est le fait qu'on peut exprimer une infinité de type d'évènement : l'envoie de messages, l'envoie des signaux, l'appel d'opérations, etc. Mais il y'a un évènement particulier est que y'a du temps qui s'est écoulé. Ce qui nécéssite le besoin de controler à quel moment un message apparait, à quel moment sera la reception d'une telle donnée, ou bien l'execution d'un tel comportement. Il y'a aucun moyen de le faire, du fait que le temps n'apparait pas dans les premières version d'UML. Cependant, ce qui manquait dans le méta-modèle UML, est de pouvoir contrôler ces évènements. La solution proposée par l'OMG "*Object Group Management*", était d'étendre ces concepts, en ajoutant des stéréotypes, appelés *"Clock"*. En effet, les évènements dans MARTE deviennent différents de ceux d'UML, avec la particularité de pouvoir employer ces évènements comme des *références temporels*, utilisées dans le but de guider ce qui se passe dans le système. Ces références temporelles peuvent étre *logiques* ou *physiques*.

En MARTE, le temps est vu comme un ordre partiel d'instants, c'est l'idée reprise dans le modèle semantique. Le modèle de temps introduit dans MARTE complète les fonctionnalités fournies par le package *SimpleTime* d'UML 2. L'idée de base de cette spécification est la capacité de relier les actions et les événements au temps. Ces derniers doivent étre référencés explicitement à une *horloge*, en se limitant pas juste au temps physique, mais en prennons en compte à ce qu'on appel le *temps multi-formes*.

Le package de Modèle de temps de base en MARTE

Le package de Modèle de temps de base (Figure 2.7), fournit une vue structurelle du temps représentée comme un ensemble d'instants ordonnés. Ce modèle ne fait aucune référence à la notion du temps physique. Par concequent, il supporte commodément le temps logique, qui est largement utilisé dans les systèmes distribués et les langages synchrones. Ce modèle de temps se focalise sur l'ordre des instants, tout en ignorant la durée physique entre les instants successifs.

Une base de temps peut étre vue comme un conteneur d'instants. La structure de temps est spécifiée par l'attribut "*nature*" qui prend sa valeur dans l'énumération " *TimeNatureKind*", dont les valeurs possibles sont "discret" ou bien "*dense*". Concernant le temps dense, pour chaque paire d'instant, il existe toujours au moins un instant entre les deux. Une base de temps posséde un ensemble ordonné d'instants, ou seulement les ensembles dénombrables sont considérés.

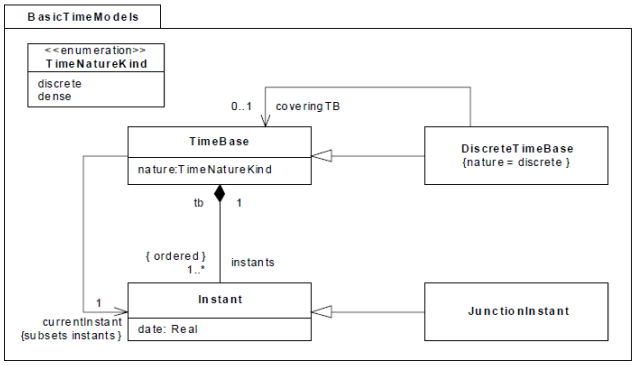

Figure 2.7: Le diagramme de base du modèle de temps en MARTE [OMG10]

Pour une base de temps discrete, les instants peuvent étre indexés par un entier positif. Autrement, concernant la base de temps dense, les instants peuvent étre indéxés par des nombres rationnels. Notons que des modèle de temps continue, les indices peuvent étre des nombres réels, qui ne peuvent pas étre totalement représentés par des ensembles dénombrables. Du fait que la sémantique du comportement d'UML est uniquement traitée par des comportements discrets. Le temps physique est considéré comme une progression continue et non limitée des instants physiques, d'ou le temps physique est supposé progressé de façon monotone.

Les Relations des Instants du Temps Concret dans MARTE

Comme le montre la Figure 2.8, trois sous classes concrétes de la classe abstraite "*TimeInstantRelation*" sont définies : La relation de "*Coinsidence*", de "*Precedence*" et le "*TimeIntervalMembership*". La relation de "*Coinsidence*" est une forme robuste de relation d'instant temporels. Les instants de jonction appartenant à différentes bases de temps peuvent être coïncidents.

Au point de vue modélisation, la notion de *Coïncidence* n'a pas nécéssairement ce sens relativiste stricte. En effet, Ceci peut représenter des synchronisations d'horloges ou méme des choix de conceptions diverses. Par exemple, la relation de *Coïncidence* doit étre symétrique et transitive. De plus, chaque jonction d'instant est supposée coinsidée avec soi méme, ceci indique que la relation de *Coïncidence* est une relation d'équivalence sur les instants. Une éxigence stricte, est que l'ajout d'une *Coïncidence* n'introduit pas des dépendances cyclique dans l'ordre temporel. Mathématiquement, l'ensemble d'instants liés à la relation de coïncidence doit être un ensemble partiellement ordonné. Par consequent,

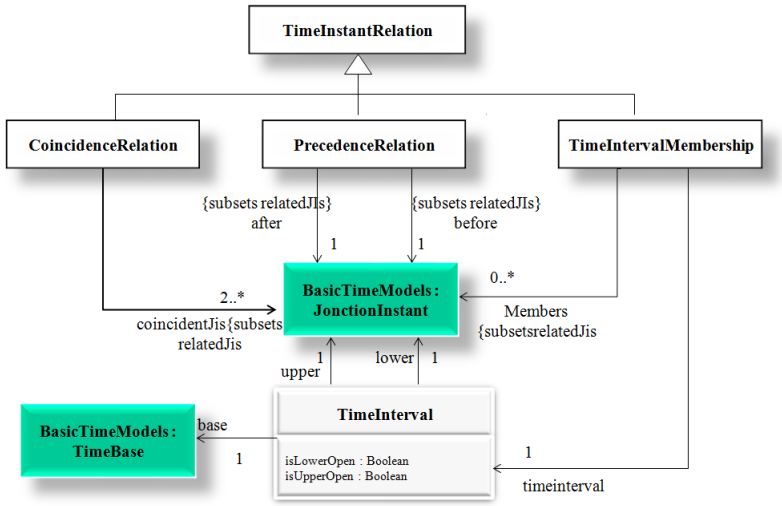

Figure 2.8: Diagramme des relations d'instants temporels *"TimeInstantRelation"* du modèle de temps [OMG10].

la relation de *Coïncidence* est souvent représentée par des diagrammes liant des paires d'instants coïncidents.

La Figure 2.9 illustre un exemple de base de temps multiple, contenant trois bases de temps. La junction des instants *a2* et *b2* sont coïncidents. De méme pour *b2* et *c2*, *a2* et *c2* qui sont également des instants de jonction coïncidents (par transitivité).

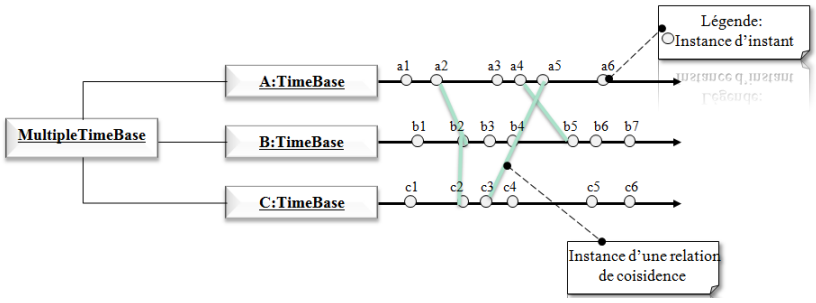

Figure 2.9: Exemple de base de temps multiple avec la notion de Coinsidence [OMG10].

La relation de *Precedence* entre les jonctions d'instants à partir de différentes bases de temps, est une relation d'instants temporels, envisagée plus faible que la *Coinsidence*. Elle

exprime une dépendance directionnelle : Un instant de jonction appartenant à une base de temps qui peut précéder ou peut être suivit des instants de jonction appartenant à d'autres bases de temps.

La Base de Relation du Temps Concrêt dans MARTE

Les relations de base temporelles sont un moyen de haut niveau pour imposer les dépendances entres les jonctions d'instants.

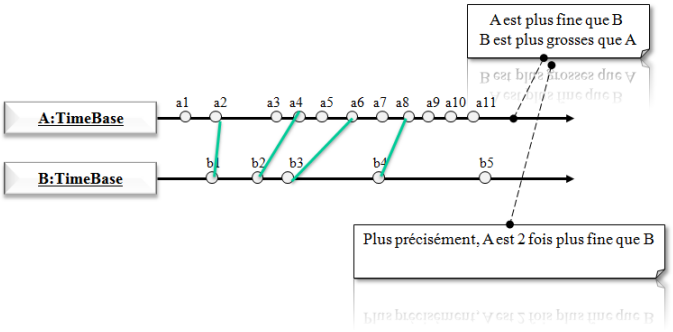

Figure 2.10: Exemple de relation entre deux bases de temps[OMG10]

Une relation de base de temps spécifie un ensemble de relations d'instants temporels. Comme le montre la Figure 2.10. Pour chaque deux bases de temps A et B, on définit une relation, d'ou A est plus fine que B si pour chaque instant de jonction dans B il existe un et un seul instant de jonction coinsident dans A. Cette relation peut être caractérisée par un mappage M de la plus ***grosses ***(coarser) base B jusqu'à la base la plus fine base A. Quand la base de temps la plus fine est également une base de temps discrète, des relations plus précises peuvent être spécifiées. Par exemple, la relation *k-finer* est définie comme suit . A est *k-finer* que B pour k entiers, k sup ou égale à 1, si A est plus fine que B et pour chaque deux consécutive instants dans B, il exist k1 instants entre les correspondants instants coinsidents dans A. La Figure 2.10 illustre un exemple ou k=2.

Des relations prédéfinies sont suggérées dans la structure de temps des relations de la librairie de MARTE "*TimeStructureRelation*". Les sémantiques de ces relations sont spécifiées dans OCL[Gro12].

Accès au Temps : Les Horloges en MARTE

D'une façon générale, les horloges sont considérées comme des dispositifs qui permettent de mesurer la progression du temps Physique. Le profil MARTE adopte un concept plus abstrait, dont une horloge est définie comme un élément du modèle qui permet l'accés à la

stucture du temps. Cependant, une horloge fait référence à une base de temps discréte. Par conséquent, cette dernière fait référence aux instants de cette base de temps. En MARTE, une horloge peut être associée à un événement, d'ou les occurences de ces évènements correspondent aux *tics* de cette horloge.

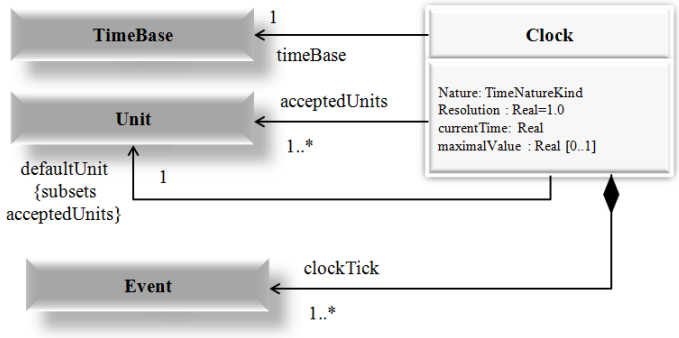

Figure 2.11: Accés à la stucture de temps, *Clock*[OMG10]

Les attributs "*resolution*" et "*nature*" de chaque horloge caractérisent les valeurs de temps données par l'horloge. En effet, l'attribut *resolution* représente l'écart minimal entre deux valeurs pour qu'elles soient distinguables. En d'autre terme, cet attribut indique la granualité de l'horloge. La nature de l'hologe est spécifiée dans l'attribut *nature*, qui peut être dense ou discrete.

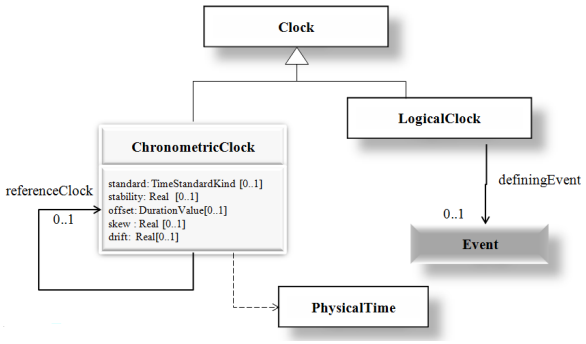

Figure 2.12: Horloges logiques et chronométriques[OMG10]

Il existe deux sous classes de la classe abstraite *Clock* : *LogicalClock* et *Chronometric-Clock*, cette dernière fait référence au temps physique (d'une façon implicite). Par contre, une hologe logique fait référence à un évènement, dont les occurences définissent les instants de cette horloge.

Les stéréotypes "*Clock*" sont liés (map) aux éléments du domaine des horloges dans l'annexe F de MARTE (sub clause F.3.2, [OMG10]). Une horloge est un élément du modèle représentant une instance d'un type d'horloge *ClockType*. Une horloge "*Clock*" donne accés au temps. Cette dernière est définie dans le domaine du temps dans MARTE "*TimeDomain*". Une horloge dans MARTE est liée à une base de temps "*TimeBase*" dans le domaine de la sémantique. L'attribut *UnitType* spécifie l'unité de l'horloge. Une horloge est caractérisée par sa résolution, et par son "*Offset*" (sa valeur d'instant initial) ainsi que sa valeur maximale. Une "*Clock*" peut être une propriété stéréotypée, ce qui peut étre utilisé dans les structures composées et les infrastructures.

Alternativement, n'importe quel évènement peut être vu comme une horloge, du fait que le stéréotype *Clock* étend la méta-classe "Event". Cette extension met en évidence le concept du domaine de "$defingEvent$" (voir sub clause F.3.16) et permet de définir des contraintes d'horloges sur un évènement précis, et non pas seuleument sur un "*TimeEvent*". Dans ce cas le type de l'horloge doit être *logique*.

Les Contraintes d'Horloges : ClockConstraint

Le stéréotype "*ClockConstraint*" correspond à l'élèment du domaine *ClockConstraint*, dénoté dans l' Annex F de MARTE (sub clause F.3.3). Une contrainte d'horloge impose une dépendance entre les horloges ou de types d'horloges, (ç-à-d, une contrainte d'horloge fait référence à un ensemble d'horloges ou de types d'horloges).

La spécification de la contrainte est généralement une expression opaque qui utilise un langage dédié: *CCSL,Clock Constraint Specification Language*, définit dans Annex C de MARTE. Une contrainte peut définir une ou plusieurs relations d'horloges, souvent un nombre infinits des relations d'instants. Quand des relations d'instants sont reliés à une *Coinsidence*, l'attribut *isCoincidenceBased* doit être mis à vrai. Dans le cas d'une *Précedence*, l'attribut isPrecedenceBased doit être mis à vrai (noter que ceci c'est pas exclusif). Cependant, si seulement *isCoincidenceBased* est à vrai, la contrainte et pûrement synchrone. Dans le cas ou seulement isPrecedenceBased est à vrai, la contrainte est dite pûrement asynchrone. Mise à part ces deux distinction structuelle, une contrainte peut aussi définir des éxigences reliées à des aspects chronometriques d'horloges comme par exemple (la stabilité, offset, skew...). Dans ce cas l'attribut "*isChronometricBased*" doit être mis à vrai.

Le Type d'Horloge : ClockType

Le stéréotype *ClockType* correspond au domaine d'élèment de la base de temps $TimeBase$ de MARTE, dénoté dans l'Annex F (sub clause F.3.21). Il est aussi indirectement relié au stéréotypes $Clock$ (sub clause F.3.2) et *ChronometricClock*(sub clause F.3.1). Un *Clock-*

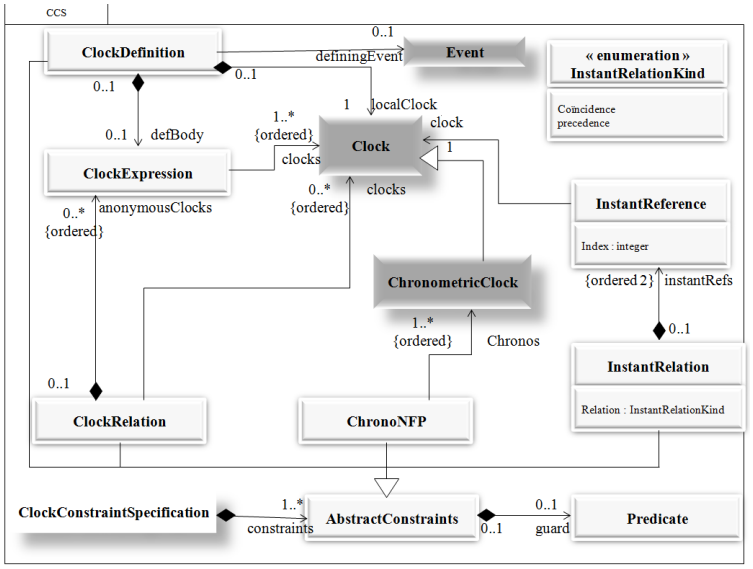

Figure 2.13: Les Contraintes d'Horloges [OMG10]

Type est vu comme un classificateur d'horloges. Les attributs de ce stérotype définissent la nature du temps représenté (discret ou dense) ainsi que le type des unités, que ce soit des instants d'une horloge *logique* ou *chronométrique*.

Afin de créer des horloges, il suffit d'importer le package de MARTE du modèle de temps, *MARTE::TimeLibrary*. Il est nécéssaire de commencer d'abord à définir des types d'horloges. Par exemple, dans la Figure 2.14, nous déclarons un type d'horloge, par la création d'une classe nommée *Chronometric*, dont laquelle on applique le stéréotype *Clock-Type*. Par la suite un package est crée, nommé *ApplicationTimeDomain*, dans lequel le stéréotype *TimedDomain* est appliqué. Dans ce package, trois instances d'horloges sont crées : Deux horloges discretes de type *Chronometric*, *cc1* et *cc2* et une instance de l'horloge *idealClock*, nommée *idealClk*, qui est une horloge à temps dense. Le stéréotype *Clock* est appliqué à ces hologes Concernant le corp de la contrainte, les relations entre les horloges déclarées sont exprimées en *CCSL*.

La Figure 2.15 illustre le domaine de temps de l'application, On y trouve des instances, représentant les horloges, et des contraintes d'horloges. *pr* est une instance du *Processor*, considérée comme une horloge logique stéréotypée par *Clock*. Par default *tick* est l'unité des horloges logiques déclarées. *c1* et *c2* représentent deux instances du *Controleur*. Le *TimedEvent*, "tev" est associé au déclenchement d'un évènement bien précis, référencé par l'horloge *on=idealClk*.

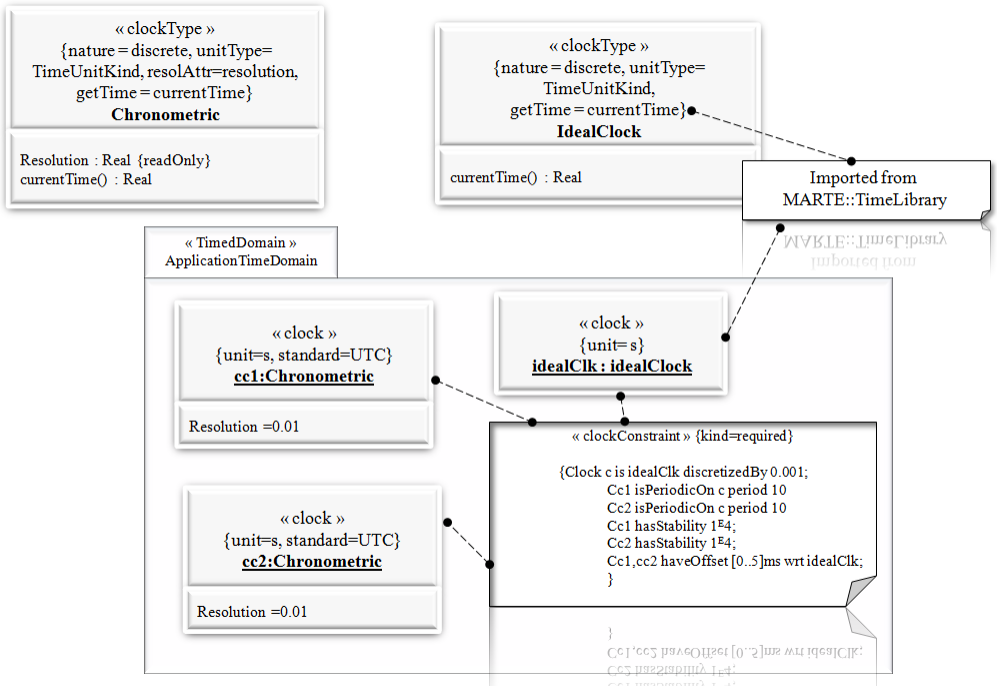

Figure 2.14: Exemple de Spécification d'horloges Chronométriques en MARTE[OMG10]

2.4.3 Discussion

UML-MARTE n'impose pas une sémantique précise, ce qui permet de l'utiliser de façon très souple. Cependant, soit on donne peu de détails dans l'architecture parce-que beaucoup d'informations sont implicites. Autrement, une modélisation trop précise devient difficile à comprendre, et il peut être plus compliqué de modéliser que de programmer directement. Par ce fait, dans notre démarche de transformation, on a choisis de s'arrêter à un niveau d'abstraction raisonnable, qui dépend de ce qu'on veut transformer. Ce niveaux de modélisation est choisi en sélectionnant des stéréotypes de MARTE qui nous paraissaient les plus pertinents pour notre transformation. L'un des objectifs de cette thése est de fournir des régles méthodologiques permettant de faciliter les activités de conception et d'analyse, en prenant en compte qu'une sous partie du Profil MARTE.

En effet, nous ne prenions pas en compte l'aspect matériel et nous limitons l'analyse au niveau fonctionnel – d'ou la prise en compte de l'application ainsi que l'architecture fontionnelle. Nous adoptons cette approche pour ajuster l'activation de comportement des composants communicants du système, dans le but de valider des contraintes temporelles exprimées en *CCSL*.

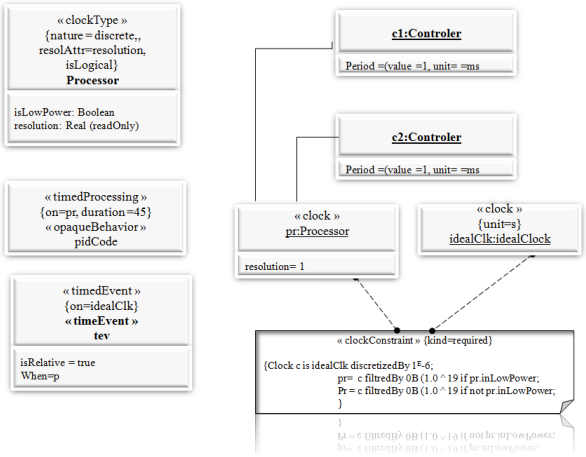

Figure 2.15: Exemple de spécification d'horloges logiques en MARTE[OMG10]

Dans notre approche de transformation, nous envisageons à utiliser une approche de type DSL "*Description Specification Language*", afin de passer du formalisme MARTE à la vérification. Cela est dans le but d'avoir une sémantique assez présice, puisque notre langage intermédiaire est censé être utilisé seulement pour une plateforme donnée, possèdant une sémantique d'exécution bien définie, ce qui évite beaucoup de conflits sémantiques et d'ambiguité.

En contrepartie, on est restreint à la plateforme prévue, puisqu'il s'agira typiquement d'extraire les aspects sémantique du modèle MARTE. Le problème qui se posera immédiatement est que la sémantique d'UML n'est pas très précise: c'est surtout une syntaxe, pour permettre l'adaptation aux besoins. Nous indiquons donc clairement comment nous interprétons les différentes constructions MARTE. Nous expliquons dans le chapitre suivant les techniques de vérification par model checking et l'exploration des modèles à vérifier avec la plate forme OBP.

Chapitre 3

Techniques de vérification par model-checking

"*It has been an exiting twenty years, which has seen the research focus envolve [...] from a dream of automatic program verification to a reality of computer-aided design debugging.*"

Thomas A. Henzinger

3.1 Introduction

Nous nous intéressons ici à une classe particulière de programmes : *les systèmes réactifs*. Parmi les caractéristiques remarquables de ces systèmes : (1) les types de données gérés est plutôt simples alors que le contrôle est complexe (e.g., fortes contraintes de l'ordre d'exécution de tâches, exécution de plusieurs composants en parallèle). (2) ils maintiennent un ensemble d'intéractions (e.g., ils intéragissent avec l'environnement par le biais de capteurs ou même des actionneurs).

Les propriétés que l'on veut prouver sur ce type de systèmes sont généralement différentes de celles que l'on veut prouver sur des programmes standards. Par exemple, dans tels systèmes on veut prouver des propriétés d'entrelacements d'évènements durant l'exécution d'un programme, ou par exemple est t-il possible durant l'execution de revenir dans un état quelquanque.

Pour les systèmes réactifs l'aspect temporelle est très élaboré. Afin d'exprimer des propriétés temporelles, on utilise ce qu'on appel *la logique temporelle*. De nombreuse logique temporelles existent, telles que CTL, LTL et CTL* [PNU77][PNU81] [eEAE81] [eJYH82] [eJYH86]. Lors de la phase de spécification, ces logiques peuvent être utilisées avantageusement du fait qu'elles expriment le comportement d'une manière non ambigue.

Dans ce chapitre, nous décrivons l'outillage qu'on utilise dans notre approche de la vérification formelle de propriétés sur des modèles logiciels. L'outillage de vérification exploite des scenarios de contextes afn de réduire la compléxité des modèles durant les vérifications.

3.2 Model checking

"Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for that model" [Clarke Emerson 1981].

Selon la définition de [Bar08], un model checking est un ensemble de techniques de vérification automatique de propriétés temporelles sur un système réactif. L'algorithme de model checking prend en entrée : (1) une abstraction du comportement du système réactif, sous forme d'un système de transitions. (2) une formule d'une certaine logique temporelle, et répond si l'abstraction satisfait ou non la formule. On dit alors que le système de transitions est un modèle de la formule, d'où le terme anglais " *model checking*". Plusieurs attentions ont été prétées aux techniques de model checking. Ceci est fondamentalement dû à leur capacité d'être implémenter efficacement avec les différentes technologies récentes tels que les microprocesseurs, les BDDs (Binary Decision Diagrams), SAT (SATisfiability) solveurs et les SMT (Satisfiability Modulo Theories), SMT (Satisfiability Modulo Theories) solveurs, et d'autre part fournissent une manière automatique (i.e; push-button way) de vérification. L'avantage principal de model checking (L'exploration de l'espace des états) est de faire une exploration des états du système jusqu'à exploration de tout l'espace des

Table 3.1: Résumé des caractéristiques de différents model-checkers.

Model checker	Outils de vérif.	Langage	Spéc. de SETR.	Type ab-strait	Vérification
SPIN	LTL Promela	non formel	RT-Spin	Non	Logiciel de syst. distribués
NuSMV	LTL/CTL, BDD	formel	–	Non	Circuit (Hardware)
Maude	LTL,RL	formel	RT-Maude	Oui	Architecture distribuées, syst. de composants critiques
UPPAAL	Timed automata	formel	–	Non	SETR

états en entier, ou bien jusqu'à la détection d'un état d'erreur. De plus, ces modèles n'exigent pas une compréhention profonde des concepts mathématiques.

Le tableau 3.1 représente un résumé de caractéristiques de différents model checkers les plus utilisés.

Parmi les avantages de la technique de model checking est qu'elle est automatique, et que généralement un contre-exemple est retourné dans le cas ou la propriété n'est pas vérifiée. Du point de vue théorique, la limitaion majeure du model checking est que le système de transition doit être fini, i.e; le programme doit gérer que des variables à domaine fini. Du point de vu pratique, la limitation principale de cette technique est la grande taille du système de transitions due au *phénomène d'explosion combinatoire* du nombre d'état du comportement du système dû à l'entrelacement du comportement des différents composants du système. Le problème d'explosion combinatoire évolue exponentiellement et il est principalement causé d'une part par l'augmentation du nombre de variables ainsi que leurs tailles, et d'autre part, par le nombre de composants du système (le cas de composants concurrents).

3.3 De la théorie à la pratique

Les recherches avec le model checking continuent, afin d'augmenter leur efficacités. Dans ce contexte on peut par exemple citer : le *Bounded model checking* [BCC$^+$], le *model checking modulaire* ainsi que le *raffinement automatique d'abstractions*. De plus, plusieurs travaux de recherche sont aller encore plus loin que le model cheking fini, tels que les systèmes probabilistes et temporisés.

3.3.1 Domaines d'application

D'un point de vue académique, la technique de model checking a contribué à la génération de tests ainsi qu'à la démonstration automatique. Cependant, les différents techniques du procesus de model checking s'impliquent notamment dans d'autres domaines, i.e; le raffinement automatique d'abstractions par contre-exemple (analyse statique et la logique temporelle : *model base testing, run-time verification*).

La technique du model checking est généralement appliquée à des systèmes finis, d'une manière oû on peut facilement leur représentés par une abstraction finie. Parmi les domaines d'applications est la validation de protocoles d'applications, les services Web, ainsi que le tests de composants éléctroniques des systèmes de nature finie.

3.3.2 Le model checking dans le cycle de développment

Afin de détecter les erreurs au plus tôt possible, il est nécéssaire d'appliquer une exploration avant l'étape d'implantation. Pour cela, le processus de model checking prend place au niveau de la phase de conception. Afin d'appliquer la vérification par model checking, on a besoin que le système et ses spécifications soient représentés dans un modèle formel.

Le tableau 3.2 représente une comparaison subjective entre les différentes méthodes de vérification et la technique de model checking [Bar08]. La technique de vérification par l'outil OBP est détaillée dans la partie 7.

3.4 La vérification de propriétés par exploration de modèle -L'outil OBP

Pour établir la vérification d'un ensemble d'exigences sur un modèle, il faut disposer d'un modèle simulable et des exigences formalisées sous la forme, par exemple, de formules logiques ou d'automates observateurs (Figure 3.1). Le modèle simulable, les exigences et les interactions avec l'environnement (le contexte) constituent les données pertinentes et suffisantes pour pouvoir mener les vérifications. Pour pouvoir exécuter une exploration du modèle, il faut intégrer dans le modèle le comportement de l'environnement.

Ce modèle est ensuite simulé et exploré par un outil de vérification. L'exploration génère un *Système de Transitions (SdT)*. Celui-ci représente tous les comportements du modèle dans son environnement sous la forme d'un graphe de configurations et de transitions. Sur ce SdT, une vérification des propriétés peut être conduite, soit en appliquant des algorithmes de model-checking sur les formules logiques, soit en appliquant une analyse d'accessibilité des états d'erreur des observateurs. La difficulté liée à cette technique est la production du SdT qui peut être de grande taille, dépassant la taille mémoire disponible (explosion combinatoire).

Table 3.2: Comparaison subjective entre les différentes méthodes de vérification et la technique de model checking.

–	phase du cycle	prise en main	assisté par ordi.	surcoût	debug	validation
preuve	conception	–	-	preuves	-	++
model checking	conception	+	+	modélisation concrétisation	++	+
tests autom. (modèle)	conception	+	++	modélisation concrétisation	+	-
analyse statique	code	++	++	faux negatifs stub	-	+
tests autom. (code)	code	++	++	stubs	+	-
tests autom. (code+assert)	code	++	+	assertions stubs	++	-
tests standard	code	++	-	stubs jeux de tests	-	–

3.5 Exploration du modèle avec OBP

Pour explorer le modèle, nous utilisons deux outils :

- Un premier outil est le model-checker TINA-SELT [BBV04] (Figure 3.2). Celui-ci est particulièrement bien adapté à la vérification d'exigences exprimant des invariants formalisées en logique temporelle. Il intègre des algorithmes de model-checking sur les formules logiques au format SELT. Pour utiliser TINA, nous choisissons de décrire les comportements des acteurs, déduits des scénarios, sous la forme d'automates au format Fiacre [FGP+08]. Le modèle d'entré est également spécifié par un réseau d'automates Fiacre. L'exploration de l'ensemble de ce modèle génère un SdT, qui représente tous les comportements du modèle dans son environnement, sous la forme d'un graphe de configurations et de transitions.

- Le second est OBP Explorer [DBR11], un explorateur de modèle couplé à un analyseur d'accessibilité (Figure 3.3). Celui-ci est plus adapté à la vérification de propriétés d'accessibilité qui s'expriment à l'aide d'observateurs.

  Ces deux outils, Tina et OBP Explorer, sont connectés en amont à un troisième outil

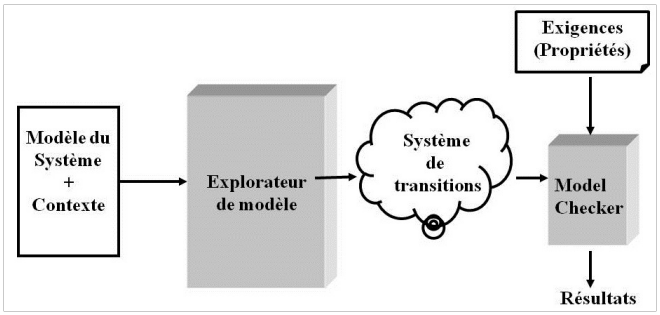

Figure 3.1: Vérification de propriétés par exploration de modèle.

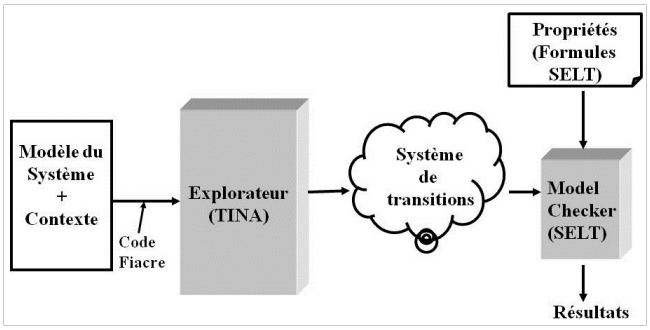

Figure 3.2: Outil de vérification expérimenté TINA-SELT.

OBP [1]. Nous utilisons les deux outils [2] pour la vérification de deux types d'exigences associées à l'étude de cas illustré au chapître 7: un invariant exprimé en logique temporelle, vérifié avec TINA, et un observateur exprimé dans le langage CDL que nous présenterons à la section suivante et vérifié avec OBP Explorer.

3.6 Le Langage CDL

Ce DSL [3] s'inspire des Use Case Chart de [Whi06] basé sur des diagrammes d'activités et de séquences. Nous étendons ce formalisme pour permettre de décrire les entités (nommées acteurs) contribuant à l'environnement et pouvant s'exécuter en parallèle. Doté d'une syntaxe textuelle [4], un modèle CDL décrit, d'une part, des scénarios à l'aide de diagrammes

[1] D'un point pratique, OBP intègre l'outil OBP Explorer

[2] Dans les deux cas, les vérifications sont effectuées sur une machine de type PC, possédant 3 giga-octets de mémoire.

[3] Domain Specific Language.

[4] La documentation complète du langage CDL est accessible sur http://www.obpcdl.org

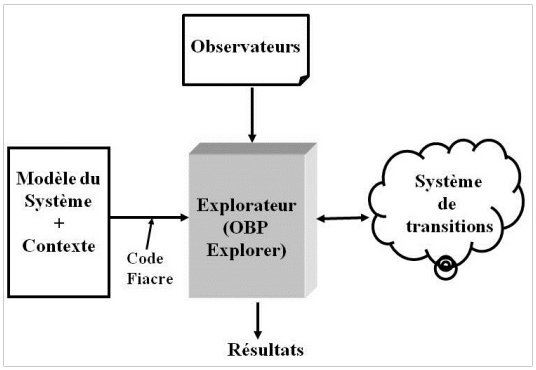

Figure 3.3: Outil de vérification expérimenté OBP Explorer.

d'activités et de séquences. Le comportement de l'environnement est considéré comme des enchainements de scénario qui décrivent les interactions entre le modèle soumis à la simulation et des entités (acteurs) composant l'environnement de ce modèle. D'autre part, un modèle CDL décrit également des propriétés à vérifier. Un méta-modèle de CDL a été défini et une sémantique décrite en terme de traces [DBR11], s'inspirant des travaux de [Whi06]. Un programme CDL est structuré en plusieurs parties déclaratives :

- déclarations des évènements (event_declaration)
- déclarations des activités (activity_declaration)
- déclarations des predicats (predicate_declaration)
- déclarations des propriétés (property_declaration)
- déclarations des contextes (cdl_declaration)

Lors de la compilation d'un modèle CDL, les diagrammes correspondant à chaque acteur sont dépliés (mise à plat de chaque boucle finie) puis entrelacés entre eux (conformément à la sémantique asynchrone de l'opérateur de parallélisme de langages tels que LOTOS, FIACRE, etc.). L'entrelacement de l'ensemble des MSC, décrivant le comportement du contexte, génère un graphe représentant toutes les exécutions des acteurs de l'environnement considéré. Ce graphe est ensuite partitionné, de manière à générer un ensemble de sous-graphes correspondant aux sous-contextes. Lors de l'exploration par le vérificateur, chaque sous-graphe est composé avec le modèle à valider et les propriétés sont vérifiées sur le résultat de cette composition.

3.7 Discussion et Conclusion

Un model checker est vu comme un testeur automatique, qui a la capacité de vérifier tous les comportements possibles du système pour des chemins d'exécution bien fixés. Cette technique de vérification a fait une percée remarqué dans les champs d'application industrielles. Citant quelques produits commerciaux : Bull, IBM, Microsoft et Siemens. On peut par exemple citer le langage PSL qui a été standarisé au niveau industriel dans le but de spécifier des propriétés temporelles (mis au point par un consortium entre Intel et IBM). Cette technique est également fréquèment utilisée dans le domaine de systèmes critiques, i.e; Bosch et Airbus dans une approche de *Model Driven Developping (MDD)*.

Dans ce qui suit, nous exposons un aperçu des travaux concernant les transformations des modèles, l'intégration des modèles semi-formels dans le processus de validation. En outre, nous discutons les approches facilitants la spécification des transformations (mapping) à l'aide des régles de transformation, dans le but de faciliter l'utilisation de ces modèles pour la vérification formelle. Nous citons les différentes approches de transformation de modèles semi-formels en langages formels. Et enfin, les travaux de transformations de spécifications CCSL pour la vérification formelles avec le processus de model-checking.

Chapitre 4

Travaux Connexes

"I haven't failed I've just found 10,000 ways that won't work."

Thomas Edison

4.1 Techniques et Langages de modèles de transformation: outils et standards

Nous introduisons dans cette section les différentes techniques et langages de transformation de modèles.

Dans un premier temps, nous introduisons deux méthodes de transformation (par les graphes de grammaire et par la représentation XMI). Nous citons quelques techniques et langages de transformation de modèles, ainsi que les outils standarisés dans ce domaine.

4.1.1 Transformation par les TGG (Triple Graph Grammars)

Triple Graph Grammars, est une technique utilisée pour définir la relation entre deux types de modèles. Et par la suite, transformer un modèle d'un type à un autre, afin de calculer la correspondance entre deux modèles existants. Ainsi, pour le but de maintenir la cohérence entre deux types de modèles, tel qu'il est définit en TGG. Lorsqu'un des modèles est changé, le modèle cible peut être modifié, ce qui signifie que la transformation ou la synchronisation peut être appliquée progressivement [KW07a] [Kon05].

Les TGGS nécéssitent des concepts additionnels avancés, qui sont primordiales pour l'utilisation de la transformation de modèles en pratique. Par exemple, lorsqu'on a besoin de transformer des attributs de noeuds (resp. objects), et, pour quelques exemples de transformation, on aura besoin des régles de TGGs (*TGG-rules* avec des contraintes additionnelles "*additional constraints*". Comme par exemple la non existence de quelques objects, appelés parfois les noeuds négatifs.

L'avantage principal des TGG est qu'ils permettent de définir des transformations d'une façon déclarative, avec une exécution dans les deux directions de la transformation. Cette définition de relation entre les différents modèles a une bonne implication. Dans le sens ou, premièrement, les régles de TGGs " *TGG-rules*" peuvent être rendues opérationnelles dans les scénarios d'application (modèle de tranformation, modèle d'intégration et modèle de synchronisation). Deuxièment, le processus de transformation est possible dans les deux directions, (i.e., la définition est donc bi-directionnelle). En effet, les régles de TGGs participent à l'intégration du modèle et au scénario de sa synchronisation [KW07b]. De plus, la définition locale de la relation entre les deux modèles est fortement similaire au style de la *Sémantique Opérationnelle Structurelles (SOS)* [Plo04], ce qui permet de vérifier la correction sémantique des relations entre ces modèles.

Parmi les incovénients de la méthode de transformation avec les TGGs, est que toute les contraintes sont ajoutées au moment de l'application des régles et pendant la construction des modèles. Par contre, ces contraintes ne peuvent être évaluées seulement qu'aprés la construction du modèle complet.

EMF Henshin [ABJ+10] est considéré comme la continuation du langage de transformation EMF Tiger [BEK+06]. C'est un langage de modèle de transformation qui utilise *Triple graph grammars (TGG)*, basé sur le Framework *Eclipse Modeling Framework EMF* [SBP+09].

La description de transformation est un modèle de transformation qui se base sur le mécanisme de *left-hand-side graph*, un *right-hand-side* et une liste de correspondance (Mapping).

Les noeuds du graphe sont des instances des éléments du modèle de méta-model source et le méta-model cible, respectivement. Il est ainsi possible de créer des transformations d'ordre-élevé (higher -order) avec EMF Henshin. Par contre, il n'existe pas de support (no built-in support) pour créer les traces, pour le modèle de transformation incrémental, ou même le multiple-directionel.

4.1.2 Transformation par méta-modèle : utilisant la représentation XML/ XMI

Parmi les approches bien connues pour la transformation des modèles est la technique de XSLT [W3C99], utilisée pour la transformation des modèles représentés par des documents XML. En effet, un modèle doit être exporté dans cette représentation et par la suite transformé en une seule étape. Le changement de propagation incrémentielle n'est donc pas pris en charge. De plus, le modèle de transformation doit étre exprimé en utilisant les concepts offerts par XML et XSTL. Par ce fait, le modèle de transformation doit étre assez expressif, tout en utilisant les concepts offerts par le XMI et le XSLT, qui est difficile à lire.

ATL Le langage de transformation ATLAS [JABK08] (ATL) est un langage de transformation hybride de type modèle-à-modèle (model-to-model).

ATL supporte les constructions déclaratives et impératives. Le style préféré est celui de constructions déclaratives, qui permet de simplifier l 'implémentatin du mapping. Par contre, les constructions impératives fournissent des mapping plus au moins complexes à gérer déclarativement peuvent plus au moins être spécifiés. Un programme de transformation ATL est composé d'un ensemble de régles qui servent à décrire comment créer et initialiser les élèments du modèle cible. Le langage est spécifié à la fois comme un méta-modèle et une syntaxe textuelle concréte. ATL est intégré sur l'environnement de développement Eclipse et peut gérer les modèles basés sur EMF. Ce langage fournit ainsi un support pour les modèles en utilisant EMF basé sur les profils UML .

Query/View/Transformation (QVT) Query/View/Transformation (QVT) est un langage de standarisation pour les modèles de transformation [1] établie par le groupe (OMG) . QVT utilise le langage (OCL) Object Constraint Language [2], Meta Object Facility (MOF)

[1] MOF 2.0 Query,View,Transformation, QVT a: http://www.omg.org/spec/QVT
[2] OCL: http://www.omg.org/spec/OCL/2.2

[3] et il est aligné avec la méthode MDA (Model Driven Architecture) [4].

QVT définit trois langages pour la transformation model-to-model. Il définit ainsi une syntaxe concrète et un méta-modèle basé sur XMI pour la création de la représentation de la transformation QVT. QVT a le mécanisme de boite noire qui permet d'appeler un code externe dans lequel figure la transformation. Pour l'outil d'implémentation de chacun des trois langages, QVT définit quatres classes conformes pour l'interopérabilité qui sont : une syntaxe exécutable (la capacité d'exécuter le QVT dans une syntaxe concrête), un XMI exécutable (capacité d'exécuter un QVT dans un modèle XMI serialisable, une syntaxe exportable, i.e; la capacité d'exporter le QVT dans une syntaxe concrète). Les outils de transformation de modèles qui sont conformes au standard QVT incluent par exemple le langage de transformation SmartQVT (cf. 4.1.2).

SmartQVT SmartQVT [5] est une implémentation d'un outil de transformation pour un QVT opérationel. C'est un langage impératif pour la transformation model-to-model pour le EMF basé sur les modèles. La description de transformation est compilée en code Java et supporte le mécanisme de boite noire de QVT pour appeler un code externe.

Ce type de transformation offre un support de traçage. En outre, il offre des réflexions d'accès aux informations de traçage, telles que l'objet ciblé correspondant à un objet source ou l'objet source correspondant à un objet cible. Il contient un support pour contrôler les paramètres ainsi que des règles d'ordre supérieur.

OpenArchitecureWare (OAW) OAW [SEV07] intègre un nombre d'outils pour la transformation de modèle en un framework cohérent. OAW fournit un langage de spécificat-ion (appelé : *ow specification*), et un langage de transformation (appelé Xpand). Le langage "ow" est utilisé pour contrôler le processus de transformation et de spécifier la séquence de transformation entre les différents modèles. Le langage de transformation Xpand est basé sur un template, un langage impératif pour la transformation model-to-model. OAW est distribué comme un plugin dans la plateforme Eclipse et il est capable de gérer les modèles EMF (Eclipse Modeling Framework).

Kermeta Kermeta [rFHN] [TMK] est un langage de modélisation impératif, permettant d'effectuer la transformation. Ce langage offre un métamodeling basé sur EMF, les contraintes, la vérification, la transformation et le support de comportement. Les modèles et méta-modèles seront explicitement chargés et stockés. Les élèments cibles seront explicitement instantiés et ajoutés au modèle cible. Les régles de contrôle d'ordonnoncement de l'application doivent être explicitement décrites par l'utilisateur.

[3] MOF: http://www.omg.org/spec/MOF/2.0/
[4] MDA:http://www.omg.org/mda/
[5] http://smartqvt.elibel.tm.fr

ETL Le langage de transformation Epsilon "The Epsilon Transformation Language (ETL)" [KPP08] est un langage de transformation hybride (de type model-to-model). Ce langage peut gérer différentes sources et plusieurs modèles cibles. Il offre des fonctionnalités de régles d'ordonnoncement: Un code externe peut être exécuté à partir des régles de transformation.

XML Stylesheet Language Transformations (XSLT) XSLT [6] est un langage de transformation fonctionnel pour la manipulation de données XML, dont les régles doivent être appelées explicitement. Les régles sont strictement unidirectionelles sans aucun support de traçabilité.

Les descriptions de transformation avec XSLT sont eux même des documents XML, par la suite une transformation d'ordre-elevé peut être réalisée. Du fait que le XSLT est initialement développé pour transformer les documents XML en documents HTML, il est considéré limité par de simples transformations [7].

4.2 Discussion

Beaucoup d'autre langages de transformation de modèles existent, on peut citer quelques réferences: Moflon, mediniQVT, Textual Concrete Syntax (TCS), XText, Tefkat, MOLA, MT, SiTra, MofLog, GreAT, GenGen, Beanbag, UMT, UMLX, ATOM, VIATRA, BOTL, XDE Transformations, Codagen Architect Transformations, b+m Generator Framework, OptimaJ Transformations, ArcStyler Transformations, MPS Transformation, Microsoft DSL Tool Transformations, Metaedit+ Transformations, AndroMDA, JET, FUUT-je, GMT, Jamda, Fujaba Transformations, TXL, Stratego.

4.3 Approches de transformation des modèles pour la vérifi--cation formelle

4.3.1 Approches de transformation des modèles semi-formels

Plusieurs approches de transformation à partir d'UML MARTE vers des spécifications formelles ont été proposées. Une approche dirigée par la vérification (A verification-driven approach) [GPC12] consiste à traduire des diagrammes d'activités à des réseaux de Petri (*Time Petri Nets (TPN)*) dans le but de guarantir la correction des propriétés temporelles. Les authors utilisent une sémantique formelle des TPN afin d'éviter l'explosion d'espace des états du model checker lors de l'exploration. L'approche est limitée dans le sense ou elle prend en compte uniquement un type particulier de propriétés (i.e., synchronisation et ordonnonçabilité).

[6] XSLT: http://www.w3.org/TR/xslt
[7] Disponible: http://dx.doi.org/10.1016/S0306-4379(01)00033-3

[NGC12] présente un caneva (cadre) dédié à la vérification des propriétés temporelles pour les modèles MARTE, reliés à des transormations dirigées par les propriétés des architectures UML ainsi que le comportement des modèles à un modèle *TPN* executable et vérifiable. Ceci en transformant les propriétés temporelles à un ensemble de pattrons de propriétés bien spécifique correspondant à des observateurs TPN. Ce qui est insuffisant dans ce travail est que cette approche ne prend en considération qu'une partie d'analyse de propriétés temporelle dans le profil UML MARTE sans exploiter la notation formelle du langage CCSL.

Dans [RMGF12], une approche a été proposée pour implémenter les architectures dirigées par les modèles dans le domaine de logiciel d'espace temps-réel (real-time space software), basés sur fUML et les spécifications UML MARTE. Cependant, du fait que cette approche se repose sur fUML, il est difficile de modéliser et de contrôler le comportement des algorithmes par le biais des diagrammes d'activités. Une autre insuffisance de cette approche, est qu'elle ne prend pas en compte la transformation des contraintes temporelles, principalement les contraintes qui peuvent être annotées formellement en CCSL dans le cas de spécifications MARTE.

Sorel *et al.* [PFS08] propose une apporoche dirigée par le modèle (model-driven approach) basée sur MARTE pour la validation de comportement temps-réel du modéle fonctionnel vis à vis de contraintes temporelles ainsi que la plateforme d'exécution spécifique. Les characteristiques temporelles et les contraintes de temps de l'application sont extraintes d'un niveau d'abstraction élevé du modèle impliqué, et sont par la suite analysées en utilisant la technique d'analyse d'odonnonçabilité. L'approche ne prend pas en compte la vérification d'exigences fonctionnelles.

Une autre approche pour la formalisation et la validation d'exigences temporelles a été proposée dans [CRST11]. Un formalisme pour représenter et analyser des exigences en utilisant le model checker NuSMV a été proposé. Les contraintes de temps sont exprimées par le biais d'opérateurs temporelles, resultant dans un ensemble de fragments de logique temporelle du premier ordre. Le formalisme se repose sur des diagrammes de classes, et de fragments combinés de logique temporelle de premier ordre avec des opérateurs de temps afin de décrire l'évolution des scénarios. L'inconvénient de cette approche est que seulement une partie du scénario qui est considérée controllable.

4.3.2 Spécification de temps, formalismes : Extention de langage formel (Promela) avec du temps discret

Un type de variable temporelle est définit dans [BD] correspondant à du temps discret " countdown timer" (Une extention avec du temps discret pour Promela pour les systèmes concurrents). En effet, l'extention proposée sera difficile à utiliser pour exprimer la coinsidence entre les ticks d'horloges. Un autre travail dans [BD98], qui propose de modéliser les spécifications temporelles à partir de l'algébre ACP, par une définition sur le niveau d'abstraction de Promela. Cette approche présente une extention de Promela avec SPIN,

avec du temps discret, qui fournit l' opportunité de modéliser des systèmes que leurs correction fonctionnelle se repose et depends fortement sur l'exactitude des paramètres de temps.

4.3.3 CCSL, observateurs et simulation, vérification formelle des contraintes CCSL

"Les modèles créés sont ceux qui sont utiles pour s'assurer que le système est correctement construit (vérification)", "et qu'il satisfait un ensemble de propriétés spécifiées en amont dans le processus de développement (validation)" [Ren09].

De nombreux travaux ont été menés pour vérifier formellement les contraintes CCSL. Talpin *et al.* [YTB+11] propose de convertir CCSL à une spécification synchnone. Les opérateurs CCSL sont traduits à des systèmes polynomilals dynamiques. L'approche utilise le model checker Sigali [Dut92] afin de vérifier les contraintes temporelles. Néanmoins, l'approche met l'accent que sur l'implémentation des contraintes CCSL en un programme de Signal. Par contraste, notre approche fournie un support de caneva de vérification dans le cadre de l'ingénierie dirigée par les modèles (Model-Driven Engineering), tout en se concentrant sur la vérification des propriétés temporelles sur les modèles MARTE. De plus, notre approche permet ainsi de vérifier des propriétés fonctionnelles.

L'approche proposée présente une extension en réponse aux spécifications CCSL. Cependant, cette approche est trop restrictive, car elle se concentre uniquement sur la mise en œuvre des contraintes CCSL avec Signal. En contraste avec ce travail, nous proposons une approche plus générale de la traduction qui vérifie non seulement les contraintes CCSL, mais également des propriétés fonctionnelles.

Une approche est proposée dans [And10] pour la mise en oeuvre d'observateurs [HLR93] pour la vérification formelle des spécifications CCSL. Les observateurs, encodant les contraintes CCSL sont traduits en codes Esterel et une analyse d'atteignabilité permet de vérifier si un observateur atteint un état d'erreur.

L'environnement Times Square [DMA08], est dédié à la résolution de contraintes CCSL et au calcul de solutions informatiques, met en œuvre un générateur de code dans l'Estérel. L'outil permet en plus de détecter des inconsistences dans les spécifications CCSL en mettant en évidence des interbloquages dans la simulation. Il fournit à l'utilisateur des chronogrammes lui montrant des exécutions possibles pour faciliter la mise au point de ses spécifications.

En contraste avec ces travaux, nous proposons une approche plus générale de la traduction qui vérifie non seulement les contraintes CCSL, mais également des propriétés fonctionnelles. De plus, dans notre approche l'expression de propriétés sont séparées de l'application du modèle grâce à notre langage CDL, d'oû la séparation des préoccupations.

L'interprétation équivalente du noyau CCSL avec Signal [BGJ91b] et les réseaux de Petri temporels [Gol87] a été proposée dans [AM08]. Les auteurs font une comparaison des

contraintes CCSL avec les Réseaux de Petri et signaux. L'approche mathématique montre une définition pour chaque contrainte sélectionnée et sa mise en œuvre équivalente avec les Réseaux de Petri et le Signal. Cette approche diffère de notre approche car nous ciblons la langage Fiacre, basée sur des automates, à la place des Réseaux de Petri.

Dans le contexte de la modélisation et la conception les *Systèmes à puce* "System-On-chip", des travaux ont été proposés utilisant le langage CCSL comme un langage de specification de haut niveau, à partir duquel un réseau d'observation peut étre conçu. Ces réseaux d'observations sont utilisés pour observer des prototypes d'implémentations précoce du système sous conception, ainsi de vérifier sa conformité tout en respéctant les spécifications CCSL. L'approche [Mal12a] consiste à concevoir manuellement une librairie de noeuds d'observateurs pour chaque opérateur CCSL, ainsi de définir un mécanisme générique représentant ces noeuds. L'auteur propose une technique de génération complète d'observateurs à partir des spécifications CCSL. L'approche introduit une sémantique à base d'états (state-based sémantics) visant un sous ensemble des opérateurs CCSL.

Meme si le modèle de temps de MARTE offre un support assez riche pour décrire des horloges discrètes et denses. Effectivement, on constate que l'effort a été plus fait dans le but de spécifier et d'analyser des modèles discrets de MARTE. Afin de mettre en évidence les systemes embarqués et les systèmes hybrides temps-réel. L'approche dans [JL12] propose d'étendre les diagrammes d'états (statecharts) en utilisant MARTE, ainsi que la théorie d'automates hybrides. Cette proposition fournit une amélioration vis à vis des automates hybrides de façon que les variables du temps logique et du temps chronometrique soient unifiées. La syntaxe et la sémantique formelle des diagrammes d'états de MARTE hybrides proposées sont basées sur les systèmes de transition étiquetés et les systèmes de transition dynamiques.

Les auteurs de l'article [GMD12] contribuent à comparer l'expressivité de CCSL et du langage PSL. Ils identifient les constructions CCSL qui ne peuvent pas être exprimées dans la logique temporelle et proposent des restrictions de ces gestionnaires. L'article a également désigné la classe des formules PSL qui peuvent être encodées en CCSL et définit des traductions entre ces fragments et PSL en utilisant le formalisme des automates comme une représentation intermédiaire. Cependant, cette approche ne tient pas compte des constructions CCSL qui ne peuvent être exprimées en PSL.

[YM11] propose des transformations de spécifications CCSL en un modèle Promela afin de vérifier formellement les contraintes CCSL par le model-checker SPIN. Les propriétés à vérifier sont exprimées dans cette approche en logique LTL. Nous avons été inspirés par ce travail pour concevoir la traduction automatique de contraintes CCSL en automates Fiacre. En outre, dans cette approche, les propriétés à vérifier sont exprimées dans la logique LTL. Nous proposons d'exprimer des propriétés avec des automates observateurs, exprimés en langage CDL, qui permettent une meilleure expression. Par exemple, dans notre thèse, nous montrons une propriété (cf Figure 7.5) qui serait fastidieux à exprimer en LTL.

Le travail de [Mal12b], présente une technique de réutilisation du standard de l'environnement basé sur la conception VHDL et les langages synchrones (Esterel). L'article définit des sémantiques à base d'états pour quelque opérateurs CCSL basés sur les systèmes de transitions étiquetées.

Un autre travail a été présenté dans [RM13], traitant les opérateurs CCSL non bornés. Plus précisement, il définit une représentation basée sur les états des opératuers CCSL. Dans ce travail, une évaluation paresseuse (lazy evaluation) est utilisée pour représenter intentionnellement des systèmes de transition infinis, tout en fournissant un algorithme afin de maintenir une production de synchronisation pour chaque système, avec la prise en compte de leurs compléxité.

Un travail récent à été publié dans [SSMP13], présentant une technique de transformation de modèles comportementaux exprimés en MARTE-CCSL en un ensemble d'automates temporels par la méthode de model-checking utilisant l'outils UPPAAL, qui permet de vérifier des propriétés de temps logique et chronométrique du système.

4.4 Bilan et Positionnement

La plupart des travaux sur CCSL considèrent uniquement des spécifications temporelles et leurs analyses. Par contre, ce que nous proposons est une approche de transformation plus générale qui sert à vérifier non seulement des contraintes exprimées formellement en CCSL, mais aussi des propriétés fonctionnelles. De plus, dans notre approche ces propriétés sont séparées du modèle applicatif grâce au langage CDL, d'ou la séparation de préoccupations. Ces propriétés, dans le cas ou elles sont liées au temps ou bien au model fonctionnel, elles sont exprimées en langage CDL [DR11a]. Par ailleurs, la notion de contexte utilisée, exprimée ainsi en CDL, permet de réduire l'explosion combinatoire durant l'exploration du modèle dans certains cas.

Nous nous inspirons du travail publié dans [YM11] pour la conception de la transformation des contraintes CCSL en automates Fiacre. Par contre, nous préferrons exprimer les propriétés à vérifier par des observateurs que nous considérons plus aisés à rédiger. Ces propriétés sont exprimées en langage CDL [DR11b] et concernent non seulement les contraintes CCSL mais aussi les exigences de la partie fonctionnelle du modèle à valider. De plus, nous exploitons la notion de contextes, exprimés également en CDL, qui permet, dans certains cas, de réduire l'explosion combinatoire lors de l'exploration des modèles. Nous présentons en détails nos contributions dans la partie suivante.

Part III

Contribution

Chapitre 5

Approche pour la validation de modèles temporisés

" Il y a dans la création des Cieux et de la Terre et dans la succession de la Nuit et du Jour, des signes pour ceux qui sont doués d'intelligence "

Coran, Sourate 3 - Verset 190

5.1 Introduction

Nous nous intéressons aux problématiques liées au développement des systèmes réactifs. Dans ce domaine, les architectures logicielles doivent être conçues pour assurer des fonctions critiques soumises à des contraintes très fortes en termes de fiabilités et de performance temps réel. Les exigences à respecter lors de la modélisation des logiciels concernent non seulement la correction et le déterminisme sur le plan functionnel mais aussi sur le plan temporel. Aussi, dans le processus de développement des systèmes, les concepteurs recherchent des méthodes et des langages leurs permettant de décrire leurs architectures, tout au long du cycle et à différents niveaux d'abstraction.

Actuellement, les industries engagent tous leurs efforts dans les processus de test et de simulation à des fins de certification. Néanmoins, si les tests se révèlent souvent efficaces pour débusquer les erreurs, ils ne permettent généralement pas de démontrer exhaustivement l'absence d'erreurs. Il est alors nécessaire d'utiliser d'autres méthodes pour garantir la fiabilité des logiciels. Parmi celle-ci, les méthodes formelles ont contribué, depuis plusieurs années, à l'apport de solutions rigoureuses et puissantes pour aider les concepteurs à produire des systèmes non défaillants. Dans ce domaine, les techniques de model-checking (cf Figure 5.1) [QS82] [CES86] ont été fortement popularisées grâce à leur faculté de d'executer automatiquement des preuves de propriétés sur des modèles logiciels. De nombreux outils *model-checkers* ont été développés dans ce but: SPIN [1], UPPAAL [2], TINA [3], CADP [4] et plus récemment OBP [5](Observer-Based Prover) développé à l'Ensta Bretagne. Malgré la performance croissante de ces outils, leur utilisation reste difficile en contexte industriel. Leur intégration dans un processus d'ingénierie industriel est encore trop faible comparativement à l'énorme besoin de fiabilité dans les systèmes critiques. Cette contradiction trouve en partie ses causes dans la difficulté réelle de mettre en oeuvre des concepts théoriques dans un contexte industriel.

Parmi les difficultés rencontrées lors de l'utilisation des outils model-checkers, nous pouvons citer le problème bien identifié de l'explosion combinatoire du nombre de comportements des modèles, induite par la compléxité interne des logiciels qui doivent être vérifier. Cela est particulièrement vrai dans le cas des systèmes embarqués temps réel, qui interagissent avec des environnements impliquant un grand nombre d'entités. Une autre difficulté est liée à l'expression formelle des propriétés nécessaires à leur vérification. Traditionnellement, cette expression s'effectue à l'aide de formalismes tels que les logiques temporelles. Bien que ces logiques aient une grande expréssivité, elles ne sont pas faciles à

[1] http://spinroot.com
[2] http://www.uppal.com
[3] http://laas.fr/tina
[4] http://www.inrialpes.fr/vasy/cadp
[5] http://www.obpcdl.org

utiliser par des ingénieurs dans des contextes industriels.

Ensuite, les modèles spécifiés par les utilisateurs sont d'un niveau trés différent en terme d'abstraction des modèles manipulés par les outils de preuves de type model-checking. En effet, ces derniers manipulent des systèmes de transition, temporié ou non, alors que les utilisateurs expriment leur spécification dans des formalismes manipulant des abstractions de types processus, canaux de communications, données structurées, algorithmes de calcul, etc. C'est le cas par exemple d'UML et des différents profils qui ont été proposés dans la communauté académique et industrielle. Aussi, lors du développement des systèmes, les concepteurs modélisent leurs architectures et logiciels à différents niveaux d'abstraction. La modélisation implique de disposer d'expressivité riche pour prendre en compte les spécificités des applications en terme fonctionnel et temporel. Des transformations sont donc nécessaires pour générer automatiquement, à partir de modèles utilisateurs, des codes formels exploitables par les outils de vérification. Ces transformations posent alors des problèmes d'équivalence sémantique entre ces différentes expressions.

Toutes ces difficultés font qu'aujourd'hui, il n'existe pas encore de résultats de travaux et d'outils satisfaisants à proposer aux industriels pour qu'ils puissent mener des preuves d'exigence dans le cadre de leur processus de développments industriels. Ces donc à ces défis que veux répondre cette thèse dans le cadre de l'utilisation de modèles UML-MARTE.

La modélisation des logiciels des systèmes temps réel impliquent de référencer explicitement un ensemble d'horloges lors de la construction des modèles. Par exemple, un modèle de temps logique a été proposé à l'OMG comme une part du profile UML MARTE [OMG10] pour enrichir ce formalisme. Cette intégration vise à permettre à l'utilisateur non seulement de décrire ses logiciels mais aussi d'analyser les contraintes temporelles. Ce modèle de temps est associé au langage de spécification de contraintes d'horloge CCSL (Clock Constraint Specification Language)[And09]. Une fois le logiciel modélisé, la vérification formelle propriétés nécessite de les exprimer formellement. Elle nécessite aussi de transformer le modèle MARTE dans un modèle formel compatible avec le vérificateur (model-checker) mis en jeu. Durant la transformation, les contraintes CCSL viennent enrichir ce modèle formel.

Nous présentons dans ce chapitre, une application d'une technique [MD13] de vérification de propriétés de ce type de modèle avec l'outillage OBP et le langage CDL. Ces vérifications impliquent des transformations automatiques des contraintes CCSL. La technique s'appuie sur une traduction des modèles MARTE et des contraintes CCSL en code Fiacre. CDL permet d'exprimer des prédicats et des observateurs. Ceux-ci sont vérifiés lors de l'exploration exhaustif du modèle complet par OBP.

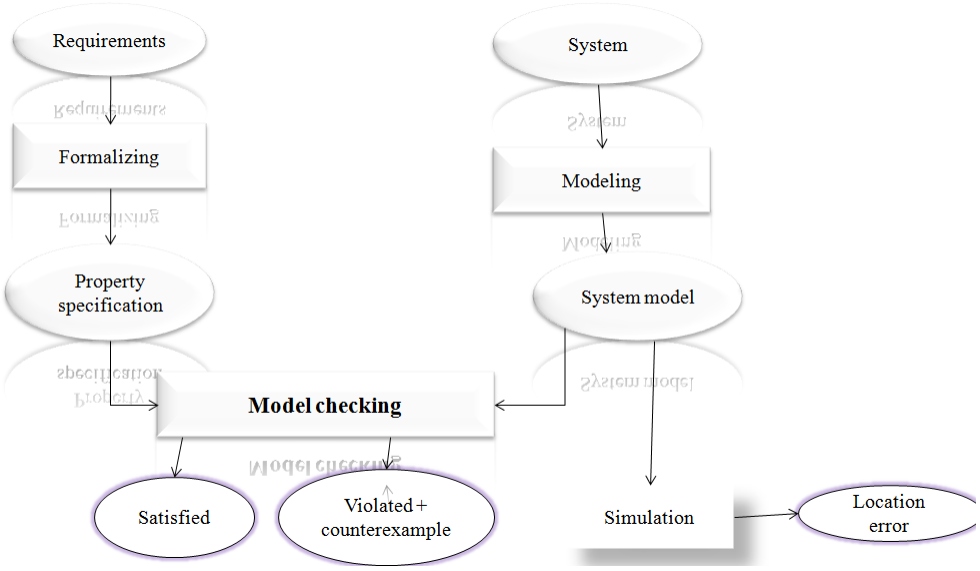

Figure 5.1: Shéma globale du principe de model checking.

5.2 Approche pour la validation de propriétés temporelles

L'approche est basée sur une modélisation multi-langages permettant la modélisation des différents aspects d'une application et la transformation vers les outils de vérification (Figure 5.2) : Modélisation de l'architecture, de la communication, du comportement, des données, des contraintes temporelles, des propriétés fonctionnelles et des observateurs, et des contextes.

Lors de passage d'un modèle semi formel (e.g UML MARTE) à un modèle Fiacre, les deux problèmes principaux à résoudre sont :

- La large fossé sémantique entre MARTE et Fiacre, avec en amont (MARTE vers Fiacre) des objets actifs et en aval des automates et composants .
- La variabilité des outils d'édition des modèles UML, qui n'implante pas tous les mêmes formats de séréalisation et méta-modèles.

Ainsi qu'évoqué précédemment, nous nous interéssant à la modélisation de SETR en MARTE afin de vérifier des propriétés sur nos modèles. Puisque le champs d'application de MARTE est plus large que le domaine qui nous intrésse, nous n'avons pas besoin de tous les élements du langage. De plus, notre objectif nécéssite une sémantique précise qui permet d'interpréter de manière non ambigue nos modèles. Or, le standard UML définit plusieurs sémantiques possibles [CD07][FSKdR05].

Le concept de modélisation abstraite des horloges logiques a été mis en place avec le langage CCSL (*Clock Constraint Specification Language*) [And09] au sein de MARTE

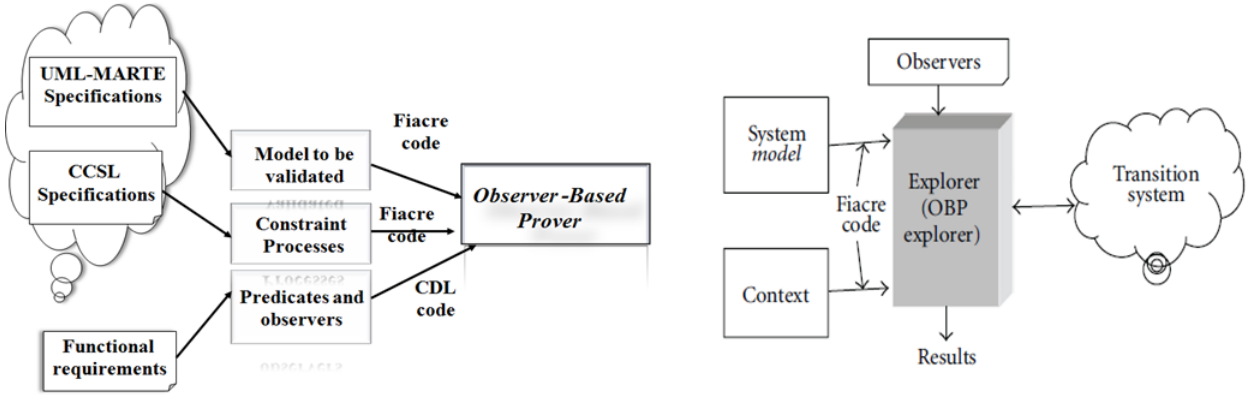

Figure 5.2: La vérification avec l'outils OBP Explorer.

[MAS08] et adopté par l'OMG [OMG10]. CCSL est un langage qui permet de définir les relations causales, chronologique et temporelle. Doté d'une sémantique formelle, il vise à compléter les formalismes de conception existants pour la spécification de modèles peuvant être analysées. CCSL vient enrichir la palettte de langages de conception et compenser le manque de formalisation constaté avec les notations semi-formelles telles que UML. Nous montrons dans ce travail, une approche où nous cherchons à vérifier des propriétés à la fois fonctionnelles et temporelles des programmes en combinant les contraintes CCSL avec l'architecture logicielle modélisée.

L'objectif de ce travail est une contribution pour une meilleure intégration des techniques de vérification formelle d'exigences comportementales sur des modèles logiciels. L'idée sous-jacente à ce travail est de contribuer à outiller l'activité de validation formelle de modèles logiciels, que nous nommons *modèles applicatifs* dans le but de renforcer la correction des logiciels.

Ces travaux s'appuient sur le formalisme de modélisation UML-MARTE [OMG10] et des transformations de modèles permettant de générer des programmes formels pouvant être exploités par des vérificateurs de propriétés.

Nous proposons d'une part, de définir un sous-ensemble d'UML-MARTE et une sémantique formelle pour lequel nous avons développé des transformations vers le langage Fiacre [FGP+08]. La correction des transformations est assurée par la vérification de propriétés nécessaires et suffisantes sur les programmes Fiacre générés. Celles-ci sont exprimées dans le langage CDL (*Context Description Language*) [DBR11, DPC+09] et vérifiées par l'outil OBP (*Observer-Based Prover*) [DBR+12].

D'autre part, nous prenons en compte dans les modèles UML-MARTE la possibilité de modéliser des contraintes temporelles adaptées aux logiciels temps réels que nous voulons

cibler dans le domaine des logiciels embarqués. Cette modélisation temporisée s'appuie sur le langage CCSL [And09] et des transformations de ces contraintes vers les modèles Fiacre.

Nos contributions sont les suivantes: (1) nous générons un programme Fiacre à partir d'un modèle UML MARTE; (2) nous exploitons des spécifications CCSL qui enrichissent ce modèle en les traduisant en processus Fiacre correspondant aux contraintes CCSL, en s'inspirant de l'approche décrite dans [YM11]. Nous décrivons les principes de la génération de code Fiacre pour ces contraintes; (3) nous montrons comment spécifier des propriétés sous la forme d'automates observateur. Pour cela nous les formalisons en langage CDL et utilisons l'outil OBP (Figure 5.2 pour la vérification basées sur une exploration de l'ensemble du code Fiacre généré; (4) nous illustrons notre contribution avec un exemple et décrivons les résultats des preuves menées.

Pour cela, notre approche consiste à une chaine de transformation (tool-chain) qui utilise trois outils existants : Fiacre, CDL, et OBP. Apres avoir motivé nos choix de ces outils, nous présentons CCSL et nous présentons dans ce qui suits des définitions brèves de chaque outils.

5.2.1 Définitions préliminaires

Fiacre. Fiacre est un langage pour utilisé pour définir des représentation intermédiaire des SETR, à base de machines d'états, dédié à être utiliser comme un "input" pour la vérification formelle et des besoins de simulation. Le choix du langauge Fiacre dans notre approche est se justifie par (1) le besoin de decrire formellement les SETR (2) le besoin de rendre les modèles (dans les sense de l'ingénierie des modèles) de ces systèmes vérifiable.

Context Description Language (CDL). CDL est un langage dédié à exprimer formellement les exigences sur le comportement du système. Ces exigences représentent les scénarios appropriés, dans lequel un système doit executé

De cette manière, CDL contribue à réduire le fossé entre les exigences de l'utilisateur et le modèle formel du système. De plus, ce langage sert dans notre approche à réduire l'espace des états du comportement du sytème.

Observer-based Prover (OBP) OBP [6] est model-checker qui permet d'explorer l'execu--tion des états du système. Il prend CDL et la spécification Fiacre comme entrées. Nous utilisons l'outils OBP afin de conduire notre exprérimentations et vérifier le système. (initiallement décrit en MARTE) vis-à-vis les contraintes temporelles (en CCSL), ainsi l'expression formelle des exigences (en CDL) de l'utilisateurs sur le système.

[6] http://www.obpcdl.org

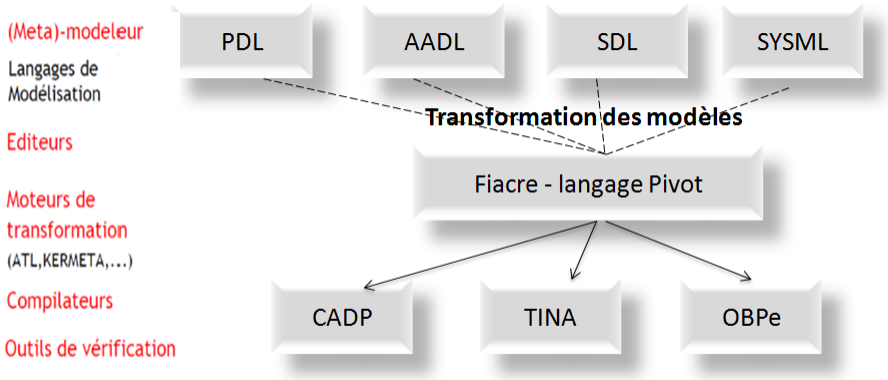

Figure 5.3: Fiacre comme un langage ciblé de transformation.

5.2.2 Le Langage Fiacre

Fiacre [FGP+08] (*Intermediate Format for the Architectures of Distributed Embedded Components*) [FGP+08] a été développé dans le cadre du projet TOPCASED[7] comme langage pivot reliant des formalismes de haut niveau comme UML, AADL, SDL et les outils d'analyse formelle. L'utilisation d'un langage formel intermédiaire apporte l'avantage de réduire l'écart sémantique entre les formalismes de haut niveau et les descriptions manipulées en interne par les outils de vérification comme les réseaux de Petri, les algèbres de processus ou les automates temporisés. Fiacre peut être considéré comme un langage disposant d'une sémantique formelle et servant de langage d'entrée à différents outils de vérification.

Ce langage s'inspire des nombreux travaux de recherche, à savoir V-Cotre [BBL03], NTIF [GL02], ainsi que les décennies de recherche sur la théorie de la concurrence et la théorie du système en temps réel. Fiacre est conçu à la fois comme le langage ciblé de transformation des modèles à partir de différents modèles tels que PDL, AADL, DSL, ou UML, et que le langage ciblé Fiacre pourra exploité dans les boîtes à outils de vérification, à savoir CADP [FGK+96],TINA [BRV04],et OBP Explorer (cf Figure 5.3). On n'a pas exploité les syntaxes et les sémantiques du langage Fiacre dans cette étude bibliographique, pour en savoir plus détail vous pouvez consulter [FGP+08].

Fiacre est un langage avec une sémantique formelle qui sert de langage d'entrée pour différents outils de vérification. C'est un langage formel de spécification pour décrire les aspects comportementaux et temporisés des systèmes temps réels. Il intègre les notions de *processus* et de *composants* :

- Les automates (*processus*) sont décrits par un ensemble d'états, une liste de transitions entre ces états avec des constructions classiques (affectations des variables,

[7] http://www.topcased.org

if-elsif-else, while, compositions séquentielles), des constructions non déterministes et des communications par ports et par variables partagées.

- Les composants (*component*) décrivent les compositions de processus. Un système est construit comme une composition parallèle (clause *par* la || opérateur) de composants et/ou processus qui peuvent communiquer entre eux par des ports ou variables partagées. Les priorités et les contraintes de temps sont associées aux opérations de communication.

Dans ce langage trois types de communication entre processus existent:

- Synchronisation avec ports: Un port peut être utilisé comme une variable bloquante, à l'intérieur d'une transition de deux ou plusieurs processus. Dans le but de contrôler (sychroniser) l'évolution de l'ordre de leur exécution.
- Variables partagées: les processus peuvent communiquer d'une façon asynchrone, en utilisant des variables partagées (pour la lecture /écriture).
- Buffers: Plutôt que d'utiliser une seule variable, un "buffer" peut être utilisé pour la communication asynchrone, comme une file d'attente d'une taille ajustable, contenant des données du même type. Avec la possibilité de vérifier l'état du buffer plein/vide.

L'expression du temps dans le langage Fiacre est basée sur la sémantique des Systèmes de Transitions Temporisés (TTS) [HMP91]. Toute transition est associée à des contraintes de temps (temps minimum et maximum). Ces contraintes assurent que les transitions sont tirables dans des intervales de temps bien définis (ni trop tôt, ni trop tard).

Pour une description detaillée, le lecteur peut se référer à la documentation Fiacre [8].

5.2.3 Le temps logique en MARTE

Dans un modèle MARTE, un évènement quelconque (e.g; une action de communication, émission ou réception, un début d'un calcul) peut servir à définir une base de temps. Cette dernière peut être considérée comme une horloge logique (*clock*). Une horloge représente un ensemble d'occurences d'évènements discrets, appelés *instants*. Ces instants sont strictement ordonnés et constituent un référentiel temporel. MARTE se base alors sur des instants qui sont des occurences d'évènements.

CCSL [And09], introduit comme une annexe de MARTE est considéré comme un langage déclaratif permettant de spécifier des relations binaires entre évènements logiques basées sur la notion d'horloge. CCSL est fondé sur un modèle mathématique donnant une sémantique formelle au temps, servant comme un pivot, apportant l'interopérabilité entre les langages existants, dans le but d'exprimer des phénomèmenes importants presentés dans de divers formalismes.

[8] http ://projects.laas.fr/fiacre/papers.php

Une spécification CCSL exprime formellement des relations basées sur un ordre partiel. Chaque occurrence de l'evenement est présentée par un *tick* sur l'horloge correspondante. La distance entre deux *instants* est mesurée en terme de *ticks*, ce qui est rien avoir avec la durée physique.

Le langage CCSL sert à bâtir un modèle de causalité temporelle dans les modèles MARTE. Ce langage founit une manière de spécifier des relations entre évènements dans lequelles l'ensemble d'horloges est vu comme des evènements "*events*", et les instants comme des occurences d'évènements "*event occurrences*". Cependant, ce langage peut exprimer une variété d'évènement supportant du temps, menants à la notion de temps mutiforme.

La logique temporelle est interprétée dans MARTE comme une structure de temps. Tout les opérateurs présentés sont définits avec une sémantique restrictive, avec l'hypothèse (assuming that) que les horloges sont discrète.

5.2.4 Le langage CCSL

CCSL fournit un ensemble de constructions basées sur les horloges et les contraintes d'horloges afin de spécifier des propriétés temporelles. Ces horloges peuvent être soit *denses*, soit *discrètes*. Dans notre contexte, nous prenons compte que les horloges discrètes, qui sont constituées de séquences d'instants de durées *abstraites*. Le langage propose des contraintes d'horloges très riches, qui spécifient des contraintes sur l'ordre des instants de différentes horloges. Il est ainsi pratiquement possible de définir des horloges périodiques et des échantillonnages d'horloges périodiques. CCSL est un langage très récent et n'est pour l'instant pas destiné à la génération de code mais à la vérification formelle. Comme il est très général, une génération de code efficace nécessiterait de ne retenir qu'un sous-ensemble des contraintes proposées par le langage.

Pour pouvoir modéliser des applications distribuées (e.g. sytèmes répartis, circuits numériques), il est nécessaire d'identifier un ensemble de référentiels temporels ou horloges et des relations de causalité entre des évènements. Les horloges logiques peuvent être référencées dans l'expression de contraintes temporelles pour exprimer les relations de causalité comme par exemple des contraintes de synchronisation ou de précédence. Ces horloges peuvent aussi être exploitées pour constituer des horloges échantillonnées (sous-horloges) ou de filtrage. (voir dans [And10], les spécifications d'un ensemble de relations). [And07] précise que cette vision du temps permet de manipuler la notion de *simultanéité* dans une succession d'instants discrets. Dans un instant donné, s'exécutent des évènements pouvant être dépendants causalement entre eux, et considérés comme simultanés à l'image de la notion de *réaction instantanée*, abstraction exploitée dans les langages synchrones [BB91]. Nous rappelons succintement ci-dessous les bases formelles du langage CCSL.

5.2.5 Bases formelles du langage CCSL

En reprenant la formalisation décrite dans [And07], une horloge est considérée comme un quintuplé $\langle I, \preceq, \mathcal{D}, \lambda, \mu \rangle$ où I est un ensemble d'instants, $\preceq$ est une relation ordre, transitive, binaire sur I, $\mathcal{D}$ est un ensemble d'étiquettes, $\lambda : \mathcal{I} \rightarrow \mathcal{D}$ est une fonction d'étiquetage, μ est un symbole représentant une *unité*.

Une *structure de temps* est un quadruplé $\langle \mathcal{C}, \mathcal{R}, \mathcal{D}, \lambda \rangle$ où $\mathcal{C}$ est un ensemble d'horloges, $\mathcal{R}$ est une relation sur $\bigcup_{a,b \in \mathcal{C}, a \neq b} (\mathcal{I}_a \times \mathcal{I}_b)$, $\mathcal{D}$ est un ensemble d'étiquettes, $\lambda : \mathcal{I}_{\mathcal{C}} \rightarrow \mathcal{D}$ est une fonction d'étiquetage. $\mathcal{I}_{\mathcal{C}}$ est l'ensemble des instants de la structure de temps, quotient de cet ensemble par la relation de coïncidence induite par la relation $\mathcal{R}$.

Les relations entre horloges sont basées sur la relation de précédence $\preceq$ de laquelle dérivent de nouvelles relations sur des instants : La coincidence ($\equiv \stackrel{def}{=} \preceq \bigcap \preceq^{-1}$), la stricte precedence ($\prec \stackrel{def}{=} \preceq \setminus \equiv$), l'independence ($|| \stackrel{def}{=} \overline{\preceq \bigcup \preceq^{-1}}$), et l'exclusion ($\# \stackrel{def}{=} \prec \bigcup \prec^{-1}$).

Relation entre les instants

Une base de temps *TimeBase* dans MARTE est considérée comme un conteneur d'*Instants*, pour toute paire d'instants, il existe toujours au moins un instant entre ces deux.

La Figure 5.4 (b) illustre un exemple de base de temps multiple composée de trois bases : A, B et C. CCSL considère deux types de relations : *Causale* and *Temporelle* ones.

- La relation causale, nommée ainsi *causalité* ou relation de *dépendance*, est exprimée comme suit : $i \leq j$ signifiant que i provoque j , ou bien j depend de i, (i, $j \in I$, D'ou I represente un ensemble d'instants indexé par des nombres naturels. Cette relation sert à exprimer des chemins de causalité. La Figure 5.4 (a) illustre un *chemin de causalité*, tout en montrant une relation de dependance ainsi que le comportement temporel du modèle de temps impliqué. L'instant $c1$ marque le début de la trace. Cependant, le chemin de temps inclue les occurences suivantes (*c1, c2, a2, c3, a3*) ce chemin fait référence à une causalité. Dans cette figure, le chemin causale est illustré par des lignes pointillés. L'evènement (a2) doit être exécuté aprés l'execution de deux occurences de l'horloge C ($c1$ and $c2$). Commençant par l'evènement (a2) l'execution de chaque évènement de l'horloge A doit être suivi par l'exécution d'un évènement de l'horloge C selon l'ordre : $a2$, $c3$, $a3$, $c4$, etc.

- Les relations temporelles entres les horloges logiques sont basées sur la relation de Précedence $\preceq$ à partir de laquelle dérivent de nouveaux relations d'instants : Coincidence ($\equiv \triangle= \preceq \bigcap \preceq^{-1}$), Strict precedence ($\prec \triangle= \preceq \setminus \equiv$), Independence ($||\triangle= \overline{\preceq \bigcup \preceq^{-1}}$), and Exclusion ($\#\triangle= \prec \bigcup \prec^{-1}$).

Une spécification CCSL consiste en la déclation d'un ensemble d'horloges et d'un ensemble de relations entre horloges. Ces relations s'appliquent sur les horloges et sur des

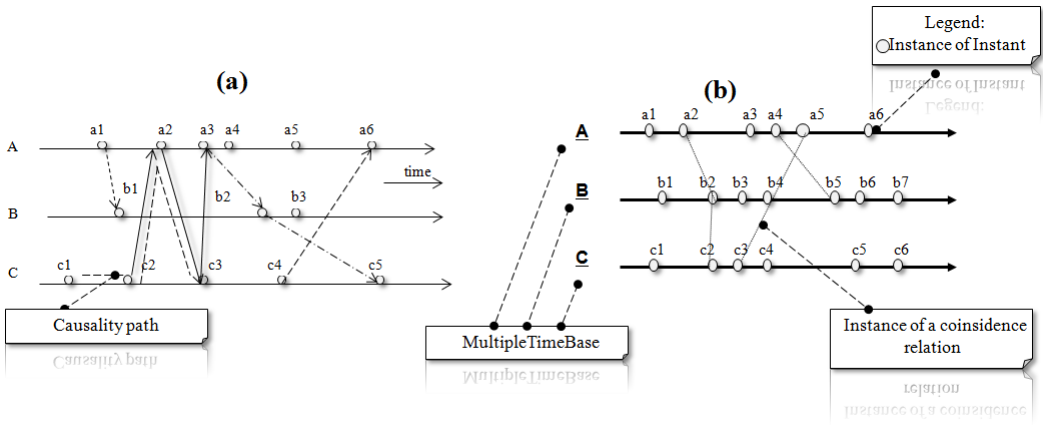

Figure 5.4: Exemple d'une base de temps: Temps logique en MARTE -Ordre partiel entre les évènements- [OMG10].

expressiosn référençant les horloges. Une exécution (*run*) représente une évolution possible des horloges, conformes aux relations exprimées. A chaque pas (*step*) d'exécution, un ensemble d'horloges (ou évènements) surviennent (*tic* d'horloge). Ces *tics* d'horloges respectent la spécification CCSL c'est à dire les relations entre horloges. A chaque pas d'exécution, la sémantique opérationnelle de CCSL permet d'évaluer les conditions pour lesquelles une horloge *tic*.

Une relation entre horloges est la généralisation de relations entre instants sur tous les instants d'une horloge. L'ensemble d'instants I est indexé par des nombres naturels pour respecter l'ordre dans I. Soit $N^* = N \setminus 0$. $idx : I \rightarrow N^*$. $\forall i \in I$, idx(i) = k si et seulement si i est le k^{eme} instant I. Pour tout horloge $c = \langle I_c, \prec_c, \mathcal{D}_c, \lambda_c, \mu_c \rangle$, $c[k]$ denote le k^{eme} instant dans I_c c'est-à-dire, $k = idx_c(c[k])$.

Dans la modélisation $CCSL$, il existe deux relations d'hologes de base : La relation de*Coincidence* et de *Precedence*. On présente dans ce qui suit quelques relations décritent dans [YM11] .

Exemples de contraintes CCSL

Nous ne présentons ici que quelques unes des relations décrites dans [And10, YM11], celles qui sont nécessaires pour la mise en œuvre du modèle du cas d'étude illustratif décrit dans cette thèse, à savoir la relation de coinsidence, d'alternance, de précédence stricte et de filtrage.

La relation de coincidence est une forte forme de relation temporelle entre instants. Nommée ainsi une relation de jonction d'instants "*junction instants*" appartenant à différentes bases de temps qui peuvent être coinsidées (i.e., en même temps), dans lequel n'importe

quelle jonction d'instant coinside avec elle même. En effet, ce type de relation est une équivalence d'instant sur l'ensemble d'instants des bases impliquées. La relation de coinsidence (dénotée *equiv*) entre deux horloges A et B impose une forte dépendence synchrone, où e $k^{ème}$ instant de A doit coinsider avec le $k^{ème}$ instant de B.

Cette relation n'a pas obligatoirement cette stricte relativité, cela veut dire que çà peut représenter une synchronisation d'horloges ou bien même un choix de conception.

L'ensemble d'instants impliqués par cette relation doit être un ensemble partiellement ordonés. Cependant, la relation de coinsidence est généralement représentée par des diagrammes liants des paires d'instants coinsidants. La Figure 5.4(b) montre un exemple de base de temps multiple. Les jonctions d'instants $a2$ and $b2$ coincident. Ainsi que $b2$ et $c2$. On peut ainsi remarquer que, $a2$ et $c2$ coinsident aussi, –par transitivité.

La relation $A\ CoincidentWith\ B$ est formellement exprimée par l'expression (1):

$$A\ CoinsidentWith\ B \Leftrightarrow (\forall k \in N^*).\ A[k] \equiv B[k]) \qquad (1)$$

La relation d'alternance (dénotée $alternatesWith$) est une relation asynchrone entre deux horloges C_1 et C_2. Elle spécifie que pout tout nombre naturel k, le k^{eme} instant de C_1 survient avant le k^{eme} instant de C_2, et que le k^{eme} instant de C_2 survient avant le $k + 1^{eme}$ instant de C_1. En reprenant encore la formalisation décrite dans [And07], la relation $C_1\ alternatesWith\ C_2$ s'exprime formellement par l'expression (1) :

$$C_1\ alternatesWith\ C_2 \Leftrightarrow (\forall k \in N^* \\ C_2[k] \in I_{C2})((C_1[k] \in I_{C1})\ \wedge\ C_1[k] \prec C_2[k]\ \wedge\ C_2[k] \prec C_1[k+1]) \qquad (1)$$

Pour notre cas d'étude, nous illustrons la relation $read_i$ alternatesWith $write_i$ par le chronogramme Figure 5.5(a) et l'automate Figure 5.5(b).

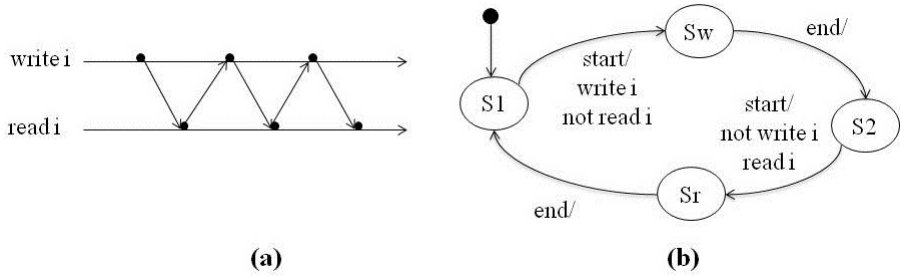

Figure 5.5: Illustration de la contrainte d'alternance : $read_i$ alternatesWith $write_i$.

Notons que pour l'alternance non stricte, il faut remplacer, dans l'expression (1) ci-dessus, le symbole $\prec$ par $\preceq$.

La relation de précédence (dénotée *strictPrec*) est une relation asynchrone entre deux horloges C_1 et C_2. C_1 est dite strictement plus rapide que C_2, ou "C_1 précède strictement

C_2", noté C_1 *strictPrec* C_2, spécifie que pour tout nombre naturel k, le k^{eme} instant de C_1 survient avant le k^{eme} instant de C_2, c'est à dire $\forall k \in N^*,\ C_1[k] \prec C_2[k]$. La relation C_1 *strictPrec* C_2 s'exprime formellement par l'expression (2):

$$C_1\ strictPrec\ C_2 \Leftrightarrow (\forall k \in N^*, \\ C_2[k] \in I_{C2})((C_1[k] \in I_{C1})\ \wedge\ C_1[k] \prec C_2[k]) \qquad (2)$$

Pour notre cas d'étude, nous illustrons la relation $read_i$ strictPrec *comput* par le chronogramme figure 5.6(a).

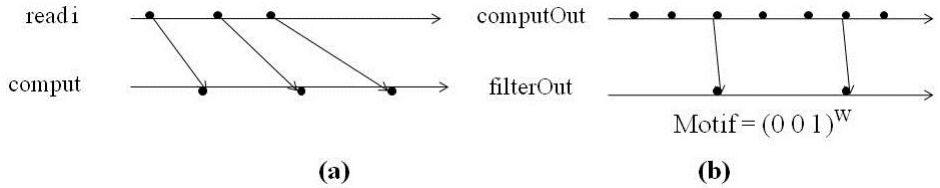

Figure 5.6: Illustration des contraintes de StrictPrecedence (a) et de filtrage (b).

La relation de filtrage (dénotée $filteredBy$) est une relation qui définit une sous-horloge à partir d'une horloge discrète donnée. L'application entre les deux horloges est caractérisée par un *motif de filtrage* (ou plus simplement filtre) codé par un mot binaire fini ou infini $w \in \{0,1\}^* \cup \{0,1\}^w$.

$C_1\ filteredBy\ w$, définit la sous-horloge C_2 de C_1 telle que $\forall k \in N^*,\ C_2[k] \equiv C_1[w \uparrow k]$, où $w \uparrow k$ est l'indice du k^{eme} 1 dans le motif w. Les mots binaires sont utilisés pour représenter des séquences d'activations. Quand ces dernières ont un caractère périodique, elles peuvent être représentées par des mots binaires périodiques dénotés par $w = u(v)^w$. u et v sont des mots binaires finis, appelés respectivement préfixe et période [And11].

La relation $C_2\ =\ C_1$ FilteredBy w s'exprime formellement par l'expression (3):

$$C_2\ =\ C_1\ filteredBy\ w \Leftrightarrow ((\forall k \in N^*, C_1[k] \in I_C) \\ (C_2[k] \equiv C_1[w \uparrow k])) \qquad (3)$$

5.3 Méthodologie pour la vérification des modèles temporisés

Après avoir définit les langages et outils de notre transformation en Section 5.2, nous présentons dans cette section le principe général de notre méthodologie ainsi que la structure du modèle de transformation proposé.

5.3.1 Méthodologie : Structure du modèle de transformation proposé

Une vue générale des composants requis pour la vérification est illustrée dans la Figure 5.7. Nous détaillons dans ce qui suit chaque composant.

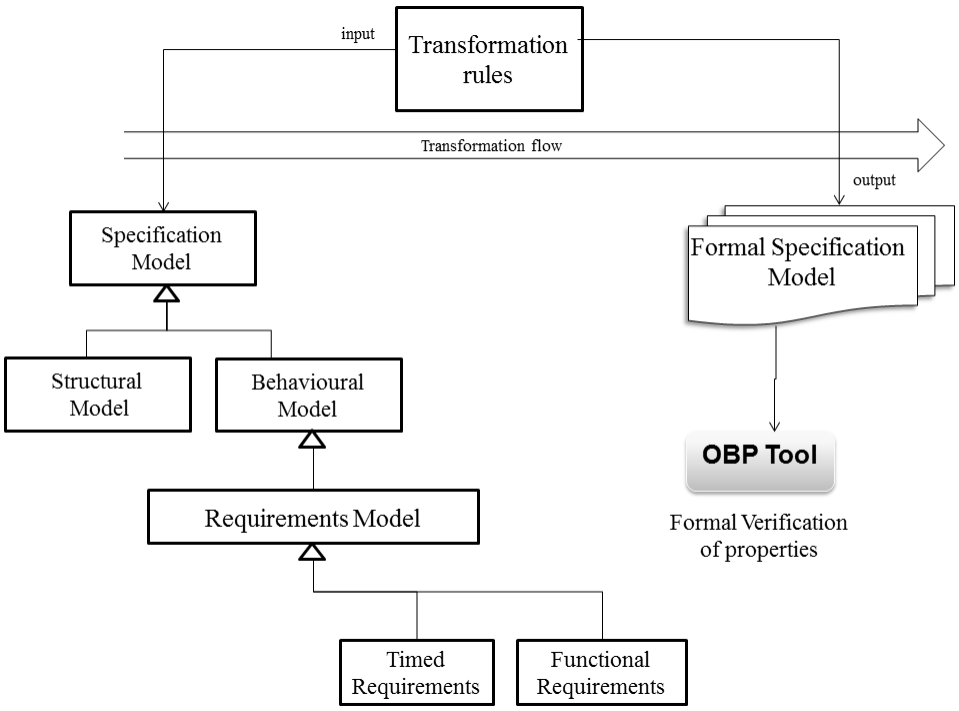

Figure 5.7: Structure du modèle de transformation proposé.

Modèle de spécification Le modèle de spécification est une représentation de l'architec--ture du système temp-réel (Modèle structuel) et son comportement (Modèle comportemental).

UML MARTE permet la modèliser une large variété de système temps réel. Pour mettre en oeuvre notre approche, on a choisis un sous ensemble d'UML MARTE qui contient des concepts spécifiques et pértinents considérés suffisants pour la vérification. Par exemple, nous somme seulement interessé par des concepts de temps logique, les unités temps-réel (pour la modélisation de la structure), des contraintes d'horloges, ports synchrone/asynchrone (pour le comportement). Afin d'intégrer le processus de vérification dans notre approche, nous avons défini une sémantique précise pour le sous ensemble de MARTE pris en compte dans notre transformation.

Modèle d'exigences Le modèle d'exigence représente les contraintes temporelles ainsi que fonctionnelles. Les exigences fontionnelles décrivent une trace d'execution qui doit être assurée par le comportement du système. Les contraintes temporelles expriment des exigences sur le temps pendant lequel une execution doit être accomplie. Ce type de contraintes doit être défini tout en respectant les base de temps prédéfinie. Nous utilisons le langage CCSL pour représenter les contraintes temporelles, et CDL pour exprimer formellement les contraintes temporelles et fonctionnelles.

Modèle de spécification formelle Le modèle de spécification formelle est une représent--ation formelle du système et son comportement i.e., en utilisant un vocabulaire spécifique, une telle représentation est basée sur des outils dont la sémantique est formellement définie. Dans le but de garder la cohérence entre le modèle de spécifications et le modèle d'exigences. Ces modèles sont transformés en une spécification formelle. Pour cela, nous utilisons Fiacre comme un langage clé pour faire le lien entre les spécifications UML MARTE et les outils d'analyse formelle.

Régles de transformation Les régles de transformations spécifient la correspondance entre les élements du modèle source (UML MARTE/CCSL) et leurs représentations formelle dans un langage cible (Fiacre). Le modèle cible permet la distingtion entre les contraintes temporelles et fonctionnelles. Nous nous sommes inspirés des régles de transformation d'UML vers Fiacre décrites dans [FJ14].

Nous détaillons dans ce qui suit le processus de transfomation du modèle de spécification.

5.3.2 Modèle de sortie : Une méthodologie pour la vérification des modèles

Notre méthodologie pour la vérification de modèles temps réel est illustrée dans la Figure 5.8. A partir de la spécification du modèle source, on fait extraire le modèle fonctionnel et les exigences attendues en deux partie distingtes, le modèle d'application d 'UML MARTE

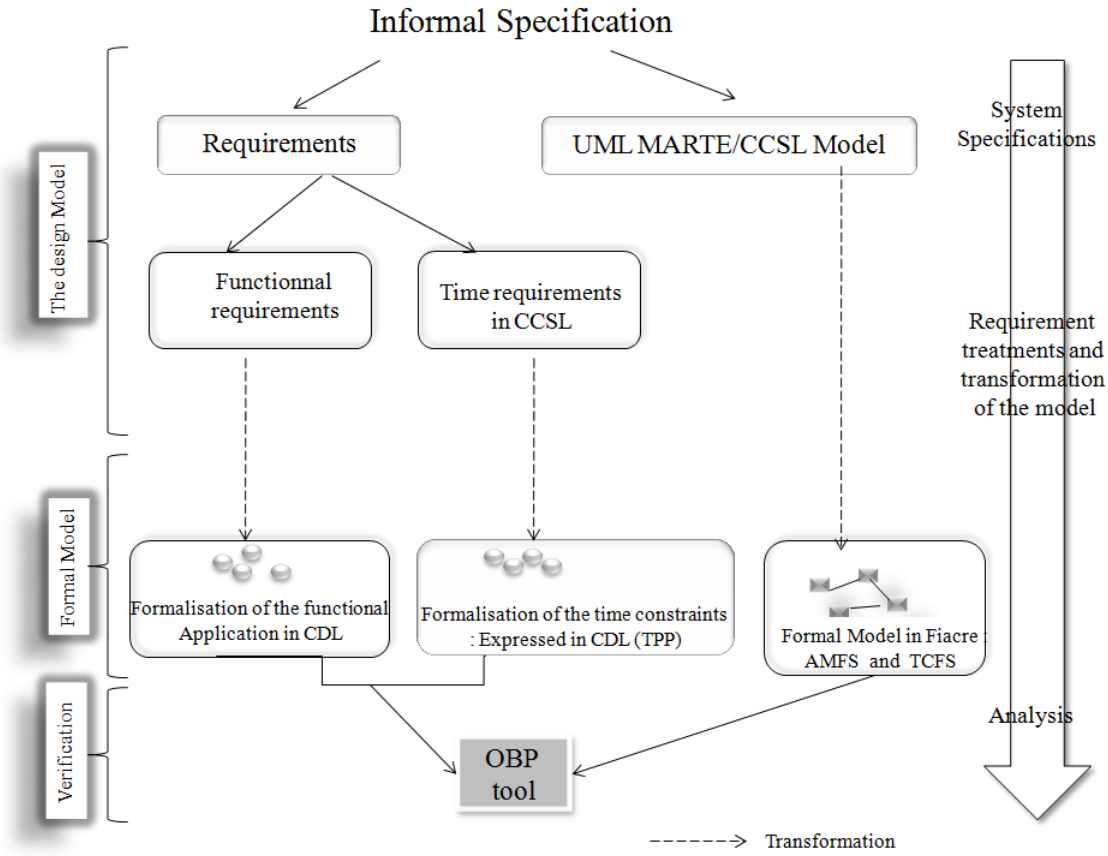

Figure 5.8: Vue générale de la méthodologie proposée.

et les exigences attendues (incluant les contraintes temporelles et fonctionnelles), respectivement. Deuxièment, le modèle applicatif est transformé en modèle Fiacre, que nous le nommons *Application Model Formal Specification (AMFS)*. De même, les contraintes CCSL sont transformées en spécifications formelles de contraintes temporelles *Time Constraint Formal Specification(TCFS)*. De plus, le document de contraintes CCSL sont par la suite transformées en un ensemble de patrons de propriétés *Time Property Patterns (TPP)*, correspondant à des observateurs en langage CDL. Chaque patron de propriétés correpond à une relation CCSL. Le modèle formel obtenu est complété avec un ensemble de propriétés fonctionnelles, exprimées ainsi en CCSL, dans le but de vérifier le compotement du modèle applicatif. Finalement, le programme Fiacre et les observateurs CDL sont combinés pour être vérifier avec l'outils OBP[DPC+09].

Nous examinons maintement la structure de programme Fiacre en sortie. Le programme Fiacre résultant de la transformation de modèle de spécification est composé de quatre

parties :

(1) un processus correspondant aux objets actifs du modèle applicatif, que nous nommons processus fonctionnels,

(3) un processes correspondant aux contraintes CCSL, que nous nommons procesus de contraintes,

(2) un processus ordonnonceur *Scheduler*,

(4) un composant contenant les instances des processus générés.

La Figure 5.9 illustre comment ces parties sont reliées les unes aux autres.

Le modèle UML MARTE est transformé (i.e., sans une relation hiérarchique) en un ensemble de processus communicants. Spécifiquement, chaque objet actif d'UML MARTE est transformé à une instance de processus Fiacre, que nous nommons processus fonctionnels "*functional process*".

Les processus fonctionnels communiquent les uns aux autres avec des variables partagées ou des ports pour les communications synchrones, comme c'est illustré dans Figure 5.9.

Pour permettre le déclenchement de ces processus, un ensemble d'horloges logiques (tab_Clock dans Figure 5.9) est déclaré au debut du modèle Fiacre généré. [9] : Chaque processus fonctionnel est associé à une de ces horloges, avec au moins un port de synchronisation (e.g., the $sync\_pr_3$ port pour le processus *Functional_Proc_3*). Une priorité peut être associée à une horloge, si un ordre de priorité est exigé sur le déclenchement de processus fonctionnels. Le processus ordonnonceur *Scheduler* utilise ces états d'horloges pour synchroniser les processus fonctionnels, et les processus de contraintes sont en charge de mettre à jour les états d'horloges. Nous detaillons ce point dans la section 5.3.3. Le rôle du processus ordonnonceur est de définir un ordre d'activation des processus fonctionnels par rapport aux priorités attribuées aux horloges.

Nous détaillons dans ce qui suit le modèle d'exécution en code Fiacre.

5.3.3 Modèle d'exécution

L'algorithme de l'ordonnonceur *Scheduler* est illustré dans Figure 5.10. Ce dernier itère sur la séquence des instants, chaque itération correspond à un instant qui est composé de quatre phases principales. Durant la phase de début *start* d'un instant donné, les horloges sont initialisées ainsi que la synchronisation des processus de contraintes, permettant le déclenchement de leur exécution (step 1).

A la phase *update*, les états des horloges sont modifiés par les processus de contraintes (step 2). A la phase *end*, l'ordonnonceur se synchronise avec la fin de l'exécution des processus de contraintes (step 3). A la phase *active*, l'ordonnonceur sélectionne les processus

[9] Chaque occurence d'evenement dans une contrainte CCSL se réfer à une horloge spécifique, déclarée dans la structure Fiacre *tab_Clocks*.

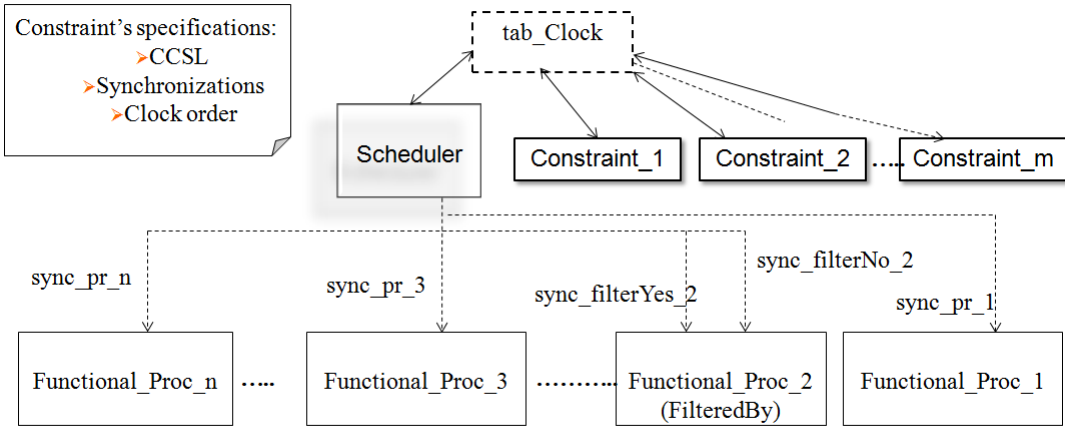

Figure 5.9: Le Modèle formel Fiacre généré.

Table 5.1: Règles d'évaluation de la valeur *enable_tic*

clock_state	enable_tick	*Commentaire*
0	false	Indéterminé
1	false	Non actif dans l'instant courant
2	true	Actif dans l'instant courant

fonctionnels selon les états des horloges (mises à jour à la phase the *update*) pour qu'ils soient déclenchés aprés leurs synchronisation (step 4).

En pratique, une horloge de *tab_Clock* est une structure de données de trois variables, *dead*, *enable_tic*, et *clock_state*.

Ces variables conditionne le *tick* de l'horloge. La variable spéciale *dead* est fixée à *true* quand l'horloge associée ne doit plus *tiquer* dans le reste de l'execution. A chaque itération, une fois un processus de contrainte met à jour la valeur *clock_state*, *Scheduler* lit ces valeurs et met à jour *enable_tic* selon de la Table 5.1. Si *enable_tic* est évaluée à *true*, les processus fonctionnels associés à l'évènement se synchronise avec *Scheduler*, ce qui déclenche un pas d'exécution dans le processus fonctionnel. Sachant que la valeur 0 pour la variable *clock_state* indique un état indeterminé (e.g., dans le cas ou un conflit s'occure entre deux contraintes).

Nous détaillons dans ce qui suit un cas d'étude que nous avons modélisé en MARTE, avec lequel nous illustrons le processus de notre transformation.

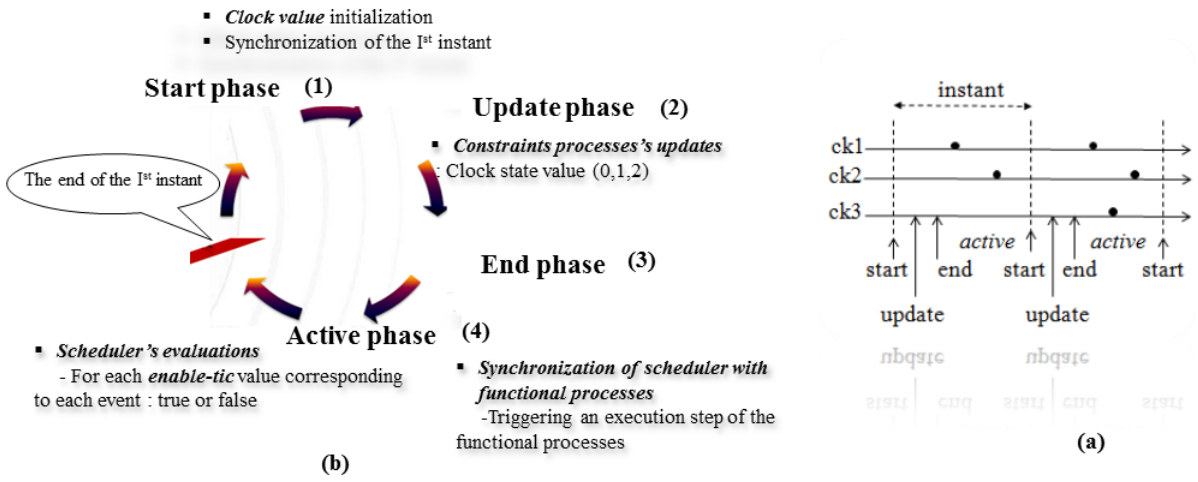

Figure 5.10: Les itérations du processus Ordonnonceur (Scheduler).

5.4 Illustration sur un cas d'étude

Nous considérons un circuit d'acquisition (à n voies) (Figure 5.11) de données constitué de composants d'acquisition ($Sensor_i$ et Acq_i) ($i \in [0 \ldots n-1)$, d'un composant ($Comput$) de traitement des données acquises et d'un filtre ($Filter$) échantillonnant les valeurs calculées. Chaque voie i d'acquisition est associée à un couple de composants $Sensor_i$ et Acq_i.

Pour l'illustration, nous supposons que, pour une voie i, le composant $Sensor_i$ reçoit des données de l'environnement (de Dev_i externe au circuit) et transmet la valeur à Acq_i par l'intermédiaire une mémoire Mem_i. Les composants Acq_i fournissent à $Comput$ chaque donnée $data_i$ par un port de communication synchrone $portAcq_i$. $Comput$ fournit une valeur $data$, calculée à partir des données $data_i$, au filtre via une fifo. Le filtre fournit à l'environnement des données échantillonnées (Dev_{out} externe au circuit).

Les contraintes temporelles associées à ce circuit sont les suivantes : $Req1$: Chaque donnée acquise $data_i$ est écrite dans la mémoire Mem_i avant la lecture par Acq_i. $Req2$: $Comput$ débute le calcul de la donnée $data$ après n réceptions de $data_i$ provenant de chaque Acq_i. $Req3$: $Comput$ termine le calcul de la donnée $data$ après le début du calcul. $Req4$: $Filter$ fournit à l'environnement une valeur échantillonnée suivant une séquence d'une valeur toutes les 3 valeurs calculées par $Comput$. Le chronogramme (Figure 5.12) illustre une exécution partielle du circuit.

En complément aux contraintes temporelles précédentes, nous exprimons des exigences

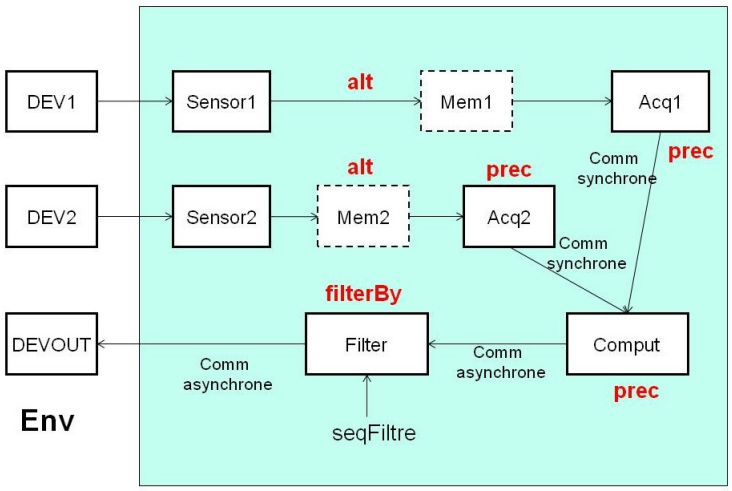

Figure 5.11: Architecture du composant.

qui sont associées spécifiquement au comportement attendu du circuit. En guise d'illustration, nous suposons que, sur chacune des 2 voies d'acquisition $DEV_n, n \in \{1,2\}$, N données $data_{ni}, i \in [0 \dots N-1]$ sont acquises de l'environnement. Nous supposons, également, que le composant *Comput* applique par exemple aux 2 données reçues $data_{1i}$ et $data_{2i}$ reçues respectivement de DEV_1 et DEV_2, la somme $data_{1i} + data_{2i}$. Nous pouvons donc exprimer l'exigence suivante : $Req5$: les données $result_j, j \in [0 \dots (N-1)/3]$ fournies à l'environnement après un échantillonnage (d'une valeur sur 3) ont pour valeurs $data_{1k} + data_{2k}$ *avec* $k = (3 * j) + 2$.

Pour notre cas d'étude, nous illustrons la relation $filterOut = computOut\ FilteredBy\ (001)^w$ par le chronogramme Figure 5.6(b).

En résumé, l'ensemble des exigences temporelles, pour notre cas d'étude, décrites en Section 5.4 sont spécifiées en CCSL comme suit :

$$
\begin{array}{ll}
read_1\ alternatesWith\ write_1, read_2\ alternatesWith\ write_2 & (Req1) \\
read_1\ strictPrec\ comput,\ read_2\ strictPrec\ comput & (Req2) \\
comput\ strictPrec\ computOut & (Req3) \\
filterOut\ =\ computOut\ FilteredBy\ (001)^w & (Req4)
\end{array}
$$

5.4.1 La Modélisation du circuit

La Figure 5.13 illustre le modèle de ce circuit en utilisant les concepts du profil MARTE [OMG10]. Nous définissons un package nommé *CircuitApplication* contenant la description du circuit. Nous considérons *Sensor1*, *Sensor2*, *Acq1*, *Acq2*. *Comput*, *Filter*, *Dev1*, *Dev2* et *DevOut* comme des objets actifs (stéréotypés avec *RtUnit*). Chacune de ces unités a la possibilité d'invoquer l'envoi et la réception de données.

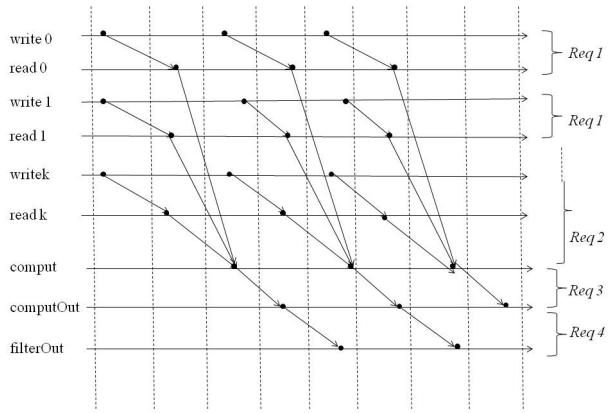

Figure 5.12: Chronogramme d'une exécution du circuit.

Le *CircuitApplication* package introduit également deux ressources partagées, appelées respectivement, *M1* et *M2*, stéréotypés par *SharedDataComResource*. Ces ressources partagé--es permettent l'échange de données entre les objets $Sensor1$ et $Acq1$ (resp. $Sensor2$ et $Acq2$). Nous définissons des services $read$ / $write$ qui permettent de lire / d'écrire les données patagées, en établissant *op_Write1* en tant que service d'écriture et *op_Read1* en tant que service de lecture. *Comput* et $Acq1$ (resp. $Acq2$) échangent des données par communication synchrone. Ainsi, nous spécifions cette communication par l'instanciation des ports *portAcq1* et *portAcq2*. Nous les connectons respectivement à *ReceivedFrom_Acq1* et *ReceivedFrom_Acq2* port.

Quand *Comput* exécute un calcul, il génère une donnée pour *Filter* à travers un port dédié connecté à un *Datapool* nommé *FifoFrom_Comput*. Cette $fifo$ est caractérisée par l'attribut *ordering* qui est fixé à la valeur *FIFO* pour indiquer le type de communication asynchrone.

Tous les objets actifs fournissent des actions grâce à leurs interfaces qui offrent des opérations stéréotypés par *rtAction* ou *rtService*. Les opérations sont définies par les *op_Write1 op_Write2*, *op_Read1*, *op_Read2 op_Comput* et *op_filter* interfaces, limités aux ports dédiés. Ces ports invoquent un stéréotype *clock*, qui indique que les actions des interface fournies sont considérés comme des événements qui sont invoqués par ces horloges. Par exemple, *Acq1* accède à l'objet ressource partagée (*M1*) par un service de lecture, en invoquant l'action de lecture. CCSL définit les relations entre horloges associées aux opérations de lecture et d'écriture garantissant la politique d'accès à la même ressource empêchant les lecteurs et les écrivains pour accéder simultanément à la même ressource.

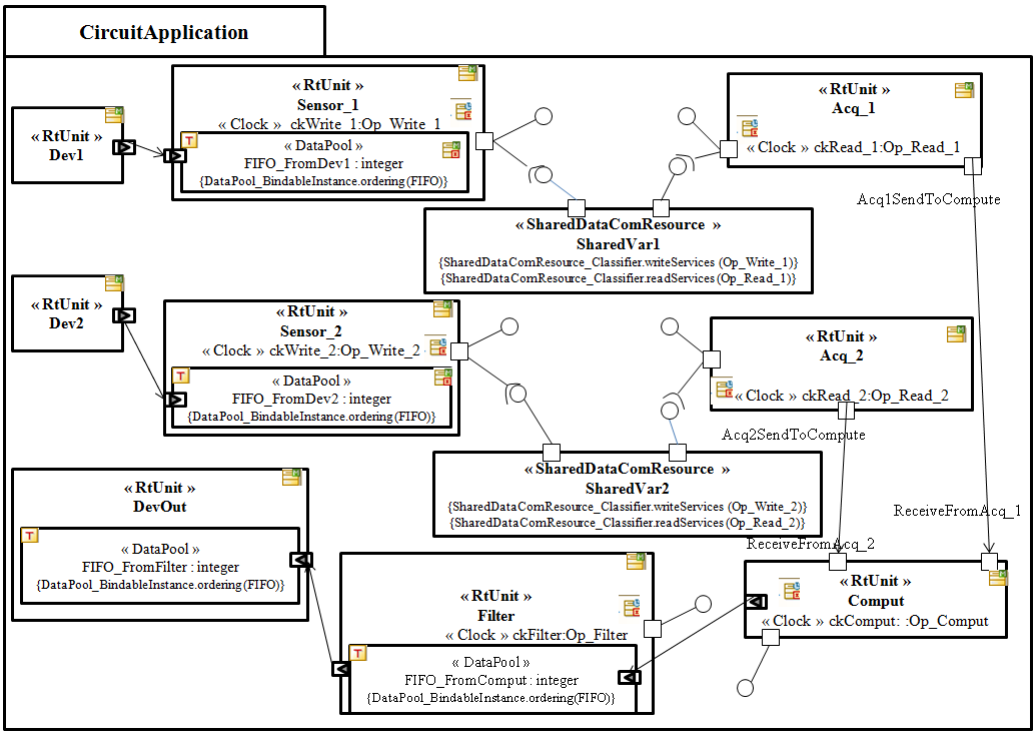

Figure 5.13: Illustration de l'architecture du circuit en MARTE.

5.4.2 Modélisation du temps

En outre, nous représentons un extrait du sous-profil UML pour la modélisation de temps de MARTE, qui spécifient les horloges, en utilisant les concepts de *MARTE::TimeLibrary* pour décrire les aspects temporels. La description (Voir Figure 5.14) représente une spécification des contraintes sur les horloge. Nous définissons une classe stéréotypée par *ClockType*, nommé *L_clock*. A partir de *L_clock*, nous définissons les autres horloges logiques, comme illustré dans la Figure 5.14. Nous déclarons les instances d'horloges appartenant au domaine de temps. Dans cet exemple, un seul domaine temporel est considéré, et elle possède des horloges qui sont les instances de *L_clock*. Nous introduisons le stéréotype *ClockConstraint* pour spécifier des contraintes associé aux horloges. Par exemple, la dépendance entre l'écriture et la lecture, de em Sensor1 (resp. *Sensor2*) et *Acq1* (resp. *Acq2*) est spécifié par la contrainte *AlternatesWith* qui impose à *write* et *read* d'être alternés. Une autre contrainte (*FilteredBy*) spécifie la contrainte de filtrage pour *filter* avec le patron de filtrage 001.

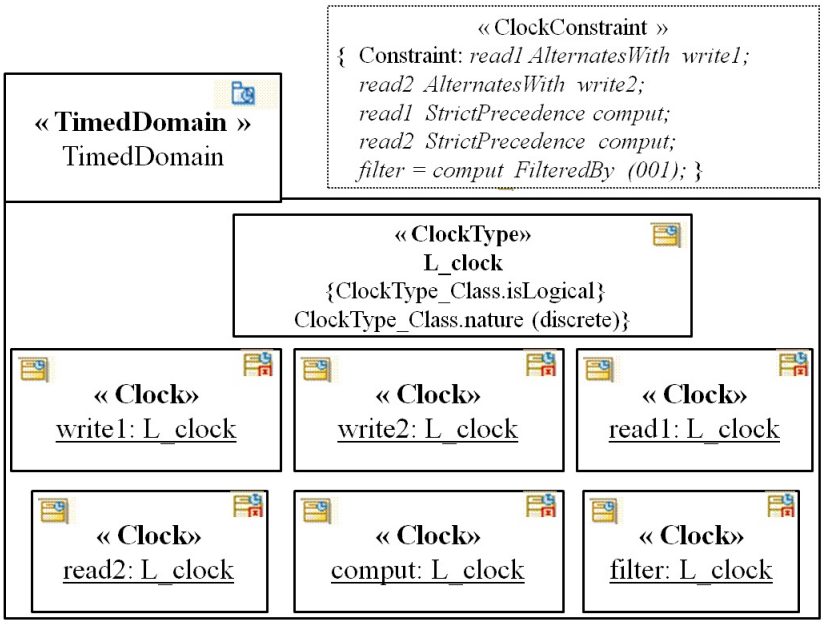

Figure 5.14: Illustration des horloges en MARTE.

5.5 Conclusion

Nous avons montrer dans ce chapitre la stucture du modèle de transformation proposé, la méthodologie pour vérifier des modèles temporisés en MARTE et un aperçu sur le modèle de sortie. Nous allons détailler dans ce qui suit notre application à la validation de modèles temporisés avec les contraintes CCSL, ainsi que les principes d'implantation pour la génération d'architecture Fiacre pour la vérification formelle avec le processus de model checking.

Chapitre 6

Transformations des modèles : Application à la validation de modèles de temps

"*Any verification using model-based techniques is only as good as the model of the system.*"

"The principles of model checking", by *Christel Baier* and *Joost-Pieter Katoen*. "The MIT press, Massachusetts Institue of Technology. Cambrigde, Massachusetts .

6.1 Introduction

Dans le but de pouvoir vérifier les propriétés sur le modèle, il est nécessaire de disposer d'un modèle formel simulable et de propriétés formelles. Dans une approche de *model−checking*, le modèle doit être exploré exhaustivement et les propriétés vérifiées lors de l'exploration. Dans notre approche, nous choisissons de transformer le modèle du circuit en un programme au format Fiacre [FGP+08]. Nous choisissons également d'exprimer les propriétés sous la forme d'observateurs et d'exécuter une analyse d'atteignabilité des états d'erreurs des observateurs. Nous montrons par la suite, Section 7.2, l'expression des observateurs basée sur le langage CDL [DR11b].

Nous ne décrivons pas ici le langage Fiacre qui a été largement présenté dans plusieurs publications[1]. Nous présentons les principes d'implantation des programmes Fiacre à partir de la description de l'architecture du circuit décrit en 5.4 et des contraintes CCSL associées correspondant aux exigences *Req*1 à *Req*4 listées en 5.4.

Cette section présente également les principes de traduction du modèle UML MARTE avec contraintes CCSL en des programmes Fiacre. Ces principes ont été mis en œuvre dans notre générateur de code.

Le schéma présenté dans la Figure 6.1 illustre la plateforme de transformation.

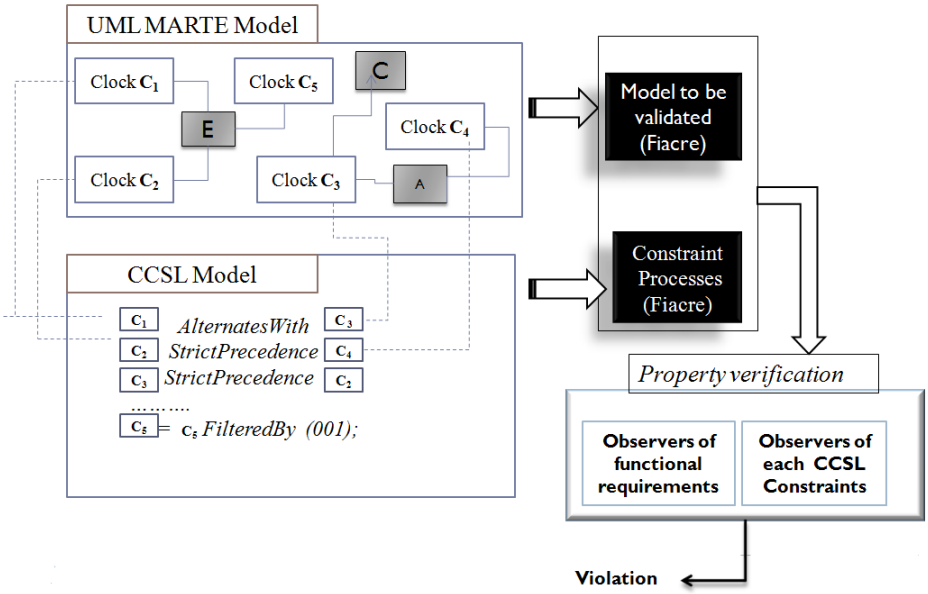

Figure 6.1: Plateforme de transformation des modèles MARTE annotés avec CCSL.

[1] voir http://hal.inria.fr/inria-00262442)

6.2 Transformation de modèles MARTE-CCSL à un programme Fiacre

Nous preséntons dans cette partie les principes de transformation des modèles MARTE enrichis avec des contraintes CCSL (Fig.6.2.a) en un code Fiacre (Fig.6.2.b). Nous illustrons ces principes sur l'architecture de circuit introduit dans la Section 5.4. Plus spécifiquement, nous détaillons comment les contraintes (*Req*1*a*, *Req*1*b*, *Req*2*a*, *Req*2*b* and *Req*3) sont tranformées en processus Fiacre.

6.2.1 Vue générale des principes de tranformation et d'implantation

L'idée générale de la traduction d'un modèle UML-MARTE consiste à la génération des élements Fiacre suivants (Fig.6.2.b): (1) la génération d'un ensemble de processus correspondants aux objets actifs du modèle MARTE, (2) la génération d'un ensemble de processus correnpondants aux contraintes CCSL, (3) la génération d'un processus Fiacre ordonnanceur " *Scheduler*", pour synchroniser l'execution des processus générés ainsi, (4) la génération un composant (*Component*) Fiacre decrivant l'architecture contenant tout les instances des processus générés.

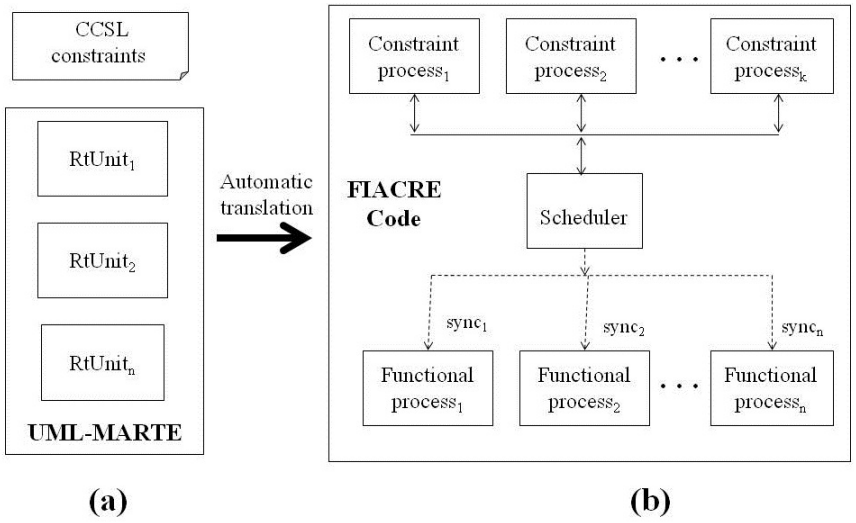

Figure 6.2: Vue globale des principes de transformation.

Les principes de traduction des contraintes CCSL en processus Fiacre sont inspirés du travail décrit dans [YM11]. Ils se basent, premièrement, sur l'implantation, pour chaque contrainte CCSL, d'un processus Fiacre qui implante l'automate (cf section 5.2.5) correspondant à la contrainte (nous nommons ces processus *processus de contrainte*). Deuxièment,

Table 6.1: Mappage de MARTE vers les concepts du langage Fiacre.

MARTE	**FIACRE**
RtUnit	functional process
Clock Constraint	constraint process
DataPool	`queue` structure
SharedDataComResource	Shared variable
Synchronous port $\longrightarrow$	Synchronous communication port
Port with interface $\circ\!)\!\longrightarrow$	Triggering port

un processus que nous nommons *Scheduler* est en charge d'activer chaque processus de contrainte et de contrôler l'état des automates implantés dans ces processus. Enfin, les parties fonctionnelles du circuit, c'est-à-dire *Sensor*, *Acq*, *Comput* et *Filter* sont implantées chacune dans un processus, que nous nommons "*processus fonctionnels*". Nous illustrons dans ce qui suit les principes de traduction sur l'étude de cas.

6.2.2 Mappage entre le modèle MARTE et le langage Fiacre

La transformation de concepts d'UML MARTE en constructions Fiacre est résumée dans le tableau 6.1. Notons que ne nous expliquerons pas les principes de transformation complète, qui sont le sujet de ce travail (Cf. [JTD+14, JD14]). Dans notre cas d'étude, la transformation est appliquée aux élements *RtUnit* suivants : *Sensor1*, *Sensor2*, *Acq1*, *Acq2*, *Comput*, *Filter*. Ces objets actifs du modèle UML sont traduits dans les instances de processus Fiacre, appelés *processus fonctionnels* et correspondent aux parties fonctionnelles du modèle. Cette traduction est basée sur une sémantique formelle du langage UML que nous avons choisi et qui a été décrite dans [Hen12]. Nous ne la détaillons pas ici.

Les contraintes CCSL attachées au modèle MARTE sont ainsi transformer en processus Fiacre, nommé *constraint processes*. Les éléments *DataPool* (pour la communication asynchrone) sont transformés en une structure de données *"Fifo "* prédéfinie en Fiacre, les resources partagées deviennent des variables partagées associées aux processus Fiacre correspondant aux éléments *RtUnit* impliqués. Les deux ports utilisés pour la communication synchrone de deux éléments *RtUnit* sont transformés en un port constructeur prédéfinis dans Fiacre. Finalement, les ports associés à une interface sont transformés en une déclaration de conditionnel Fiacre que nous detaillons dans la section Section 6.2.4.

La Figure 6.3 illustre partiellement la génération de code Fiacre pour $Sensor1$, $Acq1$ et la contrainte $alternatesWith$. Dans cette figure, nous illustrons en pointillé les liens de synchronisation. Par exemple, $Sensor1$ est synchronisé avec *Scheduler* via le port $sync\_pw1$ pour l'exécution d'une opération d'écriture de la donnée *data* dans la mémoire $M1$, patagée entre les processus $Sensor1$ et $Acq1$. Le processus $AlternatesWith$ est synchronisé avec

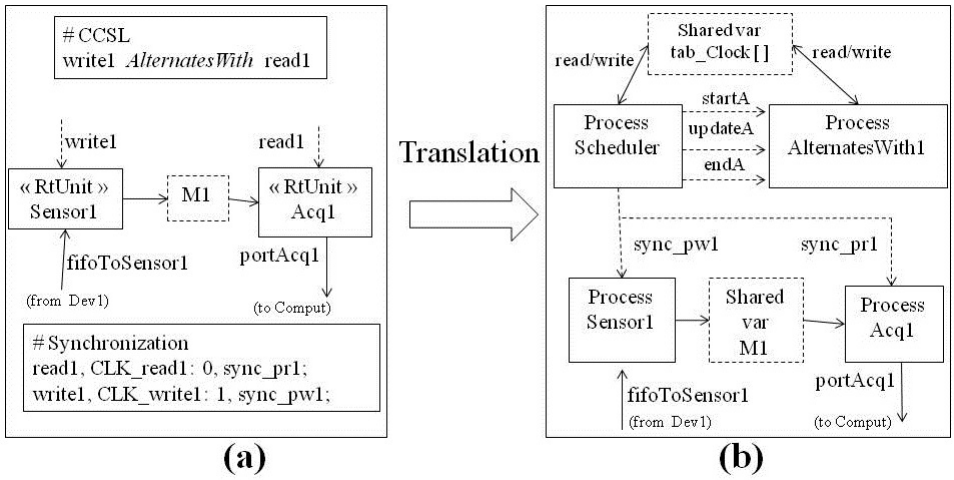

Figure 6.3: Shéma illustrant les principes de transformations : Traduction pour la contrainte d'alternance

Scheduler via les ports $startA1$, $updateA1$ et $endA1$.

Le même type de traduction est appliqué aux autres processus fonctionnels $Sensor2$, $Acq2$, $Comput$, $Filter$ et les autres processus de contrainte $StricPrec$ and $FilteredBy$.

Dans ce modèle, nous mettons en œuvre les objets $Dev1$, $Dev2$ et $DevOut$ avec le langage CDL, en considérant que ces objets s'exécutent dans l'environnement du circuit, comme nous pouvons le constater dans la Section 5.4.

6.2.3 Mappage des contraintes CCSL vers le langage Fiacre.

Chaque contrainte CCSL est implantée par un processus Fiacre qui encode l'automate (cf Section 5.2.5) correspondant à la contrainte (Nous nommons ce processus *constraint process*).

Ces processus de contraintes sont synchronisés par un processus spécifique, *Scheduler*, qui est décrit dans la Section 6.2.4. *Scheduler* synchronise les processus de contraintes via trois ports, *start*, *update* and *end*, pour l'activation de la transition de l'automate de contrainte. Par exemple, dans notre cas d'étude, les transitions du processus $AlternatesWith$ sont synchronisées avec le *Scheduler* via les ports : $startA$, $updateA$ et $endA$. L'automate $AlternatesWith$ met à jour les valeurs de *clock_state* qui permettent de déclencher le processus $Sensor1$ et $Acq1$, (respectivement $Sensor2$ et $Acq2$) par les ports $sync\_pw1$ et $sync\_pr1$ (respectivement $sync\_pw2$ et $sync\_pr2$) (cf section 6.2.4). Nous encodons en Fiacre les automates des contraintes. Nous montrons ici le code pour la contrainte $alternatesWith$ qui correspond à l'implantation de l'automate présenté en Figure 5.5(b). Le principe de codage pour les deux autres contraintes, precedence stricte et filtrage, est

similaire[2].
Comme par exemple, Listing 2 montre le code de la contrainte *AlternatesWith* correspondante au automate représenté dans la Figure 6.4.

Le principe de codage pour les deux contraintes *strict precedence* et *filtering* est similaire.

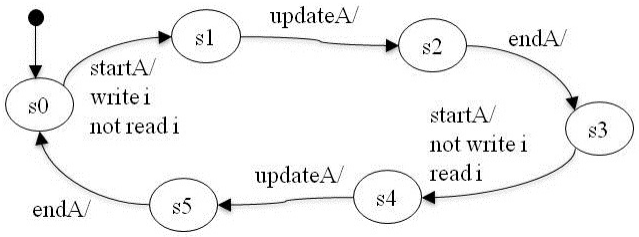

Figure 6.4: Automate de la contrainte $write_i$ $alternatesWith$ $read_i$

```
process AlternatesWith [startA, updateA, endA: in none] // ports
  (&c1: nat, &c2 : nat, &tab_Clocks: T_ARRAY_CLOCK)    // shared variables
is states s1, s2, s3, s4, s5
init to s0
from s0  startA;
         tab_Clocks [c1].clock_state := 2;
         tab_Clocks [c2].clock_state := 1; to s1
from s1  updateA;  to s2
from s2  endA; to s3
from s3  startA;
         tab_Clocks [c1].clock_state := 1;
         tab_Clocks [c2].clock_state := 2; to s4
from s4  updateA; to s5
from s5  endA; to s0
```

Listing 2. FIACRE code of an *alternateWith* constraint.

6.2.4 Interprétation de contraintes de temps avec l'ordonnanceur

Le rôle du processus *Scheduler* est de déterminer l'ordre d'execution des processus fonctionnels en se basant sur l'état des processus de contraintes. Il est en charge d'activer chaque processus fonctionnel. Pour cela, *Scheduler*, les processus de constrainte et fonctionnels sont tous synchronisés au travers de ports de communication synchrones. L'interprétation

[2]Le code complet Fiacre pour le cas d'étude peut être trouvé sur le site http://www.obpcdl.org.

de contraintes de temps est faite par le biais de l'ordonnanceur qui est reponsable d'invoquer (activer) les processus fonctionnels selon les états des processus de contraintes.

L'ordonnonceur "The scheduler" et ses connexions avec les processus

La Figure.6.5 est un extrait de programme Fiacre généré pour les deux processus fontionnels (*Sensor1* and *Acq1*) et le processus de contrainte ($alternatesWith$). Les lignes pointiés représentent des liens de synchronisation.

Par exemple, *Sensor1* est synchronisé avec *Scheduler* via le port *sync_pw*1 afin d'executer une opération d'écriture d'une donnée *data* dans la mémoire $M1$ partagée entre les processus *Sensor1* et *Acq1*.

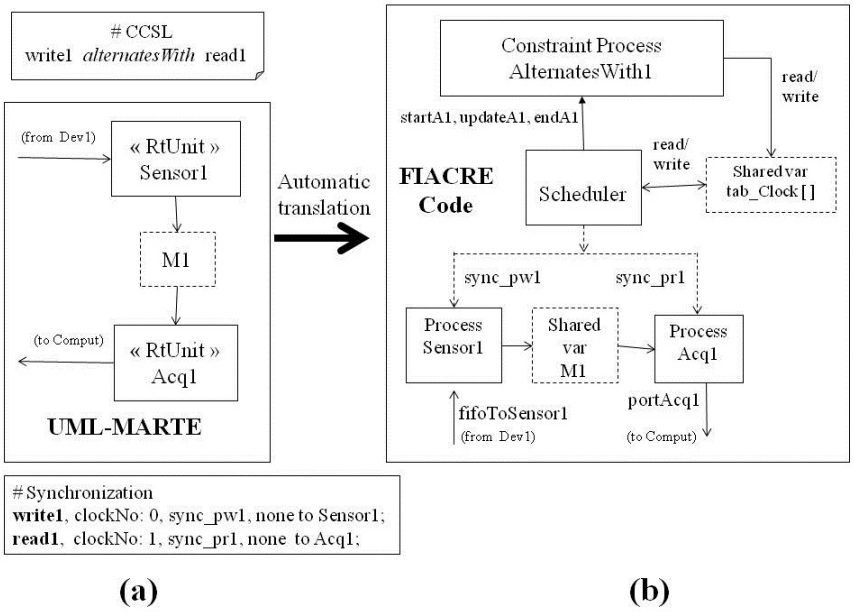

Figure 6.5: Illustration d'une partie d'un code Fiacre généré.

Acq1 et *Comput* communiquent via le port *portAcq*1 avec une valeur entière. *Comput* et *Filter* communiquent par une variable partagée *fifoFromComput* de type *fifo*. *Filter* est synchronisé avec le *Scheduler* via $sync\_filter$ pour l'opération de filtrage et $sync\_filter$ contient une variable de valeur type "boolean" utilisée pour le comportement du *Filter*.

Le comportement de l'ordonnanceur "Scheduler"

Le processus *Scheduler* a pour rôle de déterminer l'ordre des exécutions des processus fonctionnels en fonction de l'état des processus de contrainte. Pour cela, le *Scheduler*, les

processus de contraintes et les processus fonctionnels se synchonisent par des ports. Le processus *Scheduler* ainsi que les processus de contraintes partagent entre eux des horloges logiques (*Table tab_Clocks* structure, cf 6.2.4) qui correspndent à des occurences d'évenements dans le circuit de calcul ($write1$, $write2$, $read1$, $read2$, $comput$, $filterOut$). Le principe de la simulation est le suivant : *Scheduler* itère et exécute, à chaque boucle, plusieurs phases : Une phase *start* d'initialisation des horloges déclarées et d'activation des processus de contraintes. Une phase *end* de synchronisation sur la fin des processus de contrainte. Une phase *active* durant laquelle le *Scheduler* se synchronise avec chaque processus fonctionnel pour que ce dernier s'exécute. Un *instant* de la simulation correspond à la période entre deux phases *start*. Une phase intermédiaire *update* est intercalée entre les phases *start* et *end* pour synchroniser au besoin certaines contraintes (Figure 6.7). L'algorithme exécuté par le *Scheduler* est répété pour simuler la séquence des instants coïncidents.

L'entrelacement ou la simultanéité des exécutions des processus fonctionnels est simulé par des synchronisations entre le *Scheduler* et les processus fonctionnels impliqués à chaque instant délimité temporellement. Par exemple, la Figure 6.6 montre deux horloges $ck1$ et $ck2$ qui sont activées à chaque fois durant le même instant. $ck3$ est échantillonnée avec $ck1$ ou $ck2$.

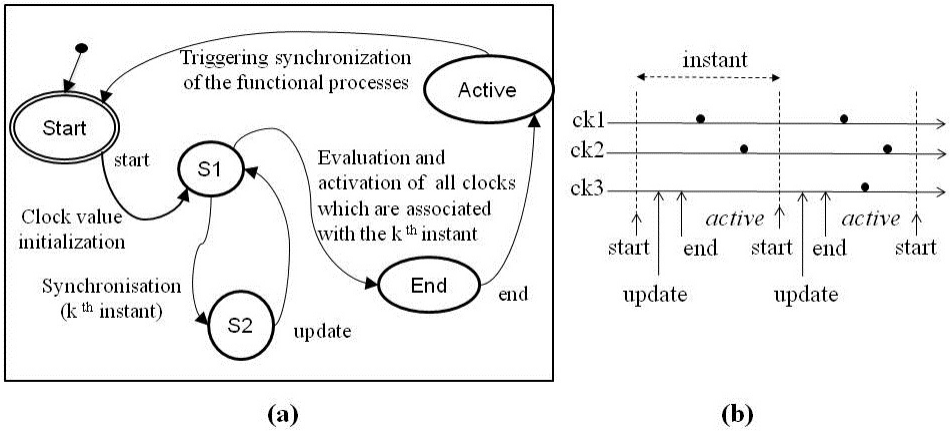

Figure 6.6: L'automate prosessus de l'ordonnanceur "Scheduler".

Chaque événement donne naissance à une horloge *clock* qui sera localisée par une structure Fiacre *tab_Clocks*. Cette structure est déclarée comme suit (Listing 3): D'un point de vue implantation Fiacre, et prenant en compte le principe d'implantation des programmes Promela décrit dans [YM11], chaque évènement dans le modèle est associé à une horloge (*clock*) qui est localisée dans une structure Fiacre *tab_Clocks*. Cette structure est déclarée comme suit :

```
type T_CLOCK is record clock_state:nat, enable_tick, dead: bool end
type T_ARRAY_CLOCK is array 7 of T_CLOCK
tab_Clocks: T_ARRAY_CLOCK
```

Listing 3. La déclaration FIACRE de la structure structure T_CLOCK.

A chaque itération de *Scheduler*, chaque processus de contrainte met à jour la valeur *clock_state* qui peut prendre les valeurs entières 0, 1 ou 2, en accord avec l'exécution de l'automate qui l'encode. Une fois que *Scheduler* a exécuté une itération, il évalue les valeurs *clock_state* pour mettre à jour les valeurs *enable_tic* à *true* ou $false$. Si *enable_tic* est évaluée à *true*, *Scheduler* synchronise le processus fonctionnel associé à cette horloge (par exemple $Sensor1$ est synchronisé avec le port $sync\_pw1$ comme montré dans la Figure.6.5). La valeur *enable_tic* est mis à *true* seulement si *clock_state* est égale à 2. Dans les autres cas, *enable_tic* est mis à $false$. La valeur *dead* est mise à *true* quand l'horloge associée ne doit plus être active dans la suite de l'execution.

La génération produit automatiquement le code de *Scheduler* incluant la partie executée durant la phase *Active* :

```
... if (tab_Clocks [0].enable_tick) then sync_pw1
elsif (tab_Clocks [1].enable_tick) then sync_pr1
elsif (tab_Clocks [2].enable_tick) then sync_pw2
elsif (tab_Clocks [3].enable_tick) then sync_pr2
elsif (tab_Clocks [4].enable_tick) then sync_comput
elsif (tab_Clocks [5].enable_tick) then sync_filter (true)
elsif (tab_Clocks [6].enable_tick) then sync_filter (false)
end ...
```

Listing 4. Extrait du code Fiacre de l'ordonnanceur *Scheduler* pour la synchronisation des processus fonctionnels.

D'un point de vue implantation en Fiacre, et reprennant le principe d'implantation de programmes Promela décrit dans [YM11], chaque évènement à prendre en compte dans le modèle donne lieu à une horloge *clock* qui est implantée par une structure Fiacre (*record*). Cette structure contient les champs suivants : *must_tic*, *cannot_tic*, *act_tic* et *dead*. A chaque boucle d'exécution du *Scheduler*, chaque processus de contrainte met à jour les valeurs *must_tic*, *cannot_tic* en fonction de l'exécution de l'automate qu'il encode. Une fois, les processus de contrainte ayant exécuté une boucle, *Scheduler* évalue ces valeurs

pour mettre la valeur *act_tic* à *true* ou à $false$. Si *act_tic* est évalué a *true*, le processus fonctionnel associé à l'évènement est synchronisé avec le *Scheduler*, ce qui déclenche un pas d'exécution dans le processus fonctionnel. L'évaluation de la valeur *act_tic* est régie par le tableau 5.1.

La valeur *dead* est mise à *true* lorsque l'hologe associée ne doit plus être active dans la suite de la simulation.

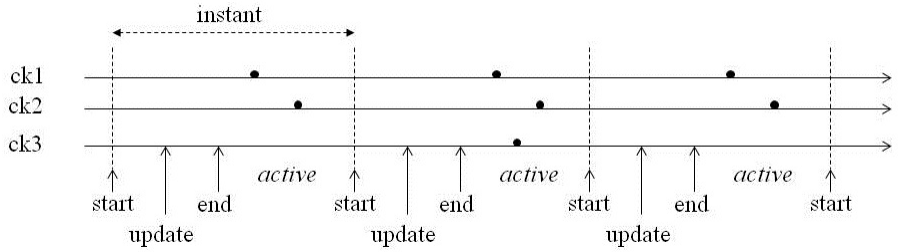

Figure 6.7: Séquencement du processus Scheduler

6.2.5 Génération de l'architecture Fiacre

La Figure. 6.8 illustre l'architecture FIACRE de notre cas d'étude, résulant de la transformation du modèle MARTE/ CCSL source.

Dans le cas d'étude, le générateur de code produit 12 processus : *Scheduler*, 5 processus de contrainte (2 pour $alternatesWith$, 2 pour *strictPrec*, 1 pour $filterBy$) et 6 processus fonctionnel ($Sensor1$, $Sensor2$, $Acq1$, $Acq2$, *Comput* and $Filter$). La Figure.6.8 montre que le processus "Scheduler" est connecté avec chaque processus fonctionnel via une communication synchrone avec port nommée *triggering communications*. *Scheduler* contrôle l'execution des processus fonctionnels et permet de donner un ordre explicit "*explicit rhythm*" pour l'execution des différents procesus. Les processus fonctionnels *Sensor1*, *Sensor2*, *Acq1*, *Acq2*, *Comput* et *Filter* sont respectivement synchronisés avec *Scheduler* via *sync_pw1*, *sync_pr1*, *sync_pw2*, *sync_pr2*, *sync_comput* et *sync_filter* ports. Par exemple, *Sensor1* et *Acq1* sont respectivement synchronisés par *Scheduler* pour les opération d'écriture et de lecture dans $M1$. Egalement, *Filter* est synchronisé avec with *Scheduler* via *sync_filter* pour l'opération de filtrage, d'où *sync_filter* contient une valeur booléenne.

De même, *Scheduler* est connecté avec tout les processus de contraintes déclarés via le tableau *Table tab_clock*partagé, dans le but de mettre à jour les valeurs des états d'horloges selon les états de contraintes afin de prendre une decision si une horloge donnée peut ou pas tiquer dans un instant spécifique.

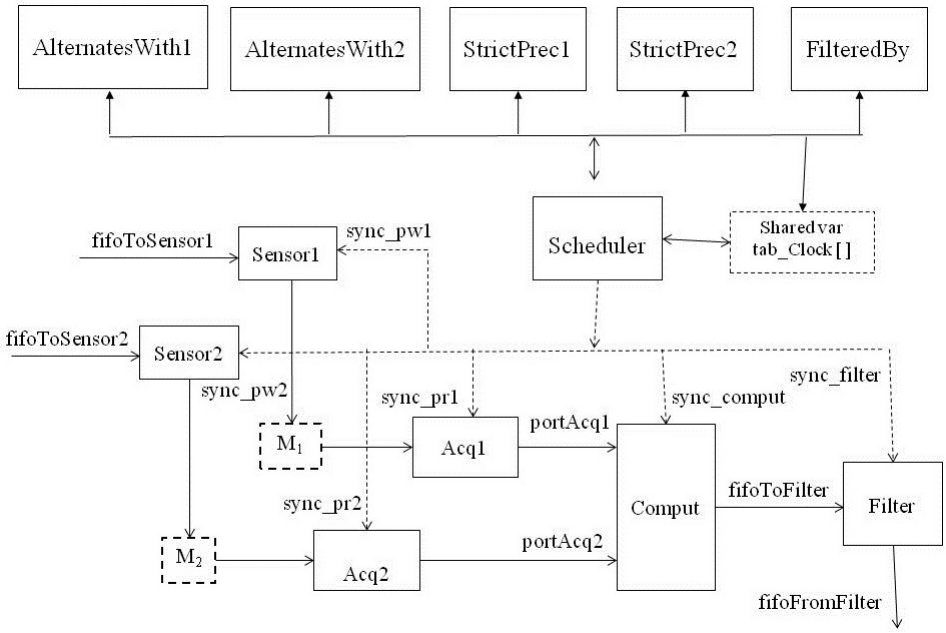

Figure 6.8: Structure du code Fiacre généré.

6.2.6 Architecture du code Fiacre

Nous décrivons ici l'organisation du programme Fiacre implémenté. Dans un modéle donné, en supposant n le nombre de contraintes et d le nombre d'hologes logiques associées à ces contraintes, nous instancions n processus Fiacre implantant les contraintes (un par contrainte), et déclarons d structures de type *clock*.

Le processus *Scheduler* dispose de ports de synchronisation avec les processus de contrainte : n ports de synchronisation de type *start*, n ports de type *update*, n ports de type *end*. Il disposera également de d ports de synchronisation pour chacune des horloges. Par exemple, pour le cas d'étude, en supposant que nous ne considérons que deux voies d'acquisition de données, le *Scheduler* est déclaré comme suit :

```
process Scheduler [startAlt1, startAlt2, startPrec1, startPrec2, startPrec3,
            startFilteredBy, updateAlt1, updateAlt2, updatePrec1,
            updatePrec2, updatePrec3, updateFilteredBy, endAlt1, endAlt2, endPrec1,
          endPrec2, endPrec3, endFilteredBy, pr1, pw1, pr2, pw2,
          comput, computOut, filter: out none ]
         (&fifo_fromEnv: read write t_fifo_External, &tab_Clocks : T_ARRAY_CLOCK) is ...
```

Les processus de contrainte sont déclarés comme suit :

```
process AlternatesWith [start, update, end : in none]
       (&c1 : nat, &c2 : nat, &tab_Clocks : T_ARRAY_CLOCK) is ...
```

```
process StrictPrecedence [start, update, end : in none]
     (&c1 : nat, &c2: nat, &tab_Clocks : T_ARRAY_CLOCK)  is ...
process FilteredBy [start, update, end : in none]
       (&c1 : nat, &c2: nat, &tab_Clocks : T_ARRAY_CLOCK)  is ...
```

Les processus fonctionnels sont déclarés comme suit :

```
process Sensor [sync_pw : in int]
       (&numSensor : read write int, &fifo_fromEnv : read write t_fifo_External,
        &Mem : read write T_ARRAY_INT) is ...
process Acq  [sync_pr : in int, port_Acq_Comput : out int]
      (&numSensor : read int, &Mem : read write T_ARRAY_INT) is ...
process Comput [sync_calcul: in int, sync_calculOut: in int,
        port_Acq_Comput1: in int, port_Acq_Comput2 : in int]
       (&fifo_toFilter : read write t_fifo_Internal) is ...
process Filter [sync_filter: in int]
      (&fifoFromComput: read write t_fifo_Internal, &fifo_toEnv: read write t_fifo_External) is ...
```

Ces processus fonctionnels sont synchronisés avec le *Scheduler* par des ports synchrones. Par exemple le *Sensor* est synchronisé sur le port *sync_pw* pour exécuter une écriture d'une donnée *data* dans la mémoire partagée *Mem* entre lui et processus *Acq*. Un extrait du code Fiacre de *Sensor* implantant cette opération d'écriture est montré ci-dessous :

```
from Start
             if (empty fifo_fromEnv) then loop end; // lecture d'une data dans la fifo d'entree
             data := first fifo_fromEnv;
             fifo_fromEnv := dequeue (fifo_fromEnv);
             to waitSynchro
from waitSynchro
             sync_pw;      // synchronisation avec Scheduler
             Mem := data;  // memorisation de data
             to Start
```

Génération du code au niveau

Le progamme Fiacre inclut un composant nommé C (cf. Listing 1) qui contient les instances des processus s'exécutant en parallèle (operateur ||). Les codes des processus fonctionnels, de contrainte et du *Scheduler* sont le résultat de l'exécution de notre algorithme de génération automatique.

Les procesus représentant les élements de MARTE, les contraintes CCSL, et les l'interprétation de ces contraintes sont finalement instantiés dans le composant Fiacre, nommé C, et spécifié comme une entité d'execution indépendante via l'opérateur || . Le scheduler, les processus de contraintes, les processus fonctionnels, sont synchronisés via des port de communication.

Le code Fiacre du composant C est partiellement généré dans le Listing 6[3]. Comme un résultat de la génération de notre algorithme, le code des processus fonctionnels et le *Scheduler* sont instantiés dans C.

Pour que la génération automatique de code soit possible, nous devons déclarer des attributs qui sont les numéros d'horloges (clockNo), les relations entre ces horloges et les signaux de synchronisation générés par *Scheduler*. Par exemple, l'horloge *read*1 est associée au port *sync_pr*1 pour synchroniser la première instance (*Acq:1*) du processus *Acq*. L'horloge $filter$ est associée au port $sync\_filter$ qui transporte une valeur booléenne. Pour l'implantation de cette dernière contrainte de filtrage, deux indices sont nécessaires pour chaque valeur booléenne.

Ces attributs sont spécifiés comme suit (Listing 5):

```
# Synchronization
write1: clockNo: 0, synchro: sync_pw1 none  to:  Sensor:1;
read1:  clockNo: 1, synchro: sync_pr1 none  to:  Acq:1;
write2: clockNo: 2, synchro: sync_pw2 none  to:  Sensor:2;
read2:  clockNo: 3, synchro: sync_pr2 none  to:  Acq:2;
comput: clockNo: 4, synchro: sync_comput none  to: Comput:1;
filterOut: clockNo: 5, synchro: sync_filter bool:true,
           clockNo: 6, synchro: sync_filter bool:false to: Filter:1;
```

Listing 5. Déclaration explicite des numéros d'horloges, les liens entres elles et les déclencheurs de synchronisation .

Avec ces attributs, le principe d'implantation et de synchronisation avec *Scheduler* est appliqué similairement aux deux autres processus fonctionnels *Acq* et $Filter$. Enfin, le composant est déclaré comme suit (Listing 6):

```
component C is
var  ckWrite1, ckRead1, ckWrite2, ckRead2, ckComput,
      ckFilterTrue, ckFilterFalse: int,
tab_Clocks: T_ARRAY_CLOCK, Mem: T_ARRAY_INT, Data_Enable: T_ARRAY_BOOL,
// Shared Fifos between process Comput and Filter
fifoComput_Filter: t_fifo_Internal,
// Shared Fifos with Environment
fifoToSensor_1: t_fifo_External,// from DEV1
fifoToSensor_2: t_fifo_External,// from DEV2
toContext: t_fifo_External,      // to DEVOUT
NumSensor1, NumSensor2: int
```

[3]Le code complet du cas d'étude peut être trouvé sur le site : http://www.obpcdl.org.

```
port  // Synchro ports for the constraints
startAlt1, startAlt2, startStrictPrec1, startStrictPrec2, startFilteredBy,
UpdateAlt1, UpdateAlt2, UpdateStrictPrec1, UpdateStrictPrec2,
UpdateFilteredBy,endAlt1, endAlt2, endStrictPrec1, endStrictPrec2,
endFilteredBy: none,
sync_pw1, sync_pr1, sync_pw2, sync_pr2, sync_filter, sync_comput,
  port_Acq_Comput1, port_Acq_Comput2: int
init
// Clock numbers
ckWrite1 := 0; ckRead1 := 1; ckWrite2 := 2; ckRead2 := 3; ckComput := 4;
ckFilterTrue := 5; ckFilterFalse := 6;
Mem := [0, 0]; // Shared data: memory between Acq and Sensor
// Shared Fifo initialization
fifoComput_Filter  := {||}; fifoToSensor_1 := {||};
fifoToSensor_2  := {||}; toContext      := {||};
NumSensor1 := 1; NumSensor2 := 2
par
//---------- Scheduler process ---------
Scheduler [startAlt1, startAlt2, startStrictPrec1, startStrictPrec2,
    startFilteredBy, endAlt1, endAlt2, endStrictPrec1, endStrictPrec2,
    endFilteredBy, UpdateAlt1, UpdateAlt2, UpdateStrictPrec1,
    UpdateStrictPrec2, UpdateFilteredBy, sync_pr1, sync_pw1, sync_pr2,
    sync_pw2, sync_comput, sync_filter] (&tab_Clocks)

//---------- CCSL constraint processes ----------
|| AlternatesWith [startAlt1,UpdateAlt1,endAlt1]
             (&ckWrite1,&ckRead1,&tab_Clocks)
|| AlternatesWith [startAlt2,UpdateAlt2,endAlt2]
             (&ckWrite2,&ckRead2,&tab_Clocks)
|| StrictPrecedence [startStrictPrec1, UpdateStrictPrec1, endStrictPrec1]
           (&ckRead1, &ckComput, &tab_Clocks)
|| StrictPrecedence [startStrictPrec2, UpdateStrictPrec2, endStrictPrec2]
           (&ckRead2, &ckComput, &tab_Clocks)
  || FilteredBy [startFilteredBy, UpdateFilteredBy, endFilteredBy]
           (&ckComput, &ckFilterTrue, &ckFilterFalse, &tab_Clocks)
//---------- Sensor processes ---------
|| Sensor [sync_pw1] (&NumSensor1, &fifoToSensor_1, &Mem)
|| Sensor [sync_pw2] (&NumSensor2, &fifoToSensor_2, &Mem)
//---------- Acq processes -------------
  ||Acq [sync_pr1, port_Acq_Comput1] (&NumSensor1, &Mem)
  ||Acq [sync_pr2, port_Acq_Comput2] (&NumSensor2, &Mem)
```

```
//---------- Comput and Filter processes ---------
|| Comput [sync_comput, port_Acq_Comput1, port_Acq_Comput2]
         (&fifoComput_Filter)
|| Filter [sync_filter](&fifoComput_Filter, &toContext)
end Comp
```

Listing 6. Programme du composant généré.

6.3 Conclusion

Notre objectif est de tirer parti des avantages de la vérification basée sur les contextes tout en partant de modèles temporisés en MARTE, considérés plus facile à définir que des modèles formels (e.g Fiacre). Par conséquent, l'approche que nous avons retenue consiste à transformer les modèles MARTE définis par les utilisateurs en modèles formels exploitables par les outils existants tels qu'OBP. Cependant, l'utilisation d'une telle transformation ne résout pas tout les problèmes. Il faut par exemple être capable de remonter les informations produites par les outils de vérification au niveau de UML MARTE de manière à ce qu'elles puissent être facilement interprétable par le concepteur. Nous présentons dans ce qui suit les principes de vérifications des exigences sur les modèles temporisés, ainsi que les expérimentations sur le cas d'étude.

Chapitre 7

Vérification formelle d'exigences

"*Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied maths is necessary part of the education of other engineers.*"

"The principles of model checking", by *Christel Baier* and *Joost-Pieter Katoen*. "The MIT press, Massachusetts Institue of Technology. Cambrigde, Massachusetts.

7.1 Introduction

7.1.1 Les principes de la vérification

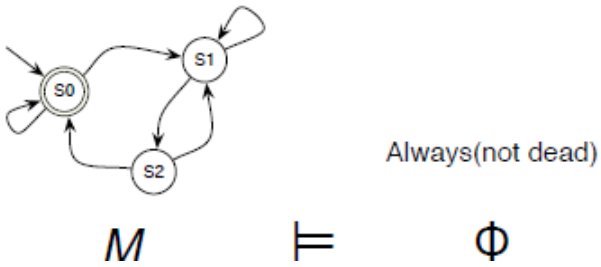

Figure 7.1: Le modèle M du Sytème S satisfait-il la propriété Φ

La Figure 7.1 montre le principe de la vérification formelle. Etant donné un modèle M du système S, et la spécification d'une propriété attendue Φ, $Msatisfait\text{-}il\Phi$?

Grâce à la vérification automatique, l'exploration exhaustive de l'espace d'état du système [CD03], nous permet de répondre de manière efficace à une telle question. Pour établir la vérification d'un ensemble d'exigences sur un modèle, il faut pouvoir, d'une part, l'explorer exhaustivement, et d'autre part disposer d'une expression formelle des propriétés à vérifier. Des propriétés de sûreté et de vivacité peuvent alors être vérifiées. Ces propriétés sont généralement exprimées sous une forme de logique modale, par exemple la logique temporelle linéaire (Linear-time Temporal Logic, abrégé LTL) [PNU77][PNU81], et la logique temporelle arborescente (Computation Tree Logic, abrégé CTL, CTL*) [eEAE81] [eJYH82] [eJYH86].

Dans l'approche décrite dans [YM11], l'expression des propriétés est basée sur la logique linéaire LTL [PNU77] et la vérification s'exécute avec l'outil SPIN[1]. Dans notre approche, nous choisissons [DBR$^+$12] d'exprimer les propriétés sous la forme d'automates observateurs. En effet, les formules de logiques linéaires peuvent parfois être difficiles à manipuler dans un contexte industriel. Une exigence peut référencer de nombreux évènements, liés à l'exécution du modèle ou de l'environnement, et peut être dépendante d'un historique d'exécution à prendre en compte au moment de sa vérification. Leur expression par des formules logiques demandent alors une grande expertise de la part des ingénieurs. L'écriture d'observateurs, basée sur le formalisme d'automate bien connu des ingénieurs, est par contre beaucoup plus aisée à manier.

Une fois les observateurs spécifiés, le modèle est ensuite exploré et l'exploration génère un *Système de Transitions (SdT)*. Celui-ci représente tous les comportements du modèle dans son environnement sous la forme d'un graphe de configurations et de transitions. Sur ce SdT, la vérification des propriétés est conduite en appliquant une analyse d'accessibilité

[1] http://spinroot.com

des états d'erreur des observateurs. La difficulté, bien connue, liée à cette technique d'analyse est la production du SdT qui peut être de grande taille, dépassant la taille mémoire disponible – connu par le phénomème d' "*explosion combinatoire*". Dans des travaux précédents, nous argumentons que l'explosion combinatoire peut être contenue dans certains cas applicatifs par l'exploitation de *contextes* formalisés qui permettent de restreindre le nombre de comportements explorés lors de la vérification. Cet aspect n'est pas détaillé dans ce mémoire mais peut être consulté dans [DBRL12, DBR$^+$12].

7.1.2 Le processus de vérification

Afin de vérifier les exigences pertinentes sur le modèle, il est primordial de l'explorer exhaustivement et de posséder des expressions formelles des propriétés à vérifier; par exemple, sous forme de formules logiques ou bien d'automates observateurs.

Une fois les observateurs sont bien spécifiés, le modèle est exploré et cette dernière génère un système de transition étiqueté : *Labeled Transition System (LTS)*. Ce dernier représente tous les comportements du modèle ainsi que son environnement sous forme d'un graphe de configurations et d'un ensemble de transitions. Sur ces LTL, la vérification des propriétés est faite par l'analyse d'attégnabilité des états d'erreurs d'observateurs.

7.2 Expression des propriétés en CDL

Le langage CDL permet à l'utilisateur d'exprimer des propriétés sous la forme de prédicats et d'observateurs représentés sous forme d'automates.
La déclaration des prédicats en CDL peut porter sur des valeurs de variables comme par exemple : $predicate\ pred1\ is\ Proc1 : v = value$,signifiant que $pred1$ est vrai si la variable v de l'instance 1 du processus $Proc$ est égale à la valeur $value$. Les prédicats peuvent porter aussi sur l'état des processus. Prenant l'exemple : $predicate\ pred2\ is\ Proc1@stateX$, qui signifie que $pred2$ est vrai si l'instance 1 du processus $Proc$ est dans l'état $stateX$. Les prédicats peuvent porter également sur le nombre de données contenues dans une $fifo$ ou sur une expression booléenne combinant les types de prédicats précédents. Ces possibilités fournissent un mode d'expressivité très riche qui permettent, en complémentarité avec les observateurs, d'exprimer aisemment des propriétés qu'il serait plus difficile à exprimer en logique linéaire. Elles permettent d'instrospecter en profondeur le comportement d'un modèle tout en offrant une expression facile à manier et à comprendre pour le concepteur.

Dans cet étude, nous exprimons des propriétés CDL en suivant deux objectifs complémen--taires: un pour vérifier que la mise en œuvre des contraintes CCSL est correcte, et l'autre pour s'assurer que les parties fonctionnelles du circuit ($Sensor1$, $Sensor2$, $Acq1$, $Acq2$, $Comput$, $Filter$) sont correctement mises en œuvre.

Propriétés associées aux constraintes CCSL :

Ici, nous illustrons l'écriture de propriétés associées aux contraintes CCSL incluses dans notre modèle.

Propriétés d'alternance

Pour vérifier les propriétés $P1a$ $P1b$ associées aux exigences d'alternance $Req1a$ et $Req1b$, décrites en section 5.2.5, nous déclarons les événements evt_write1, evt_read1, evt_write2 et evt_read2 comme suit :

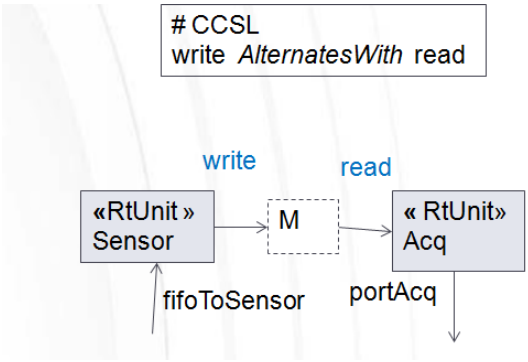

Figure 7.2: Représentation de la contrainte d'alternance selon notre cas d'étude

```
event evt_write1 is {sync sync_pw1 from {Scheduler}1 to {Sensor}1}
event evt_write2 is {sync sync_pw2 from {Scheduler}1 to {Sensor}2}
event evt_read1  is {sync sync_pr1 from {Scheduler}1 to {Acq}1}
event evt_read2  is {sync sync_pr2 from {Scheduler}1 to {Acq}2}
```

Avec ces événements, nous spécifions les observateurs, illustrés dans la Figure 7.3, encodant la propriété $P1a$ (resp. $P1b$) qui satisfait l'alternance de $write1$ et $read1$ (resp. $write2$ et $read2$). L'état initial de l'observateur $P1a$ (resp. $P1b$) est l'état $Start$ et a un état d'erreur ($reject$). Chaque transition de l'observateur est déclenchée par l'occurrence d'un événement evt_write1 ou evt_read1 (resp. evt_write2 ou evt_read2).

Si nous voulons vérifier que les horloges $write1$ et $read1$ (resp. $write2$ et $read2$) ne "tiquent" pas dans un même instant, nous pouvons déclarer les prédicats suivants:

```
predicate act_tick_pr1_true is {{MyCircuit}1:tab_Clocks [CLK_read1].act_tick = true}
predicate act_tick_pw1_true is {{MyCircuit}1:tab_Clocks [CLK_write1].act_tick = true}
predicate act_tick_rw1_together is {act_tick_pw1_true and act_tick_pr1_true}

predicate act_tick_pr2_true is {{MyCircuit}1:tab_Clocks [CLK_read2].act_tick = true}
predicate act_tick_pw2_true is {{MyCircuit}1:tab_Clocks [CLK_write2].act_tick = true}
predicate act_tick_rw2_together is {act_tick_pw2_true and act_tick_pr2_true}
```

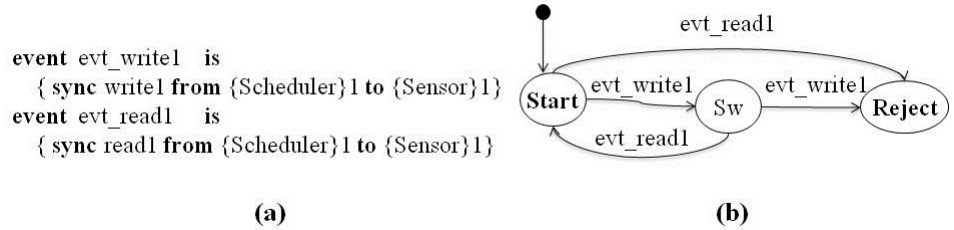

Figure 7.3: Automate observateur correspondant aux propriétés P1a et P1b.

Nous pouvons maintenant déclarer, avec l'opérateur *assert*, les invariants suivants: *not act_tick_rw*1*_together* et *not act_tick_rw*2*_together*. Au cours de l'exploration du modèle, l'outil OBP vérifie que les invariants ne sont pas violés.

```
assert not act_tick_rw1_together
assert not act_tick_rw2_together
```

Propriétés de précédence

De la même manière, nous pouvons spécifier des observateurs pour vérifier les propriétés des exigences *Req*2*a* et *Req*2*b*. Nous déclarons l'événement *evt_comput* :

```
event evt_comput is  {sync sync_comput from {Scheduler}1 to {Comput}1}
```

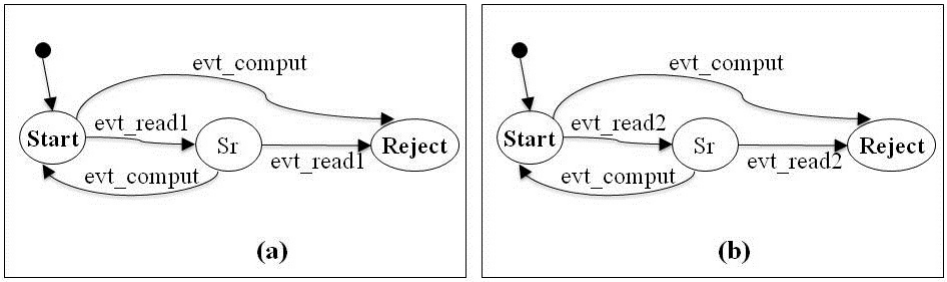

Figure 7.4: Automates observateur correspondants aux propriétés P2a etP2b.

Propriété de filtrage

L'exigence *Req*3 associée à la production de données par *Comput* et la contrainte de filtrage est exprimé par le terme CCSL: $filterOut = comput\ FilteredBy\ (001)^w$. Au cours de l'exploration, nous devons vérifier que la séquence de données générées à partir de *Filter* est la séquence générée par *Comput* avec un échantillon d'une valeur sur 3. Dans la version actuelle du modèle, le mot filtre (001) est stocké dans un tableau

variable $tabFilter$ du processus de contrainte $FilteredBy$. Les $(i \ modulo \ 3)^{emes}$ données de la séquence générée par $Comput$ seront copiées dans la séquence issue de $Filter$ si la valeur $tabFilter[i \ modulo \ 3]$ est égale à 1. Sinon, elle n'est pas copiée dans la séquence des données fournies à l'environnement. Pour vérifier cette contrainte, nous déclarons donc les prédicats suivants (pour $x \in \{0, 1, 2\}$) :

```
predicate bitx_true  is {{FilteredBy}1:tabFilter[x] = 1}
predicate bitx_false is {{FilteredBy}1:tabFilter[x] = 0}
```

Les transitions d'un observateur peuvent être décorées avec un des prédicats conjointement avec les événements evt_comput, $evt\_filterTrue$ et $evt\_filterFalse$ qui déclenchent les transitions de l'observateur. Ces eux derniers événements sont déclarés comme suit:

```
event evt_filterTrue  is {sync filter (true)  from {Scheduler}1 to {Filter}1}
event evt_filterFalse is {sync filter (false) from {Scheduler}1 to {Filter}1}
```

La Figure 7.5 illustre l'observateur encodant la propriété $P3$ et référençant les prédicats et les événements ci-dessus. Il est à noter que la syntaxe CDL permet l'expression de ce type de propriétés qu'il serait difficile d'exprimer en logique linéaire.

Si nous voulons vérifier d'autres propriétés sur les parties fonctionnelles de notre modèle, nous spécifions ces propriétés qui caractérisent le comportement du modèle. Par exemple, l'exigence $Req4$, exprimée dans la section 5.4 peut être exprimée par un automate observateur utilisant des prédicats et des événements appropriés.

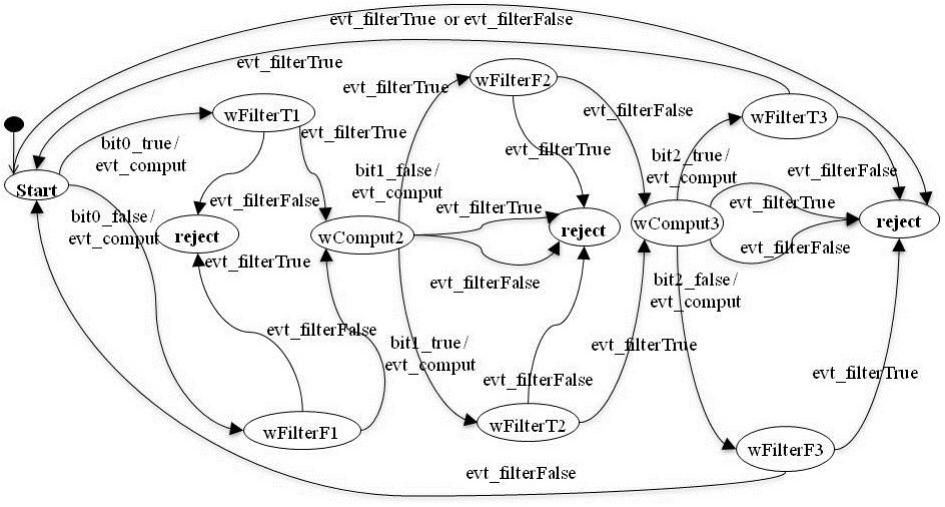

Figure 7.5: Automate observateur correspondant à la propriété P3.

Le code CDL de la propriété P3 est indiqué ci-dessous.

```
\begin{verbatim}
property  pty_FilteredBy  is {
// bit 0 in the filter word
   start        --  / bit0_true  / evt_comput      /  -> wFilterT1;
   start        --  / bit0_false / evt_comput      /  -> wFilterF1;
   wFilterT1    --  /            / evt_filterTrue  /  -> wComput2;
   wFilterF1    --  /            / evt_filterFalse /  -> wComput2;

// bit 1 in the filter word
   wComput2     --  / bit1_true     / evt_comput      /  -> wFilterT2;
   wComput2     --  / bit2_false    / evt_comput      /  -> wFilterF2;
   wFilterT2    --  /               / evt_filterTrue  /  -> wComput3;
   wFilterF2    --  /               / evt_filterFalse /  -> wComput3;

// bit 2 in the filter word
   wComput3     --  / bit2_true  / evt_comput      /  -> wFilterT3;
   wComput3     --  / bit2_false / evt_comput      /  -> wFilterF3;
   wFilterT3    --  /            / evt_filterTrue  /  -> start;
     wFilterF3    --  /            / evt_filterFalse /  -> start;

   // --------Errors -----------
    start       --  / / evt_filterTrue  /    -> reject;
   start        --  / / evt_filterFalse /    -> reject;
   wComput2     --  / / evt_filterTrue  /    -> reject;
   wComput2     --  / / evt_filterFalse /    -> reject;
   wComput3     --  / / evt_filterTrue  /    -> reject;
   wComput3     --  / / evt_filterFalse /    -> reject;
   wFilterT1    --  / / evt_filterFalse /    -> reject;
   wFilterF1    --  / / evt_filterTrue  /    -> reject;
   wFilterT2    --  / / evt_filterFalse /    -> reject;
   wFilterF2    --  / / evt_filterTrue  /    -> reject;
   wFilterT3    --  / / evt_filterFalse /    -> reject;
   wFilterF3    --  / / evt_filterTrue  /    -> reject
}
```

Propriétés associées aux contraintes CCSL

Nous illustrons ici l'écriture de quelques propriétés associées aux contraintes CCSL intégrées à notre modèle de circuit.

Pour vérifier une des propriétés $P1$ associée à l'exigence d'alternance $Req1$, par exemple $read_1$ $alternatesWith$ $write_1$, nous déclarons (voir Figure 7.3(a)) les évènements CDL evt_write1 et evt_read1. Avec ces évènements, nous spécifions l'observateur, illustré dans la Figure 7.3(b), encodant la propriété $P1$ qui vérifie l'alternance des synchronisations $write_1$ et $read_1$. L'état initial de l'observateur est l'état $Start$ et dispose d'un état d'erreur $Reject$. Chaque transition de l'observateur est déclenchée par l'occurence d'un évènement (evt_write1 ou evt_read1).

D'une manière similaire, nous pouvons spécifier des observateurs pour vérifier les pro-

priétés associées aux exigences $Req2$ et $Req3$ en déclarant les évènements evt_read_2, evt_comput, $evt\_computOut$.

```
event evt_read2  is  { sync read1 from {Scheduler}1 to {Sensor}2}
event evt_comput  is  { sync comput from {Scheduler}1 to {Comput}1}
event evt_cOut      is  { sync computOut from {Scheduler}1 to {Comput}1}
```

Le langage CDL permet également de spécifier des prédicats qui peuvent être vérifiés lors de l'exploration du modèle. Par exemple, si on souhaite vérifier que, dans un instant, les horloges $write_1$ et $read_1$ ne "tiquent" pas dans le même instant, nous pouvons déclarer les prédicats suivants :

```
predicate act_tick_CLK_pr1_true is {{MyCircuit}1:tab_Clocks [0].act_tick = true}
predicate act_tick_CLK_pw1_true is {{MyCircuit}1:tab_Clocks [1].act_tick = true}
predicate act_tick_rw1_together is {act_tick_CLK_pw1_true and act_tick_CLK_pr1_true}
```

Nous pouvons maintenant déclarer, dans une clause *cdl* du programme CDL et avec l'opérateur $assert$[2] l'invariant suivant : $not\ act\_tick\_rw1\_together$. L'outil OBP vérifie, lors de l'exploration du modèle, que l'invariant n'est pas violé.

Les prédicats CDL permettent aussi de faciliter l'écriture d'observateurs plus complexes quand ils référencent, par exemple, un grand nombre d'évènements. Par exemple, l'exigence $Req4$ associée à la contrainte de filtrage des données générées par $Comput$, s'exprime en CCSL par l'expression : $filterOut = computOut\ FilteredBy\ (001)^w$. Lors de l'exploration, nous devons vérifier que la séquence de données issues du filtre $Filter$ est la séquence générée par $Comput$ avec un échantillonnage d'une valeur sur 3. Dans la version actuelle du modèle, le mot de filtrage (001) est mémorisé dans une variable tableau $tabFilter$ du processus contrainte $FilteredBy$. La $(i\ modulo\ 3)^{eme}$ donnée de la séquence issue de $Comput$ sera recopiée dans la séquence issue de $Filter$ si la valeur $tabFilter[i\ modulo\ 3]$ est égale à 1. Sinon, elle n'est pas recopiée dans la séquence de données fournies à l'environnement. Pour vérifier cette contrainte, nous déclarons donc les 6 prédicats suivants (pour $i \in \{0, 1, 2\}$) :

```
predicate biti_true  is {{FilteredBy}1:tabFilter [i] = 1}
predicate biti_false  is {{FilteredBy}1:tabFilter [i] = 0}
```

Les transitions d'un observateur peuvent être décorées d'un des 6 prédicats coinjointement à l'évènement (par exemple evt_cOut) qui déclenche la transition. La Figure 7.5 illustre l'observateur encodant la propriété $P3$. et référençant les prédicats ci-dessus.

[2]Voir la syntaxe détaillée du langage CDL disponible sur http://www.obpcdl.org.

Propriétés associées aux parties fonctionnelles

Pour vérifier les parties fonctionnelles de notre modèle, nous spécifions des propriétés caractérisant le comportement du circuit modélisé. Par exemple, l'exigence $req5$ exprimée en section 5.4 peut s'exprimer par un automate observateur en utilisant les prédicats et les évènements adéquats.

7.2.1 La spécification des contextes avec le langage CDL

Pour compléter la spécification du modèle CDL, on décrit les interactions entre le système et les devices Dev1, Dev2 and DevOut. Leurs comportements est décrit comme expliqué dans [DBRL12, DBR11].

A partir des diagrammes du modèle CDL, OBP génére un ensemble de graphes de contexte acyclique. Actuellement, chaque graphe est transformé en un automate Fiacre . Cela représente tous les intéractions entre le modèle et l'environnement. Afin de valider le modèle, il est nécéssaire de traiter chaque modèle graphique. Chaque propriété référencée en modèle CDL doit être vérifier sur les résultats de ces compositions.

Par exemple, si deux devices Dev1, Dev2 supposés transmettre 3 valeurs, on décrit ces intéractions avec CDL par (*event*) comme suit:

```
  event evt_send_data1_sensor1 is {send DATA1 to {Sensor}1};
  event evt_send_data2_sensor1 is {send DATA2 to {Sensor}1};
  event evt_send_data3_sensor1 is {send DATA3 to {Sensor}1};
  event evt_send_data1_sensor2 is {send DATA1 to {Sensor}2};
  event evt_send_data2_sensor2 is {send DATA2 to {Sensor}2};
  event evt_send_data3_sensor2 is {send DATA3 to {Sensor}2};
```

Ces évènements permettent au comportements des devices Dev1, Dev2 à être spécicifiés avec des activités "*activities*" comme suit:

```
activity Dev1 is {
  event evt_send_data1_sensor1;
  event evt_send_data2_sensor1;
  event evt_send_data3_sensor1
 }
 activity Dev2 is {
  event evt_send_data1_sensor2;
  event evt_send_data2_sensor2;
  event evt_send_data3_sensor2
 }
```

Le comportement de DevOut que reçoit les trois valeurs du processus *Filter* est spécifié de la manière suivante :

```
event evt_recv_dataOut is {receive ANY from {Filter}1 to {env}1};

activity DevOut is {
   loop 3 { event evt_recv_dataOut }
}
```

Le contexte est finalement décrit de la manière suivante :

```
cdl cdl_2dev is {
 properties P1a, P1b, P2a, P2b, pty_FilteredBy
 assert  not act_tick_rw1_together;
 assert  not act_tick_rw2_together

 main is { Dev1 || Dev2 || DevOut }
```

Il est spécifié que l'environnement est composé de 3 devices Dev1, Dev2 et DevOut. Durant l'exploration par OBP, les propriétés $P1a$, $P1b$, $P2a$, $P2b$ $pty\_FilteredBy$, $not\ act\_tick\_rw1\_together$ and $not\ act\_tick\_rw2\_together$ seront vérifiées.

7.3 Experimentation sur le cas d'étude et discussion

7.3.1 Outillage OBP (Observer-Based Prover) pour le model-checking

Pour mener les expérimentations, nous mettons en œuvre l'outil OBP (Figure 7.6). OBP est architecturé en 3 modules. Le *front end* OBP importe les modèles Fiacre provenant d'une traduction des spécifications UML-MARTE et des spécification CCSL. Il importe également les modèles CDL décrivant les propriétés et les scénarios de contextes si besoin. *OBP Explorer* explore le modèle et, après chaque transition du modèle exécuté, laisse la main au dispositif d'observation (*Observation Engine*). Celui-ci capte les occurences d'évènements et évalue, à chaque pas d'exécution du modèle, la valeur des prédicats et l'état de tous les observateurs impliqués. Une vérification des invariants et une analyse d'accessibilité des états d'erreurs des observateurs est donc ainsi menée.

A partir des diagrammes d'un modèle CDL, OBP génère un ensemble des graphes acycliques de contexte. Actuellement, chaque graphe est transformé en un automate Fiacre. Ceux-ci représentent l'ensemble des interactions possibles entre le modèle et l'environnement. Pour valider le modèle, il est nécessaire de composer chaque graphe avec le modèle. Chaque propriété référencée dans le modèle CDL doit être vérifiée sur le résultat de cette composition.

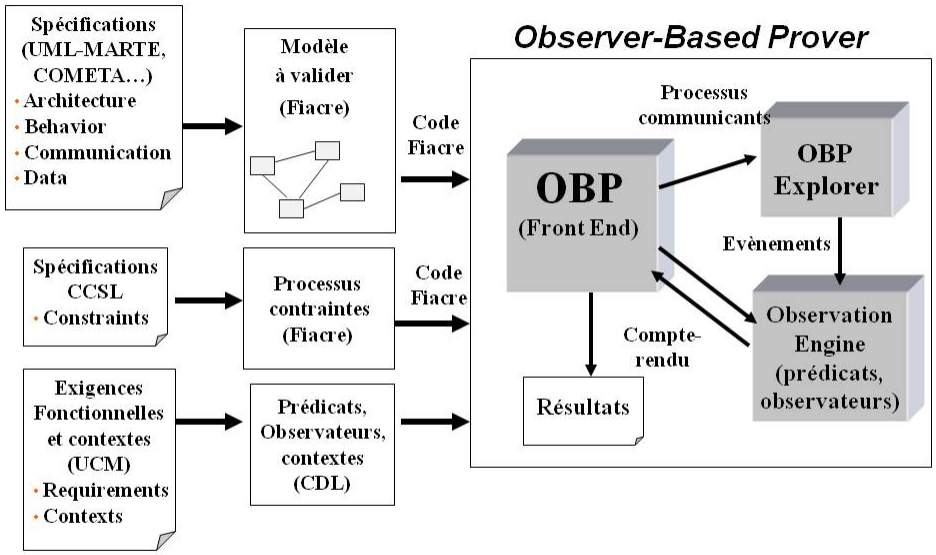

Figure 7.6: Architecture de l'outil OBP.

En fin d'exploration, un compte rendu est généré par OBP renseignant sur l'évaluation des propriétés et en fournissant des contre-exemples à la demande sur des états de *reject* ou de *success* des observateurs ou de violations des invariants. Ces indications peuvent aiguiller l'utilisateur sur le scénario ayant mis en échec les propriétés. Des travaux en cours visent à obtenir des facilités pour restituer des données de plus haut niveau dans le modèle de l'utilisateur, lui permettant de faciliter son diagnostic.

7.3.2 Partie 1: Résutats des vérifications avec l'outil OBP

Nous indiquons ici la complexité de l'exploration dans le cas d'étude proposé[3]. En guise d'exemple, nous supposons que l'environnement est composé de 2 voies (cf Table 7.1). Chaque voie délivre 5 valeurs (de 1 à 5) au circuit. Le nombre de configurations est alors de 77 828 et le nombre de transitions est de 285 187. Le temps d'exploration est de 7,8 secondes. En supposant maintenant que l'environnement est composé de 3 voies. Chaque voie délivre 5 valeurs (de 1 à 5) au circuit. Le nombre de configurations est alors de 942 480 et le nombre de transitions est de 4 320 996. Le temps d'exploration est de 109,4 secondes.

[3]Les tests sont exécutés sur une machine type Windows 7, 64 bits - 4 Go de mémoire RAM, avec OBP vers.1.3.4.

Table 7.1: Complexité avec 2 et 3 devices, avec variation de nombre de valeurs reçues de l'environnement (RAM 4GO)

Splitting	Nbr de voies	Nbr of valeurs	LTL config	LTL trans	temps	sous-contexte
3*Sans splitting	2	5	77 828	285 187	7,8	-
	3	5	942 480	4 32 996	109,4	-
	3	12	-	Explosion	-	-
1*Avec splitting	3	12	7 631 664	35 552 788	-	4

Si on augmente le nombre de voies ou le nombre de valeurs transmises différentes sur chaque voie, nous constatons une explosion du nombre de comportements du modèle. Dans ce cas, l'analyse des propriétés ne peut pas être conduite à son terme. Nous pouvons exploiter la capacité d'OBP à partitionner les scénarios d'interactions de l'environnement avec le modèle. Nous mettons alors en oeuvre la technique de *splitting* comme décrit dans [DBR11]. Cette technique permet d'explorer des modèles ayant un nombre plus important d'états parcourus. Par exemple, dans le cas de 3 voies avec une transmission de 12 valeurs différentes sur chaque voie, nous constatons que l'exploration ne termine pas. En appliquant alors le partitionnement des contextes, nous obtenons les résultats suivants : Le comportement de l'environnement est partionné en 4 sous-contextes. Le nombre de configurations explorées (en cumulé) est 7 631 664, le nombre de transitions est 35 552 788.

Ces résultats nous montre que, lorsque nous pouvons contenir l'explosion combinatoire intrasèquement associée à la technique d'exploration des modèles, l'outil OBP peut fournir une évaluation de la correction des propriétés, même sur une machine possédant une mémoire de taille limitée (4 Go). Si le modèle est d'une taille, en nombre de comportements, compatible avec une exploration exhaustive, le résultat sur les propriétés peut être obtenu sans utiliser la technique de partitionnement implanté dans OBP. Si le circuit est connnecté à un environnement qui interagit avec lui, le partitionnement peut alors être utilisé. L'environnement est, dans ce cas, spécifié par un ensemble de scénarios qui donne lieu lors du partitionnement à la génération automatique d'un ensemble de sous-contextes. La vérification est ensuite exécutée par un ensemble de vérifications, chacune associée à un sous-contexte. Mais ce n'est pas toujours le cas si le circuit présente intrasèquement un grand nombre de comportements sans pouvoir interagir avec un environnement. Dans ce cas, la méthode ne peut pas s'appliquer et il faut donc soit disposer d'une machine avec une mémoire plus importante, soit se focaliser sur des parties du modèle.

Table 7.2: Complexité avec 3 devices et 16 valeurs reçues de l'environnement

Splitting	Taille de fifo	Nbr of valeurs	LTL configurations	LTL transitions	souscontexte
3*Sans splitting	1	16	744,592	3,295,261	-
	2	16	3,328,269	17,797,040	-
	3	16	-	Explosion	-
1*Avec splitting	3	16	27,564,280	159,993,196	4

7.3.3 Partie 2 : Résultats de l'exploration avec la technique de partitionnement de scénarios

Nous avons explorer le modèle d'exigences du circuit en utilisant la méthode du model checking par OBP Explorer.

Avec CDL, on a spécifié des observateurs et des invariants pour appliquer la vérification des différentes propriétés concernant les exigences ($Req1a$ à $Req4$) exprimées en Section 5.4.

Les résultats sont illustrés dans le Tableau 7.2 et 7.3. Ils montrent le nombre de configurations LTS et de transitions générées durant l'exploration par OBP.

Le Tableau 7.2 presente les résultats du modèle de Circuit avec trois devices, en augmentant la taille de la file partagée entre l'environnement et le composant *Sensor*. Pour ce cas d'étude, la compléxité d'exploration [4] est d'une taille raisonable.

Comme un exemple, si on suppose que la taille de la file est égale à 1, le nombre de configurations explorées est donc 744 592 et le nombre de transition est de 3 295 261.

Avec 16 valeurs et une taille de file egale à 3, on remarque une explosion d'états à cause de la limitaion d'espace mémoire. Si on augmente le nombre de voies ou bien le nombre des différente valeurs transmises pour chaque voies, on remarque une explosion dans le nombre de comportements. Dans ce cas, l'analyse des propriétés ne peut être complétée.

Dans ce cas, on peut exploiter la capacité de l'outils OBP, pour partitionner les intéractions des scénarios de l'environnement avec le modèle. Cette technique permet au modèles possèdant un grand nombre d'états d'être exploités. Par exemple, dans le cas de trois voies avec la transmission de 16 differentes valeurs, on trouve que l'exploration ne se termine pas. Ainsi, après l'application du partitionnement des contextes, on obtient les resultats suivants: Le comportement de l'environnement est partitionné en quatres sous contextes. Le nombre de configurations explorés (cumulatives) est de 27 564 280, ainsi que le nombre de transitions est de 159 993 196.

Ces résultats nous montrent quand est ce que on pourra tomber dans le cas d'explosion combinatoire intrinsèquement associée avec des modèles de technique d'exploration, l'outil

[4]Les tests sont exécutés sous une machine machine Windows 32-bit - 10 GB RAM avec OBP vers.1.4.5

Table 7.3: Compléxité avec 6 devices, taille de fifo egale à 3, et 3 valeurs reçues de l'environnement

Splitting	LTL configurations	LTL transitions	sous-context
no	Explosion	-	-
yes	77 225 206	607 639 474	7

OBP peut fournir une vérification de correction de propriétés, même avec une machine avec une mémoire limitée de taille (10 GB). Si le modèle a un nombre de comportement compatible avec la recherche exhaustive, les résultats de vérification peuvent être obtenus sans utiliser la technque de partitionnement implémentée en OBP. Dans le cas ou le circuit est connecté avec l'environnement interagissant le partitionnement peut être utilisé.

Dans le présent cas, l'environnement est spécifié par l'ensemble de scénarios résultants à la génération de l'ensemble de sous contexte, durant le partitionnement automatique. La vérification est donc accomplie par l'ensemble de vérification des sous-contextes. Cependant, cela n'est pas toujours la cas si le circuit a un grand nombre de comportements qui n'intéragissent pas avec l'environnement. Dans ce cas de figure, la méthode ne peut être appliquée. Même s'il est nécéssaire d'avoir une machine d'espace mémoire plus grand, et de prendre en compte juste une partie du modèle.

L'exploration débute par la création de synchronisation entre le modèle et le système, les contextes ainsi que les automates observateurs. Si ces automates contiennent différentes locations et différentes horloges, prenons en compte l'observateur comme une entrée de production de la synchronisation peut augmenter significativement le nombre d'états et les transitions explorées.

En effet, les propriétés sont souvent reliées à un cas d'étude spécifique (comme par exemple : l'initialisation, la reconfiguration, et les modes dégradés). Cependant, il est pas nécéssaire pour une propriétés donnée de prendre en considération tous les possibilités du comportement de l'environnement, mais seulement une sous partie concernée de la vérification. De plus , la description du contexte permet une première limitation de l'espace exploré et par conséquent une première réduction de l'explosion combinatoire.

7.4 Version standalone

Nous avons spécifié, sous la forme d'observateurs CDL, les exigences $Req1$ à $Req4$, exprimées dans la section 5.4. La complexité de l'exploration[5] est de taille raisonnable. Par exemple, pour 2 canaux d'acquisition de données, chaque canal fournissant 16 valeurs (1 à

[5]Les tests sont effectués sur une machine Windows 7, 64 bits - 4 Go RAM avec OBP v.1.4.5

16) pour le circuit, la taille des fifo étant de 1, le nombre de configurations générées lors de l'exploration du modèle est alors de 159 404 et le nombre de transitions est de 589 607.

Pour 42 valeurs émises, nous constatons une explosion du graphes exploré. En appliquant le splitting, avec 3 sous-contextes générés, nous obtenons un nombre de configurations de 1 270 634 et 5 543 171 transitions.

Pour montrer la correction de notre modèle Fiacre généré, nous tenons le raisonnement suivant : Le comportement des processus Fiacre ne dépend pas des valeurs acquises en entrée par les processus *Sensor*. Nous transformons notre modèle en intégrant les 2 acteurs $Dev1$ et $Dev2$. dans le modèle comme des processus Fiacre. Ces processus itèrent tous les deux sur l'envoi de la même valeur. La compléxite de l'exploration du modèle est alors de 45 328 configurations et 168 664 transitions. Nous pouvons vérifier ainsi les propriétés avec les observateurs décrits précédemment.

7.5 Discussion et conclusion

L'idée de l'exprimentation de la version *Standalone* est la suivante : Nous montrons que si les valeurs sont émises avec une valeur constante, et en nombre de fois infini, le graphe d'exploration des configurations est fini. Or le comportement du circuit est indépendant de la valeur des données d'entrées. Ceci prouve que nous pouvons sur ce graphe vérifier toutes les propriétés qui sont exprimées indépendamment des valeurs de données.

Maintenant, la question qui se pose : " *Est ce que ce qui a été prouvé dans notre cas pour l'instant reste applicable pour le nombre de valeurs égales à 16?*". Dans la partie précédente, on affirme que la correction des propriétés est toujours vérifiée pour tous les nombres naturels envoyés par des équipements donnés. Pour démontrer la correction du code Fiacre généré, on prend en compte le raisonnement suivant : Le comportement des processus Fiacre ne dépend pas de la valeur aquise par les processus *Sensor*.

Pour atteindre cet objectif, on introduit une version du code *standalone* pour le but d'indiquer que le comportement du modèle de circuit impliqué n'est pas échangeable quelque soit les valeurs envoyées de l'environnement.

On a transformé notre modèle par l'intégration des deux acteurs, Dev1 Dev2 dans le modèle Fiacre sous forme de processus. Les deux processus itèrent sur l'envoie de la même valeur. La compléxité de l'exploration du modèle est donc de 45 328 configuations et 168 664 transitions. Donc, les propriétés peuvent être vérifiées par des observateurs et des invariants décrits au paravant.

Part IV

Conclusion et Perspectives

Chapitre 8

Conclusion et Perspectives

"*Research is formalized curiosity. It is poking and prying with a purpose.*"

Zora Neale Hurston

8.1 Bilan et discussion

Nous avons présenté dans cette thèse une démarche pour l'intégration des méthodes formelles aux démarches de développement suivant les bases de l'ingénierie dirigée par les modèles, pour la la vérification formelle de SETR. Notre but est de contribuer à une meilleure fiabilité des systèmes. Plus spécifiquement, en renforçant la qualité de leurs spécifications comportementales en termes de contraintes de temps logique.

Le contexte de cette thèse concerne le développement des systèmes embarqués et réactifs. Dans ce domaine, les architectures logicielles doivent être conçues pour assurer des fonctions critiques soumises à des contraintes très fortes en termes de fiabilité et de performances temps réel. Les exigences à respecter lors de la modélisation des logiciels concernent non seulement la correction et le déterminisme sur le plan fonctionnel mais aussi sur le plan temporel.

L'utilisation des méthodes formelles, avec le model checking est d'autant plus efficace qu'elle est mise en oeuvre au plus tôt dans le développement des systèmes. Le problème majeur liée a l'utilisation des techniques de vérification formelle avec le model checking provient du problème d'explosion combinatoire du nombre du comportement possible de modèle, ce dernier est provoqué par la compléxité interne du logiciel à vérifier. La deuxième difficulté est liée à l 'expression formelle des propriétés à vérifier.

Dans le but de permettre l'analyse de ces modèles de temps, par des techniques formelles, de modèles exprimés dans un language semi-formel, nous avons introduit dans cette thèse une chaine de transformation prenant un sous ensemble d'UML MARTE annoté avec des exigences de temps logique, et ciblant le language Fiacre en sortie. À travers cette thèse, nous avons abordé le domaine de la transformation de modèle. Notre travail se repose sur deux idées, la première est réduire l 'écart entre les modèles semi formelle exprimés en MARTE -CCSL, la deuxième est de proposer une description des propriétes temporelles à vérifier exploitable lors de la vérification automatique.

Le point de départ de notre transformation est des modèles exprimés en utilisant le profile UML MARTE. La conception d'un profile UML demande d'une part une grande maitrise de spécifications UML; et d'une autre part, une très bonne connaissance en métamodélisation. Les exigences imposées par le standard UML n'autorisent pas l'ajout de tout les aspects formels, pour cela des sémantiques ont été éxprimées en langage natuel. Dans ce contexte, certaines propositions ont été lancé pour les integrer dans le profile de temps, dans l'espoir qu'elles devienent standarisées (e.g., l'association explicite des hologes à des élément du modèle). Les informations correspondantes au temps "attachées aux élements du modèle" sont considérées avoir une sémantique profonde dans le sens de la néssecité de pouvoir exprimer temps et concurrence dans la sémantique.

Le sous profile "*Time*" de MARTE, étend largement les versions précédentes de modélisa--tion de temps (e.g SimpleTime de MARTE). En effet, il permet de faire référence explicitement à plusieurs référentiels de temps, par le biais des concepts d'horloges (stéréotype

Clock). Le profil MARTE permet ainsi d'exprimer des différences dans le perception du temps qui peuvent être caractérisées par des propriétés attachées aux horloges (e.g., dérive, decalage, stabilité. etc). Pour le besoin d'offrir des facilités d'utilisation de concepts temporels, un langage d'expréssion de contraintes d'horloges a été fournis sous forme de "*langage spécialisés*", permettant d'exprimer d'une manière concise des dépendances entre différents instants d'horloges.

Nous avons ainsi mis en place une démarche articulée en trois phases :

- (i) proposer et formaliser des transformations de modèles qui prennent en compte un sous-ensemble d'UML, suffisant et pertinent, dotés d'un pouvoir d'expressivité suffisant pour le domaine des logiciels embarqués,
- (ii) implanter et intégrer ces transformations dans un outillage basé sur l'outil de preuve OBP ,
- (iii) identifier une méthodologie claire et précise à proposer aux industriels pour rendre opérationnelle la chaîne de transformation réalisée.

Pour répondre à ces objectifs pré-cités, nous nous somme concentrer alors sur plusieurs axes de travail :

- Identification du périmètre UML-MARTE, choix de la sémantique d'interpétation des modèles, La définition du périmètre UML-MARTE s'appuie sur des travaux précédents qu'a mené l'Ensta Bretagne.

  Nous selectionnons des parties de MARTE decrivant du comportement temporelle du systeme à l aide de declatation d'un ensemble d horloges logiques, les communications, les composants (RtUnit, PpUnit), les données, ainsi que l'architecture.

- Conception, formalisation et implantation des règles de transformation vers Fiacre. S'inspirant de travaux déjà effectués dans l'équipe sur les techniques de transformation de modèle UML vers Fiacre [Hen12], l'objectif est de concevoir des transformations qui ciblent le langage Fiacre. Nous améliorons la méthode de transformation proposée par [Hen12] en les dotant d'un caractère générique. En effet, le travail proposé n'est adapté que pour les modèles issus d'un modeleur spécifique (IBM-Rhapsody) . Nous souhaitons donc définir des transformations paramétrables qui puissent prendre en compte les spécificités des modeleurs UML. Celles-ci sont spécifiées dans un fichier regroupant les informations liées à la configuration du modeleur utilisé .

  Nous avons choisis de ne transformer les contraintes temporelles, exprimées en CCSL au sein des modèles UML, que dans la phase aval de la chaîne de transformation vers Fiacre .

- Traduction des contraintes CCSL en Fiacre. Les modèles UML-MARTE sont enrichis de contraintes qui expriment des relations temporelles entre des évènements spécifiques de l'exécution du modèle UML. Pour cela, nous exploitons les spécifications décrites en langage CCSL pour enrichir les modèles UML. Lors des transformation d'UML-MARTE, les contraintes CCSL sont traduites lors de la phase de transformation vers Fiacre .

- Correction des traductions par vérification d'observateurs. Pour exécuter les vérifica--tions de propriétés sur les modèles Fiacre générés, nous exploitons le langage CDL (*Context Description Language*) et l'outil OBP. Le langage CDL a pour but d'aider l'utilisateur à la formalisation des contextes et des propriétés au sein d'un même modèle afin de pouvoir, d'une part, limiter la portée des exigences par rapport à des cas d'utilisation (les contextes) et, d'autre part, limiter le problème de l'explosion combinatoire afin de garantir la mise à l'échelle de la vérification. Lors de la transformation CCSL vers Fiacre, les propriétés nécessaires à la correction des transformation sont générées en langage CDL et vérifiées avec OBP.

Dans ce travail, nous avons conçu une implantation des contraintes CCSL en langage Fiacre et avons exprimé des propriétés en langage CDL. La manipulation d'expressions CCSL dans le cadre de la modélisation avec des formalismes UML-MARTE permet d'étendre l'expressivité en intégrant des contraintes temporelles sur les modèles. Le modèle de temps logique proposé à l'OMG pour enrichir le formalisme UML MARTE permet la description et l'analyse de contraintes temporelles.

Nous avons rendu compte de la technique de vérification de propriétés par model-checking exploitant le language CDL et l'outil OBP . La technique s'appuie sur une traduction des modèles MARTE et des contraintes CCSL en code Fiacre. CDL permet d'exprimer aisemment des prédicats et des observateurs qui sont vérifiés lors de l'exploration exhaustive du modèle complet par OBP. Nous avons montré que le language permet une expression des propriétés avec une granularité très fine, référençant les variables et les états des processus implantés.

Avec le langage CDL, nous pouvons exprimer des propriétés pour vérifier si l'implantation des contraintes CCSL est correcte, soit pour vérifier que les parties fonctionnelles du circuit sont correctement implantées. L'expression des propriétés associées à l'implantation des contraintes a été évoquée dans cette thèse dans un but d'illustration des capacités d'expression du langage CDL. Une fois le schéma de traduction de CCSL en Fiacre étant fixé, la traduction des contraintes CCSL doit être vérifié une seule fois car elle ne dépend pas de l'application modélisée. Ce travail sera poursuivi par une étude de la traduction automatique de modèles UML-MARTE, enrichis des expressions CCSL.

8.2 Feedback de l'approche

L'approche développée dans nos travaux se repose sur deux idées conjointes : D'une part, réduire l'écart entre les modèles semi-formelles exprimés en MARTE-CCSL pour décrire les comportements du système à valider et les langages formels utilisés commme entrée pour le processus de model-checking de manière à exploiter ce que ces modèles permettent d'exprimer. D'autre part, décrire des propriétés temporelles réutilisables et exploitables lors de la vérification automatique. Cet axe de recherche a donné lieu au développement de travaux dans deux directions complémentaires :

- La première s'est intéréssée à une approche de transformation des modèles UML MARTE/CCSL en code formel.
- Une deuxième direction d'ordre méthodologique. On a choisis d'exploiter l'expresivité du langage CDL, afin de coder des propriétés qui sert à vérifier des contraintes temporelles bien spécifiques. L'utilisation de ce langage est associée à une méthode qui permette son intégration dans un processus plus large de développement industriel de logiciels et plus particulièrement qui permette une aide à la mise en oeuvre des techniques de vérification d'exigences de type logique temporelle.

8.3 Les bénéfices de l'approche

- L'utilisation d'un langage intermédiaire Fiacre permet de réduire l'écart sémantique entre les modèles de haut niveau exprimés en UML MARTE, ****by making it possible to precisely specify the semantics of the input language for system modeling***. Cela permet ainsi le Partage des spécifications via différents chaine d'outils de vérification.
- Codage d'automates CCSL : Une partie des automates observateurs sont des entrées réutilisable pour appliquer la vérification.
- Le langage CDL : Avec ce langage, les ingénieurs peuvent facilement exprimé des prédicats et des observateurs qui seront par la suite vérifiés durant l'exploration exhaustive du modèle, au lieu d'utiliser la logique LTL (Linear Temporal Logic) ou CTL (Computation Tree Logic) qui sont généralement des formules difficile à utiliser.
- L'approche contribue à clarifier son rôle par l'expression des propriétés temporelles dédiée au contraintes CCSL.

8.4 Perspectives

Nous avons pu à travers ce travail atteindre nos objectifs fixés, cependant un certain nombre d'améliorations peuvent être envisagés.

En termes de perspectives, la chaine de transformation est toujours en amélioration pour, d'un coté, lever certaine restrictions posées sur le language d'entrée et, de l'autre, permettre de cibler d'autres outils de vérification. Nous travaillons ainsi sur les aspects pratiques qu'impliquent les différentes méthodologies tant en amont qu' en aval.

Nous présentons ici quelques améliorations possibles à notre approche :

- Elargir l'approche à d'autres types de contraintes temporelles comme par exemple, le deadline, periode, et contrainte du temps physique .
- Etendre le champ de vérification par l'ajout des élements MARTE capable d'être transformer en langage Fiacre .
- Une autre amélioration qu'on peut porter, est de proposer d'autres algorithmes simulant l'ordonnanceur, afin de dinminuer le nombre d'êtat de l'algorithme tout en gardant le même principe de synchronisation des contraintes à vérifier (voir chapitre 6).

Nous pensons que cette approche de traduction peut être une étape importante vers un processus de vérification formelle des modèles MARTE et des spécifications CCSL. Dans ce travail, nous avons conçu une implantation des contraintes CCSL en langage Fiacre et avons exprimé des propriétés en langage CDL. Avec le langage CDL, nous pouvons exprimer des propriétés pour vérifier l'implantation des contraintes CCSL est correcte.

Part V

Publications

Publication

- Nadia Menad, Philippe Dhaussy: Real-Time and Embedded Systems Challenges - Key Requirements and Issues. ICSOFT 2013: 107-113

- Nadia Menad, Philippe Dhaussy, Belhadri Messabih: MDA Approach for Distributed and Real-Time embedded Systems Analysis and Verification - Overview and First Proposal. ICSOFT 2013: 114-123

- Nadia Menad, Philippe Dhaussy: A Transformation Approach for Multiform Time Requirements. SEFM 2013: 16-30

- Nadia Menad, Philippe Dhaussy, Zoé Drey : Towards a Transformation Approach of Timed UML MARTE Specifications for observer-based formal verification. Jounal. Computing and Informatics (CAI)

Part VI

Bibliographie

Bibliography

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, pages 121–135, 2010.

[ACD] P. Marquet A. Cuccuru and J.-L. Dekeyser. Uml2 as an adl hierarchical hardware modeling. In *Tech. Rep., Apr. 2004.*

[AGD11] Adolf Abdellah, Abdoulaye Gamati, and Jean-Luc Dekeyser. Modelisation uml/marte de soc et analyse temporelle basee sur l'approche synchrone. Technical Report 30.1089-1113, TSI, Architecture des ordinateurs, 2011.

[AM08] Charles Andre and Frederic Mallet. Clock constraints in uml/marte ccsl. Number 6540, 2008.

[AMdS07] Charles André, Frédéric Mallet, and Robert de Simone. Modeling time(s). In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, pages 559–573, 2007.

[And07] Charles Andre. Le temps dans le profil uml marte. Technical Report ISRN I3S/RR-2007-19-FR, Laboratoire I3S, 2007.

[And09] Charles Andre. Syntax and semantics of the clock constraint specification language ccsl. Technical Report 6925, INRIA, 2009.

[And10] Charles Andre. Verification of clock constraints: Ccsl observers in esterel. Technical Report 7211, INRIA, 2010.

[And11] Charles Andre. Modeles de temps et de contraintes temporelles de marte et leurs applications. Technical Report 7788, INRIA, 2011.

[Bar08] Sebastien Bardin. Introduction au model checking ensta. Technical report, CEA,LIST, Laboratoire de Surete logicielle Boite 65, Gif-sur-Yvette, F-91191 France, 2008.

[BB91] A. Benveniste and Gerard Berry. The sysnchronous approach to reactive and real-time systems. In *IEEE*, pages 1270–1282, 1991.

[BBL03] F. Vernadat J. Bernartt J. M. Farines J. P. Bodeveix M. Filali G. Padiou P. Michel P. Farail P. Gaufillet P. Dissaux B. Berthomieu, P. O. Ribet and J. L. Lambert. Towards the verification of real-time systems in avionics: the cotre approach. Number Elsevier, 201â216, 2003.

[BBV04] P.-O. Ribet B. Berthomieu and F. Verdanat. Construction of abstract state spaces for petri nets and time petri nets-the tool tina -. *International Journal of Production Research*, pages 42:2741–2756, July 2004.

[BCC+] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer, Strichman, and Yunshan Zhu. Bounded Model Checking, year = 2003, editor = 58 of Advances in Computers.

[BD] Dragan Bosnacki and Dennis Dams. Integrating real time into spin: A prototype implementation. In *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE XI / PSTV XVIII'98, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII)*, pages 3–6 November, 423–438. 1998, Paris, France.

[BD98] Dragan Bosnacki and Dennis Dams. Discrete-time promela and spin. 1998.

[BdS91] Frederic Boussinot and Robert de Simon. The esterel language. Number 79 (9), 1991.

[BEK+06] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, pages 425–439, 2006.

[BGJ91a] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[BGJ91b] Albert Benveniste, P.L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. Technical Report 7788, 1991.

[BH05] Dragan Bosnacki and Gerard J. Holzmann. Improving spin's partial-order reduction for breadth-first search.in spin. volume 3639, pages 91–105, 2005.

[BRR87] Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors. *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, volume 255 of *Lecture Notes in Computer Science*. Springer, 1987.

[BRV04] B. Berthomieu, P.-O. Ribet, and F. Verdanat. The tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42, 2004.

[CCGR00a] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *Int. J. on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CCGR00b] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.

[CD03] J.-M. ILIe et D. POITRENAUD C. DUTHEILLET, I. VERNIER-MOUNIER. State-space-based methods and model checking. In *Springer Verlag, Petri nets and system engineering (Claude Girault and Rudiger Valk Eds), first edn*, pages chap. 14, p. 201–276. . Reference 2 fois, pages 3 et 8, 2003.

[CD07] Michelle L. Crane and Jürgen Dingel. UML vs. classical vs. rhapsody statecharts: not all models are created equal. *Software and System Modeling*, 6(4):415–435, 2007.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 178–188, 1987.

[CRST11] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta. Formalizing requirements with object models and temporal constraints. *Software and System Modeling*, 10(2):147–160, 2011.

[DAB+08] Philippe Dhaussy, Julien Auvray, Stéphane De Belloy, Frédéric Boniol, and Eric Landel. Using context descriptions and property definition patterns for software formal verification. In *Workshop Modevva'08 (hosted by ICST'08)*, Lillehammer, Norway, 2008.

[DBR11] Philippe Dhaussy, Frédéric Boniol, and Jean-Charles Roger. Reducing state explosion with context modeling for model-checking. In *13th*

IEEE International High Assurance Systems Engineering Symposium (Hase'11), Boca Raton, USA, 2011.

[DBR+12] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, Amine Raji, Yves Le Traon, and Benoit Baudry. Formalisation de contextes et d'exigences pour la validation formelle de logiciels embarqués. *Technique et Science Informatiques*, 6/2012:797–826, 2012.

[DBRL12] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux. Improving model checking with context modelling. *Advances in Software Engineering*, ID 547157:13 pages, 2012.

[DCL11] Papa Issa Diallo, Joel Champeau, and Vincent Leilde. Model based engineering for the support of models of computation: The cometa approach. In *4th International Workshop on Multi-Paradigm Modeling (MPM'11) (hosted by Models'11)*, Wellington, New Zealand, october 2011.

[DCL13] Papa Issa Diallo, Joël Champeau, and Loïc Lagadec. A model-driven approach to enhance tool interoperability using the theory of models of computation. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, pages 218–237, 2013.

[DGK02] Shang-Wen Cheng David Garlan and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. sci. comput. program. page 23â49, 2002.

[DMA08] Julien DeAntoni, Frederic Mallet, and Charles Andre. Timesquare: on the formal execution of uml and dsl models. In *Tool session of the 4th Model driven development for distributed real time systems*, 2008.

[DPC+09] Philippe Dhaussy, Pierre-Yves Pillain, Stephen Creff, Amine Raji, Yves Le Traon, and Benoit Baudry. Evaluating context descriptions and property definition patterns for software formal validation. In Bran Selic Andy Schuerr, editor, *12th IEEE/ACM conf. Model Driven Engineering Languages and Systems (Models'09)*, volume LNCS 5795, pages 438–452. Springer-Verlag, 2009.

[DR11a] Philippe Dhaussy and Jean-Charles Roger. Cdl (context description language) : Syntax and semantics. Technical report, ENSTA-Bretagne, 2011.

[DR11b] Philippe Dhaussy and Jean-Charles Roger. Cdl (context description language) : Syntaxe et semantique. Technical report, Disponible sur http://www.obpcdl.org, ENSTA-Bretagne, 2011.

[Dut92] B. Dutertre. *Specification et preuves de systemes dynamiques*. PhD thesis, Universite de Rennes I, IFSIC, 1992.

[EBS06] Matthieu Leclercq-Vivien Quema Eric Bruneton, Thierry Coupaye and Jean-Bernard Stefani. The fractal component model and its support in java : Experiences with autoadaptive and reconfigurable systems, softw. pract. exper. Number 36, no. 11-12, 1257â1284., 2006.

[ECS86] E.A. Emerson E.M. Clarke and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In . *ACM Trans. Program. Lang.Syst.*, pages 8(2):244–263, 1986.

[eEAE81] E. M. CLARKE et E. A. EMERSON. Design and synthesis of synchronization skeletons using branchingtime temporal logic. In *In D. KOZEN, ed. : Logic of Programs, vol. 131 de Lecture Notes in Computer Science, p. 52-71, Yorktown Heights, New York, USA,*, pages Springer. ISBN 3–540–11212–X. Reference une fois, page 3., May 1981.

[eFB07] Philippe Dhaussy et Frederic Boniol. Mise en oeuvre de composants mda pour la validation formelle de modeles de systemes d'information embarques. Technical Report pages 133-157, 5,37, Revue des Siences et Techniques Informatiques, 2007.

[EJP97] E.A. Emerson, S. Jha, and D. Peled. Combining partial order and symmetry reductions. In *In E. Brinksma, editor, Tools and Algorithms for the Construction and Analysis of Systems, Enschede, The Netherlands, Springer Verlag, LNCS.*, volume 1217, pages 19–34, 1997.

[eJYH82] E. A. EMERSON et J. Y. HALPERN. Decision procedures and expressiveness in the temporal logic of branching time. In *In STOC„ San Francisco, California, USA,. ACM. Reference une fois, page 3.*, pages p. 169–180., May 1982.

[eJYH86] E. A. EMERSON et J. Y. HALPERN. Sometimes and not never revisited: on branching versus linear time temporal logic. In *J. ACM, page 3.*, pages 33(1):151–178„ 1986.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp: A protocol validation and verification toolbox. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 437–440, London, UK, 1996. Springer-Verlag.

[FGP+08] Patrick Farail, Pierre Gaufillet, Florent Peres, Jean-Paul Bodeveix, Mamoun Filali, Bernard Berthomieu, Saad Rodrigo, Francois Vernadat, Hubert Garavel, and Frédéric Lang. FIACRE: an intermediate language for model verification in the TOPCASED environment. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008*. SEE, janvier 2008.

[FJ14] Jerome Delatour Luka Le Roux Philippe Dhaussy Frederic Jouault, Ciprian Teodorov. Transformation de modeles uml vers fiacre, via les langages intermediaires tuml et abcd. Technical report, Real time UML workshop for Embedded Systems, Juin 2014.

[FSKdR05] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, pages 52–65, 2005.

[GL02] Hubert Garavel and Frederic Lang. Ntif: A general symbolic model for communicating sequential processes with data. Number 276â 291.Springer Verlag, Novembre 2002.

[GMD12] Regis Gascon, Frederic Mallet, and Julien Deantoni. Logical time and temporal logics: Comparing uml marte/ccsl and psl. Technical Report ISBN: 978-0-7695-4508-0, INRIA, 2012.

[Gol87] U. Goltz. Synchronic distance. - petri nets: Central models and their properties. Technical report, 1987.

[GPC12] Ning Ge, Marc Pantel, and Xavier Crégut. Time properties dedicated transformation from uml-marte activity to time transition system. *ACM SIGSOFT Software Engineering Notes*, 37(4):1–8, 2012.

[Gro04] Object Managament Group. Enhanced view of time specification. Technical Report Version 1.2, formal /04-10-04, 492, Old Connecticut Path, Framing Ham, October 2004.

[Gro06] Object Management Group. Uml 2.1 superstructure specification. Technical Report Framing-ham, MA 01701, OMG document number : ptc/2006-04-02, Inc., 492 Old Connecticut Path, April 2006.

[Gro09a] Object Management Group. Uml 2.2 superstructure and infrastructure. Technical Report formal 2009-02-04, http//www.omg.org/spec/UML/2.2, February 2009.

[Gro09b] Object Management Group. Omg: Uml profile for marte, v1.0. Technical Report formal/2009-11-02, November 2009.

[Gro12] Object Management Group. Omg object constraint language (ocl), version 2.3.1. Technical report, 2012.

[Hen12] Sotharith Heng. Transformation from uml into fiacre programs. Technical report, Lab-STICC ENSTA Bretagne, Brest, 2012.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and

G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HMP91] T.A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *Proceedings of the 1991 REX Workshop*, 1991.

[Hol97a] G.J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[Hol97b] G.J. Holzmann. The model checker spin. In *Software Engineering*, pages 23(5):279–295, 1997.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[JCFS96] Alain Kerbrat Laurent Mounier Radu Mateescu Jean-Claude Fernandez, Hubert Garavel and Mihaela Sighireanu. Cadp: A protocol validation and verification toolbox. *In CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification, London, UK. Springer-Verlag*, pages 437–440, 1996.

[JD14] Frederic Jouault and Jereme Delatour. tUML : syntax and semantic. Technical report, ESEO, 2014.

[JL12] Jifeng He Frédéric Mallet Zuohua Ding Jing Liu, Ziwel Liu. Hybride marte statecharts. Number DOI 10. 1007/s11704-012-1301-1, 2012.

[JTD+14] Frederic Jouault, Ciprian Teodorov, Jerome Delatour, Luka Le Roux, and Philippe Dhaussy. Transformation de modeles UML vers FIACRE, via les langages intermediaires tUML et ABCD. In *Revue Genie Logiciel*, number 109, June 2014.

[KCS09] Ali Koudri, Joel Champeau, and Philippe Soulard. Mopcom/marte process applied to a cognitive radio system design and analysis. In *Model Driven Architecture - Foundations and Applications*, 2009.

[Kon05] Alexander Konigs. Model transformation with triple graph grammar. Septembre, 2005.

[Kop11] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer, 2011.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The epsilon transformation language. In *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, pages 46–60, 2008.

[KW07a] Ekkart Kindler and Robert Wagne. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Technical Report, University of Paderborn, D-33098 Paderborn, Germany, 2007.

[KW07b] Ekkart Kindler and Robert Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, University of Paderborn, 2007.

[L.L78] L.Lamport. Time, clocks, and the ordering of events in a distributed system. Technical Report 21(7) :558-565, Communications of the ACM, 1978.

[LPY97a] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LPY97b] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[LSV98] E. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. Technical Report 17(12): 1217-1229, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, December 1998.

[Mal12a] Frédéric Mallet. Automatic generation of observers from marte/ccsl. Number 978-1-4673-2789-3, 2012.

[Mal12b] Frédéric Mallet. Automatic generation of observers from marte/ccsl. In *RSP*, pages 86–92, 2012.

[MAS08] Frederic Mallet, Charles Andre, and Robert De Simone. Ccsl: Specifying clock constraints with uml/marte. In *ISSE*, volume 4, pages 309–314, 2008.

[MD13] Nadia Menad and Philippe Dhaussy. A transformation approach for multiform time requirements. Number SEFM, 16-30, 2013.

[Mes90] David G. Messerschmitt. Synchronization in digital system design. *IEEE Journal on Selected Areas in Communications*, 8(8):1404–1419, 1990.

[MP95] K. L. Mc.Millan and D. K. Probst. A technique of state space search based on unfolding. *In Formal Methods in System Design*, 6:pages 45–65, January 1995.

[NGC12] Marc Pantel Ning Ge and Xavier Cregut. A framework dedicated to time properties verification for uml-marte specifications. 2012.

[OMG05] OMG. Uml profile for schedulability, performance, and time specification. Technical Report Framing-ham, MA 01701 OMG document number : formal/05-01-02 (v1.1), Inc., 492 Old Connecticut Path, January, 2005.

[OMG10] OMG. Uml profile for marte, v1.1. In *Object Managment Group*, Document number: PTC/10-08-32, August 2010.

[OMG11] OMG. Omg mof 2 xmi mapping specification. Technical report, Object Management Group Version 2.4.1, 2011.

[OSE04] OSE. Ose, real time operating sytem. 2004.

[Pel98] D. Peled. Ten years of partial order reduction. In *In CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification. Springer-Verlag.*, pages 17–28, 1998.

[PFS08] Marie-Agnès Peraldi-Frati and Yves Sorel. From high-level modelling of time in marte to real-time scheduling analysis. Number Int. Workshop of MODEL'S 2008 - Int. Workshop on Model Based Architecting and Construction of Embedded Systems 503 (2008) 129-144, 2008.

[PGB00] Luca Abeni Marco di Natale Paolo Gai, Giuseppe Lipari and Enrico Bini. Architecture for a portable open source real-time kernel environment. November 2000.

[PGB01] Massimiliano Giorgi Paolo Gai, Luca Abeni and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. June 2001.

[PGS96] Doron Peled Patrice Godefroid and Mark G. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. In *In International Symposium on Software Testing and Analysis, San Diego, CA, S. J. Zeil and W. Tracz, ACM Press, New York.*, pages 261–269, January 1996.

[PLG03] Jean Christophe Le Lann Paul Le Guernic, Jean-Pierre Talpin. Polychrony for system design. tachnical report, rr-4715. Technical report, INRIA, , Renne, 2003.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[PNU77] A. PNUELI. The temporal logic of programs. In *In FOCS, Providence, Rhode Island, USA. IEEE*, pages p. 46–57, page 3., 1977.

[PNU81] A. PNUELI. The temporal semantics of concurrent programs. In *Theoretical Computer Science*, pages 13:45–60, 1981. Reference une fois, page 3., 1981.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[QS08] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of conurrent systems in cesar. In *25 Years of Model Checking*, pages 216–230, 2008.

[RAR97] Thomas A. Henzinger Shaz Qadeer Rajeev Alur, Robert K. Brayton and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *In Computer Aided Verification,Springer Verlag, LNCS*, volume 1254, pages 340–351, 1997.

[RDA10] Amine Raji, Philippe Dhaussy, and Bruno Aizier. Automating context description for software formal verification. In *Workshop (MoDeVVa'10)*, Oslo, Norway, 2010.

[RDb11] Amine Raji, Philippe Dhaussy, and Benoit baudry. Extension of use cases for context-aware verification. In *Workshop Modevva'11 (hosted by Models'11)*, Wellington, New Zealand, 2011.

[Ren09] Xavier Renault. Mise en oeuvre de notations standardisees, formelles et semi-formelles dans un processus de developpement de systemes embarques temps-reel repartis. Technical report, Laboratoire d'Informatique de Paris 6, 2009.

[rFHN] Jean remy Falleri, Marianne Huchard, and ClÃ©mentine Nebut. C.: Towards a traceability framework for model transformations in kermeta.

[Riv95] Wind River. Vxworks programmer's guide : Algorithms for real-time sheduling problems. 1995.

[RM13] Yuliia Romenska and Frédéric Mallet. Lazy parallel synchronous composition of infinite transition systems. In *ICTERI*, pages 130–145, 2013.

[RMGF12] Alessandro Gerlinger Romero, calves Mauricio Gon and Vieira Ferreira. An approach to model-driven architecture applied to space real-time software. Number National Institute for Space Research (INPE), 12227-010, Brazil, 2012.

[Rog06] Jean-Charles Roger. Exploitation de contextes et d'observateurs pour la verification formelle des modeles. Technical Report Universite de Rennes I, PhD thesis, ENSTA-Bretagne, 2006.

[SAE] SAE. Architecture analysis design language (standard sae as5506. In *September 2004, available at http ://www.sae.org.*

[SBP+09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks, Frank Budinsky, Dave Steinberg, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse modeling framework*. The eclipse series. Addison-Wesley, Boston, London, 2009.

[Sch94] F. A. Schreiber. Is time a real time,an overview of time ontology in informatics. Technical Report F127 :283-307, Real Time Computing, 1994.

[Sel04] Bran Selic. On the semantic foundations of standard UML 2.0. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 181–199. Springer, 2004.

[SEV07] A. Haase C. Kadura B. Kolb D. Moroff K. Thoms S. Efftinge, P. Friese and M. Voelter. user guide," openarchitectureware community. 2007.

[SM94] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[SSMP13] Jagadish Suryadevara, Cristina Cerschi Seceleanu, Frédéric Mallet, and Paul Pettersson. Verifying marte/ccsl mode behaviors using uppaal. In *SEFM*, pages 1–15, 2013.

[TMK] Daniel Varro Tom Mens, Pieter Van Gorp and Gabor Karsai. Applying a model transformation taxonomy to graph transformation technology.

[Val91] Antti Valmari. Stubborn sets for reduced state space generation. In *In Proceedings of the 10th International Conference on Applications and Theory of Petri Nets , London, UK, Springer-Verlag.*, pages 491–515, 1991.

[W3C99] W3C. Xsl transformations (xslt) version 1.0. Technical report, http://www.w3.org/tr/xslt, 1999.

[Whi06] Jon Whittle. Specifying precise use cases with use case charts. In *In 9th IEEE/ACM conf. Model Driven Engineering Languages and Systems (MoDELS'06), Satellite Events. Genova, Italy.*, number 290301, October 2006.

[YM11] Ling Yin and Frederic Mallet. Correct transformation from ccsl to promela for verification. Technical Report 7491, INRIA, 2011.

[YTB+11] Huafeng Yu, Jean-Pierre Talpin, Loic Besnard, Thierry Gautier, Herve Marchand, and Paul Le Guernic. Polychronous controller synthesis from marte ccsl timing specifications. In *Memocode*, 2011.

Annexe A

Automates Fiacre des contraintes CCSL

A.1 Implémentation des contraintes CCSL

Nous présentons ici quelques-unes des relations décrites dans (André, 2010; Yin et al. 2011), à savoir la relation *AsFrom*, *Préemption*, *Await* et la relation *Concaténation*.

A.1.1 La relation AsFrom

La relation "A partir de " (dénotée AsFrom) est une relation synchrone entre deux horloges C_1, C_2. Elle spécifie que pour tout nombre naturel k, le 1^{er} instant de C_2 survient après le $k^{ème}$ instant de C_1 (k connu et fixé selon les besoins).

En reprenant la formalisation décrite dans (André, 2007), la relation :
$C_2 = C_1$ *AsFrom* k s'exprime formellement par l'expression(1) :

$$C_2 = C_1 k \Leftrightarrow (C_2 \in C_1) \wedge (C_2[1] \equiv C_1[k+1]) \quad (1)$$

Pour notre cas d'étude, nous illustrons la relation $B = AAsFromk$ par l'automate Figure A.1(a) et le chronogramme Figure A.1(b).

A chaque tique de A (S1) on incrémente n, et on teste sa valeur par rapport à k ce qui détermine le tique de B ou pas. Notons qu'il est impossible pour $B(S4)$ de tiquer sans A même au-delà de $k+1$. Notons aussi qu'après avoir atteint $k+1$, nous n'avons plus besoin de tester n, cela devient un peu comme pour l'alternance A ensuite B ensuite A. La naissance de l'horloge B est comprise entre l'instant k et $k+1$ de l'horloge A.

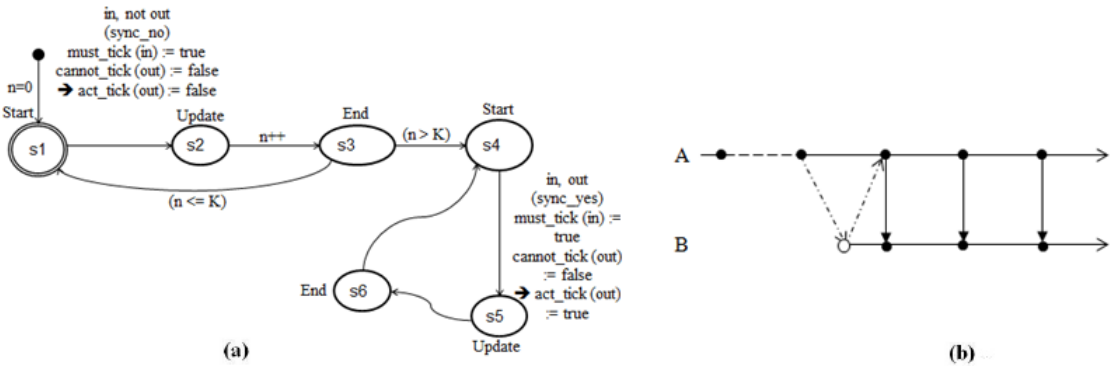

Figure A.1: Illustration de la contrainte AsFrom : B = A AsFrom k

SyncYes : est délivré lorsque $n > k$, les deux horloges A, B tiquent. **SyncNo :** est délivré lorsque $n <= k$, seul l'horloge A tique.

Dans ce cas d'étude le tic de B signifie que le Scheduler envoie un *SyncYes* au processus *Proc*. Ce dernier reçoit un signal provenant du processus *Prod* à destination de *Acq*. Donc le processus transmet ce signal s'il reçoit un SyncYes ou le détruit s'il reçoit un *SyncNo*.

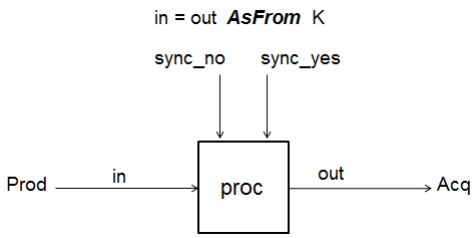

Figure A.2: Cas illustratif du AsFrom

A.1.2 La relation Await

La relation *Await* est une relation synchrone entre deux horloges C_1 et C_2. Elle spécifie que pour tout nombre naturel k, l'horloge C_2 tique a un seul instant précis qui est le kème tique de l'horloge C_1, elle meurt par la suite. En reprenant la formalisation décrite dans (André, 2007), la relation $C_2 = C_1$ Await k s'exprime formellement par l'expression :

1) $(|I_C 2| = 1) \wedge$

2) $((\exists C_1[k] \in I_C 1)(C_2[1] \equiv C_1[k]))$

Pour notre cas d'étude, nous illustrons la relation B = A Await k par le chronogramme Figure A.3(a) et l'automate Figure A.3(b).

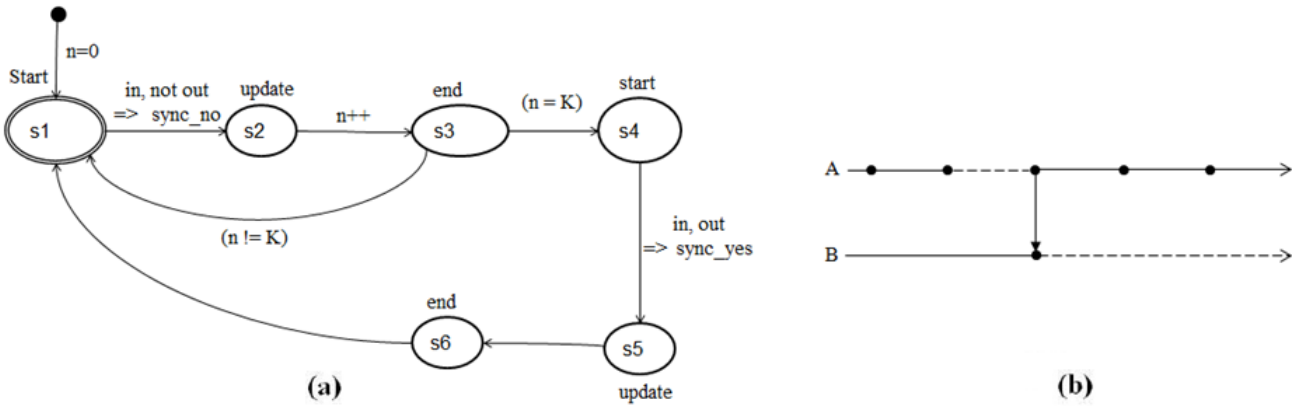

Figure A.3: Illustration de la contrainte Await : B = A Await k

SyncYes : est délivré lorsque $n = k$, les deux horloges A, B tiquent. **SyncNo :** est délivré lorsque $n \neq$k, $seull'horlogeAtique$.

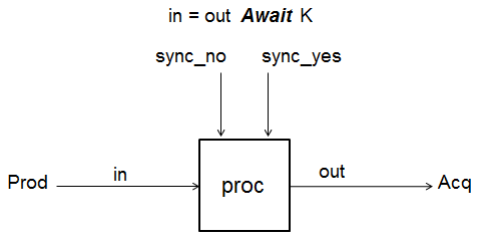

Figure A.4: Cas illustratif du Await

Dans ce cas d'étude le tic de B signifie que le Scheduler envoie un *SyncYes* au processusProc. Ce dernier reçoit un signal provenant du processus *Prod* à destination de *Acq*. Donc le processus transmet ce signal s'il reçoit un SyncYes ou le détruit s'il reçoit un *SyncNo*.

A.1.3 La relation Préemption

La relation de *préemption* (dénotée Upto) définit une horloge C qui tique lorsque l'horloge A tique et dont le cycle de vie s'achève définitivement lorsque l'horloge B commence à tiquer. En reprenant la formalisation décrite dans (André, 2007), la relation C = A Upto B s'exprime formellement par les propriétés :

1) $(C \subset A) \wedge$

2) $((\forall k \in N*, A[k] \in I_A)(A[k] < B[1]) \Rightarrow (C[k] \equiv A[k]))$

Pour notre cas d'étude, nous illustrons la relation $C = AUptoB$ par l'automate Figure A.6(a) et le chronogramme Figure A.6(b).

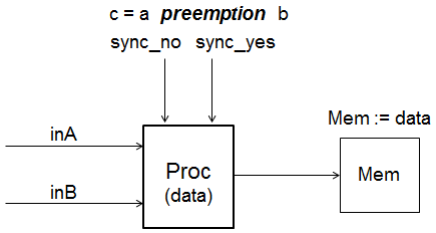

Figure A.5: Cas illustratif de la préemption

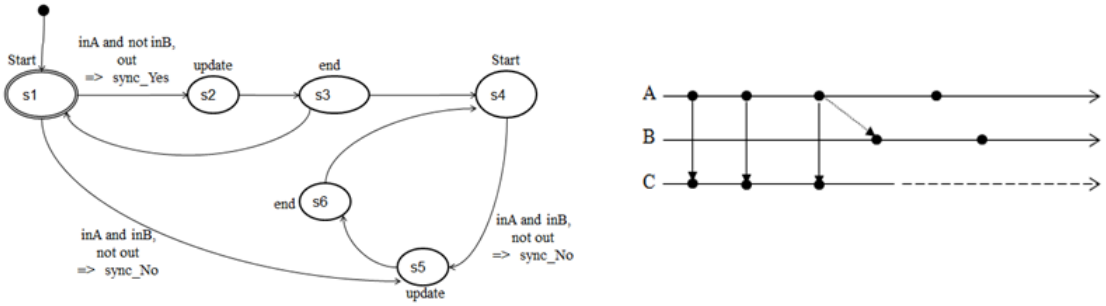

Figure A.6: Illustration de la contrainte préemption : C = A Upto B

SyncYes : est délivré lorsque l'horloge A tique.

SyncNo : est délivré l'horloge B entre en exécution.

Dans ce cas d'étude le tic de B signifie que le Scheduler envoie un SyncYes au processus Proc. Ce dernier reçoit un signal provenant du processus Prod à destination de Acq. Donc le processus (Proc) mémorise la donnée transmise s'il reçoit un SyncYes ou le détruits'il reçoit un SyncNo.

A.1.4 La relation Concaténation

La relation concaténation (dénotée Concat) définit une horloge C qui tique en même temps que A, tant que celle-ci est vivante le tique de l'horloge B n'a pas d'impact sur celui de C. dès lors ou l'horloge A meurt, B prend le relais ce qui fais de l'horloge C le résultat d'une concaténation entre les deux horloges. En reprenant la formalisation décrite dans (André, 2007), la relation C = A Concat B s'exprime formellement par l'expression :

$Let\ l = |I_A|, (\forall k \in N*, C[k] \in I_C)$

1) $((k \leq l) \wedge (C[k] \equiv A[k])) \vee$

2) $((k > l) \wedge (\exists m \in N*, B[m] \leq A[l] < B[m+1]) \wedge (B[k+m-l] \in I_B)$
$\wedge (C[k] \equiv B[m+k-l]))$

Pour notre cas d'étude, nous illustrons la relation C = A Concat B par l'automate Figure A.7(a) et le chronogramme Figure A.7(b).

SyncYes : est délivré lorsque l'horloge A tique ou lorsque A meurt et l'horloge B prend le relais.

SyncNo : est délivré soit lorsque A est en execution et B tique ce dernier n'est pas pris en consideration, ou lorsque A meurt sans que B ne prene le relais.

A.1.5 La relation Intersection (*)

Nous présentons ci-dessous quelques-unes des relations décrites dans (André, 2010; Yin et al. 2011), à savoir les relations Intersection , Union, Sup et Inf .

La relation Intersection est une relation synchrone entre deux horloges C_1, C_2, en résulte une troisième horloge **Out** qui tique à l' instant où les deux horloges

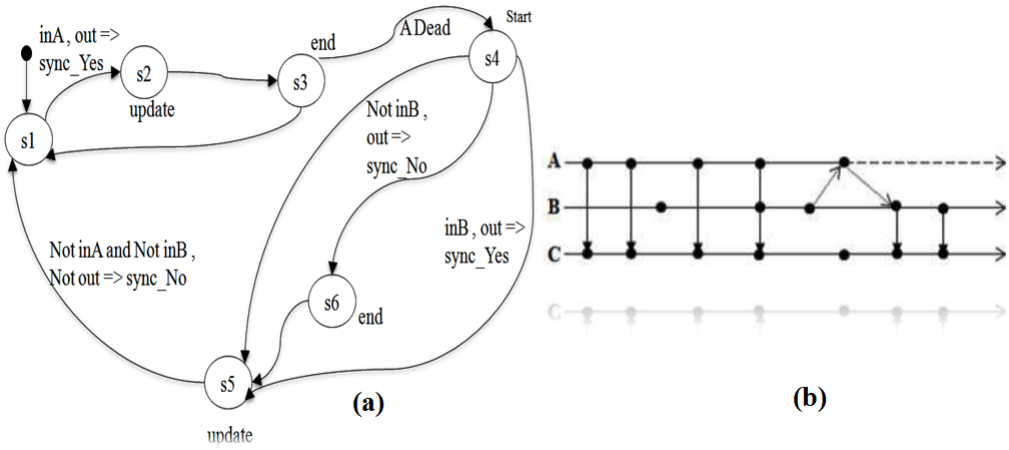

Figure A.7: Illustration de la contrainte concaténation : C = A Concat B

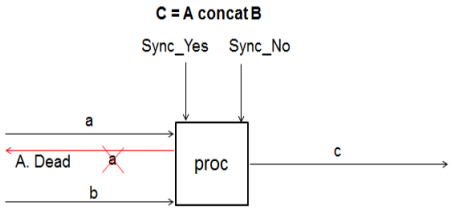

Figure A.8: Cas illustratif de la relation de concaténation

précédentes C_1 & C_2 tiquent, en résumé pour qu'il y'ait un tique de lhorloge **Out**, il faut impérativement les tiques de C_1 & C_2 (l'ordre nest pas important), l'absence de tique de l'horloge C_1 ou C_1 implique l'absence de l'horloge Intersection résultante.

Dans nos implémentation, l'horloge **Out** est activée par deux synchronisations **SYNC_YES** et **SYNC_NO**, qui permettent respectivement la présence et l'absence de l'horloge **Out**. En reprenant la formalisation décrite dans (André, 2007).

La relation **Out** = C_1 * C_2, s'exprime formellement par les expressions suivantes :

$$(C \subset A) \wedge \tag{A.1}$$

$$(C \subset B) \wedge \tag{A.2}$$

$$((\forall i \in$$

$I_a)\ (\forall j \in I_b)(\ \exists k \in I_c)(i \equiv j) \Rightarrow (i \equiv k)).$(A.3)

Pour notre cas d'étude, nous illustrons la relation **Out= C1 Intersect C2** par l'automate Figure A.10 (a) et le chronogramme Figure A.10 (b).

Notons que la condition irréversible pour avoir en sortie lhorloge **Out** en d'autre termes l'horloge **SYNC_YES** mentionné ci-dessus est le tique de chacune des horloges **C1 & C2** (peu importe lordre).

L'absence de l'horloge **Out**, c'est-à-dire le non lieu de l'intersection est du au not_tick de l'horloge **C1 ou C2**.

Automate et chronogramme de la contrainte Intersection

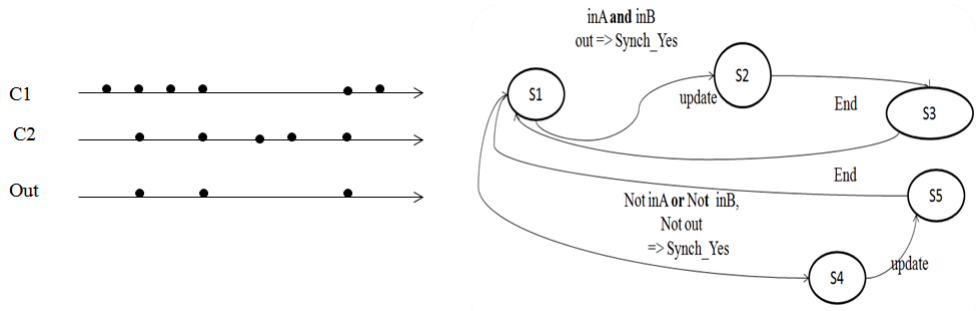

Figure A.9: Automate et chronogramme de la contrainte Intersection

L'utilisation de la contrainte est démontrée dans la Figure A.10 :

- **Sync_yes** : est délivré lorsque Out tique.
- **Sync_no** : est délivré lorsque les deux horloges **C1 ou C2** ne tiquent pas.

Dans ce cas d'études les tiques de C1 et C2 signifient que l'ordonnanceur envoie un **Sync_Yes** au processus proc.

Ce dernier reçoit un signal provenant du processus capteur à destination d'ACQ. Donc le processus transmet ce signal q'il reçoit un **Sync_Yes** ou le détruit s'il reçoit un **Sync_No**.

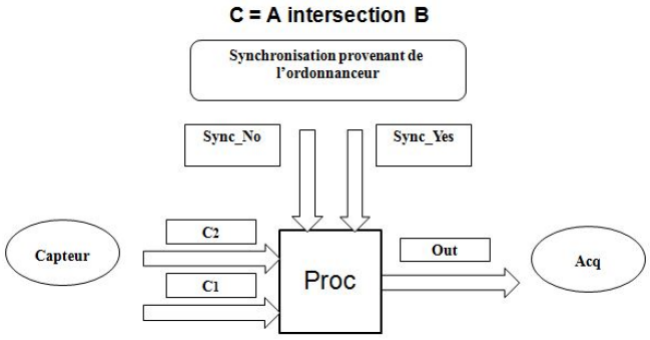

Figure A.10: Cas illustratif de l'utilisation des contrainetes Intersection

A.1.6 La relation Union (+)

La relation *Union* est une relation synchrone entre deux horloges **C1, C2**, en résulte une troisième horloge **Out** qui tique à l' instant où une des deux horloges précédentes **C1 ou C2** tiquent, en résumé pour qu'il y'ait un tique de l'horloge **Out**, il faut soit un tique de **C1** ou un tique de **C2** (l'ordre nest pas important), l'absence de tique de l'horloge C1 & C2 implique l'absence de l'horloge Union résultante.

Dans nos implémentation, l'horloge **Out** est activée par deux synchronisations **SYNC_YES** et **SYNC_NO**, qui permettent respectivement la présence et l'absence de l'horloge **Out**. En reprenant la formalisation décrite dans (André, 2007).

La relation $C_3 = C_1 + C_2$, s'exprime formellement par les expressions suviantes :

$$(A \subset C) \wedge \tag{A.4}$$

$$(B \subset C) \wedge \tag{A.5}$$

$$((\forall i \in$$

$\mathrm{I}_c)$ ($\exists j \in \mathrm{I}_a \cup \mathrm{I}_b)(\mathrm{i} \equiv j)$.(A.6)

Pour notre cas d'étude, nous illustrons la relation **Out= C1 union C2** par l'automate (cf Figure **??**(a)) et le chronogramme (cf Figure **??** (b)).

Notons que la condition irréversible pour avoir en sortie l'horloge **out** en d'autre termes la synchronisation **SYNC_Yes** mentionné ci-dessus est le tique des horloges **C1 ou C2** (peu importe l'ordre).

L'absence de l'horloge **out** qui est la synchronisation **Sync_No**, c'est-à-dire le non lieu de l'union qui est due au not_tick des horloges **C1 & C2**.

Automate et chronogramme de la contrainte Union

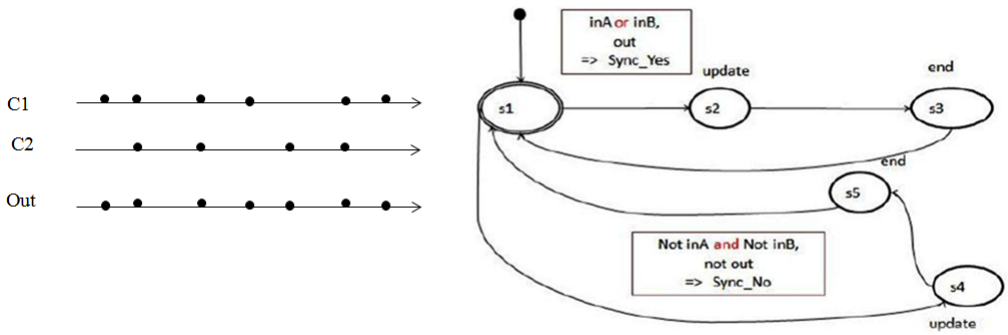

Figure A.11: Illustration de la contrainte d'union

Illustration de l'utilisation de la contrainte

L'utilisation de la contrainte est démontrée dans la Figure A.12(a) ci-dessous :

- **Sync_yes** : est délivré lorsque Out tique.
- **Sync_no** : est délivré lorsque les deux horloges **C1 et C2** ne tiquent pas.

Dans ce cas d'études les tiques de C1 ou C2 signifient que l'ordonnanceur envoie un **Sync_Yes** au processus proc.

Ce dernier reçoit un signal provenant du processus capteur à destination d'ACQ.

Donc le processus transmet ce signal q'il reçoit un **Sync_Yes** ou le détruit s'il reçoit un **Sync_No**.

A.1.7 La relation Supérieure ($\vee$)

La relation *Supérieur* est une relation synchrone entre deux horloges **C1, C2**, en résulte une troisième horloge **Out** qui tique, l'horloge **Out** tique avec l'hologe la plus

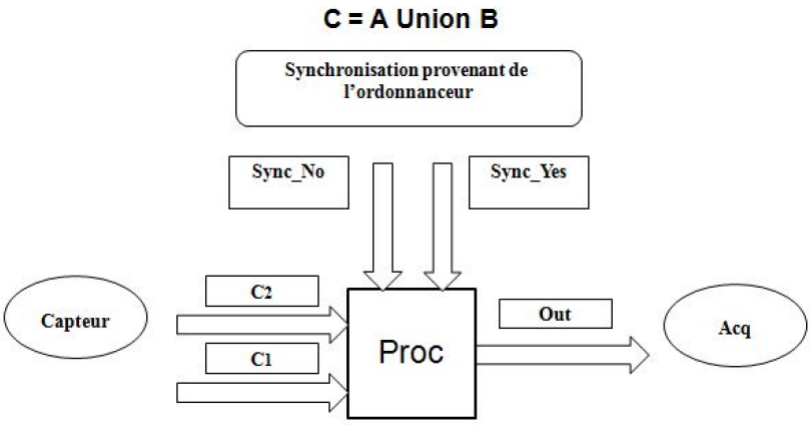

Figure A.12: Cas illustratif de l'utilisation de la contraint d'union

lente entre **C1 & C2**. En d'autre termes l'horloge **out** tique au moment ou l'horloge avec le moindre nombre de tiques, tique.

Dans nos implémentation, l'horloge **Out** est activée par deux synchronisations **SYNC_YES** et **SYNC_NO**, qui permettent respectivement la présence et l'absence de l'horloge **Out**. En reprenant la formalisation décrite dans (André, 2007).

La relation **Out** $= C_1 \vee C_2$, s'exprime formellement par les expressions suivantes :

$$Leta, bC = \{d \in C | (a \leq d) \wedge (b \leq d)\}, (\forall C' \in a, bC)((a \wedge b) \leq C') \tag{A.7}$$

Comme on peut exprimer la contrainte par les expressions suivantes :
Let c = (a $\wedge b$) : $(\forall k \in N*, c[k] \in \mathrm{I}_c)$

$$(a[k] \in$$

I_a) $\wedge$(A.8)

$$(b[k] \in$$

I_b) $\wedge$(A.9)

$$(C[k] \equiv \{ \begin{array}{lll} a[k] & if & b[k] \leq a[k] \\ b[k] & if & a[k] \leq b[k] \end{array}) \tag{A.10}$$

Pour notre cas d'étude, nous illustrons la relation **Out= C1 sup C2** par l'automate Figure A.13(a)et le chronogramme Figure A.13(b)

Notons que la condition irréversible pour avoir en sortie lhorloge **out** en d'autre termes la synchronisation **SYNC_Yes** mentionné ci-dessus est le tique de l'horloge la plus lente entre C_1 et C_2, ou l'horloge avec le moins de tiques.

L'absence de l'horloge **out** qui est la synchronisation **Sync_No**, c'est-à-dire le non lieu de la relation supérieur qui est due au not_tick de l'horloge la plus lente entre **C1 & C2**.

Automate et chronogramme de la contrainte Supérieure

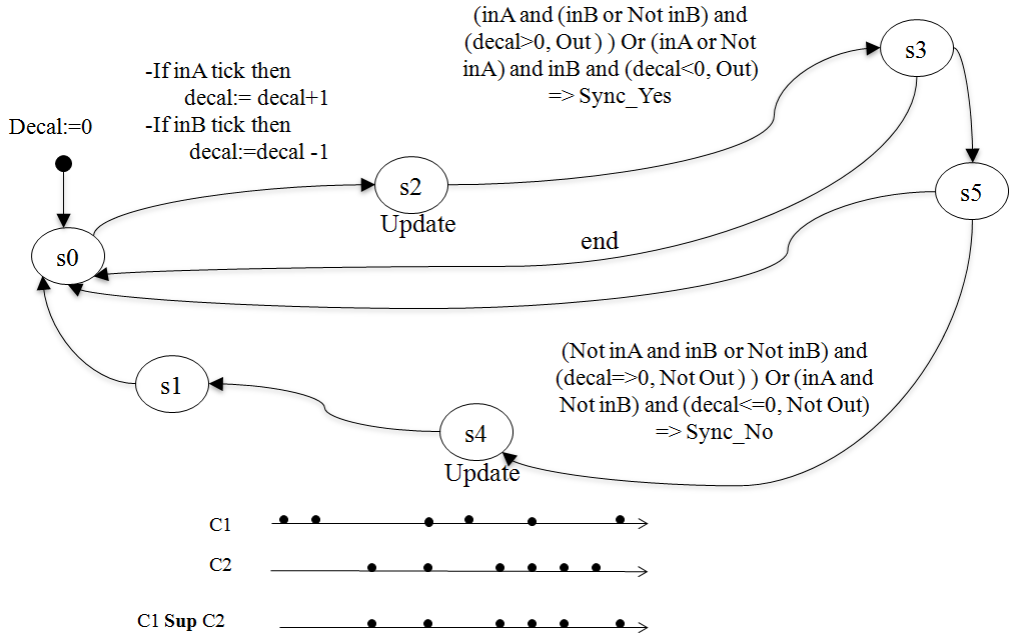

Figure A.13: Illustration de la contrainte Supérieur

Illustration de l'utilisation de la contrainte

L'utilisation de la contrainte est démontrée dans la Figure **??** ci-dessous :

- **Sync_yes** : est délivré lorsque Out tique.
- **Sync_no** : est délivré lorsque l'horloge la plus lente entre **C1 et C2** ne tiquent pas.

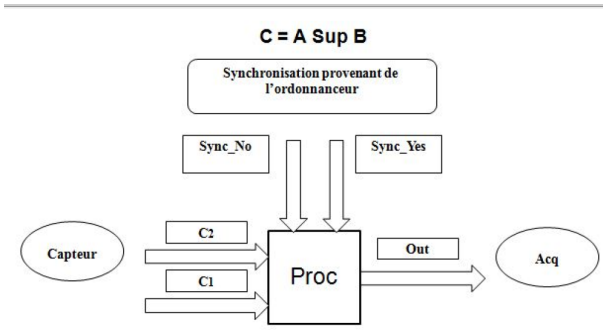

Figure A.14: Cas illustratif de l'utilisation de la contrainte Supérieure

Dans ce cas d'études le tique de l'horloge la plus le lente **C1 ou C2** signifient que l'ordonnanceur envoie un **Sync_Yes** au processus proc.

Ce dernier reçoit un signal provenant du processus capteur à destination d'ACQ.

Donc le processus transmet ce signal q'il reçoit un **Sync_Yes** ou le détruit s'il reçoit un **Sync_No**.

A.1.8 La relation Inférieur ($\wedge$)

La relation *inférieur* est le dual de la relation supérieure qu'on a cité ci-dessus,c'est aussi une relation synchrone entre deux horloges **C1, C2**, en résulte une troisième horloge **Out**, l'horloge **Out** tique avec l'hologe la plus rapide entre **C1 & C2**. En d'autre termes l'horloge **out** tique au moment ou l'horloge avec le plus de nombre de tiques.

Dans nos implémentation, l'horloge **Out** est activée par deux synchronisations **SYNC_YES** et **SYNC_NO**, qui permettent respectivement la présence et l'absence de l'horloge **Out**. En reprenant la formalisation décrite dans (André, 2007).

La relation **Out** $= C_1 \wedge C_2, s'exprime formellement par les expressions suivantes$:

$$Let a, bC = \{d \in C | (d \leq a) \wedge (d \leq b)\}, (\forall c' \in a, bC)(c' \leq (a \wedge b)) \qquad (A.11)$$

Comme on peut exprimer la contrainte par les expressions suivantes :
Let c = (a $\wedge b$) : $(\forall k \in N*, c[k] \in \mathrm{I}_c)$

$$((a[k] \in$$

$\mathrm{I}_a) \wedge (b[k] \in \mathrm{I}_b) \Rightarrow a[k]b[k]) \wedge (c[k] \equiv a[k])) \wedge$ (A.12)

$((b[k] \in$

$\mathrm{I}_b) \wedge (a[k] \in \mathrm{I}_a) \Rightarrow b[k]a[k]) \wedge (c[k] \equiv b[k]))$(A.13)

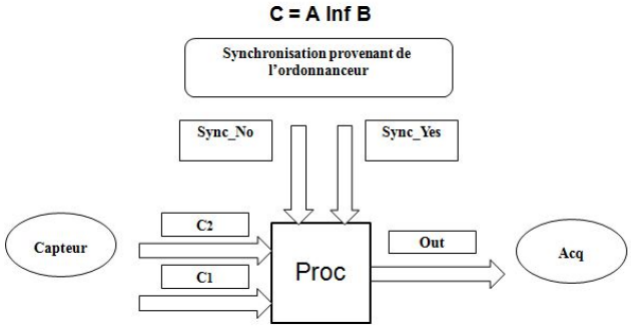

Figure A.15: Cas illustratif de l'utilisation de la contrainte inférieure

Pour notre cas d'étude, nous illustrons la relation **Out= C1 inf C2** par l'automate **??** (a) et le chronogramme figue A.16 Notons que la condition irréversible pour avoir en sortie

l'horloge **out** en d'autre termes la synchronisation **SYNC_Yes** mentionné ci-dessus est le tique de l'horloge la plus rapide entre C_1 et C_2, ou l'horloge avec le plus de tiques.

L'absence de l'horloge **out** qui est la synchronisation **Sync_No**, c'est-à-dire le non lieu de la relation inférieure qui est due au not_tick de l'horloge la plus rapide entre **C1 & C2**.

Automate et chronogramme de la contrainte inférieure

Illustration de l'utilisation de la contrainte

L'utilisation de la contrainte est démontrée dans la Figure A.15 ci-dessous :

- **Sync_yes** : est délivré lorsque Out tique.
- **Sync_no** : est délivré lorsque l'horloge la plus rapide entre**C1 et C2** ne tiquent pas.

Dans ce cas d'études le tique de l'horloge la plus rapide **C1 ou C2** signifient que l'ordonnanceur envoie un **Sync_Yes** au processus proc.

Ce dernier reçoit un signal provenant du processus capteur à destination d'ACQ.

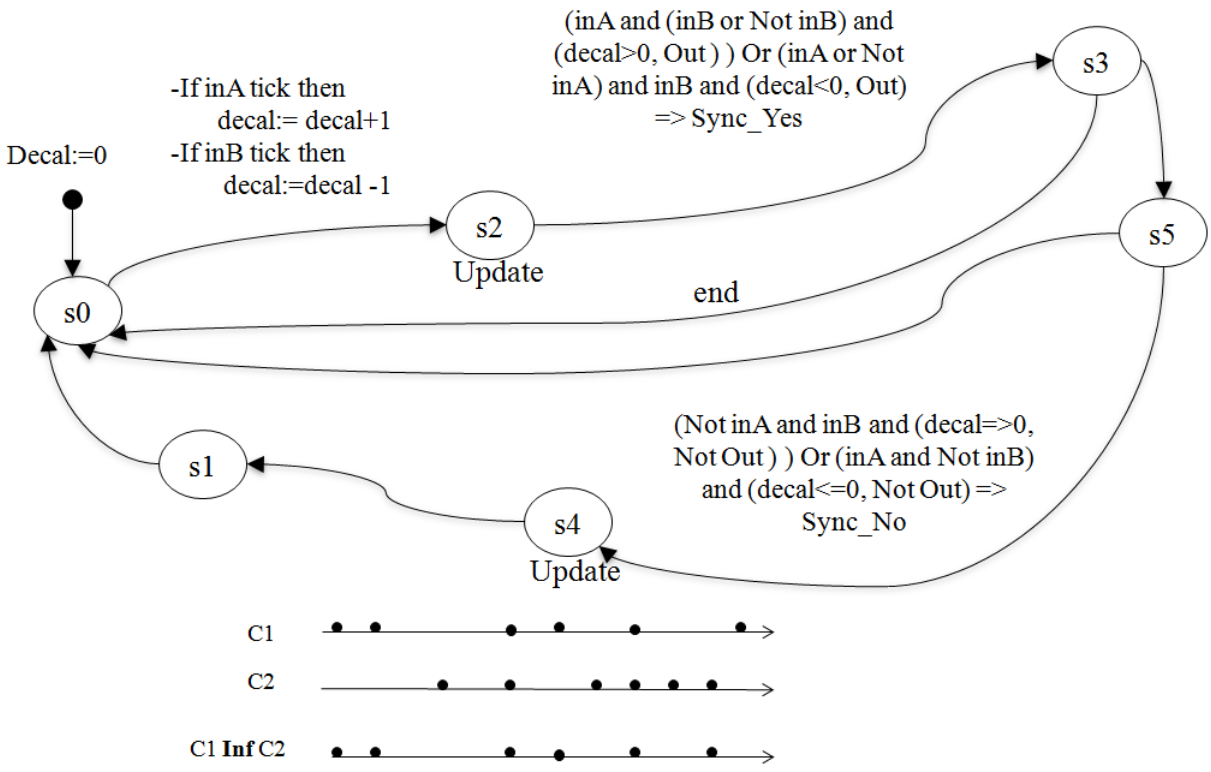

Figure A.16: Illustration de la contrainte Inférieure

Donc le processus transmet ce signal q'il reçoit un **Sync_Yes** ou le détruit s'il reçoit un **Sync_No**.

Nous pouvons maintenant déclarer, dans une clause CDL du programme CDL et avec l'opérateur -assert- l'invariant suivant : not_tick_Yes_No. L'outil OBP vérifie, lors de l'exploration du modèle, que l'invariant n'est pas violé.

Annexe B

Automates CDL des contraintes CCSL

B.1 La vérification formelle des propriétés CCSL

Pour établir la vérification d'un ensemble d'exigences sur un modèle, il faut pouvoir, d'une part, l'explorer exhaustivement et disposer d'une expression formelle des propriétés à vérifier sous la forme par exemple de formules logiques ou d'automates observateurs.

Dans l'approche décrite dans (Yin et al., 2011), l'expression des propriétés est basée sur la logique linéaire LTL (Pnueli, 1977) et la vérification s'exécute avec l'outil SPIN. Dans notre approche, nous choisissons (Dhaussy et al., 2012b) d'exprimer les propriétés sous la forme d'automates observateurs. En effet, les formules de logiques linéaires peuvent parfois être difficiles à manipuler dans un contexte industriel. Une exigence peut référencer de nombreux évènements, liés à l'exécution du modèle ou de l'environnement, et peut être dépendante d'un historique d'exécution à prendre en compte au moment de sa vérification. Leur expression par des formules logiques demandent alors une grande expertise de la part des ingénieurs. L'écriture d'observateurs, basée sur le formalisme d'automate bien connu des ingénieurs, est par contre beaucoup plus aisée à manier.

Une fois les observateurs spécifiés, le modèle est ensuite exploré et l'exploration génère un système de transitions (SdT). Celui-ci représente tous les comportements du modèle dans son environnement sous la forme d'un graphe de configurations et de transitions. Sur ce SdT, la vérification des propriétés est conduite en appliquant une analyse d'accessibilité des états d'erreur des observateurs. La difficulté, bien connue, liée à cette technique d'analyse est la production du SdT qui peut être de grande taille, dépassant la taille mémoire disponible (explosion combinatoire).

Le langage CDL permet à l'utilisateur d'exprimer des propriétés sous la forme de prédicats et d'observateurs exprimés sous la forme d'automates. La déclaration des prédicats en CDL peut porter sur des valeurs de variables comme par exemple predicate pred1 is Proc1 : v = value signifiant que pred1 est vrai si la variable v de l'instance 1 du processus Proc est égale à la valeur value. Les prédicats peuvent porter aussi sur l'état des processus. predicate pred2 is Proc1@stateX signifie que pred2 est vrai si l'instance 1 du processus Proc est dans l'état stateX. Les prédicats peuvent porter également sur le nombre de données contenues dans une fifo ou sur une expression booléenne combinant les types de prédicats précédents. Ces possibilités fournissent un mode d'expressivité très riche qui permettent, en complémentarité avec les observateur, d'exprimer aisemment des propriétés qu'il serait plus difficile à exprimer en logique linéaire. Elles permettent d'instrospecter en profondeur le comportement d'un modèle tout en offrant une expression facile à manier et à comprendre pour le concepteur.

B.1.1 Propriétés associées aux contraintes CCSL

Nous illustrons ici l'écriture de quelques propriétés associées a nos contraintes CCSL.

AsFrom : Pour vérifier les propriétés associées a l'exigence AsFrom, nous déclarons (Figure B.1 (a)) les évènements CDL evt_tickIn_Yes (1), evt_tickIn_No (2), et evt_nSupK (3). avec ces évènements, nous spécifions l'observateur, illustré Figure B.1 (b), encodant les propriétés (1), (2), (3) qui vérifient que l'execution de la 2eme horloge respecte et dépend bien de la valeur de k. l'état initial de l'observateur est l'état *Start* et dispose d'un état d'erreur *Reject*. chaque transition de l'observateur est déclenchée par l'occurence d'un évènement.

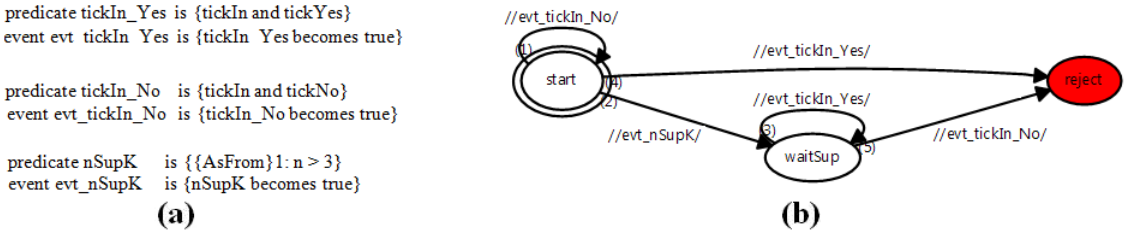

Figure B.1: Automate observateur de la contrainte "AsFrom"

Le langage CDL permet également de spécifier des prédicats qui peuvent être vérifiés lors de l'exploration du modèle. Par exemple, si on souhaite vérifier que, dans un instant, les horloges Yes et No ne tiquent pas dans le même instant, nous pouvons déclarer les prédicats suivants :

predicate tickYes is {{Comp}1:tab_Clocks[1].act_tick = true}

predicate tickNo is {{Comp}1:tab_Clocks[2].act_tick = true}

predicate tick_Yes_No is {tickYes and tickNo}

predicate not_tick_Yes_No is {not (tickYes and tickNo)}

Nous pouvons maintenant déclarer, dans une clause cdl du programme CDL et avec l'opérateur -assert- l'invariant suivant : not_tick_Yes_No. L'outil OBP vérifie, lors de l'exploration du modèle, que l'invariant n'est pas violé.

D'une manière similaire, nous pouvons spécifier des observateurs pour verifier les propriétés associées aux autres contraintes en déclarant les évènements propre a chaque contrainte.

Intersection : Pour vérifier les propriétés associées a l'exigence Intersection, nous déclarons (Figure B.2 (a)) les évènements CDL. avec ces évènements, nous spécifions l'observateur, illustré (Figure B.2 (b)), encodant les propriétés qui vérifient que l'execution de l'horloge Out respecte et dépend bien de la valeur des tiques des horloges (C_1 et C_2). l'état initial de l'observateur est l'état *Start* et dispose d'un état d'erreur *Reject*. chaque transition de l'observateur est déclenchée par l'occurence d'un évènement.

Le langage CDL permet également de spécifier des prédicats qui peuvent être vérifiés lors de l'exploration du modèle. Par exemple, si on souhaite vérifier que, dans un instant, les horloges Yes et No ne tiquent pas dans le même instant, nous pouvons déclarer les prédicats suivants :

predicate tickYes is {{Comp}1:tab_Clocks[2].act_tick = true}

predicate tickNo is {{Comp}1:tab_Clocks[3].act_tick = true}

predicate tick_Yes_No is {tickYes and tickNo}

predicate not_tick_Yes_No is {not (tickYes and tickNo)}

Predicate tickIn_A is {{Comp}1 : tab_clocks [0]. act_tick =true}
Event evt_tickIn_A is {tickIn_A becomes true}

Predicate tickIn_B is {{Comp}1 : tab_clocks [1]. act_tick =true}
Event evt_tickIn_B is {tickIn_B becomes true}

(a)

Predicate tickYes is {{Comp}1 : tab_clocks [2]. act_tick =true}
Event evt_tickYes is {tickIn_B becomes true}

Predicate tickNo is {{Comp}1 : tab_clocks [3]. act_tick =true}
Event evt_tickNo is {tickNo becomes true}

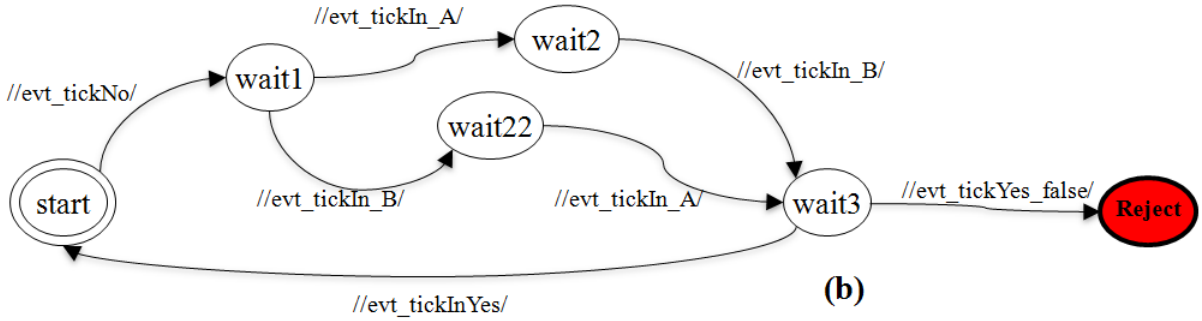

Figure B.2: Automate observateur de la contrainte "Intersection"

Nous pouvons maintenant déclarer, dans une clause cdl du programme CDL et avec l'opérateur -assert- l'invariant suivant : not_tick_Yes_No. L'outil OBP vérifie, lors de l'exploration du modèle, que l'invariant n'est pas violé.

Supérieur : Pour vérifier les propriétés associées a l'exigence "Supérieur", nous déclarons (Figure B.3 (a)) les évènements CDL. avec ces évènements, nous spécifions l'observateur, illustré Figure B.3 (b), encodant les propriétés qui vérifient que l'execution de l'horloge Out respecte et dépend bien de la valeur des tiques des horloges (C_1 et C_2) et l'ordre de ces dernières (lenteur et rapidité) qu'on a représenté avec la variable Décal. l'état initial de l'observateur est l'état *Start* et dispose d'un état d'erreur *Reject*. chaque transition de l'observateur est déclenchée par l'occurence d'un évènement.

Inférieur : Pour vérifier les propriétés associées a l'exigence inférieur, nous déclarons (Figure B.4 (a)) les évènements CDL. avec ces évènements, nous spécifions l'observateur,

predicate tickIn_A is {{Comp}1:tab_Clocks[0].act_tick = true}
event evt_tickIn_A is {tickIn$_A$*becomestrue*}
predicate tickIn_B is {{Comp}1:tab_Clocks[1].act_tick = true}
event evt_tickIn_B is {tickIn_B becomes true}
predicate tickIn_B_false is {{Comp}1:tab_Clocks[1].act_tick = false}
event evt_tickIn_B_false is {tickIn_B becomes true}
predicate tickYes is {{Comp}1:tab_Clocks[2].act_tick = true}
event evt_tickYes is {tickYes becomes true}
predicate tickYes_false is {{Comp}1:tab_Clocks[2].act_tick = false}
event evt_tickYes_false is {tickYes_false becomes true}
predicate tickNo is {{Comp}1:tab_Clocks[3].act_tick = true}
event evt_tickNo is {tickNo becomes true}
predicate decal_sup_0 is {{Sup}1:decal $\leq$ 0}
event evt_decal_sup_0 is {decal_sup_0 becomes true}
predicate decal_inf_0 is {{Inf}1:decal $\geq$ 0}
event evt_decal_inf_0 is {decal_inf_0 becomes true}

(a)

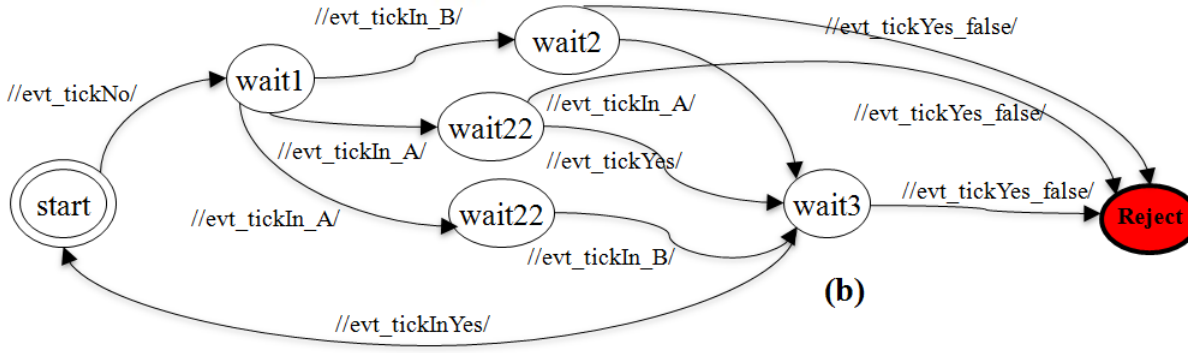

Figure B.3: Automate observateur de la contrainte "Supérieur"

illustré Figure B.4 (b), encodant les propriétés qui vérifient que l'execution de l'horloge Out respecte et dépend bien de la valeur des tiques des horloges (C_1 et C_2) et l'ordre de ces dernières (lenteur et rapidité) qu'on a représenté avec la variable Décal. l'état initial de l'observateur est l'état *Start* et dispose d'un état d'erreur *Reject*. chaque transition de l'observateur est déclenchée par l'occurence d'un évènement.

predicate tickIn_A is {{Comp}1:tab_Clocks[0].act_tick = true}

event evt_tickIn_A is {tickIn$_A$*becomestrue*}

predicate tickIn_A_false is {{Comp}1:tab_Clocks[0].act_tick = false}

event evt_tickIn_A_false is {tickIn_A_false becomes true}

predicate tickIn_B is {{Comp}1:tab_Clocks[1].act_tick = true}

event evt_tickIn_B is {tickIn_B becomes true}

predicate tickIn_B_false is {{Comp}1:tab_Clocks[1].act_tick = false}

event evt_tickIn_B_false is {tickIn_B_false becomes true}

predicate tickYes is {{Comp}1:tab_Clocks[2].act_tick = true}

event evt_tickYes is {tickYes becomes true}

predicate tickNo is {{Comp}1:tab_Clocks[3].act_tick = true}

event evt_tickNo is {tickNo becomes true}

predicate tickIn_Yes is { act_tick_Yes_if_tickIn_A_and_decal_Sup_0 or act_tick_Yes_if_tickIn_A_and_decal_egal_0
}

event evt_tickIn_Yes is {tickIn_Yes becomes true}

predicate tickIn_No is { act_tick_No_if_tickIn_A_and_decal_Sup_0 or act_tick_No_if_tickIn_A_and_decal_egal_0
}

event evt_tickIn_No is {tickIn_No becomes true}

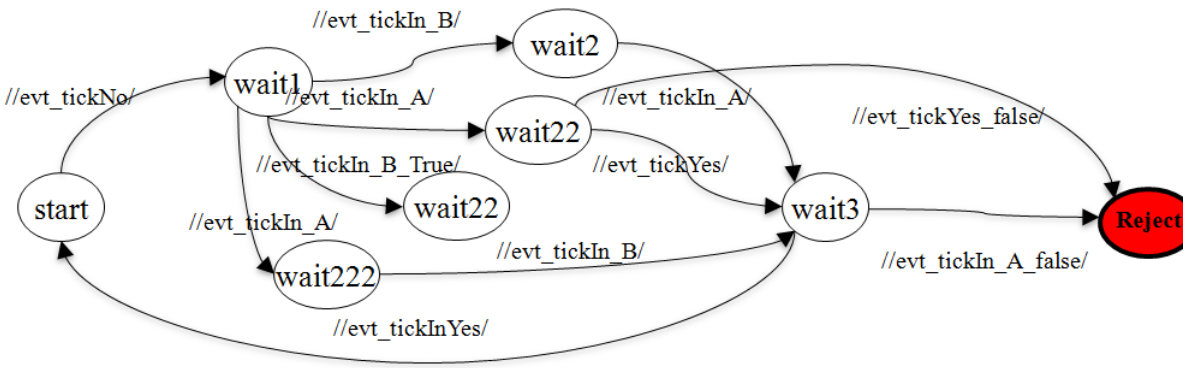

Figure B.4: Automate observateur de la contrainte "Inférieur"

Union : Pour vérifier les propriétés associées a l'exigence d'Union, nous déclarons (Figure B.5 (a)) les évènements CDL. avec ces évènements, nous spécifions l'observateur, illustré (Figure B.5 (b)), encodant les propriétés qui vérifient que l'execution de l'horloge Out respecte et dépend bien de la valeur des tiques des horloges (C_1 et C_2). l'état initial de l'observateur est l'état *Start* et dispose d'un état d'erreur *Reject*. chaque transition de l'observateur est déclenchée par l'occurence d'un évènement.

predicate tickIn_A is {{Comp}1:tab_Clocks[0].act_tick = true}
event evt_tickIn_A is {tickIn$_A$ *becomestrue*}
predicate tickIn_B is {{Comp}1:tab_Clocks[1].act_tick = true}
event evt_tickIn_B is {tickIn_B becomes true}
predicate tickIn_B_false is {{Comp}1:tab_Clocks[1].act_tick = false}
event evt_tickIn_B_false is {tickIn_B_false becomes true}
predicate tickYes is {{Comp}1:tab_Clocks[2].act_tick = true}
event evt_tickYes is {tickYes becomes true}
predicate tickYes_false is {{Comp}1:tab_Clocks[2].act_tick = false}
event evt_tickYes_false is {tickYes_false becomes true}
predicate tickNo is {{Comp}1:tab_Clocks[3].act_tick = true}
event evt_tickNo is {tickNo becomes true}

(a)

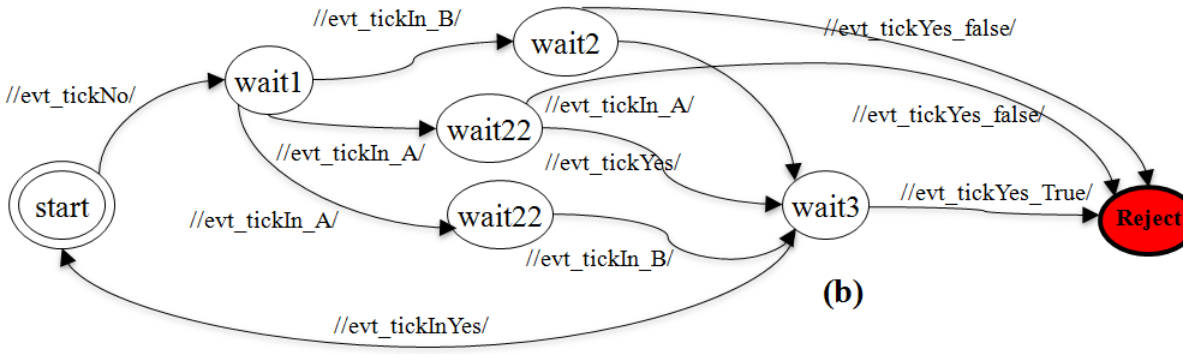

Figure B.5: Automate observateur de la contrainte ”Union”