Université de Montréal

Mécanismes d'introspection pour la vérification semi-formelle de modèles au niveau système

par
Michel Metzger

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M. Sc.)
en Informatique

Février, 2006

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Mécanismes d'introspection pour la vérification semi-formelle de modèles au niveau système

présenté par :

Michel Metzger

a été évalué par un jury composé des personnes suivantes :

Abdelhakim Hafid
président-rapporteur

El Mostapha Aboulhamid
directeur de recherche

Julie Vachon
codirectrice

Frédéric Rousseau
membre du jury

Mémoire accepté le 24 avril 2006

Résumé

Avec la complexité grandissante des systèmes numériques, le processus de vérification prend une part de plus en plus importante dans le cycle de développement. La vérification semi-formelle est une tendance récente qui vise à intégrer des méthodes formelles pour vérifier la validité d'une simulation.

Ce mémoire présente un environnement de vérification semi-formelle pour ESys.Net. ESys.Net est un outil de modélisation et de simulation au niveau système construit sur la plateforme Microsoft .NET. Notre travail est basé sur la logique temporelle linéaire et sur la théorie des automates qui lui est associée. Nous tirons partie des fonctionnalités avancées de .NET, telles que l'introspection et la programmation basée sur les événements.

Nous présentons les formalismes et l'implémentation de l'outil, ainsi qu'un ensemble d'interfaces graphiques facilitant la spécification des propriétés et l'exploitation des résultats en fin de simulation.

Une étude de cas et une analyse des performances mettent en valeur la pertinence de notre approche.

Mots-clefs : Systèmes électroniques, vérification de modèle, vérification semi-formelle, vérification basée sur les assertions, logique temporelle linéaire.

Abstract

As the modern digital systems' complexity increases, the part of the verification task in the development cycle is becoming more and more significant. Semi-formal verification is a recent trend which aims to bring formal verification methods to simulation.

This paper presents a semi-formal verification environment for ESys.Net. ESys.Net is a system-level modeling and simulation tool built on top of the Microsoft .Net framework. Our work is based on linear temporal logic and the related automata theory. We take advantage of .NET's advanced programming features such as introspection and event-driven programming to provide a flexible verification environment.

Formalisms and the implementation of our tool are presented, along with graphical user interfaces developed to facilitate the specification of properties and the examination of results at the end of a simulation. A case-study and a performances analysis are detailed to emphasis the relevance of our approach.

Keywords: Electronic systems, model verification, semi-formal verification, assertion-based verification, linear temporal logic.

Table des matières

Liste des figures

Liste des tableaux

Liste des codes sources

Liste des sigles et abréviation

AHB	*Advanced High-performance Bus*
AMBA	*Advanced Microcontroller Bus Architecture*
API	Interface de Programmation d'Applications, *Application Programming Int.*
AST	Arbre Syntaxique Abstrait, *Abstract Syntax Tree*
BA	Automate de Büchi, *Büchi Automaton*
CLI	*Common Language Infrastructure*
CLS	*Common Language Specification*
CTL	*Computation Tree Logic*
DST	*Design Structure Tree*
ECMA	*European Computer Manufacturers Association*
GBA	Automate de Büchi Généralisé, *Generalized Büchi Automaton*
GDL	*General Description Language*
HDL	*Hardware Description Language*, Langage de description de matériel
IEEE	*Institute of Electrical and Electronics Engineers*
IP	*Intellectual Property*
ISO	*International Organization for Standardization*
LALR	*Look-Ahead, Left to rightmost derivation*
LGBA	Automate de Büchi Généralisé Étiqueté, *Labeled Generalized Büchi A.*
LL	*Left to right, leftmost derivation*
LTL	Logique Temporelle Linéaire, *Linear Temporal Logic*
LTL2BA	*Linear Temporal Logic to Büchi Automaton*
MTL	*Metric Temporal Logic*
OBE	*Optional Branching Extension*
OSCI	*Open SystemC Initiative*
OVA	Assertions OpenVera, *OpenVera Assertions*
POA	Programmation Orientée Aspect
POO	Programmation Orientée Objet
PSL	*Property Specification Language*
RTL	Niveau Transfert de Registre, *Register Transfer Level*
RTLTL	Logique Temporelle Linéaire Temps-Réel, *Real-Time Linear Temporal*
SCV	*SystemC Verification Standard*
SDL	*System Description Language*, Langage de description de système
SERE	*Sequential Extended Regular Expression*
SVA	*SystemVerilog Assertions*
SVC	*SystemC Verification*
VES	*Virtual Execution System*
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*
GPVW95	*Gerth, Peled, Vardi, Wolper 1995,* Algorithme de transformation de formules LTL en automate de Büchi

Remerciements

Je tiens tout d'abord à remercier El Mostapha Aboulhamid, Nicolas Gorse et James Lapalme, les initiateurs de ce projet, pour leurs conseils et leur soutien.

J'aimerai également remercier Julie Vachon et Frédéric Rousseau pour leur aide et leurs contributions.

Introduction

La complexité grandissante des systèmes électroniques est un réel défi pour les développeurs d'outils de conception. D'après la loi de Moore, le nombre de transistors dans un circuit intégré double tous les dix huit mois. Depuis longtemps, ces systèmes ont atteint une taille telle qu'ils ne sont plus compréhensibles dans leur ensemble par un être humain. Ce problème a été résolu en augmentant le niveau d'abstraction, comme il l'a été dans les langages de programmation. Dans la conception de systèmes électroniques, le niveau d'abstraction du modèle est de plus en plus élevé et la transformation du modèle vers des niveaux d'abstraction plus bas est automatisée au maximum.

La vérification tient un rôle primordial dans le flux de conception de systèmes numériques. Contrairement aux logiciels, une erreur dans du matériel en production n'est pas corrigible simplement. La seule solution possible est bien souvent le changement intégral du système, avec les coûts que cela engendre. Par exemple, une erreur de conception dans la première version du processeur Pentium a coûté environ un million de dollars à Intel. Dans les systèmes embarqués critiques, le risque peut être la perte de vies humaines, ou la destruction d'équipements coûteux.

1.1. La conception de matériel

Une première étape dans l'élévation du niveau d'abstraction a été l'introduction dans les années 80 des langages de description de matériel comme VHDL (Very High speed integrated circuit Hardware Description Language) et Verilog. Le premier objectif de VHDL était la documentation de systèmes électroniques. Il a rapidement été complété pour pouvoir être simulé et permettre la vérification du comportement du modèle avant la fabrication. L'étape suivant fut la synthèse directe à partir d'un modèle VHDL, c'est-à-dire la construction automatique d'une implémentation sous forme de portes logiques.

Par rapport à des langages de programmation impératifs, les langages de description de matériel introduisent les concepts suivants :

- Le parallélisme des opérations. Sur du matériel, toutes les opérations sont concurrentes, contrairement aux instructions logicielles qui sont fondamentalement séquentielles ;
- Le découpage hiérarchique du système sous forme d'entités exposant une interface ;
- La communication entre les entités ;
- Les contraintes de temps.

Ces notions se retrouvent dans la plupart des langages de modélisation.

VHDL et Verilog sont principalement utilisés pour de la modélisation au niveau transfert de registres (RTL – *Register Transfer Level*), où le système est décrit par un ensemble de registres et par les fonctions combinatoires qui les connectent. Ce niveau d'abstraction est bien adapté pour la synthèse, mais il est trop bas pour la modélisation au niveaux système [1].

Ces dix dernières années, le développement spectaculaire des systèmes embarqués a provoqué un changement dans les méthodes de développement de système. Historiquement, les parties matériels et logicielles étaient développées séparément. La modélisation au niveau système permet le développement conjoint du matériel et du logiciel et facilite l'exploration de différentes architectures pendant la conception. Avec ces nouvelles techniques sont apparues de nouveaux langages, les langages de description au niveau système (SDL – *System-level Description Langage*). Ils permettent entre autre l'abstraction des communications entre les blocs qui composent le système et facilitent le développement simultané du matériel et du logiciel. Parmi ces langages on retrouve SystemVerilog [2], SystemC [3] et ESys.Net [4].

Les recherches récentes [5, 6] sur les SDL ont montré l'importance d'utiliser des environnements de programmation modernes et standard – tels que .NET [7] ou Java [8] – pour le développement d'outils de modélisation et de simulation au niveau systèmes. Les fonctionnalités de programmation avancées proposées par ces environnements peuvent grandement faciliter la conception matériel/logiciel en permettant l'intégration de composants logiciels dans le système. Le développement d'outils autours des environnements de conception est également simplifié par l'utilisation d'un format intermédiaire standard.

Comme nous l'avons vu, le coût des erreurs sur des systèmes électroniques est très élevé. Pour tenter les réduire, plusieurs techniques de vérification ont été mises au point. Dans la conception de matériel, la vérification est la tâche qui consiste à s'assurer qu'un système donné ou qu'un modèle de ce système implémente le comportement voulu. La principale vérification effectuée sur un système concerne ses fonctionnalités, mais il existe d'autres types de vérification portant par exemple sur le respect des contraintes de temps ou de consommation. Dans la suite de ce mémoire, nous nous consacrerons uniquement à la vérification fonctionnelle.

D'après l'International Technology Roadmap for Semiconductors [1], la vérification est aujourd'hui la tâche la plus coûteuse du cycle de développement d'un système, aussi bien en termes de temps, d'argent ou de ressources humaines. Sur les projets les plus complexes, la vérification emploie environ deux à cinq fois plus d'ingénieurs que la conception elle-même. Ce déséquilibre peut s'expliquer de plusieurs façons. Premièrement, la loi de Moore prévoit une progression exponentielle du nombre de transistors. Mais la progression du nombre de configuration distinctes à vérifier croît d'une manière doublement exponentielle [1]. De plus, le processus de vérification est faiblement automatisé par rapport aux autres tâches comme la synthèse ou le placement-routage.

À l'heure actuelle, la technique la plus utilisée est la vérification par simulation du modèle. Elle n'explore que partiellement l'espace d'états du modèle, avec des durées de simulation souvent très longues. L'ITRS prédit ainsi que,

> *«Without major breakthroughs, verification will be a non-scalable, show-stopping barrier to further progress in the semiconductor industry.»*

De nouvelles méthodes, algorithmes et structures de données sont nécessaire pour résoudre ce problème. L'avancée la plus prometteuse semble venir de l'introduction de méthodes formelles et semi-formelles dans le flux de conception. Une méthode formelle est une preuve mathématique exhaustive qu'un système vérifie une propriété. Les méthodes formelles ne sont pour l'instant applicables que sur des sous-systèmes. Elles sont généralement incapables de gérer des systèmes complets avec les niveaux d'abstraction requis actuellement. Ce problème se réduit d'année en année avec les progrès des techniques formelles et avec l'augmentation de la puissance des machines. Mais les

techniques traditionnelles telles que la simulation restent indispensables. Une tendance récente est l'introduction de méthodes formelles dans les processus de vérification classiques. Parmi ces méthodes semi-formelles se trouve la vérification par assertions *(assertion-based verification)* ou vérification par observateurs. Dans cette technique, le modèle est simulé et son évolution au cours du temps est vérifiée par des méthodes formelles. Ces méthodes sont bien entendues moins « fortes » que les méthodes formelles, mais elles ont l'avantage de pouvoir s'intégrer simplement dans le flux de conception actuel, sans nécessiter un autre niveau d'abstraction. Cependant, il est recommandé d'utiliser la vérification fonctionnelle à un plus haut niveau d'abstraction dans le flux de conception [1, 9]. Cette vérification se faisait auparavant au niveau des portes logiques. Actuellement, elle est effectuée au niveau RTL, mais pour satisfaire les contraintes des systèmes actuels, des technologies standardisées pour la vérification au niveau système sont nécessaires.

Pour finir, les tâches de vérification doivent être automatisées au maximum et les concepteurs doivent pouvoir utiliser les outils de vérification sans connaissances des théories mathématiques sous-jacentes [9]. Les méthodes actuelles de vérification formelle ne sont que difficilement exploitables par des ingénieurs qui n'on pas une connaissance profonde des concepts mathématiques employés. Même si une meilleure formation des ingénieurs dans ce domaine permettrait de changer cette situation, un perfectionnement des outils est indispensable [9]. Il faut notamment les rendre plus accessibles en simplifiant les formalismes et en proposant par exemple à l'utilisateur des interfaces graphiques adaptées.

1.2. Contribution

L'objectif principal de ce projet est de proposer un environnement de vérification semi-formelle pour ESys.Net. Bien qu'ESys.Net soit un environnement de modélisation au niveau système, la vérification fonctionnelle est concentrée sur l'aspect « modélisation de matériel ». Notre outil repose sur la logique temporelle linéaire (LTL) et sur le concept des observateurs pour vérifier que le modèle en cours de simulation respecte le comportement désiré par le concepteur. Ce comportement est formalisé par un ensemble de formule LTL qui sont transformé en automates. Ces derniers sont exécutés en parallèle de la simulation et détectent les comportements contraires aux spécifications.

L'observation du modèle emploie les mécanismes d'introspections offerts par l'environnement .NET. Ces mécanismes permettent d'accéder à la structure des modèles ESys.NET et d'observer l'évolution de l'état du modèle, c'est-à-dire les valeurs des différents signaux, registres et variables composant le système. Cette technique est au cœur de la vérification fonctionnelle par simulation.

Sur le plan de la vérification, nous étudions l'utilisation d'algorithmes et de méthodes de vérification formelle dans le cadre de la vérification par simulation. Un aspect important est l'impact du caractère fini d'une simulation sur le résultat de la vérification. Les algorithmes de construction des automates à partir de formules en logique temporelle sont également abordés.

En ce qui concerne les environnements de conception de systèmes embarqués, nous démontrons la pertinence de l'approche d'ESys.NET. L'intérêt de l'introspection dans un environnement de modélisation et de simulation est particulièrement mis en avant ; de même que les avantages offerts par l'architecture du simulateur d'ESys.NET en termes d'extensibilité et d'interopérabilité.

Ces travaux ont fait l'objet de trois publications dans des conférences internationales [10-12].

1.3. Organisation du mémoire

Dans le Chapitre 2, nous donnons un aperçu des techniques de vérification existantes, ainsi qu'une présentation des outils et des langages actuellement employés ou en cours d'élaboration. L'environnement de développement .NET et ESys.Net sont également présenté en détail.

Les bases formelles de notre environnement sont données dans le Chapitre 3. Nous présentons les différentes logiques utilisées et leurs sémantiques. Les algorithmes de vérification sont également détaillés.

Le chapitre principal de ce mémoire est le Chapitre 4. Il contient les détails de l'architecture et de l'implémentation de notre outil de vérification. Le langage de

spécification que nous avons conçu y est présenté. Enfin, la méthode d'exécution de l'outil en parallèle de la simulation est décrite.

Dans le Chapitre 5, les interfaces graphiques proposées à l'utilisateur sont présentées.

Le Chapitre 6 présente l'étude de cas qui a été effectuée avec notre outil. Elle présente également une analyse des performances de l'application et les optimisations effectuées.

Enfin, le contenu de ce mémoire est résumé dans un dernier chapitre. Nous proposons également quelques idées pour les évolutions de ce projet.

Chapitre 2 État de l'art

La complexité grandissante des systèmes numériques implique un niveau d'abstraction de plus en plus élevé, pour la modélisation autant que pour la vérification. Les techniques de modélisation s'éloignent de plus en plus de l'implémentation physique du système pour se rapprocher d'une vue plus symbolique, accessible à l'esprit humain. Il en va de même pour la vérification. Si la vérification manuelle d'une trace de simulation peut suffire pour un système peu complexe, la taille des architectures actuelles rend cette méthode difficilement exploitable. Ainsi, des méthodologies de vérification plus automatiques sont apparues.

Ce chapitre se compose de quatre parties. La première présente les différentes techniques de vérification de modèles. Dans la deuxième partie, nous introduisons les environnements de modélisation et de vérification, en mettant l'accent sur leurs fonctionnalités de vérification par simulation. Les troisièmes et quatrièmes parties introduisent les deux solutions logicielles sur lesquelles se base le projet : la plateforme de développement .NET et l'environnement de modélisation et de simulation ESys.NET.

2.1. Vérification de modèles

L'objectif de la vérification de modèle est d'assurer la conformité d'un modèle par rapport à un ensemble de propriétés décrivant des comportements ou des fonctionnalités désirées du système final. On distingue deux grandes tendances dans la vérification : la vérification formelle et la vérification par simulation.

2.1.1 Vérification formelle

La vérification formelle vise à produire une preuve exhaustive de la validité du modèle par rapport à la spécification. Pour cela, elle nécessite une description formelle du système lui-même mais aussi des propriétés désirées. La sémantique de ces descriptions doit être clairement définie et lever tout ambiguïté. On distingue principalement trois types

de méthodes de vérification [13, 14] : Les systèmes de déductions, les approches basées sur les automates et les techniques basée sur la modélisation du comportement du modèle.

Les systèmes de déductions déduisent, à partir d'un ensemble de règle d'inférences, les conséquences d'un ensemble d'axiomes. Le système est spécifié dans des logiques classiques comme la logique du premier ordre ou dans des logiques d'ordre plus élevées. Produire une preuve pour un système complexe peut être fastidieux et des erreurs peuvent rapidement être introduites dans la preuve. Des techniques automatiques de preuves permettent de s'assurer de la validité de la preuve. Les vérificateurs de preuves (*proof checkers*) garantissent qu'une déduction est correcte. Les « prouveurs » de théorèmes (*theorem provers)* [15, 16] utilisent des techniques heuristiques pour produire la preuve mais nécessitent l'intervention de l'utilisateur pour produire une preuve complexe.

Dans l'approche basée sur les automates, le système est modélisé sous la forme d'une machine à états finis. Ces techniques de vérification permettent uniquement de prouver l'équivalence de deux automates, par exemple la validité d'une implémentation matérielle par rapport à sa spécification à plus haut niveau. Mais elles ne peuvent pas servir à vérifier qu'un système vérifie bien une fonctionnalité donnée.

La troisième approche permet de prouver que certaines propriétés sont vérifiées par une modélisation mathématique du comportement du système. La nécessité de spécifier un modèle précis du système permet déjà de mettre en évidence des lacunes, des ambiguïtés ou des contradictions dans la spécification. De plus, ces techniques sont, contrairement au prouveurs de théorèmes, totalement automatiques. La vérification par évaluation sur un modèle (*model-checking*) [17] fait partie de cette catégorie. Dans le *model-checking*, le système est modélisé par un automate représentant la totalité des états possibles du système et les différentes transitions possibles entre ces états. La propriété à vérifier est formalisée dans une logique temporelle. Ces logiques permettent d'exprimer l'évolution des systèmes au cours du temps. L'idée d'utiliser une logique particulière pour formaliser un comportement dynamique remonte au moins à [18] en 1977. La logique temporelle est formée de propositions dont la valeur booléenne évolue au cours du temps et d'opérateurs temporels tels que « jusqu'à », « fatalement », « toujours », « suivant », etc. Il existe un grand nombre de logiques temporelles, qui diffèrent par leur expressivité et la complexité

des algorithmes de vérification. On distingue principalement CTL (*Computation Tree Logic*) et LTL (*Linear Temporal Logic*). CTL [19] permet d'exprimer des comportements sur l'arbre d'exécution du système. Le modèle temporel considéré est donc non linéaire : on peut exprimer des propriétés où le temps se divise en plusieurs cours. Le *model-checking* de CTL est simple : son exécution se fait en temps polynomial sur la taille de la formule [19]. Le principal inconvénient de CTL est la difficulté d'exprimer certaines propriétés (voir par exemple [20]). La logique temporelle linéaire [18] (LTL ou bien PTL pour *Propositional Temporal Logic*) exprime des propriétés sur un seul chemin d'exécution. La taille du modèle joue un rôle important dans la complexité de la vérification. Les algorithmes de *model-checking* rencontrent le problème dit d'explosion du nombre d'états qui limite leur utilisation. Ces méthodes nécessitent un modèle très abstrait du système à vérifier afin de pouvoir effectuer l'exploration du domaine des exécutions possibles dans un espace mémoire raisonnable.

Les méthodes de vérification formelles requièrent une grande expertise de la part du concepteur. Même dans le cas des méthodes complètement automatiques, la formalisation des propriétés ainsi que la modélisation du système implique une bonne connaissance des algorithmes utilisés pour faire la preuve. La vérification formelle peut difficilement être utilisée pour la totalité du développement [14]. Elle est cependant utilisée pour la vérification de sous-systèmes comme des unités de calcul d'un microprocesseur, des protocoles de cohérence de cache ou des bus de communication [14, 21].

2.1.2 Vérification par simulation

La vérification formelle, et en particulier le *model-checking*, nécessitent une exploration de la totalité des exécutions possibles du système et donc un modèle extrêmement abstrait. Dans le processus de conception de matériel, les modèles utilisés et les langages utilisés sont souvent de trop bas niveau pour qu'une exploration de la totalité des exécutions soit faisable.

La technique de vérification la plus utilisée dans le domaine est donc la vérification par simulation. Les ports d'entrées du modèle sont stimulés par des vecteurs de test et son comportement est observé pour en vérifier la validité [22]. Le taux de couverture des vecteurs de test détermine la pertinence de la vérification. Les vecteurs de tests ciblant une

propriété particulière sont utiles, mais pour s'assurer que toutes les zones du modèle sont explorées, des tests pseudo-aléatoires sont utilisés conjointement aux tests dirigés. Des contraintes peuvent également être appliquées sur les valeurs aléatoires pour réduire les temps de simulation.

Suivant le niveau d'observabilité du modèle, on distingue deux types de vérification. Dans la première, la vérification par boîte-noire (blackbox), seules les sorties du système sont observées. Cette approche est parfois nécessaire pour vérifier des blocs IP dont ne connaît pas le contenu. La seconde, la vérification boîte-blanche (whitebox), permet de vérifier le comportement des composants internes du modèle. Elle nécessite d'avoir accès à un environnement de simulation permettant l'observation du contenu du modèle pendant la simulation. L'analyse du résultat de la simulation est souvent difficile, en partie à cause de la complexité des traces de simulation.

La vérification par assertions est une tendance récente pour améliorer l'analyse du comportement et du taux de couverture [23]. Une assertion, également appelée moniteur ou observateur, est une description formelle d'un comportement désiré dans le modèle. Les outils vérifient automatiquement pendant la simulation que les assertions sont respectées. Ces dernières années, beaucoup de travaux [24-26] ont appliqué certains concepts de la vérification formelle à la vérification basée sur les assertions pour former les techniques de vérification dite semi-formelles. Ainsi, la logique temporelle linéaire est utilisée pour exprimer des comportements complexes du modèle dans le temps. De tels comportements seraient difficilement vérifiables avec une simple analyse manuelle de la trace.

Bien évidemment cette vérification est beaucoup moins forte que celle des méthodes formelles. Mais puisque la vérification se fait sur une simulation du système, la taille du modèle n'est plus un critère limitatif de l'approche. Elle n'aura de fait un impact que sur le temps d'exécution de la simulation. La description du modèle peut ainsi se faire dans les langages de description de matériel plus classiques tels que VHDL [27] , SystemVerilog [27] et SystemC [3, 28].

Les deux approches – vérification formelle et semi-formelle – ne sont pas totalement antagonistes. D'une part parce sont utilisées conjointement à différents niveaux

dans le cycle de développement. Et d'autre part parce qu'elles sont étroitement liées d'un point de vue théorique.

2.2. Langages et outils pour la vérification par simulation

La vérification prend de plus en plus d'importance dans les environnements de modélisation et de simulation de matériel. Ces dernières années, plusieurs extensions aux langages de description de matériel ont été standardisées pour améliorer la vérification. Dans cette partie, nous allons présenter les principaux langages de modélisation et de vérification.

2.2.1 VHDL et Verilog

Le langage VHDL (*Very High Speed Integration Circuit Hardware Description Language*) [29, 30] a été développé au début des années 1980 par le Département de Défense des Etats-Unis. Sa syntaxe est basée sur celle d'ADA et ajoute les notations nécessaires pour la description de matériel à plusieurs niveaux d'abstraction. Même s'il a été développé pour la simulation, un sous ensemble normalisé du langage est synthétisable et c'est aujourd'hui un des langages les plus utilisés pour la synthèse de matériel. Le modèle est découpé en modules, dont l'interface est séparée de l'implémentation via les concepts d'entités et d'architectures. Le comportement du modèle est défini dans un ensemble de processus qui communique à travers des signaux. Les structures proposées par VHDL restent d'assez bas niveau. C'est un langage de description purement matériel, même s'il permet quelques opérations de haut niveau comme la lecture ou l'écriture de fichiers. L'absence de primitives logicielles et de constructions orientées objets ainsi que la difficulté d'interaction avec des programmes C ou C++ font que VHDL n'est pas adapté à la modélisation au niveau système [30]. En ce qui concerne la vérification fonctionnelle, VHDL ne propose que des assertions propositionnelles, comme dans les langages de programmation classique. Ces assertions permettent la vérification de l'état immédiat du modèle, mais ne peuvent pas décrire un comportement dans le temps. Malgré ces inconvénients, VHDL reste très utilisé dans l'industrie. De nombreux standards sont développés pour pallier l'absence de structures de vérification.

Verilog [31] est le concurrent historique de VHDL. Par rapport à ce dernier, il se différencie essentiellement par un typage faible, l'impossibilité de définir des types

utilisateurs et l'absence de la notion d'interface [32]. Aucune construction n'est prévue pour la vérification. Il autorise cependant la création dynamique de processus et facilite les interactions avec d'autres langages via une API standard en C. Cette API permet d'étendre simplement le langage et l'environnement de simulation et par exemple lui ajouter des fonctionnalités de vérification ou d'analyse du taux de couverture.

2.2.2 *Property Specification Language*

PSL est un langage de spécification de propriétés indépendant du langage de modélisation. Son objectif est de fournir un langage standard pour la vérification par simulation et pour la vérification formelle [25, 33]. Il a été développé à partir de 1994 par le laboratoire d'IBM à Haifa, sous le nom de *Sugar* [34]. L'objectif de départ était d'ajouter des sucres syntaxiques à la logique temporelle CTL pour faciliter la définition de propriétés pour des systèmes numériques. Un sucre syntaxique est une extension à la syntaxe d'un langage qui ne modifie pas son expressivité. En 2001, l'organisation Accellera [35] cherchait à standardiser un langage de description de propriétés compatible avec VHDL et Verilog. Accellera considéra plusieurs langages existants dont *Sugar*, *Temporal e* [36, 37] (voir partie 2.2.6) et *ForSpec* [38] (un langage de spécification basé sur la logique temporelle linéaire). C'est *Sugar* qui est sélectionné en 2002, après l'ajout du support de la logique temporelle linéaire. En septembre 2005, PSL/Sugar est normalisé par l'IEEE [39].

Une des exigences d'Accellera était l'indépendance vis-à-vis du langage de description du modèle. PSL permet cette indépendant grâce au concept de « saveurs » *(flavors)* [33]. La syntaxe d'une partie du langage varie en fonction du langage utilisé pour la modélisation. Actuellement, PSL supporte quatre HDL : VHDL, Verilog, SystemVerilog et GDL (General Description Language [40]). L'ajout de SystemC est en cours [39]. La saveur permet notamment de définir la syntaxe de certains opérateurs.

Le langage est découpé en quatre couches. Les expressions d'une couche donnée sont construites d'expressions des couches inférieures et/ou de sous expressions de la même couche.

Couche booléenne. Elle exprime l'état du modèle à un instant donné. Une expression de la couche booléenne est donc évaluée immédiatement. Elle peut être composée d'expressions du HDL ciblé, d'expressions PSL ou bien d'appels à des fonctions prédéfinies de PSL. La

couche booléenne permet également de définir une ou plusieurs horloges pour l'échantillonnage des variables et définir ainsi la notion de cycle utilisé dans les couches supérieures.

Couche temporelle. Elle est utilisée pour définir des propriétés qui décrivent le comportement du modèle dans le temps. Les propriétés sont formées d'expressions booléennes, d'expressions séquentielles et de formules de logiques temporelles.

Les expressions séquentielles (*SERE, Sequential Extended Regular Expression*) sont des expressions rationnelle construites sur des expressions booléennes. Par exemple l'expression `{req;[*];ack}` est une séquence spécifiant qu'une requête doit être suivie d'un acquittement.

PSL possède deux classes de formules temporelles. La première, nommée FL (*Foundation Language*) est dérivé de LTL et permet d'exprimer des propriétés sur une séquence d'états. La seconde, OBE (*Optional Branching Extension*) est plus ou moins équivalente à CTL et travail sur les arbres d'exécutions. Elle se destine plus à la vérification formelle. PSL supporte les opérateurs classiques de LTL (toujours, fatalement, suivant, jusqu'à, ...).

Couche de vérification. Elle contient les directives et les unités de vérification.

Les directives de vérification indiquent à l'environnement de vérification ce qu'il doit faire avec les propriétés. Les trois directives principales sont `assert`, `assume` et `cover`. La première indique que cette propriété doit être vérifiée. La deuxième permet d'indiquer à l'environnement qu'il ne doit considérer que les comportements qui sont valides par rapport à cette contrainte. Cette directive est souvent utilisée pour contraindre les signaux d'entrée du modèle. Enfin, la directive `cover` permet de s'assurer qu'un scénario spécifié par une séquence a été couvert.

Les unités de vérification permettent de préciser comment les propriétés sont liées avec le modèle en associant une unité de vérification avec un module du système.

La couche de modélisation. Elle permet de définir le comportement des entrées du modèle et de déclarer des signaux et des variables supplémentaires ainsi que d'en définir le comportement. Ceux-ci pourront être utilisés dans la vérification.

PSL est prévu pour que la spécification soit séparée du modèle, via le concept d'unité de vérification. Mais certains outils proposent de l'embarquer dans le code, sous la forme de commentaires qui sont interprétés par le compilateur [41].

De nombreux outils supportent PSL [42]. On peut citer ModelSim de Mentor Graphics [43] qui fait de la vérification par simulation ou Solidify d'Averant qui fait de la vérification formelle. L'outil FoCs d'IBM transforme automatiquement des propriétés PSL en vérificateurs écrits en VHDL ou Verilog [44]. Ces vérificateur peuvent ensuite être intégrés au modèle pour de la vérification par simulation.

2.2.3 SystemC

Publié en 1999 par l'OSCI (Open SystemC Initiative) SystemC [3, 45] est un langage de description de systèmes à code source ouvert. Il est bâti sur le langage C++ et propose une bibliothèque de classes définissant les constructions de base nécessaires à la modélisation (modules, processus, signaux,...). Le modèle est compilé avec un compilateur c++ classique et l'exécutable obtenu est une version simulable du modèle. La sémantique du langage est définie par rapport à cette simulation. SystemC définit des classes de base pour les éléments de modélisation au niveau système. Le modèle est constitué d'une hiérarchie de modules dont le comportement est défini par des processus. Les modules possèdent des ports d'entrée/sortie qui permettent de les connecter via des canaux de communication.

SystemC lui-même ne définit pas de construction pour la vérification, mais l'OSCI a développé un standard spécifique, SCV (*SystemC Verification Standard*) [46, 47] qui permet de faire de la vérification basée sur les transactions [48]. C'est une méthode de vérification boite noire : on stimule les entrées du modèle avec un banc de test et on observe le résultat sur les sorties. L'objectif de cette méthodologie est d'élever le niveau d'abstraction de l'infrastructure de test. Le système de vérification est composé de trois parties : Les tests, les « transacteurs » (transactor) et le modèle à vérifier. Les transacteurs font l'interface entre les tests et le modèle à bas niveau (RTL). La communication entre les tests et les transacteurs se fait via des tâches. Ces tâches sont des commandes de haut niveau, indépendantes des signaux du modèle. Les transacteurs font la conversion entre les

tâches et les signaux du modèle. Ils enregistrent également la trace des transactions pour une analyse ultérieure.

Pour automatiser au maximum les tests, le système de vérification doit être capable de manipuler des types de données dont il ne connaît pas la structure à la compilation. Pour cela, il utilise les techniques d'introspection qui lui permettent, entre autre, d'obtenir à l'exécution des informations sur les types de données. L'introspection de C++ est limitée et SCV définie donc un API qui offre des fonctionnalités d'introspection étendues sur les types primitifs de C++, sur ceux de SystemC et sur les types définis par l'utilisateur. Les informations sur le type comme le nombre de champs et leurs tailles sont accessibles via cette API. Des méthodes de génération de données aléatoires avec des contraintes sur les valeurs et sur leur distribution sont également proposées pour faciliter la génération de vecteurs de tests. Enfin, l'API d'introspection de SCV permet de lier des fonctions de rappel (*callbacks*) sur des objets. Ces fonctions seront déclanchées à chaque changement de valeur. Toutes ces techniques facilitent la définition de tests et de transacteurs génériques, relativement indépendant des types de données utilisés.

SCV standardise une méthode de construction de bancs de test mais ne propose pas de méthode pour la vérification du modèle pendant la simulation. Cet aspect doit être couvert par PSL. En 2005, le groupe de travail de PSL à l'IEEE annonce l'ajout d'une saveur SystemC à PSL [27, 39]. Plusieurs projets proposent de la vérification par assertion avec PSL [41, 49, 50].

Malgré son support important par l'industrie, SystemC est limité par le manque de solutions standard pour l'introspection de C++ [5]. Pour la vérification, ce problème est critique puisque c'est l'introspection qui permet de comprendre la structure du modèle et d'en observer l'état pendant la simulation. Pour palier ce manque, l'utilisation d'analyseurs lexical et syntaxique (le *front-end* d'un compilateur) pour C++ est souvent nécessaire. Plusieurs approches sont possibles dans ce cas. La première est d'utiliser un *front-end* commercial comme EDG [51]. C'est ce que font certains outils commerciaux comme ceux de Synopsys ou Cadence. On peut également réutiliser un *front-end* déjà existant. Le projet PINAPA [52] utilise une version modifiée du compilateur GCC pour extraire la structure du modèle ainsi que le contenu des processus. SystemCXML [53] utilise Doxygen [54]

pour construire un fichier XML contenant les informations sur la structure du modèle : hiérarchie des modules, les signaux et leurs types, les ports ainsi que l'interconnexion des modules. Enfin, certains outils utilisent des *front-ends* développés spécialement pour SystemC [55]. Quelques soit la technique utilisée pour fournir l'introspection à SystemC, elle nécessite d'avoir les sources du modèle et de les analyser. Cette contrainte peut être problématique pour vérifier des modèles dont les sources sont inaccessibles.

2.2.4 SystemVerilog

SystemVerilog, adopté par IEEE comme standard en 2005, est une extension de Verilog [26, 30, 56]. L'objectif est de fournir les structures nécessaires à la modélisation de systèmes et à la vérification [32]. En ce qui concerne la modélisation, la structure des types a été entendu pour supporter un typage fort et la définition de types par utilisateur. Des constructions pour la modélisation au niveau système ont été ajoutées, notamment la possibilité de définir des classes (au sens de la programmation orienté objet) et la notion d'interfaces. Ces dernières permettent de connecter des modules avec un plus haut niveau d'abstraction en regroupant tous les signaux de la connexion dans une même interface. Cette interface peut également contenir des directives de vérifications, des méthodes et des variables. La possibilité d'inclure des vérifications à ce niveau permet par exemple de faire de la vérification de protocole sur des interfaces complexes.

Le langage de vérification est un des ajouts les plus important de SystemVerilog. Baptisé SystemVerilog Assertions (SVA), c'est un langage de définition de propriétés inspiré des OpenVera Assertions (voir partie 2.2.5), du langage Temporal ***e*** (voir 2.2.6) et de PSL. Les SVA sont de deux types : les assertions immédiates et les assertions concurrentes. Les assertions immédiates vérifient des propriétés sur l'état courant du modèle, sans tenir compte de son évolution dans le temps. On peut les comparer aux assertions de VHDL ou à celles des langages de programmation procéduraux. Contrairement à VHDL, Verilog ne permettait pas la définition d'assertions et les assertions immédiates sont venues combler cette lacune. Les assertions concurrentes décrivent le comportement du modèle dans le temps. Elles sont composées d'expressions booléennes, de définitions de séquences et d'expressions temporelles (dans une logique temporelle linéaire). Le modèle d'évaluation des assertions est défini sur une horloge. Une assertion

est ainsi évaluée uniquement sur le top d'horloge. Ceci définit également l'instant d'échantillonnage des variables et des signaux. L'horloge est définie par l'utilisateur et peut varier d'une assertion à l'autre. Plusieurs horloges différentes peuvent également être utilisée dans une même assertion. Les SVA sont en fait très proches de PSL [57, 58]. Les deux groupes travaillent ensemble pour aligner la sémantique des deux langages et unifier certains opérateurs. Contrairement à PSL, SVA ne propose pas d'accès direct aux opérateurs LTL, même si des constructions équivalentes existent [57]. De plus, SVA est conçu pour la vérification par simulation, il ne propose donc pas d'équivalent à *OBE*. Les assertions SystemVerilog peuvent être embarquées directement dans le code du modèle ou bien séparées pour faciliter la réutilisabilité.

Enfin, SytemVerilog améliore les capacités d'interaction de Verilog avec d'autres langages et avec l'environnement de simulation, via des API. Il propose notamment les appels directs de fonctions C ou C++ dans le modèle lui-même. Une API permet d'accéder aux assertions pour les activer ou les désactiver par exemple.

2.2.5 OpenVera

OpenVera est un langage de vérification de matériel [59, 60] à sources ouvertes, géré par Synopsys. Sa syntaxe est proche de celle de C++ ou de Java. C'est un langage de programmation de haut niveau, relativement générique, orienté objet. Les objectifs d'OpenVera sont de faciliter la création de bancs de tests, de fournir des outils d'analyse du taux de couverture et de permettre la vérification du comportement du modèle via des assertions. Les assertions OpenVera (OVA) [23, 24] sont proches des SVA . Elles sont formées d'expressions rationnelles et de formules temporelles linéaires. L'échantillonnage des variables se fait également avec une ou plusieurs horloges. OpenVera est utilisé par de nombreux outils, dont Vera de Synopsys.

2.2.6 Le langage *e*

Le langage ***e*** [36, 61] est un langage de programmation générique, spécialisé dans la vérification fonctionnelle. Il est orienté objet et orienté aspects [62]. La programmation orienté aspect (POA) permet de capturer des fonctionnalités réparties sur plusieurs objets du système en introduisant la notion d'aspect. La POA permet par exemple d'ajouter des membres (champs, méthodes, ...) dans des classe préexistantes ou bien de remplacer une

méthode par une autre. ***e*** est principalement utilisé pour la création de bancs de test et la vérification de comportement. Il possède notamment un mécanisme de génération de données aléatoires avec contraintes, intégré au langage. La programmation orienté aspect peut être utilisé pendant la conception des bancs de test pour ajouter des contraintes sur des données déjà définies. Elle permet également l'exploration rapide de différentes solutions, sans devoir modifier la totalité du système et ainsi faciliter le travail en équipes. Des primitives de programmation concurrente comme les processus légers (« thread »), les événements et les sémaphores permettent un haut niveau d'abstraction. Enfin, des mécanismes permettent de mesurer le taux de couverture des tests. Le langage de définition de propriétés *Temporal* ***e*** est un sous-ensemble du langage ***e***. Il est basé sur la logique temporelle linéaire. Comme OVA, SVA et PSL, Temporal ***e*** permet la modélisation de comportement via des séquences et possède la notion d'événements d'échantillonnage [37]. Contrairement à PSL, il n'a pas de logique temporelle équivalente à CTL. Temporal ***e*** permet la vérification formelle de modèles ***e*** ou la vérification par assertion de modèles HDL.

Le langage ***e*** a été créé par Verisity (maintenant Candence). Il est utilisé dans l'outil Specman Elite [63] de Verisity pour l'automatisation de la génération de bancs de test.

2.3. .NET Framework

De plus en plus, les environnements de modélisation se rapprochent de langages de programmation classiques, en intégrant des concepts permettant la modélisation au niveau système. SystemC et SystemVerilog, qui offrent entre autre la conception orientée objet, la notion d'interfaces et les canaux de communication abstraits, illustrent bien cette tendance. En profitant des fonctionnalités avancées des outils de développement logiciel modernes, de nouveaux environnements de modélisation et de simulation sont créés. C'est le cas d'ESys.NET, qui tire partie des fonctionnalités de la plateforme logicielle Microsoft .NET.

2.3.1 Présentation générale

Depuis une dizaine d'années, la popularité des plateformes logicielles reposant sur des machines virtuelles ne cesse de grandir. Les machines virtuelles offrent plusieurs avantages [64] :

- La portabilité : pour implémenter *n* programmes sur *m* plateformes on a besoin de *n* + *m* traductions au lieu de *n* × *m* traductions avec un compilateur classique ;
- La réduction de la taille : en général, le code est plus compact quand il est traduit dans un format intermédiaire ;
- La sécurité : le déploiement à l'exécution d'une infrastructure de sécurité et la vérification du typage est plus simple sur un code intermédiaire de haut-niveau ;
- L'interopérabilité : elle est facilité par la traduction vers un code intermédiaire et par le partage d'un système de type commun ;
- La flexibilité : la combinaison d'un code intermédiaire de haut niveau et de méta-données permettent la construction de concepts de méta-programmation comme la réflexion et la génération dynamique de code.

La première plateforme avec machine virtuelle largement acceptée par l'industrie est Java, de Sun Microsystems [8]. En 2000, Microsoft annonce la plateforme .NET [7], elle aussi basée sur le concept de machine virtuelle. Le cœur de .NET, représenté par le CLI (*Common Language Infrastructure*) est une plateforme d'exécution normalisée par l'ECMA en décembre 2001 et en avril 2003 par l'ISO.

Ce qui différencie .NET de Java, c'est principalement son caractère multi langage. Le CLI a été conçu pour permettre l'interopérabilité entre différents langages de programmation, de manière transparente pour l'utilisateur. Deux composants permettent cette interopérabilité : le système de type unifié, le *Common Type System* (CTS), et le langage intermédiaire. Le langage commun intermédiaire (CIL, *Common Intermediate Language*) est un jeu d'instructions supportant des notions de haut niveau (classes, exceptions,...). Tous les programmes, quelques soient leurs langages, sont compilés dans ce code intermédiaire et utilisent le CTS.

Le CLI est composé que quatre éléments :

Le *Common Type System*. Le CTS fournit un système de type qui supporte la plupart des types et des opérations présents dans les langages de programmation.

Les Méta-données. Les méta-données sont utilisées par le CLI pour décrire et référencer les types définis par le CTS. Les méta-données sont stockées de manière indépendante du

langage de programmation. Les méta-données offrent ainsi un mécanisme commun pour les outils travaillant avec les programmes (compilateurs, débuggeurs,...). Elles sont également utilisées pour annoter les programmes à l'aide d'attributs.

Le *Common Language Specification*. Le Common Language Specification est un accord entre les concepteurs de langages et les concepteurs de bibliothèques. Il spécifie un sous ensemble du CTS et un ensemble de conventions. Les langages facilitent l'accès aux bibliothèques en implémentant au moins la partie du CTS définie par le CLS. Les bibliothèques seront plus largement utilisées si elles exposent uniquement des aspects (classes, interfaces, méthodes, ...) qui font partie du CLS.

Le *Virtual Execution System*. Le *Virtual Execution System* (VES) est responsable du chargement et de l'exécution de programmes en CIL. Il fournit les infrastructures nécessaires à l'exécution de code géré : gestion automatique de la mémoire (ramasse-miettes), gestion des processus légers, gestion des méta-données, ...

Le CLI défini également un ensemble de bibliothèques de classes fournissant des services importants comme les interactions avec les processus légers, la manipulation de données et des structures de données fondamentales (listes, tableaux, tables de hachage, ...).

2.3.2 Le langage C#

Le langage C# est un langage de programmation moderne, orienté-objet et normalisé par l'ECMA et l'ISO. Sa syntaxe s'inspire de celle de Java et C++ et il propose des concepts comme le typage fort, la vérification des bornes des tableaux, la détection des utilisations de variables non initialisées et la libération automatique de la mémoire. La plupart des implémentations utilisent le CLI comme support d'exécution et comme bibliothèque de classes. Il hérite ainsi des capacités du CLI et la frontière entre le langage C# et le CLI est parfois difficile à définir.

2.3.3 Introspection et réflexion

L'introspection est la capacité d'un programme à voir, comprendre et modifier sa propre structure, pendant son exécution [65]. Un langage de programmation est dit réflexif si sa structure lui est accessible. L'information accessible via l'introspection est appelée méta-information ou méta-données. Ces méta-données permettent la création simple

d'outils travaillant sur les programmes comme des débuggeurs, des explorateurs d'objets ou des interpréteurs. Il existe beaucoup de langages de programmation réflexifs dont Java et C#. Ces langages fournissent aux programmes les méta-informations nécessaires à l'introspection via une API de réflexion. Ils implémentent l'introspection grâce à un environnement d'exécution particulier, les machines virtuelles.

Le langage C# permet cette introspection en obtenant auprès de la CLI la structure d'un objet. Pour décrire un objet, le CLI retourne un objet qui est une instance d'une méta-classe appelée `Type` qui documente complètement la structure de l'objet. Le Tableau I présente quelques une des classes qui rendent les méta-données accessibles aux programmes.

Membres	Description
`Type`	Contient les informations sur la déclaration d'un type (classe, interface, tableaux, énumération …).
`Assembly`	Décrit une « assembly », qui est une collection de types et de ressources formant une unité logique de fonctionnalités.
`MethodInfo`	Contient les méta-données sur une méthode, comme son nom, son type de retour ou ses paramètres.
`ParameterInfo`	Contient les méta-données sur un paramètre d'un méthode ou d'un constructeur (nom, type…).
`FieldInfo`	Contient les méta-données sur un champ (variable d'instance). Par exemple son nom, son type ou sa visibilité.
`PropertyInfo`	Contient les méta-données sur une propriété.
`EventInfo`	Contient les méta-données sur un événement.
`ConstructorInfo`	Contient les méta-données sur un constructeur, comme la liste de ses paramètres.
`MemberInfo`	La classe base pour `MethodInfo`, `ConstructorInfo`, `FieldInfo`, `EventInfo` et `PropertyInfo`.

Tableau I : Les classes d'accès aux méta-données.

L'exemple de code ci-dessous illustre quelques-unes des opérations que l'on peut faire à l'aide de l'introspection.

```
1.  public class TestIntrosp{
2.    private string _chaine = "Une chaîne de caractères";
3.    private void MaMethode(){
4.      //du code
5.    }
6.    public string Chaine{
7.      get{return _chaine;}
8.    }
```

```
9.    public static void Main(){
10.     TestIntrosp objet = new TestIntrosp();
11.     Type monType = objet.GetType();
12.      foreach(MemberInfo member in monType.GetMembers())
13.       System.Console.WriteLine("{0} est un {1}", member.Name,
14.                                member.MemberType);}}
```

Code Source 1 : Parcours des membres d'un type avec l'introspection.

Voici un extrait de la sortie du programme :

```
_chaine est un(e) Field
MaMethode est un(e) Method
.ctor est un(e) Constructor
Chaine est un(e) Property
```

La ligne 10 instancie un objet du type `TestIntrosp`. On récupère un objet `Type` décrivant la classe `TestIntrosp` à la ligne 11. Les lignes 12 et 13 itèrent à travers tous les membres de la classe et les affichent sur la sortie standard.

Les méta-données sur un membre d'un type permettent d'interagir avec le membre en question. Par exemple de modifier la valeur d'une variable d'instance. L'exemple ci-dessous illustre ceci en lisant la valeur d'une variable privée et en la modifiant.

```
1.  public class MaClasse{
2.    private string chaine = "Je suis caché!";
3.    public string EnMajuscule(string param){
4.      return param.ToUpper();
5.    }}
6.  public class Test{
7.    public static void Main(){
8.      Type monType = typeof(MaClasse);
9.      FieldInfo field = monType.GetField("chaine",
10.                  BindingFlags.Instance|BindingFlags.NonPublic);
11.     MaClasse monObjet = new MaClasse();
12.     System.Console.WriteLine("monObjet.chaine = {0}",
13.                                        field.GetValue(monObjet));
14.     field.SetValue(monObjet, "Et non !");
15.     System.Console.WriteLine("monObjet.chaine = {0}",
16.                                        field.GetValue(monObjet));
17.   }}
```

Code Source 2 : Accès à un champ par introspection.

Les lignes 9-10 montrent comment récupérer un objet `FieldInfo` correspondant à la déclaration d'un champ privé d'une classe, à partir de son nom. À la ligne 13 on lit la valeur de ce champ pour une instance donnée. Cette valeur peut être modifiée (ligne 14). Une nouvelle lecture ligne 16 permet de vérifier que la valeur a effectivement changée. Ce

mécanisme de lecture d'une valeur est au cœur de l'outil de vérification présenté ici. C'est grâce à lui que l'état du modèle peut être observé pendant la simulation.

L'introspection permet également d'accéder à la déclaration d'une méthode à partir de son nom et de l'appeler. Dans le Code Source 3, la méthode `EnMajuscule` de la classe `MaClasse` (cf. Code Source 2) est appelée.

```
1.  public static void Main()
2.  {
3.    Type monType = typeof(MaClasse);
4.    object unObjet = Activator.CreateInstance(monType);
5.    MethodInfo method = monType.GetMethod("EnMajuscule",
6.                                    new Type[]{typeof(string)});
7.    object[] parametres = new object[]{"en majuscule ?"};
8.    object resultat = method.Invoke(unObjet, parametres);
9.    Console.WriteLine("{0}", resultat);}
```

Code Source 3 : Découverte et appel de méthode avec l'introspection.

Ligne 4, une instance de la classe `MaClasse` est créée, à partir de l'objet `Type` correspondant. Les lignes 5-6 montrent comment la découverte dynamique de la méthode est effectuée à partir de son nom et du type de ses paramètres. Avant l'appel dynamique la méthode, on crée un tableau qui contient les paramètres (ligne 7). L'appel lui-même se fait à la ligne 8.

L'introspection permet également de charger dynamiquement des « assemblies » et d'en explorer le contenu. De manière simple une « assembly » est le résultat d'une compilation d'un programme .NET. C'est un fichier contenant un ensemble de définitions de types au format CIL.

```
1.  public static void Main(){
2.    Assembly assembly = Assembly.LoadFrom(@"c:\MonAssembly.dll");
3.    Type[] typesAssembly = assembly.GetTypes();
4.    foreach(Type monType in typeAssembly)
5.      Console.WriteLine("{0}", monType.FullName);}
```

Code Source 4 : Chargement dynamique d'assembly

L'« assembly » est stockée sur le disque dur sous la forme d'un fichier. Elle est chargée dans le domaine d'exécution de l'application à la ligne 2. L'ensemble des types définis dans cette assembly sont accessibles (ligne 3). Les lignes 4-5 itèrent à travers ces types et les affichent sur la sortie standard.

Un programme peut également modifier sa propre structure via l'introspection. Dans .NET, un programme peut émettre du code CIL pendant son exécution [66]. Ce mécanisme peut en particulier servir à écrire relativement simplement des compilateurs pour .NET. L'exemple ci-dessous montre la création d'une classe et la définition d'une méthode, pendant l'exécution.

```
1.  public static void Main(){
2.    //Le code pour initialiser moduleBuilder n'est pas montré
3.    TypeBuilder typeBuilder = moduleBuilder.DefineType("MaClasse",
4.                      TypeAttributes.Class | TypeAttributes.Public,
5.                      typeof(object));
6.     MethodBuilder mBuilder = typeBuilder.DefineMethod("Value",
7.                 MethodAttributes.Public| MethodAttributes.Virtual,
8.                 typeof(string), new Type[]{});
9.    ILGenerator ilGen = mBuilder.GetILGenerator();
10.    ilGen.Emit(OpCodes.Ldstr, "toto");
11.    ilGen.Emit(OpCodes.Ret);
12.    Type monType = typeBuilder.CreateType();
13.
14.    MethodInfo maMethode = monType.GetMethod("Value");
15.    object monObjet = Activator.CreateInstance(monType);
16.    Console.WriteLine(maMethode.Invoke(monObjet, new object[0]));}
```

Code Source 5 : Création d'un type pendant l'exécution.

Une nouvelle classe publique héritant d'`object` est déclarée lignes 3 à 5. Une méthode nommée `Value` est ajoutée lignes 6 à 8. Cette méthode retourne une chaîne de caractères et ne prend aucun argument. Les lignes 9 à 11 définissent le corps de cette méthode. Les instructions sont en langage intermédiaire. Sans entrer dans le détail du langage intermédiaire, la méthode charge une chaîne de caractères sur la pile (ligne 10) et retourne l'élément situé en haut de la pile (ligne 11). Ligne 12, le type est créé. Les opérations d'introspection présentées dans les autres exemples sont accessibles sur ce nouveau type (lignes 14 à 16).

2.3.4 Programmation par attributs

C# et le CLI permettent d'ajouter des informations déclaratives sur des entités d'exécution. Ce mécanisme est appelé programmation par attributs. Les attributs peuvent être ajoutés sur tous les éléments du langage, excepté le corps des méthodes, des propriétés et des constructeurs.

L'exemple du Code Source 6 montre comment annoter des membres d'une classe et comment accéder à cette information avec l'introspection.

```
1.  public class MaClasse{
2.    [Afficher]
3.    public string chaine = "une chaîne";
4.
5.    public int entier = 14;
6.
7.    [Afficher]
8.    public double reel = 3.14}
9.
10. public class Test{
11.   public static void Main(){
12.     Type monType = typeof(MaClasse);
13.     foreach(MemberInfo membre in monType.GetMembers()){
14.       object[] att = membre.GetCustomAttributes(
15.                         typeof(AfficherAttribute), false);
16.       if(att.Length != 0)
17.       Console.WriteLine("{0}", membre.Name);}}}
```

Code Source 6 : un exemple de programmation par attributs.

Certains membres de la classe `MaClasse` sont annotés avec l'attribut `Afficher` (lignes 2-3 et lignes 7-8). Lors de l'exploration du type, si un membre est annoté avec un attribut de type `AfficherAttribute` (lignes 14 à 16), il est affiché ligne 17.

2.3.5 Délégués

Les fonctions de rappels (*callback*) sont un concept important pour la gestion des événements. Une fonction de rappel est une fonction liée à un événement. Quand cet événement survient, ce n'est pas le programme qui appel la méthode liée, mais c'est l'environnement qui va appeler la fonction de rappel. Ces fonctions de rappels sont implémentées par les pointeurs sur fonctions en C++ et par les délégué (*delegate*) en C#. Par rapport aux pointeurs sur fonctions, les délégués offrent un typage fort. De plus, les délégués permettent nativement de connecter plusieurs méthodes à un seul délégué. Les méthodes sont appelées les unes à la suite des autres. L'exemple du Code Source 7 montre comment déclarer, instancier et appeler un délégué.

```
1.  delegate void MonDelegue();
2.  class Test{
3.    static void F() {System.Console.WriteLine("Test.F");}
4.    static void Main() {
5.      MyDelegate d = new MonDelegue (F);
6.      d();}}
```

Code Source 7 : Un exemple simple de délégué.

Le prototype du délégué est déclaré ligne 1. La méthode qui sera liée au délégué est définie ligne 3. Ligne 5 un délégué est instancié et on le lie à la méthode `F`. Enfin, le

délégué est appelé ligne 6. Pour lier une autre méthode à `d`, on peut ajouter la ligne `d+=new MonDelegue(G);` entre les ligne 5 et 6. Les deux méthodes seront alors appelées en séquence.

C# ajoute la notion d'événement aux délégués, via le mot-clef `event`. Ce mot-clef s'applique aux délégués déclarés comme membres d'une classe. Depuis l'intérieur de la portée de la classe, le délégué n'a pas changé. Mais depuis l'extérieur de la classe, le délégué ne peut être invoqué. Il peut seulement être utilisé comme opérande de gauche avec les opérateurs += et -=. L'opérateur += lie une nouvelle méthode au délégué, -= la retire.

```
1.  public delegate void Delegue();
2.
3.    public class Producteur{
4.      public event Delegue donneeDispo;}
5.
6.      public class Consommateur{
7.        Producteur prod = new Producteur();
8.        public Consommateur() {
9.          prod.donneeDispo += new Delegue(Afficher);}
10.       void Afficher(){
11.         Console.WriteLine("La donnée est prête!");}}
```

Code Source 8 : Synchronisation avec des événements.

Dans l'exemple du Code Source 8, un événement est utilisé pour synchroniser un producteur et un consommateur. La ligne importante est la ligne 9, où le consommateur accroche la méthode `Afficher` à l'événement `donneeDispo` du producteur. Le code de la notification par le producteur n'est pas montré.

2.4. ESys.NET

ESys.NET [6, 67] est un environnement de modélisation et de simulation au niveau système. A l'instar de SystemC, il est construit sur un langage de programmation et lui ajoute les constructions nécessaires à la modélisation de systèmes matériel/logiciel. L'objectif d'ESys.NET est de fournir un environnement extensible, performant et utilisable simplement. L'extensibilité de l'environnement permet l'ajout de composant de vérification, d'analyse du modèle ou encore de visualisation. C'est une caractéristique d'ESys.NET qui est au cœur de notre projet. L'utilisation de fonctionnalités avancées de la plateforme .NET et de C#, telles que l'introspection, la programmation par attribut et le support de plusieurs langages, facilites grandement l'interaction avec des outils externes.

ESys.NET a été influencé par l'architecture de SystemC, la différence majeure provient des capacités d'introspection limitées de C++ [5], ce qui implique une utilisation importante de macro et limite l'extensibilité du système. La complexité du langage C++, par exemple pour la gestion de la mémoire, rend la syntaxe d'un modèle SystemC relativement complexe.

ESys.NET est une bibliothèque de classe qui définie les éléments nécessaires à la modélisation. Elle propose également un simulateur basé sur les événements. Un modèle ESys.NET est donc écrit dans un des langages supportés par .NET. Ce programme est compilé dans du code CIL et stocké dans une « assembly » (un fichier exécutable ou une bibliothèque chargée dynamiquement). Le simulateur explore ensuite le modèle avec l'introspection pour en extraire la structure et enregistrer les entités de simulation (processus, signaux, etc.).

2.4.1 Un premier exemple

Afin de donner un aperçu global d'ESys.NET, un premier exemple de modèle est donné dans cette partie. Il s'agit d'un additionneur à trois entrées dont les ports d'entrées sont stimulés par un banc de test. La Figure 1 donne une représentation graphique du modèle.

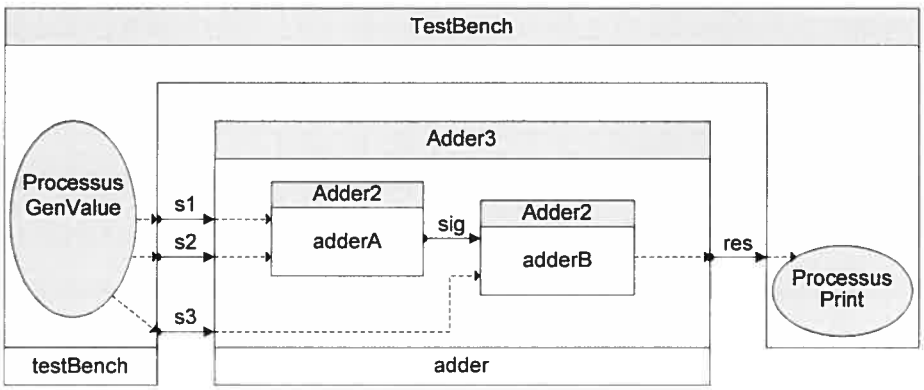


Figure 1 : Un modèle ESys.NET simple.

Le processus `GenValue` génère des données aléatoires qui sont utilisées comme entrées sur le module testé `adder`. Ce module est un système combinatoire (i.e. il n'a pas d'horloge) qui calcule la somme de ses trois entrées et écrit le résultat sur sa sortie. Pour mettre en évidence le caractère hiérarchique de la modélisation avec ESys.NET, ce module

est découpé en deux sous modules qui sont des additionneurs à deux entrées. Ces deux additionneurs sont deux instances d'un module dont le code est donné dans le Code Source 9.

```
1.  public class Adder2 : BaseModule{
2.    public inInt inA;
3.    public inInt inB;
4.    public outInt outRes;
5.
6.    [MethodProcess]
7.    [EventList("sensitive","inA")]
8.    [EventList("sensitive","inB")]
9.    private void Calc()
10.   {
11.     outRes.Value = inA.Value + inB.Value;}}
```

Code Source 9 : Code d'un additionneur à deux entrées.

Le module Adder2 possède deux ports d'entrées `inA` et `inB` et un port de sortie `outRes`. Ce module doit additionner les valeurs présentent sur les ports `inA` et `inB` et écrire le résultat sur le port `outRes`. Dans un module, le traitement des données est effectué par un ou plusieurs processus. Un processus est une méthode étiquetée avec un attribut particulier défini par ESys.NET. Dans le Code Source 9, la méthode `Calc` est étiquetée avec l'attribut `MethodProcess`, pour indiquer que cette méthode doit être exécuté à chaque occurrence d'un ou plusieurs événements. Les événements qui déclenchent l'exécution d'un processus constituent la liste de sensibilité du processus. Dans ESys.NET, cette liste est définie grâce aux attributs `EventList`. Ici, le processus est sensible à deux événements, l'événement nommé `sensitive` appartenant au port `inA` et l'événement `sensitive` du port `inB`. Ces événements sont déclenchés à chaque changement de valeur sur un port. Ainsi, dès qu'une des valeurs change, le processus `Calc` est exécuté et le résultat est calculé.

```
1.  public class Adder3 : BaseModule{
2.    public inInt inA, inB, inC;
3.    public outInt outRes;
4.
5.    private Adder2 adderA = new Adder2();
6.    private Adder2 adderB = new Adder2();
7.    private IntSignal sig = new IntSignal();
8.
9.    public override void BindingPhase()
10.   {
11.     adderA.inA = inA;
12.     adderA.inB = inB;
13.     adderA.outRes = sig;
14.
```

```
15.     adderB.inA = sig;
16.     adderB.inB = inC;
17.     adderB.outRes = outRes;}}
```

Code Source 10 : Code d'un additionneur à trois entrées.

Le Code Source 10 présente la définition du module Adder3. Ce module comporte trois ports d'entrées et un port de sortie. Il est composé de deux sous modules `adderA` et `adderB` connectés entre eux par un signal nommé `sig`. Ces deux sous module sont des instances du module `Adder2` présenté dans le Code Source 9. Un signal est une entité qui transporte une donnée entre deux modules. C'est en quelque sorte la modélisation d'un fil dans un circuit électronique. La connexion entre les ports d'entrées/sorties du module Adder3 et les ports d'entrées/sorties des ses deux sous modules ne nécessite pas de signal. La référence vers le port de niveau supérieur peut simplement être recopiée au niveau inférieur. Il faut cependant effectuer cette opération une fois le modèle complètement instancié, lors de la phase « d'accrochage » (`BindingPhase`).

```
1.  public class TestBench:BaseModule{
2.    public outInt outX, outY, outZ;
3.    public inInt inResult;
4.
5.    [ThreadProcess]
6.    private void GenVal()
7.    {
8.      Random random = new Random();
9.      while(true)
10.     {
11.       Wait((ulong)random.Next(30, 50))
12.       outX.Value = random.Next(10);
13.       Wait((ulong)random.Next(30, 50));
14.       outY.Value = random.Next(10);
15.       Wait((ulong)random.Next(30, 50));
16.       outZ.Value = random.Next(10);}}
17.
18.   [MethodProcess]
19.   [EventList("sensitive","inResult")]
20.   private void Print(){
21.     Console.WriteLine("r = {0}", inResult.Value);}
```

Code Source 11 : Définition du banc de test.

Dans le Code Source 11, le module correspondant au banc de test est défini. Outre les ports d'entrées/sorties, il comporte un processus `GenVal` et un processus `Print`. Le processus `GenVal` est de type `ThreadProcess`, c'est-à-dire qu'il va être exécuté dans un thread parallèle au processus d'exécution du simulateur. Un `ThreadProcess` ne possède en

générale pas de liste de sensibilité et contient souvent une boucle infinie qui va tourner pendant toute la simulation. Le `ThreadProcess` peut cependant s'endormir en attente d'un événement ou pour une durée donnée. Ainsi, dans l'exemple du Code Source 11, le processus ajoute un délai aléatoire avec l'instruction `Wait` qui va endormir le processus.

```
1.  class TopLevel : SystemModel{
2.    private TestBench testBench = new TestBench();
3.    private Adder3 adder = new Adder3();
4.
5.    private IntSignal s1 = new IntSignal();
6.    private IntSignal s2 = new IntSignal();
7.    private IntSignal s3 = new IntSignal();
8.    private IntSignal sRes = new IntSignal();
9.
10.   public TopLevel(ISystemManager manager,
11.                   string name):base(manager, name){
12.     adder.inA = s1;
13.     adder.inB = s2;
14.     adder.inC = s3;
15.     adder.outRes = sRes;
16.
17.     testBench.outX = s1;
18.     testBench.outY = s2;
19.     testBench.outZ = s3;
20.     testBench.inResult = sRes;}
21.
22.   static void Main(string[] args){
23.     Simulator simulator = new Simulator();
24.     TopLevel topLevel = new TopLevel(simulator, "TestAdder");
25.     simulator.Run(3000);}}
```

Code Source 12 : Le modèle complet.

Le modèle complet est défini dans le Code Source 12 en déclarant une classe qui hérite de `SystemModel`. L'additionneur et le banc de test sont instanciés et connectés avec des signaux. Enfin, aux lignes 23 et 24, le simulateur et le modèle lui-même sont instanciés. La simulation est lancée ligne 25, pour une durée de 3000 unités de temps.

La partie suivante détaille chacun des concepts introduits ici.

2.4.2 Modélisation

Modules. Le module est l'entité de base pour le découpage hiérarchique du modèle. Ce découpage est nécessaire pour permettre la conception de systèmes complexes en les divisant en sous composants plus simples. Le module permet d'encapsuler les fonctionnalités et de présenter une interface aux composants extérieurs. Cette interface est composée entre autre des ports d'entrées/sorties et des événements. L'utilisation d'un

module se fait en deux parties : la déclaration et l'instanciation. Dans ESys.NET, un module est déclaré en étendant la classe `BaseModule` et en lui ajoutant les composants internes nécessaires. L'interface est composée d'un ensemble d'éléments publics. Les composants internes peuvent être des sous modules ou des processus qui doivent être privés pour assurer une encapsulation propre. Puisque les modules sont des objets, leur instanciation se fait de manière naturelle avec le mot clef `new`.

Processus. Les processus sont chargés d'effectuer les traitements des données en parallèle. Un processus est une méthode privée sans paramètres annotée avec un attribut défini par ESys.NET. Puisque c'est une méthode appartenant à un module, il peut accéder au contenu du module. En particulier, il peut lire ou écrire sur les ports, modifier les variables d'instance ou appeler des sous méthodes. Il ne doit cependant pas appeler directement d'autres processus, car la synchronisation entre les processus est assurée par le simulateur et nécessite l'utilisation d'événements. ESys.NET permet de déclarer deux types de processus différents : les processus « méthode » et les processus « thread ». La différence fondamentale vient de la méthode d'exécution des processus. Un processus « méthode » sera exécuté via un simple appel de méthode par le simulateur tandis qu'un processus « thread » sera lancé dans un processus léger séparé du simulateur. Cela implique notamment qu'un processus méthode ne peut pas effectuer d'opération bloquante, comme l'attente d'un événement par exemple.

Processus Méthode. Un processus méthode est déclaré avec l'attribut `MethodProcess`. Quand il est déclanché, le corps complet de la méthode est exécuté. Il est déclanché quand un événement appartenant à sa liste de sensibilité survient. Le processus possède en fait deux types de sensibilité : la sensibilité statique et la sensibilité dynamique. La sensibilité statique est définie par un attribut `EventList`. L'attribut est paramétré avec le nom de l'événement et le(s) nom(s) des composants contenant cet événement. Par exemple,

```
[EventList("posedge","clk1","clk2")]
```

déclare une liste de sensibilité contenant les événements `posedge` des signaux d'horloge `clk1` et `clk2`. La sensibilité dynamique est définie pendant l'exécution du modèle par des appels à la méthode `Wait`. Cette méthode peut être appelée avec un événement comme

paramètre ou bien avec une durée. Dans le cas où elle est appelée avec une durée *n*, le processus va être déclanché *n* unités de temps plus tard.

Processus Thread. Un processus thread est déclaré avec l'attribut `ThreadProcess`. Il est exécuté dans un thread parallèle à celui du simulateur. La méthode du processus est exécutée jusqu'à ce qu'une méthode `Wait` soit appelée et que le thread soit ainsi endormi. Il sera réveillé dès qu'un événement de sa liste de sensibilité survient.

Événements. Un événement est implémenté dans ESys.NET par la classe `Event`. C'est le mécanisme de base pour la synchronisation de processus et pour la simulation. Certains événements sont prédéfinis par l'environnement, comme par exemple le front montant d'un signal d'horloge ou bien le changement de la valeur d'un signal. Mais le concepteur peut aussi utiliser les événements pour modéliser des comportements de haut niveau.

Signaux. Les signaux servent à interconnecter les modules. Ils sont chargés de transporter les données d'un port à un autre et de donner l'illusion que ce transfert se fait en parallèle. Pour simuler ce parallélisme, on introduit la notion de delta-cycle. Quand une donnée est écrite sur un signal, le signal va stocker cette valeur et la rendra disponible qu'au prochain delta-cycle. Ce concept est implémenté dans la classe `BaseSignal`. Cette classe est ensuite spécialisé pour les types de données courants (entier, réels, booléens, chaîne de caractères,...). Le Tableau II donne un aperçu des principaux champs d'un signal.

Champs	Type	Description
`Value`	Suivant le type de donné transporté	Contient la valeur actuelle du signal.
`transaction`	`Event`	Déclanché quand un processus écrit une valeur sur le signal.
`sensitive`	`Event`	Déclanché quand un processus écrit une valeur différente de la valeur actuelle du signal.

Tableau II : Principaux champs d'un signal.

Un concepteur peut également implémenter un signal transportant un type de donnée particulier. Contrairement au C++, la notion de *template* n'existe pas. Un *template* est une classe paramétrée. Dans SystemC, les signaux sont implémentés sous la forme d'un *template* paramétré par le type de donnée. Pour déclarer un signal transportant des objets de type `MaClasse`, il suffit d'écrire sc_signal<MaClasse>. Par contre, dans ESys.NET, il faut

étendre la classe `BaseSignal` et en particulier définir, avec le type voulu, la propriété permettant de lire ou écrire la valeur du signal.

Un signal définit deux événements : `transaction` et `sensitive`. Le premier est déclenché quand un processus écrit une valeur sur le signal, quelque soit cette valeur. Le second est déclanché uniquement si la valeur est différente de la valeur actuelle du signal.

Ports. Dans ESys.NET, et contrairement à SystemC, la notion de port ne passe pas par une classe particulière. Elle est implémentée par un mécanisme d'interface au sens orienté objet du terme. Ces interfaces vont servir à masquer les opérations de lecture ou écriture suivant le sens du port. Les signaux implémentent toutes les interfaces de définitions des ports (entrée, sortie, bidirectionnelle). Pour connecter un signal à un port, une simple affectation est nécessaire. De l'intérieur du module, le signal est vu à travers l'interface et les opérations de lecture/écriture sont limitées par cette dernière.

Horloges. Une horloge est modélisée par la classe `Clock`. A l'instanciation, la période de l'horloge est définie et l'horloge déclenche automatiquement les événements correspondants (voir Tableau III).

Champs	Type	Description
`Value`	`bool`	Contient la valeur actuelle de l'horloge (lecture seule).
`posedge`	`Event`	Déclenché à chaque front montant.
`negedge`	`Event`	Déclenché à chaque front descendant.
`sensitive`	`Event`	Déclenché à chaque front montant ou descendant

Tableau III : Principaux champs d'un signal d'horloge.

Canaux. La classe `BaseChannel` permet de modéliser des canaux de communication complexes entre les modules (par ex. une FIFO). On peut voir les canaux comme un croisement entre un signal et un module. Comme un module, un canal peut contenir des sous-modules. Et comme un signal, il gère les delta-cycles.

2.4.3 Simulation

La simulation d'un modèle ESys.NET se découpe en trois phases : l'élaboration, l'initialisation et l'ordonnancement de processus.

Élaboration. C'est dans cette phase que les éléments d'un modèle sont instanciés et connectés entre eux et que le simulateur explore le modèle pour enregistrer les éléments dont il a besoin (les horloges, les modules, les processus, les signaux, ...). Cette phase est faite entièrement à l'exécution, grâce à l'introspection.

Initialisation. Au cours de cette phase, les processus sans liste de sensibilité statique sont exécutés. Les méthodes marquées avec l'attribut `SimInit` sont également exécutées, pour initialiser des signaux par exemple.

Ordonnancement des processus. L'ordonnanceur d'ESys.NET est chargé de gérer le temps, l'ordre d'exécution des processus, la notification des événements, la mise à jour des signaux et des canaux ainsi que l'exécution des fonction des rappels. Les différentes étapes d'une simulation sont données ci-dessous. L'ordonnancement proprement dit se fait entre les étapes 3) et 11).

1) *Phase d'élaboration ;*
2) *Phase d'initialisation ;*
3) *Phase d'évaluation.* Sélectionner un processus parmi l'ensemble des processus prêt à être exécutés. L'ordre de sélection n'est pas défini ;
4) Reprendre l'exécution du processus sélectionné ;
5) Exécuter le processus. L'exécution d'un processus peut déclancher une notification immédiate d'un événement et ainsi rendre d'autres processus prêt à être exécutés dans la même phase d'évaluation ;
6) Répéter l'étape 3) pour tous les autres processus prêts à être exécutés ;
7) *Phase de mise à jour.* Mettre à jour les éléments dépendant d'un delta-cycle (signaux et canaux) ;
8) Si il y a des notifications de delta-cycle en attente, déterminer les processus prêts à être exécuté et aller à l'étape 3) ;
9) S'il n'y a plus de notification d'événement temporisé, aller à l'étape 12);
10) Avancer le temps de simulation courant jusqu'à la prochaine notification d'événement temporisé ;
11) Déterminer quels processus sont prêts à être exécutés à cause des événements en attente à l'instant courant et aller à l'étape 3) ;

12) Fin de la simulation.

Les événements temporisés sont générés par une méthode quand elles appellent la méthode `Wait`. Les horloges génèrent également des événements temporisés pour faire leurs changements de valeur tous les demi-cycles.

La Figure 2 donne une vue schématique de l'algorithme. Les numéros dans les différents blocs correspondent aux numéros des étapes dans l'algorithme donné à la page précédente.

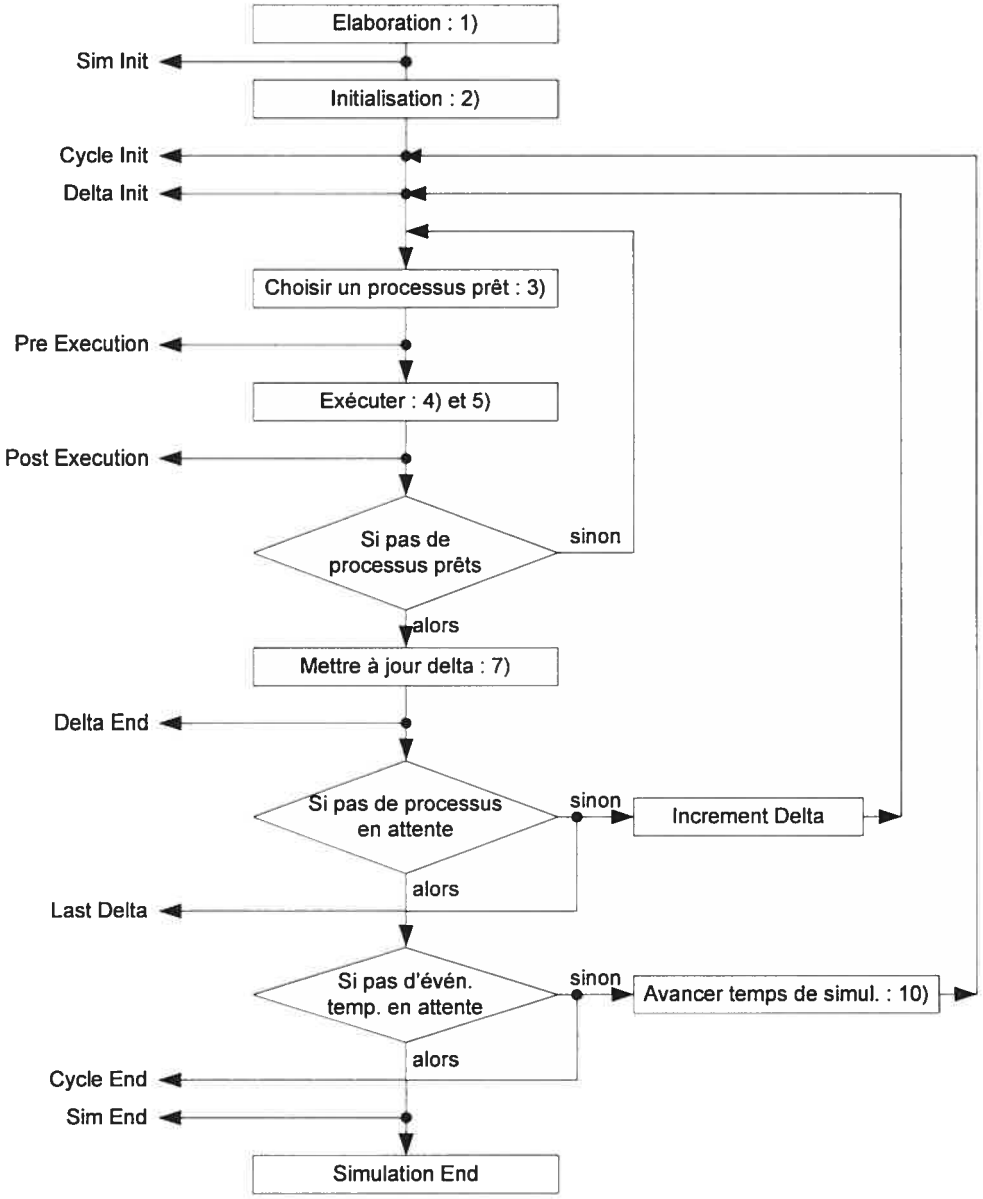


Figure 2 : L'ordonnanceur d'ESys.NET et les points d'encrage.

Points d'encrage. Les points d'encrage sont des événements déclanchés par le simulateur d'ESys.Net afin de faciliter les interactions entre le simulateur et l'environnement. ESys.NET propose des points d'encrages à différentes étapes de la simulation. Dans la Figure 2 les différents points d'encrage sont indiqués [67]. Un point d'encrage important est `CycleEnd`. Il est appelé juste avant le saut de temps discret. C'est à ce moment qu'on peut observer un modèle ou tous les signaux sont stables, quelque soit l'ordonnancement exact des processus dans les delta-cycles.

2.5. Discussion

On peut distinguer deux grandes méthodes de vérifications, la vérification formelle, qui vise à produire une preuve exhaustive de la correction du modèle, et la vérification par simulation, qui s'assure de la validité d'une exécution du modèle par rapport à un ensemble de règles. Parmi les méthodes formelles, le *model-checking* est la plus employée. Mais malgré les progrès récents dans le domaine, elle reste difficilement sur des models très complexes comme ceux qu'on peut trouver dans la conception de matériel. Il est clair que est difficilement faisable sur un modèle ESys.NET. Calculer le graphe de tous les états possible est impossible dans la plupart des cas. La vérification par simulation est applicable plus simplement et permet de détecter des erreurs dans la fonctionnalité du modèle. En ce qui concerne les langages de définition de propriétés, il semble clair que la tendance actuelle est d'utiliser une logique temporelle linéaire conjointement à des expressions rationnelles étendues. Pour assurer une compatibilité maximum avec les outils existants, un nouvel environnement a tout intérêt à se baser sur la logique temporelle linéaire.

Pour vérifier un modèle pendant la simulation, il faut pouvoir en observer l'état dynamiquement. De plus, pour qu'un environnement de vérification soit générique, il faut qu'il soit non intrusif, c'est-à-dire qu'il soit le plus indépendant du modèle possible. Il doit être capable de vérifier un modèle quelconque, et doit notamment pouvoir manipuler des types arbitraires. De telles opérations sont possibles grâce à l'introspection. Enfin, la collaboration avec un simulateur existant implique l'existence de points d'encrages qui permettent à l'outil de vérification de se synchroniser.

.NET et l'environnement ESys.NET offrent donc des avantages certains pour la conception d'un environnement de vérification. Les capacités d'introspection de .NET

facilitent l'exploration et l'observation du modèle pendant la simulation. Les points d'encrage proposés par le simulateur d'ESys.Net permettent d'ajouter aisément des composants externes au simulateur, sans avoir besoin de le recompiler.

Chapitre 3 Bases Formelles

Dans ce chapitre, nous allons présenter les différentes bases formelles et les algorithmes utilisés pour la définition et la vérification de propriétés temporelles.

3.1. Présentation informelle

Une propriété exprimée par une formule de logique temporelle linéaire (LTL) [13, 17] est composée de propriétés élémentaires (ou propositions atomiques), d'opérateurs booléens et d'opérateurs temporels. Les propositions atomiques décrivent l'état du système à un instant donné. Par exemple, dans un modèle de four à micro-onde, on pourrait définir les propriétés :

- p_0 : « la porte du four est fermée » ;
- p_1 : « un programme de cuisson a été sélectionné » ;
- p_2 : « le four est en train de chauffer ».

La valeur de ces propositions varie suivant l'état dans lequel se trouve le modèle. Par exemple, à l'initialisation, la porte du four est ouverte, aucun programme n'est sélectionné et le four ne chauffe pas. Les trois propositions sont donc fausses. Une fois la porte fermée, la proposition atomique p_0 devient vraie. Les propositions atomiques vont servir de base pour la construction des propriétés LTL devant être vérifiées. Ainsi, la propriété « dans le futur p_2 sera vraie » spécifie que le four va effectivement fonctionner. Cette propriété est formalisée en LTL par $\Diamond\, p_2$, où l'opérateur $\Diamond$ (fatalement) indique que la propriété qui suit doit être vérifiée dans le futur. De même, la propriété « la porte du four doit toujours être fermée pour que le four chauffe » s'écrit $\Box\, (p_2 \rightarrow p_0)$ où $\Box$ désigne l'opérateur « globalement » et $\rightarrow$ l'implication logique.

3.2. Formalisation de LTL

3.2.1 Structure de Kripke.

La sémantique de LTL est définie sur les structures de Kripke.

Structure de Kripke. Étant donné un ensemble de propriétés atomique AP, une structure de Kripke est un quadruplet $(S, I, R, Etiquette)$ où :

- S est un ensemble fini d'états.
- $I \in S$ un ensemble d'état initiaux.
- $R \subseteq S \times S$ est une relation de transition définie par :

$$\forall s \in S.\ (\exists s' \subseteq S.\ (s, s') \in R)$$

- $Etiquette : S \rightarrow 2^{AP}$ est une fonction d'interprétation qui associe à chaque état s $\in$ S l'ensemble des propositions atomiques valides dans l'état s.

Chemins et Exécution. Un chemin dans une structure de Kripke est une séquence finie ou infinie d'états $s_0, s_1, s_2 \ldots$ tel que $(s_i, s_{i+1}) \in R$ pour tout $i \geq 0$. Pour un chemin σ et un entier $i \geq 0$, on note $\sigma[i]$ le (i-1)^{ème} état de σ, c'est-à-dire s_i. On note également σ' le suffixe de σ obtenu à partir du i^{ème} état, c'est-à-dire $s_i, s_{i+1}, s_{i+2} \ldots$ En particulier $\sigma^0 = \sigma$. Une exécution est un chemin qui commence par l'état initial s_0. On note $|\sigma|$ la longueur, possiblement infinie du chemin, c'est-à-dire le nombre d'états dans σ.

3.2.2 Syntaxe de LTL

La Figure 3 définit la syntaxe de LTL sur un ensemble AP de propositions atomiques.

p est une formule LTL si $p \in AP$;
Si ϕ est une formule LTL alors $\neg\phi$ (négation logique) est une formule LTL ;
Si ϕ et ψ sont des formules LTL alors $\phi \vee \psi$ (ou logique) est une formule LTL ;
Si ϕ est une formule LTL alors $\mathsf{X}\phi$ est une formule LTL ;
Si ϕ et ψ sont des formules LTL alors $\phi \mathsf{U} \psi$ est une formule LTL.

Figure 3 : Syntaxe de LTL.

L'opérateur $\mathsf{X}\phi$ signifie « la formule ϕ doit être vraie dans le prochain état ». L'opérateur $\phi \mathsf{U} \psi$ signifie « la formule ϕ doit être vraie jusqu'à ce que la formule ψ le devienne ».

Les opérateurs donnés dans la Figure 3 sont les opérateurs primitifs de LTL. Pour faciliter l'expression de formules plus complexes on définit dans la Figure 4 des opérateurs additionnels.

$\phi \wedge \psi \Leftrightarrow \neg(\neg\phi \vee \neg\psi)$	(*et*)	$\Diamond\phi \Leftrightarrow \mathbf{true}\mathsf{U}\phi$	(*fatalement*)
$\phi \rightarrow \psi \Leftrightarrow \neg\phi \vee \psi$	(*implique*)	$\Box\phi \Leftrightarrow \neg\Diamond\neg\phi$	(*toujours*)
$\phi \leftrightarrow \psi \Leftrightarrow (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$	(*équiv. à*)	$\phi\mathsf{R}\psi \Leftrightarrow \neg(\neg\phi\mathsf{U}\neg\psi)$	(*relâche*)
		$\phi\mathsf{W}\psi \Leftrightarrow (\phi\mathsf{U}\psi) \vee \Box\phi$	(*jusqu'à faible*)

Figure 4 : Opérateurs additionnels pour LTL.

Les opérateurs booléens additionnels sont présentés dans la colonne de gauche. Par exemple, l'opérateur d'implication est défini à partir de l'opérateur de négation et du ou logique. L'opérateur R est le dual de U. On peut interpréter p R q par « la proposition q doit rester vraie, jusqu'à ce que p la relâche de cette obligation ». Enfin, l'opérateur p W q (jusqu'à faible, weak until) est une variante de U qui permet que la propriété q ne soit jamais vraie. Mais cela implique que p devra être toujours valide.

Notations alternatives. Suivant les domaines et les outils, l'opérateur suivant (X) est parfois noté ○, fatalement (◊) est également noté F ou <> et enfin toujours (□) est aussi noté G ou [ ].

3.2.3 Sémantique de LTL

Sémantique de LTL sur des chemins infinis. La sémantique de LTL sur un chemin infini σ dans une structure de Kripke est déterminée par la relation de satisfiabilité $\sigma \vDash \phi$ (la formule ϕ est valide pour le chemin σ). La Figure 5 définit cette relation, avec $p \in AP$ et ϕ, ψ des formules LTL.

$\sigma \vDash$ vrai		$\sigma \vDash \mathsf{X}\phi$	ssi $\sigma^1 \vDash \phi$,
$\sigma \nvDash$ faux		$\sigma \vDash \phi\mathsf{U}\psi$	ssi $\exists i \geq 0$ tel que $\sigma^i \vDash \psi$,
$\sigma \vDash p$	ssi $p \in \mathit{Etiquette}(\sigma[0])$,		et $\forall 0 \leq k < i$ on a $\sigma^k \vDash \phi$
$\sigma \vDash \neg\phi$	ssi $\sigma \nvDash \phi$,		
$\sigma \vDash \phi \vee \psi$	ssi $\sigma \vDash \phi$ ou $\sigma \vDash \psi$,		

Figure 5 : Sémantique de LTL sur un chemin infini.

La sémantique des cinq premières clauses reprend la définition de la logique propositionnelle. On remarquera que dans le cas du Until, i peut être égal à 0, la propriété ψ est alors vérifiée immédiatement et la validité de ϕ n'est pas prise en compte.

3.3. LTL et automates

3.3.1 Lien entre LTL et la théorie des langages

Pour vérifier qu'une exécution de longueur infinie vérifie une formule LTL donnée, l'approche la plus souvent utilisée est celle de la théorie des langages. L'idée est de spécifier la forme que doit avoir une exécution pour être valide à l'aide d'une expression ω-rationnelle (ou ω-régulière). Les expressions ω-rationnelles étendent les expressions rationnelles avec l'opérateur ω. Il indique une répétition *infinie* d'une partie de l'expression, contrairement à l'étoile de Kleene des expressions rationnelles qui indique un nombre *fini* de répétitions. Prenons la formule LTL $\Diamond p$ (p est fatalement vrai). Pour vérifier cette formule, une exécution $s_0, s_1, s_2 \ldots$ doit contenir un état vérifiant p. On dira que cette exécution doit être de la forme $(p \vee \neg p)^*.p.(p \vee \neg p)^\omega$. On peut associer à une formule LTL ϕ une expression ω-rationnelle $\mathcal{E}_\phi$ qui décrit les exécutions acceptées par ϕ. De plus, on peut construire un automate $\mathcal{B}_\phi$ qui va reconnaître les mots de $\mathcal{E}_\phi$ [68, 69]. Par exemple, pour la formule $\phi : \Diamond p$ les exécutions satisfaisant ϕ sont celles où à un moment donné, p devient vraie. La Figure 6 présente un automate reconnaissant de telles exécutions.

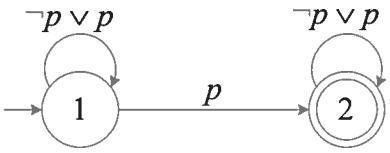


Figure 6 : $\mathcal{B}_\phi$ pour $\phi : \Diamond p$.

Il est important de noter que l'automate n'est pas déterministe. En effet, il peut « choisir », à n'importe quel moment où p est vérifié, de passer de l'état 1 à l'état 2. S'il reste infiniment dans l'état 2 il aura reconnu un comportement satisfaisant ϕ. L'état 2 est donc un état d'acceptation pour l'automate.

3.3.2 Automates de Büchi

Comme expliqué dans la partie 3.3.1, la vérification d'une formule LTL passe par la construction d'automates reconnaissant des mots infinis. Il en existe plusieurs type [70], mais la plupart des algorithmes se basent sur les automates de Büchi [71].

Automate de Büchi (BA – Büchi Automaton). Un automate de Büchi $\mathcal{B}$ est un quintuplet (Σ, S, S^0, F, T) où :

- Σ est un alphabet.
- S est une ensemble non vide d'états
- $S^0 \subseteq S$ est un ensemble non vide d'états initiaux.
- $F \subseteq S$ est un ensemble d'états acceptants.
- $T : S \times \Sigma \to 2^S$ est une relation de transition.

Critère d'acceptation de Büchi. Puisque les mots reconnus par les automates de Büchi sont infinis, on ne peut pas déterminer l'acceptation d'un mot à partir de l'état dans lequel se trouve l'automate à la fin de son exécution. Prenons le mot infini $w = a_0, a_1, a_2, \ldots$ sur Σ. Une exécution r de l'automate $\mathcal{B}$ sur w est une séquence d'état $s_0, s_1, s_2, \ldots$ tel que $s_0 \in S^0$ et $s_{i+1} \in T(s_i, a_i)$ pour tout $i \geq 0$. On note $\inf(r)$ l'ensemble $\{s \mid s = s_i \text{ pour une infinité de } i\}$, c'est-à-dire l'ensemble des états de r qui se répètent infiniment souvent. L'exécution r est « acceptante » si est seulement si $\inf(r) \cap F \neq \varnothing$, c'est-à-dire si il existe au moins un état final qui est visité infiniment souvent. Un mot w est accepté par $\mathcal{B}$ si il existe au moins une exécution de w sur $\mathcal{B}$ qui est « acceptante ».

Un automate de Büchi est de manière générale non déterministe, puisqu'il peut avoir plusieurs états initiaux et qu'il peut effectuer plusieurs transitions à la lecture d'un symbole donné. Mais contrairement à un automate reconnaissant des mots finis, un automate de Büchi déterministe est moins expressif qu'un automate de Büchi non-déterministe. C'est-à-dire qu'il existe des mots infinis pouvant être reconnus par un automate de Büchi non-déterministe mais pas par un automate de Büchi déterministe.

Automate de Büchi et LTL. Étant donné une formule LTL ϕ , on peut construire un automate de Büchi $\mathcal{B}_\phi = \left(\Sigma, S, S^0, T, F\right)$ où $\Sigma = 2^{AP}$ tel que le langage reconnu par $\mathcal{B}_\phi$ soit exactement l'ensemble des exécutions reconnues par ϕ [68].

Les étiquettes des transitions de l'automate obtenu sont les propositions atomiques devant être vérifiées pour que les transitions soient valides (voir Figure 6).

Il existe d'autres types d'automates sur des mots infinis, différant principalement par leurs critères d'acceptation [70]. On peut notamment citer les automates de Büchi généralisés, les automates de Muller et les automates de Rabin. Les automates de Büchi généralisés seront utilisés par la suite, ils seront donc définis formellement.

Automate de Büchi généralisé (GBA – Generalized Büchi Automaton). On défini un automate de Büchi généralisé par le tuple $\left(\Sigma, S, S^0, T, F\right)$. La différence par rapport à un automate de Büchi est la nature de l'ensemble F. Dans le cas des automates de Büchi généralisés on a $F \subseteq 2^S$; c'est-à-dire un ensemble de sous-ensembles acceptants de S. $F = \left\{F_1, \ldots, F_k\right\}$ pour $k \geq 1$ avec $F_i \subseteq S$. Une exécution r est acceptante si $\inf(r) \cap F_i \neq \varnothing$, pour tout $0 \leq i \leq k$. C'est-à-dire si tous les ensembles acceptants sont visités infiniment souvent.

3.3.3 Transformation de LTL vers un Automate de Büchi

Il existe un grand nombre d'algorithmes pour effectuer la transformation de LTL vers un automate de Büchi [68, 72-74]. La plupart passent par la forme normale négative de la formule LTL à transformer, souvent afin de simplifier les règles de transformation.

Forme normale négative. Une formule LTL est dite en forme normale négative si elle n'utilise que des éléments de *AP*, la négation d'éléments de *AP*, les opérateurs booléens $\wedge$ et $\vee$ ainsi que les opérateur temporels X, U et R. Cela implique notamment que l'opérateur de négation ne peut pas se trouver devant un opérateur temporel et que les opérateurs *toujours* et *fatalement* doivent être transformé en « *jusqu'à* ». On peut transformer une formule LTL quelconque dans une formule LTL en forme normale négative équivalente

sans en changer la sémantique. Le Tableau IV présente les règles de transformation à appliquer à une formule.

	Formule de départ	Forme normale nég.
Opérateurs booléens	$\phi \rightarrow \psi$ $\phi \leftrightarrow \psi$	$\neg \phi \vee \psi$ $(\phi \wedge \psi) \vee (\neg \phi \wedge \neg \psi)$
Opérateurs temporels	$\Box \phi$ $\Diamond \phi$	faux R ϕ vrai U ϕ
Négation op. booléens	$\neg \neg \phi$ $\neg (\phi \vee \psi)$ $\neg (\phi \wedge \psi)$ $\neg (\phi \rightarrow \psi)$ $\neg (\phi \leftrightarrow \psi)$	ϕ $\neg \phi \wedge \neg \psi$ $\neg \phi \vee \neg \psi$ $\neg \psi \wedge \phi$ $(\neg \phi \wedge \psi) \vee (\phi \wedge \neg \psi)$
Négation op. temporels	$\neg \Box \phi$ $\neg \Diamond \phi$ $\neg$ X ϕ $\neg (\phi$ U $\psi)$ $\neg (\phi$ R $\psi)$	vrai U $\neg \phi$ faux R $\neg \phi$ X $\neg \phi$ $\neg \phi$ R $\neg \psi$ $\neg \phi$ U $\neg \psi$

Tableau IV : Transformation de formule LTL en forme normale négative.

Ces règles proviennent essentiellement de la définition des opérateurs additionnels pour LTL présentée dans la Figure 4. Une fois la formule sous forme normale négative obtenue, elle peut être transformée en un automate de Büchi. On distingue trois grandes approches différentes.

La première [68] repose sur la construction de deux automates : l'automate local et l'automate des fatalités. Le premier va vérifier les incompatibilité locales du modèle, c'est-à-dire les incohérences sur les proposition atomiques. Il va également s'assurer que les formules temporelles sont localement (dans un état donné) vérifiées. Le deuxième automate, celui des fatalités, va quant à lui vérifier que pour les formules de fatalités de la forme ϕ U ψ, ψ va fatalement être vérifiées. Les deux automates sont enfin combinés par un produit synchronisé pour donner l'automate de Büchi final. Si l'algorithme est relativement simple, il a cependant un inconvénient majeur de par le nombre d'états générés. On obtient directement le pire cas avec un nombre d'état de l'ordre de $2^{\mathcal{O}(n)}$ où n est le nombre de sous-formules dans la formule de la propriété [72]. Un grand nombre d'états ne sont souvent pas atteignables.

Une autre approche passe par la construction d'un automate alternant [69, 74]. Un tel automate comporte un maximum de n états, avec n le nombre de sous formules [74]. Comme les algorithmes de model-checking nécessitent un automate de Büchi, il faut transformer cet automate alternant. Les algorithmes usuels pour cette transformation produisent en générale des automates à $2^n \times 2^n$ états. Une approche présenté dans [74] permet cependant d'obtenir un automate moins complexe. Le nombre d'état est réduit en utilisant un propriété particulière des automates alternants générés et en faisant des simplifications sur la formule de départ ainsi que sur les automates produits.

Enfin, un certain nombre d'algorithmes de transformation de LTL vers des automates de Büchi utilisent la technique des tableaux [75]. Le premier algorithme, utilisant la méthode dite des tableaux déclaratifs, produit un automate de Büchi généralisé avec un nombre d'états exponentiel dans le meilleur des cas [76]. Pour éviter de produire des états inaccessibles, la méthode des tableaux incrémentaux va ajouter les états au fur et à mesure de la décomposition de la formule. La version la plus connue de cette technique est l'algorithme présenté dans [72]. Dans la suite de ce mémoire, on fera référence à cette algorithme sous le nom de GPVW95. Il est capable de produire l'automate « à la volée », c'est-à-dire pendant le processus de vérification. L'automate généré est un automate de Büchi généralisé étiqueté. Une version modifiée de cet algorithme a été implémentée dans le cadre de ce projet, on va donc le présenter en détail.

Automate de Büchi généralisé étiqueté (LGBA – Labeled Generalized Büchi Automaton). La différence entre un GBA et un automate de Büchi généralisé étiqueté vient de la position des étiquettes. Dans un automate de Büchi généralisé étiqueté, les étiquettes sont placées non pas sur les transitions mais sur les états. Un LGBA est donc le sextuplet $\left(\Sigma, S, S^0, F, T, \mathcal{L}\right)$ où :

- Σ est un alphabet.
- S est une ensemble non vide d'états
- $S^0 \subseteq S$ est un ensemble non vide d'états initiaux.
- $F \subseteq 2^S$ est un ensemble d'ensembles d'états acceptants $F = \left\{F_1, \ldots, F_k\right\}$
- $T \subset S \times S$ est une relation de transition.

- $\mathcal{L}: S \to 2^{\Sigma}$ est une fonction d'étiquetage qui associe chaque état un ensemble d'étiquettes de Σ.

Critère d'acceptation. Un mot infini $w = a_0, a_1, a_2, \ldots$ sur Σ est accepté par un LGBA $\mathcal{A}$ s'il existe une exécution acceptante $r = s_0, s_1, s_2 \ldots$ sur $\mathcal{A}$ tel que $\forall i \geq 0, a_i \in \mathcal{L}(s_i)$.

L'algorithme construit un graphe représentant les états et les transitions de l'automate. La formule de départ est décomposée en sous-formules qui servent à étiqueter les états. La formule est découpée suivant sa structure booléenne et elle est « dépliée » suivant les opérateurs temporels. L'idée est de séparer ce qui doit être vérifié dans l'état courant et ce qui doit l'être dans l'état immédiatement suivant, selon la formule $\phi \mathsf{U} \psi \equiv \psi \vee (\phi \wedge \mathsf{X}(\phi \mathsf{U} \psi))$. Pour vérifier $\phi \mathsf{U} \psi$ on a deux possibilités : soit ψ est vérifié dans l'état courant, soit ϕ est vrai et $\phi \mathsf{U} \psi$ est vérifié dans l'état immédiatement suivant. Avant de détailler l'algorithme lui-même, on va présenter les structures de données utilisées pour représenter un nœud du graphe.

Un nœud est une structure de donnée contenant les attributs suivants :

Name : Une chaîne de caractères représentant le nom de l'état.

Incoming : L'ensemble des noms des nœuds précédant le nœud courant dans le graphe.

New : L'ensemble des propriétés temporelles qui doivent être vérifiées dans l'état courant et qui n'ont pas encore été traitées.

Old : L'ensemble des propriétés devant êtres vérifiées dans l'état courant et qui ont déjà été traitées.

Next : L'ensemble des propriétés devant être vérifiées dans les états qui suivent immédiatement l'état courant dans le graphe.

On suppose que la formule donnée en entrée est sous forme normale négative. Elle ne comporte donc plus que des opérateurs booléens $\wedge$ et $\vee$, des opérateurs temporels X et U et des propositions atomiques ou leurs compléments. L'algorithme est présenté ci-dessous (Code Source 13). On désigne par `node.New` le champ `New` du nœud `node`, etc. La fonction `NewName()` génère un nouveau nom pour un nœud à chaque appel.

```
1.  Node NewNode(string Name, StringSet Incoming,
2.                FormulaSet New, FormulaSet Old,
3.                FormulaSet Next);
4.
5.  NodeSet Expand(Node node, NodeSet set){
6.    if(node.New.IsEmpty){
7.      if(∃ ND ∈ set avec ND.Old==node.Old && ND.Next==ND.Next)
8.        ND.Incoming.Add(node.Incoming);
9.        return set;
10.     else{
11.       set.Add(node);
12.       return Expand(NewNode(NewName(), {node.Name},
13.                             node.Next, {}, {}),
14.                     set);
15.     }
16.   }else{
17.     Formula η = node.New.GetFormula();
18.     node.New.Remove(η);
19.     case(η){
20.       η = (P_n || !P_n || true || false):{
21.         if(η=false ou node.Old.Contains(Neg(η)))
22.           return set;
23.         else{
24.           node.Old.Add(η);
25.           return(Expand(node,set));
26.         }
27.       }
28.       η = φ U ψ || φ R ψ || φ ∨ ψ:{
29.         Node n1 = NewNode(NewName(), node.Incoming,
30.                           node.New ∪ New1(η).Remove(node.Old),
31.                           node.Old ∪ {η},node.Next ∪ Next(η));
32.         Node n2 = NewNode(NewName(), node.Incoming,
33.                           node.New ∪ New2(η).Remove(node.Old),
34.                           node.Old ∪ {η}, node.Next);
35.         return Expand(n2,Expand(n1, set));
36.       }
37.       η = φ ∧ ψ:{
38.         Node n = NewNode(node.Name, node.Incoming,
39.                          node.New ∪ {φ,ψ}.Remove(node.Old),
40.                          node.Old ∪ {η}, node.Next);
41.         return Expand(n, set);
42.       }
43.       η = X φ:
44.         return Expand(NewNode(NewName(), node.Incoming,
45.                               node.New, node.Old ∪ {η},
46.                               node.Next ∪ {φ}));
47.     }//case
48.   }//else
49. }//Expand
50.
51. NodeSet CreateGraphe(Formula φ){
52.   return Expand(NewNode(NewName(), {init}, {φ}, {}, {}), {});
53. }
```

Code Source 13 : L'algorithme de transformation GPVW95. Adapté de [72].

La méthode `NewNode` crée un nouveau nœud à partir des paramètres donnés. Le cœur de l'algorithme est la fonction `Expand`, ligne 5. Elle prend en paramètre le nœud à traiter ainsi que l'ensemble des nœuds déjà ajoutés au graphe. L'algorithme est initialisé (lignes 51-52) avec un nœud unique `n` et la propriété à vérifier est ajoutée à l'ensemble `n.New`. L'ensemble `n.Incoming` est initialisé avec le mot-clef « init » pour marquer les nœuds initiaux.

L'algorithme commence par vérifier s'il reste des formules à décomposer dans l'ensemble `node.New` (ligne 6). Si ce n'est pas le cas, c'est que le nœud a été entièrement traité et que toutes les formules devant être vérifiées dans ce nœud ont été décomposées. On ajoute alors ce nœud au graphe en vérifiant tout de même qu'un nœud équivalent `ND` n'existe pas déjà. Si c'est le cas, l'ensemble des transitions entrantes de `ND` est mis à jour en lui ajoutant les transitions entrantes du nœud courant (ligne 8). Deux nœuds sont dits équivalent si leurs ensembles `Old` et `Next` sont identiques (ligne 7). Si aucun nœud équivalent n'a été trouvé, le nœud courant est ajouté au graphe (ligne 11) et un nouveau nœud est construit (ligne 12). Ce nœud est le nœud suivant immédiatement le nœud courant. Les propriétés à vérifier dans ce nouveau nœud sont donc les propriétés de l'ensemble `Next` du nœud courant. Une transition du nœud courant vers le nouveau nœud est créée et la fonction `Expand` est rappelée sur ce nœud.

Si il reste des formules à décomposer dans l'ensemble `node.New`, une formule η est sélectionnée et supprimée de `node.New` (lignes 17 et 18). Suivant sa forme, la formule est décomposée :

- Si η est une proposition, la négation d'une proposition ou un littéral (*vrai* ou *faux*), on vérifie que le nœud ne contient pas de contradiction. C'est-à-dire qu'il ne contient pas p et $\neg p$ dans les propriétés devant être vérifiées (ligne 21). Si c'est le cas, le nœud est ignoré. Sinon, η est ajouté dans l'ensemble `node.Old` des formules vérifiées dans ce nœud et le traitement continue (lignes 24 et 25).
- Si η est de la forme $\varphi \wedge \psi$, alors φ et ψ sont ajoutés à l'ensemble `New`, puisque la validité des deux opérande est requise pour que la formule soit vraie.
- Si η est de la forme $\varphi \vee \psi$, le nœud est divisé en deux (lignes 29 à 34). Les deux nouveaux nœuds représentent les deux alternatives pour vérifier la formule. φ est

ajouté à l'ensemble `New` du premier nœud et ψ à celui du second nœud. Les autres ensembles du nœud de départ sont recopiés dans les nouveaux nœuds. Il faut remarquer qu'il s'agit bien d'une *division* du nœud courant et non pas la création de deux nouveaux successeurs.

- Si η est de la formue φ U ψ, alors il y a deux façon de vérifier la propriété : soit ψ est vérifiée dans l'état courant, soit φ est vérifiée dans l'état courant et φ U ψ est vérifiée dans l'état suivant. Ceci se traduit par la formule : φ U ψ ≡ ψ ∨ (φ ∧ X (φ U ψ)). Le nœud est donc divisé (lignes 29 à 34) pour vérifier les deux alternatives. Dans le premier on va ajouter ψ à `New`. Dans le second on ajoute φ à `New` et φ U ψ à `Next`.
- Si η est de la forme φ R ψ, on divise le nœud (lignes 29 à 34) en observant l'équivalence suivant : φ R ψ ≡ ψ ∧ (φ ∨ X (φ R ψ)).
- Si η est de la forme X φ, φ est simplement ajouté à l'ensemble `Next` du nœud.

Dans le Code Source 13, la fonction `Neg()` retourne le complément de la proposition atomique passée en argument. I.e. `Neg(p)` = ¬p et *Neg(¬p)* = *p*. Les fonctions `New1`, `Next` et `New2` déterminent les formules à ajouter aux différents ensembles `New` et `Next`, suivant le type d'opérateur rencontré. Elles sont définies dans le Tableau V.

Formule	New1	Next1	New2
φ U ψ	{φ}	{φ U ψ}	{ψ}
φ R ψ	{ψ}	{φ R ψ}	{φ, ψ}
φ ∨ ψ	{φ}	{}	{ψ}

Tableau V : Définitions de fonctions `New1`, `Next1` et `New2`.

A la fin de l'exécution, l'algorithme retourne un ensemble de nœud constituant les états de l'automate de Büchi généralisé étiqueté. Les états initiaux sont les nœuds dont l'ensemble `Incoming` contient le mot-clef « init ». Les transitions sont définies par les ensembles `Incoming` des différents nœuds : pour un état *s*, on a une transition $q \rightarrow s$ si $q \in s.Incoming$. L'alphabet Σ est 2^{AP} et l'étiquette d'un état *s* est l'ensemble des ensembles de 2^{AP} qui sont compatibles avec `s.Old`. De manière formelle, prenons $Pos(s) = s.Old \cap AP$ et $Neg(s) = \{\eta \mid \neg\eta \in s.Old \wedge \eta \in AP\}$. En d'autres termes, $Pos(s)$ est l'ensemble des

propositions atomiques apparaissants dans `s.Old` et $Neg(s)$ est l'ensemble des propositions atomiques complémentées apparaissants dans `s.Old`. Ainsi, la fonction d'étiquetage est définie par $\mathcal{L}(s) = \{X \mid X \subseteq AP \wedge X \supseteq Pos(X) \wedge X \cap Neg(X) = \varnothing\}$. Enfin, il faut déterminer l'ensemble F d'ensembles d'états acceptants. Pour chaque sous-formule η de la forme $\phi \mathsf{U} \psi$, on crée un ensemble $F_i \in F$ qui inclut tous les états q tel que soit $\phi \mathsf{U} \psi \notin q.Old$ ou $\psi \in q.Old$. Ainsi, toutes les exécutions acceptées vérifieront chacune des fatalités composant la formule à vérifier.

3.4. LTL pour la vérification par simulation

Les formalismes et algorithmes présentés jusqu'ici se rapportent au model-checking. Ils considèrent toutes les exécutions, possiblement infinies, du système à vérifier. Dans le cas de la vérification par simulation, seule une unique exécution finie est vérifiée. Dans [77], une taxonomie des techniques de vérification par simulation est proposée. Plusieurs critères de classification sont proposés. Le premier est le stockage de la trace d'exécution. Dans le cas de la simulation de systèmes matériels, les traces d'exécution peuvent être très longues et suivant le niveau d'abstraction du modèle, la quantité d'information à enregistrer à chaque cycle peut devenir importante. On préférera donc un traitement à la volée de la trace de simulation, c'est-à-dire pendant la simulation elle-même. Un autre critère est le caractère synchrone ou asynchrone de la détection d'erreur dans la trace. Si une violation de la règle est constatée au cours de la simulation, on veut pouvoir en avertir l'utilisateur et par exemple arrêter la simulation. Les techniques de vérification qui travaillent sur une trace complète sont de fait asynchrones, puisqu'elles doivent attendre la fin de la simulation avant de faire la vérification. On va ici considérer une technique synchrone et sans stockage de la trace de simulation.

3.4.1 Sémantique de LTL sur une trace finie

On peut considérer la vérification de trace comme un sous problème de la vérification formelle. Plusieurs adaptations sur les définitions sont nécessaires, principalement à cause de la nature finie des traces de simulation. La sémantique de LTL, traditionnellement donnée sur une exécution infinie, doit être modifiée. Ce problème a été

discuté dans [78, 79]. On considère généralement qu'un chemin fini est une portion de chemin infini et qu'il constitue ainsi le préfixe du chemin infini.

Sémantique de LTL sur des exécutions finies. Une exécution finie est une séquence d'états $\sigma = s_0, s_1, s_2, \ldots, s_{n-1}$, tel que $s_i \subseteq AP$ et $n \in \mathbb{N}$. Une formule LTL ϕ est valide sur l'exécution σ si la relation de satisfiabilité $\sigma \vDash \phi$ est vérifiée. La Figure 7 définit cette relation, avec $p \in AP$ et ϕ, ψ des formules LTL.

$\sigma^i \vDash$ vrai	
$\sigma^i \nvDash$ faux	
$\sigma^i \vDash p$	ssi $p \in s_i$,
$\sigma^i \vDash \neg\phi$	ssi $\sigma^i \nvDash \phi$,
$\sigma^i \vDash \phi \vee \psi$	ssi $\sigma^i \vDash \phi$ ou $\sigma^i \vDash \psi$,
$\sigma^i \vDash \mathsf{X}\phi$	ssi $i < n$-1 et $\sigma^{i+1} \vDash \phi$,
$\sigma^i \vDash \phi \mathsf{U} \psi$	ssi $\exists i \leq k < n$ tel que $\sigma^k \vDash \psi$
	et $\forall i \leq j < k$ on a $\sigma^j \vDash \phi$

Figure 7 : Sémantique de LTL sur une exécution finie.

Les opérateurs additionnels définis dans la Figure 4 restent valides, en adaptant évidemment leurs sémantiques.

Cette nouvelle sémantique est l'adaptation directe de la sémantique classique de LTL. Si on considère qu'une exécution finie est en fait une exécution infinie tronquée, certaines formules prennent un sens particulier. Par exemple, la signification des formules de la forme « □ φ » (toujours φ) changent légèrement. En effet, on peut vérifier que la propriété φ est valide sur la totalité du chemin dont on dispose mais on ne peut rien conclure sur la validité du chemin infini d'origine. Le même problème se pose pour des formules du type « ◊ φ » sur des chemins finis où φ n'est jamais valide.

Les auteurs de [79] proposent donc trois interprétations du résultat. Dans la première, une formule est déclarée valide s'il y a un doute sur sa validité. Ainsi, la formule « ◊ φ » est valide pour n'importe quel chemin fini, même si la propriété φ n'est pas vérifiée sur la portion observée. De plus, « □ φ » est valide si φ est valide sur la totalité du chemin. Cette interprétation est appelée « vue faible » (*weak view*). Dans la seconde interprétation,

la « vue forte » (*strong view*), une formule est déclarée fausse si il y a un doute sur sa validité. Ainsi, la formule « ◊ φ » est valide si φ est vérifiée sur au moins un état du chemin. Par contre, « □ φ » n'est jamais vérifiée avec une vue forte. Car même si φ est vérifiée sur la totalité du chemin fini, on ne sait rien de son état sur le chemin complet. L'interprétation intermédiaire, baptisée « vue neutre » (*neutral view*) est la plus intuitive. Dans cette vue, une formule « □ φ » est vérifiée si φ est vérifiée sur toute la longueur de la trace et « ◊ φ » est vérifiée si φ devient vraie dans la trace. C'est cette sémantique que l'on a adoptée et qui est présentée dans la Figure 7.

3.4.2 Vérification de LTL

Pour vérifier une formule LTL sur une trace finie, plusieurs techniques sont possibles. On peut distinguer deux grandes approches : l'utilisation d'automates finis ou la manipulation directe de la formule. Dans la première, la formule est transformée en un automate qui est utilisé pour vérifier la trace. Dans la seconde, la propriété à vérifier reste sous forme de formule temporelle et est transformée pendant l'exécution de la trace [80-82]. L'idée est qu'à chaque nouvelle étape de l'exécution, on regarde si l'état du système est compatible avec la formule actuelle, puis on réécrit cette formule suivant ce qui doit être vérifié dans le prochain état. Par exemple, si on a la formule $\phi \wedge \mathsf{X}(\psi \wedge \eta)$ on vérifie que ϕ est vrai et la formule est transformée en $(\psi \wedge \eta)$ qui sera valide à la prochaine étape. Pour l'opérateur *jusqu'à*, on utilise la relation $\phi \mathsf{U} \psi \equiv \psi \vee (\phi \wedge \mathsf{X}(\phi \mathsf{U} \psi))$. Cet algorithme s'apparente à celui présenté dans la partie 3.3.3, mais avec une exécution dynamique. Alors que la construction de l'automate se fait avant le début de la simulation, la transformation de la formule se fait pendant le déroulement de cette dernière.

L'utilisation d'un automate pour reconnaître une trace d'exécution valide est plus classique [83-85]. Les algorithmes de transformation de LTL vers des automates sont connus et largement étudiés. Mais l'automate obtenu n'est pas déterministe, ce qui implique un algorithme particulier pour l'exécution de l'automate. Prenons l'automate de Büchi de la Figure 8, généré à partir de la formule $(a \wedge \mathsf{X}c) \vee (a \wedge b \wedge \mathsf{X}d)$. Cet automate n'est pas déterministe, car les deux transitions sortantes de l'état 1 sont valides si les propriétés atomiques a et b sont vraies dans le premier pas de la trace. Comme on ne peut pas

transformer cet automate en un automate déterministe sans modifier le langage reconnu, il faut l'exécuter en respectant ce non-déterminisme. L'algorithme proposé par [83], effectue un parcours en largeur de l'automate. Il est présenté par le Code Source 14 dans un pseudo-code.

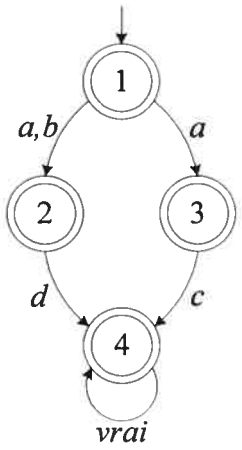


Figure 8 : Automate de Büchi pour la formule $(a \wedge \mathbf{X}c) \vee (a \wedge b \wedge \mathbf{X}d)$.

```
1.  currentStates = {initState};
2.  foreach(step in trace){
3.    newCurrentStates = {};
4.    foreach(state in currentStates){
5.      foreach(transition in state){
6.        if(transition.Label.Contains(step)){
7.          newCurrentState.Add(transition.Destination);
8.        }
9.      }
10.   }
11.   if(newCurrentStates.IsEmpty)
12.     return InvalidTrace;
13.   currentStates = newCurrentStates;
14. }
```

Code Source 14 : Algorithme d'exécution de l'automate.

Plutôt que d'avoir un seul état courant, on va considérer un ensemble d'états. Il contient l'état initial au début de l'algorithme (ligne 1). Ensuite, à chaque étape de la trace, toutes les transitions valides sont exécutées (lignes 4 à 6) et le nouvel état courant est mis à jour (ligne 13). Si, pour un pas donné, aucune transition n'est possible, c'est que la règle a été violée.

Le critère d'acceptation d'un automate de Büchi n'est plus utilisable dans le cas de mots finis. Si l'automate se termine sans arriver dans un état d'acceptation, cela signifie que

la portion de trace examinée n'est peut-être pas suffisamment longue pour pouvoir conclure de façon certaine sur sa validité. L'automate de Büchi est donc considéré comme un automate fini classique [83] et un mot est accepté si au moins un des états dans lequel se trouve l'automate à la fin de la trace est un état d'acceptation.

3.5. Logique Temporelle Temps-Réel

Dans la plupart des logiques temporelles, la notion de temps est qualitative. Il s'agit de définir l'ordre des états, sans donner de valeurs temps quantitatives. Plusieurs extensions ont été proposées pour étendre ces logiques avec des opérateurs « temps-réel » [86]. Elles se différencient principalement par le modèle de temps utilisé (continu ou discret) et leur expressivité. Une logique temporelle temps-réel, *Metric Temporal Logic* (MTL), étend LTL avec des bornes de temps sur les opérateurs temporels, représentant leurs périodes de validité [87]. Par exemple, $\Diamond_{\leq 4}\phi$ exprime la propriété « ϕ doit devenir vraie dans les 4 prochaines unités de temps ». La sémantique habituelle de MTL est définie sur une séquence d'états avec une étiquette de temps. Les algorithme de vérification de traces à partir de spécification MTL utilisent souvent les techniques de réécriture de la formule [82]. La Logique Temporelle Linéaire Temps-Réel (RTLTL), présentée dans [84], est similaire à MTL. Le modèle de temps est discret, mais sa sémantique est définie sur une trace sans étiquette de temps. Les valeurs de temps se réfèrent alors au nombre d'états dans la trace. Par exemple, la propriété $\Diamond_{\leq 4}\phi$ signifie « ϕ doit être vérifié avant le $4^{\text{ème}}$ état ».

Syntaxe de RTLTL. La Syntaxe de RTLTL est définie dans la Figure 9. Les opérateurs additionnels définis pour LTL sont aussi valides pour RTLTL.

---

p est une formule RTLTL si $p \in AP$;

Si ϕ est une formule RTLTL alors $\neg\phi$ est une formule RTLTL ;

Si ϕ et ψ sont des formules RTLTL alors $\phi \wedge \psi$ est une formule RTLTL ;

Si ϕ est une formule RTLTL alors $\mathsf{X}\phi$ et $\phi \mathsf{U} \psi$ sont des formules RTLTL ;

Si ϕ et ψ sont des formules RTLTL et $a, b \in \mathbb{N}$ alors $\mathsf{X}_a \phi$ et $\phi \mathsf{U}_{a,b} \psi$ sont des formule RTLTL.

Figure 9 : Syntaxe de RTLTL.

Sémantique de RTLTL. Comme pour LTL, elle est définie sur une exécution finie σ par la relation de satisfiabilité $\sigma \vDash \phi$. La Figure 10 définit cette relation.

$\sigma^i \models$ vrai	
$\sigma^i \not\models$ faux	
$\sigma^i \models p$	ssi $p \in s_i$
$\sigma^i \models \neg\phi$	ssi $\sigma^i \not\models \phi$
$\sigma^i \models \phi \vee \psi$	ssi $\sigma^i \models \phi$ ou $\sigma^i \models \psi$
$\sigma^i \models \mathsf{X}\phi$	ssi $i < n$-1 et $\sigma^{i+1} \models \phi$
$\sigma^i \models \mathsf{X}_{a+1}\phi$	ssi $i < n - a$ et $\sigma^{i+a} \models \phi$
$\sigma^i \models \phi \mathsf{U} \psi$	ssi $\exists i \le k < n$ tel que $\sigma^k \models \psi$
	et $\forall i \le j < k$ on a $\sigma^j \models \phi$
$\sigma^i \models \phi \mathsf{U}_{a,b} \psi$	ssi $\exists i + a \le k < i + b, k \le n$ tel que $\sigma^k \models \psi$
	et $\forall i \le j < k$ on a $\sigma^j \models \phi$

Figure 10 : Sémantique de RTLTL.

On remarque que RTLTL est sémantiquement équivalent à LTL [84], puisqu'on peut dérouler une formule RTLTL en une formule LTL équivalente avec les relations d'équivalences de la Figure 11.

$$\mathsf{X}_{a+1}\phi \equiv \mathsf{X}\mathsf{X}_a\phi$$
$$\phi \mathsf{U}_{a+1,b+1} \psi \equiv \phi \wedge X\left(\phi \mathsf{U}_{a,b} \psi\right)$$
$$\phi \mathsf{U}_{0,b+1} \psi \equiv \psi \vee \left(\phi \wedge X\left(\phi \mathsf{U}_{0,b} \psi\right)\right)$$
$$\phi \mathsf{U}_{0,0} \psi \equiv \psi$$

Figure 11 : Equivalence entre RTLTL et LTL.

Les opérateurs temporels bornés de RTLTL sont considérés comme des sucres syntaxiques de LTL. Cette équivalence permet également d'employer les algorithmes définis pour LTL, notamment pour la transformation en un automate de Büchi. L'approche suggérée dans [84] est de modifier l'algorithme présenté dans la partie 3.3.3 en utilisant les équivalences données dans la Figure 11. Cette technique est particulièrement élégante, puisque seules les fonction `New1()`, `Next1()` et `New2()` (voir le Code Source 13) doivent être modifiées. Le Tableau VI donne la nouvelle version de ces fonctions.

Formule	New1	Next1	New2
$\varphi \, U \, \psi$	$\{\varphi\}$	$\{\varphi \, U \, \psi\}$	$\{\psi\}$
$\varphi \, U_{a+1,b+1} \, \psi$	$\{\varphi\}$	$\{\varphi \, U_{a,b} \, \psi\}$	$\{\}$
$\varphi \, U_{0,b+1} \, \psi$	$\{\varphi\}$	$\{\varphi \, U_{0,b} \, \psi\}$	$\{\psi\}$
$\varphi \, U_{0,0} \, \psi$	$\{\psi\}$	$\{\}$	$\{\}$
$\varphi \, R \, \psi$	$\{\psi\}$	$\{\varphi \, R \, \psi\}$	$\{\varphi, \psi\}$
$\varphi \, R_{a+1,b+1} \, \psi$	$\{\psi\}$	$\{\varphi \, R_{a,b} \, \psi\}$	$\{\}$
$\varphi \, R_{0,b+1} \, \psi$	$\{\psi\}$	$\{\varphi \, R_{0,b} \, \psi\}$	$\{\varphi,\psi\}$
$\varphi \, R_{0,0} \, \psi$	$\{\varphi\}$	$\{\}$	$\{\}$
$\varphi \vee \psi$	$\{\varphi\}$	$\{\}$	$\{\psi\}$

Tableau VI : Définition des fonctions *New1*, *Next1* et *New2* pour RTLTL.

3.6. Patrons de spécification

Dans cette partie, nous allons donner quelques exemples de propriétés temporelles qui se retrouvent souvent dans les spécifications. Un important travail de recensement et de classification a été effectué par les auteurs de [88]. Ils partent du constat que si les méthodes de vérification formelles sont efficaces, elles sont rarement adoptées par les concepteurs ; principalement à cause du manque d'expertise et de la difficulté de formaliser les spécifications. Ils proposent donc d'identifier certains patrons de formules qui reviennent souvent dans les spécifications et de les classer, comme ce qui a été fait en génie logiciel par [89]. Des implémentations de ces patrons sont données dans plusieurs logiques temporelles, dont LTL. L'ensemble de ces patrons est disponible en ligne sur [90]. La Figure 12 donne la classification arborescente adoptée.

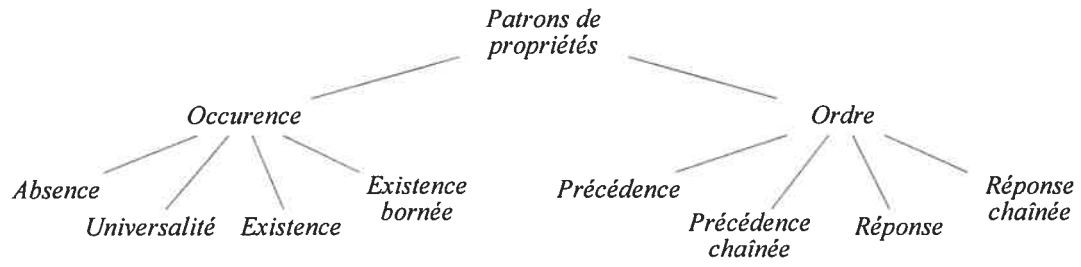


Figure 12 : La hiérarchie des patrons de spécification [88].

Les patrons sont des propriétés temporelles paramétrables. Huit ont été identifiés (les feuilles de l'arbre dans la Figure 12), répartis dans deux catégories : Les propriétés sur l'occurrence d'états et les propriétés sur l'ordre des états. Dans la première on retrouve les propriétés suivantes :

- L'absence : un état n'est jamais atteint, ce qu'on peut traduire en LTL par $\neg\Diamond P$. On peut utiliser ce patron pour traduire des propriétés d'exclusion mutuelles.
- L'universalité : On reste toujours dans un état, $\Box P$ en LTL.
- L'existence : On doit attendre un état donné, $\Diamond P$.
- L'existence bornée : On peut atteindre un état au maximum n fois. Pour n=2, on a la formule LTL $\left(\neg P \mathrm{W}\left(P \mathrm{W}\left(\neg P \mathrm{W}\left(P \mathrm{W} \Box \neg P\right)\right)\right)\right)$.

Les patrons d'ordre sont :

- Précédence : Un état P doit toujours être précédé d'un état S.
- Réponse : Un état P doit toujours être suivi d'un état S. Ce patron est particulièrement utile dans la vérification de protocole. En LTL, il se traduit par $\Box\left(P \rightarrow \Diamond S\right)$.
- La précédence chaînée : Une séquence d'états $P_1,\ldots,P_n$ doit toujours être précédée d'une séquence d'états $S_1,\ldots,S_m$.
- La réponse chaînée : Une séquence d'états $P_1,\ldots,P_n$ doit toujours être suivie d'une séquence d'états $S_1,\ldots,S_m$.

Les auteurs introduisent en plus la notion de portée (*scope*) pour les patrons. Par exemple, on peut vérifier une propriété d'absence uniquement entre deux états définissant la portée de la propriété. Les portées sont données dans la Figure 13.

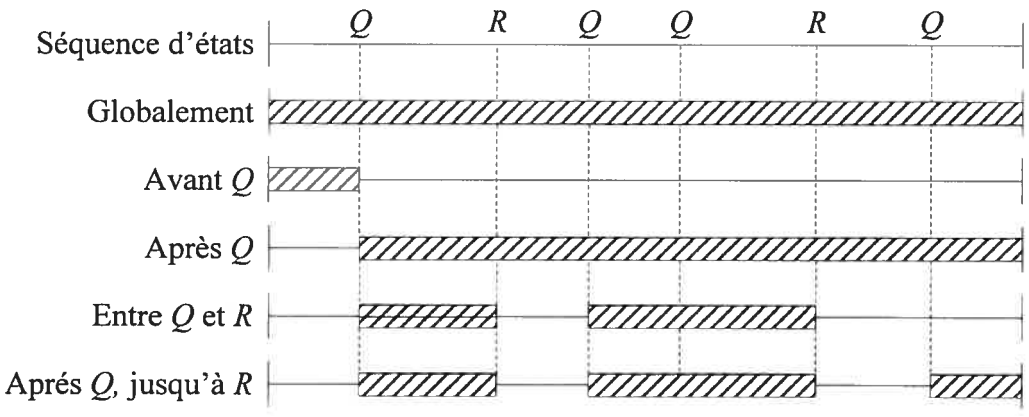


Figure 13 : Portées des patrons de spécification.

Ainsi, la propriété de réponse « P est toujours suivi de S » avec une portée « entre Q et R » devient $\square\left((Q \wedge \neg R \wedge \Diamond R) \rightarrow \left(P \rightarrow \left(\neg R \mathsf{U} (S \wedge \neg R)\right)\right) \mathsf{U} R\right)$. C'est sur de telles formules que les patrons de conception deviennent intéressants. En effet, exprimer des propriétés complexes en LTL n'est pas trivial et le risque d'erreur est grand.

Pour écrire une propriété temporelle à partir d'un patron, les paramètres (P, Q, R, S,...) sont remplacés par des formules décrivant les états. Ces formules peuvent être de simple proposition atomique, des formules propositionnelles ou bien des formules LTL plus complexes. Dans ce cas, il faut faire attention à la réelle signification de la formule obtenue, qui peut être subtile. C'est souvent le cas pour les patrons utilisant les portées [90].

3.7. Discussions

Dans cette partie, nous avons présenté un ensemble de formalismes et d'algorithmes nécessaire à la vérification semi-formelle. Ils ont été pour la plupart développé pour le *model-checking*. Comme nous l'avons vu, la vérification formelle n'est pas applicable à des modèles ESys.NET. Mais l'utilisation de techniques semi-formelle permet d'améliorer l'efficacité du processus de vérification, notamment en permettant de spécifier des comportements complexes avec la logique temporelle linéaire. Le chapitre suivant présente notre implémentation d'un outil de vérification semi-formelle dans l'environnement ESys.NET.

Chapitre 4 Implémentation – Moteur de vérification

Dans ce chapitre, nous allons présenter en détail l'architecture et l'implémentation de notre moteur de vérification pour ESys.Net. Le moteur de vérification est constitué d'un ensemble de bibliothèques permettant de construire une application complète de simulation et de vérification.

Dans une première partie, nous allons introduire un exemple de modèle ESys.Net qui sera utilisé par la suite pour illustrer les différents concepts présentés. Dans la partie suivante, nous donnerons un aperçu global du flux de vérification et des différentes étapes qui le compose. La partie 4.3 présentera la bibliothèque facilitant l'introspection d'un modèle ESys.Net et encapsulant les opérations courantes. Le langage de spécification de propriétés sera introduit dans la partie 4.4. Les parties 4.5 et 4.6 présenteront la construction des automates et du moteur de vérification lui-même. Enfin, l'exécution du processus de vérification sera détaillée dans la dernière partie.

4.1. Exemple de modèle

Le système modélisé est un système de producteur/consommateur synchrone. Il est composé d'un sous-système qui produit des nombres aléatoires et qui les transmet à un deuxième sous-système qui imprime les données reçues sur la sortie standard. Ils sont synchronisés par un signal d'horloge. La Figure 14 donne une représentation graphique du modèle.

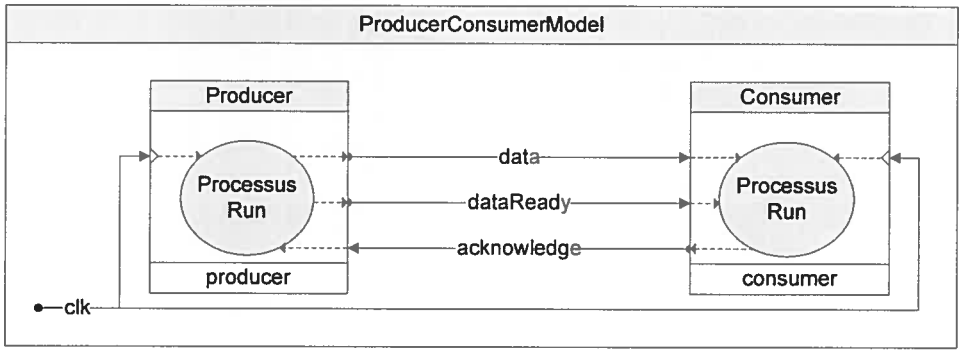


Figure 14 : Représentation graphique du modèle de producteur/consommateur.

Les deux sous-systèmes sont modélisés par des modules `Producer` et `Consumer`. Ces modules communiquent avec trois signaux. Le premier transmet les données (signal `data`). Le signal `dataReady` indique qu'une nouvelle donnée est disponible. Enfin, le dernier signal transmet l'acquittement de réception du consommateur (signal `acknowledge`). Le protocole de transmission est donné dans le chronogramme de la Figure 15.

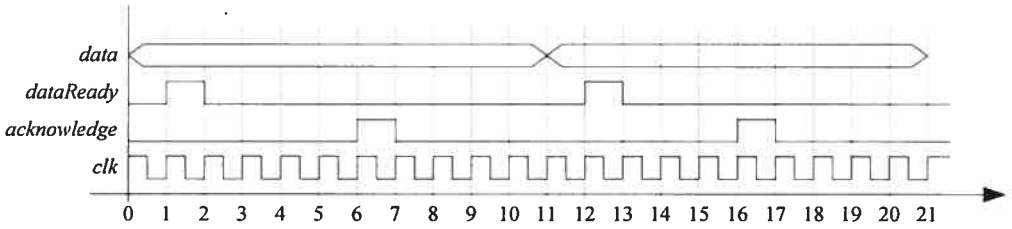


Figure 15 : Chronogramme du protocole de communication.

Les délais de production d'une donnée et de réponse sont aléatoires. Les signaux `dataReady` et `acknowledge` sont à l'état haut pendant un cycle maximum et ne peuvent être actifs en même temps. Le producteur doit obligatoirement attendre la réponse du consommateur avant d'émettre une nouvelle donnée.

Le code source du modèle est donné dans l'Annexe A.

4.2. Présentation du flux de vérification

L'outil de vérification développé utilise l'architecture des observateurs, également appelés moniteurs. Le modèle est simulé par ESys.Net et le moteur de vérification est exécuté en parallèle. Ce dernier va observer l'évolution de l'état du modèle et vérifier la conformité du comportement par rapport à un ensemble de règles. La Figure 16 représente les flux de simulation et de vérification.

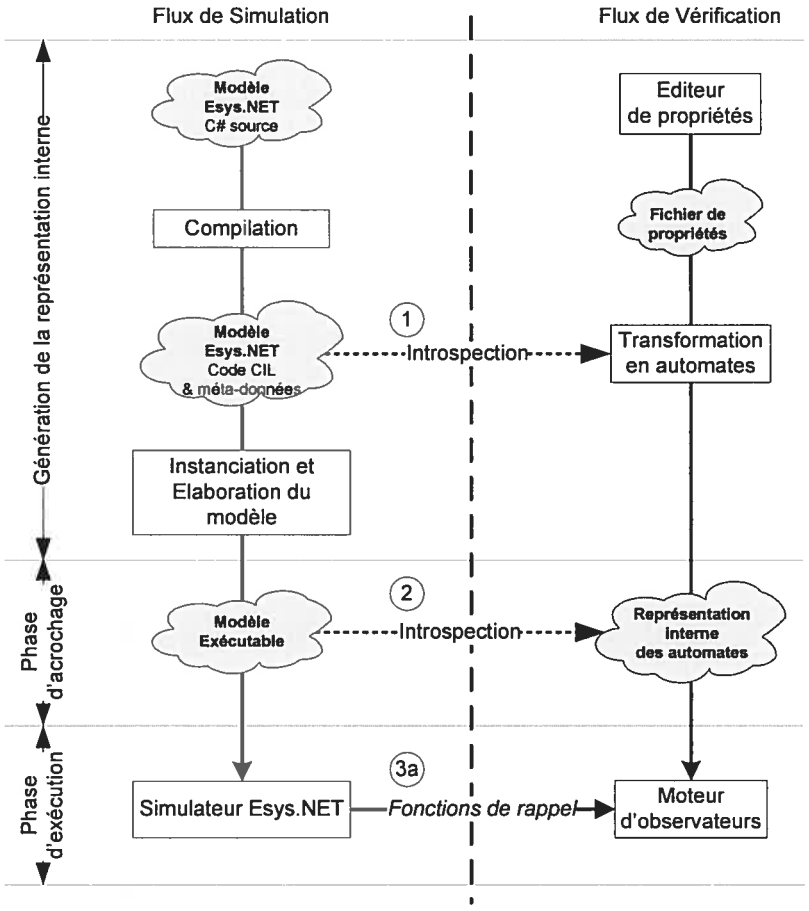


Figure 16 : Les flux de simulation et de vérification.

La partie gauche de la figure représente le flux de simulation. Le modèle ESys.Net est un programme utilisant la bibliothèque de classes d'ESys.Net. Il est écrit dans un des langages pris en charge par .NET. Le résultat de la compilation de ce programme est une « assembly » contenant le code CIL et les méta-données représentant le modèle. Le modèle est ensuite instancié et élaboré par ESys.Net, puis simulé.

Le flux de vérification est décrit dans la partie droite. La première étape est la spécification d'un ensemble de propriétés à vérifier. Ces propriétés sont formalisées en LTL et sont regroupées dans un fichier texte. La syntaxe utilisée et le format de ce fichier sont donnés dans la partie 4.4. Ces propriétés sont ensuite transformées en automates (partie 4.5). Chaque formule est transformée en un automate qui constituera un observateur. Diverses vérifications sont alors faites sur la syntaxe des formules et sur leur compatibilité avec le modèle vérifié. Ces vérifications sont faites à l'aide de l'introspection sur la représentation binaire du modèle (opération 1 sur la Figure 16). Une fois les observateurs

construits, ils sont liés au modèle au cours de la phase d'accrochage (opération 2 sur la Figure 16). Enfin, ils rassemblés dans le moteur d'observateur qui va assurer la synchronisation entre le simulateur et les observateurs (opération 3 sur la Figure 16). La partie 4.6 détaille la liaison entre les observateurs et le modèle, ainsi que le fonctionnement du moteur.

4.2.1 Comparaison avec SystemC

Avant d'expliquer la structure et le fonctionnement du moteur d'observateur, nous allons comparer brièvement notre approche avec un environnement de vérification *hypothétique* basé sur SystemC. L'architecture et les fonctionnalités des deux environnements sont similaires afin de bien mettre en valeur l'intérêt de .NET et ESys.Net pour le développement d'outils de vérification.

Même si ESys.Net et SystemC ont des architectures et des sémantiques de simulation proches, ils diffèrent énormément du point de vue de leur extensibilité. Cette différence est principalement due aux environnements sur lesquels ils sont bâtis, notamment en ce qui concerne l'introspection. Les capacités de réflexion de C++ limitent SystemC dans le développement d'environnement de vérification. L'introspection est nécessaire pour construire un outil souple, c'est-à-dire dont le processus de vérification reste indépendant de la modélisation et qui permet la manipulation de types de données arbitraires. Différent projets ajoutent une couche d'introspection à SystemC, sur la structure du modèle [52, 53] ou sur les données manipulée [46, 91]. Outre le problème de l'introspection, SystemC a besoin d'être modifié pour permettre l'interaction avec l'outil de vérification.

La Figure 17 présente les flux de vérification et de simulation du système *hypothétique* qui servira à la comparaison.

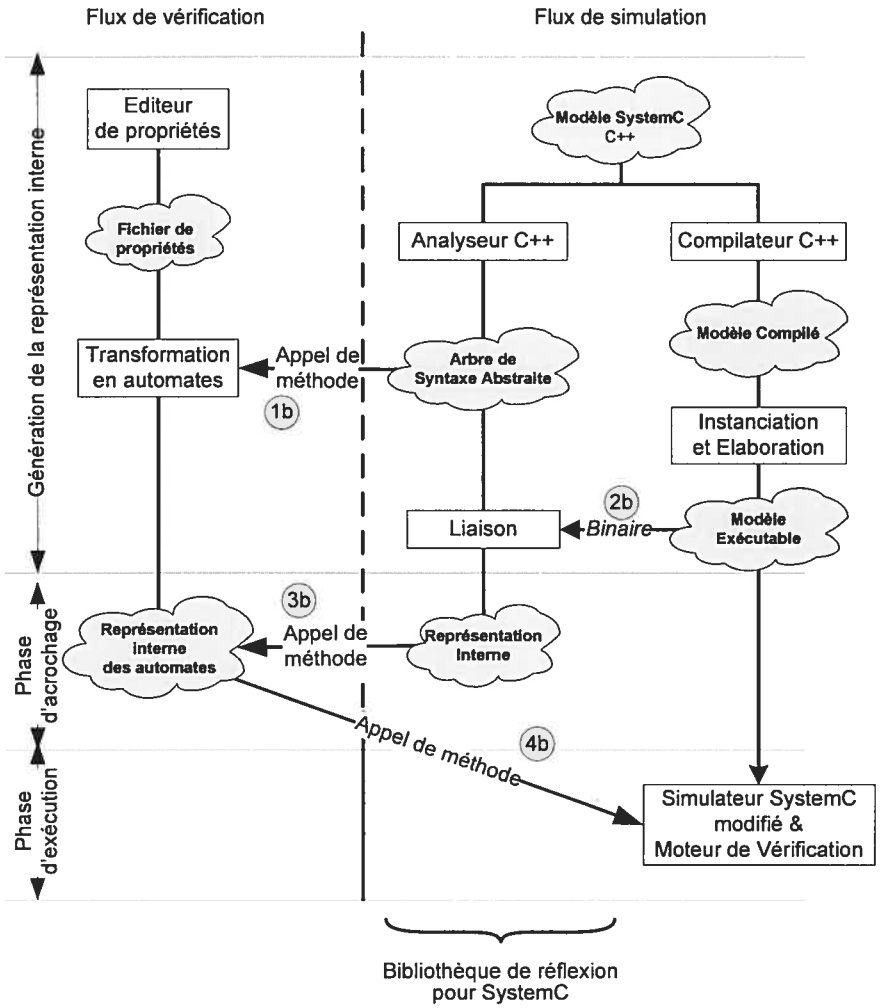


Figure 17 : Un environnement de vérification pour SystemC.

La tâche la plus complexe est certainement l'ajout d'une bibliothèque de réflexion à SystemC. La réflexion est nécessaire pour extraire la structure du modèle et obtenir l'état du modèle pendant la simulation. De plus, elle permet de manipuler des types de données arbitraires. Une première méthode serait d'analyser directement le fichier binaire résultant de la compilation pour en extraire l'architecture du modèle. La lecture de l'état du modèle peut se faire sur les structures de données en mémoire. Cette approche est difficilement praticable car la dépendance est très forte vis-à-vis de la plateforme ciblée par la compilation, du système d'exploitation et du compilateur. L'approche présentée ici, qui s'apparente à celle proposée par PINAPA [52], repose sur un analyseur lexical qui extrait la structure du modèle. L'arbre de syntaxe abstraite construit par le *front-end* d'un compilateur contient l'information nécessaire dans la définition des classes qui composent

le modèle. Cet arbre est utilisé pour vérifier la syntaxe et le type des propriétés (Figure 17 – 1b). Pour extraire l'état du modèle pendant la simulation, un lien doit être fait en la hiérarchie abstraite du modèle et l'instance qui est simulée. Cette liaison se fait en explorant les objets binaires en mémoire pour obtenir l'adresse de leurs membres (Figure 17 – 2b). Le compilateur utilisé pour compiler le modèle doit être modifié pour exporter les informations requises. Le résultat de cette opération permet de lier les observateurs à l'instance du modèle simulé (Figure 17 – 3b). Enfin, la synchronisation entre la simulation et l'exécution des observateurs implique la modification de SystemC. Le simulateur offre des fonctions de rappel à différents points de la simulation. Ils peuvent être utilisés pour la synchronisation mais ne sont accessibles qu'aux éléments du modèle. Il faut donc incorporer le moteur de vérification au simulateur ou exporter les fonctions de rappels. Dans la Figure 17, l'opération 4b représente l'appel du moteur de vérification aux observateurs. La manipulation de types de données arbitraires par l'outil de vérification peut se faire avec une approche similaire à SCV [46] ou à [91].

Le développement d'une couche d'introspection pour SystemC est une tâche fastidieuse. Même si de nombreux projets sont en cours pour pallier ce manque, l'absence de solutions simples d'utilisation et standards pénalise le développement d'outils autours de ce langage.

4.3. Bibliothèque d'introspection pour ESys.Net

Avant de présenter l'outil de vérification proprement dit, nous allons décrire la bibliothèque qui sert de base aux opérations d'introspection sur des modèles ESys.Net. Un des avantages majeur d'ESys.Net par rapport aux autres environnements de modélisation est sa capacité d'introspection, héritée de la plateforme .NET. L'introspection est utilisée par ESys.Net lui-même pendant la phase d'élaboration du modèle. L'introspection est ici employée pour vérifier le comportement du modèle pendant la simulation, mais d'autres utilisations sont envisageables : la visualisation de la structure et de l'état du modèle, l'analyse du taux de couverture d'une simulation ou encore la synthèse comportementale. Afin de faciliter les développements futurs autour d'ESys.Net, une bibliothèque de classes indépendante du processus de vérification a été développée. Elle encapsule les opérations d'introspection courantes sur des modèles ESys.Net. La bibliothèque est composée de deux

parties indépendantes. La première partie est chargée de construire en mémoire une représentation abstraite de la structure du modèle à partir de la hiérarchie des modules. La structure de donnée représentant la hiérarchie du modèle est appelée *Design Structure Tree*. La seconde partie, le *ModelResolver*, permet de récupérer des données dans le modèle pendant la simulation.

4.3.1 *Design Structure Tree*

Le *Design Structure Tree* (DST) est une représentation abstraite de la structure hiérarchique du modèle. L'objectif est de construire un arbre qui contient les informations sur la structure et sur les types définissants le modèle. Ces informations sont utilisées pour la visualisation et pour vérifier la syntaxe des propriétés. L'arborescence représente l'imbrication des modules des divers éléments du modèle. La structure de données enregistre les informations sur les éléments suivants : les signaux, les canaux de communication, les horloges, les événements, les propriétés C# et les variables d'instances non spécialisées (qui ne sont pas des éléments de modélisation comme des modules ou des signaux).

Une représentation graphique partielle du DST du modèle de producteur /consommateur est donnée dans la Figure 18.

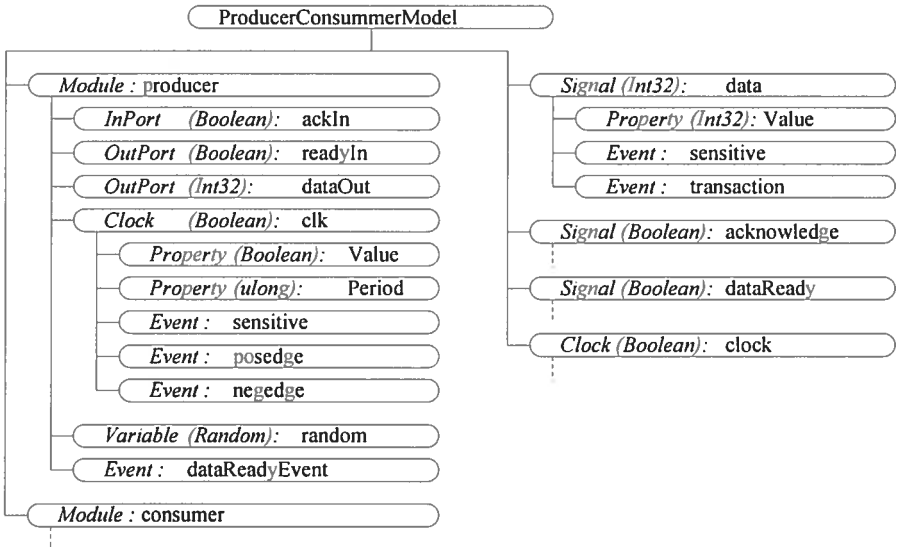


Figure 18 : Le DST partiel du modèle de producteur/consommateur.

La racine de l'arbre représente le type `ProducerConsumerModel` héritant de `SystemModel` et qui contient la totalité du modèle (voir code source en Annexe A). Au premier niveau, les deux modules constituant le système sont représentés (dans la colonne de gauche), ainsi que les signaux de communication et l'horloge (dans la colonne de droite). Pour les signaux de données et l'horloge, le type de la donnée transportée par le signal en question est enregistré. Il est indiqué entre parenthèse. C'est également le cas pour les variables non spécialisées et pour les propriétés. Cette information n'a pas de sens pour les modules et les événements qui ne véhiculent pas de données. L'interface du module `producer` est enregistrée : les ports d'entrées/sorties ainsi que le signal d'horloge sont des nœuds fils du nœud `producer`. La variable d'instance `random` interne au module figure également dans l'arbre.

La construction du DST se fait par l'exploration récursive du type héritant de `SystemModel`. Chaque fois qu'un nouvel élément devant être enregistré est découvert, il est ajouté à l'arbre. Cette exploration est statique dans le sens où elle se fait non pas sur une instance particulière du modèle, mais sur le type définissant ce modèle. La structure de donnée utilisée pour stocker l'arbre en mémoire est représentée par le schéma UML de la Figure 19.

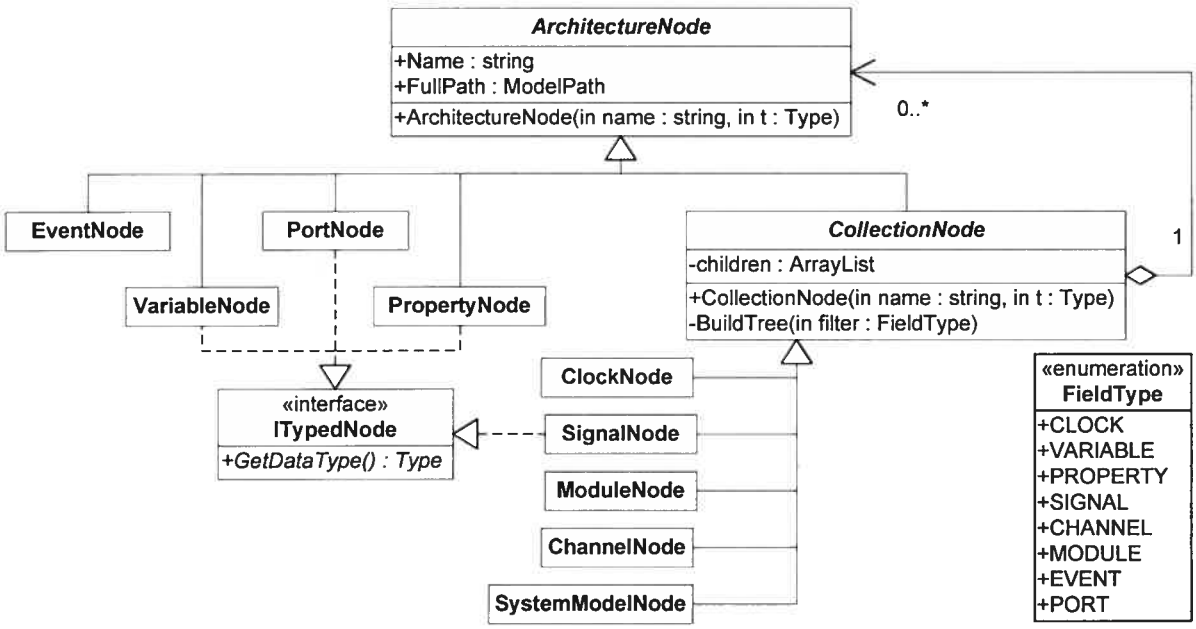


Figure 19 : Diagramme de classes du *Design Structure Tree*.

La structure de donnée implémente le patron de conception « composite » [89]. Chaque classe représentant un nœud de l'arbre hérite de la classe abstraite `ArchitectureNode`. Les constructeurs des différents nœuds prennent deux paramètres, le nom de l'élément représenté et son type. Les feuilles de l'arbre sont les objets de type `EventNode`, `VariableNode`, `PortNode` et `PropertyNode`. Ces éléments ne peuvent contenir de sous-éléments. Les composants pouvant contenir d'autres éléments sont d'un type héritant de `CollectionNode`. Cette classe contient une liste d'objets de type `ArchitectureNode` qui représente les nœuds fils dans l'arborescence. Le type passé en paramètre est utilisé pour peupler cette liste. Même s'ils ne contiennent pas réellement des sous-éléments de la hiérarchie, les nœuds `Clock` et `Signal` héritent de `CollectionNode` car ils doivent exposer les propriétés et les événements qu'ils possèdent.

Dans une hiérarchie d'un modèle ESys.Net, on peut distinguer deux catégories d'éléments. D'une part on a les éléments de structures tels que les modules et le `SystemModel`. D'autre part, on a les éléments de données comme les signaux, les horloges, les variables non-spécialisées et les propriétés. Ces éléments ont pour but de stocker ou de transmettre des données, contrairement aux éléments de structure qui servent à décomposer le modèle en sous-éléments. Cette distinction entre éléments de structure et éléments de données se retrouve avec l'interface `ITypedNode` du DST. Tous les éléments de données implémentent cette interface qui déclare une propriété `DataType`. Cette propriété retourne le type des données manipulées par l'élément en question. Par exemple, dans le DST de la Figure 18, la propriété `DataType` du nœud représentant le signal `data` retourne `Int32` car ce signal transmet des nombres entiers signés sur 32 bits. Ce type est déterminé par le type de la propriété `Value` du signal.

Chaque nœud de l'arbre conserve un certain nombre d'informations concernant l'élément qu'il représente. Il stocke notamment le nom de l'élément et son chemin complet dans l'arbre. Dans le cas des éléments de données, le type des données manipulées est également stocké. Enfin, les nœuds représentant des ports d'entrée/sortie enregistrent la direction du port.

La construction du DST passe par l'exploration récursive des types de données constituant le modèle. Le cœur de l'algorithme se trouve dans la méthode `BuildTree` de la

classe `CollectionNode`. Elle est appelée dans le constructeur des différentes classes qui étendent `CollectionNode`. Le contenu de cette méthode est donné dans le Code Source 15.

```
1.  private void BuildTree(Type t, FieldType filter){
2.    foreach(FieldInfo f in type.GetFields(BindingFlags.Instance |
3.                    BindingFlags.Static | BindingFlags.NonPublic |
4.                    BindingFlags.Public)){
5.      if(IsInIgnoreList(f.Name)) continue;
6.      if(typeof(Clock).IsAssignableFrom(f.FieldType) &&
7.        (filter & FieldType.CLOCK) == FieldType.CLOCK) {
8.          new ClockNode(f.Name, f.FieldType, this, Tree);
9.          continue;}
10.     if(typeof(BaseSignal).IsAssignableFrom(f.FieldType) &&
11.       (filter & FieldType.SIGNAL) == FieldType.SIGNAL) {
12.         new SignalNode(f.Name, f.FieldType, this, Tree);
13.         continue;}
14.     if(typeof(BaseModule).IsAssignableFrom(f.FieldType) &&
15.       (filter & FieldType.MODULE) == FieldType.MODULE) {
16.         new ModuleNode(f.Name, f.FieldType, this, Tree);
17.         continue;}
18.     if(typeof(Event).IsAssignableFrom(f.FieldType) &&
19.       (filter & FieldType.EVENT) == FieldType.EVENT) {
20.         new EventNode(f.Name, this, Tree);
21.         continue;}
22.     if((filter & FieldType.VARIABLE) == FieldType.VARIABLE) {
23.       new VariableNode(f.Name, f.FieldType, this, Tree);
24.       continue;}}
25.
26.   if((filter & FieldType.PROPERTY) == FieldType.PROPERTY)
27.   {
28.     foreach(PropertyInfo p in type.GetProperties(
29.             BindingFlags.Instance | BindingFlags.Static |
30.             BindingFlags.NonPublic | BindingFlags.Public)){
31.       if(!Tree.IsInIgnoreList(p.Name))
32.         new PropertyNode(p.Name,p.PropertyType, this, Tree);}}}
```

Code Source 15 : Construction du Design Structure Tree.

La méthode se compose de deux boucles. La première, lignes 2 à 24, parcourt les différentes variables d'instance et construit les nœuds suivant le type d'élément rencontré. Toutes les variables d'instances sont explorées, qu'elles soient publiques, privées, statiques ou non statiques (lignes 2 à 4). La seconde boucle, lignes 28 à 32, parcourt les propriétés et construit un objet `PropertyNode` pour chacune d'elle.

Pour identifier le type d'un élément, la méthode `type1.IsAssignableFrom(type2)` est utilisée. Cette méthode détermine si un objet de type `type2` peut être assigné à une instance de `type1`. Par exemple, à la ligne 10, si le champ courant peut être affecté à une variable de type `BaseSignal`, cela signifie que le

champ est d'un type héritant de `BaseSignal` et qu'il doit donc être considéré comme un signal.

Afin de limiter la taille de l'arbre et d'enregistrer uniquement les éléments pertinents, deux systèmes de filtres sont utilisés. Le premier permet de masquer les variables internes d'ESys.Net. Par exemple, un module contient une table des processus lui appartenant. Cette structure de données ne doit pas être accessible à l'utilisateur et ne doit donc pas figurer dans le DST. Ce filtrage se fait sur le nom des variables. Les noms des variables devant être ignorées sont stockés dans une liste. Avant d'ajouter un élément au DST, on vérifie que son nom ne figure pas dans cette liste (ligne 5 du Code Source 15). Cette méthode a l'avantage d'être simple à implémenter mais interdit l'utilisation de certains noms de variables dans les modèles. Une autre approche serait d'annoter les variables internes d'ESys.Net avec un attribut particulier. Cette méthode est plus élégante mais nécessite de modifier lourdement ESys.Net et impose une contrainte sur ses développements futurs. Le deuxième filtrage mis en place permet de limiter l'exploration de certaines parties du modèle. Ce filtrage se fait avec le paramètre `filter` de la méthode `BuildTree` qui permet d'ignorer certains types d'éléments. Par exemple, pour les signaux d'horloge, on ne veut considérer que les événements et les propriétés contenus dans le signal. La méthode `BuildTree` est donc appelée avec le filtre `FieldType.EVENT|FieldType.PROPERTY`.

Afin de faciliter la manipulation de l'arbre, ce dernier est encapsulé dans un objet de type `DesignStructureTree`. Cet objet offre des opérations suivantes :

- `ArchitectureNode[] GetChildrenList(ModelPath path)` : à partir du chemin d'un élément dans le modèle, retourne la liste des sous-composants de l'élément en question.
- `bool ElementExists(ModelPath path, FieldType type)` : vérifie qu'un élément d'un type donné existe dans le modèle.
- `ArchitectureNode GetElement(ModelPath path)` : retourne le nœud correspondant à un chemin donné.

Le DST est utilisé dans l'outil de vérification pour offrir à l'utilisateur une représentation graphique de l'arborescence du modèle et faciliter la spécification de propriétés (voir partie 5.1). Il est également utilisé lors de l'analyse des propriétés LTL pour vérifier la syntaxe des références vers le modèle, par exemple que le signal `sig(data)` existe vraiment. Enfin, le DST est utilisé pour obtenir le type d'un objet du modèle et vérifier la compatibilité des types dans les comparaisons.

L'avantage du DST est qu'il évite les opérations d'introspection redondantes. L'introspection est un mécanisme coûteux et son impact sur les performances globales de l'application est conséquent. On peut remarquer que le DST ne contient aucune information sur l'interconnexion entre les modules. Cette information est contenue dans le corps des constructeurs des modules ou dans le corps des méthodes `BindingPhase`, mais elle n'est pas nécessaire pour le processus de vérification. Quoiqu'il en soit, cette fonctionnalité pourra être ajoutée par la suite, de deux façons. Premièrement en analysant le code CIL pour détecter les affectations. Deuxièmement en exécutant la phase d'élaboration du modèle et en utilisant les structures données internes de ESys.Net qui contiennent les informations sur l'interconnexion. La première méthode reviendrait à faire un interpréteur de CIL très simplifiée, alors que la deuxième utilise l'interpréteur de l'environnement d'exécution.

4.3.2 *Model Resolver*

Les informations obtenues par le DST sont des informations statiques sur la structure du modèle. L'objectif de `ModelResolver` est d'obtenir des informations dynamiques sur l'état du modèle pendant la simulation. On veut obtenir la valeur d'un objet du modèle à partir du chemin de l'objet et d'une instance du modèle. Ceci permet par exemple de lire la valeur d'un signal à partir d'une chaîne de caractères donnant son chemin. Ce mécanisme est au cœur de l'évaluation de l'état du modèle pendant la vérification.

Le chemin d'un objet dans le modèle est stocké dans un objet de type `ModelPath`. Le chemin est représenté sous la forme d'une liste de chaînes de caractères. Par exemple, `["producer", "clk", "negedge"]` représente l'événement `negedge` de l'horloge `clk` du module `producer`. La classe `ModelPath` offre deux opérations. La propriété `Path.Root`

retourne la racine du chemin (`producer` dans l'exemple précédent) et `Path.GetDown()` retourne un nouveau chemin en « descendant » dans la hiérarchie (`["clk", "negedge"]` avec l'exemple précédent).

Le Code Source 16 présente une version simplifiée de la méthode `Resolve`. La récursion est initialisée en appelant cette méthode avec l'instance du modèle comme racine et le chemin complet de l'objet recherché.

```
1.  public static object Resolve(object root, Path path)
2.  {
3.    Type rootType = root.GetType();
4.    FieldInfo fInfo = rootType.GetField(path.Root,
5.        BindingFlags.Instance | BindingFlags.Static |
6.        BindingFlags.NonPublic | BindingFlags.Public);
7.    if(fInfo != null){ //field found
8.      if(path.Length == 1)
9.        return fInfo.GetValue(root); // end of recursion
10.     else{
11.       return Resolve(((FieldInfo)fInfo).GetValue(root),
12.                      path.GetDown());}
13.   }
14.   return null;}
```

Code Source 16 : Le *Model Resolver*.

L'objet `fInfo` décrivant le champ recherché est obtenu lignes 3 et 4. Le champ recherché est la racine du chemin. Si ce champ n'est pas trouvé, la valeur `null` est retournée. Quand la fin du chemin est atteinte (ligne 8), on retourne la valeur du champ (ligne 9). S'il reste des éléments dans le chemin, on appelle à nouveau la méthode. La valeur du champ courant devient la nouvelle racine (ligne 11) et on supprime l'élément courant du chemin (ligne 12).

4.3.3 Discussion

La simplicité de ces algorithmes montre la puissance du mécanisme d'introspection de .NET et son utilité pour l'observation de modèles ESys.Net. Sans ces capacités d'introspection, la réalisation d'environnements de vérification et de simulation serait plus complexe.

4.4. Spécification des propriétés

Dans cette partie, nous allons présenter le langage de spécification de propriétés utilisé par notre environnement de vérification. Ce langage se base sur la logique

temporelle linéaire pour formaliser les comportements du modèle qui doivent être vérifiés pendant la simulation. Notre outil prend en charge deux types de logique temporelle : la logique temporelle linéaire classique (LTL) et la logique temporelle linéaire temps-réel (RTLTL). Chaque propriété constituera un observateur dans l'environnement de vérification. Les propriétés relatives à un modèle sont regroupées dans un fichier. Chaque observateur est synchronisé avec le modèle via un événement qui définit l'instant d'échantillonnage des variables et des signaux. Il définit également les pas d'évaluation des propriétés. Cet événement peut être n'importe quel événement du modèle, par exemple le front montant d'un signal d'horloge.

4.4.1 Propositions atomiques

Les propositions atomiques constituent la base du langage de spécification. Ce sont des propriétés décrivant l'état du modèle observé. L'observabilité du modèle est limitée aux variables et objets suivants:

- Les signaux ;
- Les horloges ;
- Les canaux ;
- Les ports ;
- Les variables d'instance des modules et de canaux. I.e. les variables d'instances qui ne sont pas des éléments de la structure du modèle. Ces variables seront par la suite appelées variables non-spécialisées, ou plus simplement variables ;
- Les propriétés des modules et des canaux ;
- Les événements des modules ;
- Les événements des canaux ;
- Les propriétés et les événements des signaux ;
- Les propriétés et les événements des horloges.

Les propositions atomiques sont constituées d'une comparaison de deux objets du modèle ou bien d'un objet et d'un littéral. Les opérateurs de comparaison classiques sont pris en charge : égal (==), différent (!=), inférieur (<), supérieur (>), inférieur ou égal (<=) et supérieur ou égal (>=). Les objets du modèle sont référencés par le chemin absolu de l'objet annoté d'un qualificateur. La syntaxe est `chemin.qualificateur(nom_champ)`. Le

qualificateur est un des mots-clefs donné dans le Tableau VII. Ce qualificateur facilite la lecture de la propriété.

Objet	Qualificateur
Signal	`sig`
Port	`port`
Horloge	`clk`
Canal	`ch`
Variable	`var`
Propriété	`prop`
Événement	`event`

Tableau VII : Les qualificateurs.

Par exemple, la propriété `consumer.port(dataIn)` référence le port `dataIn` du module `consumer`. La proposition atomique `consumer.port(dataIn) <= 5` est vraie quand la valeur sur le port d'entrée `dataIn` est inférieure ou égale à cinq.

Toutes les propriétés utilisent la même syntaxe, à deux exceptions près : les canaux et les événements. Les canaux sont des entités de transmission de données de haut niveau. Contrairement aux signaux, la notion de « valeur courante » n'a pas de sens pour un canal. Il faut donc indiquer à l'environnement de vérification quelle valeur du canal il doit lire. La syntaxe utilisée pour les canaux est donc : `chemin.ch(mon_canal, un_champ)`, où `mon_canal` est le nom du canal et `un_champ` est le nom du champ (variable d'instance ou propriété) contenant la valeur à observer. L'évaluation de cette expression retourne la valeur de `mon_champ`. Les événements ne sont pas utilisables dans des propositions atomiques. Ils peuvent uniquement servir à spécifier l'événement de synchronisation d'un observateur.

On peut remarquer que certaines expressions sont équivalentes. Par exemple, `sig(data)` et `data.prop(Value)` représentent la même chose : la valeur du signal data. De même, `chemin.ch(mon_canal, un_champ)` est équivalent à `chemin.mon_canal.var(un_champ)`.

Les constantes littérales sont de quatre types : booléen (`true`, `false`), entier, réels et chaînes de caractères. Cependant, les autres types de données sont pris en charge grâce à

l'utilisation de chaînes de caractères. Par exemple, si un signal transmet des adresses IP, on pourra écrire `sig(adresse) == "127.0.0.1"`.

Pour faciliter la définition de propriétés sur des modèles dont la hiérarchie est profonde, le langage permet la définition d'alias sur des chemins dans l'arbre. L'alias est un nom qui est associé à un chemin. Dans les propositions atomiques, on peut remplacer le chemin complet par le nom de l'alias. Par exemple, l'alias local_clk est défini par : `#local_clk = producer.clk`. On peut ensuite écrire `#local_clk.Prop(value)==true` qui est équivalent à `producer.clk.Prop(value)==true`.

4.4.2 Syntaxe

Les propositions atomiques sont connectées par des opérateurs. Le Tableau VIII donne la liste des opérateurs.

Type	Nom	Symbole
Booléen	non	!
	et	&&
	ou	||
	implique	->
	équivalent à	<->
Temporels	toujours	[]
	fatalement	<>
	prochain	X
	jusqu'à	U
Temporels RTLTL	toujours	[][a,b]
	fatalement	<>[a,b]
	prochain	X[a]
	jusqu'à	U[a,b]
	jusqu'à faible	W[a,b]

Tableau VIII : Opérateurs du langage de spécification.

La syntaxe des formules correspond à la syntaxe de LTL et de RTLTL donnée dans la Figure 3 et dans la Figure 9, avec AP l'ensemble de propositions atomiques définies comme indiqué dans la partie 4.4.1. Les parenthèses sont utilisées pour lever les ambiguïtés sur la précédence des opérateurs. Par exemple, la formule `[](sig(acknowledge)==true -> X sig(acknowledge)==false)` spécifie que le signal `acknowledge` ne peut pas être à l'état haut plus d'un cycle.

Chaque formule LTL est englobée dans le corps d'un observateur. Un observateur est déclaré avec les mots-clefs `ltl_observer` ou `rtltl_observer`. À chaque observateur

on associe un nom unique qui est utilisé pour identifier la propriété dans l'outil de vérification. Le corps de l'observateur ne contient qu'une seule formule. La syntaxe pour déclarer un observateur est :

```
1. ltl_observer nom_de_l_observateur(evenement_de_synchronisation){
2.    une formule ltl
3. }
```

Où evenement_de_synchronisation est le chemin vers l'événement synchronisant l'observateur et le modèle ; par exemple `clk.event(posedge)`.

Le langage de définition offre également la possibilité d'ajouter des commentaires à l'extérieur du corps des observateurs. Deux types de commentaire sont autorisés : les commentaires multi-lignes, qui commencent par `/*` et se terminent par `*/`, et les commentaires sur une seule ligne qui commencent par `//`. Le commentaire qui précède la définition d'un observateur est associé à l'observateur en question. Il explique le rôle de l'observateur.

4.4.3 Sémantique

La sémantique des observateurs est identique aux sémantiques de LTL et de RTLTL données dans la Figure 7 et la Figure 10. Ces sémantiques sont définies sur des traces d'exécution finies, sans étiquettes de temps. Chaque pas dans la trace est défini par l'occurrence de l'événement de synchronisation. La sémantique de l'opérateur **X** est donc « à la prochaine occurrence de l'événement de synchronisation, la propriété qui suit doit être vraie ». Cette sémantique est claire si l'événement de synchronisation est périodique, par exemple si c'est un événement sur une horloge. Par contre, si l'occurrence de l'événement est irrégulière, la sémantique devient moins naturelle. Le même constat s'applique pour les opérateurs RTLTL dont la sémantique des bornes de temps dépend de l'événement de synchronisation. Ces bornes de temps sont en fait des contraintes sur le nombre d'occurrences de l'événement de synchronisation : [3, 5] signifie « entre la troisième et la cinquième occurrence de l'événement ». La sémantique peut être moins explicite si cet événement est irrégulier.

4.4.4 Format du fichier

Les observateurs concernant un modèle sont regroupés dans un fichier texte. Ce fichier comporte 3 sections.

init. Cette section contient le nom du type définissant le modèle, avec son espace de nommage complet. Elle contient également le chemin relatif vers l'« assembly » qui contient le modèle. Pour le modèle du producteur/consommateur la définition est :

```
1.  init{
2.    model ProducerConsumer.ProducerConsumerModel@"ProducerConsumer.dll"
3.  }
```

Cette section est prévue pour être étendue suivant les fonctionnalités futures de l'outil, par exemple pour contenir les références vers d'autres « assemblies ».

aliases. Dans cette section sont définis les alias. Les alias sont communs à l'ensemble des observateurs et doivent possédent un identifiant unique. Par exemple, les alias `#p_clk` et `#c_clk` sont définis par :

```
1.  aliases{
2.    #p_clk=producer.clk
3.    #c_clk=consumer.clk
4.  }
```

un ensemble d'observateurs. Cette section comporte une suite de définitions d'observateurs.

Le Code Source 17 donne un exemple de fichier de propriétés sur le modèle producteur/consommateur. Deux alias `#p_clk` et `#c_clk` sont déclarés dans la section `aliases` (lignes 3 à 5). Deux observateurs sont définis : `dummy_observer` (lignes 6 et 7) et `dataReadyNotRepeat` lignes 9 à 12.

```
1.  init {
2.    model ProdCons.ProducerConsumerModel@"ProducerConsumer.dll"}
3.  aliases {
4.    #p_clk=producer.clk
5.    #c_clk=consumer.clk}
6.  ltl_observer dummy_observer(#p_clk.event(sensitive)){
7.      [](#p_clk.prop(Value) == #c_clk.prop(Value))}
8.
9.  ltl_observer dataReadyNotRepeat(clk.event(posedge)){
10.     [] (sig(dataReady) == true -> X(
11.             (sig(dataReady) != true U sig(acknowledge) == true)
12.             || [] sig(dataReady) != true))}
```

Code Source 17 : Exemple de fichier de propriétés.

Le premier observateur est trivialement vrai, il compare la valeur de deux objets qui sont en fait identiques. Le second vérifie que le signal `dataReady` ne se répète pas avant que l'acquittement ait été reçu (ligne 11). Il peut également rester faux jusqu'à la fin de la simulation (ligne 12) si l'acquittement n'arrive pas. Une propriété identique peut être écrite pour vérifier que l'acquittement n'est pas envoyé deux fois.

4.4.5 Discussion

Le langage de définition de propriétés présenté dans cette partie constitue la base de notre environnement de vérification. Par rapport à des langages comme PSL ou SystemVerilog Assertions, notre langage reste cependant limité. Les observateurs sont en quelque sorte équivalents aux assertions de PSL et aux assertions concurrentes de SystemVerilog. Mais l'absence d'équivalent aux expressions rationnelles séquentielles de PSL ou de SVA réduit la facilité d'utilisation du langage et limite l'expressivité. Ces expressions permettent de définir plus simplement certaines propriétés, notamment celles sur des états successifs d'une trace. Par exemple, la propriété « a est vrai au maximum trois cycle consécutifs » se traduit par la SERE (*Sequential Extended Regular Expression*) {!a;a[3];!a}. Cette propriété peut être formalisée en LTL par `(!a && X a) -> (X a && XX a && XXX a && XXXX !a)`. RTLTL permet de simplifier l'écriture de cette formule avec `(!a && a) -> X(a U[3,3] !a)`, mais l'expression reste plus difficile à manipuler que son équivalent SERE.

La liaison entre les observateurs et les éléments du modèle est également moins souple dans notre langage. En effet, il n'offre pas d'équivalent aux notions de `v_unit` de PSL ou de `property` de SVA qui permettent de lier une assertion à plusieurs éléments du modèle. Dans le langage développé ici, la liaison entre les observateurs et le modèle est plus direct puisque les propriétés contiennent directement un chemin absolu vers les éléments du modèle. Si on écrit une propriété vérifiant le fonctionnement d'un module donné, il va falloir répéter la propriété pour chaque instance du module.

Malgré ces limitation, notre langage de spécification fourni une base pour l'implémentation d'un langage plus complet comme PSL. La prise en charge complète de PSL est une tâche complexe qui était difficilement faisable dans le cadre de ce projet.

Enfin, les limitations du langage et de l'observabilité du modèle peuvent être pénalisantes pour l'expression de certaines propriétés. Prenons l'exemple d'un canal de communication implémentant une file. Ce canal contient un tableau représentant les éléments dans la file. Notre langage ne permet pas d'exprimer des propriétés sur les éléments du tableau ou sur le nombre d'éléments du tableau. Une façon simple de pallier ce manque est de définir une propriété dans le code source du modèle qui va effectuer le calcul. Par exemple, on peut ajouter le code suivant dans la définition d'une file :

```
1.  public bool ValidData{
2.    get{
3.      foreach(Packet p in dataArray)
4.        if(!p.CRC.IsValid) return false;}}
```

Ce code va vérifier que le code d'erreur de chacun des paquets se trouvant dans la file est valide. Et on peut ensuite écrire la propriété LTL suivante :

```
1.  [](fifo.prop(IsActiv) == true ->
2.         (fifo.prop(ValidData) == true && fifo.prop(NbElement) <= 100))
```

Cette propriété vérifie que quand la file est active, les données ont un CRC valide et que le nombre d'élément dans la file est inférieur à 100.

4.5. Construction des observateurs

Au cours de cette phase du processus de vérification, le fichier contenant l'ensemble des propriétés est chargé. La syntaxe des propriétés est vérifiée et elles sont transformées en observateurs qui seront par la suite liés avec le modèle simulé. La Figure 20 représente le flux de données pendant la construction des observateurs. Le processus est divisé en deux phase : l'analyse du fichier de propriétés et la transformation en automates. Cette dernière phase est différente suivant le type d'observateur que l'on considère. Si c'est un observateur LTL, il est transformé en automate par l'algorithme LTL2BA, présenté dans la partie 3.3.3. L'implémentation de LTL2BA est un outil générique préexistant, qui ne peut pas manipuler des formules LTL avec des propositions atomiques complexes. La formule est donc réécrite en remplaçant les propositions atomiques par de étiquette (p1, p2, p3...). C'est ce qu'on entend par « formule LTL simplifiée » dans la Figure 20. Dans le cas d'un observateur RTLTL, c'est notre implémentation de l'algorithme GPVW95 modifié qui est utilisé (voir parties 3.3.3 et 3.5). Cet algorithme manipule directement l'arbre de syntaxe abstraite (AST) représentant la formule. Quelque soit le type d'observateur, les propositions

atomiques sont extraites au cours de la phase d'analyse et sont intégrées aux automates pendant leurs constructions. Au début de la phase d'analyse, l'« assembly » contenant le modèle en cours de vérification est chargé et un « Design Structure Tree » est construit pour faciliter la construction des automates.

Dans une première partie nous présentons l'analyse lexicale et syntaxique du fichier de propriétés. Puis dans la partie 4.5.2, la construction des propositions atomiques est détaillée. Enfin, la partie 4.5.3 présente les deux méthodes de transformation en automates.

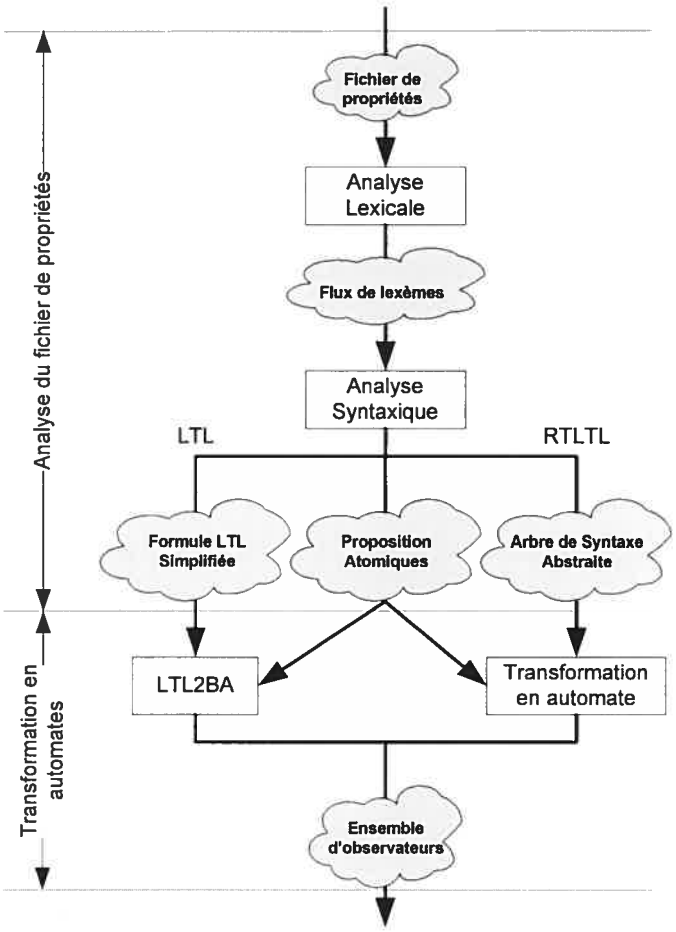


Figure 20 : Flux de données pour la construction des observateurs.

4.5.1 Analyse du fichier de propriétés

L'objectif de l'analyse du fichier de propriétés est d'extraire les formules LTL et RTLTL ainsi que les propositions atomiques pour préparer la construction des automates. Les formules LTL sont réécrites en remplaçant les propositions atomiques par des

étiquettes *p1*, *p2*, *p3*, ... Ainsi, la formule `[](sig(acknowledge)==true -> X !sig(acknowledge)==true)` est réécrite en `[](p1 -> X !p1)`. Les formules RTLTL sont quant à elles transformées en arbres de syntaxe abstraite qui sont analysés pendant la transformation en automate. Quelque soit la logique temporelle utilisée, les propositions atomiques sont extraites au cours de la phase d'analyse pour construire des fonctions d'évaluation. Les propositions atomiques sont au cœur du processus d'évaluation de l'état du modèle et sont détaillées dans la partie 4.5.2.

L'analyse du fichier est décomposée en deux opérations. La première, l'analyse lexicale, découpe le fichier de propriétés en un ensemble de lexèmes (*tokens* en anglais). Puis l'analyse syntaxique décompose le flux de lexème suivant des règles pour extraire la structure du fichier. Ces deux phases sont similaires au *front-end* d'un compilateur classique. Le logiciel ANTLR [92, 93] est utilisé pour concevoir les analyseurs lexical et syntaxique. Cet outil prend en entrée deux fichiers, donnés dans l'Annexe B, définissant les lexèmes et la grammaire du format du fichier. À partir de ces fichier, ANTLR génère deux programmes en C# correspondant à l'analyseur lexicale à l'analyseur syntaxique. ANTLR permet de placer des blocs d'instructions dans la définition de la grammaire qui sont exécutées à certaines étapes de l'analyse du fichier, par exemple quand une certaine structure a été reconnue.

Le but de l'analyse lexicale est de transformer une suite de caractères en une suite de lexèmes. Les lexèmes sont des unités lexicales qui correspondent principalement aux mots-clefs, aux opérateurs, aux identifiant et aux littéraux. Le Tableau IX présente la liste des lexèmes des fichiers de spécification de propriétés.

Lexèmes	Définition
Mots-clefs	`init aliases ltl_observer ...`
Opérateurs LTL	`U X <> [] ...`
Opérateurs booléens	`! && || -> <->`
Op. de comparaison	`== != < <= > >=`
Littéraux	`true false` entiers, réels et chaîne de caractères
Qualificateurs	`sig( var( port( clk( ...`
Identifiants	Expressions de la forme `([a-Z]|_)([a-Z]|_|[0-9])*`
Symboles	`[ ] { } ( ) . , # = @`

Tableau IX : Lexèmes des fichiers de propriétés.

Outre les mots-clefs et les opérateurs, l'analyseur lexical reconnaît les identifiants. Ils correspondent au nom d'un élément du modèle ou au nom d'un observateur. Ils sont définis par une expression rationnelle. Un identifiant est une suite de caractères alphanumériques qui peut contenir le caractère *souligné* « _ » et qui ne peut commencer par un chiffre. La définition des qualificateurs inclut la parenthèse ouvrante pour lever l'ambiguïté entre un qualificateur et un identifiant.

L'analyse syntaxique est effectuée sur le flux de lexèmes issu de l'analyse lexicale. Elle vérifie que la suite de lexèmes est correcte par rapport à la grammaire du langage. La grammaire est décrite dans un fichier qui contient également des actions à effectuer quand certaines structures sont reconnues. Ces actions sont représentées par des blocs d'instructions en C# qui sont copiés dans le code de l'analyseur syntaxique généré par ANTLR. Le Code Source 18 est un extrait de la grammaire des fichiers de propriétés. La grammaire complète est donnée dans l'Annexe B. Il correspond à la définition du bloc `init`.

```
1.  path returns [Path p = new Path();]:
2.    m1:ID{p.Append(m1.getText());}(DOT m2:ID {p.Append(m2.getText());})*
3.    ;
4.
5.  init:
6.    INIT LCURLY MODEL p=path AT s:STRING_LITERAL RCURLY
7.    {
8.      Assembly assembly = Assembly.LoadFrom(s);
9.      Type modelType = assembly.GetType(p.ToString());
10.     dst = new DesignStructureTree(modelType);
11.   }
12.   ;
```

Code Source 18 : Fragment de la grammaire des fichiers de propriétés.

La règle qui définit un chemin est donnée aux lignes 1 à 3. Un chemin est une suite d'identifiants séparés par des points. La règle retourne un objet `p` de type `Path` qui est initialisé ligne 1 avec un chemin vide. La règle proprement dite est donnée ligne 2. `ID` et `DOT`, respectivement identifiant et point, représentent les lexèmes identifiés par l'analyse lexicale. Un chemin est constitué par au minimum un identifiant, référencé par le nom `m1`. Les instructions entre accolades définissent les actions. Quand le premier identifiant m1 est reconnu par l'analyseur, l'action qui suit est exécutée et l'identifiant est ajouté à la structure de donnée `Path`. La fin de la règle entre parenthèse est suivie du symbole * indiquant que

cette partie peut être répétée un nombre fini de fois. Elle reconnaît la suite d'identifiants et les ajoute au fur et à mesure à la fin du chemin. Les lignes 5 à 12 définissent la syntaxe du bloc `init` et l'action à effectuer à la fin de la reconnaissance. La définition de la syntaxe est donnée ligne 6. Elle est formée de lexèmes et d'une référence à la règle `path`. L'objet retourné par cette règle est référencé par p. Le lexème `STRING_LITERAL` représente une chaîne de caractères qui contient le nom de l'« assembly » du modèle. A la fin de la reconnaissance du bloc `init`, l'« assembly » est chargée et un DST est construit (lignes 8 à 10). Les instructions de gestion des erreurs ne sont pas montrées dans cet extrait.

La grammaire des expressions LTL ou RTLTL est donnée dans le Code Source 19. Les actions et les valeurs de retours ont été omises pour plus de clarté.

```
1.  expr: mexpr (binaryOp mexpr )*;
2.  mexpr: atom | TRUE| FALSE
3.       | unaryOp mexpr
4.       | LPAREN expr RPAREN;
5.  unaryOp: OP_NOT | OP_NEX | OP_ALW | OP_EVE [...];
6.  binaryOp: OP_OR | OP_AND | OP_EQU | OP_IMP | OP_UNT [...];
```

Code Source 19 : Grammaire simplifiée de LTL et RTLTL.

La structure particulière de cette grammaire est due au fait que ANTLR génère des analyseurs LL(k), qui n'acceptent pas de grammaire avec une récursivité gauche. Dans la grammaire, LPAREN et RPAREN sont les lexèmes des parenthèses ouvrantes et fermantes et les lexèmes OP_ représentent les opérateurs temporels et booléens. Les grammaires de LTL et de RTLTL sont identiques, à la définition des opérateurs temporels près. Cependant, les actions effectuées pendant la reconnaissance des formules ne sont pas les même suivant la logique temporelle utilisée. Dans le cas de LTL, la formule est réécrite dans une chaîne de caractères, en remplaçant les propositions atomiques par des étiquettes. Pour RTLTL, un arbre de syntaxe abstraite est construit. Cet arbre est utilisé par la suite lors de la transformation de la formule en automate. La Figure 21 représente le diagramme de classes partiel de l'arbre de syntaxe abstraite.

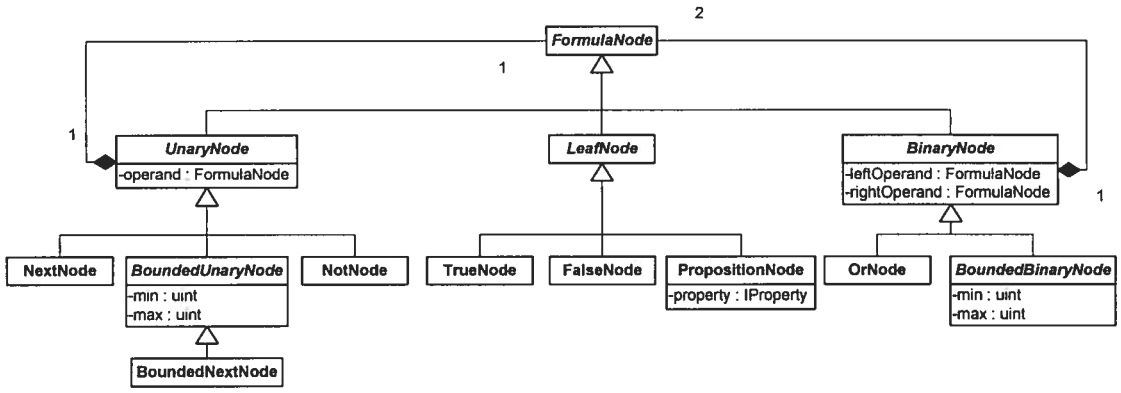


Figure 21 : Diagramme de classes partiel pour l'AST de RTLTL.

Chaque nœud de l'arbre est d'un type qui hérite de `FormulaNode`. Ce dernier est spécialisé par trois classes, suivant l'arité de l'opérateur : `LeafNode`, `UnaryNode`, `BinaryNode`. Les nœuds `PropositionNode` représentent les propositions atomiques. Ils contiennent un objet IProperty dont la construction est détaillée dans la partie suivante. Les classes `BinaryNode` et `UnaryNode` sont spécialisées pour représenter les opérateurs temporels avec une borne de temps. Bien que l'on puisse considérer les opérateurs temporels classiques comme des cas particuliers des opérateurs avec des bornes de temps, cette unification n'est pas faite ici. En effet, l'algorithme de transformation gère différemment les opérateurs LTL et les opérateurs RTLTL.

4.5.2 Construction des propositions atomiques

Pendant l'analyse du fichier, les propositions atomiques sont analysées pour vérifier leur conformité avec le modèle. La construction des propositions atomiques se fait en deux temps. On commence par créer les deux opérandes de la comparaison, en vérifiant que les objets référencés existent réellement dans le modèle. On va pour cela utiliser le DST présenté dans la partie 4.3.1. Dans un deuxième temps, les deux opérandes sont assemblés dans la comparaison en s'assurant que les types de deux opérandes soient compatibles. Le diagramme de classes de la structure de données est donné Figure 22.

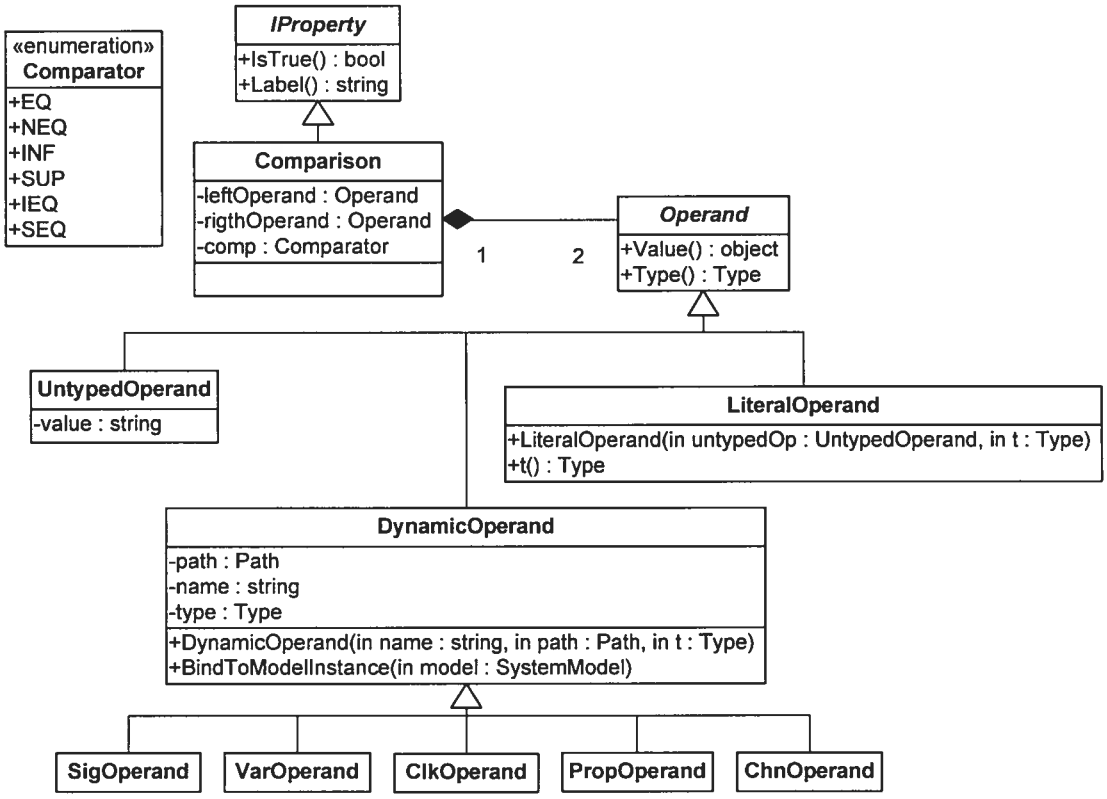


Figure 22 : Diagramme de classes des propositions atomiques.

L'interface `IProperty` est une abstraction d'une proposition atomique. Elle définie deux méthodes. `IsTrue` est la méthode d'évaluation de la proposition : elle retourne vrai si la proposition est valide à un instant donné. La méthode `Label` retourne l'étiquette de la proposition. Cette étiquette correspond à une chaîne de caractères permettant d'identifier une proposition atomique. Elle est utilisée dans la réécriture des formules LTL et pour la visualisation des observateurs. À l'heure actuelle, les seules propositions atomiques prisent en charge par l'environnement sont les comparaisons entres des objets du modèle. Cette comparaison est représentée par la classe `Comparison` qui étend la classe `IProperty`. La classe `Comparison` est composée de deux opérandes et d'un opérateur de comparaison. On distingue trois types d'opérandes :

Les opérandes littéraux. Ce sont les opérandes ayant une valeur littérale donnée dans le fichier de spécification sous la forme d'une constante. Il contient un objet `value` qui représente la valeur constante.

Les opérandes dynamiques. Les opérandes dynamiques sont des références vers des objets du modèle dont la valeur peut changer au cours de la simulation. Ils contiennent trois attributs : un chemin, possiblement vide, un nom et un type. Par exemple, pour l'opérande `consumer.port(inData)`, le chemin est consumer et le nom est `inData`. Le type stocké correspond au type de la donnée retournée par l'opérande. Dans l'exemple précédent, ce type est `int32`. Il est obtenu en interrogeant le Design Structure Tree du modèle. Ces informations sont utilisées pendant la simulation pour évaluer l'opérande.

Les opérandes non-typés. Ce sont des opérandes intermédiaires qui sont utilisés pendant la création des opérateurs littéraux. Les opérandes littéraux sont d'abord stockés dans un opérande non-typé car le type exact de l'opérande est déterminé en fonction du type de l'autre opérande. Cette opération est effectuée au moment de la construction de la comparaison.

La construction de la comparaison elle-même se fait quand les deux opérandes sont construits. On distingue deux cas. Si les deux opérandes sont dynamiques, une simple vérification de type est effectuée en s'assurant que le type des deux opérandes est identique. Si l'opérande de droite est une constante, il faut créer un objet dont le type est compatible avec l'opérande de droite. Par exemple, si on a la propriété atomique `producer.port(inData) == 12`, il faut construire un objet `int32` à partir de la chaîne de caractères « `12` ». À cet instant de la construction, l'opérande de droite est un UntypedOperand. Il doit muter en un LiteralOperand type avec le type correct. L'objectif est donc de construire un objet d'un type arbitraire à partir d'une chaîne de caractères représentant sa valeur. Cette opération est effectuée dans le constructeur de LiteralOperand dont le code est donné dans le Code Source 20.

```
1.  public LiteralOperand(UntypedOperand untypedOp, Type t)
2.  {
3.    string lit = (string) untypedOp.Value;
4.    if(t == typeof(string))
5.      value = lit;
6.    else{
7.      object [] parameter = new object[1];
8.      parameter[0] = lit;
9.      if(t.IsEnum)
10.       value = System.Enum.Parse(t, lit);
```

```
11.     else{
12.       ConstructorInfo constructor =
13.                         t.GetConstructor(new Type[] {typeof(string)});
14.       if(constructor != null)
15.         value = constructor.Invoke(parameter);
16.       else
17.       {
18.         MethodInfo parseMethod =
19.                     t.GetMethod("Parse", new Type[] {typeof(string)});
20.         if(parseMethod != null && parseMethod.IsStatic &&
21.            parseMethod.ReturnType == t)
22.           value =
23.            parseMethod.Invoke(t.ReflectedType, new string[]{lit});}}}}
```

Code Source 20 : Constructeur de *LiteralOperand*.

Le constructeur prend deux paramètres : l'objet UntypedOperand devant être transformé et le type de la constante devant être créée. Il s'agit du type de l'opérande de droite dans la comparaison. Si c'est une chaîne de caractères qui doit être créée, aucune opération n'est nécessaire et la valeur est obtenue directement depuis l'opérande non-typé (lignes 4 et 5). Si le type de l'objet devant être créé est un type énuméré (ligne 9), on fait appel à la méthode `System.Enum.Parse` définie par .NET (ligne 10). Cette méthode construit un objet de type énuméré à partir d'une chaîne de caractères représentant sa valeur. Si le type n'est pas un type énuméré, alors c'est une classe. Les types primitifs de .NET (nombres entiers, nombres réels, booléens, caractères,...) peuvent être encapsulés automatiquement dans des classes par des opérations de *boxing/unboxing*. On peut donc les manipuler comme des classes, de manière transparente. Pour construire un objet à partir d'une chaîne de caractères, deux méthodes ont été retenues. La première en utilisant un constructeur prenant une chaîne de caractères comme unique paramètre (ligne 12 à 15). Si un tel constructeur n'existe pas, on essaye la deuxième méthode. On recherche une méthode `Parse` qui prend une chaîne de caractères comme paramètre (lignes 18-19). Si cette méthode est statique et qu'elle retourne le bon type (lignes 20 et 21), elle est appelée pour construire l'objet. La méthode Parse est une convention implicite largement utilisée dans l'environnement .NET. Quand elle est implémentée, elle permet d'analyser une chaîne de caractères et de construire un objet avec la valeur de la chaîne. Cette technique permet de prendre en charge de manière transparente les types primitifs de .NET et un certain nombre de classes définies dans la bibliothèque de .NET comme DateTime ou IPAddress. Les types de données définis par les concepteurs sont également supportés sans modification de l'application. Il suffit que le l'utilisateur définissent un constructeur avec une chaîne de

caractères ou une méthode `Parse` et l'environnement de vérification peut manipuler ce type.

Quand les deux opérandes sont construits, une dernière vérification est effectuée : il faut s'assurer que la comparaison demandée est possible. .NET définit une interface `IComparable` qui est implémentée par les types dont les valeurs peuvent être ordonnées. Si l'opérateur de comparaison est « inférieur », « inférieur ou égal », « supérieur » ou « supérieur ou égal », le type des opérandes doit implémenter `IComparable`.

L'ensemble des propositions atomiques d'un observateur est stocké dans une table de hachage. Cette table permet de réutiliser la même proposition atomique à chaque fois qu'elle est rencontrée dans une formule.

4.5.3 Transformation en automates

Au cours de cette étape, les formules LTL et RTLTL sont transformées en automates qui constituent les observateurs. Suivant la logique temporelle, l'outil de transformation utilisé est différent. Les structures de données utilisées pour représenter les automates générés sont donc elles aussi différentes. Le diagramme de classes des observateurs est donné Figure 23. La classe `Observer` représente une généralisation du concept d'observateur, indépendante de la notion d'automate. Ceci facilite l'extension du moteur avec d'autres types d'observateurs, par exemple avec d'autres logiques, d'autres algorithmes de vérification ou encore pour de la visualisation. Un observateur est identifié par un nom. Celui-ci correspond au nom donné par l'utilisateur dans le fichier de spécifications. L'observateur contient aussi le chemin de l'événement utilisé pour la synchronisation avec le modèle en cours de simulation. Enfin, un commentaire facultatif peut être associé à l'observateur. Par exemple une description de la propriété vérifiée.

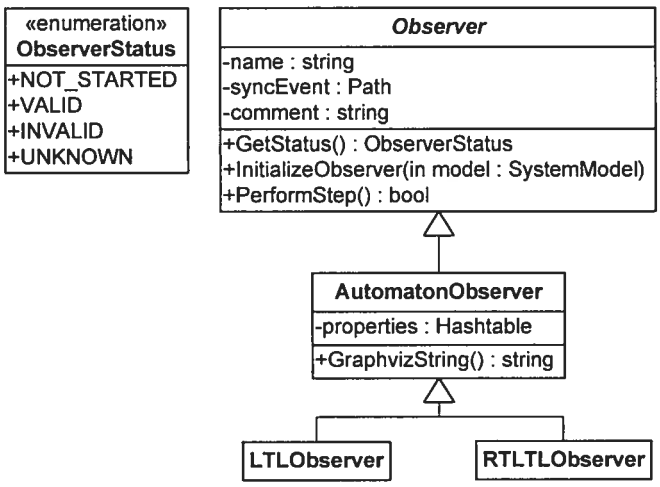


Figure 23 : Diagramme de classes des observateurs.

La classe `Observer` déclare en outre trois méthodes virtuelles qui devront êtres surchargées par les sous-classes :

GetStatus. Cette méthode retourne l'état actuel de l'observateur. L'état peut être :

- `NOT_STARTED` : l'observateur n'a jamais été exécuté. C'est l'état dans lequel se trouve l'observateur immédiatement après sa construction ;
- `VALID` : La propriété décrite par l'observateur a été vérifiée ;
- `INVALID` : La propriété décrite par l'observateur a été violée ;
- `UNKNOWN` : L'état de la vérification est inconnu. L'observateur se trouve dans cet état si le processus de vérification n'est pas terminé et que la simulation n'a pas été assez longue pour vérifier la propriété.

PerformStep. Elle est appelée à chaque occurrence de l'événement de synchronisation pour effectuer un pas dans le processus de vérification. Le détail de cet appel est donné dans la partie 4.6.

InitializeObserver. Cette méthode est appelée entre la phase d'élaboration et l'initialisation de la simulation. Elle permet de remettre l'observateur dans son état initial et de lier l'observateur à une nouvelle instance du modèle.

La classe `Observer` est raffinée par la classe `AutomatonObserver`. Cette dernière abstrait les fonctionnalités d'enregistrement de la trace d'exécution d'un automate. Cette trace est utilisée pour la visualisation de l'automate. Cependant, la classe

`AutomatonObserver` ne définit pas de structures de données pour l'exécution des automates, comme les états ou les transitions. En effet, celles-ci sont différentes pour LTL et RTLTL. Elles sont donc définies dans deux sous-classes `LTLObserver` et `RTLTLObserver`.

La transformation d'une formule LTL vers un automate est effectuée par une version modifiée de LTL2BA, un outil développé à l'origine pour la vérification formelle. La sortie de la version standard de LTL2BA est une description de l'automate dans un langage destiné au *model-checker* SPIN. Plutôt que d'analyser la sortie textuelle de LTL2BA, nous avons préféré le modifier pour traduire ses structures de données internes vers les structures de données de notre outil de vérification. Le programme LTL2BA est écrit en C. L'environnement .NET ne permet pas la compilation de programme C en code intermédiaire, mais il facilite l'interopérabilité entre les programmes en mode géré (*managed*) et ceux en mode non géré (unmanaged). Un programme en mode géré est un programme .NET classique, qui s'exécute dans le Common Language Runtime (CLR). Il bénéficie des caractéristiques de .NET comme la gestion automatique de la mémoire et l'introspection. Un programme en mode non géré est un programme compilé en code natif, c'est-à-dire directement dans le jeu d'instructions de la plateforme sur laquelle il est exécuté. Ici, LTL2BA est compilé en mode non-géré tandis que l'environnement de vérification est en mode géré. .NET offre plusieurs techniques d'interopérabilité. La solution mise en place ici est une interface entre le programme géré et le programme non géré. L'interface est écrite en C++ géré, un langage développé pour .NET avec une syntaxe proche de C++. Bien que s'exécutant en mode géré, un programme en C++ géré peut appeler facilement un programme non géré. .NET permet de compiler dans une même « assembly » du code C++ géré et du code non géré. L'interface et LTL2BA sont donc compilés dans une bibliothèque commune. L'interface est chargée de traduire les paramètres venant du mode géré et d'analyser la sortie de LTL2BA pour construire une structure de données gérée. La Figure 24 donne une vue globale des interactions entre l'environnement de vérification et le programme LTL2BA.

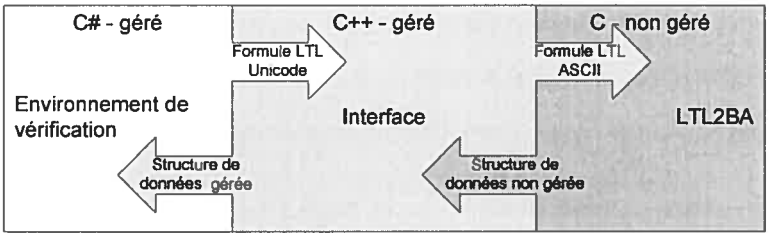


Figure 24 : Interaction avec LTL2BA.

Dans l'environnement de vérification, la formule LTL à transformer est stockée dans un objet de type string au format Unicode, avec 16 bits pour représenter un caractère. Cependant, LTL2BA doit être appelé avec une chaîne de caractères au format C, avec un codage sur 8 bits. L'interface fait la transformation en utilisant la méthode `System.Marshal.StringToHGlobalAnsi` qui retourne un pointeur vers la suite de caractères terminée par un octet nul. Ce pointeur est utilisé comme paramètre lors de l'appel à LTL2BA. La mémoire allouée pour la chaîne est ensuite libérée manuellement. Les données générées par LTL2BA sont analysées pour extraire la structure de l'automate. Au cours de cette transformation, le lien entre les propositions atomiques et l'observateur est reconstruit. Dans LTL2BA, les étiquettes des transitions sont des conjonctions de propositions atomiques. Ces dernières sont identifiées par une étiquette. A partir de ces étiquettes et de la table de hachage contenant les propositions atomiques (voir partie 4.5.2), l'interface construit les conditions de validité des transitions dans l'automate. Le résultat de la transformation est une structure de donnée en mode géré, directement manipulable par un programme C#.

Pour la transformation de formules RTLTL en automates, nous avons implémentés l'algorithme GPVW95 présenté dans les parties 3.3.3 en y apportant les modifications données dans la partie 3.5. Bien que conçu pour être exécutable à la volée, c'est-à-dire pendant le processus de vérification, il est ici exécuté pendant la construction du moteur de vérification, avant la simulation. La transformation d'une formule se fait en deux étapes. Dans un premier temps, la formule est mise en forme normale négative. Dans un second temps, elle est découpée pour construire les états et les transitions de l'automate.

La transformation en forme normale a pour objectif la réécriture de la formule RTLTL en une formule plus simple, ne comprenant que des opérateurs primitifs. La définition de la forme normale négative est donnée dans la partie 3.3.3, elle comprend deux

aspects. D'une part la suppression des opérateurs simplifiables (`[]`, `<>`, `->` ...) et d'autre part la transformation pour repousser l'opérateur de négation jusqu'aux propositions atomiques. L'entrée de l'algorithme de transformation est un arbre de syntaxe abstraite représentant la formule (voir Figure 21). La sortie est un autre arbre de syntaxe abstraite, utilisant la même structure de données, représentant la formule sous forme normale négative. Cette structure de données est non mutable, dans le sens où son état ne peut être modifié. En d'autres termes, un arbre de syntaxe abstraite ne peut être transformé directement, il faut construire un nouvel arbre pour représenter la nouvelle formule. La formule doit en quelque sorte être clonée. La raison de cette non mutabilité est d'assurer la robustesse de l'algorithme de transformation qui manipule profondément les formules. Il faut éviter qu'un sous arbre soit modifié dans un des nœuds alors qu'il est référencé dans d'autres. Ceci a peu d'impact sur les performances car les formules restent de tailles modestes avec quelques dizaines de nœuds dans l'arbre de syntaxe abstraite.

L'algorithme de transformation en forme normale négative implémente directement les règles de transformation données dans le Tableau IV. Le cœur de l'algorithme est une fonction qui explore récursivement l'arbre et qui construit le nouvel arbre. Les étapes de transformation d'un nœud sont :

1) Identifier le type d'opérateur ;
2) Si besoin, construire le ou les opérande(s) pour le nouvel opérateur ;
3) Appeler récursivement la fonction de transformation sur les opérandes ;
4) Construire le nouveau nœud avec le ou le(s) opérande(s) transformé(s).

À l'étape 1), la règle de transformation à utiliser est choisie suivant le type d'opérateur. Par exemple la formule `a -> b`, doit être transformée en `!a || b`. Les nouveaux opérandes sont construits à l'étape 2). Par exemple, le sous-arbre représentant `!a` est construit mais me sous-arbre de l'opérande de droite n'est pas modifié. L'appel récursif de la fonction est fait à l'étape 3), sur les deux nouveaux opérandes. Enfin, le nouveau nœud de l'arbre est construit à l'étape 4). Le Code Source 21 donne un extrait de l'implémentation de cet algorithme.

```
1.  public static FormulaNode Translate(FormulaNode node){
2.    if(node is LeafNode) return (FomrulaNode)node.Clone();
3.    if(node is ImplyNode){ // a -> b => !a || b
4.      ImplyNode n = node as ImplyNode;
5.      FormulaNode newLeft = Translate(new NotNode(n.LeftOperand));
6.      FormulaNode newRight = Translate(n.RightOperand);
7.      return new OrNode(newLeft, newRight);}
8.    //[...]
9.    if(node is NotNode){
10.     NotNode n = node as NotNode;
11.     if(n.Operand is UntilNode){ //!(a U b) => (!a R !b)
12.       UntilNode n2 = (UntilNode)n.Operand;
13.       FormulaNode newLeft = Translate(new NotNode(n2.LeftOperand));
14.       FormulaNode newRight = Translate(new NotNode(n2.RightOperand));
15.       return new ReleaseNode(newLeft, newRight);}
16.     //[...]
17.   }}
```

Code Source 21 : Extrait de la méthode de transformation en forme normale.

Les lignes 3 à 7 reprennent l'exemple donné dans le paragraphe précédent et remplace l'opérateur d'implication par un « ou logique ». Les instructions des lignes 9 à 15 permettent de « repousser » vers l'intérieur de la formule l'opérateur de négation qui se trouve devant l'opérateur Jusqu'à. Aux lignes 13 et 14, les deux opérandes sont construits puis eux-mêmes transformés. Ligne 15 le nouveau nœud correspondant à l'opérateur de « relâche » est construit.

Après que la formule soit mise sous forme normale négative, èlle est transformée en un automate par l'algorithme donné dans le Code Source 13 (partie 3.3.3). La structure de donnée principale de cet algorithme est la classe `Node`, qui correspond à un nœud potentiel du graphe décrivant l'automate. Cette classe comprend les champs donnés dans la partie 3.3.3 et contient une méthode `Expand` semblable à celle donnée dans le Code Source 13. Cette méthode décompose la formule et ajoute les nœuds de l'automate à un objet de la classe `RTLTLObserver`. Dans cette classe, les nœuds sont stockés dans une table de hachage dont la clef est le nom du nœud. Au début de l'algorithme, deux nœuds sont créés. Le premier est nommé `init` et ne contient aucune formule. Il représente l'état initial de l'automate. Le second nœud est le nœud de départ de l'algorithme qui contient la formule à décomposer. Une transition est ajoutée entre le nœud `init` et ce premier nœud. Cette transition est valide si la formule dans le deuxième nœud est vérifiée.

Quand un nouveau nœud est ajouté à l'automate, les transitions entrant dans ce nœud sont construites. Les conditions de validité pour ces transitions sont déterminées par l'ensemble `Old` du nœud. Cet ensemble contient les formules RTLTL, y compris les propositions atomiques, qui doivent être vérifiées dans le nœud courant. Ainsi, pour pouvoir « entrer » dans ce nœud, il faut que toutes les formules de l'ensemble `Old` soient vérifiées. Si cet ensemble est vide, cela signifie qu'il n'y a pas de conditions de validité particulière, la transition est toujours valide.

La définition des états acceptant est différente de celle donnée dans GPVW95 [72]. Un état de l'automate est acceptant si la totalité de la formule a pu être vérifiée dans cet état. En utilisant l'ensemble `Next`, on peut déterminer quelle partie de la formule il reste à vérifier. L'ensemble `Next` contient les formules devant être vérifiées dans les états immédiatement suivants. Ainsi, si cet ensemble est vide, cela signifie qu'il ne reste plus rien à vérifier et que l'état est acceptant. Pour certaine formule, il n'existe pas d'état acceptant. Par exemple, un formule du type `[]p` n'a pas d'état acceptant car il reste toujours une propriété à vérifier dans l'état suivant. Dans ce cas, la réponse de l'algorithme de vérification sera soit « la propriété n'est pas valide » soit « la trace d'exécution n'est pas assez longue ».

La Figure 25 donne l'ensemble de nœuds obtenus pour la formule « `a U b` ». Pour plus de clarté les proposition atomique ont été remplacées par des étiquettes `a` et `b` et le nœud `init` n'est pas représenté.

node_1	
New =	{}
Old =	{« a U b », « a »}
Next =	{« a U b »}
Incoming =	{init, node_1}

node_2	
New =	{}
Old =	{« a U b », « b »}
Next =	{}
Incoming =	{init, node_1}

node_3	
New =	{}
Old =	{}
Next =	{}
Incoming =	{node_2, node_3}

Figure 25 : Nœuds générés pour la formule « `a U b` ».

Le résultat de la décomposition de la formule « `a U b` » est un ensemble de trois nœuds. L'ensemble `Old` du nœud `node_1` contient les formule « `a U b` » et « `a` ». Ainsi, pour entrer dans le nœud `node_1`, il faut que la proposition atomique `a` soit valide. L'automate 1 de la Figure 26 représente l'automate correspondant. Les transitions entrantes de l'état 1

sont bien étiquetées avec la proposition `a`. L'ensemble `node_3.Old` est vide, car aucune propriété ne doit être vérifiée pour entrer dans ce nœud. Les transitions entrantes de l'état 3 sont donc étiquetées « `true` » pour indiquer qu'elles sont toujours valides. On peut remarquer que cet automate n'est pas optimal : les états 2 et 3 peuvent être fusionnés sans modifier le langage reconnu par l'automate.

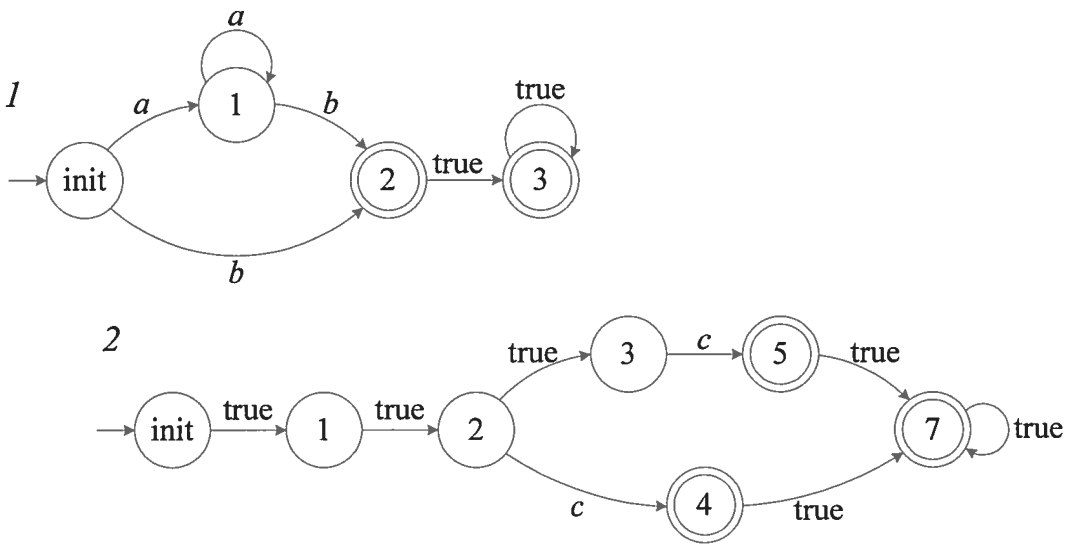


Figure 26 : Automate généré pour « `a U b` » (*1*) et « `<>[2,3] c` » (*2*).

Le deuxième automate de la Figure 26 est généré par la formule « `<>[2,3] c` ». Les deux transitions `true` qui suivent l'état initial permettent « d'attendre » le deuxième cycle imposé par l'intervalle [2,3]. Ensuite, il y a deux possibilités pour vérifier la formule, représentées par les deux branches sortant de l'état 2. Dans la branche du bas, la proposition `c` est vérifiée au cycle 2. Dans la branche du haut, elle est vérifiée au cycle 3.

4.5.4 Discussion

Dans cette partie nous avons présenté les deux techniques utilisées pour transformer les formules LTL/RTLTL en automates. La première méthode implémentée est celle basée sur LTL2BA. La seconde est une implémentation complète de l'algorithme de transformation GPVW95, pour prendre en charge RTLTL. Une autre possibilité pour la transformation de RTLTL aurait été de réécrire les formules LTLTL en LTL, puisque RTLTL est un ensemble de sucres syntaxiques sur LTL. On aurait pu également modifier LTL2BA pour prendre en charge les opérateurs RTLTL. Nous avons cependant choisi

d'implémenter un algorithme de transformation pour faciliter les extensions futures de l'environnement.

De plus, bien que notre implémentation permette également la transformation de formule LTL, nous avons décidé de continuer à utiliser LTL2BA pour la transformation de formules LTL, pour des raisons de performances. En effet, les automates générés par notre algorithme sont beaucoup moins compacts que ceux produits par LTL2BA. Une comparaison des performances des deux algorithmes est donnée dans la partie 6.2.3.

Cette séparation des flux de vérification entraîne une légère divergence des sémantiques : une formule LTL transformée avec LTL2BA n'aura pas exactement la même sémantique que la même formule transformée avec notre algorithme. Par exemple, pour une expression de la forme « `[] p` », l'automate produit par LTL2BA donnera un résultat positif si `p` reste vraie sur toute la durée de la simulation. Mais l'automate que nous générons ne donne jamais de résultat positif pour une telle formule. Si `p` est toujours vraie, le résultat est : « la trace d'exécution n'est pas assez longue pour donner un résultat définitif ». Cette différence de sémantique vient du fait que les états acceptant ne sont pas déterminés de la même façon dans les deux algorithmes. LTL2BA a été créé pour le *model-checking*, alors que nous notre algorithme a été pensé pour la vérification par simulation. Pour résoudre ce problème de sémantique, il faut améliorer notre algorithme pour obtenir des performances se rapprochant de celles de LTL2BA. On peut ensuite utiliser notre implémentation en lieu et place de LTL2BA. La principale amélioration à apporter concerne l'optimisation du nombre d'états. Dans le *model-checking*, la réduction du nombre d'états est un problème crucial et c'était l'objectif de LTL2BA. La vérification par simulation est moins pénalisée par le nombre d'états, mais il convient tout de même de le limiter pour réduire les temps d'exécution et la taille en mémoire de l'environnement.

4.6. Le moteur d'observateurs

Dans cette partie, nous allons détailler la structure et l'exécution du moteur d'observateurs. Plus particulièrement, nous allons présenter la synchronisation avec le simulateur ESys.Net, ainsi que la liaison entre les observateurs et le modèle puis le processus d'évaluation pendant la simulation.

4.6.1 Liaison avec le simulateur

La synchronisation entre les observateurs et le simulateur est assurée par le moteur d'observateur dont la structure est présentée dans La Figure 27. Le moteur d'exécution est chargé de déclencher la mise à jour des observateurs après l'occurrence d'un événement de synchronisation.

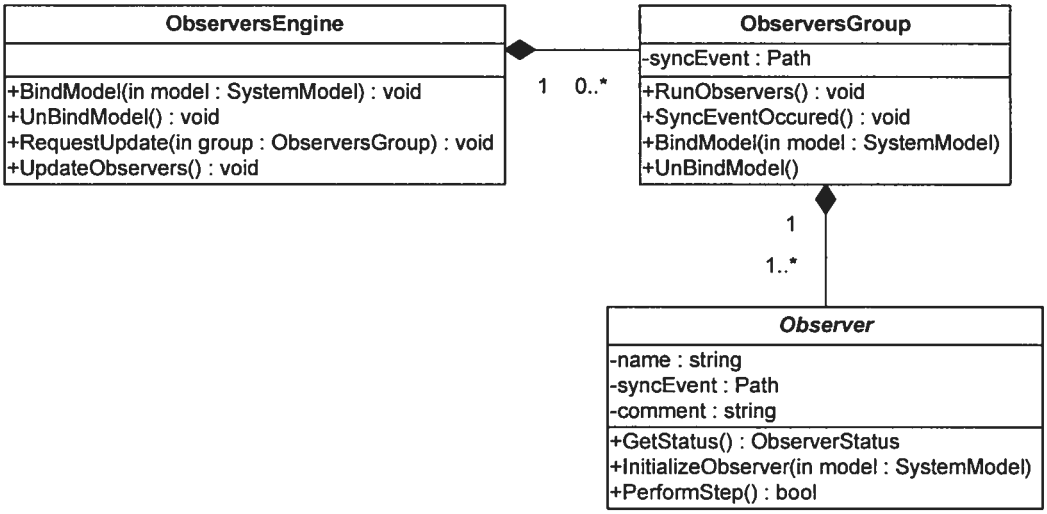


Figure 27 : Diagramme de classes du moteur d'observateurs.

La classe de base est l'observateur. Il s'agit de la classe abstraite présentée dans la partie 4.5.3. La méthode `PerformStep()` de cette classe déclenche la mise à jour de l'automate. Chaque observateur contient le chemin de l'événement qui le synchronise avec le simulateur. Tous les observateurs synchronisés avec le même événement sont regroupés dans un objet de la classe `ObserversGroup`. Cette classe maintient à jour deux ensembles d'observateurs : les observateurs en cours d'exécution et les observateurs ayant échoués. La méthode `ObserversGroup.RunObservers()` appelle la méthode `PerformStep()` sur chacun des observateurs en cours d'exécution. Le moteur d'observateurs est symbolisé par la classe `ObserversEngine` qui contient un ensemble de groupes d'observateurs. La méthode `UpdateObservers()` déclenche l'exécution de tous les groupes d'observateurs devant être mis à jour.

La synchronisation entre les observateurs et le simulateur est effectuée conjointement par l'objet `ObserverEngine` et les groupes d'observateurs. Du fait de l'algorithme d'ordonnancement des processus et de mise à jour des signaux d'ESys.Net,

l'exécution des observateurs ne peut être effectuée immédiatement à l'occurrence de l'événement de synchronisation. En effet, cet événement est déclenché pendant la phase de mise à jour des signaux ou bien pendant l'exécution des processus. Le modèle se trouve alors dans un état « instable », car la totalité des signaux n'a pas été mise à jour. De plus, l'ordre exact d'occurrences des événements est inconnu, car il dépend de l'ordre dans lequel sont exécutés les processus, et n'est pas spécifié par la sémantique du simulateur. Enfin, l'enchaînement exact des delta-cycles est difficilement prévisible. La Figure 28 schématise la simulation dans ESys.Net.

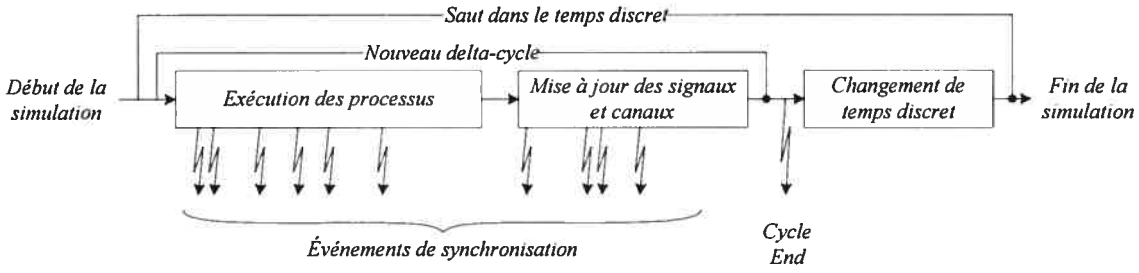


Figure 28 : Schéma de l'exécution du simulateur.

Chaque delta-cycle est composé de deux phases, l'exécution des processus et la mise à jour des signaux et des canaux. Les événements de synchronisation peuvent être déclenchés dans ces deux phases, dans un ordre indéterminé. Un certain nombre de delta-cycles peuvent survenir avant un changement du temps discret. Le modèle est stabilisé quand la totalité des delta-cycles a été exécutée, avant le changement de temps discret. Le simulateur ESys.Net déclenche l'événement `CycleEnd` à cet instant. La mise à jour des observateurs doit donc se faire à l'occurrence de l'événement `CycleEnd`.

On donc deux synchronisations : la première directement sur les événements de synchronisation des observateurs et la seconde sur l'événement `CycleEnd`. Les groupes d'observateurs se connectent aux événements de synchronisation dans le modèle. Cette liaison se fait grâce à la méthode `BindMethod` du simulateur :

```
1.  Event eventObject = (Event) ModelResolver.Resolve(model,SyncEvent);
2.  string handlerID = simulator.BindMethod(
3.          new RunnableMethod(this.SyncEventOccured),
4.          eventObject, EventID());
```

À la ligne 1, une référence vers l'instance de la classe `Event` est récupérée. Cet objet est l'événement de synchronisation lui-même. L'appel à la méthode `BindMethod` est effectué lignes 2 à 4. Le premier paramètre est un délégué qui représente la méthode `SyncEventOccured` à appeler à l'occurrence de l'événement représenté par `eventObject`. La méthode `EventID()` retourne une chaîne de caractères unique. Cette chaîne sert à générer la valeur de retour de la méthode qui identifie la liaison.

Le moteur d'observateur s'enregistre quant à lui auprès du simulateur pour être déclenché à chaque fin de cycle. Cet enregistrement se fait simplement avec l'instruction : `simulator.cycleEnd += new RunnableMethod(UpdateObservers)` où `UpdateObservers` est la méthode à exécuter.

La Figure 29 donne un exemple de l'exécution de deux observateurs.

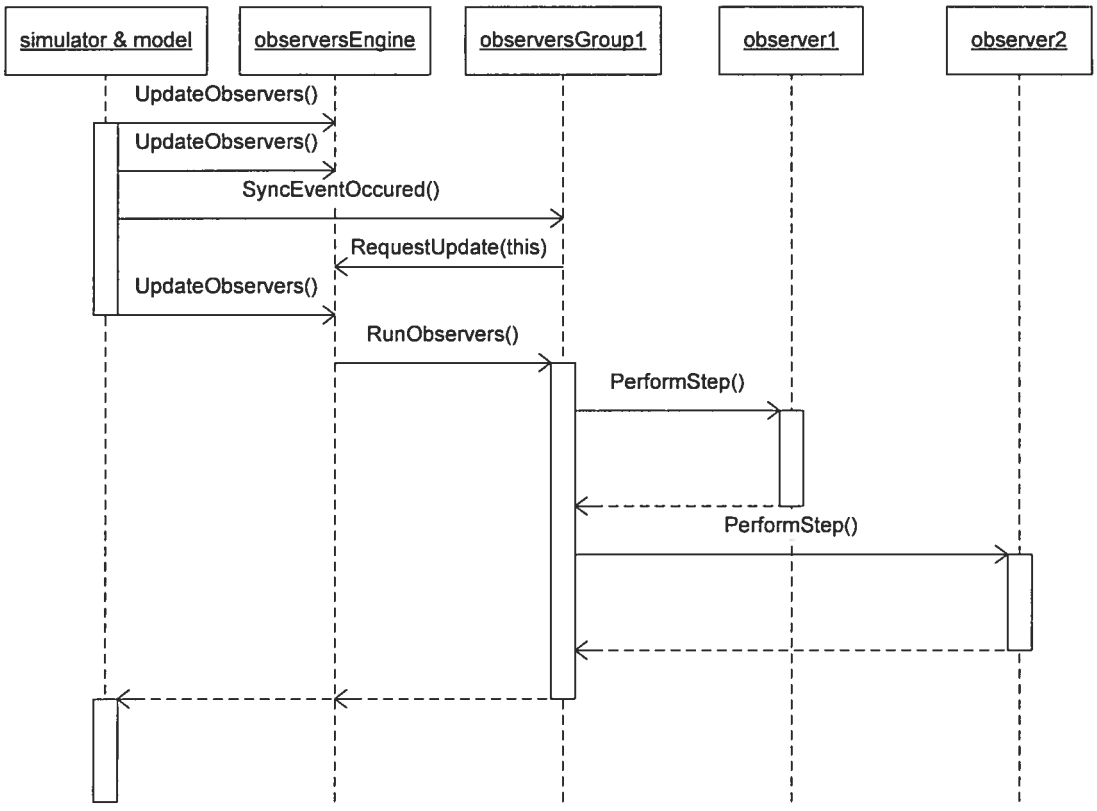


Figure 29 : Diagramme de séquence de l'exécution des observateurs.

À chaque occurrence de l'événement `CycleEnd`, la méthode `UpdateObservers()` du moteur d'observateurs est appelée. Si aucun groupe d'observateurs ne doit être mis à jour,

la simulation se poursuit. Quand un événement de synchronisation survient, la méthode `SyncEventOccured()` du groupe d'observateur concerné est appelée. Le groupe d'observateurs demande alors au moteur d'observateur d'être mis a jour à la fin du prochain cycle via la méthode `RequestUpdate()`. À la fin du cycle suivant, la méthode `UpdateObservers()` appelle `RunObservers()` pour tous les groupes ayant demandés une mise à jour. Cette dernière méthode exécute séquentiellement les observateurs qu'elle contient. Ce processus est détaillé dans la partie suivante.

Le découpage de l'environnement d'exécution des observateurs en trois entités – moteur, groupes et observateurs – permet de factoriser les déclenchements de méthodes sur les différents événements. Sans cette architecture, chaque observateur serait obligé de se synchroniser avec le simulateur et avec le modèle. Le surcoût engendré par les multiples appels de délégués diminuerait nettement les performances de l'environnement.

4.6.2 Exécution des observateurs et liaison avec le modèle

Dans cette partie, nous allons détailler l'évaluation des observateurs. Le moteur d'observateurs appelle la méthode `PerformStep()` qui va effectuer un pas dans l'exécution des observateurs. Un pas dans l'exécution d'un automate consiste à exécuter toutes les transitions sortantes des états courants (voir Code Source 14). Dans notre environnement de vérification, les transitions sont étiquetées avec des conjonctions de propositions atomiques ou bien avec des étiquettes « vraie » qui indiquent que les transitions sont toujours valides. L'exécution des observateurs consiste principalement en l'évaluation des propositions atomiques. Pour effectuer cette évaluation, la valeur des deux opérandes est nécessaire. Cette valeur est obtenue par introspection sur le modèle en cours de simulation. Afin de limiter l'impact sur les performances de l'environnement, une partie du travail d'introspection est effectuée avant le début de la simulation, pendant la phase d'initialisation.

On peut décomposer un modèle ESys.Net élaboré en deux parties. Une partie de l'information est statique, dans le sens où elle n'est pas modifiée pendant la simulation. La structure du modèle est par définition statique : un module ou un signal ne peut pas être instanciés pendant la simulation. A l'inverse, certaines informations sont dynamiques, i.e. leurs valeurs changent au cours de la simulation. La valeur d'un signal ou d'une variable est

typiquement une donnée dynamique. Les opérations d'introspection peuvent effectuées sur les données statiques avant la simulation. Mais la lecture des données dynamiques se fait impérativement pendant la simulation. Le langage de spécification de propriétés a été défini pour que le chemin d'un opérande soit statique. Par exemple, dans l'opérande `producer.clk.Prop(Value)`, `producer.clk` est une information statique alors que `Value` est dynamique. En d'autres termes, la valeur des références `producer` et `producer.clk` n'est pas modifiée au cours de la simulation, contrairement à la valeur de la propriété `Value`. On distingue principalement deux cas. D'une part, les objets ESys.Net (signaux, horloges, ports et canaux), pour lesquels l'objet lui-même est statique, mais dont la valeur est dynamique. D'autre part les propriétés et les variables non spécialisées, qui sont dynamiques.

Les objets ESys.Net qui sont observables dans notre environnement sont statiques, car on ne peut pas les instancier ou les déplacer pendant la simulation. Leur valeur est donnée par une propriété `Value`. Pour lier les opérandes de ce type, le Model Resolver présenté dans la partie 4.3.2 est utilisé pour obtenir une référence vers l'objet ESys.Net observé. Si cet objet est une horloge, on peut obtenir sa valeur directement via sa propriété `Value`. Si c'est un signal ou un port, cette propriété n'est pas accessible. En effet, l'architecture de ESys.Net définit une classe `BaseSignal` mais cette classe ne contient pas de propriété `Value`. La raison est que le type de la propriété `Value` dépend du type de donnée transporté par le signal. Or ce type est déterminé dans les sous-classes de `BaseSignal` qui implémente un signal pour un type donné. Il faut donc passer une nouvelle fois par l'introspection pour accéder à cette propriété pendant l'exécution. Le Code Source 22 donne les instructions nécessaires pour obtenir la valeur d'un signal pendant l'exécution.

```
1.  BaseSignal sig = ModelResolver.Resolve(model,path);
2.  //sig.Value est invalide, car Value n'est pas défini dans BaseSignal
3.  PropertyInfo pInfo = sig.GetType().GetProperty(name,
4.                BindingFlags.Instance | BindingFlags.Static |
5.                BindingFlags.NonPublic | BindingFlags.Public);
6.  object sigValue = pInfo.GetValue(sig);
```

Code Source 22 : Récupération de la valeur d'un signal.

À la ligne 1, la référence vers le signal observé est récupérée avec le *Model Resolver*. Les lignes 3 à 5 récupèrent les méta-données concernant la propriété `Value` du signal. Il faut

noter que le type retourné par l'appel `sig.GetType()` est le type réel du signal, héritant de `BaseSignal` et définissant la propriété `Value`. À la ligne 6, la valeur du signal est lue. Cet appel est fait pendant la simulation, alors que les instructions des lignes 1 à 5 sont effectuées à l'initialisation.

Les propriétés et les variables non spécialisées sont des données dynamiques, qui doivent être évaluées pendant la simulation. Cependant, le langage de spécification de propriétés permet uniquement d'accéder aux propriétés et aux variables qui sont contenues dans des modules, des canaux, des signaux ou des horloges. Ces objets étant statiques, l'introspection peut être effectuée sur l'objet contenant la variable ou la propriété à observer. Par exemple, la propriété `consumer.clk.Prop(Value)` est contenue dans l'objet `consumer.clk` qui est statique. On va donc récupérer avant la simulation une référence vers cet objet. Pendant la simulation, la méthode `GetValue` permettra d'obtenir la valeur de la propriété. La valeur d'une variable est obtenue de manière similaire.

Cette méthode permet de récupérer simplement la valeur des objets observables. Cependant, le coût des opérations d'introspection pendant la simulation, essentiellement des appels à `GetValue`, peut réduire les performances de l'application. Afin de limiter ce surcoût chaque proposition atomique n'est évaluée qu'une seule fois par pas d'exécution de l'automate, même si elle apparaît sur plusieurs transitions d'un automate. Une autre approche, passant pas l'émission dynamique d'instructions CIL, permet de supprimer ces appels. L'idée est de construire, pendant l'initialisation, une classe qui va lire la donnée désirée. La définition de cette classe pour un signal MyTypeSignal est donnée dans le Code Source 23.

```
1.  public interface ValueReader{
1.    object Value();}
2.  //Classe générée dynamiquement:
3.  public class MyTypeValueReader{
4.    MyTypeSignal sig;
5.    public ValueReader(MyTypeSignal s){sig=s}
6.    public object Value(){return sig.Value;}}
```

Code Source 23 : Récupération de la valeur par émission de code.

L'interface ValueReader est définie statiquement. Pour chaque type de signal observé, on construit dynamiquement une classe implémentant cette interface suivant le gabarit donné

des lignes 3 à 6. Le type `MyTypeSignal` n'a pas besoin d'être connu à la compilation, puisque cette classe est construite dynamiquement. Cette méthode à l'avantage d'avoir un surcoût pendant la simulation inférieur à la méthode présentée précédemment. En contrepartie, seules les membres publics sont accessibles, contrairement à l'approche introspective qui permet d'accéder à tous les membres, quelque soit leur visibilité. De plus, les versions actuelles de .NET n'offrent pas de méthode pour libérer la mémoire occupée par la définition d'une classe, ce qui implique de redémarrer l'application régulièrement si on génère un grand nombre de classes.

Une fois les valeurs des deux opérandes récupérées, la proposition atomique peut être évaluée. Pour cela les méthodes `Equals` ou `CompareTo` sont utilisées. Si le type des opérandes implémente l'interface `IComparable`, la méthode `lValue.CompareTo(rValue)` est appelée. Cette méthode retourne un nombre entier négatif, nul ou positif si `lValue` est respectivement inférieur, égal ou supérieur à `rValue`. Cette méthode doit obligatoirement être implémentée si l'opérateur de comparaison est différent de « == » ou « != ». Pour ces opérateur, la méthode `lValue.Equals(rValue)` peut être utilisée si `CompareTo()` n'est pas définie.

4.6.3 Discussion

Dans cette partie, nous avons présenté le moteur d'observateur et son exécution. Les observateurs sont exécutés de manière synchrone avec le simulateur ESys.Net, afin d'observer le modèle dans un état stable. Puisque la simulation et la vérification sont exécutées sur une machine séquentielle, nous avons choisi de suspendre la simulation pendant l'évaluation des observateurs. Leur temps d'exécution a donc un impact direct sur le temps total de simulation. Avec une exécution distribuée, il est cependant possible d'évaluer les observateurs en parallèle de la simulation puisqu'ils se contentent d'observer l'état du modèle et ne le modifient pas.

On peut remarquer que notre façon d'évaluer les observateurs diffère légèrement par rapport aux assertions de SystemVerilog. Dans SystemVerilog, les assertions sont elles aussi évaluées avant le saut dans le temps discret, une fois que le modèle est stable. Cependant, les variables et les signaux utilisés dans les assertions sont échantillonnés avant l'exécution des processus. La valeur utilisée dans une assertion est donc la valeur de la

variable au cycle précédent, afin de simuler l'échantillonnage fait par une bascule. Ceci permet d'éviter les « situations de compétition » (*race condition*) entre le modèle et les assertions. Nous n'avons pas rencontré ce problème avec notre environnement. Cependant, s'il survenait, le changement à effectuer pour adopter la même technique que SystemVerilog serait très simple : il suffirait d'utiliser l'événement `CycleInit` à la place de `CycleEnd`.

Chapitre 5 Implémentation – Interfaces Utilisateurs

Dans ce chapitre nous allons présenter les différentes interfaces graphiques développées pour faciliter l'utilisation de l'environnement de simulation. Dans la première partie, l'éditeur d'observateurs est présenté. L'application permettant de contrôler la simulation et la vérification d'un modèle est introduite dans la seconde partie. Enfin, la dernière partie détaille l'outil de visualisation de signaux. Des captures d'écran de ces trois applications sont données dans les Annexe C et D.

5.1. Éditeur d'observateurs

L'éditeur d'observateurs est une interface graphique qui permet à l'utilisateur d'éditer les propriétés à vérifier (Annexe C). L'objectif est de simplifier le processus de vérification en proposant une visualisation du modèle et en offrant des fonctionnalités d'édition avancée comme la coloration syntaxique, l'auto-complétion ou l'indentation automatique. L'éditeur intègre en outre une bibliothèque de patrons de formules LTL qui facilitent l'expression de propriétés complexes. Enfin, il cache à l'utilisateur le format des entête du fichier de propriétés en gérant dans l'interface les alias et le lien avec le modèle.

5.1.1 Visualisation du modèle

Ce module affiche l'arborescence du modèle dans un panneau à droite de l'éditeur proprement dit. Il permet d'avoir une vision globale du modèle et facilite la vérification de modèle dont les sources ne sont pas disponibles.

L'« assembly » contenant le modèle à vérifier est chargée via un menu, puis le Design Structure Tree correspondant est construit. Une représentation graphique de ce DST est donnée en utilisant un composant graphique de .NET. Des icônes permettent de symboliser les types des différents éléments du DST (modules, signaux …).

Afin de faciliter le travail sur des modèles dont la hiérarchie est profonde, la notion « d'arbre local » est introduite. L'utilisateur peut sélectionner un sous-arbre du DST qui représente la zone du modèle sur laquelle il est en train de travailler. Ce sous-arbre constitue « l'arbre local » et il est affiché dans un nouvel onglet. Cette notion concerne

seulement la représentation graphique, elle n'a aucune répercussion sur la syntaxe des formules.

Enfin, le composant d'affichage permet à l'utilisateur d'effectuer des « glisser / déposer » vers le composant d'édition pour faciliter la saisie de référence vers des objets du modèle.

5.1.2 Composant d'édition avancée

Ce composant est au cœur de l'éditeur d'observateurs. Il propose à l'utilisateur des fonctionnalités d'édition avancée qui réduisent les risques d'erreurs et le temps de spécification des propriétés.

Une barre d'outils permet d'insérer facilement les opérateurs temporels et booléens.

La coloration syntaxique affiche les mots-clefs et les opérateurs du langage dans une couleur différente. Elle met également en évidence la correspondance des parenthèses et des accolades. Quand le curseur se trouve à proximité d'une parenthèse, la parenthèse ouvrante ou fermante correspondante est affichée dans une autre couleur. Les parenthèses et les accolades qui ne sont pas correctement agencées sont affichées en rouge.

L'auto-complétion facilite la saisie de références vers des éléments du modèle en proposant à l'utilisateur une liste de choix suivant ce qu'il a déjà entré. Par exemple, s'il entre « `consumer.` », l'éditeur va lui proposer une liste contenant les sous-éléments du module `consumer`. Cette information est obtenue à partir du DST représentant le modèle. L'auto-complétion se déclanche automatiquement quand l'utilisateur saisie un point qui sépare les éléments d'un chemin dans le modèle. L'utilisateur peut également demander l'auto-complétion à n'importe quel moment avec les touches `ctrl+espace`.

Afin d'aider le formatage d'expressions LTL complexes, l'éditeur propose d'indenter automatiquement les lignes en ajoutant les espaces nécessaires pour représenter les niveaux d'imbrication dans une formule. L'auto-indentation est effectuée à l'insertion d'une nouvelle ligne. Si le dernier caractère entré est une parenthèse ou une accolade ouvrante, le niveau d'indentation est augmenté. Quand l'utilisateur ferme la parenthèse ou l'accolade, le niveau est décrémenté.

Les fonctionnalités d'édition avancée sont fournies par le logiciel libre Scintilla [94]. C'est une bibliothèque qui propose un composant d'édition portable (Windows et Linux). Le composant implémente les fonctions de base pour construire un éditeur et peut être personnalisé pour correspondre aux besoins du langage édité et de l'application. Nous avons défini un analyseur lexical basic pour reconnaître les mots-clefs et les opérateurs. L'auto-complétion est également gérée en partie par Scintilla. Pour personnaliser l'auto-complétion, il faut définir l'action qui la déclenche et construire la liste proposée à l'utilisateur.

5.1.3 Alias

L'éditeur d'observateurs gère également l'ensemble de alias définis dans le fichier de spécification. L'utilisateur peut ajouter, supprimer ou modifier les alias affichés dans la liste. Il peut également créer des alias à partir de la représentation graphique du modèle. La gestion des alias via une interface graphique permet de proposer l'auto-complétion sur des chemins comportant un alias.

5.1.4 Bibliothèque de patrons

Pour simplifier l'expression de formule LTL complexes, une bibliothèque de patrons est proposée. Il s'agit d'un ensemble de formules rencontrées communément dans les processus de vérification. La bibliothèque se base sur les travaux de [88] et reprend le classement de la Figure 12. Elle est stockée dans un fichier XML dont un extrait est donné dans le Code Source 24.

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <LTLPatterns xmlns="http://tempuri.org/LTLPatternsSchema.xsd" >
3.  <category label="Occurence">
4.    <subcategory name="Response" caption="S responds to P">
5.       <intent>To describe cause-effect relationships between a
6.  pair of states. </intent>
7.       <mapping scope="global" scope-label="Globally">
8.          <formula>[] (P -&gt; &lt;&gt;S)</formula>
9.          <param-list>
10.            <param>S</param>
11.            <param>P</param>
12.         </param-list>
13.      </mapping>
14. [...]
```

Code Source 24 : Extrait du fichier XML de définition de patrons.

Chaque patron est stocké dans un élément `subcategory`. Le sous-élément `intent` donne une courte description du patron. Chaque patron est ensuite décliné suivant les différentes portées dans les éléments `mapping`. Dans l'exemple ci-dessus, le patron de « réponse » est avec la portée « globale » est représenté. Le sous-élément `formula` donne la formule LTL correspondante et `param-list` contient la liste des paramètres du patron.

Le fichier XML contenant la bibliothèque de patrons est chargé automatiquement au démarrage de l'éditeur. Celui-ci va construire un ensemble de sous-menus représentant l'organisation des patrons. Les entrées dans ces menus permettent de créer un nouvel observateur à partir du patron en question. Il suffit de lui fournir le nom de l'observateur, l'événement de synchronisation et de définir la valeur des paramètres. La formule complète est générée à partir des valeurs des paramètres et elle est insérée dans l'éditeur.

5.2. Application de simulation

Cette application permet de contrôler la simulation et le processus de vérification. Elle charge le fichier de propriétés produit par l'éditeur et construit le moteur d'observateur. La liste des observateurs définis est affichée ainsi qu'une représentation graphique des automates. Une fois le fichier de propriétés chargé, l'utilisateur peut démarrer une simulation et en choisir la durée. L'application va alors instancier et élaborer le modèle, puis elle va le lier au moteur d'observateur avant de lancer la simulation. Quand elle se termine, l'utilisateur peut observer l'évolution des signaux du modèle et l'exécution des automates. Le résultat de l'évaluation des observateurs est également affiché. Une capture d'écran de l'application est donnée dans l'annexe D – 1.

5.2.1 Affichage des observateurs

Les observateurs, groupés suivant leur événement de synchronisation, sont affichés dans une liste. Une icône symbolise le résultat de l'évaluation de chaque observateur lors de la dernière simulation. Une représentation graphique de la structure de l'automate (états, transitions…) correspondant à l'observateur sélectionné est également affichée. Cette représentation permet des débuguer des observateurs simples, mais l'automate devient difficilement compréhensible quand la propriété se complexifie. Cet affichage se fait avec un outil nommé Graphviz [95] qui permet de générer une image représentant un graphe orienté à partir d'une chaîne de caractères décrivant les états et les transitions. Des

algorithmes pour le placement des états et des transitions permettent d'obtenir une représentation relativement claire. *Graphviz* est à l'origine un outil en ligne de commande, écrit en C. La bibliothèque *NGraphviz* permet de l'utiliser dans une application .NET. La licence de cette bibliothèque est problématique, car elle ne semble pas respecter la licence de *Graphviz*. Pour éviter tous problèmes dans notre application, *NGraphviz* est prise en charge sous forme de *plug-in* qui est chargé dynamiquement au démarrage de l'application. Ainsi il n'y a aucune dépendance vis-à-vis de *NGraphviz* à la compilation.

L'exécution des automates pendant la simulation est enregistrée. Á la fin de chaque pas dans l'exécution, l'état de l'automate est enregistré sous la forme d'une chaîne de caractères pour Graphviz. L'utilisateur peut ainsi revoir l'exécution avec l'évolution des états courants et des transitions effectuées. Cet enregistrement est coûteux, en temps autant qu'en mémoire. Il est donc désactivé par défaut et l'utilisateur peut choisir dans la liste des observateurs ceux dont l'exécution doit être enregistrée.

5.2.2 Gestion des erreurs et des sorties

Les erreurs qui surviennent à tous les niveaux ont besoin d'être rapportées à l'utilisateur. Des erreurs au chargement du fichier, pendant son analyse ou pendant la construction des observateurs doivent être clairement signalées. Pour gérer ces erreurs, l'application se repose sur le mécanisme des exceptions et sur la bibliothèque NLog [96].

NLog est une bibliothèque de gestion des journaux (*log* en anglais). Un journal est une séquence d'entrées. Chaque entrée est composée entre autre d'une date, d'un niveau d'alerte, d'une source et du message luis. Le niveau d'alerte (`Debug`, `Info`, `Warn`, `Error` ou `Fatal`) indique le degrés de sévérité de l'événement. La source indique la provenance du message, par exemple « Analyseur syntaxique » ou « Moteur d'observateurs ». Un aspect important de NLog est l'indépendance entre l'opération d'ajout d'une entrée et le stockage ou l'affichage du journal. Le journal peut être affiché sur la sortie d'erreur standard, dans une fenêtre, ou bien stocké dans une base de donnée. Cependant, le code pour émettre un message reste le même et c'est la configuration de NLog qui permet de choisir la ou les destination(s) des messages. Il permet également un filtrage sur le niveau d'alerte et sur la source du message.

Dans notre application, le journal est affiché dans une liste de la fenêtre principale. L'utilisateur a ainsi accès à l'historique des événements survenus. Quand une erreur survient, elle est reportée directement dans le journal en choisissant la sévérité. Une exception est également lancée pour alerter les niveaux supérieurs de l'application. Les erreurs sont ainsi détectées et reportées le plus tôt possible. Ceci permet une gestion plus fine des réponses à l'erreur : ignorer et alerter l'usager, stopper le processus, écrire un fichier contenant les informations pour le débuguage... L'utilisation de NLog permet également d'informer l'utilisateur sur l'état de l'application.

Certains modèles ESys.Net utilisent la sortie standard et la sortie d'erreur standard pour afficher des informations pendant la simulation. Comme l'application est graphique, elle ne possède pas de console et les sorties standards ne sont normalement pas affichées. Pour que l'utilisateur puisse accéder aux sorties, elles sont affichées dans la fenêtre de l'application. On utilise pour cela une méthode de .NET qui permet de détourner les sorties standards vers un nouveau flux qui est affiché dans une zone de texte.

5.3. Affichage des signaux

La visualisation de l'évolution des signaux et des horloges d'un modèle pendant la simulation est un outil important pour les processus de modélisation et de vérification. Utilisée conjointement avec les observateurs, elle facilite la détection d'erreurs de conception ou d'implémentation. C'est pourquoi nous avons développé un ensemble de bibliothèques permettant l'enregistrement de la valeur de signaux pendant la simulation et l'affichage dans une fenêtre. Les bibliothèques ont été conçues pour être réutilisable dans différents contextes. Les dépendances vis-à-vis de Windows, d'ESys.Net et de l'environnement de vérification ont été réduites au minimum.

L'outil de visualisation est composé de trois bibliothèques. *SimpleTrace* gère le stockage d'une trace d'exécution en mémoire. Le lien avec ESys.Net est fait par la bibliothèque *Trace4ESys*. Enfin, l'affichage est géré par *WinWaveForm*.

5.3.1 *SimpleTrace*

Cette bibliothèque est chargée du stockage en mémoire et de la manipulation de traces. Elle est totalement indépendante d'ESys.Net et de l'affichage. Une trace est une

séquence ordonnée de couples *(temps, valeur)* qui représente un changement de valeur du signal à un instant donné. Entre deux état i et i+1, la valeur du signal est égale à l'élément *valeur* de l'état i. Les données stockées dans une trace ne sont pas typées. Les opérations proposées sur une trace sont l'ajout d'une nouvelle valeur à la fin de la trace et la lecture de la valeur du signal à un instant donné. La modification des valeurs déjà stockées n'est pas permise. Afin de faciliter la manipulation de la trace et notamment l'affichage, la notion de segment de trace est introduite. Un segment de trace est une portion de la trace entre deux instants donnés.

5.3.2 *Trace4ESys*

Trace4ESys permet l'enregistrement de traces à partir d'ESys.Net. Elle est principalement composée d'une classe TraceRecorder qui effectue la connexion entre une trace et le modèle simulé. L'enregistrement se fait de la même façon que la récupération de valeurs dans le moteur d'observateurs : l'événement `sentitive` du signal observé est utilisé pour déclancher l'enregistrement d'un nouveau pas dans la trace. L'enregistrement de variables n'est pas supporté, mais il peut l'être simplement en définissant un événement pour l'échantillonnage de la variable pendant la simulation.

5.3.3 *WinWaveForm*

L'affichage d'un ensemble de traces sous la forme d'un graphique est assuré par la bibliothèque WinWaveForm. Le temps est représenté en abscisse et la valeur des signaux en ordonnée. Une capture d'écran est donnée dans l'Annexe D – 2. L'affichage est indépendant d'ESys.Net.

L'architecture modèle-vue-contrôleur est utilisée. Le modèle représente les données à afficher, la vue est la représentation graphique elle-même et le contrôleur permet de gérer la synchronisation entre le modèle et la vue et de répondre aux requêtes de l'utilisateur. Ce découpage permet entre autre de garder une certaine indépendance entre les modules et de faciliter le portage. Dans notre application le modèle est constitué d'un ensemble de traces. La vue est un composant graphique utilisant les *WinForms* de .NET. Les WinForms sont un ensemble de classes permettant le développement d'interfaces graphiques pour Windows. La vue est composée d'une liste affichant le nom des signaux, d'une échelle de temps, d'une barre de navigation et de l'affichage du graphique lui-même. La barre de navigation

permet à l'utilisateur de choisir la portion de trace à afficher. Enfin, le contrôleur gère la portion de trace affichée. Il répond aux requêtes de l'utilisateur (zoom avant, zoom arrière, déplacement) et modifie la zone affichée en conséquent. Il notifie la vue de ces changements.

Ces bibliothèques sont utilisées par notre application de simulation pour faciliter la vérification de modèles. Outre l'enregistrement de signaux et d'horloge, nous avons implémenté l'enregistrement des valeurs des propositions atomiques des observateurs.

Chapitre 6 Étude de cas et analyses de performances

L'étude de cas que nous allons présenter ici est basée sur un modèle de bus AMBA (*Advanced Microcontroller Bus Architecture*) écrit par un autre étudiant qui a ensuite utilisé notre environnement pour vérifier le comportement de son modèle. Nous nous sommes servi de cette étude de cas pour évaluer notre outil, en particulier ses performances.

Nous allons dans un premier temps présenter le modèle et les propriétés. Dans une deuxième partie, l'analyse des performances de l'application est détaillée.

6.1. Présentation du modèle et des propriétés

Le bus AMBA, spécifié par ARM [97] est un bus conçu pour les System-On-Chip. Trois types de bus sont spécifiés dont le *Advanced High-performance Bus* (AHB), un bus dédié à l'interconnexion de processeurs, de mémoires et de périphériques dans une puce. Il permet également la connexion avec des mémoires externes. Le bus AHB est un bus à hautes performances, avec une fréquence d'horloge élevée. Uns système conçu autour d'un bus AHB est composé d'un ou plusieurs maîtres, d'un ou plusieurs esclaves, d'un arbitre et d'un décodeur. Un maître est autorisé à déclencher une opération de lecture/écriture sur le bus. Un seul maître peut être actif sur le bus à un instant donné. Typiquement, un processeur, une interface de test ou un processeur de traitement de signal sont des maîtres sur un système AHB. Un esclave est l'entité qui répond à une requête d'un maître. L'arbitre est chargé de résoudre les conflits entre les maîtres et de s'assurer qu'un seul maître est actif à un instant donné. Enfin, le décodeur décode l'adresse de chaque transfert et génère un signal d'activation pour l'esclave impliqué dans une transaction.

Un sous ensemble d'AHB, nommé *AHB Lite* définie une architecture plus simple à partir de AHB [98]. Un bus AHB Lite est composé d'un seul maître. L'arbitre n'est donc pas nécessaire. De plus, le protocole de réponse est simplifié. Le modèle développé pour cette étude de cas, un bus AHB Lite, est représenté dans la Figure 30. Il peut contenir jusqu'à seize esclaves. Le nombre exact d'esclaves est défini à la compilation. Le banc de test, contenu dans les processus de maître et des esclaves, génère des transferts aléatoires.

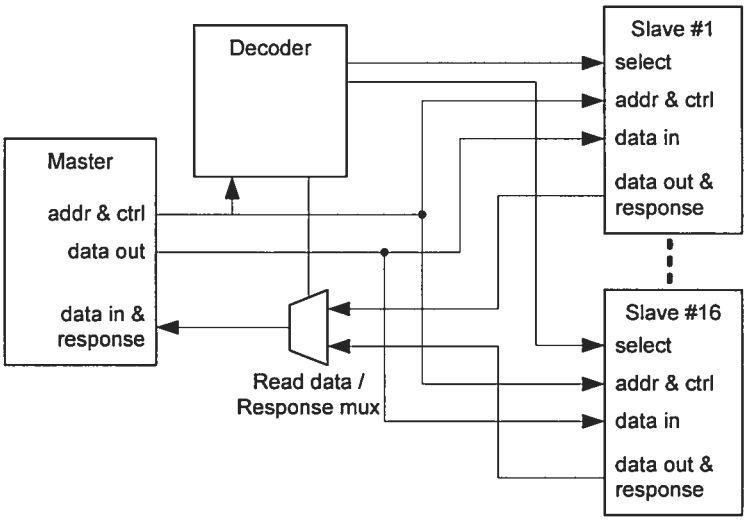


Figure 30 : Représentation sous forme de schéma bloc d'un bus AHB-lite.

Les propriétés vérifiées sur ce modèle sont dérivées d'un ensemble de règles sous la forme de texte en anglais. Ces spécifications nous ont été fournies par une société privée qui les utilise pour vérifier le bus et ses implémentations. Par exemple, la règle

> *« In a write transfer, after the address phase of the transfer is sampled, the master should provide valid data in immediate next cycle »,*

décrit un transfert en écriture. Les spécifications fournies contiennent environ quarante règles. La plupart de ces règles ont été traduites en LTL, à l'aide de l'éditeur. Certaines règles trivialement vérifiées par l'implémentation n'ont pas été incluses. Dans le Code Source 25, l'observateur LTL correspondant à la règle donnée ci-dessus est présenté.

```
1.  ltl_observer prop(clk.event(posedge)){
2.  [] (((sig(htrans) == "NSEQ" || sig(htrans) == "SEQ")
3.        && sig(hwrite) == true
4.        && sig(hready) == true)
5.       -> X (sig(hwdata) == "DATA"))}
```

Code Source 25 : Exemple d'observateur pour le bus AHB Lite.

La formule est composée de deux parties. La première (lignes 2 à 4) vérifie que le modèle est dans une phase de transfert. Le signal `htrans` identifie le type de trame actuellement transféré. Il peut être soit non séquentiel, soit séquentiel. Si la trame est la première d'un transfert en rafale, elle est de type non séquentiel. Les trames suivantes sont séquentielles. Le signal `hwrite` indique que le maître est en train d'écrire des données vers un esclave et le signal `hready` est la réponse de l'esclave pour signifier qu'il est prêt à transférer. La

deuxième partie de la formule, ligne 5, vérifie qu'une donnée valide est fournie par le maître au cycle suivant. Les deux parties sont connectées par l'opérateur d'implication. L'opérateur *toujours* (« [] »), placé devant la formule, s'assure qu'elle est valide durant toute la simulation. Outre les propriétés qui vérifient fonctionnement du modèle, des propriétés évaluant le taux de couverture de la simulation ont été ajoutées. Ces observateurs testent que tous les types de transactions ont été testés et que la totalité des esclaves a été activée. Ils sont de la forme `<> sig(htrans) == "NSEQ"` ou `<> sig(hsel) == 0`. Un observateur de cette forme a été écrit pour chaque type de transfert et pour chaque esclave. Au total, le fichier contient quarante deux observateurs LTL.

Cette étude de cas a permis de détecter un certain nombre d'erreurs, aussi bien au niveau du modèle que de l'outil de vérification. En ce qui concerne le modèle, plusieurs bugs concernant les cas limites et des problèmes de synchronisation ont été identifiés. Quant à l'environnement de vérification, des erreurs dans les interfaces graphiques et dans le moteur ont été détectées et corrigées. De plus, cette étude de cas nous a fourni une base pour une analyse des performances de l'application, présentée dans la partie suivante.

6.2. Analyse de performances

Les objectifs de l'analyse de performances sont d'essayer d'évaluer le surcoût du processus de vérification sur la simulation, d'identifier les goulots d'étranglement et d'optimiser les parties critiques, en terme de temps d'exécution. L'analyse a été effectuée à l'aide d'un *profiler*, AQtime [99]. Cet outil permet de mesurer les temps d'exécution d'un programme .NET, au niveau des méthodes ou au niveau des instruction C#. Ces mesures sont relativement précises, mais le processus d'analyse de performances augmente le temps d'exécution. L'information recueillie peut donc être utilisée pour effectuer des comparaisons entre les différents composants de l'application, mais elle ne constitue pas un indicateur absolu de sa performance. Le nombre d'appels d'une méthode ou le nombre d'exécutions d'une instruction C# peuvent également être comptés par le *profiler*. Ceci permet d'identifier les routines fréquemment exécutées et de concentrer les efforts d'optimisation sur celles-ci. Enfin, une évaluation de la consommation de mémoire de l'application est également possible.

Cette partie est composée de trois sous-parties. La première présente les performances de la phase d'initialisation. Dans la seconde, le surcoût des observateurs sur la simulation est évalué. Enfin, les performances des deux algorithmes de transformation sont comparées dans la dernière partie.

6.2.1 Résultats pour la phase d'initialisation

L'initialisation de l'environnement est constituée de toutes les opérations qui précèdent le lancement de la simulation et l'exécution des observateurs. Elle consiste principalement au chargement du fichier de propriétés, à la construction des automates et à la construction du modèle. Une partie du lien entre le modèle et les observateurs est également construite. L'objectif de cette analyse de performances est l'évaluation du coût de l'introspection en terme de temps d'exécution dans la construction de l'environnement de vérification. Les résultats sont présentés dans le Tableau X.

Opération	Pourcentage du temps d'exécution
Analyse du fichier	60%
Chargement du modèle et construction du DST	4%
Construction des observateurs Transformation en automates (96%) Construction des propositions (4%)	24%
Instanciation du modèle et élaboration	10%
Liaison du modèle et des observateurs	1%

Tableau X : Résultats de l'analyse de performances de la phase d'initialisation.

L'opération la plus lente est clairement l'analyse du fichier de propriétés. Ceci s'explique par le grand nombre d'entrées/sorties qui sont extrêmement lentes par rapport aux autres opérations. De plus, l'analyseur LL(k) peut-être contre performant. En effet, pour lever des ambiguïtés dans la grammaire, un *lookahead* de deux lexèmes est utilisé. Le lookahead d'un analyseur lexical est le nombre maximum de lexèmes que le l'analyseur peut « voir » pour décider quelle règle employer. Plus le *lookahead* est élevé, moins les performances sont bonnes.

Après l'analyse des fichiers, l'opération la plus coûteuse est la transformation des formules en automates. Cette opération comprend l'appel à LTL2BA et la construction des

structures de données de haut niveau à partir du résultat de LTL2BA. Le temps d'exécution de cette dernière par rapport à celui de l'appel à LTL2BA est négligeable.

Les opérations faisant appel à l'introspection sont le chargement du modèle, la construction du DST, la construction des propositions atomique et la liaison du modèle et des observateurs. Comme on le voit dans le Tableau X, elles ne représentent qu'environ 5% du temps total d'exécution.

6.2.2 Résultat pour la phase de simulation

L'objectif est d'évaluer le surcoût des observateurs sur le temps de simulation et d'identifier à l'aide du « profiler » les opérations les plus lentes ou les plus souvent répétées, afin de les optimiser. Pour calculer le surcoût, la méthode utilisée est la suivante : dans un premier temps, on mesure le temps d'exécution d'une simulation sans observateurs, puis avec l'ensemble des observateurs. Ceci nous permet d'obtenir le temps d'exécution des observateurs seuls, et de calculer le pourcentage qu'il représente par rapport à la durée de la simulation seule. La formule est donc $\frac{temps\ avec\ obs. - temps\ sans\ obs.}{temps\ sans\ obs.}$. Afin d'obtenir des données représentatives, une dizaine de simulations de 100 000 cycles ont été exécutées et le calcul a été effectué sur les valeurs moyennes.

La première évaluation des performances a été faite sur une version du modèle utilisant exclusivement des `ThreadProcess`, extrêmement lents à cause des multiples changements de contextes. Les temps d'exécution étaient très long : plusieurs minutes pour une simulation de 100 000 cycles. Le surcoût des opérateurs sur la simulation, de l'ordre de 2%, était alors négligeable.

Une seconde version du modèle, utilisant des `MethodProcess`, a ensuite été développée et a servi de base pour le reste de l'étude de cas. Le temps d'exécution moyen est devenu environ 1,5 seconde, entraînant une augmentation du surcoût des observateurs, de 70% environ. Ce résultat nous a amené à rechercher quelles étaient les opérations critiques et à tenter de les optimiser. Le Tableau XI présente les résultats obtenus.

Optimisations	Pourcentage du temps d'exécution consacré aux observateurs
Première version, pas d'optimisation	72,7%
Seconde version, optimisation de la lecture de l'état du modèle	70,1%
Troisième version, optimisation de la structure de données des automates	68,8%
Quatrième version, optimisation de l'évaluation des propositions atomiques	67,0%

Tableau XI : Surcoût des observateurs après les différentes optimisations.

La version dite « sans optimisation » est la première version de l'outil de vérification utilisée pour l'étude de cas. Elle sert de référence pour évaluer les gains des différentes optimisations. Les optimisations sont cumulatives, c'est-à-dire qu'une version *n* profite également des optimisations de la version *n-1*. La première amélioration apportée concerne la lecture de l'états du modèle, c'est-à-dire l'évaluation des opérandes des propositions atomique. Dans la version de référence, certaines opérations d'introspection étaient encore effectuées pendant la simulation, alors qu'elle pouvait se faire à l'initialisation. Cette optimisation a permis de gagner 2,6%. La seconde optimisation a été apportée sur les structures de données utilisées pour représenter les automates. En observant en détail les performances de l'exécution des automates, il est apparu que la manipulation des ensembles d'états était coûteuse. Ces opérations (ajout d'éléments, suppression, parcours) sont à la base de l'algorithme d'exécution. Elles sont élémentaires et ne devraient pas être si pénalisantes. En fait, c'est l'utilisation de collections fortement typées qui est problématique. Dans .NET, les collections de bases, comme les listes ou les tables de hachage sont dites faiblement typées, car elles manipulent des objets, sans contraintes sur leurs types. Pour améliorer la robustesse, on peut utiliser les collections spécialisées, fortement typées. La vérification du type est faite dynamiquement, contrairement aux *templates* de C++ où cette vérification se fait à la compilation. La vérification à l'exécution fait appel à l'introspection, relativement coûteuse. En utilisant des collections faiblement typées, au détriment de la robustesse de l'API, les performances ont été sensiblement améliorées. Enfin, dans la dernière version, la méthode de comparaison entres les opérandes lors de l'évaluation des propositions atomiques a été améliorée. La méthode `CompareTo` n'est utilisée que si elle est réellement nécessaire, i.e. quand l'opérateur de

comparaison est « inférieur », « inférieur ou égal », « supérieur » ou « supérieur ou égal ». Après ces trois optimisations, le pourcentage du temps d'exécution consacré aux observateurs est de 67%. L'utilisation du « profiler » nous a également permis de réduire la quantité de mémoire utilisée par le programme, notamment lors de l'enregistrement des traces d'exécution.

Plusieurs raisons expliquent le surcoût très élevé dans cette étude de cas. La première vient du modèle utilisé, qui est très optimisé. D'une part, il comporte un faible nombre de processus, un peu moins d'une vingtaine. D'autre part, son implémentation fait que l'ordonnanceur d'ESys.Net est « sous utilisé », car les processus sont quasiment ordonnancés statiquement. En effet, la synchronisation des processus avec des événements est relativement coûteuse dans ESys.Net. En « court-circuitant » cette opération, le modèle d'AHB réduit substantiellement le temps de simulation. Enfin, le nombre d'observateurs est relativement important par rapport à la taille du modèle. Ce surcoût n'est pas une valeur absolue. Il dépend complètement de la complexité du modèle et de son implémentation. Il est également directement relié au nombre d'observateurs et à leur complexité.

6.2.3 Comparaison des algorithmes de transformation

Comme nous l'avons vu, le temps d'exécution des observateurs dépend de la complexité des automates. Dans cette partie, nous comparons les performances des deux algorithmes utilisés pour effectuer la transformation de formule logique temporelle en automates, LTL2BA et notre implémentation de GPVW95 [72]. Dans un premier temps, nous analysons les performances de la transformation. Les critères d'évaluation sont le temps d'exécution, le nombre d'états de l'automate généré et le nombre de transitions. Ensuite, nous évaluons l'impact de la complexité des automates générés sur le coût d'exécution des observateurs.

Pour comparer les algorithmes, nous avons sélectionné neuf formules LTL. Cinq proviennent de l'étude de cas sur le bus AHB, deux sont des patrons de formules fréquemment rencontrées et les deux dernières sont extraites de [72], où elle sont également utilisées pour évaluer les performances de l'algorithme. Ces formules sont présentées dans le Tableau XII. Pour chacune de ses formules, le temps d'exécution a été mesuré, ainsi que le nombre d'états et de transitions. Afin d'obtenir des données fiables, les temps

d'exécution ont été mesurés sur plusieurs exécutions successives. Les valeurs données dans le Tableau XII sont les valeurs moyennes sur 1000 exécutions. Pour mesurer l'impact de la complexité des automates générés sur la durée des simulations, nous avons mesuré le temps d'exécution de la simulation avec un automate généré par LTL2BA (*t1*), puis avec un automates généré par notre implémentation (*t2*). La dernière colonne du Tableau XII présente le rapport $\frac{t2}{t1}$, soit le « facteur de ralentissement » introduit par notre algorithme par rapport à LTL2BA. Les valeurs de *t1* et *t2* ne sont pas représentées dans le tableau.

Formule LTL			LTL2BA			Notre implémentation			facteur de ralentissement de la simulation
			temps de transf. (ms)	nombre d'états	nombre de trans.	temps de transf. (ms)	nombre d'états	nombre de trans.	
Etude de cas	1	`[](((a||b)&&c&&d) ->Xe)`	0,6	2	8	4,7	9	36	1,7
	2	`[](((a||b) &&c&&d&&Xd) ->X(eUd))`	1,0	4	17	38,4	25	169	5,7
	3	`[]((a&&Xb&&Xc) ->X(e&&f))`	0,7	4	16	12,3	17	68	2,7
	4	`[]((a&&b&&c) ->X((d||e)Uf))`	0,7	3	16	34,8	17	148	3,4
	5	`[](((a||b)&&c) ->X(c&&d))`	0,5	2	5	2,8	6	16	1,6
Patrons	6	`[](a->(b-><>c))`	0,5	2	6	6,9	11	52	N/A
	7	`[]((a&&!b&&<>b) ->(cUb))`	0,7	4	9	18,2	22	96	N/A
GPWV	8	`a U (b U c)`	0,4	3	6	0,8	7	13	N/A
	9	`!(a U (b U c))`	0,5	3	6	1,6	10	26	N/A

Tableau XII : Comparaison des performances de LTL2BA et de notre implémentation.

En ce qui concerne le temps d'exécution de la transformation, notre implémentation est de deux à cinquante fois plus lente. Ce résultat n'a rien de surprenant, puisque l'objectif de LTL2BA était justement de trouver un algorithme plus performant que GPVW95 [74]. De plus, LTL2BA est une application écrite en C, fortement optimisée. A l'inverse, notre implémentation est écrite en C# et elle utilise beaucoup de constructions de haut niveau, connues pour être lentes (appels de méthodes virtuelles, utilisation de collections fortement typées, introspection). Il faut également remarquer que l'on compare deux algorithmes fondamentalement différents, bien qu'ils résolvent le même problème. Des optimisations de

notre implémentation sont possibles, mais il faut bien voir que cette pénalité en terme de temps d'exécution a un impact uniquement sur le temps de construction du moteur d'observateurs. Le temps de transformation des formules n'a pas d'influence sur le temps de simulation. Par contre, la taille des automates générés est importante.

Sur les exemples sélectionnés, notre implémentation produit des automates avec deux à sept fois plus d'états et deux à dix fois plus de transitions. Encore une fois, ce résultat n'est pas surprenant puisqu'une des caractéristiques de LTL2BA est de simplifier au maximum les automates qu'il utilise au cours de la transformation. Les simplifications se font de manière « statique », c'est-à-dire sur l'automate complet, mais aussi « à la volée », pendant les constructions des automates. A l'inverse, notre implémentation n'effectue aucune simplification de l'automate. Certaines simplifications de LTL2BA pourraient cependant être transposées dans GPVW95.

La taille des automates à un impact sur les temps d'exécution des observateurs pendant la simulation, représenté par le « facteur de ralentissement » donné dans la dernière colonne du Tableau XII. Ce facteur n'a été calculé que pour les propriétés provenant de l'étude de cas faute d'un modèle approprié pour les autres formules. On voit que ce facteur dépend directement du nombre d'états et de transitions dans l'automate. L'impact du nombre de transitions est limité par l'utilisation d'un « cache » pour l'évaluation de propositions atomiques qui composent les conditions de validité des transitions. Cette opération est la plus coûteuse dans l'exécution de l'automate et le cache permet de n'évaluer les propositions qu'une fois par cycle même si la propositions apparaît dans plusieurs les conditions de validité. Sans ce cache, le facteur de ralentissement sur les automates comportant de nombreuses transitions devient très important. Il est par exemple de 9 dans le cas de l'automate n°2.

L'occupation de la mémoire n'a pas été comparé, mais elle a déjà été effectuée par l'auteur de LTL2BA dans [74]. Cependant, on peut remarquer que la formule n°9, connue pour l'explosion combinatoire qu'elle génère, est transformable sans problème par notre algorithme. Par contre, une formule de même structure avec six opérateurs « jusqu'à » imbriqués provoque un débordement de la pile alors que LTL2BA est capable de la transformer.

6.3. Discussion

Dans cette partie nous avons présenté l'étude de cas effectuée avec notre outil, ainsi que les résultats de l'évaluation des performances.

L'étude de cas proprement dite, c'est-à-dire le développement du modèle et de l'ensemble a été effectué par un autre étudiant. Ceci nous a permis d'avoir un retour pertinent sur l'ergonomie et sur l'utilisation de notre application. Nous avons pu ainsi améliorer les interfaces utilisateur, ajouter certaines fonctionnalités manquantes et corriger des bugs. Nous en avons également retiré des informations sur l'évolution possible de l'application, surtout en ce qui concerne le langage de spécification. LTL n'est pas toujours pratique pour exprimer certaines propriétés, et le manque d'expressions rationnelles étendues s'est fait sentir.

L'étude de cas a en outre permis une analyse des performances de l'application sur des données plus complètes. Nous avons ainsi pu identifier les points critiques de l'application et les améliorer.

Enfin les deux algorithmes de transformation ont été comparés selon des critères de temps d'exécution et de complexité des automates généré. LTL2BA apparaît comme nettement plus performant que notre implémentation de GPVW95. C'est pourquoi il est toujours utilisé par défaut quand RTLTL n'est pas nécessaire. Cependant, notre implémentation peut être améliorée et servir de base pour la transformation de langages plus complexes, par exemple pour prendre en charge les expressions rationnelles étendues.

Chapitre 7 Conclusion et travaux futurs

La complexité de systèmes actuels implique de nouvelles méthode et outils pour la conception. En ce qui concerne la modélisation, une augmentation du niveau d'abstraction est indispensable pour permettre à l'être humain de définir les fonctionnalités et l'architecture des systèmes modernes. Les techniques de vérifications sont également adaptées pour faire face à cette tendance. Le développement récent qu'a connu la vérification par assertions ou la vérification semi-formelle illustre bien cette volonté d'automatiser les processus de vérifications complexes.

Ce mémoire présente un nouvel outil de vérification basé sur les observateurs, pour l'environnement de modélisation et de simulation ESys.Net. L'objectif est de démontrer par la pratique la pertinence de l'approche proposée par ESys.Net et d'illustrer l'intérêt de l'introspection pour les outils de conception de matériel. Notre outil de vérification tire partie des capacités de programmation avancée proposée par .NET et C#, notamment en ce qui concerne l'introspection, l'interopérabilité avec des bibliothèques natives et la programmation par événement. Nous proposons un environnement de vérification relativement complet et un ensemble d'outils réutilisable. Plusieurs interfaces graphiques sont offertes pour facilitent la spécification et la vérification de modèles.

7.1. Résumé

Dans le premier chapitre d'introduction, nous avons présenté le contexte dans lequel se situe notre travail, les problématiques rencontrées et la solution que nous proposons. Dans le chapitre suivant, nous avons dressé un état de l'art des techniques de modélisation et de vérification de matériel. Dans un premier temps, nous avons comparé les approches formelles et semi-formelles de la vérification. Nous avons ensuite donné une revue des différents langages et outils pour la vérification par simulation, ceci pour permettre au lecteur de situer notre travail par rapport aux solutions existantes, en terme de fonctionnalités. SystemC a été détaillé pour permettre une comparaison plus précise par la suite. La description de PSL, en tant que futur standard pour les langages de spécification

de propriétés a également été approfondie. La plateforme .NET et le langage C#, qui servent de base logicielle à notre outil ont été présentées en détaillant les fonctionnalités de programmation avancées telles que l'introspection ou les événements. Enfin nous avons décrit ESys.Net l'environnement de modélisation et de simulation pour lequel notre outil a été développé. L'algorithme d'ordonnancement a fait l'objet d'une analyse précise car il est au cœur du mécanisme de synchronisation avec les observateurs.

Les bases théoriques de l'application ont été données dans le chapitre 3. Dans un premier temps, la logique temporelle linéaire a été présentée à travers sa syntaxe sémantique et sa sémantique. Par la suite, nous avons expliqué le lien qui unit LTL et les automates de Büchi, ainsi que les algorithme permettant la transformation de LTL en un automate. Dans la partie suivante, l'algorithme utilisé pour la vérification de traces d'exécutions finies avec un automate a été expliqué. Nous avons également présenté RTLTL, une autre logique temporelle qui ajoute la notion quantitative du temps à LTL. Enfin, la notion de patrons de spécifications a été introduite.

Le chapitre 4 est le chapitre principal de ce mémoire. Nous y avons exposé en détail l'architecture et l'implémentation du moteur de vérification. Dans un premier temps, nous avons présenté le flux de vérification de manière globale, en donnant un aperçu du fonctionnement général de l'application. Pour souligner l'importance de l'introspection, nous avons comparé notre travail avec une solution hypothétique se basant sur SystemC. Dans la partie suivante, la syntaxe et la sémantique du langage de spécification de propriétés que nous avons conçu a été introduit. Puis nous avons présenté en détail des outils réutilisables qui facilitent l'introspection de modèles ESys.Net. Ils permettent notamment d'obtenir des informations sur la structure du modèle et sur son état pendant la simulation. Enfin, dans les dernières parties ont été détaillés l'architecture et le fonctionnement du moteur d'observateurs lui-même. Celui-ci constitue le cœur de l'environnement de vérification. Il est construit à partir du fichier de propriétés. Nous avons en particulier présenté l'implémentation de l'algorithme de transformation de formule RTLTL ainsi que l'extraction des propositions atomiques. L'exécution des observateurs pendant la simulation passe par un processus de synchronisation entre le moteur

d'observateur et le simulateur. Cette synchronisation tire partie des points d'ancrage proposés par ESys.Net.

Dans le chapitre 5, les deux interfaces graphiques proposées aux utilisateurs ont été présentées. La première est un éditeur d'observateur qui facilite la spécification de propriétés en offrant des fonctionnalités telles que l'auto-complétion ou la coloration syntaxique. L'application de simulation permet quant à elle de contrôler la simulation et le processus de vérification. Elle propose également une visualisation des résultats et une représentation graphique des automates. En plus de ces deux interfaces, un composant générique d'affichage de traces d'exécution a été présenté. Il a été conçu pour être le plus réutilisable possible et peut être utilisé indépendamment d'ESys.Net.

Enfin, le dernier chapitre porte sur l'étude de cas et l'analyse de performances. Nous avons présenté le modèle AHB et les propriétés développés par un autre étudiant. Cette étude de cas nous a permis d'avoir un retour d'information de la part d'un utilisateur, d'améliorer les différents outils et de corriger des bugs. Le modèle et les propriétés ont été utilisés pour analyser et améliorer les performances de l'application.

7.2. Travaux futurs

De nombreuses évolutions sont possibles autour du projet présenté ici. On peut cependant retenir deux grands axes.

- L'amélioration des performances. Elle peut se faire à plusieurs niveaux. Premièrement, la complexité des automates générés par notre implémentation de GPVW95 doit être réduite. Deuxièmement, l'implémentation elle-même de cet algorithme peut être optimisée. Enfin, l'exécution des automates est déjà optimale, mais des structures de données plus adaptées pourraient réduire les temps d'exécution.
- L'utilisation de PSL comme langage de spécification. PSL est en train de devenir le standard des langages de spécification de propriétés. Sa prise en charge par notre outil serait un atout majeur. La première étape de cette intégration serait la création d'un « parfum » ESys.Net pour PSL. De plus, le moteur doit être modifié pour respecter la sémantique avec plusieurs horloges des propriétés. Les expressions

rationnelles étendues doivent également être reconnues, peut être en modifiant l'algorithme de transformation.

Finalement, .NET et ESys.Net sont eux même en train d'évoluer. Notre outil devra certainement être adapté pour prendre en charge ces nouvelles versions. L'introduction de « génériques » va certainement modifier la méthode d'évaluation de l'état du modèle. En ce qui concerne ESys.Net, les changements porteront essentiellement sur la construction du « Design Structure Tree » et sur la méthode d'évaluation des propositions atomiques. La synchronisation avec le simulateur ne devrait pas changer.

Références

[1] "International Technology Roadmap for Semiconductors," 2005, http://www.itrs.net/Common/2005ITRS/Home2005.htm

[2] SystemVerilog, http://www.systemverilog.org

[3] SystemC, http://www.systemc.org

[4] ESys.Net, http://www.esys-net.org/

[5] F. Doucet, S. Shukla, and R. Gupta, "Introspection in system-level language frameworks: meta-level vs. integrated," presented at Design, Automation and Test in Europe, 2003.

[6] J. Lapalme, E. M. Aboulhamid, and G. Nicolescu, "A New Efficient EDA Tools Design Methodology," *ACM Transactions on Embedded Computing Systems*, 2005.

[7] Microsoft .NET, http://www.microsoft.com/net/

[8] Java Technology, http://java.sun.com/

[9] "Embedded Systems Roadmap 2002," http://www.stw.nl/NR/rdonlyres/3E59AA43-68B1-4E83-BA95-20376EB00560/0/ESRversion1.pdf

[10] N. Gorse, M. Metzger, J. Lapalme, E. M. Aboulhamid, Y. Savaria, and G. Nicolescu, "Enhancing ESys.Net with a Semi-Formal Verification Layer," presented at International Conference on Microelectronics, Tunis, 2004.

[11] M. Metzger, F. Bastien, F. Rousseau, J. Vachon, and E. M. Aboulhamid, "Introspection Mechanisms for Semi-Formal Verification in a System-Level Design Environment," presented at International Workshop on Rapid System Prototyping, Darmstadt, 2006.

[12] M. Metzger, F. Bastien, F. Rousseau, J. Vachon, and E. M. Aboulhamid, "Semi-Formal Verification Tool Implementation using Introspection Mechanisms," presented at Forum on specification and Design Language, Chania, 2006.

[13] J.-F. Monin, *Introduction aux méthodes formelles*, 2e édition ed. Hermès Science Publications, 2000.

[14] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, pp. 123-193, 1999.

[15] G. Huet, G. Kahn, and C. Paulin-Mohring, "The Coq proof assistant - a tutorial," INRIA Technical Report 178, 1995,

[16] T. F. Melham and M. J. C. Gordon, *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge
University Press, 1993.

[17] B. Bérard, *Systems and software verification : model-checking techniques and tools*. Berlin ; New York. Springer, 2001.

[18] A. Pnueli, "The temporal logic of programs," presented at 18th IEEE Symp. on Foundations of Computer Science (FOCS'77), 1977.

[19] 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.*, vol. 8, pp. 244-263, 1986.

[20] D. J. Steven, S. M. Paul, and C. Albert John, "Studies of the Single Pulser in Various Reasoning Systems," in *Proceedings of the Second International Conference on Theorem Provers in Circuit Design - Theory, Practice and Experience*: Springer-Verlag, 1994.

[21] Y.-A. Chen, E. M. Clarke, P.-H. Ho, Y. V. Hoskote, T. Kam, M. Khaira, J. W. O'Leary, and X. Zhao, "Verification of All Circuits in a Floating-Point Unit Using

Word-Level Model Checking," in *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*: Springer-Verlag, 1996.

[22] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, 2nd Edition ed. Kluwer Academic, 2003.

[23] "Assertion-Based Verification," Synopsys Juillet 2002,

[24] "OpenVera Assertions," Synopsys, Juillet 2003,

[25] PSL/Sugar Consortium, http://www.pslsugar.org/

[26] Accellera, "SystemVerilog 3.1a Language Reference Manual," 2004, http://www.eda.org/sv/SystemVerilog_3.1a.pdf

[27] H. Foster, E. Marschner, and Y. Wolfsthal, "IEEE 1850 PSL: The Next Generation," presented at Design and Verification Conference, 2005.

[28] S. Swan, "An Introduction to System Level Modeling in SystemC 2.0," http://www.systemc.org

[29] S. Yalamanchili, *Introductory VHDL: From Simulation to Synthesis*. Prentice Hall, 2000.

[30] E. M. Aboulhamid and J. Lapalme, "Recent trends in hardware/software description languages," presented at 15th International Conference on Microelectronics, 2003.

[31] S. Palnitkar, *Verilog HDL*. Prentice Hall, 2003.

[32] S. Bailey, "Comparison of VHDL, Verilog and SystemVerilog," Mentor Graphics - Functional Verification White Paper, 2003, http://www.mentor.com/products/fv/techpubs/

[33] Accellera, "Property Specification Language - Reference Manual," Version 1.1, 2004, http://www.eda.org/vfv/docs/PSL-v1.1.pdf

[34] IBM PSL/Sugar website, http://www.haifa.il.ibm.com/projects/verification/sugar/

[35] Accellera, http://www.accellera.org/home

[36] S. Iman and S. Joshi, *The e Hardware Verification Language*. Kluwer Academic Publisher, 2004.

[37] "Semantics of Temporal *e*," Verisity Ltd., 2003, http://www.ieee1647.org/downloads/temporale_denotational.pdf

[38] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar, "The ForSpec Temporal Logic: A New Temporal Property-Specification Language," in *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*: Springer-Verlag, 2002.

[39] IEEE P1850 PSL Working Group, http://www.eda.org/ieee-1850/

[40] GDL: General Description Language, http://standards.ieee.org/downloads/1850/1850-2005/gdl.pdf

[41] S. Swan, "Enabling PSL Assertions in SystemC," 2003, http://www.pslsugar.org/papers/pm2_stuart_psl_sysc.pdf

[42] PSL/Sugar-based Verification Tools, http://www.haifa.il.ibm.com/projects/verification/sugar/tools.html

[43] ModelSim PSL/Assertions, http://www.model.com/products/assertions.asp

[44] FoCs, http://www.haifa.il.ibm.com/projects/verification/focs/index.html

[45] "Draft Standard SystemC Language Reference Manual," Open SystemC Initiative, 2005, http://www.systemc.org

[46] "SystemC Verification Standard Specification," SystemC Verification Working Group, mai 2003, http://www.systemc.org

[47] C. N. Ip and S. Swan, "A Tutorial Introduction on the New SystemC Verification Standard," 2003, http://www.systemc.org

[48] D. S. Brahme, S. Cox, J. Gallo, M. Glasser, W. Grundmann, C. Norris Ip, W. Paulsen, J. L. Pierce, J. Rose, D. Shea, and K. Whiting, "The Transaction-Based Verification Methodology," Cadence Berkeley Labs technichal report #CDNL-TR-2000-0825, août 2000,

[49] A. Habibi, A. Gawanmeh, and S. Tahar, "Assertion based verification of PSL for SystemC designs," 2004.

[50] A. Habibi and S. Tahar, "Design for Verification of SystemC Transaction Level Models," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*: IEEE Computer Society, 2005.

[51] Edison Design Group C++ Front-End, http://edg.com/cpp.html

[52] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip," in *Proceedings of the 5th ACM international conference on Embedded software*. Jersey City, NJ, USA: ACM Press, 2005.

[53] D. Berner, H. Patel, D. Mathaikutty, S. Shukla, and J.-P. Talpin, "SystemCXML: An Extensible SystemC Front End Using XML," Formal Engineering Research using Methods, Abstractions and Transformations, Technical Report No: 2005-06, 2005, http://systemcxml.sourceforge.net

[54] Doxygen, http://www.stack.nl/~dimitri/doxygen/

[55] SystemPerl, http://www.veripool.com/systemperl.html

[56] S. Sutherland and D. Mills, "SystemVerilog 3.1 The Hardware Description AND Verification Language," 2003, http://www.sutherland-hdl.com/papers/2003-SNUG-paper_SystemVerilog.pdf

[57] J. Havlicek and Y. Wolfsthal, "PSL and SVA: Two Standard Assertion Languages, Addressing Complementary Engineerins Needs," presented at Design and Verification Conference, 2005.

[58] S. Bailey, "Comparison of PSL and SystemVerilog Assertions," Mentor Graphics - Digital Simulation White Paper, 2003, http://www.mentor.com/products/fv/techpubs/

[59] OpenVera, http://www.open-vera.com/

[60] Synopsys, "OpenVera Language Reference Manual," 2002, http://www.open-vera.com

[61] Y. Hollander, M. Morley, and A. Noy, "The e Language: A Fresh Separation of Concerns," in *Proceedings of the Technology of Object-Oriented Languages and Systems*: IEEE Computer Society, 2001.

[62] The Aspect Oriented Software Development website, http://aosd.net/

[63] Verisity, Specman Elite, http://www.verisity.com/products/specman.html

[64] E. Meijer and J. Miller, "Technical Overview of the Common Language Runtime," http://research.microsoft.com/~emeijer/Papers/CLR.pdf

[65] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern Oriented Software Architecture: A System*
of Patterns. JohnWiley and Sons, 1996.

[66] J. Gough, *Compiling for the .NET Common Language Runtime*. Prentice Hall, 2001.

[67] J. Lapalme, *ESys.Net A New .Net Based System-Level Design Environment*. M. Sc. Thesis, Université de Montréal, 2003.

[68] M. Y. Vardi and P. Wolper, "Reasoning about infinite computations," *Inf. Comput.*, vol. 115, pp. 1-37, 1994.

[69] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata: structure versus automata*. Banff, Canada: Springer-Verlag New York, Inc., 1996.

[70] W. Thomas, "Automata on infinite objects," in *Handbook of theoretical computer science (vol. B): formal models and semantics*: MIT Press, 1990, pp. 133-191.

[71] J. R. Büchi, "On a Decision Method in Restricted Second Order Arithmetic," presented at International Congress of Logic, Method and Philosophy of Science, 1962.

[72] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*: Chapman & Hall, Ltd., 1995.

[73] D. Giannakopoulou and F. Lerda, "From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata," in *Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*: Springer-Verlag, 2002.

[74] D. Oddoux, *Utilisation des automates alternants pour un model-checking efficace des logiques temporelles linéraires*. Thèse de doctorat, Universitée Paris 7, Paris, 2003.

[75] P. Wolper, "The tableau method for temporal logic: An overview," *Logique et Analyse*, pp. 110-111:119-136, juin-sep. 1985.

[76] M. Fujita, H. Tanaka, and T. Moto-oka, "Logic Design Assistance with Temporal Logic," presented at Conference on Computer Hardware Description Languages and their Applications (CHDL), Amsterdam, 1985.

[77] G. Rosu and K. Havelund, "Rewriting-Based Techniques for Runtime Verification," *Automated Software Engineering*, vol. 12, pp. 151-197, 2005.

[78] Z. Manna and A. Pnueli, *Temporal verification of reactive systems: safety.* Springer-Verlag New York, Inc., 1995.

[79] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout, "Reasoning with Temporal Logic on Truncated Paths," presented at 15th International Conference on Computer Aided Verification, 2003.

[80] K. Havelund and G. Rosu, "Monitoring Programs Using Rewriting," in *Proceedings of the 16th IEEE international conference on Automated software engineering*: IEEE Computer Society, 2001.

[81] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Program monitoring with LTL in EAGLE," presented at Parallel and Distributed Processing Symposium, 2004.

[82] P. Thati and G. Rosu, "Monitoring Algorithms for Metric Temporal Logic Specifications," presented at Fourth Workshop on Runtime Verification, 2004.

[83] D. Giannakopoulou and K. Havelund, "Automata-based verification of temporal properties on running programs," presented at 16th Annual International Conference on Automated Software Engineering, 2001.

[84] R. Deutschmann, M. Fruth, H. Reichel, and H.-C. Reuss, "Trace Checking with Real-Time Specifications," presented at the 5th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004), 2004.

[85] B. Finkbeiner and H. Sipma, "Checking Finite Traces Using Alternating Automata," *Formal Methods in System Design*, vol. 24, pp. 101-127, 2004.

[86] P. Bellini, R. Mattolini, and P. Nesi, "Temporal logics for real-time system specification," *ACM Comput. Surv.*, vol. 32, pp. 12-42, 2000.

[87] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Systems*, vol. 2, pp. 255-299, 1990.

[88] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st international conference on Software engineering*. Los Angeles, California, United States: IEEE Computer Society Press, 1999.

[89] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[90] Specification Patterns, http://patterns.projects.cis.ksu.edu/

[91] L. Charest, E. M. Aboulhamid, and G. Bois, "Using Design Patterns for Type Unification and Introspection in SystemC," in *Proceedings of the International Workshop on System-on-Chip for Real-Time Applications*: IEEE Computer Society, 2004.

[92] ANTLR, ANother Tool for Language Recognition, http://www.antlr.org/

[93] T. J. Parr and R. W. Quong, "ANTLR: a predicated-LL(k) parser generator," *Software - Practice & Experience*, vol. 25, pp. 789-810, 1995.

[94] Scintilla - A free source code editing component, http://www.scintilla.org/

[95] Graphviz - Graph Visualization Software, http://www.graphviz.org/

[96] NLog - A .NET Logging Library, http://nlog.sourceforge.net/

[97] AMBA 2.0 Specification, http://www.arm.com/products/solutions/AMBA_Spec.html

[98] "AHB Lite Overview."

[99] AQtime - Atomated Profiling and Debugging, http://www.automatedqa.com/products/aqtime/

Annexe A Code source du modèle de producteur/consommateur

```
1. namespace ProdCons{
2. public class Producer:BaseModule{
3.   public inBool   ackIn;
4.   public outBool  readyOut;
5.   public outInt   dataOut;
6.   public Clock    clk;
7.
8.   private Event   acknowledgeEvent;
9.   private Random  random = new Random();
10.
11.    public Producer():base(){}
12.
13.    [ThreadProcess]
14.    private void Run(){
15.      readyOut.Value = false;
16.      while(true){
17.        uint delay = (uint)random.Next(3,8);
18.        Console.Error.WriteLine(
19.                    string.Format("Wait {0} cycles",delay));
20.        Wait(delay * clk.Period);
21.        int dataValue = random.Next(0, 5);
22.        dataOut.Value = dataValue;
23.        Wait(clk.posedge);
24.        readyOut.Value = true;
25.        Wait(clk.posedge);
26.        readyOut.Value = false;
27.
28.        Wait(acknowledgeEvent);
29.        while(ackIn.Value == true)
30.          Wait(acknowledgeEvent);}}
31.
32.    public override void BindingPhase(){
33.      if(ackIn is BoolSignal)
34.        acknowledgeEvent = ((BoolSignal)ackIn).sensitive;
35.      base.BindingPhase ();}
36. }
37.
38.
39. public class Consumer:BaseModule{
40.    public outBool  ackOut;
41.    public inBool   readyIn;
42.    public inInt    dataIn;
43.    public Clock    clk;
44.
45.    private Event dataReadyEvent;
46.    private Random random = new Random();
47.
48.    public Consumer():base(){}
```

```
49.
50.    [ThreadProcess]
51.    private void Run(){
52.      ackOut.Value = false;
53.      while(true){
54.        Wait(dataReadyEvent);
55.        while(readyIn.Value == true)
56.          Wait(dataReadyEvent);
57.        uint delay = (uint)random.Next(3,8);
58.        Wait(delay * clk.Period);
59.        int dataValue = dataIn.Value;
60.        Console.WriteLine(string.Format(
61.             "New Value :  {0}, t={1}", dataValue, CurrentTime));
62.        ackOut.Value = true;
63.        Wait(clk.posedge);
64.        ackOut.Value = false;}}
65.
66.    public override void BindingPhase(){
67.      if(readyIn is BoolSignal)
68.        dataReadyEvent = ((BoolSignal)readyIn).sensitive;
69.      base.BindingPhase ();}
70.  }
71.
72.  public class ProducerConsumerModel:SystemModel{
73.    private IntSignal data = new IntSignal();
74.    private BoolSignal acknowledge = new BoolSignal();
75.    private BoolSignal dataReady = new BoolSignal();
76.    private Clock clk = new Clock(10);
77.
78.    private Consumer consumer = new Consumer();
79.    private Producer producer = new Producer();
80.
81.    public ProducerConsumerModel(ISystemManager manager,
82.                                 string name):base(manager, name){
83.      producer.clk = clk;
84.      consumer.clk = clk;
85.      producer.ackIn = acknowledge;
86.      consumer.ackOut = acknowledge;
87.      producer.readyOut = dataReady;
88.      consumer.readyIn = dataReady;
89.      producer.dataOut = data;
90.      consumer.dataIn = data;}
91.
92.    public static void Main(string [] args){
93.      Simulator sim = new Simulator();
94.      ProducerConsumerModel model =
95.              new ProducerConsumerModel(sim, "PCModel");
96.      sim.Run(1000);
97.      System.Console.ReadLine();}
98.  }
99.  }
```

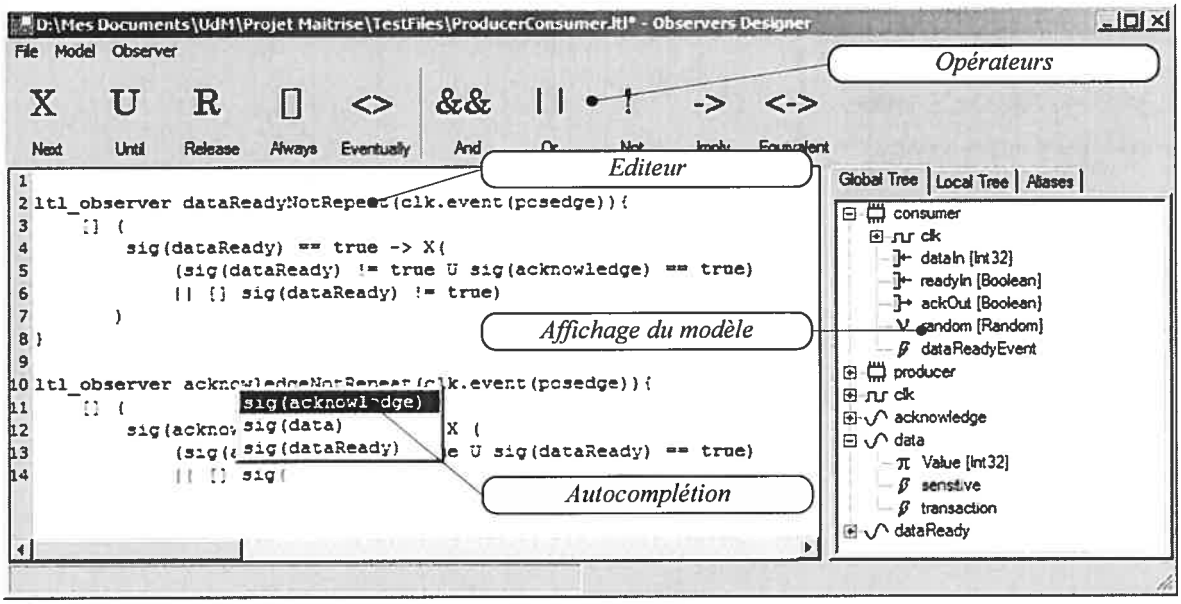
Annexe B Grammaire du langage de spécification de propriétés

```
1.  //TOKENS
2.
3.  tokens {
4.    INIT="init"; ALIASES="aliases"; IMPORT="import";
5.    MODEL="model"; LTL_OBSERV="ltl_observer";
6.    MTL_OBSERV="rtltl_observer"; OP_UNT="U";
7.    OP_REL="R"; OP_NEX="X"; TRUE="true";
8.    FALSE="false"; INTEGER; FLOAT;}
9.  LPAREN: '(';
10. RPAREN: ')';
11. LCURLY: '{';
12. RCURLY: '}';
13. LSBRACKET: '[';
14. RSBRACKET: ']';
15. COM: ',';
16. DOT: '.';
17. AT: '@';
18. OP_EQ: "==";
19. OP_NEQ: "!=";
20. OP_INF: '<';
21. OP_SUP: '>';
22. OP_IEQ: "<=";
23. OP_SEQ: ">=";
24. OP_NOT: '!';
25. OP_EQU: "<->";
26. OP_IMP: "->";
27. OP_AND: "&&";
28. OP_OR: "||";
29. OP_ALW: "[]";
30. OP_EVE: "<>";
31. ID:('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
32. WS : ( ' ' | '\t' | '\n' { newline(); } | '\r' )+;
33. Number :
34.   ('-')? ('0'..'9')+
35.     ( '.' ('0'..'9')*     { $setType(FLOAT); }
36.     | /* empty */     { $setType(INTEGER); }
37.     );
38. SIG : "sig(";
39. VAR : "var(";
40. CH  : "ch(";
41. CLK : "clk(";
42. PORT: "port(";
43. PROP: "prop(";
44. EVNT: "event(";
45. STRING_LITERAL :
46.   '"' ('a'..'z'|'A'..'Z'|'_'|'0'..'9'|' '|':'|'\\'|'.')* '"';
47. ALIAS: '#' ID;
48. AFFECT: '=';
```

```
49. SL_COMMENT: // Single-line comments
50.   "//" (~('\n'|'\r'))* ('\n'|'\r'('\n')?);
51. ML_COMMENT: // multiple-line comments
52.   : "/*" ('*' | '\r' '\n' | '\r' | '\n' | ~('*'|'\n'|'\r'))* "*/";
53.
54. //GRAMMAIRE - *Sans les blocs d'actions*
55. top:
56.   init aliases (comment | ltl_observer | mtl_observer)+ EOF;
57.
58. comment: SL_COMMENT | ML_COMMENT;
59.
60. // HEADER
61. init: INIT LCURLY initContent RCURLY;
62.
63. initContent: modelDefinition;
64.
65. modelDefinition: MODEL path AT STRING_LITERAL;
66.
67. aliases: ALIASES LCURLY aliasesContent RCURLY;
68.
69. aliasesContent: (aliasDefinition)*;
70.
71. aliasDefinition: ALIAS AFFECT path;
72.
73. //LTL OBSERVER
74. ltl_observer:
75.   LTL_OBSERV ID LPAREN signal RPAREN LCURLY expr RCURLY;
76.
77. binaryOp:
78.   OP_OR | OP_AND | OP_EQU | OP_IMP | OP_UNT | OP_REL;
79.
80. expr: mexpr (binaryOp mexpr)*;
81.
82. unaryOp: OP_NOT | OP_NEX | OP_ALW | OP_EVE;
83.
84. mexpr:
85.     atom | TRUE | FALSE
86.   | unaryOp mexpr | LPAREN expr RPAREN;
87.
88. //MTL OBSERVER
89. mtl_observer:
90.   MTL_OBSERV ID LPAREN signal RPAREN LCURLY mtl_expr RCURLY;
91.
92. mtl_expr: mtl_mexpr (mtl_binOp mtl_mexpr)*;
93.
94. mtl_binOp:
95.     OP_OR  | OP_AND | OP_IMP | OP_EQU
96.   | OP_UNT | OP_REL | OP_WUNT
97.   | (OP_UNT LSBRACKET)  =>
98.                       OP_UNT  LSBRACKET INTEGER COM INTEGER RSBRACKET
99.   | (OP_REL LSBRACKET)  =>
100.                        OP_REL  LSBRACKET INTEGER COM INTEGER RSBRACKET
101.    | (OP_WUNT LSBRACKET) =>
102.                       OP_WUNT LSBRACKET INTEGER COM INTEGER RSBRACKET;
103. mtl_mexpr:
104.      atom | TRUE | FALSE
105.    | OP_NOT mtl_mexpr
```

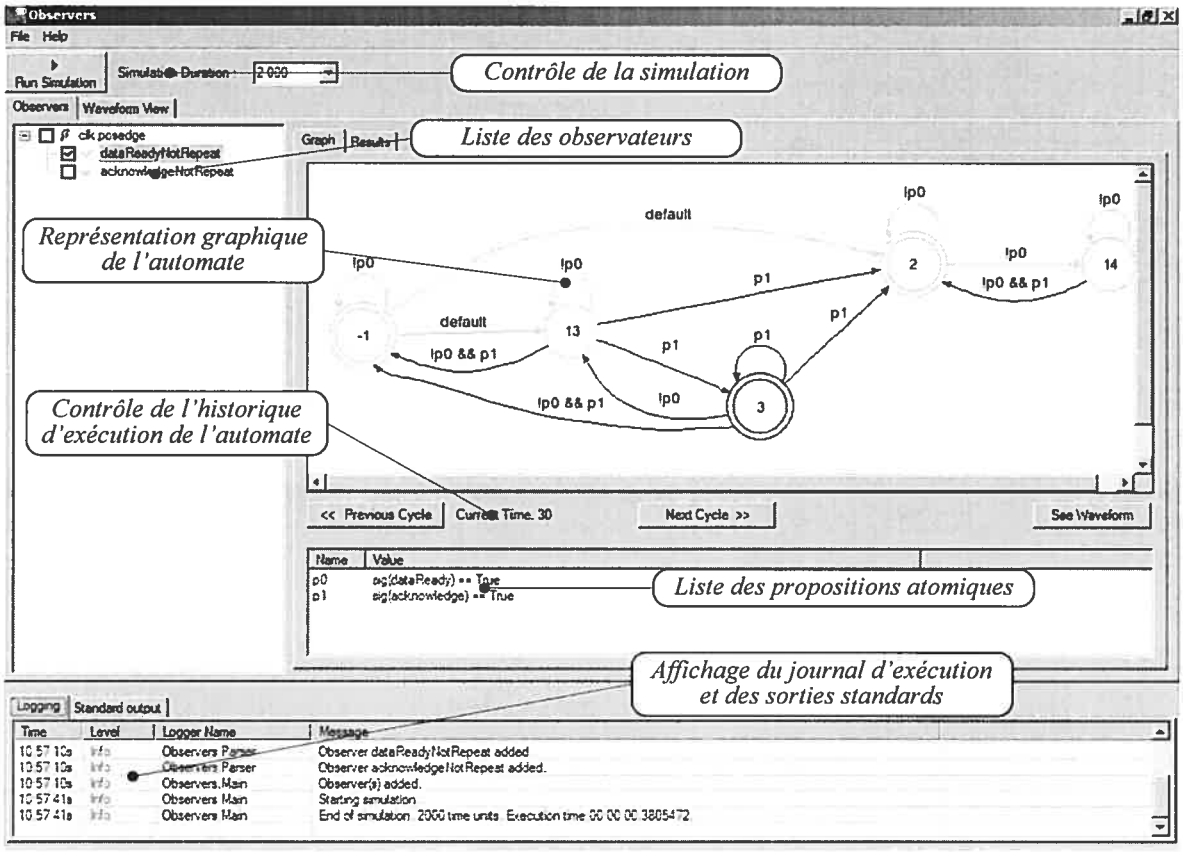
```
106.     | OP_NEX mtl_mexpr
107.     | (OP_NEX LSBRACKET) =>
108.                         OP_NEX LSBRACKET INTEGER RSBRACKET mtl_mexpr
109.     | OP_ALW mtl_mexpr
110.     | (OP_ALW LSBRACKET) =>
111.               OP_ALW LSBRACKET INTEGER COM INTEGER RSBRACKET mtl_mexpr
112.     | OP_EVE mtl_mexpr
113.     | (OP_EVE LSBRACKET) =>
114.               OP_EVE LSBRACKET INTEGER COM INTEGER RSBRACKET mtl_mexpr
115.     | LPAREN mtl_expr RPAREN;
116.
117.   //SIGNAL
118.   ////////////////
119.   fieldName:
120.       ID | INIT | ALIASES | IMPORT | MODEL | LTL_OBSERV
121.     | MTL_OBSERV | OP_UNT | OP_REL | OP_NEX;
122.   signal:
123.       (aliasPath DOT SIG) => aliasPath DOT SIG fieldName RPAREN
124.     | SIG fieldName RPAREN
125.     | (aliasPath DOT VAR) => aliasPath DOT VAR fieldName RPAREN
126.     | VAR fieldName RPAREN
127.     | (aliasPath DOT CLK) => aliasPath DOT CLK fieldName RPAREN
128.     | CLK fieldName RPAREN
129.     | (aliasPath DOT PORT) => aliasPath DOT PORT fieldName RPAREN
130.     | PORT fieldName RPAREN
131.     | (aliasPath DOT CH) =>
132.                   aliasPath DOT CH fieldName COM fieldName RPAREN
133.     | CH fieldName COM fieldName RPAREN
134.     | (aliasPath DOT PROP) => aliasPath DOT PROP fieldName RPAREN
135.     | PROP fieldName RPAREN
136.     | (aliasPath DOT EVNT) => aliasPath DOT EVNT fieldName RPAREN
137.     | EVNT fieldName RPAREN;
138.
139.   path:
140.     ID (DOT ID)*;
141.
142.   aliasPath:
143.     ALIAS (DOT ID)* | path;
144.
145.
146.   //Atomic Propositions
147.   ////////////////////////
148.   atom:
149.     (signal comparator) => signal comparator (signal | val);
150.
151.   comparator:
152.       OP_EQ  | OP_NEQ | OP_INF
153.     | OP_SUP | OP_SEQ | OP_IEQ;
154.
155.   val:
156.       FLOAT | INTEGER | TRUE
157.     | FALSE | STRING_LITERAL;
```

Annexe C Capture d'écran de l'éditeur



Annexe D Captures d'écran de l'application de simulation

1 – Visualisation des observateurs



2 – Visualisation des signaux