REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ EL-HADJ LAKHDAR – BATNA
FACULTÉ DES SCIENCES
DEPARTEMENT D'INFORMATIQUE

THESE

Présentée par

REBAIAIA Mohamed-Larbi

pour obtenir le titre de

DOCTEUR en Sciences

Spécialité

INFORMATIQUE

Sujet de la thèse :

Spécification et Vérification des Systèmes Critiques :

Extension de l'Environnement VALID pour la Prise en Charge du Temps Réel

---

Soutenue publiquement le 16 février 2011 devant le jury composé de :

M. Noureddine BOUGHECHEL	Professeur	Président
M. Mohamed BENMOHAMED	Professeur	Directeur de Thèse
M. Djamel-Eddine SAIDOUNI	Professeur	Examinateur
M. Allaoua CHAOUI	Maître de conférences	Examinateur
M. Azzedine BILAMI	Maître de conférences	Examinateur
M. Brahim BELATAR	Maître de conférences	Examinateur

A la mémoire de ma mère et de mon père.
A Saliha, ma femme.
A mes enfants : Aniss, Islem, Rayan et Assil.

REMERCIEMENTS

Les travaux exposés dans cette thèse ont été réalisés grâce aux fruits d'un long travail de recherche réalisé à l'Université de Batna, sous la direction de M. Mohamed Benmohamed. Les premières étapes du projet de recherche de la thèse prenaient naissance des idées amorcées dans deux projets de recherche financés par le MESR : l'Intelligence Artificielle Distribuées (Multi-Agents en 1997-1998) et la *Bisimulation* des Systèmes temps-réel (1999-2000) dont je suis l'initiateur et le responsable du groupe. Plus tard, l'incorporation de la notion même de *spécification* et de *vérification* des systèmes réactifs ainsi que l'extension du système *VALID* pour couvrir l'aspect temps-réel et la description *UML/OCL* pour le temps réel a été amorcée grâce au concours de mon ami, Ammar Attoui, professeur au laboratoire CESALP/ILP à l'Université d'Annecy que je remercie énormément.
Cette thèse a été déposée le 4 mars 2005 pour être défendue devant un jury. Mais, pour des circonstances diverses elle a été reculée jusqu'au jour de la soutenance.
Je tiens à remercier tout particulièrement, Mr. Mohamed Benmohamed, maître de conférences au Département d'Informatique de l'université de Constantine, qui a encadré cette thèse. Je le remercie aussi pour ses conseils, pour sa disponibilité et pour la confiance qu'il m'a accordée durant toute la période de recherche. Je voudrai aussi lui témoigner toute ma reconnaissance pour m'avoir laissé une totale liberté d'action.
Je voudrais aussi remercier M. Boughechel Nouredine, Professeur et doyen de la faculté des sciences de l'ingénieur de l'université El-Hadj Lakhdar de Batna, d'avoir accepté la présidence du jury. Je remercie aussi mes collègues et amis Belatar Brahim et Bilami Azzedine, Maîtres de Conférences au département d'informatique de l'Université de Batna, d'avoir accepté la d'être membres du jury.
Je tiens aussi à exprimer toute ma gratitude à Messieurs Djamel-Eddine Saidouni et Alaoua Chaoui, respectueusement Professeur et Maître de Conférence au département d'informatique de l'université de Constantine d'avoir accepté la tache d'être rapporteurs de ma thèse.
Aussi, je témoigne toute ma sympathie à Mr. Zidani Majid et Mr. Zidat Samir responsables du département d'informatique de Batna pour avoir fait l'impossible pour que la soutenance ait lieu dans des délais raisonnables.
Je voudrai aussi remercier toutes les personnes avec qui, j'ai eu des discussions scientifiques, et qui d'une manière ou d'une autre m'ont été d'une utilité quelconque. Je citerai plus particulièrement mes co-auteurs dans plusieurs publications, Mr Jihad Mohamad Jaam et Mr Ahmad Hasnah, Professeurs au département d'Informatique de l'Université du Qatar, Mr Robert Valette, Professeur et Directeur de Recherche au LAAS CNRS de Toulouse, et surtout Paul Pettersson professeur à l'Université Uppsala (Suède) de m'avoir fournit le système de vérification UPPAAL. Que M. S. Eker et M. P. Olveczky assistants Professeurs et membres de l'équipe du Professeur Meseguer au laboratoire CSL/SRI à l'Université de Stanford (USA), reçoivent toute ma gratitude, pour les conseils prodigués et le programme «Maude Real-time » qui m'a été envoyé, et surtout pour toutes les bonnes choses qu'ils ont accompli pour promouvoir le système MAUDE.
Je remercie énormément mes étudiants fin de cycle Ingénieur (Informatique) qui m'ont aidé dans l'écriture du programme ROBDD, pour la programmation des algorithmes génétiques et du module de simulation.
J'ose aussi témoigner de l'implication bénéfique dans la préparation de cette thèse de certaines personnes. Je citerai entre autres, ma femme pour m'avoir encouragé à reprendre la recherche après pratiquement une dizaine d'année de vide scientifique. Je la remercie aussi pour le soin dont elle a fait preuve lors de la préparation de ce document et les corrections qu'elle a pu apporter.

Résumé : *La technologie des systèmes mixtes matériels et logiciels connaît de nos jours une révolution sans pareille dans tous les domaines technologiques. Le fait de mélanger matériel et logiciel, ceci a donné naissance à des systèmes très complexes. La conception de tels systèmes, nécessite l'intervention de ressources humaine, financière et technologique très importante. Pour veiller au bon fonctionnement de leurs conceptions, ces systèmes doivent être soumis à des processus de validation très rigoureux. Une telle tâche n'est point évidente vue l'accroissement rapide de la complexité et de l'hétérogénéité des nouvelles architectures (matériel/logiciel). Aussi correcte que possible, la conception doit tenir compte de certaines propriétés dites critiques, qui sans leur présence risque d'engendrer une situation de catastrophe et même de ternir l'image de la compagnie.*

Dans ce travail, nous nous sommes intéressées à la mise en place d'un environnement intégré pour le développement des systèmes réactifs temps-réel. A cet effet, nous avons proposé plusieurs solutions pour la modélisation, la spécification et la vérification de plateformes matériel/logiciel. Dans un premier temps, nous avons utilisé une technique simple et efficace pour implémenter les diagrammes de décision binaire utilisés pour la représentation et la vérification des réseaux booléens symbolisant les systèmes réactifs. Nous avons ensuite, montré que l'axiomatisation des logiques temporelles linéaire LTL et arborescente CTL, permet de cerner de peu le problème de la complexité combinatoire spacio-temporelle. Les automates temporisés et la logique TCTL ont permis de mettre en place un modèle très utile pour la spécification et la vérification des systèmes temps-réels.
En se basant d'un coté sur la logique de réécriture comme modèle de spécification algébrique et philosophie universelle de raisonnement formel, et du système Maude comme outil de programmation et d'exécution de ces spécifications, nous avons intégré le tout en tant que noyau de programmation de l'environnement en question et sur lequel un ensemble de modules intégrés et spécialisés sont venus se greffer. Nous avons d'autant plus compris durant cette recherche, que la description UML et son extension UML/OCL et UML-RT, constituent pour le moment les meilleurs moyens pour la modélisation "semi-formelle" des systèmes réactifs temps-réel. Finalement, nous avons montré l'aisance même de l'utilisation de cet environnement à travers plusieurs exemples et benchmarks connus.

Mots-clés : *Spécification, Vérification, ROBDD, Logique de Réécriture, Maude, UML/OCL, UML-RT, Méthodes Formelles, Systèmes Réactifs Temps-réel.*

Abstract : *With the increasing emergence of hardware/software distributed systems, designers need in some ways efficient methods and tools to improve the safeness of such systems and to reduce drastically their design engineering time. In general, some traditional methods as simulation and testing are used to detect errors and bugs, but they are inadequate to certify the correctness of complex systems. To accelerate the design and to avoid distributed systems malfunctions, new mathematical-based techniques have been used to improve the deficiency of the old methods. During the last two decades, Model-checking and theorems proving have been the major research verification fields intensively used to check the design correctness of some big projects (NASA lance rockets, Pentium chips, etc.) In fact, despite the generalization of formal methods, some problems remain in suspense in producing coherent specification fully integrated semantics and avoiding the size complexity of designs.*

In this Thesis, we present a hierarchical approach and an open environment to describe and to verify distributed and concurrent systems. The system currently uses UML and extension UML/OCL and UML-RT notations and provides rewriting logic, model checking, theorem proving, and simulation techniques. The Model checking algorithm developed in this research work is based on the axiomatization of LTL linear temporal model and tree temporal logic CTL. We have used temporal automata and TCTL to describe formal behavioral description and to verify timed-based properties. Our experience has shown that combining Temporal Logics and a theoretical well based techniques as the Rewriting logic embodied in the core of the fastest formal system (Maude), can improve the verification of the correctness of large circuits and critical software, and can decrease the time-consuming to perform a solution. Such good solutions are obtained due to the strategy rules incorporated in Maude system which are based on the recent theoretic researches by the rewriting community. When one would like to analyze the results depicted in the benchmark experiments results, he will be surprised because due to the execution CPU time which is, in general insignificant comparing with other verification commercial tools.

Keywords : *Specification, Verification, ROBDD, Rewriting Logic, Maude Language, UML/OCL, UML-RT, Formal Methods, Reactive Real-Time Systems.*

SOMMAIRE

CHAPITRE 7 : SPECIFICATION DES SYSTEMES TEMPS REEL DANS LE CADRE DE LA LOGIQUE DE REECRITURE

CHAPITRE 8 : PROPOSITION D'UN ENVIRONNEMENT POUR LA MODELISATION DES SYSTEMES REACTIFS

CHAPITRE 9 : IMPLANTATION

TABLE DES FIGURES

SOMMAIRE DES TABLES

CHAPITRE 1

Introduction et Motivation

1.1 Problématique

La technologie des systèmes mixtes matériels et logiciels en anglais hardware-software (H/S) connaît de nos jours une révolution sans pareille dans tous les domaines technologiques. Le fait de mélanger matériel et logiciel, a donné naissance à des systèmes très complexes (plusieurs millions de composants électroniques et des centaines de milliers de lignes de code et même plus). La conception de tels systèmes, nécessite l'intervention de ressource humaine, financière et technologique très importante. Aussi important soit-elle, leur modélisation, avant d'être fabriquée permet aux ingénieurs concepteurs de réduire le nombre d'erreurs de conception, connue sous l'appellation de ***bugs***.

Pour veiller au bon fonctionnement de leurs conceptions, ces systèmes doivent être soumis à des processus de validation très rigoureux. Une telle tâche n'est point évidente vue l'accroissement rapide de la complexité et de l'hétérogénéité des nouvelles architectures H/S [Bal97]. Aussi correcte que po.ssible, la conception doit tenir compte de certaines propriétés dites critiques, qui sans leur présence risque d'engendrer une situation de catastrophe sur le plan financier, humain et même de ternir l'image de la compagnie. En général, ces propriétés se répartissent en deux classes : propriétés de performance (durée du cycle de conception par exemple) et propriétés fonctionnelles (problèmes d'interblocage par exemple.)

La conception de composantes H/S d'un système intégré s'effectuait traditionnellement de manière séparée, c'est-à-dire que la partie logicielle obtenue, était exécutée sur le prototype de la partie matérielle. Si les contraintes de la spécification requises pour le système ne sont pas satisfaites par le design, le processus subit des modifications puis il est réitéré dans l'espoir d'obtenir le bon prototype. Cette technique de validation s'avère être très coûteuse en temps et en coût de réalisation.

Pour parer à de telles inconvenances, plusieurs idées ont émané de la communauté scientifique pour l'élaboration de nouvelles techniques de vérification et d'environnements de co-design groupés sous l'appellation d'outils et techniques CAD (System-level Computer Aid Design) qui, en réalité, sont des plate-formes de modélisation, de validation et de synthèse, surtout de circuits VLSI (Very Large Scale Integration Circuits) [Bry91]. Comme la majeure partie des VLSI sont essentiellement des processeurs, la conception de la partie matérielle et de la partie logicielle est un problème très critique et complexe. Aussi, compte tenu d'une compétitivité très prononcée dans un marché très ouvert tout produit conçu dans des délais moindres devient la clé du succès de tout projet.

Il est bien connu que les architectures H/S s'apparentent en général avec les systèmes enfouis (embedded), critiques et temps-réel, nécessitant qu'au moment de la phase de validation, une attention très particulière mettant en place des mécanismes de vérification tendant à éviter toute situation malencontreuse.

Au cours des dernières années, la complexité de ces systèmes (Embedded H/S Systems)[Ern98] n'a cessé de se développer. Les progrès technologiques favorables, notamment en terme de finesse de gravure, ont permis de diminuer considérablement leur taille. Cette miniaturisation entraînait incontestablement d'énormes avantages, tant au niveau de leur consommation (électronique grand public : téléphones cellulaires, distributeurs ATM...), de leur puissance et de leur rapidité.

Lors de la conception des systèmes enfouis, la phase de vérification est l'étape la plus longue. Elle consomme jusqu'à 70 % du temps de conception. La notion de vérification elle-même, date des années soixante avec les travaux de Dijkstra et Floyd. Celle-ci, s'est étalée sur plus de trois décennies de découvertes théoriques extraordinaires qui a vu naître carrément une nouvelle science dite « *vérification formelle* », qui a vite déclassé certaines techniques traditionnelles telles que la simulation, le *testing* et les anciennes méthodes de validation. Une prise de conscience de l'importance de la vérification en tant que telle a émergée lors de la médiatisation de l'erreur de conception du Pentium d'Intel en 1994 [Pra95]. La perte a été estimée à plus de 400 millions de dollars. Ce cas n'est pas isolé, puisque de nombreux exemples d'erreurs de conception ont été médiatisées notamment le cas de l'accident lors du premier vol du lanceur européen ARIANE-5 [Ari96, RSS96] en 1996 (Figure 1.1) qui a été suivi de la vérification du système complet par des

techniques d'interprétation abstraite, ce qui a permis la suppression de certaines erreurs parmi elles, celles ayant entraîné la destruction de la fusée.

Actuellement, les circuits et gros logiciels deviennent de plus en plus élaborées rendant le processus de conception très complexe susceptible d'engendrer un nombre maximum d'erreurs et ceci à tous les niveaux du processus de conception, depuis l'analyse du cahier des charges jusqu'à la fabrication du circuit lui-même [CCL+99]. Il devient donc de plus en plus judicieux de disposer de moyens théoriques et pratiques permettant de s'assurer de la validité d'un système avant d'en arriver à la phase ultime de fabrication et de sa mise en service. Il faudrait aussi avant cela de disposer d'outils sinon d'une méthodologie de modélisation et de conception basée sur les nouvelles idées tels que UML [OMG99a, OMG01], SDL [ITU01], MSC [ITU-T99, RGG96], Statecharts [Har87] ou autre, pourvues d'une bonne représentativité soit de rigueur.

En général, la nécessité de recourir aux techniques de vérification basées sur les méthodes formelles s'impose de nos jours. Il s'agit donc de produire un modèle qui est une représentation mathématique, graphique (réseau, automate) ou autres (logique, Statecharts, …) d'un système H/S de façon à ce que son comportement se confonde exactement avec sa spécification. Dans la réalité, au moment de la modélisation certaines caractéristiques doivent être clairement définies compte tenu de leur importance pour départager les conceptions, à savoir : l'aspect séquentiel ou concurrent (peut-être bien les deux à la fois), l'aspect temporel et temporisé (peut-être bien les deux à la fois), les aspects critiques, déterministes ou non-déterministes, discrets ou continus (peut-être bien les deux à la fois appelés communément par le terme hybride), etc.…

De nos jours, un grand nombre d'outils de vérification académiques et commerciaux ont été écrit. Nous citons à titre d'exemples *CheckOff-core technology* développé par Siemens, *RuleBase-core technology* [BBL+97], développé au CMU (Carnegie Mellon University (USA)). Quelques universités ont aussi développé les leurs, tels que SMV [McM99], écrit par McMillan au CMU d'ailleurs basé essentiellement sur un model checking symbolique. L'Université de Montréal possède aussi le sien, appelé MDG (Multiway Decision Graph) [ZB96], qui supporte la vérification de propriétés (*Property checking)* et l'*equivalence checking*. Le plus populaire parmi ces systèmes de vérification est VIS (Verification Interacting with Synthesis) développé à l'université de Berkeley California (USA). VIS est constitué de programmes de vérification, de simulation et de synthèse des systèmes sous forme de machines d'états finis [Bra95]. Il utilise le langage de description *Verilog* et permet la vérification par les model checking et par l'équivalence-checking des systèmes séquentiels et combinatoires, la simulation *« cycle-based »* et la synthèse hiérarchique. Nous citons aussi Uppaal [UPPAAL] développé dans le cadre d'un travail de collaboration entre l'Université Uppsala en Suède et Aalborg en Norvège sous la direction du professeur Wang Yi. UPPAAL est un système très efficace qui permet de vérifier les systèmes temps-réel utilisant des algorithmes de résolution de contraintes et d'autres techniques tels que les BDD (Binary Decision Diagrams) [Bry86]. Il est disponible et téléchargeable au http://www.uppaal.com. D'autres outils de vérification sont disponibles sur différents sites et chacun d'eux possède ces propres caractéristiques. Nous reviendrons un peu plus tard pour citer quelques-uns parmi eux dans la partie *Etat de l'Art*.

1.2 Contexte de Nos Travaux

Nos travaux de recherche, rentrent dans le cadre de la prospection et de l'utilisation des modèles de spécification et de vérification des systèmes informatiques réactifs (H/S). Nous utilisons une approche formelle basée sur la logique de réécriture [Mes90, Mes92, Mes98, Mes99, Mes00], du système MAUDE [CDS+00, CDE+99] et de l'environnement VALID [AM94, AS94, AS96]. La démarche de spécification utilise un langage de description du type UML et étendue à UML-RT (UML Real-Time). Les modèles de vérification utilisent deux types de logiques temporelles, l'une linéaire (LTL-PTL)[Pnu77] et l'autre arborescente (CTL) [eme96]. Nous tenons à citer que certains de nos travaux ont contribué à montrer le bien fondé de l'utilisation de certaines méthodes hybrides basées sur les algorithmes génétiques et les réseaux de neurones pour la description et la vérification des systèmes critiques, loin de toute ambiguïté créée par

l'écriture de spécifications formelles très lourdes, bénéficiant de méthodes simples à programmer et à exécuter sur des plate-formes maintenant très rapides et disponibles sur le Web [RBJ+03c, RJH03b, Reb03b, RB02, RBJ+03b].

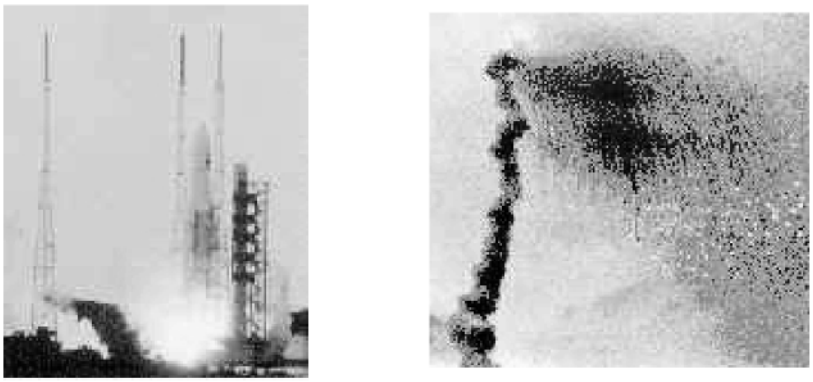


Figure 1.1 : Lancement (à gauche) et explosion (à droite) du lanceur européen Ariane-5 le 4 juin, 1996; 36 secondes après son lancement.

Ce projet (objectifs de la thèse), influencé de façon globale par le projet VALID développé par l'équipe de A. Attoui, Professeur à Annecy, a été batisé VALID 2 [RJ04], pour ne pas sortir de la méthodologie suivie par VALID. Néanmoins une grande différence subsiste entre les deux projets.

Le projet VALID 2 dixit VALID, permet la réalisation d'un environnement intégré pour le développement des systèmes réactifs temps réel complexes. VALID 2, quoi que pour le moment est en phase d'expérimentation et d'assemblage permet de couvrir les phases de développement, à savoir, l'analyse, la spécification, la conception, la vérification et la validation des systèmes. La philosophie de VALID 2 tire profit comme il a été dit de l'expérience accumulée par VALID qui d'ailleurs tourne autour de trois architectures indépendantes, à savoir VALID (le noyau), ANALD (outil d'aide au diagnostic en temps réel et/ou en différé) et RESOLVE (utilisé pour la vérification des différentes contraintes temporelles, topologique et de performance, mais aussi pour l'auto-apprentissage de la recherche de stratégies pour pallier à certains événements sous estimés pendant la phase de développement et qui risquent de mettre en cause la cohérence globale du fonctionnement du système). VALID 2 est constitué d'un noyau qui est VALID, remodelé par l'incorporation d'une interface graphique mettant en place un système de modélisation et de spécification de systèmes concurrents écrit en Java. Cette application génère deux types de codes, l'un fournit un code Java, utilisé pour l'exécution d'une simulation et de son animation en directe, et l'autre un code formel écrit en MAUDE. VALID 2, incorpore aussi un résolveur de contraintes basées sur la logique de réécriture, générant ainsi un code exécutable efficace et très rapide. Aussi un vérifieur de spécification écrit en MAUDE a été adjoint au tout. Ce vérifieur appelé aussi model checking est un puissant outils qui permet de déboguer et de vérifier des plate-formes *logiciels et matériels*, il utilise deux types de logiques temporelles, l'une pour les modèles linéaires (LTL-PTL-PLTL) et l'autre pour les modèles arborescents (CTL). Une troisième logique qui est particulière aux systèmes temps réel a été définie et écrite en MAUDE (Timed-CTL). Indépendamment à cela, nous avons développé, un système de représentation des plate-formes Hardware/Software sous forme de ROBDD pour leur simplification d'un côté, mais aussi pour la vérification de l'équivalence entre une spécification et son implémentation. Le système ROBDD est aussi utilisé comme module SAT(isfaction) de contraintes d'une façon abrégée mais il sera considéré comme un sous module récursif lors de l'utilisation des algorithmes de model checking dus à Emerson et clarke mais pour la vérification de systèmes et qui est basée sur le CTL et le mu-calcul.

Le système VALID 2 à été enrichi, d'un module de vérification écrit en Java et qui s'articule autour de la résolution de contraintes temporelles qui se présentent sous forme de système linéaire. Celui-ci (le module) utilise d'un coté les algorithmes génétiques et les réseaux de neurones. Ces deux applications ne seront pas présentées par la suite, le lecteur est invité à consulter les articles [RBJ+03b, Reb03b]

pour mieux se situer. D'autres modules ont été écrits tels que la résolution des systèmes logiques par la déduction naturelle, les systèmes de Gentzen, le réseau de Petri, la logique linéaire, l'implémentation des bisimulations etc. Ces modules ne seront pas présentés, du fait de la lourdeur engendrée par l'écrit de cette thèse. Notons qu'en fin du chapitre 9, nous donnons un tableau exhaustif de plusieurs benchmark qui ont été vérifié par le model-checker. Nous notons aussi dans le chapitre 9, quelques problèmes test de systèmes réactifs qui seront présentés pour faciliter la compréhension de la notion de formalisation, de spécification et de vérification.

La figure 1.2, nous donne un aperçu général sur les différents composants du projet de cette thèse.

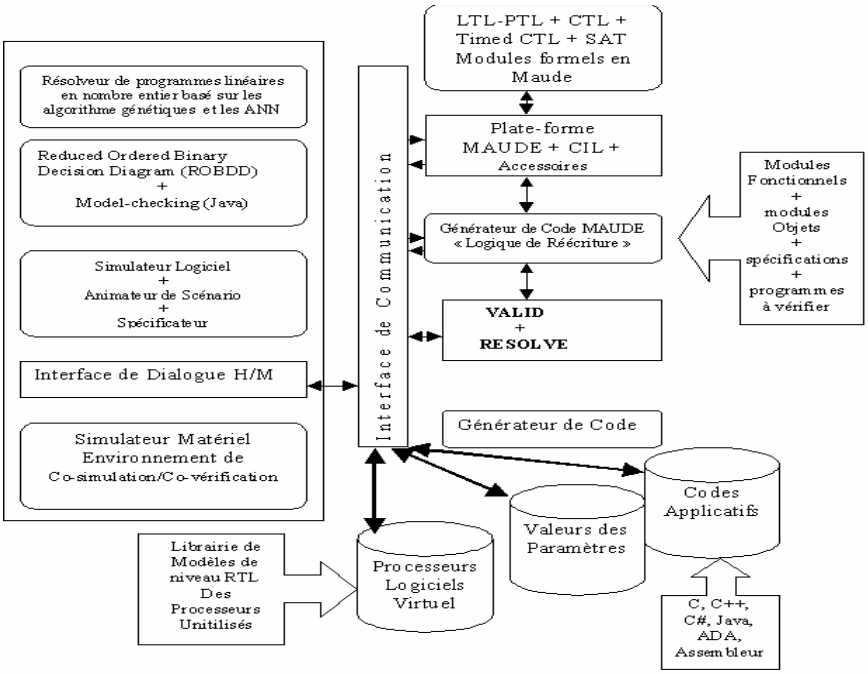


Figure 1.2 : Architecture générale du projet

1.3 Plan de la thèse

Donnons à présent le plan de ce document. Mis à part le premier chapitre qui introduit la problématique, la motivation et le contexte de nos travaux de façon presque succincte, voici brièvement les centres d'intérêts de chacun des chapitres restants :

- Le chapitre II dresse un plan détaillé de l'état de l'art du contexte de notre thèse, à savoir les méthodes formelles, la spécification, la vérification, les outils de vérification temporelles-temporisées etc.
- Le chapitre III introduit quelques définitions de base sujettes à extension, mais utiles pour la présentation de la problématique. Nous citons les systèmes réactifs et temps réel, les multi-ensembles et la bisimulation.
- Le chapitre IV présente les méthodes de représentation des systèmes matériels et logiciel (H/W) sous forme symbolique par un diagramme binaire de décision (BDD) et dresse des algorithmes pour sa réduction et sa transformation en diagramme binaire de décision ordonné et réduit (ROBDD). Les ROBDD sont très utiles dans notre cas puisqu'ils constituent un noyau d'une grande importance pour la vérification des systèmes et sont d'un apport théorique et pratique considérable pour l'implantation des algorithmes d'Emerson pour le CTL et le mu-calcul.
- Le chapitre V définie méticuleusement la logique temporelle linéaire (LTL) et la logique arborescente temporelle (CTL). Beaucoup d'exemples ont été proposés pour bien expliquer

leur signification mais aussi leur force pour vérifier les différentes facettes de systèmes complexes. Nous avons présenté des algorithmes très puissants sous trois formes : simple, c'est à dire algorithmique; axiomatique mais aussi en se plaçant dans un cadre bien précis à savoir les automates de Buchi. Cette partie est considérée comme le cœur même des méthodes de vérification. Ce chapitre traite aussi les problèmes des systèmes temps réel définis dans une forme abstraite qui est l'automate temporisé, mais aussi comme des systèmes de transitions temporisées. Dans cette partie nous introduisons une logique très puissante (TCTL) pour vérifier les systèmes temporisés.

- Le chapitre VI introduit le concept théorique de la réécriture, de la logique de réécriture et du système MAUDE. Nous verrons plus en détail la force d'expression de la logique de réécriture et la puissance du système MAUDE pour la spécification, la vérification des systèmes complexes.
- Le chapitre VII reprend une partie des définitions introduites dans le chapitre VI afin de tracer dans ce même cadre un chemin correct pour la spécification des systèmes temps réel.
- Dans le chapitre VIII nous proposons un environnement pour la modélisation des systèmes réactifs, qui d'un coté utilise un langage particulier et normalisé pour la description graphique des systèmes complexes : le langage UML. L'adjonction du protocole de description de contraintes appelé OCL est d'un apport considérable ce qui enrichi UML d'un moyen en soit peu formel. Le temps réel est modélisé particulièrement en utilisant une méthode singulière les machines d'états etc. Nous proposons une autre façon d'appréhender le temps réel autrement, juste en utilisant UML et OCL, un exemple très intéressant est même dressé.
- Enfin, le dernier chapitre explique à la lettre presque tous les éléments de notre système (environnement) et nous dressons un tableau récapitulatif de la vérification de certains benchmarks.

CHAPITRE 2

Etat de l ’ Art

2.1 Introduction

Dans ce chapitre nous définissons l'état de l'art d'une manière très sélective pour montrer le bien-fondé des méthodes de vérification, leur utilité mais aussi leur pouvoir d'abstraction sur les architectures matérielle et logiciel.

Tout d'abord nous présentons les méthodes formelles en tant qu'outils mathématiques très forts pour modéliser, spécifier et vérifier les systèmes critiques temps réels. Nous donnons aussi des définitions qui pour le moment sont générales sur la manière de prendre en charge un système à vérifier, de départager sa spécification formelle de son implémentation et de constituer une base de donnée de ses propriétés les plus cruciales.

Dans la suite de ce chapitre nous définissons le processus de conception dans un cadre unifié que ce soit sur une architecture matériel ou bien logicielle, le processus dans les deux cas diffère de très peu.

Nous introduisons aussi différentes techniques de vérification et nous mettons plus de détail sur le model-checking, puisque c'est ce model qui sera uilisée dans la suite de notre travail. A la fin de ce chapitre nous relatons des cas réels de systèmes très connus qui ont vus ou bien leur fonctionnement s'arrêter subitement, mais aussi des systèmes qui ont carrément étaient détruits, ce qui a causé des pertes énormes en vies humaines mais aussi en centaines de millions de dollars.

2.2 Méthodes Formelles

Historiquement, les méthodes formelles ont été définies dés les premiers temps (vers la fin des années 60) comme des méthodes associant une logique, un certain ensemble d'axiomes et des techniques de preuve. De nos jours et plus généralement, ces méthodes (formelles) sont définies comme un cadre mathématique (framework) utilisé pour modéliser les systèmes et vérifier leur comportement loin de toute ambiguïté. On parlera alors, de vérification formelle le cas de tester une spécification par rapport à son implémentation. La possibilité de prouver certaines propriétés d'un système s'appelle technique de vérification de programme.

Les premières tentatives pour décrire formellement le comportement des systèmes sont dues en majeure partie à Turing, Church, McCarthy, Strachey, Floyd et bien d'autres [AHU74]. Leurs travaux avaient pour objectifs de représenter un programme dans un cadre mathématique pur, loin de toute métaphore linguistique. Les résultats de leurs recherches ont dans un premier temps pu établir les premiers fondements des systèmes de *preuve mécanique*.

Ces dernières décennies, la recherche à produit de nouvelles techniques de vérification impliquant les machines d'états finis (FSM : Finite states Machines) et abstraites (Abstract machines) [TSL+90], les automates généralisés [Wol97, VW86] et les automates de Buchi [Buc77], les logiques modales [Che80, HC72] et temporelles [CES86, LPS99, Lam94, SE84, Koz83, SB00], les méthodes d' interpretatation abstraite des prédicats [CC77, Dam96, LGS95, MRS01, TY01], les modèles d'équivalences basées sur les bisimulations [Mou92, FV98, FM90, Lar86], les langages de spécification etc. Ces théories s'attachent à définir un cadre formel pour décrire les systèmes critiques et en étudier leur comportement à l'aide d'un model formel. D'une façon informelle, les comportements d'un système sont l'ensemble des exécutions possibles. Le but de cette théorie sera d'une part de permettre de vérifier la consistance d'une conception et de vérifier que la réalisation correspond aux spécifications, c'est à dire que le comportement est conforme à celui prévu dans le cahier de charges. D'un point de vue formel, deux classes de modèles s'imposent : les modèles de comportement qui s'intéressent aux comportements de la machine (programme) et les modèles structurants qui s'attachent au système lui-même. Ces deux modèles sont pris en charge différemment selon des caractéristiques fondamentales très précises, donc selon des paradigmes différents.

En théorie il existe plusieurs modèles abstraits pour traiter le comportement d'un système, les plus étudiés dans la littérature sont:

- Les méthodes formelles Z [Led96], B [Ab96] et Hol [GM93] sont des systèmes de spécification et de vérification des systèmes séquentiels. Les états sont décrits à l'aide de structures mathématiques bien connues tels que les ensembles, les relations et les fonctions; quant aux transitions elles sont modélisées comme des termes de pré- et de post-conditions.

- Les langages formels CSP [Hoa85], CCS [Mil80, KS80], Statecharts [Har87], Temporal Logic (TA) [Lam94], les I/O automata, les automates de Buchi [Buc77] etc., tous s'emploient à spécifier le comportement des systèmes concurrents; les états sont décrits à l'aide de domaines très simples, tels que les entiers naturels, par contre le comportement est décrit en terme de séquences, d'arbres ou d'ordre partiel d'événements.

Nous citons aussi :

- Les *Pomsets* de Pratt [Pra86] qui sont un modèle de sémantique causale et de temps linéaire. Les dépendances causales y sont représentées par un ordre partiel entre les actions (relation de causalité). Une exécution est alors vue comme une suite de multi-ensembles d'actions indépendantes, respectant la relation de causalité.

- Les *structures d'événements* de Winskel [Win86, Pen88], sont aussi un modèle de sémantique causale et de temps arborescent. A une relation de causalité, on ajoute une notion de conflit. Deux événements sont en conflits si l'occurrence de l'un est exclusive de celle de l'autre.

- Les réseaux de Petri [NPW81] qui sont très bien connus, PVS [ORS92], Lotos [FGK+96], et Lustre [CHP+91], ainsi que d'autres, la liste est vraiment bien longue.

2.3 Processus de conception

Le processus de conception est un ensemble d'actions, ordonnées et bien définies, à exécuter par l'outil ou l'opérateur de conception pour satisfaire une spécification d'un système, des consignes ou des objectifs prédéfinis. Ces actions sont étroitement dépendantes de la nature du système en cours de conception. Fournie par le client, la spécification englobe toutes ou quelques consignes que l'implémentation doive satisfaire. L'implémentation peut être élaborée à n'importe quel niveau d'abstraction en fonction des paramètres des propriétés de la spécification qui doivent être bien explicites pour que le test d'équivalence de l'implémentation avec la spécification puisse être accompli sans aucun problème. Aussi, le processus de conception peut être définit comme l'acte d'écrire les choses d'une façon très précises, ce qui permettra aux développeurs d'éviter ou de déceler les inconsistances, les ambiguïtés et les incomplétudes. En plus, la spécification est un moyen de communication mettant en place les relations entre le client et le designer, entre le designer et l'implementeur, entre l'implementeur et le testeur. Elle sert aussi comme un document qui accompagne le code source du système, mais à un haut niveau de description.

La philosophie de conception dépend étroitement qu'il s'agisse d'un modèle Hardware ou d'un modèle Software. Le fossé n'est pas très large puisque à chaque fois on rencontrera les mêmes phases à un degré moindre.

1. Pour le cas des systèmes Hardware, les processus de conception sont classifiés en deux grandes catégories :

- La conception personnalisée (« *custom design* ») : Son utilisation est très restreinte et s'accommode très bien avec les applications spécifiques telles que les processeurs.

- La conception semi-personnalisée (« *semi-custom design* ») : Mieux utilisée que la première, se réparti en deux catégories : le design à base de cellules et le design à base de tableaux. Le premier s'effectue en utilisant des cellules de la librairie (des cellules); quant au second, il utilise des modules MPGAs et FPGAs.

En générale, le processus de conception des systèmes Hardware comprend les étapes suivantes :

- *Modélisation* : Partant d'une description écrite dans un langage de spécification ou même dans le langage naturel, un modèle HDL est établi et doit être validé en premier lieu. En général, deux langages de description s'imposent pour la description du Hardware ; les mieux situés sont les langages VHDL et Verilog.

- *Synthèse* : Dans cette étape, des outils de synthèse sont utilisés pour optimiser au mieux selon des contraintes spécifiques relatives au temps d'exécution, à la surface du circuit et à son coût. Synopsys et Cadence en tant qu'outils de synthèse sont la référence à travers le monde.

- *Validation* : Cette étape finale vérifie si le modèle implémenté reproduit exactement la fonctionnalité du modèle spécifié (modèle de référence). Après l'étape de validation, viennent les phases de mise en prototype et de fabrication.

2. Pour le cas de la conception des systèmes logiciels, les différentes méthodes sont très bien connues et faisant partie du génie logiciel englobent à un degré prêt les étapes représentées par un organigramme en V et qu'il n'est pas nécessaire de détailler. Elles se présentent comme suit :

- Orientation et faisabilité des besoins
- Conception préliminaire d'un système
- Conception détaillée d'un système
- Spécification fonctionnelle du logiciel
- Conception détaillée du logiciel
- Codage du Logiciel
- Test Utilitaire du Logiciel
- Intégration et Test d'Intégration
- Validation du Logiciel
- Intégration Matériel-logiciel (Production)
- Recette Système Validation du Système
- Maintenance du Logiciel

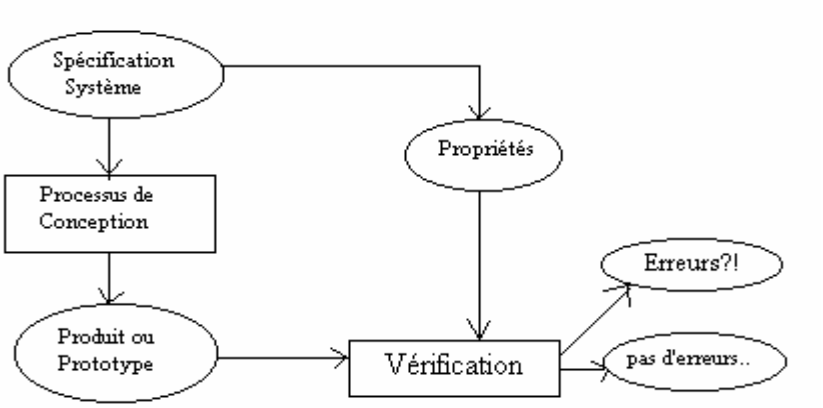


Figure 2.1 : Vue des étapes a posteriori d'un système de vérification

2.4 Spécification Formelle

Une spécification formelle d'un logiciel est une description précise et complète, dans un langage de style mathématique, des comportements que doit avoir ce logiciel. L'existence d'une spécification

formelle permet de vérifier que les comportements du logiciel sont bien ceux qu'on en attend. Elle peut aussi être utilisée pour faciliter la génération automatique de code exécutable, et en tout cas pour vérifier que l'implémentation est correcte. Etudiées depuis longtemps pour les programmes séquentiels, ces méthodes formelles s'appliquent aussi à la programmation dite parallèle, en particulier dans le domaine des protocoles de communication, et, plus récemment, dans celui des systèmes réactifs. Bien que les méthodes dites classiques (ou semi-formelles) ont apporté des améliorations très sensibles (clarté et facilité), il est difficile, voire impossible, de vérifier la cohérence et la complétude des besoins exprimés à l'aide d'outils automatiques. Un investissement important en termes humains et en termes de temps est nécessaire pour élaborer seulement les premiers prototypes. Dans certains domaines critiques comme les systèmes temps-réel embarqués (systèmes de conduite et de pilotage automatique de fusée ou d'avion, systèmes de contrôle ferroviaire ou de centrale nucléaire…), la moindre erreur est fatale. Elle peut entraîner des dommages matériels et humains. Les solutions pragmatiques (tests), mais aussi de simulation sont défaillantes et sont généralement bien difficiles à mettre en œuvre, et ne couvrent pas tout l'espace des solutions.

2.5 Vérification Formelle

La vérification *formelle* est une approche algorithmique de la vérification logique. Plus généralement, ce type de vérification s'associe à tout outil qui peut effectuer une preuve sur une description, de manière mathématique et exhaustive. Un outil de vérification formelle ne dit pas simplement si un problème existe mais aide aussi le concepteur à corriger le problème. La vérification formelle consiste à comparer la description d'un système (l'implémentation) avec la description des services attendus (les spécifications). Cette comparaison s'effectue selon une relation de satisfaction définie elle aussi formellement, en générale elle est notée |=.

Une approche possible de vérification formelle est l'utilisation de démonstrateurs automatiques, de simulateurs ou bien des systèmes de réécriture. L'utilisation de ces systèmes est en général ardue et au mieux semi-automatisable. Le modèle le plus couramment considéré pour la vérification est un système de transitions étiquetées, c'est-à-dire un automate dont les transitions entre états sont étiquetées par les actions du système. Le processus de vérification se décompose alors en deux phases : compilation d'un programme exprimé dans un langage de description vers un système de transitions étiquetées, un graphe de Kripke ou bien un réseau de Petri (en ce qui concerne le choix des modèles—la liste est vraiment longue), puis vérification des spécifications sur ce système de transitions étiquetées (ou autres). Un intérêt pratique de cette approche est qu'elle est complètement automatisable. Elle est par contre limitée par la taille des modèles. La technique de vérification appliquée dépend du formalisme utilisé pour les spécifications.

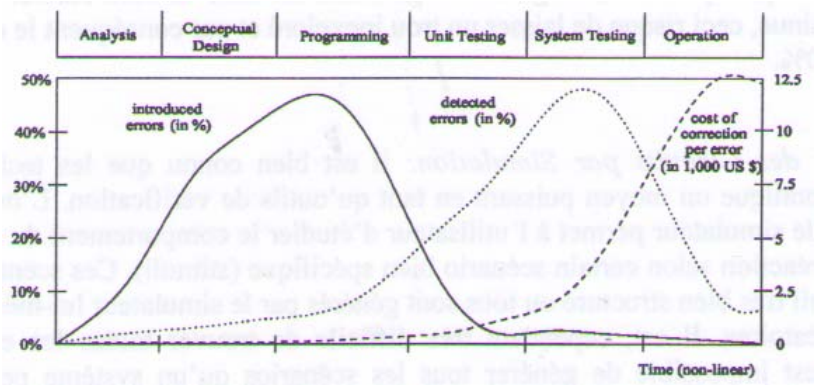


Figure 2.2 : Cycle de Vie de Logiciels, la Détection et la correction des erreurs

2.6 Vérification du Matériel (Hardware)

L'importance de la vérification du matériel. La prévention du risque d'erreur dans la conception du hardware est vitale pour toute entreprise. Le hardware a toujours été sujet à un coût de fabrication très élevé. La découverte d'un défaut de fabrication après la distribution d'un produit risque de coûter très chère à l'entreprise en question. Le fait de rappeler le produit pour le corriger certes de nos jours une telle situation est acceptée par le client, mais entraîne en général la banqueroute de l'entreprise si ce n'est plus grave. Nous pouvons avoir une idée sur les conséquences d'une telle débâcle, justement sur le cas du rappel et la correction du processeur Pentium II, qui a causé une perte sèche de 475 Millions de Dollars à Intel.

L'un des indicateurs de l'évolution du nombre de portes logiques (logical gates) dans un circuit, la très célèbre loi de Moore, d'ailleurs de tout temps vérifiée en pratique vraie, atteste que ce nombre à tendance à doubler tous les 18 mois ce qui est par conséquent un obstacle de taille pour fabriquer des produits corrects. Certaines études empiriques ont indiqué que plus de 50% des ASICs (Application-Specific Integrated Ciruit) au début de leur design et fabrication ne fonctionnent pas normalement. Donc, il n'est pas surpris de voir des manufactures investir des sommes colossales pour que leurs produits soient dépourvus d'erreurs, ce qui constitue en soit l'importance qu'on donne aux méthodes de vérification qui sont d'ailleurs une partie très importante est bien-établie dans le processus de conception. Nous constatons aussi que 27% des efforts consentis au processus de conception sont en générale réservés au design lui-même, le reste soit à peu prêt 73% du temps est réservé à la phase de vérification, c'est-à-dire, la recherche et la prévention des bugs.

2.7 Techniques de Vérification du matériel

Les techniques de vérification du Hardware se répartissent en général en trois axes importants, qui sont :

Emulation: est l'une des méthodes de test. Un système matériel générique (re-configurable) appelé l'émulateur est configuré de façon à ce qu'il se comporte exactement comme le circuit cible pour qu'il soit testé. Comme pour le testing du logiciel, l'émulation essaye de se munir d'un ensemble de stimulis injectés au circuit et en retour rassemble les réactions de ce dernier. La comparaison s'effectue alors avec les données fournies par la spécification. Pour bien tester le circuit toutes les combinaisons possibles seront alors utilisées. Cette manière de tester les circuits risque dans un certain sens de se confronter au problème de l'espace temps qui risque de faire défaut. Comme nous l'avons déjà signalé, tester toutes les possibilités est carrément impossible. Donc les concepteurs essayent toujours de choisir l'ensemble test le plus pertinent, en général c'est selon un canevas bien étudié. Donc, si le nombre de possibilités diminue, ceci risque de laisser un trou inexploré et par conséquent le circuit risque de ne pas être fiable à 100%.

Vérification des Circuits par Simulation: il est bien connu que les techniques de simulation constituent en pratique un moyen puissant en tant qu'outils de vérification. L'outil de vérification qui n'est autre que le simulateur permet à l'utilisateur d'étudier le comportement du système sur la base de la notion de la réaction selon certain scénario bien spécifique (stimuli). Ces scénarios sont établis selon un plan de travail très bien structuré ou tous sont générés par le simulateur lui-même et construit à partir de nombres aléatoires. Il est, cependant très difficile de trouver toutes les erreurs parce que tout simplement il est impossible de générer tous les scénarios qu'un système peut comporter. A titre d'exemple, pour un système simple, disons un téléphone portable avec un nombre très réduit de choix par étape. Supposons que l'on considère des scénarios à 20 étapes, donc on aura à vérifier 5^{20}, c'est-à-dire à peu prêt 100,000,000,000,000 possibilités, un nombre absolument très grand. Nous savons qu'en pratique un nombre très limité de scénarios possibles sont pris en considération. Par conséquent, il

existe bel et bien un risque réaliste pour que des erreurs indétectables c'est-à-dire non prévues par ces scénarios subsistent toujours, donc pourraient engendrer l'erreur *fatale*.

Les techniques de simulation procèdent toujours sur une représentation approchée (graphique, mathématique ou autres …) du système qui est le modèle. Ce modèle est écrit dans un langage de description; les mieux situés sont Verilog et VHDL, standardisés par IEEE. Basé sur les stimulis, les chemins d'exécutions sont examinés par un simulateur. La simulation étant le moyen le plus utilisé pour la recherche d'erreur dans les ASIC's, elle est utilisée dans différentes phases du cycle de conception (exemples : register-transfer level (RTL), gate et transistor level). Typiquement, environ 21% du temps de vérification est utilisé par l'émulation, 63% par la simulation et 16% par les techniques de l'analyse structurelle.

Vérification par Model-checking : le Model-checking est une technique de vérification qui explore tous les états possibles du système, d'ailleurs nous pouvons assimiler le model-checker à un programme de jeux d'échec qui, avant de bouger une pièce doit calculer tous les mouvements possibles en prenant soins de respecter les variantes pré-programmées (stratégies). Cette manière brute de parcourir tout l'espace engendré par les différents scénarios nous incite à voir du coté des algorithmes de parcours de l'ensemble des solutions, c'est-à-dire que si l'on associe, des algorithmes efficaces et des structures de données fiables il est tout à fait possible que des systèmes très complexes de l'ordre de 10^{20} jusqu'à 10^{476} et même plus puissent être vérifiés. Une vue exhaustive des étapes suivies par le Model-checker est consignée dans la Figure 2.3.

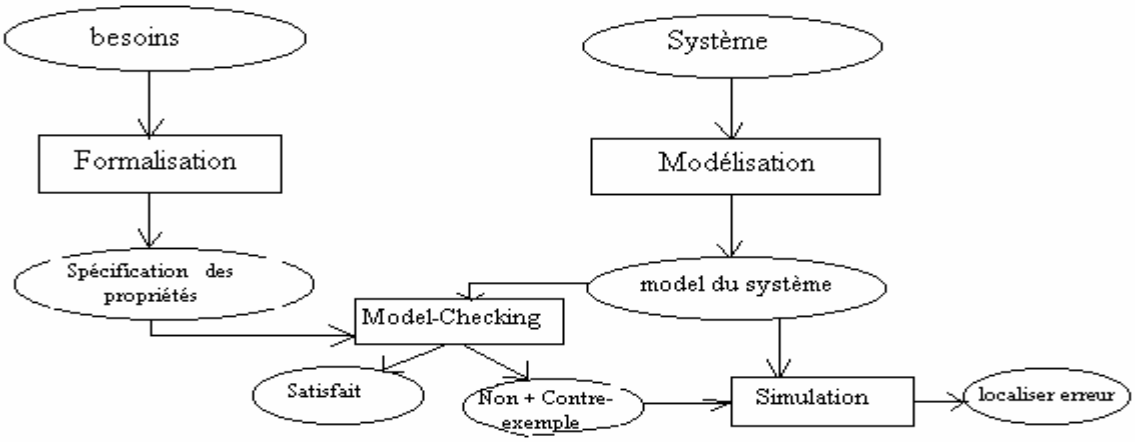


Figure 2.3 : Une vue schématique d'une approche basée sur le model-checking

2.8 Les spécifications comportementales

C'est un ensemble de formalismes utilisés pour décrire le comportement (souhaité) d'un système (programme). Donc vérifier (dans un sens) revient à comparer le système à sa spécification moyennant une certaine relation (mathématique). En général, au moment de la conception du système on dresse dans le cahier de charge un certain nombre de spécifications. Ces spécifications sont préparées minutieusement par des experts dans le but de prévoir un maximum de situations critiques (comportement général, mais aussi, des situations bien précises ou uniques). En théorie et dans la pratique plusieurs formalismes sont utilisés. Nous citons à titre d'exemples les systèmes de transitions étiquetées (Labelled Transition Systems), les Statecharts de Harel, les réseaux de Petri, les machines d'états, les automates généralisés et ceux de Buchi, les structures de Kripke [CES86] et j'en passe. L'on n'oublie pas que la description peut se faire en utilisant un langage formel et les mieux adaptés sont : CCS [Mil80, Mil89], CSP [Hoa78, Hoa85], ACP [BK85], SDL [ITU01] etc. Il existe tellement de formalismes qu'il serait très difficile de choisir le bon à moins d'être un expert, même selon certaines caractéristiques (synchronicité, parallélisme, non-déterminisme, etc.). De la même manière pour le

choix du formalisme de description, la relation de comparaison pose problème. En général, pour le cas du comportement on utilise des relations d'ordre partiel [GPS96] ou bien des relations d'équivalence, qui prennent compte du critère d'abstraction qui permet de ramener la description détaillée (système) vers une description plus abstraite (celle de la spécification). Pour le cas des formalismes formels descriptifs, ces relations sont basées sur les relations de bisimulation et sont modulées par une notion d'observabilité des actions du programme. Le choix de la relation à utiliser dépend essentiellement du type de propriété que l'on souhaite préserver lors de la comparaison

Ces deux styles de spécification sont complémentaires et dans la plus part des cas, ils s'accommodent très bien avec les Model-checking qui consistent à déterminer l'ensemble des états du modèle satisfaisant une formule de la logique temporelle.

2.8.1 Les spécifications Axiomatiques

C'est la forme la plus simple des spécifications formelles. La méthode axiomatique consiste à définir un système déductif décrivant la relation entre le programme et la spécification. Chaque relation est spécifiée en utilisant des pré- et des post-conditions. Une pré-condition spécifie les valeurs en entrée de la fonction, tandis qu'une post-condition spécifie les valeurs en sortie. Les pré- et post-conditions sont en pratique des prédicats qui peuvent être quantifiés et dont les variables sont les paramètres de la relation spécifiée. La vérification devient alors une preuve modulo l'ensemble des axiomes et des règles d'inférence. Donc, il va falloir axiomatiser l'équivalence choisie, la spécification et le programme. Lorsque la spécification est logique, la relation de satisfaction procédera par induction sur la structure du programme. Ce type de spécification s'apparente très bien avec la sémantique axiomatique dont le fondement a été établi par Floyd et c'est Hoare [Hoa69] qui lui donna sa forme actuelle définie sous forme d'un système d'axiomes et de règles d'inférence. Cette forme de sémantique est la base logique du langage de spécification des processus CSP.

2.8.2 Les spécifications Logiques

Dans ce cadre de spécification, les propriétés (spécifications) sont décrites par des formules logiques. La spécification est alors déclarative, et le problème à résoudre est de montrer que le programme appartient à l'une des logiques de choix en termes de satisfaction. La plus part d'entre elles, sont : la logique propositionnelle, la logique de 1er ordre, la logique de 2^{ième} ordre et d'ordre supérieur, mais aussi, des logiques modales tel que le mu-calcul de Kozen [Koz77], plus précisément elles sont temporelles [Pnu77]. Elles s'avèrent être les mieux appropriées compte tenu de leur pouvoir d'abstraction qui tient compte du temps en tant qu'entité symbolique et globale, donc qualitative. Contrairement, à l'autre type de logique, plutôt une extension pour couvrir l'aspect du comportement d'un système temps-réel. Cette particularité, tire profit du temps non plus comme une abstraction symbolique, mais bien quantitative et dense.

2.8.3 Les spécifications Algébriques

Les spécifications algébriques utilisent la notion d'algèbre pour définir en même temps des types de données (data-type) et leur dynamique (comportement). Une algèbre est définie par un ensemble d'opérations à appliquer sur les éléments de cet ensemble. Par analogie, un type définit un ensemble d'objets. Le comportement dynamique de ces objets peut être décrit à travers des opérations qui respectent certaines lois (composition de séquentialité, de non-déterminisme, de parallélisme, ...). On peut ainsi définir une algèbre et donc parler de spécifications algébriques de ces objets. Cette approche utilise les axiomes pour spécifier les propriétés des systèmes, mais les axiomes sont restreints à des équations. Entre autres, les algèbres de processus constituent aussi une autre approche de modélisation, de spécification, mais aussi ce qui est fort intéressant, une technique de preuve. ACP de Bergstra & Klop constitue un très bon modèle de spécification algébrique qui se présente sous forme d'un système d'axiomes ou toutes les opérations de concurrence sont présentes.

2.9 Explosion des états et Génération de Modèles

Les méthodes de vérification basée sur les modèles (model-checking) présentent un problème crucial en rapport avec la taille de ce modèle. Dès que les programmes étudiés sont de complexité réaliste, la taille du modèle correspondant devient rapidement prohibitive. Ce problème est connu sous le nom de l'explosion combinatoire.

Ce problème s'avère donc complexe dans la pratique des outils classiques de vérification qui travaillent sur le modèle complètement généré. Ces outils se trouvent de fait limités à la vérification de modèles ayant environ 10^6 états, ce qui est très en deçà dans des cas réalistes.

Un certain nombre de méthodes ont été étudiées et mis en oeuvre pour tenter de traiter différents aspects du problème de l'explosion des états. Ces méthodes jouent sur un ou plusieurs des paramètres qui sont les suivants :

La génération du modèle. L'idée est alors d'essayer de générer un modèle réduit, mais équivalent au modèle original du point de vue vérification des propriétés à valider. D'autre part, nous pouvons aussi avoir des méthodes plus ou moins efficaces de représentation du modèle, en fonction de ses caractéristiques.

L'exploration du modèle. Il est possible d'améliorer considérablement les performances, en temps ou en mémoire, lors de l'exploration du modèle, en n'effectuant qu'une exploration partielle ou en ne mémorisant qu'une partie du modèle.

La qualité du résultat Le but de la vérification formelle est de démontrer qu'une propriété est correcte ou fausse pour un système donné. Mais quand une propriété est fausse, il est souvent possible d'appliquer des méthodes plus simples qui se contenteront de la recherche d'un contre exemple, comme par exemple les différentes méthodes de test d'un programme. Ce sont des méthodes procédant par une vérification partielle, permettant de démontrer la non-satisfaction d'une propriété, mais pas leur satisfaction.

Dans le même cadre d'idée, il existe une multitude de méthodes pour minimiser d'un côté les modèles (systèmes à vérifier) et de les vérifier. Dans ce qui suit, nous présentons un nombre très restreint, mais les plus en vue, c'est-à-dire celles qui ont attiré un plus grand nombre de chercheurs et implémentées dans des outils de vérification très connus qui sont comme suit :

1. *Vérification à la volée*. Ce type d'algorithme est du à Fernandez et Mounier [FM90]. Au lieu de construire tout le modèle, puis de vérifier la propriété sur ce modèle, l'algorithme procède par une vérification instantanée dite à la volée (on the fly) pendant le parcours du graphe de transition relatif au modèle. Pour ce faire, il est en général suffisant de ne garder en mémoire que la pile permettant un parcours en profondeur. Dans ce cas, le parcours du graphe s'effectue en utilisant des algorithmes de parcours tels que DFS (Depth First Search) et BFS (Breadth First Search). Pour accélérer le calcul, la plupart des algorithmes à la volée utilisent au maximum des piles (mémoire) pour garder le plus possible des états déjà atteints. Les mêmes algorithmes sont souvent utilisés pour une vérification partielle, grâce à des méthodes de codage astucieux des ensembles d'états qui ne garantissent pas forcément l'exploration de tout le modèle.

2. *Ordres partiels* [Pra86, Pel96, GW93]. Une des causes reconnues de l'explosion des états est liée à la représentation du parallélisme par l'entrelacement d'actions parallèles asynchrones. Une idée consiste alors à éviter de parcourir tous les chemins construits par entrelacement d'actions parallèles, pour n'en parcourir qu'un sous-ensemble caractéristique pour la propriété à vérifier. Le modèle ainsi parcouru peut être beaucoup plus petit que le modèle complet, surtout si le système à vérifier contient beaucoup d'actions asynchrones. Cette technique se combine très bien avec la vérification à la volée, chaque méthode tirant parti des avantages offerts par l'autre.

3. *La Réduction*. Puisque la taille du modèle pose problème, une idée fréquemment utilisée est de réduire cette taille en tenant compte soit de la relation de comparaison utilisée, soit des spécifications à vérifier. S'il s'agit de spécifications comportementales, la comparaison s'effectue alors pour une relation d'équivalence R. La réduction du modèle consiste alors à effectuer la vérification sur un modèle

équivalent pour *R*, mais plus petit en nombre d'états (de préférence minimale). De nombreux travaux ont abouti à des algorithmes de réduction très performants [Hop71, PT87, FV98, BCM+92, EH00, GO01].

4. *Abstraction* L'abstraction consiste à construire un modèle abstrait à partir du modèle d'origine dit modèle concret, telle que chaque action du modèle concret peut être simulée par une action dans le modèle abstrait. Cette technique qui est basée sur une approche introduite par Cousot & Cousot vers la fin des années 70 [CC77, CH78] et s'est développée très rapidement surtout ces dernières années [Uri98, LGS95, MRS01, Dam96]. Les propriétés vérifiables par cette méthode doivent être préservées par la méthode d'abstraction. Dans ce cas, une propriété vérifiée sur le modèle abstrait est valide sur le modèle concret. Si la propriété n'est pas vérifiée sur le modèle abstrait, nous ne pouvons tirer aucune conclusion, car la méthode d'abstraction peut éventuellement être trop grossière pour la vérification de cette propriété. Un grand nombre de travaux ont été consacré aux méthodes d'abstraction et se sont généralisées pour la vérification les systèmes critiques, les systèmes temps-réel [TY01, MRS01], et même les logiques temporelles n'ont pas été épargnées etc.

5. *Composition.* Un système distribué étant construit par la mise en parallèle de ces sous systèmes, des méthodes ont été proposées pour tirer partie de cette technique. En particulier, la construction d'une abstraction du modèle peut se faire par la composition d'abstractions de chacun des composants parallèles du modèle, l'abstraction de chaque composante étant éventuellement plus facile à réaliser. De même, nous pouvons utiliser cette technique pour la réduction d'un modèle, en construisant le modèle global d'un système à partir du modèle minimal pour une relation d'équivalence *R* de chacune des composantes de ce système. Une autre utilisation de la composition consiste à vérifier une propriété séparément sur chaque composant, et à en déduire la validité d'une propriété globale. Néanmoins, un inconvénient majeur s'impose lors de l'utilisation de la composition c'est celui de l'explosion des états. Même si chaque composant ne décrit qu'une partie du comportement du système, le modèle correspondant est souvent de taille très grande. En fait, lorsque certains composants sont modélisés sans tenir compte de leurs interactions avec les autres (relâchement de contraintes de synchronisation) leur modèle peut même être de taille largement supérieure au modèle du système complet.

6. *Techniques symboliques*. La représentation énumérative d'un modèle se trouve directement confrontée au problème de l'explosion des états, la taille de la représentation mémoire plus grande pour des modèles de moyenne dimension et qu'en serait-il pour ceux de dimension très importante. Une idée est alors de ne plus représenter chaque état individuellement, mais d'adopter des méthodes de représentation efficaces d'ensembles d'états, et d'adapter les opérateurs utilisés dans les algorithmes de vérification classiques pour travailler directement sur des ensembles. Cette approche a véritablement pris son essor avec la définition par Bryant des Diagrammes de Décision Binaires (Bdds)[Bry92] et leur extension de Reduced Ordered BDD (ROBDD), qui permettent la manipulation efficace de fonctions booléennes. De nombreux outils de vérification utilisent les Bdds avec succès, en particulier dans le domaine de la vérification de circuits. D'autres représentations symboliques plus numériques sont aussi utilisées dans la vérification de systèmes temporisés et de systèmes hybrides [HNS+94, AD90, AH92], il s'agit de versions plus ou moins particulières des polyèdres convexes [CH78]. Nous notons aussi que les BDD ont été retenus pour notre étude et un logiciel à été réalisé et constitue un module du système global.

7. *La vérification d'équivalence***.** Les outils de vérification d'équivalence dominent actuellement le marché, principalement grâce à leur facilité d'utilisation. La vérification d'équivalence n'est pas simplement un substitut à la simulation fonctionnelle au niveau porte [Rit00]. Elle vérifie si deux circuits ou bien plus simple encore, deux modèles, décrits dans un langage précis sont équivalents. Ceci peut être utile si on possède déjà une représentation "vérifiée" d'un circuit et que l'on veuille valider une nouvelle implémentation. Mais cela permet aussi de vérifier le bon déroulement d'une étape de synthèse. Pratiquement, la vérification d'équivalence est utilisée à des niveaux variés d'implémentation : à partir du haut niveau RTL jusqu'au niveau « netlist » obtenue après placement-routage. Ainsi, elle sert à comparer deux descriptions, soit pour des conversions FPGA-ASIC (niveau porte/niveau porte), des réutilisations de descriptions (niveau porte/RTL), des migrations de langages (VHDL vers Verilog), des raffinements de descriptions (RTL/RTL), soit enfin pour des insertions de

BIST (« Built-In Self Test ») ou des remontées de fonctions à partir des transistors [Ger01]. L'emploi de la vérification d'équivalence par rapport à la simulation est rentable lorsque la description dépasse les 200 000 portes.

8. *La vérification de modèle***.** (Model-checking) Les outils de Model-Checking (*Model-Checkers*) veulent apporter une réponse à une question très délicate : « Ai-je écrit la description que je voulais? ». Cette technique requiert de l'utilisateur qu'il crée une spécification en entrant des propriétés sur sa description. Ce type de vérification permet de déceler des failles, comme les "deadlocks" (états qui bloqueraient le système) ou des "exclusions mutuelles" (un système ne fonctionne pas avec un autre). Le Model-Checking est une technique qui consiste à vérifier les systèmes dont le comportement se modélise par des machines d'états finis tels que les circuits séquentiels [BBE01]. Le Model-Checking symbolique possède ses racines dans les premiers concepts du Model-Checking temporel, propose par Clarke et Emerson en 1981 [CE81]. Dans le Model-Checking temporel, les spécifications sont exprimées dans une logique appelée CTL (Computation Tree Logic), et les descriptions à valider sont modélisées en systèmes états-transitions. Une procédure efficace basée sur le calcul du point-fixe est utilisée pour déterminer automatiquement si les spécifications correspondent aux systèmes états-transitions. Soit l'outil de Model Checking s'arrête avec la réponse «True», indiquant que la description satisfait la spécification, soit l'outil retourne «False». Dans ce dernier cas, l'outil produit un contre-exemple explicite. L'inconvénient de cette approche d'origine est le problème de *l'explosion combinatoire :* l'espace d'état des descriptions du monde industriel est typiquement trop grand pour permettre une recherche exhaustive.

9. *Model-Checking Symbolic***.** En 1986, Bryant [Bry86] développa une nouvelle représentation des fonctions booléennes, appelées OBDDs (Ordered Binary Decision Diagrams). La contribution cruciale de Bryant a été de montrer qu'en fixant un ordre dans le test des variables booléennes présentes dans les diagrammes de décisions binaires, ces diagrammes devenaient des représentations canoniques. McMillan réalisa alors que les OBDDs peuvent représenter symboliquement les espaces d'états dans les systèmes états-transitions, et en 1987, il développa un logiciel appelle SMV (Symbolic Model Verifier) dans lequel les systèmes contenant 10^{20} états pouvaient être vérifiés, la technique du *model checking symbolique* était née. Actuellement, les model-checking symboliques sont largement utilisés dans les industries des semi-conducteurs, SMV étant l'un des outils les plus utilisés. Plusieurs compagnies ont intégré les model-checking symboliques dans leurs propres outils de conception propriétaires. Par exemple, l'outil RuleBase est utilisé dans des projets consistant à vérifier des protocoles de bus ou d'autres composants. Le model-checker *SVE* développé à Siemens est appliqué dans de nombreux projets de développements internes, il a été commercialisé à des clients externes. D'autres outils commerciaux, basés sur le model-checking symbolique, incluent *FormalCheck* de Lucent [Cad99], commercialisé par Cadence et Design Insight.

10. *Les démonstrateurs de théorèmes***.** La démonstration de théorèmes est une technique qui met en place une description mathématique du système et des propriétés à vérifier sur ce système. Cette logique (Theorem-Proving) est donnée en terme de système formel, qui définie un ensemble d'axiomes et un ensemble de règles d'inférence. La démonstration de théorèmes est le processus de rechercher la preuve d'une propriété à partir des axiomes et des règles du système. L'acte de prouver une propriété est en général long est fastidieux, il requière l'intervention directe de l'utilisateur puisque le système doit à chaque étape générer des lemmes et des définitions intermédiaires. Les démonstrateurs de théorèmes sont des outils très puissants qui contrairement aux model-checker, ils peuvent vérifier des systèmes infinis.

L'utilisation du démonstrateur de théorème pour les architectures matérielles (PVS et NQTHM) [ORS92, BM88], consiste à formaliser le circuit dans une logique mathématique. En le définissant à un tel niveau d'abstraction, il sera converti complètement dans une approche formelle. Par exemple, cela fournit le moyen de créer une spécification mathématique sur la manière dont un multiplieur fonctionne, et d'être capable de le prouver. Cette approche permet ainsi des raisonnements arithmétiques qui peuvent être impossibles avec les autres techniques.

Il existe également des dizaines de prototypes universitaires et industriels, la plupart étant en licence publique gratuite pour les utilisateurs. Les plus connus sont:

- PVS, développé par le SRI, est un démonstrateur inductif basé sur une logique d'ordre supérieur [ORS92]. Il dispose de nombreuses règles de déduction et est utilisé dans l'industrie. La démonstration n'est pas automatique et doit être guidée par l'utilisateur.

- ACL2 [KM96], développé au CLI (Computational Logic Inc, Austin, Texas), puis à l'Université de Texas à Austin par J Moore et Matt Kaufmann, est un démonstrateur inductif basé sur une logique de premier ordre, sans quantificateurs. Il s'agit d'un démonstrateur automatique qui utilise un moteur d'enchaînement de règles de déduction, le programme à vérifier est écrit dans un pseudo-code proche du langage LISP, d'ailleurs ACL2 lui-même est écrit en LISP. Cet outil est le successeur du démonstrateur de Boyer-Moore, appelé Nqthm [BM88], afin de le rendre compatible pour un usage industriel sur de gros exemples.

- Coq [HKP97], développé à l'INRIA, est un démonstrateur de théorème inductif, basé sur une logique d'ordre supérieur. Plus précisément, cet outil est basé sur le calcul des constructions inductives, c'est-à-dire, il permet l'expression des démonstrations comme des termes, et donc leur traçabilité. Ces démonstrations peuvent être revérifiées par un système indépendant et la correction d'un système ne repose que sur celle d'un noyau très réduit. Coq propose donc une intégration du raisonnement et du calcul.

- HOL [GM93], développé initialement à l'Université de Cambridge, est un démonstrateur inductif basé sur une logique d'ordre supérieur. Cette logique est simple mais expressive. La démonstration d'un théorème est manuelle. HOL a été utilisé pour la validation de nombreux systèmes matériels, logiciels ainsi que des protocoles de communication.

2.10 Quelques Exemples Notables de cas vérifiés

- *SRT division algorithm*. En 1996, RueB, Shankar, and Srivas [RSS96] utilisèrent une technique de démonstration de théorèmes basée sur l'outil PVS pour trouver l'erreur qui a causée le bug du Pentium 94. Cette méthode de vérification fonctionne automatiquement et pourrait trouver le cas d'erreur sur le Pentium qui est causée par un mauvais calcul du quotient de la table de sélection

- *Motorola 68020*. En 1991 Boyer et Yu utilisèrent une spécification du Microprocesseur de Motorala 68020 en Lisp et l'exécutèrent sur le démonstrateur de théorème Nqthm. Cette spécification sera utilisée par la suite pour vérifier plusieurs programmes sous forme de code-machine produits par des compilateurs à partir de code source écrit en Ada, Lisp, et C.

- *AMD5K86*. En 1995 Moore et Kaufmann du Computational Logic, Inc., et Lynch de l'Advanced Micro Devices, Inc., Collaborèrent pour prouver la correction écrit en ACL2 du Micro-code de Lynch relative à l'algorithme de la division en virgule flottante implémenté sur l'AMD5K86.

2.11 Conclusion

La première partie de ce chapitre a été consacrée, principalement, aux techniques de spécification et de vérification basées sur les méthodes formelles. Nous y avons quelque peu détaillé le processus de spécification des plate-formes hardware/software en utilisant un cycle de vie tout à fait ordinaire s'appuyant sur un graphique mettant en valeur l'évolution du nombre d'erreurs dans les conceptions (H/W).

Dans la deuxième partie, nous avons expliqué le problème de l'explosion combinatoire des solutions relatives aux méthodes de vérification. Nous avons en outre proposé un ensemble de modèles qui contournent dans certains cas ce problème. Enfin de compte, nous avons dressé une récapitulation des outils de vérification existants ainsi que certains bugs très médiatisés.

CHAPITRE 3

Définitions de Base

3.1 Introduction

Dans ce chapitre, nous donnons certaines définitions de base qui seront citées directement ou indirectement dans la suite de notre thèse. Nous définissons en premier lieu, les différentes catégories de systèmes (informatiques). Ce sont les systèmes réactifs qui nous intéressent de prêt puisque nous mettrons l'accent sur les systèmes réactifs temps réels et la notion même du temps (mutli-forme) et de contraintes temporelles.

Les approches synchrones, asynchrones, flot de données et synchrones/asynchrones (mixte) sont introduites pour montrer le bien-fondé et l'utilité de chacune d'entres-elles puisque ce sont des modèles de base utilisés pour l'interprétation et la vérification des architectures materiel-logiciels.

Nous rappelons à travers quelques définitions de la théorie des langages et automates, éléments clés de notre prospection pour manipuler la syntaxe et la sémantique des méthodes formelles en relation avec les logiques temporelles (LTL, CTL, TCTL) qui seront abordées par la suite.

La notion même de muti-ensemble est un atout indéniable pour représenter le nombre d'occurrence d'un élément dans un ensemble. Ce paradigme est très apprécié pour décrire le comportement des systèmes concurrents (modèles) entre autres les réseaux de Petri et les machines abstraites de réécriture. Nous apprécierons leurs utilités quand on abordera les chapitres 6 et 7.

La dernière partie de ce chapitre discute quelques fondements théoriques sur l'équivalence de ce que l'on appelle les LTS (labelled transition systems), communément appelés les graphes (systèmes) de transitions étiquetées. Ce type de systèmes formels est utilisé pour modéliser les systèmes concurrents. La notion même de bisimulation fait ressortir des points forts pour la comparaison des ces LTSs, moyennant une relation, mais aussi dans certains cas des axiomes. Nous notons aussi que la bisimulation permet de mettre en place un mécanisme pour minimiser la taille des LTS et ceci dépendra de la relation utilisée qui peut être forte, faible, de branchement, etc. Nous introduirons par la suite différentes catégories de bisimulation.

3.2 Concepts fondamentaux

D'après la classification établie par David Harel et Amir Pnueli [PH85], les systèmes informatiques se répartissent en trois sous-classes :

1. Les systèmes *transformationnels* régis par des programmes de type classique qui disposent de leurs entrées dés leur initialisation, et délivrent des résultats lorsqu'ils terminent l'exécution ;

2. Les systèmes *interactifs* qui interagissent continuellement avec leur environnement, mais à leurs propres rythmes tels que les systèmes d'exploitation ;

3. Les systèmes *réactifs*, c'est la classe de systèmes qui nous intéressent en premier lieu. Ce sont des programmes (systèmes) qui réagissent continuellement sans intermittence avec leur environnement (**déterminisme**), mais avec une cadence déterminée par cet environnement. Cette classe englobe la plupart des systèmes industriels dits critiques, c'est-à-dire qu'un disfonctionnement minime soit-il risque de causer beaucoup de désagréments, ils sont dits « temps réel » tels les systèmes de contrôle-commande, automatismes, systèmes de surveillance de processus, de traitement du signal, etc., mais aussi les protocoles de communication.

Ces systèmes doivent posséder les caractéristiques suivantes :

- Ils sont des systèmes **parallèles** : en effet, leur conception doit au moins prendre en compte le parallélisme d'exécution du système et de son environnement. Donc, ils peuvent aussi être vus comme un ensemble de composants (sous-systèmes **concurrents**) s'exécutant en parallèle et coopérant à la réalisation du comportement du système souhaité sur une architecture unique ou parallèle.

- Ils sont soumis à des **contraintes temporelles** strictes, c'est-à-dire qu'il s'agit de tenir compte de la réaction du système à des stimulis extérieurs et à des délais bien précis.

- Ils sont aussi soumis à des **contraintes de sûreté** particulièrement sévères, c'est-à-dire qu'ils doivent être vérifiés rigoureusement pour qu'ils fonctionnent dans des environnements hostiles et à des cadences bien précises avec un risque carrément nul. Sinon, toute erreur risque d'entraîner des situations d'extrême gravité (perte de vie humaine, de somme d'argent qui peuvent dans des situations dépasser le milliard de dollars, retard sur le plan d 'exécution projetée, etc.).

- **Maintenance** : c'est aussi la clé des systèmes réactifs qui pousse à les concevoir pour durer le plus longtemps possible (plusieurs dizaines d'années si c'est possible), donc c'est aussi de pouvoir les re-compiler et les modifier à outrance, sans naturellement perturber leur fonctionnement auquel ils sont assujettis.

Comme les systèmes réactifs temps-réels sont souvent très complexes pour être étudiés tel qu'ils sont, alors on essaye de les abstraire tout en gardant leurs fonctions les plus importantes pour mieux cerner le problème de la vérification des propriétés du système.

La conception des systèmes réactifs considère plusieurs approches mises en place pour prendre en charge tous les aspects cités ci-dessus. Naturellement, plusieurs équipes de recherche se sont carrément spécialisées dans l'une ou dans l'autre (approche) pour mieux cerner tous les aboutissants théoriques et l'écriture d'outils très puissants en travail mixte avec certains lobbies industriels. Par la suite, une présentation exhaustive sera donnée pour détailler, l'outil conçu et la tâche auquel il est subordonné.

3.3 Systèmes temps réel

De nombreuses disciplines scientifiques se sont intéressées aux problématiques soulevées par les Systèmes Temps Réel (STR). Chacune étudiant un domaine différent, utilisant un vocabulaire spécifique et définissant des concepts qui lui sont propres, la terminologie associée au temps réel ne fait pas actuellement l'objet d'un consensus. Dans le domaine informatique, le terme même de "temps réel" suscite encore différentes interprétations.

Afin d'éviter toute ambiguïté d'interprétation, certaines définitions relatives au temps réel seront présentées dans ce qui suit :

3.3.1 Définition d'un système temps réel

De nombreuses définitions ont été données pour les systèmes temps réel, nous nous consacrerons à celle qui sera fournit ci-après. De plus amples explications formelles seront étayées par la suite.

Définition 3.1 (Système temps réel) *Un système temps réel est un système dont l'exactitude (correctness) dépend non seulement des résultats logiques, mais également des instants où ces résultats sont fournis.*

Cette définition permet de mettre de côté les définitions courantes qui s'attachent à le définir aussi comme un système constitué de processus dont l'exécution est très rapide. Un système ayant des délais à respecter exprimés en heures peut être considéré comme un STR (Chaudières à vapeur par exemple). Ce n'est donc pas uniquement la « rapidité des temps de réponse » (au sens valeurs très faibles des durées) qui caractérise un STR. Néanmoins, la plupart des STR ont des temps de réponse exprimés en milli-secondes (ms). Le respect de telles contraintes de promptitude rend complexe le développement des systèmes et constitue une des caractéristiques importantes des STR.

Cette définition s'explique essentiellement par le fait que les STR sont des systèmes appartenant à la classe des systèmes dits réactifs. Par exemple, et de manière simplifiée, le rôle du système réactif de pilotage automatique d'un avion sera justement d'empêcher sa chute, c'est-à-dire de le maintenir à une altitude tolérable (ce que nous appelons le comportement maîtrisé).

3.3.2 Temps et contraintes temporelles

Le concept de temps est une notion difficile à définir, ainsi que le rappelle saint Augustin :

« Qu'est ce donc que le temps ? Si personne ne me le demande, je le sais ; mais si on me le demande et que je veuille l'expliquer, je ne le sais plus. » *(Confession, 11, X*V).

Pour notre part, nous ne donnons pas une définition philosophique ou mathématique du temps. De manière pragmatique, nous considérons simplement le temps comme un moyen d'exprimer des synchronisations entre des activités concurrentes au sein d'un STR, mais aussi entre ces activités et celles de l'environnement. Nous utilisons, ici, le terme "activité" au sens de fait temporel. Nous définissons l'occurrence d'un événement comme la manifestation (généralement le début ou la fin) d'une activité.

Les indications sur l'ordonnancement des événements temporels sont précisément les contraintes temporelles (CT).

3.3.2.1 Les types de temps

La communauté informatique distingue deux grands types de temps [AH92]:

- *Le temps continu*. C'est un temps que l'on suppose généralement dense (et complet), au sens mathématique du terme. Il est supposé qu'entre deux instants t et t', il y ait toujours un troisième instant t", ce qui engendre en générale par coupure une infinité de temps intermédiaires entre deux temps. L'observation de la simultanéité d'événements n'est donc pas possible. C'est l'ensemble des réels qui s'emploie à le définir mathématiquement.

- *Le temps logique* (ou discret). Correspond à une succession d'instants discrets, prédéfinis et ordonnés. Ces instants peuvent être caractérisés par des événements ou par leur périodicité. Le temps logique entre deux instants n'est pas défini, il est non dense. La simultanéité des événements est donc possible, c'est un temps échantillonné aussi appelé discret pour les systèmes informatiques. Il est en général matérialisé par l'ensemble des entiers naturels.

- *Temps local/global*. Cette définition n'a de sens que dans un contexte où plusieurs horloges, généralement distribuées existent dans le système. Elle décrit le fait que l'écoulement du temps ne soit pas mesuré de la même façon dans chacun des nœuds du système. Il s'agit d'un temps global si l'écoulement du temps est le même pour tous les nœuds du système sinon il s'agit d'un temps local à un nœud.

- *Temps relatif/absolu*. Il dépend de la référence initiale de ce temps selon que cette référence est partagée par tous les temps (temps absolu) ou est spécifiques à chacun des temps.

3.3.2.2 Les types de contraintes temporelles

a) Par rapport aux types de temps utilisés

Les contraintes temporelles (**CT**) pouvant être exprimées avec les différents types de temps, nous retrouvons les mêmes catégories que celle du temps. Par exemple, l'ordonnancement des événements peut être formulé :

- En reliant les évènements temporels à des dates (utilisation d'un temps physique ou échantillonné),

- En les reliant à des événements de référence (utilisation d'un temps logique absolu),

- Où en les reliant entre eux par des relations d'ordre (utilisation d'un temps logique relatif). Souvent, on parle de contraintes temporelles implicites (ou qualitatives) lorsqu'elles sont exprimées avec un temps sans métrique (temps continu ou logique). On parle de contraintes temporelles explicites (ou quantitatives) lorsqu'elles sont exprimées à l'aide d'un temps avec métrique (physique ou échantillonné).

b) Par rapport à leurs origines

Lors du développement de STR, il est courant de différencier les contraintes temporelles suivant leurs origines. On parle alors de CT externes et des CT internes :

- Les CT externes sont issus des besoins du système. Elles expriment des relations d'ordre entre le système et des entités qui lui sont extérieures. Elles constituent un ensemble d'informations permettant de caractériser le procédé à piloter. Elles vont se traduire par des exigences sur la façon d'observer et/ou de piloter l'environnement. Ce sont des contraintes définissant le comportement naturel et maîtrisé du procédé. Cela peut être des lois de commande, des vitesses de réaction, des temps de réponse (délai entre la réception d'un stimulus et la génération d'une réponse pour le système). Elles peuvent aussi provenir de besoins non plus dictés par l'environnement mais plutôt par les clients et utilisateurs.

- Les CT internes expriment des relations d'ordre entre les entités internes au système. Ce sont des CT inhérentes au fonctionnement du système. Par exemple : la périodicité (ou non) des actions à effectuer, des synchronisations entre activités, des délais d'attente entre chaque communication (protocole de communication avec attente limité), la validité temporelle d'une donnée, mais aussi des informations concernant les processeurs, leurs vitesses, le système d'exploitation utilisé, les délais de communication, le débit du réseau de communication, etc.

3.4 Les classes de systèmes temps réel

Une distinction est faite entre les STR suivant la criticité des temps de réponse de ces systèmes. On parle alors de :

- STR critiques ou strict *(hard real-time*),
- STR fermes ou durs *(firm real-time*),
- STR souples, lâches ou mous *(soft real-time*).

D'autres critères de différenciation existent entre ces catégories : par exemple, des critères statistiques sur les temps moyens des contraintes temporelles. Cependant, la criticité des contraintes temporelles est une condition nécessaire et suffisante pour distinguer les trois types de STR présentés.

Naturellement, le développement de STR stricts ou fermes nécessite impérativement des activités de vérification et de validation du système en construction.

La validation (au sens de validation des besoins) consiste à s'assurer que le système en construction est bien celui attendu par les clients. Le "bon" système est construit. La vérification (au sens de vérification des propriétés mathématiques) consiste à s'assurer que le système est cohérent. Le système est "bien" (correctement) construit.

Dans ce qui suit nous présentons rapidement quelques approches pour mieux élucider certaines caractéristiques, ce sont :

3.5 L'Approche Synchrone

Les langages synchrones ont été introduits pour faciliter la tâche du programmeur en lui fournissant des primitives, permettant de raisonner comme si le programme réagissait instantanément aux événements externes. Chaque événement interne du programme est précisément daté par rapport au flot des évènements externes, et le comportement du programme est complètement déterministe, tant de point de vue fonctionnel que temporel. En pratique, l'hypothèse revient à supposer que le programme réagisse assez vite pour percevoir tous les évènements externes en bon ordre. Il est bien connu que les langages synchrones sont susceptibles d'être implantés de manière très efficace. Le code produit n'est

autre qu'un automate fini, dont une transition correspond à une réaction du système. Comme le code d'une transition est linéaire (il ne comporte pas de boucle) et son temps d'exécution est maximal est donc par conséquent calculable, on peut ainsi vérifier le bien-fondé de l'hypothèse de synchronisme. La notion de synchronicité ne peut être bien établie que si l'on considère les algèbres de processus telle que CCS.

Les principaux langages synchrones sont ESTEREL [BB91] qui est un langage impératif bien approprié aux applications dont la partie « contrôle » est prédominante (système à alarme, protocole de communication …), c'est aussi un langage orienté automate. SIGNAL [BCG99] et LUSTRE [HCR+91a, HCR+91b] sont des langages déclaratifs « flots de données » appropriés pour les applications dont la partie « calculs » prédomine (contrôle de processus, traitement de signal…) et les Statecharts [Har85] langage purement graphique très puissant.

3.6 L'Approche Flot de Données

Dans le domaine des systèmes réactifs, il existe une frange d'utilisateurs de spécialité tout à fait différentes de l'informatique, telles que l'électronique et l'automatique, qui nécessitent dans leurs études ou bien dans leurs recherches un outil (i.e modèles) qui soit adapté à leurs exigences. Dans ce cas de figure, il est important de leur offrir un concept leur permettant d'interpréter leurs doléances. L'approche flot de données (modèles étudiés en informatique sous le nom de « flots de données » [Kah74, McG82]) constitue justement une très bonne démarche qui permet de mettre en place l'aspect théorique et pratique de tels systèmes (électronique, automatique, etc.). On parlera dans ce cas, de langages flot de données. Ces langages présentent des avantages non négligeable et se caractérisent par :

1. Il s'agit d'un modèle parallèle à grain fin : conceptuellement, dés qu'un opérateur possède les données nécessaires, il peut élaborer les sorties correspondantes. Ainsi, les seules contraintes de séquencement et de synchronisation sur les actions sont les contraintes de dépendance entre les données. Une description à parallélisme rend possible toute une variété d'implantations, allant du séquentiel pur jusqu'à l'exécution sur architectures massivement parallèles.

2. Le modèle possède en général une propriété mathématique qui fait défaut aux langages impératifs classiques, dans lesquels les notions de mémoire et d'affectation introduisent la possibilité d'effet de bord complexe. Ce caractère mathématique facilite l'application de méthodes formelles pour vérifier, construire et transformer les programmes.

3. La description sous forme de réseaux d'opérateurs fournit directement une représentation graphique des programmes. Une notion de hiérarchie est présente aussi de façon à représenter un sous-réseau comme un opérateur. L'existence d'un formalisme textuel, équationnel équivalent au formalisme graphique, permet aussi de combiner ces deux représentations.

Le langage LUSTRE est un langage synchrone fondé sur le modèle flot de données. Dans ce cas, un programme peut être vu comme un réseau d'opérateurs, et l'hypothèse de synchronisme consiste à supposer que chaque opérateur de ce réseau réponde instantanément à ses entrées. Dans le langage LUSTRE il est fournit les moyens d'écrire la spécification de vérifier son comportement à travers les opérateurs temporels disponibles, de générer différents codes, entre autres l'automate correspondant, et de minimiser le code et le graphe associé à cet automate.

3.7 L'Approche Asynchrone

La majeure partie des systèmes distribués est naturellement asynchrone du fait qu'ils ne peuvent garantir un délai fixe pour assurer les communications entre leurs différents composants. Ce sont différents processus (tâches, activités, agents …) qui s'exécutent en parallèle à des vitesses différentes (pas d'horloge centrale) n'ayant pas forcément de mémoire commune avec des délais de communication pouvant varier. En général, la modélisation de ces systèmes nécessite de considérer les horloges générant des tick à grains fins, c'est-à-dire que les évènements peuvent être générés dans un espace temporisé dense (continu).

3.7.1 L'Approche Synchrone / Asynchrone

Les méthodes algébriques de spécification des systèmes distribués sont considérées comme un puissant outil pour d'un côté décrire mathématiquement les différents aspects des processus communicants, mais aussi de vérifier leur comportement. Algébriquement, deux processus a et b qui s'exécutent en parallèle sont interprété par (a || b). Il existe deux models de base qui permettent de différencier les systèmes distribués, c'est le synchronisme et l'asynchronisme. L'exemple suivant permet de mieux cerner leur différence.

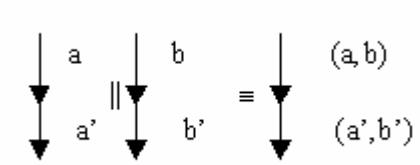
– Synchrone : a || b = (a, b)

– Asynchrone : a || b = a. b + b. a

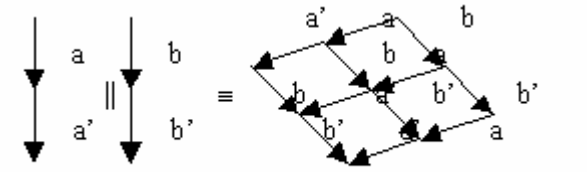
exemple:

1. composition parallèle de deux processus synchrones (l'opérateur est noté double barres verticales « || ») :

a.a' || b.b' ≡ (a, b) . (a', b')



2. composition parallel asynchrone (entrelacement = interleaving)



3.8 Langages et automates

On appelle un ensemble fini de symboles représentant des signaux (de communication) $X = \{x_0, \ldots, x_{n-1}\}$ un alphabet de dimension n. Un évènement synchrone x de X est un ensemble de signaux. La dénomination synchrone signifie que tous les signaux d'un événement sont supposés survenir au même instant.

Une exécution d'un programme synchrone est vue par un observateur extérieur comme une suite d'événements composés par des signaux de son interface. On définit ainsi une exécution d'un programme comme étant une trace (ou mot synchrone) sur *X*. Un mot *m* définit sur *X* est une suite finie ou infinie d'événements synchrones sur *X* ($m = (x_0,x_1, \ldots) \mid \forall i, x_i \in 2^X$). La longueur d'un mot *m* est notée $|m|$ ($0 \leq |m| \leq \infty$). On note ε le mot de longueur nulle.

Exemple 3.1

Soit X = $\{x_1,x_2,x_3\}$. Alors $\{x_1\}$, $\{x_1, x_3\}$, $\{x_1,x_2,x_3\}$ et $\varnothing$ sont des événements synchrones sur X. ($\{x_1\}$, $\{x_2, x_3\}$, $\{x_1,x_2,\}$, $\varnothing,\varnothing,\{x_1, x_2\}$, …) est un mot synchrone sur X.

Définition 3.2 *Un ensemble fini ou infini de mots est appelé un langage. Un programme est décrit par l'ensemble de ces traces donc par un langage.*

Si $m = (x_0, \ldots, x_n)$ est un mot fini et $m' = (x'_0, \ldots, x'_n, \ldots)$ un mot de taille quelconque, on note $m.m'$ la concaténation de ces deux mots, i.e., $m.m' = (x_0, \ldots, x_n, x'_0, \ldots, x'_n, \ldots)$. Notons en outre que X^∞ l'ensemble des mots (finis ou infinis), X^ω l'ensemble des mots infinis, X^* l'ensemble des mots finis et X_n l'ensemble des mots de taille n sur X.

Définition 3.3 *Soit X un ensemble de signaux, et soit 2^X l'ensemble des événements sur X.*

Un automate à entrée/sortie (I/O) A sur X est un tuple $(Q_A,q^0_A,I_A,O_A,\delta_A)$ ou

- *Q_A est un ensemble fini d'états contenant l'état initial q^0_A.*
- *$I_A \subseteq X$ et $O_A \subseteq A$ sont les ensembles disjoints, respectivement des signaux d'entrée et de sortie.*

Définition 3.4 *On appelle configuration un couple formé d'un état et d'un événement d'entrée $(q,i) \in Q_A \times 2^{IA}$ ou $\delta_A \subseteq Q_A \times 2^{IA} \times 2^{OA} \times Q_A$ est la relation de transition.*

Si l'automate est dans la configuration $(q,i) \in Q_A \times 2^{IA}$, il peut passer dans l'état q' en émettant la sortie o si et seulement si $(q,i,o,q') \in \delta_A$. Quand il n'y aura pas d'ambiguïté sur la relation de transition, on notera $q \xrightarrow[o]{i} q'$ s'il existe une suite infinie $(q_0, q_1, \ldots, q_n, \ldots)$ d'états, avec $q_0 = q^0_A$ et pour tout $n \geq 1$, $q_{n-1} \xrightarrow[x_{n-1} \cap O_A]{x_{n1} \cap I_A} q_n$.

Définition 3.5 *Un tel automate peut donc être interprété comme un reconnaisseur de langage sur X ou éventuellement un générateur de langage.*

Définition 3.6 *Un langage des traces est dit régulier s'il existe un automate A reconnaissant ce langage ($T = T_A$).*

Définition 3.7 *Un automate réactif ne peut pas refuser un événement d'entrée. Il possède donc la propriété*

$$\forall q \in Q_A, \forall i \in 2^{I_A}, \exists o \in 2^{O_A}, \exists q' \in Q_A, q \xrightarrow[o]{i} q'$$

Définition 3.8 *Un automate déterministe a au plus une réaction possible à un événement d'entrée donné. Il possède donc la propriété*

$$\forall q \in Q_A, \forall i \in 2^{I_A}, \exists o_1,o_2 \in 2^{O_A}, \forall q'_1,q' \in Q_A, q \xrightarrow[o_1]{i} q'_1 \quad et \quad q \xrightarrow[o_2]{i} q'_2 \Rightarrow o_1 = o_2 \quad et \quad q'_2 = q'_1$$

Définition 3.9 *Le produit synchrone est l'opération de base pour faire communiquer des automates.*

Soient $A_1 = (Q_1,q^0_1,I_1,O_1,\delta_1)$ et $A_2 = (Q_2,q^0_2,I_1,O_2,\delta_2)$, deux automates. Leur produit synchrone $A_1 \times A_2$ est un automate,

$$A_1 \times A_2 = (Q_1 \times Q_2,(q^0_1, q^0_2), (I_1 \setminus O_2) \cup (I_2 \setminus O_1), (O_1 \setminus I_2) \cup (O_2 \setminus I_1), \delta_{12})$$

où

$$(q_1,q_2) \xrightarrow[o]{i} (q'_1,q'_2) \Leftrightarrow \exists l_1 \in I_1 \cap O_2, \quad \exists l_2 \in I_2 \cap O_1.$$

$$q_1 \xrightarrow[(l_2 \cup o) \cap o_1]{l_1 \cup (i \cap I_1)} q'_1 \qquad\qquad q_2 \xrightarrow[(l_2 \cup o) \cap o_2]{l_2 \cup (i \cap I_2)} q'_2$$

Intuitivement, ce produit synchrone est équivalent à lier les entrées/sorties de A_1 avec celles de A_2 portant le même nom.

Un état $q \in Q_\varphi$ est dit accessible si et seulement s'il existe une exécution finie $(q^0{}_\varphi, q_1, q_2, \ldots, q)$ atteignant q.

Soit $\mathcal{R} \subseteq Q_\varphi$ l'ensemble des états accessibles. Cet ensemble est défini par un plus petit point fixe.

3.9 Les multi-ensembles

Définition 3.10 (Multi-ensembles*) Etant donné un ensemble X, un multi-ensemble sur X est une fonction M : X → N. Pour tout x ∈ X, nous appellerons M(x) la multiplicité de x dans M.*

Nous noterons **M(X)** l'ensemble des multi-ensembles sur X. Nous dirons qu'un multi-ensemble est fini pour tout x ∈X, M(x) = 0 sauf pour un nombre fini d'entre eux.

Nous notons le multi-ensemble vide de la même manière que l'ensemble vide : ∅.

Notations 3.11 (Opérations sur les multi-ensembles)

$$M_1 = M_2 \overset{déf}{\Leftrightarrow} \forall x \in X \quad M_1(x) = M_2(x)$$

$$M_1 \subseteq M_2 \overset{déf}{\Leftrightarrow} \forall x \in X \quad M_1(x) \leq M_2(x)$$

$$M_1 \not\subset M_2 \overset{déf}{\Leftrightarrow} M_1 \subseteq M_2 \wedge \exists x \in X : \quad M_1 = M_2(x)$$

$$M_1 = M_2 + M_3 \text{ ou } M_1 \cup M_2 \overset{déf}{\Leftrightarrow} \forall x \in X : \quad M_1(x) = M_2(x) + M_3(x)$$

$$M_1 = M_2 \cap M_3 \overset{déf}{\Leftrightarrow} \forall x \in X : \quad M_1(x) = \min\{M_2(x), M_3(x)\}$$

$$M_1 = M_2 \;/\; M_3 \overset{déf}{\Leftrightarrow} M_3 \subseteq M_2 \wedge \forall x \in X : \quad M_1(x) = \{M_2(x) - M_3(x)\}$$

$$M_1 = M_2 - M_3 \overset{déf}{\Leftrightarrow} M_1 = M_2 \;/\; \{M_2 \cap M_3\}$$

$$|M_1| \overset{déf}{\Leftrightarrow} \sum_{x \in X} M_1(x)$$

Exemple 3.12 Soit X = { x_1, x_2,x_3} et soient M_1 = { x_1, x_1, x_1, x_1} et M_2 = { x_1, x_2, x_2,x_3} deux multi-ensembles sur X, alors nous avons :

$M_1 + M_2 = \{x_1,x_1,x_1,x_2,x_2,x_3,x_3\}$

$M_1 \cap M_2 = \{x_1,x_2\}$

$M_2(x_2) = 2$

$M_1 \not\subset M_2$ *et* $M_2 \not\subset M_1$

$M_1 \setminus M_2$ n'est pas défini.

3.10 Equivalences de Bisimulation

Les systèmes de transitions permettent certes de bien représenter les systèmes parallèles, mais offrent peu de moyens pour vérifier que deux graphes de transitions seraient égaux. Dans ce cas de figure, il est alors plus intéressant de considérer la relation d'équivalence dite de bisimulation forte [Mil80, Par81].

Dans la littérature de Glabeek [Gla94], 11 sémantiques de concurrence ont été développées et définies sur les LTS (labelled transitions systems). La plus faible d'entre elles étant la *trace semantics* [Hoa85]. La plus intéressante d'entre elles étant la sémantique de bisimulation telle que développée par Milner et utilisée comme sémantique de base pour décrire le langage des processus CCS [Mil80]. L'équivalence de bisimulation est le raffinement de la bisimulation observationelle, comme introduite par Hennessy et Milner dans [HM85].Une autre sémantique d'équivalence développée par Bakker and Zucker sitée dans [Gla94], coïncide avec la sémantique de bisimilation. Entre les deux (la forte et la trace bisimulation). Par exemple *failure semantics* introduite par Brookes, Hoare & Roscoe dans [BHR84]. Elle est plus fine que la trace bisimulation utilisée comme système de base pour introduire le langage algébrique de processus CSP de Hoare [Hoa78, Hoa85]. De Nicola & Henessy [DH84] ont développé la *Testing equivalence*. Olderog & Hoare développèrent la *readiness semantics* bisimulation. Nous retrouvons aussi *ready trace* semantics introduite par Pnueli dans [Pnu85] ainsi de suite.

La bisimulation constitue l'équivalence de base qui permet d'identifier des systèmes ayant un comportement de même structure. Son nom vient du fait qu'elle permet de définir la capacité de deux systèmes $A1$ et $A2$ à s'imiter mutuellement, c'est à dire que toute action effectuée par $A1$ peut être imitée par une action de $A2$, et réciproquement. D'autre part les états atteints après ces actions doivent être bisimilaires, c'est-à-dire qu'ils doivent permettre aux systèmes d'avoir à nouveau des comportements bisimilaires.

3.10.1 Notions d'équivalence de comportement

Soit un système A qui évolue par changement d'états, et que l'on peut interrompre à tout instant. L'observation d'un tel système consiste en une succession d'actions qu'il peut effectuer. L'équivalence de traces identifie les systèmes qui admettent le même ensemble d'observations. C'est-à-dire, plus formellement :

Soient $A1$ et $A2$ deux graphes de transition (LTS, graphe de Kripke ou autres similaires). Dire que $A1$ et $A2$ sont bisimilaires, c'est,

- Lorsqu'il n'est pas possible de les distinguer l'un de l'autre
- Lorsque l'on parcourt de l'intérieur l'un d'eux, il ne nous sera pas possible de dire dans quel model nous y sommes.

L'une des bisimulations les plus utilisées dans certains systèmes de vérification synchrones, tel que Aldebaran [FGK+96], incorporé dans la *boite à outils* « Toolbox », et développé par l'INRIA étant l'équivalence de traces. Dans ce cas de figure, l'on considère toutes les exécutions qui sont les traces, uniquement depuis l'origine (le sommet), sans tenir compte de l'arborescence, c'est-à-dire des exécutions non déterministes.

3.10.2 Equivalence de traces

Définition 3.13 *Deux états q_1, q_2 ont les même traces, noté $q_1 \sim_{Tr} q_2$, si $Tr(q_1) = Tr(q_2)$.*

Deux systèmes de transition $A1$ et $A2$, ont les mêmes traces, noté $A1 \sim_{Tr} A2$, si les exécutions partent de leurs initiaux ont les mêmes traces, i.e, Tr($A1$) = Tr($A2$).

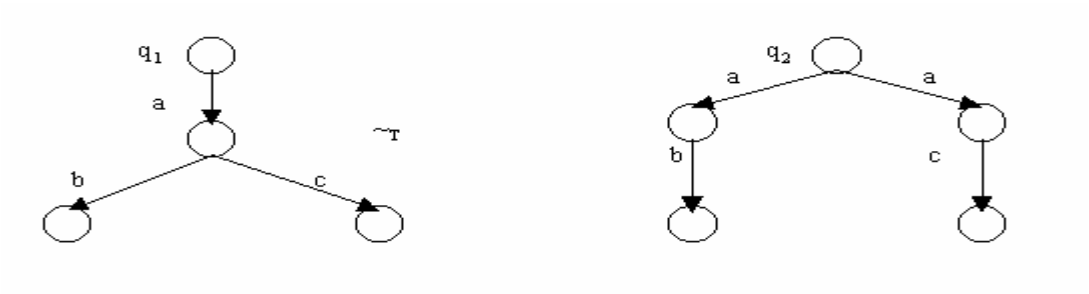


Figure 3.1 : Deux systèmes de transition ayant les mêmes traces

Il est clair que ces deux systèmes de transition sont équivalents modulo la relation d'équivalence qui est notée $\sim_T$. Nous pouvons remarquer que les deux graphes respectent les mêmes exécutions qui sont les traces (a.b, a.c).

Exemple 3.14 supposons que nous voulions modéliser le comportement d'un distributeur automatique de boissons (café et thé). Ceci se fera comme dans les graphes de transition de la figure 3.1. Supposons en outre que les étiquettes (actions) a, b et c soient renommées en Pièce (P), Café (C) et Thé (T). Il va de soit que les deux systèmes possèdent le même comportement, c'est-à-dire qu'ils acceptent le même langage, soit P.(C + T) pour le premier et P.C + P.T pour le deuxième graphe.

Une remarque de taille s'impose d'elle-même, en vérité le comportement de ces deux systèmes n'est pas exact si l'on désire tenir compte des moments du déroulement des évènements, à savoir, quant on met une pièce dans la machine, l'on aura le choix d'appuyer sur le bouton **Café** ou bien celui de **Thé**. Par contre dans le deuxième graphe celui de droite, nous aurons le choix tout au début que lorsque l'on mette une pièce de monnaie la machine nous sert thé, ou bien lorsque l'on met une pièce l'on est servit café. On aura l'impression que la machine possède uniquement deux boutons, l'un autorise la distribution du café et l'autre le thé. Naturellement il faut que l'on mette la pièce avant.

Cet exemple introduit la notion du temps de branchement (arborescent) pour le premier cas et celui de temps linéaire pour le deuxième cas.

Pour mieux apprécier l'utilité de la relation de bisimulation, l'on doit étudier la relation de bisimulation simple et celle de bisimulation forte.

Définition 3.15 *Soit un système de transition étiqueté, une relation binaire* $\mathfrak{R} \subseteq Q \times Q$ *est une bisimulation si et seulement si :*

$\forall (p_1, p_2) \in \mathfrak{R} \;.\; \forall a \in A.$

$$\forall q_1 (p_1 \xrightarrow{a} q_1 \;\Rightarrow\; \exists q_2 (p_2 \xrightarrow{a} q_2 \;\wedge\; (q_1,q_2) \in \mathfrak{R})) \wedge$$

$$\forall q_2 (p_2 \xrightarrow{a} q_2 \;\Rightarrow\; \exists q_1 (p_1 \xrightarrow{a} q_1 \;\wedge\; (q_1,q_2) \in \mathfrak{R}))$$

Donc, intuitivement deux états p et q sont bisimilaires si pour chaque état p accessible à partir d'un état q par l'exécution d'une action, il existe un état p' accessible à partir d'un état q' par l'exécution de cette même action de façon à ce que p' et q' soient bisimilaires, et vice-versa.

Si $\mathfrak{R}$ est une relation de bisimulation, alors,

- $\forall p \in Q, \;(p,p) \in \mathfrak{R}$ [**réflexivité**]
- *Si* $(p_1,p_2) \in \mathfrak{R}$, *alors* $(p_2,p_1) \in \mathfrak{R}$ [**symétrie**] et
- *Si* $(p_1,p_2) \in \mathfrak{R}$, $(p_2,p_3) \in \mathfrak{R}$, *alors* $(p_1,p_3) \in \mathfrak{R}$ [**transitivité**]

Ce qui montre que la bisimulation est bien une relation d'équivalence.

Définition 3.16 (Bisimulation forte). *Une relation* $\Re \subseteq S_{A1} \times S_{A2}$ *ayant les propriétés suivantes est dite bisimulation forte si,*

1. *Pour toute transition* $p \rightarrow^{a} A_1 p'$, *il existe un état* q' *de* A_2 *telque* $q \rightarrow^{a} A_2 q'$ *et* $p' \Re q'$.
2. *réciproquement, pour toute transition* $q \rightarrow^{a} A_2 q'$, *il existe un état* p' *de* A_1 *telque* $p \rightarrow^{a} A_1 p'$ *et* $p' \Re q'$.

Exemple :

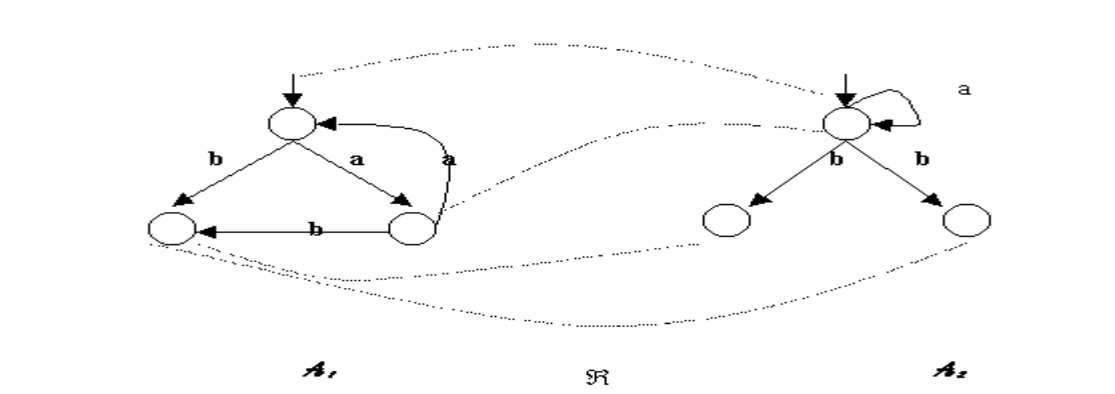


Figure 3.2 : Deux systèmes bisimilaires

Sur l'exemple de la Figure 3.2, on a représenté par les lignes pointillées la relation $\Re$ établissant la bisimilarité entre A_1 et A_2.

Préposition 3.17 *Pour tous systèmes de transitions* A, A_1, A_2, A_3,

- Id_A *une bisimulation entre* A *et lui-même*
- *Si* $\Re$ *est une bisimulation entre* A_1 *et* A_2, *alors* $\Re^{-1}$ *est une bisimulation entre* A_2 *et* A_1.
- *Si* $\Re$, $\Re'$ *sont deux bisimulations respectivement entre* A_1 *et* A_2 *et entre* A_2 *et* A_3, *alors* $\Re' \circ \Re$ *est une bisimulation entre* A_1 *et* A_3,
- *Si pour tout* $i \in I$ *(I un ensemble quelconque)* $\Re_i$ *est une bisimulation entre* A_1 *et* $A2$, *alors* $\cup_{i \in I} \Re_i$ *est une bisimulation entre* A_1 *et* A_2.

3.11 Conclusion

Dans ce chapitre nous présenté certains concepts fondamentaux pour bien appréhendé la suite de notre travail. Nous avons donné dans un premier temps certaines définitions de base en relation directes avec les systèmes réactifs, puis nous avons quelque peu introduit une typologie des systèmes temps-réel et quelques-unes de leurs caractéristiques (concurrence, synchronicité, déterminisme, interleaving, etc.).

Nous avons dans un deuxième temps rappelé certains éléments de la théorie des langages et surtout pour nous avons définit formellement les automates.

En dernier lieu, la bisimulation des systèmes concurrents a été abordée pour montrer la capacité des algorithmes basés sur les techniques de bisimulation à minimiser leur complexité ce qui leur confère un moyen d'abstraction et de raffinement par l'intermédiaire des systèmes de transitions étiquetées, des structures de Kripke mais aussi des automates de Buchi.

CHAPITRE 4

Les Diagrammes de Décision Binaire

4.1 Introduction

La conception des systèmes Hardware/Software est une tâche très complexe compte tenu de l'entrelacement de l'implémentation de logiciels spécialisés et de hardware dans une seule et unique puce. Un ensemble de puces réparties dans un système embarqué devient un problème critique qui nécessite une sûreté de fonctionnement. La nécessité de rechercher d'autres algorithmes de vérification s'est imposé au début des années quatre-vingts. Bryant et al [Bry92, Bry86, BRB90] sont les premiers à avoir utilisé brillamment les travaux de Akers [Ake78] sur l'utilisation des diagrammes de décision binaire pour la vérification symboliques des circuits intégrés. L'implémentation des BDD à travers les algorithmes « restrict », « apply » et ITE [BRB90] permettent de vérifier le bon fonctionnement des circuits de petite taille. Ruddel utilisa un algorithme basé sur les techniques de programmation dynamique pour la réduction de la taille des BDD ce qui a permit de vérifier des circuits de grande taille atteignant 2^{100} états [Rud93]. Plus récemment l'extension des BDD en ROBDD (Reduced Ordered Binary Decision Diagram) tels que les K*BMDs [DBR96] ont été développés pour représenter les fonctions arithmétiques d'une façon plus compacte et donner ainsi les moyens pour prendre en charge le cas de la vérification des circuits séquentiels modélisés par les machines d'états finis (FSM : finite states machines) dite analyse d'accessibilité. Le processus de vérification s'établit par la comparaison d'une implémentation qui doit avoir les mêmes valeurs en sortie que la spécification.

Dans cette partie de notre thèse nous décrivons une technique pour la manipulation de fonctions booléennes basée sur les diagrammes de décision binaire réduits et ordonnés (ROBDD). Cette représentation est basée sur une implémentation efficace de l'opérateur if-then-else (ITE). Une table de hachage est utilisée d'un côté pour entretenir la forme canonique dans le diagramme ROBDD et pour constituer avec le ROBDD lui-même une structure compacte afin de gagner une grande partie de la mémoire utilisée pour leur manipulation. Cette technique fait appel à une fonction de mémorisation « hash-based cache » pour implémenter l'algorithme qui manipule la fonction récursive ITE. L'accélération de cette manipulation s'effectue par l'utilisation d'un ensemble de règles qui détectent l'équivalence entre fonctions calculées.

Il est bien connu que la logique propositionnelle est un formalisme que l'on peut coder sous forme de matériel : on peut construire des circuits en reliant des portes logiques (gates) les unes aux autres, chaque porte réalisant un connecteur logique. Par exemple, une porte NAND est un bout de circuit à deux entrées et une sortie; dans les réalisations positives de la logique, lorsque la tension des deux entrées est au niveau bas, représentant *F*, la sortie est haute, et si une entrée est haute, alors la sortie est basse. Elle ne peut pas décrire des circuits avec des connexions cycliques (comme l'on fait usuellement pour créer des dispositifs instables comme des horloges, mais presque tous les circuits dans un ordinateur contemporain sont des superpositions de portes logiques. En simplifiant, un ordinateur réel est une grosse formule propositionnelle codée sous forme de matériel, plus une horloge (qui envoie des suites de *T(rue)* et *F(alse)* à une vitesse prédéfinie) et des composants d'interface. En somme, la logique propositionnelle est essentiellement suffisamment expressive pour décrire n'importe quel ordinateur réel. Comme il a déjà été cité, l'élément de base de la logique propositionnelle étant la proposition qui est considérée comme étant une variable propositionnelle qui prend ces valeurs du domaine sémantique qui correspond à ses deux interprétations : Vrai et Faux ou **1** et **0**. En plus des définitions introduites dans la partie Annexe, nous présenterons ce qui suit :

Définition 4.1 *Une algèbre booléenne est un ensemble non vide A contenant deux éléments T, F, et muni d'opérations* $\wedge$, $\vee$, $\neg$ *obéissant aux équations suivantes :*

$$a \wedge b = b \wedge a,\ a \wedge (b \wedge c) = (a \wedge b) \wedge c,\ a \wedge a = a,\ a \wedge T = a,\ a \wedge F = F,$$

$$a \vee b = b \vee a,\ a \vee (b \vee c) = (a \vee b) \vee c,\ a \vee a = a,\ a \vee T = T,\ a \vee F = a,$$

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c),\ a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c),$$

$$a \wedge \neg a = F,\ a \vee \neg a = T,\ \neg\neg a = a,$$

$$\neg(a \vee b) = \neg a \wedge \neg b,\ \neg(a \wedge b) = \neg a \vee \neg b.$$

Définition 4.2 *Une clause est la disjonction d'un nombre fini de littéraux. On parle aussi de clause conjonctive pour des clauses construites comme la conjonction d'un nombre fini de littéraux.*

Définition 4.3 *(*Forme Normale Conjonctive) *Une Forme Normale Conjonctive, noté CNF, est l'écriture de la formule sous la forme d'une conjonction de clauses (des ET de OU de littéraux).*

Théorème 4.4 *Toute formule admet une forme normale conjonctive qui lui est logiquement équivalente.*

Définition 4.5 (Forme Normale Disjonctive) *Une Forme Normale Disjonctive, notée DNF, est l'écriture de la formule sous la forme d'une disjonction de clauses (des OU de ET de littéraux). Contrairement à la CNF, il n'est pas toujours possible de calculer de façon polynomiale une transformation en DNF pour une formule quelconque. En effet, cela reviendrait à énumérer des solutions de la formule et il n'existe pas à ce jour d'algorithme polynomial qui sache le faire.*

4.2 Diagrammes de Décision Binaire

Une autre idée qui fonctionne pour prouver des formules propositionnelles et qui vient d'idées sémantiques est celle des *diagrammes de décision binaire*, ou *BDD (Binary Decision Diagram)*. Le créateur des BDD tels que nous le connaissons aujourd'hui est Randall E. Bryant en 1986 [Bry86]. Cependant, les BDD sont juste des arbres de décision avec quelques astuces bien connues en plus, et les arbres de décision remontent à George Boole (1854). Les idées sont très simples, mais les réalisations informatiques sont usuellement plus complexes qu'avec les méthodes précédentes.

En gros, les techniques qui font que les BDD fonctionnent sont : d'abord, au lieu de représenter les arbres de décision comme des arbres en mémoire (où il y a un unique chemin de la racine à n'importe quel noeud, nous les représentons comme des *graphes orientés acycliques* ou *DAG*, autrement dit nous partageons tous les sous-arbres identiques. Ensuite, nous utilisons la règle de simplification suivante : si un sous-arbre a deux fils identiques, alors remplacer le sous-arbre par ce fils; Essentiellement, ce sous-arbre signifie « si A est vrai, alors on utilise le fils de droite; si A est faux, utilise le fils de gauche »: comme les fils de gauche et de droite coïncident, il n'y a pas lieu d'effectuer une sélection fondée sur la valeur de A. Enfin, nous ordonnons les variables dans un ordre total donné $<$, et exigeons que, si nous descendons dans le BDD sur n'importe quel chemin, nous rencontrons les variables en ordre croissant. Cette dernière propriété assurera que les BDD sont des *représentants canoniques* de formules à équivalence logique près, c'est-à-dire que si Φ et Φ' sont deux formules équivalentes, alors leurs BDD construits sur le même ordre sont identiques.

Les diagrammes de décision binaires constituent un modèle de représentation des fonctions booléennes. Un arbre de décision binaire est un arbre orienté composé d'une racine, de sommets intermédiaires et de sommets terminaux valant **0** ou **1**. La racine et les sommets intermédiaires sont indexés et possèdent deux sommets "fils", un fils gauche et un fils droit. Le fils gauche est atteint en empruntant la branche **"0"**, le fils droit en empruntant la branche **"1"**. Un arbre de décision binaire est obtenu en appliquant récursivement la première forme du théorème de Shannon sur l'ensemble des variables de la fonction.

Théorème 4.6 (de Shannon) $F(x_1,x_2,\ldots,x_i,\ldots,x_n) = x'.F(x_1,x_2,\ldots,0,\ldots,x_n) + F(x_1,x_2,\ldots,1,\ldots,x_n)$.

Si ce théorème est appliqué à F pour x_1, puis aux deux sous-fonctions obtenues pour x_2, et ainsi de suite jusqu'à x_n, on peut réaliser un arbre de décision binaire (un exemple est donné dans la Figure 4.1).

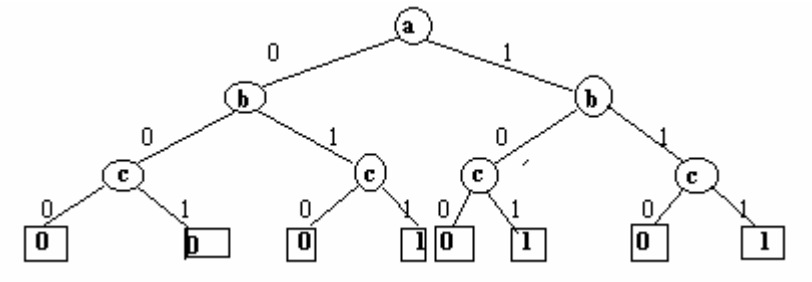


Figure 4.1: Arbre de décision binaire associé à la fonction *F(a,b,c)=a'bc+ac.*

Notons qu'il est possible de réduire la taille de l'arbre en s'arrêtant dans l'application du théorème de Shannon dès que l'on se trouve en présence d'une constante 0 ou 1. Ainsi, dans l'exemple précédent nous voyons que *F(0,0, c) = 0*, c'est à dire que *F = 0* quelle que soit la valeur qui sera assignée à *c*.

4.2.1 Diagrammes de Décision Binaire Ordonnés (OBDD)

Après sa construction, un arbre de décision binaire peut être réduit par transformation en un graphe acyclique orienté appelé graphe de décision binaire (BDD). Il est bien connu qu'un arbre de décision binaire possède des feuilles, et engendre en son sein des sous-graphes identiques. La représentation peut être simplifiée en ne conservant qu'une seule représentation des feuilles et sous-graphes concernés, et en remplaçant les autres feuilles et sous-graphes par un arc vers le premier. Le graphe de décision binaire ainsi obtenu peut également être réduit par élimination des sommets redondants, c'est à dire des sommets ayant un fils gauche et un fils droit identique (figure 4.2).

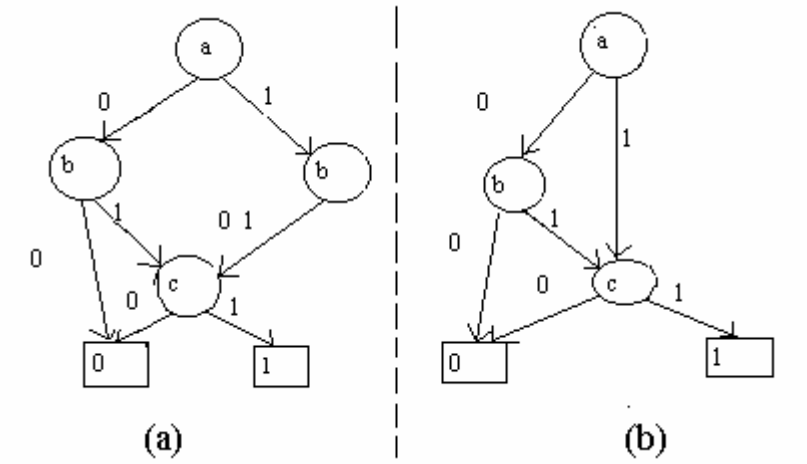


Figure 4.2: Réduction par élimination: (a) des feuilles et sous graphes identiques ; (b) des sommets redondants

En cherchant et remplaçant tous les sous-arbres équivalents d'un diagramme de décision binaire, on obtient le diagramme de décision binaire minimal pour l'ordre correspondant des variables (*ROBDD*). Il est important de noter que pour un ordre donné de variables, le graphe de décision binaire minimal est *unique*. Cette unicité peut évidemment être utilisée pour montrer l'équivalence de deux expressions logiques.

Remarque 4.7 La complexité de la procédure de réduction d'un BDD est *O(nlog n)*.

La structure du BDD (et en particulier sa taille) dépend de l'ordre dans lequel sont considérées les variables. Ainsi, pour deux ordres différents, les BDD peuvent être de tailles très différentes.

4.2.2 Comment Construire les BDDs [CYB98]

Afin de garantir la canonicité de la représentation, les contraintes suivantes sont imposées :

- chaque variable ne peut apparaître qu'une fois au plus sur chaque chemin entre la racine et une feuille
- les variables sont ordonnées de telle façon que si un sommet de label x_i a un fils de label x_j alors $ord(x_i) < ord(x_j)$

Il y a deux stratégies principales de construction des BDDs :

- De la racine vers les feuilles (top-down). Cette méthode est utilisée lorsqu'on part d'une formule algébrique
- Des feuilles vers la racine (bottom-up) lorsque l'on part d'une description structurelle d'un circuit.

4.2.2.1 Méthode top-down

On utilise la formule de Shannon. Par exemple, soit *f(a,b,c)=a'bc' + ac* avec *ord(a)<ord(b)<ord(c)*

- Si l'on fait l'expansion par rapport à *a* on obtient la Figure 4.3 a
- En faisant l'expansion par rapport à *b* on obtient la Figure.4.3.b
- En faisant l'expansion par rapport à *c* on obtient la Figure 4.3.c .

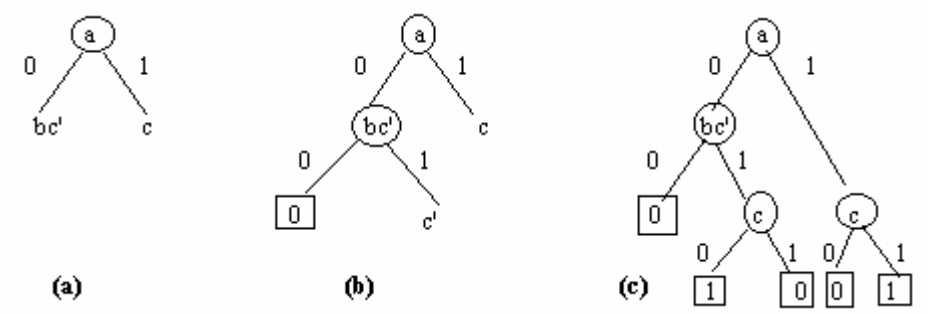


Figure 4.3: Construction du BDD de *f(a,b,c)=a'bc'+ac*

4.2.2.2 Méthode bottom-up :

Définition 4.8 *Le niveau d'une formule est défini par :*

- *les variables d'entrée sont de niveau 0.*
- *chaque sous-formule f = g <op> h a un niveau égal à Max(niveau(g),niveau(h)) + 1.*

La procédure suivante permet de construire le BDD d'une fonction *f* de *n* variables :

- Construire les BDD des variables.
- Construire les BDDs des formules de niveau 1 à l'aide de la fonction :

  Appliquer (*f1* : BDD, *f2* : BDD, opérateur <*op*>) : BDD.
- Répéter l'étape 2 jusqu'à ce que tous les niveaux soient considérés.

Par exemple si $f = x.y + z$: on construit les BDD de *x, y* et *z* puis celui de $x.y$ puis celui de $x.y + z$.

4.2.2.3 Opérations Logiques Entre deux Fonctions Représentées par BDD

Les opérations logiques applicables sur les BDD ont été définies par Bryant. L'algorithme permettant de déterminer le BDD résultant d'une opération logique entre deux fonctions est basé sur l'application récursive du théorème de Shannon selon ce qui suit :

Etant donné deux fonctions *F* et *G*, un opérateur *Op* et un ordre des variables donné. Si nous utilisons l'expansion de Shannon nous aurons :

$$F<Op>G = x_i.(F/x_i=0 \ <Op> \ G/x_i=0) + x_i.(F/x_i=1 <Op> G/x_i=1)$$

Le BDD résultant d'une opération logique entre deux fonctions peut être conçu à partir des deux BDD originaux en appliquant la procédure suivante :

Considérer les sommets racines des BDD. En fonction de la nature de ces deux sommets, appliquer récursivement l'une des règles suivantes. Itérer le processus jusqu'à la génération des sommets terminaux.

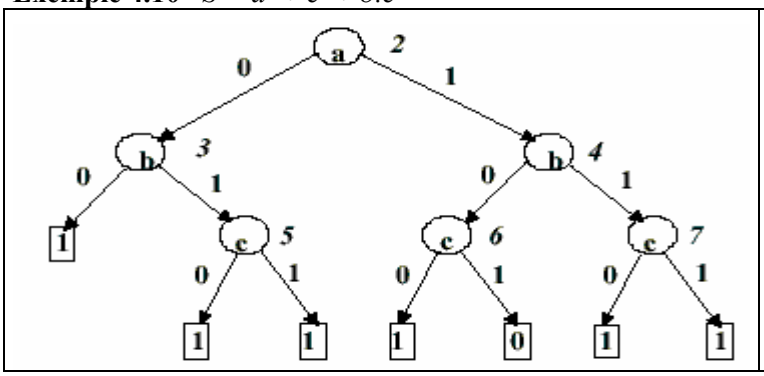
- *Règle1:* Si les deux sommets node_f et node_g sont des sommets terminaux alors le sommet résultant est un sommet terminal de valeur Valeur(node_f) <Op> Valeur(node_g).
- *Règle2 :* Si le sommet node_f est un sommet terminal mais pas le sommet node_g, créer le sommet node_g en lui associant comme fils droit le résultat de la comparaison (node_g, fils droit de node_f) et comme fils gauche le résultat de la comparaison (node_g, fils gauche de node_f).
- *Règle 3 :* Si ni le sommet node_f ni le sommet node_g ne sont des sommets terminaux alors :
- *Règle 3.1* : Si node_f et node_g représentent la même variable, créer un sommet représentant la variable en lui associant comme fils droit le résultat de la comparaison (fils droit de node_g, fils droit de node_f) et comme fils gauche le résultat de la comparaison (fils gauche de node_g, fils gauche de node_f).
- *Règle3.2* : Si node_f et node_g ne représentent pas la même variable, créer un sommet S représentant la variable intervenant la première dans l'ordonnancement considéré (S=min[ord(node_f), ord(node_g)]) en lui associant comme fils droit (gauche) le résultat de la comparaison entre le fils droit (gauche) de min[ord(node_f), ord(node_g)] et l'autre sommet (max[ord(node_f), ord(node_g)]).

Remarque 4.9 Etant donné deux fonctions F et G dont les BDD respectifs ont n et m sommets. En termes d'appels récursifs, la complexité de la procédure d'application d'une opération entre ces deux BDD est 2.n.m. La figure suivante récapitule le fonctionnement de la procédure

4.3 Implantation Informatique Des BDDs

La figure 4.4, nous donne un aperçu sur la manière utilisée pour implémenter les BDD en mémoire. Il suffit pour cela d'utiliser une table à trois entrées. La première stockera les variables (nœuds). Dans la deuxième et la troisième on mettra, les nœuds de référence pointés par ces variables.

Exemple 4.10 $S = a' + c' + b.c$



	0	-	-
	1	-	-
2	a	3	4
3	b	1	5
4	b	6	7
5	c	1	1
6	c	0	1
7	c	1	1

Figure 4.4: Représentation du BDD en Mémoire

4.3.1 Algorithme De Réduction Des BDDs

Principe : suppression des sommets redondants et les sous-graphes isomorphes

Algorithme Suppression_Sommets

Faire *id(v) = 0* pour chaque feuille *v* avec *valeur(v) = 0* ;

Faire *id(v)* = 1 pour chaque feuille *v* avec *valeur (v)* = 1 ;

Initialiser le ROBDD avec 2 feuilles avec *id =0* et *id* = 1 ;

Nextid = 1; **** Nextid = identificateur suivant disponible

Pour (*i* = *n* to 1 avec *i*=*i*-1) { *V(i)* = { *v* ∈ *V* / *niveau(v)* = *i* };

Pour chaque *v* ∈ *V(i)* { if (*id(gauche(v))* = *id (droit(v)*)

{ *id(v) = id(gauche(v))*

enlever *v* de *V(i)*}

else *clef(v) = id (gauche(v)) , id (droit(v))*

/* la clef est définie comme la paire d'identificateurs des fils*/ }

ancienne_clef = 0 , 0;

Pour chaque *v* ∈ *V(i)* trié selon la clef {

if *clef(v) = ancienne_clef*

id(v) = nextid /* le graphe de racine v est redondant */

else { *Nextid++;*

id(v)=Nextid;

ancienne_clef = clef (v);

/*ajouter *v* au ROBDD avec des arcs allant aux sommets dont l'*id* est égale à ceux de *gauche(v)* et *droit(v)*) }}}

4.4 Manipulation Des ROBDDs

Définition 4.11 *Deux BDDs D et D' sont isomorphes s' il existe une application δ de sommets de D sur les sommets de D' tel que pour tout sommet v : si δ(v) = v', alors v et v' sont des sommets terminaux avec value(v) = value(v'), ou v et v' sont des sommets non terminaux avec index(v) = index(v'), δ (low(v)) = low(v') et δ(high(v)) = high(v').*

4.4.1 L'approche Depth-First search (profondeur d'abord)

Les algorithmes typiques de BDD sont basés sur la procédure Depth-first search introduite par Bryant. Nous présentons trois algorithmes importants.

Le premier algorithme calcul n'importe quelle opération logique de base, par exemple AND, OR,…. Bryant a montré que la complexité de ces opérations est quadratique dans la taille du graphe des arguments des entrés.

Cet algorithme prend l'opérateur logique (*op*) et ses deux opérandes (*f* et *g*), et retourne un BDD comme résultat de l'opération *f op g*. Le résultat BDD est construit par l'exécution récursive de la

décomposition de Shannon, dans la manière Depth-first search. Cette décomposition récursive s'arrête quand le nouvel opérateur crée est un cas terminal (*ligne 1)* ou quand le résultat est disponible dans la table calculée (*ligne 2)*. La *ligne 3* détermine le top variable de *f* et *g*. Les *lignes 4* et *5* exécutent récursivement la décomposition de Shannon sur le cofacteur. A la fin de cette décomposition récursive, l'étape de réduction (*ligne 6*) assure que le résultat est un BDD réduit. Le BDD est ainsi inséré dans la table *unique* (*lignes 7-11*) et finalement l'opérateur est inséré dans la table *calculée* avec son résultat.

```
Procedure dfs-op(op,f,g)
/*compute f op g, where f and g are two BDDs and op is a logical
operator*/
1   if (terminal case) return simplified result ;
2   if the operation (op,f,g) is in computed cache , return result found in
cache;
3   let v be the top variable of f and g
4   r0← df_op(op,f|v←0, g|v←1)
5   r1← df_op(op,f|v←0,g|v←1);
6   if (r0 = = r1) return r0
7   b← BDD node (v,r0,r1);
8   result ← lookup (unique table, b)
9   if(BDD node b does not exist in the unique table )
10  insert b into unique table
11  r ←b;
12  insert this operation and its result into computed table
13  return r
```

Figure 4.5: La procédure Depth-first Search de Bryant

Le deuxième algorithme est appelé le produit relationnel (*relational product*). Celui-ci calcule $\exists v. f \wedge g$. L'idée de base est que le produit $f \wedge g$ sélectionne l'ensemble des transitions valides et la quantification existentielle $\exists v$. La complexité de cet algorithme est NP-hard.

Le troisième algorithme est appelé *restrict* est un algorithme utilisé pour minimiser la représentation BDD d'une fonction booléenne par le choix des valeurs appropriées et ne se soucie pas de l'espace utilisé d'où son appellation en anglais (*don't care-space*).

Procedure RP(v,f,g)

/*calcul du produit relationnel : $\exists v.f \wedge g$, ou *v est l' ensemble de* variables à quantifier et *f* et *g* sont deux BDDs*/

if (cas terminal) return simplified result

if the result of (RP,*v,f,g*) is cached, return the result

let τ be the top variable of *f* and *g*

r0 ← RP($v,f|_{\tau\leftarrow 0},g|_{\tau\leftarrow 1}$) /*Shannon expansion on 0-cofactors*/

if (τ ∈*v*) /* existential quantification on τ ≡ OR (*r0,* RP($v,f|_{\tau\leftarrow 1},g|_{\tau\leftarrow 1}$*/

if (r0 ==1)/* OR (*1,* RP($v,f|_{\tau\leftarrow 1},g|_{\tau\leftarrow 1}$) ≡ 1*/ r←1

else *r1*← RP($v,f|_{\tau\leftarrow 1},g|_{\tau\leftarrow 1}$)/* Shannon expansion on 1-cofactors*/

r←df_op(OR,*r0,r1*)

else *r1*← RP($v,f|_{\tau\leftarrow 1},g|_{\tau\leftarrow 1}$)/* Shannon expansion on 1-cofactor*/

r ←reduced, unique BDD node for(τ,*r0,r1*)

return *r*

Procedure restrict (f,c)

*compute care-space optimization :

* if c then f else choose some value to minimize the result

*Precondition : $c<>0$. */

if(f is constant) or (c==1) return f

if the result of(restrict,f,c) is cached, return the result

let τ be the top variable of f and c

if(τ is not the top variable of f) /*c has variables not in f. Quantify, then recurse*/

$r \leftarrow$ restrict(f,df_op(OR, $c|_{\tau\leftarrow 0}$, $c|_{\tau\leftarrow 1}$))

else if ($c|_{\tau\leftarrow 0}$= = 0) /* don't care about 0-branch*/

$r \leftarrow$ restrict($f|_{\tau\leftarrow 1}$, $c|_{\tau\leftarrow 1}$))

else if ($c|_{\tau\leftarrow 1}$= = 0) /* don't care about 1-branch*/

$r \leftarrow$ restrict($f|_{\tau\leftarrow 0}$, $c|_{\tau\leftarrow 0}$))

else /*normal Shannon decomposition */

$r0 \leftarrow$ restrict($f|_{\tau\leftarrow 0}$, $c|_{\tau\leftarrow 0}$)) /*0-cofactors*/

$r1 \leftarrow$ restrict($f|_{\tau\leftarrow 1}$, $c|_{\tau\leftarrow 1}$)) /*1-cofactors*/

$r \leftarrow$ reduced, unique BDD node for (τ,$r0$,$r1$)

return r

4.4.2 L' Approche Breadth-First Search

La deuxième approche qui est aussi bien connue et utilisée pour la construction du BDD est l'approche Breadth-First search. Dans cette approche, l'expansion de Shannon pour une opération de niveau supérieur est exécutée de la variable la plus haute (position) vers celle qui possède le rang le plus bas. Toutes les opérations avec le même ordre de variables seront étendues en même temps et de la même façon. La phase de réduction est exécutée de bas en haut selon l'ordre des variables et toutes les variables de même ordre seront étendues en même temps.

Pour illustrer le processus de construction du BDD avec l'algorithme breadth-first, on utilise une approche graphique pour visualiser le processus de construction du BDD. Figure (4.6) illustre l'expansion de Shannon pour l'opération f *op* g. Dans la partie gauche de cette figure, l'opérateur est représenté avec un nœud d'opérateur lequel fait référence aux représentations BDD de f et g comme des opérandes de cet opérateur. La partie droite de cette figure montre l'expansion de Shannon de cet opérateur et ou le nœud original fera référence au top variable τ. Les deux nouveaux nœuds d'opérateur crées représentent l'opération sur les cofacteurs-0 et cofacteurs-1.

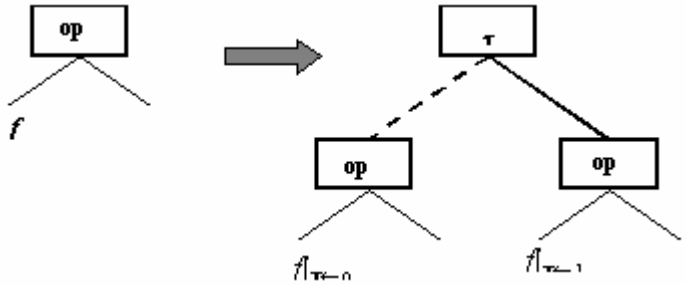


Figure 4.6 : Représentation graphique de la décomposition de Shannon

La Figure (4.6) illustre les deux règles utilisées dans la phase de réduction. La Figure (4.7.a) montre le cas où les résultats de deux branches (bdd0 et bdd1) sont distincts. La Figure (4.7.b) illustre le cas où les deux branches sont les mêmes et donc le BDD résultant ne dépend pas de la variable τ.

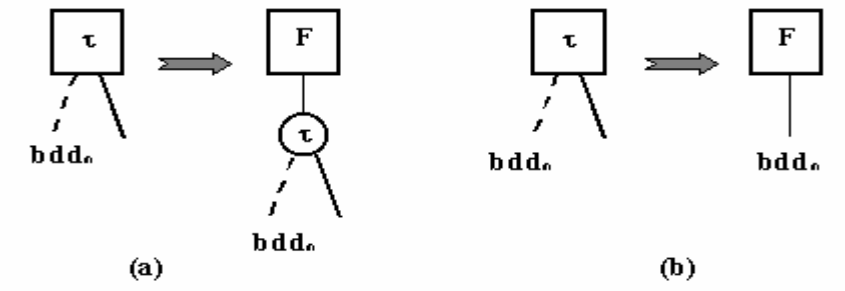


Figure 4.7 : Opération de La réduction

4.5 Algorithme ITE

On peut manipuler de façon simple les ROBDD à l'aide de la fonction ITE(*f,g,h*) (If Then Else) qui est définie par : $ite(f,g,h) = f.g + f'.h$ (i.e. If *f* Then *g* Else *h*)

Soit *x* la première variable (i.e. *Ordre(x) = 1*) de *f,g* et *h*.

Soit $z = ite(f,g,h)$. La fonction *z* est associée au sommet de variable *x* et dont les fils implantent $ite(f_x, g_x, h_x)$ et $ite(f_{x'}, g_{x'}, h_{x'})$. En effet :

$$z = x.Z.x + x'.Z.x'$$
$$= x(f.g + f'.h)x + x'(f.g + f'.h)x'$$
$$= x(f_x . g_x + f'_x . h'_x) + x'(f_{x'} . g_{x'} + f'_{x'} . h_{x'})$$
$$= ite(x, ite(f_x, g_x, h_x), ite(f_{x'}, g_{x'}, h_{x'}))$$

En plus du calcul récursif de la fonction *ite*, nous utilisons des identités remarquables pour éviter de les recalculer chaque fois. Ces identités sont consignées dans ce qui suit:

$ite(f, 1, 0) = f$; $ite(1, g, h) = g$; $ite(0, g, h) = h$; $ite(f, g, g) = g$; $ite(f, 0, 1) = f'$

De plus toutes les opérations habituelles peuvent être traduites en termes de l'opérateur *ite* :

Table 4.8: Fonctions Booléennes Pré-calculées et Exprimées sous forme de la fonction ITE

Expression	Forme Equivalent
0	0
1	1
$f.g$	$ite(f,g,0)$
$f.g'$	$ite(f,g',0)$
f	f
$f'g$	$ite(f,0,g)$
$f \oplus g$	$ite(f,g',g)$
$f+g$	$ite(f,1,g)$
$(f+g)'$	$ite(f,0,g')$
f'	$Ite(f,0,1)$
$f+g'$	$ite(f,1,g')$
$f'+g$	$ite(f,g,1)$
$(f.g)'$	$Ite(f,g',1)$

```
L'Algorithme ite( f,g,h) {
if (cas terminal) {  /* c-à-d 0 ou 1 figurant dans la table calculée */
return ( r = résultat trivial);
   } else if ( la table-calculee a une entrée ( (f,g,h),r) ) {/* déjà calculé */
                return (r à partir de la table-calculee) ;
          }else {
                x = variable top de f, g, h ;
                t = ite (fx, gx, hx);
                e = ite (fx', gx', hx');
                if ( t == e)          /* fils gauche et droit isomorphes */
                return (t);
                r = trouve-ou-ajoute-dans-Table (x, t, e);  /* ajout r à table */
                mise-a-jour-Table-calculée avec ( (f,g,h), r);
                return (r);
                }
           }
```

Figure 4.9: L'algorithme du Calcul de la Fonction ITE

Cet algorithme utilise deux tables : Table et Table calculée. **Table** est un tableau dont chaque ligne représente un sommet (c-à-d le tableau précédent). Ses colonnes représentent l'identificateur du sommet, le nom de la variable associée, l'identificateur du fils gauche et l'identificateur du fils droit.

4.6 Ordre Des Variables

Un des problèmes liés aux OBDDs qui a été bien mis en valeur par Bryant [Bry86] est l'ordonnancement des variables de la fonction. En effet, pour une même fonction suivant l'ordre que l'on prend sur les variables, la taille de l' OBDD peut passer de l'exponentiel au linéaire (Figure 4.10).

Il y a plusieurs approches utilisées pour le classement des variables. On peut citer quelques-unes :

A*pproche dynamique* [Rud93] : Cette approche consiste à faire des modifications directement dans l'OBDD. En effet, les ordres des variables dans l'OBDD ne sont plus statiques mais dynamiques (changeant au fur et à mesure de la construction de l' OBDD).

Approche échange de variables [ISY91] : Propose de pondérer la permutation à l'aide du concept d'énergie. Le déplacement des variables dans l'OBDD est assuré, comme le ferait un électron sur les couches électroniques d'un atome.

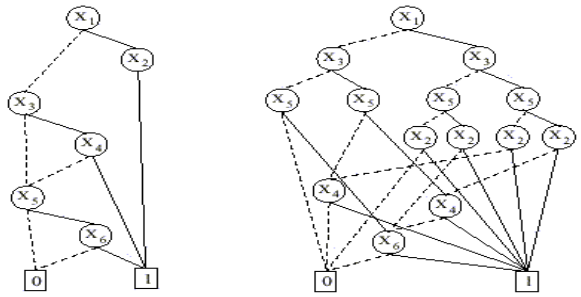


Figure 4.10: L'influence des ordres sur la taille du BDD, appliquée à la formule « $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$ »

4.7 Implantation des techniques de représentation et de minimisation des systèmes concurrents

L'outil que nous avons développé pour prendre en charge la normalisation au sens de minimisation des descriptions symboliques des systèmes concurrents, nous fournit un graphe ROBDD (Reduced Ordered Binary Decision Diagram) (Figure 4.11) plus le nombre de nœuds de départ et de fin, le temps d'exécution ainsi que d'autres informations sans grande importance. Le fonctionnement d'un tel outil est très simple, il suffit de lui communiquer la formule logique symbolique qui modélise la structure et le comportement du système en question.

L'outil a été écrit en Java 1.2 et constitue une partie importante de notre système de vérification (Module) épousant par la même occasion tous les éléments qu'on vient de voir, à savoir l'algorithme ITE, l'algorithme de recherche DFS, la méthode de représentation minimale des BDD en mémoire etc. Il constitue un autre apport d'exactitude en plus de l'outil SAT construit utilisant le concept des méthodes génétiques. Pour avoir une idée sur son emplacement dans l'architecture générale du système global, il suffit de jeter un coup d'œil sur les modèles de vérification des programmes CTL et du Mu-calcul, c'est justement lors des calculs des point-fixes.

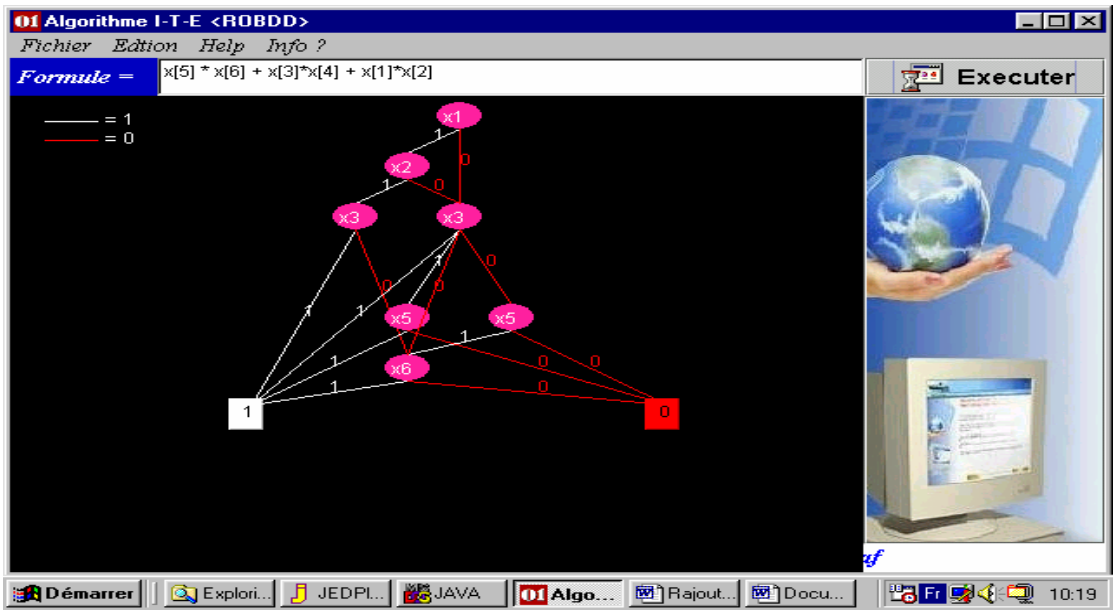


Figure 4.11: Interface Graphique du Module ROBDD

4.8 Conclusion

Dans ce chapitre nous nous sommes intéressés à un modèle de représentation et de simplification des réseaux booléens qui pourraient modéliser les systèmes réactifs. A travers cette mécanique, nous avons tenté de répondre à certaines exigences exprimées durant tout le projet. En effet :

- Nous avons montré le bien fondé de l'utilisation de l'utilisation du théorème de Shannon comme étant l'outil de base des BDD qui sans lequel l'algorithme ITE ne pourrait fonctionner.
- Nous avons donné toutes les informations pour la bonne représentation et la manipulation des fonctions booléenne à travers les algorithmes RESTRICT et EXPAND.
- Nous avons montré dans une telle philosophie deux algorithmes peuvent être utilisés, l'un basé sur l'algorithme DFS et l'autre sur BFS.
- Finalement, nous nous sommes intéressés à l'implémentation des BDD, puis des OBDD et des ROBDD utilisant une procédure intelligente basée sur la représentation de ces derniers en mémoire de façon optimale.

CHAPITRE 5

Théorie du Model-Checking

5.1 Introduction

Les logiques temporelles les plus connues d'entre elles sont CTL et LTL [Pnu77, CE81, Eme96]. Elles sont très utilisées pour la spécification de systèmes réactifs et en général concurrent. Elles utilisent un modèle simple et disposent d'algorithmes de model-cheking très puissant [QS82, EC82, CES86, Eme83]. Les logiques temporelles, sont utilisées pour exprimer des propriétés relatives à des comportements très complexes des systèmes critiques telles que : pour tout chemin, à tout moment sur des chemins, si une alarme se déclenche, alors elle conduit inévitablement à une réponse, ce qui s'écrit *AG(alarme* → AF *réponse*).

Les logiques temporelles CTL, LTL ou d'autres de même types tel que le mu-calcul [SE84, Wal93, Koz83, KPS84] et TPA [Lam94], comme elles revêtent plutôt un aspect *qualitatif* ne permettant pas (ou pas facilement) d'exprimer des aspects *quantitatifs* sur le temps. Cette limitation est parfois gênante : par exemple, si le fait de vouloir exprimer « existe-t-il une exécution menant à une certaine situation *avant un délai maximal* disons *critique.*

Les logiques temporisées dites aussi temps-réel (timed logic ou simplement real-time logic)[AH92, AD94] permettent d'exprimer ce genre de propriétés pour un modèle de temps continu, discret ou carrément hybride, sur des automates temporisés. Prenons l'exemple qu'on vient de citer et exprimons la propriété « la réponse se produit moins de 5 unités de temps après l'alarme », celle-ci s'écrit $AG(alarme \rightarrow A F_{\leq 5}$ réponse). De telles logiques sont actuellement mises en oeuvre dans des outils tels que HyTech [HHW97], Uppaal [UPPAAL], ou Kronos [Yov97], la complexité de tel model-checking est au moins *PSPACE-dur* [ACD93, AL99].

L'un des avantages des logiques est qu'elles reposent sur un modèle simple et général appelé *structure de Kripke*, c'est-à-dire un automate pour lequel chaque état vérifie un certain nombre de propositions atomiques. Un autre avantage de ces logiques est l'existence d'algorithmes de model-cheking efficaces.

Formellement, une *Structure de Kripke* est aussi un automate ou simplement un graphe. Il est admis que les sommets (nœuds ou états) dans les structures de Kripke, chacun d'eux doit avoir au moins un sommet successeur ce qui pousse à ce que la relation de transition soit totale.

Supposons que AP soit un ensemble fini et non vide de propositions atomiques utilisées pour décrire les propriétés du système d'états.

Définition 5.1 (Kripke structure) *Une structure de Kripke est un quadruplet* $M = (S, \rho, s_0, L)$ *ou :*

- S *est l'ensemble fini des états du système,* $\rho \subseteq S \times S$ *est la relation de transition qui satisfait la condition* $\forall s \in S : \exists s' \in S : (s, s') \in \rho$,

- $L : S \rightarrow 2^{AP}$ *est une fonction d'étiquetage qui associe à chaque état un ensemble de propositions atomiques.*

Définition 5.2 [Chemin] *Un chemin définit sur une structure de Kripke est une séquence infinie d'états* $(s_0, s_1, s_2, \ldots) \in S^{\omega}$ *tel que* $(s_i, s_{i+1}) \in \rho$ *pour tout* $i \geq 0$.

Quant on utilise les structures de Kripke comme formalisme, on doit considérer uniquement les chemins (exécutions) qui commencent à partir des états initiaux (pour le moment l'on ne considérera qu'un seul état initial). Pour un chemin donné qu'on notera $\sigma = s_0\ s_1\ s_2\ \ldots$ et un indice (entier) $i \geq 0$, $\sigma[i]$ dénotera le $(i+1)^{ième}$ état de σ. En outre on utilisera σ^i pour dénoter le suffixe de σ obtenu par suppression de ces i états, c'est-à-dire $\sigma^i = s_i\ s_{i+1}\ s_{i+2}\ \ldots$Nous notons que $\sigma^i[j] = \sigma[i+1]$ et en particulier $\sigma^0 = \sigma$. Pour les chemins d'une certaine régularité on écrira $\sigma = (s_0)^{\omega}$ au lieu de $\sigma = s_0\ s_0\ s_0\ s_0\ \ldots$ L'ensemble des chemins qui commencent à partir d'un état s sont dénotés *Paths*(s). Noter aussi que *Paths*(s) = $\varnothing$ si $s \notin S_0$.

On définit aussi la fonction d'étiquetage (renommage) L utilisée pour projeter tout chemin de la structure comme étant une séquence infinie de labels sur les états. Ces séquences de labels peuvent être considérées comme des mots infinis sur l'alphabet des lettres (2^{AP}). Le plus souvent tout sous-ensemble

L de $(2^{AP})^{\omega}$ sera considéré comme un langage des mots infinis sur l'alphabet 2^{AP}. Dans la suite nous utilisons $\mathbf{L}_M$ pour définir l'ensemble des mots générés par les exécutions d'une structure de Kripke M. Nous dirons alors que $\mathbf{L}_M$ est généré par M.

L'exemple suivant démontre tous les concepts décrit ci-dessus.

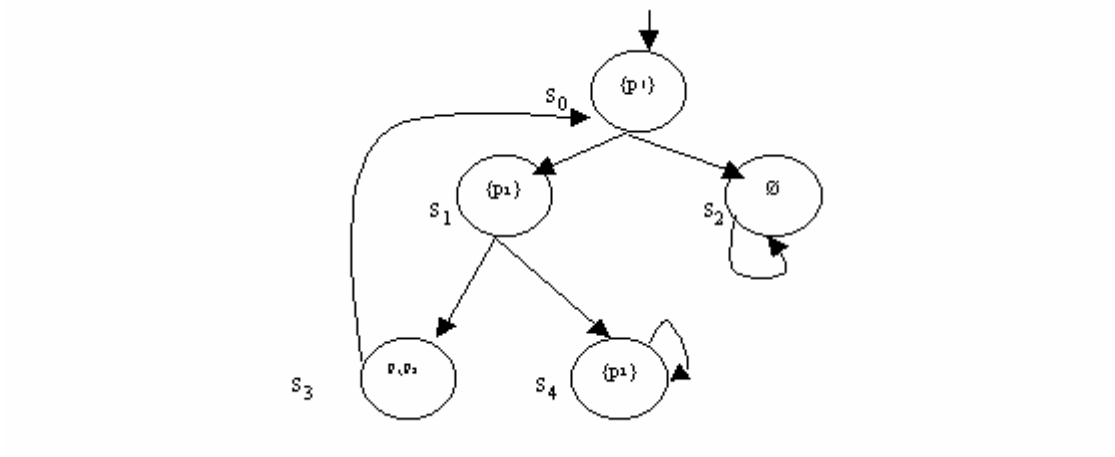


Figure 5.1 : Une structure de Kripke simple

Dans cette structure nous avons les éléments suivants :

$AP = \{p_1, p_2\}$; $M = (S, \rho, S_0, L)$ et tels que ;

$S = \{s_0, s_1, s_2, s_3, s_4\}$; $\rho = \{(s_0,s_1), (s_0,s_2), (s_1,s_3), (s_1,s_4), (s_2,s_2), (s_3,s_0), (s_4, s_4)\}$

$s^0 = s_0$; $L(s_0) = \{p_1\}$; $L(s_1) = \{p_1\}$; $L(s_2) = Ø$; $L(s_3) = \{p_1, p_2\}$, $L(s_4) = \{p_2\}$.

Les deux exécutions de M sont : $(s_0,s_1,s_3,s_0,s_1,s_3, \ldots)$ et $(s_0,s_2,s_2,s_2, \ldots)$ correspondent aux séquences infinies suivantes :

$(L(s_0), L(s_1), L(s_3), L(s_0), L(s_1), L(s_3), \ldots) = (\{p_1\}, \{p_1\}, \{p_1, p_2\}, \{p_1\}, \{p_1\}, \{p_1, p_2\}, \ldots)$

et $(L(s_0), L(s_2), L(s_2), L(s_2), \ldots) = (\{p_1\}, Ø, Ø, Ø, \ldots.)$

Le chemin $(s_1,s_3,s_0,s_1,s_4,s_4,s_4, \ldots)$ n'est pas une exécution car il ne commence pas par l'état s_0.

La séquence infinie $(\{p_2\}, \{p_1\}, \{p_2\}, \{p_1\}, \{p_2\}, \{p_1\}, \ldots)$, n'appartient pas au langage L_M, M ne possède pas d'exécution de ce type.

Dans ce qui suit, deux types de logiques temporelles seront présentées, à savoir :

5.2 Modèles Linéaires

5.2.1 La Logique Temporelle Linéaire (LTL)

Les logiques temporelles permettent de représenter le comportement des systèmes réactifs au moyen de propriétés qui décrivent l'évolution du système. Dans le cadre des logiques temporelles linéaires le temps se déroule, comme son nom l'indique, linéairement. En clair, on spécifie le comportement attendu d'un système sans réelle possibilité de spécifier plusieurs futurs possibles. Dans la section suivante nous présentons une logique temporelle linéaire proposée par Manna et Pnueli [Pnu86]. Cette logique a depuis été grandement améliorée d'un point de vue expressif par l'ajout de nombreux opérateurs [MP92, MP95]. Mais, les principes de base restent les même et dans un premier temps nous nous contenterons de ce langage.

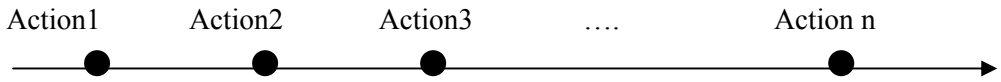


Figure 5.2 : Le déroulement temporel d'un système dans une logique linéaire temporelle

La signification de formule dans cette logique est définit par une relation de satisfaction (dénotée |=) entre un chemin σ et une formule de la logique (LTL, PLTL, …). Le concept est $\sigma \models \Phi$ si est seulement si Φ est valide pour σ.

L'ensemble des formules de la logique temporelle linéaire est défini par induction sur la structure des formules comme suit. On admettra que *AP* représente l'ensemble des formules atomiques de la logique propositionnelle.

Définition 5.3 (Linear Temporal Logic) *L'ensemble des formules de la logique temporelle linéaire est construit des chaînes de caractères finies qui peuvent être obtenues par application des règles suivantes :*

- *Toutes les propositions atomiques de AP sont des formules de LTL.*
- *Si φ est une formule de LTL alors $\neg\varphi$ est une formule de LTL.*
- *Si φ et Ψ sont des formules de LTL, alors $(\varphi \vee \Psi)$ est une formule de LTL.*
- *Si φ est une formule de LTL, alors $X\varphi$ est une formule de LTL.*
- *Si φ et Ψ sont des formules de LTL, alors $(\varphi \, U \, \Psi)$ est une formule de LTL.*

La sémantique de la logique temporelle linéaire est définie sur les séquences de sous-ensembles des propositions atomiques *AP* comme suit :

Définition 5.4 (Sémantique de LTL) *Soit $p \in AP$ une proposition atomique, σ un chemin infini et Φ, ψ des formules de LTL. La relation de satisfaction $\models$ est définie par :*

- $\sigma \models p$ *ssi* $p \in L(\sigma[0])$, *le premier élément de la séquence.*
- $\sigma \models \neg\varphi$ *ssi non* $(\sigma \models \varphi)$.
- $\sigma \models (\varphi \vee \psi)$ *ssi* $\sigma \models \varphi$ *ou* $\sigma \models \psi$.
- $\sigma \models X\varphi$ *ssi* $\sigma^1 \models \varphi$.
- $\sigma \models (\varphi \, U \, \psi)$ *ssi il existe* $i \geq 0$ *tel que* $\sigma^i \models \psi$, *et pour tout* $0 \leq j < i$, $\sigma^i \models \varphi$.

D'autres opérateurs logiques et booléens peuvent être définit comme abréviation dans le sens usuel par : $T \overset{déf}{\equiv} (p \vee \neg p)$ pour toute proposition atomique.

Soit $p \in AP$, on définit T : la constante booléenne « True » ; et son cnjugué $\perp \overset{déf}{\equiv} \neg T$ (« false »)). On définit en outre certaines équivalences en rapport avec la logique propositionnelle.

Intuitivement, de la définition 5.4, la satisfaction des formules de LTL dépend seulement du premier sous-ensemble de la logique propositionnelle dans la séquence, seulement si la formule ne contient aucun opérateurs X ou U. Dans le cas contraire, la satisfaction de la formule dépendra de la séquence entière, du moins d'une partie d'elle. Pour mieux comprendre la notion de séquence d'un chemin et le raisonnement sur les opérateurs temporels tel que le X (Next) et le U (Until), par exemple, la formule X φ est satisfaite dans une séquence infinie σ si la formule φ tient compte de la sous séquence σ^1 de σ. Donc l'opérateur X φ est dit opérateur « Next time » ou bien opérateur du futur immédiat. La formule (φ U ψ) est satisfaite si et seulement si ψ est valide maintenant ou bien dans quelque temps dans le future, et φ est valide jusqu'à ce que ψ le devient à son tour. D'autres opérateurs peuvent être dérivés de la définition de l'opérateur U.

5.2.1.1 Opérateurs Temporels Auxiliaires

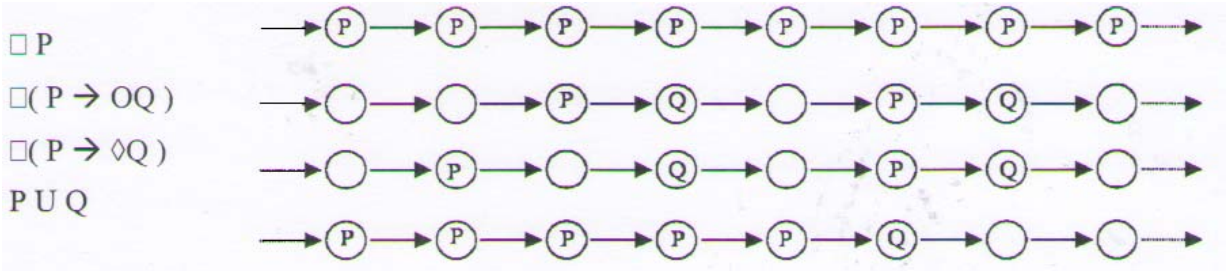
Pour soulager la spécification de propriétés appropriées, quatre opérateurs auxiliaires peuvent être introduits. Ces opérateurs sont définis en utilisant certains opérateurs déjà définis auparavant. L'opérateur temporel □ (qu'on prononce Always « toujours ») et l'opérateur ◊ (prononcé « eventually » ou Future) sont définis comme suit :

L'opérateur $\Diamond\varphi \overset{déf}{\equiv} (T\ U\ \varphi)$ (« eventually »),

$\Box\varphi \overset{déf}{\equiv} \neg\Diamond\neg\varphi$ (« always »), et l'opérateur « release »

$(\varphi\ R\ \psi) \overset{déf}{\equiv} \neg\ (\neg\varphi\ U\ \neg\psi)$.

Exemple 5.5 (vérification de formule LTL)



Soit $AP = \{p_1, p_2\}$, et $\sigma = (y_0, y_1, y_2, \ldots) = (\{p_1\}, \{p_1\}, \{p_1, p_2\}, \{p_1\}, \{p_1\}, \{p_1, p_2\}, \ldots)$ une séquence infinie sur 2^{AP}. Vérifions que cette séquence vérifie la formule LTL $\Box\Diamond p_2$ utilisée pour dire que p_2 est toujours éventuellement vraie dans la séquence, qui signifie aussi que p_2 est vraie dans la séquence infiniment souvent.

Avant de procéder à la vérification, on aborde le problème de la simplification sur la structure de la formule par réécriture, on obtient ainsi :

$\sigma \models \Box\Diamond p_2$

$\sigma \models \Box(T\ U\ p_2)$

$\sigma \models \neg\Diamond\neg(T\ U\ p_2)$

$\sigma \models \neg(T\ U \neg(T\ U\ p_2))$

$non(\sigma \models (T\ U \neg(T\ U\ p_2)))$.

D'après la sémantique de LTL, $\sigma \models (T\ U \neg(T\ U\ p_2))$ si il existe un $i \geq 0$ tel que $\sigma^i \models \neg(T\ U\ p_2)$, et pour tout $0 \leq j < i$, $\sigma^j = T$. Donc $\sigma \lrcorner T\ U \neg(T\ U\ p_2)$.

1. Il n'existe pas de $i \geq 0$ tel que $\sigma^i \models \neg(T\ U\ p_2)$ est vraie, donc $\sigma^i \models (T\ U\ p_2)$ est vraie pour tout $i \geq 0$.

2. Pour tout $i \geq 0$ tel que $\sigma^i \models (T\ U\ p_2)$ il existe un $0 \leq j < i$ pour lequel non ($\sigma^i \models T$).

En fait, le premier cas est vrai dans la séquence suivante. Avant cela nous devons considérer que pour tout $i \geq 0$, $\sigma^i \models T$:

$\sigma^i \models T$

$\sigma^i \models p_1 \vee \neg p_1$

$\sigma^i \models p_1$ ou $\sigma^i \models \neg p_1$

$\sigma^i \models p_1$ ou $non(\sigma^i \models p_1)$,

qui est vraie, puisque ou bien $p_1 \in y_i$ ou bien $p_1 \notin y_i$ pour tout sous-ensemble $y_i \in 2^{AP}$ et $i \geq 0$.

Nous montrons maintenant que pour tout $i \geq 0$, $\sigma^i \models T \, U \, p_2$. Notons que pour tout $k \geq 0$, $\sigma^{k+3} = \sigma^k$. Par définition, $\sigma^i \models T \, U \, p_2$ si et seulement s'il existe un $i' \geq i$ telle que $\sigma^{i'} \models p_2$ et pour tout $i \leq j < i'$, $\sigma^j \models T$. Nous avons montré auparavant et ceci pour tout $j \geq 0$. Nous savons aussi que $\sigma^2 \models (p_1, p_2\},\{p_1\},\{p_1\},\{p_1, p_2\}, \ldots) \models p_2$. Parce que $\sigma^{k+3} = \sigma^k$ pour tout $k \geq 0$, il s'ensuit que $\sigma^{2+3k} \setminus p_2$ et pour tout $i \geq 0$ il doit exister un $i' \geq i$ tel que $\sigma^i \setminus p_2$, alors $\sigma \setminus \Box\Diamond p_2$.

Dans certaines littératures, l'opérateur du futur est dénoté par F au lieu de $\Diamond$, l'opérateur « always » $\Box$ est dénoté par G pour dire globalement (« Global ») et O est remplacé par X. Donc on définira F est G par :

$$F \, \Phi \equiv T \, U \, \Phi$$

$$G \, \Phi \equiv \neg \, F \, \neg\Phi$$

L'opérateur U est appelé « the strong Until »(le plus fort). On définit quelque fois « the weak Until » (le plus faible) qui définit la situation que lorsque Φ « hold » continuellement ou bien jusqu'à ce que ψ « holds » pour la première fois ou à travers le chemin. Cet opérateur est défini par :

$$\Phi \; W \; \psi \; = \; G \, \Phi \; \vee \; (\Phi \, U \, \psi)$$

ou de façon équivalente nous aurons :

$$\Phi \; W \; \psi \; = \; F \, \neg\Phi \; \Rightarrow \; (\Phi \, U \, \psi)$$

Le dernier opérateur auxiliaire s'appelle « the release operator » dénoté R ; il est défini par :

$$\Phi \; R \; \psi \; = \; \neg \, (\neg\Phi \; U \; \neg\psi)$$

Intuitivement il est interprété comme suit :

La formule $(\Phi \, R \, \psi)$ est vraie sur le chemin σ si ψ est toujours vraie, la nécessité est qu'il est relâcher aussi vite que Φ devient valide.

Exemple 5.6 $G \, p \Rightarrow \neg X(\neg p \; \wedge G \, \neg p)$ est une tautologie.

Preuve :

$\sigma \models \neg G \, p \vee (\neg X(\neg p \; \wedge G \, \neg p))$ [par remplacement de l'implication] $\Leftrightarrow$

$\neg(\sigma \models G \, p \wedge X(\neg p \; \wedge G \, \neg p))$ [par sémantique de $\neg$] $\Leftrightarrow$

$\neg(\sigma \models G \, p \wedge \; \sigma \models X(\neg p \; \wedge G \, \neg p))$ [sémantique de la conjonction] $\Leftrightarrow$

$\neg((\forall j \geq 0.\sigma^j \models p) \wedge \; \sigma^1 \models \neg p \; \wedge (\forall j \geq 0.(\sigma^1)^j \models \neg p))$ [sémantique de G deux fois, conjonction et sémantique de X] $\Leftrightarrow$

$\neg((\forall j \geq 0.\sigma^j \models p) \wedge \; (\forall j \geq 0.\sigma^j \models \neg p))$ [par simple calcul].

Notons que toutes les étapes sont équivalentes, ce qui fait que le chemin inverse est aussi vrai.

5.2.1.2 Axiomatisation

La validité d'une formule LTL de la forme $\Phi \Leftrightarrow \psi$ peut être dérivée en utilisant la sémantique de LTL et en prouvant :

$\sigma \models \Phi$ si et seulement si $\sigma \models \psi$ sur tous les chemins.

Dans le même sens d'idée essayons de vérifier l'équivalence qui nous parait très importante puisqu'elle sera utilisée un peu plus tard, à savoir :

$\Phi \, U \, \psi \; \Leftrightarrow \; \psi \vee (\Phi \wedge X(\Phi \, U \, \psi))$

Considérons la partie droite, soit :

$\sigma \models \psi \vee (\Phi \wedge X(\Phi \, U \, \psi))$

$\Leftrightarrow$ [sémantique de $\vee$ et $\wedge$]

$(\sigma \models \psi) \vee (\sigma \models \Phi \wedge \sigma \models X(\Phi \, U \, \psi))$

$\Leftrightarrow$ [sémantique de X]

$(\sigma \models \psi) \vee (\sigma \models \Phi \wedge \sigma^1 \models \Phi \, U \, \psi)$

$\Leftrightarrow$ [sémantique de U]

$(\sigma \models \psi) \vee (\sigma \models \Phi \wedge (\exists j \geq 0. (\sigma^1)^j \setminus \psi \wedge \forall 0 \leq k < j. (\sigma^1)^k \models \Phi)$

$\Leftrightarrow$ [par calcul en utilisant $(\sigma^1)^j = \sigma^{j+1}$]

$(\sigma \models \psi) \vee (\exists j \geq 0. \sigma^{j+1} \models \psi \wedge \forall 0 \leq k < j. (\sigma^{k+1} \models \Phi) \wedge \sigma \models \Phi))$

$\Leftrightarrow$ [par calcul en utilisant $\sigma^0 = \sigma$]

$(\sigma \models \psi) \vee (\exists j \geq 0. \sigma^{j+1} \setminus \psi \wedge \forall 0 \leq k < j+1. (\sigma^k \models \Phi)))$

$\Leftrightarrow$ [par calcul en utilisant $\sigma^0 = \sigma$]

$(\exists j = 0. \sigma^0 \models \psi \wedge \forall 0 \leq k < j+1. \sigma^k \models \Phi \vee (\exists j \geq 0. \sigma^{j+1} \setminus \psi \wedge \forall 0 \leq k < j. (\sigma^k \models \Phi)))$

$\Leftrightarrow$ [par calcul des prédicat]

$(\exists j = 0. \sigma^j \models \psi \wedge \forall 0 \leq k < j+1. \sigma^k \models \Phi)$

$\Leftrightarrow$ [sémantique de U]

$\sigma \models \Phi \, U \, \psi$

5.3 Model-checking à la Volée

Rappelons-nous que pour représenter un système (programme, protocole, etc.), il nous fallait utiliser une certaine abstraction que nous avons appelé *model* qui est définit par la représentation M = (S, $\rightarrow$, L) qui peut être un LTS (labelled transition system) ou bien une structure de Kripke tels que définis auparavant. Nous reproduisons exactement les mêmes définitions pour S qui représente l'ensemble des états dans lesquels le système peut s'y trouver, $\rightarrow$ étant la relation de transition qui décrit comment le système évolue d'un état à un autre. Nous ne voulons pas pour le moment être très concis sur la nature des états du système. Ce qui nous importe c'est plutôt, la fonction étiquetage qui associe à chaque état un label, c'est-à-dire une valuation *v* que nous supposons pour le moment par mesure d'intégrité être une formule propositionnelle (booléenne). On supposera aussi que *v* est vraie si $s \in L(s)$, autrement *v* est faux en s. L'idée essentiellement qui nous intéresse pour le moment, c'est de pouvoir représenter le système et la spécification utilisant le même langage. Cette idée à été bénéfique puisqu'il existe dans la pratique un certain nombre d'implémentations qui fonctionnent selon ce principe. Les plus connues sont : SPIN développé au Bell Labs [Hol97, SPIN02] muni de son langage de programmation formelle [Ger97], qui est un outil très populaire utilisé pour la vérification des systèmes distribués. Il a reçu en 2001 le prestigieux prix (System Software Award) du meilleur logiciel de vérification qui est distribué par ACM (Association of Computing Machinery). Ce type d'outils procède en général sur la spécification au sens des automates de Büchi, un puissant concept théoriquement complet utilisé pour décrire le système et ses propriétés selon ce qui suit.

5.3.1 Les Automates de Büchi

Pour mieux appréhender la notion de vérification telle que développée jusqu'à présent, nous devons nous donner les moyens théoriques pour représenter les états de la spécification. Il s'agit donc de pouvoir représenter deux modèles M_1 et M_2 (model du système et model de la spécification), de les exécuter en parallèle de façon à ce que, chaque étape (exécution) dans M_1 (respectivement M_2) sera suivie d'une exécution dans M_2 (respectivement M_1).

L'une des solutions présente dans la théorie étant de composer les deux modèles (M1 et M2) pour en créer un troisième (M). Le problème qui se pose dans ce cas étant l'explosion combinatoire essentiellement lorsque l'on utilise des invariants sur les états. Pour remédier à un tel problème, il suffit d'adjoindre la fonction d'étiquetage sur les transitions et non plus sur les états. Intuitivement, un label sur une transition de s vers s' indiquera que le système empruntera cette transition uniquement que lorsque la condition sur le label soit vraie. Cette façon de traduire le problème permet maintenant de le faciliter et permettre ainsi d'exécuter en parallèle les deux systèmes (M_1 et M_2). C'est-à-dire, on passera d'un état vers un autre que lorsque deux transitions de M_1 et de M_2 soient syntaxiquement équivalentes (même label).

Avec ce nouveau type de raisonnement, nous avons besoins de prendre certaines libertés (respectivement restrictions) pour dire que seulement des exécutions d'un certain type doivent être autorisées (acceptées). Dans le même cadre d'idée c'est uniquement certains états qui seront notifiés par le titre *d'état accepté*, et qu'une exécution est dite *acceptable* si elle comporte infiniment plusieurs états acceptés.

Dans ce qui suit nous détaillons tous les points qui viennent d'être cités. A savoir la notion de modèle qui sera interprétée par un automate de Büchi, de l'exécution en parallèle de deux modèles qui n'est d'autre que l'intersection dans la théorie des ensembles et ainsi de suite.

5.3.2 Quelques définitions de Base

Définition 5.7 (Automate de Büchi)[VW86, Var87, SE84, EH00] *Un automate de Büchi est un quintuplé* $\mathcal{A} = (S, T, S_0, F, \Sigma)$ *où*

- S *est un ensemble d'états*
- $S_0 \subseteq S$ *est l'ensemble des états initiaux*
- $T \subseteq S \times \Sigma \times S$ *est l'ensemble des transitions*
- $F \subseteq S$, *un ensemble d'états finaux dit aussi états acceptés par l'automate*
- Σ *est un alphabet (alphabet de* $\mathcal{A}$*)*

Définition 5.8 (Critère d'acceptation de Büchi). *Un mot (chaîne de caractère, string) sur* Σ *est reconnaissable par* $\mathcal{A}$ *si la chaîne des états visités par l'automate en le lisant en partant du sommet* s_0*, passe par F infiniment souvent.*

Exemple 5.9 Le graphe de la Figure 5.3 représente l'automate de Büchi $\mathcal{A} = (S, T, S_0, F, \Sigma)$ dont les composantes sont définies comme suit :

$S = \{s_0, s_1, s_2\}$
$S_0 = \{s_0\}$
$F = \{s_2\}$
$\Sigma = P([p, q, r])$
$T = \{(s0 \xrightarrow{pq} s1), (s1 \xrightarrow{pq} s1), (s1 \xrightarrow{qr} s2), (s0 \xrightarrow{qr} s2), (s2 \xrightarrow{pq} s0)\}$

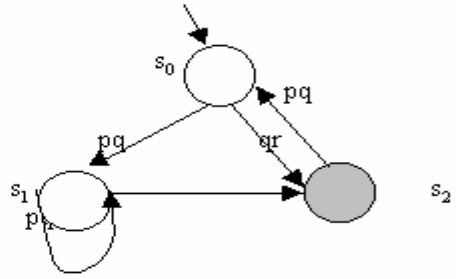


Figure 5.3 : Un automate de Büchi

Définition 5.10 *Une exécution dans* $\mathcal{A}$ *est définie par* $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ *avec* $s_0 \in S_0$.

Définition 5.11 *Une exécution est dite acceptée (reconnaissable) par l'automate si* $s_i \in F$ *infiniment plusieurs i.*

Dans ce qui suit on notera L(B), l'ensemble des mots sur Σ reconnaissable par l'automate B.

Exemple de mot (exécution acceptable) reconnaissable par l'automate de la Figure 5.3,

$s_0 \xrightarrow{\{p,q\}} s_1 \xrightarrow{\{q,r\}} s_2 \xrightarrow{\{p\}} s_0 \xrightarrow{\{p,q\}} s_1 \xrightarrow{\{q,r\}} s_2 \xrightarrow{\{p\}} \cdots$ donc le mot sera {p,q} {q,r} {p} {p,q} {q,r} {p} …

$s_0 \xrightarrow{\{p,q\}} s_1 \xrightarrow{\{p,q\}} s_1 \xrightarrow{\{p,q\}} s_1 \xrightarrow{\{p,q\}} s_1 \xrightarrow{\{p,q\}} s_1 \xrightarrow{\{p,q\}} \cdots$ est une exécution non reconnaissable par l'automate.

Exemple 5.12 On considère l'automate de Büchi de la Figure 5.4 ou F = $\{s_1\}$ et S = $\{s_0\}$. L'ensemble des mots reconnaissable par cet automate est de la forme $(a^*ba^*)^\omega$. Nous pouvons remarquer que les mots de la forme $(a^*ba^*)a^\omega$ ne sont pas reconnaissables pour cet ω-mots (un ω-mot est reconnaissable s'il contient une infinité de b).

Exemple 5.13

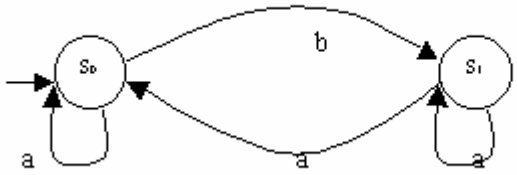


Figure 5.4 : Un Automate de Büchi

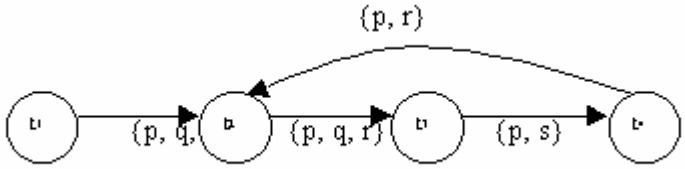


Figure 5.5 : Un Système de Transitions

Supposons que p, q, r, s, t soient des propositions atomiques. La satisfaction d'une formule de la logique temporelle linéaire (LTL) par le système (représenté par le chemin $t_1(t_2t_3t_4)^\omega$ est déterminée par le ω-mot

$$(1,1,0,0,1)[(1,1,1,0,0)(1,0,0,1,0)(1,0,1,0,0)]^\omega$$

est reconnaissable par l'automate de Büchi obtenu de l'automate de la figure en posant F = S et en transportant l'étiquetage de l'état vers la transition issue de cet état. On obtient alors l'automate de Büchi illustré à la figure suivante :

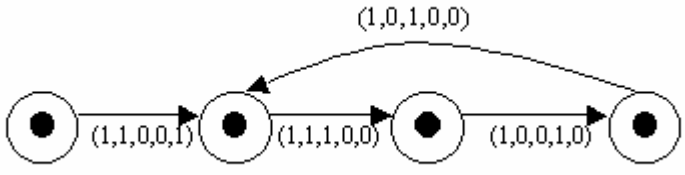


Figure 5.6 : Automate de Büchi obtenu de la figure 5.5

Le théorème suivant nous donne le moyen de faire correspondre à une formule de la logique temporelle LTL un automate de Büchi, il en va de même pour toute formule d'une logique linéaire.

Dans l'énoncé l'on utilise $|\phi|$ pour dénoter la longueur d'une formule ϕ et $|S|$ pour dénoter la cardinalité d'un ensemble S.

Théorème 5.14 (Wolper-Vardi-Sistla) (dans [GPV+95]) *Etant donné une formule de PLTL, on peut construire un automate de Büchi* $A_\phi = (S, T, S_0, F, \Sigma)$ *avec* $\Sigma = 2^{AP}$ *et* $|S| \leq 2^{O(|\phi|)}$ *tel que L(A)* soit *exactement l'ensemble de tous les modèles de* ϕ.

5.4 L'Algorithme du Model-checking d'une Formule de LTL

Soient A et S deux automates qui représentent respectivement le système et la spécification. Supposons que $L(A)$ et $L(S)$ définissent tous les comportements possibles respectivement du système et de la spécification. Le problème qui se pose maintenant c'est de se donner les moyens théoriques pour dire que le système A puisse satisfaire la spécification S (qui s'écrit $A \models S$), c'est-à-dire en terme plus clair dans la théorie des langages, ceci s'interprète par la notion de contenance (containment en anglais) qui s'exprime par $L(A) \subseteq L(S)$. Malheusement, l'implantation d'un tel calcul s'avère être très coûteuse puisque la complexité d'un tel problème est NP. Jouant sur la notion ensembliste de l'intersection de deux ensembles, le problème de contenance est ramené vers un problème d'intersection. Ce qui fait qu'au lieu de considérer $L(A) \subseteq L(S)$ on considérera $L(A) \cap \overline{L(S)}$ qui est plus simple à concrétiser que celle de l'inclusion (problème NP) d'une part et d'autre part, calculer le complément d'un automate de Büchi est difficile. Toutefois il serait meilleur de calculer $S_{\neg\phi}$ plutôt que de calculer d'abord l'automate S_ϕ puis de calculer son complément.

Le model checking suivant permet de résoudre le problème de la vérification dans le sens d'idée développé dans ce cadre.

Algorithme.

- Modélisation du système (M_1) sous forme d'un automate A,
- Trouver un automate $\overline{S}$ qui représente la négation de la spécification désirée,
- Vérifier si $L(A) \cap \overline{L(S)} = \phi$ en recherchant d'abord l'automate B qui est le produit synchrone et tel que $L(B) = L(A) \cap \overline{L(S)} = \phi$.

5.4.1 Transformation d'un graphe de Kripke en un graphe de Büchi

L'algorithme est très simple, il suffit pour cela de reproduire le graphe de Kripke tel qu'il est, en faisant attention de garder les états et les transitions ainsi que leurs liaisons, de faire glisser les labels (propositions) qui étiquettent les états (sommet du graphe) sur les transitions incidentes vers l'intérieur, et les boucles seront étiquetées par la formule de l'état lui-même. On obtient ainsi le graphe de Büchi (automate de Büchi).

Exemple 5.15

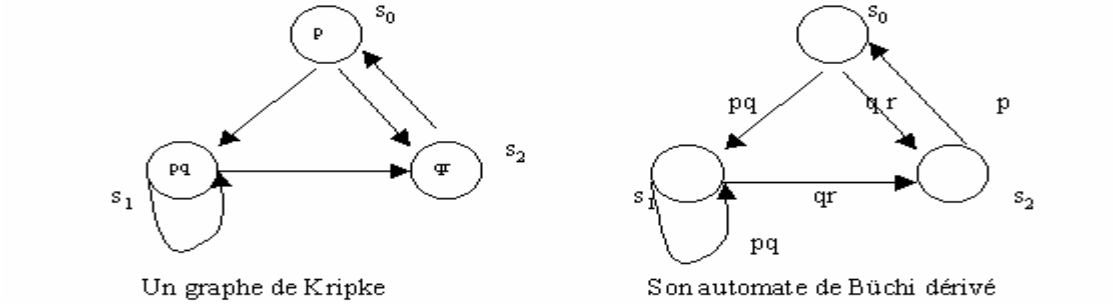


Figure 5.7 : Transformation d'une structure de Kripke en un automate de Büchi

5.4.1.1 Calcul de l'intersection de deux Automates

Pour vérifier qu'une spécification est valide, on se ramène à déterminer tous les ω-mots qui satisfont l'automate obtenu à partir de la composition de l'automate qui représente le système (l'implémentation) et celui de la spécification (formule de la logique temporelle)[GPV+95, SB00]. La composition n'est autre que l'intersection de deux automates.

Soit $\mathcal{A} = (S_1, T_1, S_{01}, F_1, \Sigma)$ et $\mathcal{B} = B_{\neg\phi} = (S_2, T_2, S_{02}, F_2, \Sigma)$, alors $L(A) \cap L(B_{\neg\phi})$ est l'ensemble des mots reconnaissables par l'automate de Büchi $\mathcal{P} = \mathcal{A} \otimes \mathcal{B} = (S, S_0, T, F, \Sigma)$ où $S = S_1 \times S_2$ et $S_0 = S_{01} \times S_{02}$ et T étant la synchronisation de T_1 et T_2 sur les actions identiques et $F = F_1 \times F_2$.

5.4.2 Implémentation de la Procédure du Model-checking

Pour implémenter une telle procédure, nous aurons besoin de deux choses: un moyen pour former la composition parallèle de deux automates, et un autre moyen pour vérifier si un automate contient des exécutions d'acceptance [SB00, EH00].

Le premier moyen en principe est direct, il suffit de former la composition parallèle $\mathcal{P} = \mathcal{A} \times \mathcal{B}$ de deux automates. Pour cela on calcule les éléments suivants :

$\mathcal{P} = \mathcal{A} \otimes \mathcal{B} = (S_1 \times S_2, S_{01} \times S_{02}, T, F_1 \times F_2, \Sigma)$ [le produit synchrone]

T est définit par : $(s_1, s_2) \xrightarrow{a}_p (s'_1, s'_2) \equiv s_1 \xrightarrow{a} s'_1 \;\&\; s_2 \xrightarrow{a} s'_2$

Le deuxième moyen étant de vérifier que $\boldsymbol{L}(\mathcal{P}) \neq \emptyset$ si et seulement s'il existe des exécutions $s_0^1 \xrightarrow{a1} s_1^1 \xrightarrow{a2} s_2^1, \cdots$ et $s_0^2 \xrightarrow{a1} s_1^2 \xrightarrow{a2} s_2^2, \cdots$ telle que pour chaque $i \in \{1,2\}$, $s_0^i \in S_0^i$ et infiniment plusieurs des $s_0^i, s_1^i, s_2^i, \cdots$ sont dans $\boldsymbol{F_i}$.

Ce point peut facilement être prouvé. Supposons, que $\boldsymbol{L}(\mathcal{P}) \neq \emptyset$. Alors il existe une exécution acceptante dans $\mathcal{P}$ qui a pour forme : $(s_0^1, s_0^2) \xrightarrow{a1}_p (s_1^1, s_1^2) \xrightarrow{a2}_p (s_2^1, s_2^2) \cdots$, avec $(s_0^1, s_0^2) \in S_0^1 x S_0^2$ et tel qu'il existe infiniment plusieurs des (s_j^1, s_j^2) *dans* $F_1 x F_2$. L'existence des deux exécutions avec la propriété désirée est maintenant immédiate à cause de la définition de la transition $\xrightarrow{a}_p$. De la même manière nous pouvons vérifier dans l'autre sens que $(s_1, s_2) \xrightarrow{a}_p (s'_1, s'_2) \Leftarrow s_1 \xrightarrow{a} s'_1 \;\&\; s_2 \xrightarrow{a} s'_2$. Comme $(s_j^1, s_j^2) \xrightarrow{aj+1}_p (s_{j+1}^1, s_{j+1}^2)$ pour tout $j \geq 0$, et $(s_0^1, s_0^2) \in S_0^1 x S_0^2$. Il reste maintenant de montrer que (s_j^1, s_j^2) *dans* $F_1 x F_2$. *Comme* $F_1 = S_1$, alors $(s_j^1, s_j^2) \in F_1 x F_2$ si et seulement si $s_j^2 \in F_2$, et ceci arrive le plus souvent par supposition.

La deuxième partie de notre model-checking permet de déterminer si un automate $\mathcal{A}$ possède des exécutions d'acceptations, ou, d'une façon équivalente, si $\boldsymbol{L}(\mathcal{A}) \neq \varnothing$. Pour cela, il suffit d'imaginer que l'automate sera représenté par un graphe orienté et ou les sommets sont les états et l'existence d'une transition suppose l'existence d'un sommet initial et un sommet terminal reliés par une transition (un arc) qui est étiqueté par une action $a \in \sum$. Alors $\boldsymbol{L}(\mathcal{A}) \neq \varnothing$ est vraie si et seulement il existe un chemin s_0, s_1, … , s_k dans $\mathcal{A}$ tel que s_k soit un état d'acceptation et qu'il existe un cycle de s_k vers s_k. S'il existe un tel chemin et un tel cycle il est claire que $\boldsymbol{L}(\mathcal{A}) \neq \varnothing$, et que nous pouvons obtenir un cycle d'acceptation en suivant s_0 à s_k et nous répétons ceci en suivant le cycle. Réciproquement, supposons $\boldsymbol{L}(\mathcal{A}) \neq \varnothing$, et il existe une exécution $s_0 \xrightarrow{X_1} s_1 \xrightarrow{X_2} s_2, \cdots$ Par la définition de l'exécution d'acceptation et par le fait que **F** est fini, il existe un cycle de s_k vers lui-même qui contient un état d'acceptation.

En utilisant ce fait, il serait plus judicieux par exemple de rechercher les composantes fortement connexes. Pour cela, l'un des meilleurs moyens qui permet de les déterminer étant celui qui utilise une procédure basée sur une procédure du type le *depth-first search* (une recherche en profondeur d'abord (***dfs***)) [HPY98] pour vérifier que s_k satisfait les conditions citées plus haut. C'est-à-dire, en premier lieu, c'est de vérifier que s_k est un état d'acceptation, en principe ceci est trivial. Par contre pour vérifier que s_k appartient à un cycle, nous seront obligé d'utiliser une deuxième procédure **dfs** qui s'effectue dans le pire des cas en une complexité temps de O(nm), avec n le nombre de sommets (états) et m le nombre de transitions. Mais en entrelaçant deux ***dfs**'s* nous pouvons réduire cette complexité à O(m). Les deux procédures *dfs* récursives qui utilisent une pile sont présentées comme suit :

```
function nmt(A)
     recursive dfs1(u)
      push(u, S) ;
      mark u visited;
      for each edge u →v do
         if v not visited then defs1(v);
       if u is an accepting state then dfs2(u);
       pop(S)

      recursive dfs2(u)
        flag u ;
        for each edge u →v do
          if v is on S then return true from nmt;
             if v not flagged then dfs2(v);
         St := Ø;
         dfs1(s0);
return false
```

Figure 5.8 : L'Algorithme de Recherche des Etats Valides

5.4.3 Transformation d'une formule LTL en Automate de Büchi

La procédure de traduction permet de transformer une formule LTL ou PLTL en un automate de Büchi [GO01]. Une étape préliminaire est alors engagée pour n'autoriser qu'un certain type d'opérateurs et procéder ainsi à une forme de simplification de la spécification. Pour cela il est nécessaire d'utiliser les équivalences (LTL) suivantes et de les convertir dans leur forme normale :

- $F\, g = 1\, U\, g$
- $G\, g = \neg(1\, U \neg g)$
- $\neg(f \vee g) = \neg f \wedge \neg g$
- $\neg(f \wedge g) = \neg f \vee \neg g$
- $\neg\neg f = f$

- $\neg X f = X \neg f$
- $\neg(f\ U\ g) = \neg f\ R\ \neg g$
- $\neg(f\ R\ g) = \neg f\ U\ \neg g$
- $\neg[]f\ =\ <>\neg f.$
- $\neg<>f = []\ \neg f.$
- $\rightarrow \psi = (\neg\ \varphi) \vee \psi$
- $<>\varphi\ = (T\ U\ \varphi).$
- $[]\varphi = (F \vee \varphi).$
- Transformation de ([]<>P) → ([]<>Q) par élimination de l'implication ¬([]<>P) ∨ ([]<>Q)
- Elimination de [] et <>
- La négation est poussée vers l'intérieur: ¬(F ∨ (T U P)) ∨ (F ∨ (T U Q)) devient (T U (F U ¬ P)) ∨ (F ∨ (T U Q)).

Il est à remarquer que les nouvelles formules construites par réécriture ne contiennent plus que les opérateurs U, R, X, ∧, ∨, ¬ et les formules atomiques.

Durant toutes les étapes de construction du graphe (automate), une partie des nœuds sera marquée ***done***, qui est un terme anglais pour dire que tout sommet ainsi marqué ne sera ni modifié ni supprimé, à la seule possibilité que les arcs partant de ce sommet peuvent être modifiés.

Pour commencer cette construction, nous créons tout d'abord deux nœuds, ***init*** et $\boldsymbol{v_0}$ reliés par un arc. Le champs du nœuds *init* sera vide et v_0 aura Old(v_0) = Next(v_0) = Ø, New(v_0) = $\overline{f}$. Le nœud *init* est alors marqué par *done*.

Le reste de la construction du graphe s'exécute en faisant appel à la procédure récursive *expand(v_0)* suivante :

```
Expand(u) :
    If new(u) = Ø then {done with this node}
       If there is a node u' marked with the same fields as u
       then {a similar node has already been fully proceeded}
          redirect any edges pointing to u to now point at u' ;
          delete u from the graph ;
       else {create a "nest" state; Next(u) must be true}
          mark u done ;
          Old(v), New(v), Next(v) :=, Next(u), Ø ;
          Expand(v)
    else {process a "new" formula in the node}
       Remove a formula g from New(u) and it to Old(u) ;
       process_formula(g, u)
```

Figure 5.9 : L'Algorithme EXPAND

La procédure *process_formula()* modifie les nœuds ***u*** et fait des appels récursifs à la fonction *expand()* qui prend en charge d'autres formules. En outre, elle procède par une analyse par cas sur la structure de g. Les différentes situations qu'elle peut traiter sont :

- g = p. Si $\neg p \in Old(u)$, alors p et $\neg p$ sont supposés être vraies, ce qui est impossible comme nous le savons, alors il ne sera pas possible d'atteindre ce nœud ce qui nous oblige de supprimer u ainsi que les arcs incidents intérieurement à u. Dans l'autre cas on appelle alors la procédure *expand*(u).

- g = $\neg p$. Ce cas est similaire au cas passé. Si $p \in Old(u)$, alors on supprime u et tous les arcs incidents intérieurement. Sinon on fait un appel à *expand*(u).

- $g = g_1 \wedge g_2$. g_1 et g_2 doivent être vrais, on les rajoute alors à *New*(u) et on appelle *expand*(u).

- $g = g_1 \vee g_2$. Si l'exécution parallèle considère u, alors g_1 est vrais en u, ou bien g_2 est vrais. On éclate alors le nœud u pour représenter les deux cas. On crée alors deux nœuds u_1 et u_2 avec le même champ que u, et on rajoute u_1 à *New*(u_1) et g_2 à *New*(u_2). On supprime alors le nœud u et pour chaque arc incident à l'intérieur, on le duplique pour avoir une copie pour u_1 et une autre pour u_2. Finalement on appelle *expand*(u_1) et aussi *expand*(u_2).

- g = X g'. Ceci montre que X g' doit être vraie en tout état suivant, alors on rajoute g' à *Next*(u).

- $g = g_1 \, U \, g_2$. Nous savons que $g_1 \, U \, g_2 = g_2 \vee (g_1 \wedge X(g_1 \, U \, g_2))$. Dans ce cas il nous sera possible de dupliquer en mettant g_2 dans *New*(u_1) et $(g_1 \wedge X \, (g_1 \, U \, g_2))$ in New(u_2) et en faisant des appels récursifs.

- $g = g_1 \, R \, g_2$. Nous avons $g_1 \, R \, g_2 = g_2 \wedge (g_1 \vee X(g_1 \, U \, g_2)) = (g_2 \wedge g_1) \vee (g_2 \wedge X(g_1 \, R \, g_2))$. Puis pour les autres parties nous procédons de la même par la duplication de chaque sous formule.

5.4.3.1 Comment s'y prendre pour appliquer cet algorithme

Pour se faire, on utilisera une table à trois cases, qu'on nommera respectivement par, ***New***, ***Old*** et ***Next***.Puis on procédera comme suit :

- Pour les formules composées, on essayera à chaque fois de séparer la formule en deux parties. La première partie sera étiquetée dans l'état courant et la deuxième dans l'état suivant *Next* .

- Prendre une formule de l'état *New* et la rajouter à l'état *Old*.

- Selon la formule, ou bien on fractionne le nœud courant en deux nouveaux nœuds ou bien une nouvelle copie du nœud sera mise en place.

Exemple 5.16

$\varphi \, U \, \psi$ est fractionnée comme suit :

1. Add φ to *New*, add $\varphi \, U \, \psi$ to *Next*.
2. Add ψ to *New*.

Comme $\varphi \, U \, \psi = \psi \vee (\varphi \wedge X(\varphi \, U \, \psi))$.

$\varphi \vee \psi$, split (fractionner) :

1. Add φ, ψ to *New*.
2. Add ψ to *New*, $\varphi \vee \psi$ to *Next*.

Comme $\varphi \vee \psi = \psi \wedge (\varphi \vee X(\varphi \vee \psi))$.

$\varphi \wedge \psi$, evolve (se transforme en):

1. Add φ, ψ to *New*.
2. Xφ, evolve:
3. Add φ to *Next*.

5.5 Modèles Arborescents

5.5.1 La Logique arborescente CTL

Il est bien connu que c'est Amir Pnueli [Pnu77], qui a introduit le premier la logique temporelle à temps linéaire (LTL) aussi appelée PLTL (Propositional LTL) pour la spécification est la vérification des systèmes réactifs. Comme nous l'avons déjà dit, la logique LTL est dite linéaire compte tenu de la notion du temps qui est qualitative et considérée comme linéaire, c'est-à-dire qu'a chaque moment du temps il y a seulement un seul état successeur possible, et donc chaque moment admet un seul futur possible. Techniquement parlé, cette notion de temps linéaire est rendue possible grâce à l'interprétation des formules, définie en termes de séquences d'états. La logique linéaire LTL quoi que très utile pour la vérification des systèmes réactifs, ne permet pas de fournir les outils nécessaires pour considérer des situations très fréquentes et plus compliquées. A titre d'exemple, le fait de dire « pour chaque exécution il est toujours possible de retourner à l'état initial », toute tentative d'interpréter cette propriété dans la logique LTL échoue. Pour éviter de tomber dans ce type de problème qui est d'ailleurs multiple est très fréquent, Clarke et Emerson [CE81] aux débuts des années quatre-vingts développèrent une autre logique temporelle qu'ils ont appelé CTL (Computationnal Tree Logic). La sémantique de cette logique n'est pas du tout basée sur la notion du temps linéaire sur une séquence infinie d'états mais sur la notion du temps arborescent (de branchement), définie sur un arbre infini d'états. La notion de temps arborescent fait référence qu'a chaque moment il peut y avoir plusieurs futures possibles qui différent les uns des autres. Donc, chaque moment du temps est reparti en plusieurs futures possibles, et chaque chemin (path) est supposé représenter une seule séquence possible d'exécution et l'arbre qui prend racine en s_0, représente toutes les exécutions infinies possibles obtenues par dépliage de la structure de Kripke (Figure 5.13).

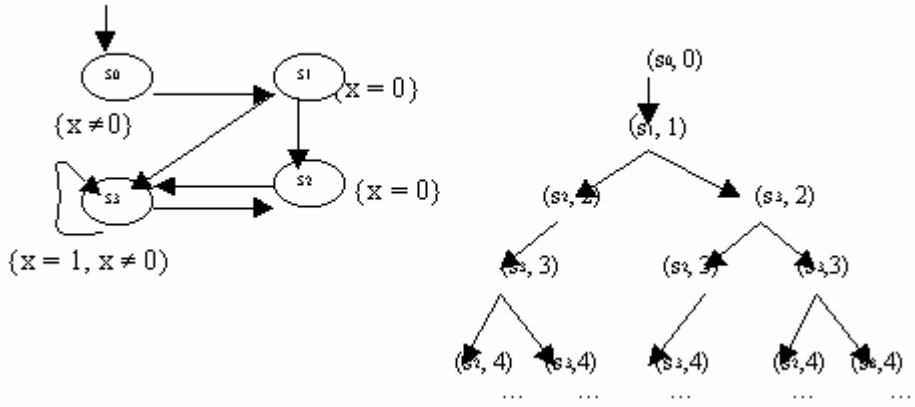


Figure 5.10 : Une structure de Kripke et son arborescence infinie

5.5.1.1 Syntaxe de CTL

Nous reconduisons dans la logique CTL, presque les mêmes éléments de base cités dans la logique linéaire (LTL), plus certains nouveaux opérateurs.

Définition 5.17 (Syntaxe de CTL) *Soit p une proposition atomique. Les formules dans la logique CTL sont ou bien des formules définies sur les états (state-formulae) ou bien sur les chemins (path-formulae). Les formules-états satisfont les règles suivantes :*

- *p est une formule-état.*
- *Si Φ est une formule-état, alors $\neg\Phi$ est une formule-état de CTL.*
- *Si Φ et Ψ sont des formules-états, alors $\Phi \vee \Psi$ est une formule-état.*

- *Si ϕ est une formule-chemin, alors $\boldsymbol{E}\phi$ et $\boldsymbol{A}\phi$ sont des formules-états.*

Les formules-états satisfont les règles suivantes :

- *Si Φ est une formule-état, alors $\boldsymbol{X}\Phi$ est une formule-chemin.*
- *Si Φ et Ψ sont des formules-états, alors $\Phi \boldsymbol{U} \Psi$ est une formule-chemin*

Donc CTL fait une distinction entre les formules-état et les formules-chemin. Intuitivement, les formules-état expriment des propriétés sur les états et les formules-chemin expriment à leur tour des propriétés sur les chemins. Les opérateurs temporels X et U ont la même signification que pour la logique temporelle linéaire. Les formules Eϕ qu'on prononce « for some paths » sont vraies dans un état *s*, s'il existe des chemins qui satisfont ϕ partant de s. Les formules du type Aϕ qu'on prononce « for all paths) sont vraies dans un état *s* si tous les chemins partant de s satisfont la formule ϕ. Nous pouvons naïvement trouver une relation de dualité entre Aϕ et Eϕ, similaire à la relation qui lie les quantificateurs existentiels et universels de la logique de premier ordre. De ce fait, nous avons $A\phi \equiv \neg E \neg\phi$.

5.5.1.2 Sémantique de CTL

Exactement comme dans la logique linéaire, l'interprétation des formules CTL est définie en terme de structure de Kripke. Il n'est pas utile de reproduire la définition des structures de Kripke et celle des chemins.

Définition 5.18 *Pour une structure de Kripke donnée M = (S, I, R, Label) et un état $s \in S$, il y a un arbre infini de racine s tel que (s', s'') est un arc dans l'arbre si et seulement $(s', s'') \in R$.*

Cet arbre est obtenu par dépliage de la structure de Kripke à partir du sommet s. Le nombre de degré extérieur d'un nœud est calculé à partir du nombre de transitions incidentes vers l'extérieur de ce nœud dans la structure de Kripke.

Un exemple est donné dans la Figure 5.10. Nous avons à gauche la structure de Kripke d'un problème quelconque et à droite l'arborescence infinie de racine s_0 (sommet initial). Par convenance, à chaque nœud de cette arborescence, on lui adjoint un numéro qui indiquera le niveau de dépliage. Les chemins sont obtenus en traversant l'arborescence de haut en bas à partir du sommet initial (racine). Les séquences suivantes sont des chemins pris dans l'arborescence du graphe,

$$s_0s_1s_2s^{\omega}_3,\ s_0s_1(s_2s_3)^{\omega} \text{ et } s_0s_1(s_3s_2)^*s^{\omega}_3.$$

La sémantique des formules CTL est définie à l'aide de deux relations de satisfactions (toutes les deux, seront dénotées par |=). Donc une relation sera utilisée pour les formules-état et l'autre pour les formules-chemin Dans la suite, on utilisera l'écriture M, s |= Φ au lieu de ((M, s), Φ) ∈ |=. La signification d'une telle écriture est : M, s |= Φ si et seulement si la formule (état) Φ est vraie dans l'état s de la structure M. Pour l'autre relation |= (sur les chemins), celle-ci est une relation entre la structure de Kripke, l'un de ces chemins et la formule-chemin Φ. On écrira dans ce cas M, σ |= φ si et seulement si le chemin σ dans le model M satisfait la formule φ. Pour donner plus de clarté dans ce qui suit, nous omettrons à chaque fois l'écriture de M.

Définition 5.19 (Sémantique de CTL) *Soit $p \in AP$ une proposition atomique, M = (S, I, R, L) une structure de Kripke, un état quelconque $s \in S$, Φ et Ψ des formules–états de CTL, et φ une formule-chemin de CTL. La relation de satisfaction |= est définie pour les formules-états par ce qui suit :*

$s \models p$ *si et seulement si (noté **ssi**) $p \in Label(s)$*

$s \models \neg \Phi$ ***ssi*** *not $(s \models \Phi)$*

$s \models \Phi \vee \Psi$ ***ssi*** *$(\sigma \models \Phi)$ or $(\sigma \models \Psi)$*

$s \models E\ \varphi$ ***ssi*** *$\sigma \models \varphi$ pour un certain nombre de chemins $\sigma \in Paths(s)$*

$s \models A\ \varphi$ ***ssi*** $\sigma \models \varphi$ *pour tous les chemins* $\sigma \in Paths(s)$.

Pour la relation sur les chemins, la relation de satisfaction |= pour les formules-chemin est définie comme suit :

$\sigma \models X\ \Phi$ **ssi** $\sigma[1] \models \Phi$

$\sigma \models \Phi\ U\ \Psi$ **ssi** $(\exists j \geq 0.\ (\sigma[j] \backslash\ \psi\ \wedge\ (\forall 0 \leq k < j.\ \sigma[k] \backslash \Phi))$

L'interprétation en CTL des propositions atomiques, de la négation et de la conjonction sont définies comme d'habitude sur les états et non sur les chemins, contrairement à leurs interprétations dans la logique LTL qui est sur les chemins.

La formule-état E φ est valide dans l'état s si et seulement s'il existe des chemins (pas forcement tous les chemins) prenant naissance en s qui satisfont φ. De la même manière on définit A φ, mais sur tous les chemins.

5.5.2 Opérateurs Temporels Auxiliaires

Les opérateurs booléens true, $\wedge$, $\Rightarrow$ et $\Leftrightarrow$ sont définis comme d'habitude.

Soit F Φ = true U Φ, on définit alors ce qui suit :

$EF\ \Phi \equiv E(true\ U\ \Phi)$

$AF\ \Phi \equiv A(true\ U\ \Phi)$

EF Φ est prononcé « Φ holds potentially » et AF Φ est prononcé « Φ is inevitable »

L'interprétation des opérateurs temporels AX Φ, EF Φ, EG Φ, AF Φ et AG Φ peuvent être dérivés en utilisant la sémantique de CTL.

Exemple de dérivation : EG Φ,

$S \models EG\ \Phi \Leftrightarrow$

{définition de EG}

$s \models AF\ \Phi \Leftrightarrow$

{définition de F}

$s \models \neg A(true\ U \neg\Phi)) \Leftrightarrow$

{sémantique de ¬}

$\neg(s \models A(true\ U \neg\Phi)) \Leftrightarrow$

{sémantique de A U }

$\neg(\forall \sigma \in Paths(s).\ (\exists j \geq 0.\ (\sigma[j])\ \neg\Phi \wedge\ (\forall 0 \leq k < j.\ \sigma[k] \models true))) \Leftrightarrow$

{s |= true pour tout état s ; calcul des prédicats}

$\exists \sigma \in Paths(s).(\forall j \geq 0.\ \sigma[j] \models \Phi)$

Donc EG Φ est valide dans l'état *s* si et seulement si un ensemble de chemins (pas forcement tous) partant de l'état *s* tel que pour chaque état de ces chemins Φ est valide.

Exemple 5.20 Soit la structure de Kripke M (Figure 5.11). Essayons de donner un aperçu informel sur le comportement de chacune des formules suivantes sur chacun des états et chemins du graphe de Kripke.

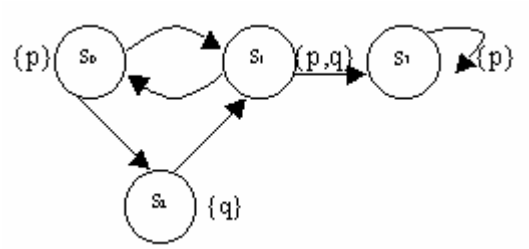


Figure 5.11 : Graphe de Kripke de Démonstration

- La formule EX p est valide pour tous les états du graphe, puisque chacun des états possède au moins un successeur qui satisfait p.

- La formule AX p n'est pas valide au niveau de l'état s_0, puisqu'un chemin possible prenant naissance en s_0 et passant par s_3 qui est le successeur immédiat de s_0 alors que l'invariant de s_3 est {q} qui est différent de {p}. Cependant tous les autres états vérifient cette propriété

- La formule EG p est valide dans les sommets s_0, s_1 et s_3, puisque nous pouvons déterminer des chemins à partir de ces trois sommets pour lequel p est globalement valide. Il reste le sommet s_2 auquel Label(s_2) est différent de p. Ce qui revient à dire que EG p n'est pas valide en s_2.

- La formule AG p est valide uniquement pour s3, puisque le seul chemin valide est s^{ω}_3 visitera ce sommet à l'infinie. Pour les autres états il est possible de trouver un chemin qui contiendra le sommet s_2 qui ne vérifie pas la condition.

- La formule EF(G p) est valide pour tous les états puisque à partir de n'importe quel sommet, un autre sommet (s_0, s_1, s_3) peut éventuellement être atteint à partir duquel des exécutions peuvent prendre naissance ou tout le long de ces exécutions p est globalement valide.

- A(p U q) n'est pas valide en s_3, puisque sa seule exécution (s^{ω}_3) n'atteindra jamais un état étiqueté par q. Pour l'état s_0 le problème ne se pose pas puisque s_0 est étiqueté par p, donc la formule est valide jusqu'à ce qu'un état étiqueté par q devienne actif, ceci d'ailleurs se réalise juste après avec l'état successeur qui est s_1. Les états s_1 et s_2 valident la formule immédiatement puisqu'ils sont étiquetés par q.

- La formule E(P U($\neg$ p $\wedge$ A($\neg$p U q))) n'est pas valide en s_3, puisque qu'à partir de s_3 on ne pourra jamais atteindre un sommet étiqueté par q. Pour les sommets s_0 et s_1, la formule est valide puisque l'état s_2 peut être atteint à partir d'un p-path prenant naissance en ces deux sommets, $\neg$ p est valide en s_2 et à partir de s_2 tous les chemins satisfont $\neg$ p U q, puisque s_2 est un q-state.

5.5.3 Axiomatisation de la logique CTL

Nous allons procéder de la même manière que pour la logique LTL, pour affirmer que CTL peut aussi être axiomatisable. Nous avons vu, que l'axiome le plus important dans la logique LTL est l'axiome de l'expansion :

$$\Phi \, U \, \psi \Leftrightarrow \psi \vee (\Phi \wedge X(\Phi \, U \, \psi))$$

Dans CTL, des axiomes de ce type existent aussi. Comme dans la logique LTL, nous pouvons préfixer l'opérateur U par un quantificateur existentiel et un autre universel, c'est-à-dire qu'on obtient E(Φ U ψ) et A(Φ U ψ). Le tableau ci-dessous nous fournit un ensemble d'axiomes *correct* « sound » et peuvent être prouvés en utilisant la logique CTL. Pour en être plus précis, nous invitons le lecteur à consulter les articles [Eme96, CES86] qui fournissent la base de démonstration d'un système axiomatisable de règles de CTL qui soit correct et *complet*. Dans ce qui suit, nous allons fournir un moyen simple pour vérifier quelques axiomes.

$EF\ \Phi \equiv \Phi \vee EX(EF\ \Phi)$
$AF\ \Phi \equiv \Phi \vee AX(AF\ \Phi)$
$EG\ \Phi \equiv \Phi \wedge EX(EG\ \Phi)$
$AG\ \Phi \equiv \Phi \wedge AX(AG\ \Phi)$
$E(\Phi\ U\ \psi) \equiv \psi \vee (\Phi \wedge EX\ E(\Phi\ U\ \psi))$
$A(\Phi\ U\ \psi) \equiv \psi \vee (\Phi \wedge AX\ A(\Phi\ U\ \psi))$

Table 5.12 : Quelques axiomes d'expansion pour CTL

5.5.3.1 Quelques exemples pour vérifier certains axiomes

1- $AF\ \Phi \equiv \Phi \vee AX(AF\ \Phi)$?

$AF\ \Phi \Leftrightarrow$ {définition de AF}

$A(true\ U\ \Phi) \Leftrightarrow$ {axiome pour $A(\Phi\ U\ \psi)$}

$\Phi \vee (true \wedge AX\ A(true\ U\ \Phi))) \Leftrightarrow$ {calcul des prédicats, définition de AF}

$\Phi \vee AX(AF\ \Phi)$.

2- $EG\ \Phi \equiv \Phi \wedge EX(EG\ \Phi)$

$EG\ \Phi \Leftrightarrow$ {définition de EG}

$\neg\ AF\ \neg\Phi \Leftrightarrow$ {résultat après la dérivation}

$\neg\ (\neg\Phi \vee AX(AF\ \neg\Phi)) \Leftrightarrow$ {calcul des prédicats}

$\Phi \wedge AX((AF\ \neg\Phi)) \Leftrightarrow$ {définition de AX}

$\Phi \wedge EX(\neg(AF\ \neg\Phi)) \Leftrightarrow$ {définition de EG}

$\Phi \wedge EX((EG\ \Phi)$.

5.6 Model-checking de CTL

Comme il a déjà été introduit, A et E sont les quantificateurs duaux sur les chemins, et G et F sont les quantificateurs duaux sur les états. A partir de ces quantificateurs, il est simple de retrouver certaines lois de De Morgan :

$\neg AF\phi \equiv EG\neg\phi$

$\neg EF\phi \equiv AG\neg\phi$

Aussi, il est vrai que pour un chemin particulier, chaque état possède un successeur unique, alors X est auto-dual :

$\neg AX\phi \equiv EX\neg\phi$

Nous savons aussi que

$AF\phi \equiv A[T\ U\ \phi]$

$EF\phi \equiv E[T\ U\ \phi]$

En plus de cela, nous considérons que toutes les équivalences valides de la logique propositionnelle sont aussi valides pour CTL. En utilisant les équivalences ci-dessus et l'équivalence suivante :

$$\phi\ U\ \psi \equiv \neg\ (\neg\psi\ U\ (\neg\phi \wedge \neg\psi)) \wedge G\psi$$

On peut ré-écrire toutes formules CTL en une formule équivalente qui ne contienne que les connecteurs temporaux EX, EG et EU :

$EF\phi \equiv \neg E[T\ U\ (\neg\phi)]$

$AX\phi \equiv \neg EX(\neg\phi)$

$AG\phi \equiv \neg EF\neg\phi \equiv \neg E[T\ U\ \neg\phi]$

$AF\phi \equiv A[T\ U\ \phi] \equiv \neg EG(\neg\phi)$

$A[\phi\ U\ \psi] \equiv \neg E[(\neg(\neg\psi\ U\ (\neg\phi \wedge \neg\psi))]$

Théorème 5.21 *Toute formule CTL est transformable en une formule équivalente qui n'utilise que les connecteurs 0, ∧, ¬, EG, EU et EX.*

5.6.1 Calcul des points fixes de fonctions monotones : une approche pour CTL

Cette partie de notre exposé s'oriente à présenter une autre approche pour mieux expliquer les logiques modales et temporelles essentiellement, le mu-calcul et la logique CTL [Arn88, EC80, AN92]. L'essentiel des éléments de base sera puisé de la théorie des travaux de Kozen [Koz83, KP84].

Définition 5.22 (Ordre Partiel) Une relation binaire ⇓ ™ A x A est une relation d'ordre partiel si et seulement si elle est réflexive, anti-symétrique et transitive. Soit, pour tout a, a', a'' 5 A :

- a ⇓ a (**réflexivité**)
- (a ⇓ a' ∧ a' ⇓ a) ⇒ a = a' (**anti-sysmétrie**)
- (a ⇓ a' ∧ a' ⇓ a'') ⇒ a ⇓ a'' (**transitvité**).

La paire (A, ⇓) est un ensemble d'ordre partiel (***poset***). Si non(a ⇓ a') et non(a' ⇓ a) alors a et a' sont dits incomparables.

Définition 5.23 (Least upper bound) *Soit (A, ⇓) un ensemble d'ordre partiel et A' ⇓ A (une partie de A).*

- *a ∈ A est une borne supérieure de A' si et seulement si ∀a' ∈ A' : a ⇓ a'.*
- *a ∈ A est la plus petite borne supérieure (least upper bound) de A', noté ®A', si et seulement si (1) a est une borne supérieure de A' et (2) ∀a'' ∈ A, a'' est une borne supérieure de A' ⇒ a ⇓ a''.*

Le concept de borne inférieure et de plus grande borne inférieur (greatest lower bound), noté ⟨A', peut être définit de la même manière que pour la borne supérieur et le plus ptite borne supérieur.

Soit (A, ⇓) un ordre partiel.

Définition 5.24 (treillis complèt)[Tar55] *(A, ⇓) est un* treillis complèt *si pour tout A' ⊆ A, ®A' et ⟨A' existent.*

Un treillis complet possède un et un seul plus petit élément et un et un seul plus grand élément.

Définition 5.25 *Soit S, un ensemble et g : 2^S → 2^S. On dira que f est monotone si pour tout X, X' ∈ S on a, X ⇓ X' ⇒ f(X) ⇓ f(X').*

Définition 5.26 *Soit S, un ensemble et g : 2^S → 2^S. On dira que X est un point fixe de f si f(X) = X. Un point fixe est un plus petit (resp. plus grand) point fixe de f qu'on note μ (resp. ν), si pour tout point fixe de f, Y ⇓ X (resp. X ⇓ Y).*

La monotonicité de S est suffisante pour assurer l'existence d'un plus petit et d'un plus grand point fixe *f* :

Théorème 5.27 *Toute fonction monotone définie sur une lattice complète possède une lattice complète de points fixes.*

Preuve : c'est une conséquence du théorème de Kleene qui est présenté juste après.

Théorème 5.28 (Knaster-Tarski). *Si* $f : 2^S \rightarrow 2^S$ *est monotone, alors :*

- f *possède un plus petit point fixe donné par* $\bigcap\{X \subseteq S : f(X) \subseteq X\}$;
- f *possède un plus grand point fixe donné par* $\bigcup\{X \subseteq S : X \subseteq f(X)\}$.

Définition 5.29 *Une fonction* $f : 2^S \rightarrow 2^S$ *est* $\cup$*-continue (resp.* $\cap$*-continue) si pour toute suite* $[X_i]$ *croissante (resp. décroissante) pour l'inclusion, on a :*

$$f(\bigcup_i X_i) = \bigcup_i f(X_i)$$

respectivement

$$f(\bigcap_i X_i) = \bigcap_i f(X_i)$$

Remarque 5.30 Toute fonction continue est monotone car

$$\begin{aligned} X \subseteq X' &\Rightarrow X' = X \cup X' \\ &\Rightarrow f(X') = f(X) \cup f(X') \\ &\Rightarrow f(X) \subseteq f(X'). \end{aligned}$$

De la même façon pour $X = X \cap X' \Rightarrow f(X) \subseteq f(X')$.

Remarque 5.31 Si f est monotone on a toujours

$$f(\bigcup_i X_i) \supseteq \bigcup_i f(X_i)$$

et $$f(\bigcap_i X_i) \supseteq \bigcap_i f(X_i)$$

Preuve : Nous savons que pour tout i on a $X_i \subseteq \bigcup_i X_i$ alors par monotonocité pour tout i $f(X_i) \subseteq f(\bigcup_i X_i)$ et donc, $\bigcup_i f(X_i) \subseteq f(\bigcup_i X_i)$. De la même manière on définit $f(\bigcap_i X_i) \subseteq f(\bigcap_i X_i)$.

Remarque 5.32 Si f et non seulement monotone mais continue alors le plus petit (resp. plus grand) point fixe de f peut se représenter comme limite de ses itérations $i^{ème}$, f^i définies comme suit pour tout $X \subseteq S$:

$$f^0(X) = X \text{ et } f^{i+1}(X) = f(f^i(X))$$

Théorème 5.33 (Kleene)[Kle52]. *Si f est continue alors*

$$\mu f = \bigcup_i f^i(\emptyset)$$

$$\nu f = \bigcap_i f^i(S)$$

Preuve : Puisque f est continue alors la suite $\{f^i(\emptyset)\}_{i \geq 0}$ est croissante et on aura aussi,

$$f(\bigcup_i f^i(\emptyset)) = \bigcup_i f^{i+1}(\emptyset) = \bigcup_i f^i(\emptyset)$$

qui est un point fixe de f. C'est le plus petit point fixe de f car si X est un point fixe il vient alors :

$$\emptyset = f^0(\emptyset) \subseteq X$$

Si $f^0(\emptyset) \subseteq X$ alors $f^{i+1}(\emptyset) \subseteq f(X) = X$.

Théorème 5.34 *Soit* $f : 2^S \rightarrow 2^S$ *monotone où S est fini de cardinalité n alors* $\mu f = f^n(\emptyset)$ *et* $\nu f = f^n(S)$.

Preuve : Soit n_i la cardinalité de $f^i(\emptyset)$, alors la suite $\{n_i\}_{i\geq 0}$ est majorée par n et est croissante. Soit k le plus petit entier tel que $n_k = n_{k+1}$ alors $\forall_{j\geq k} f^k(\emptyset) = f^j(\emptyset)$ car $f^i(\emptyset) = f^{i+1}(\emptyset) \Rightarrow f^{i+1}(\emptyset) = f^{i+2}(\emptyset)$. En particulier,

$$\bigcup_i f^i(\emptyset) = f^k(\emptyset) = f^n(\emptyset)$$

puisque $k \leq n$ (car $n_k \leq n$).

L'algorithme suivant présente le pseudo-code pour le calcul itératif du plus petit point fixe.

```
Function mu(f: P(S) → P(S)) : P(S) ;
      Var X, Y ;
      Const Ensemble_Vide
   begin
      X := Ensemble_Vide ;
      Y := f(X) ;
      while X <> Y do
      begin
         X := Y ;
         Y := f(Y) ;
       end ;
       return X ;
end
```

Ce deuxième algorithme calcul le plus grand point fixe :

```
Function mu(f: P(S) → P(S)) : P(S) ;
      Var X, Y ;
    begin
      X := S ;
      Y := f(X) ;
      while X <> Y do
      begin
         X := Y ;  Y := f(Y) ;
       end ;
       return X ;
end
```

Exemple 5.35 : Soit M une machine d'états finis représentant le comportement d'un système réactif. Nous allons vérifier la validité de la formule temporelle AG(a ∨ c). Pour cela nous utilisons la structure de M (Figure 5.16) et l'algorithme qui calcule le plus petit point fixe.

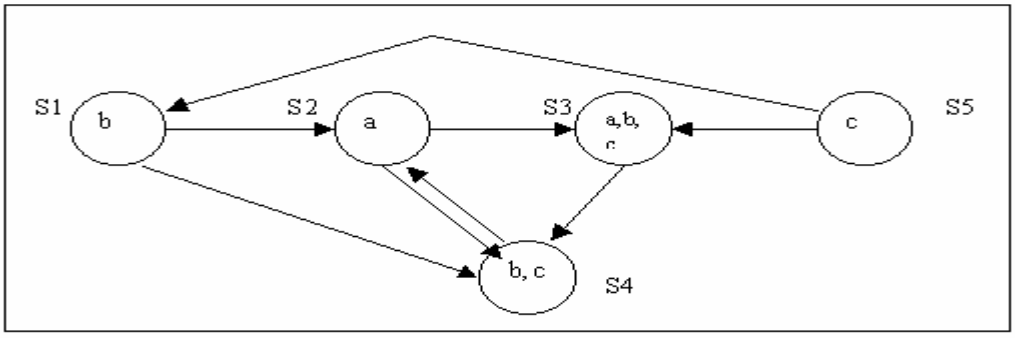


Figure 5.13 : Structure d'un FSM Relative à Un Système Réactif

Après plusieurs itérations et suite au calcul de H(AG(a ∨ c)) ={ 2, 3, 4}, nous concluons que les états états S2, S3 et S4 vérifient la propriété AG(a ∨ c).

On procédera de la même manière pour le calcul du plus petit point fixe à la seule différence que l'ensemble X soit initialisé à l'ensemble S.

Dans ce qui suit, on sera intéressé par l'introduction des fonctions monotones définies sur les treillis des formules de CTL.

5.6.2 Caractérisation des points fixe pour CTL

Nous venons de voir qu'il était possible de calculer les points fixes d'un treillis en utilisant une procédure itérative très simple. Dans ce qui suit deux procédures similaires seront utilisées pour le calcul des fonctions caractéristiques de certains opérateurs de la logique CTL. Le principe est le même, en plus de cela on tiendra compte des points suivants :

Une lattice complète doit tout d'abord être définie sur les formules logiques de CTL telles que l'existence (unicité) du plus petit et plus grand point fixe qui doit être garantie. La base de cette lattice est une relation d'ordre sur les formules CTL.

Les fonctions monotones sur les formules CTL doivent être déterminées telles que les formules $E(\phi\ U\ \psi)$ et $A(\phi\ U\ \psi)$ soient caractérisées comme des plus petits (plus grands) points fixes de ces fonctions. Pour se faire l'aximatisation de CTL est plutôt nécessaire pour introduire ce qui suit.

5.6.2.1 Un Treillis Complèt des formules-CTL

La relation d'ordre partiel $\Downarrow$ sur les formules de CTL est définie en associant à chaque formule Φ l'ensemble des états s de S dans la structure de Kripke M pour lesquels Label(s) = Φ. Cependant Φ est identifié comme l'ensemble :

$$[|\Phi|] = \{s \in S \mid M, s \models \Phi\}.$$

On défint l'ordre $\Downarrow$ par :

$$\Phi \Downarrow \Psi \text{ si et seulement si } [|\Phi|] \text{ ™ } [|\Psi|].$$

Nous pouvons remarquer, qu'à ce point de raisonnement la relation $\Downarrow$ correspond désormais à la relation ™ définit sur les sous-ensembles. Notons que la notation de $[|\Phi|]$ ™ $[|\Psi|]$ est équivalente à $\Phi \Rightarrow \Psi$. De ce fait, (CTL, $\Downarrow$) est définit comme étant une lattice complète. La limite inférieure (lower bound) est la conjonction qui sera construite autour de :

$$[|\Phi|] \cap \Psi = [|\Phi \wedge \Psi|].$$

et la borne supérieure correspond à la disjonction :

$$[|\Phi|] \cup \Psi = [|\Phi \vee \Psi|].$$

Comme l'ensemble des formules CTL est fermé sous la conjonction et la disjonction, il s'ensuit que pour tout Φ et Ψ, leur borne inférieure et supérieure existent. Le plus petit élément de la lattice (CTL, $\Downarrow$) est *false*, cependant $[|false|] = \varnothing$. De façon similaire, le plus grand élément dans (CTL, $\Downarrow$) est *true*, donc $[|false|] = S$, l'ensemble des états de M.

Rappelons un résultat vu dans la section (CTL) matérialisé par l'axiome d'expansion existentiel, à savoir :

$$E(\Phi\ U\ \psi) \equiv \psi \vee (\Phi \wedge EX\ E(\Phi\ U\ \psi))$$

La nature récursive de cette règle suggère de considérer la formule $E(\Phi\ U\ \psi)$ comme le point fixe de la fonction F définie comme suit :

$$F(Z) = \psi \vee (\Phi \wedge EX\ Z)$$

après remplacement de $E(\Phi\ U\ \psi)$ par Z juste pour donner une forme simple à la formule.

De la même manière, nous pouvons exprimer les autres opérateurs de CTL par cette notion de point fixe. Par exemple l'opérateur **Until** s'écrira :

$$F_{EU}(Z) = [|\psi|] \text{ ® } ([|\Phi|] \langle \{s \in S \mid R(s) \langle Z \neq \varnothing\}).$$

Le théorème suivant généralise les résultats de calcul des opérateurs CTL comme un calcul de point fixe :

Théorème 5.36 *Pour toutes formules ψ et Φ nous avons :*

- $[|E(\Phi\ U\ \psi)|]$ *est le plus petit point fixe (lfp) de* $F_{EU}(Z) = [|\psi|] \circledR ([|\Phi|] \cap \{s \in S \mid R(s) \cap Z \neq \varnothing\})$
- $[|A(\Phi\ U\ \psi)|]$ *est le plus petit point fixe de* $F_{AU}(Z) = [|\psi|] \circledR ([|\Phi|] \cap \{s \in S \mid R(s) \subseteq Z\})$
- $[|E\ \Phi|]$ *est le plus grand point fixe de* $F_{EG}(Z) = ([|\Phi|] \cap \{s \in S \mid R(s) \cap Z \neq \varnothing\})$
- $[|AG\ \Phi)|]$ *est le plus grand point fixe de* $F_{AG}(Z) = ([|\Phi|] \cap \{s \in S \mid R(s) \subseteq Z\})$
- $[|EF\ \Phi)|]$ *est le plus petit point fixe de* $F_{EF}(Z) = ([|\Phi|] \circledR \{s \in S \mid R(s) \cap Z \neq \varnothing\})$
- $[|AF\ \Phi)|]$ *est le plus petit point fixe de* $F_{AF}(Z) = ([|\Phi|] \circledR \{s \in S \mid R(s) \subseteq Z\})$.

5.6.2.2 Algorithme du Model Checking Symbolique

L'algorithme du Model Checking Symbolique procède comme suit :

• Soit p un ensemble d'états (graphe d'états) et **p** sa représentation symbolique Booléenne sous forme d'un ROBDD , nus avons alors

$$p = \lambda (v\,1 , v\,2 , ..., v\,n)\ \mathbf{p}$$

• Pour une relation R sur les états, il existe une seule représentation **R** telle que

$$R = \lambda (v\,1 , v\,2 , ..., v\,n , v\,1\,' , v\,2\,' , ..., v\,n\,').\ \mathbf{R}$$

Ces résultants permettent de calculer la forme symbolique des opérateurs temporels de CTL, par exemple :

Le calcule de **EXp** s'effectue comme suit :

• $EXp = \lambda\ v.\ \exists\ v'(R(v, v') \wedge p(v'))$, ou $v = (v\,1 , v\,2 , ..., v\,n)$, $v' = (v\,1\,' , v\,2\,' , ..., v\,n\,')$

R(v, v') (relation) = **R**

p(v') (expression logique) = **p**', ou **p'** = $p[v\,i \leftarrow v\,i\,']$ $\Rightarrow$ $EXp = \lambda\ v.\ \exists\ v'(\mathbf{R} \wedge \mathbf{p}')$.

Algorithme: Soit **p** la représentation booléenne de p;

Début;

p' := $p[v\,i \leftarrow v\,i\,']$;

$S(v) := \exists\ v'$ **(R** $\wedge$ **p**');

Check if initial state $s\,0 \in S(v)$.

Fin.

Exemple 5.37

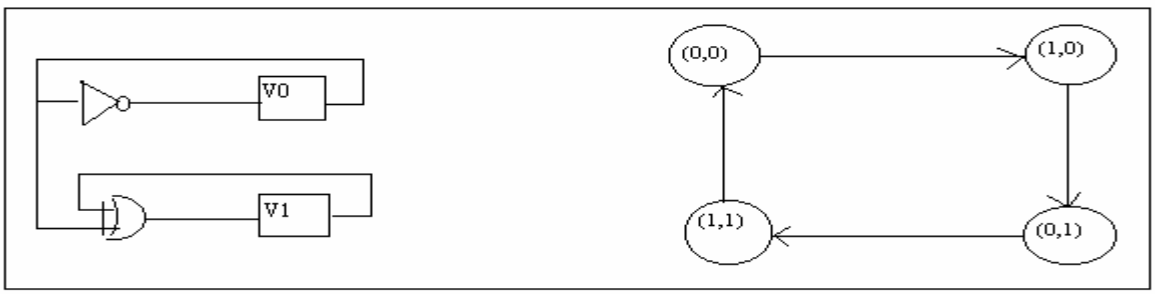


Figure 5.14 : Circuit Analogique d'un Compteur et son Graphe d'Etat (FSM)

De la spécification du Compteur nous tirons les paramètres suivants :

• State variables: v 0 , v 1 , {v = (v 0 , v 1)}

• Variables d'Etats Next: v 0 ', v 1 ', {v' = (v 0 ', v 1 ')}

• Relation de Transition: **R** = (v 0 ' $\Leftrightarrow \neg$ v 0) $\wedge$ (v 1 ' $\Leftrightarrow$ (v 0 $\oplus$ v 1))

Vérification de : EX(v 0 $\wedge$ v 1)

= $\exists$ v'. **(R** $\wedge$ **p'**)

= $\exists$ (v 0 ', v 1 '). **(R** $\wedge$ (v 0 ' $\wedge$ v 1 '))

= $\exists$ (v 0 ', v 1 '). ([(v 0 ' $\Leftrightarrow \neg$ v 0) $\wedge$ (v 1 ' $\Leftrightarrow$ (v 0 $\oplus$ v 1))] $\wedge$ (v 0 ' $\wedge$ v 1 '))

= $\exists$ v 0 ' . ((v 0 ' $\Leftrightarrow \neg$ v 0) $\wedge$ (v 0 $\oplus$ v 1) $\wedge$ v 0 ')

= $\neg$ v 0 $\wedge$ (v 0 $\oplus$ v 1)

= $\neg$ v 0 $\wedge$ (($\neg$ v 0 $\wedge$ v 1) $\vee$ (v 0 $\wedge \neg$ v 1))

= $\neg$ v 0 $\wedge$ v 1

• **Ce résultat signifie que l'état** (0, 1) satisfait EX(v 0 $\wedge$ v 1)

Vérification de : EF(v 0 $\wedge$ v 1) = μ y. ((v 0 $\wedge$ v 1) $\vee$ EXy)

τ 1 [0] = (v 0 $\wedge$ v 1) $\vee$ EX0 = (v 0 $\wedge$ v 1)

τ 2 [0] = (v 0 $\wedge$ v 1) $\vee$ EX(v 0 $\wedge$ v 1)

= (v 0 $\wedge$ v 1) $\vee$ ($\neg$ v 0 $\wedge$ v 1) {from the result of EX(v 0 $\wedge$ v 1)}

= v 1

τ 3 [0] = (v 0 $\wedge$ v 1) $\vee$ EX(v 1)

= (v 0 $\wedge$ v 1) $\vee$ [$\exists$ (v 0 ', v 1 '). (**R** $\wedge$ v 1 ')]

= (v 0 $\wedge$ v 1) $\vee$ [$\exists$ (v 0 ', v 1 '). ((v 0 ' $\Leftrightarrow \neg$ v 0) $\wedge$ (v 1 ' $\Leftrightarrow$ (v 0 $\oplus$ v 1)) $\wedge$ v 1 ')]

= (v 0 $\wedge$ v 1) $\vee$ [$\exists$ v 0 '. ((v 0 ' $\Leftrightarrow \neg$ v 0) $\wedge$ (v 0 $\oplus$ v 1))]

= (v 0 $\wedge$ v 1) $\vee$ [(v 0 $\wedge$ (v 0 $\oplus$ v 1)) $\vee$ ($\neg$ v 0 $\wedge$ (v 0 $\oplus$ v 1)]

= (v 0 $\wedge$ v 1) $\vee$ (v 0 $\oplus$ v 1) = (v 0 $\wedge$ v 1) $\vee$ ($\neg$ v 0 $\wedge$ v 1) $\vee$ (v 0 $\wedge \neg$ v 1) = v 0 $\vee$ v 1

τ 4 [0] = (v 0 $\wedge$ v 1) $\vee$ EX(v 0 $\vee$ v 1)

= (v 0 $\wedge$ v 1) $\vee$ [$\exists$ (v 0 ', v 1 '). (**R** $\wedge$ (v 0 ' $\vee$ v 1 '))]

= (v 0 $\wedge$ v 1) $\vee \neg$ v 0 $\vee$ (v 0 $\oplus$ v 1)

= (v 0 $\wedge$ v 1) $\vee \neg$ v 0 $\vee$ ($\neg$ v 0 $\wedge$ v 1) $\vee$ (v 0 $\wedge \neg$ v 1) =1

• EF(v 0 $\wedge$ v 1) ={(0, 0), (0, 1), (1, 0), (1, 1)}. Ce résultat signifie que tous les états vérifient la propriété en question.

5.6.3 La génération de Contre-exemples

Tous les model-checker(s) et sans exception utilisent des mécanismes qui leur permettent de vérifier la validité d'une formule temporelle. Dans le cas affirmatif, un message simple et court est communiqué à l'utilisateur disant que le programme en question est valide, c'est-à-dire que la formule est vérifiée sur la structure de Kripke. Dans le cas contraire, le model-checker imprime un contre-

exemple qui montre clairement la non-validité de la formule. A partir de ce contre-exemple, le concepteur revoit le modèle et lui apporte les corrections nécessaires et ainsi de suite.

Dans ce qui suit, nous allons donner le concept même de la notion très importante dans les model-checking de contre-example.

Un contre-example est supposé être une séquence finie ou infinie d'états qui indiquent clairement pourquoi la formule en question n'est pas valide (refusée). Par exemple, considérons la formule AG p prise dans l'état initial (notre exemple test de la Figure 5.11). Cette formule est refusée et un contre-exemple possible qui est le chemin $s_1s_3s_0s_2$ (Figure 5.11). Nous pouvons voir que label(s_2) est différent de p, ce qui revient à dire que la formule AG p n'est pas vérifiée en s_2, et donc elle ne l'est pas sur le chemin et par la même occasion AG p est refusée sur tout le graphe de Kripke. Nous pouvons remarquer que comme AG p est considéré sur un état initial qui est d'ailleurs le préfixe d'un chemin d'exécution satisfaisant $\neg$ AG p, donc son équivalent EF $\neg$p. Si nous voulons généraliser ce résultat, nous pouvons dire que pour les formules du type AG Φ, un contre-exemple est une séquence qui conduit à un état qui à pour label $\neg\Phi$. De façon similaire, le contre-exemple relatif à la formule AX Φ est une séquence satisfaisant EX $\neg\Phi$. En général, un contre-exemple d'une formule quantifiée universellement, serait un témoin quantifié par l'opérateur existentiel et vice-versa. Notons, qu'il est impossible de générer des séquences de contre-exemple pour les formules du type E Φ lorsque la vérification échoue, puisque E Φ est refusée si et seulement si aucun chemin ne vérifie Φ. En revanche, en cas de succès, l'exécution peut être retournée

5.7 Vérification des Systèmes Temps-réel

Dans la suite de ce chapitre nous étudions la logique TCTL [ACD90, Alu91, HNS+92] qui est l'extension de CTL permettant d'exprimer des propriétés temps-réel. Une façon simple d'introduire le temps explicitement dans la syntaxe consiste à borner la portée dans le temps des opérateurs temporels. Pour bien comprendre l'utilité d'une telle logique, prenons le fait d'écrire en CTL $\forall\Box\exists\Diamond$p qui exprime qu'il est toujours possible d'atteindre un état qui satisfait la propriété p. Cette propriété ne nous renseigne guère sur le temps qui va s'écouler pour la réalisation d'une telle propriété. L'extension de CTL en TCTL permet justement de considérer ce type de propriétés. Par exemple nous pouvons écrire $\exists\Diamond_{< 3}$ p pour exprimer qu'il existe une exécution où la propriété p est vraie avant 2 unités de temps. Dans la littérature plusieurs autres notations (approches) ont été introduites pour prendre en charge les systèmes temporisés, nous pouvons citer les travaux de Alur [Alu98], Henziger et al. [HHW97] Manna et Pnueli [Hal93], de Sifakis et al [HNS+94]., Alur et Henzinger [AH90, AFH91], Bery et Al. [BB91]. Une autre possibilité pour la spécification des contraintes temporelles des systèmes réactifs, n'introduit aucun opérateur nouveau, mais au lieu de cela, on utilise une variable appelée **now** juste pour mémoriser le temps qui passe. Cette approche est appelée *Explicit-clock approach*. La première idée d'une telle approche fut introduite dans [Har88] puis améliorée est appelée *Real-time Temporal Logic*. Pour bien comprendre ces deux approches et les comparer, prenons un exemple simple. Considérons la réponse *q* à un stimulis *p* qui doit s'effectuer dans au plus 3 unités de temps. Cette propriété s'exprime dans la première approche par $p \rightarrow \Diamond_{\leq 3} q$.

Dans la deuxième approche, cette propriété s'écrit par :

$$(p \wedge now = t) \rightarrow \Diamond(q \wedge now \leq t + 3),$$

L'invariant t dans ce cas est associé à l'état p.

5.7.1 Modélisation des systèmes temps-réel

Nous définissons la sémantique formelle des systèmes temps–réel comme un ensemble de séquences d'exécutions qui s'effectuent en deux étapes. La première étape, introduit la notion d'abstraction des systèmes de transitions temporisées et identifie les séquences d'exécutions d'un système quelconque (temporisé). Dans l'étape 2, la notion de système temps-réel concret est abordée pour montrer que les

systèmes temps-réel peuvent être modélisés par les systèmes abstraits. La présentation de ces deux notions sera explicitée à travers des exemples.

5.7.1.1 Automates Temporisés

Les logiques temporelles qu'on a cité auparavant, sont des logiques interprétées sur les graphes de Kripke qui permettent de décrire comment évolue un système réactif d'un état vers un autre état. Les aspects temporisés sont jusqu'à l'instant non couverts, c'est-à-dire qu'aucune information n'a été présentée pour modéliser en tant que quantité matériel à prendre en compte. Si nous voulons par exemple représenter le temps qu'un processus doive mettre pour résider dans un état, ceci pour le moment est un petit peu impossible.

Les systèmes réactifs qui dans leur ensemble sont à caractéristique temps-réel doivent réagir dans le temps, ils sont de fait des systèmes temps-réel critique. Le comportement de tels systèmes est alors soumit à des contraintes temporelles. Pour mieux exprimer ce type de comportement, prenons le cas d'un train qui s'approche d'un passage à niveau. Il est par conséquent essentiel qu'une fois le train soit détecté à une distance connue, la barrière du passage se refermera de suite pendant un certain temps qui doit être connu à l'avance de façon à ce que les automobilistes sur l'autre route soient empêchés de traverser le passage. Prenons un autre exemple, il est impératif que la dose qui doit être injectée par un dispositif de radiation (radio-thérapie) à un patient cancéreux ne dépassera jamais un laps de temps, ne serait-ce très petit, ce qui forcement peut causer la mort.

En définitive, comme le temps est d'une importance vitale aux systèmes réactifs, il est essentiel que les contraintes temporelles du système doivent être absolument assurées. Dans ce cas de figure, le comportement du système sujet à de telles contraintes doit absolument être vérifié. Cette vérification utilise des algorithmes bien spécifiques dépendant de l'aspect temps. Un grand nombre d'algorithmes et de systèmes se trouvent de fait exploré. Nous citons à titre d'exemple Uppaal [UPPAAL, LPY97], Step [STEP], Lustre [Rat92], Kronos [KRONOS] etc.

Les ingrédients essentiels utilisés dans les model-checking temporisés sont (1) un langage de description du modèle, (2) un langage de spécification des propriétés et (3) des algorithmes de vérification.

Pour mieux situer notre présentation des model-checking temps-réel quelques éléments doivent être tout d'abord introduits. Nous présenterons un type particulier d'automate dit temporisé, qui est un automate à états finis dont les transitions sont étiquetées par des variables appelées horloges (clocks) utilisées pour mesurer le passage du temps.

5.7.1.1.1 Eléments de Base des Automates Temporisés

Un automate temporisé est utilisé pour modéliser un système temps-réel critique. Par conséquent, il est essentiel de définir le temps généré par une horloge (clock).

Définition 5.38 (Clock) *Une horloge est une variable prenant ces valeurs dans* IR^+.

Dans la suite, les lettres x, y et z seront utilisées pour représenter des variables de type clocks (horloges). Un état de cet automate définit l'état actuel de la *location* plus les valeurs courantes des variables clocks.

Les horloges peuvent être initialisées à zéro quand le système procède à une transition. Une fois initialisées à zéro, les variables horloges incrémentent leurs valeurs implicitement d'une unité de temps et ceci simultanément pour toutes les horloges. La valeur d'une horloge à un moment précis fournit le temps passé depuis son initialisation. Informellement, ces horloges peuvent être considérées comme des chronomètres qui évoluent indépendamment les uns des autres. Le plus important dans tous ces comportements c'est le système global qui possède quant à lui sa propre horloge appelée aussi l'horloge globale à laquelle toutes les autres horloges se réfèrent.

Le système évolue donc en restant carrément dans une location (état), sinon il peut se déplacer vers une autre location empruntant une transition. Le passage sur une transition est instantané est supposé ne pas prendre de temps (free of time). Les contraintes sur les horloges sont utilisées pour permettre le franchissement ou non des transitions, ce sont des gardes (guards or enabling conditions). Pour le moment et par simplicité les conditions d'autorisation (enabling conditions) sont supposées ne dépendre que des horloges, mais en réalité le problème est autre, c'est-à-dire que le passage peut dépendre aussi d'autres facteurs. Des invariants sur les horloges sont aussi utilisés pour limiter le temps qu'il faut mettre pour rester dans une location. En d'autres termes, les invariants et les conditions de passage (guards) sont des conditions utilisées sur les horloges.

Définition 5.39 (Clock constraints). *Soit C un ensemble d'horloges, et soit x une horloge de C ($x \in C$) et c un nombre naturel. Les contraintes sur les horloges satisfont les règles suivantes :*

- $x < c$ *et* $x \leq c$ *sont des contraintes sur les horloges.*
- *Si* α *est une contrainte horloge alors* $\neg \alpha$ *est aussi une contrainte horloge.*
- *Si* α *et* β *sont des contraintes horloges, alors* $\alpha \wedge \beta$ *est une contrainte horloge.*
- *Toute autre formule n'est pas une contrainte horloge.*

L'ensemble des contraintes sur les horloges est dénoté ***Constraints(C).***

Dans ce qui suit nous considérons les abréviations comme $x \geq c$ pour $\neg (x < c)$ and $x = c$ for $x \leq c \wedge x \geq c$. Le choix de contraintes légales est très important. Nous pouvons par exemple autoriser l'addition de contraintes (e.g $x < c+d \mid d \in IN$) sans causer de problème. Par contre l'addition $x+y \leq 3$, risque de créer un problème et le model-checking devient indécidable. Aussi, si c est un nombre réel tel que $x \leq 2\pi$, alors le domaine ainsi utilisé sera un domaine dense (continu) et le problème devient de fait indécidable. Cependant c de préférence doit être un nombre naturel. La décidabilité ne sera pas affectée si c doit être un nombre rationnel puisque c peut être converti en un nombre naturel. Nous pouvons aussi autoriser d'autres variantes de c et à chaque fois de choisir le bon multiplicateur.

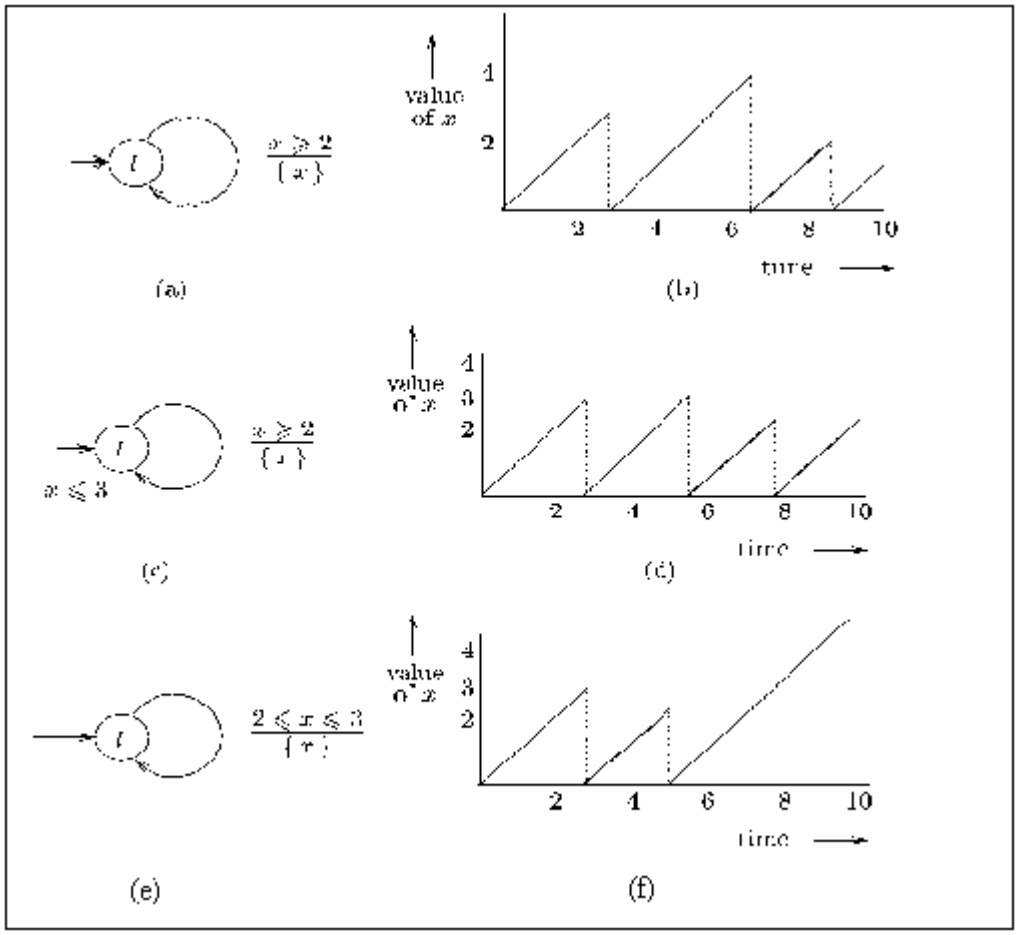


Figure 5.15 : Trois automates pourvus d'une seule horloge et différentes contraintes sur les transitions

Définition 5.40 (Automate temporisé) *Un automate temporisé A est un sixtuplet (L, I, C, →, Label, inv) avec :*

- *L, un ensemble non-vide et fini de location.*
- *$I \subseteq L$, l'ensemble de locations initiales.*
- *C, un ensemble fini d'horloges.*
- *$\rightarrow \subseteq L \times (Constraints(C) \times 2^{c}) \times L$, la relation de transition*
- *$Label : L \rightarrow 2^{AP}$, une fonction d'interprétation sur L*
- *$inv : L \rightarrow Constraints(C)$, une fonction d'assignation des invariants.*

Un automate temporisé peut être assimilé à une structure de Kripke équipée d'un ensemble C d'horloge. Les transitions sont étiquetées par de couple (α, C') ou α est une contrainte sur les horloges et C' ⊆ C est un ensemble d'horloges. Par la suite, on adoptera l'écriture $l \xrightarrow{\alpha, C} l'$ au lieu de *(l, α, C, l')* ∈ →. L'interprétation intuitive de cette écriture est que l'automate peut évoluer de la location l vers la location l' lorsque la contrainte α devient vraie. Aussi le déplacement d'une location l vers la location l' réinitialise toute horloge de C' à zéro. La fonction Label possède la même fonction que pour les structures de Kripke telle que vu auparavant. Finalement, la fonction *inv* affecte un invariant à toute location du système. Pour toute location l, *inv*(l) définie la durée du temps qui doit passer dans la location l. Une fois que *inv*(l) devienne invalide la location l doit être libérée immédiatement. Si une

telle situation n'est plus possible peut être qu'il n'y a plus une transition qui doit être empruntée et comme le temps passe inéluctablement, alors une telle situation est dite *timelock*.

Pour mieux représenter graphiquement les automates temporisés, nous adopterons les conventions suivantes. Les cercles dénoteront les locations (états) et les transitions sont représentées par des flèches. Les invariants sont écrits à l'intérieur des locations sinon sur le coté, sauf si aucune écriture n'est présente, dans ce cas la location est *true* dans toutes les situations. Les transitions (flèches) sont équipées d'étiquettes qui consistent d'une contrainte sur les horloges (optionnelle) et d'une autre qui représente la variable horloge qui doit être réinitialisée (optionnelle), séparées par une ligne droite. Si la contrainte sur l'horloge est égale à *true* et qu'aucune variable horloge ne doit être initialisée alors la transition est libre de toute indication.

Exemple 5.41 La Figure 5.15-(a) représente un automate temporisé très simple pourvu d'une seule horloge x est d'une location équipée d'une boucle. La transition peut être traversée si la valeur de x est au moins égale à 2, et quand ceci est possible l'horloge x est ré-initialisée à 0. La Figure 5.15-(b) présente un exemple d'exécution de cet automate en donnant les valeurs du temps selon x. Chaque fois que la valeur de x est au zéro, l'automate procède à un déplacement de la location l vers elle-même. Aussi compte tenu que la valeur de l'invariant qui est *true* sur l, ceci permet au temps de progresser sans restriction aucune.

Procédons à un petit changement en incorporant un invariant $x \leq 3$ dans la location l, ceci conduit au fait que x ne pourra plus évoluer librement. Si $x \geq 2$ (enabling constraint) et $x \leq 3$ (invariant) la transition dans ce cas doit être traversée (Figure 5.1(c) et (d)).

Observons que le même effet ne peut pas survenir en changeant la contrainte sur l'horloge à $2 \leq x \leq 3$ et en gardant l'invariant à true. Dans ce cas la transition de sortie ne peut être traversée que pour le cas $2 \leq x \leq 3$, mais ceci peut ne pas se faire en laissant passer le temps (figure 5.15-c et f).

De cet exemple, il est clair que les horloges prennent des valeurs réelles continues par morceaux et la discontinuité du temps peut survenir lorsqu'une transition est traversée.

Les horloges peuvent être initialisées à des temps divers, ce qui en fait, exclu toute limite de leur différence. Ceci n'est pas possible pour le cas d'un système à temps discret puisque les valeurs des horloges sont des multiples de un click (une unité).

Example 5.42. Figure 5.16 représente l'automate temporisé d'un interrupteur. Deux locations sont tout à fait distinctes $l_1 = \{off\}$ et $l_2 = \{on\}$, et deux variables d'horloge x et y. Au début, toutes les horloges sont initialisées à zéro.

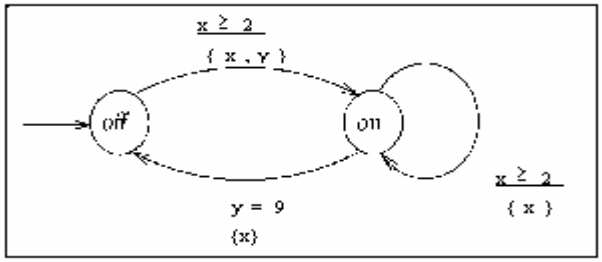


Figure 5.16 : L' automate temporisé d'un interrupteur

L'interrupteur peut à tout instant rester dans la position (location) ***on*** pourvu qu'il reste sur le ***on***. Comme les deux horloges évoluent au même temps et avec le même taux, l'automate passe de la position ***on*** à la position ***off*** exactement 9 unités de temps après que l'interrupteur sera passé de ***off*** à ***on***. L'horloge ***x*** retient le temps depuis la dernière fois que l'interrupteur est passé à ***on***. La transition portant la contrainte sur l'horloge $x \geq 2$ modélise la condition de passage vers la location ***on***. La variable horloge ***y*** est utilisée pour compter le temps depuis la derrière fois que l'interrupteur est passé de ***off*** vers le ***on***, donc, il contrôle aussi l'extinction de la lumière.

5.7.1.1.2 Sémantique des Automates Temporisés

L'exemple de la figure 5.16 démontre que l'automate temporisé est déterminé par :

- Sa position actuelle (location)
- La valeur courante de ces horloges

En fait, l'automate temporisé est assimilé à un système à transitions étiquetées, pourvu d'un nombre infini d'états (possiblement) et d'un nombre de transitions à branchement infini (possiblement).

Définition 5.43 **(**Clock valuation**)** *Une valuation d'horloge v pour un ensemble C d'horloges est la fonction* $v : C \rightarrow IR^+$*, affectant à chaque horloge* $x \in C$ *sa valeur courante v(x).*

Supposons que *Val*(C) dénote l'ensemble de toutes les valuations d'horloge sur C. Un état de *A* est la paire *(l, v),* ou l est une location dans A et *v* une valuation sur C, l'ensemble des horloges de *A*.

Exemple 5.44 *Considérons l'automate de la Figure 5.16. Quelques états de cet automate sont les paires (off, v) telles qu'avec* $v(x) = v(y) = 0$ *et l'état (off, v') avec* $v'(x) = 4$ *et* $v'(y) = 13$ et l'état *(on, v") avec* $v''(x) = 2.7$ *and* $v''(y) = 13$.

Nous remarquons que le dernier état n'est pas accessible.

Soit *v* une valuation d'horloge de C. Pour un nombre positif *d*, la valuation de l'horloge *v+d* dénote que toutes les horloges ont augmenté de *d* unités. Ceci est définit par $(v+d)(x) = v(x)+d$ pour toutes les horloges $x \in C$. La valuation de l'horloge re-initialise x, et la valuation *v* avec l'horloge x, est définie par :

$$(reset \quad x \quad in \quad v) \quad = \quad \begin{cases} v(y) & if \quad y \neq x \\ 0 & if \quad y = x \end{cases}$$

Exemple 5.45 *Reprenons notre exemple et considérons les valuations v and v'. La valuation v+9 est définie par (v+9) (x) = (v+9) (y) = 9. Dans la valuation* reset *x* in (v+9), l'horloge *x reçois la valeur 0 et l'horloge y la valeur 9. La valuation v' sera égale à* (*reset x in (v+9)*) + 4.

Nous pouvons maintenant donner un sens à la notion de validité pour une contrainte horloge. Ceci s'effectue exactement de la même manière que pour la logique temporelle simple en définissant une certaine relation de satisfaction |= qui est une relation entre les valuations horloges et les contraintes horloges.

Définition 5.46 *(Evaluation of Block constraints) Pour tout* $x \in C$, $v \in Val(C)$, *et soient c un nombre naturel et* $\alpha, \beta \in$ *Constraints(C) nous avons:*

- $v \models x \leq c$ *iff* $v(x) \leq c$
- $v \models x < c$ *iff* $v(x) < c$
- $v \models \neg\alpha$ *iff* $\neg (v \models \alpha)$
- $v \models \alpha \wedge \beta$ *iff* $(v \models \alpha) \wedge (v \models \beta)$.

Pour la négation et la conjonction, les règles sont identiques à celles de la logique propositionnelle. Pour vérifier que $x \leq c$ est valide dans *v,* il serait alors simple de vérifier que $v(x) \leq c$. Il en est de même pour $x < c$.

Exemple 5.47 *Considérons la valuation v, v+9 et* reset *x* in *(v + 9) de l'exemple 5.42. Supposons que nous voudrions vérifier la validité de* $\alpha = x \leq 5$. *Il s'ensuit que* $v \models \alpha$, *et* $v(x) = v(y) = 0$. *Nous avons* $\neg(v+9 \models \alpha)$, *comme* $(v+9)(x) = 9 > 5$, *il s'ensuit alors que* reset *x* in *(v+9)* $\models \alpha$.

5.7.2 Systèmes de Transitions Temporisées

L'interprétation des automates temporisés [AD90, Alu98, AH92] est définie en terme de système de transitions infinies (S, →), avec S, le nombre des états (paires de location et de valuation d'horloge) et → étant la relation de transition qui définie que le système sera en mesure d'évoluer d'un état à un autre. Cette relation de transition est tout à fait différente que celle définie pour les automates temporisés. Nous savons déjà que l'automate temporisé se déplace d'un état à un autre uniquement sous les deux conditions, à savoir (1) en traversant une transition de l'automate ou bien (2) en laissant le temps passer en restant dans la même location. On rencontre alors deux types de transitions; l'une à valeurs discrètes et l'autre à valeurs continues prenant ses valeurs dans l'ensemble des nombres réels positifs.

Définition 5.48 *Soit A = (L, I, C, →, Label, inv) un automate temporisé. Le système de transitions étiquetées associé à l'automate A qu'on dénote par [A], est définie par (S, I', →) ou*

- $S = \{ (l, v) \in L \times Val(C) \mid v \models inv(l) \}$
- $P = \{(l_o, v_o) \mid l_o \in I\}$ *et* $v_o(x) = 0$ *pour tout* $x \in C$

La relation de transition $S \times (IR^+ \cup \{*\}) \times S$ *est la plus petite relation définie par les règles :*

1. $(l, v) \xrightarrow{*} (l', \text{reset } C' \text{ in } v)$ *si les conditions suivantes sont respectées :*
   - (a) $l \xrightarrow{\alpha, C} l'$
   - (b) $v \models \alpha$, *et*
   - (c) $(\text{reset } C' \text{ in } v) \models (inv(l'))$
2. $(l, v) \xrightarrow{d} (l, v + d)$ *, pour les valeurs de d positif et la condition suivante soit vraie:* $\forall d' \leq d.\ v+d' \models inv(l)$.

Définition 5.49 *(Chemin) Un chemin de A est une séquence infinie* $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$ *Telle que* $s_i \xrightarrow{a_i} s_{i+1}$ *pour tout* $i \geq 0$.

Définition 5.50 (Elapsed time on a path) Pour un chemin $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$et i un nombre naturel, on définie le temps écoulé entre s_0 et s_i dénoté $\Delta(\sigma, i)$, le temps accumulé depuis s_0 jusqu'à ce que le s_i eut été atteint (continu ou discret).

Exemple 5.51 *Rappelons l'exemple de l'interrupteur de courant.*

$\sigma = (Off, v_0) \xrightarrow{3} (Off, v_1) \xrightarrow{*} (on, v_2) \xrightarrow{4} (on, v_3) \xrightarrow{*} (on, v_4) \xrightarrow{1} (on, v_5) \xrightarrow{2} (on, v_6) \xrightarrow{2} (on, v_7) \xrightarrow{*} (Off, v_8) \ldots$

avec $v_o(x) = v_o(y) = 0$, $v_1 = v_o+3$, $v_2 = $ reset x, y in v_1, $v_3 = v_2+4$, $v_4 = $ reset x in v_3, $v_5 = v_4+1$, $v_6 = v_5+2$, $v_7 = v_6+2$ and $v_8 = $ reset x in v_7.

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
x	0	3	0	4	0	1	3	5	0
y	0	3	0	4	4	5	7	9	9

Table 5.17 : Calcul du Comportement d'une exécution

La transition (off, v_1) $\xrightarrow{*}$ (on, v_2) est par exemple possible puisqu'il existe une transition de la location **off** vers la location **on**, aussi comme $v_1 \models x \geq 2$, et puisque $v_1(x) = 3$, et $v_2 \models in\ v(On)$. Nous avons aussi par exemple que $\Delta(Q, 3) = 7$ and $\Delta(Q, 6) = 12$.

Une autre possibilité du comportement de l'interrupteur, c'est de rester dans la location off infiniment longtemps lorsque l'horloge prend ses valeurs dans IR^+. De façon similaire, il peut rester dans la position **on** infiniment longtemps. Ce comportement est causé par le fait que inv(of) = inv(on) = *true*.

Définition 5.52 (Time-divergent)[ACH+95] *Un chemin σ est dit divergent si la limite $lim_{i\to\infty}$ $\Delta(\sigma, i) = \infty$.*

L'ensemble des chemins divergents prenant naissance en *s* sont dénotés *Paths'(s). Un exemple d'un chemin* non time-divergent est un chemin qui visite un nombre infini d'états pendant un temps borné. Par exemple, le chemin

$$\sigma = s_0 \xrightarrow{2^{-1}} s_1 \xrightarrow{2^{-2}} s_2 \ldots s_k \xrightarrow{2^{-k+1}} s_{k+1} \ldots$$

n'est pas divergent, puisqu'un nombre infinie d'états est visité pendant un intervalle de temps [1/2, 1]. L'automate temporisé est exclu totalement de tel chemin (zenoness paths).

Définition 5.53 (Non-Zeno timed automaton) [AH92] *Un automate temporisé A est dit non-zeno si à partir de n'importe quel état de cet automate un nombre de chemins divergents peuvent prendre naissance.*

5.7.3 Composition d'Automates Temporisés

Pour pouvoir réaliser la composition de deux automates temporisés, nous devons adopter quelques conventions. Nous supposerons que sur les transitions on aura une condition qui autorisera le passage plus un ensemble d'horloge qui peuvent être ré-initialisés pouvant étiqueter les transitions. Les actions sont supposées prendre leurs valeurs dans un ensemble E d'alphabet. Maintenant les transitions sont étiquetées par le triplet *(a,α, C')* *ou* *a est une action, la* condition de passage α et l'ensemble d'horloges C'. L'addition des actions nous aide à spécifier les systèmes temporisés comme une composition de ces derniers.

Définition 5.54 (Composition d'automates temporisés) *Soient A_i un ensemble d'automates temporisés (S_i, L_i, I_i, C_i, $\rightarrow_i$, $Label_i$, inv_i). Pour $i = 1, 2$ et $C_1 \cap C_2 = \varnothing$. La composition de A_1 et A_2, dénoté $A_1 // A_2$, est l'automate temporisé ($\Sigma_1 \cup \Sigma_2$, $L_1 \times L_2$, $I_1 \times I_2$, $C_1 \cup C_2$, $\rightarrow$, Label, inv) ou :*

- $\rightarrow$ est la plus petite relation de transition définie par les règles suivantes :
  - Pour $a \in \Sigma_1 \cap \Sigma_2$,

$$\frac{l_1 \xrightarrow{a_1,\alpha_1,C_1}{}_1 l'_1, \quad l_2 \xrightarrow{a_1,\alpha_2,C_2}{}_2 l'_2}{(l_1,l_2) \xrightarrow{a_1,\alpha_1\wedge\alpha_2,C_1\cup C_2} (l'_1,l'_2)}$$

  - Pour $a \in \Sigma_1 \setminus \Sigma_2$: $\dfrac{l_1 \xrightarrow{a_1,\alpha_1,C_1}{}_1 l'_1}{(l_1,l_2) \xrightarrow{a_1,\alpha_1,C_1} (l'_1,l_2)}$

  - Pour $a \in \Sigma_2 \setminus \Sigma_1$: $\dfrac{l_2 \xrightarrow{a_2,\alpha_2,C_2}{}_2 l'_2}{(l_1,l_2) \xrightarrow{a_2,\alpha_2,C_2} (l_1,l'_2)}$

- $Label(l_1,l_2) = Label_1(l_1) \cup Label_2(l_2)$
  - $inv(l_1,l_2) = inv_1(l_1) \cap inv_2(l_2)$

5.7.4 La Logique Temporelle Temporelle (TCTL)

5.7.4.1 Syntaxe et sémantique de *TCTL*

Les formules de TCTL [AD90, ACD90] sont construites à partir des propositions et de contraintes sur les horloges au moyen d'opérateurs logiques, d'un opérateur d'initialisation d'Horloge noté z., et de deux opérateurs temporels notés AU et EU.

Intuitivement, un état *q* satisfait la formule $E[\varphi_1 U \varphi_2]$ s'il existe une séquence à partir de *q* pour laquelle la formule φ_1 est continuellement vraie jusqu'à un état qui satisfait φ_2.

Un état q satisfait $A[\varphi_1 U \varphi_2]$ si pour toute séquence à partir de q, φ_1 est continuellement vraie jusqu'à un état où φ_2 est vraie.

L'opérateur d'initialisation z. sert à introduire une horloge auxiliaire $z \in C$ qui permet de mesurer le temps écoulé à partir de l'état présent. Dans la formule z.φ, les occurrences de z dans φ sont liées par l'opérateur z. Lorsque la formule z.φ est interprétée sur un système temporisé T, nous exigeons que z ne soit pas une horloge de T, c'est-à-dire, si T est le modèle d'un graphe temporisé G dont l'ensemble d'horloges est $C' \subseteq C$, alors $z \notin C$.

Définition 5.55 *Les formules de TCTL son définies par la syntaxe suivante :*

$$\varphi ::= p \mid \alpha \mid \neg\varphi \mid \varphi \vee \varphi \mid z.\varphi \mid E[\varphi\ U\ \varphi] \mid A[\varphi\ U\ \varphi]$$

où p est une formule propositionnelle, $x,y,z \in C'$, $c \in Z$ et $\updownarrow \in \{<,\leq\}$.

Les formules de TCTL sont interprétées sur les états d'un système temporisé.

Les formules E[φ U φ] et A[φ U φ] sont définies exactement comme en CTL.

Définition 5.56 *Pour un système temporisé $T = (Q, \rightarrow, q_1)$, un ensemble de séquences $\Sigma \subseteq \Sigma_T$, un état $q \in Q$ et une formule φ de TCTL, la relation $s, \omega \models \varphi$ est définie par induction sur φ de la façon suivante :*

- $s, \omega \models p$ *ssi* $p \in Label(s)$
- $s, \omega \models \alpha$ *ssi* $v \cup \omega w \models \alpha$
- $s, \omega \models \neg\varphi$ *ssi* $\neg (s, \omega \models \varphi)$
- $s, \omega \models \varphi_1 \vee \varphi_2$ *ssi* $(s, \omega \models \varphi_1)$ *ou* $(s, \omega \models \varphi_2)0$
- $s, \omega \models z.\varphi$ *ssi* s, *reset* z *in* $\omega \models \varphi$
- $s, \omega \models E(\varphi_1\ U\ \varphi_2]$ *ssi* $\exists\sigma \in \mathbf{P}_M^{\infty}(s).\exists(i,d) \in Pos(\sigma).\ (\sigma(i,d),\ \omega+\Delta(\sigma,i) \models \varphi_2 \wedge (\forall(j,d')\ \updownarrow = (i,d).\ \sigma(j,d'),\ \omega+\Delta(\sigma,j) \models \varphi_1 \vee \varphi_2))$
- $s, \omega \models A(\varphi_1\ U\ \varphi_2]$ *ssi* $\forall\sigma \in \mathbf{P}_M^{\infty}(s).\exists(i,d) \in Pos(\sigma).\ (\sigma(i,d),\ \omega+\Delta(\sigma,i) \models \varphi_2 \wedge (\forall(j,d')\ \updownarrow = (i,d).\ \sigma(j,d'),\ \omega+\Delta(\sigma,j) \models \varphi_1 \vee \varphi_2))$
- $(i,d) \updownarrow (i',d')$ *ssi* $(i<j)$ *ou* $((i=j)$ *et* $(d<d'))$.

Avec s : une location, ω : la valuation d'une formule horloge, $\mathbf{P}_M^{\infty}$ ensemble des chemins de sommet s initial, Pos(σ) position dans σ le l'exécution, Δ(σ,i) : temps écoulé depuis la dernière fois.

Exemple 5.57 AG [send(m) ➔ $AF_{<5}$ receive(m)]

La réception du message m se fera juste avant 5 unités de temps après son envoie.

5.7.4.2 Opérateurs dérivés

$\mathbf{A}[\varphi 1\ U_{\leq 7}\ \varphi 2] \equiv z \text{ in } \mathbf{A}[(\varphi 1 \wedge z \leq 7)\ U\ \varphi 2]$

EF$<5\ \varphi \equiv$ z in **EF**$[\varphi \wedge z < 5]$.

L'ensemble caractéristique d'une formule φ relativement à l'ensemble de séquences $\sum$, noté $[|\varphi|]_{\sum}$, est :

$$[|\varphi|]_{\sum} = \{q \in Q |\ q|=_{\sum}\varphi\}$$

i.e., l'ensemble des états qui satisfont φ relativement à $\sum$.

Dans ce qui suit il est nécessaire d'après la définition de la relation de satisfaction pour les formules $\varphi_1 \exists u \varphi_2$ et $\varphi_1 \forall u \varphi_2$, que la formule $\varphi_1 \vee \varphi_2$ soit vérifiée le long de la séquence, au lieu d'exiger que seulement φ_1 soit vérifiée comme dans le cas de CTL pour éviter des anomalies dues à la densité du temps.

Exemple 5.58 Considérons la formule $z.((z < 2 \vee p_1) \exists u(z = 5 \wedge p_2))$.

En appliquant la définition ci-dessus, q satisfait la formule s'il existe une séquence $\sigma \in \sum(q[z := 0])$ telle que :

Il existe $i \geq 0$ et $(q',i) \in \chi(\sigma)$, tels que $q'(z) = 5$ et $q'(p_2) =$ true, $q'(z)$ est le temps t' écoulé depuis le début de la séquence. Par conséquent, la proposition p_2 est satisfaite exactement 5 unités de temps après le début de la séquence.

De plus, tout état (q",j) qui procède à (q',i), satisfait $q''(z) < 2$ ou $q''(p1) =$ true ou $q''(z) = 5$ et $q''(p2) =$ true, $q''(z)$ est le temps t'' écoulé depuis le début de la séquence. De plus $t'' \leq t'$, par conséquent, la proposition p_1 est vraie pour $2 \leq t'' \leq 5$. En effet, puisque les transitions temporelles ne changent pas les valeurs des propositions, la proposition p_1 doit être satisfaite aussi à l'instant 5.

Donc la formule exprime la propriété suivante : il existe une séquence où la proposition p_2 est vraie à l'instant 5 et la proposition p_1 est continuellement satisfaite dans l'intervalle de temps [2,5].

Les opérateurs temporels additionnels comme $\exists\Diamond, \exists\Box, \forall\Diamond$ et $\forall\Box$ sont définis comme des abréviations à partir des opérateurs temporels $\exists u$ et $\forall u$ de la façon suivante :

- $\exists\Diamond\varphi$ pour true$\exists u\varphi$
- $\forall\Diamond\varphi$ pour true$\forall u\varphi$
- $\exists\Box\varphi$ pour $\neg\forall\Diamond\neg\varphi$
- $\forall\Diamond\varphi$ pour $\neg\exists\Diamond\neg\varphi$

5.8 Conclusion

Nous venons de traiter dans ce chapitre deux points importants qui se résument en des modèles de vérification des systèmes réactifs, les premiers indépendamment de l'aspect implicite du temps et les deuxièmes sont les systèmes réactifs temps-réel. Dans la première catégorie de model-checking, nous avons abordé les modèles linéaires basés sur une procédure simple qui procède tout d'abord par simplification syntaxique sur les termes puis par réduction axiomatique sur les équations. La deuxième catégorie définie le système comme une structure de Kripke, la transforme (cette dernière) en automate généralisé qui à sont tour est transformé en automate de Buchi. La vérification s'effectue sur le graphe de la composition de l'implémentation avec celui de la spécification.

La deuxième catégorie procédait sur les modèles arborescents en utilisant la logique CTL sur les modèles de Kripke.

Enfin, nous avons présenté une technique de pointe pour représenter les automates temporisés et les graphes de transitions étiquetées par des contraintes sur les arcs et des invariants sur les états. Le modèle de vérification étant bien entendu TCTL.

CHAPITRE 6

La Logique de Réécriture, Maude et Full-maude

6.1 Introduction

Nous abordons dans cette section un modèle de calcul basé sur la réécriture comme modèle de base des machines séquentielles et parallèles. A l'origine, la réécriture est apparue et utilisée comme une technique de preuve en logique équationnelle et aussi comme un outil d'exécution de spécifications. Durant le début des années quatre-vingts, plusieurs idées sur l'implémentation de la réécriture ont apparu dans plusieurs écrits, essentiellement dans [O'Do77, O'Do85] qui est à la base de plusieurs implémentations tels OBJ [GW88]. Récemment, plusieurs systèmes basés sur la logique de réécriture ont pris naissance, constituant des outils très puissants utilisés pour les calculs de spécification mais aussi de vérification. Maude [CDS+99], [CDS+00a], ELAN [BKK96] et Cafe-Obj [FD98] sont les trois systèmes les plus performants, certes ils sont différents dans leur style de programmation, mais ils se basent sur le même principe, c'est-à-dire qu'ils utilisent la logique de réécriture en tant que model de calcul plus des stratégies. Naturellement d'autres aspects théoriques sont venus se greffer sur le noyau (rewrites core), telles que la réflexion, l'orienté objet, la réécriture conditionnelle, Membership equational etc. Nous verrons, tous ces éléments un peu plus tard quand on présentera le système MAUDE.

Cette section présente un aperçu général de la logique de réécriture ([Mes00, Mes98, Mes92, MM96, CDE+00, CDE+99, CDM+00, GM88] pour une introduction à la logique de réécriture et ces applications), le langage Maude [CDE$^+$99, CDE$^+$00a, CDE+00b], le langage Full-Maude et son système [CDE$^+$99, Dur99]. Nous supposons que le lecteur est familier avec les concepts de base des spécifications algébriques dont une partie sera introduite juste après. Pour mieux appréhender cette notion, le lecteur est invité à consulter l'ouvrage de Wirsing [Wir90], les travaux de Kosiuczenko [SK00, KW00, KW97a, KW97b, OKW96, KW96] et l'excellent travail de Harman [FH98, HT96, Har00]. Nous attirons aussi l'attention du lecteur pour bien comprendre cette section, certaines notions sur la théorie de réécriture des termes (term rewriting) sont nécessaires. Un travail colossal dans ce sens à été réalisé par Klop dans [Klo92] sans oublier les travaux de Huet et Dershowitz [DJ87, DJ90, HO80,HH80, Hue80] sur la réécriture. Aussi pour que les lecteurs se sentent à l'aise en lisant cette section, quelques éléments fondamentaux seront introduits, essentiellement la réécriture, les systèmes algébriques, la logique de réécriture et naturellement certaines définitions et théorèmes très importants seront présentés.

6.2 La réécriture

6.2.1 Quelques concepts fondamentaux

Le concept des systèmes de réécriture TRS (Termes Rewriting System) est en général, paradigmatique, surtout quand il s'agit des problèmes liés aux procédures de calcul (computationnal). Depuis pratiquement un siècle le TRS le plus connu fût le λ-calcul. Il a été très utilisé dans la logique mathématique. Un peu plus tard, les mêmes modèles ont figuré dans les travaux de Plotkin [Plo77] pour décrire les sémantiques dénotationnelles dans les langages de programmation. Plus récemment, la logique combinatoire catégorielle émergea conduisant à une connexion entre la théorie des catégories et la spécification des étapes élémentaires dans les calculs des machines.

Les TRSs sont des systèmes attractifs du fait d'une représentation simple des syntaxes et des sémantiques. Cette simplicité facilite l'expression de la notion de *satisfaction* dans l'analyse mathématique ainsi que la conception des machines parallèles. Ils jouent un rôle très important dans l'analyse et l'implémentation des types de données abstraits pour mettre en place les moyens de vérification des propriétés de consistance, de calculabilité, de décidabilité ainsi que la démonstration de théorèmes.

Dans ce qui suit nous allons introduire quelques notions pour définir les TRS 's en tant qu'outils pour la représentation (spécification) des systèmes parallèles et concurrents. Tous ces éléments constituent une

approche importante pour mieux appréhender la logique de réécriture. Nous démontrons aussi, que la logique de réécriture est l'assemblage des systèmes de réécriture et des spécifications algébriques.

6.2.1.1 Définitions de base

Définition 6.1 *(Opérateurs) F est un ensemble d'opérateurs (symboles de fonctions) muni de la fonction arité : F → N qui associe à un symbole de fonction f d'arité du symbole f. F^n d ésigne le sous-ensemble de fonction d'arité n.*

Nous définissons les termes qui sont construits sur des opérateurs plus un ensemble de variables. Ayant un ensemble d'opérateurs F et un ensemble de variables X, nous pouvons définir des expressions construites sur des opérateurs définis.

Définition 6.2 *(Termes) Les termes sur F sont définis comme un ensemble Ter(F,X), par les règles :*

- *Chaque variable $x \in X$ est un terme de Ter(F,X),*
- *Si $t_1, \ldots, t_n$ sont des termes de Ter(F,X), respectivement, et si $f \in F^n$, alors $f(t_1, \ldots, t_n)$ est un terme de T(F,X).*

L'ensemble des variables d'un terme t est noté par *Var(t)*. L'ensemble de terme clos (termes sans variables) est noté T(F). Un terme peut être vu comme un arbre où les feuilles sont des constantes (opérateurs d'arité 0) ou des variables et les nœuds internes vus comme des opérateurs d'arité strictement positive. Afin de décrire la position d'un nœud dans cet arbre, nous définissons la notion d'occurrence ou de position.

Définition 6.3 *(Occurrence) Une occurrence ou position dans un terme t est une suite p d'entiers positifs décrivant le chemin de la racine du terme t ou sous terme à cette position d'entiers positifs décrivant le chemin de la racine du terme ou du sous terme à cette position noté $t_{|p}$.*

Définition 6.4 *(F-Algèbre) Une F-Algèbre **A** est un ensemble non vide de valeurs A et une interprétation τ de F est notée $(A, \tau(F))$, $\tau(F)$ est noté F_A et l'interprétation du symbole f est notée f_A.*

Définition 6.5 *(Identité) Une identité (égalité) est un couple (t, t') avec $t, t' \in ter(F, X)$, noté $t = t'$.*

Définition 6.6 *Soit un ensemble d'égalités de E et l'ensemble des termes Ter(F,X), la théorie équationnelle de E, noté TH(E) est l'ensemble d'égalités obtenues par application des règles d'inférence décrites dans le système.*

Définition 6.7 *(Algèbre-quotient des termes) Soit F, X un ensemble de variables et $\equiv_E$ la plus petite congruence contenant l'ensemble d'égalité E. Alors l'ensemble quotient des termes T(F,X)/E est l'ensemble $T_E(F,X) = \{<u>_E \mid u \in T(F,X)\}$ telle que $<u>_E$ étant la classe d'équivalence définie par $<u>_E = \{v \mid v \in T(F,X), v \equiv_E u\}$.*

Définition 6.8 *Une algèbre A est un modèle de l'axiome s = t, si pour toutes les interprétations θ des variables de s et t, on a $\theta(s) = \theta(t)$. Une algèbre A est un modèle de E si elle est un modèle de tous les axiomes de E. $A \models s = t$ si $s = t$ est valide dans A. L'ensemble des modèles de E est noté MOD(E).*

Définition 6.9 **(***Système Abstrait de réduction) Un système de réduction abstrait ARS (Abstract Réduction System) est une structure **A** = $<A, (\rightarrow_\alpha)_{\alpha \in I}>$ ou A est un ensemble (pour le moment quelconque) et $\rightarrow_\alpha$ est une séquence de relations binaires définies sur A, appelées réductions dites aussi relations de réécriture.*

Par la suite une réduction à une seule étape notée «$\rightarrow\alpha$» est remplacée par « → ».

Un ARS avec une seule réduction (règle) est dit système de *remplacement* ou bien système de *réduction*, si $(a, b) \in \rightarrow_\alpha$, on écrit $a \rightarrow_\alpha b$ et nous disons que b est one-step (α-)reduct de a.

Définition 6.10 *(Fermeture réflexo-transitive) La fermeture reflexo-transitive de $\rightarrow_\alpha$ s'écrit $a \twoheadrightarrow_\alpha b$ et est définie s' il existe une séquence de réduction finie (possiblement vide) $a \equiv a_o \twoheadrightarrow_\alpha a_1 \twoheadrightarrow_\alpha \ldots \twoheadrightarrow_\alpha a_n \equiv b$ (la relation $\equiv$ définie l'identité des éléments de A) et b est dit α-réduit de A.*

A partir de là, on défini la relation d'équivalence qui s'écrit $=_\alpha$ appelée aussi relation de convertibilité (convertibility relation). Il nous est aussi possible maintenant d'écrire certaines propriétés définies par:

- La fermeture réflexive de $\rightarrow_\alpha$ est notéé $\rightarrow\equiv_\alpha$
- La fermeture transitive de $\rightarrow_\alpha$,, est notée $\rightarrow_\alpha+$
- La relation inverse de $\rightarrow_\alpha$ est notée $\leftarrow_\alpha$
- L'union de $\rightarrow_\alpha \cup \rightarrow_\alpha$ est notée $\rightarrow_{\alpha\beta}$.
- La composition de $\rightarrow_\alpha$ o $\rightarrow_\beta$ o si il existe a,b, et c de A tel que $a \rightarrow_\alpha c \rightarrow_\beta b$.

Si α et β sont des relations de réduction sur A, on dit que α *commute faiblement* avec β si $\forall a,b,c \in A\ \exists d \in A\ (c \leftarrow_\beta a \rightarrow_\alpha b \Rightarrow c \rightrightarrows_\alpha d \leftrightarrows_\beta b)$.

La relation $\rightarrow$ est dite *confluente* ou de *Church-Rosser* (CR) si elle vérifie $\forall a,b,c \in A\ \exists d \in A\ (c \leftrightarrows a \rightrightarrows b \Rightarrow c \rightrightarrows d \leftrightarrows b)$.

Définition 6.11 *(Ordre partiel) Un ordre partiel (RPO) noté $\succ$ sur un ensemble S est une relation binaire et irreflexive sur S. L'ensemble S est dit partiellement ordonné. L'ordre $\succ$ est total si pour deux éléments distincts t_1, t_2 de S, $t_1 \succ t_2$ ou bien $t_2 \succ t_1$.*

Définition 6.12 (*Ordre lexicographique sur les mots) soit $\prec$ un ordre sur un ensemble E. L'ordre lexicographique $\prec_{lex}$ sur E* est défini $e_1 \ldots e_n \prec_{lex} l_1 \ldots l_m$ ssi $e_1 \prec l_1$ ou bien $e_1 = l_1$ et $e_2 \ldots e_n \prec l_2 \ldots l_m$, ou bien $n = 0$ et $m > 0$.*

Définitions 6.13 *Soit **A** = $<A, \rightarrow>$ un ARS. Nous disons que $a \in A$ est une forme normale s'il n'existe aucun $b \in A$ telle que $a \rightarrow b$.*

***A** (ou $\rightarrow$) est inductive si pour toute séquence de réduction (possible infinie) $a_0 \rightarrow a_1 \rightarrow \ldots$, il existe un $a \in A$ tel que $a_n \rightrightarrows a$ pour tout n.*

***A** (ou $\rightarrow$) est à branchement finit (FB) si pour tout $a \in A$ l'ensemble des réduction one-step de a, $\{ b \in A \mid a \rightarrow b \}$, est fini,*

- *Si la relation inverse de $\rightarrow$ est FB, on dit que **A** (ou $\rightarrow$) est FB^{-1}.*
- *Un ARS qui est confluent et qui se termine (CR et SN) est dit complet ou bien canonique.*

6.3 Notions de base des systèmes de réécriture

6.3.1 Syntaxe des systèmes de réécriture

Définition 6.14 *On défini un système de réécriture par un TRS, c'est-à-dire la paire (Σ, R) formée d'un alphabet (signature) Σ est d'un ensemble de règles de réduction (règles de réécriture) R.. Cette signature est définie par :*

- *Un ensemble infini dénombrable de variable x_1, x_2, ... aussi dénotés par x, y, z, x', y'...*
- *Un ensemble de symboles de fonctions non vide (symbole d'opération) dénotés F, G,… , ou chaque symbole d'opération est équipé d'une arité (nombre naturel) qui peut être 1-aire, binaire, n-aire, mais aussi 0-aire (symbole de constante).*

Définition 6.15 *On défini aussi l'ensemble des termes (ou expressions) de Σ par Ter(Σ) qui est défini inductivement par:*

- *$x, y, z \ldots \in Ter(\Sigma)$*
- *Si F est un symbole d'opérations est $t_1 \ldots, t_n \in Ter(E)$ ($n > 0$) alors $F(t_1, \ldots, t_n) \in Ter(\Sigma)$ et t_i ($i = 1, \ldots, n$) sont les arguments du dernier terme.*

Définitions 6.16

- *Termes clos : Les termes qui ne contiennent pas de variables sont dits termes clos (Ground Terms) et $Ter_G(E)$ est dit l'ensemble des termes clos (Ground Termes).*
- *Termes linéaires : Ce sont les termes qui contiennent une seule variable.*
- *Contexte : Se sont les termes qui contiennent une occurrence d'un symbole spécial [] dénotant une place vide. Un contexte est généralement dénoté par C[]. Si $t \in Ter(\Sigma)$ est t est substitué dans [], le résultat $C[t] \in Ter(\Sigma)$; t est dit sous terme de C[t] (ou symbole dans le terme).*

Exemple 6.17 Soit $\Sigma = \{A,M,S,O\}$, les arités sont 2,2,1,0 respectivement. Alors A(M(x,y),y) est un terme non-linéaire, A(M(x,y),z) est un terme linéaire, A(M(S(0),0),S(0)) est un terme clos, A(M([],0),S(0)) est un contexte, S(0) est un sous-terme de A(M(S(0),0),S(0)) possédant deux occurrences : A(M(S(0),0),S(0)).

Définition 6.18 *(Substitution) Une substitution est une application de $Ter(\Sigma)$ vers $Ter(\Sigma)$ satisfaisant: s. On écrit alors t^σ au lieu de $\sigma(t)$.*

Définition 6.19 *(Règle de réécriture) C'est la paire (t, s) de termes $\in Ter(\Sigma)$. Elle peut aussi être écrite comme $t \rightarrow s$ et recevra si nécessaire un nom r: $t \rightarrow s$.*

Néanmoins deux conditions doivent être imposées:

1. La partie gauche ne doit pas être une variable; on dit partie gauche ou bien LHS (left-hand-side).

2. Les variables dans la partie droite RHS (Right-hand-ride) doivent être contenues dans t.

Une règle de réduction r: t $\rightarrow$ s détermine un ensemble de réécriture $t^\sigma \rightarrow_r s^\sigma$ pour toutes les substitutions σ. Le coté gauche (LHS) t^σ est appelé le *Redex* (Reducible expression) est plus précisément r-redex. Un redex t^σ peut être remplacé par son contractum s^σ à l'intérieur d'un contexte C[]. Et cette opération est dite étape de réduction ou bien réécriture en une seule étape (one step rewriting) $C[t^\sigma] \rightarrow_r C[s^\sigma]$.

Théorème 6.20 *Soient R1 et R2 deux TRS. $R1 \oplus R2$ (ou $\oplus$ est la relation de disjonction) est confluent si et seulement si R1 et R2 sont confluents.*

Termes clos (GROUNDS) (t $\rightarrow$ s ou t et s sont tous deux des termes clos) + confluence implique la décidabilité.

Théorème 6.21 (Toyama, Klop et Barendrest)[TKB89]. *Soient R1 et R2 deux TRSs linéaires à gauche, alors $R1 \oplus R2$ est complet si et seulement si RI et R2 le soient tous les deux.*

Définitions 6.22

- *Une règle de réécriture est une règle `collapsing' (réduction) si s est une variable.*
- *Une règle de réécriture $t \rightarrow s$ est une règle dupliquée si certaines variables ont plus d'occurrences dans s que dans t.*

6.4 Spécification équationnelle (syntaxe-sémantique)

6.4.1 Notions Générales

Définition 6.23 *Une spécification équationnelle est juste un TRS sans orientation.*

Plus précisément une spécification équationnelle est la paire (Σ, E) ou Σ est une signature (alphabet) et E est l'ensemble des équations de la forme $s = t$ entre les termes $s, t \in Ter(E)$.

Si une équation $s = t$ est *dérivable* à partir des équations dans E, on écrit:

$$(\Sigma, E) \vdash s = t \quad ou \quad s =_E t.$$

Formellement la *dérivabilité* est définie par application des règles d'inférence du système suivant (Table 6.1).

- $(\Sigma, E) \vdash s = t$ si $s = t \in E$
- $\dfrac{(\Sigma,E) \vdash s=t}{(\Sigma,E) \vdash s^{\sigma}=t^{\sigma}}$ pour toute substitution σ
- $\dfrac{(\Sigma,E) \vdash s=t_1,\ldots,(\Sigma,E) \vdash s_n=t_n}{(\Sigma,E) \vdash F(s_1,\ldots,s_n)=F(t_1,\ldots,t_n)}$ pour tout n-aire $F \in \Sigma$
- $(\Sigma,E) \vdash t=t$
- $\dfrac{(\Sigma,E) \vdash t_1=t_2 \ , \ (\Sigma,E) \vdash t_2=t_3}{(\Sigma,E) \vdash t_1=t_3}$
- $\dfrac{(\Sigma,E) \vdash s=t}{(\Sigma,E) \vdash t=s}$

Table 6.1 : Système d'Inférence

Définition 6.24 *Si Σ est une signature, une Σ-algèbre A est un ensemble A munit d'une fonction $F^A : A^n \rightarrow A$ pour tout symbole de fonction $F \in \Sigma$, (si F est dépourvue d'argument, alors F est une constante, donc $F^A \in A$.). Les variables contenues dans les termes s et t sont universellement quantifiables et si s et t appartiennent à Ter(Σ) et s'ils sont valides dans A, on écrit : $A \models s = t$ pour dire que $s = t$ est valide dans A.*

Définition 6.25 *A est dit model d'un ensemble E d'équations, si toute spécification équationnelle (Σ,E) qu'on notera par Alg(Σ,E) est la classe de toutes les Σ-algèbres A telles que $A \models E$, au lieu d'écrire $\forall A \in Alg(\Sigma,E)$, $A \models F$, ou F est un ensemble d'équations définit entre les Σ-termes on écrira $(\Sigma,E) \models F$.*

Le théorème suivant dû à Birkoff [Bir35] est un théorème très connu qui énonce la complétude d'une spécification équationnelle.

Théorème 6.26 *Soit (Σ,E) une spécification équationnelle . Alors pour tous s, $t \in Ter(\Sigma)$:*

$$(\Sigma, E) \vdash s = t \Leftrightarrow (\Sigma,E) \models s = t .$$

Le fait d'avoir un système d'équations qui soit complet, donc qui soit SN + CR est très important dans la théorie équationnelle, ceci témoigne bel est bien de la finitude des transformations que subiront les équations. En vérité, un algorithme simple peut être trouvé pour vérifier cela, il sera construit autour des points suivants :

- *Réduire* s et t en vue d'obtenir leurs formes normales s' et t'
- *Comparer* s' et t' : $s =_R t$ si et seulement si $s' \equiv t'$.

6.4.2 La déduction dans les Systèmes de réécriture

La réécriture est à l'origine une méthode de preuve en logique équationnelle. Dans cette logique les axiomes sont des équations et les théorèmes sont déduits des axiomes par une règle d'inférence très simple qui est le remplacement d'égaux par des égaux. Un exemple simple peut être introduit,

c'est celui des fonctions sur les entiers s'exprimant alors sous forme d'équations entre termes (avec variables) :

$$x + 0 = x \qquad \text{(I)}$$

$$x + s(y) = s(x + y) \qquad \text{(II)}$$

qui sont deux axiomes équationnels définissant l'addition sur les entiers. Un théorème élémentaire de ce système d'axiomes est le suivant :

$$s(x) + 0 = x + s(0).$$

En effet, $s(x) + 0 = s(0)$ d'après l'axiome (I) utilisé de gauche à droite, $s(x) = s(x + 0)$ toujours d'après l'axiome (I) utilisé de droite à gauche cette fois-ci, enfin $s(x + 0) = x + s(0)$ d'après l'axiome (II) utilisé de droite à gauche.

Ce type de théorème est appelé *théorème équationnel*. Malheusement l'obtention de tels théorèmes n'est pas toujours aisée, il va falloir chaque fois, choisir les bons axiomes et aussi de bien les ordonner.

Le remplacement des égaux s'appelle réécriture, et la méthode qui consiste à choisir un sens d'utilisation des axiomes, de gauche à droite ou de droite à gauche étant la base de la réécriture. Une fois le choix fixé, l'axiome en question devient une équation orientée et sera appelée *règle de réécriture*. Un terme qui ne peut être réécrit est dit terme *irréductible* ou en *forme normale*. La réécriture en elle-même est un processus *indéterministe* et ce qui fait qu'un terme pourra donc avoir plusieurs formes normales. En général, la propriété la plus intéressante étant la confluence qui stipule que le calcul du résultat ne dépend aucunement du choix de la règle de départ, c'est-à-dire que n'importe quelle règle nous conduit directement vers la même forme normale. En général d'après le théorème de Church-Rosser, un système de réécriture qui possède la propriété de terminaison (qui est en générale indécidable) et celle de confluence est dite convergente. Il est donc clair que dans ce cas on est ramené à remplacer l'idée de preuve par celle de calcul des formes normales des termes d'une équation.

Comme nous venons de noter que la confluence est une propriété indispensable, même si le système ne l'est pas, alors on essaie en utilisant une procédure très connue appelée la procédure de *complétude de Knuth et Bendix* [KB70], qui ajoute les paires critiques non convergentes au système de réécriture initial en les orientant de manière à conserver la propriété de terminaison.

6.5 La logique de réécriture

6.5.1 Introduction

La logique de réécriture est définie comme une logique de changement dans laquelle sa partie dynamique est spécifiée par des règles de réécriture conditionnées et étiquetées [Mes00, Mes98, Mes92, MM96]. Par contre la partie statique est spécifiée par des équations. Une *signature* dans la logique de réécriture est une théorie équationnelle (Σ,E), ou Σ est une signature équationnelle (alphabet) et E un ensemble de Σ-équations. La réécriture opère sur les classes d'équivalence de termes modulo E.

Soit une signature (Σ,E), les séquents en logique de réécriture sont de la forme

$$[t]_E \rightarrow [t']_E ,$$

ou t et t' sont des Σ-termes qui peuvent contenir des variables, et $[t]_E$ définie une classe d'équivalence du terme t modulo le système d'équations E. Une théorie de réécriture $\mathcal{R}$ est un quadruplet $\mathcal{R} = (\Sigma,E,L,R)$ *ou* Σ une *signature* (ranked alphabet) de symboles de fonctions, E est l'ensemble des Σ*-équations*, *L est* l'ensemble des *étiquettes et R étant l'ensemble des paires* $R \subseteq L \times T_{\Sigma,E}(X)^2$ ou $X = \{x_1, \ldots, x_n, \ldots\}$ est un ensemble dénombrable, possible infinie de

variables. Les éléments de *R* sont appelés les *règles de réécriture*. Une règle (l, ([t],[t'])) est alors considérée comme un séquent étiqueté qui prend la forme l : [t] → [t'].

Pour indiquer que $\{x_1, \ldots, x_n\}$ est l'ensemble de variables composant les termes t ou t ', en écrit alors

$$l : [t(x_1, \ldots, x_n)] \rightarrow [t'(x_1, \ldots, x_n)],$$

ou de manière abrégée on utilise la notation $l : [t(\bar{x})] \rightarrow [t'(\bar{x})]$.

Soit une théorie de réécriture $\mathcal{R}$ on dit que [t]→[t'] est un (concurrent) $\mathcal{R}$ – rewrite, et on écrit $\mathcal{R}$|- [t]→ [t'] si et seulement si [t] → [t'] peut être obtenu par application des règles de déduction de la figure 6.2 (nous supposerons que tous les termes sont bien-formés et que $t(w_i/x_i)$ dénote la substitution simultanée de w_i pour x_i dans le terme t):

1. **Réflexivité.** Pour chaque $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t]\rightarrow[t']}$

2. **Congruence.** Pour chaque $f \in \Sigma_n$, $n \in \mathbf{N}$,
$$\frac{[t]\rightarrow[t'] \cdots [t_n]\rightarrow[t'_n]}{[f(t_1,\cdots,t_n)]\rightarrow[f(t_1,\cdots,t_n)]}$$

3. **Remplacement inconditionnel.** Pour chaque règle $l : [t(x_1,\ldots, x_n)] \rightarrow [t'(x_1, \ldots, x_n)]$ dans R,
$$\frac{[w_1]\rightarrow[w'_1] \cdots [w_n]\rightarrow[w'_n]}{[t(\bar{w}/\bar{x})]\rightarrow[t'(\bar{w}/\bar{x})]}$$

4. **Transitivité**
$$\frac{[t_1]\rightarrow[t_2] \quad [t_2]\rightarrow[t_3]}{[t_1]\rightarrow[t_3]}$$

Figure 6.2 : Système de règles de Réécriture de Base

La logique équationnelle (modulo un ensemble d'axiome E) peut être obtenue à partir de la logique de réécriture en rajoutant la règle suivante,

5. Symétrie

Nous pouvons remarquer que si l'on rajoute cette nouvelle règle au sous-système des règles 1-4, les séquents dérivables dans la logique équationnelle deviennent toujours bi-directionnels, dans ce cas précis nous pouvons adopter la notation [t] ↔ [t'].

6. Substitution

$$\frac{[t(x_{i1},\cdots,x_{in})] \rightarrow [t'(x_{i1},\cdots x_{in})] \quad [w_1] \rightarrow [w'_1]\cdots[w_n] \rightarrow [w'_n]}{[t(w_1/x_{i1},\cdots,w_n/x_n)] \rightarrow [t'(w'_1/x_{i1},\cdots,w'_n/x_{in})]}$$

Définition 6.27. *Etant donné une théorie $\mathcal{R} = (\Sigma,E,L,R)$, un (Σ,E)-séquent* [t] → [t'] *est dit :*

- *Un 0-step concurrent R-rewrite si et seulement s'il peut être dérivé de R par application (finie) des règles 1 et 2 des règles de déduction*
- *Un one-step rewrite si et seulement s'il peut être dérivé (déduit) de R par application des règles (1) - (3) un nombre fini de fois, avec au moins une application de la règle (3). Si la règle (3) est appliquée une seule fois alors le séquent est dit " one-step séquentiel rewrite".*

- *Un "concurrent R-rewrite" (ou juste rewrite) si et seulement s'il peut être dérivé de R par application finie des règles 1-4.*

Lemme 6.28 *Pour chaque R-rewrite concurrent* [t] → [t'], *ou bien* [t] = [t'] *ou il existe un nombre n∈ N est une chaîne de réécriture de "one-step"*

$$[t_1] \rightarrow [t_2] \rightarrow \ldots \rightarrow [t_n] \rightarrow [t']$$

Nous appelons une chaîne une "*step séquence*" pour [t] → [t']. Par conséquent nous pouvons choisir toutes les étapes pour quelles soient séquentielles, dans ce cas on appellera une telle séquence, séquence d'*entrelacement* (*interleaving ou firing séquence*).

Les définitions de terminaison, de forme normale, de Church-Rosser (confluence) sont exactement similaires à celles déjà définies dans les systèmes de réécriture.

Théorème 6.29 *Si R est un système de règles de réécriture qui vérifie la confluence, alors une équation [t] → [t'] est prouvable à partir de R par déduction équatorienne, c'est-à-dire par application des règles 1-4 et la règle 6, si et seulement s'il existe un terme [t"] et les règles de réécriture [t] → [t"], [t'] → [t"]. Et si en plus R possède la propriété de terminaison alors tout terme [t] possède une forme normale unique, appelée forme canonique et dénotée* $can_R[t]$*; sous ces conditions, une équation [t] → [t'] est prouvable à partir de R par déduction équationnelle si* $can_R[t] = can_R[t']$.

Exemple 6.30 Considérons la théorie de réécriture R = (∑,R) avec ∑ = (F, E, X)

$F_0 = \{zero\}$, $F_1 = \{s\}$, $F_2 = \{plus\}$; $F = F_0 \cup F_1 \cup F_2$;

E = {plus(x,y) = plus(y,x)}, X = {x,y}, l = $\{l_0, l_1\}$;

Soit R = $\begin{cases} l_0(x): plus(zero,x) \rightarrow x \\ l_1(x,y): plus(s(x),y) \rightarrow s(plus(x,y)) \end{cases}$

Cette théorie de réécriture implique le séquent :

$$\langle\langle(s(zero))\rangle\rangle_E : \langle plus(zero, s(zero))\rangle\rangle_E \rightarrow \langle\langle(s(zero))\rangle\rangle_E$$

qui explique une application de la règle l_0 en appliquant les règles de déduction.

Une preuve de la même formule est donnée par le séquent :

$$l_1(<zero>_E, <zero>_E); s(l_0(<zero>_E : <plus(zero, s(zero))>_E \rightarrow <s(plus(zero), zero)>_E \ ,$$

les deux termes de preuve :

$$<l_0(s<zero>)>_E : \quad \text{et} \quad <l_1(<zero>)>_E, <zero>_E); s(l_0(<zero>_E)):$$

sont deux preuves différentes de $<plus(zero, s(zero))>_E \rightarrow <s(plus(zero), zero)>_E$

6.5.2 Expression de la Logique de Réécriture dans La Théorie des Catégories

La logique de réécriture est une logique utilisée pour modéliser les systèmes concurrents. Le raisonnement dynamique d'une telle logique utilise la notion d'état et de transitions. La signature de la théorie de réécriture décrit une structure particulière pour les états du système (e.g, les multisets, arbres binaires etc.). Les règles de réécriture dans cette théorie décrivent quelles transitions élémentaires peuvent être distribuées compte tenu de cette structure. Toute cette dynamique se traduit par le fait que l'étape de réécriture est une transition locale et parallèle dans un système concurrent. Alternativement à cela, nous pouvons considérer les règles de la logique de réécriture comme des meta-règles pour une bonne déduction dans le système logique. Logiquement, chaque étape de réécriture (rewriting step) définie un comportement logique dans le système formel.

Un model initial d'une théorie de réécriture est une catégorie $\mathcal{TR}$ ou les objets sont les classes d'équivalences [t] des termes clos modulo les équations dans E, et ou les flèches

(transitions) sont les preuves $r : [t] \rightarrow [t']$ dans la logique de réécriture. Les règles de déduction de la logique de réécriture définies dans la théorie des catégories sont comme suit :

1. Identités. Pour chaque $[t] \in T_{\Sigma,E}(X)$,

$$\frac{}{[t] : [t] \rightarrow [t']}$$

2. Σ-structure. Pour chaque $f \in \Sigma_n$, , $n \in N$,

$$\frac{\alpha_1 : [t] \rightarrow [t'] \cdots \alpha_n : [t_n] \rightarrow [t'_n]}{f(\alpha_1,\ldots,\alpha_n) : [f(t_1,\cdots,t_n)] \rightarrow [f(t_1,\cdots,t_n)]}$$

3. Remplacement. Pour chaque règle de réécriture

$$r : [t(\overline{x})] \rightarrow [t'(\overline{x})] \quad if \quad [u_1(\overline{x})] \rightarrow [v_1(\overline{x})] \wedge \ldots \wedge [u_k(\overline{x})] \rightarrow [v_k(\overline{x})]$$

$$\frac{\alpha_1 : [w_1] \rightarrow [w'_1] \quad \ldots \quad \alpha_n : [w_n] \rightarrow [w'_n] \qquad \beta_1 : [u_1(\overline{w}/\overline{x})] \rightarrow [v_1(\overline{w}/\overline{x})] \quad \ldots \quad \beta_k : [u_k(\overline{w}/\overline{x})] \rightarrow [v_k(\overline{w}/\overline{x})]}{r(\overline{\alpha}^n, \overline{\beta}^k) : [t(\overline{w}/\overline{x})] \rightarrow [t'(\overline{w}'/\overline{x})]}$$

4. Composition.

$$\frac{\alpha : [t_1] \rightarrow [t_2] \quad \beta : [t_2] \rightarrow [t_3]}{\alpha;\beta : [t_1] \rightarrow [t_3]}$$

5. Inversion.

$$\frac{\alpha : [t_1] \rightarrow [t_2]}{\alpha^{-1} : [t_2] \rightarrow [t_1]}$$

Les preuves sont égales modulo la notion d'équivalence qui en terme d'exécution correspond au vrai parallélisme.

1. Catégorie

(a) Associativité. Pour tous α,β,γ, $(\alpha;\beta);\gamma = \alpha;(\beta;\gamma)$.

(b) Identités. Pour chaque $\alpha : [t] \rightarrow [t']$, $\alpha;[t'] = \alpha$ et $[t];\alpha = \alpha$.

2. Fonctorialité de la structure Σ-algébrique. Pour chaque $f \in \Sigma_n$,

(a) Préservation de la composition. Pour tous les $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$,

$f(\alpha_1;\beta_1, \ldots, \alpha_n;\beta_n) = f(\alpha_1, \ldots, \alpha_n);(\beta_1, \ldots, \beta_n)$

(b) Préservation des identités. $f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, tn)]$.

1. **Axiomes dans E**. Pour $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ un axiome définie dans E pour tous les $t(\alpha_1, \ldots, \alpha_n) = t'(\alpha_1, \ldots, \alpha_n)$.

2. **Echange**. Pour chaque règle $r : [t(x_1, \ldots, xn)] \rightarrow [t'(x_1, \ldots, x_n)]$ dans *R*,

$$\frac{\alpha_1 : [w_1] \rightarrow [w_1'] \cdots \alpha_n : [w_n] \rightarrow [w'_n]}{[r(\overline{\alpha}) = r([\overline{w}]) \; ; \; t'(\overline{\alpha}) = t(\overline{\alpha}) \; ; \; r([\overline{w}'])}$$

6.6 Différents Types d'Algèbres Engendrées par la Réécriture

6.6.1 Many Sorted Algebra

Définition 6.31 *Soit S un ensemble quelconque, un ensemble S-sorted est une collection d'ensembles A_s indexés par les élements $s \in S$.*

Une fonction S-sorted f : A → B est une collection de fonctions indéxées par S telle que fs : As → Bs pour chaque $s \in S$. De façon similaire, une relation S-sorted R de A vers B est une collection de relations indexées par S et telle que Rs est définie de As vers Bs pour chaque $s \in S$.

Définition 6.32 *Une signature many sorted étant la paire (S, ∑), ou S est l'ensemble des sortes et ∑ est un ensemble (S* x S)-sorted de noms d'opérations. Si $\omega \in$ * et $s \in$ alors $\sum \omega,s$ est l'ensemble de noms d'opérations. L'écriture ∑[ ],s définit les symboles constantes de sort s.*

Définition 6.33 *Pour une signature many sorted ∑, une ∑-algèbre A sera définie par :*

- *Un ensemble S-sorted dénoté A, est appelé le carrier de l'algèbre*
- *Un élément $A\sigma \in As$ pour chaque $s \in S$ et $\sigma \in \sum[\ ],s$*
- *Pour chaque liste non vide $\omega \in S^*$, et pour chaque $s \in S$ et $\sigma \in \sum w,s$, une opération $A\sigma : A\omega \rightarrow As$, ou $\omega = s1 \ldots sn$ alors $A\omega = As1 \times \ldots \times Asn$.*

Soit A et B deux ∑-algèbres, on définit un ∑-homomorphisme h : A → B une fonction S-sorted A → B telle que :

- Soit un symbole de constante $\sigma \in \sum_{\omega,s}$, $h_s(A\sigma) = B\sigma$;
- Soit une liste non-vide $\omega = s1 \ldots sn$ et $\sigma \in \sum_{\omega,s}$ et a_i VA si pour i = 1,…,n, alors

$$H_s(A\sigma(a_1, \ldots, a_n)) = B\sigma(h_{s1}(a_1), \ldots, h_{sn}(a_n)).$$

Une algèbre pour une signature interprète les noms des sortes comme des ensembles et les noms d'opérateurs comme des opérations, cependant l'homomorphisme préserve la structure de l'algèbre par distribution sur les opérations de l'algèbre.

Nous écrivons $T_\sum$ pour l'algèbre des termes construit à partir des opérations dans ∑, et $T_\sum(X)$ pour la ∑-algèbre des termes contenant les variables d'un ensemble S-sorted.

Définition 6.34 *Une ∑-équation est le triplet (X,l,r) ou X étant un ensemble S-sorted et l et r sont des termes définis dans $T_\sum(X)$ de même sorte. Nous écrivons $(\forall X)\ l = r$.*

Une spécification est le triplet (S,∑,E) ou (S,∑) étant une signature et E est l'ensemble des ∑-équations.

Exemple 6.35 Considérons la spécification suivante écrite dans le langage Obj :

```
Obj  FLAG  is
    protecting BOOL .
    sort  Flag .
    op  new : → Flag .
    op up_ : Flag → Flag .
    op  up?_ : Flag → Flag .
    var  F : Flag .
    eq up ? new = false .
    eq up? Up F = true .
    eq up ? rev F = not(up ? F) .
endo
```

Ce programme spécifie une classe de cellules prenant des valeurs booléennes; leur état est représenté par la sorte Flag, et la valeur donnée dans un état quelconque par l'opération up? Cette valeur est mise à *true* par l'opération *up* et elle change de valeur par l'opération *rev*.

Définition 6.36 . Une ∑-algèbre A satisfait une ∑-équation (∀X) l = r si et seulement si $\bar{\theta}(l)=\bar{\theta}(r)$ pour tout θ : X → A. Nous écrivons A |= e pour indiquer que A satisfait l'équation e. Pour un ensemble E d'équations, nous dénotons par A |= E si et seulement si e∈E, et aussi nous dénotons par E|=e si et seulement A |= E implique A|=e pour toutes les ∑-algèbres.

Définition 6.37 Soit (∑,E), un (∑,E)-model est une ∑-algèbre A telle que A|=E. Une algèbre satisfait une équation donnée si et seulement si la partie droite et la partie gauche de l'équation sont égales sous toutes les interprétations des variables.

6.6.2 Order-Sorted Algebra.

L'algèbre order-sorted (OSA) [GM88, Gna92], est formée d'une structure d'ordre partiel sur les ensembles des sortes, la relation en question impose une restriction sur une algèbre A (S-sorted) telle que : *$s \leq s'$ est définie dans S alors $As \subseteq As'$* avec As qui dénote l'ensemble des éléments de sortes S. La motivation d'une telle algèbre étant de trouver des solutions à certains problèmes telles que les expressions insignifiantes. Nous pouvons citer à titre d'exemple certaines situations incongrues du type Top d'une pile, pile vide, division par zéro qui ne pouvaient pas être formalisés par les algèbres des années quatre-vingts. Donc, c'étaient des problèmes parmi tant d'autres à qui il fallait apporter des solutions.

Donc le concept théorique OSA est venu répondre à une foule de préoccupations en ADT. Ce formalisme qui n'est qu'une approche supporte ce qui suit:

- Plusieurs formes de polymorphismes et de surcharge (overloading)
- La définition des erreurs (détection et correction)
- L'héritage multiple, notion de sélecteurs (cas de constructeurs multiples)
- Retracts (inversion gauche de l'inclusion des opérations partielles qui rendent les sous-sortes définies équationnellement
- Des sémantiques opérationnelles qui supportent les équations (règles de réécriture)
- Des sémantiques de modèles théoriques rigoureux.

La signature de OSA consiste en une famille de sortes ordonnées par une relation d'ordre partiel d'héritage, plus une famille de symbole d'opérations. L'importance de l'OSA réside dans le fait que d'un coté elle permet de combiner différents types de sortes (naturels et rationnels, entiers et rationnels ect...) on appelle ceci l'overloading. Ce qui fait que cette situation est interprétée pour dire que les sous-sortes sont polymorphiques.

6.7 Le Langage Maude

6.7.1 Généralités

MAUDE [CDE+99, CDE+00a, CDE+00b] est un langage et un système en même temps, développé au CSL SRI International (Stanford University). L'élément de base du langage MAUDE étant la *théorie* qui est déclarée sous forme de module. Il existe naturellement plusieurs types de modules et le plus général étant le *module "system"*. Chaque module *system* se présente syntaxiquement sous la forme de ***mod*** $\mathcal{R}$ ***endm*** ou $\mathcal{R}$ étant une théorie de réécriture (le lecteur est invité à consulter le papier [Mes00] pour une description détaillée de la syntaxe des modules). Le système d'équations E dans la théorie équationnelle (∑,E) se présente comme l'union E = E' ∪ A, ou A étant l'ensemble d'axiomes

équationnels introduits comme des attributs de certains opérateurs, par exemple l'opérateur + peut être déclaré associatif et commutatif par le mot clé ***assoc*** et ***comm***, et ou E' est l'ensemble d'équations supposées respectant la propriété de *Church-Rosser* et celle de terminaison modulo les axiomes de A. De plus, la théorie $\mathcal{R}$ est supposée être *cohérente* (ground) [Vir94]. Le langage MAUDE supporte aussi la réécriture modulo différentes combinaisons des attributs de A. Les opérateurs dans MAUDE peuvent être déclarés avec certaines combinaisons de certaines propriétés telles que l'*associativité*, la *commutativité* et avec des attributs d'identités à gauche, à droite ou bien des deux cotés.

Une théorie équationnelle peut être vue comme une théorie de réécriture supportant l'ensemble L et R des labels et de règles vides. MAUDE possède un sous-langage dit *module fonctionnel* de la forme ***fmod*** (Σ,E) ***endfm,*** qui possède les même propriétés énoncées auparavant, c'est-à-dire la *confluence* et la *terminaison*. La logique équationnelle des modules fonctionnels et de la partie équation des modules système est une logique equationnelle de M*embership* [BJM00, Mes98], les expressions dans cette logique sont les équations t = t' et les prédicats *Membership* s'écrivent t : s avec s une sorte. La logique équationnelle *Membership* supporte aussi les sortes, les sous-sortes et les opérateurs de surcharge (overloaded).

MAUDE supporte aussi un troisième type de module, appelé *object-oriented* module qui spécifie les systèmes concurrents orienté-objets, sa syntaxe étant ***omod*** $\mathcal{R}$ ***endom.*** Au moment de l'exécution ces modules (orienté-objet) sont traduits en modules "*system"* ordinaires. MAUDE met en place tous les moyens pour représenter certains concepts de l'orienté-objets tels que l'objet lui-même, les messages, les classes, le polymorphisme et la notion d'héritage multiple.

Les modules dans MAUDE possèdent une sémantique initiale. Le module système tel qu'il est présenté (**mod** $\mathcal{R}$ **endm**) spécifie le model initial $\mathcal{TR}$ de la théorie de réécriture $\mathcal{R}$**.** De façon similaire un module fonctionnel ***fmod*** (Σ,E) ***endfm*** spécifie une algèbre initiale $T_{\Sigma,E}$ de la théorie équationnelle (Σ,E).

En plus de cela, MAUDE supporte un autre type de module appelé *module algebra* avec des modules : *parametrized modules*, *parameter theories views* et des *modules expressions*. Dans MAUDE cette algèbre de module (model algebra), appelée Full-Maude, est définie à l'intérieur même du langage par réflection ce qui lui donne une facilité d'extension et d'enrichissement. Full-Maude constitue un moyen pour prendre en charge la notion même de meta-programmation à travers les modules programmables META-LEVEL modules [CM96, CDEM00]. Cette notion de réflection met en place des fondements théoriques très puissants pour prendre en charge et mettre en application ce qu'on appelle les stratégies afin de guider de l'intérieur même l'exécution des théories de réécritures sous forme de règles de réécriture au niveau *metalevel*.

Donc**,** le système MAUDE et son langage spécifient et exécutent des spécifications basées sur la logique de réécriture. Une forme simple de MAUDE comme langage pour l'expression des systèmes dynamiques et concurrents spécifie une configuration (état) comme une collection et des messages utilisant l'opérateur **A.C.I** (Additive, Commutative et Identité) et l'élément neutre $\varnothing$.

L'expression d'une configuration c'est-à-dire de l'objet et des messages qui l'accompagnent (input & output) sont représentés par la règle suivante :

$$M_1, \ldots, M_n \langle O: C \mid a_1: : v_1 \ldots . a_n:: v_n \rangle Mn_1, \ldots, Mn_m$$

qui signifie que l'objet est un terme $\langle O: C \mid a_1: : v_1 \ldots . a_n:: v_n \rangle$ ou O étant le nom de l'objet, C la classe auquel appartient l'objet, a_i le nom de l'attribut numéro i et v_i la valeur de l'attribut i. Le système évolue selon des règles de réécriture conditionnelles qui prennent la forme suivante :

crl [r] : $M_1, \ldots, M_n \langle O_1:C_1 \mid atts_1 \rangle \ldots \langle O_m :C_m \mid atts_n \rangle$

=>

$\langle O_{i1}:C_{i1} \mid atts_{i1} \rangle \ldots \langle O_{ik}:C_{ik} \mid atts_{ik} \rangle \langle Q_1:D_1 \mid atts_1 \rangle \ldots \langle Q_p : D_p \mid atts_p \rangle M'_1, \ldots, M'_q$ ***if*** *Cond*.

Donc, cette règle exprime l'occurrence d'un événement de communication dans laquelle n messages et m objets distincts sont impliqués. Dans une phase donnée les messages M_1, ... , M_n sont consommés (disparaissent) et de nouveaux messages M'_1, ... , M'_q sont alors crées par le système et émis aux objets ou modules qui leur font référence. Les objets O_1,…O_m qui n'apparaissent que dans la partie gauche de la règle sont supprimés, par contre ceux de la partie droite Q_1 ... Q_p sont crées et ceux qui se trouvent sur les deux cotés changent uniquement leurs états locaux. L'option conditionnelle *Cond* restricte la règle à des situations particulières, donc c'est aussi un invariant contrôlant l'application de la règle. Quand plusieurs objets ou messages apparaissent sur la partie gauche de la règle, ils ont besoin de se synchroniser. Ce type de règles est dit synchrone. Par contre celles qui possèdent un seul objet et un seul message dans leur partie gauche sont des règles asynchrones.

Comme dans la philosophie orienté-objet, l'héritage dans une classe est directement supporté par la notion de structure de type order-sorted [MM96, GM88]. La déclaration d'une sous-classe s'exprime par C < C' qui est un cas particulier par lequel tous les attributs, les messages et les règles des superclasses caractérisent sa structure et son comportement.

A titre d'exemple le module fonctionnel suivant définit les déclarations des types de données des transactions bancaires.

fmod bank_operation is
protecting Int .
sort Int, Nat, Account, Msg .
subsort Nat < Int .
class Account .
att solde : Account → Nat .
msgs credit, debit : Account Nat → Msg .
msg transfer_from_ to _ :Nat Account Account → Msg
vars C1 C2 : Account, var mont : Nat .
endfm

Cet exemple, présente les opérations simples qu'une banque pourrait exécuter, définies par la balance, le débit et le crédit de deux comptes C1 et C2. Les informations contenues dans ces deux comptes pourraient être manipulé par deux types de règles— l'une synchrone et l'autre asynchrone ou (transfer Mont From C1 to C2) , credit(C1, Mont) and debit(C2, Mont) sont les messages de transfert et de manipulation des comptes. Ces deux règles sont équivalentes.

6.7.2 Expression des Communications Synchrones et Asynchrones

Les deux règles suivantes nous donnent une représentation approximative de la possibilité qu'offre la logique de réécriture supportée par le langage MAUDE, afin d'identifier et de mettre en place tous les mécanismes pour exprimer d'une manière générale le parallélisme et l'échange synchrone et asynchrone d'information entre des processus communicants et concurrents.

1er cas: *Règle synchrone:*

crl [rule1] (transfer Mont From C1 to C2) < C1: Account | solde: S1> < C2: Account | solde: S2> **=>** <C1: Account | solde: S1-Mont> < C2: Account | solde: S2 + Mont> ***if*** (S1 – Mont >= 0)

2ème cas: *Règle asynchrone:*

crl [rule2] credit(C1, Mont) < C1: Account | solde: S1 > => < C1: Account | solde: S1-Mont> ***if*** (S1– Mont >= 0)

rl [rule3] debit(C2, Mont) < C2: Account | solde: S2> => <C2: Account | solde: S2 + Mont> .

Dans la règle synchrone (cas 1), les objets C1 (< C1: Account | solde: S1 >) et C2 (< C2: Account | solde: S2>) interagissent en parallel (concurrently). Influencés par le message (transfer Mont From C1 to C2), ils échangent la valeur Mont et le système la retranche de solde: S1 et rajoute cette quantité à solde: S2. Le résultat de la règle (< C1: Account | solde: S1-Mont> < C2: Account | solde: S2 + Mont>) nous fournit une nouvelle configuration ou le message (transfer Mont From C1 to C2) est supprimé.

Nous pouvons imaginer à partir de la structure de ces opérations que pour les règles asynchrones, aucune interaction entre objets n'est possible puisqu'ils s'exécutent séquentiellement. Le premier objet perdra de fait la quantité Mont déduite de solde : S1, et rajoutée au solde : S2 du deuxième objet.

6.7.2.1 Règles de Fonctionnement Interne

Ce type de règles est écrit selon la forme suivante :

$$O_i \rightarrow O_i\, O_m \ldots O_n\, M_q \ldots M_r\ [T]\ .$$

Ces règles permettent d'exprimer les changements d'états, non pas à une excitation externe, mais plutôt à une logique interne qui n'est pas visible à ce niveau. Elle concerne donc un objet complexe actif, qui peut évoluer dans le temps suite à une activité interne. Cela permet de spécifier, entre autres, le comportement de certains objets qui obéissent à des lois internes plus ou moins complexes. Par l'intermédiaire de [T], ces lois peuvent être aléatoires quelconques, on introduit l'indéterminisme dans le comportement.

6.7.3 Caractéristique du Module META-LEVEL et de la Réflection

Clavel et Meseguer ont démontré que la logique de réécriture étant réflective [CM96, Dur99, CDE+00]. Ce qui fait, qu'il existe une théorie de réécriture $\mathcal{U}$ qui est universelle dans le sens ou il nous est possible de représenter toute théorie de réécriture (finitely) $\mathcal{R}$ (incluant $\mathcal{U}$ elle-même) et tous termes t, t' dans $\mathcal{R}$ comme des termes $\overline{\mathfrak{R}}$ et $\bar{t}$, $\bar{t}'$ dans $\mathcal{U}$, de façon à avoir l'équivalence suivante :

$$\mathfrak{R} \ \vdash\ t \ \rightarrow\ t' \ \Leftrightarrow\ \mathcal{U} \ \vdash\ \langle \overline{R}, \bar{t} \rangle \ \rightarrow\ \langle \overline{R}, \bar{t}' \rangle .$$

La théorie $\mathcal{U}$ peut être vue comme un interpréteur universel pour la logique de réécriture qui peut simuler les réécritures d'une théorie de réécriture $\mathcal{R}$. Cependant, il serait vraiment très coûteux de simuler les $\mathcal{R}$-rewrites d'une implémentation naïve de $\mathcal{U}$, puisque la simulation d'une seule étape de réécriture dans $\mathcal{R}$ peut engendrer plusieurs $\mathcal{U}$–étapes. La théorie universelle peut cependant être exécutée avec des fonctions (*descent functions*) qui simulent des opérations (rewrite step), sur les paramètres $\bar{t}$ *et* $\overline{R}$ par un chemin efficace de descente (going down) en un seul niveau de réflection et utilise le moteur de réécriture pour calculer dans $\mathcal{R}$ le résultat t' de l'application de la réécriture à t. Le terme résultant t' et alors transformé dans sa représentation en $\bar{t}'$.

Dans le langage Maude, les modules pré-définis META-LEVEL sont supposés modéliser une théorie universelle $\mathcal{U}$. Dans le module META-LEVEL, les termes sont réifiés comme des éléments d'un type de données Term, possédant la signature suivante:

```
subsort Qid < Term .
subsort Term < TermList .
op {_}_ : Qid  Qid  → Term .
```

op _[_] : Qid TermList → Term .

op _,_ : TermList TermList → TermList [assoc] .

op error* : → Term .

Dans cet exemple, "subsort Qid < Term . " déclare la sorte Qid comme une sous-sorte de Term, utilisée pour représenter les variables dans un terme par les identificateurs correspondants (quoted identifiers). Cependant si nous voulons représenter une variable, par exemple N, elle sera représentée par 'N. L'opérateur "{ _ }_" est utilisé pour représenter les constantes comme des paires, avec le premier argument la constante dans une forme quottée et le deuxième argument la sorte de la constante toujours dans une forme quottée. Par exemple, la constante zero de sorte ***Time*** est représentée par {'zero}'Time. L'opérateur _[_] correspond à une construction récursive de termes hors des sous-termes, ou la concaténation d'une liste est dénotée par " _,_". Par exemple, le terme f(a,b) (pour a et b deux constantes de sorte ***s***) est représenté dans la description META-LEVEL par 'f[{'a}'s, {'b}'s] et le terme {f(a,b)} (pour {_} un opérateur infixé) est représenté comme "{_'}[f[{'a}'s, {'b}'s]].

La syntaxe des modules *system* et *functional* est exactement la même que leur syntaxe originale. C'est-à-dire :

sorts Fmodule Module .

subsort Fmodule < Module .

op fmod_is_ _ _ _ _ _ _endfm : Qid ImportList SortDecl SubsortDeclSet OpDeclSet VarDeclSet MembAxSet EquationSet → Fmodule .

op mod_is_ _ _ _ _ _ _ _ endf : Qid ImportList SortDecl SubsortDeclSet OpDeclSet VarDeclSet MemAxSet EquationSet RuleSet → Module .

A titre d'exemple l'ensemble de regle dans un module seront représentés comme suit:

sort Rule RuleSet .

subsort Rule < RuleSet .

op rl[_] : _=>_. : Qid Term Term → Rule .

op crl[_] : _=>_if_=_. : Qid Term Term Term Term → Rule .

op none → RuleSet .

op _ _ : RuleSet RuleSet → RuleSet [assoc comm id: none] .

Le module META-LEVEL fournit des fonctionnalités sous forme de fonctions. Par exemple le processus de réduction d'un terme vers sa forme normale peut être réifié par la fonction :

Op meta-reduce : Module Term → Term [special …] .

Ou meta-reduce($\overline{M}$, $\overline{t}$) retourne la meta-représentation $\overline{u}$ de la forme normale u du terme t dans le module M. Le processus de l'application d'une règle est aussi réifié par la fonction :

op meta-apply : Module Term Qid Substitution MachineInt → ResultPair [special …],

Meta-apply($\overline{M},\overline{t},\overline{l},\overline{\sigma},\overline{n}$) retourne le résultat de l'application de la règle l, instanciée partiellement par σ, dans M vers la forme normale du terme t, utilisant $(n+1)^{ième}$ égalités (application). Le résultat est retourné sous forme d'un terme $\{\overline{t'},\overline{\sigma'}\}$ ou t' étant la forme normale du terme résultant, et σ' est le pas utilisé par la règle d'application. Le résultat {error*, none} est aussi retournée si moins de (n+1) pas de réduction sont utilisés. MAUDE est réifié par la fonction meta-rewrite. L'analyse syntaxique est réalisée par la fonction :

op meta-parse : Module QidList → Term [special …] .

6.7.4 Full-Maude

Les modules MAUDE constituent des objets de première classe, il est relativement aisé de définir en MAUDE (META-LEVEL), différentes opérations sur les modules. Le système plutôt sous-système Full-Maude écrit par Duran et Meseguer [Dur99, DCE+98, CDE+04], étant une spécification écrite en totalité dans MAUDE (core), définie un langage mais aussi un outil pour la spécification et l'exécution des modules écrits en MAUDE. Les modules de Full-Maude sont constitués d'un ensemble de modules spécialisés tels que le *module-algebra* avec *module-hierarchies*, *module-parametrization*, views, theories et un module expressions. Il supporte aussi la spécification de modules "orienté-objet" utilisant une syntaxe mettant en place tous les éléments de déclaration dans le paradigme de l'orienté objets tels que la notion de classe, d'objet, de message, d'héritage, de polymorphisme etc. Full-Maude utilise LOOP-MODE pour lire les modules et "system commands" pour que la base de données des modules soit présentés d'une façon persistante ainsi que pour mettre à la disposition de l'utilisateur les résultats des commandes en les plaçant dans un *buffer* en sortie.

6.8 Conclusion

Dans ce chapitre nous avons présenté le fondement même de la logique de réécriture qui est basée sur deux concepts théoriques très connus depuis bien longtemps : les systèmes de réécriture et les spécifications équationnelles.

Nous avons aussi montré que comme la logique de réécriture est universelle, c'est donc qu'elle possède un pouvoir d'abstraction et de description de tout type de système. Nous avons explicitement définit l'existence d'un isomorphisme fonctionnel permettant le passage d'un modèle vers sa description formelle en logique de réécriture qui à son tour permet la projection de ce modèle dans la théorie des catégories qui est un puissant formalisme mathématique de représentation et de démonstration. Le chemin inverse est aussi aisé que le premier.

Le mécanisme de la logique de réécriture et de son langage porteur qui est Maude à été décrit montrant la facilité avec lequel Maude permet de décrire les programmes sous forme de module fonctionnel, de module objet mais aussi de module système. La refléction, la logique de *Membership*, la notion d'algèbre *Many-sorted* et de *Order -sorted* confortent la puissance de Maude.

CHAPITRE 7

Spécification des Systèmes Temps-Réel dans le Cadre de la Logique de Réécriture

7.1 Introduction

Nous avons montré dans le chapitre précédent que la logique de réécriture est un modèle formel très expressif à travers duquel différents types de modèles concurrents peuvent être exprimés et spécifiés de façon naturelle. Son aspect théorique lui permet de mettre en place des mécanismes (des réécritures) pour la spécification et l'intégration de caractéristiques temporisées des systèmes temps-réel qui s'expriment aisément à travers une syntaxe ordinaire et une sémantique opérationnelle faisant ressortir le temps comme une variable de type (sorte) *Time*.

La première contribution en matière de recherche pour l'application de la logique de réécriture dans la spécification du facteur temps réel fut le travail de Kosiuczenko et Wirsing dans le cadre des TRL (Timed Rewriting Logic) [KW96] et de [OM01a, OM01b, Olv02, Reb00]. Dans ce qui suit, nous allons montrer comment la logique de réécriture permet de spécifier les systèmes temps réels. Ceci se fera à travers l'introduction d'un paramètre explicite du temps satisfaisant des axiomes généraux de passage du temps. L'aptitude d'exprimer le temps dans la logique de réécriture est réalisée par des théories de réécriture temporisée qui seront traduites vers des théories de réécriture ordinaire non-temporisées en considérant une ou un ensemble d'horloges explicites.

Nous abordons dans ce chapitre les systèmes temps réel d'un point de vue purement formel dans une approche orientée vers la logique de réécriture. La notion de temps est alors introduite en utilisant une certaine axiomatisation définie dans des domaines variés (entiers, réels, etc) selon le type. Ces définitions sont écrites sous forme de théories fonctionnelles de la logique de réécriture (modules fonctionnels et objets Maude). Le système abstrait temps réel est ensuite définit moyennant une structure statique interprétée à l'aide d'un type de données abstrait (ADT : Abstract Data-type), plus une structure dynamique définie par un ensemble de règles de réécriture.

Le modèle objet temps réel est alors introduit à travers un exemple pour montrer la puissance de la logique de réécriture à prendre en charge le modèle formel orienté-objet réactif. Par la suite un modèle d'une importance capitale est introduit, montrant la simplicité naturelle pour déclarer le comportement sous forme de règles étiquetées par le paramêtre « temps ». Ces déclarations sont éffectuées par une sémantique opérationnelle à la Plotkin.

Toutes ces définitions mettent en place des mécanismes théoriques très puissants pour formaliser, spécifier et vérifier les systèmes réatifs temps réels.

Le but de cette section est de montrer que la logique de réécriture est une plate forme théorique universelle qui permet d'exprimer les systèmes concurrents à travers leurs structures, leurs comportements et surtout de détailler le modèle de communications.

7.1.1 Spécification des systèmes temps réels

Nous utilisons les symboles r, r', rl,.... pour dénoter les valeurs du temps; x_r, y_r, z_r ..., pour dénoter les variables d'une sorte *Time* et t_r, t'_r,…… pour dénoter des termes de sorte *Time*.

7.1.1.1 Domaines Temporels

Le temps est considéré dans cette partie comme une entité abstraite utilisée pour exprimer les contraintes temporelles sur des actions temporelles.

Définition 7.1. *Un domaine temporel peut être considéré comme un monoïde commutatif (D,+,0) qui satisfait les besoins suivants :*

- $d + d' = d \Leftrightarrow d' = 0$
- *left-cancellative:* $d + d' = d + d'' \Rightarrow d' = d''$
- *anti-symetrie:* $(d + d' = 0) \Rightarrow d = d'$

- *le preorder $\leq$ défini par $d \leq d' \Leftrightarrow \exists d'' / d + d'' = d'$est un ordre total.*

Les propriétés suivantes peuvent aisément être prouvées à savoir-- *(i) 0* est le plus petit élément de **D**, *(ii)* pour tous d, d' $\in$ **D**, si d < d', alors il existe un élément d'' $\in$ **D** tel que d + d" = d' et qui soit unique, il sera dénoté par (d' – d). Nous dénotons **D** – {*0*} par **D** *. Nous écrirons d < d'au lieu de d $\leq$ d' $\wedge$ d $\neq$ d'. **D** est dit dense *si* $\forall$d, d' : d < d' $\Rightarrow \exists$ d": d<d"<d', **D** est dit discret *si* $\forall$d,$\exists$ d': d<d' $\wedge$ d": d < d" $\Rightarrow$ d' < d". Comme la relation d'ordre est totale entre les éléments de **D**, tout élément de **D** peut être obtenu à partir de zéro en additionnant le successeur de ce dernier. Exemple de domaines temporisés: Les nombres naturels (domain discret), Q^+ et $\Re^+$ (dense) ou bien le singleton {0}.

7.1.1.1.1 Déterminisme du Temps

Il est admit que lorsqu'un processus P par exemple est inactif (n'exécute aucune action) pendant une durée d, le comportement résultant est complètement déterminé par le processus lui-même P et la durée d'exécution. La progression du temps peut être exprimée par : [Reb03, Olv00]

$$\forall P, P', P'', d : P \xrightarrow{d} P', P \xrightarrow{d} P'' \Rightarrow P' = P''$$

ou '=' étant l'égalité syntactique. P, P', et P" sont des processus et d étant la quantité de temps que prenne un processus pour passer d'un état à un autre.

7.1.1.1.2 Additivité du Temps

Afin d'assurer la nécessité en terme mathématique de la notion de temps, il est demandé de respecter

(i) un processus qui peut devenir inactif pendant d + d'unités de temps, doit rester inactif pendant d unités de temps, puis pendant d'unités de temps, et vice-versa.

(ii) *(ii)* dans les deux cas le résultat doit être le même.

La propriété d'additivité du temps est formellement définie par :

$$\forall P, P', d, ,d' : (\exists P'': P \rightarrow^{d} P'', P'' \rightarrow^{d'} P') \Leftrightarrow P \rightarrow^{d+d'} P'$$

Nous pouvons remarquer un peu plus tard que la propriété d'additivité est exprimée naturellement dans la logique de réécriture temporisée par l'axiome de transitivité

7.2 Modèle temporisé

Le temps est modélisé par un monoïde commutatif (*Time, +, 0*) muni d'opérateurs additionnels <, - (« Monus ») écrits dans le langage Maude (donc il satisfait la théorie Maude).

fth *TIME* is
Protecting Bool
sort Time .
op 0 : → *Time* .
op _ + _ : *Time Time* [*assoc comm id* : 0] .
op _<_,_<=_ : *Time Time* → *Bool* .
op _ - _ : *Time Time* → *Time* .

vars x_r, y_r, z_r, w_r : *Time* .

ceq $y_r = z_r$ **if** $x_r + y_r == x_r + z_r$.

eq $(x_r + y_r) - y_r = x_r$.

ceq $(x_r < y_r) = true$ **if** $(x_r < y_r)$ **and** $y_r < z_r$.

eq $(x_r < x_r) = false$.

eq $0 <= x_r = true$.

eq $(x_r <= y_r) = (x_r < y_r)$ **or** $(x_r == y_r)$.

ceq $x_r + y_r <= z_r + w_r = true$ **if** $x_r <= z_r$ **and** $y_r <= w_r$.

eq $x_r <= (x_r - y_r) + y_r = true$.

ceq $(x_r - y_r) + y_r = x_r$ **if** $y_r <= x_r$.

ceq $x_r - z_r <= y_r - z_r = true$ if $x_r < y_r$.

endft

Nous remarquons que la relation - - est une relation d'ordre partiel telle que pour tout : x_r, y_r: Time, x_r = true, et $y_r <= x_r$ si et seulement s'il existe z_r unique tel que $x_r = y_r + z_r$.

Pour la simulation et l'exécution de la spécification proposée, nous seront plutôt intéressés par les modèles calculables de la théorie *Time* en question. Cela signifie que toutes les opérations sont calculables. Ce type de *modèles* est « finitely » spécifiable *comme* une algèbre initiale pour un ensemble E de Church-Rosser et d'équations terminant «terminating » [OM01b].

S'il nous arrive de rajouter une valeur additionnelle TG (Très Grand qui tend vers l'infinie) de type temps (*Time*), celle-ci peut être spécifiée dans le langage Maude par ce qui suit.

fth $TIME_{TG}$ is

including *TIME* .

sort $Time_{TG}$.

subsort *Time* < $Time_{TG}$.

op TG : → $Time_{TG}$.

op _<=_ : $Time_{TG}$ $Time_{TG}$ o → Bool .

op _ - _ : $Time_{TG}$, *Time* → $Time_{TG}$.

op _ + - : $Time_{TG}$ $Time_{TG}$ → $Time_{TG}$ [assoc comm id : 0] .

var x_r : *Time* .

var y_r : $Time_{TG}$.

eq $y_r <= TG = true$.

eq $TG <= x_r = true$.

eq $x_r < TG = true$.

eq $TG < y_r = false$.

eq $TG - x_r = TG$.

eq $TG + y_r = TG$.

endft

Dans le cas ou le temps peut être supposé linéaire, il peut être spécifié dans la théorie suivante:

fth *LTIME* is
including *Time* .
op min : *Time Time* → *Time* [comm] .
vars x_r, y_r, : *Time* .
ceq $x_r = y_r$ **if** *not*($x_r < y_r$) and *not*($y_r < x_r$) .
ceq min (x_r, y_r) = y_r if $y_r <= x_r$.
endft

Nous pouvons aussi étendre cette théorie avec le caractère **TG** du temps très grand comme suit :

fth $LTIME_{TG}$ is
extending *LTIME* $TIME_{TG}$.
op min : $Time_{TG}$ $Time_{TG}$ → $Time_{TG}$ [comm] .
var x_r : $Time_{TG}$.
eq min (TG, x_r) = x_r .
endft

7.3 Expression du temps comme une action

Le passage du temps peut être exprimé par une action qui aura un effet sur chaque composante du système. Dans ce cas, et comme il a déjà *été* dit, le temps est modélisé par un monoïde commutatif muni d'une structure additionnelle. L'action dans ce cas est alors axiomatisée par la fonction *f* de type *f : State Time* → *State* satisfaisant les axiomes usuels suivants:

$$f(x,0) = x$$

$$f(f(x,y_r),z_r) = f(x,y_r + z_r) .$$

La fonction *f* permet d'exprimer l'effet du passage du temps dans le système.

L'effet du passage du temps est matérialisé par des règles du type *tick*, donc un tick correspond à une unité de temps. Ce type de règle s'exprime par :

$$\textbf{crl}\ [\text{tick}] : \{t\} \xrightarrow{r} \{f(t,r)\}\ =\ \text{if}\ C\ .$$

Exemple: La spécification suivante suppose modéliser un temps discret "clock", ou *Time* est une sorte prenant ses valeurs dans **N** (nombres naturels), qui est réinitialisée quand elle atteint la valeur 24.

sorts *State System* .
op clock : Nat → State .
op *f* : State Nat → State .
op {-} : State → System .
vars x_r, y_r : Nat .
var z : state .

eq $f(\text{clock}(x_r),y_r) = \text{clock}(x_r + y_r)$.

rl [reset] : clock(24) → clock(0) .

rl [tick] : $\{z\} \xrightarrow{x_r} \{f(z,x_r)\}$.

Supposons aussi que le temps agit sur un terme **a** pour deux unités de temps. Dans ce cas le terme **a** se réécrit en **b**, qui peut alors à son tour se réécrire en c, si le temps ne peut agir sur lui. Les équations d'un tel système se réécrivent :

$$\{f(a,2) \rightarrow b,\ b \rightarrow c,\ f(c,2) \rightarrow d\}$$

7.4 Logique de réécriture en tant que sémantique pour les systèmes temps réels.

7.4.1 Application aux automates temporisés

Nous pouvons omettre les détails sur les états initiaux et les conditions d'acceptation. Donc nous pouvons d'ores et déjà définir un automate par ce suit et qui consiste-en :

Rappelons la définition d'un automate temporisé telle que définie auparavant. Donc, nous supposons que l'automate temporisé A = (∑,S,C,E) peut être représenté d'une façon naturelle dans la logique de réécriture par la théorie de réécriture temps-réel $\psi_{TA}(A) = ((\sum_A,E_A,L_A,R_A),\phi_A,T_A)$ comme suit :

- $(\sum_A,E_A)$ contient une axiomatisation équationnelle du domaine temporisé R^+. De plus, la signature $\sum_A$ content une sorte TAState avec une constante s : → TAState pour chaque sorte s∈S, une sorte State munie d'un opérateur (n+1)-aire _, _ ,_ , … , _ : TAState TIME … Time → State, et un opérateur {-} : State → System .
- L'ensemble L_A des labels est $\sum \cup \{\text{tick}\}$.
- L'ensemble des règles RA et son affectation (duration) contient une règle instantanée,

$$[a] : \{s,x_1, \dots , x_n\} \rightarrow \{s',t_1, \dots , t_n\} \text{ if } \varphi(x_1, \dots, x_n)$$

pour chaque transition $(s,s',a_i,\lambda, \varphi) \in E$, telles que les variables x_i sont de type sort *Time* et les termes t_i sont nuls si $c_i \in \lambda$ et $t_i = x_i$ sinon, et $\varphi(x_1, \dots, x_n)$ est obtenu à partir de φ par substitution de chaque horloge c_i avec la valeur x_i et en remplaçant les opérateurs ∧ et ¬ par "and" et le "not". En plus de cela une règle du type :

$$[\text{tick}] : \{z,x_1, .. ,x_n\} \xrightarrow{y_r} \{z,x_1 = y_r, .. ,x_n + y_r\}$$

(ou z, yr, $x_1, \dots , x_n$ sont des variables) est rajoutée au model du passage de temps.

L'automate temporisé de la figure 7.1 sera représenté par la théorie de réécriture temporisée qui contient les règles suivantes :

rl [a] : $\{s_0,x,y\} \rightarrow \{s_1,0,0\}$.

crl [b] : $\{s_1,x,y\} \rightarrow \{s_2,0,y\}$ if $5 \le x \le 10$

crl [a] : $\{s_2,x,y\} \rightarrow \{s_1,0,0\}$ if $y \le 15$ and $4 \le x \le 8$

rl [tick] : $\{z,x,y\} \xrightarrow{y_r} \{z, x + y_r, y + y_r\}$

pour x, y, et yr des variables de sorte Time, et z une variable de type TAState.

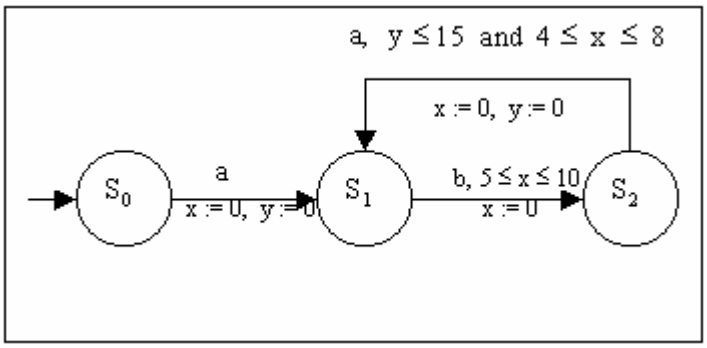


Figure 7.1 : Graphe d'un Automate

7.4.2 Systèmes temps-réel orienté-objets

Dans un système concurrent orienté-objet [Olv00, Reb00], l'état qui est appelé *configuration* possède la structure d'un multi-ensemble constitué d'objets et de messages. La déclaration en MAUDE d'une telle configuration s'écrit selon la syntaxe suivante :

op _ _ : Configuration Configuration → Configuration [assoc comm id : none] .

Ou l'opérateur de l'union (multi-ensemble) est déclaré pour satisfaire les lois structurelles de l'associativité, de la commutativité et de l'identité *none*. La déclaration du sous-sorte (subsort)

Object Msg ≤ Configuraton

qui déclare les objets et les messages comme un singleton (multi-ensemble) de configurations à partir duquel des configurations plus complexes peuvent être générées par l'union de multi-ensembles.

Object ≤ ObjConfiguration ≤ Configuration

et l'opérateur

_ _ : ObjConfiguration ObjConfiguration → ObjConfiguration [assoc comm id : none].

Les objets sont des termes de type Object de la forme déclarée de la même manière qu'auparavant, c'est-à-dire $< O : C \mid att_1 : val_1, \ldots, att_n : val_n >$ dénotant un objet O, ou O est de sorte Oid (identificateur d'objets).

Le comportement dynamique d'un système concurrent orienté-objet sera axiomatisé en spécifiant chacune de ces transitions courantes par une règle de réécriture. Par exemple la règle suivante définie une transition temporisée :

rl [l] : m(O,w) < O : C |att1 : x, att2 : y, att3 : z >

→

< O : C |att1 : x + w, att2 : y,att3 : z > m'(y, x + w)

En résumé, un système orienté objet peut être spécifié à l'aide de théories de réécriture temps-réel comme c'est déjà fait, plus une extension à l'aide de l'opérateur suivant :

{_} Configuration → System .

Nous illustrons cette partie de notre présentation des systèmes temps-réel dans le cadre de la logique de réécriture par l'exemple suivant [Reb02a].

7.4.3 Exemple d'application

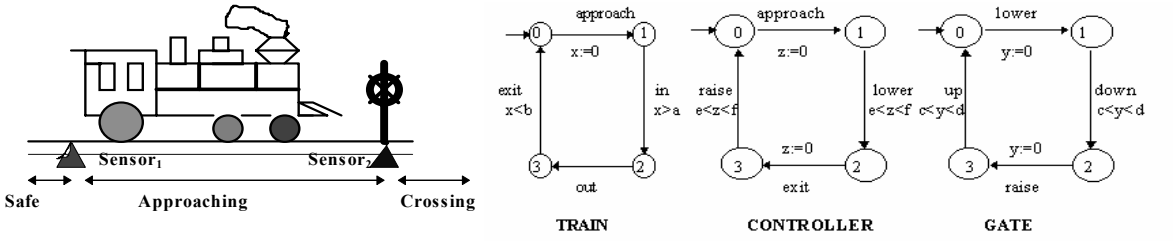


Figure 7.2 : Un passage à niveau et son automate correspondant

7.4.3.1 Spécification d'un croisement au niveau d'un passage à niveau

Un système concurrent gérant le croisement au niveau d'un passage à niveau est un dispositif composé formellement d'un ou de plusieurs trains, d'un contrôleur et d'une barrière automatique. La partie la plus intéressante dans ce système étant de gérer ce passage pour qu'il n'y ait pas d'accidents entre trains et automobilistes passant en sens croisé. Ce système peut être représenté par trois processus qui évoluent tout à fait en parallèle, disons *Train* pour le train, *Controller* pour le contrôleur et *Gate* pour la barrière. Relatif au passage à niveau tout train peut se trouver dans l'un des trois modes suivants : *safe*, *approaching* ou *crossing*. Tout train passe d'un état *safe* lorsqu'il se trouve très loin de l'intersection, puis arrivé à une distance connue, il passe à l'état *approaching*, et finalement à l'état *crossing* et lorsqu'il quitte le passage à niveau il devient *safe* (Figure 7.2).

Quand le train pénètre dans la surface *approaching*, un objet de sorte *Timer* est forcé avec la valeur d_1 qui est le temps nécessaire avant que le système déclenche le signal *lower*. Lorsque d_1 unités de temps auraient été consommées le Timer déclenche une opération interne de Time-out et le signal *Approach_Signal* est transmis au système de contrôle. Quand le Contrôleur reçoit ce message, il positionne son état interne à "lower" et envoie le signal *lower_signal* au système qui contrôle la barrière qui lui donne l'ordre de se mettre à la position "lowered" qui doit se faire dans un délai de d_2 unités de temps qui est l'intervalle de temps prévu entre le dernier time-out l'aire de croisement "crossing area" et la position de la barrière "gate position". Lorsque la barrière se met à baisser, le contrôleur de cette dernière réinitialise son Timer à d_3 unités de temps qui correspond exactement au temps qu'il faut pour un train de dépasser l »aire de croisement. Quand le quantité de temps d_3 est consommée le Timer time-out, le signal Exit_Signal est alors envoyé au contrôleur qui à son tour repositionne son état interne à la position "raising" (remonter) et transmet le signal raising_signal au contrôleur de la barrière pour qu'elle se remette à la position "raised" pour libérer la route. L'un des critères les plus importants étant de s'assurer que lorsque le train se trouve sur l'aire d'intersection la barrière doit être fermée (critère de sûreté). Pour assurer une telle spécification nous devons supposer que le train passe de l'état safe à l'état crossing pendant un temps égale à $d_1 + d_2$ unités de temps.

Nous nous arrêtons à ce point de définition et nous redirigeons le lecteur sur d'autres détails de la spécification et de la vérification dans le langage MAUDE du système global et de certaines propriétés qui sont consignées (voir les articles [Reb00, Reb02a]). Nous informons aussi le lecteur qu'une autre approche similaire pour définir ce système (critique temps réel) se trouve introduite dans le chapitre 8.

7.5 Conclusion

Dans ce chapitre, nous avons montré les possibilités qu'offrent la logique de réécriture et le langage MAUDE pour spécifier les systèmes temps-réel ordinaires mais aussi ceux à caractéristiques objets ainsi que les automates temporisés. Nous avons vu, qu'avant d'entamer la description du système structurellement, il faut tout d'abord définir méticuleusement le ou les types de temps qu'il faut au préalable mettre en place. Ces définitions sont simplistes puisque nous utilisons des théories (modules fonctionnels) élémentaires qui sont ensuite sont généralisées.

CHAPITRE 8

Proposition d'un Environnement pour la Modélisation des Systèmes Réactifs

8.1 Introduction

Le développement des systèmes temps réel revêt une importance qui ne cesse de grandir. En effet, l'essor croissant des technologies permet de définir des systèmes informatiques sophistiqués et d'une complexité difficilement maîtrisable.

Dans cette partie de notre travail, nous nous intéressons à la modélisation et à la conception des systèmes réactifs et plus spécialement ceux qui sont temps réels. Cette modélisation certes passera par plusieurs étapes, mais, nous ne retiendrons qu'une approche plutôt graphique qui sera communiquée au système (le noyau), et qui a son tour ferra en sorte de lui faire subir certaines transformations pour la faire passer du mode graphique vers un mode textuel, donc, qui sera réécrite en programmes qui fourniront différents codes et machines d'états.

L'objectif le plus important de toutes ces transformations étant de ramener la forme graphique d'un programme qui est de surcroît formel vers une version condensée, formelle (langages algébriques) est minimale (équivalence de bisimulation). En effet, pour des critères essentiellement économiques, le système doit être modélisé à l'aide de langages dit formels très tôt dans le cycle de vie, cela, permet d'assurer sa cohérence et sa faisabilité. Le surcoût indéniable entraîné par l'utilisation de ce type de langages reste largement inférieur à l'impact d'un dysfonctionnement lors de l'exécution du système ou d'une correction tardive au cours du développement.

Malheureusement, les langages formels se révèlent être d'une grande complexité, leur compréhension reste réservée à des spécialistes de la modélisation formelle. Or, le développement de systèmes temps réel fait appel à différentes spécialités, que ce soit en informatique (sûreté de fonctionnement, protocole, temps réel, base de données), en automatique, en électronique… Les langages formels sont alors peu adaptés aux échanges pour lesquels il est nécessaire de favoriser la communication entre les différentes spécialités. De plus, ils obligent généralement l'analyste-concepteur à entrer rapidement dans des niveaux de détails qui ne facilitent pas une compréhension globale du système en construction.

Pour palier à ce type de perturbation dans la conception des systèmes, l'utilisation conjointe de notations semi-formelles et de langages formels (approches dites mixtes) est de plus en plus utilisée. Les notations semi-formelles simplifient la capture des besoins, le dialogue entre les différents spécialistes intervenant dans le développement, le passage des fossés sémantiques *(semantic ga*p) que représentent les transitions entre les phases d'analyse des besoins *(requirement*s), de spécification du système *(analysi*s), de conception préliminaire *(global desig*n) et surtout, la validation du modèle qui sort de son carcan traditionnel pour épouser un aspect purement formel pour la vérification de propriétés (désirs) qui sont au préalables mises à la disposition du concepteur.

Avec l'apparition de la notation UML *(Unified Modelling Languag*e)[OMG99a, OMG01], la question du choix d'une représentation semi-formelle ne se pose plus. Son statut de standard «de facto», sa richesse d'expression, ses capacités d'extension, son large spectre de couverture des phases du cycle de vie, le nombre d'outils informatiques l'implémentant et son adoption par la plupart des universitaires et des industriels la rend accessible à un large public.

Dans un contexte ou un nombre important de méthodes et de langages prolifèrent (Plus de 50 méthodes objet sont apparues durant la période 1990-95, mais aucune ne s'est imposée (SA, SA-RT, Booch, Classe-Relation, Fusion, HOOD, OMT, OOA, OOD, OOM, OOSE...) [RBP91], l'idée de standardiser un langage rassembleur pour modéliser d'un côté les systèmes dans des contextes divers, puis de les coupler avec un langage à résonance formelle est devenue la priorité en matière de recherche de plusieurs équipes universitaires et concepteurs industriels.

Dans ce cadre d'idée, nous joignons notre effort pour promouvoir cet axe de recherche et ainsi apporter un plus, donc nous nous sommes intéressés à faire coupler UML et Maude [Reb04a, Reb04b, ATH00] à travers naturellement la logique de réécriture. Pourquoi un tel choix ? Nous avons deux réponses qui vont étayer le fondement d'une telle approche. L'une a été à moitié élucidée dans le chapitre 6, elle se base sur le fondement théorique du langage Maude à savoir la logique de réécriture et les spécifications algébriques, contexte théorique prouvé certes mais qui se trouve de même être un

puissant outil pratique, mais aussi qui modélise de façon remarquable tous les processus et leur comportement fonctionnel et dynamique, sans oublier le fait que Maude incorpore en son sein (core = noyau) cette notion d'algèbre universelle et de réflection lui permet d'engendrer, mais plutôt de générer une multitude de théories mais aussi de langages. Une preuve tangible est consignée dans les travaux de Meseguer et al., mais aussi ceux d'autres auteurs [OM01a , OM01, MM96, DCE+98, Mes98, FH98, HR00, VMO03, EMS02, Cla01]. A titre d'exemple, la Figure 8.1 présente un exemple de modélisation par réseaux de Pétri et sa spécification en Maude. Nous pouvons continuer ainsi et décrire des problèmes de différents types dans des langages ou des formalismes, qu'ils soient formels, semi-formels ou naturel et leur trouver une description équivalente en Maude. En plus de cela, c'est l'aisance même qu'offre le langage Maude à rendre la spécification exécutable et fournir ainsi des résultats qu'ils soient ciblés ou bien selon un jeu de simulation. La deuxième réponse étant que la syntaxe et la sémantique du langage Maude permettent de donner une représentation très précise de tous les éléments descriptifs de la notation UML [DV01, ATH00, AT00, FT99, Reb03, Reb04].

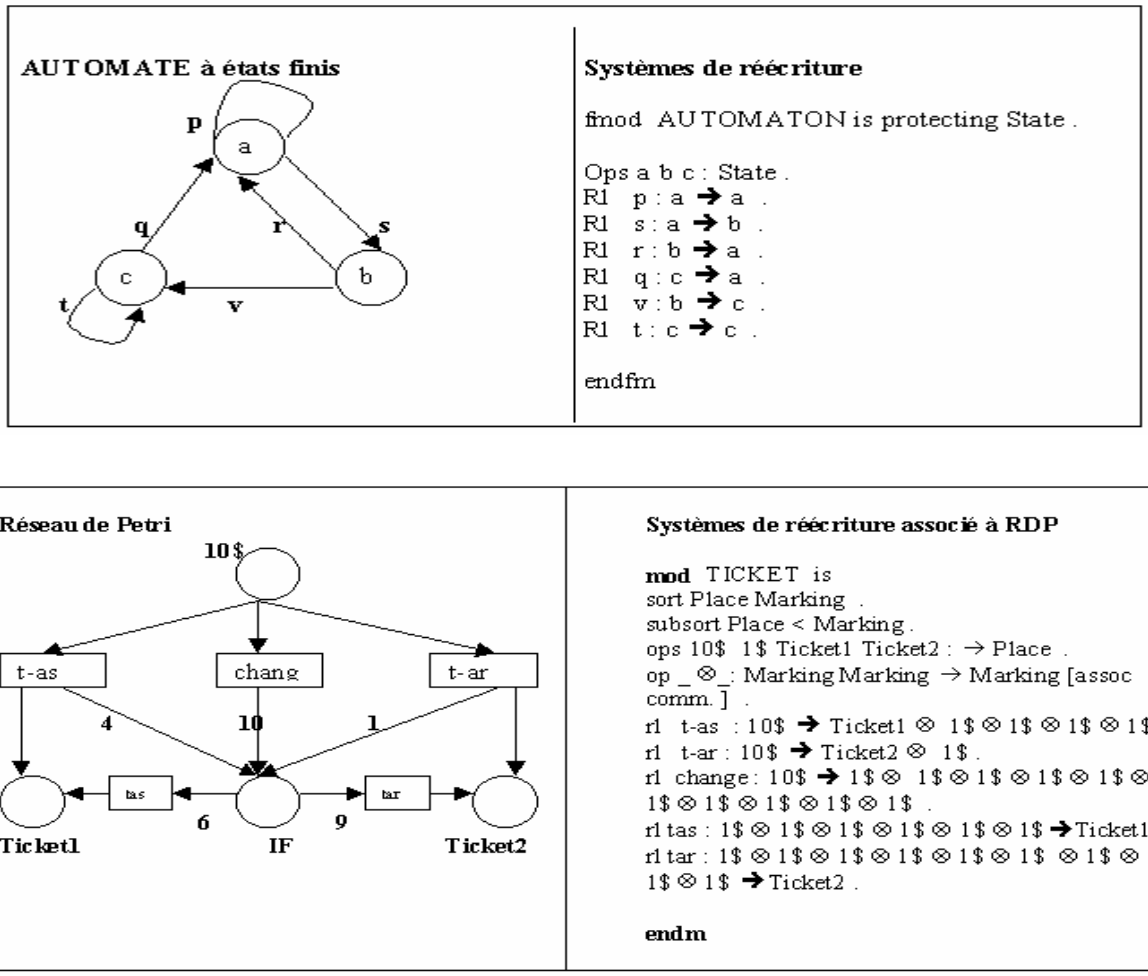


Figure 8.1 : Représentation de différents Modèles dans le Langage Maude

8.2 Choix d'une méthodologie de modélisation

Lors de l'implémentation d'un système temps réel, le logiciel est écrit d'une part dans un langage de programmation comme le langage "C" ou ADA, et d'autre part il s'appuie sur les services d'un noyau temps réel qui fournit au concepteur tous les outils pour contrôler l'environnement logiciel des tâches. L'utilisation de la programmation multitâche basée sur les algorithmes d'ordonnancement temps réel

permet de mieux respecter les exigences temps réel (respect des contraintes temporisées, délais et autres).

Les contraintes temporelles qui sont essentielles lors de la construction de l'architecture permettent justement de valider le comportement temps réel du système.

8.2.1 Choix d'un langage pour la méthodologie

Il est bien connu que SA-RT (Structured Development for Real-Time System) [MW85] est un langage fonctionnel qui permet de spécifier un système embarqué temps réel tant du point de vue logiciel que matériel (architecture). Contrairement à la méthode S.A qui représente seulement un réseau de processus transformant des données, SA-RT a le mérite d'être une méthode bien adaptée pour le temps réel. Elle prend en compte la notion d'événement et de séquencement de l'activation des différents processus ainsi que leur synchronisation. A partir d'une spécification SA-RT, CODARTS définit des principes de mise en place d'un modèle d'exécution multi-tâches temps réel. SA-RT et CODARTS proposent des principes intéressants mais restent dans le cadre des approches informelles non validables. Son modèle de contrôle permet de modéliser la dynamique du système et son comportement vis à vis des stimulations externes. L'aspect graphique et la nature hiérarchique font que SA-RT soit une méthode conviée pour définir la spécification de l'architecture temps réel. Afin de bénéficier des atouts de l'approche objet, UML-RT (UML Real-Time) [SGW94, SR98] est utilisée pour la description des systèmes embarqués temps réel. Malheusement, bien que la méthode SA-RT offre un formalisme plus complet, elle comporte néanmoins certaines lacunes qui la rende moins fréquentée, nous citons à titre d'exemple une syntaxe pas du tout conforme à certaines tâches, une mauvaise interprétation des règles de connexion, et la circulation des données entre processus ne sont pas toujours clairement établies. Le dictionnaire de données et les mini-spécifications peuvent être décrites en langage naturel ce qui en général, est un handicap latent causant des interprétations tout à fait ambiguës.

Devant de telles problématiques des méthodes structurées qui engendrent en général une lourdeur dans l'interprétation des modèles, mais aussi une complexité de plus en plus difficile à maîtriser, les méthodes orientées objets se sont tout de suite illustrées, assurant par la même occasion, une efficacité d'interprétation, une représentation des plus exiguës et une réutilisation raffinée mais aussi la génération de code épousant par la même occasion les concepts des différents langages de programmation orientés objets. A titre d'exemple, nous pouvons passer très facilement de la description UML vers les langages Java, C++ ou autres… Comme les approches objets permettent justement une représentation proche du naturel, une adéquation permet de les différencier [You91, Hil93] des autres techniques à savoir :

Approche Orientée Objet = Classes et Objets + Héritage + Communication par Messages

Plusieurs méthodes se sont imposées, les unes pour le temps réel telle que la méthode HOOD [Hei87], d'autres pour les systèmes d'information et de gestion telles que la méthode OOA Object Oriented Analysis) de Grady BOOCH l'un des concepteurs de UML, Mérise Objet, mais d'autres aussi qui se veulent être plus général telle que la méthode OMT (Object Modeling Technique) [Hil93], et finalement celle qui a eu un consensus très large appelée UML. Cette dernière (UML) apporte un langage de modélisation qui se veut universel pour les systèmes d'information mais ne propose pas de méthodologie ou de cadre formel au développement de logiciel. De plus, l'approche objet, avec son modèle de communication par message, induit une architecture qui peut ne pas être performante du point de vue temporel. Des corrections sont alors nécessaires sur le modèle d'exécution généré pour introduire des politiques temps réel. Enfin, la sémantique d'UML n'est pas précise du fait de son caractère universel et de la mise en place des différentes variations. De nombreuses études utilisent UML comme langage d'analyse et lors de la phase de conception, utilisent un langage formel spécifique pour l'implémentation des modèles [Sel98, OMG99a, OMG99b, OMG01, Jar98, Har87, Har88, ABD98, Sch04]. Ceci permet de préciser la sémantique du modèle et de réaliser des preuves. Il est aussi possible de décrire directement le système à l'aide d'un langage formel. Il existe de nombreux langages formels pour la modélisation des systèmes concurrents.

Toutes les études présentées jusque là, apportent des principes, des solutions ou des modèles pour la mise en place du modèle d'exécution multitâches. Le but de notre étude est de proposer un cadre formel au développement de système temps réel, c'est à dire de formaliser les principes et les solutions exprimées à partir de modèles formels. On s'intéresse plus particulièrement à la formalisation de la phase du développement qui aboutit à la définition du modèle d'exécution de l'application. Nous utiliserons le langage UML et nous présentons maintenant les diverses phases de l'approche proposée.

8.3 La notation UML (Unified Modelling Language)

8.3.1 Les bénéfices d'UML

Le principal argument en faveur d'UML est son statut de standard *de fact*o. Parmi la multitude de méthodes OO (et les notations qui leur sont associées), disposer d'un consensus dans leur représentation est un avantage certain.

Bien qu'issu des notations OMT et OOD, UML possède un plus grand pouvoir d'expression que ses aînées et couvre une plus grande partie du cycle de développement. Dans la description des aspects statiques du système, la richesse de description est grande et rien ne semble avoir été oublié. La question de la représentation d'un objet ou d'une classe et de ses relations statiques ne se pose plus.

Cet état de fait nous permet d'espérer un meilleur dialogue avec les clients, qui n'ont plus qu'un seul formalisme à connaître (puisque UML est un standard). De plus, les diagrammes UML sont assez intuitifs et aisés à comprendre, car ils sont une reprise de formalismes éprouvés et utilisés depuis une décennie. Par ailleurs, les possibilités d'extension [Gue00, TG00, WTB+00] de cette notation la rendent extrêmement adaptable. Ces mécanismes d'extension sont multiples et très intéressants. En effet, UML repose sur une description en 4 couches afin d'être compatible avec le standard MOF *(Meta-Object Facilit*y) de l'OMG. Chaque couche est une instance de sa couche supérieure, la plus haute couche étant une instance d'elle-même. Ces quatre couches sont :

- Le méta-méta-modèle : Il définit la structure du méta-modèle et le langage nécessaire à la définition du méta-modèle. Il est défini de manière réflexive.

- Le méta-modèle : C'est une instance du méta-méta-modèle. Il définit le langage de la couche "modèle", en l'occurrence ici les diagrammes et les éléments UML.

- Le modèle : Il représente l'ensemble des modèles permettant de décrire un domaine d'information. Ce sont, par exemple, les classes définies pour le système en construction.

- Les objets utilisateur : C'est une instance du modèle. Ce sont les objets du système exécuté.

Il est ainsi possible de définir des extensions au niveau du modèle ou au niveau du méta-modèle.

Cela se fait, par exemple, en spécialisant certains éléments UML (par utilisation des stéréotypes et des valeurs étiquetées). Les extensions du méta-modèle pouvant être regroupées dans un profil UML, elles peuvent être alors directement employées dans des modèles UML adaptés à un domaine d'étude particulier.

UML est un langage conçu pour spécifier, visualiser et documenter les systèmes en général et industriels en particulier. Il est le résultat d'un large consensus (industriels, méthodologiques...) d'experts reconnus, donc il est issu directement du terrain et couvre toutes les phases d'un cycle de développement.

Comme la complexité des systèmes (industriels) augmente, l'utilisation de techniques de modélisation efficaces devient aussi important que le projet lui-même. Donc, la meilleure façon de choisir le bon modèle c'est d'en choisir un standard qui soit visuel pour toucher un maximum d'utilisateurs de différentes catégories et de surcroît à conception orienté objets. Donc, il doit prendre en charge dans ce sens ce qui suit :

- Eléments du model — concepts et sémantique de modélisation fondamentale.

- . Notation — description visuelle des éléments du modèle.
- Guidelines — idiomes d'utilité.

Le choix du type de modèle et du type de diagrammes une fois crée permet d'avoir une influence profonde sur la façon de prendre en charge le problème, ce qui fournira nécessairement une bonne ou une mauvaise solution selon ce choix. Un autre point aussi important étant de faire en sorte d'utiliser l'abstraction comme une clé non des moindres à cause de :

- Chaque système aussi complexe soit-il, est mieux prit en charge une fois décomposé en de petits ensembles de vues indépendantes mais très proches, ce qui fait qu'une seule vue est totalement absurde pour donner un modèle très fiable et représentatif.
- Chaque modèle peut être exprimé à des niveaux différents de fidélité, mais aussi d'abstraction.
- Les meilleurs modèles sont en relation intrinsèques avec la réalité.

En d'autres termes, UML permet de définir et de visualiser un modèle, à l'aide de diagrammes. Un diagramme UML est une représentation graphique, qui s'intéresse à un aspect précis du modèle ; c'est une perspective du modèle, pas le modèle lui-même. Chaque type de diagramme UML possède une structure (les types des éléments de modélisation qui le composent sont prédéfinis). Combinés, les différents types de diagrammes UML offrent une vue complète des aspects statiques et dynamiques d'un système.

En terme de vues d'un modèle, UML définie les diagrammes qui se présentent sous forme graphique et qui sont : use case diagram; class diagram; behavior diagrams; statechart diagram; activity diagram; interaction diagrams; sequence diagram; collaboration diagram etc…

Ces diagrammes constituent une plate-forme de modélisation pour mettre en place toutes les facettes et prendre en charge aussi toutes les perspectives de l'analyse du système en question mais aussi sa construction et son développement. Ces diagrammes si on leur rajoute un moyen de documentation deviennent les premières étapes qu'un concepteur verra.

Le langage UML se présente sous des vues différentes – statiques et dynamiques et qui sont :

8.3.2 Les Classes

Une *classe* est une description d'un ensemble d'objets qui partagent les même attributs, opérations, méthodes, relations et sémantique. Une classe utilise un ensemble d'interfaces pour spécifier des collections d'opérations mises à la disposition de son environnement. Elle définie la structure de donnée des objets. Les classes peuvent être abstraites, c'est-à-dire ne possédant pas les moyens de créer directement leurs propres objets. Tout Objet instantié à partir de sa classe contient son propre ensemble de valeurs correspondant aux caractéristiques structurelles (BehavioralFeatures) et tous les objets d'une classe partagent la définition de la BehavioralFeature à partir de la classe, et tous, ont accès à la seule valeur pour chaque attribut. C'est un *classifieur* qui permet justement de déclarer ces caractéristiques, ce qui fait qu'il possède un nom, donc c'est une *metaclasse*.

Classe : sémantique et notation

Une classe est un type abstrait caractérisé par des propriétés (attributs et méthodes) communes à un ensemble d'objets et permettent de créer des objets :

Classe = attributs + méthodes + instantiation

La figure suivante nous donne une idée comment une classe doit être déclarée graphiquement. Donc, nous voyons clairement que la classe possède un nom (voiture), des attributs (marque, immatriculation …) qui peuvent être public, protégé ou privé. De la même manière, les méthodes déclarées peuvent être public, privé ou bien protégé.

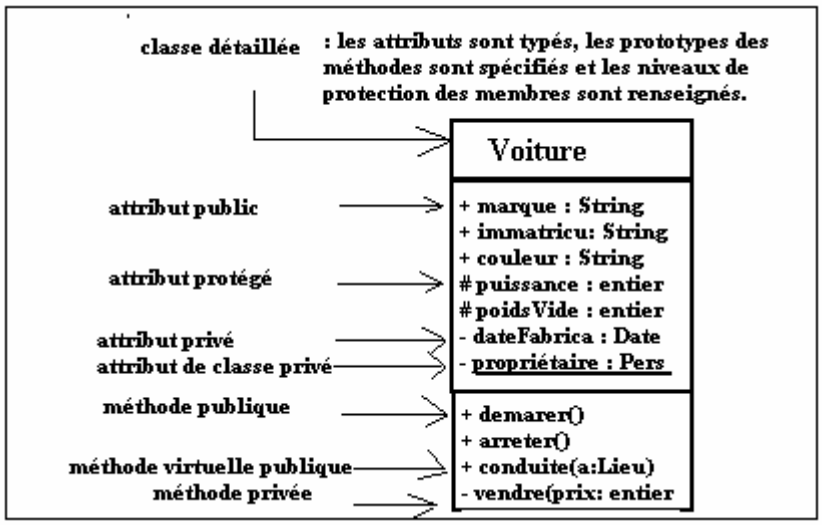


Figure 8.2 : Caractéristiques d'une Classe

8.3.3 Diagramme de Classes

Un diagramme de classes est une collection d'éléments utilisés pour la modélisation statique d'un ensemble d'objets. En d'autres termes, le diagramme de classes fait abstraction des aspects dynamiques et temporels d'un modèle. Naturellement au cas où le modèle est complexe et fastidieux à établir, la nécessité d'utiliser des diagrammes de classes complémentaires s'impose. En général, lors de l'écriture du modèle, le concepteur se focalise plutôt sur les classes qui participent à un *cas d'utilisation* (*use case*), à celles qui sont associées à la réalisation d'un scénario précis, à celles qui composent un paquetage et d'une façon globale, une attention particulière sera donnée en premier lieu à la structure hiérarchique d'un ensemble de classes.

Si l'on considère deux classes appartenant au même diagramme, ces deux classes peuvent être reliées par une connexion sémantique bidirectionnelle qui est l'*association*. Cette association (Figure 8.3) peut être instanciable sous forme de liens. Dans le même sens, on définit ce que l'on appelle le ou les r**ôles** qui spécifient la fonction d'une classe pour une association donnée (indispensable pour les associations réflexives).

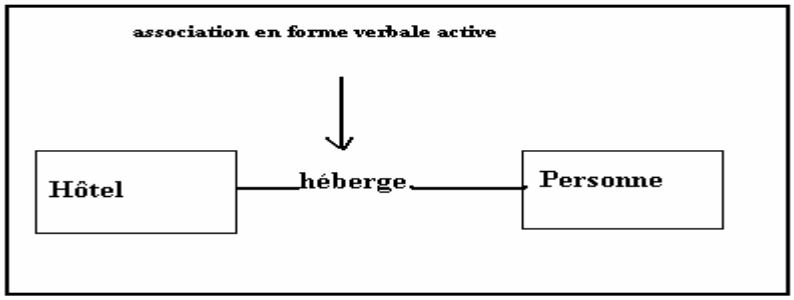


Figure 8.3 : Association entre deux objets

8.3.4 Diagramme de Séquence

8.3.4.1 Diagramme de séquence : sémantique

Les diagrammes de séquences permettent de représenter des *collaborations* entre objets selon un point de vue temporel, on y met l'accent sur la chronologie des envois de messages. Contrairement au diagramme de collaboration, on n'y décrit pas le contexte ou l'état des objets, mais la représentation se concentre sur l'expression des interactions. Les *diagrammes de séquences* peuvent servir à illustrer un

cas d'utilisation (use case). Par contre l'ordre d'envoi d'un message dans une collaboration est déterminé par sa position sur l'axe vertical du diagramme ce qui permet que le temps s'écoule "de haut en bas" de cet axe. Nous remarquons que les diagrammes de séquences et les diagrammes d'états-transitions (statecharts) sont les vues dynamiques les plus importantes d'UML

Dans un diagramme de séquence, plusieurs types de messages peuvent être considérés et qui se présentent dans un diagramme comme des stéréotypes graphiques. Bien entendu, nous avons les messages simples dépourvus des caractéristiques d'envoie et de réception. Les messages minutés (timeout), qu'on connaît bien, c'est une sorte de contrainte temporisée imposée à l'expéditeur pendant un certain laps de temps jusqu'à ce que le destinataire du message se manifeste pendant cet espace de temps. Passé ce délai l'expéditeur est libéré. Les messages synchronisés bloquent l'expéditeur jusqu'à prise en compte du message par le destinataire. Le flot de contrôle passe de l'émetteur au récepteur (l'émetteur devient passif et le récepteur actif) à la prise en compte du message. Les messages asynchrones n'interrompent pas l'exécution de l'expéditeur. Le message envoyé peut être pris en compte par le récepteur à tout moment ou laissé pour compte (jamais traité).

Sur un diagramme de séquence, il est aussi possible de représenter de manière explicite les différentes périodes d'activité d'un objet au moyen d'une bande rectangulaire superposée à la ligne de vie de l'objet. On peut aussi représenter des messages récursifs, en dédoublant la bande d'activation de l'objet concerné. Pour représenter de manière graphique une exécution conditionnelle d'un message, on peut documenter un diagramme de séquence avec du pseudo-code et représenter des bandes d'activation conditionnelles.

8.3.4.2 Envoie de Messages

L'envoie de messages entre les diagrammes d'états peut être représenté par une flèche pleine (noir) de l'objet émetteur vers le récepteur. Les messages doivent être envoyés entre objets, c'est à dire que le diagramme doit contenir des objets (non des classes). La flèche est libellée du nom d'un événement et les arguments qui indiquent clairement la réception de l'événement. Chaque diagramme d'états doit apparaître avec un symbole d'objet représentant un objet de collaboration. Les diagrammes d'états représentent les états de la collaboration des objets.

L'émetteur (du message) peut être :

- *Une transition*. Le message est envoyé suite à une action de déclenchement de la transition.
- *Un objet*. Le message est envoyé par un objet d'une classe à des points du diagramme. Le message est reçu par l'objet et peut déclencher une transition sur l'événement correspondant. Il peut y avoir plusieurs transitions impliquant l'événement en question. Cette notation ne doit pas être utilisée quand l'objet récepteur est déterminé dynamiquement, mais dans ce cas une expression (commentaire textuel) doit être utilisée.
- *Une transition*. La transition doit être l'unique transition de l'objet qui implique un événement donné, ou du moins l'unique transition qui peut être déclenchée par le message. Cette notation ne doit pas être utilisée quant une transition choisie dépend aussi de l'état de l'objet recevant est pas uniquement de l'émetteur.

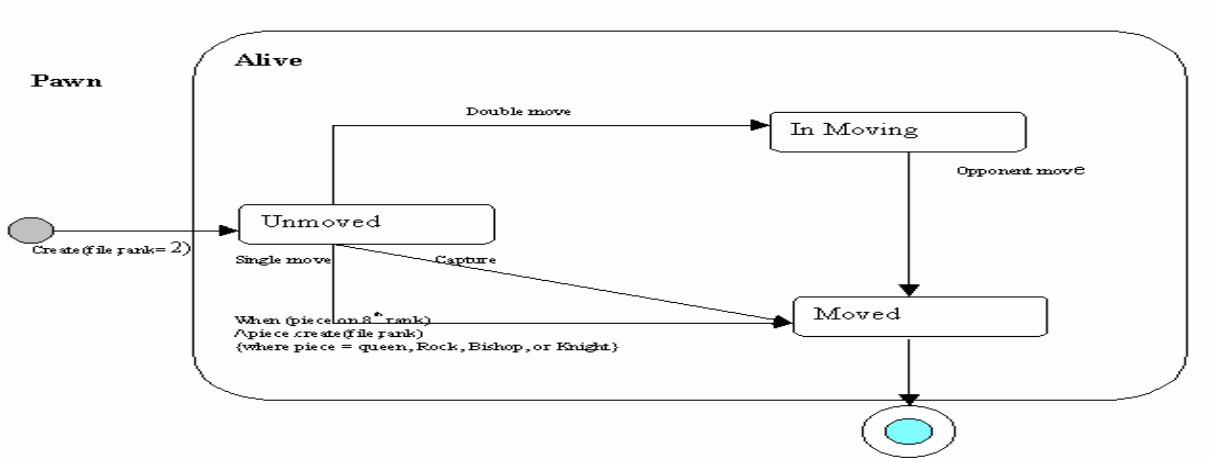


Figure 8.4 : Création et destruction d'objets

La désignation de classe. Cette notation pourrait être utilisée pour modéliser les invocations des opérations des classes, comme par exemple la création d'une nouvelle instance. La réception d'un tel message peut être utilisé pour déclencher le passage d'une transition, ce qui constitue un moyen pour faire passer les informations du créateur vers le nouvel objet ainsi obtenu.

8.3.5 Diagrammes de Collaboration

Un diagramme de collaboration (Figure 8.5) montre clairement une interaction organisée autour des objets et leurs liens. Comme dans un diagramme de séquence, le diagramme de collaboration permet de tisser les relations entre objets, sans tenir compte du temps comme une dimension les séparant, cependant les séquences de messages et les threads concurrents doivent être déterminés en utilisant une séquence de nombres.

Le comportement d'un diagramme de collaboration est implémenté grâce à des ensembles d'objets échangeant des messages qui interagissant pour accomplir un but. Pour comprendre les mécanismes de fonctionnement des diagrammes de collaboration, il est important de ne voir que les objets et les messages qui s'associent pour accomplir un objectif tracé par le système dans lequel ils sont partie prenante.

Une collaboration étant un ensemble de participants et des relations ayant une signification pour accomplir un objectif, ce qui fait que l'identification des participants n'a pas une grande importance.

Une collaboration doit être liée à une opération ou un *use case* pour décrire le contexte dans lequel leur comportement survient. Le comportement actuel peut être spécifié par des interactions, tels les diagrammes de séquences ou les diagrammes de collaboration. Une collaboration peut aussi être reliée à une classe pour définir la structure statique de celle-ci.

Une collaboration paramétrée représente un design qui peut être utilisé de manière itérative dans différentes architectures. Les participants dans une collaboration hormis les classes et leurs relations, mais aussi les paramètres d'une collaboration générique. Une collaboration peut être exprimée à des niveaux différents de granularité. Une collaboration au sens large peut être raffinée pour produire une autre collaboration à granularité plus fine.

La description du comportement d'une collaboration conduit à considérer deux aspects : la description *structurelle* de ces participants et la description de *comportement* de ses exécutions. Les deux aspects sont souvent décrits ensemble dans un même diagramme mais à des périodes utiles à leur description séparée. La structure des objets jouant un rôle dans ce comportement et leurs relations est appelée *collaboration*, et cette collaboration montre clairement le contexte dans lequel cette interaction se produit. Le comportement dynamique des séquences de messages échangés par les objets est appelé *interaction*.

Cependant, un diagramme de collaboration peut aussi être défini comme un graphe qui fait référence aux objets et leurs liens avec les flots de messages. La collaboration est utilisée pour décrire une opération associant ses arguments et les variables locales crées durant une exécution aussi bien que les associations ordinaires.

Les objets crées durant une exécution (Figure 8.4) sont désignés par {*new*}; les objets détruits durant cette exécution sont désignés par {*detroyed*}. Par contre ceux crées puis détruits durant une exécution portent désormais le label {*transient*}. Ces changements dans un cycle sont dérivables à partir des messages envoyés par les objets.

Le diagramme montre aussi les liens entre les objets, incluant les liens représentant les arguments, les variables locales, et les *self* links.

Les valeurs des attributs individuels ne sont en général pas montré explicitement. Si les messages doivent être envoyés à des valeurs d'attributs, les attributs doivent être dans ce sens modélisés en utilisant des associations.

Un diagramme de collaboration dépourvu de messages doit montrer clairement le contexte dans lequel les interactions surviennent.

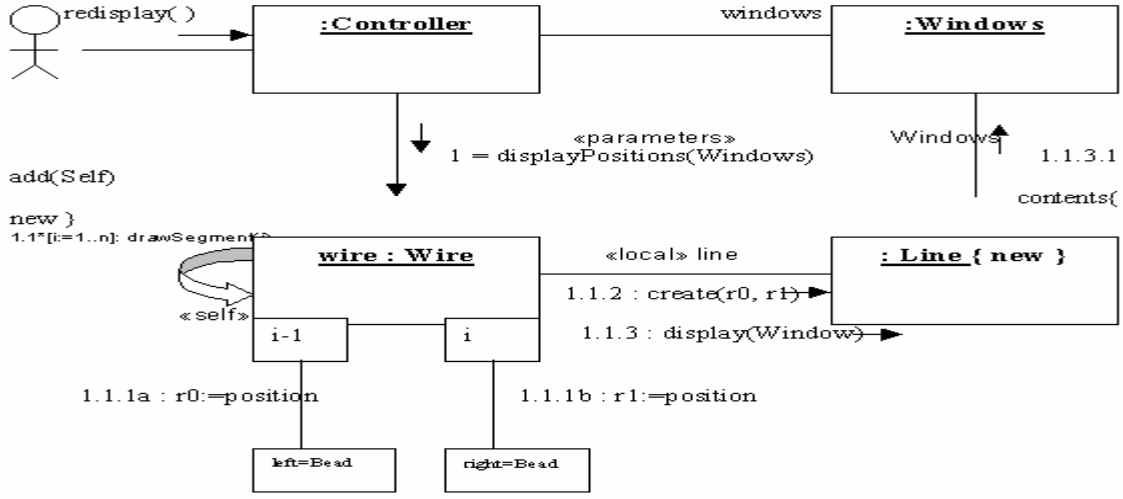


Figure 8.5 : diagramme de Collaboration

8.3.6 Diagramme de Statechart (Diagramme de transitions)

Un diagramme statechart [Har87, Har88, ABD98] (Figure 8.6) représente une machine d'états. Les états sont représentés par des symboles d'états et les transitions sont représentées par des flèches, faisant la connexion des états. Les états peuvent contenir des sous diagrammes.

8.3.6.1 Etats

Un *état* dans ce sens est une condition établie durant le cycle de vie d'un objet, mais aussi, il peut être considéré comme une interaction durant lequel il (l'état) satisfait certaines conditions, exécute une action ou bien il attend un évènement. Un objet par convention ne doit rester dans un état que pendant un temps fini (non-instantané).

Les *actions* sont atomiques. Un état peut correspondre à une activité qui se met en route. Cette activité est représentée par une paire d'actions, l'une qui déclenche l'activité sur l'état et l'autre qui termine l'activité en sortie de l'état.

Chacune des sous régions d'un état peut contenir des états initiaux et des états finaux. Une transition incidente à l'intérieur d'un état est considérée comme une transition qui concoure vers un état initial. Une transition vers un état final représente l'achèvement de l'activité dans une région enfermée, l'achèvement de l'activité dans toutes les régions concurrentes représente l'achèvement de l'activité

par l'état enfermé et provoque un évènement particulier qui est « achèvement de l'activité » dans cet état (enfermé). L'achèvement d'un état qui se trouve à l'extérieur constitue le signe que l'objet en question est mort.

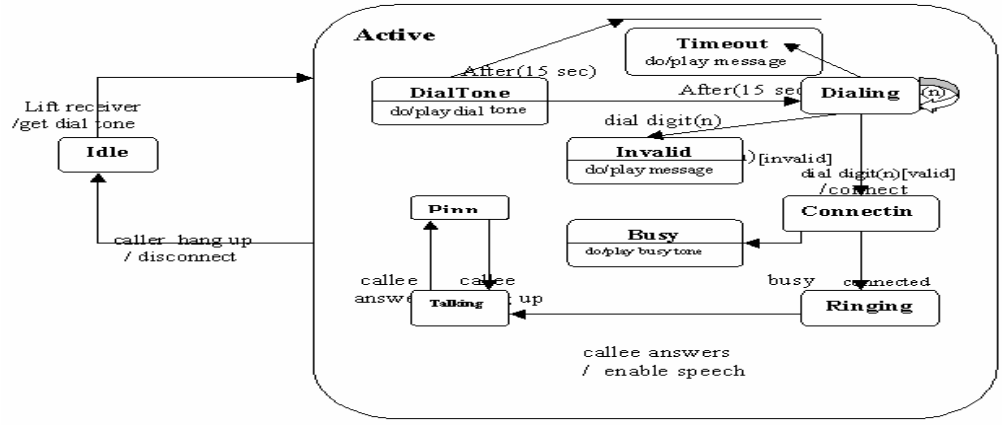


Figure 8.6 : Un Statechart correspondant à un Systéme d'appel Téléphonoque tiré de la litérature.

Un état est matérialisé graphiquement par un rectangle aux coins arrondis. Il peut contenir un ou plusieurs compartiments qui sont d'ailleurs optionnels. Chaque compartiment doit posséder :

- Un nom de compartiment. Une chaîne de caractères. Les états dépourvus de nom sont considérés comme quelconques "anonymous" et doivent être distincts. Il est indésirable de rencontrer le même nom deux fois et plus, ceci risque de causer la confusion.

- Transition interne à un compartiment. Tiens dans une liste d'actions internes ou d'activités exécutées en réponse aux réponses reçus, le moment ou l'objet était dans l'état en question, sans changement d'état. Cette transition est déclarée dans le format suivant :

event-name argument-list '[' guard-condition ']''/' *action-expression.*

- Chaque nom d'évènement ou de pseudo-évènement peut apparaître au plus une seule fois dans un état singulier.

Les actions spéciales suivantes possèdent la même forme mais représentent les mots réservés qui ne peuvent pas être utilisés comme noms d'évènements :

- 'entry' '/' *expressio*n d'*action* Une action atomique qui s'exécute à l'entrée de l' état
- 'exit' '/' *expressio*n d'*action* Une action atomique qui s'exécute en sortie de l'état

Les actions e*ntry* et *exit* ne doivent pas contenir dans leur corps des arguments ni des conditions guard (tout simplement parce qu'ils sont invoqués implicitement et non explicitement). Cependant, l'action entry au top niveau de la machine d'états doit avoir des paramètres qui représentent les arguments qu'elle reçoit au moment de sa création.

Les expressions de type action, peuvent utiliser les attributs et les liens de l'objet et des paramètres qu'ils possèdent des transitions incidentes intérieurement (s'ils apparaissent dans toutes les transitions incidentes intérieurement).

Le mot-clé suivant représente l'invocation d'une machine d'états (incorporée)

'do' '/' *machine-name (argument-lis*t)

Exemple 8.1

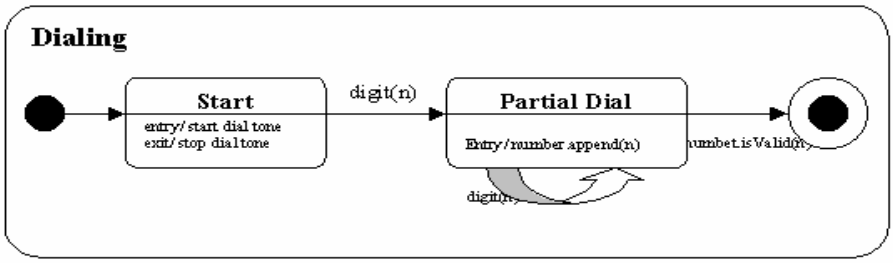


Figure 8.7 : Sous-états Séquentiels

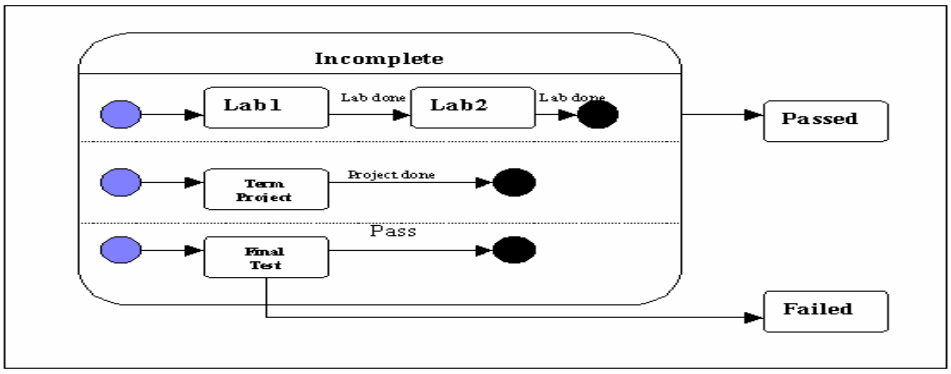


Figure 8.8 : Sous-états Concurrents dans un Statechart

8.3.6.2 Etats concurrents et barre de synchronisation

Pour représenter des états concurrents sur un même diagramme d'états-transitions, on utilise la notation un symbole spécial : "**la barre de synchronisation**". La barre de synchronisation permet de représenter graphiquement des points de synchronisation. Les transitions automatiques qui partent d'une barre de synchronisation ont lieu en même temps. On ne franchit une barre de synchronisation qu'après réalisation de toutes les transitions qui s'y rattachent

Exemple 8.2

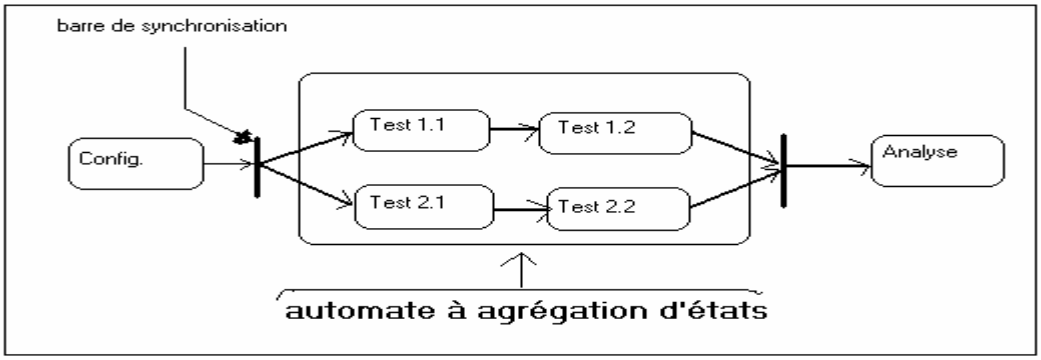


Figure 8.9 : Expression des Barres de Synchronisations dans un Statechart

8.3.7 Diagramme d'Activité

Un diagramme d'activité (Figure 8.10) est un cas spécial d'un diagramme d'états dans lequel tous les états (ou du moins un certain nombre) du système sont des états-actions et dans lequel toutes les transitions (ou du moins un certain nombre) peuvent être tirées par complétude des actions dans les états sources. Le diagramme d'activité est dans ce cas attaché à une classe ou à l'implémentation d'une

opération ou d'un cas d'utilisation (use case). La proposition d'un tel diagramme étant de se focaliser sur les flux (flows) internes. L'exemple suivant nous donne une idée sur la possibilité d'écrire des diagrammes d'activités qui soient très expressifs.

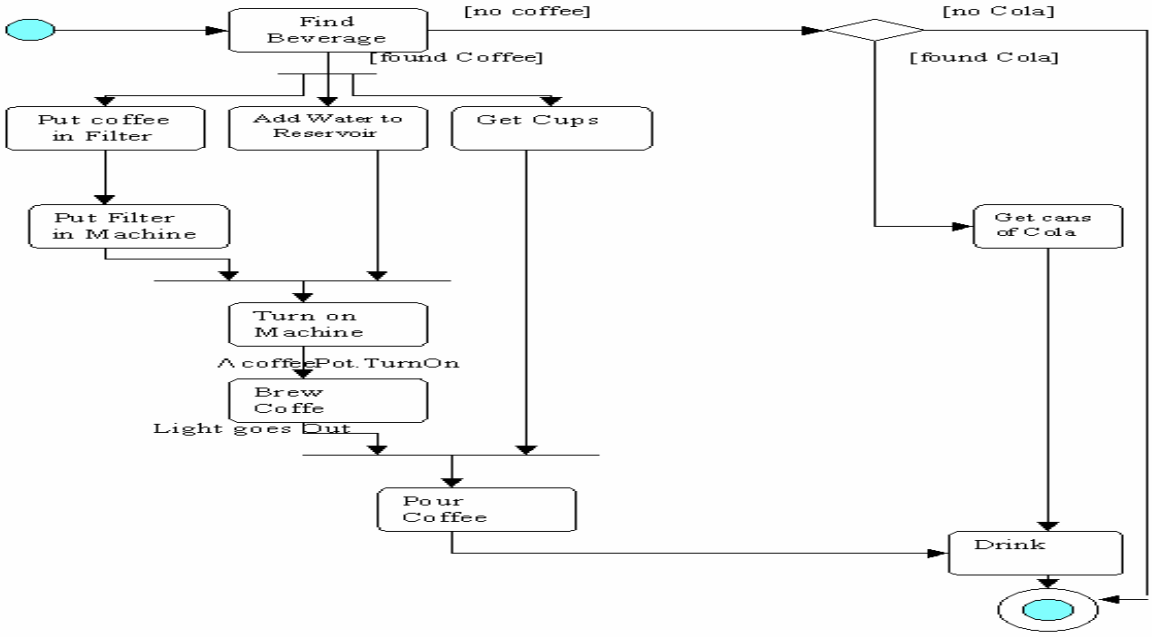


Figure 8.10 : Un Diagramme d'Activité

8.4 Vue Générale Sur le Langage OCL

Dans cette partie, nous introduisons Object Constraint Language (OCL) [WK99, OMG01, JAR98, RG00] un langage formel utilisé pour spécifier les contraintes ainsi que d'autres expressions associées aux modèles UML.

Pour être plus précis, OCL est utilisé pour exprimer les règles bien-formées du metamodel UML. Chaque règle bien-formée contient une expression OCL, qui est en réalité un invariant. Naturellement OCL possède une grammaire consignée dans les manuels de l'OMG.

8.4.1 POURQUOI OCL?

Dans la modélisation orientée-objet, un model graphique, orienté classe n'est pas assez précis pour décrire tous les éléments qui lui sont rattachés. Dans ce cas, nous avons besoin de concepts pour décrire les contraintes additionnelles qui sont en général, décrites dans le langage naturel (l' Anglais dans la plus part des cas). Devant une multitude de cas (modélisation), l'expérience a montré qu'un tel procédé génère en général des spécifications ambiguës.

Le langage OCL a été développé depuis la version 1.3 de l'ML est utilisé comme un langage de modélisation des processus adaptés au business (affaires) par IBM Insurance division justement pour éviter les problèmes liés à l'inconsistance et à l'ambiguïté des modèles développés. C'est un langage qui s'écrit dans une syntaxe formelle tout à fait simple. Une expression écrite dans le langage OCL ne peut en aucun cas influencer le système (changement d'état), mais elle peut être utilisée expressément pour un changement de l'état du système, mais dans ce cas en utilisant une primitive de post-condition. Quand une expression OCL est évaluée, elle délivre tout simplement une valeur. Nous notons qu'OCL n'est pas du tout un langage de programmation dans le sens ou aucune logique ne ferra suite à une succession de directive dans le sens d'obtenir des réponses logiques pour matérialiser le fonctionnement d'un processus. D'ailleurs OCL n'est même pas adapté pour le contrôle des flux de données. OCL est

un langage typé dans la mesure ou toute expression possède un type et donc chaque expression est par conséquent conceptuellement atomique.

8.4.2 Dans quel cas utiliser OCL

OCL peut être utilisé pour les besoins suivants :

- Pour spécifier les invariants sur les classes et sur les types dans une classe.
- Pour spécifier les invariants typés sur les stéréotypes.
- Pour décrire les pré- et post-conditions sur les opérations et sur les méthodes.
- Pour décrire les Gardes.
- Comme un langage de navigation
- Pour spécifier les contraintes sur les opérations:

8.4.3 Invariants

L'expression OCL peut être considérée comme un invariant lorsque c'est une contrainte stéréotypée avec «invariant». Cette expression dans ce cas doit être vraie pour toutes les instances du type concerné et à tout moment.

Dans l'expression,

```
self.numberOfEmployees
```

self est une instance de type compagnie. Nous pouvons voir que self se comporte comme un objet à partir duquel nous pouvons commencer l'expression.

Dans ce document, le type de l'instance contextuelle de l'expression OCL qui fait partie de l'invariant, est écrite avec le nom du tuple souligné comme suit :

```
Company
self.numberOfEmployees
```

Dans la plupart des cas l'écriture de *self* peut être remplacée par un nom de variable. Dans l'exemple suivant, il est remplacé par le nom c.

```
c : Company
c.numberOfEmployees
```

Ces deux expression sont tout à fait équivalentes.

8.4.3.1 PRE- et POSTCONDITIONS

Une expression OCL peut aussi exprimer une Pré ou Post-condition, qui n'est d'autre qu'une contrainte stéréotypée par les expressions «précondition» et «postcondition». La notation de l'écriture d'une telle expression OCL s'effectue en posant les labels « pre » et « post » des Précondion et Postcondition comme suit :

PostconditionsTypename::operationName(parameter1 : Type1, ...): ReturnType

```
pre : parameter1 > …
post: result = ...
```

Exemple :

```
Person::income(d : Date) : Integer
post: result = ...some function of self and parameter1 ...
```

8.4.4 Expressions Générales

Certaines expressions OCL peuvent être utilisées comme des valeurs des attributs de classes ou l'un de ces sous-types.

8.4.4.1 Valeurs de Base et Types

Le langage OCL admet les types de valeurs et des opérateurs selon ce qui suit :

type	**values (example)**
Boolean	true, false
Integer	1, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be...'

type	**operations**
Integer	*, +, -, /, abs
Real	*, +, -, /, floor
Boolean	and, or, xor, not, implies, if-then-else
String	toUpper, concat

Chaque expression OCL est écrite dans le contexte du model UML, un nombre de types/classes leurs caractéristiques et leurs généralisations. Toutes les types/classes dans UML sont considérés comme des types dans OCL.

8.4.4.2 Types et Enumération

Les types énumération peuvent être définis en utilisant l'en tête enum comme :

```
enum{value1, value2, value3}
```

Les valeurs de l'énumération *(value1, ...*) peuvent être utilisées avec des expressions. L'utilisation d'une valeur de type énumération doit être préfixée par le symbole # . Exemple :

```
#value1
```

Le type d'un attribut énumération est Enumération, avec restriction sur les valeurs de l'attribut.

8.4.4.3 Conformation

OCL est un langage typé ce qui fait que les types de bases sont organisés selon une hiérarchie, ce qui assure une certaine conformation des différents type vis-à-vis les uns des autres. Par exemple, il n'est pas possible de comparer un integer avec un booléen ou avec les *string*. Les règles qui assurent cette conformation sont très simples selon :

- Chaque type doit être conforme à son supertype.
- Le type de conformation est transitive: if *type1 est conforme au type*2, and *type2 est* conforme au *type*3, alors *type1 est* conforme au *type3.*

Example :

OCL expression	**valid?**	**error**
1 + 2 * 34	yes	

1 + 'motorcycle'	no	type Integer n'est pas conforme au type String
23 * false	no	type Integer n'est pas conforme au type Boolean
12 + 13.5	yes	

8.4.4.4 Propriétés: Attributs

Exemple, l'age d'une Personne est écrit comme,

```
Person
self.age
```

La valeur de cette expression est la valeur de l'attribut age à Person *sel*f. Le type de cette expression est le type de l'attribut age qui est un type de base « Integer ».

Donc, avec les attributs et les opérations définies sur des valeurs de type de base, nous pouvons ainsi réaliser des calculs. Par exemple "the age of a Person is always greater or equal to zero." Ceci s'écrira comme un invariant :

```
Person
self.age >= 0
```

8.4.4.5 Propriétés : Opérations

Dans le langage OCL, les opérations peuvent avoir des paramètres. Par exemple, un objet Person possède un revenu, ceci s'exprime par la fonction (date) :

```
aPerson.income(aDate)
```

L'opération elle-même peut être définie par une contrainte post-condition. L'objet retourné par l'opération est referé par le label *result*. Il prend la forme suivante :

```
Person::income (d: Date) : Integer
post: result = -- some function of d and other properties of person
```

Une opération ou une méthode qui ne prend pas d'argument peut simplement être déclarée avec des parenthèses vide.

Exemple :

```
Company
self.stockPrice()
```

8.4.4.6 Propriétés : Fins d'Association

A partir d'un objet spécifique, nous pouvons adjoindre une association sur le diagramme des classes pour se référer à d'autres objets et leurs propriétés. Dans ce cas, considérons l'association en utilisant l'association opposée (association-end) :

```
object.rolename
```

La valeur de cette expression est l'ensemble des objets sur l'autre coté de l'association rolename. Si la multiplicité de l'association possède un maximum de 1 ("0..1" or "1"), alors la valeur de cette expression est un objet. Par exemple (i.e. *self* est une instance de Company), nous pouvons écrire :

```
Company
```

```
self.manager -- is of type Person
self.employee -- is of type Set(Person)
```

L'évaluation de la première expression résultera dans un objet de type person, parce que l'association de la multiplicité est un. L'évaluation de la deuxième expression, résultera dans l'ensemble des personnes.

Les collections de type ensemble, Bags et Séquences sont tous prédéfinis Type dans le langage OCL. Une propriété de collection elle-même est accessible en utilisant la flèche « ➔ ». Exemple :

person:

```
Person
self.employer->size
```

Cette écriture définit la taille de la propriété sur l'ensemble *self.employe*r, qui est le nombre des employés de Person *sel*f.

```
Person
self.employer->isEmpty
```

L'évaluation de cette expression est "true" si l'ensemble des employés est vide (empty), sinon "false"

8.4.4.7 Combinaison de Propriétés

Les propriétés peuvent être combinées les unes aux autres pour construire des expressions plus compliquées.

Les expressions suivantes définissent des invariants qui utilisent des propriétés combines :

[1] Married people are of age >= 18

```
self.wife->notEmpty implies self.wife.age >= 18 and
self.husband->notEmpty implies self.husband.age >= 18
```

[2] a company has at most 50 employees

```
self.employee->size <= 50
```

[3] A marriage is between a female (wife) and male (husband)

```
self.wife.sex = #female and
self.husband.sex = #male
```

[4] A person can not both have a wife and a husband

```
not ((self.wife->size = 1) and (self.husband->size = 1))
```

8.4.5 Caractéristiques Prédéfinies sur Tous les Objets

Il existe plusieurs caractéristiques appliquées à tous les objets et qui sont prédéfinies dans OCL, à savoir :

```
oclType : OclType
oclIsTypeOf(t : OclType) : boolean
oclIsKindOf(t : OclType) : boolean
```

L*' oclType* redéclare le type d'un objet. Par exemple, l'expression

```
Person
self.oclType
```

L'opération *isTypeOf* fournie true si le type de self if the *type* of self et *t sont équivalent*. Par exemple:

```
Person
self.oclIsTypeOf( Person ) -- is true
self.oclIsTypeOf( Company) -- is false
```

8.4.6 Collections

Dans le langage OCL, il est possible de définir des Collections qui est un type abstrait de données, à partit duquel nous pouvons définir un sous type qui est du type collection mais concret. OCL fait la distinction entre trois types de collections qui sont :

- *set* : ensemble au sens mathématique, mais ne doit pas contenir des éléments dupliqués,
- *sequence* (séquences = chaine) et
- *bag* (sac) : c'est exactement la même définition que bag seulement les éléments doivent être ordonnés.

Exemple 8.3

Ensembles

```
Set { 1 , 2 , 5 , 88 }
Set { 'apple' , 'orange', 'strawberry' }
```

Sequence

```
Sequence { 1, 3, 45, 2, 3 }
Sequence { 'ape', 'nut' }
```

Les séquences suivantes définissent une séquence de nombres entiers compris entre l'élément devant et après les deux points successifs « .. ». Ce qui veut dire que les deux expressions suivantes ont même valeurs :

```
Sequence{ 1..(6 + 4) }
Sequence{ 1..10 }
```

et ils sont identiques à l'expression : `Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }`

Bag

```
Bag {1 , 3 , 4, 3, 5 }
```

Exemple 8.4

```
Company
self.employee
collection1->union(collection2)
```

Il est aussi admis d'imbriquer les collections entre elles, par exemple :

```
Set{ Set{1, 2}, Set{3, 4}, Set{5, 6} }
Set{ 1, 2, 3, 4, 5, 6 }
```

Donc ces deux expressions sont équivalentes.

Nous admettons d'une façon générale que :

- Tout type de Collection (X) est un sous-type de OclAny. Les types Set (X), Bag (X) and Sequence (X) sont tous des sous-type de Collection (X).

Les règles de conformation des types sont définies comme suit :

- *Type1 est conforme avec Type2 lorsqu'ils sont identiques* (rule standard pour tous les types).
- *Type1 est conforme avec Type2 lorsque c'est un sous-type de Type2* (rule standard pour tous les types).
- *Collection (Type1) est conforme à Collection(Type2*), lorsque *Type1 est conforme avec Type*2.
- La conformation des Types est transitive: if *Type1* conforms to *Type*2, and *Type2* conforms to *Type*3, then *Type1* conforms to *Type3* (rule standard pour tous les types).

Par exemple, si *Bicycle et Car sont deux sous-type différents de Transpor*t, nous pouvons définir :

Set(Bicycle) est conforme à Set(Transport)

Set(Bicycle) est conforme à Collection(Bicycle)

Set(Bicycle) est conforme à Collection(Transport)

Notons que Set(Bicycle) n'est pas conforme à Bag(Bicycle).Ils sont tous les deux des sous-types de Collection(Bicycle) au même niveau de hiérarchie.

8.4.7 Valeurs Précédentes dans Les Postconditions

Nous savons qu'il est possible de spécifier pre- et post-conditions sur les opérations et les méthodes en UML/OCL. Dans une postcondition, l'expression peut se référera à deux ensembles de valeurs pour chaque propriété d'un objet:

- La valeur d'une propriété au début d'une opération ou d'une méthode.
- La valeur d'une propriété après l'achèvement de l'exécution d'une opération ou d'une méthode.

Pour se referez à la valeur d'une propriété dans au début d'une opération, il suffit de postfixer le nom de la propriété avec le signe commercial '@', suivit du mot clé 'pre':

```
Person::birthdayHappens()
post: age = age@pre + 1
```

Dans cet exemple, la propriété age fait référence à la propriété de l'instance de Person sur laquelle on effectue l'exécution. La propriété *age@pre fait référence à la valeur de la propriété age de Person* qui exécute 'opération au début de l'opération.

Si la propriété possède des paramètres, '@pre' est postfixé à propertyname, avant les paramètres.

```
Company::hireEmployee(p : Person)
post: employees = employees@pre->including(p) and
stockprice() = stockprice@pre() + 10
```

L'opération précédente peut aussi être spécifiée par une post- et pre-condition écrite ensemble :

```
Company::hireEmployee(p : Person)
pre : not employee->includes(p)
post: employees->includes(p) and
stockprice() = stockprice@pre() + 10
```

Quand une pré-value est une propriété qui évalue un objet, toutes les propriétés qui accèdent à cet objet sont les nouvelles valeurs à la fin de l'exécution de l'opération de cet objet, alors :

```
a.b@pre.c -- prend l'ancienne valeur de la propriété b de a, disons x, puis la
nouvelle valeur de c de x.

a.b@pre.c@pre -- prend l'ancienne valeur de la propriété b de a, disons x et puis
l'ancienne valeur de c de x.
```

Nous avons défini jusque là une partie très importante du langage OCL, sans pour cela introduire toutes les opérations admissibles qui nous auraient permis de manipuler à titre d'exemple les Collections. Nous invitons le lecteur à consulter des ouvrages spécialisés, entre autres la version 1999 de UML/OCL syntaxe et sémantique [OMG99a] et la version 2001 [OMG01].

Dans la suite de notre exposé, nous allons nous atteler à concrétiser notre point de vue sur l'utilisation du langage UML/OCL à travers un exemple de système réactif que nous avons déjà cité auparavant. Il s'agit du système Train-gate-controller (passage à niveau).

8.5 Modélisation du système Train-Gate-Controller en UML/OCL

La description des propriétés du système, est une activité décisive pour assurer qu'un produit sera correct dans le future. La consistance du système dépend de l'habilité des concepteurs pour mettre en place toutes les propriétés qui définissent le comportement du système avec un risque zéro.

Cependant, une description détaillée du système TGC est donnée dans [EN97, JS00, Be96]. Cette description inclus la connaissance du domaine comme un élément très important pour la spécification formelle du système. Le système TGC, considérée comme une application répartie en trois sous-systèmes : le train qui est un système qui se déplace sur des rails d'un point (disons) de départ vers un autre point (disons) d'arrivée de façon continue. Donc, nous considérons que le train contient en son sein un dispositif de communication qui peut envoyer des messages renseignant sur sa position sur un tronçon de rail. Un contrôleur automatique utilisé pour manipuler une barrière automatique lui donnant des ordres de mouvement d'une position précise (position de repos) vers le haut, et dans le cas opposé un ordre de mouvement vers le bas lui est intimé vers la position de repos. Le système contrôleur-barrière se trouve à proximité de toute intersection d'une route usagée pour véhicule et une autre (rails) pour les trains. La zone d'intersection est appelée la zone dangereuse, qu'on appellera aussi la zone de passage. Le troisième sous-système étant un système de contrôle centralisé.

8.5.1 L'approche de modélisation par le langage UML

Donc, selon ce qui vient d'être cité, le système étant constitué de trois sous-systèmes qui communiquent entre eux. Le système complet est décrit par le diagramme de classe de la Figure 8.12. Pour ne pas trop tergiverser sur le fonctionnement du système complet qui est formé de trois sous-systèmes, une attention particulière est donnée dans la Figure 8.11 et 8.12. Le lecteur ne trouvera aucun difficulté à comprendre l'architecture mais aussi le fonctionnement c'est-à-dire son comportement. Nous comprendrons mieux tous les aboutissants à travers les règles OCL

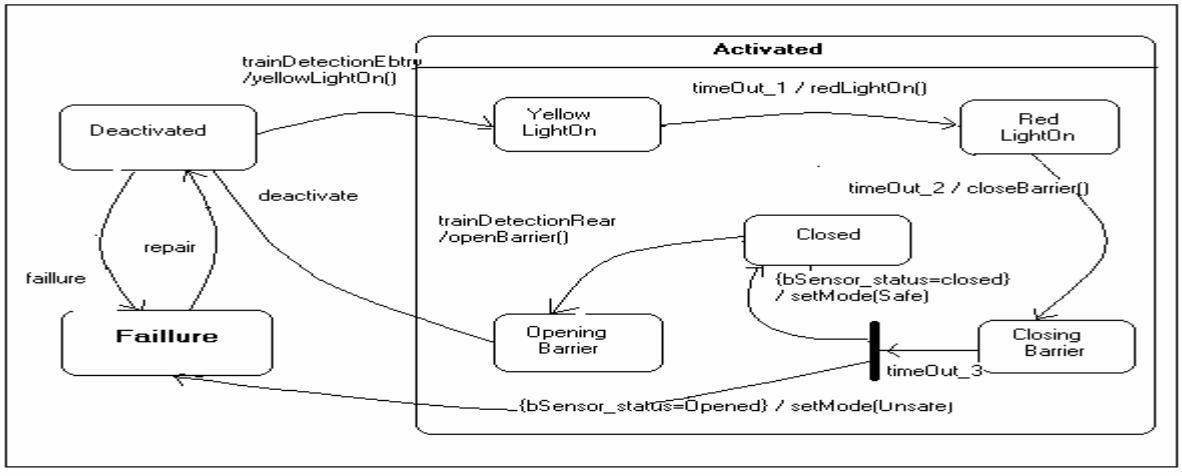


Figure 8.11 : Diagramme d'état du TGC

8.5.2 Spécification de Contraintes en OCL

L'utilisation d'un langage unifié et standardisé permet d'outre passer toutes les divergences au sein d'un groupe de concepteurs qui en général possède des connaissances diverses touchant pratiquement à plusieurs disciplines. Comme nous l'avons vu, l'idée de l'utilisation du langage OCL, permet d'associer des contraintes aux objets d'un système pour qu'il soit spécifié formellement, préservant toutes les caractéristiques structurelles et dynamiques du système. Cette manière de représentation permet ainsi d'associer le formel avec le graphique, ce qui à coup sûr permet d'éviter l'incompréhension, l'ambiguïté et ainsi assuré l'exactitude de la spécification, et de traiter des problèmes très complexes sans risque majeur. Nous avons vu, qu'OCL met en place des mécanismes d'écriture assurant la référence aux éléments des classes, à la notion de typage, à la représentation de contraintes sous forme de pre et post-conditions, à l'expression de caractéristiques du temps etc.

Dans ce qui suit, nous montrons certaines recommandations du langage OCL, pour spécifier un certain nombre de propriétés de notre système, à savoir le TGC (train-gate_controller). Nous commencerons par exemple par les propriétés qui expriment la sûreté du système du moins certains comportements du système (Safety).

Comme la propriété la plus importante du système étant de préserver le trafic routier d'un côté et de permettre au train de passer sur la zone dangereuse (l'intersection) sans encombre, cependant à un niveau d'abstraction (élevé) de la spécification de control, il est tout à fait suffisant de modéliser la zone du passage à niveau et la barrière automatique et ceci à tout instant.

Les contraintes OCL suivantes (invariants) spécifient ce qui suit (voir Figure 8.15) :

1. S'il existe un train traversant la zone d'intersection alors la barrière doit être fermée :

```
context CrossingArea inv:
not(self.train ->isEmpty()) implies self.barrier.state=Closed
```

2. Si la barrière automatique est ouverte alors aucun train ne s'est approché de la zone dangereuse :

```
context Barrier inv:
self.state=Opened implies self.guards.train ->isEmpty()
```

3. La barrière automatique doit être fermée lorsqu'un train passe sur la zone dangereuse :

```
context PhysicalTrain inv:
self.crosses.barrier=Closed
```

Ces contraintes doivent être respectées (vraie). Pour le moment il n'est pas nécessaire d'expliquer physiquement comment l'avant des trains et leurs queues sont détectées pour connaître les positions de certaines variables du modèle de spécification soient vrais. Ceci n' a pas une très grande importance pour le moment, il suffit pour le moment que le train s'approche de la zone, qu'il soit dans la zone ou bien qu'il soit sorti de la zone.

La spécification des positions (états) de la barrière peut être décrite par les contraintes OCL suivantes (Figure 8.12):

4. Le feu rouge est allumé (switched on) chaque fois que la barrière est fermée et que le feu jaune soit allumé lorsque la barrière se ferme. Si le feu rouge et le feu jaune sont éteints alors la barrière est ouverte:

```
context TGC System inv:
self.theBarrier.state=Closed implies self.redLight.state=On
and
self.theBarrier.state=Closing implies
self.yellowLight.state=On and
self.yellowLight.state=Off and self.redLight.state=Off
```

```
implies self.theBarrier.state=Opened
```

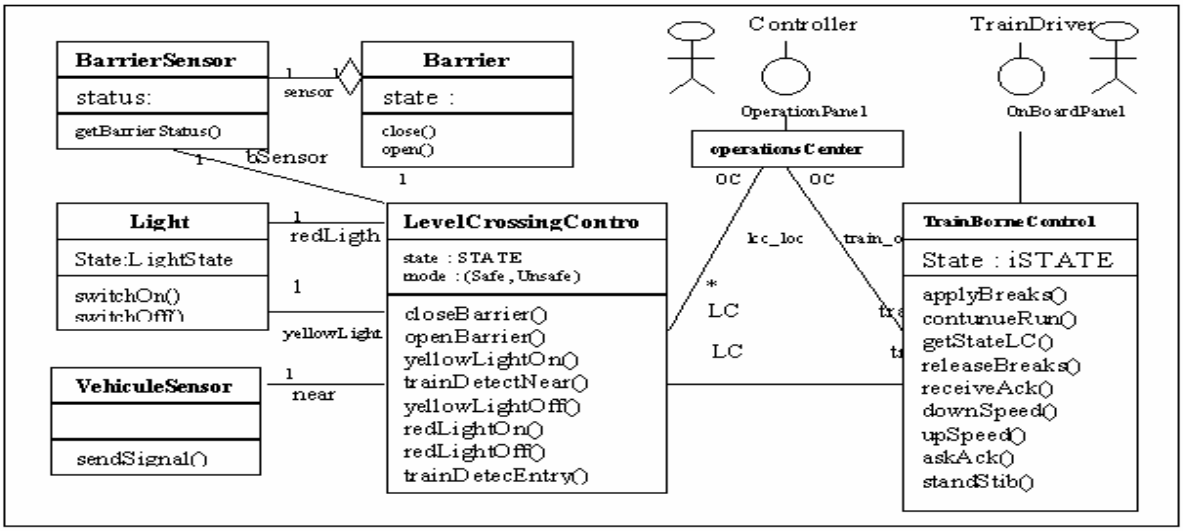


Figure 8.12 : Diagramme de Class du TGC

5. Lorqu'il reste encore un train dans la zone dangereuse, l'état level crossing est en position activated state. La variable *activated state* est composée de trois sous-états (WaitingAck, Closing, Closed, Opening) :

```
context TGC System inv:
not(self.train ->isEmpty()) implies self.state=Activated
and
Set(Activated)=Set(WaitingAck->Union(Closing)->Union(Closed)->
Union(Opening))
```

6. Lorsque le système TGC est positioné à *activated state* et la barrière soit ouverte alors l'état *level crossing* est en mode *unsafe* :

```
context TGC System inv:
self.state=Activated and self.bSensor.state=Opened
implies self.mode=Unsafe
```

7. Si la barrière est fermée alors que le sensor indique que la barrière est ouverte alors l'état *the level crossing* est dans le mode unsafe. Ceci, ne se fait que si la barrière soit dans l'état « entrain de se fermer », dans ce cas le système reste *unsafe* jusqu'à ce que la barrière soit complètement fermée. :

```
context TGC System inv:
self.bSensor.state=Opened and self.theBarrier.state=Closed)
implies self.mode=Unsafe
```

Les opérations de la classe TGC sont spécifiées avec les pré- et post-conditions de l'OCL.

OCL est aussi utilisé pour la spécification des contraintes dans le diagramme de séquences pour l'écriture des pre-conditions et des invariants sur les opérations (diagramme de séquence *train approaching*). Par contre le diagramme des états est utilisé pour dériver une première spécification de chaque opération. Dans ce cas les contraintes OCL sont d'une importance capitale pour donner plus de précisions aux informations qui ne peuvent pas figurées sur le diagramme d'état.

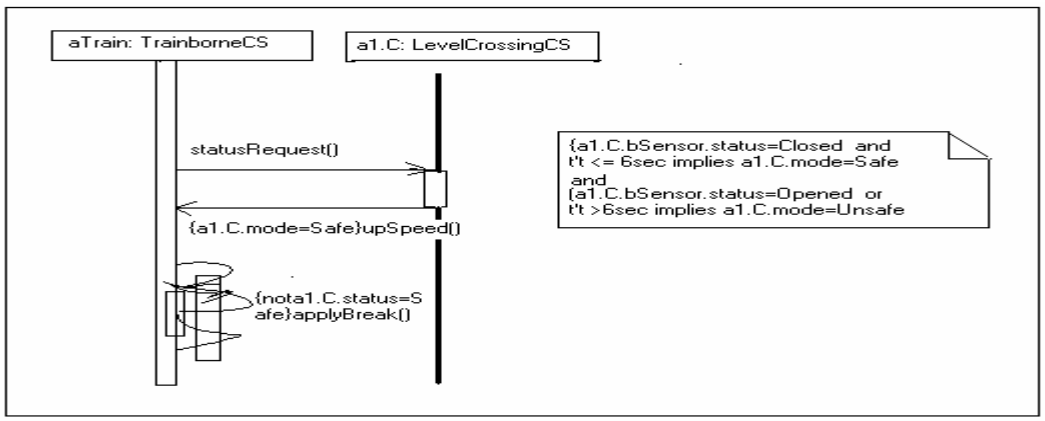


Figure 8.13 : Diagramme de Séquence du Sous-système Train

Considérons à titre d'exemple, la fermeture de la barrière actionnée par l'événement *timeOut 1*. La pré-condition de l'opération *closeBarrier* doit vérifier que le feu jaune soit allumé avant d'envoyer l'ordre de fermeture à la barrière. Elle a aussi à vérifier que la barrière n'est pas encore fermée. La post-condition assure que le feu jaune soit éteint, le feu rouge soit allumé et que la barrière soit fermée. Cette opération est spécifiée par :

```
context TGC System::closeBarrier
pre:
self.yellowLight.state=On and self.theBarrier.state=Opened
post:
self.yellowLight.state=Off and self.redLight.state=On and
self.theBarrier.state=Closed
```

La pré-condition de l'opération *openBarrier* qui est activée par l'événement trainDetectionRear vérifie que la barrière soit fermée et le système TGC est en mode *safe*.

La post-condition assure que la barrière est dans l'état ouvert "opened state":

```
context LCC System::openBarrier
pre: self.theBarrier.state=Closed and self.mode=Safe
post: self.theBarrier.state=Opened
```

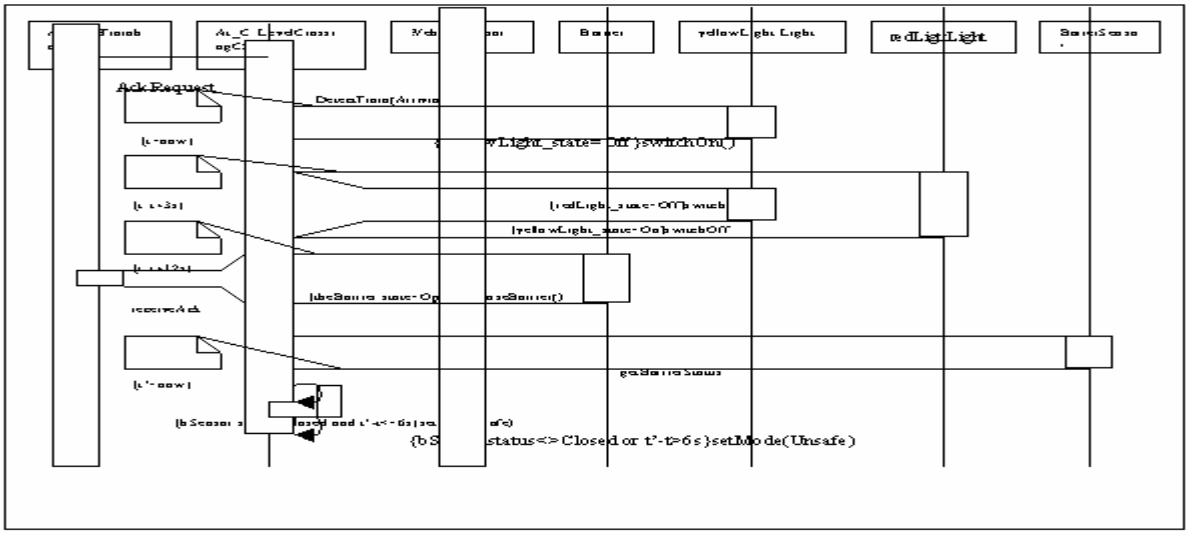


Figure 8.14 : Diagramme de Séquence : Scénario d'un train qui approche

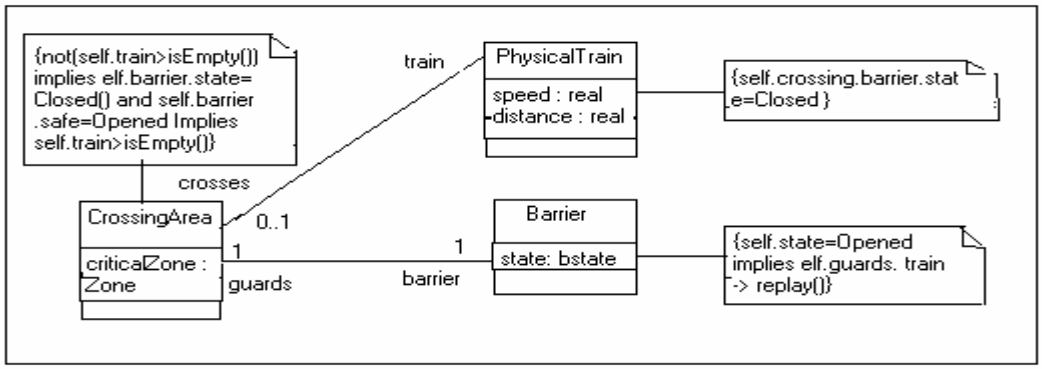


Figure 8.15 : Contraintes pour spécifier le passage à Niveau

8.6 L'abstraction des contraintes temporelles en UML/OCL

Plusieurs auteurs ont démontré [Ste03, GO03,RW99,GOO03, RKT+00] qu'OCL n'admet pas ou n'est pas capable dans sa version actuelle de modéliser les contraintes temporelles, sauf qu'à un degré moindre les outils mis à la disposition du concepteur ne traitent qu'une partie infime du langage UML, à savoir les messages. Les contraintes temporelles, même si celles-ci peuvent être exprimées dans le type de diagrammes suivants :

- State diagrams, ou elles apparaissent étiquetant les transitions.
- Sequence diagrams, ou les contraintes temporelles peuvent se baser sur le fait de choisir entre les observations temporisées ou temporelles.

Dans tous ces diagrammes les contraintes temporelles ont toutes la même signification et la même expressivité.

De nombreux chercheurs se sont intéressés à la normalisation d'UML, sa formalisation ou à son utilisation dans une approche mixte. Pour cette raison, dresser un état de l'art exhaustif de toutes les approches existantes serait illusoire.

En effet, pour la quasi-totalité des modèles formels présentés précédemment, il existe des travaux de couplage avec UML. Ainsi, des propositions sont disponibles pour : les langages Z [Led96], B [LM00, MLL00, Mar00], LOTOS [CLA00], des logiques temporelles [KNR02] des automates temporisés [RKT00] etc.

8.7 Traduction du langage UML vers MAUDE

Dans l'ordre d'exprimer le bien-fondé de traduire les modèles UML vers le langage MAUDE et ainsi obtenir une forme universelle exécutable, nous allons le démontrer à travers un exemple simple.

L'exemple en question représente, le modèle d'une bibliothèque. Pour bien expliquer notre idée, il s'agit de ne considérer qu'une partie aussi générale que possible et nous lessons le soin aux lecteurs de deviner les détails qui se trouvent consignés dans les travaux de F. Duran et al. [DV01] et [Reb05].

Dans un premier temps nous commencerons par une spécification informelle d'une bibliothèque, qui sera écrite dans le langage UML et par la suite elle sera raffinée et écrite dans le langage MAUDE.

Dans cette bibliothèque, nous identifions, quatre rôles, appelés communément, *borowers*, *library items*, *calendars* et *librarians*. Par mesure d'intégrité, nous préférons garder la dénomination de tous les éléments de ce modèle dans la langue de Shakespeare, mais ça n'empêche pas de les traduire, d'ailleurs la traduction est élémentaire, par exemple *borrowers* veut dire emprunteurs donc les lecteurs

d'une façon juste. *Labrary items*, ce sont les livres, les journaux et autres, *calendars* veut dire calendriers ou plus précisément dates (d'emprunt et de retour), *librarians,* il s'agit de l'agent libraire. Revenons maintenant à notre modèle. Il y a trois types de lecteurs (personnel académique, étudiants en graduation et étudiants en postgraduation, appelons-les : academic staff, undergrads et postgrads. Les lecteurs peuvent emprunter deux catégories d'articles *books* et *periodicals* (livres et periodiques). La communauté de la bibliothèque peut être vue comme étant composée d'un calendrier, d'un ou de plusieurs agents libraires, d'articles et des lecteurs. Un lecteur peut emprunter des livres et/ou des périodiques. Toutes ces informations sont consignées dans la Figure 8.16. Nous devons noter que chaque objet (information) remplie un et un seul rôle, quoique nous devons considérer par exemple, le staff académique ou les étudiants se comporter exactement comme les libraires, il suffit pour cela d'introduire des relations d'héritages en sus. L'utilisation des diagrammes de classes pour la description de la structure de la communauté, a été proposée par plusieurs auteurs [DV01]. Dans notre cas, nous lui avons apporté quelques modifications pour étayer notre point de vue publié dans [Reb05].

Nous notons que les attributs et les opérations de chaque classe définissent l'information de base de la spécification. Mis à part les attributs (title, author, et ISBN des livres et nom et adresses des lecteurs), d'autres attributs méritent quelques explications :

- La période de prêt prescrite et la limite qui est imposée sur le nombre d'article à emprunter à chaque emprunt qui est spécifié respectivement par les attributs *maxloans* et *loanPeriods* de la classe **Library**.

- Le lecteur qui ne retourne pas le ou les articles en sa possession au moment de la date de retour fixée doit être pénalisé. Le montant de cette pénalité ainsi que le nombre d'articles empruntés sont gardés (mémorisés) dans les attributs *fines* et *borrowedItems* de la classe **Borrower**.

- Les lecteurs peuvent être suspendus par l'agent libraire, s'ils ne payent pas l'amande en question, cette situation sera consignée par l'attribut *status* de la classe **Borrower** and *suspendedUsers* de la classe **Library**. La valeur de ces attributs doit être consistante, ce qui nécessitent une représentation d'une telle situation par l'utilisation de contrainte qui sera considérée comme un invariant.

- Les attributs status de la classe Item définissent l'état d'un article matérialisé par les valeurs possibles « on loan », « free », ou « disposed » au cas ou le livre à été détruit ou tout simplement perdu. L'attribut location de la classe item décrit l'emplacement sur les étagères de l'article correspondant quant il est disponible dans la bibliothèque.

- Finalement, nous définissons les opérations borrow, return et pour illustrer comment les opérations sont modélisées dans notre approche.

Supposons, maintenant que nous voulions décrire le modèle UML de notre cas d'étude (bibliothèque), c'est-à-dire de traduire cette spécification directement en Maude, la traduction se présentera comme suit :

Dans un premier temps, nous présenterons tous les acteurs intervenant dans la collaboration et nous commencerons par les lecteurs qui est une classe à part entière à savoir :

```
class reader |  code : Nat, name : string, address : string, status : ReaderStatus,
                    ReaderItem : Nat, fines : pointsNomber .
Including MACHINE-INT .
Including Qid .
Including CONFIGURATION .
sort Reader ReaderStatus .
subsort Reader < Cid .
ops suspended released : → ReaderStatus .
var Academic-staff  PostGrad-stud UnderGrad-stud :   Reader  .
```

var reader : → Reader .

var Rayan : Oid .

rl [*change_status*] <Rayan : reader | readerStatus : nil> <L : Librarian >(set_Suspended)

→

<Rayan : reader | reader_status : suspended><L : Librarian | suspendedUsers : reader_name> .

La règle *change_status* (ci-dessus) traduit le comportement concurrent des objets appartenant à la classe **reader** et ceux de la classe **Librarian**. L'interaction influencée par le message "set_suspended" permet de balancer l'état du reader de "nil" vers "Suspended" tout au début, et rajoute le nom du lecteur à la liste des lecteurs supendus.

Comme le lecteur lui appartient à interagir sur l'une des classes suivantes, ces derniers sont traduites en MAUDE comme suit :

class academic | dept : string, faculty : string .

class postGrad | dept : string, authoriser : string .

class undergrad | faculty : string, dept : string, tutor : string .

Les trois classes (academic, postGrad, undergrad) sont des sous-classes de la classe **reader** et par conséquent elles seront déclarées par :

subclasses academic postGrad undergrad < reader .

L'une des classes les plus importantes qui est d'ailleurs l'interface entre le lecteur (reader class) et les classes Book et Periodic étant la classe item. Ces trois classes seront traduites comme suit avec une petite différence c'est que les classes Book et Periodic sont considérées comme des sous-classes de la classe item :

class Item | location : string, status : ItemStatus, title : string .

class Book | author : string, ISBN : string, quantity : Nat, year : Nat, code : string,

observation : string .

class Periodical | ISSN : string, date : Date, number : Nat .

subclasses Book Periodical < Item .

L'opération suivante permet de procéder à une diminution de qtyAvaillable d'une unité lorsque un article (book ou periodical) vient d'être rendu. Cette opération matérialisée par une règle de réécriture, témoigne de l'aisance même du langage MAUDE à exprimer d'une façon naturelle l'échange d'information entre objets concurrents et par conséquent intervient directement sur le changement dynamique des variables intervenant sur le comportement des objets. Par exemple, les quatre classes (objets) se synchronisent pour décroître le nombre d'articles qui seront disponibles dans la bibliothèque.

Rl [*decreaseqtyavailableOfaBook*] <R : reader | code : ?, ISBN = ?>(getBookin) <BookLend : BookLending | InitialDate : date, ExpiredDate : date + 15, Observation : "nil">(RentBook)<book : Book | Author : "Peter Moses", ISBN : "#=1252514", Year : 2004, Quantity : N>(getLending) →<book : Book | Author : "Peter Moses", ISBN : "#=1252514", Year : 2004, Quantity : N - 1>

La règle *decreaseqtyavailableOfaBook* ne fournie pas toutes les informations mises en valeurs lors de l'interaction entre l'objet reader, l'objet book et la relation *book lending.* Nous pouvons à titre

d'exemple étendre l'interaction et faire intervenir la sous-classe **TypeOfBook** appartenant à la classe Book puis faire intervenir les objets *labrarian* et *library* pour mettre à jour la base de données.

Il serait aussi possible d'élargir la classe Book et la sous-classe Periodic afin de donner plus de détails à leurs caractéristiques. Par exemple rajouter le type de livres et le numéro de l'instance du livre. Ce qui fait que par similitude nous déclarons la même chose pour les articles (item) de Periodical, ceci se fait comme suit :

```
class TypeofBook |  code : string,  description : string .

class BookCopy |  number : string,  position : string,  status : bookStatus .

subclasses typeOfBook BookCopy < Book .

class TypePeriodic |  number : string,  description : string .

class PeriodicCopy |  number : string,  position : string,  status : PeriodicStatus .

subclasses typePeriodic PeriodicCopy < Book .

class BookReservation |  reservationDate : date, reader : Oid, item : Oid, observation : string .

class  Booklending | lendingDate : Date, initialDate : Date, Observation : string,
       reader : Oid, item : Oid .

class  PeriodicalReservation |  reservationDate : date, reader : Oid, item : Oid,
       observation :  string .

class PeriodicalLending | lendingDate : Date, initialDate : Date, Observation : string,
       reader : Oid, item : Oid .

class Calendar | date : Date .

rl [new-date] : <C : calendar | date : D> ➔ <C : Calendar | date : D + 1> .

class Library | intitution : string, address : string, loanPeriod :   maxLoans: Pfun(Cid, Nat),
        suspendedsers : set(Oid) .

class Pay_fines | reader : string,  pay_date : Date, amount :  money .
```

Dans le diagramme des classes de la Figure 8.16, la classe Librarian ne possède pas d'attributs, mais elle offre trois méthodes qui sont déclarées comme des messages crées pour permettre la communication et l'échange d'information et de notification.

```
class Librarian .

msgs borrow return : Oid Oid Oid ➔ Msg .

msg payCharges : Oid Money Oid ➔ Msg .
```

Chaque message sera renommé juste après l'opération qu'il modélise, et le premier et le dernier argument du message sera l'identificateur des objets appelant respectivement. Le reste des arguments correspondent aux arguments déclarés pour l'opération. Par exemple :

```
msg payCharges : Oid Money Oid ➔ Msg .
```

détermine les charges qui doivent être payées par le lecteur pénalisé.

8.7.1 Contraintes: passage de l'OCL vers Maude

Une fois avoir spécifié la forme statique qui détermine la structure de notre système, nous passons à la deuxième exigence à savoir interpréter les contraintes sur les différents diagrammes. Dans ce cadre d'idée nous pouvons nous imposer certaines spécifications qui contraignent le comportement du

système. Pour bien exprimer le passage de la spécification semi-formelle donc dans le langage OCL vers une réécriture dans le langage MAUDE, prenons en considérations les trois exigences suivantes :

1. Premier Invariant : Il ne doit y avoir qu'une seule bibliothèque et qu'un seul calendrier dans le système
2. Deuxième Invariant : Le nombre d'objets à emprunter doit être égal à la somme des valeurs des attributs borrowedItems de tous les lecteurs dans le système
3. Troisième Invariant : La liste suspendedUsers de la bibliothèque doit contenir exactement les identificateurs de tous les lecteurs suspendus.

Le premier invariant limite le nombre d'objets d'une classe donnée dans le système. Le prédicat suivant *thereIsOneAndOnlyOne* impose le fait qu'un seul (unique) objet d'une classe donnée dans une configuration donnée.

```
op  thereIsOneAndOnlyOne : Cid Configuration → Bool .
op numberOfOccurrences : Cid Configuration → Nat .
var M : Message .
var Ats : Attributes
var O : Oid .
vars A A' : Cid .
eq thereIsOneAndOnlyOne(A, C) = numberOfOccurences(A, C) == 1 .
eq numberOfOccurences(A, <O : A'| > C) = if <O : A' | > : A then 1 + numberOfOccurences(A, C)
     else numberOfOccurences(A, C) fi .
eq numberOfOccurences(A, M C) = numberOfOccurences(A, C) .
eq numberOfOccurences(A, none) = 0 .
```

Pour le deuxième invariant, la spécification s'écrira :

```
op consistentNumberOfLoans : Configuration → Bool .
op borrowedItemsSum : Configuration → Nat .
var N : Nat .
eq   consistentNumberofLoans( C ) =   numberOfOccurences(Loan, C) == borrowedItemSum( C ) .
eq   borrowedItemSum(< 0 : Borrower | borreowedItems : N> C) = N + borrowedItemsSum( C )  .
ceq  borrowedItemSum(< 0 : A | > C)  =  borrowedItemsSum( C ) if not  < O : A|> : Borrower .  .
eq   borrowedItemSum(M C) =  borreowedItems( C) .
eq    borrowedItemsSum(none) = 0 .
```

Le troisième invariant, est décrit comme suit :

```
op consistentSuspendedUsers : Configuration → Bool .
eq consistentSuspendetUsers(<O : Library | suspendedUsers : O' OS >
   <O' : Borrower | status : BS > C) = (BS == suspended)  and
   consistentSuspendetUsers(<O : Library | suspendedUsers :  OS > C) .
eq consistentSuspendetUsers(<O : Library | suspendedUsers : none >
```

<O' : Borrower | status : BS > C) = (BS =/= suspended) and consistentSuspendedUsers(< O : Library | > C) .

ceq consistentSuspendedUsers(< O : Lbrary | > <O' : A | > C) = cocistentSuspendedUsers(C) if not <O : A | > : Borrower .

eq consistentSuspendetUsers(M C) = cocnsistentSuspendedUsers(C) .

eq cocnstentSuspendedUsers(< O : Library | suspendedUsers : OS > = OS == none .

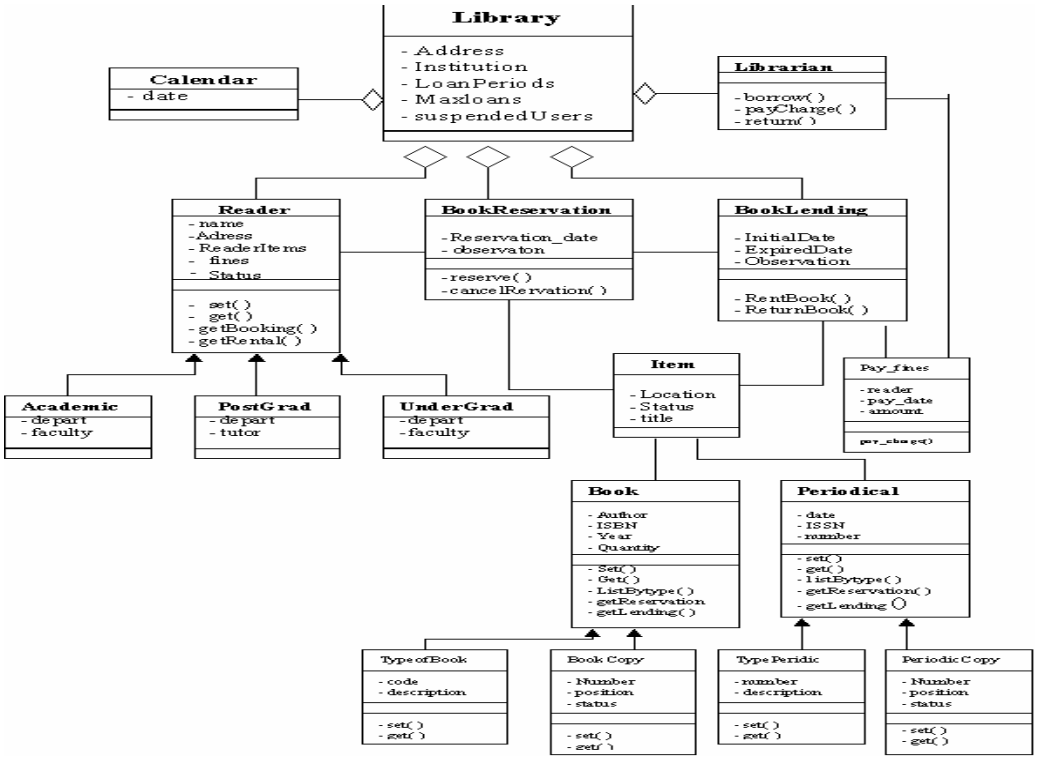


Figure 8.16 : Diagramme des classes d'une bibliothèque

Cependant, nous devons dans une certaine mesure pouvoir déclarer tous les invariants pour qu'ils soient reconnus par le système, pour cela il est nécessaire de déclarer dans un premier temps l'opération "{_}" comme suit : op {_} : Configuration → WholeConfig .

qui est un constructeur de sorte WholeConfig. Dans ce cas, il serait en outre necessaire de differencier entre le système global et une de ces partie. Ce qui fait que cet invariant sera declaré par l'équation speciale dite de membership.

var C : Configuration .

subsort ValidLibrary < WholeConfig .

cmb {C} : ValidLibrary if thereIsOneAndOnlyOne(Calendar, C)

and therieIsOneAndOnlyOne(Library, CD)

and consistentNumberOfLoans(C)

and consistentSuspendedUsers(C) .

8.8 UML et le temps réel

8.8.1 Une représentation limitée du temps

L'expression des CT explicites reste limitée en UML 1.3. [OMG99a]. Sur les diagrammes de séquence, le temps s'écoule de haut en bas. L'axe vertical peut être gradué afin d'exprimer des contraintes temporelles. Deux primitives *(Event.ReceiveTime* et *Event.SendTim*e) permettent de différencier les instants où les messages entre objets sont envoyés et reçus. Pour le reste, des notes dans un format non spécifié, peuvent être utilisées afin de définir des contraintes temporelles.

Dans les diagrammes d'états, en plus d*'Event.ReceiveTime* et d*'Event.SendTim*e, les primitives *When* et *After* peuvent être utilisées. *When* (date = janvier 1999) permet de spécifier un temps logique (ou échantillonné) absolu. *After* (x unités de temps) permet de représenter un temps relatif.

Il n'est donc pas possible, sans créer une extension à UML, de spécifier, par exemple : des événements ou des activités périodiques, de donner des informations probabilistes, de définir des relations entre temps physiques et temps logiques, de différencier simplement les CT internes et externes… Dans ce qui suit nous allons montrer comment la modelisation des contraintes temporelles est apprehendée, nous parlerons aussi de l'aspect vérification en UML-RT (UML real time)[SR98, BJO00, Ste03].

8.8.2 Illustration de UML pour la Modélisation des Systèmes Temps Réel

Dans cette partie de notre exposé, nous allons clarifier le problème de la spécification des systèmes temps réel et leurs propriétés dans le langage UML. Considérons un système temps réel très simple. Il s'agit d'un contrôleur qui est susceptible de réagir à des évènements extérieurs e = '*la valeur de la concentration c est plus grande qu'une certaine valeur M donnée'* ou le délai T de deux évènement successifs est considéré comme l'une des propriétés du système.

Compte tenu des éléments présentés jusqu'ici, nous construisons le système qui est composé du sous-système CS (Control Subsystem) et le RS (Registration Subsystem). Il y a aussi un buffer B dans le système utilisé pour l'échange d'information entre les deux sous-systèmes. Quand un évènement se produit les deux sous-systèmes commencent à fonctionner simultanément. Le sous-système de contrôle CS peut travailler dans l'un de deux modes de façon alternative. Si la concentration c se trouve dans le rang ($M \leq c < Max$), CS fonctionne alors en **mode normale**. Dans ce mode le sous-système calcule la position x de control lid, enregistre le résultat dans le buffer et donne l'ordre (commande) de faire bouger le lid. Si ($c \geq Max$), alors CS se met dans le mode **emergency mode**. Dans ce mode le sous-système ouvre plusieurs lids au maximum, envoie une notification puis enregistre la position courante de x du contrôle lid dans le buffer. Le sous-système RS peut se positionner *off* et *on* par le personnel. S'il est positionné à *on* alors et en réaction à l'évènement e, RS enregistre la concentration actuelle et la position courante du contrôle lid du Buffer.

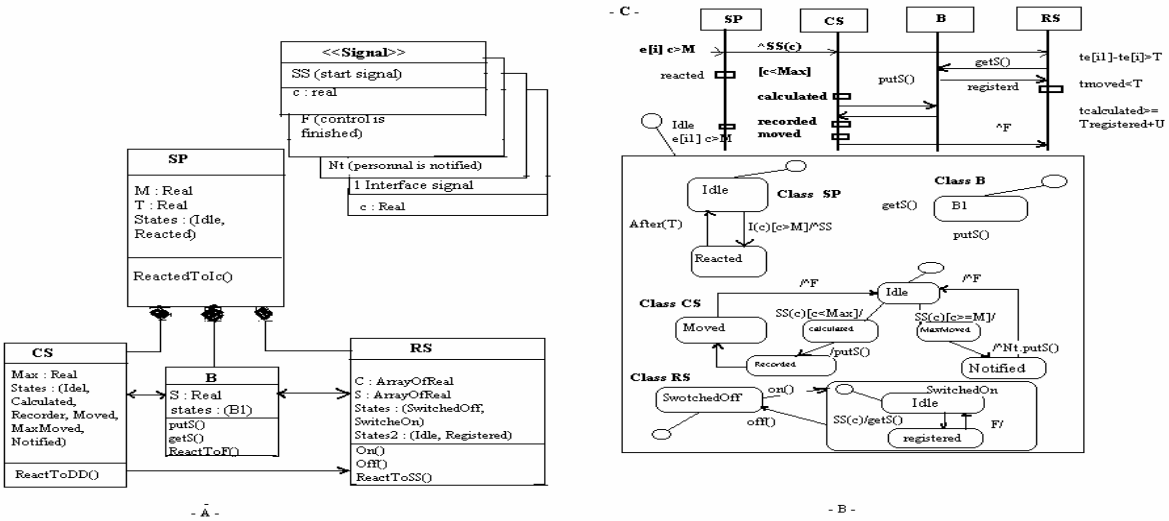


Figure 8.17 : Les diagrammes de classes (A), les diagrammes Statechart (B) et le diagramme des séquences (C)

8.8.3 Une approche pour la spécification des systèmes temps réel en UML

Nous avons montré auparavant, qu'un système temps réel peut être modélisé par une extension, disons intelligente mais lourde de conséquence. Cette approche qui définissait un système temps réel comme un ensemble d'éléments du langage UML (diagramme de classe, collaboration, statecharts etc..), plus quelques artifices qui correspondent merveilleusement aux caractéristiques propres des STRs, à savoir les capsules, les ports et les protocoles. Cette façon de représenter les STRs permet justement de retracer la partie statique et le comportement, disons la partie dynamique. Naturellement, il existe d'autres approches [ABD98, WTB+00, TER00, KNR02, Bjo00] qui ont permis d'apporter plus d'éclaircis compte tenu de la sensibilité des STRs. Ces méthodes qui puisent leur fondement théorique de travaux de recherches déjà cités, tels que ceux de Dill, Alur, Henzinger, Manna et Pnueli et bien d'autres permettent non seulement de proposer des architectures temps réel, mais en plus définissent une manière noble de traiter l'expression de propriétés dans un langage ou dans un autre épousant par la même occasion un aspect temporel. Donc, du temporisé exprimé en général selon la caractéristique du problème par des valeurs prises dans un domaine la plupart du temps dense pour le continu, c'est-à-dire prenant des valeurs dans IR^+, et IN^+ pour le discret.

Dans le présent travail, nous proposons un metamodel de UML. Le profile sélectionne des éléments du metamodel UML utile pour la conception mais dans un domaine d'application spécifique. Le profile peut aussi inclure des règles pour la validation et la transformation des diagrammes.

Le profile en question tourne autour des définitions suivantes :

- Un domaine temporel Dense,
- Quelques notions d'UML pour la spécification des systèmes temps réel : real time class, class diagram, object diagram, statechart diagram,
- Quelques notions d'UML pour la spécification de propriétés temps réel : spécification class, extended class diagram, extended statechart diagram, rules of deriving the properties spécification des diagrammes de classes.

Cette approche consiste-en :

- Un diagramme de classes pour le temps réel, construit par mixage des classes temps réel et des classes traditionnelles.
- Un diagramme objet qui définie l'ensemble des objets et leurs associations.
- Un diagramme statechart est construit à partir des statcharts de classes.
- Un diagramme de classe étendu construit en utilisant les classes de spécification avec des contraintes prédéfinies. La spécification de propriétés est dérivée automatiquement à partir des diagrammes de classes étendues.
- Un diagramme de statechart est dérivé automatiquement à partir des classes et des statecharts étendus. La spécification est sémantiquement équivalente à un graphe de transition temporisé.

8.8.4 Spécification des systèmes temps-réel

Rappelons qu'une classe dans le metamodel UML est le tuple CL = (At, Op, C), avec At l'ensemble (fini) des attributs, Op est un ensemble fini des opérations et C est une contrainte.

Supposons pour le moment que C soit vrai. Nous définissons une classe temps réel comme une classe formée des attributs traditionnels et des attributs à caractéristique temps réel de type DenseTime.

Pour modéliser le comportement d'une telle classe, nous utilisons les diagrammes statechart de l'UML. Une configuration de statechart est définie comme un arbre statique formé d'états (états composites, états à histoire, sous-états, états synch, etc.) Une transition est présentée par la notation

suivante : t : $s \xrightarrow{e[g]/\hat{}\,a} s'$, avec s une configuration source, s' est une configuration cible. Une fonction *trigger* (déclenchement) affecte un évènement e dit de lancement à une transition, une fonction de garde associe un prédicat défini sur l'ensemble des attributs de la classe correspondante, et une fonction d'effet qui affecte une action à la transition. Une transition sera empruntée si sa condition de garde [g] soit vraie et son évènement e prend place. Si plusieurs transitions sont choisies, alors uniquement seule l'une d'elles doit être empruntée immédiatement et ce choix est déterministe.

Un diagramme de statechart est le tuple Sch = (S, E, G, A, TR) ou S est l'ensemble de configurations, E est l'ensemble des évènements, G est l'ensemble des conditions sur les gardes. A est un ensemble d'actions, et TR = $\{(s_i, s_j, e, g, a)|s_i, s_j \in S ; e \in E, g \in G, a \in A\}$ est l'ensemble de transitions.

L'introduction du temps est donnée par ce qui suit :

- L'ensemble des évènements doit inclure des évènements sur les horloges, par exemple, t = T (horloge t possède la valeur T), $t_1 \geq t_2$.
- Un prédicat de garde [g] peut être défini sur l'ensemble des horloges.
- Parmi ces actions il existe des actions de réinitialisations des horloges.
- Le temps passera dans la configuration aussi longtemps qu'une transition ne vérifie pas les conditions de passage.
- Le temps passera dans une transition aussi longtemps jusqu'à ce que l'action termine son exécution.

Nous définissons maintenant, la spécification d'un système temps réel comme le tuple :

SP = (Class Diagram, Object Diagram, Statechart Diagram)

Dans ce tuple la Class diagram = (Classes, Associations) est l'ensemble (finie) de classes et d'associations entres les classes. Une association de deux classes A et B dans un diagramme de classe est le tuple R = (Atype, A, B, N_A, N_B), ou AType $\in$ Atypes.

L'ensemble des associations possibles de deux classes est représenté par des arêtes entre les classes avec une terminaison spécifique : Atypes = {aggregation, composition, généralisation, association}.

N_A, N_B sont un nombre d'objets des classes A et B. L'ensemble d'association peut être vide. Il existe un ensemble de signaux Signals $\subset$ Cl parmi les classes. Ils sont marqués par le stéréotype <<Signal>>. Un objet diagramme est utilisé pour éviter les ambiguïtés, et c'est la paire OD = (Objects, $Associations_0$), ou Objects est l'ensemble d'objets des classes du diagramme des classes. $Assocoations_0$ est l'ensemble de relations de ces objets. $Associations_0 \subseteq$ Associations, seulement des relation un à un des objets est autorisée.

Nous notons en fin de compte que l'arbre de calcul temporisé (Timed Computational Tree) peut être spécifié en utilisant TCTL.

8.8.5 Spécification de l'extension du langage UML

Pour spécifier les propriétés d'un système temps réel, nous faisons une extension du langage UML à travers le diagramme d'états et du diagramme statecharts. Les deux spécifications système et propriétés sont maintenant représentées par la paire : diagramme des classes et diagramme de statecharts (Figure 8.26). Nous pressentons dans cette figure une extension du diagramme des classes et du diagramme des statecharts de la Figure. Dans ce graphique quatre spécifications sont notifiées chacune correspond à une propriété définie auparavant. Les propriétés 3 et 4 sont modelisées par les instances du stéréotype Deadline-stereotype. La propriété 2 est représentée par une instance de stéréotype Rearlier-stereotype. La propriété 1 est modélisée par la spécification class Property 1. La classe possède un attribut disons k : Integer qui contient la série des réactions consécutives du système de contrôle CS aux événements e en mode emergency. Quand le système de contrôle réagit en mode

emergency (SS(c) [c ≥Max]), la specification-class Property 1 fait une transition de l'état P1 vers l'état P2 et l'attribut k est incrémenté. Quand le sous-système de contrôle réagit dans le mode normal (SS(c) < Max]), la specification-class property 1 fait une transition de l'état P2 vers l'état P1 et l'attribut k est réinitialisé à zéro. La contrainte de la spécification specification-class

C(SP)property 1 AG(k < 2)

signifie que pour tout chemin et pour tout état de l'arbre de calcul (computational tree) du système du contrôle de concentration (k < 2), i.e. il ne doit y avoir deux réactions consécutives à l'évènement e quand le sous-système CS fonctionne en mode emergency en deux temps.

Une contrainte d'une classe de spécification déclare une propriété de l'arbre de calcul du système. La propriété du système est la conjonction des contraintes de toutes les classes de spécification à partir du diagramme de classe étendu du système.

Nous pouvons à partir de ce système établir quelques propriétés, qui sont :

Proprieté 1 : Il ne peut y avoir deux réactions consécutives à l'événement e, tel que le sous-système CS fonctionne dans le mode emergency pendant deux périodes consécutives.

Propriété 2 : L'enregistrement de la position x du lid à partir du buffer par le sous-système d'enregistrement doit être complété U unités de temps avant l'enregistrement d'une nouvelle position lid au buffer par le sous-système de contrôle.

Propriété 3 : Une réaction à l'occurrence de l'évènement e (du CS en mode normal) serait achevée dans les limites du temps de séparation de l'évènement.

Propriété 4 : Une réaction à l'évènement e (du CS dans le mode emergency) serait achevée dans les limites du temps de séparation de l'évènement.

Dans la figure suivante il est noté les éléments suivants :

- Il existe une interface externe I(c) de concentration c comme paramètre.
- La classe SP modélise une réaction sporadique à l'évènement e. La classe possède une opération ReactToIc() et deux attributs : T ! Real – c'est le temps de séparation ; M : Real – est une valeur de la concentration à partir duquel le système doit réagir. Quand l'interface I(c) prend place et la condition du garde soit vraie, c'est-à-dire [c > M] = true, la classe SP réalise une transition de l'état Idle vers l'état Reacted. La transition Reacted vers idle sera aussi possible après le temps de séparation.
- L'occurrence du signal SS(c) pousse les deux sous-systèmes (CS et RS) à réagir. La concentration c étant le paramètre du signal.
- La classe CS correspond au sous-système de contrôle. Ces attributs Max : Real – est une valeur de la concentration. La valeur de cet attribut définie le mode de réaction du sous-système de contrôle.
- F est le signal, qui est envoyé par le système de contrôle après avoir terminé son travail.
- Nt est un signal de notification du personnel.
- La classe RS est un model du sous-système RS. La classe possède les opérations on(), off(), ReactToSS() et deux attributs :

C : ArrayOf(Real – est un array de valeur de concentration ;

S : ArrayOfReal – est un array de valeurs des positions lid.

- La classe B modélise le buffer. La classe contient l'attribut : S : Real pour sauvegarder une valeur de la position lid et deux opérations : putS() – pour écrire l'attribut S, getS() – pour lire l'attribut S.

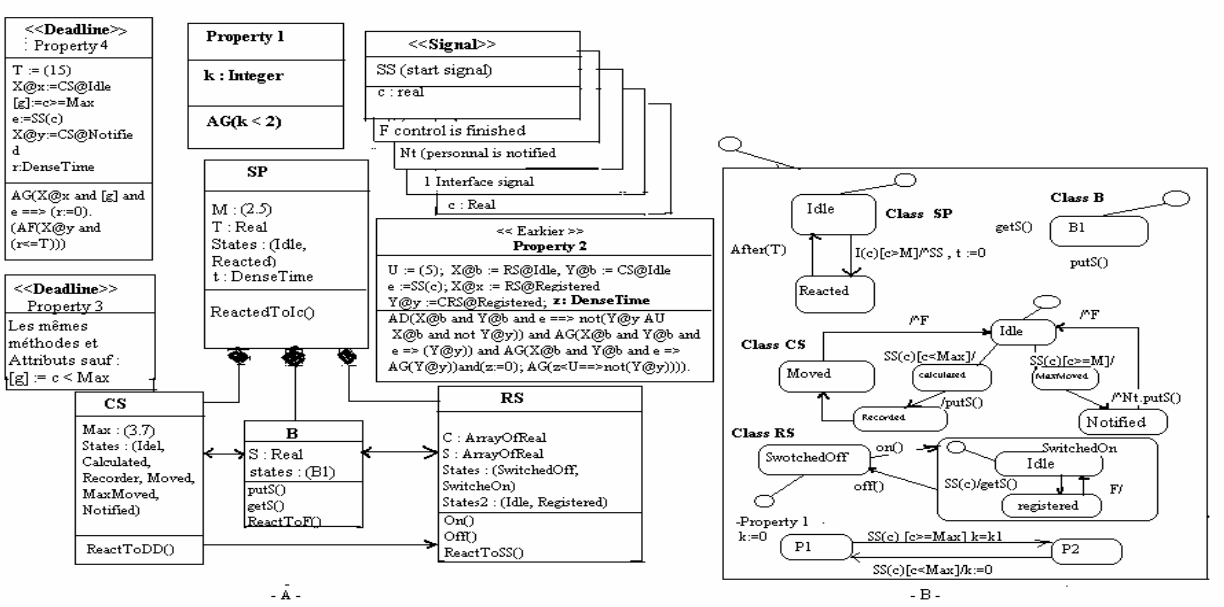


Figure 8.18 : Les diagrammes de classes et de statecharts du système du Contrôle de concentration

Le statechart (Figure 8.26) modélise le comportement du système. Par contre le diagramme de séquence permet de spécifier les propriétés du système. Cette spécification montre certains problèmes : premièrement, nous ne sommes pas capables de spécifier que l'évènement e arrive sporadiquement. Cependant, ce problème peu être résolu par une simple extension syntaxique. Deuxièmement : nous ne pouvons pas représenter toutes les séquences de réactions à l'évènement e sur le diagramme des séquences et nous ne pouvons pas définir leurs propriétés. Ce qui montre que nous ne pouvons pas représenter la propriété 1 en UML.

En général, UML est incapable de représenter les propriétés temporelles et essentiellement la propriété d'atteignabilités (reachability property), ainsi que les propriétés relatives aux séquences de réactions à différentes instances d'un évènement.

Nous proposons dans ce travail une extension du langage UML pour la spécification de propriétés temps réelle.

8.8.6 Exemple de specification-stereotypes

1. Deadline-stereotype. Soit SPx la spécification d'une classe X donnée. $ClassDiagram_X$ définie cette classe. Le $statechart_X$ fixe l'ensemble des états de la classe X, les conditions de garde et les actions.

2. Nous définissons aussi le *deadline* T d'un état de X qui peut être atteint à partir d'un autre état de la classe. Cependant le *Deadline-spereotype* doit être spécifié. La contrainte du deadline-Stereotype est alors écrite comme suit :

CD : AG((X@x) ∧ [g] and ∧ e ➔ (r := 0).(AFX@y) ∧ (r ≤ T)))),

ou r:DenseTme est un attribut. Nous définissons l'ensemble suivant des paramètres du Deadline-stereotype :

- T – la valeur du deadline,
- X@x - la configuration du statechart à partir duquel nous commençons à calculer le temps qui passe,
- g – la condition sur les gardes d'une transition de X@x,
- e – le prédicat sur l'évènement e d'une transition à partir de X@x

- X@y - la configuration de statthalter à partir duquel nous terminons le temps de calcul.

La contrainte ci-dessus signifie que pour tout chemin, pour tout état, si nous somme dans la configuration X@x et le garde [g] d'une transition prenant naissance en X@x est vrai, l'évènement e est activé et nous réinitialisons l'horloge *r* à zéro, alors dans tout chemin, qui commence à partir de cet état, l'état X@y est atteignable avant le deadline ($r \leq T$).

3. *Counter-stereotype*. Quand on spécifie des réactions instantanées à un évènement e, nous utilisons un stéréotype du type counter (compteur) pour compter les différentes occurrences d'un évènement. Le stéréotype *Counter* possède un attribut (i : integer). La contrainte du stéréotype est true. Quand un évènement se produit, la valeur de la l'attribut est incrémenté de 1.

4. *Earlier-Stereotype*. Soit la spécification de deux classes X et Y, quand les classes sont à l'état $X@b \wedge Y@b$, elles réagissent à la même occurrence de l'évènement e. Supposons que les réactions des classes soient des séquences d'états apériodiques, un stéréotype *Earlier* modélise la propriété qu'un état X@x de la classe X se produit U unités de temps avant que l'état Y@y de la classe Y soit atteint.

5. La contrainte de la specification-stereotype :

6. $C_{earlier}$: $AG(X@b \wedge Y@b \wedge e \Longrightarrow \neg(Y@y)\ AU(X@x \wedge \neg Y@y)) \wedge$

$AG(X@b \wedge Y@b \wedge e \Longrightarrow AF(Y@y))$

$AG(X@b \wedge Y@b \wedge e \Longrightarrow AG(X@x \rightarrow (z := 0 . AG(z < U \rightarrow (\neg Y@y)))$

Qui signifie qu' en réaction à l'évènement e :

- Pur tout état (chemin) X@x sera atteint tel que l'état Y@y ne peut être atteint ni avant X@x ni simultanément avec X@x.
- Pour tous les chemins, l'état Y@y sera atteint par moment,
- Pour tous les chemins, toujours, si nous réinitialisons l'horloge dans l'état X@x, alors l'état Y@y ne peut pas être atteint avant $z \geq U$.

8.9 Conclusion

Au cours de ce chapitre nous avons abordé le problème du choix d'une ou des méthodologies de modélisation adaptées de façon particulière aux systèmes réactifs et de préférence temps-réel. Nous avons admis que dans les temps actuels, UML constitue le meilleur choix. Naturellement nous avons explicité dans les détails ce choix et présenté quelques éléments du langage de description UML ainsi que son extension OCL pour l'adapter à nos besoins qu'est la spécification des systèmes temps-réel. Au fil de cet écrit, nous avons montré à travers un exemple très intéressant (Train-Gate-Controller) qu'UML et OCL s'adaptent très bien à la spécification du type de problèmes traités dans notre thèse. Le problème qui se pose actuellement et d'ailleurs qui est sujet à un investissement intense en matière de recherche, c'est justement comment adapté UML/OCL pour l'appliquer en tant qu'outil de description du problème même de la vérification. Dans ce cadre d'idée nous avons proposé une approche singulière et l'avoir adapté au problème du TGC, ce qui a conforté dans un certain sens l'un des soucis majeurs des concepteurs.

Dans un deuxième temps, nous avons su adapter les modèles UML/OCL pour qu'ils soient spécifiés dans la logique de réécriture et MAUDE à travers des modules fonctionnels/objets et utilisant des règles d'inférence pour les adapter aux règles de l'OCL, un exemple très intéressant (modèle de la bibliothèque) a été fournit pour montrer le bien-fondé de nos idées.

CHAPITRE 9

Implantation

9.1 Introduction

Dans ce chapitre, nous présentons les différentes parties composant le système global. Nous retardons un petit peu nos explications pour bien définir l'architecture, le modèle logique, le fonctionnement les interfaces et les différents modules composant le système VALID et son extension VALID 2. La partie vérification est matérialisée par l'outil ROBDD définie dans le chapitre 4, des modules de vérification basés sur la logique linéaire temporelle LTL et PTL. Nous notons qu'un module basé sur la logique CTL a été développé mais ne sera pas présenté. A la fin de ce chapitre nous détaillons deux exemples typiques de systèmes (l'un hardware et l'autre software) qui ont été spécifié puis vérifier et finalement un tableau récapitulatif d'un certain nombre de benchmarks avec leurs solutions et comparaison.

9.2 Module de Spécification et de Simulation des Systèmes Réactifs

Les méthodes de description issues de nos travaux utilisent en partie la philosophie UML et le système VALID tel qu'il est présenté dans [AM94,AS94,AS96] par Attoui et al. Nous montrons dans ce qui suit que VALID a été enrichi de différentes manières. Nous fournirons par la suite une table mettant en valeur les caractéristiques de VALID, de VALID2 et leurs différences, avant cela une introduction rapide mettra en valeur ces deux systèmes.

9.2.1 Le Système VALID

Contrairement à certains systèmes de modélisation des systèmes concurrents tels que LOTOS et ESTELLE, VALID présente un formalisme qui s'apparente d'avantage au langage Maude. D'abord, il ne fait appel qu'à une seule théorie pour la spécification formelle des structures de données et de la dynamique des systèmes. Le langage adopté par VALID peut être considéré comme un sous-ensemble du langage MAUDE, aussi minime soit-il mais toute fois très expressif. Il s'agit d'une généralisation de la programmation fonctionnelle orientée objet. La principale différence entre VALID et MAUDE est que VALID retient un seul concept de base qui est le module formel. Un module formel peut décrire un module de façon similaire telle que OBJ1, OBJ2 et OBJ3. Le module formel une fois instancié est utilisé pour générer, filtrer et réduire les entités syntaxiques qui le concernent dans les phases du système formel correspondant.

Le système VALID, durant le processus de spécification considère tout objet comme un système complexe composé si nécessaire d'autres objets. Le processus de décomposition est défini alors comme une suite hiérarchique de composants. Dans ce cas, un module formel peut faire appel pour sa spécification à d'autres modules formels, ce qui forcément met en place un mécanisme de prise en charge de la notion de niveau et de visibilité qui est similaire à la notion de métarègles telles que définies dans les systèmes experts. C'est-à-dire qu'un système A est d'un niveau immédiatement supérieur à un niveau B, si le système B intervient directement dans la composition du système A. De cette façon le système A ne connaît alors que les attributs des systèmes de son niveau qui sont rendus visibles ou publiques.

9.2.1.1 La Signature du Langage VALID

La figure 9.1 donne la grammaire du formalisme textuel pour la spécification des systèmes concurrents temps réel de VALID.

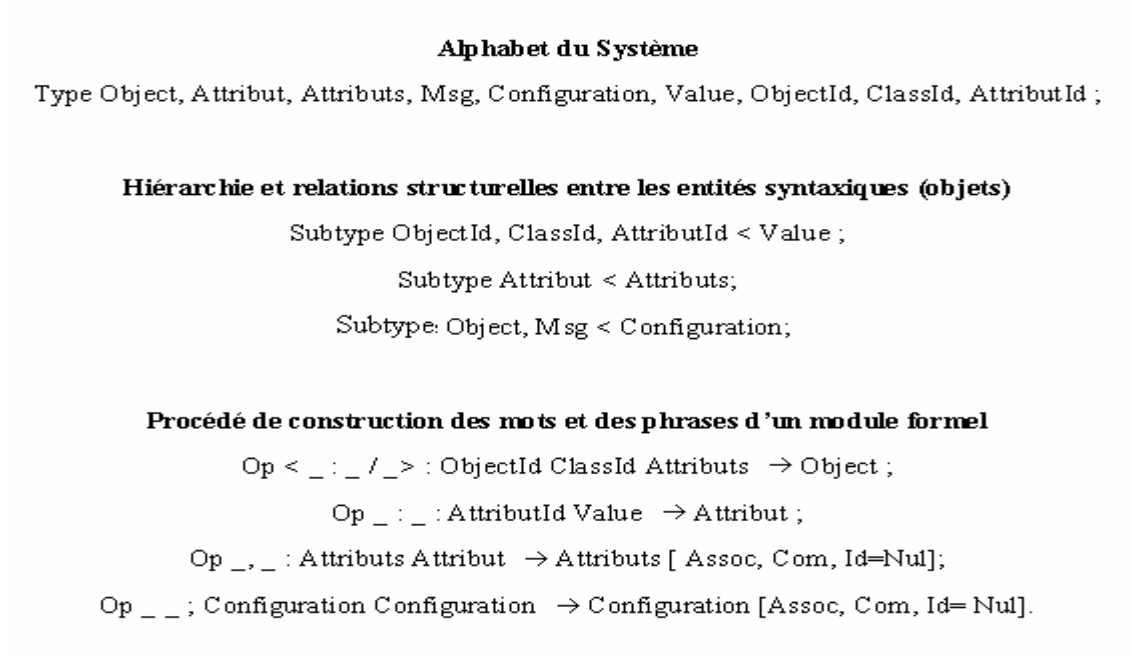


Figure 9.1 : Signature des Systèmes Concurrents supportés par VALID

Cette signature précise la construction syntaxique du système formel. Neaumoins elle ressemble un tout petit peu à une partie de la grammaire de MAUDE. Nous pouvons très facilement faire une correspondance, à titre d'exemple du mot clé « Type » de VALID est le mot réservé « sort » de MAUDE. Il n'est pas nécessaire d'introduire les définitions de chaque partie de cette syntaxe, le lecteur est invité à consulter les travaux de Attoui et al. [AS96, Has97].

La description d'un système consiste à instancier les méta-types des éléments du vocabulaire (Object, Attributs, Msg, …), en fonction de la réalité perçue de façon à définir les objets qui le compose ainsi que les messages ou les évènements susceptibles d'agir sur le comportement de l'objet ou de l'interaction de plusieurs objets.

9.2.1.2 Les règles de réécriture dans VALID

Un objet en VALID peut être sollicité pour un service. Cette sollicitation est représentée par un message qui peut contenir des paramètres. L'effet d'un message sur un objet est décrit à travers une règle. Un tel message ne peut être émis que par des objets du même niveau ou du niveau supérieur. Un message peut être intercepté par une règle et redirigé vers n'importe quel niveau. Les règles dans ces conditions permettent de raisonner par changement d'états des objets du système et de tirer des conclusions valides sur l'évolution. Une règle indique ainsi que le système passe d'une configuration à une autre exactement comme dans le langage MAUDE. Une règle peut aussi être activée lorsque toutes les conditions sont réunies. La forme générale d'une règle dans VALID est donnée par :

Messages1 .. n Objets1..m ➔ ObjetS1..k NewObjetsS1..p NewMessages1..q [Timing]

Pour comprendre le comportement d'une telle règle, il vaudrait mieux pour nous de se refferer au langage Maude.

Pour mieux appréhender la syntaxe et la sémantique de VALID un exemple simple est sollicité dans ce cas. Nous rappelons l'exemple traité dans la partie MAUDE, il s'agit de la gestion de comptes bancaires qui sera réécrit dans le langage VALID, cette spécification est représentée par les deux modules formels suivants (Figure 9.2) :

ModuleFormel COMPTE :

(**Importe** Date, Montant, Liste, Compte ;

Signature

{

Type Compte, Id.Compte, ObjectId, Msg, Valeur, Configuration ;

SousType Id.Compte < ObjectId < Valeur ;

SousType Msg.Compte < Msg ;

SousType Msg, Compte < Confgration ;

Op <_ : Compte / solde : _, écriture : _> : Id.Cmpte Montant Liste[Coupe[Date, Montant]] → Compte ;

Op (créditer_de_) : Compte Montant → Msg.Compte ;

Op (débiter_de_) : Compte Montant → Msg.Compte ;

Op (solde de_pour _) : Compte Montant ObjetId → Msg .Compte;

Op (solde de_est _pour) : Compte Montant ObjetId → Msg .Compte;

Op _ _ : Configuration Configuration Cnfiguration [Assoc, Com, Id= Nul]

}

Règles

{

Var C : Compte ;

M, S : Montant ;

H : Triplet ;

I : Oid ;

(créditer C de M) <C : Compte / solde : N, écriture : N, écriture : H> ==>

<C : Compte / : N + M, écriture : (date_aujourd'hui, +M) H> ;

(débiter C de M) <C : Compte / solde : N, écriture : H> ==>

<C : Compte / : (solde : N – M) >= 0, écritures : (date_aujourd'hui, -M) H> ;

(solde de C pour I) <C : Compte / solde : N, écriture : H> ==>

<C : Compte / solde :: N, écriture : H> (le solde de C est N pour I) ;

}

}

ModuleFormel BANQUE

{

Importe Compte, Montant, Date, Nat, Liste, Triplet, Pourcentage ;

Signature

{

SousType Cpt_dépôt < Compte ;

SousType Cpt_librett < Compte ;

SousType Codevi < Cpt_livret , Compte ;

SousType Cpt_courant < Cpt_livret ;

Op (Triplet) : Cpt_dépôt → Liste[Triplet[Date, Nat, Argent]] ;

Op (Taux_) : Cpt_livret → Poucentage ;

Op (plafond_) : Codevi → Montant ;

Op (datefermeture_) : Codevi → Date ;

Op (chèque à enregistrer numéro _ de valeur _ sur le compte _) : Nat Montant Compte → Msg.Cpt.dépôt ;

Op (intérêt de *) : Compte* → *Cpt*livret ;

Op (fermet _ dans _) : Compte Compte → Msg.Codevi ;

Op (transférer _ de _ dans _) : Argent Compte Compte → Msg ;

}

Règles

{

Vars C, CV : Compte ;

N : Montant ;

N : Nat ;

(chèque à enregistrer numéro N de valeur M sur le compte C) <C : Cpt_dépôt/Triplet :H> = => C :Cprt_dépôt / Triplet : insérer(date_aujourd'hui, N, M) dans H)> (débiter C de M) ;

(intérêt de C) <C :Cpt_livret / taux : R> = = > <C :Cpt_livret / taux : R> (créditer C de (solde de C)* R) ;

(fermer CV dans C) <CV : Codevi /plafond : P, date_fermeture :D> = = > (créditer C de (solde de CV)) si D < date_aujourd'hui sinon (créditer C de (/*formule permettant le calcul des intérêts*/)) ;

(transferet M de C dans CV) = = > (débiter C de M)(créditer CV de M) ;

Figure 9.2 : Spécification des comptes bancaires dans VALID

Nous pouvons remarquer que l 'alphabet de ce module formel est constitué d'objets complexes importés (Date, Montant, Liste et Couple etc.). Il existe par contre un deuxième type de module dit module fonctionnel dont la description est tout à fait identique à celle d'un module orienté objet. La seule différence entre les deux types de modules au niveau conceptuel, réside dans leur comportement. Un module fonctionnel à un comportement purement fonctionnel. Il sert à définir un type abstrait de données qui représente une classe d'entités qui ne change pas dans le temps (statique). Un module de ce genre contiendra les éléments qui permettent de construire, d'interroger et de manipuler ses éléments. Dans ce module le terme « équation » est utilisé au lieu de « règle ».

En outre VALID permet de déclarer des règles synchrones et asynchrones juste pour différencier entre le vrai parallélisme et celui d'interleaving.

Le système VALID admet une représentation particulière des règles dite internes, elle se présente sous cette forme :

$$O_i = = > \ O_{i\,m} \ldots O_n \, M_q \ldots M_r \, [T].$$

Ce type de règles permet d'exprimer les changements d'états dus non pas à une excitation externe, mais à une logique interne qui n'est pas visible à ce niveau. Elles concernent donc un objet complexe actif, qui peut évoluer dans le temps suite à une activité interne. Cela permet de spécifier, entre autres, le comportement de certains objets qui obéissent à des lois internes quelconque, on introduit l'indéterminisme dans le comportement.

9.2.2 Processus de Modélisation dans VALID

9.2.2.1 Architecture de VALID

Le système VALID est un système ouvert qui épouse différents concepts de modélisation et repose sur un processus de modélisation cyclique dit de Gourgant (Figure 9.3). Le cycle dans ce cas est incrémental et itératif, chaque étape étant susceptible d'alimenter les autres. Il est arrêté que lorsque le fonctionnement du système est jugé conforme aux désirs du concepteur.

L'environnement VALID est formé de deux grandes parties aussi importantes l'une que l'autre à savoir :

- Le modèle de connaissance (ou de fonctionnement), contient les spécifications de topologie et de fonctionnement attendu du système. Ce modèle étant la représentation de la partie statique du système, ensuite la partie dynamique sera spécifiée formellement par des règles de réécriture telles que définies auparavant. La description de cette partie est assurée par une interface graphique très conviviale et simple à la fois.

- Le modèle d'action : c'est la traduction du modèle de connaissance, d'abord en un système de réécriture concurrente pour prouver que la spécification du système satisfait les propriétés de confluence, de terminaison. Ensuite une série de transformations formelles de spécifications et dont l'étape finale est l'obtention du code en Prolog qui assurera par la même occasion le comportement du système par l'intermédiaire de l'exécution des règles de réécriture générées par PROLOG lui-même.

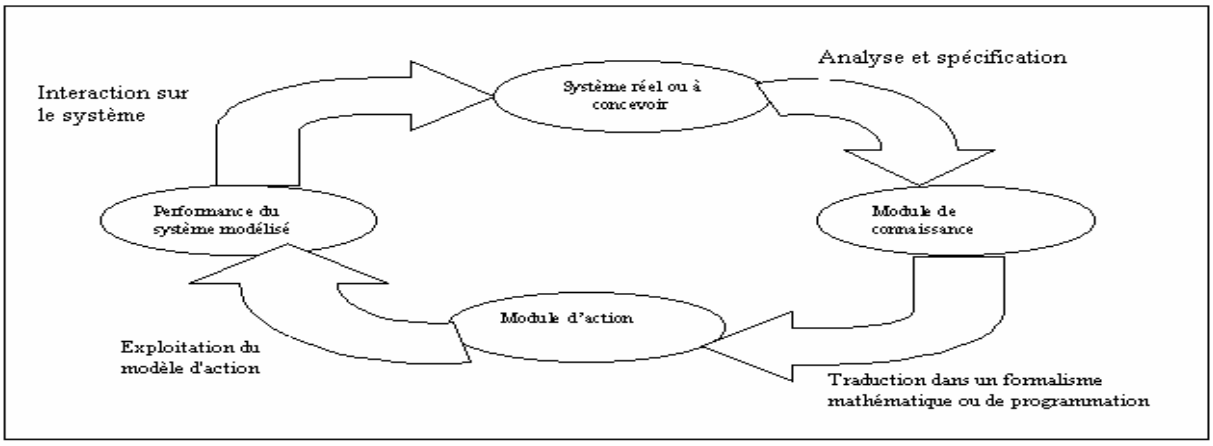


Figure 9.3 : Processus de modélisation

La figure suivante (Figure 9.4) définie au mieux l'architecture de VALID quelque peut amélorée dans le cadre de nos travaux. Comme extension apportée à VALID, plusieurs détails furtifs sont consignés dans les articles [RBJ+03a, RBJ+03a, RJ04a, RJ04b].

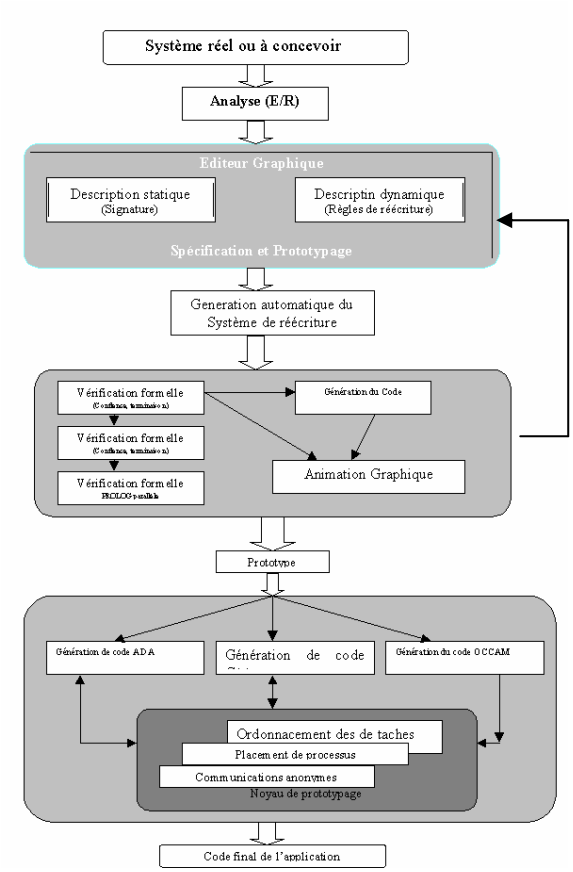


Figure 9.4 : Architecture de VALID

9.2.2.2 Méthodologie de spécification des systèmes concurrents avec VALID

La méthode VALID utilise deux phases de spécification distinctes reliées entre elles.

L'éditeur graphique étant la pièce maîtresse et sert de support. Pendant une session de spécification d'un système, il permet d'instancier et de spécialiser les modèles pour chaque objet du système global. La spécialisation consiste essentiellement à préciser le type réel des méta-types : Objet, Valeur, ObjetId, … Pour les méta-types qui ne peuvent pas être spécialisés automatiquement, soit par manque d'information, soit en raison d'ambiguïtés de description, une interaction avec l'utilisateur est ainsi requise pour plus de clarté.

Le système VALID procède selon les phases suivantes :

9.2.2.2.1 Phase de mise en place de l'architecture et de l'interface

La première phase consiste à mettre en place les objets du système en procédant par niveau d'abstraction. Dans cette phase il est important de respecter la hiérarchie naturelle des objets du système modélisé. A un niveau donné, seuls les objets visibles sont mis en évidence.

Dans VALID, un système complexe sera représenté par les éléments suivants :

- Des variables d'état,
- Des flots de données et de contrôle en entrée (vecteur d'entrée)
- Des flots de données et de contrôle en sortie (vecteur de sortie),
- Des règles et des lois de génération des flots en sortie à partir des flots en entrée (règle de transformation)

- Une dynamique du système, exprimée facilement à l'aide des règles que nous avons introduites précédemment,
- Des attributs visibles des sous-systèmes composites nécessaires pour sa partie dynamique

Cette représentation est récursive et s'applique aussi aux sous-systèmes.

Un objet complexe est considéré comme un système qui est susceptible d'être décomposé en une suite de sous-systèmes. Cette décomposition est guidée en privilégiant les relations de composition qui existent au sein même du système. A ce niveau de spécification des éléments tels que les relations de spécialisation / généralisation, de référence etc., peuvent être prises en charge par le système VALID. Ce qui revient d'une façon globale à découper les spécifications en modules de spécialisation qui s'emboîtent les uns aux autres.

Le système VALID procède selon les étapes suivantes :

- Création du système principal. Dans le cas de l'éditeur VALID le système principal est toujours de niveau 0 et ne peut contenir qu'un seul sous-système
- Création de l'architecture.
    1. Création des systèmes et sous-systèmes.
    2. Création des attributs pour les systèmes spécifiés.
- Création de l'interface.
    1. Création des messages.
    2. Description des attributs visibles.
    3. Phase de vérification de l'architecture et de l'interface : Cette vérification est importante, car avant d'entamer la description des règles il est impératif que l'architecture soit complète et cohérente. Cette phase évitera plus tard de nombreuses étapes de modifications des règles dues à la mise à jour de l'architecture.
- Description du comportement de chaque système.

La visualisation du système à spécifier se présente sous sa forme réduite selon la figure 9.5. Cette façon de représenter tous les sous-systèmes sur une même fenêtre se fait justement dans cette forme (réduite) qui laisse apparaître que l'interface (messages en entrée et en sortie du système et les attributs visible). A ce stade de spécification, nous omettons la description du contenu.

Remarque : Pour le moment nous mettons de côté tous les éléments techniques de création, de modification, de suppression d'un système ou d'un de ces sous-systèmes. Nous ferrons de même pour l'écriture de toutes les appellations (descriptions). Par contre un système développé c'est la représentation graphique de tous les sous-systèmes (sous leur forme réduite) et attributs qui le composent.

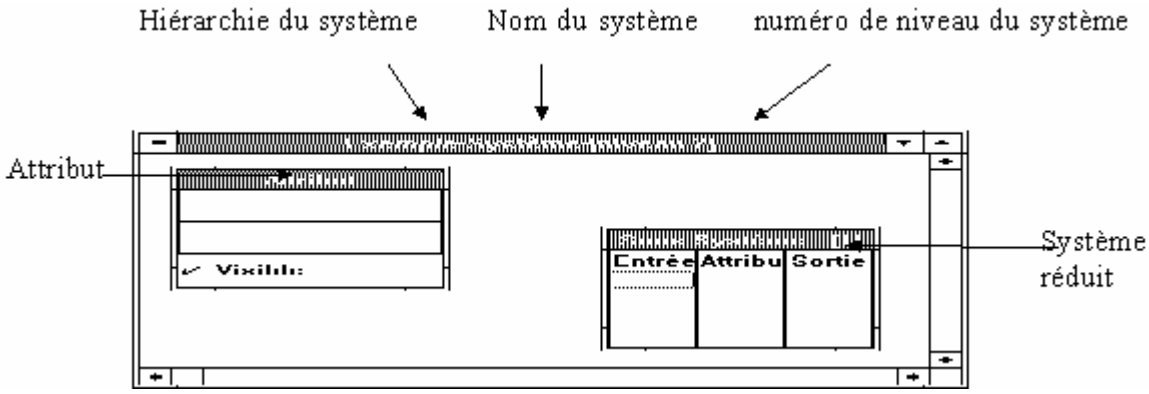


Figure 9.5 : Exemple de système développé.

L'objet Attribut

L'attribut représente un objet qui ne peut être décomposé hiérarchiquement. Il devient un composant à part entière du système et est décrit par les caractéristiques suivantes :

- Un nom qui l'identifie de manière unique dans le système considéré.
- Une description qui représente sa documentation.
- Le Type de l'attribut qui représente l'ensemble des valeurs que peut prendre cet attribut.
- Une case de sélection de visibilité. Si elle est cochée la portée n'est pas seulement locale mais aussi exploitable par le système père de celui qui contient cet attribut.

Pour mieux comprendre la notion d'attribut, il peut être assimilé à une variable locale à un système mais qui peut être modifiable par un système père (ou hiérarchiquement supérieur) en cas de sélection de visibilité.

Les messages

Les messages qui sont pris en charge par VALID sont similaires à ceux de MAUDE (figure 9.6). La boîte de dialogue relative à la description d'un message s'affiche et il ne reste plus qu'à renseigner les différents champs. Après validation des informations, le message apparaît dans la liste concernée par cette création.

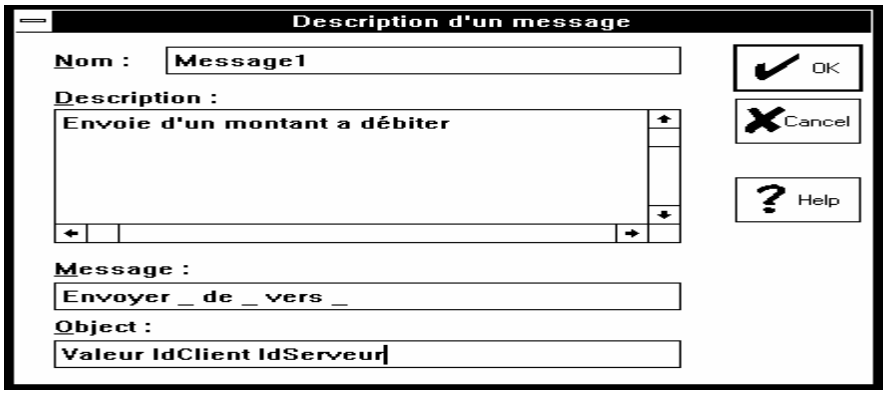


Figure 9.6 : Boite de dialogue de description d'un message.

La réduction dans VALID est un moteur d'ordre 0^+. Il utilise la stratégie du parcours en profondeur d'abord (DFS). Il gère également un échéancier pour la réalisation des contraintes temporelles. Pour créer un scénario de validation, il faut respecter les étapes suivantes :

- Sélectionner un système développé de niveau n, puis sélectionner l'option ***Tester un objet*** du menu ***Réduction***. A ce moment une fenêtre apparaît et affiche le système instancié du système sélectionné précédemment.
- Choisir dans le menu ***Fichier*** l'option nouveau ***Scénario***.
- Décrire les différents états nécessaires au scénario.

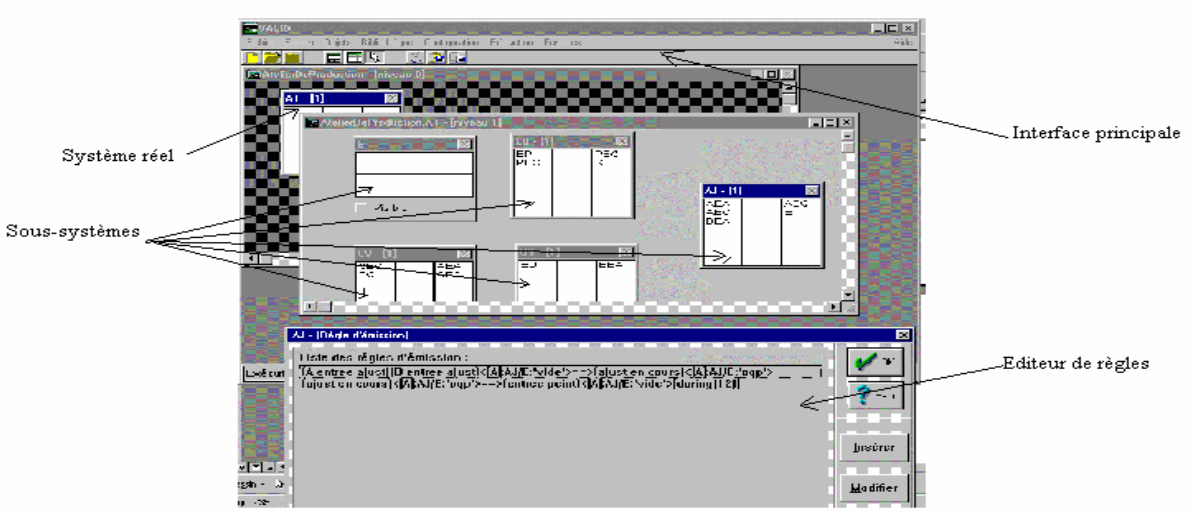


Figure 9.7 : Schéma exhaustif des différentes phases de spécification sous VALID de l'atelier de fabrication

9.3 Simulation et Animation dans VALID 2

Le système de simulation et d'animation tel qu'il a été conçu est écrit dans le langage Java version 1.2.. Le programme utilise des fonctions prédéfinies des classes SIMJAVA une bibliothèque de modèles. SimJava est une collection de trois packages, *eduni.SimJava*, *eduni.SimAnim* et *eduni.SimDiag*.

- **Eduni.SimJava :** est un package utilisé pour construire le modèle de la simulation en java, et produit un fichier de traces comme résultat par défaut. Simjava contient un nombre d'entités qui fonctionnent en totale indépendance, c'est-a-dire en parallèle. Le comportement d'une entité utilise la méthode *Body() ou corps*. Les entités utilisent des primitives de simulation et qui sont :

1. **Sim_schedule() :** envoie un événement à d'autres entités à travers des ports.
2. **Sim_holde() :** suspend la simulation pour un certain temps.
3. **Sim_wait() :** attente pour l'arrivée d'un événement d'objet.
4. **Sim_select() :** sélectionne des événements de la queue reportée.
5. **Sim_trace() :** écrit un message temporaire dans un fichier de trace.

- **Eduni.SimAnim :** fournit un squelette d'Applet pour construire une animation du modèle de simulation. Deux méthodes de SimAnim créent l'animation :

1. **Anim_init() :** pour ajouter des boutons de contrôles.
2. **Anim_layout() :** est étendu pour placer les icônes d'entité.

- **Eduni.SimDiag :** est une collection de classes JavaBeans utilisée pour l'affichage des résultats de la simulation (diagramme, histogramme, etc.). SimDiag fournit deux types de JavaBeans, qui sont (TraceEventListeners), et (GraphEventListeners).

Le graphique de la Figure 9.11 nous donne une idée sur le résultat fournit par le simulateur. Nous pouvons voir sur ce graphique, un réseau de processus (Alternating Bit Protocol), à gauche l'emeteur, à droite le receveur et au milieu (en haut) le bus data-channel et en bas le bus acknowledge (Figure 9.9) .

9.3.1 Architécture de l'éditeur de simulation

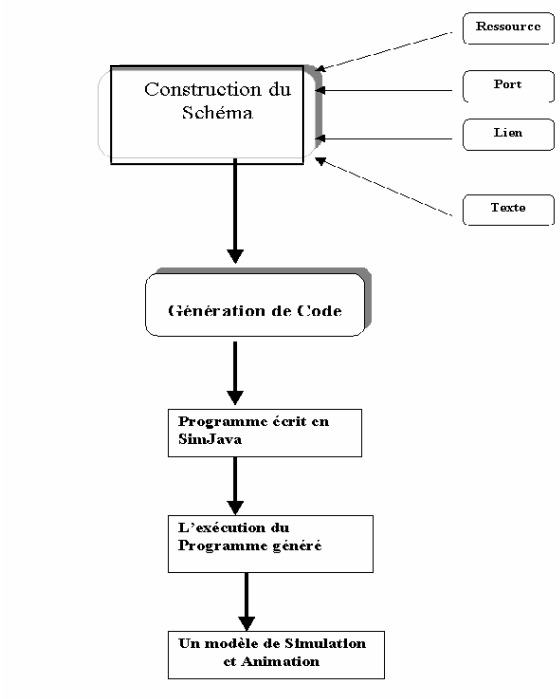


Figure 9.8 : Architecture du simulateur et de l'animateur d'objets concurrents

9.3.1.1 Description des différents composants du système

9.3.1.1.1 Construction de schéma

Notre système est basé sur une interface graphique qui met à la disposition de l'utilisateur un ensemble d'icônes ayant chacune une sémantique propre. Ces icônes sont généralement disponibles à travers des menus, et peuvent être sélectionnées par le biais de la souris ou du clavier. Le schéma est représenté graphiquement par la connexion des objets graphiques (icônes) entre elles pour former une représentation du système à modéliser. Chaque objet (icône) possède généralement des paramètres qui sont spécifiés à travers des boites de dialogues, de ce fait le module en question hérite les particularités de VALID et des descriptions UML (Figure 9.9).

9.3.1.1.2 Le générateur de code

Il est réalisé grâce aux connaissances des composants de schéma. Le générateur de code accepte des connaissances simples de schéma et produit un code source exécutable (écrit en langage Java) (Figure 9.10)

9.3.1.1.3 L'Exécution du programme généré

L'exécution du programme obtenu par l'étape de génération de code, nous donne une animation du modèle simulé (Figure 9.11).

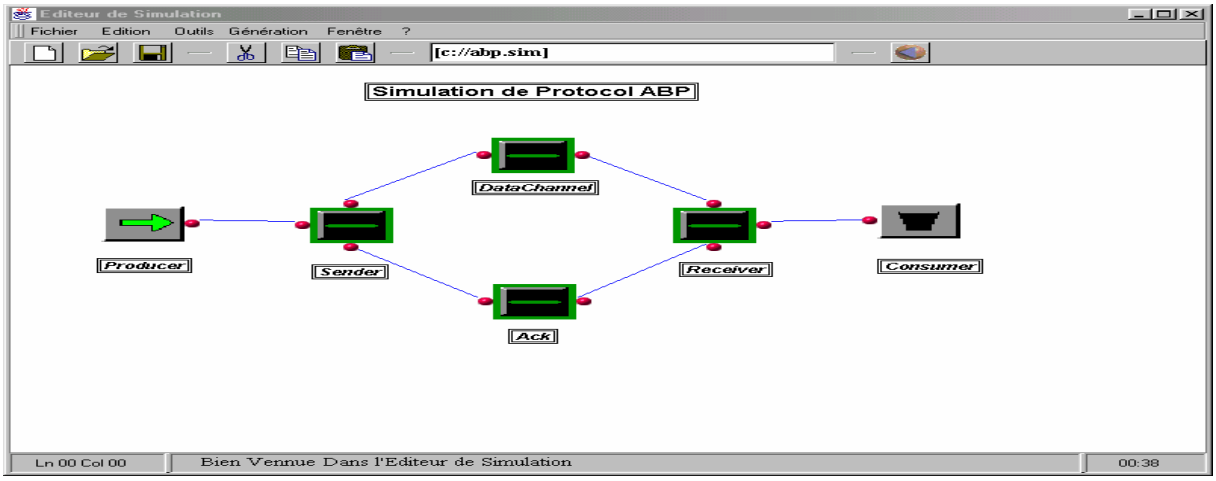


Figure 9.9 : Représentation (modélisation) du Protocole ABP

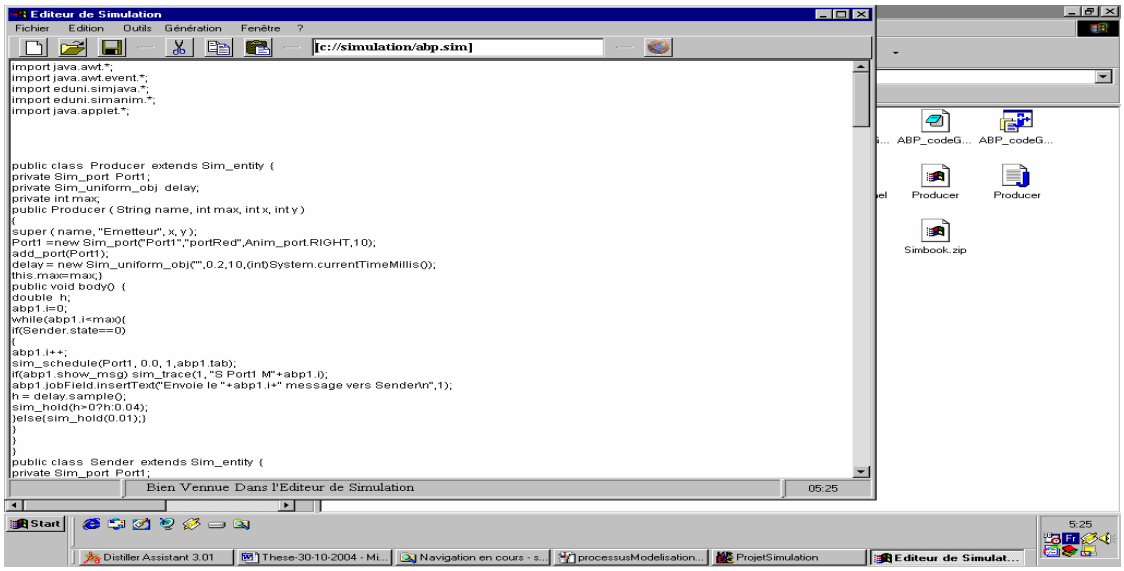


Figure 9.10 : Une partie du code-Java générée du protocole ABP

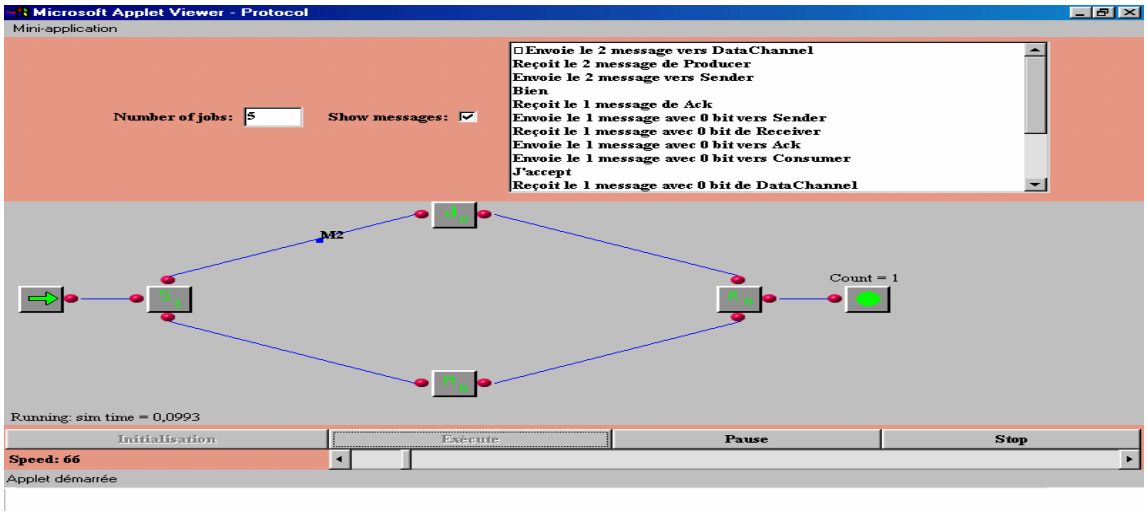


Figure 9.11 : Le résultat de la simulation et de l'animation du Protocole ABP

9.4 Transformation de Spécifications VALID en MAUDE

9.4.1 Vérification de la terminaison et de la confluence d'une spécification

Les règles de réécriture générées par le système VALID lors de leur exécution ne fournissent qu'un code permettant de mettre en valeur le comportement de l'interaction des règles de réécriture entre elles (concurrence). Le comportement tout a fait linéaire, est formée d'une succession de règles consignées dans un fichier trace (*.trc). Ce fichier quelquefois est d'une longueur incommensurable et c'est pour cela que l'outil ORME [Les94a, Les94b], un outil de première génération qui est d'ailleurs un laboratoire de réécriture, fût utilisé pour vérifier dans un premier temps et avant toute exécution la terminaison et la confluence des règles ainsi générées.

L'outil ORME tel qu'il a été conçu offre plusieurs possibilités pour manipuler des ensembles d'équations et des systèmes de réécriture. Les interactions avec l'utilisateur permettent de choisir une commande spécifique permettant d'agir plus finement sur la spécification ainsi prête pour être manipulée. Dans ce cas, l'une des opérations les plus importance étant de donner la possibilité à l'utilisateur de choisir lui-même l'ordre des équations, de les orienter à sa guise et d'opter pour l'une des méthodes de simplification du système, à savoir :

- La méthode des interprétations,
- L'ordre récursif des chemins.

Après la manipulation de la spécification, ORME fournit les résultats de l'une des deux situations suivantes :

- L'ensemble des règles ne possède pas la propriété de terminaison, donc ne possède pas la propriété de confluence. ORME indique ainsi la règle qui est à l'origine de cette non-terminaison. L'ordre établi au sein des symboles d'opérateurs, généralement source de conflits doit être revu.
- L'ensemble des règles se termine. Dans ce cas ORME indique l'ensemble des paires critiques. Si cet ensemble est vide, la vérification est terminée, sinon ces paires critiques sont éliminées et ainsi on rajoutera d'autres règles.

Pour illustrer l'étape de traduction de la spécification (ensemble des règles) pour qu'elle soit prise en charge par l'outil ORME pour la vérification, l'exemple suivant définit au mieux cette opération :

Considérons un sous-ensemble de règles de réécriture tiré de la spécification du gestionnaire de compte bancaire à savoir :

R01 : (contenu de A de C pour O)<C : Client/A=V, …> ➔ <C : Client/A=V,…>(contenu de A est V pour O)

R02 : (mettre A de C à V))<C : Client/A=S, …> ➔ <C : Client/A=V,…>

R1 : (Débiter C de S)<[M] : Client/état : actif> ➔ (connexion serveur de C pour [M] :Client/état :actif>(attendre Débiter C de S)

Ces règles sont traduites dans ORME dans le système d'équations suivantes :

1. Config1(Contenpour(a,c,o),Structobj(c,a,v),Fictif1)=Config2(Structobj(c,a,v),Contenuest(a,v,o),Fictif)
2. Config1(Mettre(a,c,v), Structobj(c,a,s),Fictif1) = Config2(Structobj(c,a,v), Fictif, fictif2)
3. Config1(Debiter(c,s),Structobj(client,etat),Fictif1)= Config2(Connectionserveur(c,client), Structobj(client,etat), AttendreDebiter(c,s))

L'ordre lexicographique des termes ainsi choisis est selon ce qui suit :

Config1 > Config2 > Debiter > Cintenypour >Mettre >Fictif1 > Fictif2 > Structobj > Contenuest > Connectionserveur > AttendreDebiter.

9.4.2 Génération de code MAUDE à partir de Spécifications VALID

Dans le chapitre 6, nous avons montre que la logique de réécriture véhiculée par le langage MAUDE étant une logique universelle prouvée, c'est à dire que du fait qu'elle utilise un mécanisme mathématique très puissant qui est la refléction, permet dans un certain sens à toute logique mathématique ou bien toute description (modèles, langages, machines abstraite, machine d'états, logique temporelle, etc.) de prendre forme dans cette même logique (logique de réécriture), donc d'être décrit dans le langage MAUDE.

Le fait, dorénavant de raisonner en toute liberté en MAUDE, nous donne certes des avantages énormes que nous avons pris le soin de citer auparavant (spécifications entièrement formelles, réécriture, système très puissant et très rapide, plus l'ensemble des outils qui vont avec). Cette possibilité, nous a poussé à écrire un mini-interpréteur écrit entièrement en MAUDE qui permet aux spécifications VALID d'être interprétées et exécuter dans le système MAUDE d'un coté mais aussi d'utiliser les outils de vérification de la confluence, de la terminaison qui sont beaucoup plus performants que l'outil ORME, plus d'autres caractéristiques propres aux systèmes de réécriture mais aussi aux systèmes réactifs en général. En gros, cette façon de procéder nous permet d'intégrer tous les formalismes ainsi développés dans une même plate-forme unificatrice.

Le code suivant introduit juste une partie du mini-interpreteur de transformation de spécifications VALID en théories MAUDE.

```
======
*** déclaration des opérateurs ***
    op Op_ : → _ : Token Sort → OpDecl  .
    op Op_ : →_[_] . : Token Sort AttrList → OpDecl .
    op Op _ : _ →_ . : Token SortList Sort → OpDecl .
    op Op_ : →_[_] . : Token SortList Sort AttrList → OpDecl .
    op Op _ : _ →_ . : NeTokenList  Sort → OpDecl .
    op Op_ : →_[_] . : NeToken Sort AttrList → OpDecl .
    op Op _ : _ →_ . : NeTokenList  SortList Sort → OpDecl .
    op Op_ : →_[_] . : NeToken SortList Sort AttrList → OpDecl .
=======
   *** déclaration de variables ***
    op vars_ :_ . : NeTokenList Sort → VarDecl .
    op var_ :_ . : NeTokenList Sort → VarDecl .
     =======
     *** déclaration des règles ***
    op RL_ : -=>_. : Token Buble Buble → RuleDecl .
    op rl_ : -=>_. : Token Buble Buble → RuleDecl .
    op CRL_ : _=>_if_. : Token Buble Buble Buble→ RuleDecl .
    op crl_ : -=>_if_. : Token Buble Buble → RuleDecl .
     =====
    *** déclaration du module fonctionnel
     op ModuleFormel_ _ FinMF : ModuleName FdeclList → PreModule .
```

Pour continuer dans le même sillage à savoir l'utilisation du système MAUDE comme plate-forme de spécification, nous proposons dans ce qui suit dans un premier temps un Model-checker basé sur la logique temporelle PTL pour vérifier les systèmes réactifs qui présentent des comportements linéaires et un deuxième basé sur la logique arborescente CTL tels qu'ils ont été introduit dans le chapitre 7. Nous donnerons par la suite juste un fragment du code du Model-checker suivant :

9.5 Proposition d'un Model Checker basé sur la Logique Temporelle PTL

Nous rappelons toutes les définitions, la syntaxe et la sémantique de la logique propositionnelle (chapitre 3) et de la logique temporelle linéaire (chapitre 5) à juste titre pour ne pas alourdir ce manuscrit.

Nous savons déjà qu'un Model-checker support deux niveaux de spécification :

- niveau de spécification du système : utilisé pour la formalisation du système concurrent,
- niveau de spécification des propriétés qui doivent être vérifiées annonçant le comportement du système.

Dans le langage MAUDE une théorie de réécriture bien construite peut être spécifiée par un module-système. Aussi en supposant dans un modèle (module-système M) un état initial, disons *init* de sort $State_M$, nous pouvons ainsi vérifier toutes les formules temporelles linéaires de LTL (PTL, PLTL).

Un module qui représente la logique propositionnelle s'écrit alors :

```
fmod PROPOSITIONNELLE is
Protecting TRACE .
sort Boolean.
subsort Bool < Boolean.
op _<_ : Boolean Boolean -> Boolean.
op _and_,_*_: Boolean Boolean -> Boolean [assoc comm prec 55].
op _or_,_+_ : Boolean Boolean -> Boolean [assoc comm prec 59].
op _->_ : Boolean Boolean -> Boolean.
op not_ _!_: Boolean -> Boolean [prec 53].
op _<->´_ : Boolean Boolean -> Boolean.
op _nor_ : Boolean Boolean -> Boolean [assoc comm prec 54]
op _xor_ : Boolean Boolean -> Boolean [assoc comm prec 54]
vars X Y Z: Boolean.
eq false < true = true.
eq true < false = false.
eq X and X = X.
eq true and X = X.
eq X or X = X.
eq X or false = X.
eq X -> Y = (not X) or Y.
eq false nor X = X.
```

```
  eq X nor X = false.
  eq X and (Y nor Z) = X and Y nor X and Z.
  eq X or Y = X and Y nor X nor Y.
  endfm
```

Comme les formules temporelles sont définies sur des graphes de Kripke il est nécessaire dans ce cas d'introduire des constructions sur les chemins du graphe ce qui veut dire autrement, sur des traces.

L'interprétation d'une telle situation est assurée par le module suivant :

```
fmod TRACE is
   sorts Event  Event*  Trace .
   subsorts Atom < Event < Event* < Trace .
   sort Formula .
   subsort Atom < Formula .
   ops true false : -> Formula .
   op [_] : Formula -> Boolean [strat (1 0)] .
   op _|=_ : Trace Boolean -> Boolean .
   op _{_} : Formula Event* -> Formula [prec 10] .
   op nil : -> Event .
   op __ : Atom Event -> Event [prec 23] .
   op _* : Event -> Event* .
   op _,_ : Event Trace -> Trace [prec 25] .
   var X Y Z : Boolean .
   var T : Trace . vars A A' : Atom .
   var E* : Event* .
   var E : Event .
   eq T |= X /\ Y = T |= X and T |= Y .
   eq (X /\ Y){E*} = X{E*} /\ Y{E*} .
   eq [X /\ Y] = [X] and [Y] .
   eq [X ++ Y] = [X] xor [Y] .
   eq [true] = true .
   eq [false] = false .
   eq true{E*} = true .
   eq false{E*} = false .
   eq A{nil} = false .
   eq A{A'} = if A == A' then true else false fi .
   eq A{A' E} = if A == A' then true else A{E} fi .
   eq A{E *} = A{E} .
   op _|=_ : Trace Formula -> Bool [prec 30] .
   eq T |= true = true .
   eq T |= false = false .
   eq E |= A = [A{E}] .
   eq E,T |= A = E |= A .
endfm
```

Les formules temporelles sont aisément vérifiables par le module suivant :

```
fmod LTL is extending PROPOSITIONNELLE .
*** syntaxe
   op []_ : Formula -> Formula [prec 11] .
   op <>_ : Formula -> Formula [prec 11].
   op _U_ : Formula Formula -> Formula [prec 14] .
   op o_ : Formula -> Formula [prec 11] .
*** semantique
   vars X Y : Formula .
```

```
   var E : Event .
   var T : Trace .
   eq <> false = false .
   eq <> true = true .
   eq o (X * Y) = o X * o Y .
   eq o (X + Y) = o X + o Y .
   eq X U (Y + Z) = (X U Y) + (X U Z) .
   eq false U X = X .
   eq X U false = false .
   eq true U X = <> X .
   eq E |= [] X = E |= X .
   eq E,T |= [] X = E,T |= X and T |= [] X .
   eq E |= <> X = E |= X .
   eq E,T |= <> X = E,T |= X or T |= <> X .
   eq E |= X U Y = E |= Y .
   eq E,T |= X U Y =
   E,T |= Y or E,T |= X and T |= X U Y .
   eq E |= o X = E |= X .
   eq E,T |= o X = T |= X .
   vars X Y : Formula .
   var E : Event . var T : Trace .
   eq ([] X){E} = [] X /\ X{E} .
   eq ([] X){E *} = X{E *} .
   eq (<> X){E} = <> X \/ X{E} .
   eq (<> X){E *} = X{E *} .
   eq (o X){E} = X .
   eq (o X){E *} = X{E *} .
   eq (X U Y){E *} = Y{E *} .
   op _|-_ : Trace Formula -> Bool [strat (2 0)] .
   eq E |- X = [X{E *}] .
endfm
```

9.6 Expérimentation

Nous avons conduit plusieurs vérifications sur des systèmes dont nous avons pris le soin de formaliser et de spécifier ainsi que certaines de leurs propriétés. La table 9.12 récapitule à juste titre une partie de notre expérimentation.

Nous donnons tout de suite le texte descriptif de la spécification de certains de ses systèmes. Comme nous l'avons dit auparavant, les systèmes réactifs peuvent se présentés dans une forme « Hard »: un circuit ou équivalent et une forme « soft » : un logiciel, un programme ou équivalent.

9.6.1 Exemple de spécification d'un système Hardware: Muller-C circuit

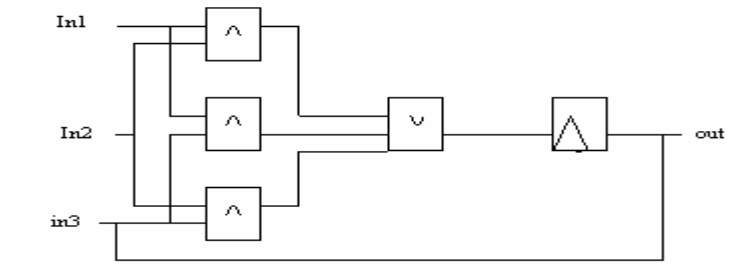


Figure 9.12 Le Circuit Muller C- avec retard

Le circuit Muller-C est réalisé à partir de trois circuits majeurs selon le graphique de la figure 9.12. Comme nous pouvons le constater, la structure et le comportement d'un tel circuit peut être formalisé par la formule de la logique temporelle suivante :

$$[\,](e \leftrightarrow ((in1 \wedge in2) \vee (in1 \wedge in3) \vee (in2 \wedge in3)))$$

avec un décalage d'un délai d'une unité selon :

$$[\,](@out \leftrightarrow ((in1 \wedge in2) \vee (in1 \wedge in3) \vee (in2 \wedge in3)))$$

Lorsque l'on relie la sortie (out) avec l'une des trois entrées, disons In3 par exemple, nous obtenons l'expression du circuit Muller-C comme suit :

$$[\,](@out \leftrightarrow ((in1 \wedge in2) \vee (in1 \wedge out) \vee (in2 \wedge out)))$$

En suivant une telle description du circuit Muller-C, nous pouvons alors développer un outil très large formé par des spécifications pour décrire les comportements du circuit concerné, mais aussi d'autres circuits équivalents. Par exemple nous pouvons mettre en place les spécifications suivantes et démontrer par la même occasion leurs validités. Nous proposons trois spécifications qui se résument comme suit :

Spécification1. Deux signaux en entrée (par exemple In1 et In2) sont au niveau "high" conduit (implique) que la sortie soit à un niveau "low" jusqu'à ce qu'elle devienne (éventuellement) "high" (peut être bien immédiatement) et restera à ce niveau "low", jusqu'à ce que In1 et In2 redeviennent en même temps au niveau "low" de nouveau, ou bien si cette dernière conditions ne surviendra jamais (restera pour toujours de mise). La spécification est écrite comme suit :

$$[\,]((in1 \wedge in2 \rightarrow \neg out\ U(out \wedge (out\ U (\neg in1 \wedge \neg in2) \vee [\,] out)))$$

De la même manière nous decrivons la deuxième et la troisième spécification selon ce qui suit :

Specification 2.

$$[\,]((\neg in1 \wedge \neg in2 \rightarrow \neg out\ U(\neg out \wedge (\neg out\ U (in1 \wedge in2) \vee [\,] \neg out)))$$

Specification 3.

$$out\ U (in1 \leftrightarrow in2) \vee out\ U(in1 \leftrightarrow in2) \vee [\,] (@out \leftrightarrow out)$$

Notre outil permet de façon simple la verification du comportement circuit Muller-C qui se résume à la spécification globale suivante :

$$\text{Muller-C} \rightarrow \text{specification1} \wedge \text{specification2} \wedge \text{specification3}$$

La solution d'un tel comportement est executé en 63 opérations de réécriture en moins d' une millli-seconde.

9.6.2 Exemple de specification d'un "Software" : Peterson Algorithm's

Pour mieux apprécier l'étendue de cet algorithme, j'ose le présenter dans sa version originale qui permet de garder son sens et sa porté. La traduction est aussi aisée que la version anglaise.

Le Problème :

A paradigmatic problem in concurrent programming is the mutual-exclusion problem, which asks for a programming solution to ensure that no two processes simultaneously access a common resource, such as an I/O devise or a write-shared variable.

In a practical way there is a finite numbers of algorithms modelling the problem of the mutual-exclusion control. Among such paradigm, Peterson's algorithm is the simpler solution to the problem of mutual exclusion.

Peterson's Mutual Exclusion Algorithm for 2 processes each of which can adopt 4 states (non-critical section, request section, wait section and critical section). Such states are defined for each process using Boolean equations as follows :

We admit that the parameters c1, c2, d1 and d2 are used to control the accessibility of the critical section, and the unary operator ' ~ ' is the negation form.

```
**** Process  P1
NCS1     = ~c1 /\ ~c2;          # non-critical section
TRY1     =  c1 /\ ~c2;          # request section
WAIT1   = ~c1 /\  c2;# wait section
CS1       =   c1 /\  c2;          # critical section
*** Process  P2
NCS2         = ~d1 /\ ~d2;
TRY2         =  d1 /\ ~d2;
WAIT2 = ~d1 /\  d2;
CS2          =  d1 /\  d2;
```

The processes are modeled using interleaving, which is represented by a program counter' bit (pc). If pc (true) then the first process is running, if ~pc (false) the second is so. There are other 'global' variables (bits), in1 (in2): process 1 (2) requests the critical resource. If in1 /\ in2 /\ t (t is the 'tie-breaker') then process 1 enters the critical section, if in1 /\ in2 /\ ~t then process 2 enters.

/* The complete system: */

peterson = start /\ []first /\ []second /\ fairness

/* The specification to prove mutual exclusion

mut_exc = [] ~(CS1 /\ CS2);

/* To verify that the implementation (Peterson algorithm) meets its requirement (verify the specification), we write :

peterson -> mut_exc

Start, first, second, fairness are defined as follows :

***** The behavior of the first process

first = pc -> (d1 <-> @ d1) /\ (d2 <-> @ d2) /\ (in2 <-> @ in2) /\ (NCS1 /\ @ TRY1 /\ @ in1 /\ (t <-> @ t) \/ TRY1 /\ @ WAIT1 /\ @ ~t /\ (in1 <-> @ in1) \/ WAIT1 /\ @ WAIT1 /\ in2 /\ ~t /\ (t <-> @ t) /\ (in1 <-> @ in1) \/ WAIT1 /\ @ CS1 /\ (~in2 \/ t) /\(in1 <-> @ in1) /\ (t <-> @ t) \/ CS1 /\ @ NCS1 /\ @ ~in1 /\ (t <-> @ t));

***** The behavior of the second process

second = ~pc -> (c1 <-> @ c1) /\ (c2 <-> @ c2) /\ (in1 <-> @ in1) /\ (NCS2 /\ @ TRY2 /\ @ in2 /\ (t <-> @ t) \/ TRY2 /\ @ WAIT2 /\ @ t /\ (in2 <-> @ in2) \/ WAIT2 /\ @ WAIT2 /\ in1 /\ t /\ (t <-> @ t)/\ (in2 <-> @ in2) \/ WAIT2 /\ @ CS2 /\ (~in1 \/ ~t) /\ (in2 <-> @ in2) /\ (t <-> @ t) \/ CS2 /\ @ NCS2 /\ @ ~in2 /\ (t <-> @ t));

**** The initial start

start = t /\ ~in1 /\ ~in2 /\ NCS1 /\ NCS2;

/* Infinitely often pc high and infinitely often pc low: */

fairness = []<> pc /\ []<> ~pc;

The solution of such formula is performed in 103 rewrites and at less than 160 ms.

La table suivante résume au mieux un certain nombre de circuits et logiciels. Dans la première colonne nous présentons le nom du système en question, dans la deuxième sa durée d'exécution utilisant un outil de vérification PTL-tool [PTL-Tool], dans la troisième colonne, le nombre de réécritures qui ont permis la vérification et finalement dans la quatrième colonne le temps d'exécution des spécifications.

Dénomination du model	Temps d'exécution (PTL-tool)	Nombre de réécritures	Temps d'exécution (ms)(VALID2)
Muller_C	0.1 s	70	40
Lars1	3.0 s	124	0
adder	0.1 s	195	0
Mul4	158.7s	216	10
Dff1	0.0	53	20
Josko3	105.6 s	21	10
Josko4	18.0 s	20	10
Mutex1	1.3 s	91	30
Mutex2	19.1 s	93	20
Peterson	-	103	160
Scc-event	-	23	10
Josko5	-	20	10
Muller-C2	-	20	10
Mutex	-	65	10
CallHear	-	34	10
Hailpern	-	70	20

Table 9.12 : Résultats d'expérimentations vérifiées par Notre Model-Checker

9.7 Conclusion

Cette partie de notre thèse concerne les résultats obtenus suite à un effort considérable pour la mise en pratique des idées développées pour la spécification et la vérification des systèmes réactifs. En effet nous avons écrit les programmes suivant :

- Une implémentation des diagrammes de décision binaire qui constitue en lui-même (le BDD) un choix incontournable pour la vérification des systèmes software/hardware décrits comme des réseaux booléens.
- Une implémentation d'un module mixte basé sur les algorithmes génétiques et la programmation linéaire bivalente. Ce module n'a pas été décrit dans cette thèse mais l'a été dans certaines de nos publications.
- L'extension du système VALID en VALID 2, plus certains outils de modélisation, de spécification, de simulation et de vérification.
- L'écriture d'un ensemble de modules exécutés dans le système MAUDE sur une plate-forme Linux.

Nous signalons aussi, que plusieurs expérimentations ont été réalisées à travers la spécification, la programmation et la vérification de plusieurs problèmes (circuits et algorithmes) mettant en œuvre les applications développées dans le cadre de nos recherches et des résultats très encourageants ont été obtenus.

Conclusion et Perspectives

Un système embarqué complexe comme ceux que l'on trouve dans les satellites, les avions, les fusées, les robots, etc., se compose de plusieurs cartes dédiées. Chaque carte assure une fonction particulière (télécommunication, traitement de signal, surveillance des équipements, pilotage et conduite du système, etc.). Chaque carte est composée d'une partie matérielle (matériel RISC, un circuit ASIC, un DSP (processeur de traitement de signal), un microprocesseur classique et d'une partie logicielle (applicatif). Si tous les outils qui relèvent du domaine du génie logiciel des systèmes temps réel permettent actuellement de bien mener à terme les projets de développement des logiciels de ce type de systèmes, la phase d'intégration du matériel et du logiciel reste encore assez laborieuse et très coûteuse, et représente, dans certain cas jusqu'à 50% du cycle de développement du système embarqué.

Les travaux de recherches abordés dans cette thèse, permettent justement de prendre en charge plusieurs étapes du cycle de développement de ce type de problématique. Avant d'entamer une récapitulation des résultats de nos travaux, dont le sujet même nous a influencé pour lancer successivement plusieurs étapes de recherche. Dans la première étape, il était question de prospecter l'état de l'art du sujet et d'apporter une contribution notable en choisissant une projection dans une optique purement formelle. La deuxième étape s'est approchée de très prêt du cadre même de notre objectif premier qui est de réunir tous les moyens théoriques et pratiques pour concrétiser le fait de représenter les systèmes critiques temps réel dans un formalisme universel plutôt une science en elle-même, qu'est la logique de réécriture et ainsi de les pourvoir d'un outil d'exécution suffisamment puissant pour permettre aux spécifications d'être exécutées de façon simple et naturelle. Pour raisonner en terme de réécriture formelle, nous avons eu l'embarras du choix entre Cafe-Obj, Elan, PVS et MAUDE. Une étude très poussée nous a permis de choisir le bon et cela selon plusieurs critères, qu'il n'est pas nécessaire de relater. Notre choix s'est porté finalement sur le langage et système MAUDE.

Dans l'étape suivante, nous avons eu à connaître, la complexité même des systèmes temps réel embarqués puisqu'elle impose d'adopter une méthodologie rigoureuse pour les appréhender et pour garantir une cohérence certaine entre les différentes phases de développement. Ces systèmes (critiques temps réel embarqués) constitués d'entités matérielles et logicielles (co-design) nécessitent l'utilisation d'outils permettant de prendre en charge indifféremment les parties logicielles et les parties matérielles toutes considérations technologiques et ceci dans les premières phases du cycle de développement.

La phase d'intégration logiciel/matériel qui est une phase cruciale, peut être facilitée par le recours à des outils de co-vérification matériel/logiciels. Pour l'aspect logiciel nous avons eu recours au développement de systèmes de spécification et de vérification basés sur des méthodes semi-formelles dans un premier temps (UML et UML-Real Time) pour une représentation fiable des structures internes et externes par des ADTs (abstract data type), mais aussi de leur comportement qui, la plus part des cas est modélisé selon une représentation formelle sous forme de machines d'états ou simplement d'automates généralisés. De façon globale il s'agit aussi de tisser tous les liens entre les différents objets (modules) d'un système. Nous avons eu recours dans ce cas à des formalismes algébriques tel le langage CCS et sa forme étendue Timed-CCS. A partir de là, il était facile d'appréhender la notion de bisimulation (timed et untimed) pour la comparaison et la simplification des systèmes de transitions étiquetés (une représentation abstraite des systèmes concurrents). Dans ce cadre d'idée un algorithme du à Paige & Tarjan et un autre du à Fernandez et Mounier ont été revus (ces algorithmes n'ont pas été repris dans cette thèse). Ces deux algorithmes permettent justement la simplification des systèmes de transition en partitions plus fines, d'où la notion de forte-connexité qui s'impose (Paige & Tarjan Algorithm) et une vérification à la volée (Fernandez et al.) moyennant la notion d'états accessibles et de deadlock qui sont des critères de sûreté inaliénables.

La partie vérification s'est imposée d'elle-même par le recours de l'une des méthodes les plus efficaces de nos jours, c'est la méthode de vérification symbolique implémentée dans les meilleurs outils de vérification (SVM, SVC et autres) et qui est basée sur les diagrammes binaires de décision. Nous notons qu'un module BDD a été écrit dans le langage Java de JDK 1.2. Ce module BDD, tel qu'énoncé dans le texte de cette thèse a été utilisé par la suite comme module SAT, pour la résolution

des systèmes d'équations de la programmation linéaire en nombre bivalents et ainsi de satisfaire des contraintes donc les spécifications.

La partie la plus intéressante de notre thèse et qui nous a valu un nombre important de publications étant la logique de réécriture et le langage MAUDE. Dans le même sens d'idée, plusieurs modules ont été écrits en MAUDE, les uns donnant des codes d'une rapidité certaine pour la simulation symbolique et la vérification de différents systèmes concurrents réactifs. Les autres essayant tant bien que mal de représenter les systèmes temps réels, leur spécification et leur vérification, des cas réels ont même étaient revus et tester.

Comme le passage d'une description formelle vers sa représentation en MAUDE est d'une lourdeur ennuyeuse compte tenu de la longueur même du code à écrire qui avoisine très facilement les centaines, voir, les milliers de lignes, l'écriture d'une interface graphique, de spécification, de génération de code, de deboggage et de simulation s'est faite sentir un peu plus tard. Dans un premier temps nous avons enjoint nos efforts à celle de l'équipe de Attoui, Professeur au laboratoire de recherche CESALP-LLP Université de Haute-Savoie pour adopter l'environnement VALID juste comme une interface graphique qui génère du code et que ce code est transférable via un traducteur dans le langage MAUDE pour qu'il puisse être exécuté. Nous avons ensuite écrit notre propre interface qui reste encore à un stade primitif

Dans le sillage de ce travail, plusieurs techniques de vérification ont été rappelées, les unes réécrite dans le langage MAUDE selon l'algorithme original qui les régit, telles que CTL de Emerson et PTL de Manna. Elles ont été ensuite traduite dans une forme axiomatisable, puisque MAUDE et la logique de réécriture s'adaptent bien à cette forme d'expression en plus de leur symbiose à utiliser leur schéma de déduction basé sur les règles ACI (associativité + commutativité + idempotence).

Finalement, nous notons qu'un ensemble d'idées ont été proposées qui montre l'aisance même du langage de réécriture MAUDE d'adopter UML et OCL et par la même occasion les modèle de vérification.

PERSPECTIVES

Deux champs de vision s'offrent à nous :

- Une implémentation du noyau de réécriture sur machines fortement parallèles : en effet deux aspects importants qui caractérisent la logique de réécriture sont :

    1. La déclaration même des règles de réécriture sous forme d'équations mettant en place la notion de réflexivité, transitivité, symétrie, congruence permet d'exprimer le parallélisme sous ces deux facettes (free & d'interleaving). Le premier énonce une architecture massivement parallèle et le deuxième, la décomposition d'exécutions parallèles en une série d'exécutions qui se relayent par séquencement.
    2. La puissance de la refléction de la logique de réécriture nous donne la possibilité de réécrire un code MAUDE minimal, de l'implémenter sur des puces et de lui associer les programmes que nous avons écrits (spécification, vérification, etc.) et d'en créer autant de machine MAUDE miniaturisées qui puissent constituer une partie d'un système Hardware/Software monté sur une carte électronique.

- Etendre l'interface graphique pour qu'elle soit conviviale le plus possible et qu'il puisse y créer une certaine complicité entre le système informatique et l'utilisateur ou carrément les utilisateurs dans un environnement coopératif. En plus de cela, il s'agit d'étoffer les programmes de spécification et de vérification par de nouvelles techniques, puisqu'il ne se passe un jour pour qu'une nouvelle méthode ne soit trouvée et implémentée. Cet aspect est très encouragé, puisque le système que nous avons développé ainsi que MAUDE sont des applications systèmes fortement ouvertes.

BIBLIOGRAPHIE

[ABD98] André C., Boufaïed H., et Dissoubray S., "Les SyncCharts: un modèle graphique synchrone pour systèmes réactifs complexes", RTS, Paris, pp. 175-196, 1998.

[ACD90] Rajeev A., Courcoubetis C., et Dill L. D.. "Model-checking for real-time systems", 5th Symp. on Logic in Computer Science (LICS 90), pp. 414-425, 1990.

[ACH+95] Alur R., Courcoubetis C., Halbwachs N., Henzinger T. A., Ho P.-H., Nicollin X., Olivero A., Sifakis J., et Yovine S., "The algorithmic analysis of hybrid systems." *Theoretical Computer Science*, 138:3–34, 1995.

[AD90] Alur R. and Dill D. L., "Automata for modeling real-time systems. In Automata, Languages et Programming", 17th International Colloquium, vol. 443, *LNCS*, pp. 322-335, Springer-Verlag, July 1990.

[AD94] Alur R. et Dill D.L., "A theory of timed automata", *Theoretical Computer Scienc*e, 126(2):183-235, 1994.

[AH92] R. Alur R. et Henzinger. T.A. "Logics and models of real time: a survey", In J.W. de Bakker,K. Huizing, W.-P. de Roever, and G. Rozenberg (eds.), Real Time: Theory in Practice,vol. 600, *LNCS*, pp. 74–106. Springer, 1992.

[AHU74] Aho A. V., Hopcroft J. E., et Ullman J. D., "Design and Analysis of Computer Algorithms", *Addison-Wesley*, Reading, MA, 1974.

[Ake78] Akers S.B., "Functional Testing with Binary Decision Diagrams", in Eighth Annual Conference on Fault-Tolerant Computing, pp. 75-82, 1978.

[AKZ96] Awad M., Kusela J., et Ziegler, J., "Object-Oriented Technology for Real-Time Systems", Prentice Hall, NJ, 1996.

[Alu98] Alur R., "Timed automata", in NATO-ASI Summer School on Verification of Digital and Hybrid Systems, 1998.

[AM00] Arajo, J., et Moreira, A., "Specifying the Behaviour of UML Collaborations Using Object-Z", Proc. of the Americas Conference on Information Systems, California, 2000.

[AN92] Arnold. A. et Niwinski. D., "Fixed point characterisation of weak monadiclogic definable sets of trees", in Nivat, M. & Podelski, A. (eds.): Tree Automata and Languages, Elsevier, pp. 159-188, 1992.

[Ari96] Ariane 5 flight 501 failure: Report by the inquiry board, July 19, 1996.

http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

[Arn88] Arnold. A., "Logical definability of fixed points", in TCS, vol. 61, pp. 289-297, 1988.

[AT00] Aleman. J.L.F. et Toval. A, "Formally Modeling and Executing the UML Class Diagram", in Rodriguez, M.J., Paderewski, P. (eds.): Proc. of the V Workshop MENHIR, Universidad de Granada, Spain, March 2000.

[ATH00] Aleman. J.L.F., Toval. A., et Hoyos. J.R, "Rigorously Transforming UML Class Diagrams", in Rodriguez, M.J., Paderewski, P. (eds.): Proc. of the V Workshop MENHIR, Universidad de Granada, Spain,2000.

[AM94] Attoui. A. et Maouche A., "A tool for Parallel Applications Designs", in Proceedings 2nd Euromicro Workshop on Parallel and Distributed Processing, Malaga, Spain, 1994.

[AS94] Attoui. A. et Schneider, M., "A Formal Approach for Prototyping Distributed Information Systems", in Proc. IEEE International Workshop on Rapid Prototyping, Grenoble, France, June 21-23, 1994.

[AS96] Attoui. A. et Schneider. M., “A Formal Approach to the Specification and the Behavior Validation of Real-Time Systems Based on Rewriting Logic”, J. of Real Time Systems, Kluwer Academic Publishers, vol. 10, pp. 5-22, 1996.

[Bal97] Balarin. F. et al., “Hardware-Software Co-Design of Embedded Systems”, Kluwer Academic Publishers, 1997.

[BB91] Benveniste. A. et Berry G., “The synchronous approach to reactive and real-time systems”, Proc. of the IEEE, 79(9):1270-1282, 1991.

[BBE01] Boniol. F., Bel G., et Ermont J., “Modélisation et vérification de systèmes intégrés asynchrones: étude de cas et approche comparative”, 9 ième conférence internationale sur les systèmes temps-réel, RTS’01, Mars 2001.

[BBL+97] Beer. I., Ben-David S., et Landver A., “RuleBase: an Industry-oriented Formal Verification Tool”, Proc. Design Automation, Conference (DAC'96), pp. 665-660, 1996.

[BCG99] Benveniste A., Caillaud B., et Le Guernic P., “From synchrony to asynchrony”, 10th International Conference on Concurrency Theory, LNCS, vol. 1664, pp.162-177, 1999.

[BCM+92] Burch J.R., Clarke E.M., McMillan K.L., et Dill D.L.. “Symbolic model checking: 10^{20} states and beyond. Information and Computation”, 98:142-170, 1992.

[Be96] Betriebliches Lastenheft ffur FunkFahrBetrieb, Stand 1.10., German Railways, 1996.

[BHR84] Brookes D.D., Hoare CAR, et Roscoe A.W., “A theory of Communicationg Sequential Processes”, J. of the ACM, 31 (3), pp. 560-599, 1984.

[Bir35] Birkhoff G., “On the Sructure of Abstract Algebras”, in Proc. Of the Cambridge Philosophical Society, 31, 433-454.

[BJM00] Bouhoula A., Jouannaud J-P., et Meseguer J., “Specification and proof in membership equational logic”, Theoretical Computer Science, 236:35-132, 2000.

[BJO00] Bjorkander M., “Real-Time Systems in UML and SDL, Embedded System Engineering”, October/November 2000, (http://www.telelogic.com).

[BK85] Bergstra J. A. et Klop. J. W., “Algebra of communicating processes with abstraction”, Theoretical Comput. Sci., 37(1):77{121, 1985.

[BKK+96] Borovansky P.,. Kirchner C, Kirchner H., Moreau P-E, et Vittek M., “ELAN: A logical framework based on computational systems”, in José Meseguer (eds), Proc. WRLA’96, vol. 4 of ENTCS, pp. 35-50. Elsevier, 1996.

[BM88] Boyer R.S. et Moore J.S., “A Computational Logic Handbook”, Academic Press,Boston, 1988.

[Bra95] Brayton R.K., “VIS: A System for Verification and Synthesis”, Technical Report UCB/ERL M95, University of Berkley, December 1995.

[BRB90] Brace K.L., Rudell R.L. et Bryant R.E, “Efficient Implementation of a BDD Package”, 27th ACM/IEEE Design Automation Conference, pp. 40-45, 1990.

[Bry86] Bryant R. E., “Graph-based algorithms for Boolean function manipulation”, IEEE-Transactions on Computers, C-35(8):677-691, Aug, 1986.

[Bry91] Bryant R. E., “On the complexity of VLSI implmentations and graph representations of boolean functions with application to integer multiplication”, IEEE Transactions on Computers 40, 2, 205-213, 1991.

[Bry92] Bryant R.E, “Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams”, ACM Computing Surveys, vol.24(3), pp.293 –318, Sep. 1992.

[Buc77] Buchi, J. R., “Using determinacy of games to eliminate quantifiers, in Fundamentals of Computation Theory”, LNCS, vol. 56, pp. 367-378, 1977.

[Cad99] Cadence, USA., “Formal Verification Using FormalCheck, Version 2.3”, 1999.

[CC77] Cousot P. et Cousot R., “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints”, in the 4th ACM Symp. Princ. of Prog. Lang., pp. 238-252, ACM Press, 1977.

[CCL+99] Chauhan P., Clarke E., Lu Y., et Wang D., “Verifying IP-Core based System-On-Chip Designs”, in Proc. of the IEEE ASIC Conference, September 1999.

[CDE+04] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Mart´-Oliet, J. Meseguer, et C. Talcott., Maude manual (version 2.1). http://maude.cs.uiuc.edu/manual/, 2004.

[CDE+99] Clavel M., Duran F., Eker S., Lincoln P., Mart-Oliet N., Meseguer J., et Quesada J.. “Maude: Specification and programming in rewriting logic”, Computer Science Laboratory, SRI International, January 1999.

[CDE+00] Clavel M., Duran F., Eker S., et Meseguer J., “Building equational proving tools by reflection in rewriting logic”, in Cafe: An Industrial-Strength Algebraic Formal Method, Elsevier, 2000. http://maude.csl.sri.com.

[CDS+00a] Clavel M., Duran F., Eker S., Lincoln P.,. Mart-Oliet N, Meseguer J., et. Quesada J.F. , Maude tutorial, March 2000. March 25, 2000. http://maude.csl.sri.com/tutorial.

[CDS+00b] Clavel M., Duran F, Eker S, Lincoln P., Mart-Oliet N, Meseguer J., et Jose F. Quesada., “Towards Maude 2.0”, in 3rd WRLA’00, vol. 36 of ENTCS, Elsevier, 2000.

[CDS+99] Clavel M., Duran F., Eker S., Lincoln P., Mart-Oliet N., Meseguer J., et. Quesada J. F. “Maude: Specification and programming in rewriting logic”, January 1999. http://maude.csl.sri.com/manual.

[CE81] Clarke E. et Emerson E.A., “Design and synthesis of synchronization skeletons using branching-time temporal logic”, in Workshop on Logic of Programs, *LNCS 13*1, 1981.

[CES86] Clarke E.,. Emerson E., et Sistla A., “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic”, in Proc. of ACM Transactions on Programming Languages and Systems, 8(2):244-263, April 1986.

[CH78] Cousot P. and Halbwachs N., “Automatic discovery of linear restraints among the variables of a program”, in 5th ACM Symp, Princ. of Prog. Lang., pp. 84-97, Jan. 1978.

[Che80] Chellas B. F., “Modal Logic: an Introduction”, Cambridge University Press,1980.

[CHP+91] Caspi P., Halbwachs N., Pilaud P., et Raymond P.., “Programmation et vérification des systèmes réactifs : le langage LUSTRE”, *TSI*, vol.10(2), pp.139-158, 1991.

[CLA00] Clarck R.G., et Moreira, A.M.D., “Use of E-LOTOS in Adding Formality to UML, Journal of Universal Computer Science”, vol.6 (11), pp. 1071-1087, 2000.

[Cla01] Clavel M., “The ITP tool”, in A. Nepomuceno, J. F. Quesada, et F. J. Salguero, (eds), Logic, Language, and Information : Proc. of the First Workshop on Logic and Language, pp. 55–62. Kronos, 2001.

[CM96] Clavel M. et Meseguer J., “Reflection and strategies in rewriting logic”, in J. Meseguer, (ed) : Proc. Of the First Intl. Workshop on Rewriting Logic and its Applications, 4 of ENTCS, Elsevier, 1996.

[CYB98] Chen T. A., Yang B., et Bryant R. E., “Breadth-First with Depth-First BDD Construction: A hybrid Approach.”, Carnegie Mellon University, CMU-98-120.

[Dam96] Dams D. R., “Abstract Interpretation and Partition Refinement for Model Checking”, PhD thesis, Eindhoven University ofTechnology, July 1996.

[DBR96] R. Drechsler, B. Becker, and S. Ruppertz, “K*BMDs: A New Data Structure for Verification”, in Proc. European Design & Test Conference, pp. 2–8, 1996.

[DCE+98] Duran F, Clavel M, Eker S, et Meseguer J., “Building equational proving tools by

reflection in rewriting logic", in Proc. of the CafeOBJ Symposium '98, 1998.

[DDL99] Dellacherie S., Devulder S., et Lambert J.-L., "Software Verification Based on Linear Programming", in: J.M. Wing, J. Woodcock, J. Davies (Eds.): FM'99 Proc., Vol. II, Springer LNCS 1709, 1999.

[Der87] Dershowitz N., "Termination of rewriting", J. of Symbolic Computation, vol. 3, pp.69-116,1987.

[Dic91] Dick A. J., "An introduction to Knuth-Bendix completion", The Computer Journal, 34(1):2-15, 1991.

[DJ90] Dershowitz N. et Jouannaud J.-P., "Rewrite systems", in J. van Leeuwen, (ed), Handbook of Theoretical Computer Science, vol. B, chap. 6, Elsevier, Amsterdam, 1990.

[DP60] Davis M. et Putnam H., "A Computing Procedure for Quantification Theory", J. of the ACM for Computing Machinery, vol. 7, pp. 201-215, July 1960.

[DPP03] Dovier, Piazza, et Policriti., "Fast Bisimulation Algorithm", CAV 2001, Theoretical Computer Science, 2003.

[Dur99] Durch R.., "A Reflective Module Algebra with Applications to the Maude Language", PhD thesis, University of Malaga, 1999.

[DV01] Duran F. et Vallecillo A., "Writing ODB Entreprise Specifcations in Maude", http://www.lcc.uma.es/ãv/Publicaciones/01/ITI-2001-8-pdf

[EC80] Emerson E. A. et Clarke E. M., "Characterising correctness properties of parallel programs using fixpoints", in 7th ICALP, Proc., LNCS vol. 85, pp. 169-181, 1980.

[EC82] Emerson E.A. et Clarke E., "Using Branching-time Temporal Logic to Synthesize Synchronization Skeletons", Science of Computer Programming, 2:241-266, 1982.

[EH00] Etessami K et Hotzmann G.J., "Optimizing Buchi Automata," in CONCUR 2000, LNCS 1877, pp. 153–167, 2000.

[Eme83] Emerson E. A., "Alternative semantics for temporal logics", in J. TCS, vol. 26, pp. 121-130, 1983.

[Eme96] Emerson E. A., "Automated temporal reasoning about reactive systems", in Logics for Concurrency, Structure versus Automata, LNCS 1043, Springer, pp. 41-101, 1996.

[EMS02] Eker S., Meseguer J., et Sridharanarayanan A., "The Maude LTL model checker", in Proc. Of the WRLA'02, ENTCS71, Elsevier, 2002.

[EN97] EN 50128 CENELEC, "Railway Application – Software for Railway Control and Protection Systems", 1997.

[Ern98] Ernst R., "Codesign of Embedded Systems: Status and Trends", J. of IEEE Design & Test of Computers, pp. 45-54, April-June 1998.

[FD98] Futatsugi K. et Diaconescu R.., "CafeOBJ Report", World Scientific, AMAST Series, 1998.

[FGK+96] Fernandez C., Garavel H., Kerbrat A., Mateescu R., Mounier L., et Sighire M., "CADP Caesar Aldebaran Development Package Protocol Validation and Verification Toolbox", in proc. CAV'96, LNCS Springer Verlag, 1996.

[FH98] Foxand C. J. et Harman N. A., "Algebraic models of correctness for microprocessors", Technical Report CSR 8-98, University of Wales Swansea, 1998.

[FM90] J.C. Fernandez et L. Mounier., "Verifying bisimulations on the fly", in J. Quemada, J. Manas et E. Vasquez, (eds), Proc. FORTE '90, North Holland, Amsterdam, 1990.

[FT99] Fernandez J. L. and Toval A., "A formal syntax and static semantics of UML statecharts", Technical Report, Department of Informatics: Languages and systems, University of Murcia, Spain, 1999.

[FV98] Fisler K. et Vardi M.Y., “Bisimulation Minimization in an Automata-Theoretic Verification Framework”, in Formal Methods in CAD, pp. 115-132, 1998.

[FV99] Fisler K. et Vardi M.Y., “Bisimulation and Model Checking”, in CHARME 1999.

[Ger97] Gerth R., Concise Promela Reference, 1997.

http://cm.bell-labs.com/cm/cs/what/spin/Man/Quick.html.

[Gla94] van Glabbeek R.J., “What is Branching time Semantics and why to use it?”, Report Paper, Stanford University 1993. ftp://Boole.stanford.edu/pub/DVI/branching.dvi.gz,

[GM88] Goguen J. A et Meseguer J., “Order-sorted algebra I”, Technical Report, SRI International, Stanford University, 1988.

[GM93] Gordon M J C. et Melham T F., “Introduction to HOL”, Cambridge University Press, 1993.

[Gna92] Gnaedig I., “Termination of order-sorted rewriting”, in Proc. 3rd Conference on Algebraic and Logic Programming, Pisa1, LNCS 632, pages 37-52, 1992.

[GO01] Gastin P et Oddoux D., “Fast LTL to Buchi Automata Translation”, in Thirteenth Conference on Computer Aided Verification (CAV’01), LNCS 2102, pp. 53-65, 2001.

[GO03] Graf S. et Ober I., “A real-time profile for UML and how to adapt it to SDL”, in Proc. of SDL Forum, LNCS, 2003.

[GOO03] Graf S., Ober I. et Ober I., “Timed annotations with UML”, SVERTS’03, 2003.

[GPS96] Godefroid P., Peled D. et Staskauskas M., “Using partial-order methods in the formal validation of industrial concurrent programs”, J. of IEEE Transactions on software engineering, 22(7), July 1996.

[GPV+95] Gerth R., Peled D., Vardi M. et Wolper P., “Simple on-the-fly Automatic Verification of Linear Temporal Logic”, in Protocol Specification Testing and Verification, pp. 3-18, 1995.

[GUE00] LE GUENNEC A., “Méthodes formelles avec UML : Modélisation, validation et génération de tests”, Actes du 8è Colloque Francophone sur l’Ingénierie des Protocoles CFIP’2000, Toulouse, Editions Hermès, Paris, pp. 151-166, 17-20 octobre 2000.

[GW88] Goguen J. A et Winkler T., “Introducing OBJ3”, Technical report, SRI, Computer Science Lab, 1988.

[GW93] Godefroid P. and Wolper P., “Partial-order methods for temporal verification”, CONCUR '93, LNCS 715, pp. :233-246, August 1993.

[Hal93] Halbwachs N., “Synchronous programming of reactive systems”, Kluwer Academic Pub., 1993.

[Har00] Harman N. A. “Verifying a simple pipelined microprocessor using maude”, Technical Report; Computer Science Report, University of Wales Swansea, 2000.

[Har87] Harel D., "Statecharts: a Visual Formalism for Complex Systems," Science Computer Program, Vol8, pp 231-274, 1987.

[Har88] Harel D., "On Visual Formalism", Communications of the ACM, Vol 31, No 5, May 1988.

[Has97] Hasbani A., "Contribution à la Conception d’un Environnement de Spécification Formelle et de Validation des Systèmes Réactifs”, Thèse de Doctorat, Université Blaise Pascal, Clermont II, France.

[HC72] Hughes, G. E. et Cresswell, M. J., “An Introduction to Modal Logic”, 2nd ed., Routledge, 1972

[HCR+91a] Halbwachs N., Caspi P., Raymond P. et Pilaud D., “The synchronous data flow programming language Lustre", Proc. of the IEEE, 79(9), septembre 1991.

[HCR+91b] Halbwachs N., Caspi P., Raymond P. et Pilaud D., “Programmation et verification des systèmes réactifs à l'aide du langage et de données synchrone Lustre", TSI, 10(2), 1991.

[Hei87] Heite M., "Hood une méthode de Conception Hiérarchisée Orienté-objet pour le Développement de Gros Logiciels Techniques Réel", Bigre N° 57, Journées ADA France, Dec 1987.

[HHW97] T. A. Henzinger, P.-H. Ho, et H. Wong-Toi., “HyTech: A model checker for hybrid systems. Software Tools for Technology Transfer”, vol. 1, pp. 110–122, 1997. http://www-cad.eecs.berkeley.edu/Utah/HyTech/.

[Hil93] HILL D. R. C., "Analyse Orienté Objet et Modélisation par Simulation", Ed.Addison-Wisley, France, Juillet 1993.

[HKP97] Huet G., Kahn G., et Paulin-Mohring C., “The Coq Proof Assistant”, A Tutorial Version 6.1, INRIA, France, 1997.

[HM85] Hennessy, M. et Milner, R., “Algebraic laws for nondeterminism and concurrency”, in Journal of the ACM, vol. 32, 1985, pp. 137-162

[HNS+94] Hezinger T.A., Nicollin X., Sifakis J. et Yovine S., “Symbolic model checking for real-time systems.” *Information and Computatio*n, 111(2) :193-244, 1994.

[Hoa69] Hoare, C. A. R., “An axiomatic basis for computer programming”, in Communications of the ACM, vol. 12, 1969, pp. 576-580.

[Hoa78] Hoare C.A.R.., “Communicationg sequential processes”, in Communication of the ACM, 21:666-677, 1978.

[Hoa85] Hoare C. A. R., “Communicating Sequential Processes”, Prentice-Hall International, 1985.

[Hol97] Holzmann G.J., “The Model Checker SPIN”, IEEE Transactions on Software Engineering, 23:279-295, May 1997.

[Hop71] J. E.Hopcroft., “An n log n algorithm for minimizing states in a finite automaton”, in Theory of Machines and Computations, Academic Press.1971.

[HPY97] Holzman G, Peled D., et Yannakakis M., “On nested Depth First Search”, Design: An International Journal vol.13, pp. 289-307, 1998.

[HR00] Havelund K. et Rosu G., “Testing linear temporal logic formulae on finite execution traces”, 2000. http://citeseer.nj.nec.com/havelund00testing.html

[HT96] Harman N A et Tucker J V., “Algebraic models of microprocessors: Architecture and organisation”, Acta Informatica, vol.33, pp. 421-456, 1996.

[Hue80] Huet G., “Confluent Reductions : Abstract Properties and Applications to Ter Rewriting Systems”, J. of the ACM, vol..27, n°.4, pp.797-821, 1980.

[Hue81] Huet G., “A Complete Proof of Correcteness of the Knuth-Bendix Completion Algorithm”, J. of CSS, vol. 23, pp.11-21, 1981.

[Hul80] Hullot J. M., “Canonical Forms and Unification”, in Proc. Fifth Conf. On Automated deduction, Les Arcs, France, pp.318-334, 1980.

[HyTech] http://www-cad/eecs.berkeley.edu/~tah/HyTech.

[ISY91] Ishiura N., Sawada H., and Yajima S., “Minimization of Binary Decision Diagrams Based on Exchanges of Variables”, in Proc. Of the (ICCAD-91, pp. 472–475, 1991.

[ITU01] ITU-T, "Specification and Description Language (SDL)", ITU-T – Recommendation Z.100 Corrigendum 1, October 2001.

[ITU94] Z.100 ITU, “Specification and description language (SDL)”, June 1994.

[ITU99] ITU-T "SDL combined with UML", ITUtT – Recommendation Z.109, November 1999.

[ITU-T99] "Message Sequence Charts (MSC-2000)", ITU-T – Recommendation Z.120, November 1999.

[JAR98] Jard C., Jazquel J.-M., et Pennanac'H F., "Vers l'utilisation d'outils de validation de protocoles dans UML", TSI, vol. 17, n°9, pp. 1083-1098, Hermès, Paris, 1998.

[Java] Java 2 Platform Enterprise Edition. http://java.sun.com/j2ee

[JS00] Jansen L., et Schneider J., "Traffic Control System case Study; Problem description and a note on Domain-based Software Specification", Technical report,2000.

[Kah74] Kahn G., "The semantic of a simple language for parallel programming", in IFIP 74, North Holland, 1974.

[Kat99] Katoen, J.-P., "Concepts, Algoritms, and Tools for Model Checking", 1999.

[KB70] Knuth D. E. et Bendix P. B., "Simple word problems in universal algebras", in J. Leech, (ed), Computational Problems in Abstract Algebra, pp. 263-297. Pergamon Press, Oxford, 1970.

[KC99] Kim S.-K. et Carrington D., "Formalizing the UML class diagram using Object-Z", in R. France and B. Rumpe, (eds), Proc., UML, Fort Collins, USA, LNCS 1723, pp. 83-98, Springer, 1999.

[Kle52] Kleene, S. C., "Introduction to Metamathematics", North-Holland Edition, 1952.

[Klo92] Klop J. W., "Term rewriting systems", in S. Abramsky, D. Gabbay, and T. Maibaum, (eds), Handbook of Logic in Computer Science, vol. 2, pp. 1-116. Oxford University Press, 1992.

[KM96] Kaufmann M.et Moore J.S., "ACL2 an Industrial Strenght of Nqthm", in Conf. COMPASS'96, pp. 23-34, IEEE, Computer Society Press, 1996.

[KNR02] Knapp A., Merz S. et Rauh C., "Model checking – Timed UML State Machines and Collaborations", 2002.

[Koz83] Kozen, D., "Results on the propositional mu-calculus", in TCS, vol. 27, pp. 333-354, 1983.

[KP84] Kozen, D. et Parikh, R., "A decision procedure for the propositional mu-calculus, in Logics of Programs", Workshop, CMU University, 1983, LNCS. 164, Springer-Verlag, 1984, pp. 313-325.

[KP99] King P. et Pooley R., "Using UML to derive stochastic Petri net models", in N. Davies and J. Bradley, editors, UKPEW '99, University of Bristol, July 1999.UK Performance Engineering Workshop.

[KRONOS] http://www-verimag.imag.fr/TEMPORISE/kronos.

[KS80] Kannellakis P. et Smolka S., "CCS Expressions", Informationand Computation, 86:43–68, 1990.

[KW00] Kosiuczenko P. et Wirsing M., "Towards an Integration of Message Sequence Charts and Timed Maude", J. of Integrated Design & Process Science, Vol. 5(1), 2001.

[KW96] Kosiuczenko P. et Wirsing M., "Timed Rewriting Logic", in Proc. of the Third AMAST, Salt Lake City, Utah, pp. 242-257, March 6-8, 1996.

[KW97a] Kosiuczenko P. et Wirsing M., "Timed Rewriting Logic with Applications to Time-Sensitive Systems", H. Schwichtenberg (ed.): Logic of Computation, Computer and System Science, Vol. 157, pp. 229-264, Springer 1997.

[KW97b] Kosiuczenko P. et Wirsing M., "Timed Rewriting Logic with an Application to Object-Based Specification Science of Computer Programming. 28(2-3), Elsevier 1997, pp.

225-246.

[Lam94] Lamport L., “The temporal logic of actions”, J. of the ACM Transactions on Programming Languages and Systems, 16(3):872-923, May 1994.

[Lar86] Larsen K. G., “Context-dependent bisimulation between processes”, PhD thesis, 1986.

[Led96] Ledru Y., “Completing Semi-Formal Specifications with Z”n in Proc. of KBSE'96: 52-61, 1996.

[LGS95] Loiseaux C., Graf S., Sifakis J., A. Bouajjani, and S. Bensalem., “Property preserving abstractions for the verification of concurrent systems”, Formal Methods in System Design, 6:1-35,1995.

[LM00] Laleau R. et Mammar A., “An overview of a method and its support tool for generating B specifications from UML Notations”, in Proc. of the 15th Int. Conf. on Automated Software Engineering, ASE’2000, Grenoble, France, September 2000.

[LN00] Leucker M. et Noll T., “Rewriting Logic as a Framework for Generic Verification Tools”, ENTCS 36, 2000.

[LPS99] Laroussinie F., Petit A., et Schnoebelen P., “Vérification de logiciels : Techniques et outils du model-checking, Vuibert éditions, 1999.

[LPY97] Larsen K, Pettersson P., Yi W., “UPPAAL”, in a nutshell. International Journal on Software Tools for Technology Transfer, 1(1-2) :134-152, 1997.

[Les94a] Lescanne P., "Elementary Interpretations in Proofs of Termination", Technical Report, Centre de Recherche en Informatique (CNRS), INRIA-Loraine, 1994.

[Les94b] Lescanne P., "An introduction to ORME", Rapport Technique disponible sur ftp.loria.fr:/pub/loria/ORME.LIFHT, 13 Avril, 1994.

[Mar00] Marcano R., “Formalizing Patterns Applicability: An approach based on UML and B”, in Proc. of ASE’2000, IRISA technical report, n° PI-1353, Sept 2000.

[McG82] Mcraw J.R., “The Val language : description and analysis”, TOPLAS, 4(1), Janvier 1982.

[McM93] McMillan K. L., “Symbolic Model Checking”, Kluwer, 1993.

[McM99] McMillan K.L., “The SMV language”, Cadence Berkeley Labs, 1999.

[Mes00] Meseguer J., “Rewriting logic and Maude: a wide-spectrum semantic frameworkfor object-based distributed systems”, in S. Smith and C. Talcott, editors, Formal Methods for Open Object-based Distributed Systems IV, Kluwer, 2000.

[Mes90] Meseguer, J., “Rewriting as a unified model of concurrency”, in Proc. Concur’90, LNCS, Volume 458, pp. 384–400, 1990..

[Mes92] Meseguer J., “Conditional rewriting logic as a unified model of concurrency.” TCS, 96:73-155, 1992.

[Mes98] Meseguer J.,. “Membership algebra as a logical framework for equational specification”, in F. Parisi-Presicce, (ed), Proc. WADT'97, LNCS, pp. 18-61, 1998.

[Mil80] R. Milner., " A calculus of communicating systems", LNCS 92, 1980.

[Mil89] Milner. R., "Communication and Concurrency", International Series in Computer Science, Prentice Hall, 1989.

[MLL00] Marcano R., Levy N. et Losavio F., “Spécification et Spécialisation de Patterns en UML et B", in Actes LMO’00 – Langages et Modèles à Objets, Ed. Hermès. – Montréal (Ca), janvier 2000.

[MM96] Marti-Oliet N. et Meseguer J., “Rewriting logic as a logical and semantics Framework ”, Workshop on Rewriting Logic and its Applications, vol. 4 of ENTCS, Elsevier, 1996.

[Mou92] Mounier. L., "Méthodes de vérification de spécifications comportementales", étude et mise œuvre, Thèse, Université Joseph Fourier, Grenoble, janvier 1992.

[MP92] Manna Z. et Pnueli A., "The Temporal Logic of Reactive and Concurrent System's Specification", Springer-Verlag, 1992.

[MP95] Manna Z. et Pnueli A., "Temporal Verification of Reactive Systems: Safety", Springer-Verlag, New York, 1995.

[MRS01] Moller M. O., RueB H. et Sorea M., "Predicate Abstraction for Dense Real-Time Systems", BRICS Report Series, RS-01-44, ISSN 0909-0878, November 2001.

[MS00] Marques-Silva J.P. et Sakallah K.A.., "Boolean Satisfiability in Electronic Design Automation", DAC, pp. 675–680. 2000.

[MW85] Mellor S. et Ward P., "Structured Development for Real Time Systems", Youdon Press, Prentice Hall, New York, USA, 1985.

[NPW81] Nielsen M., Plotkin G. et Winskel G., "Petri Nets, Event Structures and Domains, part 1", TCS, 13:85-108, 1981.

[Odo77] O'Donnell M.J., "Computing in Systems Described by equations", in LNCS 58, 1977.

[Odo85] O'Donnell M.J, "Equational Logic as a programming Language", The MIT Press, Cambridge MA, 1985.

[OKW96] Olveczki P., Kosiuczenko P. et Wirsing M., "Steamboiler specification problem: an algebraic object-oriented solution", in J. Abrial, E. Boerger, H. Langmaack (eds.): LNCS 1165, Springer, 1996, 17 pages.

[Olv02] Olveczky P.C. "Formal modeling and analysis of distributed systems in Maude", Lecture notes for INF220, University of Oslo, Department of Informatics, June 2002. http://www.ifi.uio.no/inf220/Kompendium/komp2.pdf.

[Olve00] Olveczky P. C., "Specification and Analysis of Real-Time and Hybrid System", in Rewriting Logic. PhD thesis, University of Bergen, 2000.. http://maude.csl.sri.com/

[OM01a] Olveczky P. C. et Meseguer J., "Specification of real-time and hybrid systems in rewriting logic", disponible au at http://maude.csl. sri.com/papers, January 2001.

[OM01b] Olveczky P. C. et Meseguer J.. "Specifying and analyzing real-time object systems in Real-Time Maude", in P. Pettersson and S. Yovine, editors, Proc. Workshop on Real-Time Tools, Aalborg University, Denmark, 2001, 2001. Technical report 2001-14, Department of Information Technology, Uppsala University.

[OMG01] OMG. – Object Constraint Language Specification. – in : OMG Unified Modeling Language Specification, Version 1.4, September 2001. Object Management Group, Inc., Framingham, Mass., Internet http://www.omg.org, 2001.

[OMG99a] OMG. UML notation guide version 1.3. http://www.rational.com/uml/, June 1999.

[OMG99b] OMG, Unified Modeling Language Specification Version 1.3, Object Management Group http://www.omg.org/technology/uml/index.htm, 1999.

[OMG99c] OMG GROUP, UML Profile for Scheduling, Performance, and Time, Request for Proposal, 1999, ftp://ftp.omg.org/pub/docs/ad/99-03-13.doc.

[ORS92] Owre S., Rushby J., et Shankar N., "PVS: A prototype veri½cation system", in Automated Deduction CADE-11, vol. 607 of LNAI, pp. 748-752, 1992.

[Par81] Park D., "Concurrency and automata on infinite sequnces", in Proceedings of the 5th GI Conference, LNCS 104, pp. 167-183. Springer-Verlag, 1981.

[Pen88] Penczek W., "A temporal logic for event structures", Fundamenta Informatica, 11(3):297-326, 1988.

[PH85] Pnueli A. et Harel D., "On the Development of Reactive Systems", vol. F13 of NATO

ASI, pp. 477-498, Springer-Verlag Berlin Heidelberg, 1985.

[Plo77] Plotkin G.D, "LCF as a Programming Language", TCS 5, 223-257.

[Pnu77] Pnueli A., "The temporal logic of programs", in Proc. 18th IEEE Symp. Found. of Comp.Sci., pages 46-57. IEEE Computer Society Press, 1977.

[Pnu85] Pnueli A., "Linear and Branching Structures in the Semantics and Logics of Reactive Systems", in Proc. Of the 12th ICALP, LNCS 194, pp. 15-32, 1985.

[Pnu86] Pnueli A., "Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends", in W.-P. de Roever and G. Rozenberg, editors, Current trends in Concurrency: Overviews and Tutorials, vol. 224 of LNC, Springer-Verlag, NewYork, N.Y., 1986.

[Pra86] Pratt V. R., "Modeling concurrency with partial orders", International Journal of Parallel Programming, 15(1):33-71, February 1986.

[Pra95] Pratt V.R, "Anatomy of the Pentium Bug", in P. D. Mosses, M. Nielsen, and H. I. Schwartzbach, (ed), TAPSOFT'95, vol. 915 of LNCS, pp. 97-107, Springer, 1995..

[PT87] Paige R. et Tarjan R.E., "Three partition refinement algorithms", SIAM Journal on Computing, 16(6):973–989, 1987.

[QS82] Queille J. et Sifakis J., "Specification and verification of concurrent systems in CESAR", in M. Dezani-Ciancaglini and U. Montanari, (eds), Intl. Symposium on Programming, vol. 137 of LNCS, pp. 337-351, Springer-Verlag, 1982.

[Rat92] C. Ratel., "Définition et réalisation d'un outil de vérification formelle de programmes LUSTRE : le système LESAR, Thèse de doctorat, INPG, Grenoble. Juillet 1992.

[RBJ+03a] Rebaiaia M.L, Benmohamed M., Jaam J.M. et Hasnah Ahmad, "A Toolset for the Specification and Verification of Embedded Systems", in Hamid R. Arabnia, Youngsong Mun (Eds.): Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '03, June 23 - 26, 2003, Las Vegas, Nevada, USA, Vol. 4. pp. 1539-1545, CSREA Press 2003, ISBN 1-892512-44-0.

[RBJ+03b] Rebaiaia M.L, Benmohamed M., Jaam J.M. et. Hasnah A, "A Rewriting Logic-Based Computation and Deduction Approach to Avoid Reactive System Malfunctions", in Hamid R. Arabnia, Youngsong Mun (Eds.): Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '03, pp. 573-579, June 23 - 26, 2003, Las Vegas, Nevada, USA, Vol. 4. CSREA Press 2003, ISBN 1-892512-44-0.

[RBP91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. "Object-Oriented Modeling andDesign", Prentice Hall, Englewood Cliffs, 1991.

[Reb00] Rebaiaia M. L., "Specification of Real-Time Systems in Timed Rewriting Logic", In Proc. International Arab Conference of Information Technology, Zarka, Jordan, October, 2000.

[Reb04a] Rebaiaia M.L., "Embedded Systems Certification using Temporal Logic", in the Proc. of IEEE International Conference, Information and Communication Technologies: From Theory to Applications, pp. 577- 578, Syria 19-23 April 2004,. IEEE : ISBN: 0-7803-8482. http://www.ieeexplore.ieee.org/xpl/.

[RBJ+03c] Rebaiaia M.L, Benmohamed M., Jaam J.M et Hasnah A., "Symbolic Simulation: Toward and Approach Using Prepositional Logic Integer Programming and Neural Networks", Proc. of, ACIT'03, Alexandria, Egypt, 2003.

[Reb02a] Rebaiaia M.L., "Real-Time Systems Specification in Timed Rewriting Logic," Zarka Journal for Research and Studies, Vol 4, N°2, pp. 37-53, December 2002, ISSN 1561-9109.

[RBA+01] M.L Rebaiaia, Benfifi K., Abdellaoui F., Rezki B.et Moustari S, "VERIF: an Automatic Specification and Verification Tool for Industrial and Reactive Systems", Proc. of the International ACIT'01, Jordan, 2001..

RJH03 Rebaiaia M. L., Jaam J. M et Hasnah A, "A Decision Support System Based on Ranking Methods", in H. R. Arabnia,(Ed.),Proc. of the International Conference on Information and Knowledge Engineering. IKE'03, June 23 - 26, 2003, Las Vegas, Nevada, USA, Vol. 2. pp. 591-597, CSREA Press 2003, ISBN 1-932415-08-4.

[RG00] Richters M. et Gogolla M., " Validating UML Models and OCL Constraints", in: Proc. of UML 2000: 3th Int. Conf. The Unified Modeling Language, York, UK. Springer, Berlin, LNCS 1939, 2000.

[Rit00] Ritter G., "Sequential equivalence checking by symbolic simulation", in FMCAD2000, vol. LNCS 1954, Springer, November 2000.

[RJ04a] Rebaiaia ML. et Jaam J.M., "VALID-2: A Practical Modeling, Simulation and Verification Software for Distributed Systems", in the Workshop PMEO-PDS'04, IEEE-IPDPS 2004, San Diego, USA, ISBN : 0-7695-2132-IEEE-2004

[RJ04b] Rebaiaia ML. et Jaam J.M, "A Hierarchical Approach to the Formal Verification of Distributed Systems", in Proc. of the Internationnal Conference IEEE-ICICS, King Fahd University, 29/11/ au 01/12, 2004, Dharhan, Seoudia Kingdom, 2004.

[Reb04b] Rebaiaia M.L, "A Deductive Approach to Verify e-commerce Protocols", iIn the Proc. of the Internationnal Conference IEEE-ICICS, King Fahd University, 29/11/ au 01/12, 2004, Dharhan, Seoudia Kingdom, 2004.

[RBJ04b] Rebaiaia ML., Benmohamed M, et Jaam J.M, "Improving PTL Model-Checker using Maude Rewriting Logic", in : Proc. of the SETIT'04 International Conference, Sousse, Tunisie, 2004, ISBN : xxxx-xxxx.

[RJH03a] Rebaiaia M.L, Jaam M. J et Hasnah A., "An Efficient Model Checker Based on the Axiomatization of Propositional Temporal Logic in Rewriting Logic", in : the Proc. of the 10th IEEE International Conference on Electronics, Circuits and Systems, pp. 866-869, December 14-17, 2003 - Sharjah - United Arab Emirates, IEEE Catalog Number: 03EX749C. ISBN : 0-7803-8164-5.

[RJH03b] Rebaiaia M.L, Jaam M. J et Hasnah A., "A Neural Network Algorithm for Hardware-Software Verification", in : the Proc. of the 10th IEEE International Conference on Electronics, Circuits and Systems, pp. 1332-1335, December 14-17, 2003 - Sharjah - United Arab Emirates. ISBN : 0-7803-8164-5.

[RB03] Rebaiaia M.L et Benmohamed M., "A Specification and Verification Software for Critical Systems", in : Proc. of the Second International Conference on Signals Systems and Information Technology SSD'03, March 26-28-2003, Sousse, Tunisia..

[Reb03b] Rebaiaia M.L, "Mixing Propositional Logic, Integer Programming and Neural Network for the Hardware/Software Verification", In : Proc. of the Second International Conference on Signals Systems and Information Technology SSD'03, March 26-28-2003, Sousse, Tunisia..

[Reb03c] Rebaiaia M.L, "Modelling Real-Time Systems using Rewriting Logic", In : Proc. of the Second International Conference on Signals Systems and Information Technology SSD'03, March 26-28-2003, Sousse, Tunisia..

[RB02] Rebaiaia M.L et Benmohamed M., "Verification Algorithms for Integer Programming and Genetic Algorithms", in : Proc. of International Arab Conference of Information Technology, ACIT'02, Doha, Qatar, 2002.

[Reb02b] Rebaiaia M.L, "A Verification Tool Based on Integer Programming and Genetic Algorithms", In : Proc. of the Algerian Conference on Microelectronics, ACM'02, pp.53-61, October 13-15, Algiers, Algeria, 2002.

[RBJ+03b] Rebaiaia ML., Benmohamed M, Jaam J.M et Hasnah A., “Verification Algorithms for Integer Programming and Genetic Algorithms”, In the International Journal of Applied Sciences & Computations, Vol. 10, N°3, December, 2003, pp.191-204. ISSN : 1089-0025.

[Reb05] Rebaiaia M.L, “Monitoring and Reuse Software Patterns Analysis in Maude”, to be appreared in the Proc. of the IEEE-AICCSA Workshop in Untimed Patterns Analysis, to be held in American University, CAIRO, Egypt, 3 January, 2005.

[RKT+00] Roubstova E.E., van Katwijk J., Toetenel W.J., Ponk C. et de Rooij R.C.M., “Specification of Real-Time Systems in UML”, ENTCS 39, 2000.

[Rob65] Robinson J.A., “A Mache-Oriented Logic Based on the Resolution Principle”, Journal of the ACM, pp. 23-41, 216-226.

[RSS96] RueB H, Shankar N. et. Srivas M. K., “Modular verification of SRT division”, in R. Alur and T. A. Henzinger, (eds), CAV '96, LNCS 1102, pp. 123-134. Springer-Verlag, 1996.

[Rud93] Rudell R., “Dynamic Variable Ordering for Ordered Binary Decision Diagrams”, in IC-CAD'93, IEEE Computer Society Press, 1993.

[Rus95] Russino M.D., “A Formalization of a Subset of VHDL in the Boyer-Moore Logic”, in Formal Methods in System Design, vol.7, pp. 7-25, August 1995.

[RW99] Reggio G. et Wieringa R. J., “Thirty one problems in the semantics of UML 1.3 dynamics”, Workshop Rigorous Modeling and Analysis of the UML Challenges and Limitations, OOPSLA'99, November 1999.

[Saw99] Sawada J., “Formal Verification of an Advanced Pipelined Machine”, PhD thesis, University of Texas at Austin, December 1999.

[SB00] Sommenzi F et Bloem R., “Efficient Buchi Automata from LTL formulae”, in CAV’00, LNCS 1633, pp. 247-263, 2000.

[SC85] Sistla A. P. et Clarke E. M., “The complexity of propositional linear temporal logics”, Journal of the ACM, 32(3):84–103, 1985.

[Sch04] Schütz B., “UML-RT – Solution for Embedded Software?”, TU München, Faculty for Computer Science, Boltzmannstr.3, D-85748 Garching (Munich), 2004.

[SE84] Streett R. S. et Emerson E. A., “The propositional mu-calculus is elementary”, in ICALP'84, LNCS 172, Springer-Verlag, 1984, pp. 467-472.

[Sel03] Selic B., “An Overview of UML 2.0 (and MDA)”, Tutorial presented at OTS’2003 18-19 June 2003. Available at http://lisa.uni-mb.si/cot/ots2003/predkonferenca/ .

[Sel98] Selic B., "Recursive Control", in Robert Martin, Dirk Riehle, and Frank Buschmann (ed.) "Pattern Languages of Program Design 3", Addison Wesely Longman Inc 1998, chapter 10, pp147.

[SGW94] B. Selic, G. Gullekson et Paul T. Ward., “Real-Time Object-Oriented Modeling”, John Wiley and Sons, Inc. 1994.

[SK00] Steggles J. et Kosiuczenko P., “A Formal Model for SDL Specification Based on Timed Rewriting Logic”, ASE’00, Vol.7(1), Kluver, 2000, pp. 59-88.

[SM02] Shanchez-Alonso M. et Murillo J. M., "Specifying Cooperation Environment Requirements using Formal and Graphical Techniques", WER, pp. 225-239, 2002.

[SPIN02] Spin Formal Verification, 2002. http://netlib.bell-labs.com/netlib/spin/whatspin.html.

[SR98] B. Selic et J. Rumbaugh., “Using UML for Modeling Complex Real-Time Systems”. http://www.objectime.com, 1998.

[SS99] Silva J. et Sakallah K., “GRASP: A Search Algorithm for Propositional Satisfiability ”, IEEE Transactions on Computers, pp.506 –521, May 1999.

[Ste03] Stephen F., “Temporal ocl extensions for specification of real-time constraints”, in Workshop Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS’03), San Francisco, CA, USA, October 2003. UML 2003.

[STP] The Stanford Temporal Prover. http://www-step.stanford.edu/.

[Tar55] Tarski, A., “A lattice-theoretical fixpoint theorem and its applications”, in Pacific Journal of Mathematics, vol. 5, 1955, pp. 285-309.

[TG00] Terrier, F., et Gerard, S., “Real Time System Modeling with UML: Current Status and Some Prospects”, Proc. of the 2nd Workshop of the SDL Forum society on SDL and MSC, SAM 2000, Grenoble, France, 2000.

[TKB89] Toyama Y, Klop J.W.et Barendregt H.P., "Termination for Direct sums of left-linear complete term rewriting Systems", Report CS-R8923, Centre for Mathematics and Computer Science, Amsterdam.

[TSL+90] Touati H. J., Savoj H., Lin B. et. Brayton R. K, “Implicit State Enumeration of Finite State Machines using BDDs,” in Proc of ICCAD-90, pp. 130–133, 1990.

[TY01] Tripakis S. et Yovine.S, “Analysis of timed systems using time-abstracting bisimulations", Formal Methods in System Design, 18(1):25-68, January 2001.

[UML01] UML-OMG. Unified Modeling Language Specification. Version 1.4, September 2001.

[UML03] IBM. UML 2.0 implementation efforts. Available at http://www.rational.com/uml/resources/uml2/implementations.jsp 22 July 2003.

[UPPAAL] http://www.uppaal.com

[Uri98] Uribe T. E., “Abstraction-based Deductive-Algorithmic Verification of Reactive Systems”, PhD thesis, Computer Science Department, Stanford University, Dec. 1998. Technical Report STAN-CS-TR-99-1618.

[Var87] Vardi, M. Y., “Verification of concurrent programs: the automata-theoretic framework”, in Proc. of the 2nd IEEE Symposium on Logic in Computer Science, 1987, pp. 167-176, 1988, pp. 250-259.

[Vir02] Viry P., “Equational rules for rewriting logic”, TCS, 285(2):487–517, 2002.

[Vir94] Viry P., “Rewriting: An effective model of concurrency”, in C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, (eds), Proc. PARLE’94, LNCS 817, pp. 648–660. Springer, 1994.

[VMO03] Verdejo A. et Mart-Oliet N., “Executable structural operational semantics in Maude”, Technical Report 134-03, Departamento de Sistemas Informaticos y Programacion, Universidad Complutense de Madrid, 2003.

[VW86] Vardi, M. Y. et Wolper, P., “Automata-theoretic techniques for modal logics of programs”, in Journal of Computer and System Sciences, vol. 32, 1986, pp. 183-221.

[Wal93] Walukiewicz, I, “On completeness of the mu-calculus,” in Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science, 1993, pp. 136-146.

[Win86] Winskel G., “Event structures”, in W. Brauer, editor, Petri nets: central models and their properties; advances in Petri nets, LNCS, Berlin-Heidelberg-New York, 1986, Springer.

[Wir90] Wirsing M., “Algebraic specification”, in J van Leeuwen, (ed), Handbook of TCS, Vol. B: Formal Models and Semantics, pages 675 -788. Elsevier, 1990.

[WK98] Warmer J. et Kleppe A., “The Object Constraint Language”, Precise Modelling with UML, Addison-Wesley, 1998.

[WK99] Warmer J. and Kleppe A., “The Object Constraint Language, precise modeling with UML”, Addison-Wesley, Object Technology Series, 1999.

[Wol97] Wolfgang T., “Languages, automata, and logic”, in Handbook of formal languages, vol. 3,

pp. 389–455, Springer, Berlin, 1997.

[WTB+00] Wang Y. , Talpin J-P., Benveniste A. et Le Guernic P., “a semantics of UML state-machines using synchronous pre-order transition systems”, International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE Press, Mars 2000.

[You91] Coad C P et Yourdon E., “Object Oriented Analysis”, 2nd Edition, YOURDON Press Computng Series, 1991.

[Yov97] Yovine S., “Kronos: A verification tool for real-time systems”, 1997. http://www-verimag.imag. fr/TEMPORISE/kronos/.

[Yov98] Yovine. S., “Model-checking timed automata”, in G. Rozenberg and F. Vaandrager, (eds), Embedded Systems, LNCS 1494, pp. 114-152, Springer-Verlag, 1998.

[ZB96] Zhou Z. et Boulerice N., “MDG Tools (V1.0) User’s Manual”, Dept. of Information and Operational Research, University of Montreal, Montreal, Canada, 1996.

ANNEXE A

LA SIGNATURE DU MODULE DE TRANSFORMATION DU CODE VALID EN CODE MAUDE

Nous informons le lecteur que le code suivant n'est qu'une partie du code original, il peut être étendu à outrance pour prendre en charge d'autres paradygmes de la programmation. Aussi, compte tenu que Full-MAUDE a été écrit en Core-MAUDE, ce qui veut dire que nous aussi nous avons été influencé par Full-Maude ce qui par la même occasion montre qu'une partie du code écrit est similaire à celle de Full-MAUDE.

```
fmod SIGNS&VIEW-EXPRS is
 sorts Token NeTokenList Bubble
       SortToken Sort SortList SortDecl
       ViewToken ViewExp
       SubsortRel SubsortDecl
       OpDecl Attr AttrList Hook HookList .
 subsorts SortToken < Sort < SortList .
 subsort ViewToken < ViewExp .
 subsort Attr < AttrList .
 subsort Hook < HookList .
 op ((_)) : Token → Token .
 op _[_] : Sort ViewExp → Sort [prec 40] .
 op __ : SortList SortList → SortList [assoc] .
 op _,_ : ViewExp ViewExp → ViewExp [assoc] .
 *** déclaration des types
 op Types_. : SortList → SortDecl .
 op Type_. : SortList → SortDecl .
 *** déclaration des sous-types
 op Subtype_. : SubsortRel → SubsortDecl .
 op Sutypes_. : SubsortRel → SubsortDecl .
 op _<_ : SortList SortList → SubsortRel .
 op _<_ : SortList SubsortRel → SubsortRel .
 *** operator attributes
 op __ : AttrList AttrList → AttrList [assoc] .
 op assoc : → Attr .
 op com : → Attr .
 op id:_ : Bubble →Attr .
 op strat(_) : NeTokenList →AttrList .
```

```
  op strategy(_) : NeTokenList →AttrList .
  op prec_ : Token → Attr .
  op gather(_) : NeTokenList → Attr .
  op gathering(_) : NeTokenList → Attr .
  op idem : → Attr .
  *** operator declaration
  op Op_: →_. : Token Sort → OpDecl .
  op Op_: →_[_]. : Token Sort AttrList → OpDecl .
  op Op_:_→_. : Token SortList Sort → OpDecl .
  op Op_:_→_[_]. : Token SortList Sort AttrList → OpDecl .
  op Ops_: →_. : NeTokenList Sort → OpDecl .
  op Ops_: →_[_]. : NeTokenList Sort AttrList → OpDecl .
  op Ops_:_→_. : NeTokenList SortList Sort → OpDecl .
  op Ops_:_→_[_]. : NeTokenList SortList Sort AttrList → OpDecl .
endfm

fmod F&S-MODS&THS is
  including SIGNS&VIEW-EXPRS .
  sorts FDeclList SDeclList PreModule
  ImportDecl ModExp Parameter ParameterList
  ModuleName EquationDecl RuleDecl MembAxDecl
  VarDecl VarDeclList .
  subsort Parameter < ParameterList .
  subsorts Token < ModExp ModuleName .
  subsort VarDecl < VarDeclList .
  subsorts VarDecl ImportDecl SortDecl SubsortDecl
  OpDecl MembAxDecl EquationDecl VarDeclList < FDeclList .
  subsorts RuleDecl FDeclList < SDeclList .

  *** déclaration des variables
  op Vars_:_. : NeTokenList Sort →VarDecl .
  op Var_:_. : NeTokenList Sort → VarDecl .
  *** déclaration des règles
  op R[_] :_=>_. : Token Bubble Bubble → RuleDecl .
  op CRl[_]:_=>_if_. : Token Bubble Bubble Bubble → RuleDecl .

*** déclaration d'importation (*** l'importation n'existe pas en VALID nous l'avons incorporé***)
  op including_. : ModExp → ImportDecl .
```

```
  op protecting_. : ModExp →ImportDecl .
  *** déclaration des listes
  op __ : VarDeclList VarDeclList →VarDeclList [assoc] .
  op __ : SDeclList SDeclList → SDeclList [assoc] .
  op __ : FDeclList FDeclList → FDeclList [assoc] .
  *** déclaration des classes
  op class_|_. : Sort AttrDeclList → ClassDecl .
  op class_. : Sort -> ClassDecl .
  op _,_ : AttrDeclList AttrDeclList → AttrDeclList [assoc] .
  op _:_ : Token Sort → AttrDecl [prec 40] .
  *** déclaration des sous-classes
  op subclass_. : SubsortRel → SubclassDecl .
  op subclasses_. : SubsortRel → SubclassDecl .
  op _<_ : SortList SortList → SubsortRel .
  op _<_ : SortList SubsortRel → SubsortRel .
  *** déclaration de message
  op msg_:_→_. : Token SortList Sort → MsgDecl .
  op msgs_:_→_. : NeTokenList SortList Sort → MsgDecl .

  *** Déclaration du Module formel
  op moduleFormel_is_} : ModuleName FDeclList → PreModule .
endfm

fmod COMMANDS is
  including MOD-EXPRS .
  sorts PreCommand .
  *** down function
  op down_:_ : ModExp PreCommand → PreCommand .
  *** reduce command
  op red_. : Bubble → PreCommand .
  op red-in_:_. : ModExp Bubble → PreCommand .
  op reduce_. : Bubble → PreCommand .
  op reduce-in_:_. : ModExp Bubble → PreCommand .

  *** rewrite command
  op rew_. : Bubble → PreCommand .
  op rew[_]_. : Token Bubble → PreCommand .
  op rew-in_:_. : ModExp Bubble → PreCommand .
```

```
  op rew-in[_]_:_. : Token ModExp Bubble → PreCommand .
  op rewrite_. : Bubble → PreCommand .
  op rewrite[_]_. : Token Bubble → PreCommand .
  op rewrite-in_:_. : ModExp Bubble → PreCommand .
  op rewrite-in[_]_:_. : Token ModExp Bubble → PreCommand .
  *** select command
  op select_. : ModExp → PreCommand .
  *** show commands
  op show module . : → PreCommand .
  op show module_. : ModExp → PreCommand .
  op show all . : → PreCommand .
  op show all_. : ModExp → PreCommand .
  op show sorts . : → PreCommand .
  op show sorts_. : ModExp → PreCommand .
  op show ops . : → PreCommand .
  op show ops_. : ModExp → PreCommand .
  op show vars . : → PreCommand .
  op show vars_. : ModExp → PreCommand .
  op show mbs . : → PreCommand .
  op show mbs_. : ModExp → PreCommand .
  op show eqns . : → PreCommand .
  op show eqns_. : ModExp → PreCommand .
  op show rls . : → PreCommand .
  op show rls_. : ModExp → PreCommand .
  op show view_. : ViewExp → PreCommand .
  op show modules . : → PreCommand .
  op show views . : → PreCommand .
endfm

fmod FULL-MAUDE-SIGN is
  including VIEWS .
  including COMMANDS .
endfm
```

ANNEXE B

La logique propositionnelle

Les propriétés des systèmes séquentiels peuvent s'exprimer par des relations de pré- et post-conditions telles que définies par Hoare [Hoa69]. Dans une sémantique axiomatique ou bien dans un modèle de vérification, une pré-condition décrit en générale un certain ensemble d'états de départ tels que un *input*(s) désiré(s). Les post-conditions décrivent l'*output*(s). De ce fait, la paire pré- et post-condition permet de spécifier le comportement d'un programme. A titre d'exemple, le programme séquentiel {x = x + 1 ; x = x + 2} établit la post-condition « x égal 3 », une fois que la pré-condition serait « x égal 0 ». Le comportement de ce programme ne fait que transformer la valeur de la variable x de zéro vers 3, après avoir exécuté les deux instructions l'une après l'autre.

Comme tout langage de modélisation ou bien de programmation, la logique propositionnelle possède elle aussi une syntaxe et une sémantique propre à elle.

Dans ce qui suit, nous allons étudier certains fondements théoriques de cette logique, puisqu'elle constitue la base de presque toutes les logiques modales et temporelles.

4.2.1 Syntaxe de la Logique Propositionnelle

La syntaxe de la logique propositionnelle est fondée sur des *variables* appelées propositions a*tomiques* qui expriment des faits sur les états du système considéré (exemples : « *x égal à 0 », « x supérieur à 2 », « il pleut », « il n'y a plus de processus dans la file d'attente »)*. Nous dénotons par *AP* l'ensemble des *propositions atomiques* que nous notons avec des lettres majuscules ou bien des lettres quelconques au gré des convenances, possiblement avec des indices ou des primes. Ces lettres sont censées représenter des propriétés de base, qui sont soit vraies (*true*) soit fausses (*false*). Ces variables sont combinées au moyen de *connecteurs logiques* pour former des *formules* qu'on appellera *propositions* que nous noterons par les lettres grecques Φ et Ψ (avec des indices ou des primes).

On admet que pour chaque état dans le système, il est connu quelle proposition atomique sera considérée pour établir ce qu'on appelle la fonction d'interprétation définie comme suit,

Définition 4.1 (Fonction d'Interprétation) *La fonction d'interprétation* $L : S \rightarrow 2^{AP}$ *attribut à chaque état* $s \in S$ *la proposition atomique L(s) qui est valide en s. Autrement dit la fonction L indique quelle sont les propositions atomiques qui sont valides pour tout état s. Si par exemple à l'état s,* $L(s) = \varnothing$*, ceci signifie qu'aucune proposition n'est valide en s. Un état s pour lequel une proposition p est valide* ($p \in L(s)$)*, s'écrit p-state*

Définition 4.2 *Soit X un ensemble infini dit de variables de proposition. L'ensemble AP des formules propositionnelles est le plus petit ensemble contenant toutes les variables, et tel que si* Φ *et* Ψ *sont des formules, alors* $\Phi \vee \Psi$*, et* $\neg\Phi$ *sont des formules.*

En forme textuelle, nous utilisons des parenthèses et des priorités pour éliminer les ambiguïtés: $\neg$ (opérateur de négation) est plus prioritaire que $\wedge$ (opérateur de conjonction) qui l'est plus que $\vee$ (opérateur de disjonction) qui l'est plus que $\Rightarrow$. Donc la notation de la formule $\Phi\wedge\Phi'\vee\Phi''$ est exactement la même chose que d'écrire $(\Phi\wedge\Phi')\vee\Phi''$, $\neg\Phi\vee\Phi'$ représente $(\neg\Phi)\vee\Phi'$, et $\Phi\wedge\Phi'\Rightarrow\Phi''\vee\Phi'''$ $(\Phi\wedge\Phi')\Rightarrow(\Phi'\vee\Phi''')$, en particulier. De plus, $\wedge$ et $\vee$ sont associatifs à gauche, i.e. $\Phi\wedge\Phi'\wedge\Phi''$ représente $(\Phi\wedge\Phi')\wedge\Phi''$ et $\Phi\vee\Phi'\vee\Phi''$ $(\Phi\vee\Phi')\vee\Phi''$. L'opérateur $\Rightarrow$ est associatif à droite, par exemple $\Phi \Rightarrow \Phi' \Rightarrow \Phi''$ représente $\Phi \Rightarrow (\Phi'\Rightarrow\Phi'')$.

Nous définissons aussi des abréviations pour d'autres opérations. Par exemple, l'*équivalence logique* $\Leftrightarrow$ est définie par: $\Phi \wedge \Psi = \neg(\neg\Phi \vee \neg\Psi)$ et $\Phi\Leftrightarrow\Phi'$ égale $(\Phi\Rightarrow\Phi')\wedge(\Phi'\Rightarrow\Phi)$.

La relation d'équivalence sur les formules permet de regrouper celles qui possèdent la même signification.

Définition 4.3. *Par définition de la valeur d'une formule, deux formules* Φ et Ψ *sont équivalentes si et seulement si* $\Phi \Leftrightarrow \Psi$.

4.2.2 Sémantique de la Logique Propositionnelle

La sémantique des langages de programmation ou tout autre langage s'attache à donner une signification à toutes ces constructions, c'est-à-dire aux mots du langage selon des règles bien précises. Contrairement aux langages de programmation, chaque formule (mot) construite dans la logique propositionnelle est censée être soit vraie soit fausse. Formellement, l'ensemble des *valeurs de vérité* est l'ensemble $B = \{T,F\}$ des booléens, où $T \neq F$. La signification des connecteurs est définie à l'aide de fonctions des booléens vers les booléens. Ces fonctions sont représentées par des *tables de vérité*. Les voici pour quelques connecteurs de cette logique.

Par exemple, si Φ est supposé vraie (*T*(rue)), et Φ' est supposé fausse (*F(alse)*), alors $\Phi \Rightarrow \Phi'$ doit être faux, (Le premier argument est en colonne, le second en ligne dans les tables de vérité). Ceci définit des fonctions binaires ***et ou*** et ***implication*** de $\boldsymbol{B}\times\boldsymbol{B} \rightarrow \boldsymbol{B}$, et une fonction unaire ***non.***

La signification d'une formule Φ dépend des valeurs de vérité supposées des variables, et nous définissons :

Définition 4.7 *Une affectation ou interprétation* ρ *est une application de l'ensemble X des variables propositionnelles vers* $\boldsymbol{B}$.

Définition 4.8 *La sémantique* $[\Phi]\rho$ *d'une formule* Φ *dans l'affectation* ρ *est définie par récurrence structurelle sur* Φ *par :*

- $[A]\rho = \rho(A)$ *si A est une variable propositionnelle.*
- $[\Phi \wedge \Phi']\rho$ = $[\Phi]\rho$ *et* $[\Phi']\rho$,
- $[\Phi \vee \Phi']\rho$ = $[\Phi]\rho$ *ou* $[\Phi']\rho$,
- $[\Phi \Rightarrow \Phi']\rho$ = $[\Phi]\rho$ *implique* $[\Phi']\rho$,
- $[\neg\Phi]\rho$ = *non* $[\Phi]\rho$.

Nous dirons qu'une formule Φ est vraie *dans l'affectation* ρ si et seulement si $[\Phi]\rho = T$, elle est fausse *dans* ρ si et seulement si $[\Phi]\rho = F$.

Définition 4.9 *Soit* Φ *une formule propositionnelle, et* ρ *une affectation. Nous disons que* ρ *est un modèle de* Φ *ou que* ρ *satisfait, et nous écrivons* $\rho \models$, *si et seulement si* $[\Phi]\rho = T$.

Définition 4.10 *Nous disons qu'un ensemble* Γ *de formules entraîne* Φ, *et nous écrivons* $\Gamma \models \Phi$, *si et seulement si toutes les affectations satisfaisant toutes les formules de* Γ *en même temps (les modèles de* Γ*) sont aussi des modèles de* Φ, *c'est-à-dire quand* $\rho \models \Psi$ *pour tout* $\Psi \in \Gamma$ *implique* $\rho \models \Phi$.

Définition 4.11 Φ *est valide si et seulement si* Φ *est vraie dans toute affectation* $[\Phi]\rho = T$ *pour tout* ρ, *(noté* $\models\Phi$*), et est invalide sinon. Une formule propositionnelle valide est aussi appelée une* ***tautologie***.

Définition 4.12 Φ *est satisfiable si et seulement si elle est vraie dans au moins une affectation, et est non satisfiable sinon.*

Remarque 4.13 Toutes les formules valides sont satisfiables, et toutes les formules non satisfiables sont invalides. Ceci divise l'espace des formules en trois catégories : les formules valides (toujours vraies), les formules non satisfiables (toujours fausses), et les formules à la fois invalides et satisfiables (parfois vraies, parfois fausses).