République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITE MOHAMED KHIDER BISKRA

N° d'ordre : ……..

N° de Série : ……

Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie
Département d'Informatique

THÈSE

Présentée pour obtenir le diplôme de

DOCTORAT EN SCIENCES EN INFORMATIQUE
Option : Informatique

Par
BENDIAF Messaoud

THÈME

SPÉCIFICATION ET VÉRIFICATION DES SYSTÈMES EMBARQUÉS TEMPS RÉEL EN UTILISANT LA LOGIQUE DE RÉÉCRITURE

Soutenue le : 15/05/2018

Devant le jury composé de :

CHERIF Foudil	Professeur	Université de Biskra	Président
BOURAHLA Mustapha	Professeur	Université de M'sila	Rapporteur
CHAOUI Allaoua	Professeur	Université de Constantine 2	Examinateur
BENNOUI Hammadi	MCA	Université de Biskra	Examinateur

À ma mère
À mes frères et mes sœurs
À toute ma famille
À tous mes amis et collègues
À tous ceux qui m'aiment et que j'aime

Remerciements

El-Hamdou Li ALLAH le Tout Puissant pour m’avoir donné la force morale et physique pour achever cette thèse.

Mes vifs remerciements vont également à Monsieur **BOURAHLA Mustapha** d’avoir assuré l’encadrement de cette thèse ainsi que pour tous ses conseils, sa confiance, ses idées, ses encouragements, ses corrections et ses remarques tout au long de ce travail.

Je souhaite adresser mes sincères remerciements à Monsieur **CHERIF Foudil** qui m’a fait l’honneur de présider le jury de thèse, et aux personnes qui ont accepté la tâche délicate d'examiner cette thèse et qui ont eu la patience d'évaluer ce travail, Monsieur **CHAOUI Allaoua** et Monsieur **BENNOUI Hammadi.**

Ma gratitude infinie va à toute ma famille, notamment, ma mère qui m’a soutenu tout au long de mes années d’étude.

Enfin, je remercie les personnes m’ayant soutenue et encouragé pendant ces années de thèse.

RÉSUMÉ

Les systèmes embarqués temps réel doivent être correctement validés et vérifiés avant de les fabriquer et déployer afin d'augmenter leurs fiabilités et de réduire leurs coûts de maintenance. Les modèles sont utilisés depuis longtemps pour construire des systèmes complexes, dans pratiquement tous les domaines de l'ingénierie. C'est parce qu'ils fournissent une aide inestimable pour prendre des décisions de conception importantes avant la mise en œuvre du système.
Dans une activité classique d'Ingénierie Dirigée par les Modèles (IDM), les systèmes sont modélisés à l'aide d'une notation semi-formelle et sont par la suite validés puis implantés. L'étape de validation, basée sur ces modèles, est particulièrement cruciale pour les Systèmes Embarqués Temps-Réel (SETR), afin de s'assurer de leur bon fonctionnement. Cependant, une démarche IDM reste insuffisante dans le sens où elle n'indique pas comment utiliser les modèles pour appliquer l'analyse. Face à cette situation, l'intégration de méthodes formelles dans les cycles de développement de ces systèmes est devenue primordiale. Ces méthodes sont depuis longtemps reconnues afin d'aider au développement de systèmes fiables, en raison de leurs fondements mathématiques, réputés rigoureux sur l'exhaustivité de la vérification formelle qu'ils permettent de l'activer.
Dans cette thèse, nous proposons une approche basée sur la transformation de modèle pour obtenir une spécification formelle de notre système, puis utilisons les techniques de vérification formelles pour prouver que la conception de tel système est correcte par rapport à sa spécification. Nous commençons par l'utilisation de diagramme d'états-transitions (Statechart), qui décrit un système temps réel, comme modèle source pour générer un code RT-Maude, ce code représente le module Maude du système temps réel (modèle cible). La deuxième partie de notre travail est de ; en se basant sur le module généré ; vérifier le système par rapport aux propriétés exprimées en logique temporelle linéaire (LTL) en utilisant Maude LTL Model-Checker. En outre, nous utilisons RT-Maude pour rechercher et analyser le système pour trouver des comportements indésirables. L'approche est illustrée à travers trois études de cas.

Mots-clés : RT-Maude, systèmes en temps réel, transformation de modèles, grammaires de triples graphes, Interpréteur TGG, modèle-à-texte, spécification et vérification, logique de réécriture, vérification de modèle.

ABSTRACT

Real-time embedded systems must be properly validated and verified before their manufacturing and deployment in order to increase their reliability and reduce their maintenance cost. Models have been used for a long time to build complex systems, in virtually every engineering field. This is because they provide invaluable help in making important design decisions before the system is implemented.
In a typical Model Driven Engineering (MDE) activity, systems are modeled using semi-formal notation and then are validated and implemented. The validation step, based on these models, is particularly crucial for Real-Time Embedded Systems (RTES), in order to ensure their correct function. However, an MDE approach remains insufficient in the sense that it does not indicate how to use the models to apply the analysis. Faced with this situation, the integration of formal methods into the development cycles of these systems has become paramount. These methods have been recognized since a long time in order to help the development of reliable systems, because of their mathematical foundations, which are reputedly rigorous on the completeness of the formal verification that they allow to activate it.
In this thesis, we propose an approach based on model transformation to get a formal specification of our system, and then use formal verification techniques to demonstrate that a system design is correct with respect to its specification. First, we start by using the state chart diagram (also known as state machine diagram), which describes a real-time system, as a source model to generate a RT-Maude code, this code represents the real time Maude module of the system (a target model). The second part of our work is to; based on this generated module; verify the system against specified LTL properties using Maude LTL Model-Checker. In addition, we use the Real-Time Maude to search analysis found a previously unknown behavior that led to missed deadlines. The approach is illustrated through three cases study.

Keywords: RT-Maude, Real-Time Systems, Model Transformation, Triple Graph Grammars, TGG Interpreter, Model-to-Text, Specification and Verification, Rewrite Logic, Model Checking.

الملخص

يجب أن يتم التأكد من صحة الأنظمة المضمنة ذات الوقت الفعلي والتحقق منها قبل تصنيعها ونشرها من أجل زيادة موثوقيتها وتقليل تكاليف الصيانة. وقد استخدمت النماذج منذ فترة طويلة لبناء أنظمة معقدة، في كل مجال الهندسة تقريبا. وذلك لأنهم يقدمون مساعدة لا تقدر بثمن في اتخاذ قرارات التصميم الهامة قبل إنشاء النظام.

في عملية الهندسة المبنية على النموذج (MDE)، الأنظمة تتم نمذجتها باستعمال تدوين شبه رسمي ومن ثم يتم التحقق من صحتها وتنفيذها. خطوة التحقق، استنادا إلى هذه النماذج، أمر بالغ الأهمية خاصة بالنسبة إلى الأنظمة غير المجزأة ذات الوقت الحقيقي، من أجل ضمان وظيفتها الصحيحة. ومع ذلك، لا يزال نهج الهندسة المبنية على النموذج غير كاف بمعنى أنه لا يشير إلى كيفية استخدام النماذج لتطبيق التحليل. وفي مواجهة هذا الوضع، أصبح إدماج الأساليب الرسمية في مراحل تطوير هذه النظم حاجة قصوى. وقد تم الاعتراف بهذه الطرق منذ وقت طويل من أجل المساعدة في تطوير نظم موثوقة، بسبب أسسها الرياضية، والتي هي صارمة ومكتملة باستعمال التحقق الرسمي التي تسمح بتنشيطه.

في هذه الأطروحة، نقترح منهجا يستند إلى التحول النموذجي للحصول على مواصفات رسمية لنظامنا، ومن ثم استخدام تقنيات التحقق الرسمية لإثبات أن تصميم النظام هو الصحيح بالنسبة لمواصفات النظام الأولية. أولا، نبدأ باستخدام الرسم البياني للحالة Statechart الذي يصف النظام ذو الوقت الحقيقي، كنموذج مصدر لتوليد رمز RT-Maude، وهذا الرمز يمثل وحدة ذات الوقت الحقيقي RT-Maude للنظام (نموذج مستهدف). الجزء الثاني من عملنا هو؛ استنادا إلى هذه الوحدة المولدة؛ التحقق من النظام على حساب خصائص منطق الزمن الخطي (LTL) محددة باستخدام مدقق النماذج Maude LTL. وبالإضافة إلى ذلك، فإننا نستخدم RT-Maude للبحث والتحليل وإيجاد سلوكيات غير مرغوبة في النظام. هذا المنهج طبقناه على ثلاث حالات.

الكلمات المفتاحية: RT-Maude، نظم الوقت الحقيقي، نموذج التحول، تحويل الرسوم البيانية الثلاثي، مترجم TGG، نموذج إلى نص، التوصيف والتحقق، منطق إعادة الكتابة، نموذج التحقق.

Liste des abréviations

AADL : Architecture Analysis and Design Language
AGG : Attributed Graph Grammar
API : Application Programming Inteface
AToM^3: A Tool for Multi-formalism and Meta-Modelling
CASE : Computer Aided Software Engineering
CATIA : Conception Assistée Tridimensionnelle Interactive Appliquée
CIM : Computation Independent Model
CSP : Communicating Sequential Processes
CTL : Computational Tree Logic
DARTS : Design Approach for Real-Time Systems
EMF : Eclipse Modeling Framework
ET-LOTOS : Enhanced Timed Language of Temporal Ordering Specifications
GMF : Graphical Modeling Framework
GReAT : Graph Rewriting And Transformation
HOOD : Hierarchic Object-Oriented Design
HRT-HOOD : Hard Real-Time Hierarchic Object-Oriented Design
JMI : Java Metadata Interface
KDE : K Desktop Environment
LHS : Left Hand Side
LOTOS : Language of Temporal Ordering Specifications
LTL : Linear Temporal Logic
M2M : Model to Model
M2T: Model to Text
MDA : Model-Driven Architecture
MDE : Model-Driven Engineering
MOF : Meta-Object Facility
NuSMV : New Symbolic Model Verifier
OCL : Object Constraint Language
OMG : Object Management Group
OMT : Object Modeling Technique
OOD : Object-Oriented Analysis and Design
OOSE : Object Oriented Software Engineering
PAISLey : Process-oriented Applicative and Interpretable Specification Language
PIM : Platform Independent Model
PSM : Platform Specific Model
RHS : Right Hand Side
RTIL : Real-Time Interval Logic
RTL : Real-Time Logic
RT-LOTOS : Real-Time Language of Temporal Ordering Specifications
SADT : Structured Analysis and Design Technique
SDL : Specification and Description Language

SDRTS : Structured Design for Real-Time Systems
SETR : Système Embarqué Temps Réel
SPIN : Simple Promela Interpreter
STR : Système Temps Réel
TCTL : Timed Computational Tree Logic
TGG : Triple Graph Grammars
TLTL : Timed Linear Temporal Logic
TPTL : Timed Propositional Temporal Logic
UML : Unified Modeling Language
UP : Unified Process
XMI : XML Metadata Interchange
2TUP : 2 Tracks Unified Process

Table des matières

3 La logique de réécriture comme formalisme de spécification et vérification des systèmes temps réel

4 La modélisation avec UML

5 Ingénierie Dirigée par Modèles (IDM)

Table des figures

Liste des tableaux

CHAPITRE 1

Introduction générale

Sommaire

1.1 Contexte

Les systèmes temps réel sont devenus omniprésents, dans plusieurs domaines tels que les systèmes de contrôle de processus, les centrales nucléaires, l'avionique, le contrôle du trafic aérien, les télécommunications, les applications médicales, les technologies multimédias et les applications de défense. Ils sont généralement hétérogènes du point de vue composants matériels et logiciels ; ces systèmes sont de nature complexes.
Dans le domaine des systèmes embarqués temps réel (SETR) et les systèmes réactifs, les architectures logicielles doivent être conçues pour assurer des fonctions très critiques soumises à de très fortes exigences en termes de performances temps réel et de faisabilité. En effet, avec la croissance de la capacité des calculateurs embarqués, la taille de ces systèmes accroît ainsi les risques d'erreurs.

Concevoir des applications temps-réel embarquées constitue de nos jours une tâche très difficile. Ceci est dû à la complexité de ces applications, au nombre de contraintes qu'elles doivent respecter et aux problèmes à prendre en compte et à résoudre lors de leur construction. Le développement de ces systèmes est difficile du fait qu'il est nécessaire de respecter non seulement l'exactitude des résultats, mais aussi les contraintes temporelles liées à la validité des données et à l'échéance des opérations ainsi que la fiabilité et la sécurité de ces systèmes. Un défaut de comportement ou le non-respect d'une contrainte de temps peut provoquer des conséquences catastrophiques. Des défauts sur ce type de systèmes peuvent entraîner des pertes financières considérables ou même mettre en danger des vies humaines. Certains systèmes sont à sûreté critique, comme les avions, d'autres moins, comme les montres ou des machines à laver. La complexité du développement reste un problème majeur dans le cas de systèmes de grande envergure. Pour gérer cette complexité, on se tourne vers l'idée de modélisation. Aujourd'hui, le langage de modélisation unifié (UML) [1], standardisé par l'Object Management Group (OMG) a été largement accepté par l'industrie et s'est établi comme le langage commun pour l'analyse et la conception en génie logiciel orienté objet. En informatique, plus particulièrement dans l'ingénierie et le développement logiciel, les modèles sont utilisés sur une grande étendue pour décrire un système. Ces modèles peuvent représenter d'une manière précise sa structure, sa logique ou/et son comportement.

Il est clair que la spécification et la vérification de systèmes temps réel est une étape fondamentale dans le cycle de développement des systèmes informatiques. En effet, l'utilisation des méthodes de spécification et de vérification contribue au développement de systèmes sûrs de fonctionnement. Les architectures logicielles doivent être conçues pour assurer des fonctions critiques soumises à des contraintes très fortes en termes de fiabilité et de performances temps réel. Toutefois, si les tests s'indiquent souvent efficaces pour détecter les erreurs, ils ne permettent généralement pas de démontrer exhaustivement l'absence d'erreurs.

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. Pour ces raisons, l'utilisation conjointe de notations semi-formelles et de langages formels (approches dites mixtes) est de plus en plus demandée.

Face à ce constat, de nombreuses techniques ont été explorées, parmi lesquelles celles de la famille des méthodes formelles qui ont contribué, depuis de nombreuses années, à l'apport de solutions précises et rigoureuses afin d'aider les concepteurs à produire des systèmes non défaillants. Les méthodes formelles fournissent des outils permettant de garantir mathématiquement l'absence de certaines erreurs. Les méthodes formelles représentent une solution intéressante à ce problème. Les spécifications formelles auront pour effet d'éliminer les ambigüités au niveau de l'interprétation des modèles. La vérification formelle de modèles (model checking) [2], quant à elle, tente de valider le comportement d'un système à l'aide de diverses propriétés énoncées à l'aide d'une logique (temporelle, etc.). La combinaison d'un langage de spécification formelle éprouvé et de techniques de vérification de modèles *(model checking)* solides permettra de valider formellement les applications développées. Cette démarche permet d'éliminer certaines erreurs de façon précoce avant de passer aux phases de conception et de programmation. Elle peut également supporter la vérification du code ultérieurement.

1.2 Problématique

Le défi pour pouvoir maitriser une complexité, au cours des différentes phases dans le processus de développement des systèmes embarqués, est sans doute de proposer une démarche rigoureuse basée sur une sémantique formelle pour consolider la description structurelle des différents composants de ces systèmes, évaluer au plus tôt leur comportement et vérifier leurs propriétés d'exécution. Plusieurs approches de développement des systèmes embarqués ont été proposées dans la littérature dont certaines ont été appliquées dans l'industrie. L'approche classique suit une démarche informelle basée sur le prototypage et le débogage.

Dans ce contexte, l'OMG cherche à promouvoir plus encore l'utilisation de modèles au niveau d'abstraction le plus pertinent tout au long du processus d'analyse, de conception et de développement, en proposant l'architecture dirigée par les modèles (MDA, Model-Driven Architecture). L'Ingénierie Dirigée par les Modèles (IDM) [3] offre un cadre méthodologique outillé qui se base sur des modèles abstraits plutôt que sur des concepts d'algorithmique et de programmation. Elle permet ainsi de mieux maîtriser la complexité du développement logiciel de ces systèmes et de valider les choix de conception avant la phase de codage, impliquant une réduction du coût et de durée de développement.

Aujourd'hui l'utilisation de méthodes formelles (*MFs*) est essentielle pour le développement de systèmes complexes, en particulier pour les systèmes temps réel et critiques où les questions liées à la sûreté/fiabilité sont fondamentales. D'autre part, l'IDM a atteint un bon niveau de maturité et est devenue une nouvelle démarche en génie logiciel qui conçoit l'intégralité du cycle de développement en se basant sur la méta-modélisation et transformation de modèles.
L'enjeu consiste donc à concevoir la vraie complémentarité entre ces deux méthodologies : comment ces deux approches peuvent être combinées.

Le développement des systèmes embarqués temps réel nécessite des spécifications précises, complètes et non ambiguës afin d'obtenir des logiciels sûres, fiables et efficaces. L'ingénierie dirigée par les modèles apporte de bonnes pratiques pour le développement de logiciels. Ces pratiques serviront également aux applications temps réel embarquées pour plusieurs raisons [4] :

- La spécification des applications temps réel embarquées comprend différents points de vue (ex. fonctionnel, temps-réel, tolérance aux fautes, etc.). Cela nécessite des techniques d'abstraction lors du développement.
- Les options d'implémentation visées peuvent varier considérablement ; différents modèles d'exécution peuvent être envisagés pour un même modèle en fonction de contraintes de réalisation particulières (modèle multi-tâches, communication synchrone, programmation en boucle, etc).
- La contrainte de performance des SETRs s'oppose aux techniques standard du développement logiciel. Les optimisations potentiellement réalisées peuvent produire un code fonctionnel qui pénalise la maintenabilité de l'application finale.
- Le test et la validation des applications embarquées temps réel sont des activités critiques qui nécessitent la mise en place de modèles et d'outils d'analyse sophistiqués et spécifiques.

L'utilisation de l'IDM pour le développement des STREs, nécessite la définition d'un ensemble cohérent et complet d'artéfacts (règles méthodologiques, transformations de modèles, génération automatique de code) préalablement testés, outillés et évalués. Il reste néanmoins que l'ingénierie dirigée par les modèles présente un inconvénient mineur qui consiste à produire un code généré moins performant qu'un code optimisé directement pour une plateforme cible.
L'objectif des travaux présentés dans ce manuscrit est de répondre, principalement, aux questions suivantes :

- Comment générer automatiquement des spécifications d'un système temps réel à partir d'un modèle comportemental d'un système temps réel écrit en UML ?
- Comment vérifier et analyser les spécifications générées, en se basant sur une logique temporelle assez expressive et en disposant d'un outil de vérification par Model-Checking ?

1.3 Contributions

Dans cette thèse, nous nous intéresserons en générale de tous les types de systèmes temps réel, particulièrement aux systèmes embarqués temps-réel (SETR). Cette thèse propose une approche mixte de spécification de systèmes temps réels basée sur la notation UML (langage semi-formel) et le formalisme de RT-Maude [5] qui est un langage formel destiné aux systèmes temps-réel. Cette approche se propose de combiner les avantages des spécifications formelles et de la vérification de modèles dans une seule et unique technique unifiée. Cette technique exploite les outils de l'ingénierie dirigée par les modèles (IDM). La génération automatique et massive de code permet de réduire le cycle de développement de l'application et réduit le risque d'erreur par le développeur. L'une des devises les plus populaires de l'IDM prend tout son sens « *Model once, generate everywhere* » (« Modéliser une fois, générer partout »).

Les spécifications ainsi construites doivent pouvoir être non seulement simulés mais également exploités lors d'analyses formelles pour vérifier les exigences temporelles spécifiées par le concepteur. Dans notre travail, nous nous focalisons sur les techniques de vérification de type *Model-Checking*. Celles-ci ont été fortement popularisées grâce à leur capacité d'exécuter automatiquement des preuves de propriétés sur des modèles logiciels. De nombreux outils (model-checkers) ont été développés dans ce but.

Les objectifs poursuivis tout au long de cette thèse étaient de développer un cadre formel par lequel il serait possible de translater des diagrammes d'états-transitions d'UML vers une notation formelle du langage RT-Maude. Par la suite, cette description formelle RT-Maude sera validée à l'aide de deux des outils de l'environnement RT-Maude : les simulations et la vérification de modèles (Model-checking). Le processus développé se divise en quatre étapes majeures :

1. La description de comportement d'un système temps réel à l'aide de diagrammes d'états-transitions UML (Statechart).
2. En basant sur l'IDM, Construire une grammaire de graphes (règles) permettent de transformer un modèle de diagramme d'états-transitions vers un modèle RT-Maude. C'est une phase de transformation de modèle ;
3. Génération d'une description formelle RT-Maude. Cette étape consiste à dériver une description formelle basée sur le langage RT-Maude à partir de diagramme d'états-transitions UML (génération de code) ;
4. Une fois l'étape de spécification réalisée, celle-ci peut être validée à l'aide de méthodes formelles par une vérification de la description formelle du système à l'aide du vérificateur intégré de l'environnement Maude et des propriétés comportementales du système exprimées à l'aide de la logique temporelle linéaire LTL.

1.4 Organisation du manuscrit

Ce document est organisé selon le plan suivant :

• **Chapitre 2** donne une vue générale du domaine de spécification et vérification des systèmes embarqués temps-réel (SETR) et les méthodes formelles et leurs classifications. Tout d'abord, nous introduisons les systèmes embarqués temps réel. Ensuite, les formalismes de spécification telles que les automates temporisés et les réseaux de Petri seront expliqués. La fin de ce chapitre sera consacrée aux logiques temporelles les plus connus.

• **Chapitre 3** nous présentons la logique de réécriture et les Model-Checkers les plus répandus. Une étude comparative entre les Model-checkers nous a permet de choisir Maude comme étant un Model-Checker assurant le meilleur rapport puissance/expressivité. La deuxième partie de ce chapitre est dédiée au langage RT-Maude qui est une variante de Maude utilisé pour les systèmes temps réel.

• **Chapitre 4** est une présentation de la modélisation orientée objet, l'utilisation du langage UML ainsi que l'interprétation de la conception à travers la notation UML. Les diagrammes d'états-transitions seront étudiés en détails.

• **Chapitre 5** décrit les concepts de base de l'Ingénierie Dirigée par les Modèles (IDM), utilisée pour leur mise en œuvre dans cette étude, avec ses différentes variantes. Nous nous intéresserons en particulier à l'Architecture Dirigée par les Modèles (ADM) dans le cadre des systèmes temps réel. Á la fin, nous aborderons les transformations de graphes, leurs grammaires et les systèmes de transformation.

• **Chapitre 6** donne un aperçu global des contributions de cette thèse. D'abord, nous introduisons les outils et les langages utilisés pour la mise en œuvre de notre approche (La plateforme Eclipse, EMF/Ecore, TGG Interpreter et le langage de génération de code Xpand). Puis, nous présenterons les règles de transformation de modèle Statechart décrit un système

temps réel en modèle RT-Maude. Ensuite une phase de génération de code utilisé pour obtenir des spécifications formelles RT-Maude de tel système. Finalement, le model-checker LTL de Maude permet une analyse et une vérification de ces spécifications.

• **Chapitre 7** illustre la validité et l'utilisabilité de notre approche. Ils sont montrés sur trois systèmes temps réel : un feu de circulation tricolores, un thermostat et un robot industriel. A la fin de ces études de cas, des expérimentations menées et des résultats obtenus dans ce chapitre pour évaluer l'approche proposée.

• **Conclusion générale** qui résume les contributions de cette thèse, et discute des limites et des perspectives de notre travail.

CHAPITRE 2

Spécification et Vérification des Systèmes Embarqués Temps-réel

Sommaire

2.1 Introduction

Les systèmes temps-réel sont souvent complexes et critiques, et nécessitent un développement rigoureux pour affirmer leur correction fonctionnelle et temporelle. Ces systèmes sont en interaction avec leur environnement et contraints par le temps. Plus particulièrement, Les systèmes embarqués ont souvent des contraintes d'énergie et des contraintes temps-réel [6]. En effet, les évènements temps-réel sont des évènements externes au système et ils doivent être traités au moment où ils se produisent (en temps-réel). Le disfonctionnement ou le bug de tels systèmes peut, ainsi, causer des pertes économiques et humaines. Cependant, ces « systèmes critiques » présentent souvent des imperfections et l'Histoire a connu de nombreux faits divers qui en témoignent.

Le bug d'Intel Pentium II dans l'unité de division de virgule flottante a causé une perte de 475 millions de Dollars. Les bugs des systèmes temps réel peuvent être catastrophiques, tel que l'explosion de la fusé ARIANE5 en juin 1996 [7] et le crash d'avions AirBus qui se renouvelle jusqu'à nos jours. De même, pour les systèmes temps réel qui sont utilisés pour contrôler les centrales nucléaires et chimiques. Il est clair que leurs disfonctionnements peuvent causer un désastre mettant en jeu des vies humaines. Le bug de la machine de radiothérapie THERAC-25, qui est un appareil médical utilisé aux États-Unis et au Canada à partir de 1976 pour traiter les tumeurs cancéreuses en exposant les patients à des radiations, a causé entre 1985 et 1987 le décès de 6 patients [8], qui ont été exposés à des surdoses de radiation.

Des milliers d'appels téléphoniques passés depuis la France sont restés sans réponse au cours des samedi 30 et dimanche 31 octobre 2004. Et pour cause, ces appels n'étaient pas traités par les commutateurs ! C'est l'analyse menée par les experts de *France Telecom* qui a permis de mettre en évidence une erreur logicielle sur un équipement de traitement de la voix sur IP situé à Reims. Les ingénieurs ont dû procéder au développement d'un logiciel de remplacement et l'ont implanté sur chacun des commutateurs, ce qui a nécessité l'ensemble du week-end [9].

Un rapport complet [10] sur les dangers du vote électronique a été publié par une commission d'experts du « Brennan Center », de l'Université de New York de Justice. Cette commission émet des conclusions critiques concernant l'utilisation actuelle des machines de vote électroniques, et propose un ensemble de mesures permettant d'augmenter la confiance dans ce système de vote.

Il apparaît primordial de pouvoir garantir une *sûreté de fonctionnement* de ces systèmes critiques. Les méthodes habituellement utilisées lors du développement de systèmes informatiques (comme des logiciels) consistent essentiellement à « tester » le fonctionnement du système dans un ensemble de situations correspondant au cadre d'utilisation « attendu », *i.e.* pour lequel le système a été conçu. Cependant, de telles méthodes ne permettent pas de tester le comportement du système dans *toutes* les situations, l'ensemble des scénarios étant le plus souvent infini. Une autre faiblesse de cette approche provient du manque de précision de ces tests. Les descriptions souvent informelles utilisées dans les cahiers des charges ne décrivent pas de façon satisfaisante les comportements admissibles et non-admissibles. Bien que le test du système permette de détecter des erreurs, il est nécessaire de compléter les résultats obtenus par une autre approche. Les *méthodes formelles* sont de plus en plus utilisées pour répondre aux exigences auxquelles sont soumis les systèmes temps-réel. Ces méthodes reposent sur l'utilisation de modèles formels de spécification dotés de sémantiques rigoureuses et de techniques de vérification formelle. L'idée principale de cette approche est de plonger la démarche de vérification du système dans un cadre formel afin de permettre l'utilisation de raisonnements mathématiques lors de l'analyse du système. Ainsi, il est important de spécifier

avec précision les systèmes temps réel et par la suite vérifier les propriétés exigées par l'utilisateur, afin d'assurer un développement fiable de tels systèmes, de réduire le coût de maintenance et de minimiser le risque de pertes économiques et humaines.

2.2 Les systèmes embarqués temps réel

Dans cette première partie, nous définissons le domaine d'étude des systèmes embarqués temps réel. Les systèmes sont vus au travers de leurs contraintes de qualité de service qui impactent fortement leur processus de développement. Du fait des avancées technologiques qui permettent une plus grande miniaturisation des systèmes, les systèmes embarqués temps réel (SETR), souvent cachés aux utilisateurs, sont de plus en plus présents dans notre environnement et de plus en plus complexes.

Les applications les plus connues des SETR sont les systèmes de transport (voiture, avion, train) et les systèmes mobiles autonomes (robot, fusée, satellite). De même, les systèmes liés à la gestion d'un périphérique (imprimante, souris sans fil), à la mesure (acquisition en temps réel) et les systèmes domotiques (électroménager) sont des systèmes embarqués pouvant posséder des contraintes temps réel liées aux capteurs ou actionneurs utilisés. Enfin, la notion d'embarqué peut été étendue aux objets portables grand public (cartes à puce, assistants personnel, téléphones mobiles, lecteurs vidéo, consoles de jeu).

2.2.1 Définitions

Définition 2.1. ***Les systèmes temps réel*** sont des systèmes informatiques qui doivent obligatoirement réagir à l'environnement dans les contraintes précises du temps. En conséquence, le comportement correct de ces systèmes dépend non seulement de la valeur du résultat logique, mais aussi du temps de réponse auquel le résultat est produit [11].

Définition 2.2. Un système embarqué (*embedded system*) est un système électronique et informatique intrinsèquement lié à un équipement (industriel ou bien de consommation) pour lequel il est conçu et dans lequel il est intégré. Il est entièrement dédié à une ou plusieurs fonctionnalités attendues de cet équipement [12] et n'a pas de raison d'être en dehors de celui-ci [13].
Les activités identifiées d'un tel système concernent les traitements liés au fonctionnement de l'équipement, les interactions avec l'équipement, les interactions avec d'autres équipements et les interactions avec l'environnement physique ou humain [14]. Le terme « système enfoui » souligne parfois la capacité limitée ou l'absence d'interaction directe avec un utilisateur.

Définition 2.3. Un système embarqué temps réel (*real-time embedded system*) est un système embarqué dans un équipement (ou procédé) dont il doit contrôler et commander le comportement [15]. Le qualificatif « temps réel » du système est dû à l'évolution dynamique du procédé auquel il est connecté et qu'il doit piloter [16]. Il reflète une capacité de réaction à l'échelle de temps du procédé[1].
Les interactions entre un système embarqué temps réel et le procédé qu'il pilote prennent généralement la forme : de mesures prélevées par le système sur le procédé, d'évènements que le procédé signale au système et de signaux de commande adressés au procédé par le système [17].

[1] Le terme « système réactif » est pour cela parfois utilisé.

2.2.2 Système de contrôle/commande temps réel

L'architecture classique des systèmes temps réel est illustrée par la figure 2.1. Ces systèmes sont constitués essentiellement de deux sous-systèmes : le procédé à contrôler et le système de contrôle.

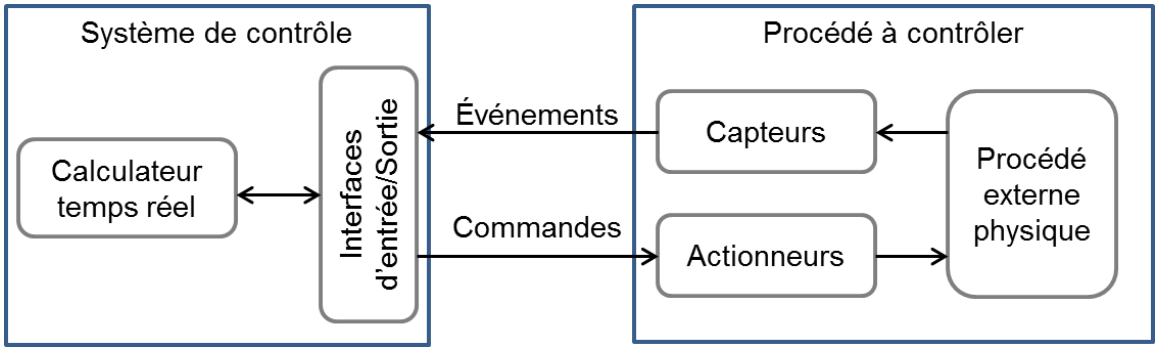


Figure 2.1. Architecture d'un système temps réel [18].

Le procédé communique via des capteurs et des actionneurs. Les capteurs récupèrent des informations du procédé et les transmettent au système de contrôle sous forme d'événements et de données. Les actionneurs commandent le procédé afin de réaliser une tâche bien définie. Le système de contrôle est composé d'un calculateur auquel sont liées des interfaces d'entrées/sorties qui servent à communiquer avec les capteurs et les actionneurs.
Le calculateur exécute un algorithme de contrôle en respectant des propriétés temporelles et envoie des ordres de commande aux actionneurs via l'interface de communication. Le procédé contrôlé est souvent appelé l'environnement du système de calcul temps réel. Un système temps réel est dit embarqué lorsque le système de contrôle est enfoui à l'intérieur de l'environnement avec lequel il interagit, comme un calculateur dans un avion ou une voiture.

2.2.3 Structure interne d'un système temps réel

Un système informatique temps réel est composé de deux couches, la première concerne l'architecture matérielle et la seconde représente l'architecture logicielle. La figure 2.2 illustre la structure générale (gros grain) d'un système temps réel. De bas en haut, la couche matérielle représente l'ensemble des ressources physiques utilisées pour exécuter la partie logicielle du système. Celle-ci englobe les ressources de calculs ou de traitements (processeurs), les ressources de communication (réseaux), les ressources de stockage (mémoires) et les périphériques d'entrée-sortie (capteurs et actionneurs). Le rôle de la couche logicielle est d'assurer le fonctionnement correct du système temps réel en garantissant une bonne gestion de ressources matérielles. La couche logicielle comprend l'exécutif temps réel (système d'exploitation temps réel SETR) qui est composé de différents modules lui permettant de fournir des services de communication, de synchronisation et d'exécution. Parmi les exécutifs temps réel existants, on trouve ceux qui sont libres comme MicroC/OS-II, FreeRTOS et RTEMS, et ceux qui sont commerciaux comme VxWorks, QNX, LynxOS, Chorus, Nucleus RTOS, WindowsCE. D'autre part, la couche logicielle contient le programme informatique responsable du contrôle de l'environnement. Ce programme est constitué d'un ensemble d'entités d'exécution et de structuration appelées tâches et processus [19].

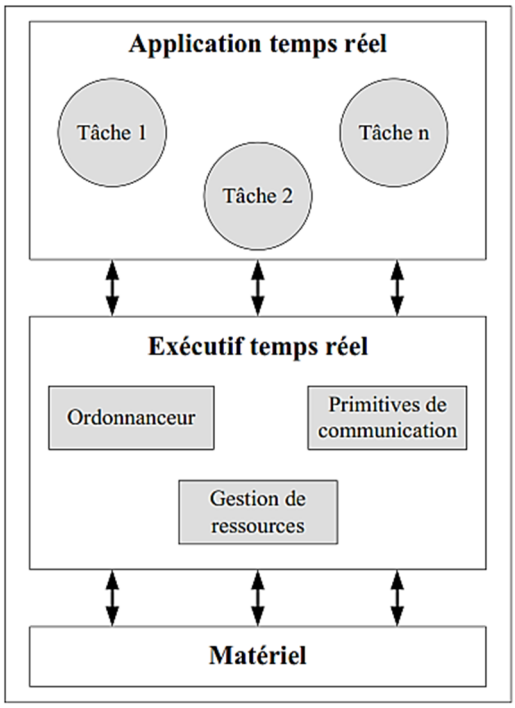


Figure 2.2. Structure interne d'un système temps réel.

2.2.4 Les catégories des systèmes temps réel

Certains systèmes temps réel sont très liés au temps, par exemple : un distributeur de billets, un radar ou un système de freinage ABS (*Antilock Brake System*). Les systèmes temps réel doivent donc généralement respecter des contraintes temporelles. Ainsi, suivant les exigences temporelles d'un système temps réel, on peut distinguer deux classes de ces systèmes :

– ***Systèmes temps réel à contraintes strictes/dures (hard real-time constraints) :***
Ces systèmes doivent impérativement respecter les contraintes de temps imposées par l'environnement, les échéances ne doivent jamais être dépassées et le non-respect d'une contrainte temporelle peut avoir des conséquences catastrophiques [20].
Le système de contrôle de trafic aérien et de conduite de missile sont deux exemples de ces systèmes.

– *Systèmes temps réel à contraintes relatives/souples (soft real-time constraints) :*
Ces systèmes se caractérisent par leurs souplesses envers les contraintes temps réel imposées par l'environnement, il s'agit d'exécuter dans les meilleurs délais les fonctions. Le non-respect d'une contrainte temporelle est toléré (acceptable)par le système, et sans que cela ait des conséquences catastrophiques [21] [22]. Par exemple des applications multimédias.

– *Systèmes temps réel à contraintes mixtes :* sont composés des tâches à contraintes strictes et des tâches à contraintes souples [23].

2.2.5 Caractéristiques des tâches temps réel

Un programme temps réel est composé d'un ensemble d'entités appelées *tâches temps réel*. Chacune ayant un rôle qui lui est propre, comme par exemple : réaliser un calcul, être associé à une alarme, traiter des entrées / sorties, etc.
La loi d'arrivée d'une tâche définit sa nature. Cette loi s'agit des contraintes temporelles qui définissent la répartition des dates d'activation des instances d'une tâche dans le temps.
Selon la loi d'arrivée, il est possible de classer les tâches en trois catégories :

– ***Une tâche périodique*** est une tâche dont l'activation est régulière et le délai Pi (*période*) entre deux activations successives est constant. Une tâche Ti est caractérisée par une durée d'exécution Ci, une période d'activation Pi, une échéance Di et la date de la première activation Ri (cf. figure 2.3) [24].

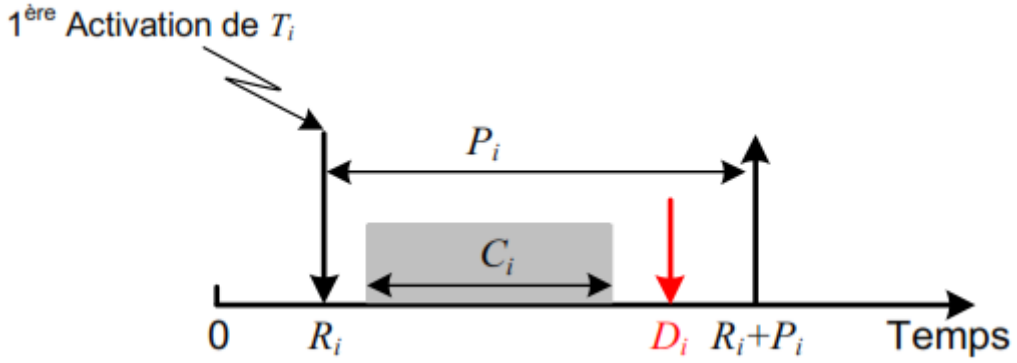


Figure 2.3. Modèle d'une tâche temps réel périodique.

– **Une tâche *sporadique*** est une tâche caractérisée par un délai minimum entre deux activations successives. Au contraire des tâches périodiques, les dates d'activation des différentes instances d'une tâche sporadique ne peuvent pas être déterminées a priori.
– **Une tâche *apériodique*** est une tâche dont on ne connaît aucune caractéristique, sauf son échéance. Elle est généralement activée par l'arrivée des évènements (message ou requête de l'opérateur) qui peuvent être produits à tout instant.
Les applications temps réel à contraintes strictes sont des applications où les contraintes temporelles doivent être strictement satisfaites. Dans ce but, une analyse des contraintes temporelles en phase de conception (off-line) est essentielle.

2.2.6 Exigences posées par les systèmes temps réel

Généralement, on distingue deux classes d'exigences dans un système [25] :
• *Les exigences fonctionnelles* représentent les différentes fonctionnalités que devra satisfaire le système et correspondent à l'aspect opérationnel de ce système. En matière de vérification formelle, on associe généralement ce type d'exigences à des « propriétés générales » telles que l'absence de blocage non désiré (« deadlock »), l'absence de fonctionnement cyclique infini non désiré (« livelock ») ou encore la possibilité de revenir à l'état initial.
• *Les exigences non-fonctionnelles* correspondent aux critères de qualité attendus du système, par exemple en termes de performances temporelles. A titre d'exemple, pensons aux différentes garanties temporelles (par exemple des délais maximum ou minimum) que doit satisfaire un système temps réel.

Dans le domaine du temps réel, les exigences non-fonctionnelles temporelles se répartissent plus précisément en deux catégories [25] :
• Les exigences où le temps est exprimé de manière *qualitative* (ou *logique*). On ne considère alors qu'un ordre partiel entre événements (par exemple un ascenseur doit être à l'arrêt pour que la porte s'ouvre).
• Les exigences où le temps est exprimé de manière *quantitative*. On considère dans ce cas l'ordre des événements mais aussi les distances temporelles entre ces derniers (par exemple la porte d'un ascenseur doit s'ouvrir deux secondes après l'arrêt de la cabine).

Hypothèse. La majorité des systèmes embarqués industriels sont des systèmes temps réel. ***Dans la suite du manuscrit on parlera de systèmes temps réel pour signifier des systèmes temps réel et embarqués***.

2.2.7 Développement classique d'un système embarqué

Le développement des systèmes embarqués temps réel (SETR) pose des défis importants pour les développeurs. Dans de tels systèmes, l'exactitude des résultats ne dépend pas seulement de l'exactitude logique des calculs, mais aussi de la date à laquelle le résultat est produit. Cela implique que le temps de réponse est aussi important que la production des résultats corrects.

À la différence du développement et de la conception d'une application logicielle sur une plateforme standard, la conception d'un système embarqué fait que le software et le hardware soient conçus en parallèle. En effet, le développement et la conception sont réalisés, comme le montre la figure 2.4, en décomposant et en assignant le système embarqué en software (SW) et hardware (HW). Ainsi, on permet une conception séparée des composants software et hardware, et finalement l'intégration des deux. Le développement dans ce cas se fait selon les phases suivantes [6] [27] :

1. Spécification des besoins
2. Partitionner la conception des composants SW et HW
3. Itération et raffinement de ce partitionnement
4. Conception des tâches SW et HW séparément
5. Intégration des composants HW et SW
6. Production des tests et validation
7. Maintenance et revalorisation

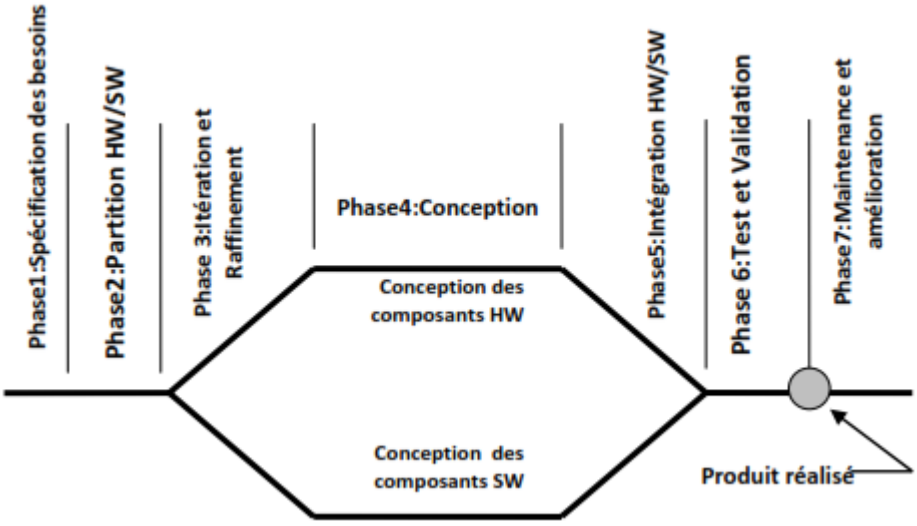


Figure 2.4. Cycle de vie de conception embarquée [6].

2.3 Vérification des Systèmes Temps-réel

Dans certains domaines critiques comme les systèmes embarqués temps réel (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. La conception des systèmes temps réel, de plus en plus complexes, se heurte à des problèmes de vérification. La vérification cherche à assurer que le système construit effectue

correctement les fonctions spécifiées. Cette tâche doit être intégrée dans tout le processus de conception du système, afin de minimiser le coût de développement.

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. En utilisant ces deux méthodes, seules les erreurs couvertes par les scénarios considérés sont détectées. Il s'avère alors impossible de tester tout le comportement du système. Cette insuffisance a été énoncée par l'informaticien Edsger W. Dijkstra dans sa fameuse citation [28] : *« Les tests peuvent servir à montrer la présence d'erreurs, mais jamais à garantir leur absence. »*

Une approche beaucoup plus ambitieuse est celle des *méthodes formelles*. Elle est apparue durant les années soixante et depuis elle a considérablement évolué. Les méthodes formelles reposent sur des fondements mathématiques qui permettent de raisonner sur des propriétés et de construire des preuves. Ces preuves montrent que les propriétés exprimées sur un programme sont bien respectées.

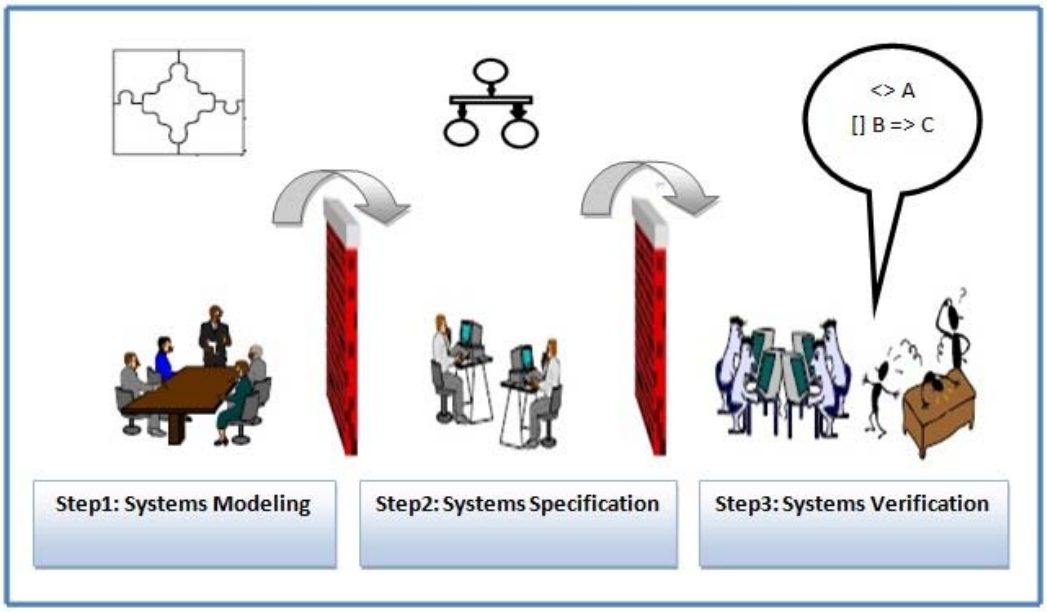


Figure 2.5. Le processus de modélisation et vérification des systèmes (temps réel) [29].

Les enquêtes ont montré que les procédures de vérifications formelles auraient révélé les défauts exposés dans le lanceur Ariane-5 (lanceur développé pour placer des satellites sur orbite géostationnaire), la sonde Mars Pathfinder (vaisseau intégrant un robot pour l'exploration de la planète Mars), le processeur Intel Pentium 2 et la machine de radiothérapie Therac-25 [30] par exemple.

Notre travail s'inscrit dans le cadre de la vérification automatique de modèles, approche que nous allons décrire plus précisément. Celle-ci présente trois grandes étapes, qui sont représentées sur la figure 2.5.

2.3.1 Modélisation du système

Un système étudié peut être donné sous la forme d'un système réel (système physique, ou code logiciel) ou sous la forme d'une description de son comportement (un protocole par exemple). Il est traduit dans un formalisme adapté, donnant ainsi un *modèle du système*, disons M.

Ainsi, des langages d'expression de modèles ont été définis spécifiquement aux différents critères à observer. Nous en énumérons quelques-uns pour donner un aperçu de la diversité du monde de la modélisation des systèmes :
– la représentation de l'architecture d'un système avec les langages de modélisation UML [1] et AADL [31] sous forme d'un ensemble de composants interagissant. Le système sera décrit essentiellement du point de vue de sa structure : les caractéristiques intra-composant ainsi que les caractéristiques inter-composants (les interactions).
– la représentation d'un système dans l'espace en trois dimensions avec IRIS [32] ou CATIA [33],
– la représentation des comportements possibles d'un système à l'aide de formalismes dédiés :
  – les automates à contraintes, utilisés par le langage AltaRica [34], dans lesquels les contraintes régissent le franchissement des transitions,
  – les automates temporisés, utilisés notamment par le langage Uppaal [35],
  – les réseaux de Petri [36] permettant de manière très simple, de modéliser les interactions entre composants du type provider fournissant une donnée, un service auprès d'un receiver en attente de cette fourniture pour à son tour accomplir sa fonction.

Pour chacun de ces types de modèles, des attributs spécifiques sont définis et caractérisent le système à modéliser :
– pour les modèles AADL, un ensemble de composants avec identificateurs, attributs et services spécifiques doivent être identifiés, ainsi que les relations existantes entre ces composants,
– pour les modèles en trois dimensions, chacun des composants du système à modéliser doit être représenté en trois dimensions, ainsi que chacune des connexions entre composants,
– pour les automates à contraintes, il s'agit de caractériser l'ensemble des états d'un automate, les transitions existantes entre ces états, et les contraintes de franchissement associées,
– pour ce qui est des réseaux de Petri, il s'agit de définir l'ensemble des places du réseau ainsi que toutes les transitions, puis enfin le marquage initial des places [37].
Dans le quatrième chapitre, nous reviendrons plus en détails sur la modélisation avec UML.

2.3.2 Spécification Formelle

La spécification formelle consiste à établir une description formelle d'un système et de ses propriétés de comportement. Elle utilise un langage formel dont la syntaxe et la sémantique sont définies mathématiquement. L'intérêt des méthodes formelles est la possibilité de vérifier de manière rigoureuse des propriétés sur un modèle formel. Pour assurer qu'un système respecte certaines propriétés, on lui associe un model formel qu'il est possible de vérifier par preuve. La spécification du système est souvent donnée par un cahier des charges. Elle est aussi traduite dans un formalisme mathématique, habituellement une logique. Parmi les nombreux formalismes proposés, citons les logiques temporelles de temps linéaire ou de temps arborescent (LTL,CTL,TCTL, etc.) , le µ-calcul, etc.

Dans cette section, nous abordons les principales techniques de spécification des applications temps réel. Il faut signaler qu'un aspect important à prendre en compte lors de la conception et la mise en œuvre d'applications temps réel est la sûreté de fonctionnement, en particulier, la spécification du comportement d'une application face aux fautes temporelles des tâches ou des messages.

Deux approches sont habituellement utilisées pour classer les techniques de spécification. La première se base sur le degré de formalisme utilisé par les techniques, la seconde sur les possibilités offertes en termes de capacité de description et de mécanismes opérationnels. Les techniques opérationnelles sont celles définies en termes d'états et de transitions, elles sont

proches de l'exécution. Les techniques descriptives sont généralement fondées sur des notations mathématiques permettant de fournir des spécifications rigoureuses qui peuvent être automatiquement traitées pour vérifier des propriétés telles que la sûreté (c'est-à-dire, absence d'interblocage, d'erreur, etc.) et la vivacité (c'est-à-dire, terminaison des actions, occurrence des événements attendus, etc.) [38].

2.3.2.1 Techniques descriptives

Ces techniques sont souvent fondées sur un formalisme mathématique et produisent des spécifications précises, rigoureuses et donnent une vue abstraite du système. Le système est décrit par des propriétés, forçant le spécifieur à exprimer ce que le système doit faire plutôt que de dire comment le système va le faire. Les spécifications formelles peuvent être traitées automatiquement pour en vérifier la complétude et la cohérence. Cette vérification fait appel à des prouveurs de théorèmes. Il faut noter que ces techniques sont peu utilisées dans le domaine du temps réel, comparées aux techniques opérationnelles, à cause, à la fois, du manque d'outils d'utilisation et des difficultés de spécifier formellement des systèmes complexes que sont les systèmes temps réel. Plusieurs classifications des techniques descriptives ont été proposées, en voici une :

 Techniques descriptives fondées sur les méthodes algébriques: il s'agit de méthodes basées sur les types abstraits. Avec ces techniques, un système est décrit avec différents niveaux d'abstraction allant du plus simple au plus compliqué. Un système est vu comme un type abstrait et sa spécification consiste à décrire sa syntaxe (décrivant les domaines des opérateurs du type) et sa sémantique (décrite par des expressions mathématiques). La complétude d'une spécification peut être vérifiée si les propriétés attendues sont vérifiées par les axiomes. Une des techniques descriptives qui commence à émerger, pour la spécification de systèmes temps réel, est LOTOS et, plus exactement ses extensions telles que RT-LOTOS et ET-LOTOS.

 Techniques descriptives fondées sur les logiques mathématiques: ces techniques permettent de décrire le comportement d'un système à l'aide de règles qui spécifient comment le système peut évoluer. Pour prendre en compte les aspects temporels, il est fait appel aux logiques temporelles. Plusieurs techniques se plaçant dans cette catégorie ont été proposées depuis de nombreuses années : RTL (Real-Time Logic), CTL (Computational Tree Logic), RTIL (Real-Time Interval Logic), TCTL (Timed CTL), TPTL (Timed Propositional Temporal Logic).

2.3.2.2 Techniques opérationnelles

Les techniques opérationnelles se divisent en deux catégories : les techniques fondées sur les modèles à transitions (machines d'états et réseaux de Pétri) et les techniques fondées sur des notations abstraites.

Les machines d'états finis (MEF) permettent de vérifier des propriétés telles que l'atteignabilité des états. Il faut noter qu'avec les MEF, le nombre d'états croit avec la complexité du problème traité et la vérification de propriétés devient complexe aussi. Pour être adaptées aux systèmes temps réel, les MEF doivent fournir des possibilités d'expression des contraintes de temps (échéances, périodes, etc.). C'est la raison pour laquelle beaucoup d'extensions de MEF ont été proposées. Par ailleurs, pour appréhender la complexité des systèmes, une MEF peut être décomposée en plusieurs MEF communicantes. Des langages fondés sur les MEF, comme PAISLey (Process-oriented Applicative and Interpretable Specification Language), SDL (Specification and Description Language), Esterel et Statecharts, sont utilisés pour la spécification de systèmes temps réel.

En ce qui concerne les techniques opérationnelles fondées sur les réseaux de Pétri (RdP), plusieurs extensions des RdP de base ont été proposées pour prendre en compte des contraintes temporelles.

Les techniques opérationnelles fondées sur les modèles à transitions ont l'avantage de conduire à des spécifications exécutables et sur lesquelles on peut prouver certaines propriétés. Des vérifications de propriétés (telle que l'absence d'interblocage) sont possibles avec les RdP.
Les techniques basées sur des notations abstraites sont adaptées à l'analyse et la conception de systèmes, par décomposition du système en sous-systèmes. Elles sont souvent destinées à fournir une représentation visuelle du système pour réduire en quelque sorte l'effort du spécifieur. En général, elles ne modélisent pas l'aspect comportemental des systèmes et par conséquent elles ne sont pas directement utilisables pour une simulation ou une exécution du système. Par ailleurs, elles sont souvent informelles ou semi-formelles. Ces techniques incluent des extensions des méthode SADT (Structured Analysis and Design Technique) : (telles que DARTS – Design Approach for Real-Time Systems) – et SDRTS – Structured Design for Real-time Systems) ou HOOD (Hierarchic Object-Oriented Design) : (telles que HRT-HOOD – Hard Real-Time HOOD) largement utilisées dans la spécification d'applications non temps réel.

2.3.2.3 Techniques mixtes

Un outil idéal pour la spécification de systèmes temps réel devrait avoir les propriétés suivantes :

- il doit être facile à comprendre et à manipuler (une interface graphique est souvent souhaitée pour construire une spécification),
- il doit permettre la réutilisation de spécification déjà existantes,
- il doit permettre de vérifier formellement et de valider une spécification à tous les niveaux du cycle de vie d'un système,
- la spécification obtenue doit permettre des validations à l'aide de simulation.

Ces propriétés sont parfois en contradiction les unes avec les autres et ne peuvent être respectées par une approche qui est soit opérationnelle, soit descriptive. C'est la raison pour laquelle certains travaux proposent des techniques mixtes respectant le plus possible des quatre propriétés précédentes. Il faut noter cependant que la politique d'extension des outils et méthodes tend à essayer d'intégrer tout dans toutes les techniques, ce qui conduit inéluctablement à des techniques complexes et difficiles à appréhender et à mettre en œuvre. Comme exemple de technique mixte, nous pouvons citer ESM/RTTL qui est une approche intégrant les machines d'états (*Extended State Machines*) et la logique temporelle RTTL.

Il faut souligner qu'une technique de spécification n'est réellement utile pour un spécifieur que lorsqu'elle est accompagnée d'outils permettant de décrire, vérifier, simuler des comportements, de générer des implantations, etc. Au niveau commercial, peu d'outils existent pour appréhender les contraintes temporelles. Depuis une date récente, la presse spécialisée présente de plus en plus de produits pour le temps réel, allant des processeurs, jusqu'aux ateliers logiciels. Rares sont les études comparatives qui permettent de situer les différentes propositions les unes par rapport aux autres. Le marché n'est qu'à ses débuts.

2.3.3 Vérification Formelle

Au cours des dernières années, la vérification formelle des systèmes critiques s'est progressivement établie comme une discipline à part entière de l'informatique. Cette évolution est, d'une part, motivé par le fait que ces systèmes sont de plus en plus présents dans notre vie quotidienne et qu'ils ont un impact majeur sur nos sociétés.

Classiquement, on distingue deux grandes familles de techniques pour vérifier formellement la correction d'un système. En premier lieu, les techniques de preuve (*theorem proving*) qui sont des démonstrations mathématiques au sens classique du terme où la vérification des propriétés est effectuée par déduction à partir d'un ensemble d'axiomes et de règles d'inférences. La seconde famille de techniques est appelée vérification de modèles, ou *model-checking*, et consiste à construire un modèle à partir d'une description formelle d'un système.

2.3.3.1 Preuve de théorèmes (*Theorem proving*)

La preuve de théorème, [39] [40] nécessite que le système soit spécifié sous forme d'une théorie mathématique, ou devrait être transformé en une telle forme. En utilisant un ensemble d'axiomes (le théorème de base), un démonstrateur (le logiciel) tente de construire : soit une preuve de théorème en générant les étapes de preuves intermédiaires ; soit de réfuter les axiomes énoncés. Les axiomes sont intégrés ou fournis par l'utilisateur. Les démonstrateurs sont également appelés assistants de preuves. Les étapes pendant la preuve font appel aux axiomes et aux règles, ainsi qu'aux définitions et lemmes qui ont été possiblement dérivés.

Au contraire du model-checking, le theorem proving peut s'utiliser avec des espaces d'états infinis à l'aide de techniques comme l'induction structurelle. Son principal inconvénient est que le processus de vérification est normalement lent, sujet à l'erreur, demande beaucoup de travail et des utilisateurs très spécialisés avec beaucoup d'expertise [41].

2.3.3.2 Vérification de modèles (*Model-Checking*)

La vérification de modèles [2] [42] [43] est une technique qui consiste à construire un modèle fini d'un système et vérifier qu'une propriété cherchée est vraie dans ce modèle. Il y a deux façons générales de vérification dans le model-checking : vérifier qu'une propriété exprimée dans une logique temporelle est vraie dans le système, ou comparer (en utilisant une relation d'équivalence ou de pré-ordre) le système avec une spécification pour vérifier si le système correspond à la spécification ou non. Au contraire du theorem proving, le model-checking est complètement automatique et rapide. Il produit aussi des contre-exemples qui représentent des erreurs subtiles dans la conception et ainsi il peut être utilisé pour aider le débogage.

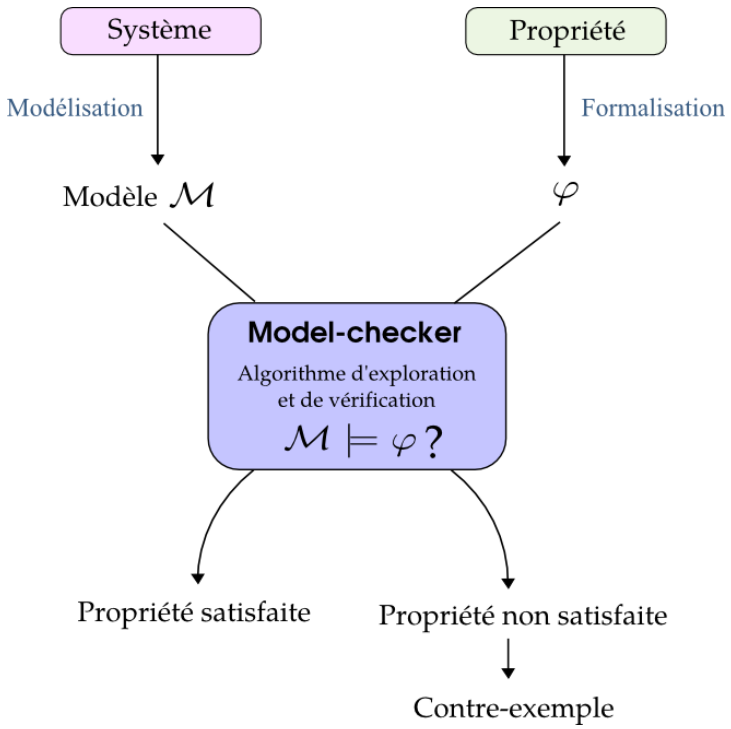


Figure 2.6. Principe du model-checking [44].

La preuve de théorème demande une grande interaction avec l'utilisateur ce qui rendrait la technique moins automatique. La technique du model checking comme méthode formelle est une méthode très performante et très utilisée. Cette technique correspond au mieux à notre domaine, car elle pourra révéler des erreurs non détectées par les autres méthodes formelles comme le test et la simulation. Dans notre travail, La technique du model-checking sera déployée pour la vérification des systèmes embarqués temps réel.

Le model checking est une technique de vérification qui explore tous les états possibles du système. Similaire à un programme d'échecs qui vérifie tous les mouvements possibles, le model checker, l'outil qui accomplit le model checking, examine tous les scénarios possibles du système d'une manière systématique. De cette manière, nous pouvons montrer que le modèle d'un système satisfait vraiment une certaine propriété. Il y a un vrai défi à examiner les espaces d'états les plus larges possible qui peuvent être traités par des moyens actuels, par exemple, des processeurs et des mémoires. Les model checkers peuvent gérer des espaces d'états d'environ 10 exposant 9 états. En utilisant des algorithmes plus performants et des structures de données adaptées, des espaces d'états plus larges (10 exposant 20 jusqu'à 10 exposant 476 états) peuvent être gérés pour des problèmes spécifiques. Même les erreurs subtiles qui ne restent pas encore découvertes en utilisant l'émulation, le test et la simulation peuvent être potentiellement révélées en utilisant la technique du model checking. Un récapitulatif des avantages et des inconvénients du model-checking est donné dans le tableau 2.1 [45].

Tableau 2.1. Avantages et Inconvénients du model-checking.

Avantages	**Inconvénients**
Assez efficace pour la détection des erreurs, automatique, exhaustif, détection des erreurs relativement tôt, bon rapport coût/bénéfice.	Espace d'état fini, explosion combinatoire d'espace d'état, erreurs de réalisation non détectées.

La phase de vérification applique un algorithme qui vérifie si le modèle du système satisfait ou non le modèle de sa spécification. Cet algorithme dépend de la nature des modèles choisis pour le système et les propriétés à vérifier. La vérification de modèles (*Model-Checking*), comme illustré dans la figure 2.6, permet de déterminer si le modèle du système satisfait la formule exprimant la spécification, ce qui est noté $M \models \phi$.
Si ce n'est pas le cas, des incohérences dans les descriptions du système et des propriétés sont mises en évidence par l'algorithme. Sinon, nous obtenons la garantie que le modèle M satisfait les propriétés exprimées par ϕ.

2.4 Classification des méthodes formelles

Dans la littérature, il existe plusieurs classifications des méthodes formelles. Selon J.M. Wing [46], on peut distinguer les méthodes :

- **Orientées opérations** pour décrire le fonctionnement du système et son comportement par des axiomes.
- **Orientées données** pour décrire les états du système
- **Hybrides** en combinant les deux orientations.

La figure 2.7 donne une classification des méthodes formelles selon le type d'approche qu'elles utilisent, axiomatique, basée sur les états ou bien hybride.

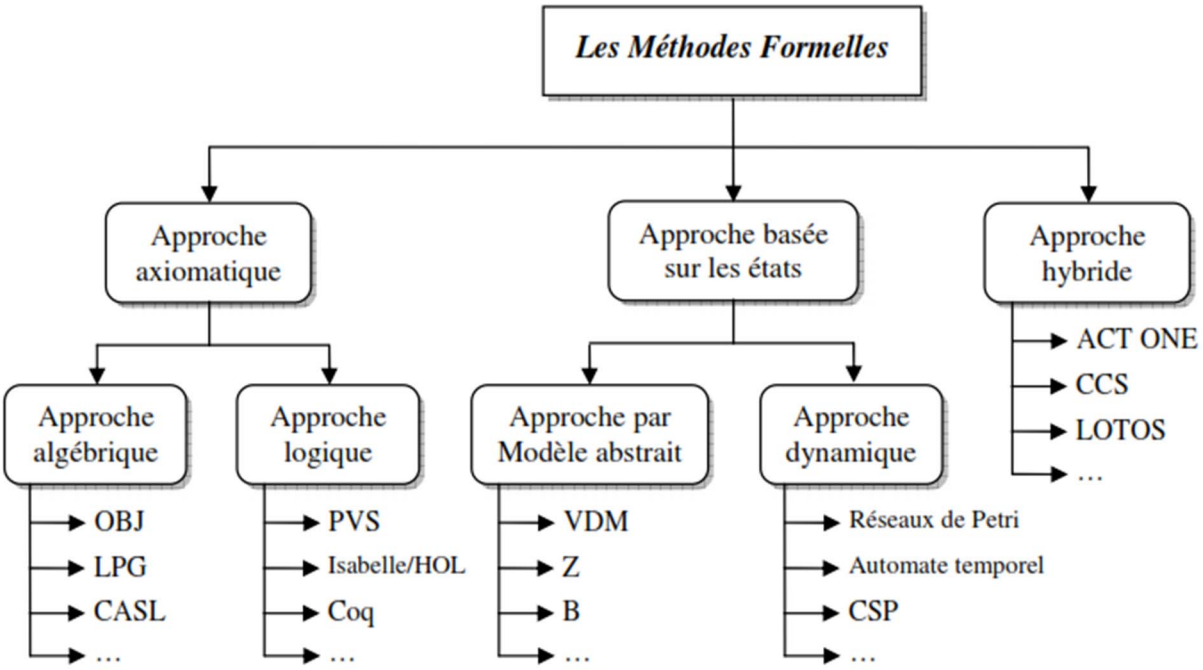


Figure 2.7. La classification des méthodes formelles [47].

Des trois approches de base, émanent les approches secondaires suivantes :
- Basées modèle
- Basées logique
- Algébriques
- Algèbre de processus
- Orientées réseaux (graphiques)

a- Approches basées modèle : Dans cette approche. Il n'y a pas de représentation explicite de la concurrence. Le système est modélisé par la définition explicite des états et des opérations qui le transforment d'un état vers un autre. Des besoins Non fonctionnels (telles que l'exigence temporelle) peuvent être exprimée dans certains cas. Exemples de méthodes : Z, VDM, et B.

b- Approches basées logique : Les propriétés du système, y compris les comportements temporels et probalistiques, peuvent être décrite en utilisant la logique. Le système d'axiomes associé à la logique permet de valider ces propriétés. Dans certains cas, un sous-ensemble de la logique peut être exécuté (par exemple, le système de tempura). La spécification exécutable est ensuite utilisée pour une simulation et un prototypage rapide. La logique de Hoare, Dijkstra, logique temporelle, etc. sont quelques exemples de cette approche.

c- Approches algébriques : On définit ici implicitement des opérations, en reliant leurs comportement sans définir la signification des états actuels. Similaire à l'approche fondée sur les modèles, dans le sens où il n'y a pas de représentation explicite de la concurrence. Exemples : OBJ, Larch.

d- Algèbre de processus : C'est dans cette approche, qu'une représentation explicite des processus concurrents est autorisée. En effet, les contraintes sur toutes les communications autorisées et observables entre processus, permettent la représentation du comportement du système. Exemples : Communicating Sequential Processes (CSP), Lotos, Timed CSP.

e- Approches basées graphes : Etant faciles à comprendre et, par conséquent, plus accessibles aux non-spécialistes, les notations graphiques sont très répandues pour la spécification des systèmes. Cette approche combine des langages graphiques avec la sémantique formelle, ce qui permet de puiser des avantages de ces deux axes dans le développement de systèmes. Exemples : Réseaux de Petri, StateCharts.

2.5 Méthodes formelles et cycle de vie

Le modèle du cycle de vie d'une application est un modèle des étapes ou des activités qui commencent quand le logiciel est conçu et se termine quand le produit n'est plus disponible pour l'utilisation [48]. Le cycle de développement d'une application temps réel comprend typiquement une étape d'expression des besoins, une étape de spécification, une étape de conception, une étape d'implémentation, une étape d'intégration et de test, une étape d'installation et de vérification ainsi que des étapes d'opération et de maintenance. Selon le cycle de développement choisi, ces étapes ou activités peuvent survenir une ou plusieurs fois dans un ordre prédéterminé. Ces étapes font partie de tous les cycles de développement de systèmes indépendamment de la nature, du domaine, de la taille et de la complexité du système à développer.
Plusieurs modèles de cycle de développement d'une application existent : le modèle en cascade [49], la cycle de vie en V [50], le prototypage rapide [51], le prototypage évolutif [52], la réutilisation de logiciel [53], le développement incrémental [54], etc.

A partir de ces modèles du cycle de développement, on peut conclure que les grandes étapes à suivre afin de développer un système sont la spécification, la conception et l'implémentation. La spécification décrit ce que le système doit faire, mais pas comment il le fait. La conception modélise une abstraction de haut niveau de l'implémentation qui cache tous les détails de l'implémentation. L'implémentation décrit précisément l'exécution de l'application.

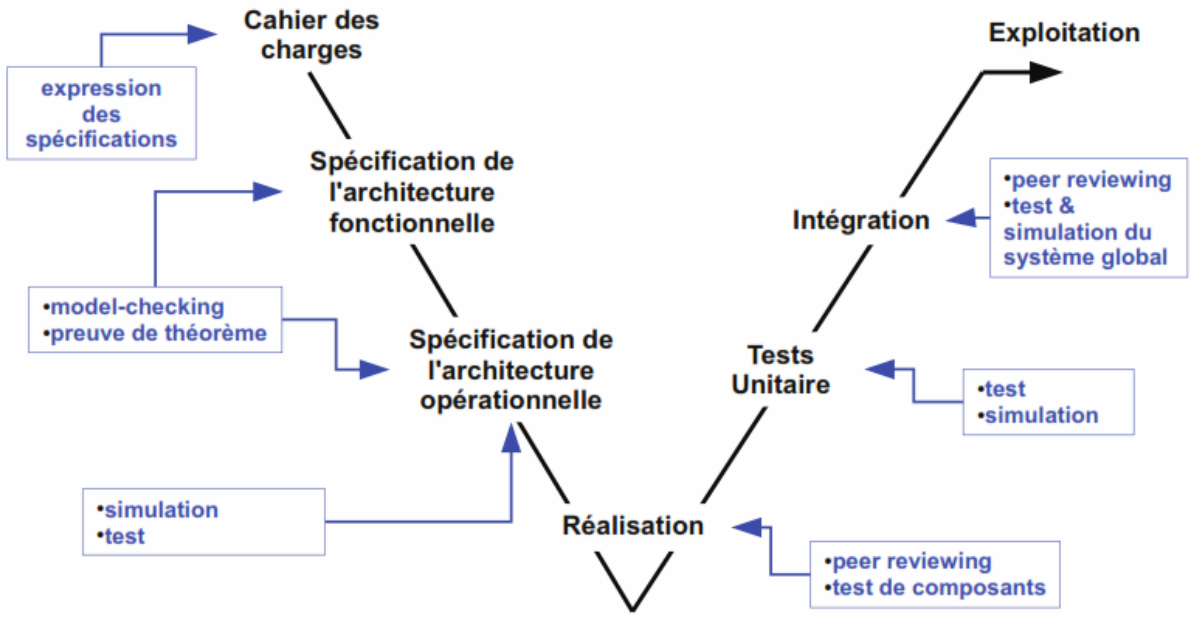


Figure 2.8. Vérification/Validation dans le cycle en V [55].

Le développement des systèmes embarqués temps réel (SETRs) suit majoritairement le parangon de développement nommé cycle en V qui, initialement, a été adopté pour le génie logiciel [56]. Ce cycle de développement anticipe les tests attendus de la partie montante dès

les phases de spécification et de conception de la partie descendante. Cette stratégie offre l'avantage de vérifier et de valider, de manière ascendante, le développement du SETR selon des exigences fonctionnelles (i.e., ce que doit faire le SETR) et non fonctionnelles (i.e., la manière dont le SETR exécute ce qu'il doit faire) établies respectivement en spécification et en conception.

Le cycle de développement en V (figure 2.8) peut être découpé en trois parties : la phase de spécification et de conception, la phase d'implémentation et finalement la phase de validation [55] :

1. Spécification et conception **:** cette étape commence par une analyse des besoins du cahier des charges afin de proposer une spécification des fonctionnalités qui est utilisée pour la conception des architectures fonctionnelles et opérationnelles.

2. Implémentation **:** en se basant sur l'architecture opérationnelle définie dans l'étape précédente, cette phase présente l'implémentation physique du produit. En effet, les différentes entités composant le futur produit sont conçues et implémentées individuellement. Ensuite, ces entités sont intégrées dans la conception du système global.

3. Validation **:** c'est la dernière étape du cycle et a comme objectif de garantir la conformité du futur produit aux exigences pré-définies. Elle est formée par deux sous-étapes : des tests sur les entités indépendantes et des tests sur le système global après intégration. L'étape de validation permet d'éliminer les erreurs de conception et/ou de réalisation ainsi que de valider l'architecture opérationnelle choisie pour le produit final.

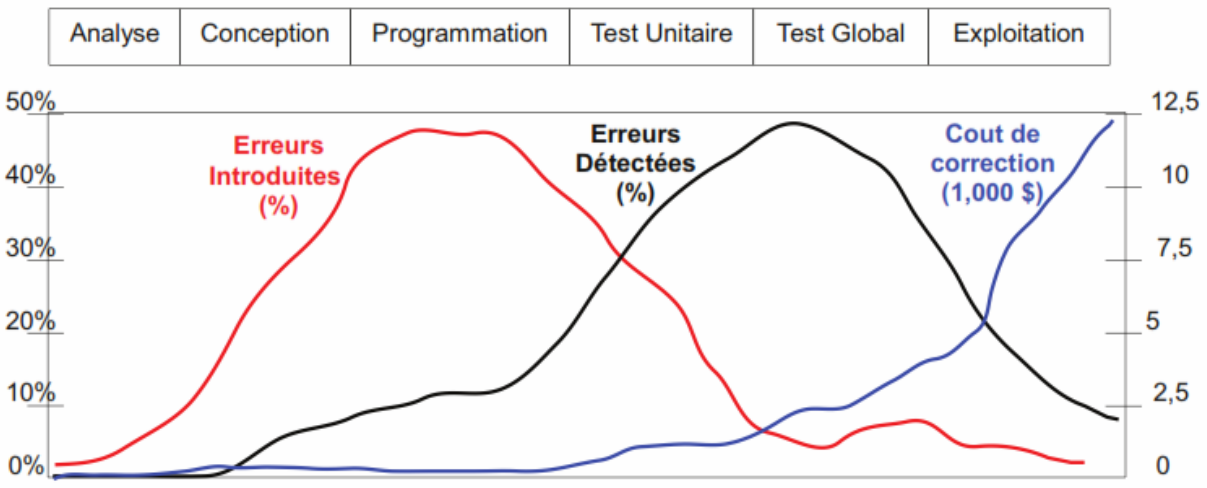


Figure 2.9. Erreurs dans le processus de développement [57].

La figure 2.9 montre que le coût de correction d'erreurs détectées dans les phases finales de développement ou après la fin de cycle de développement est souvent bien plus important que le coût de correction à un stade primaire, c'est-à-dire au début du cycle de développement [57], il est judicieux de commencer les tâches de vérification le plus tôt possible dans le cycle de développement. L'utilisation des méthodes de spécification et de vérification dans le cycle de développement des systèmes temps réel permet d'assurer un développement fiable, de réduire le coût de maintenance et d'avoir un système sûr de fonctionnement.

Les méthodes formelles peuvent être utilisées à différentes étapes du cycle de vie, et pour des objectifs différents. Nous listons ces utilisations possibles en cherchant, dans la mesure du possible, à donner des critères sur l'impact sur le cycle de vie [58].

Phase de spécification
Au niveau de la spécification, c'est à dire du passage des besoins au cahier des charges, l'intérêt des méthodes formelles est clair. Elles obligent à un effort de *formalisation* qui, *a priori*, fournit une aide à la modélisation. En effet, la formalisation peut obliger à un certain degré de questionnement, d'autant plus si cette formalisation est assistée par des outils. Cette phase est souvent la plus critique.

Conception et codage
Partant d'une spécification, les méthodes formelles peuvent être utilisées pour passer progressivement de la spécification au code. Suivant la distance entre les deux, cette étape peut essentiellement prendre deux formes :
- la production automatique de code ;
- l'assistance à la transformation de la spécification au code.

Processus automatique. Le premier type d'outils est adapté lorsqu'il existe un processus systématique de traduction qui, de plus, fournit un code ayant les qualités requises. Ceci signifie d'une part que la spécification est, d'une certaine manière, déjà exécutable et d'autre part qu'elle décrit effectivement le comportement désiré au niveau de l'exécution.
Lorsque l'automatisation est possible, le gain est indéniable puisqu'il supprime un niveau de développement. Ceci revient à offrir au développeur un langage de plus haut niveau.

Processus assisté. Lorsque la spécification est plus abstraite, ou bien n'a pas été conçue comme décrivant le comportement attendu à l'exécution, le processus de transformation peut être assisté formellement. C'est la technique de raffinement. Le développeur doit alors écrire lui-même les spécifications plus précises et la correction sera vérifiée formellement. Le gain est ici moins tangible car l'activité de développement devient plus coûteuse. Néanmoins, si cette activité est mise en place dans une approche globale de la qualité, ce gain peut être évaluable. Ceci a été le cas, par exemple, dans le projet Météor[2] [59] où les tests unitaires ont été supprimés pour les composants développés formellement.

2.6 Les formalismes de spécification
La phase de spécification repose sur un langage formel basé sur une syntaxe claire et précise avec une sémantique bien définie. Le choix de ce langage dépend du type de système à spécifier (fini, infini, séquentiel, ou concurrent) et aussi de la technique de vérification à appliquer.

Les spécifications comportementales sont 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 [60].

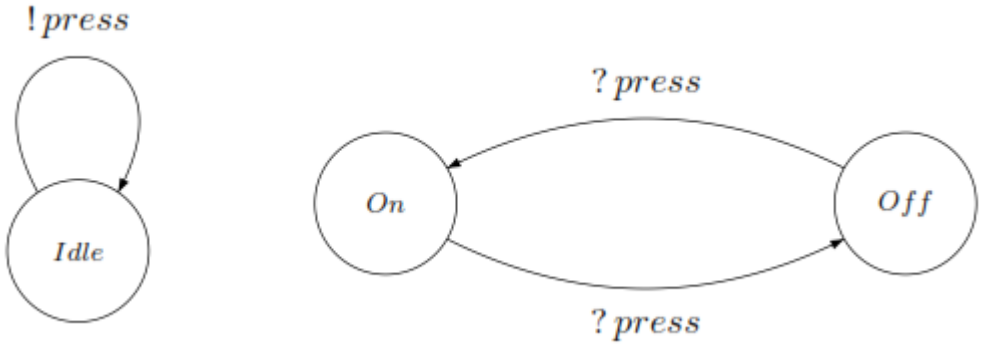
[2] Projet métro sans conducteur METEOR réalisé par *Matra Transport International.*

Il existe plusieurs approches pour la spécification des systèmes temps réel. Celles-ci sont basées sur des formalismes mathématiques de spécification de systèmes dynamiques.
Parmi ces formalismes il y a les systèmes de transitions comme les réseaux d'automates ou réseaux de Petri [61], les algèbres de processus [62] [63] ou encore les réseaux d'automates temporisés [64]. Une fois spécifiés dans un formalisme donné, les modèles des systèmes temps réels doivent être validés. Il faut donc démontrer que les contraintes non fonctionnelles sont satisfaites. L'exemple simple est de vérifier que le système produit des réponses à des stimulus de l'environnement dans un intervalle de temps défini. Étant donné le nombre de formalismes existants, nous décrivons dans ce qui suit, de façon non exhaustive, les plus communément utilisés.

2.6.1 Les automates temporisés

La modélisation de systèmes réels nécessite la plupart du temps la manipulation de *variables d'états*. Ces variables sont le plus souvent utilisées en tant que compteur, comme par exemple pour compter un nombre d'erreur ou de passage par une transition. Les systèmes qui se prêtent le mieux aux techniques de model-checking sont ceux facilement représentables par des automates (cf Définition 2.4).
Définition 2.4. (Automates) Un automate A est défini par le quintuplet $\langle Q, E, T, q_0, l \rangle$ où :
– *Q est un ensemble fini d'états ;*
– *E est l'ensemble fini des étiquettes des transitions ;*
– $T \subseteq Q \times E \times Q$ *est l'ensemble des transitions ;*
– q_0 *est l'état initial de l'automate*
– *l est l'application qui associe à tout état de Q l'ensemble fini des propriétés élémentaires vérifiées dans cet état.*

Lorsque l'on s'intéresse à un système complexe, il est souvent plus simple de le découper en sous-systèmes (ou modules). De la même manière, pour construire la modélisation globale d'un système de ce type, il est nécessaire de modéliser chaque sous-système. L'automate global modélisant le système est ainsi obtenu en synchronisant les automates de chaque module. Il existe de nombreuses manières de réaliser cette *synchronisation* mais le résultat, appelé le *produit synchronisé*, entraine une explosion du nombre d'états, et la modélisation de l'automate global en devient quasiment impossible.



(a) Interrupteur (b) Lampe

Figure 2.10. Exemple d'un réseau d'automates : la modélisation d'une lampe.

Le produit synchronisé de ces automates est un automate dont l'espace d'états est le produit des espaces d'états des automates composants, l'état initial est le *n*-uplet des états initiaux, l'alphabet d'actions est l'union des alphabets, et une transition. Le produit synchronisé offert

par la méthode de synchronisation par message est un cas particulier. Il s'agit de faire communiquer les différents automates modélisant le système global par l'envoi/réception de messages. L'émission d'un message m est notée $!m$ alors que la réception correspondante est notée $?m$. Pour que l'automate soit valide, chaque émission doit correspondre à une réception. La figure 2.10 donne un exemple de ce type de réseau avec la modélisation d'une lampe et d'un interrupteur [65].

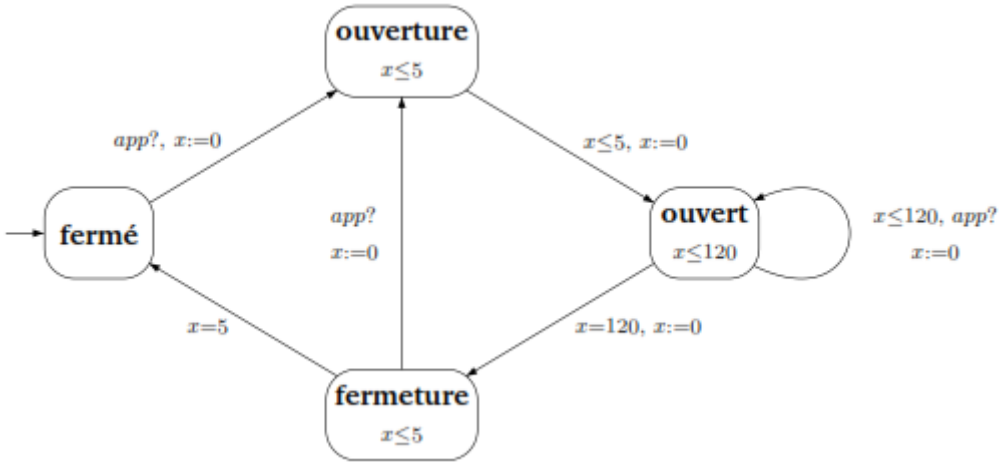


Figure 2.11. Exemple d'un automate temporisé modélisant une barrière.

Les automates temporisés ont été proposés par R. Alur et D. Dill [66] [67] en tant que modèle pour les systèmes temps-réel. Un automate temporisé est un automate fini classique qui peut manipuler des horloges évoluant de manière continue et synchrone dans le temps. En plus du nom de l'action, chaque transition de l'automate temporisé est étiquetée par une contrainte sur les valeurs des horloges (appelées aussi *gardes*), qui indique quand la transition peut être franchie, et un ensemble d'horloges à réinitialiser lors du franchissement de cette transition. Chaque état de l'automate est gardé par un invariant qui restreint les valeurs possibles des horloges pour lesquelles le système puisse demeurer dans cet état.

Définition 2.5. Un Automate temporisé A est un 5-uplet $\langle \Sigma, S, S_0, H, E \rangle$ tel que :

- Σ est un ensemble d'alphabet
- S est un ensemble finis d'états
- $S_0 \subseteq S$ est l'ensemble d'états initiaux
- H est un ensemble finis d'horloges
- $E \subseteq S \times S \times \Sigma \times 2^H_{fn} \times \Phi(H)$ est un ensemble de transitions. Un arc $\langle s, s', a, \lambda, \delta \rangle$ représente une transition de l'état s vers l'état s' en lisant le symbole a. L'ensemble $\lambda \subseteq H$ représente les horloges qui seront remises à zéro par cette transition. et δ est une contrainte temporelle sur H. La notation $\langle s, s', a, \lambda, \delta \rangle$ peut être écrite sous la forme suivante : $s \xrightarrow{a,\lambda,\delta} s'$

Dans [67] un automate temporisé est de la forme $\langle \Sigma, S, S_0, H, E, I \rangle$ où I est une fonction d'étiquetage de chaque état s par un *invariant* $I(s)$ dans $\Phi(H)$. Un système peut rester dans un état s tant que les valeurs actuelles des horloges satisfont l'invariant $I(s)$.

Les horloges d'un automate temporisé avancent toutes à la même vitesse et peuvent être comparées à des constantes et remises à zéro. Ainsi, les configurations associées à un automate temporisé ont deux types d'évolution possibles :
– soit le temps s'écoule et l'automate reste dans le même état. Les valeurs des horloges vont toutes être augmentées du délai d'attente.

– soit l'automate franchit une transition discrète, ainsi il passera à l'état suivant. Des contraintes sur les valeurs des horloges sont associées à cette transition. Elles doivent alors être satisfaites par les valeurs courantes de ces horloges avant le franchissement. Les transitions discrètes sont aussi accompagnées par des remises à zéro d'un sous-ensemble d'horloges.

Par exemple, l'automate temporisé de la figure 2.11 [68] décrit une barrière qui s'ouvre (en moins de 5 secondes) à l'approche d'une voiture et reste ouverte pendant 120 secondes avant de se refermer. L'automate part de l'état initial **fermé**. Lorsqu'une voiture s'approche de la barrière (signal **app ?**), il passe dans l'état **ouverture** pour arriver à l'état ouvert (en remettant l'horloge x à zéro) après un délai inférieur ou égal à 5 secondes. Dans cet état, le temps s'écoule et l'approche d'autres voitures (**app ?**) remet à chaque fois l'horloge x à zéro. Lorsque x vaut 120, la barrière commence à se refermer (l'automate va dans l'état **fermeture**). Elle se rabat après 5 secondes (l'automate revient à l'état **fermé**), à moins qu'une autre voiture ne s'approche.

Outre la représentation de systèmes temps-réel, les automates temporisés permettent de vérifier des propriétés temporelles sur un système. L'outil UPPAAL [69] permet la modélisation, la simulation et la vérification de systèmes temps-réel. La modélisation est basée sur une extension des automates temporisés et les propriétés à vérifier sont exprimées en logique temporelle.

2.6.2 Les réseaux de Petri

Les réseaux de Petri sont issus des travaux de Carl Adam Petri, en 1962. Ils définissent une notation formelle pour la modélisation et la vérification de systèmes concurrents. La manipulation de tels modèles par des outils d'analyse permet une vérification automatique de propriétés structurelles, ainsi que la vérification des systèmes considérés suivant des formules de logique temporelle.
Il est ainsi possible de vérifier l'absence d'un état interdit du système, la causalité entre états ou de caractériser les flux d'exécution dans l'architecture. Un réseau de Petri *(RDP)* est un graphe biparti orienté valué. Il a deux types de nœuds [70] :

1. *les places* : notées graphiquement par des cercles. Chaque place contient un nombre entier (positif ou nul) de marques (ou jetons). Ces derniers sont représentés par des points noirs.
2. *les transitions* : notées graphiquement par un rectangle ou une barre. Une transition qui n'a pas de place en entrée est appelée transition *source* et une transition qui n'a pas de place en sortie est appelée transition *puits*.

Les places et les transitions sont reliées par des arcs orientés où :

- *Un arc* relie, soit une place à une transition, soit une transition à une place mais jamais une place à une place ou une transition à une transition.
- Chaque arc est étiqueté par une valeur (ou un poids), qui est un nombre entier positif. L'arc ayant *k* poids peut être interprété comme un ensemble de *k* arcs parallèles. Un arc qui n'a pas d'étiquette est un arc dont le poids est égal à 1.

Un marquage est dénoté par un vecteur du nombre de jetons dans chaque place : la $i^{ème}$ composante correspond au nombre de jetons dans la $i^{ème}$ place [71].
Afin de pouvoir manipuler formellement ces représentations, les réseaux de Petri sont définis mathématiquement ainsi [72] :

Définition 2.6. (Réseau de Petri)
Un réseau de Petri est un réseau de type places et transitions défini par un tuple N =

(*P, T, Pre, Post*) où :
– *P* est un ensemble fini (l'ensemble des places de *N*) ;
– *T* est un ensemble fini (l'ensemble des *transitions* de *N*), disjoint de *P* ;
– *Pre, Post* ∈ N $^{|P|\times|T|}$ sont des matrices (les matrices d'incidence *amont* and *aval* de *N*).
C = *Post -Pre* est appelée *matrice d'incidence* de *N*.

La matrice d'incidence $C_{m\times n}$ d'un réseau de Petri N, où *m* est le nombre des places et *n* le nombre des transitions, est définie de la manière suivante :

$$C = [c_{i,j}] \tag{2.1}$$

Avec

$$c_{i,j} = \begin{cases} +O(t_j, p_i) & si \quad t_j \in \bullet p_i \\ -I(p_i, t_j) & si \quad t_j \in p_i \bullet \\ 0 & sinon \end{cases} \tag{2.2}$$

Où :

- $i = 1, 2, ..., m$ et $j = 1, 2, ..., n$.
- *I* et *O* sont des fonctions d'incidence avant de type $P \times T \rightarrow N$ (correspondant aux arcs allant d'une place à une transition) et respectivement arrière de type $T \times P \rightarrow N$ (correspondant aux arcs allant d'une transition à une place) ;
- On note $\bullet p = \{t | O(t, p) > 0\}$ l'ensemble des transitions d'entrée de la place *p* ;
- On note $p\bullet = \{t | I(p, t) > 0\}$ l'ensemble des transitions de sortie de la place *p*.

L'évolution d'un Réseau de Petri correspond à l'évolution de son marquage au fil du temps (évolution de l'état du système) : il se traduit par un déplacement des jetons pour une transition *t* de l'ensemble des places d'entrée vers l'ensemble des places de sortie de cette transition. Ce déplacement s'effectue par le franchissement de la transition *t* selon des règles de franchissement. Les transitions « consomment » les jetons entrants et en produisent d'autres en sortie. Elles ne peuvent être déclenchées que lorsque tous les jetons nécessaires sont disponibles ; elles permettent donc de synchroniser la circulation des jetons dans le réseau. Les places agissent comme des tampons permettant de stocker les jetons. Si la transition est validée, on peut effectuer le franchissement de cette transition : on dit alors que la transition est franchissable [70]. Le franchissement consiste à :

- Retirer *W*(*p, t*) jetons dans chacune des places en entrée *p* de la transition *t*.
- Ajouter *W*(*t, p*) jetons à chacune des places en sortie p de la transition *t*.

Où :

- *W*(*P, T*) : "*Pré(p, t)*" est le poids de l'arc allant de *P* à *T*.
- *W*(*T, P*) : *"Post (p, t)*" est le poids de l'arc allant de *T* à *P*.

Le réseau de Petri représenté par la figure 2.12 est un réseau de Petri marqué avec marquage initial M_0. Ainsi, le franchissement de t_1 est illustré dans la même figure.

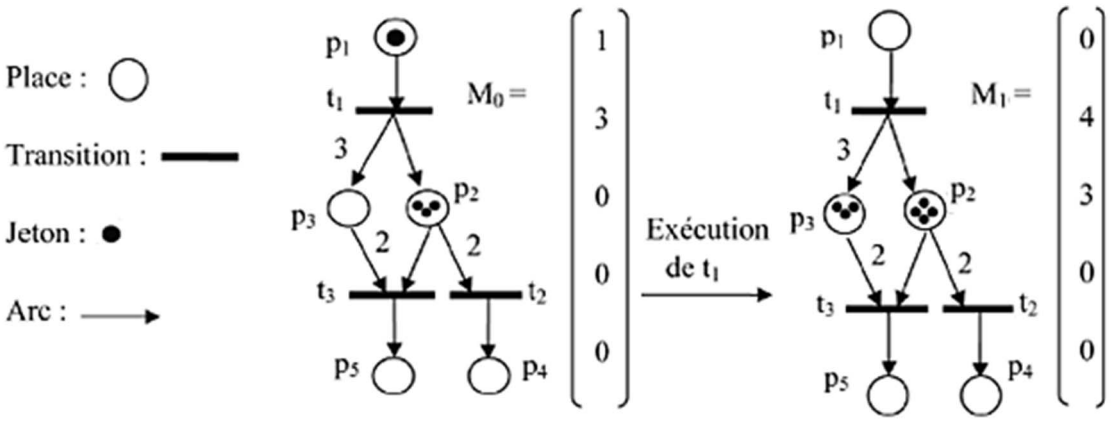


Figure 2.12 Le réseau de Petri (RdP) [73].

2.6.2.1 Propriétés comportementales des RdPs

L'analyse d'une fonction ou d'un système séquentiel passe par l'étude des propriétés du RdP qui le représente. Parmi ces propriétés, nous citerons celles qui permettent d'affirmer que les spécifications incluses dans le modèle RdP sont correctes. C'est ainsi que nous pourrons démontrer que le nombre d'états pouvant être atteints est fini (RdP borné) ou qu'un réseau (donc le système représenté) est sans blocage (RdP pseudo-vivant, RdP vivant, RdP propre). On mettra également en évidence les conflits entre plusieurs évolutions possibles (et qui sont autant d'ambiguïtés à lever) (Conflits structurel et effectif), ou encore la gestion du partage des ressources (Exclusion mutuelle) [74].

- **RdP borné**

Un RdP est borné pour un marquage initial donné si, quel que soit le marquage accessible atteint M et quelle que soit la place p considérée, le nombre de jetons contenus dans cette place est inférieur à une borne k :

$$\forall M \text{ et } \forall p : M(p) \leq k \qquad (2.3)$$

On dira également que le nombre d'états accessibles à partir de l'état initial est fini, le graphe d'états équivalent peut donc être construit.

Interprétation : un système physique présente toujours un nombre d'états fini ; il en est ainsi, par exemple, d'un stock dont la capacité est toujours limitée. Toute spécification d'un système réel par RdP doit présenter un graphe borné. Lorsque la borne est égale à 1, on dit que le RdP est sauf ou binaire.

- **RdP vivant**

Un RdP est vivant pour un marquage initial donné si, pour tout marquage M accessible à partir du marquage initial, il existe une transition t, il existe une séquence S de franchissements qui inclut la transition t :

$$\forall M \text{ et } \forall p \in P, \exists S : M \xrightarrow{S} M' \qquad \text{tel que } t \subset S \qquad (2.4)$$

Interprétation : la vivacité indique que le système représenté est sans blocage, mais également qu'il n'existe pas de branche morte dans le modèle graphique, donc de spécification incomplète.

- **RdP pseudo-vivant**

Un RdP est pseudo-vivant pour un marquage initial donné si, pour tout marquage M accessible à partir du marquage initial, il existe une transition t sensibilisée :

$$\forall M \text{ et } \forall p \in P, \exists t \in T : M(p) \geq I(p, t) \qquad (2.5)$$

Interprétation : cette propriété traduit l'absence de blocage total dans le système spécifié.

- **RdP réinitialisable**

Un RdP est réinitialisable si, pour tout marquage M accessible à partir du marquage initial, il existe une séquence S de franchissements qui ramène au marquage initial :

$$\forall M, \exists S : M \xrightarrow{S} M_0 \quad (2.6)$$

Interprétation : la plupart des processus industriels ont un fonctionnement répétitif. Il est donc très important de vérifier si les RdP qui les représentent sont réinitialisables.

- **Conflits structurel et effectif**

Deux transitions sont en conflit structurel lorsqu'elles possèdent une place d'entrée commune. Le conflit devient effectif si le marquage de la place commune sensibilise les deux transitions. Dans ce cas, le franchissement d'une transition empêche le franchissement de l'autre. Une seule transition sera franchie, mais rien dans le réseau ne permet de prévoir laquelle.
Interprétation : un conflit effectif signifie qu'il y a non-déterminisme du réseau, donc que l'évolution du système décrit présente une partie aléatoire.

- **Exclusion mutuelle**

Deux places sont en exclusion mutuelle ou mutuellement exclusives si pour un marquage initial M_0 donné, elles ne peuvent être simultanément marquées quel que soit le marquage M atteint à partir de M_0.
Interprétation : on rencontre l'exclusion mutuelle dans tout système comprenant un partage de ressource.

2.6.2.2 Les Réseaux de Petri de Haut Niveau

Pour l'utilisation des réseaux de Petri dans la modélisation des systèmes réels, plusieurs auteurs ont trouvé qu'il est convenable d'étendre le formalisme de réseau de Petri pour compacter la représentation de modèle ou pour étendre le pouvoir de modélisation du formalisme de réseau de Petri. Ce qui a donné naissance aux réseaux de Petri de haut niveau.

a) Réseaux de Petri colorés

Les réseaux de Petri colorés (CP-nets[3] ou CPN) [75] ajoutent aux réseaux de Petri, des primitives pour la définition de types de données et la manipulation de valeurs de données. Nous ne les utilisons pas dans la représentation interne du modèle formel mais uniquement pour représenter certains des réseaux de Petri génériques que nous produisons par transformation, de manière plus compacte.
Dans un réseau de Petri coloré, un jeton a une valeur appartenant à un type.
• Un type de donnée est associé à une place indiquant qu'elle ne peut contenir que des jetons appartenant à ce type. Un type peut être le produit cartésien de types existants. Une place peut contenir un multiset de jetons (i.e. contenir des jetons de même valeur).
• Un arc entrant d'une transition peut avoir des gardes sur la sensibilisation de la transition. Une transition n'est franchissable que lorsqu'un sous ensemble des jetons dans les places sources satisfait les gardes sur les arcs entrants de la transition.
• Les arcs sortants peuvent contenir des expressions arithmétiques spécifiant comment calculer les valeurs produites par la transition.
• les gardes et les expressions sont exprimées à l'aide de paramètres (ou variables) et sont évaluées en associant la valeur d'un jeton à un paramètre (*binding*).

[3] Coloured Petri Nets

Le développement des réseaux de Petri coloré a été motivé par le désir de développer un langage de modélisation bien-fondé théoriquement et assez versatile en même temps. Ils sont utilisés pour les systèmes de taille et de complexité qu'on retrouve dans les projets industriels.

b) Réseau de Petri Objet

Les réseaux de Petri Objet *(OPN)* étendent le formalisme des réseaux de Petri colorés avec une intégration complète des propriétés orientées objet y compris l'héritage, le polymorphisme et la liaison dynamique. L'orientation objet fournit des primitives de structuration puissante permettant la modélisation des systèmes complexes [76].

2.6.2.3 Extensions temporelles des réseaux de Petri

Les deux principales extensions temporelles des réseaux de Petri sont les réseaux de Petri *temporisés* [77] et les réseaux de Petri *temporels* [78]. Dans les premiers, le temps est représenté par un délai minimal (ou exact, dans le cas d'un fonctionnement « au plus tôt » du réseau) permettant l'occurrence d'un événement. Les réseaux temporisés constituent une classe de réseaux incluse dans les réseaux de Petri temporels. Le temps est intégré au sein de ces derniers sous la forme d'un intervalle contraignant les instants de tir des transitions.

Différents types de réseaux temporels ont été introduits. Ils diffèrent par l'élément auquel est associée la notion de temps : il peut tout aussi bien s'agir des arcs (réseaux de Petri *A-temporels* [79] [80]), des places (réseaux de Petri *P-temporels* [81]) ou des transitions (réseaux de Petri *T-temporels* [78]). Une étude complète de l'expressivité de ces modèles a ´été proposée dans [82]. Nous en retenons notamment qu'en apparence, les réseaux de Petri A-temporels et P-temporels sont très proches, leur règle de tir étant liée à la conjonction d'horloges locales tandis que, pour les réseaux de Petri T-temporels, seule la dernière horloge est prise en compte. Théoriquement, il s'avère qu'en sémantique forte (i.e. toute transition doit être tirée lorsque la borne temporelle supérieure de sa condition de tir est atteinte), les réseaux T-temporels et P-temporels sont incomparables, tandis que les réseaux A-temporels sont plus expressifs que les réseaux P-temporels et les réseaux T-temporels.

a) Les réseaux de Petri temporels

Parmi les différents types de réseaux temporels, nous allons décrire brièvement, par un exemple, les réseaux de Petri T-temporels (*Time Petri nets* en anglais) [78], qui étendent les réseaux de Petri avec des intervalles de temps [$a(t)$, $b(t)$] associés à chaque transition t du réseau. Pour être tirée, une transition t doit non seulement être sensibilisée mais aussi l'avoir été continûment pendant une durée comprise entre $a(t)$ et $b(t)$. Puisque nous avons fait le choix de travailler en sémantique forte, une fois atteinte la date $b(t)$, la transition t devra obligatoire être franchie, à moins d'avoir été désensibilisée puis nouvellement sensibilisée par le tir d'une autre transition.

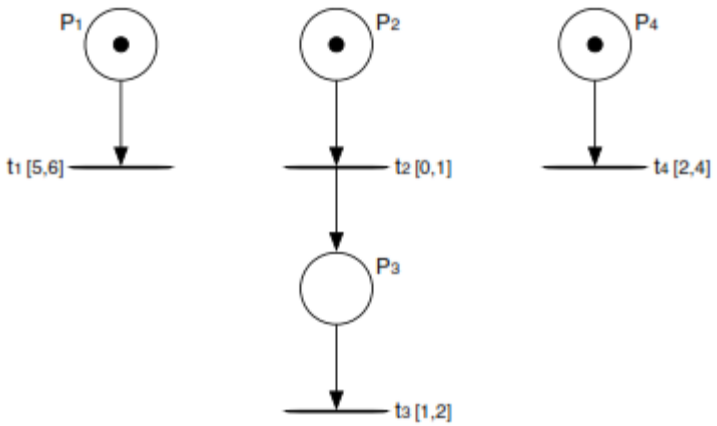


Figure 2.13. Un exemple de réseau de Petri temporel.

Considérons le réseau de Petri temporel de la figure 2.13, les transitions t_1, t_2 et t_4 sont sensibilisées au sens classique des réseaux de Petri. Si le temps s'écoule d'une unité de temps, les horloges associées à t_1, t_2 et t_4 auront alors la même valuation $x_1 = x_2 = x_4 = 1$. Dans cette configuration, seule la transition t_2 est franchissable.
Lors du franchissement de t_2, le jeton de la place P_2 est consommé et un jeton est généré dans la place P_3. La transition t_3 est alors sensibilisée et l'horloge associée est mise à zéro et déclenchée [83].

b) Les réseaux de Petri Temporisés
Il y a beaucoup de systèmes à événements discrets dont l'évolution dépend du temps qui est capitale surtout lorsque l'on veut évaluer les performances ou étudier les problèmes d'ordonnancement d'un système dynamique [70]. La nécessité de modéliser et d'étudier de tels systèmes a donné naissance aux RdPs Temporisés (RdPTs). La temporisation, indiquant la durée entre le début et la fin d'une opération, peut être associée soit aux places soit aux transitions. Il en résulte alors deux types de RdPs Temporisés : *RdPs P-Temporisés* et les *RdPs T-Temporisés* [84] [85].
Un RdPT est un doublet <*Q, Tempo*> tel que :

- *Q* est un RdP ordinaire marqué,
- *Tempo* est une application de l'ensemble *T* des transitions dans l'ensemble des réels rationnels.

La sémantique de l'application *Tempo* est que les transitions doivent rester dans la transition T_j au moins pendant la durée d_j associée à cette transition :

$$Tempo\ (T_j) = d_j \qquad (2.7)$$

La résolution des conflits pouvant se produire consiste à définir des priorités en fonction des temporisations. En effet, l'introduction des durées d_j permet de gérer la priorité de franchissement des transitions de sortie de la place P_i, l'ordre de franchissement étant donné à la transition de temporisation minimale / maximale.
Une transition *T* j, d'intervalle temporel $[\alpha j, \beta j]$, peut être franchie à un instant *t* si et seulement si on a : $t \in [\theta + \alpha j, \theta + \beta j]$, θ représentant l'instant où le marquage a validé la transition *T* j correspondant à l'instant du dernier franchissement de celle-ci. Un des problèmes posés par l'analyse des RdPTs est la gestion simultanée de plusieurs validations [86].
L'utilisation des réseaux de Petri est plus tournée vers la recherche que vers l'industrie. Le tableau 2.2 montre les domaines d'application de différents types de réseaux de Petri [87].

Tableau 2.2. Application des réseaux de Petri.

Réseau de Petri ordinaire	Modélisation de systèmes logiciels Modélisation de processus d'affaires Gestion des flux, Programmation concurrente Génie de la qualité, Diagnostic
Réseau de Petri généralisé	Gestion des flux complexes Modélisation de chaînes logistiques Utilisation pour les techniques quantitatives
Réseau de Petri temporisé	Gestion du temps Modélisation d'attentes
Réseau de Petri coloré	Modélisation des systèmes de collaboration
Réseau de Petri continu	Modélisation de réactions chimiques

2.7 Les logiques temporelles

Une fois le système formellement modélisé, on doit formaliser les propriétés à vérifier. On fait alors la distinction entre les propriétés comportementales liées au temps logique, des propriétés temps réel liées, elles, au temps physique. Les spécifications logiques décrivent des propriétés " système " à valider, par exemple, des propriétés de sûreté (« quelque chose de mauvais n'arrivera jamais ») ou de vivacité (« quelque chose de bon arrivera »). Les différentes classes des propriétés pouvant être vérifiées sont illustré dans le tableau 2.3 [88]. Ces types de propriétés pour pouvoir être vérifiées, doivent être exprimés dans des logiques temporelles, dont les plus connues sont LTL (« *Linear Temporal Logic* ») [89] [90], CTL *(« Computation Tree Logic »*) [91] et CTL* (regroupement de LTL et CTL). Les logiques temporelles sont des langages de spécification qui permettent d'énoncer des propriétés faisant intervenir des notions d'ordonnancement dans le temps, par exemple *"Après la pluie, le beau temps"*. Une logique temporelle est dite de temps arborescent quand elle permet d'examiner les états et leur arbre d'exécution, contrairement à la logique de temps linéaire qui énonce des propriétés sur une exécution donnée.

Les deux groupes de logiques linaires et arborescentes introduisent quelques opérateurs et symboles utilisés dans la plupart de ces logiques, permettant d'illustrer l'aspect temporel dans les spécifications des systèmes temps réel. Le comportement d'un système est décrit par une séquence d'états s avec s_0 un état initial. Un état est défini par des valeurs associées à des variables du système.

Tableau 2.3. Classes des propriétés pouvant être vérifiées.

Types de propriétés	**Signification**
Atteignabilité/Accéssiblité (*reachability)*	Sous certaines conditions, un état du système peut être atteint *« une panne peut survenir »*
Sûreté (*safety*)	Sous certaines conditions, quelque chose n'arrive jamais *« une panne est impossible »*
Vivacité (*liveness*)	Sous certaines conditions, quelque chose finit par avoir lieu *« une panne arrivera un jour »*
vivacité bornée (*bounded liveness*)	Sous certaines conditions, quelque chose finit par avoir lieu avant un certain temps *« une panne arrivera dans la journée »*
Equité (*fairness*)	Sous certaines conditions, quelque chose a lieu infiniment souvent *« le système marche infiniment souvent »*
absence de blocage (*no deadlock*)	Le système ne se trouve jamais dans un état qu'il ne peut plus quitter *« une panne peut toujours être réparée »*

2.7.1 Logique temporelle linéaire (LTL)

La logique LTL est un sous-ensemble de la logique CTL* [92]. Elle permet d'exprimer des propriétés d'exécution en prenant en compte le futur. Il est, par exemple, possible d'exprimer qu'une certaine propriété sera finalement vraie ou bien qu'une condition soit vraie jusqu'à ce qu'une autre le devienne. Le domaine d'interprétation d'une formule LTL est un ensemble de chemins tel que chaque chemin est une séquence infinie d'états $s \in S$. L'évaluation de la satisfaction de la propriété se fait donc pour chaque chemin d'exécution indépendamment des autres, sans prendre en compte les entrelacements des différents futurs possibles à un instant donné de l'exécution. Autrement dit, l'évaluation se fait sur un ensemble d'exécutions indépendantes plutôt que sur un arbre des exécutions possibles (figure 2.14).

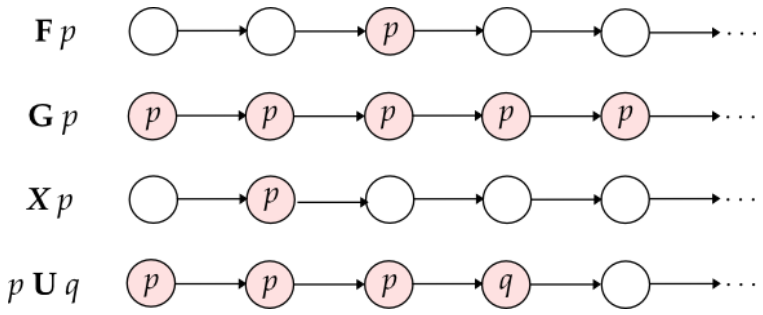


Figure 2.14. Illustration des opérateurs LTL.

Définition 2.6 (Formule LTL). Une formule LTL est définie par :

- Un ensemble fini *AP* de variables de propositions (propositions atomiques)
- Des opérateurs logiques : ¬, ∨, ∧, ⇒, ⇔, *vrai* et *faux*
- Des opérateurs modaux :
  - **X** (autre syntaxe : ○) : *neXt* (demain) tel que **X***p* signifie que *p* est vraie dans l'état suivant le long de l'exécution ;
  - **G** (autre syntaxe : □) : *Globally* (toujours) tel que **G***p* signifie que *p* est vraie dans tous les états ;
  - **F** (autre syntaxe : ◊) : *Finally* (un jour) tel que **F***p* signifie que *p* est vraie plus tard au moins dans un état ;
  - **U** : *Until* (jusqu'à ce que) tel que *p* **U** *q* signifie que p est toujours vraie jusqu'à un état où *q* est vraie ;
  - **R** : *Release* tel que *p* **R** *q* signifie que *q* est toujours vraie sauf à partir du moment où *p* est vraie, sachant que *p* n'est pas forcément vraie un jour.

Exemple 1. La propriété d'invariance *« Durant toute l'exécution, x est différent de 0 »* équivaut à la formule LTL : **G**¬ ($x = 0$).

Exemple 2. **G**($p \rightarrow$ **F***q*) signifie *« Pendant toute l'exécution, si p est vraie à un état donné, alors q sera finalement vraie au moins dans un des états suivants. »*

Exemple 3. La propriété de vivacité « *si le bouton de lancement est pressé, le missile sera lancé »* s'exprimerait donc en LTL de la manière suivante :

$$\Box(appui\_bouton \Rightarrow \Diamond lancement\_missible).$$

2.7.2 Logique temporelle arborescente (CTL)

La logique temporelle arborescente CTL permet d'exprimer des propriétés sur l'arbre d'exécution du programme. Elle considère plusieurs futurs possibles à partir d'un état du système plutôt que d'avoir une vue linéaire du système considéré. Cette logique introduit des quantificateurs de chemin par rapport à LTL. Les opérateurs temporels (comme **F**, **G** et **X**) doivent être directement précédés par les quantificateurs **E** qui signifie il existe un chemin d'exécution ou **A** qui signifie pour tous les chemins d'exécution. Nous exposons dans la figure 2.15 des exemples de propriétés exprimées avec les opérateurs CTL. Par exemple, la propriété **EF***p* se traduit par il existe un chemin où p est vraie.

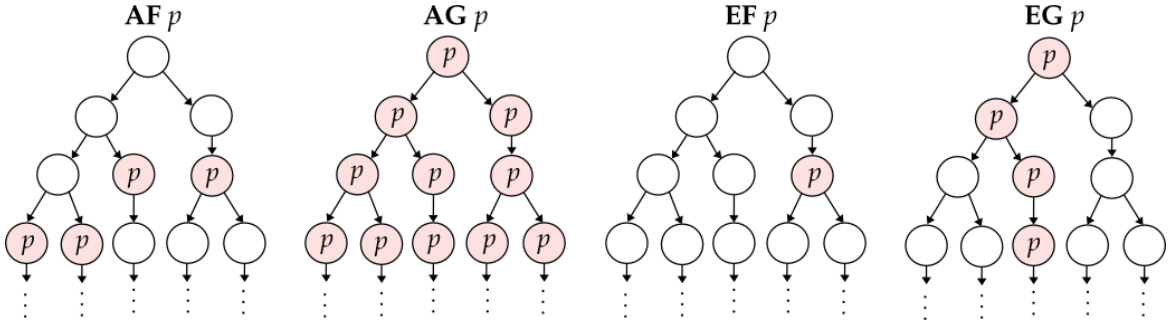


Figure 2.15. Illustration des opérateurs CTL.

2.7.3 Les logiques temporelles temporisés

Pour exprimer les propriétés temps réel, des extensions aux logiques temporelles existent, on peut noter : MTL (*Metric temporal logic*) [93], *Timed CTL* (*TCTL*) qui est une extension de CTL permettant d'exprimer les propriétés temporelles d'une manière uniforme, où la sémantique temporelle se base sur le temps dense [94]. On peut aussi utiliser une logique spécifique comme la logique temps réel [JM86]. Les logiques temporisées dites aussi temps-réel (*timed logic* ou simplement *real-time logic*) [95] [96] permettent d' exprimer ce genre de propriétés pour un modèle de temps continu, discret ou carrément hybride, sur des automates temporisés.

Il existe de nombreuses extensions de logiques temporelles permettant d'exprimer des contraintes quantitatives sur les délais des actions d'un système à analyser. La première consiste à compléter l'opérateur temporel U (*until*) avec une contrainte de la forme $\sim k$ avec $\sim \in \{=, <, \leq, \geq, >\}$ et $k \in$ N. La formule $\varphi\, U_{\sim k}\, \psi$ est vraie pour un chemin p, s'il existe un état l situé à moins de k unités de temps de l'état initial vérifiant ψ, et tel que tous les états le précédant vérifient φ. Les opérateurs F et G sont étendus de la même façon. L'exigence « l'alarme doit se déclencher en au plus 3 secondes après la détection d'une fuite de gaz » s'écrit en logique TCTL (*Timed CTL*) [97] [98] [99] :

$$AG(pb \Rightarrow AF_{\leq 3}\ alarme) \qquad (2.8)$$

Ou en TLTL *(Timed LTL)*

$$G(pb \Rightarrow F_{\leq 3}\ alarme) \qquad (2.9)$$

Une autre méthode consiste à ajouter des horloges aux logiques temporelles (on parle des horloges de formules), des opérateurs de remises à zéro, et des contraintes du même types que celles utilisées dans les automates. La formule précédente s'écrit alors de la manière suivante :

$$AG(pb \Rightarrow x\ in\ (AF(x < 3 \wedge alarme))) \qquad (2.10)$$

L'opérateur *in* réinitialise l'horloge x à zéro lorsqu'on rencontre un état vérifiant *pb*, et il suffit donc de vérifier que $x < 3$ lorsqu'on rencontre un état vérifiant *alarme* pour s'assurer que le délai séparant les deux positions est bien inférieur à 3. Cette méthode permet d'exprimer des propriétés plus fines mais est moins intuitive que la précédente. Il est également possible

d'étendre les logiques modales avec des horloges de formules et d'utiliser des points fixes pour exprimer des propriétés sur le comportement des systèmes.

2.7.3.1 Extension temporisée de CTL

La logique CTL ne permet de décrire que des propriétés qualitatives. Une façon d'introduire le temps dans cette logique est de borner la portée des opérateurs temporels. TCTL (*Timed Computation Tree Logique*) est une logique temporisée, extension de CTL, qui permet d'exprimer des propriétés sur des systèmes temps réels dits temporisés c.à.d des systèmes dont le comportement est conditionné par le passage du temps.

La logique TCTL est interprétée sur des systèmes de transitions temporisées [100]. L'opérateur U est assorti d'une contrainte $\sim c$ où $\sim$ représente un opérateur de comparaison parmi $\{=, <, \leq, \geq, >\}$, et c est une constante entière ($c \in$ N). Ce qui permet de contraindre les dates d'occurrence de la propriété ψ.

Définition 2.7. La syntaxe de TCTL est définie par :

$$\varphi, \psi ::= P \mid \neg \varphi \mid \varphi \wedge \psi \mid \varphi \; U_{\triangleright\triangleleft c} \; \psi \mid A \; \varphi U_{\triangleright\triangleleft c} \; \psi \qquad (2.11)$$

Où P est une proposition atomique.

2.7.3.2 Extension temporisée de LTL

La logique MTL (*Metric Temporal Logic*) [93] est une extension temporisée de LTL.

Définition 2.7. La syntaxe de MTL est définie par :

$$\varphi, \psi ::= P \mid \neg \varphi \mid \varphi \wedge \psi \mid \varphi \, U_I \, \psi \qquad (2.12)$$

Où P est une proposition atomique et I est un intervalle.

2.7.4 Choix d'une logique temporelle

Les opérateurs temporels permettent d'exprimer des propriétés sur les enchaînements d'états appelés exécutions. Les logiques temporelles n'ont pas toutes ni la même expressivité ni les mêmes contraintes. Par exemple les logiques LTL (*Linear Temporal Logic*) et CTL (*Computational Tree Logic*) n'ont pas le même pouvoir expressif. Toutes les deux exigent un automate à état comme modèle d'entrée. La différence entre les logiques temporelles provient de l'ensemble d'opérateurs temporels qui peut être utilisé et des objets sur lesquels ils sont interprétés. Selon les logiques, les exécutions sont soit des séquences d'états, soit des arbres qui représentent l'évolution du système. Le choix d'une logique ou de l'autre dépend donc des besoins d'analyse, et moins des outils dont on dispose.

À partir de la classification des propriétés dynamiques, présentée dans le tableau 2.4 [101], Nous pouvons voir que les logiques temporelles CTL et LTL ne permettent pas d'exprimer toutes les propriétés.

Tableau 2.4. Pouvoir d'expression des logiques LTL et CTL.

Propriété	**Logique temporelle LTL**	**Logique temporelle CTL**
Sûreté	Exprimable	Exprimable
Vivacité	Exprimable	Exprimable
Atteignabilité	Ce type de propriété n'est exprimable que dans certains cas particuliers.	Il est possible de spécifier toutes les propriétés d'atteignabilité.
Equité	Exprimable	Non exprimable
Absence de blocage	Non exprimable	Exprimable

La logique temporelle linéaire LTL s'avère plus naturelle en pratique pour spécifier les comportements attendus du système ou de l'environnement. De plus elle permet d'exprimer des notions utiles comme l'équité, les contre-exemples retournés sont simples (trace d'exécution) et ces logiques sont adaptées à la vérification. Par contre LTL manque parfois d'expressivité.

À l'inverse, la logique temporelle arborescente CTL s'avère parfois contre-intuitive (environnement, équivalence de modèle) et les contre-exemples retournés sont difficiles à interpréter (arbres d'exécution). CTL a cependant l'énorme avantage d'avoir des algorithmes de model-checking polynomiaux, alors que ceux de LTL sont PSPACE donc exponentiel en pratique. CTL peut aussi exprimer certaines propriétés utiles absentes de LTL et l'équité peut s'obtenir avec des algorithmes ad hoc.

2.8 Conclusion

Ce chapitre introduisait quelques notions de base sur la spécification et la vérification formelle de systèmes réactifs en général et temps-réel en particulier. L'utilisation des méthodes formelles dans le contexte du développement d'un système temps-réel permet d'exprimer ce système avec précision et donc disposer d'un support considérable d'outils d'analyse et de vérification. Dans ce chapitre, nous avons dressé une récapitulation des formalismes de spécification existants.
Le développement de systèmes temps réel doit s'appuyer sur des étapes de spécification, conception et implémentation. Les contraintes temps réel doivent être exprimées à chaque étape du développement.
Dans le chapitre suivant, nous introduirons les notions de base concernant la logique de réécriture et l'utilisation du langage Maude pour la spécification formelle et la vérification des systèmes temps réel.

CHAPITRE 3

La logique de réécriture comme formalisme de spécification et vérification des systèmes temps réel

Sommaire

3.1 Introduction

La logique de réécriture, dotée d'une sémantique saine et complète, a été introduite par Meseguer [102] [103]. Cette logique unifie plusieurs modèles formels qui expriment la concurrence. La logique de réécriture est une logique qui permet de raisonner d'une manière correcte sur les systèmes concurrents non-déterministes ayant des états et évoluant en termes de transitions. Elle explique n'importe quel comportement concurrent dans un système avec une sémantique de 'vraie concurrence', par des déductions dans cette logique. Le langage de la logique de réécriture Maude [104] [105] est l'un des langages les plus puissants dans la spécification formelle, la programmation et la vérification des systèmes concurrents.

Le but est de vérifier que tous les états désirés peuvent être atteints et que l'ensemble des états non désirés ne l'est pas. Une des méthodes de vérification privilégiée dans ce cas est le model-checking, méthode de vérification automatique, exhaustive et systématique de tous les états possibles d'un système à partir d'un état initial. En cas de détection d'une erreur, un contre-exemple est généré pour illustrer de quelle façon la propriété à vérifier a été violée. Lorsque ce type de vérification est inclus dans le processus de conception d'un système, il permet la détection précoce d'erreurs et de comportements indésirables.

Notre objectif est donc de choisir un langage de spécifications formel cible qui autorise l'application de techniques de model-checking.

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.

Dans ce qui suit, nous allons montrer comment la logique de réécriture permet de spécifier les systèmes temps réels. Dans ce chapitre, nous présentons la logique de réécriture et son langage Maude.

3.2 La logique de réécriture

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 [104] [106] [102] [107]. Par contre la partie statique est spécifiée par des équations. Une *signature* dans la logique de réécriture est une théorie équationnelle *(∑,E)*, ou ∑ est une signature équationnelle (alphabet) et *E* un ensemble de ∑-*équations*. La réécriture opère sur les classes d'équivalence de termes modulo l'ensemble des équations *E*.

Une théorie de réécriture $\mathfrak{R}$ est un quadruplet $\mathfrak{R}$ = *(∑, E, L, R) ou* ∑ une *signature* (ranked alphabet) de symboles de fonctions, *E* est l'ensemble des ∑-*équations*, *L est* l'ensemble des *étiquettes et R étant l'ensemble des paires* $R \subseteq L \times (T_{\sum,E}(X))^2$ tel que le premier composant est une étiquette et le second est un pair des classes d'équivalence des termes modulo les équations *E*, avec $X = \{x_1, \ldots, x_{n,\ldots}\}$ est un ensemble comptable, possible infinie de variables. Une théorie $\mathfrak{R}$ = *(∑, E, L, R)* devient une spécification exécutable du système temps réel qu'elle formalise. Les éléments de *R* s'appellent *les règles conditionnelles de réécriture*. Ils décrivent les transitions élémentaires et locales dans un système temps réel. Pour une réécriture $(r, [t], [t'])$, $([u_1],[v_1]),\ldots, ([u_k],[v_k])$), nous utilisons la notation :

$$r : [t] \rightarrow [t'] \; if \; [u_1] \rightarrow [v_1] \wedge \ldots \wedge [u_k] \rightarrow [v_k], \qquad (3.1)$$

tel que $[t]$ représente la classe d'équivalence du terme *t*. Une règle *r* exprime que la classe d'équivalence contenant le terme *t* a changé à la classe d'équivalence contenant le terme *t'* si la partie conditionnelle de la règle, $[u_1] \rightarrow [v_1] \wedge \ldots \wedge [u_k] \rightarrow [v_k]$, est vérifiée.

Soit une théorie de réécriture $\mathfrak{R}$, on dit qu'une séquence *r* : [*t*]→ [*t* '] est prouvable dans $\mathfrak{R}$, où *r* : [*t*] →_ [*t'*] est $\mathfrak{R}$-*réécriture concurrente*, et on écrit *R* |- [*t*] → [*t* '] si et seulement si [*t*] → [*t'*] peut être obtenu par une application finie des règles de déduction suivantes (figure 3.1) :

- ***La réflexivité*** *: pour chaque terme [t]* $\in T_{\sum,E}(X)$, $\overline{[t] \rightarrow [t']}$ *où* $T_{\sum,E}(X)$ *est l'ensemble des* $\sum$*-termes avec variables construits sur la signature* $\sum$ *et les équations E.*

- ***La congruence*** *: pour chaque fonction f* $\in \sum_n$ *, n*$\in$ *N,*

$$\frac{[t1] \rightarrow [t1'] \dots [tn] \rightarrow [tn']}{[f(t1, \dots, tn)] \rightarrow [f(t1', \dots, tn')]} \qquad (3.2)$$

- ***Le remplacement*** *: pour chaque règle r : [t(*x_1*, …,*x_n*)]* → *[t' (*x_1*, …,*x_n*)] dans R :*

$$\frac{[w1] \rightarrow [w1'] \dots [wn] \rightarrow [wn']}{[t(\overline{w} / \bar{x})] \rightarrow t'(\overline{w'} / \bar{x})} \qquad (3.3)$$

Sachant que $t(\overline{w} / \bar{x})$ *dénote la substitution simultanée de* x_i *par* w_i *dans t avec* $\bar{x}$ *représentant* $x_1, \dots, x_n$.

- ***La transitivité*** *:*

$$\frac{[t1] \rightarrow [t2\,]\ [t2] \rightarrow [t3]}{[t1] \rightarrow [t3]} \qquad (3.4)$$

De manière générale, la déduction dans la logique de réécriture est une itération des étapes suivantes [109] :

1. La règle de *remplacement* identifie toutes les règles de réécriture dont le membre gauche correspond à un sous-terme de l'état global courant. Comme la logique de réécriture est une logique de changement, la règle de réflexivité, appliquées aux sous-termes non identifiés, les transforme mais en eux-mêmes.

2. Les règles de réécriture, identifiées par la règle de *remplacement* ainsi que la règle de *réflexivité*, sont exécutées en concurrence et indépendamment les unes des autres. La règle de *congruence* compose les effets, membres droits, de ces règles pour construire le nouveau terme global. Les étapes (1) et (2) sont réitérées jusqu'à ce qu'il n'y ait plus de règle applicable.

3. Enfin, la règle de *transitivité* construit la séquence des réécritures faites du terme initial jusqu'au terme final. La séquence ainsi construite correspond à un calcul possible dans le système concurrent.

Remarque : Les règles de *congruence* et de *remplacement* peuvent être vues sous un autre angle. La première règle exprime que des règles de réécriture disjointes (des règles de réécriture sont disjointes si elles n'ont pas de sous-termes communs) peuvent être exécutées en concurrences. Alors que la seconde règle permet des réécritures imbriquées, i.e., imbrication de la réécriture des sous termes $w_i \longrightarrow w'_i$ dans celle du terme composite $t \longrightarrow t'$. Elle indique que deux sous termes différents peuvent être réécrits en parallèle même si leurs racines, terme composite, ne sont pas disjointes [109].
La *réflexivité* introduit le calcul « identité », c'est-à-dire que les sous-termes qui ne sont pas susceptibles de changer suite à l'application d'une étape de réécriture doivent être réécrits en utilisant la règle de *réflexivité*. Cette règle découle du fait que la logique de réécriture est une

logique de « changement », tous les termes et sous-termes doivent être changés (substitués) durant une étape de réécriture. Les termes qui ne changent pas se réécrivent donc en eux même.

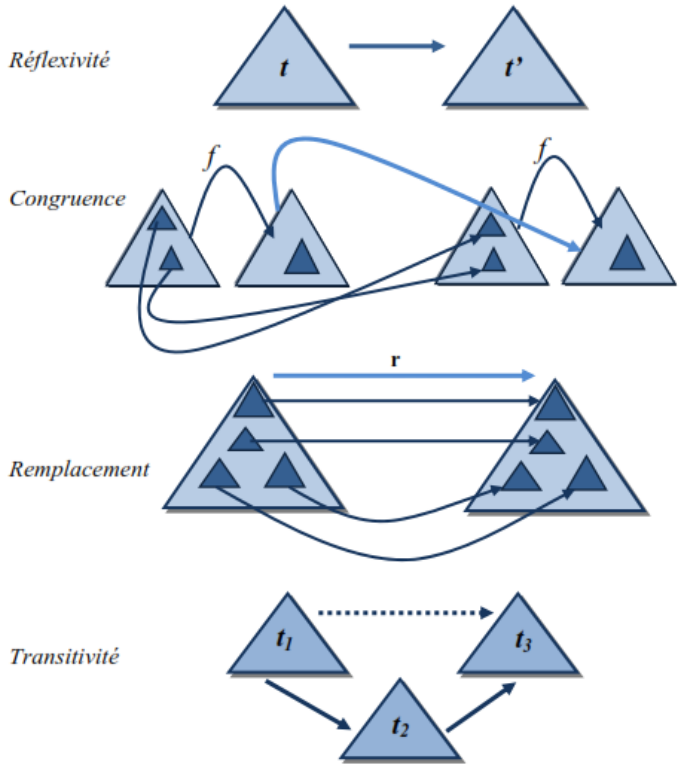


Figure 3.1. Représentation graphique des règles de déduction.

La *transitivité* détermine la composition séquentielle des étapes de réécriture. Cette règle offre la possibilité de construire la séquence de déduction à partir du terme initial jusqu'au terme final (à partir du quel plus aucune réécriture n'est possible).

La *congruence* spécifie que les réécritures peuvent être emboitées dans des contextes plus larges. Autrement dit, la composition parallèle d'un ensemble de transformations locales produit un état composite et cohérent et qui sera le nouveau terme global.

La règle de déduction qui semble la plus complexe est le *remplacement* qui identifie les termes à remplacer dans une expression d'un terme global représentant l'état actuel du système temps réel en faisant les substitutions de variables appropriées.

Une théorie de réécriture est une description statique d'un système temps réel. Sa sémantique est définie par un modèle mathématique qui décrit son comportement. Le modèle pour une théorie de réécriture étiquetée $\mathfrak{R} = (\sum, E, L, R)$, est une catégorie $\tau_{\mathfrak{R}}(X)$ dont les objets (états) sont des classes d'équivalence des termes $[t] \in T_{\sum,E}(X)$ et dont les morphismes (transitions) sont des classes d'équivalence des termes-preuves (proof-terms) représentant des preuves dans la déduction de réécriture [108].

Le calcul dans un système concurrent est une séquence de transitions (règles de réécriture) exécutées à partir d'un état initial donné. Il correspond à une preuve ou une déduction dans la logique de réécriture. Cette déduction est intrinsèquement concurrente et permet de raisonner correctement sur l'évolution du système d'un état à un autre [110] [111].

3.2.1 Théorie de réécriture temps réel

Les théories de réécriture généralisées dites « *temps réel* » ont été introduites pour spécifier les comportements des systèmes hybrides et des systèmes temps-réel dans la logique de réécriture.

Dans ce type de théories de réécriture, certaines règles appelées « *tick rules* » expriment l'écoulement du temps ou la durée dans un système tandis que des règles de réécriture ordinaires expriment les changements d'états du système qui ont lieu d'une manière instantanée. De plus, la théorie équationnelle d'appartenance sous-jacente à une théorie de réécriture temps-réel devrait contenir une axiomatisation (une interprétation abstraite) du domaine du temps et un opérateur {_} qui inclut l'état global du système. Cet opérateur est défini afin d'assurer que le temps s'écoule uniformément dans toutes les parties de ce système selon l'exécution de la règle « *tick rules* ». Une règle de réécriture « *tick rules* », qui modélise l'écoulement du temps, a la forme générale [109] :

$$l : \{t\} \xrightarrow{\tau_l} \{t'\} \text{ if Cond} \qquad (3.5)$$

où l est l'étiquette de la règle et le terme τ_l, de la sorte *Time*, exprime la durée de l'étape de réécriture.

Définition 3.1. (Théorie de réécriture temps réel) Une théorie de réécriture temps réel $\mathcal{R}_{\phi,\tau}$ est un tuple $\mathcal{R}_{\phi,\tau} = (\Re, \phi, \tau)$ où $\Re = (\Sigma, E, \phi, L, R)$ est une théorie de réécriture généralisée tel que :

- ϕ est un morphisme de théorie équationnelle $\phi : TIME \rightarrow (\Sigma, E)$ où *TIME* est une théorie équationnelle [112] dans laquelle sont spécifiés, à l'aide d'équations, les besoins généraux du modèle du temps. La théorie équationnelle *TIME* défini le temps d'une manière abstraite comme un monoïde commutatif ordonné *(Time, 0, +, <)* avec des opérateurs additionnels tels que l'opérateur de soustraction $-$ (où $x - y$ dénote $x - y$ si $y < x$ et 0 si $y \geq x$) et l'opérateur de comparaison $\leq$;

- (Σ, E) contient une sorte désignée, appelée *System* (qui exprime *l'état du système)* et une sorte spécifique *GlobalSystem* n'admettant aucune sous-sorte ni aucune super-sorte et définie uniquement sur l'opérateur suivant :
  $$\{\_\} : System \rightarrow GlobalSystem \qquad (3.6)$$
  De plus, pour tout $f : s_1 \ldots s_n \rightarrow s$ *de* Σ, la sorte *GlobalSystem* n'apparaît pas parmi $s_1 \ldots s_n$,

- τ est une attribution d'un terme $\tau_l(x_1,\ldots,x_n)$ *de la sorte* ϕ*(Time)* à chaque règle de réécriture de la forme :
  $$l : \{u(x_1,\ldots,x_n)\} \rightarrow \{u'(x_1,\ldots,x_n)\} \text{ if } Cond(x_1,\ldots,x_n) \qquad (3.7)$$
  où u et u' sont des termes de sorte *GlobalSystem*
  Ainsi, une règle l de sorte *GlobalSystem* et de durée τ_l est notée :
  $$l : \{u(x_1,\ldots,x_n)\} \xrightarrow{\tau l(x1,\ldots,xn)} \{u'(x_1,\ldots,x_n)\} \text{ if } Cond(x_1,\ldots,x_n) \qquad (3.8)$$

Etant donnée une théorie de réécriture temps réel $\mathcal{R}$, un calcul est une séquence non extensible $t_0 \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$ (c'est-à-dire, une séquence pour laquelle t_n ne peut être réécrit) ou bien une séquence infinie $t_0 \rightarrow t_1 \rightarrow \ldots$ de réécritures $t_i \rightarrow t_{i+1}$, avec t_i et t_{i+1} des termes bornés, commençant avec un terme initial t_0 de sorte *GlobalSystem* .

L'état global du système doit être de la forme $\{u\}$ de telle sorte que les règles « ***tick*** » assurent que le temps avance uniformément dans toutes les parties du système dont l'état est représenté par le terme u.

3.2.2 Réflection et stratégie de réécriture

La logique de réécriture est réflective dans un sens mathématique précis, c'est-à-dire, il existe une théorie finie de réécriture U qui est universelle dans le sens où on peut représenter dans

cette théorie *U* (comme des termes) toute autre théorie de réécriture finie *R* (y compris la théorie *U* elle-même) [113]. Ainsi, il est possible de simuler dans la théorie *U* toute réécriture inférée (déduite) dans la théorie de réécriture *R*. Par conséquent, il existe une représentation finie $\bar{R}$ sous forme de termes de *U* de toute théorie de réécriture *R*, une représentation $\bar{t}$ et $\overline{t'}$ sous forme de termes de *U* de tous termes *t, t'* et *R*, et une représentation $\langle \bar{R} ; \bar{t} \rangle$ de tout couple (*R,t*) vérifiant $R : t \rightarrow t' \Leftrightarrow (U : \langle \bar{R} ; \bar{t} \rangle \rightarrow \langle \bar{R} ; \overline{t'} \rangle)$.
La réflection permet de guider le processus de déduction induit par une théorie de réécriture R au niveau objet par le biais de stratégies de réécriture dont la sémantique peut être définie à l'intérieur de la logique de réécriture par des théories de réécriture définies à un méta-niveau. Dans ce cas, de telles stratégies de réécriture constituent des procédures d'inférence particulières spécifiques à la théorie R. Ainsi, la propriété de réfection permet à la logique de réécriture de décrire fidèlement le comportement de certains systèmes dont la sémantique complète ne peut être spécifiée simplement par un ensemble de règles de réécriture mais nécessite aussi la spécification de procédures d'exécution particulières de ces règles (par exemple, en précisant un ordre d'exécution spécifique).

3.2.3 Choix de Maude

Aux premiers abords, l'environnement Maude est relativement nouveau. La version 2.0 de l'environnement ayant été rendue publique en 2005, et sachant que l'environnement a connu quelques autres évolutions (les versions 2.1 et 2.2 respectivement), ceci suggère que cet outil est encore en voie de développement.
De plus, comme l'environnement est relativement nouveau, un nombre restreint d'auteurs s'y sont intéressés jusqu'à maintenant. De plus, l'environnement Maude, basé sur la logique de réécriture, a été spécialement développé pour supporter la programmation et la spécification de systèmes temps réel. Il s'avérait également tout aussi intéressant d'explorer les possibilités de Maude par rapport à cet aspect.
Maude est un candidat idéal pour la spécification de système. Comparativement à des langages tels *Object-Z* ou *B,* Maude aura le net avantage de permettre la définition de ses propres notations (Maude est un méta langage). Ceci rend Maude beaucoup plus expressif et beaucoup plus aisé à comprendre comparativement aux notations abstraites et mathématiques d'*Object-Z* et autres.
Encore une fois, Maude offre la possibilité de définir des opérateurs propres à l'usager, rendant les programmes beaucoup plus lisibles et compréhensibles. Maude a aussi l'avantage d'être un langage de programmation proprement dit, comparativement aux langages de spécification formelle et de vérification. Ces derniers sont très souvent spécifiques de ce type de technique, obligeant ainsi l'usager à apprendre un langage de spécification ou de vérification avant de pouvoir bénéficier des avantages de chacune de ces techniques. Il sera également obligé de connaître un langage de programmation pour pouvoir ensuite développer son système. Les notations Maude comblent donc ces deux aspects dans un seul et unique langage. Une notation Maude servira autant de spécification formelle (et à la vérification de modèles par le fait même), que de langage de programmation permettant le développement du produit. Un usager n'aura alors qu'à apprendre un seul langage, Maude, et pourra programmer son application tout en bénéficiant d'outils de vérification avancés directement dans son environnement de travail.

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 [114] [115], ELAN [116] et CafeObj [117] 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. Nous présentons, dans le tableau 3.1, une étude comparative de quelques Model-

Checkers assez connus dans la littérature tels que SPIN[4] [118], NuSMV[5] [119], Maude[6] et UPPAAL[7] [69] afin de choisir mieux qui s'adapte le mieux à la vérification de propriétés des systèmes temps réel.

Tableau 3.1. Résumé des caractéristiques de différents model-checkers [120].

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

Notre objectif est de choisir un Model-Checker puissant, de l'adopter à une logique bien expressive, pour spécifier et vérifier des propriétés de comportement des systèmes temps réel. Notre choix s'est orienté vers le langage Maude car celui-ci remplit ces exigences. Maude est un langage de spécification et de programmation fondée sur la logique de réécriture qui intègre une panoplie d'outils de validation et de vérification dont un modèle checker. Maude représente un très bon cadre logique et sémantique dans lequel différentes logiques et différents formalismes peuvent être exprimés et exécutés. De plus, il est particulièrement bien adapté pour la spécification de systèmes concurrents orientés objet et permet de décrire l'état d'un système et son comportement de façon mathématique et rigoureuse. Maude offre une modélisation plus fidèle des systèmes temps réel dont le comportement dépend du temps quantitatif ainsi que la possibilité de vérifier des propriétés temporisées.

3.3 Maude

Maude [121] [122] [123] [124] [125] [126] [127] est un environnement de programmation basé sur la logique de réécriture (« *rewriting logic* » en anglais). Maude est destiné principalement à la spécification formelle des systèmes informatiques, et plus particulièrement les systèmes concurrents et répartis. Il permet de définir des types de données algébriquement et de spécifier la dynamique d'un système à travers des règles appelées des règles de réécriture.

Dans le système Maude, les aspects structurels et statiques d'un système donné sont décrits par des équations et des types de donnés, alors que les aspects comportementaux et dynamiques sont décrits par des règles de réécriture. Les spécifications Maude sont exécutables, et elles peuvent être soumises à la simulation et l'analyse formelle en utilisant l'interpréteur Maude et plusieurs techniques et outils de vérification et d'analyse formelle tels que LTL Maude et la technique de vérification par invariants.

[4] SPIN. http://www.spinroot.com/

[5] NuSMV. http://nusmv.fbk.eu/

[6] Maude. http://maude.cs.illinois.edu/

[7] UPPAAL. http://www.uppaal.org/

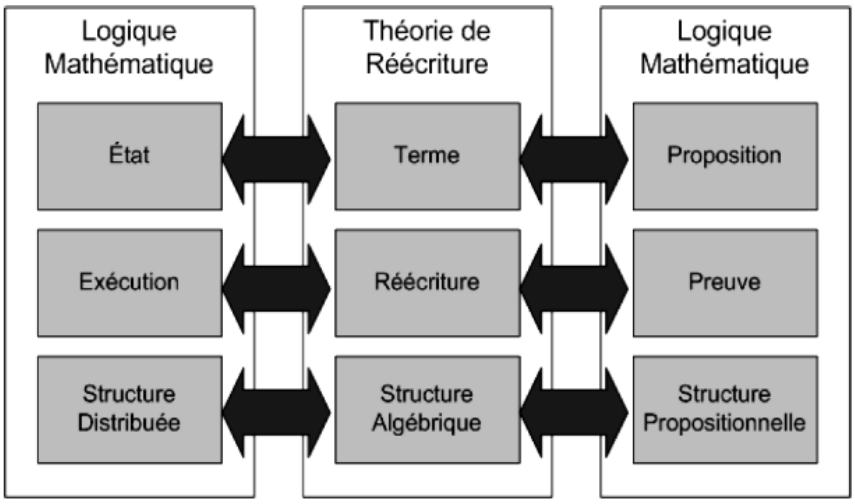


Figure 3.2. Parallèle entre un programme informatique, une théorie de réécriture et la logique mathématique [128].

Maude offre peu de constructions syntaxiques et une sémantique bien définie. Il est, par ailleurs, possible de décrire naturellement différents types d'applications. Maude est un langage qui supporte facilement le prototypage rapide et représente un langage de programmation avec des performances compétitives. Dans le langage Maude, deux niveaux de spécification sont définis. Un premier niveau concerne la spécification du système tandis que le second est lié à la spécification des propriétés [105] [129].
Un programme Maude doit être vu comme un énoncé d'une théorie de réécriture. Ainsi, l'exécution de ce programme doit être interprétée comme une déduction logique à partir des axiomes définis dans le programme. La figure 3.2 présente les équivalences qui existent entre l'exécution d'un programme informatique, une théorie de réécriture et la logique mathématique.

3.3.1 Caractéristiques du langage Maude

Le langage Maude, basé sur la logique de réécriture, où les programmes sont des théories, et les règles de déduction logique de réécriture correspond exactement au calcul concurrent, a été développé en fonction de trois grands objectifs : la simplicité, l'expressivité et la performance [130] :

 La simplicité : Maude est un langage déclaratif qui offre des notions de programmation simples et faciles à comprendre. En effet, les expressions de base de ce langage sont soit des équations soit des règles de réécriture.

 L'expressivité : Le langage Maude permet d'exprimer naturellement une vaste gamme d'applications, en commençant par les programmes déterministes arrivant aux programmes hautement concurrentiels. En outre, la sémantique rigoureuse de Maude permet non seulement d'exprimer des applications, mais aussi des formalismes entiers (d'autres langages, d'autres logiques). De ce fait, Maude est vu comme étant un « *métalangage* » avec lequel il est facile de développer un langage spécifique.

 La puissance et la performance : En plus d'être un langage de spécification formelle, Maude est un véritable langage de programmation compétitif. Il permet l'exécution de millions de règles de réécriture par seconde.

 Multi-paradigmes : Toutes les applications basées sur la logique équationnelle sont supportées facilement, donc la programmation fonctionnelle style équationnelle est

naturellement représentée par un sous-langage. Le style déclaratif de programmation concurrente, orientée-objets est aussi supporté avec une sémantique logique simple.

La plupart du temps, la simplicité et la performance sont présentes dans n'importe quel langage, ils vont de pair. Par contre, maximiser l'expressivité du langage est sans aucun doute l'un des avantages les plus marquants du langage Maude. En effet, il permet la combinaison de la programmation déclarative et orientée objet. La spécification par Maude est rapide et efficace pour l'exécution parallèle et distribuée, aussi bien que la transformation formelle des spécifications des programmes.

3.3.2 Les niveaux de programmation de Maude

Il est également important de noter que Maude permet de programmer à deux niveaux différents : *Core Maude* et *Full Maude*. On peut différencier les deux niveaux par ce qui suit :

 Core Maude : Core Maude représente le niveau de base de Maude, il est programmé directement en C++. Il implémente toutes les fonctionnalités de base du logiciel, les modules fonctionnels et les modules systèmes.
- Les modules fonctionnels sont principalement utilisés pour la définition de types de données et d'opérateurs via la théorie des équations.
- Les modules système : Les modules systèmes, quant à eux, sont utilisés pour définir des théories de réécriture. D'une certaine façon, tout comme la logique de réécriture est plus générale que la logique des équations, un module système englobe un module fonctionnel.

Core Maude intègre d'autres modules tels que : les modules prédéfinis des types de données et le module *Model-Checker* ; la réflexivité et la fonction meta-langage sont aussi prises en compte.

 Full Maude : Full Maude représente le niveau supérieur, il est programmé en *Core Maude* développé par Francisco Durân, Full Maude est une extension du premier niveau (Core Maude), avec lequel il est possible de combiner des modules pour améliorer le développement**.** Full Maude offre de nombreux avantages. Il est surtout utilisé pour spécifier le paradigme de programmation orienté-objet et les modules orientés-objet Maude. [130] [131] [132].
- Les modules orientés objets : Ce dernier type de module est développé spécifiquement pour le paradigme de développement orienté-objet. Ce type de module est utilisé au niveau de Full Maude. Il permet de définir la syntaxe nécessaire pour déclarer des classes, des sous classes et des messages.

3.3.3 Les modules de Maude

Le développement Maude se fait par l'écriture de divers modules décrivant le système que l'on écrit. Ainsi, l'unité de base pour le développement Maude est le module.
Pour une bonne description modulaire, trois types de modules sont définis dans Maude. Les modules *fonctionnels* permettent de définir les types de données. Les modules *systèmes* permettent de définir le comportement dynamique d'un système. Enfin, les modules *orienté-objet* qui peuvent, en fait, être réduits à des modules systèmes offrent explicitement les avantages du paradigme objet.

3.3.3.1 Modules Fonctionnels

Les modules fonctionnels définissent les types de données et les opérations qui leur sont applicables en termes de la théorie des équations. L'algèbre initiale est le modèle mathématique (*dénotationel*) pour les données et les opérations. Les éléments de cette algèbre sont des classes

d'équivalence des termes sans variables (*ground terms*) modulo des équations. Si deux termes sans variables sont égaux par le biais des équations, alors ils appartiennent à la même classe d'équivalence. Les équations dans un module fonctionnel sont orientées. Elles sont utilisées de gauche à droite et le résultat final de la simplification d'un terme initial est unique quel que soit l'ordre dans lequel ces équations sont appliquées. En plus des équations, ce type de modules supporte les axiomes d'adhésions (*memberships axioms*). Ces axiomes précisent l'appartenance d'un terme à une sorte. Cette appartenance peut être sous certaines conditions ou non. Cette condition est une conjonction des équations et des tests d'adhésions inconditionnels [109].
Un module fonctionnel est introduit par les mots clés `fmod` <*Corps du module*> `endfm` où le corps du module spécifie une théorie $\mathfrak{R} = (\Sigma, E \cup A, \phi)$ dans la logique équationnelle d'appartenance. La signature Σ inclut des sortes (indiqués par le mot clé `sort`), des sous-sortes (spécifiés par le mot clé `subsort`) et des opérateurs (introduits avec le mot clé `op`). La syntaxe des opérateurs est définie par les utilisateurs en indiquant la position des arguments par le symbole (_). Certains de ces arguments peuvent être spécifiés comme figés en utilisant le mot clé `frozen`(*PositionArgument*). L'ensemble ***E*** désigne les équations et les tests d'appartenance (qui peuvent être conditionnels) et ***A*** est un ensemble d'axiomes équationnels introduits comme attributs de certains opérateurs dans la signature Σ tels que les axiomes d'associativité (spécifiée par le mot clé `assoc`), de commutativité (spécifiée par le mot clé `comm`) ou d'identité (spécifiée par le mot clé `id` :), le mot clé `ctor` pour le constructeur et `prec` pour spécifier le degré de précédence. Ces derniers sont définis de manière à ce que les déductions équationnelles se fassent modulo les axiomes de *A*. Les équations sont spécifiées par le mot clé `eq` ou le mot clé `ceq` (pour les équations conditionnelles) et les tests d'adhésion ou d'appartenance sont introduits avec les mots clés `mb` ou `cmb` (pour les tests d'appartenance conditionnels). Une équation non conditionnelle suit la syntaxe suivante :

`eq` <*Terme*-1>=<*Terme*-2>[<*Attributs*>] .

Les deux termes *Terme*-1 et *Terme*-2 dans l'équation *Terme*-1 = *Terme*-2 doivent avoir la même sorte. Pour qu'une équation soit exécutable, toutes les variables dans *Terme*-1 (la partie droite de l'équation) doivent apparaître dans *Terme*-2 (la partie gauche). Les équations conditionnelles prennent la forme suivante :

`ceq` <*Terme*-1>=<*Terme*-2> `if` <*EqCondition*-1>...<*EqCondition*-k> [<*Attributs*>] .

Une condition liée à une équation ou à un test d'appartenance peut être formée par une conjonction d'équations et de tests d'adhésions inconditionnels.
Dans un module fonctionnel, les équations sont utilisées comme des règles de simplification par lesquelles chaque expression, après substitution des variables, peut être évaluée et simplifiée à sa forme réduite dite *forme canonique*. Les variables peuvent être déclarées dans les modules avec les mots clés `var` ou `vars`, ou introduites directement dans les équations et les tests d'adhésion, sous la forme d'une expression `var : sort`. La syntaxe des conditions d'une équation conditionnelle est de trois variantes :

- Équations ordinaires de la forme *t = t'*,
- Équations de correspondance (matching) *t : = t'*,
- Équations booléennes de la forme t, où t est un terme algébrique de genre [Bool], équivalent à *t = true*.

Nous donnons dans ce qui suit un exemple classique d'un type abstrait de données, celui des naturels :

```
fmod NAT is
  sort Nat . sort PositiveNat .
  subsort PositiveNat < Nat .
  op 0 : -> Nat .
```

```
  op S_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .   *** N + M = M + N
  vars N M : Nat .
  var P : PositiveNat .
  cmb P : PositiveNat if P =/= 0 .  *** axiome d'adhésion indiquant que P est de type PositiveNat
                                    *** si P est différent de 0
  eq N + 0 = N .          *** N + 0 et N dénotent le même terme (même classe d'équivalence) qui est [N]
  eq (S N) + (S M) = S S (N + M) .
endfm
```

3.3.3.2 Modules Systèmes

Les modules systèmes permettent de définir le comportement dynamique d'un système. Ce type de module augmente les modules fonctionnels par l'introduction de règles de réécriture. Un degré maximal de concurrence est offert par ce type de module. Un module système décrit une "*théorie de réécriture*" qui inclut des sortes, des opérations et trois types d'instructions : équations, adhésions et règles de réécriture. Ces trois types d'instructions peuvent être conditionnels. Une règle de réécriture spécifie une "*transition concurrente locale*" qui peut se dérouler dans un système. L'exécution de telle transition spécifiée par la règle peut avoir lieu quand la partie gauche d'une règle correspond à (*match*) une portion de l'état global du système et bien sûr la condition de la règle est valide. Un module système Maude est déclaré selon la syntaxe :

`mod` <*nom-module*> `is` <*déclarations-et-expressions*> `endm .`

Un module système dans Maude est déclaré avec le mot-clé `mod`, suivi de nom du module. Les déclarations et les expressions dans un module fonctionnel peuvent être : des importations d'autres modules fonctionnels ou systèmes, des déclarations des sortes ou des sous-sortes, déclarations des opérations, déclarations des équations et déclarations des règles de réécriture conditionnelles. Les règles de réécriture conditionnelles et non conditionnelles déclarées au sein d'un module système prennent la syntaxe suivante :

`rl` [<*Étiquette*>] : <*Terme*-1> => <*Terme*-2> [<*Attributs*>] .
`crl` [<*Étiquette*>] : <*Terme*-1> => <*Terme*-2> `if` <*Condition*-1>... <*Condition*-k>[<*Attributs* >] .

Les deux termes <*Terme*-1> et <*Terme*-2> sont des termes de même sorte qui peuvent contenir des variables. <*Étiquette*> est l'étiquette de la règle de réécriture ; elle peut être omise. Les conditions d'une règle de réécriture conditionnelle <*Condition*-k> peuvent contenir des expressions de réécriture qui testent la possibilité de réécrire des termes algébriques. Un exemple simple montrant la puissance de ce type de module est donné dans ce qui suit:

```
mod NAT is
  extending NAT .
  op _? _ : Nat Nat -> Nat .
  vars N M : Nat .
  rl [R1] : N ? M => N .
  rl [R2] : N ? M => M .
endm
```

L'opérateur ? modélise le choix aléatoire entre deux entiers. Si nous avions utilisé uniquement la logique équationnelle pour décrire ce système, alors nous aurions les équations suivantes : N ? M = N et N ? M = M. Ce qui signifie que N = M, c'est-à-dire que tous les entiers sont égaux, alors le modèle dénotationnel (*l'algèbre initiale*) de ce module s'effondre à un seul élément ! Le module NAT n'est pas protégé. La logique équationnelle est faible pour décrire ce système. La

logique équationnelle est réversible, alors que l'évolution des systèmes concurrents ne l'est pas forcément. Les règles R1 et R2 sont irréversibles et elles sont concurrentes. Dans ce cas, la sémantique de l'opérateur ? est bien définie.

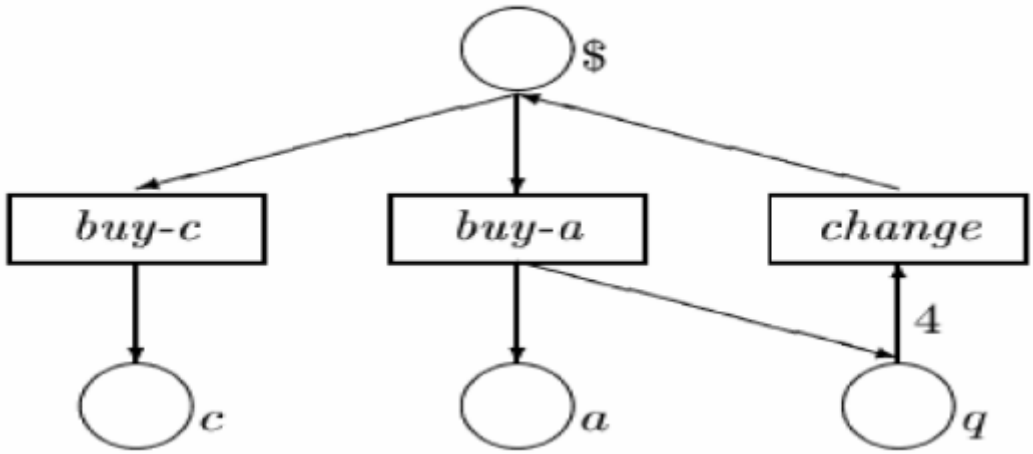


Figure 3.3. Un réseau de Petri représente le système VENDING-MACHINE.

La figure 3.3 illustre un autre exemple, représenté par un réseau de Petri, d'un système nommé VENDING-MACHINE extrait de [133]. Ce système consiste en un automate représentant une machine distributrice très simple. Les entrées sont les suivantes :
- $ représente l'entrée d'une pièce de 1$ dans la machine ;
- c représente un bonbon ;
- a représente une pomme ;
- q représente une pièce de 25 cents.

Le système permet donc l'achat d'un bonbon à 1 $ et l'achat d'une pomme pour 0,75 $. La machine fait aussi la monnaie : pour quatre pièces de 0, 25 $, elle retourne une pièce de 1$. Un tel système peut être décrit par le module système suivant :

```
mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var M : Marking .
  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change] : q q q q => $ .
endm
```

3.3.3.3 Modules orientés objet

Dans le langage Maude, les systèmes basés sur le paradigme orienté objet peuvent être définis à travers des modules orientés objet. Les modules orientés objet sont supportés par le système FullMaude [132]. Les modules orientés objet sont déclarés avec la syntaxe suivante :

`omod` < *nom-module* > `is` < *déclarations et expressions* > `endom .`

Les modules orientés objet offrent une syntaxe appropriée qui permet de déclarer les concepts de base du paradigme orienté objet :

- `Oid` pour identifier les objets,
- `Cid` pour identifier les classes,
- `Objects` pour définir les objets et
- `Msg` pour déclarer les messages.

Ils supportent la spécification et la manipulation des *objets*, des *messages*, des *classes* et de l'*héritage*. Un système orienté objet concurrent dans ce cas est modélisé par un multi-ensemble d'objets et de messages juxtaposés, où les interactions concurrentes entre les objets sont régies par des règles de réécriture.
Dans les modules orientés objet, un objet est représenté par un terme :

`< O : C | a1 : v1 ,..., an : vn >`

Où `O` est l'identificateur de l'objet, `C` est l'identificateur de sa classe, les `ai` sont les noms des attributs de l'objet et enfin les `vi` correspondent aux valeurs des attributs. Les messages dans les modules orientés objet n'ont pas de forme syntaxique prédéfinie, leur structure est définie par l'utilisateur. La seule contrainte est que le premier argument doit être l'identifiant de l'objet destination. La déclaration des classes suit la syntaxe :

`class C | a1 : s1 , … , an : sn .`

où `C` est le nom de la classe et s_i est la sorte de l'attribut `ai`. Il est aussi possible de déclarer des sous classes et bénéficier ainsi de la notion d'héritage. Les messages sont déclarés en utilisant le mot clé `msg`. La forme générale d'une règle de réécriture dans la syntaxe orientée objet de Maude est :

$$\texttt{crl [r]} : M_1 \ldots M_n < O_1 : F_1 \mid a_1 > \ldots < O_m : F_m \mid a_m > \Rightarrow < O_{i1} : F'_{i1} \mid a'_{i1} > \ldots < O_{ik} : F'_{ik} \mid a'_{ik} > M_1' \ldots M_p' \texttt{ if } Cond .$$

Où r est l'étiquette de la règle, M_s, $s \in 1..n$, et M'_u, $u \in 1..p$ sont des messages, O_i, $i \in 1..m$, et O_{il}, $l \in 1..k$, sont des objets, et *Cond* est la condition de la règle. Si la règle est non conditionnelle, nous remplaçons le mot clé *crl* par *rl* et nous enlevons la clause *if Cond*.

Notons que le système Full-Maude offre un support additionnel pour la programmation orientée objet avec les notions de classes, sous-classes et une syntaxe plus conviviale des règles de réécriture. Ainsi, il permet au système Maude de supporter la modélisation orientés objet en fournissant le module prédéfini *CONFIGURATION*. Dans ce module, les sortes représentant les concepts essentiels des objets, classes, messages et configurations, sont déclarés.

3.3.3.4 Modules prédéfinis

Les modules prédéfinis de Maude sont stockés dans une librairie spécifique et peuvent être importés par d'autres modules définis par l'utilisateur. Ils sont introduits dans les fichiers sources de Maude `prelude.maude` et `model-checker.maude` : comme par exemple les BOOL, STRING et NAT. Ces modules déclarent les sortes et les opérations pour manipuler, respectivement, les valeurs *booléennes*, les *chaînes de caractères* et les *nombres naturels*. Le fichier `model-checker.maude` contient les modules prédéfinis interprétant les outils nécessaires pour l'utilisation du *LTL Model Checker* de Maude.

3.4 Real Time-Maude (RT-Maude)

Real-Time Maude [5] est un outil et un langage de spécification, qui étend Maude dans le but de supporter la spécification formelle et l'analyse de systèmes temps réel et hybrides. Le formalisme de spécification est basé sur les théories de réécriture et il est destiné à spécifier les systèmes temps-réel distribués suivant le paradigme orienté-objet.
Les spécifications Real-time Maude peuvent être analysées par simulation ou par model checking. Real-Time Maude permet de :

- Simuler l'exécution du système par réécriture temporelle. Ainsi, un sous-ensemble des comportements possibles est exploré.
- vérifier le système par model checking borné dans le temps de propriétés en Logique Temporelle Linéaire [134] sur le modèle du système. Cela consiste à envisager tous les

comportements possibles jusqu'à une certaine durée. L'espace des états est donc réduit ce qui facilite le model checking, mais il n'y a aucune garantie sur la vérification des propriétés au-delà.

Une théorie de réécriture RT-Maude est une théorie de réécriture Maude, qui contient en plus la spécification [135] :
▪ D'une sorte ***Time*** pour décrire le domaine du temps qui peut être discret ou continue,
▪ d'une sorte ***GlobalSystem*** avec un ***constructeur*** **« {_} » :**

{_} : System -> GlobalSystem

• Et un ensemble de règles ***tick*** qui modélisent le temps écoulé dans le système qui ont la forme suivante :

{t} => {t'} in time u if cond

Où ***u*** est un terme, qui peut contenir des variables, de sorte ***Time*** qui dénote la durée de la règle, et les termes ***t*** et ***t'*** sont des termes de sorte ***System*** qui dénote l'état du système. Les règles de réécriture qui ne sont pas des règles ***tick*** sont des règles ***instantanées*** supposées prendre un temps zéro.
L'état initial doit toujours avoir la forme ***{t''}***, où ***t''*** est un terme de **sort *System***, afin que la forme des règles ***tick*** assure que le temps s'écoule uniformément dans toutes les parties du système.
Les théories de réécriture temps réel sont spécifiées en RT-Maude comme des modules temporisés ''***tmod***'' (timed modules) ou des modules temporisés orientés-objet ''***tomod*** '' (timed object-oriented modules).
Real-Time Maude est implémenté en Maude comme une extension de Full Maude, donc tous les commandes et les modules doivent être déclarés entre parenthèses :

`(tmod` *Name* `is … endtm)`

`(tomod` *Name* `is … endtom)`

Dans ce qui suit nous introduisons d'abord les ***modules temporisés (Timed modules)***. Nous présenterons ensuite quelques exemples spécifiques à chacun des types de modules (temporisés et temporisés orientés-objet), plus de détails pourront être trouvés dans [136].

3.4.1 Les Modules Temporisés (Timed modules)

Chaque module temporisé importe automatiquement le module fonctionnel *TIMED-PRELUDE* qui inclue les déclarations cruciales :

Sorts System GlobalSystem .
op {_} : System -> GlobalSystem .

L'état d'un système doit être représenté par un terme de sorte ***System.*** Un module temporisé importe aussi le squelette du domaine de temps qui est décrit ci-dessous.

3.4.1.1 Les Domaines de Temps (Time Domains)

Chaque module temporisé importe automatiquement le module *TIME* suivant :

```
fmod TIME is
  sorts Time NzTime .
  subsort NzTime < Time .
  op zero : -> Time .
  op   plus   : Time Time -> Time [assoc comm] .
  op   monus   : Time Time -> Time .
  ops  le    lt    ge    gt   : Time Time -> Bool.
  eq zero plus R:Time = R:Time .
  eq R:Time le R':Time = (R:Time lt R':Time) or (R:Time = = R':Time) .
  eq R:Time ge R':Time = R':Time le R:Time .
  eq R:Time gt R':Time = R':Time lt R:Time .
```

endfm

Ce module définie un squelette du domaine de temps, avec la valeur ***zéro*** et quelques symboles de fonction. En effet l'utilisateur a la liberté complète pour spécifier le type de données des valeurs de temps qui peuvent être discret ou bien continue par :

- Importer un type de donné spécifique de valeurs temporelles qui satisfont la théorie du Temps, avec une sorte, dit *TimeValues* pour telles valeurs, et
- Déclarer une inclusion *subsort TimeValues* < *Time* et donner les équations appropriées pour interpréter le constant zéro et les opérateurs *_plus_, _monus_,* et *_ lt_* dans *TimeValues.*

RT-Maude contient le module suivant qui définit le domaine de temps comme étant des nombres entiers :

```
fmod NAT-TIME-DOMAIN is
  including LTIME .
  protecting NAT .
  subsort Nat < Time .
  subsort NzNat < NzTime .
  vars N N' : Nat .
  eq zero = 0 .
  eq N plus N' = N + N' .
  eq N monus N' = if N > N' then sd(N, N') else 0 fi .
  eq N lt N' = N < N' .
endfm
```

Pour spécifier les domaines de temps continue, RT-Maude définit un sous ensemble (subsort) *NNegRat* de nombres rationnels non négatifs, et, dans le module *POSRAT-TIME-DOMAIN*, il définit le domaine de temps comme étant des nombres rationnels.

```
fmod POSITIVE-RAT is
  protecting RAT .
  sort NNegRat .
  subsorts Zero PosRat Nat < NNegRat < Rat .
endfm

fmod POSRAT-TIME-DOMAIN is
  including LTIME .
  protecting POSITIVE-RAT .
  subsort NNegRat < Time .
  subsort PosRat < NzTime .
  vars R R' : NNegRat .
  eq zero = 0 .
  eq R plus R' = R + R' .
  eq R monus R' = if R > R' then R - R' else 0 fi .
  eq R lt R' = R < R' .
endfm
```

Pour résumer et pour spécifier le domaine de temps, l'utilisateur peut :

- Soit laisser le domaine de temps non spécifié pour les instanciations plus tardifs, ou
- Importé soit de l'un des modules *NAT-TIME-DOMAIN* ou *POSRAT-TIME-DOMAIN*, ou
- définir explicitement son propre module temporisé.

3.4.1.2 Les règles Tick

La syntaxe des règles ***tick*** inconditionnelles est :

rl [l] : {t} => {t'} in time u .

Une règle ***tick*** conditionnelle est écrite avec la syntaxe :

crl [l] : {t} => {t'} in time u if cond .

Il faut noter que chaque module temporisé import automatiquement les sortes ***System*** et ***GlobalSystem*** et l'opérateur **{_}** aussi bien qu'un domaine de temps réduit, nous sommes maintenant prêt à en spécifier quelques exemples.

Il est nécessaire de déterminer la stratégie d'avancement du temps pour guider l'application des règles « *tick* ». Le choix d'une telle stratégie se fait par la commande de RT-Maude suivante :

`(set tick def r .)`

Où *r* est un terme de sorte *Time*, indiquant le pas d'avancement dans le temps défini par l'utilisateur et tenté par RT-Maude à chaque application d'une règle « *tick* ».

3.4.2 Exemples de Modules Temporisés : Modélisation d'une horloge

Dans une série d'exemples nous donnons la spécification d'une horloge simple qui montre le temps. Le terme *clock (t*) dénote une horloge qui montre le temps. Le module suivant spécifie une horloge discrète où le temps avance toujours par une unité de temps dans chaque pas *''tick''*. Puisque le temps est discret nous utilisons les nombres naturels comme un domaine de temps et donc nous importons le module *NAT-TIME-DOMAIN* :

```
(tmod DISCRETE-CLOCK-1 is
  protecting NAT-TIME-DOMAIN .
  op clock : Time -> System [ctor] .
  var N : Time .
  rl [tick] : {clock(N)} => {clock(N + 1)} in time 1 .
endtm)
```

Le système avance le temps par 1 dans chaque pas, avec le résultat que la valeur de l'horloge augmente par 1.

Supposons maintenant des horloges qui comptent les heures, donc quand l'horloge atteint 24 elle devrait montrer 0. Pour obtenir une spécification légèrement plus intéressante (et légèrement différente), La spécification suivante augmente juste la valeur de l'horloge modulo 24 avec une opération *reset* qui réinitialise une horloge avec la valeur 24 à 0 :

```
(tmod DISCRETE-CLOCK-24 is
  protecting NAT-TIME-DOMAIN .
  op clock : Time -> System [ctor] .
  var N : Time .
  crl [tick] : {clock(N)} => {clock(N + 1)} in time 1 if N < 24 .
  rl [reset] : clock(24) => clock(0) .
endtm)
```

L'état *{clock(24)}* passe à l'état *{clock(0)}* en utilisant une transition instantanée. La condition dans la règle *tick* assure que le temps ne peut pas s'écouler quand l'horloge en montre 24, donc *{clock(24)}* doit être réinitialisé à *{clock(0)}* avant que le temps continuer à avancer.

Bien que le temps discret soit approprié pour modéliser les systèmes temps réel tel que la planification des algorithmes temps réel et des protocoles de communication, les systèmes hybrides sont souvent modélisés en utilisant le domaine de temps continu. La spécification

suivante modélise l'horloge décrit précédemment en utilisant le domaine de temps continu, nommé les nombres rationnels non négatifs :

```
(tmod DENSE-CLOCK-1 is
  protecting POSRAT-TIME-DOMAIN .
  ops clock stopped-clock : Time -> System [ctor] .
  vars R R' : Time .
  crl [tick] :
  {clock(R)} => {clock(R + R')} in time R' if R' <= 24 monus R [nonexec] .
  rl [reset] : clock(24) => clock(0) .
  rl [batteryDies] : clock(R) => stopped-clock(R) .
  rl [tickWhenFlat] :
  {stopped-clock(R)} => {stopped-clock(R)} in time R' [nonexec] .
endtm)
```

Bien que l'exemple soit assez simple, les règles *tick* au-dessus sont typiques pour les systèmes avec le domaine de temps continu. A partir d'un état donné, le temps peut s'écouler par n'importe quelle valeur de temps jusqu' à ce que l'horloge atteigne la valeur 24. Cette augmentation non déterministe du temps est modélisée dans les règles *tick* par l'existence d'une nouvelle variable R' qui n'apparait pas dans le côté gauche de la règle, et qui n'est pas instancié dans la condition de la règle. Cette variable R' peut ensuite prendre n'importe quelle valeur entre 0 et 24-R. l'horloge peut s'arrêter à tout moment, à cause d'une batterie morte ou autres pannes, dans ce cas où l'état est *stopped-clock(r)*, où ***r*** était la valeur de l'horloge quand celle-ci est arrêtée.

3.5 Analyse formelle et vérification des propriétés avec Maude

Dans un module écrit en Maude, les règles de réécriture constituent les unités élémentaires d'exécution. Elles interprètent les actions locales du système modélisé, et peuvent être exécutées dans un temps constant et de manière concurrente (à n'importe quel moment).

Le langage Maude est entouré de nombreuses techniques et outils d'analyses formelles des comportements de systèmes tels que, la technique de vérification par invariants, le prouveur de théorèmes, le model-checker LTL, l'analyse de terminaison et l'analyse de cohérence. RT-Maude dispose d'un ensemble de commandes de simulation, d'analyse formelle et de vérification de propriétés LTL par *model checking* [137]. L'exécution et l'analyse d'un module temporisé se fait par rapport à la stratégie choisie pour l'application des règles « *tick* » de ce module. Notons que les techniques d'avancement du temps de RT-Maude procèdent à un échantillonnage de l'espace des états d'un système temporisé de telle sorte qu'au lieu de couvrir tout le domaine du temps, seulement certains moments sont visités.

Maude nous offre la possibilité de simuler l'exécution des réécritures (via des règles de réécriture) ou des réécritures équationnelles (via des équations) dans un module M par l'implémentation des deux commandes : `reduce` et `rewrite.`

La commande `reduce` (abrégée par `red`) permet à un terme initial d'être réduit par application des équations et des axiomes d'adhésion dans un module donné. Elle se présente sous la syntaxe suivante :

`Reduce {in` *module* `:}` *term* `.`

Pour simuler le comportement possible d'un système, RT-Maude dispose de deux modes de réécriture temporisée, implémentées respectivement par la commande de réécriture standard `timed rewrite` (ou `trew`) et la commande de réécriture équitable `timed fair`

`rewrite` (ou `tfrew`). Chacune de ces commandes simule un comportement possible du système, selon sa propre stratégie, jusqu'à une durée de temps donnée.

La technique de vérification par invariants est une technique simple, mais très utile, qui peut être menée dans Maude en utilisant la commande `search`. Elle représente une des techniques les plus courantes pour vérifier les propriétés de sûreté dans les différents types de systèmes informatiques. Néanmoins, elle peut être utilisée aussi pour la vérification des propriétés de vivacité.
Étant donné un système de transition *t* et un état initial s0, un invariant I est un prédicat définissant un sous-ensemble d'états qui comprend l'état s0 et tous les états accessibles à partir de s0 par un nombre fini de transitions. Par conséquent, un invariant est un prédicat qui définit un ensemble d'états contenant tous les états accessibles depuis s0. Dans ce cas, la sûreté est vérifiée si l'invariant est maintenu, ce qui signifie que rien de mauvais ne peut jamais se produire. Les propriétés de vivacité peuvent donc être vérifiées si on considère le cas contraire, c'est-à-dire, que les invariants sont considérés comme des états désirables (des états recherchés ou souhaités).
La technique de vérification par invariants est implémentée dans Maude par la commande `search`. Sa syntaxe est conforme à la forme générale suivante [138] :
`search [n, m] in` <*nom-module*>`:` <*Terme-1*> **<***flèche-de-recherche***>** <*Terme-2*> `such that` <*condition*> `.`

- `n` est un argument optionnel fournissant une limite sur le nombre de solutions souhaitées,
- `m` est un argument optionnel indiquant la profondeur maximale de la recherche,
- <*nom-module*> le module où la recherche aura lieu,
- <*Terme-1*> le terme initial,
- <*Terme-2*> le pattern qui doit être atteint,
- <*flèche-de-recherche*> est une flèche indiquant la forme de la preuve de réécriture de <*Terme-1*> au <*Terme-2*> :
    - `=>1` la preuve de réécriture comprenant exactement une seule étape,
    - `=>+` la preuve de réécriture consiste en une ou plusieurs étapes,
    - `=>*` la preuve consiste en aucune, une ou plusieurs étapes et
    - `=>!` indique que seuls les chemins à états déterministe sont exploité.
- <*Condition*> est une propriété optionnelle qui doit être satisfaite par l'état atteint.

Pour l'analyse d'accessibilité, RT-Maude offre également une commande de recherche temporisée (`tsearch`) pour vérifier si un état ***t*** dans un système est accessible à partir d'un état initial et dans une limite de temps donnée. Cette commande adopte par défaut la stratégie de recherche en largeur dans l'arbre de calcul (arbre d'accessibilité).

Tableau 3.2. Opérateurs de LTL et notation Maude.

Opérateur LTL	Notation Maude	Opérateur LTL	Notation Maude
$\top$	`True`	$\rho \rightarrow \varphi$	`_->_`
$\bot$	`False`	$\rho \leftrightarrow \varphi$	`_<->_`
$\neg\rho$	`~_`	$\Diamond\rho$	`<>_`
$\rho \wedge \varphi$	`_/\_`	$\Box\rho$	`[]_`
$\rho \vee \varphi$	`_\/_`	$\rho \mathcal{W} \varphi$	`_W_`
$O\rho$	`o_`	$\rho \rightsquigarrow \varphi$	`_|->_`
$\rho \mathcal{U} \varphi$	`_U_`	$\rho \Rightarrow \varphi$	`_=>_`
$\rho \mathcal{R} \varphi$	`_R_`	$\rho \Leftrightarrow \varphi$	`_<=>_`

La deuxième technique de vérification, que nous présentons, est basée sur la logique temporelle linéaire (LTL). Dans l'environnement Maude, deux éléments sont nécessaires à l'utilisation du vérificateur de modèles. Pour un module système Maude donné, ces deux éléments sont :

1. Une spécification système donnée par la théorie de réécriture décrite au sein de ce module,
2. Une spécification de besoins donnée par une propriété ou une série de propriétés à propos du système en modélisation.

Le Model Checking supporté par la plate-forme de Maude utilise la logique LTL essentiellement pour sa simplicité et les procédures de décision bien définies qu'il offre (pour plus de détails, voir [105] ou [129]). Dans un module prédéfini LTL, on trouve la définition des opérateurs pour la construction d'une formule (propriété) dans la logique temporelle linéaire. Une partie des opérateurs LTL est définie dans la syntaxe de Maude par le module fonctionnel LTL suivant :

```
fmod LTL is
…
*** opérateurs définis de LTL
 op _->_ : Formula Formula -> Formula .      *** implication
 op _<->_ : Formula Formula -> Formula .     *** equivalence
 op <>_ : Formula -> Formula .               *** eventualy
 op []_ : Formula -> Formula .               *** always
 op _W_ : Formula Formula -> Formula .       *** unless
 op _|->_ : Formula Formula -> Formula .     *** leads-to
 op _=>_ : Formula Formula -> Formula .      *** strong implication
 op _<=>_ : Formula Formula -> Formula .     *** strong equivalence
…
endfm
```

Les opérateurs LTL sont représentés dans Maude en utilisant une forme syntaxique semblable à leur forme d'origine. Par exemple, l'opération [] est défini dans Maude par l'opérateur (*always*). Cet opérateur s'applique sur une formule pour donner une nouvelle formule. Nous avons, par ailleurs, besoin d'un opérateur indiquant si une formule donnée est vraie ou fausse dans un certain état. Nous trouvons un tel opérateur (|=) dans un module prédéfini appelé SATISFACTION :

```
fmod SATISFACTION is
 protecting LTL .
 sort State .
 op _|=_ : State Formula ~> Bool .
endfm
```

L'état *State* est générique. Après avoir spécifier le comportement de son système dans un module système ou module rt-maude de Maude, l'utilisateur peut spécifier plusieurs prédicats exprimant certaines propriétés liées au système. Ces prédicats sont décrits dans un nouveau module qui importe deux modules : celui qui décrit l'aspect dynamique du système et le module SATISFACTION. Soit, par exemple, M-PREDS le nom du module décrivant les prédicats sur les états du système :

```
mod M-PREDS is
  protecting M .
  including SATISFACTION .
  subsort Configuration < State .
  …
endm
```

M est le nom du module décrivant le comportement du système. L'utilisateur doit préciser que l'état choisi (configuration choisie dans cet exemple) pour son propre système est sous type de type *State*. A la fin, nous trouvons le module MODEL-CHECKER qui offre la fonction model-Check, ou bien le module prédéfini TIMED-MODEL-CHECKER pour les modules temporisés. L'utilisateur peut appeler cette fonction en précisant un état initial donné et une formule. Le Model Checker de Maude vérifie si cette formule est valide (selon la nature de la formule et la procédure du Model Checker adoptée par le système Maude) dans cet état ou l'ensemble de tous les états accessibles depuis l'état initial. Si la formule n'est pas valide, un contre-exemple (*counter-example*) est affiché. Le contre-exemple concerne l'état dans lequel la formule n'est pas valide :

```
fmod MODEL-CHECKER is
including SATISFACTION .
…
op counterexample : TransitionList TransitionList -> ModelCheckResult [ctor] .
op modelCheck : State Formula ~> ModelCheckResult .
…
endfm
```

Pour les modules temporisés, les formules en logique temporelle linéaire (LTL) à vérifier doivent être définies dans un module, spécifié en termes de la logique de réécriture, qui importe le module spécifiant le système à analyser ainsi que le module prédéfini TIMED-MODEL-CHECKER.

Le Modèle-Checker intégré de Maude vérifie automatiquement la validité de la formule LTL en parcourant tous les états accessibles depuis l'état initial spécifié. Si la formule n'est pas valide, un contre-exemple sera généré. Les propositions (qui doivent être paramétrées) doivent être déclarées de sorte `Prop`, et leur sémantique doit être exprimée par des équations de la forme :

```
{état} |=  prop  =  b
```

Avec `b` un terme booléen, ce qui signifie que la proposition `Prop` tient dans tous les états `{t}` tel que `{t} |= prop` est évaluée à vraie.

Une formule de la logique temporelle est construite à partir de formules atomiques (temporisées et non temporisées), des constantes *True* et *False* et d'opérateurs de la logique temporelle (hormis l'opérateur Next) tels que la négation (¬), la conjonction (/\), la disjonction (\/), until (**U**), l'opérateur *Fatalement* (<>), l'opérateur *Globalement* ([]), …etc. Le tableau 3.2 présente chacun des opérateurs LTL vus dans le chapitre précédent et leur forme respective en Maude. Le symbole "_" représente l'emplacement d'une proposition atomique de LTL.

La commande de *model checking* (borné dans le temps) se présente avec la syntaxe suivante :

`(mc` t_0 `|= formule timeLimit .)`

Celle-ci vérifie si une formule de la logique temporelle `formule` tient dans tous les états accessibles à partir de l'état initial t_0 et dans la limite de temps `timeLimit`.

3.6 Conclusion

Nous avons présenté, dans ce chapitre, les notions relatives à la compréhension des concepts de base de la logique de réécriture, du système Maude et de son extension RT-Maude. Dans un premier temps, nous avons présenté l'aspect théorique 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. Avec ces concepts, la logique de réécriture possède un pouvoir d'abstraction et de description de tout type de système.

Ensuite, nous avons introduit le système Maude qui un langage caractérisé par la simplicité, la performance et l'expressivité. Maude est un candidat idéal pour la spécification de système car il est basé sur une logique très mathématique et dispose d'une forte sémantique et des techniques d'analyses formelles. L'extension RT-Maude est un langage très puissant et très expressif. Il offre un outil pour la spécification formelle, la simulation et l'analyse formelle des systèmes temps réel.

CHAPITRE 4

La modélisation avec UML

Sommaire

4.1 Introduction

La tâche de conception des systèmes a toujours besoin d'un langage ou d'une méthode de spécification et de modélisation pour créer et analyser des modèles et communiquer avec les collaborateurs et les clients.

L'approche orientée objet considère le logiciel comme une collection d'objets dissociés et identifiés, définis par des propriétés. Une propriété est soit un attribut soit une entité élémentaire comportementale de l'objet. La fonctionnalité du système émerge alors de l'interaction entre les différents objets qui le constituent. L'une des particularités de cette approche est qu'elle encapsule les données et leurs traitements associés au sein d'un unique objet. C'est dans ce contexte que nous présentons quelques éléments du langage UML qui s'est imposé comme un standard que rencontrent tous les développeurs dans l'industrie du génie logiciel.

4.2 La modélisation

La modélisation d'un système est le processus de développement des modèles. Un modèle est une représentation abstraite d'un système réel, construit pour un objectif donné [139]. Il contient un ensemble restreint d'informations sur le système modélisé. Le modèle construit contient toujours des informations pertinentes vis-à-vis de son utilisation future. D'autre part, la modélisation ore des avantages considérables aux concepteurs des systèmes tels que : la facilité de compréhension du fonctionnement des systèmes avant sa réalisation et un bon moyen de maîtriser sa complexité et d'assurer sa cohérence. La modélisation en informatique peut être vue comme la séparation des différents besoins fonctionnels et préoccupations extra-fonctionnelle [140] (telles que : la sécurité, la fiabilité, l'efficacité, la performance, la ponctualité, la flexibilité, etc.).

En générale, la réalisation d'un système fait l'objet d'un travail d'équipe. La communication entre les différents membres de l'équipe de travail détermine la réussite de tout projet. Le modèle se définit comme étant un langage commun aux membres de l'équipe, favorisant ainsi la communication et la compréhension entre les différentes parties concernées par le projet. Par exemple, dans l'ingénierie du logiciel le modèle permet une meilleure répartition des tâches entre les différents intervenants. Ceci joue un rôle important quant à la réduction des coûts et des délais de réalisation des projets.

4.2.1 Les types de modélisation

La classification de la modélisation peut se faire selon le degré du formalisme des langages ou des méthodes impliquées dans le processus de la modélisation. Ainsi, la modélisation peut être considérée comme étant formelle, semi-formelle ou informelle [141]. Le tableau de la figure 4.1 présente une définition des catégories de langages ainsi que des exemples de langages ou de méthodes qui l'utilisent.

Catégories de Langages			
Langage Informel		**Langage Semi-Formel**	**Langage Formel**
Simple	*Standardisé*		
Langage qui n'a pas un ensemble complet de règles pour restreindre une construction	Langage avec une structure, un format et des règles pour la composition d'une construction.	Langage qui a une syntaxe définie pour spécifier les conditions sur lesquelles les constructions sont permises.	Langage qui possède une syntaxe et une sémantique définies rigoureusement. Il existe un modèle théorique qui peut être utilisé pour valider une construction.
Exemples de Langages ou Méthodes			
Langage Naturel.	Texte Structuré en Langage Naturel.	Diagramme Entité-Relation, Diagramme à Objets.	Réseaux de Petri, Machines à états finis, VDM, Z.

Figure 4.1. Classification et utilisation de langages ou de méthodes [141].

4.2.1.1 ***Modélisation Informelle :*** Le processus de modélisation informelle à base de langages informels, se justifie selon [141] pour plusieurs raisons :
- La facilité de compréhension du langage permet des consensus entre les personnes qui spécifient et celles qui commandent un logiciel ;
- elle représente une manière familière de communication entre personnes.

Par ailleurs, l'utilisation d'un langage informel rend la modélisation imprécise et parfois ambigüe [142]. Le caractère informel de cette approche rend difficile toute tentative de standardisation.

4.2.1.2 ***Modélisation Semi-Formelle***

Le processus de modélisation semi-formelle est basé sur un langage textuel ou graphique pour lequel une syntaxe précise est définie [141]. La sémantique d'un tel langage et souvent assez faible. Néanmoins, ce type de modélisation permet d'effectuer des contrôles et de réaliser des automatisations pour certaines tâches.

La plupart des méthodes de modélisation semi-formelles s'appuient fortement sur des langages graphiques. Ceci se justifie par la puissance expressive du modèle graphique. Par ailleurs, l'appui de la modélisation semi-formelle (tels que : UML) sur des langages graphiques, permet la production de modèles assez faciles à interpréter.

Cependant, cette modélisation soufre de la déficience des aspects sémantiques impliqués dans l'approche. Afin de pallier aux insuffisances de cette approche, l'utilisation de contraintes a été introduite [143].

4.2.1.3 ***Modélisation Formelle***

Une méthode formelle est un processus de développement rigoureux basé sur des notations formelles avec une sémantique définie. Le principal avantage des aspects formelles est leurs capacités à exprimer une signification précise, permettant ainsi des vérifications de la cohérence et de la complétude d'un système. Par exemple, [144] montrent qu'avec une traduction

appropriée, les méthodes formelles peuvent aider à la compréhension d'un système par un utilisateur.

4.2.2 La Modélisation Orientée Objet

L'approche orientée objet est classée dans la catégorie des modélisations semi-formelles. Elle constitue une façon de penser les problèmes en appliquant des modèles organisés autour de concepts du monde réel.
L'orienté Objet considère le logiciel comme une collection d'objets dissociés et par conséquent son concept fondamental est l'objet, qui combine à la fois une structure de données et un comportement. La fonctionnalité du logiciel émerge de l'interaction entre les différents objets qui le constituent [145]. Les architectures logicielles à base d'objets décrivant les systèmes logiciels comme une collection de classes (les entités à abstraire et l'encapsulation des fonctionnalités) qui peuvent avoir des objets ou instances et communiquent entre eux, via des messages. Cette modélisation permet de comprendre des problèmes, de communiquer avec des experts du domaine d'applications, de modéliser le métier d'une entreprise, de réparer la documentation et de concevoir des programmes et des bases de données. Avec l'unification des méthodes de développement objet sous le langage UML (Unified Modeling Language), cette approche est largement utilisée et bien appréciée dans le monde industriel.

4.3 Historique des méthodes de conception

Dans le secteur du développement logiciel, suite à l'adoption massive de l'approche objet pour la réalisation des applications, et face au besoin pour les développeurs d'avoir à leur disposition de nouvelles méthodologies d'aide à la conception objet, une multitude de méthodes a été lancée dont une cinquantaine entre 1990 et 1995, chacune essayant de s'imposer sur le vaste marché du développement logiciel.
Comme son nom l'indique, le langage UML est né de la fusion de plusieurs langages de modélisation, issus notamment de la méthode *Object-Oriented Analysis and Design* (OOD) de Grady Booch [146],particulièrement adaptée à la conception et à l'implémentation, de la méthode OOSE (*Object Oriented Software Engineering*) de Ivar Jacobson [147] : qui permettait essentiellement l'expression des besoins, et de la méthode OMT (*Object Modelling Technique*) de James Rumbaugh [148] : pour l'analyse et applications orientées données.

En octobre 1994, Grady Booch et James Rumbaugh se sont réunis au sein de la société RATIONAL SOFTWARE dans le but de travailler à l'élaboration d'une méthode commune qui intègre les avantages de l'ensemble des méthodes reconnues, en corrigeant les défauts et en comblant les manques. Lors de la conférence OOPSLA'95 (*Object Oriented Programming Systems, Languages and Applications*, la grande conférence de la programmation orientée objet), ils présentent UNIFIED METHOD V0.8, puis en 1995 Jacobson les rejoint en RATIONAL SOFTWARE et le 14 Novembre 1997 : UML [Booch *et al.* 1998] est une notation et un langage standardisé en 1997 par l'OMG (***O**bject **M**odeling **G**roup*) [149] qui facilite la conception de programmes, ainsi que leur description pour des non-informaticiens. La figure 4.2 montre l'historique de constitution du langage UML.

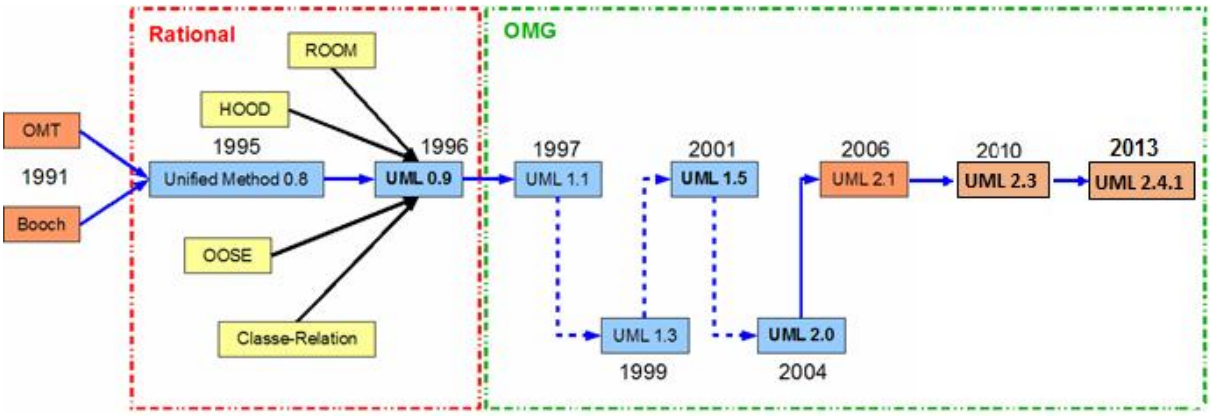


Figure 4.2. Ligne de vie du langage UML.

UML est devenu un langage standard de spécification, de visualisation, de construction et de documentation des systèmes logiciels. UML, représente une collection des meilleures pratiques d'ingénierie ayant un grand succès dans la modélisation des systèmes larges et complexes. UML utilise généralement des notations graphiques pour représenter les aspects structurels, dynamiques et fonctionnels d'un système logiciel. Mais, la notation graphique est intrinsèquement limitée lors de la spécification des contraintes complexes. Donc, UML décrit un ensemble de contraintes en langage naturel, et à l'aide du langage OCL (*Object Constraint Language* [150]). Ces contraintes peuvent être définies au niveau du méta-modèle comme au niveau du modèle. OCL permet une conception riche et correcte.

La version d'UML en cours à la fin 2004 est UML 2.0 et les travaux d'amélioration se poursuivent. On ne peut donc considérer UML comme étant uniquement un outil intéressant, mais il est également une norme et un standard en technologie à objets à laquelle se sont liés tous les grands acteurs du domaine, qui ont d'ailleurs contribué à son élaboration.

4.4 UML (Unified Modeling Language)

UML (Unified Modeling Language) est un langage de modélisation graphique destiné à visualiser, analyser, spécifier, construire des logiciels orientés objets [151] [152], ainsi que la modélisation des systèmes non logiciels. UML est aujourd'hui considéré comme un standard autant dans le milieu industriel qu'académique.

UML a la particularité de s'intéresser essentiellement à la modélisation c'est-à-dire la représentation des différents concepts qui interviendront dans l'écriture d'un logiciel [153]. Il est devenu le langage de modélisation orienté objet de référence dans le monde professionnel car il comble une lacune importante des technologies objet : il permet d'exprimer et d'élaborer des modèles objet, indépendamment de tout langage de programmation. Il a été pensé pour servir de support à une analyse basée sur les concepts objet. C'est également un langage graphique semi-formel orienté objet issu des meilleurs outils et pratiques du génie logiciel du début des années 90. Ce langage de modélisation repose sur deux concepts essentiels :
1. La modélisation du monde réel au moyen de l'approche orientée objet,
2. L'élaboration d'une série de diagrammes facilitant l'analyse et la conception des systèmes, et permettant de représenter les aspects statiques et dynamiques du domaine à modéliser ou à informatiser.

4.4.1 Les diagrammes UML

UML puise des avantages de la modélisation semi-formelle en offrant à l'utilisateur un moyen de visualiser et de manipuler des éléments de modélisation. Les éléments de représentation sont

le plus souvent des graphes connexes où les sommets correspondent aux éléments et les arcs aux relations. Ces graphes servent à visualiser un système sous différentes vues.
UML propose un certain nombre de diagrammes qui servent à visualiser un système sous différentes perspectives et pour couvrir l'ensemble des besoins de modélisation potentiellement nécessaires à la conception des logiciels, ce qui le rend relativement complet et générique. Pour les systèmes complexes, un diagramme ne représente qu'une vue partielle des éléments qui composent ces systèmes.

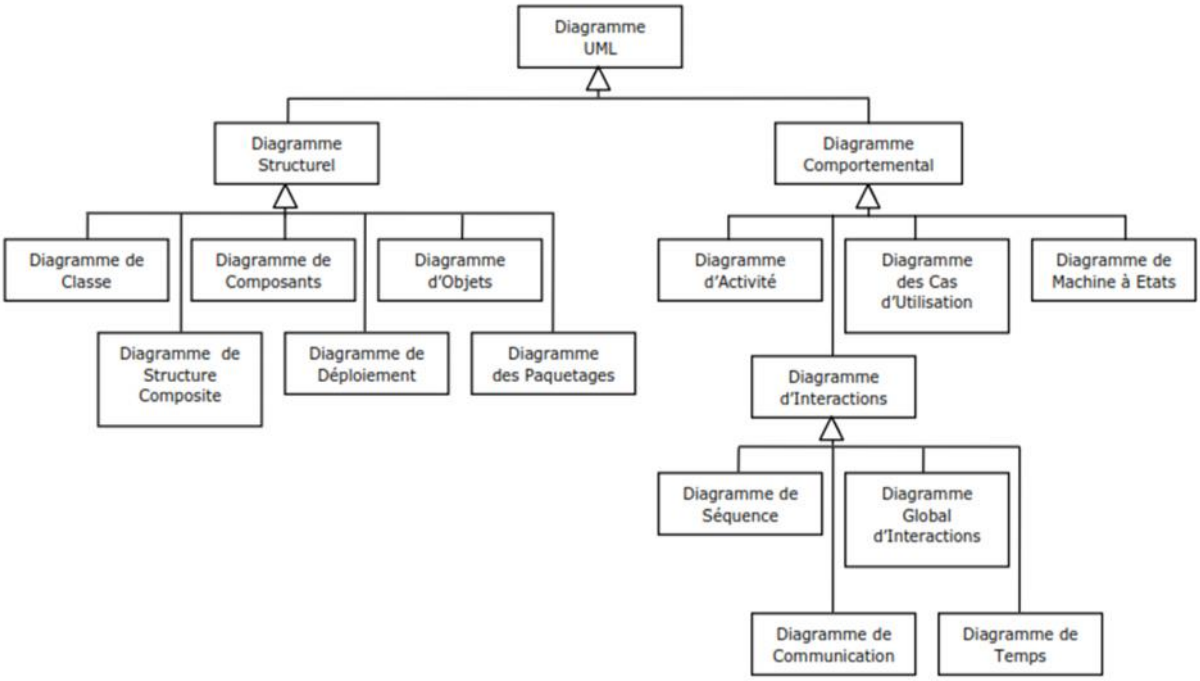


Figure 4.3. Les différents diagrammes d'UML 2.0 sous forme de diagramme de classe.

Ainsi, à travers des diagrammes (figure 4.3) [154], UML permet de modéliser les aspects statiques et dynamiques des systèmes complexes et de couvrir la plupart des phases du développement logiciel (analyse, conception, implantation, déploiement, etc.). La version d'UML 2.0 dispose de 13 diagrammes officiels (contre 9 dans la version 1.5). Les diagrammes sont répartis en trois catégories : les diagrammes structuraux, les diagrammes comportementaux, Les diagrammes d'interactions (qui peuvent être vus comme une sous-catégorie des diagrammes de comportement) :

- *Le mode de représentation statique ou structurel :* Cette vue modélise la structure des différentes classes d'une application orientée objet, elle réunit :
  - Diagramme de classes (*Class diagram*).
  - Diagramme d'objets (*Object diagram*).
  - Diagramme de composants (*Component diagram*).
  - Diagramme de déploiement (*Deployment diagram*).
  - Diagramme de paquetages (packages) (*Package diagram*).
  - Diagramme de structures composites (*Composite structure diagram*).
- *Le mode de représentation dynamique ou comportemental :* Cette vue est fonctionnelle, elle est plus algorithmique et orientée « traitement », et vise à décrire l'évolution (la dynamique) des objets complexes du programme tout au long de leur cycle de vie. De leur création à leur destruction, les changements d'états des objets sont guidés par les interactions avec les autres objets. Cette vue est présentée avec les diagrammes suivants :
  - Diagramme de cas d'utilisation (*Use case diagram*).
  - Diagramme d'activités (*Activity diagram*).

- Diagramme d'états-transition (*State machine diagram*).

- *Diagrammes d'interactions (Interaction diagram) :* Pour montrer l'interactivité, des diagrammes traitent les interactions entre les différents acteurs/utilisateurs et le système sous forme d'objectifs à atteindre d'un côté, et sous forme chronologique de scénarios d'interaction typiques de l'autre. Ces diagrammes sont les suivants :
  - Diagramme de séquence (*Sequence diagram*).
  - Diagramme de communication (*Communication diagram*).
  - Diagramme global d'interaction (*Interaction overview diagram*).
  - Diagramme de temps (*Timing diagram*).

Les diagrammes de classes : Le diagramme de classes UML est un diagramme permettant de décrire la structure statique d'un système logiciel. Il modélise les différentes parties d'un système sous forme de classes et leurs relations sous forme d'héritage, d'associations ou d'agrégation. Chaque classe du diagramme est caractérisée par un nom, un ensemble d'attributs et un ensemble de méthodes ou opérations. Le diagramme de classes est traditionnellement utilisé pour la modélisation orientée objet afin de représenter les différentes classes d'objets et les liens entre ces classes. On obtient ainsi une vue statique de la structure du système logiciel. Il permet aussi de représenter des méta-modèles.

Les diagrammes d'objet : sont destinés à représenter les instances des classes définies dans le diagramme de classe et les liens qui les connectent. Ce diagramme fige ainsi une image du système modélisé à un instant donné.

Les diagrammes de composant : manipulent la notion de composant qui permet de représenter une abstraction d'un ensemble de classes réalisant différentes fonctionnalités et qui communiquent par des interfaces avec leur environnement. Le diagramme de composant permet d'expliciter les unités logicielles qui forment le socle du système, Il inclut les classificateurs (i.e. classes) qui spécifient les composants et les artefacts qui les implémentent, tels que les fichiers de code source, de code binaire, exécutables ou scripts.

Les diagrammes de déploiement : présentent la configuration des éléments de traitement en temps d'exécution, ainsi que les composants logiciels, les processus et les objets qui les exécutent.

Les diagrammes de structure composite : Un diagramme de structure composite est un graphique représentant la structure interne d'une ou plusieurs classes. Une classe sur un diagramme de composite contient un ensemble de parties reliées par des connecteurs. Une partie possède un type et une multiplicité. Les parties peuvent être typées par des composants. Un composant possède un ensemble d'interfaces et une ou plusieurs implémentations.

Les diagrammes de cas d'utilisations : permettent pour leur part d'identifier les fonctionnalités d'un système et les conditions nécessaires à leur bon fonctionnement. Ils font apparaître les éléments fonctionnels, les acteurs et les objets en interaction. Un cas d'utilisation représente un ensemble de séquences d'actions qui sont réalisées par le système et qui produisent un résultat observable intéressant pour un observateur particulier. Chaque cas d'utilisation spécifie un comportement attendu du système considéré comme un tout, sans imposer le mode de réalisation de ce comportement. Il décrit ce que le système devra faire, sans spécifier comment il le fera [155].

Les diagrammes d'activité : Un diagramme d'activité [152] permet de représenter de façon détaillée les différentes séquences d'actions (un traitement ou une transformation) d'une activité (par exemple, un cas d'utilisation). Pour cela, il inclut des supports pour le parallélisme, la décision, l'itération, et la synchronisation entre les actions. Une activité dans ce type de diagramme est représentée par un état initial qui marque le début de l'activité, puis un flot d'éléments qui décrit l'activité, et enfin un état final qui marque la fin de celle-ci.

Les diagrammes d'états-transition (machines à états) : Un diagramme de machine à état est un graphe d'états reliés par des transitions. Le franchissement des transitions se réalise à la suite de la réception d'un signal (appel de méthode, exception, etc.). Ce type de diagramme est habituellement relié à une classe et il décrit la réponse d'une instance de la classe aux événements qu'elle reçoit. On peut également rattacher les machines à états à des comportements, des cas d'utilisation et à des collaborations pour décrire leur exécution. Une machine à états est un modèle de tous les états possibles que peut prendre un objet de classe. Elle résume toute influence en provenance du reste du monde comme un événement. Lorsqu'un objet détecte un événement, il y répond en fonction de son état actuel. Cette réponse peut comprendre l'exécution d'un effet et d'un changement vers un autre état.
Dans le cadre de cette thèse, afin d'illustrer notre proposition, nous nous sommes appuyés sur les diagrammes d'états-transition, qui seront décrites en détail par la suite dans ce chapitre.

Les diagrammes de séquence : Les diagrammes de séquence [152] permettent de représenter pour un scenario donné, les interactions séquentielles dans le temps entre les différents objets impliqués (dans ce scenario). Dans un diagramme de séquence, chaque objet est représenté par une *ligne de vie*, représentée par une ligne verticale, symbolisant la durée de vie de l'objet dans le diagramme. Les interactions s'effectuent par échange de messages représentés par différents types de flèches (selon les rôles), suivant un ordre chronologique (du haut vers le bas).

Les diagrammes de communication : décrivent des configurations types en termes d'objets collaborant. Pour cela, ils permettent de représenter d'une part la configuration d'un ensemble d'objets et d'autre part les relations dynamiques entre ces objets.

Selon [156] la Figure 4.4 décrit le nombre de diagrammes régulièrement utilisés par des experts industriels selon différents cas d'étude.

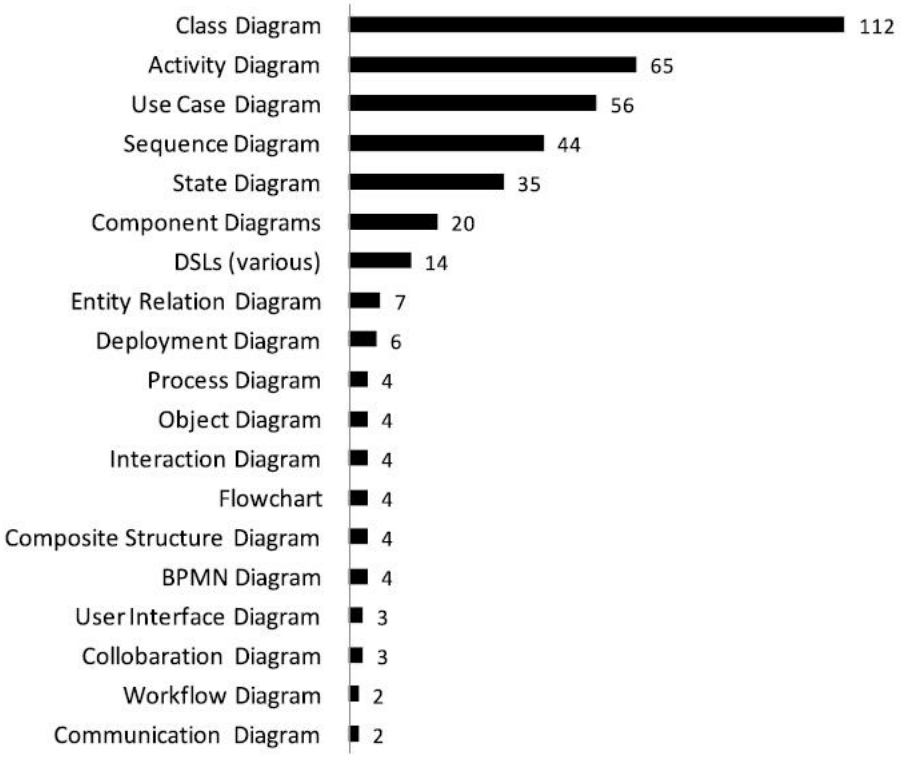


Figure 4.4. Liste des diagrammes utilisés régulièrement en UML [156].

4.4.2 Extensions d'UML

Les mécanismes d'extension d'UML (stéréotypes, valeurs marquées, contraintes et profils) peuvent servir pour raffiner la sémantique standard d'UML et permettre ainsi l'ajout de nouveaux éléments de modélisation qui seront utilisés dans la création de modèles UML spécifiques.

4.4.2.1 Stéréotypes

Le mécanisme d'extension stéréotypes définit un nouveau type d'élément de modèle basé sur un élément de modèle existant. Ainsi, un stéréotype est comme l'élément existant, avec une sémantique et des propriétés supplémentaires qui ne sont pas présents dans l'élément existant. Graphiquement, un stéréotype est rendu par un nom entouré de guillemets («» ou par << >> si les guillemets ne sont pas disponibles) et placé au-dessus de nom d'un autre élément. Un stéréotype peut aussi être indiqué par une icône spécifique [157]. La figure 4.5 représente des exemples de stéréotype.

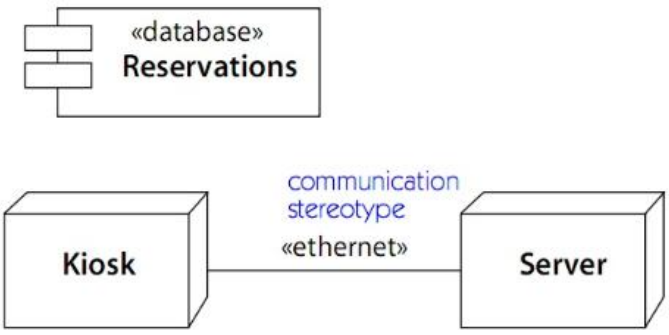


Figure 4.5. Exemple de stéréotype[8] [158].

[8] Kiosk est un cadre d'applications qui permet de limiter et d'encadrer les possibilités offertes par l'environnement KDE.

4.4.2.2 Valeurs étiquetées (tagged values)

Une valeur marquée est une paire **(nom, valeur)** qui ajoute une nouvelle propriété à un élément de modélisation [159], en plus de ces propriétés déjà définies dans le méta-modèle UML.
Les valeurs étiquetées sont un mécanisme d'extensibilité permettant d'associer des informations arbitraires à des modèles. Elles sont exprimées sous la forme *"nom = valeur"*, par exemple : *author*="David", *project_phase*=2, *OS*="Windows" ou *last_update*="1-07-02".

4.4.2.3 Le langage de contraintes OCL (Object Constraints Language)

Le langage UML est régi par un méta-modèle définissant sa structure et les règles à respecter pendant la construction des diagrammes. Cependant, lorsqu'il s'agit d'exprimer des opérations simples ou des contraintes, par exemple pour l'expression des conditions sur les diagrammes dynamiques d'UML comme les pré et post conditions ainsi que les gardes sur les transitions entre états d'objets, il n'y a pas de directives au niveau d'UML. Pour répondre à ce besoin, un langage formel de description de contraintes a été mis en place par l'OMG, notamment pour l'enrichissement des diagrammes UML en permettant de définir des contraintes et des opérations avec un langage indépendant d'un langage de programmation cible. On retrouve également ce langage au niveau du métamodèle UML pour exprimer les contraintes structurelles de construction des diagrammes et de relation entre les entités du langage. OCL est un langage formel utilisé pour spécifier les contraintes ainsi que d'autres expressions associées aux modèles UML. Une contrainte définit une condition qui doit être vraie pour la durée du contexte dans lequel elle est définie. Les contraintes définissent les invariants qui permettent de préserver l'intégrité du système [160].

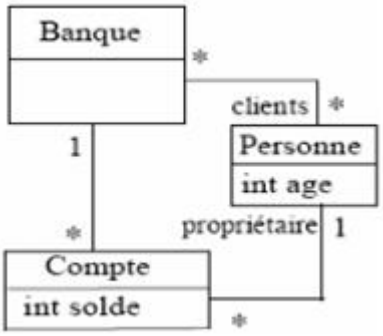


Figure 4.6. Diagramme de classe support des exemples de contraintes OCL.

A titre d'exemple, dans la figure 4.6, exprimer le fait que le solde d'un compte doit être positif ne peut pas être fait avec UML mais peut être formalisé avec la contrainte OCL suivante :

Context Compte
Inv : solde > 0

4.4.3 Vues et diagrammes UML

Les différentes vues d'un système représentent sa formulation. Elles sont indépendantes et complémentaires. Ces vues permettent de définir l'architecture du système modélisé. Chaque vue est une projection, selon un aspect particulier, dans l'organisation et la structure du système. [161] propose un modèle pour l'architecture de systèmes logiciels basé sur cinq vues, désignées par les « 4+1 » vues. L'utilisation de ces différentes vues permet de traiter séparément les préoccupations des différentes « parties prenantes » de l'architecture : les utilisateurs finaux, les développeurs, les ingénieurs systèmes, les chefs de projet, etc., et de gérer séparément les exigences fonctionnelles et non-fonctionnelles. Les cinq vues concernent :

- La *vue logique* est d'ordre conceptuel, son objectif est de modéliser les éléments et les mécanismes principaux du système. Dans ce cadre, UML implique par exemple les classes et les interfaces comme éléments de modélisation.
- La *vue des processus* qui capte les aspects de concurrence et de synchronisation pour la conception, représente la vue temporelle et technique en manipulant des notions tels que : les tâches concurrentes, la synchronisation, les processus, etc.
- La *vue physique* qui décrit la mise en correspondance du logiciel sur le matériel et reflète son aspect distribué
- La *vue de déploiement* qui décrit l'organisation statique du logiciel dans son environnement de développement,
- La *vue de cas d'utilisation* qui est redondante avec les autres vues (d'où le « +1 »). Néanmoins, elle permet de découvrir les éléments architecturaux lors de la conception et de valider cette architecture. En effet, elle assure la cohérence globale en présentant des scénarios d'utilisation qui sont en quelque sorte une abstraction des exigences les plus importantes et permet de décrire le système tel qu'il est vu par les acteurs du système lui-même.

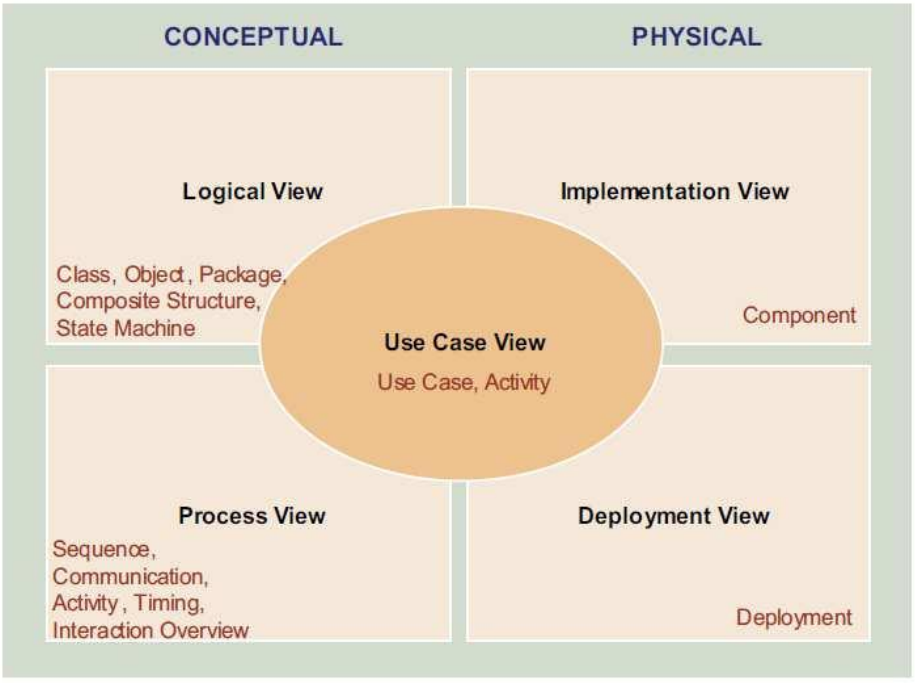


Figure 4.7. L'application de l'architecture des « 4+1 » vues de Krutchen sur UML [162].

Comme le modèle de Kruchten est apparu avant UML, plusieurs tentatives d'application de l'architecture à « 4+1 » vues sur UML existent. Comme certains diagrammes UML peuvent être utilisés dans plusieurs vues, les propositions de classification de ces diagrammes diffèrent. Cependant, la modélisation du même type de diagramme UML change d'une vue à une autre. Nous faisons référence au travail de [162] qui fait correspondre chaque type de diagramme UML à une vue architecturale. La figure 4.7 illustre cette proposition, parmi d'autres comme celle d'IBM.

4.4.4 UML et Cycle de développement

Les diagrammes UML peuvent être utilisés à tous les niveaux de l'analyse et conception orientées objet (***O**bject-**O**riented **A**nalysis and **D**esign*). Leur utilisation est subtilement différente à chaque niveau [163] :

Au **niveau de l'analyse et de la spécification des besoins** (*requierements analysis*) les diagrammes UML permettent de comprendre le domaine fondamental ; à ce moment, il n'y a pas besoin de penser aux relations avec le logiciel résultant. Les modèles du domaine sont des modèles conceptuels et ont une place importante pour débuter ensuite la conception. Les diagrammes de cas d'utilisation sont très souvent utilisés à ce moment pour capturer les besoins des utilisateurs.

Au **niveau de l'analyse** (*analysis*) et toujours dans une optique d'indépendance avec l'implantation, l'objectif est de modéliser le monde réel en déterminant les classes d'objets du monde réel dans un premier diagramme de classes et de modéliser également la dynamique du système notamment par un diagramme de collaboration. Les diagrammes d'activités et d'états-transitions peuvent être également utilisés.

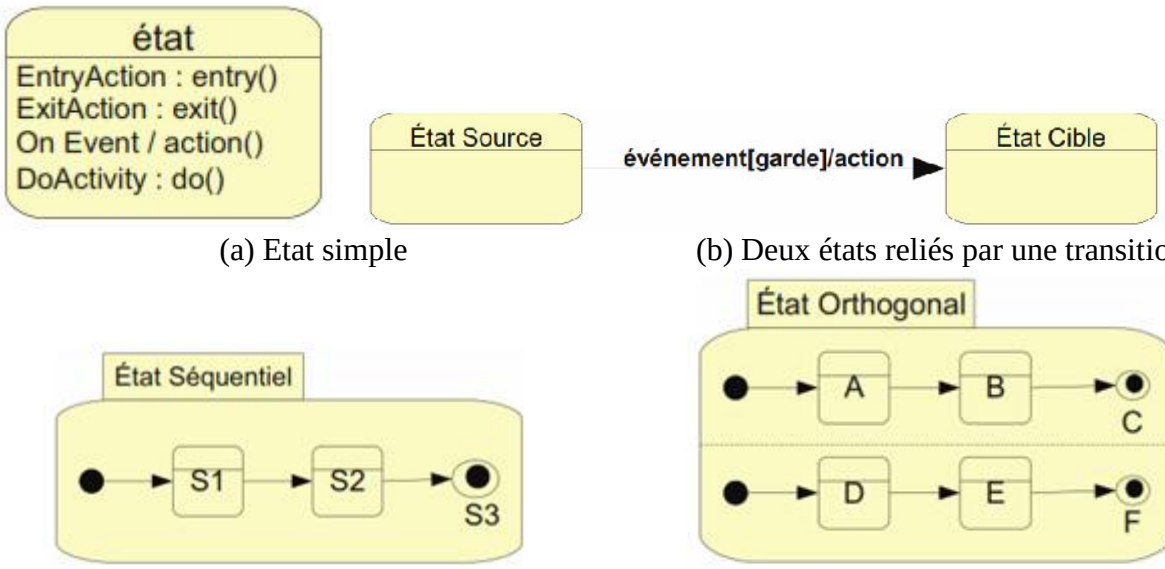
Au **niveau de la conception** (*design*), l'architecture informatique est prise en compte et les entités des diagrammes sont examinées comme des spécifications de ce que le code devrait être. C'est souvent dans cette perspective que les diagrammes UML sont les plus utilisés. À ce niveau, des classes techniques sont ajoutées pour gérer l'interface graphique, la distribution, la persistance et la concurrence. Les diagrammes manipulés lors de cette phase sont les diagrammes de classes, de séquences, de composants, de déploiement et d'états-transitions.

Au **niveau de l'implantation** (*programming*), les classes de conception sont converties vers les langages cibles (par exemple Java, SQL, C++, IDL) et les classes persistantes sont converties vers des modèles de persistances (par exemple SGBD, BDOO, langages persistants). L'attention portée sur les diagrammes à ce niveau est concentrée sur les attributs et les opérations principales pour que « l'image » du système ne soit pas trop grande.

Dans la **phase de tests**, les diagrammes UML définis dans les phases précédentes permettent de diriger des tests unitaires, des tests d'intégration et des tests du système codé de manière générale. Les diagrammes de classes sont utilisés notamment pour mener à bien des tests unitaires classe par classe et méthode par méthode alors que les diagrammes de composants sont plus utiles pour les tests d'intégration. Les diagrammes de cas d'utilisation et d'activités permettent enfin de vérifier la conformité des fonctionnalités implantées dans le système.

4.4.5 Diagramme d'états-transitions

Le diagramme d'états-transitions est le seul diagramme, de la norme UML, à offrir une vision complète de l'ensemble des comportements de l'élément auquel il est attaché (généralement une classe active).
Les diagrammes d'états-transitions (*statecharts diagram*), concept utilisé par David Harel pour son extension de notation de machine d'état à plat comprend des états imbriqués et concurrents. Cette notation a servi de base à la notation de la machine UML.
Les diagrammes d'états transition UML représentent en réalité des automates à états finis par des graphes orientés.
Le diagramme d'état-transitions illustre l'ensemble des états du cycle de vie d'un objet séparés par des transitions (figure 4.8 (b)). Chaque transition est associée à un événement [164]. Il est utilisé en génie logiciel pour représenter des automates déterministes. Il s'inspire principalement du formalisme des Statecharts [165].

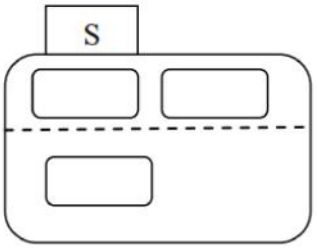
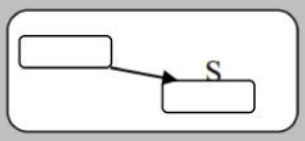
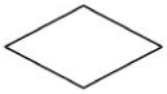


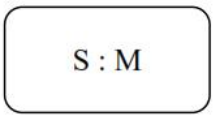
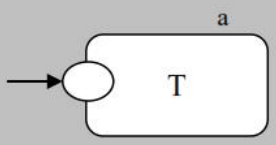
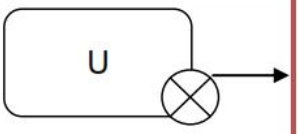
(a) Etat simple (b) Deux états reliés par une transition

(c) Etat séquentiel (d) Etat orthogonal

Figure 4.8. (a) Etat simple, (b) Deux états reliés par une transition, (c) Etat séquentiel, (d) Etat orthogonal.

Tableau 4.1. Types d'états.

Type état	description	Notation
Etat simple	Etat dépourvu de sous structure	
Etat orthogonal	Etat divisé en deux régions ou plus, un sous état direct de chaque région est simultanément actif avec l'état composite lorsque ce dernier est actif	
Etat non orthogonal	Etat composite contenant un ou plusieurs sous état directs ; un seul d'entre eux est exactement actif à la fois lorsque l'état composite est actif	
Etat initial	Pseudo –état, qui indique l'état de départ lorsque l'état enveloppant est invoqué	
Etat final	Eta spécial dont l'activation indique que l'état enveloppant est terminé	
Terminaison	Etat spécial dont l'activation achève l'exécution de l'objet de la machine d'états.	
Jonction	Pseudo–état qui relie des segments de transition en une seule transition de type RTC (run–to-completion)	
choix	Pseudo état qui crée un embranchement dynamique dans une transition de type RTC	

Etat historique	Pseudo état dont l'activation restaure l'état précédemment actif dans un état composite	
Etat de sous -machine	Etat qui indique une définition de machine d'états qui remplace conceptuellement l'état de sous machine.	
Point d'entrée	Pseudo–état visible de l'extérieur dans une machine d'états qui identifie un état interne comme cible	
Point de sortie	Pseudo état visible de l'extérieur dans une machine d'états qui identifié un état interne comme une source	

Un état peut être associés un ensemble d'actions, une activité et un invariant. Une action d'état est un traitement associé à un état. Le déclenchement d'une action d'état est lié soit à l'entrée ou à la sortie de l'état ou à l'apparition d'un événement. Une activité d'état est une séquence d'actions qui s'exécutent tant que l'objet est dans cet état [45]. On distingue trois types d'actions et une activité :

- *Entry Action* : chaque fois que l'objet entre dans l'état, l'action est exécutée,
- *Exit Action* : chaque fois que l'objet quitte l'état, l'action est exécutée,
- *OnEvent Action* : quand l'objet se trouve dans l'état, chaque fois que l'événement cité survient, l'action est exécutée,
- *Do Activity* : action exécutée en boucle dans l'état tant que ce dernier est actif.

Les états dans un diagramme d'états-transitions (DET) peuvent être de trois types : état simple, état séquentiel et état concurrent (ou orthogonal). L'état simple, représenté à l'aide d'un rectangle à coins arrondis, est utilisé pour représenter une situation élémentaire dans laquelle l'objet peut se trouver (figure 4.8 (a)). Les deux autres types d'états (séquentiel et orthogonal) sont appelés "états composites". Dans un état séquentiel, noté OR (figure 4.8 (c)) et à un instant donné, un objet ne peut se trouver que dans un seul sous-état de l'état séquentiel. L'état séquentiel contient une seule région. L'état orthogonal, nommé AND (figure 4.8 (d)), permet de décrire deux ou plusieurs sous-états concurrents au sein d'un même état. Dans un état orthogonal et à un instant donné, un objet se trouve dans un seul sous-état direct dans chacun de ses états séquentiels (régions). Les types d'états sont décrit en plus de détail dans le tableau 4.1 [166] [167].

Une transition est utilisée pour exprimer le passage instantané d'un état vers un autre. Une transition est déclenchée par un événement. Un événement est un stimulus dont l'occurrence est susceptible d'entraîner le déclenchement d'une réaction au sein du système modélisé. Deux types d'événements sont définis dans les DET : les événements *externes* échangés entre objets et les événements *internes* créés, émis et reçus au sein du même objet. Ces événements sont répertoriés en cinq classes :

- *CallEvent* : indique la réception d'un message synchrone (un appel d'opération). Il résulte de l'exécution de l'opération ainsi que d'un changement d'état (ou configuration).
- *AnyReceiveEvent* : déclenche la transition à la réception de tout message reçu.
- *TimeEvent* : relatifs aux événements temporisés associés aux annotations *after* et *when*.
- *ChangeEvent* : associé à une expression booléenne qui est continuellement évaluée.
- *SignalEvent* : indique la réception d'un message asynchrone. Il résulte de l'exécution de l'opération ainsi que d'un changement d'état (ou configuration).

Le tableau 4.2 décrit les différents types d'événements qui peuvent être présentées dans le diagramme d'états-transitions [166].

Tableau 4.2. Types d'événements.

Type d'événement	Description	Syntaxe
Appel	Réception d'une demande d'appel explicite synchrone par un objet	**Op (a :T)**
Changement	Changement dans la valeur d'une expression booléenne	**When (exp)**
Signal	Réception d'une communication explicite, nommé asynchrone entre des objets	**sname (a :T)**
Temps	Arrivée d'un temps absolu ou passage d'un laps de temps relatif	**after (time)**

Une transition est un arc orienté entre deux états conditionnés par le déclenchement de l'événement auquel elle est associée. Le déclenchement d'une transition provoque le passage instantané d'un état à un autre.
En plus des événements, il est aussi possible d'étiqueter une transition à l'aide de gardes et/ou d'actions. Une garde est une expression booléenne associée à une transition qui doit être vérifiée afin que celle-ci puisse être franchie. Une action est un traitement instantané à exécuter lorsque la transition est franchie. Le tableau 4.3 résume les différents types de transition et effets implicites [166].

Tableau 4.3. Type de transition et effets implicites.

Type de transition	description	Syntaxe
Transition entry	Spécification d'une activité d'entrée qui s'exécute lorsqu'on saisit un état	**Entry/activity**
Transition exit	Spécification d'une activité de sortie qui s'exécute lorsqu'on quitte un état	**Exit /activity**
Transition externe	Réponse à un événement qui engendre un changement d'état ou une auto transition, ainsi qu'un effet spécifié. Elle peut également entrainer l'exécution d'une activité exit et/ou d'une activité entry pour les états dont l'on sort ou dans lesquels on entre.	**(a :T)[guard]/activity**
Transition interne	Réponse à un événement qui entraine l'exécution d'un effet mais pas d'un changement d'état, ou l'exécution d'activités exit ou entry	**e (a : T) [guard]/activity**

Pour conclure cette section, nous présentons dans la figure 4.9 [168], un exemple de diagramme d'états-transitions, qui représente un système d'appel téléphonique. Ce système est non terminant où il y a un seul état de départ « *idle* ». Le système contient un *e*xemple de super-état avec transition sur l'événement raccrocher (*hangs up*) alors déconnexion (*disconnect*). Un type prédéfini d'événement appelé signal "elasped-time" (*after*) ainsi qu'un signal booléen (*when*) sont utilisés dans ce diagramme.

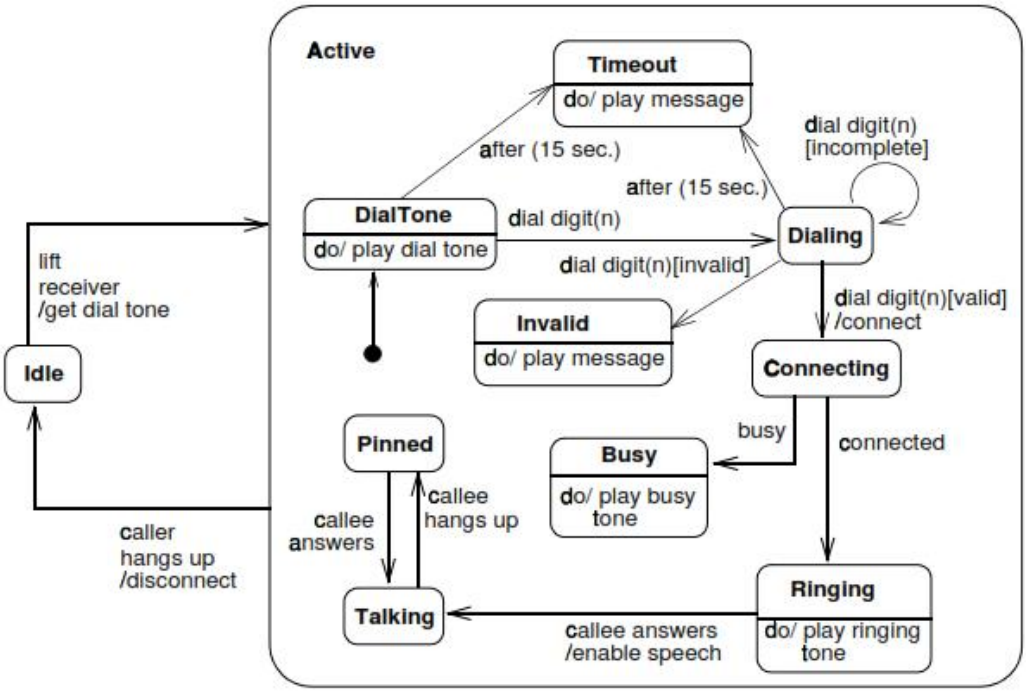


Figure 4.9. Un diagramme d'états-transitions correspondant à un système d'appel téléphonique.

4.5 Processus de développement

La conception des architectures logicielles a pour but de créer des applications répondant aux attentes des utilisateurs. Ces architectures doivent être analysées, conçues, mises en œuvre et doivent être flexibles et aisément maintenables au rythme des changements des besoins. Une description visuelle commune de l'architecture aide également les analystes, les architectes et les développeurs à rester alignés lors d'une activité de réutilisation ou de reconfiguration.

Dans la mesure où nous cherchons à améliorer l'intégration des activités de vérification aux processus de développement existants, nous nous intéressons plus particulièrement aux mécanismes proposés pour produire les artefacts nécessaires à la conduite des activités de vérifications. Nous allons présenter deux exemples de processus de développement largement utilisés pour les projets de développement de logiciels, construites autour du langage UML ; le cycle de développement selon la méthode UP et la méthode 2TUP (2 Tracks Unified Process).

4.5.1 Méthode Processus Unifié (UP)

La méthode de type UP s'impose souvent dans des grands projets (*Rational Unified Process*) et est construite autour du langage UML [169]. Selon [170], un processus unifié (UP) est "itératif et incrémental, centré sur l'architecture, construite à partir des cas d'utilisations et pilotée par les risques". La figure 4.10 montre le cycle itératif, découpé en 5 phases, de la méthode UP.

• *Le Recueil des besoins* permet de définir les besoins et faire la distinction entre les besoins *fonctionnels*, permettant la construction des modèles, et les besoins *non fonctionnels* qui forment la liste des exigences à satisfaire (diagramme de cas d'utilisation)

• *l'Analyse* a pour objectif de comprendre les besoins et les exigences afin de produire les spécifications des modèles d'analyse. Les modèles UML correspondants à cette spécification sont les cas d'utilisation pour une spécification complète des exigences fonctionnelles ainsi que leurs structures sous forme de scénarii (diagrammes de séquences et d'interactions).

• *La Conception* est l'activité où sont approfondis les connaissances des différents composants du système ainsi que leurs principales interfaces (diagrammes de classes, diagramme

d'activités). Ceci prépare la phase d'implémentation en décomposant le système en plusieurs sous-composants qui correspondent aux différentes classes du système.
• L'*implémentation et le déploiement* concerne la planification et l'intégration des composants pour chaque itération et la production des classes et des sous-systèmes sous forme de code source. Le déploiement est décrit par les diagrammes de déploiement UML.
• Les *test* permet de confronter l'implémentation aux cas de tests planifiés à chaque itération à partir de la phase du recueil des besoins.

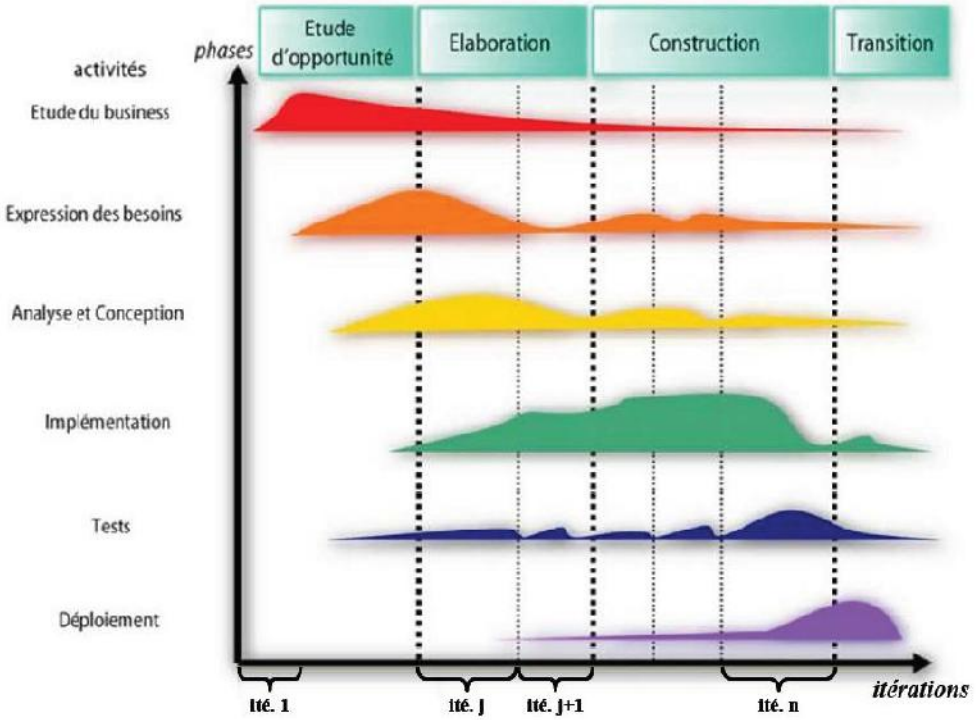


Figure 4.10. Activités et intensité de production en fonction de l'avancement du projet [171] [172] [173].

4.5.2 Méthode 2 Tracks Unified Process (2TUP)

2TUP (2 Tracks Unified Process) est un processus UP, dédié à la modélisation des systèmes d'information. Il apporte une réponse aux contraintes de changement continuel imposé aux systèmes d'information. En ce sens, il renforce le contrôle sur les capacités d'évolution et de correction de tels systèmes. Il intègre également la notion de composant selon différentes granularités : composants métier, composants logiciels, etc. [174] [175].

L'axiome fondateur de ce processus est de diviser la démarche en deux branches : *fonctionnelle* (approche par les fonctionnalités) et *technique* (étude de leur mise en œuvre). Le bénéfice attendu est de pouvoir réutiliser l'aspect fonctionnel en cas de changement d'architecture technique, et de pouvoir réutiliser l'aspect technique en cas d'évolution du fonctionnel (ou tout simplement pour développer une autre application présentant les mêmes contraintes d'architecture).

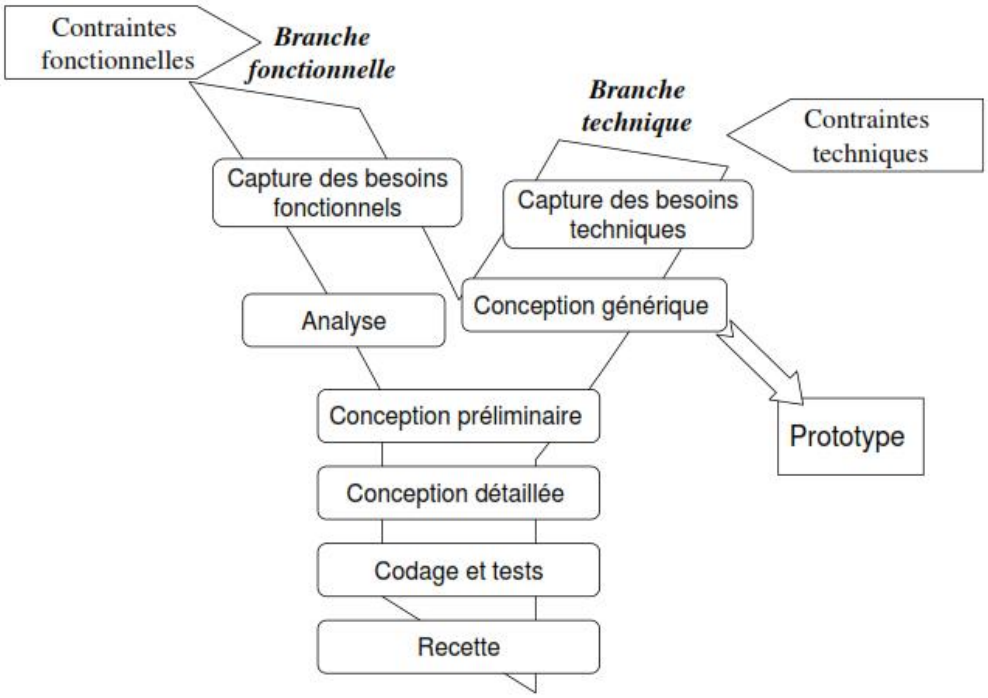


Figure 4.11. Le processus 2TUP.

A l'issue des évolutions du modèle fonctionnel et de l'architecture technique, la réalisation du système consiste à fusionner les résultats des deux branches. Cette fusion conduit à l'obtention d'un processus de développement en forme de Y qui dissocie les aspects techniques des aspects fonctionnels. Illustré par la figure 4.11, le processus en « Y » s'articule autour de 3 phases : technique, fonctionnelle et de réalisation. 2TUP est construit sur UML. Il est itératif, centré sur l'architecture et conduit par les cas d'utilisation.

4.6 Conclusion

UML est le langage de modélisation objet de référence. UML définit des notations, essentiellement graphiques, pour définir différents types de diagrammes qui représentent différentes vues et aspects d'un domaine à modéliser. Le principal avantage de UML est de disposer de notations très largement visuelles, intuitives à manipuler pour l'humain, et très connues. OCL fournit aussi un mécanisme pour exprimer des contraintes d'invariance et des règles sur les modélisations UML. Les vues UML 2.0 ont été montrées à travers les treize diagrammes faisant d'UML 2.0 le langage de modélisation de référence. Le diagramme d'états-transitions, qui intéresse le sujet de cette thèse, a été détaillé dans la deuxième partie de ce chapitre.

Le chapitre suivant va introduire les approches de la transformation des modèles qui permettent d'étendre l'utilisation de la modélisation orientée objet pour donner des solutions aux problèmes liés développement des systèmes temps réel.

CHAPITRE 5

Ingénierie Dirigée par les Modèles (IDM)

Sommaire

5.1 Introduction

Suite à l'approche objet des années 80 et de son principe du « *tout est objet* », l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM, ou Model Driven Engineering (MDE) en anglais) suivant le principe du « *tout est modèle* » [176]. La modélisation en informatique peut être vue comme la séparation des différents besoins fonctionnels et préoccupations extra-fonctionnelles (telles que : la sécurité, la fiabilité, l'efficacité, la performance, la flexibilité, etc.).

L'ingénierie Dirigée par les modèles, terme proposé par [3], est une forme d'ingénierie générative, qui se caractérise par une démarche rigoureuse par laquelle tout est généré à partir d'un modèle ce qui fait passer les modèles du statut de contemplatifs à celui de productifs, afin de faire face à la complexité croissante de la conception et de la production d'un logiciel. L'ingénierie dirigée par les modèles (IDM) est une approche de développement logiciel dont le but est d'élever le niveau d'abstraction et d'augmenter le degré d'automatisation dans le développement logiciel.

Visant à automatiser une partie du processus de développement d'un système logiciel, l'IDM requiert un effort d'abstraction plus important de la part des développeurs. En contrepartie, l'IDM promet de conserver le savoir-faire de conception proche des centres de décision grâce aux économies d'échelle dues à l'automatisation [177].
Dans ce chapitre, nous allons présenter les concepts généraux de l'IDM, la différence entre l'IDM et la MDA (*Model Driven Architecture*), nous présentons les langages/outils associés et leurs compositions. Ainsi, nous présentons un bref aperçu des transformations de modèles basées sur la transformation des graphes.

5.2 Ingénierie Dirigée par les Modèles (IDM)

Le Model Driven Engineering (MDE) [178] [179] [180] [181] est une approche de développement, très populaire dans l'industrie logicielle, qui se concentre sur la réalisation de modèles abstraits plutôt que sur des concepts informatiques ou algorithmiques. La phase de spécification est donc particulièrement importante dans une approche MDE et représente une partie conséquente du cycle de développement. Cela permet aux développeurs de se concentrer sur le comportement souhaité du système, sans se soucier de la manière de l'implémenter.
La phase d'implémentation est alors démarrée en fin de cycle, une fois la spécification terminée et validée. La génération partielle de code bas niveau à partir de la spécification permet également de réduire le temps et donc les coûts de développement.

L'idée est de réduire l'activité du concepteur à la construction de modèles métier et d'utiliser des transformations automatiques pour passer d'un modèle à l'autre ou pour générer du code [3]. L'intérêt est de faciliter la génération de code, augmenter la qualité logicielle, réduire les coûts, l'effort de développement, et le temps de commercialisation (time-to-market).
Dans un processus de développement logiciel base sur l'IDM, le logiciel est développé en créant des modèles qui sont transformés successivement en d'autres modèles et éventuellement en code source.
Les initiatives dirigées par les modèles les plus représentatives sont : Model Driven Architecture (MDA), Model Driven Development (MDD) et Model Driven Engineering (MDE), ou en français : l'architecture dirigée par les modèles, le développement dirigé par les modèles et l'ingénierie dirigée par les modèles respectivement.

Les approches modèles proposées par le Model Driven Architecture (MDA) et par l'Ingénierie Dirigée par les Modèles (IDM) sont des approches issues du domaine du génie logiciel. Leur

objectif est de rationaliser et de simplifier les démarches de conception logicielle. Pour cela, elles s'appuient sur les concepts de modèles, de langages de modélisation et de transformations de modèles. L'IDM rassemble notamment des étapes de transformation de modèles et de génération de code. Donc le code réel de l'application n'est plus considéré comme artefact de première importance car il est obtenu par raffinage (transformation) d'un modèle plus abstrait représentant le cœur métier de l'application vers un autre plus concret représentant le système selon une technologie donnée.

Le principal avantage de l'IDM est qu'elle permet de capitaliser les savoirs faire dans des modèles plutôt que dans des programmes codés manuellement. Cela ne signifie pas une rupture avec les techniques de développement existante mais, au contraire, une complémentarité [182]. En effet, à partir des modèles et en utilisant des techniques de transformation de modèles, le code des programmes peut être généré automatiquement ou semi-automatiquement.

5.2.1 Niveaux d'abstraction

Afin de mieux comprendre l'approche IDM, il convient de préciser plus en détail la nature des modèles utilisés tout au long du cycle de vie, les divers usages de ces modèles, ainsi que les possibles intentions du concepteur au moment de leur construction. Ce qui est connu sous le sigle MDA (model Driven Architecture) a proposé une architecture à quatre niveaux qui structure les différents modèles pouvant être produits lors de l'application de l'approche IDM. Cette architecture fait maintenant l'objet d'un consensus [180] [183]. On retrouve ainsi des références à cette architecture dans de nombreux travaux désirant se situer par rapport à l'IDM [184]. Cette architecture comporte quatre niveaux d'abstraction (figure 5.1) :

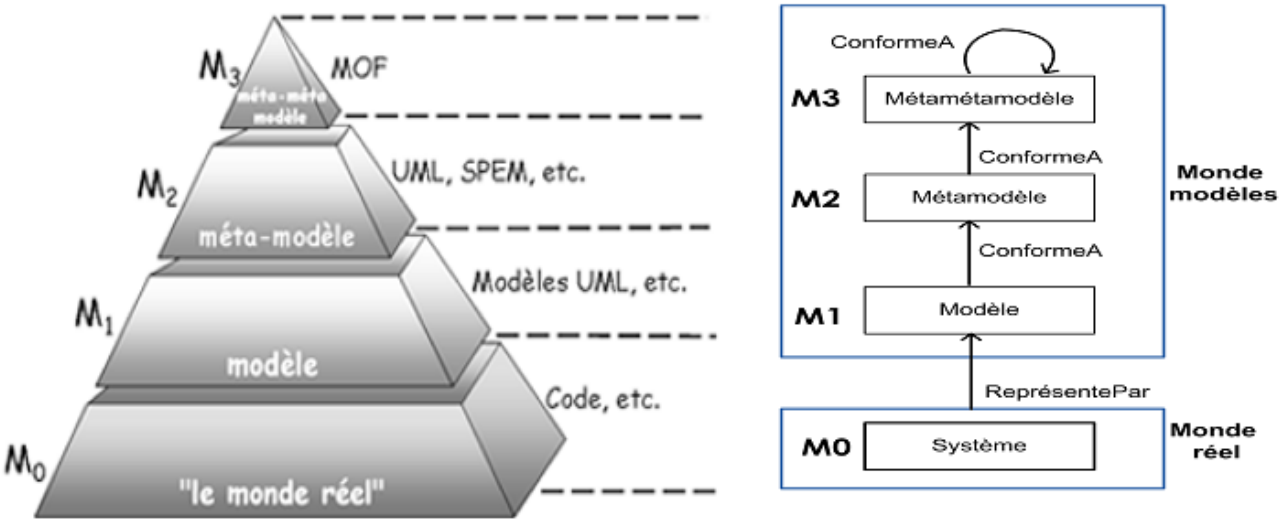


Figure 5.1. Pyramide de modélisation à quatre niveaux.

Le niveau M0. Le niveau M0, représente les objets du monde du réel. Il représente, par exemple, un compte bancaire avec son numéro et son solde actuel.

Le niveau M1. C'est au niveau M1 que les modèles sont édités. Ces modèles sont conformes aux méta-modèles définis au niveau M2. Ainsi, MDA considère que si l'on veut décrire des informations appartenant au niveau M0, il faut d'abord construire un modèle appartenant au niveau M1. De ce fait, un modèle UML (comme le diagramme de classes ou le diagramme d'état/transition) est considéré comme appartenant au niveau M1. Il représenterait des objets manipulés dans le monde réel (décrits au niveau M0).

Le niveau M2. Le niveau M2, est le lieu de définition des méta-modèles. Un *méta-modèle* est un modèle qui définit un langage de modélisation [185]. Un méta-modèle définit précisément les concepts d'un langage de modélisation ainsi que les relations entre ces concepts nécessaires à la description des modèles. Un méta-modèle est écrit dans un langage appelé méta-langage. Un modèle bien formé est *conforme* à son méta-modèle. A partir d'un *modèle bien formé*, il est possible de le transformer en un autre modèle ou bien de générer du code ou de la documentation.
Cependant, la vérification de cette relation de conformité est importante avant toute transformation ou génération. En effet, il est nécessaire de s'assurer qu'un modèle est syntaxiquement et sémantiquement conforme à son méta-modèle avant de produire une application à partir de celui-ci. Les méta-modèles contenus au niveau M2 sont tous des instances du niveau M3 (notons qu'au niveau M3, il ne peut y avoir qu'un seul méta-métamodèle). Dans le cadre de MDA, c'est le méta-modèle d'UML [186] qui est le plus utilisé, celui-ci définit la structure interne des modèles UML.

Le niveau M3. Le niveau supérieur, M3 correspond au Méta-méta-modèle. Il définit les notions de base permettant l'expression des méta-modèles (niveau M2), et des modèles (M1).
Pour éviter la multiplication des niveaux d'abstraction, le niveau M3 est réflexif, c'est-à-dire qu'il se définit par lui-même. Le plus souvent, c'est le méta-méta-modèle MOF (Meta Object Facility) qui est utilisé. Celui-ci est standardisé par l'OMG[8] [185]. Cependant d'autres méta-méta-modèles ont été proposés tels qu'*eCore* défini dans le cadre de *l'Eclipse Modeling Framework*" (EMF) [187] et d'*OWL* (Ontology Web Language) [188]. Des plates-formes de modélisation génériques implémentant le MOF permettent la production de méta-modèles spécifiques à des domaines et de produire ensuite des modeleurs spécifiques à ces domaines. Par exemple, la plateforme de méta-modélisation GME (Generic Modeling Environment) [189] est compatible avec eCore d'EMF [183].

5.2.2 Critères d'un bon modèle

Il n'y a pas de consensus sur la définition des modèles, mais l'une des plus répandues est celle de Minsky [190] : *"pour un observateur B, un objet M(A) est un modèle d'un objet A si B peut utiliser M(A) pour répondre à des questions qu'il se pose sur A."*.
L'utilisateur doit avoir "*confiance"* dans les modèles qu'il manipule. Il doit **s'assurer** de la "qualité" du modèle et être capable de juger des forces et des faiblesses du système modélisé [191].

 Qualité du modèle: Selon l'ISO 8402.94, la qualité est "*l'ensemble des caractéristiques d'une entité qui lui confèrent l'aptitude à satisfaire des besoins exprimés et implicites"*. Pour que le modèle soit efficacement manipulable par des outils informatiques, il doit respecter certaines qualités. Dans [192], Bran Selic avait identifié les caractéristiques essentielles qu'un modèle devrait avoir, ce sont principalement :

• l'**abstraction** : un modèle doit abstraire les détails non pertinents pour un point de vue sur le système qu'il modélise. Le modèle doit assurer un certain niveau d'abstraction et cacher les détails que l'on ne souhaite pas considérer pendant l'étude.

[8] L'OMG est un consortium international, à but non lucratif, pour le développement de standards industriels informatiques. Il a comme membres des entreprises telles que : AT&T, EADS, Ericsson, Hitachi, ou Lockheed Martin. Adresse internet : http ://www.omg.org/

• l**a compréhensibilité :** le modèle doit être compris intuitivement par les personnes qui l'utilisent. Donc, il doit être exprimé dans un formalisme assez compréhensible dont la sémantique est intuitive.
• la **précision** : il doit proposer une représentation du système qui répond de manière réaliste aux questions que l'utilisateur (*observateur* dans la définition de Minsky) se pose sur le système.
• la **complétude** : le modèle contient toute l'information nécessaire qui peut être utilisée pour démontrer ou infirmer sans ambiguïté ni doute la véracité d'une affirmation.
• la **cohérence :** il n'existe pas de contradiction dans l'information contenue dans un ou plusieurs modèles. La cohérence d'un modèle est donc définie par l'impossibilité de trouver deux informations contradictoires dans la spécification des exigences et, entre les exigences spécifiées et les modèles de conception.
• la **pertinence** : Un modèle est pertinent s'il représente le système avec une qualité suffisante. Ce niveau de qualité est relatif selon les langages de modélisation ou l'approche de modélisation choisis. La pertinence nécessite de s'assurer que le modèle répond aux objectifs du modeleur. Ce dernier peut être un utilisateur cible ou un concepteur du système et donc posséder des objectifs différents. L'utilisateur final décrira le système selon une approche boite noire et spécifiera les exigences fonctionnelles et non fonctionnelles du système vis-à-vis de son environnement. Le concepteur adoptera une approche boite blanche pour aboutir à une spécification de conception du système en terme de solution technologique qui soit manipulable sans introduire d'ambiguïté ou de biais (complétude vs. fidélité).
• le **coût réduit :** le modèle doit rester peu coûteux en comparaison aux coûts du développement du système réel.
Cette liste n'est pas exhaustive et elle peut contenir également d'autres caractéristiques telles que la prédiction, la maintenabilité, la traçabilité, l'exécution, etc. Ainsi, l'obtention de modèles qui respectent toutes ces caractéristiques est une tâche non évidente. Dans le cadre du développement logiciel, on s'intéresse de plus en plus à la notion de modèles exécutables et interprétables par les machines. Il nous faut donc pouvoir formaliser la syntaxe d'un modèle, c'est-à-dire définir un langage de modélisation. Pour ce faire, l'IDM propose la notion du méta-modèle.

 De pouvoir juger des forces et des faiblesses du système modélisé. Cela concerne la *performance*, *la stabilité dans le temps* (est-il possible, suite à une situation imprévue, de revenir vers un fonctionnement maîtrisé ou vers une situation de routine [193]) et d'*intégrité* (comment le système anticipe-t-il et évolue-t-il pour s'adapter à de nouvelles situations ?).
Des techniques de validation telles que la simulation et l'expertise sont alors utilisées pour analyser le système en question au travers de ses modèles.

5.2.3 Transformations de modèles

La transformation de modèles est une opération fondamentale dans l'ingénierie dirigée par les modèles. Elle peut être manuelle ou automatisée, mais dans ce dernier cas elle nécessite de la part du développeur qui la conçoit la maîtrise des méta-modèles impliqués dans la transformation.

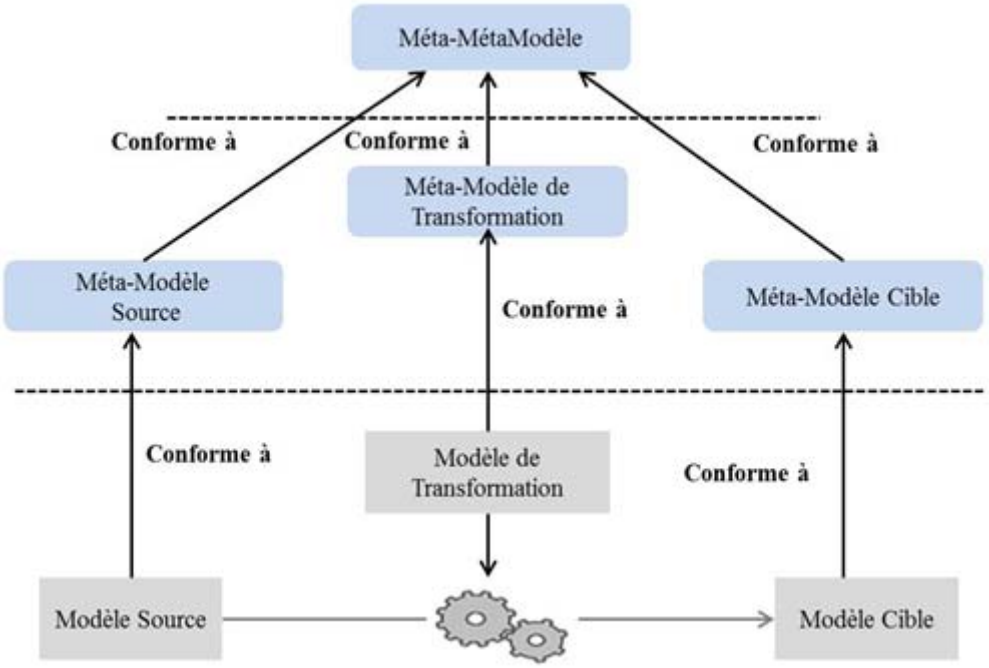


Figure 5.2. Principe d'une transformation de modèles Transformation de modèles.

La transformation de modèles est un processus de conversion d'un ensemble de modèles d'une application donnée à d'autres modèles de la même application [194]. Une transformation de modèles, comme illustré dans la figure 5.2 [195], définit un ensemble de règles pour passer d'un modèle source conforme à un méta-modèle source à un modèle cible conforme à un méta-modèle cible. Ces règles sont définies au niveau du méta-modèle et seront exécutées sur les modèles pour passer d'un modèle à un autre. La transformation de modèles peut être du type transformation de modèle en modèle dans le but de raffiner ou changer (refactoring [196]) le modèle source. Ce type de transformation peut être modélisé par les langages de transformations de graphes tels que QVT (Query/View/Transformation) [197] ou (Atlas Transformation Language) ATL [198]. La transformation de modèles peut aussi être de type transformation de modèle en texte dans le but de générer du code ou de la documentation par exemple.

5.2.3.1 Les approches de l'ingénierie dirigée par les modèles

Selon la classification proposée par Czarnecki et Helsen [199], les transformations de modèles peuvent être partagées en deux grandes classes : les transformations « *Modèle vers Modèle* » et les transformations « *Modèle vers Code* » (figure 5.3).

a) Transformations de type Modèle vers code

On distingue deux approches de transformations de type *modèle vers code* : les approches basées sur le principe du *visiteur (Visitor-based approach)* ou celles basées sur le principe des *patrons (Template-based approach*).

a.1) Les approches basées sur le principe du *visiteur* consistent à traverser le modèle en lui ajoutant des éléments (mécanismes visiteurs) qui réduisent la différence de sémantique entre le modèle et le langage de programmation cible. Le code est obtenu en parcourant le modèle enrichi pour créer un flux de texte.

a.2) Les approches basées sur le principe des *Template* sont actuellement les plus utilisées. Le code cible contient des morceaux de méta-code utilisés pour accéder aux informations du

modèle source. La majorité des outils MDA couramment disponibles supporte ce principe de génération de code à partir de modèle. Parmi les outils basés sur ce principe, on peut citer : *OptimalJ*, *XDE* (qui fournissent la transformation modèle vers modèle aussi), *JET*, *ArcStyler* et *AndroMDA* (un générateur de code qui se repose notamment sur la technologie ouverte *Velocity* pour l'écriture des templates), Acceleo [200] et XPand [201].

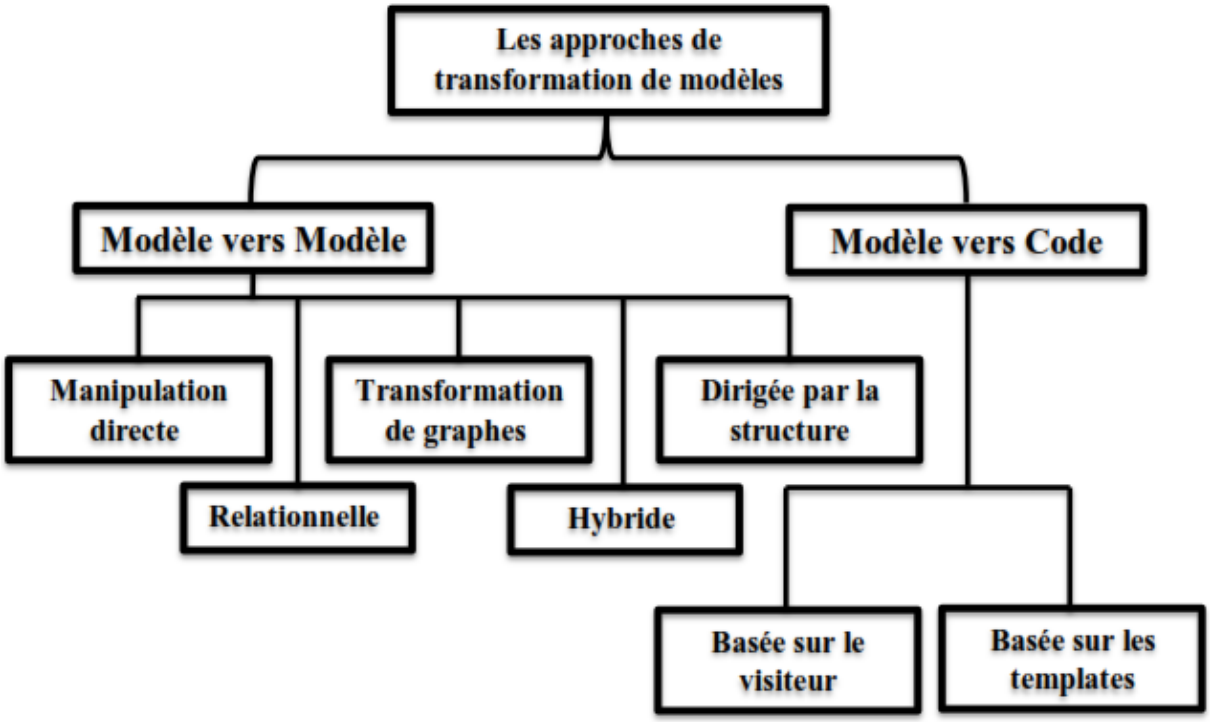


Figure 5.3. Les approches de transformation de modèles [202].

b) Transformations de type modèle vers modèle

Les transformations de type *modèle vers modèle* sont aujourd'hui moins maîtrisées, cependant, elles ont beaucoup évolué au cours de ces dernières années, et plus particulièrement depuis l'apparition du MDA [203].

Les techniques de transformations de ce type peuvent être classées en cinq catégories selon la technique de mise en œuvre et qui sont :

b.1) Approches manipulant directement les modèles : Ces approches sont basées sur l'utilisation d'APIs (Application Programming Interface) pour manipuler directement la représentation interne des modèles. Elles sont en général implémentées comme un *Framework orienté objet*. La spécification des règles de transformation et leur ordonnancement restent à la charge du développeur. L'interface JMI (Java Metadata Interface) est souvent utilisée dans la mise en œuvre de ce type d'approches.

b.2) Approches relationnelles : Pour ces approches, le principe de base consiste à établir une relation entre les éléments des modèles sources et cibles. Ces relations sont spécifiées à l'aide de contraintes. Elles sont purement déclaratives. La programmation logique est particulièrement adaptée pour ce type d'approches et la norme *QVTRelational* [204] est largement utilisée.

b.3) Approches guidées par la structure : Dans ces approches, la transformation se réalise en deux phases : la première crée la structure hiérarchique du modèle cible et la seconde vient

compléter le modèle en définissant les valeurs des attributs et des références. Comme exemples, citons *OptimalJ* [205].

b.4) Approches basées sur la transformation de graphes : Les modèles et les méta-modèles associées possèdent souvent une représentation graphique. Ainsi, les techniques de réécriture de graphes peuvent être appliquées pour réaliser des transformations de graphes.
Les traitements souhaités sont exprimés sous forme d'une grammaire de graphes. Cette catégorie d'approches est mise en œuvre par exemple dans : *GReAT* [206], *AGG* [207] et *AToM*^{3} [208].
Comme les modèles manipulés dans le cadre de cette thèse sont graphiques, nous avons opté pour cette approche. Notre contribution, basé sur ce paradigme, sera détaillé dans chapitre suivant.

b.5) Approches hybrides : C'est des combinaisons des techniques citées précédemment. On peut notamment retrouver des approches utilisant à la fois des règles déclarative et impérative. *ATLAS (ATL)* [209], *Kermeta* [210] et *ModTransf* [211] sont des approches de cette catégorie.

Les transformations opérées sur un modèle source produisent un modèle cible dont le niveau de finesse (ou de détails) est plus ou moins élevé que celui de la source. On distingue les types de transformation suivants [212] :
– une *transformation simple* (1 vers 1) transforme un élément d'un modèle source en un élément d'un modèle cible. Un exemple typique est la transformation d'une classe UML en un document XML qui définit sa structure ou en une table de base de données relationnelle ;
– une *transformation multiple* (M vers N) transforme un ou plusieurs éléments d'un modèle source en un ou plusieurs éléments d'un modèle cible. Comme exemples de transformations multiples nous pouvons citer les transformations de décomposition de modèles (1 vers N) et de fusion de modèles (N vers 1) ;
– une *transformation de mise à jour* parfois appelée *transformation sur place* qui consiste à modifier un modèle par ajout, modification ou suppression d'une partie de ses éléments. Dans ce type de transformation, le modèle source est aussi la cible. Un exemple est la restructuration de modèles (ou *Model Refactoring*) qui consiste à réorganiser les éléments du modèle source afin d'en améliorer sa structure ou sa lisibilité.

5.2.3.2 Les activités de la transformation de modèles

L'objectif principal et final de la transformation de modèles dans le contexte de l'IDM est la génération de code, mais la transformation de modèles couvre plusieurs activités qui sont [213] :

Extraction de modèle
L'extraction de modèle est une transformation de modèle exogène et verticale.
L'extraction est l'inverse de la génération de code et c'est l'une des activités essentielles dans l'ingénierie inverse et de la compréhension du programme. Elle permet de construire un modèle visuel à un niveau d'abstraction supérieur qui permettra de connaître la structure du code.

Traduction de modèle
La traduction de modèle est une transformation de modèle de type horizontale et exogène qui est utilisée dans la recherche académique. Son objectif est de transformer un modèle représenté par un langage de modélisation en un modèle d'un autre langage de modélisation.

Simulation et exécution de modèle
La simulation permet d'avoir une vision sur l'exécution d'un modèle sans l'avoir exécuté réellement. Les simulations sont faites pour des raisons de coûts, de temps et de ressources.

Vérification de modèle
La vérification d'un modèle est le synonyme de modèle correct, c'est à dire le vérifier syntaxiquement et sémantiquement. L'analyse syntaxique vérifie si le modèle est conforme à un métamodèle et s'il respecte les conditions "contraintes" imposées par le langage tandis que l'analyse sémantique vérifie les propriétés dynamiques d'un domaine.
Validation de modèle
La validation est de vérifier si un modèle répond aux exigences, ou d'une autre façon de vérifier si un modèle est fiable. La simulation et le test de modèle sont utilisés pour vérifier si une modèle est valide.
Amélioration de la qualité de modèle
Un type particulier de l'évolution du modèle pour lequel les transformations sont particulièrement utiles est l'amélioration de la qualité du modèle. Les modèles peuvent avoir différents types de critères de qualité qui doivent être satisfaits, mais cette qualité a tendance à se dégrader au fil du temps en raison de nombreux changements qui sont apportés au modèle au cours de sa vie.
Migration et coévolution de modèle
Les transformations de modèles sont également essentielles pour faire face à l'inévitable évolution des modèles. Dans ce contexte, un problème supplémentaire se pose : non seulement les modèles évoluent, mais aussi les langages de modélisation dans lesquels ces modèles sont exprimés. Avec tout changement dans le langage de modélisation (par exemple, une nouvelle version d'UML est présentée), les concepteurs du modèle sont confrontés à la nécessité de migrer leurs modèles pour cette nouvelle version, ou courir le risque que leurs modèles deviennent obsolètes ou incompatibles.
La gestion d'incohérence de modèle
La gestion d'incompatibilité de modèle est également bien adaptée pour être prise en charge par la transformation de modèles. En raison du fait que les modèles sont généralement exprimés en utilisant les points de vue multiples, ils sont en évolution constante, et sont souvent développés dans un cadre de collaboration. Donc les incohérences dans les modèles ne peuvent pas être évitées. Par conséquent, nous avons besoin de techniques basées sur la transformation de modèle pour réparer les incohérences.

5.2.3.3 Propriétés d'une transformation de modèles

Une transformation est généralement caractérisée par l'ensemble de ses règles de transformation, leur ordonnancement, leur organisation, la relation entre les deux modèles source et cible, la traçabilité et la direction [213].

o **Les règles de transformation :** Une règle de transformation est partagée en deux parties: une partie gauche (LHS *Left Hand Side*) accédant au modèle source, et une autre partie droite (RHS *Right Hand Side*) qui accède au modèle cible. La partie logique (déclarative ou impérative) de la règle comporte les calculs à effectuer sur les modèles ainsi que les contraintes appliquées. Une logique déclarative spécifie les relations entre les éléments des modèles source et cible. Une logique impérative utilise généralement des langages de programmation pour manipuler les éléments des modèles sur des interfaces dédiées (exemple JMI).

o **Les spécifications :** Les spécifications représentent des relations non-exécutables, mais peuvent exceptionnellement représenter une fonction entre les modèles source et cible et deviennent, dans ce cas, exécutables. Certaines approches fournissent des mécanismes de spécifications dédiés, tels que les pré-conditions et les post-conditions en OCL (*Object Constraint Language*).

o **La relation entre le modèle source et le modèle cible :** La relation entre le modèle source et le modèle cible dépend du type de la transformation. Nous parlons de transformation endogène lorsque le modèle source et le modèle cible sont exprimés dans le même formalisme. La transformation est exogène lorsque les deux modèles sont exprimés dans deux formalismes différents.

o **Traçabilité** : La traçabilité est la propriété d'avoir un dossier de liens entre les éléments de modèle source et ceux du modèle cible ainsi que les différentes étapes du processus de transformation. Les liens de traçabilité peuvent être stockés soit dans le modèle source, soit dans le modèle cible ou dans un modèle à part.

o **Directivité ou réversibilité** : une transformation est dite réversible si elle peut se faire dans les deux sens. Exemple : « Model to text » et « text to Model ». On dit en outre qu'une transformation est réversible s'il existe une transformation permettant de retrouver le modèle source à partir du modèle cible.

o **L'organisation des règles :** C'est une organisation qui définit la stratégie selon laquelle les règles seront appliquées. Ces règles peuvent être organisées de façon modulaire avec importation. Les règles peuvent utiliser le principe de réutilisation en passant par le mécanisme d'héritage entre les règles, ou la composition en passant par l'ordonnancement explicite. Les règles peuvent être organisées aussi selon une structure dépendante du modèle source ou du modèle cible.

o **Réutilisabilité** : la Réutilisabilité peut être mesurée avec la possibilité d'adapter et de réutiliser les règles d'une transformation de modèle dans d'autres transformations. L'identification de patrons de transformation est un moyen pour mettre en œuvre cette réutilisabilité.

o **Ordonnancement** : l'ordonnancement consiste à représenter la suite des règles à exécuter lors d'une transformation. En effet, les règles de transformation peuvent déclencher d'autres règles.

o **Modularité** : une transformation modulaire permet de regrouper les règles de transformation en faisant un découpage du problème. Un langage de transformation de modèles supportant la modularité facilite la réutilisation des règles de transformation.

o **L'incrémentalité :** Cette propriété est liée à l'aptitude des modèles cibles à s'adapter aux changements des modèles sources.

5.2.3.4 Les avantages de l'IDM

L'avantage visé par l'IDM est l'augmentation de la vitesse du développement logiciel grâce à l'automatisation : le code source peut être généré à partir des modèles en utilisant une ou plusieurs étapes de transformation. Ceci permet de réaliser un logiciel beaucoup plus rapidement. La durée allouée au développement ainsi qu'à la validation du produit logiciel en est donc raccourcie [214] [192] [215]. Le site web de l'OMG [216] expose plusieurs cas de projets logiciels qui se sont basés sur une architecture dirigée par les modèles. L'entreprise ABB [217] est un cas réussi parmi d'autres qui, selon Andreas Blaszczyk, de chez ABB a résumé: "*The code quality produced by the ArcStyler's model-based generation is consistent and clean, reducing the number of programmers and test engineers required to develop and*

maintain the test environments by approx. 45% compared to a conventional development approach.". Dans ce contexte, les travaux de Parastoo *et al.,* [218] [218] résument les avantages de l'IDM en focalisant sur la qualité des modèles :

 Traçabilité et synchronisation entre les modèles et le code source : dans une approche traditionnelle de développement logiciel, les modèles sont réalisés en marge de leur implémentation et le code source est généré manuellement, ce qui est demandant en temps. Ceci pose un problème majeur lors des modifications dans le code. En effet, elles sont rarement accompagnées d'une mise à jour dans les modèles [214]. L'IDM dans ce sens améliore la traçabilité et la synchronisation entre les modèles.
 Meilleure réutilisation des modèles : une fois définie, l'architecture du logiciel et les modèles peuvent être réutilisés dans d'autres systèmes [220].
 Logiciel de qualité : le code source généré à partir des modèles est, en principe, sans erreur parce que les modèles ont été testés et validés (ex. architecture JEE, .NET. etc). Par conséquent, la qualité du logiciel devient dépendante de l'implémentation des concepts du domaine d'affaires plutôt que des éléments de l'architecture logicielle [192].
 Séparation des préoccupations : la réutilisation des modèles offre l'avantage de focaliser davantage sur les concepts liés aux domaines d'affaires plutôt qu'aux concepts liés à leur implémentation [214] [192].

En résumé, l'IDM vise à accroître la productivité du développement logiciel. Cependant, il est important de mentionner que l'IDM reste une approche complexe qui sera maitrisée que par des spécialistes en architecture de logiciel. De plus, la difficulté de maintenir la cohérence entre les différents modèles représente un défi majeur.

5.3 L'approche MDA (Model-Driven Architecture)

Pour répondre à la difficulté croissante de concevoir et analyser des systèmes de plus en plus complexes et à une palette de plus en plus large de technologies, l'Object Management Group (OMG) a introduit l'Architecture Dirigée par les Modèles (MDA) [221]. Celle-ci peut être vue comme restriction de l'IDM à la gestion de l'aspect particulier de la dépendance d'un logiciel à une plateforme d'exécution. MDA place le modèle au centre du processus de production à travers des techniques de manipulation et de transformation. L'OMG a défini l'architecture MDA (*Model-Driven Architecture*) en 2000 [203] pour promulguer de bonnes pratiques de modélisation et exploiter pleinement les avantages des modèles. L'architecture dirigée par les modèles (MDA) est basée principalement autour des modèles qui sont exprimés dans un langage de modélisation dont le métamodèle est exprimé en MOF (Meta-Object Facility [185]). L'OMG préconise également l'UML pour la modélisation.

L'Architecture Dirigée par les Modèles permet de décrire séparément les spécifications fonctionnelles d'un système et son implémentation sur une plate-forme donnée. Partant de l'idée de séparation des préoccupations, MDA vise trois objectifs fondamentaux : la portabilité, l'interopérabilité et la réutilisabilité. Dans ce contexte, le MDA est considéré comme une approche spécifique de l'IDM, basée sur les problématiques suivantes :
• Spécification d'un système indépendamment de la plateforme sur laquelle ce système doit être déployé,
• Spécification de plateformes,
• Définition d'une plateforme pour un système,
• Transformation des spécifications d'un système en un système associé à une plateforme particulière.

Les termes « *plateforme* » et « *plateforme indépendante* » sont intensément utilisés dans la littérature abordant l'IDM. La notion de *plateforme indépendante* fait référence à la capacité d'être indépendant des spécificités d'une plateforme. La technique la plus fréquemment utilisée pour être indépendant d'une plateforme est d'utiliser une technologie neutre représentée par une machine virtuelle. Une machine virtuelle peut aussi être considérée comme étant une plateforme. Tout modèle visant une machine virtuelle est alors considéré comme étant spécifique à cette plateforme, mais indépendant de la plateforme sous-jacente.

5.3.1 Standards et espaces techniques

L'architecture du MDA se décompose en quatre couches (figure 5.4). Au centre, il existe le standard UML (Unified Modeling Language), MOF (Meta-Object Facility) et CWM (Common Warehouse Metamodel). La deuxième couche contient le standard XMI (XML Metadata Interchange) permettant le dialogue entre les middlewares (Java, CORBA, .NET, et web services). Dans la couche suivante, se trouve les services permettant de gérer les évènements, la sécurité des transactions, et les répertoires. La dernière couche propose des frameworks spécifiques au domaine d'application (Télécommunication, médecine, commerce électronique, finance, etc.) Un concepteur pour créer sa propre application peut utiliser UML comme il peut utiliser d'autres langages. En partant du centre de la figure 5.4, le concepteur dirigera son application en évoluant de couche en couche pour aller vers le domaine d'application qui l'intéresse.

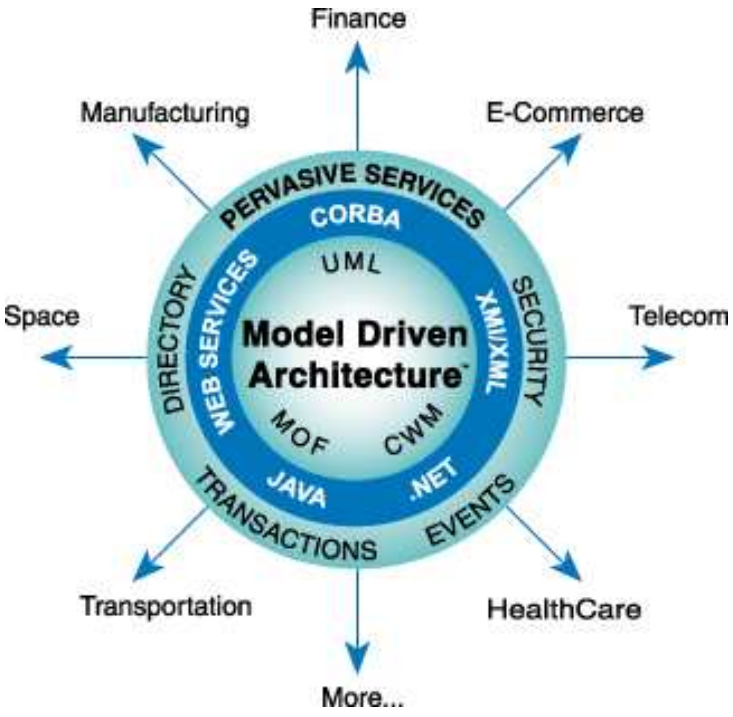


Figure 5.4. Couches de spécifications du MDA.

Donc selon cette architecture indépendante du contexte technique, MDA propose de structurer les besoins avant de se livrer à une transformation de cette modélisation fonctionnelle en modélisation technique, tout en testant chaque modèle produit [222]. Il s'agit de modéliser l'application que l'on veut créer de manière indépendante de l'implémentation cible (niveau matériel ou logiciel). Ceci permet une grande réutilisation des modèles [223]. MDA est considéré une démarche ayant l'ambition de proposer une vision le plus large possible du cycle

de vie du logiciel, ne se contentant pas uniquement de sa production. De plus, cette vision globale se veut décrite dans une syntaxe unifiée.
Un des postulats sous-jacents du MDA est que l'opérationnalisation d'un modèle abstrait n'est pas un problème trivial. Un des bénéfices du MDA est de résoudre ce problème [224]. Le MDA propose de concevoir une application au travers d'une chaîne logicielle se déclinant en quatre phases dans l'objectif d'implémentation flexible, d'intégration, de maintenance et de test :
–L'élaboration d'un modèle sans préoccupation informatique (CIM : Computer Independant Model).
–Sa transformation manuelle en un modèle dans un contexte technologique particulier (PIM : Platform Independant Model) ;
–Sa transformation automatique en un modèle associé à la plate-forme de réalisation cible (PSM : Platform Specific Model), modèle qui doit être raffiné ;
–Sa réalisation dans la plate-forme cible.

5.3.2 Types de modèles dans MDA

La MDA définit une approche qui facilite la conception en séparant les préoccupations que sont la spécification fonctionnelle (comportement du système modélisé) et l'implémentation (langage de programmation, composants logiciels). De cette manière, la validation fonctionnelle est réalisée indépendamment des choix d'implémentation. Une même spécification fonctionnelle peut ainsi être implantée sur différentes plates-formes d'exécution en étant traduite en différentes spécifications d'implémentations. On distingue ainsi les modèles indépendants de la plate-forme d'exécution (PIM : Platform Independant Model) et les modèles spécifiques à la plate-forme d'exécution (PSM : Platform Specific Model). La finalité de la MDA est de partir d'un PIM défini par l'utilisateur

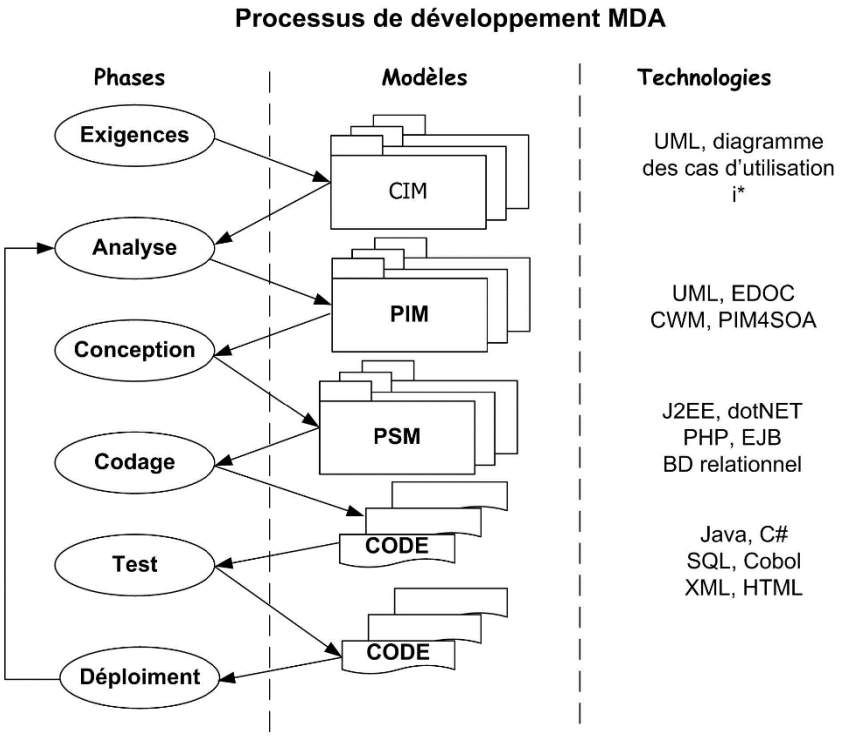


Figure 5.5. Processus de développement selon l'approche MDA.

Le principe clé et initial de MDA consiste à s'appuyer sur le standard UML pour décrire séparément des modèles pour les différentes phases du cycle de développement d'une application. Plus précisément, le MDA préconise l'élaboration de modèles (figure 5.5) :

— *d'exigence (Computation Independent Model – CIM)* : le modèle CIM spécifie la fonctionnalité (ou le comportement extérieur) d'un système sans montrer de détails de construction. Le niveau CIM joue un rôle important pour éliminer la brèche entre les experts d'analyse du domaine (à quoi sert le système) et les experts du design et mettre en œuvre le système (ou experts de TI),
— *d'analyse et de conception (Platform Independent Model – PIM)* : le modèle PIM montre les détails spécifiques du système de manière indépendante de la plateforme. Ces modèles peuvent être utilisés pour dériver des modèles liés aux différents types de plate-forme où la base conceptuelle est la même.
— *d'implémentation (Platform Specific Model – PSM)* : le modèle PSM combine les spécifications dans le modèle PIM avec les détails qui décrivent comment le système utilise un type particulier de plateforme. Notons qu'un modèle PSM peut être dérivé à partir d'un ou plusieurs modèles PIM (chaque modèle PIM a un but conceptuel différent).

L'objectif majeur du MDA est l'élaboration de modèles pérennes (PIM), indépendants des détails techniques des plates-formes d'exécution (J2EE, .Net, PHP, etc.), afin de permettre la génération automatique de la totalité des modèles de code (PSM) et d'obtenir un gain significatif de productivité.

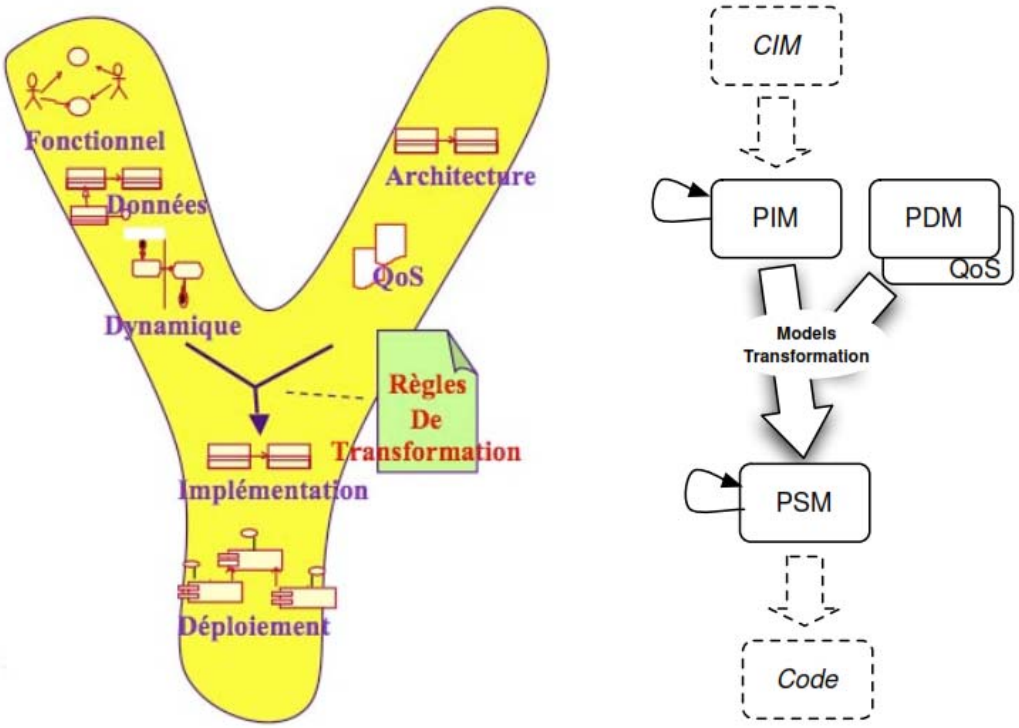


Figure 5.6. MDA : Un processus en Y dirigée par les modèles [177].

Le passage de PIM à PSM fait intervenir des mécanismes de composition et de transformation de modèle avec un modèle de description de la plate-forme (Platform Description Model – PDM), un modèle de description de la qualité de service (Quality of Service – QoS) ou autres modèles comme les non-fonctionnels (par exemple, la sécurité).
Cette démarche s'organise donc selon un cycle de développement « en Y » propre au MDD (Model Driven Development) (figure 5.6).

5.3.3 Passage entre modèles

Passer progressivement des PIM aux PSM pour préparer et faciliter la génération de code vers la plate-forme technique choisie, constitue l'idée du MDA. Ce passage des PIM aux PSM est une transformation de modèles [225]. Dans le cycle de vie et de développement d'un produit, le MDA identifie différentes catégories de transformations de modèles.

• ***PIM vers PIM (raffinement):*** Cette transformation constitue un enrichissement et une spécialisation du modèle, en y apportant des informations indépendantes des spécificités d'une technologie. La description d'un modèle de répartition, de persistance des données ou de composants peut être vue comme un raffinement.
• ***PIM vers PSM (projection*)**: C'est la traduction du modèle générique, vers une plateforme d'exécution.
• ***PSM vers PSM (réalisation*)**: C'est la mise en œuvre concrète d'un modèle générique sur une plate-forme d'exécution. Elle consiste en l'ensemble des phases qui mènent à un logiciel exécutable, telles que la génération de code source, la compilation, le déploiement, l'instanciation et l'initialisation des composants logiciels.
• ***PSM vers PIM (reverse-engineering*)**: Cette transformation permet l'élaboration du modèle générique à partir de l'implémentation existante d'un logiciel. En théorie, cette transformation est censée fournir un modèle générique décrivant l'application à partir d'une base de code accessible. En pratique, il est très complexe d'automatiser entièrement ce processus pour tout modèle PSM.

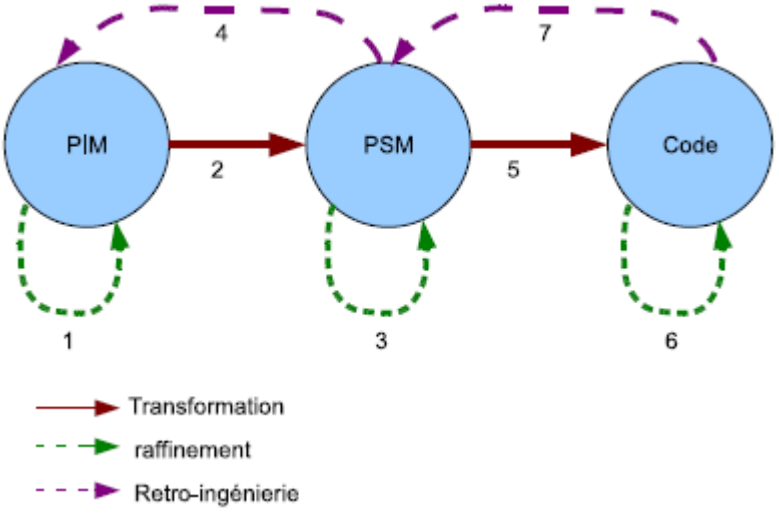


Figure 5.7. Les modèles et les transformations dans l'approche MDA.

Génération de code : dans la pratique, certains travaux font la distinction entre les PSM exécutables (ou code) et les PSM non exécutables, mais la génération de code n'est pas toujours considérée comme une transformation de modèles. La figure 5.7 montre quand même qu'il est possible de passer d'un PSM non exécutable à du code et inversement. Il est important de souligner que le passage du code au PSM est une opération de rétro-ingénierie qui est assez complexe à réaliser. Si le code n'a pas été conçu dans la démarche du MDA, il faut faire appel aux techniques traditionnelles de rétro-ingénierie pour effectuer de telles opérations. En somme, il est possible de classer les transformations de modèles possibles dans le MDA dans quatre catégories comme l'indique la figure 5.7 :
– les transformations (2) : décrivent le processus de conversion d'un PIM en un PSM ;
– les raffinements (1), (3) et (6) : introduisent ou suppriment des informations dans un modèle;

– les retro-ingénieries (4) et (7) : convertissent un modèle vers un niveau d'abstraction plus élevé ;
– la génération de code (5) : transforme un PSM non exécutable en un code exécutable. Nous avons déjà mentionné que la génération de code n'est pas toujours considérée comme une transformation de modèles dans la pratique.

5.3.4 Types de transformations de modèles dans MDA

La transformation de modèles permet de rendre les modèles productifs et d'obtenir une tracabilité entre des modèles très abstraits, proches des besoins exprimés par les utilisateurs, et des modèles très concrets, proches des plate-formes d'exécutions. Une transformation de modèles s'apparente toujours à une fonction qui prend en entrée un ensemble de modèles et qui fournit en sortie un ensemble de modèles. Les métamodèles permettent dans le processus de transformation MDA de définir les structurations possibles des modèles source et cible et de servir de base pour la définition des règles de transformation.

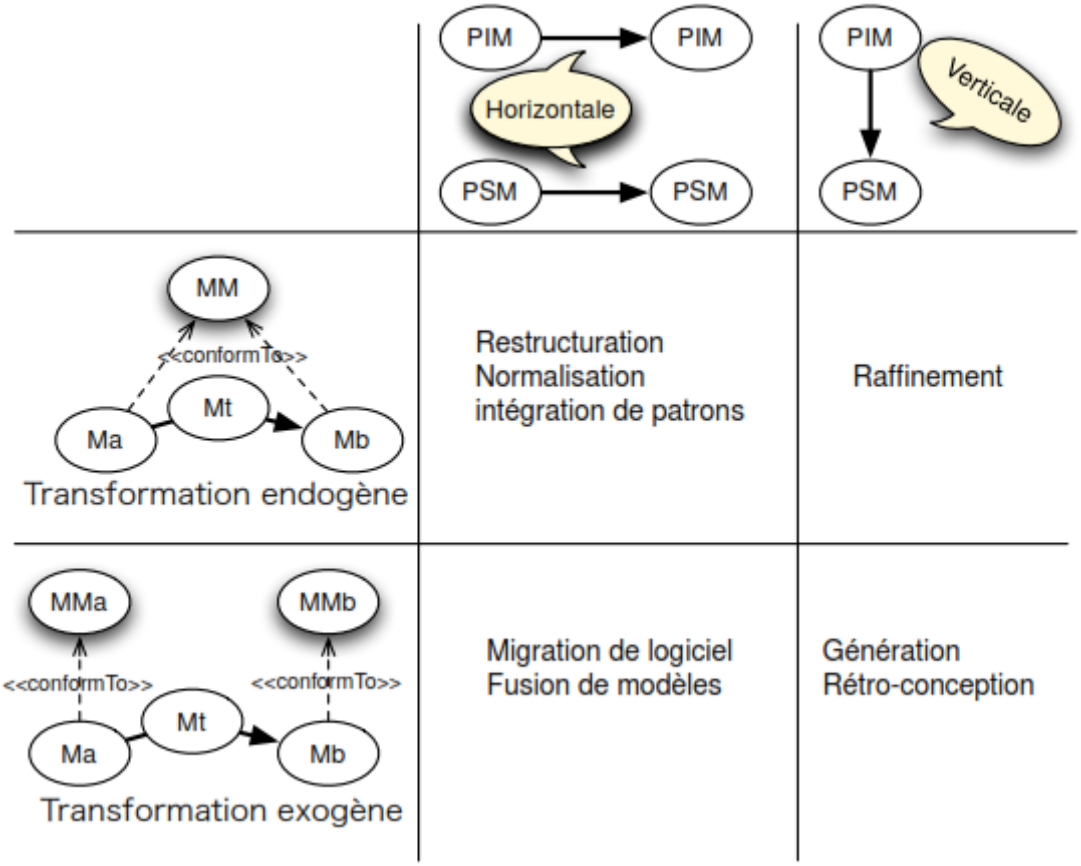


Figure 5.8. Types de transformation et leurs principales utilisations [226].

Un facteur important à prendre en considération dans les transformations de modèles est le niveau d'abstraction. Selon ce facteur, nous pouvons distinguer trois types de transformations (figure 5.8) [227] :

- ***Transformations horizontales:*** Ces transformations gardent le même niveau d'abstraction en modifiant les représentations du modèle source (ajout, modification, suppression ou restructuration d'informations).
- ***Transformations verticales:*** La source et la cible d'une transformation verticale sont définies à différents niveaux d'abstraction. Un raffinement fait référence à une

transformation qui baisse le niveau d'abstraction. Tandis qu'une abstraction désigne une transformation qui élève le niveau d'abstraction.

- ***Transformations obliques:*** Ces transformations sont généralement utilisées par les compilateurs qui optimisent le code source avant la génération du code exécutable. Elles sont le résultat de la combinaison des deux premiers types de transformations.

Selon la nature des méta-modèles sources et cibles, nous distinguons deux types de transformations :

- ***Transformations endogènes*:** La transformation de modèles est qualifiée d'endogène si les modèles sources et cibles sont conformes au même méta-modèle.
- ***Transformations exogènes :*** la transformation de modèles est dite exogène si elle se fait entre deux méta-modèles (source et cible) différents.

5.3.5 Technologies

Afin de contrôler la prolifération des technologies, l'OMG définit un cadre de pratiques de l'IDM autour d'une multitude de standards comme le MOF, XMI, OCL, UML, CWM, SPEM, QVT, etc. La plupart des approches de mise en œuvre, aussi bien dans le monde industriel qu'académique, tentent de s'aligner sur ces standards. D'un côté, Microsoft s'appuie sur les technologies XML et l'approche DSL (Domain Specific Language) pour une mise en œuvre à travers les usines logicielles. D'un autre côté, IBM propose un scénario complet allant de la définition des méta modèles jusqu'à la génération de code, et qui s'articule autour du canevas EMF (Eclipse Modeling Framework) et la plateforme Eclipse. Parmi les standards fondamentaux de l'OMG liés à l'ADM nous pouvons citer :

5.3.5.1 Méta Object Facility (MOF)

Malgré son succès, il était clair qu'UML possédait des limites et que d'autres méta-modèles adressant des besoins différents seraient nécessaires. Afin d'éviter une prolifération de métamodèles incompatibles évoluant indépendamment, il était nécessaire de mettre en place un cadre permettant de formaliser la définition des méta-modèles [176]. La solution fut de proposer un langage pour définir des méta-modèles, c'est-à-dire un méta-méta-modèle, appelé Meta-Object Facility (MOF) [228].

Dans une architecture MDA, le MOF occupe une place centrale. Le MOF est le langage standard de plus haut niveau d'abstraction dans une architecture à quatre niveaux (sommet de l'architecture MDA, figure 5.1). Deux langages de niveau deux sont conformes au MOF. Le premier langage est UML 2.x qui permet de créer des méta modèles à partir d'un méta modèle standard, par spécialisation, ou par extension, via les profils UML. Le deuxième langage est *Ecore*, qui est au coeur de la technologie *Eclipse Modeling Framework* (EMF). Ecore permet de définir des méta modèles à partir d'une spécification minimale du MOF : *Essential MOF*.

5.3.5.2 XMI (XML Metadata Interchange)

Le standard XMI permet de représenter les modèles sous forme de documents XML. En fait, MDA définit deux façons différentes de représenter les modèles : soit sous forme de documents textuels, soit sous forme d'objets de programmation.

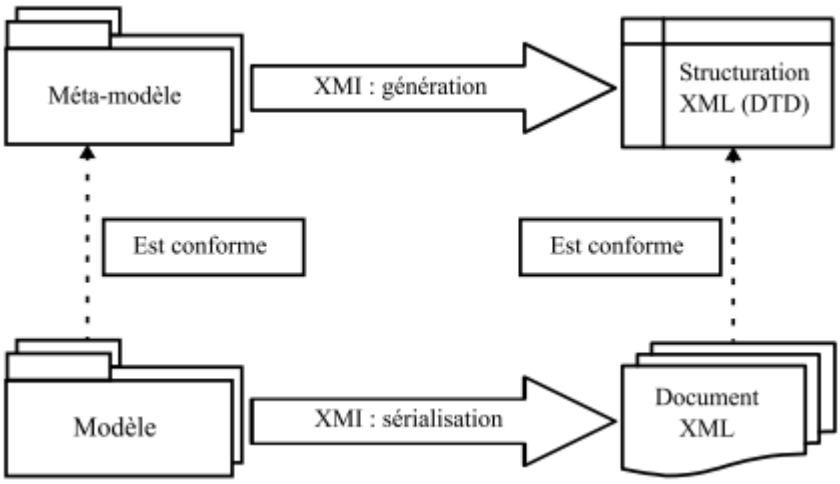


Figure 5.9. XMI et la structuration de balises XML [229].

La représentation textuelle est adaptée pour le stockage des modèles sur des mémoires de masse ou l'échange des modèles entre applications. La représentation objet est plus indiquée pour les transformations, exécutions, et validations des modèles. En plus de XMI, MDA offre deux autres standards pour la représentation des modèles : JMI (Java Metadata Interface), et EMF (Eclipse Modeling Framework). Ces standards définissent la façon de représenter les modèles sous forme d'objets Java. Le principe de fonctionnement de XMI, JMI et EMF est de générer automatiquement la structure des formats de représentation des modèles à partir de leur méta-modèle. Ces standards tirent partie de l'analogie qui existe avec la relation entre modèle et métamodèle, entre un document XML et sa DTD (figure 5.9) ou entre objet et classe.

5.3.5.3 CWM (Common Warehouse Metamodel)

CWM *(Common Warehouse Metamodel)* [230] est une interface basée sur UML, MOF et XMI pour faciliter l'échange de métadonnées entre outils, plateformes et bibliothèques de métadonnées dans un environnement hétérogène.
C'est le standard de l'OMG pour les techniques liées aux entrepôts de données. Il couvre le cycle de vie complet de modélisation, construction et gestion des entrepôts de données. CWM définit un métamodèle qui représente les méta-données aussi bien au niveau métier qu'au niveau technique. CWM définit les métamodèles des principaux types d'entrepôts de données (Relationnel, Objet, XML, ...) et propose des règles de transformation entre ceux-ci. Les méta-modèles de données permettent de modéliser des ressources comme les bases de données relationnelles et les bases de données orientées objet.

5.3.5.4 QVT (Query/View/Transformation)

Comme les transformations de modèles sont un des piliers de l'IDM, l'approche MDA se devait de proposer une façon standardisée pour les spécifier. Le standard Query/View/Transformation (QVT) [231] joue ce rôle. Ce standard doit permettre l'interrogation des modèles afin de sélectionner des éléments sources (Query). Cette interrogation demande une capacité de filtrage de modèles pour en extraire une partie spécifique (View), avant de pouvoir transformer les éléments sélectionnés (Transformation). Il définit trois langages permettant d'exprimer des transformations de modèles à modèles. Le métamodèle de QVT est conforme à MOF et OCL est utilisé pour la navigation dans les modèles. Le métamodèle fait apparaître trois sous-langages pour la transformation de modèles, caractérisés par le paradigme mis en œuvre pour la définition des transformations (déclaratif, impératif et hybride). D'après la figure 5.10, Dans sa version actuelle, le standard QVT présente un caractère hybride en supportant trois langages de transformation. La partie déclarative de QVT est définie par deux langages de niveaux

d'abstraction différents : *QVT-Relations* et *QVT-Core*. Le langage *QVT-Relations* est un langage orienté utilisateur permettant de définir des transformations à un niveau d'abstraction élevé. Il a une syntaxe textuelle et graphique. Le langage *QVT-Core* forme l'infrastructure de base pour la partie déclarative, c'est un langage technique de bas niveau, défini par une syntaxe textuelle. Ce langage sert à spécifier la sémantique du langage *QVT-Relations,* donnée sous la forme d'une transformation *Relations2Core.* Un troisième langage appelé *Operational Mappings* et un mécanisme d'invocation de fonctionnalités de transformation implémentées dans un langage arbitraire (boîte noire ou black box). *Operational Mappings* étend Relations avec des constructions impératives et des constructions OCL avec effets de bord. Le mécanisme d'extension appelé *QVT-BlackBox* permet de décrire des fonctionnalités de transformations dans un langage externe.

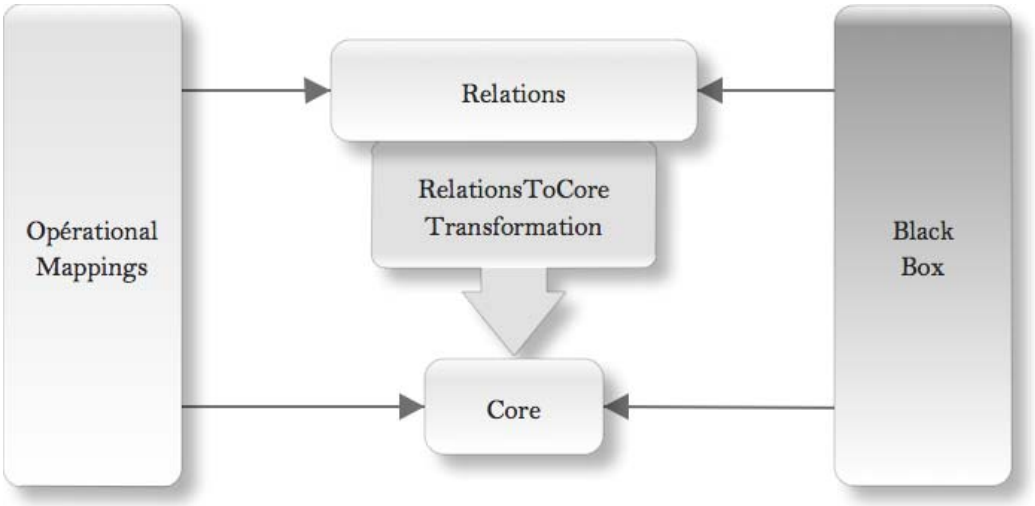


Figure 5.10. Architecture QVT des langages de spécification de modèles [204].

QVT-Relations offre aux utilisateurs la possibilité de spécifier des relations en utilisant une notation graphique, naturelle et intuitive. Cette caractéristique permet de diminuer la complexité de description des transformations de grande taille, et garantit une meilleure vision de l'ensemble des relations impliquées dans une transformation. Cette notation graphique est complémentaire de la notation textuelle. Elle permet de spécifier une relation reliant deux ou plusieurs patrons de domaine. Chaque patron est composé d'une collection d'objets, de liens et de valeurs. La structure d'un patron est exprimée à l'aide d'une notation similaire aux diagrammes d'objets d'UML. Nous illustrons cette notation graphique à travers l'exemple de la transformation UML2Rel présentée en détail dans [204], en rappelons qu'une règle est l'unité de structuration dans les langages de transformation de modèle.

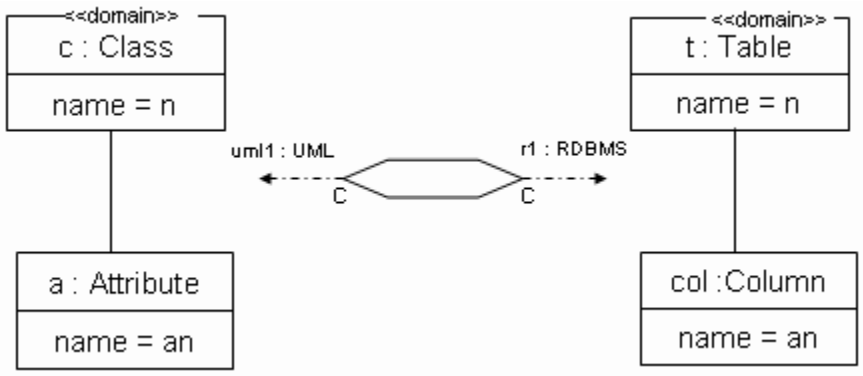


Figure 5.11. Représentation graphique de la relation UML2Rel '1ère version'.

La Figure 5.11 spécifie la relation UML2Rel. Le patron du domaine source est composé d'une classe reliée à un attribut, et le patron du domaine de sortie est composé d'une table et d'une colonne. Les déclarations 'uml1 : UML' et 'r1 : RDBMS' de chaque côté de la transformation indiquent que c'est une relation entre deux modèles candidats 'uml1' et 'r1' avec leurs métamodèles respectifs 'UML' et 'RDBMS'. Les lettres qui apparaissent sous chaque membre du symbole de la relation indiquent le mode d'exécution pour chaque domaine impliqué dans la relation ('C' pour 'checkonly').
Plusieurs constructions du langage QVT-Relational peuvent être représentées graphiquement. On peut représenter par exemple des relations avec des clauses *'Where'* ou *'When'*, en utilisant des cadres réservés à ces deux clauses (Figure 5.12). Dans cet exemple, on remarque que cette version de la relation *'UML2Rel'* étend la relation précédente pour indiquer que la relation *'AttributeToColumn'* est appliquée pour chaque attribut appartenant à une classe. De même, cette représentation graphique supporte la spécification de contraintes sur les objets ou même sur le patron de la relation. Comme dans le langage UML, on représente les contraintes graphiquement en attachant une note en OCL. Dans la Figure 5.12, une contrainte est attachée à l'objet *col* et une autre au patron UML. On remarque que le membre droit de la relation UML2Rel est marqué avec la lettre 'e', impliquant que le domaine 'RDBMS' est noté comme 'enforceable'.

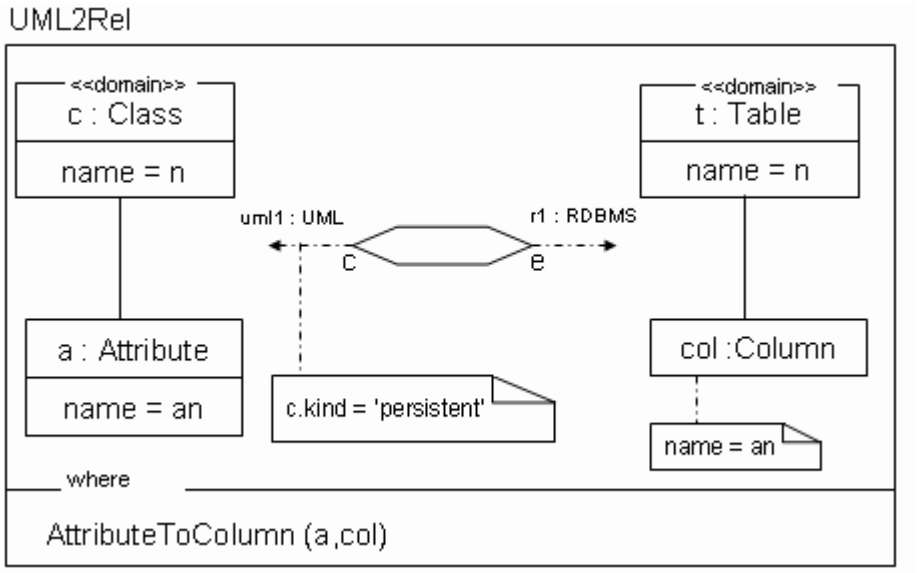


Figure 5.12. Représentation graphique de la relation UML2Rel '2ème version'.

Le standard QVT propose des solutions intéressantes pour la transformation de modèle. Il propose plusieurs styles de programmation (impératif, déclaratif, procédural, etc.) qui peuvent être combinés et utilisés par les outils compatibles QVT. Ceci laisse aux développeurs la liberté de choisir le type de paradigme le plus approprié pour chaque problème de transformation. Mais cette diversité des langages proposés par QVT peut être une source de problèmes pour les utilisateurs qui sont confrontés à plusieurs possibilités dans le développement des transformations. De plus, les outils ont tendance à n'implémenter qu'un sous-ensemble de la spécification. Ceci est confirmé par les premières implémentations de QVT comme QVT-Operational de Borland intégré dans l'outil commercial *Together*, le prototype de QVT-Relations appelé *ModelMorf* [232] de TCS (Tata Consulting Services) et l'outil opensource *SmartQVT* [233] qui est aussi une implémentation de QVT-Operational développé par France Telecom R&D.

5.4 Autres approches centrées sur les modèles

Il est à noter que l'approche ADM, bien qu'elle soit la plus importante, elle n'est pas la seule à concentrer son activité sur l'utilisation des modèles. Nous pouvons trouver l'approche CASE (Case Aided Software Engineering), l'approche MIC (Model-Integrated Computing) [234], les Software Factories [235], etc, qui concentrent également leurs activités sur les modèles.

5.4.1 Computer Aided Software Engineering (CASE)

Apparue dans les années 80, l'idée initiale de l'approche *CASE* (Computer Aided Software Engineering) était de s'inspirer des outils de type *CAD* (Computer Aided Design), qui existaient déjà pour la conception des systèmes mécaniques notamment, afin d'outiller le génie logiciel. Le terme « *CASE* » [236] a été déposé en 1982 par la Nastec Corporation of Southfield, Michigan avec leur éditeur *GraphiText*, qui a évolué plus tard pour devenir *DesignAid*, le premier outil logiciel permettant d'évaluer logiquement et sémantiquement des diagrammes de conception de systèmes et de logiciels. Sous la direction de *Albert F. Case Jr.* et de *Vaughn Frick*, la suite de produits *DesignAid* a été étendue pour supporter un large spectre de méthodologies de conception et d'analyse. Les outils de type CASE ont atteint leur heure de gloire au début des années 1990, époque à laquelle IBM avait proposé AD/Cycle [237], un framework de développement CASE sur mainframe. Avec le déclin du mainframe, les grands outils de type CASE de l'époque ont disparu pour laisser la place aux logiciels sur ordinateurs personnels.

Aujourd'hui, nous pouvons citer dans la catégorie des environnements de type CASE des suites de produits telles que celles des entreprises *IBM Rational, Telelogic* et *Computer Associates*. Il n'y a pas de consensus sur la définition de ce qu'est un outil de type CASE. Nous retenons la définition généraliste suivante, donnée par le *Software Engineering Institute* de l'université *Carnegie Mellon* [238] [239] *: "A CASE tool is a computer-based product aimed at supporting one or more software engineering activities within a software development process."* (« Un outil CASE est un produit logiciel destiné à supporter une ou plusieurs activités de génie logiciel dans un processus de développement de logiciel »).

Les premières générations des outils de type CASE ciblaient l'automatisation de tâches isolées du cycle de développement telles que la production de la documentation, la gestion des versions et le support des méthodologies de conception. Le besoin de connecter ces différents d'outils a donné naissance aux environnements de type CASE tels que définis dans [239], c'est-à-dire intégrant différents outils de manière à constituer de véritables plates-formes de développement intégrées.

5.4.2 Model Integrated Computing (MIC)

Apparue au milieu des années 90, l'approche *Model-Integrated Computing* (MIC) [240] promeut l'utilisation de modèles domaine-spécifiques à la base du développement logiciel. La motivation première de cette approche était d'apporter à l'ingénierie des systèmes logiciels complexes une méthodologie ainsi que des outils logiciels de support.

La méthodologie proposée est décomposée en deux phases. La première phase consiste à analyser le domaine d'application. L'objectif de cette phase est de trouver des paradigmes de modélisation appropriés et définir avec précision le langage de modélisation qui sera utilisé.

Un outil automatique peut alors utiliser ces informations pour générer un environnement de modélisation dédié au domaine. Cet environnement est utilisé directement dans la seconde phase, qui consiste à modéliser l'application désirée.

La plate-forme *GME* (Generic Modeling Environment) [234] est une implémentation de la méthodologie définie par l'approche MIC. Dans sa dernière version, GME est intégré à l'environnement Visual Studio .NET et propose un ensemble de langages et d'outils pour

l'ingénierie des modèles et des langages. Cependant, il ne permet pas pour le moment de spécifier la sémantique des langages développés.

5.4.3 Software Factories

Les « *Software Factories* » [235] représentent la vision de Microsoft quant à l'ingénierie dirigée par les modèles. Cette approche s'inspire des principes des lignes d'assemblage dans l'industrie et repose sur les idées principales suivantes :

– Les lignes d'assemblage ne fabriquent qu'un seul type de produit avec de faibles points de variation. Ainsi les lignes de production dans l'automobile ne produisent qu'un seul type de voiture, avec des variations possibles pour la couleur ou les combinaisons d'options.

– Les ouvriers sont souvent spécialisés. Si certains ont parfois des activités relativement variées, elles ne couvrent jamais la totalité des activités de la ligne d'assemblage.

– Les outils sont très spécialisés et très automatisés. Ils ne peuvent généralement pas être réutilisés sur d'autres lignes de production que celle pour laquelle ils ont été conçus.

– Les composants à assembler ou à usiner proviennent souvent de tierces parties (fournisseurs). Par exemple, les lignes de production automobile assemblent généralement des pièces qui sont produites dans d'autres usines.

Ces principes ont prouvé leur efficacité dans le domaine de la fabrication de familles de matériels.

L'approche des Software Factories propose d'appliquer ces principes au développement de logiciels. Suivant les deux premiers points, les fournisseurs de logiciel et les développeurs devraient être hautement spécialisés. Le troisième point suggère que les outils (de modélisation) devraient également spécialisés, c'est-à-dire être domaine-spécifiques. Ce principe inclut les langages de modélisation, les logiciels d'assistance ainsi que les transformations. Le dernier point pousse à la réutilisation de composants sur étagère. Ainsi, en résumé, une « Software Factory » est un environnement de développement configuré pour produire rapidement un type spécifique d'applications. L'environnement de développement intégré Microsoft Visual Studio .NET 2008 met en application ces idées et propose un framework de développement qui peut être configuré pour cibler un domaine particulier.

5.5 Principe de transformation de graphes

Les graphes et les diagrammes sont un moyen pratique, intuitif et direct pour la modélisation des systèmes complexes. Citons comme exemples, les diagrammes UML, les réseaux de Petri ou encore les automates finis. Si les graphes servent à visualiser les structures complexes des modèles d'une manière simple et intuitive, les transformations de graphes peuvent être exploitées pour spécifier comment ces modèles peuvent évoluer.

Les transformations de graphes ont évolué dans la répercussion à l'imperfection dans l'expressivité des approches de réécriture classique comme *les grammaires de Chomsky* et *la réécriture de termes* pour s'y prendre avec les structures non linéaires.

Une transformation de graphe [241] [242] [243] consiste en l'application d'une règle à un graphe et itérer ce processus. Chaque application de règle transforme un graphe par le remplacement d'une de ses parties par un autre graphe. Autrement dit, la transformation de graphe est le processus de choisir une règle d'un ensemble indiqué, appliquer cette règle à un graphe et réitérer le processus jusqu'à ce qu'aucune règle ne puisse être appliquée.

La transformation de graphe est spécifiée sous forme d'un modèle de grammaires de graphes. Ces dernières sont une généralisation des grammaires de Chomsky pour les graphes. Elles sont composées de règles. Une règle est constituée de deux parties, le *Left Hand Side* (*LHS*) et le *Right Hand Side* (*RHS*). Le LHS est la partie gauche de la règle, destinée à être mise en concordance avec les parties du graphe (appelé *host graph*) où on veut appliquer la règle. La

partie droite de la règle, Le RHS, décrit la modification qui sera effectuée sur le *host graph*, elle substitue dans le *host graph* la partie identifiée par la partie gauche de la règle.
Il existe plusieurs formalismes pour représenter les règles de transformations. Dans cette thèse, notre travail est basé sur une transformation dont les règles sont exprimées visuellement. La Figure 5.13 montre le principe général de l'application d'une règle sur un graphe.

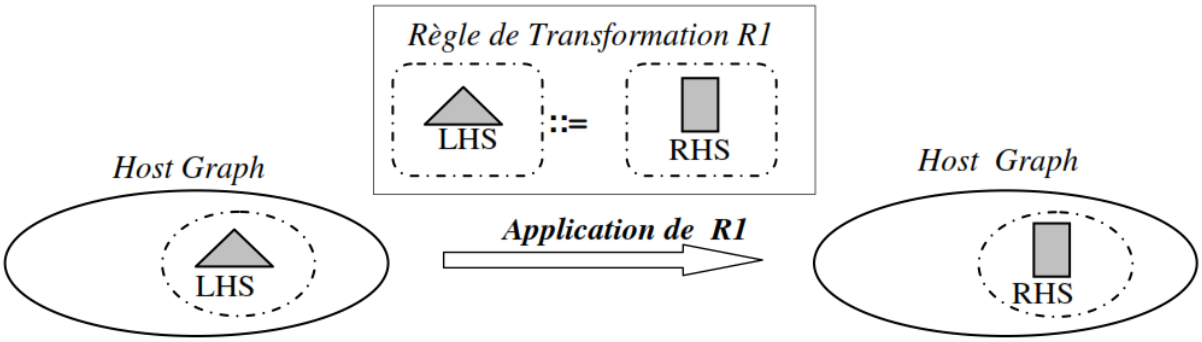


Figure 5.13. Principe de l'application d'une règle.

5.5.1 Notion de Graphe

Un *graphe* est constitué de *sommets* qui sont reliés par des *arêtes*. Deux sommets reliés par une arête sont *adjacents*. Le nombre de sommets présents dans un graphe est appelé *l'ordre du graphe*. Le *degré* d'un sommet est le nombre d'arêtes dont ce sommet est une extrémité.

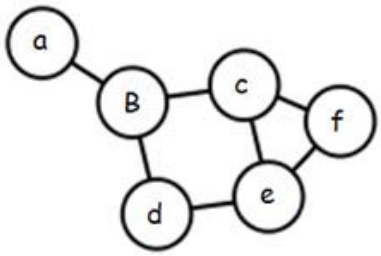


Figure 5.14. Graphe non orienté.

Un *sous-graphe* d'un graphe G est un graphe G' composé de certains sommets de G, ainsi que toutes les arêtes qui relient ces sommets.

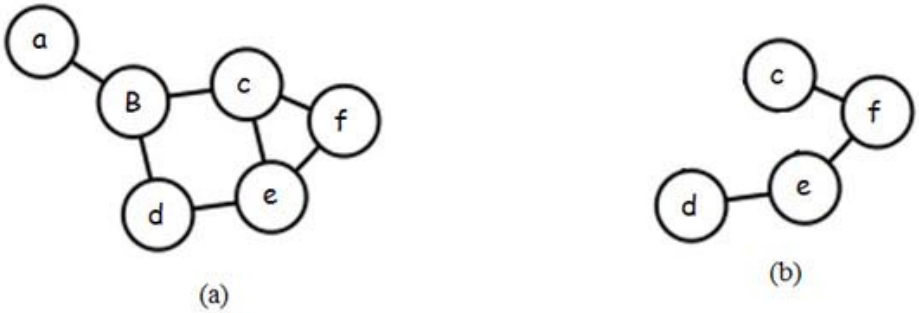


Figure 5.15. (a) graphe, G (b) sous-graphe de G.

Un *graphe orienté* est un graphe dont les arêtes sont orientées : on parle alors de l'origine et de l'extrémité d'une arête. Dans un graphe orienté une arête est dénommée *arc*.

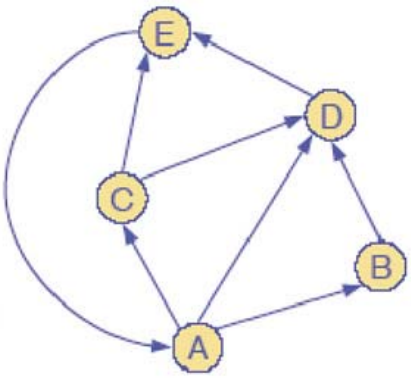


Figure 5.16. Graphe orienté.

Un *graphe étiqueté* est un graphe orienté, dont les arcs sont affectés d'étiquettes. Si toutes les étiquettes sont des nombres positifs, on parle de *graphe pondéré*.

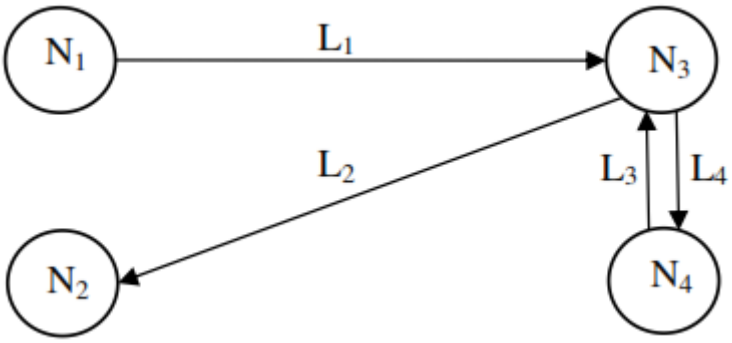


Figure 5.17. Graphe orienté étiqueté.

Un *graphe attribué* est un graphe qui peut contenir un ensemble prédéfini d'attributs.

5.5.2 Grammaire de Graphe

Une grammaire de Graphe [242] est généralement définie par un triplet:

$$GG = (P, S, T)$$

Où P : ensemble de règles, S : un graphe initial et T : ensemble de symboles.

Une grammaire de graphes distingue les graphes non terminaux, qui sont les résultats intermédiaires sur lesquels les règles sont appliquées, des graphes terminaux dont on ne peut plus appliquer de règles. Ces derniers sont dans le langage engendré par la grammaire de graphe. Pour vérifier si un graphe G est dans les langages engendrés par une grammaire de graphe, il doit être analysé. Le processus d'analyse va déterminer une séquence de règles dérivant G.

5.5.2.1 Le principe de règles

Une règle de transformation de graphe est définie par : $r = (L, R, K, glue, emb, cond)$.

Elle consiste en :

- Deux graphes : L graphe de côté gauche et R graphe de côté droit.
- Un sous graphe K de L.
- Une occurrence *glue* de K dans R qui relie le sous graphe avec le graphe de côté droit.
- Une relation d'enfoncement *emb* qui relie les sommets du graphe de côté gauche et ceux du graphe du côté droit.
- Un ensemble *cond* qui spécifie les conditions d'application de la règle.

5.5.2.2 Application des règles

L'application d'une règle r = (L, R, K, *glue*, *emb*, *cond*) à un graphe G produit un graphe résultant H. Le graphe H produit peut être obtenu depuis le graphe d'origine G en passant par les cinq étapes suivantes :

1. Choisir une occurrence du graphe de côté gauche L dans G.

2. Vérifier les conditions d'application d'après *cond*.

3. Retirer l'occurrence de L (jusqu'à K) de G ainsi que les arcs pendillés, c'est-à-dire tous les arcs qui ont perdu leurs sources et/ou leurs destinations. Ce qui fournit le graphe de contexte D de L qui a laissé une occurrence de K.

4. Coller le graphe de contexte D et le graphe de côté droit R suivant l'occurrence de K dans D et dans R. C'est la construction de l'union de disjonction de D et R et, pour chaque point dans K, identifier le point correspondant dans D avec le point correspondant dans R.

5. Enfoncer le graphe de côté droit dans le graphe de contexte de L suivant la relation d'enfoncement *emb*. Pour chaque arcs retiré incident avec un sommet v dans D et avec un sommet v' dans l'occurrence de L dans G et pour chaque sommet v'' dans R, un nouvel arc est établi (avec la même étiquette) incident avec l'image de v et le sommet v'' à condition que (v', v'') appartient à *emb*.

L'application de r à un graphe G pour produire un graphe H est appelée une dérivation directe depuis G vers H à travers r, elle est dénotée par $G \Rightarrow H$. En donnant les notions de règle et de dérivation directe comme étant les concepts élémentaires de la transformation de graphe, on peut définir les systèmes de transformation de graphe et la notion de langages engendrés.

5.5.2.3 Système de transformation de graphe

Un *système de transformation de graphe* est défini comme un système de réécriture de graphes qui applique les règles de la Grammaire de Graphes sur son graphe initial jusqu'à ce que plus aucune règle ne soit applicable [243].

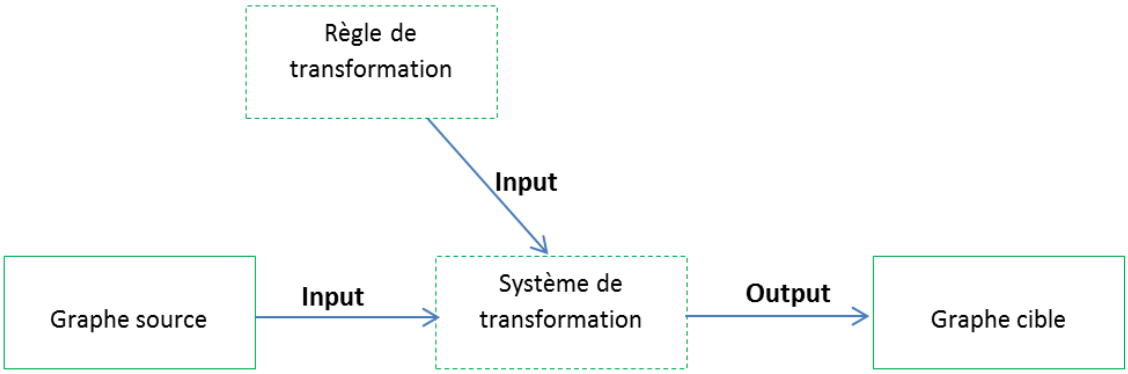


Figure 5.18. Principe de mise en œuvre de transformation de graphes.

Cette approche de transformations de modèles a plusieurs avantages par rapport aux autres approches :

- Les grammaires de graphes sont un formalisme naturel, visuel, formel et de haut niveau pour décrire les transformations.
- Les fondements théoriques des systèmes de la réécriture de graphes permettent d'aider à vérifier certaines propriétés des transformations telles que la terminaison ou la correction.

5.5.2.4 Langage engendré

Supposons que nous avons un ensemble donné P de règles et un graphe G_0, une séquence de transformations de graphe successive : $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow Gn$ est une dérivation à partir de G_0 vers G_n par les règles de P (à condition que toutes les règles utilisées appartiennent à P). G_0 est le graphe initial et G_n est le graphe dérivé de la séquence de transformation.
L'ensemble des graphes dérivés à partir d'un graphe initial S en appliquant les règles de P qui sont étiquetées par les symboles de T, est dit *langage engendré* par P, S et T et on écrit $L(P, S, T)$.

5.5.3 Outils de transformations de graphes

Il existe plusieurs outils implantant les systèmes de réécriture de graphes. Les plus intéressants sont :

 Fujaba [244]: un outil très puissant utilisé principalement pour la génération de code Java à partir de diagrammes UML. Ces dernières années, il est devenu une référence dans plusieurs axes de recherche. Particulièrement, il est très utilisé dans la modélisation et la simulation des systèmes mécaniques et électriques.

 AGG [207]: un environnement développé à l'université technique de Berlin. C'est l'un des outils de transformation de graphes les plus cités dans la littérature. Il est implémenté en langage Java. Il est muni d'une interface permettant de spécifier graphiquement les règles de transformation. Il permet des simulations pas à pas ainsi que quelques vérifications élémentaires. Les graphes manipulés sont orientés. Les nœuds et les arcs peuvent être étiquetés par des objets Java.

 TGG [245]: un environnement spécifique aux transformations bidirectionnelles. Sa puissance réside dans le fait que les mêmes règles permettant d'effectuer des translations dans les deux sens. Il est très utile pour assurer la synchronisation et le maintien de correspondance entre les deux modèles source et destination.

 GReAT [206]: un outil permettant la définition des transformations unidirectionnelles de plusieurs modèles sources vers plusieurs modèles cibles. Il est basé sur une notation graphique pour la spécification des règles de transformations, mais les expressions d'initialisation des attributs et les conditions d'application sont éditées textuellement. Les modèles manipulés doivent être conformes à leurs méta-modèles. Ces derniers peuvent être créés avec l'outil GME.

 Viatra [246]: un plugin Eclipse développé en 2005 à l'université de Budapest. Le logiciel utilise le langage VPM pour la modélisation. Pour définir les règles de transformations, l'utilisateur peut utiliser des motifs récursifs et des motifs de négations représentant les conditions d'application négatives. L'ordonnancement dans l'application des règles est basé sur une machine à états abstraite.

 Eclipse Modeling Galileo [247]: un environnement spécifique aux transformations de graphes intégré à la plateforme éclipse. Le framework principal est celui utilisé pour la modélisation. Il est appelé Eclipse Modeling Framework (EMF). Les modèles y sont décrits en XML, mais peuvent être spécifiés dans des documents UML/SysML. L'interopérabilité avec d'autres outils basés sur Eclipse est l'un des points forts de ce framework. Un de ces outils appelé Graphical Modeling Framework (GMF) permet le développement d'éditeur graphique de modèles. Les transformations de modèles sont exprimées en langage ATL (Atlas Transforming Language).

 Progres [248]: un environnement de transformation de graphes à nœuds et arcs étiquetés. La syntaxe de Progres est mi-textuelle, mi-graphique. Chaque opération de manipulation d'un graphe, est composée d'étapes basiques de recherche de motifs et de transformation des sous-graphes correspondants. Bien que basée principalement sur des règles, la programmation avec Progres peut aussi se faire de manière impérative. Les transformations sont atomiques et préservent la cohérence du graphe manipulé. Le programmeur peut faire des choix non-déterministes avec la possibilité d'initier un retour arrière plus tard si nécessaire.

 AToM3 [208]: un outil de modélisation multi-paradigme développé par le laboratoire MSDL de l'université McGill Montréal au Canada. AToM3 est un outil visuel dédié à la transformation de graphes, implémenté en langage Python [249]. Il possède une couche de Méta-Modélisation permettant une spécification syntaxique et graphique des formalismes utilisés. Les manipulations de modèles souhaitées sont définies sous forme de grammaires de graphes.

Dans le cadre de cette thèse, nous avons opté pour l'utilisation de l'outil TGG interpreter de l'environnement Eclipse EMF/Ecore pour l'implémentation des grammaires de graphes proposées. Nos contributions seront présentées dans le chapitre suivant.

5.6 Conclusion

Les techniques et langages de modélisation ont pris une place importante dans la conception et le développement des systèmes complexes. En effet, elles permettent de traiter l'interopérabilité à un niveau d'abstraction plus élevé que le niveau « code ». Ce chapitre nous place dans le contexte de cette thèse. Nous avons défini les principes de base de l'IDM et le rôle central des modèles, en montrant comment cette approche permet de pérenniser le savoir-faire grâce à l'élaboration de modèles tout au long du processus de développement, d'augmenter la productivité grâce à l'opérationnalisation de ces modèles en définissant des transformations, et finalement de prendre en compte les plateformes d'exécution en les intégrant directement dans les modèles. Nous avons également présenté la méta-modélisation dans le domaine IDM et précisément l'approche MDA, avec ce qu'elle apporte aux systèmes complexes. MDA permet l'automatisation des processus de modélisation, depuis les phases de développement jusqu'à celles de tests, en passant par la génération de code.
Nous avons focalisé sur un cadre spécifique de transformation de modèles basé sur la transformation de graphes dont l'objectif visé est d'intégrer la notion des grammaires de graphes dans le processus de transformation de modèles. Enfin, nous avons passé en revu quelque approches et outils pour la transformation de modèles.
Les concepts présentés dans ce chapitre constituent un background nécessaire pour la compréhension de nos contributions dans le cadre de cette thèse qui seront présentées dans le chapitre suivant.

CHAPITRE 6

Contributions

Sommaire

6.1 Introduction

L'Ingénierie Dirigée par les Modèles (IDM) est une approche de développement mettant à disposition de l'utilisateur des concepts, des langages et des outils. Les modèles sont considérés comme des éléments de base. Le raisonnement est entièrement à un haut niveau d'abstraction. L'application sera générée (en tout ou en partie, automatiquement ou semi-automatiquement) à partir de modèles. Les outils permettant de créer et d'exploiter ces modèles sont construits autour des concepts de *Méta-modélisation* et de *Transformation de modèles*.
Les méthodes formelles (MFs) sont des techniques basées sur les mathématiques permettant à la fois de modéliser le système et de vérifier les propriétés attendues. Les méthodes formelles reposent sur l'utilisation de langages formels. Un langage formel est un langage doté d'une sémantique mathématique rigoureuse. Les principaux défis d'utilisation des MFs dans les activités de développement des systèmes sont liés à :
-La difficulté réelle de manipuler les concepts théoriques et les méthodes d'analyse associées.
-La difficulté pour les développeurs d'exprimer les propriétés du système d'une façon aisée.

Cette thèse s'inscrit dans l'objectif de décrire une méthodologie pour la construction d'applications formellement vérifiées. Nous nous plaçons plus particulièrement dans le cadre de systèmes temps-réel embarqués. Le but étant d'optimiser les coûts liés au développement du système logiciel tout en assurant la qualité de celui-ci.

6.2 Outils d'implémentation

Un aspect de notre approche concerne le modèle de données utilisé pour représenter et manipuler les modèles. En effet, dans le monde des modèles, la structure de représentation est majoritairement le graphe. Une transformation de modèle revient alors à une transformation de graphe.

Dans cette section, nous décrivons les techniques que nous avons utilisés pour mettre en œuvre notre démarche. Du fait que notre approche est dirigée par les modèles, nous avons utilisé un environnement technique adapté à la modélisation, la méta-modélisation et la transformation des modèles. Nous avons choisi l'espace technologique très répandu Eclispe Modeling Framework (EMF[9]), qui utilise Ecore pour créer et manipuler les modèles.

6.2.1 La plateforme Eclipse

Eclipse est une communauté open source dont les projets sont axés sur la construction d'une plate-forme de développement. La plateforme Eclipse, initiée par IBM, est un atelier de génie logiciel (AGL) conçu dans le but de fournir une plateforme modulaire et extensible pour la construction, le déploiement et la gestion des logiciels, couvrant l'ensemble de leurs cycles de vie. Tout le code de la plateforme a été mis à la disposition de la communauté open source afin de contribuer à son développement.

[9] https ://www.eclipse.org/modeling/emf/, dernière consultation : 28/11/2017

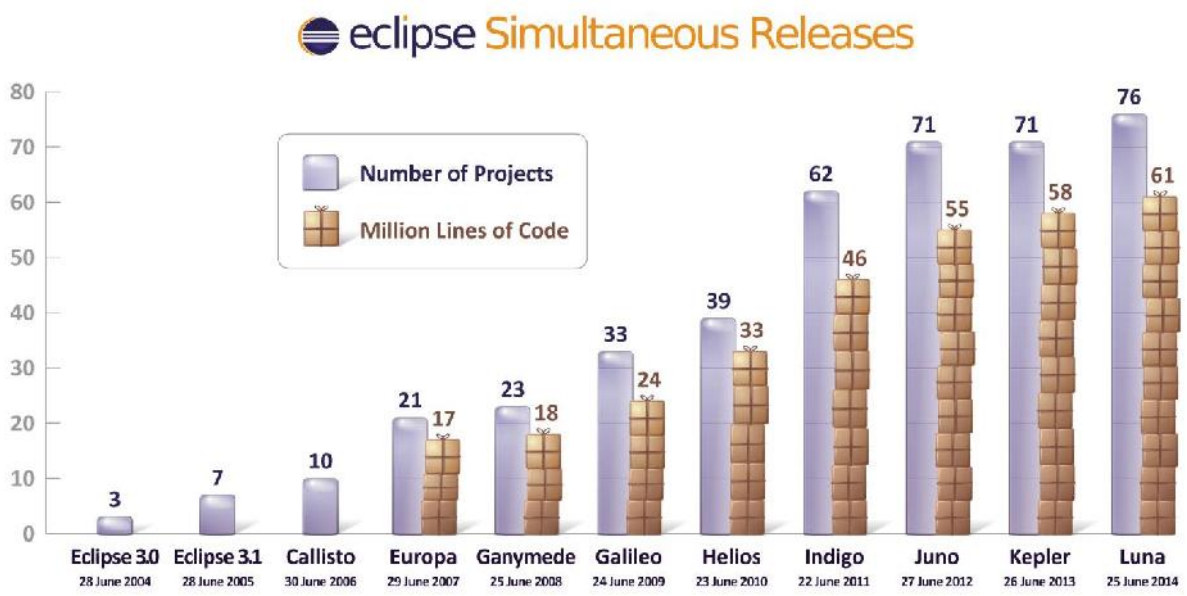


Figure 6.1. Croissance de la plateforme Eclipse dans le temps [250].

La figure 6.1, élaboré par Holger Voorman, indique le nombre de projets et de lignes de code sous les différentes distributions d'Eclipse au fil des ans. Avant 2012, la plateforme Eclipse a été publiée dans la version 3.x sous plusieurs distributions parmi lesquelles Helios (3.6), Indigo (3.7) qui sera utilisée par la suite pour la réalisation de notre proposition, etc. À partir de 2012, la version de la plateforme est passée au 4.x, avec les distributions Juno (4.2), Kepler (4.3) et Luna (4.4) sortie en 2014.

6.2.2 EMF/Ecore

EMF [251] qui signifie *Eclipse Modeling Framework*, est un environnement de modélisation et de génération de code qui facilite la construction d'outils et d'applications basées sur des modèles de données structurées. Il est à la base de nombreux outils IDM. Son métamodèle Ecore sert de pivot et permet donc l'interopérabilité entre outils. EMF permet aussi le développement rapide et l'intégration de nouveaux plug-ins Eclipse. EMF est composé d'un ensemble de briques appelées plug-ins. Parmi ces plug-ins nous citons :

- Le métamodèle *Ecore* qui est un canevas de classes pour décrire les modèles *EMF* et manipuler les référentiels de modèles.
- *EMF.Edit* est un canevas de classes pour le développement d'éditeurs de modèles EMF.
- Le Modèle de génération *GenModel* qui permet de personnaliser la génération Java.
- *JavaEmitterTemplate* qui est un moteur de template générique.
- *JavaMerge* qui est un outil de fusion de code Java.

La figure 6.2 présente une vue simplifiée du méta-modèle Ecore. On y voit que les éléments principaux sont les classes, et qu'elles peuvent être composées d'attributs (possédant un type) d'une part, et /ou de références vers d'autres classes d'autre part. En cas de l'utilisation d'Ecore comme méta-formalisme (donc pour créer des méta-modèles), on parlera alors de méta-classes, de méta-attributs…etc.

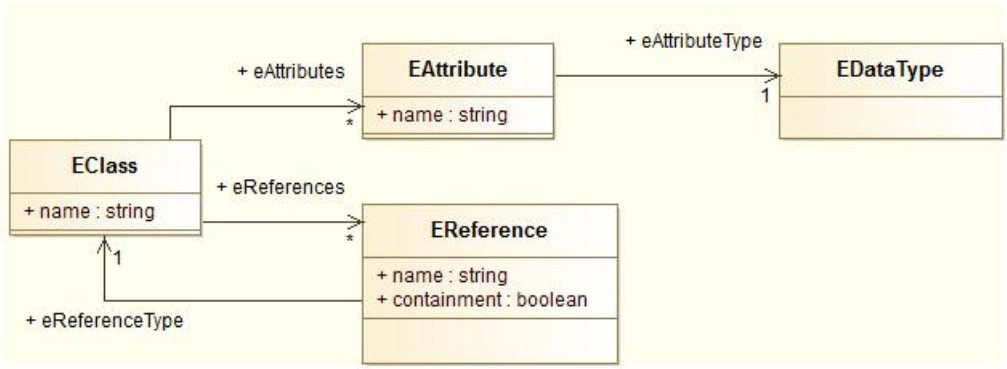


Figure 6.2. Méta-modèle simplifié Ecore [252].

Pour créer les métamodèles à partir d'Ecore, EMF propose une forme arborescente (l'extension du métamodèle créé est alors « **.ecore** ») et une forme graphique (l'extension du métamodèle créé est alors « **.ecorediag** »).
La rencontre de l'IDM et du phénomène Eclipse a donné lieu à de nombreux projets tels que : EMP (Eclipse Modeling Project) qui met l'accent sur l'évolution et la promotion des technologies de développement basées sur les modèles dans la communauté Eclipse, MoDisco qui fournit une plate-forme extensible pour développer des outils IDM, etc.
EMP est un package qui regroupe plusieurs projets qui traitent de la modélisation afin d'unifier les visions et d'améliorer l'interopérabilité. Ces projets sont : EMF, GMF (Graphical Modeling Framework) qui permet de développer des générateurs d'éditeurs de modèles, UML 2, et GMT (Generative Modeling Tools).

6.2.3 TGG Interpreter

L'approche des grammaires de graphes triples (Triple Graph Grammars : TGG), introduite par Andy Schürr [253] est une technique utilisée pour définir la relation entre deux types de modèles. Et par la suite, transformer un modèle d'un type à un autre, afin de calculer la correspondance entre deux modèles existants. Ainsi, pour le but de maintenir la cohérence entre deux types de modèles, tel qu'il est définit en TGG. Lorsqu'un des modèles est changé, le modèle cible peut être modifié, ce qui signifie que la transformation ou la synchronisation peut être appliquée progressivement [254] [255].
L'avantage principal des TGG est qu'ils permettent de définir des transformations d'une façon déclarative, avec une exécution dans les deux directions de la transformation (i.e., la définition est donc bi-directionnelle). Dans ce contexte, TGG Interpreter [256] est considéré comme le meilleur choix pour les transformations de graphes.

Le TGG Interpreter a été développé pour la transformation de modèles TGG et est un outil de mise à jour incrémentale résultant de la comparaison de TGG et de la norme de l'OMG bidirectionnel QVT (Query/View/Transformation) pour les transformations de modèles. Le TGG interpreter est basé sur Eclipse et peut être installé via l'Update Manager Eclipse. La figure 6.3 représente une capture d'écran de l'éditeur de règles TGG de la plateforme Eclipse.

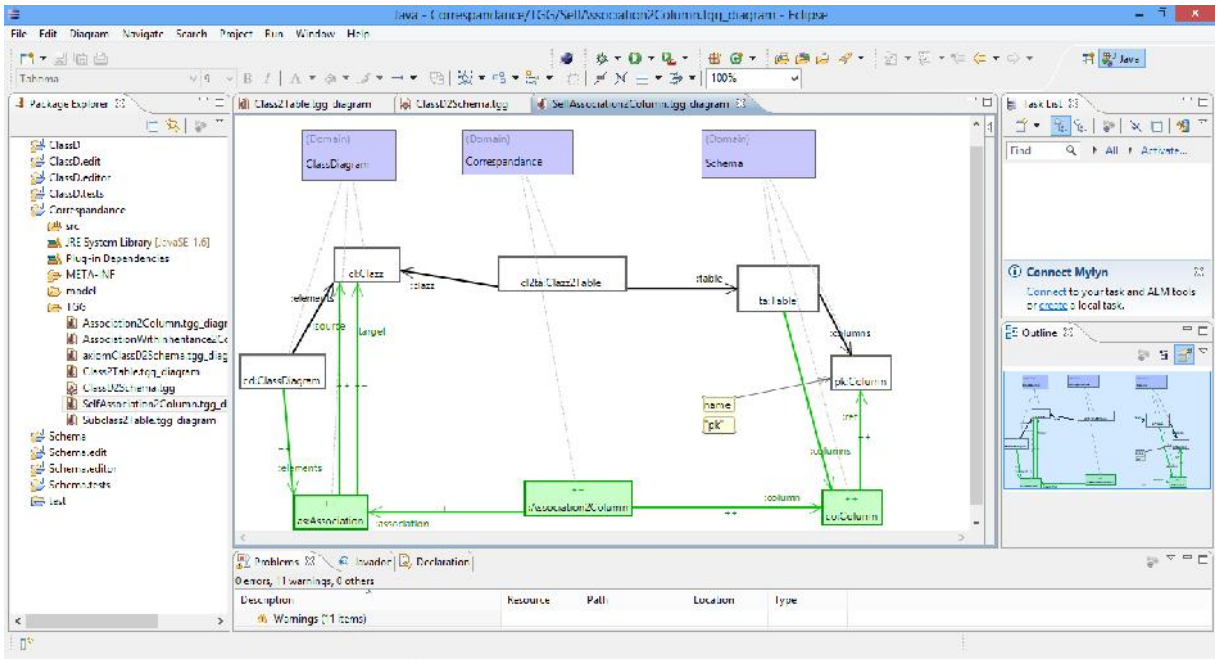


Figure 6.3. Éditeur de Règles TGG de TGG Interpreter.

La transformation de modèles se base sur la définition des règles de transformation où une bonne transformation de modèles dépend d'une bonne définition des règles de transformation. TGG se compose de trois graphes ou chaque graphe appartient à un domaine. Chaque domaine contient une règle de grammaire de graphe. Habituellement, le domaine sur la gauche est appelé source et le domaine sur la droite est appelé cible. Le domaine de correspondance relie le domaine source et cible. Chaque domaine est attribué à un métamodèle. Les nœuds de ce domaine sont les classes de ce métamodèle (on peut considérer le domaine comme une grammaire du métamodèle).
Avant de spécifier les règles, on doit définir les différents nœuds qu'on va rencontrer dans les règles. Il existe différents nœuds de règle de transformation : nœud de contexte, nœud de production, les contraintes (figure 6.3) et les nœuds réutilisables.

• nœud de contexte
Parfois, seule une partie d'un modèle est pertinente et doit être transformée. Ensuite, les règles de TGG devraient être conçues pour les parties pertinentes de la transformation. Si les pièces d'un modèle moins pertinents influencent la transformation, ils peuvent apparaître comme des nœuds de contexte dans les règles de TGG. La présentation graphique d'un nœud de contexte est présentée d'une case avec un contour noire.

Les nœuds de contexte doivent être appariés avec les objets du modèle précédemment traités. Cela signifie que le TGG, tel qu'il est présenté à ce jour, besoin de spécifier une grammaire complète pour toutes les parties de modèle. En revanche, TGG permet également de formuler une grammaire partielle sur un modèle. Là, les nœuds de contexte ne doivent pas nécessairement être préalablement traitées par une autre règle [257].

• Nœud de production
Les nœuds de production sont affichés sous forme de boîtes vertes avec une bordure verte et une label "++". Les arrêtes de production sont affichés sous forme de flèches de couleur vert foncé avec une étiquette "++". Les nœuds de production ne doivent pas correspondre à nœud

qui existe déjà. Si on trouve ces éléments dans le domaine source, les éléments de modèle cible et de modèle de correspondance peuvent être créés selon la règle. Tous les objets créés sont liés aux nœuds de contexte de la règle. En conséquence, un objet de modèle ne peut être présenté comme un nœud de production qu'une seule fois. Nous appelons cela la sémantique bind-only-once pour les de nœuds de productions [258].

• **Nœud réutilisable**
A noter que les types de composants ne doivent pas être générés, mais peuvent être réutilisés encore et encore depuis des nœuds déjà existants avec les propriétés requises. Par conséquent, nous appelons ces nœuds "les nœuds réutilisables". Graphiquement, ces nœuds sont représentés en gris. La sémantique des nœuds réutilisables est qu'ils peuvent être nouvellement générées ou réutilisés de façon arbitraire. La couleur grise reflète le fait que, sémantiquement, chaque nœud réutilisable pourrait être soit en noir (un nœud du côté gauche de la règle de TGG, à savoir qu'il est réutilisé) ou en vert (un nœud du côté droit de la règle de TGG, à savoir qu'il est nouvellement généré), qui peut être choisi à chaque fois que la règle est appliquée. Cette interprétation montre en fait que, les nœuds réutilisables ne sont pas strictement nécessaires. Nous pourrions remplacer par un nombre exponentiel de règles de TGG. Mais, le concept de nœuds réutilisables permet de réduire le nombre et la complexité des règles de TGG. En plus si l'exemple est complexe, les règles sans nœuds réutilisables peuvent générer un désordre. Les nœuds réutilisables nous permettre d'avoir plus de règles simples et de concentrer sur l'essentiel des modèles pertinents [254].

• **Les contraintes** la contrainte réelle dans un nœud de contrainte peut être toute expression OCL qui se réfère à des objets qu'il est attaché. Les restrictions sont seulement nécessaires pour les mettre en œuvre des transformations ou des synchronisations plus efficace. Une contrainte OCL représentée avec la couleur jaune et doit être attachée avec un autre nœud.

6.2.4 Le langage de génération de code Xpand

Etant donné que la génération de code à partir des modèles exécutables cible des langages de programmation de forme textuelle, elle est généralement basée sur l'approche M2T. Plusieurs langages de génération de code basés sur l'approche M2T existent tels que Xpand [259] MOFScript [260] ou encore JET [261] basé sur le langage Java. Dans cette thèse, nous avons choisi le langage Xpand, à base template, pour générer des spécifications RT-Maude à partir des modèles RT-Maude.

Le cadre de générateur Xpand fournit un langage textuel, qui sont utiles dans des contextes différents dans le processus de conception des systèmes, basé sur l'IDM (par exemple, de validation, extensions de méta modèle, génération de code, la transformation du modèle). Xpand est un langage spécialisé sur la génération de code à partir de modèles EMF.
Pour créer un projet Xpand on a besoin d'un modèle EMF, un modèle *chek* dont l'extension « **.chk** » pour définir quelques contraintes et de trois packages essentielle contiennent des fichiers de différentes extensions :

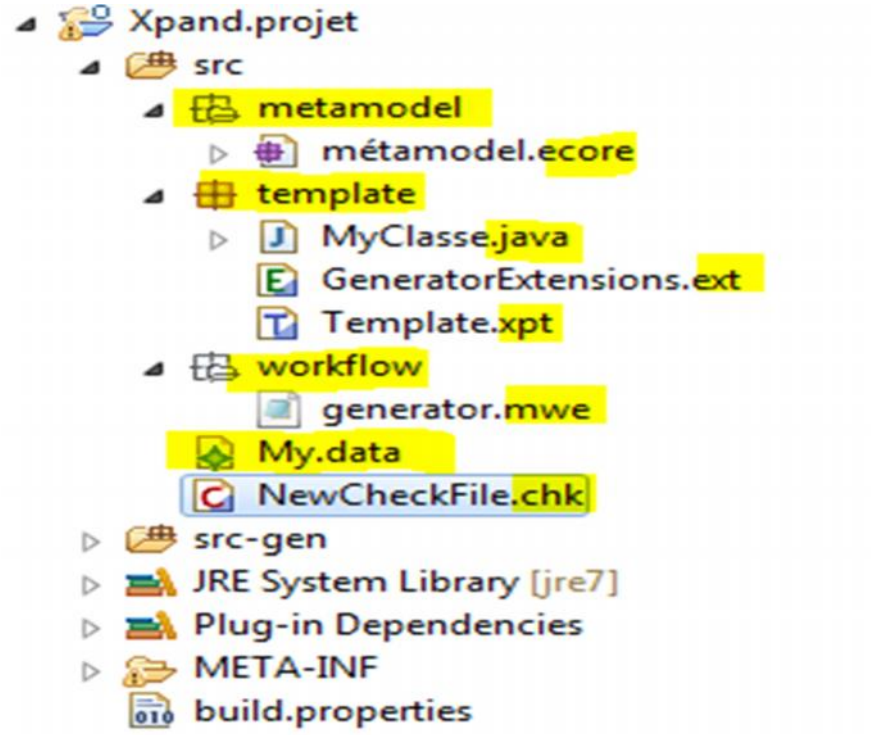


Figure 6.4. Structure d'un projet Xpand.

– le package méta model : contient un méta modèle de sortie (ex le méta modèle d'un langage).
– le package Template : contient un fichier java qui a l'extension « **.java** », un fichier xpand qui a l'extension « **.xpt** » et un fichier xtend qui a l'extension « **.ext** ».
– le package Workflow : dont l'extension « **.mwe** », contient le workflow qui permet d'appliquer le Template sur un modèle pour engendrer un ou plusieurs fichiers textes.

6.2.4.1 Structure générale d'un template Xpand

Le fichier Xpand permet le contrôle de la génération de code correspondant à un modèle [262]. Le modèle doit être conforme à un méta-modèle donné.
Le fichier d'extension « **.xpt** » se compose d'une ou plusieurs instructions **IMPORT** afin d'importer les méta-modèles, de zéro ou plusieurs **EXTENSION** avec le langage Xtend et d'un ou plusieurs blocks **DEFINE**.
Les blocs **DEFINE** constituent le concept central du langage Xpand. La balise **DEFINE** se compose d'un nom, une liste optionnelle de paramètres et du nom de la méta-classe pour laquelle le template est défini. Ils ont le format suivant :

«**DEFINE** templateName(formalParameterList) **FOR** MetaClass»
a sequence of statements
«**ENDDEFINE**»

Ainsi un bloc **DEFINE** contient un ou plusieurs instruction **FILE** qui redirige la sortie produite, à partir des instructions de son corps, vers la cible spécifiée. La cible est un fichier dont le nom est spécifié par expression.

«**DEFINE** Entity **FOR** data::Entity»
«**FILE** name + ".java"»
....

«**ENDFILE**»
«**ENDDEFINE**»

Aussi un fichier Xpand se compose d'un ou plusieurs instructions **EXPAND** appelle un bloc **DEFINE** et insère le contenu de la production "*output*" à son emplacement et un des instructions **FOR** ou **FOREACH**. Le format de l'instruction **EXPAND** se présente comme suit :

«**EXPAND** definitionName [(parameterList)]
[**FOR** expression | **FOREACH** expression [**SEPARATOR** expression]]»

« *definitionName* » est le nom du bloc **DEFINE** appelé. Si **FOR** est spécifié, la définition est exécutée pour le résultat d'une expression cible. Si **FOREACH** est spécifiée, l'expression cible doit être évaluée à un type collection. Dans ce cas, la définition spécifiée est exécutée pour chaque élément de cette collection. Il est possible de spécifier un séparateur pour les éléments générés de la collection.

«**EXPAND** Entity **FOREACH** entity»
«**ENDDEFINE**»
«**DEFINE** Entity **FOR** data::Entity»

6.2.4.2 Structure générale d'un Workflow

Pour exécuter le générateur Xpand, nous devons définir un workflow. Il contrôle les différentes étapes d'un projet Xpand (modèles de chargement, la vérification et la génération de code). Avant d'exécuter le workflow, assurez-nous que votre méta modèle peut être trouvé sur le chemin de classe avec des autres classe (plug-in).
Le fichier workflow est basé sur le langage XML ou l'élément racine est <**workflow**> cette balise contient plusieurs autres balises à l'antérieur parmi elle la balise de déclarations des propriétés (cette déclaration peut être dans un autre fichier), la déclaration des propriétés est faite par la balise <**property**> qui contient un attribut pour définit le nom '**name**' et un autre attribut pour l'emplacement de fichier '**value**' (comme le fichier exporte dans la première est l'emplacement de fichier de sortie dans la deuxième)

<workflow>
<property name= "model" value= "/home/user/target ' '| />
<property name="src-gen" value="src-gen" />

Aussi le workflow contient une balise <**component**> qui est utilisé pour importer des classes à l'aide de l'attribut **'classe'**.

6.2.4.3 langage XTEND

Comme les expressions sous-language (*sublanguage*) qui résument la syntaxe des expressions pour tous les autres langages textuels livré avec le framework Xpand, il existe un autre langage communément appelé Xtend.
Ce langage offre la possibilité de définir des bibliothèques riches d'opérations indépendantes et des extensions de métamodèle non invasives basées sur des méthodes Java ou des expressions Xtend. Ces bibliothèques peuvent être référencées à partir de tous les autres langages textuels basés sur le cadre des expressions [262].

6.2.4.4 Le langage Check

Xpand fournit également un langage pour spécifier les contraintes que le modèle doit remplir pour être correct. Ce langage est très facile à comprendre et à utiliser. Les contraintes spécifiées dans le langage Check doivent être stockées dans des fichiers avec l'extension de fichier « **.chk** ». Ce fichier doit commencer par une importation du méta-modèle selon le format « **import** *metamodel*; ». Chaque contrainte est spécifiée dans un **contexte** (*context*), soit une méta-classe du méta-modèle importé, pour lequel la contrainte s'applique. Les contraintes peuvent être de deux types :

- ***Warning*** : dans ce cas, si la contrainte n'est pas vérifiée un message sera affiché sans que l'exécution s'arrête. C'est similaire aux erreurs non bloquantes produites par le compilateur C.
- ***Error*** : dans ce cas, si la contrainte n'est pas vérifiée un message sera affiché et l'exécution sera arrêtée

L'exemple suivant présente une contrainte de type « Error » qui permet de vérifier qu'un attribut *name* doit contenir plus qu'un seul caractère, pour les modèles conforme au méta-modèle «data » :

```
import data;
context Attribute ERROR
 "Names have to be more than one character long." :
 name.length > 1;
```

6.3 L'approche proposée

Dans cette thèse, nous avons proposé une approche pour l'analyse et la vérification des systèmes temps réel. Les spécifications semi-formelles et les modèles sont une première étape pour une description précise du système. Le système est modélisé pour répondre aux besoins fonctionnels et non fonctionnels des utilisateurs.

Dans notre approche, nous avons utilisé UML (décrit dans le Chapitre 4) pour modéliser le comportement du système. Le modèle fonctionnel le plus connu dans UML est appelé le diagramme d'états-transitions (Statechart). Il représente le comportement d'une entité en termes de transitions entre états déclenchés par des événements.

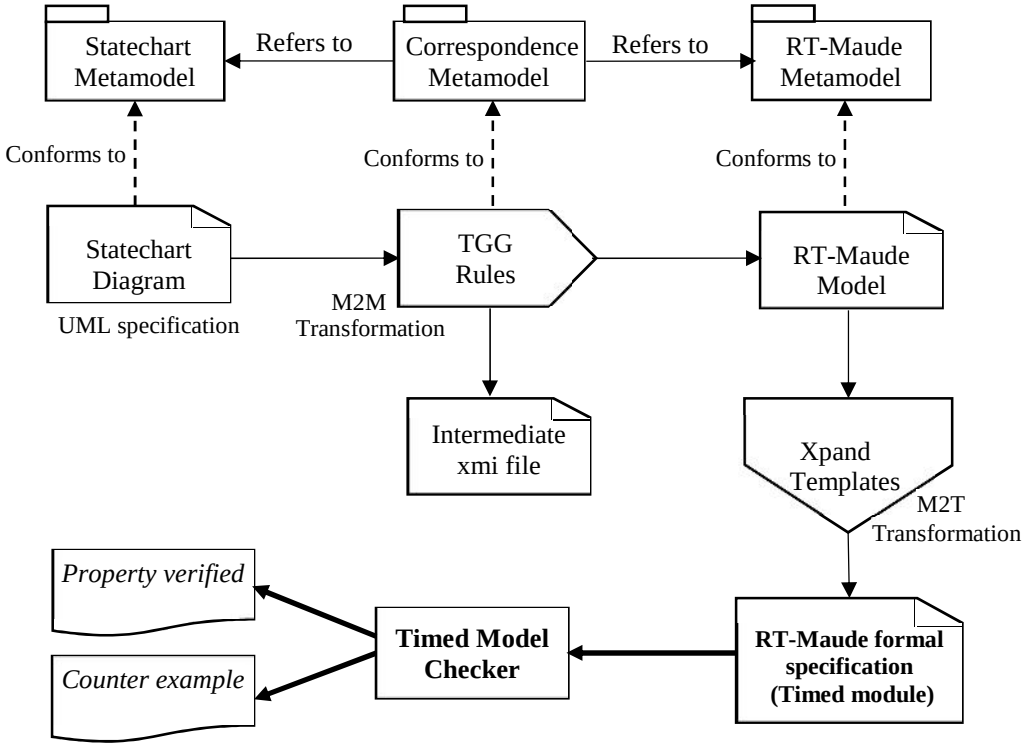


Figure 6.5. L'approche proposée.

Le but de notre travail est de générer une spécification formelle RT-Maude, fournie dans un module RT-Maude, à partir d'un diagramme d'état-système d'un système temps réel et comment ces spécifications peuvent être vérifiées à l'aide de l'outil RT-Maude [263]. Ce travail est divisé en quatre étapes majeures (figure 6.5): (1) Description des exigences fonctionnelles du système temps réel à l'aide du diagramme UML Statechart, (2) Les techniques de transformation du modèle basées sur TGGs sont utilisées pour traduire ce diagramme en modèle RT-Maude, (3) génération de spécification formelle RT-Maude (insertion dans un module temporisé) à travers la transformation du modèle en texte en utilisant le langage template Xpand, et (4) utiliser Maude LTL Model-Checker pour vérifier le module spécifié dans la logique de réécriture , un contre-exemple est également généré en cas d'une propriété non tenue. Par exemple, si la propriété *deadlock-freeness* échoue, elle affiche le contre-exemple qui est une trajectoire de l'état initial à l'état de blocage.

6.3.1 Transformations de modèle à modèle (M2M)

La notion de transformation de modèle-à-modèle définit un processus automatisé traduisant un modèle initial en un second modèle. Un métamodèle est une description formelle de toutes les constructions syntaxiques d'un langage. Le langage de modélisation est représenté par un méta-modèle qui définit les éléments et la structure d'un modèle ainsi que sa sémantique. En d'autres termes, le méta-modèle exprime le lien existant entre un modèle et le système modélisé. Dans notre travail, il s'agit d'une transformation M2M exogène (le modèle source et cible ne sont pas conforme au même méta-modèle) et horizontale (le modèle source et cible sont dans le même niveau d'abstraction).

6.3.1.1 Les métamodèles impliqués

Notre approche est implémentée dans le framework Eclipse, en utilisant notamment des modèles EMF ECore. EMF fait partie intégrante de la plateforme Eclipse. En vertu de la Fondation Eclipse, il existe un méta-modèle (Ecore) afin de définir des modèles et un support d'exécution pour les modèles incorporés dans le cadre de base de la fonction EMF. Par EMF, nous pouvons générer des interfaces pour définir nos objets, ce qui nous permet de garder notre application propre des classes d'implémentation individuelles.

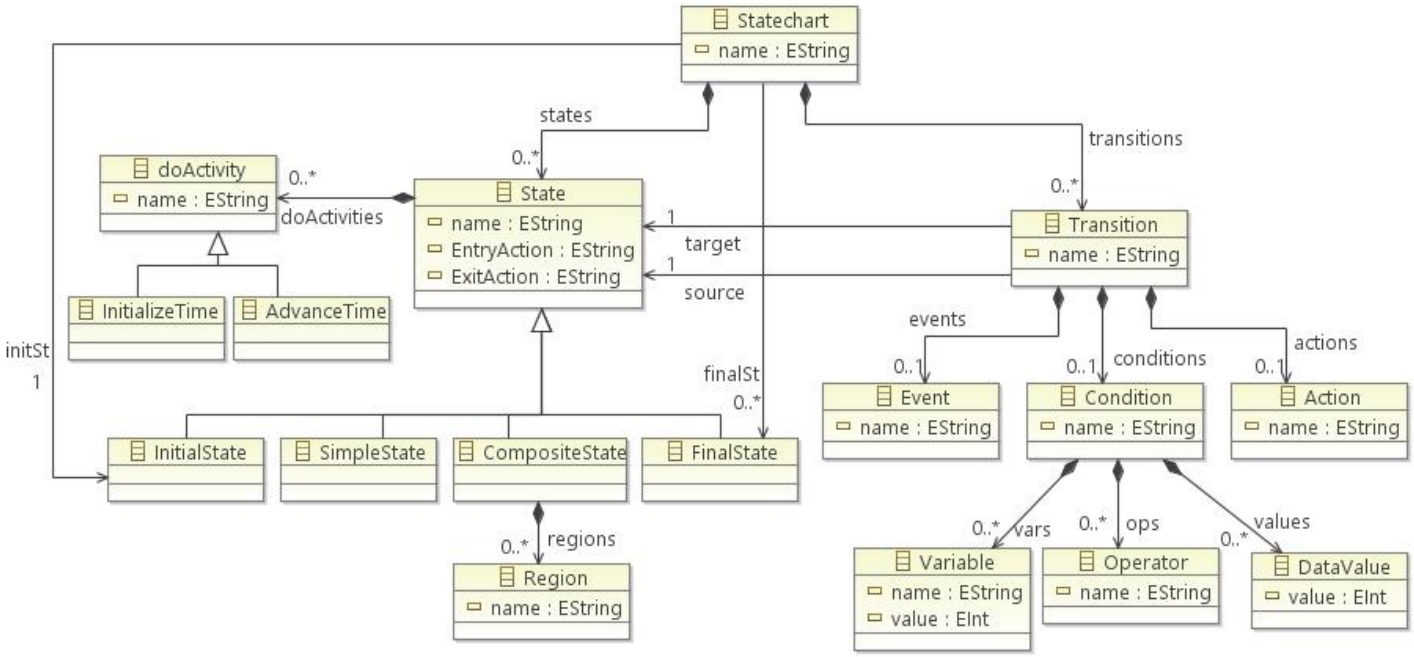


Figure 6.6. Le méta-modèle de diagramme d'états-transitions (Statechart).

Parce que l'outil visuel EMF dispose d'une couche de méta-modélisation qui nous permet de modéliser graphiquement les différents formalismes, nous devons implémenter trois méta-modèles, le premier étant un méta-modèle source pour des diagrammes d'états-transitions (figure 6.6), aussi on a besoin d'un méta-modèle cible qui est le méta-modèle RT-Maude (figure 6.7). Nous ajoutons le troisième méta-modèle, qui représente le méta-modèle de la correspondance entre eux (figure 6.8). En les créant, nous pouvons superviser et générer les différents modèles décrits dans le formalisme spécifié.

Le diagramme d'états-transitions UML permet de décrire les changements d'état d'un objet ou d'un composant, en réponse à des interactions avec d'autres objets / composants ou acteurs. C'est l'un des différents diagrammes UML utilisés pour modéliser la nature dynamique d'un système. Notre métamodèle d'états-transitions est composé des parties structurelles des états (états, régions, transitions) et des parties comportementales (événements, gardes, actions). Nous avons proposé pour méta-modéliser les diagrammes d'états-transitions les principales classes :

Classe "Statechart" : elle représente un diagramme d'états-transitions. Elle possède un attribut clé *Name*.

Classe "State" : Cette classe décrit les états simples. Elle possède trois attributs : *Name*, *EntryAction* et *ExitAction*.

Classe "CompositeState" : Cette classe représente les états composites. Elle hérite de *State* tous ses attributs, les multiplicités et les associations.

Classe "InitialState" : Cette classe représente l'état initial d'un diagramme d'états-transitions.
Classe "FinalState" : cette classe représente l'état final d'un diagramme d'états-transitions.
Classe "Transition" : Cette classe représente les actions lors du changement d'états à travers les transitions. Chaque transition est étiquetée par l'événement «event» , le nom de l'action et éventuellement d'une condition.
Puisque nous voulons décrire les systèmes en temps réel par l'intermédiaire de statecharts, nous ajoutons deux sous-classes *InitializeTime* et *AdvanceTime* (leurs rôles sont : initialiser et avancer le temps respectivement) pour la classe *doActivity*.

Le métamodèle RT-Maude montré à la figure 6.7 est inspiré de métamodèle Maude décrit dans [264]. Il est basé sur la définition du langage Maude [130]. Le méta-modèle est restreint aux éléments nécessaires de modèles et de méta-modèles à notre formalisation avec RT-Maude.

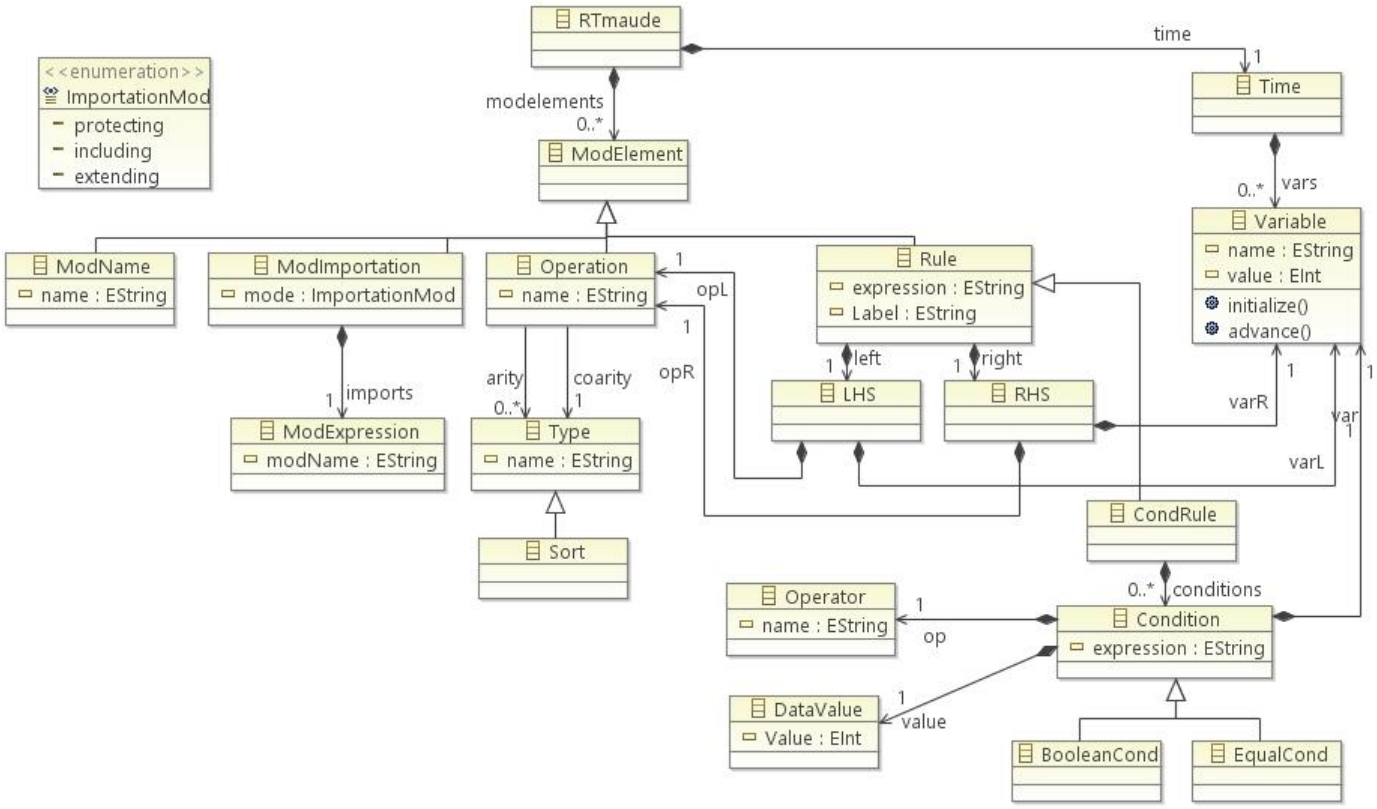


Figure 6.7. Le méta-modèle RT-Maude.

Le méta-modèle de correspondance définit les différents types de relations entre les éléments de méta-modèle source et cible. Il est composé de trois classes ; le premier *StateCh2RTmaude* crée un module RT-Maude pour le diagramme d'états-transitions. La seconde classe *InitialSt2ModElt* crée, pour l'état initial de Statechart, les éléments de base du module temporisé comme une variable de temps et un *sort* nommé *state*. La classe *Tr2Crule* est utilisée pour créer une règle pour chaque transition de Statechart. Implicitement, elle crée une règle conditionnelle pour chaque état de Satechart dans lequel nous illustrons l'avance du temps tant que la condition de la transition n'est pas satisfaite.

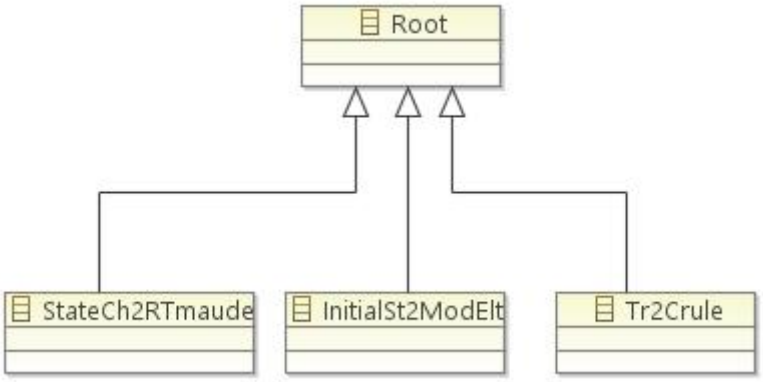


Figure 6.8. Le méta-modèle de correspondance.

Après avoir implémenté tous les méta-modèles impliqués dans ce travail, nous sommes maintenant en mesure d'automatiser la génération du modèle RT-Maude par la réalisation de nos règles TGG. *TGG Interpreter* [256] est l'outil qui permet d'exécuter nos règles de réécriture de graphe ainsi que leur automatisation. L'outil *TGG Interpreter* est composé d'autres outils permettant de spécifier et d'exécuter des transformations M2M basées sur TGG, ou comme dans le cas de notre travail, des transformations M2T à l'intérieur de la plateforme Eclipse. Nous pouvons réaliser les transformations entre les graphes de l'Eclipse Modeling Framework (EMF). Les principales propriétés de *TGG Interpreter* sont :

• Pour spécifier les règles TGG, un éditeur graphique est disponible avec un support d'édition sensible au méta-modèle
• Les instructions OCL sont utilisées pour des affectations de valeur ou pour les contraintes
• Les règles TGG sont renforcées par les relations de généralisation (héritage).

6.3.1.2 Définition des règles de TGG

L'idée générale dans la définition des règles de transformation est de créer des règles TGG. Pour notre proposition, nous avons adopté trois règles importantes à étudier. Ci-dessous leurs détails :

1. Règle "Statechart to RTmaude" (Axiome)

Pour démarrer la transformation, un noeud de départ dans le modèle d'entrée (Statechart) et une règle opérationnelle *sc2rtm* pour traduire ce noeud doivent être déterminés (figure 6.9). Une telle règle de départ valide sans aucun élément de contexte, c'est-à-dire aucune condition préalable, est appelée un *axiome*. Après la traduction du premier noeud, une stratégie est requise pour couvrir systématiquement tous les éléments du modèle d'entrée dans un ordre approprié. Dans notre axiome, le nom du module RT-Maude, dans le modèle de sortie, prend le même nom (*name*) que le Statechart.

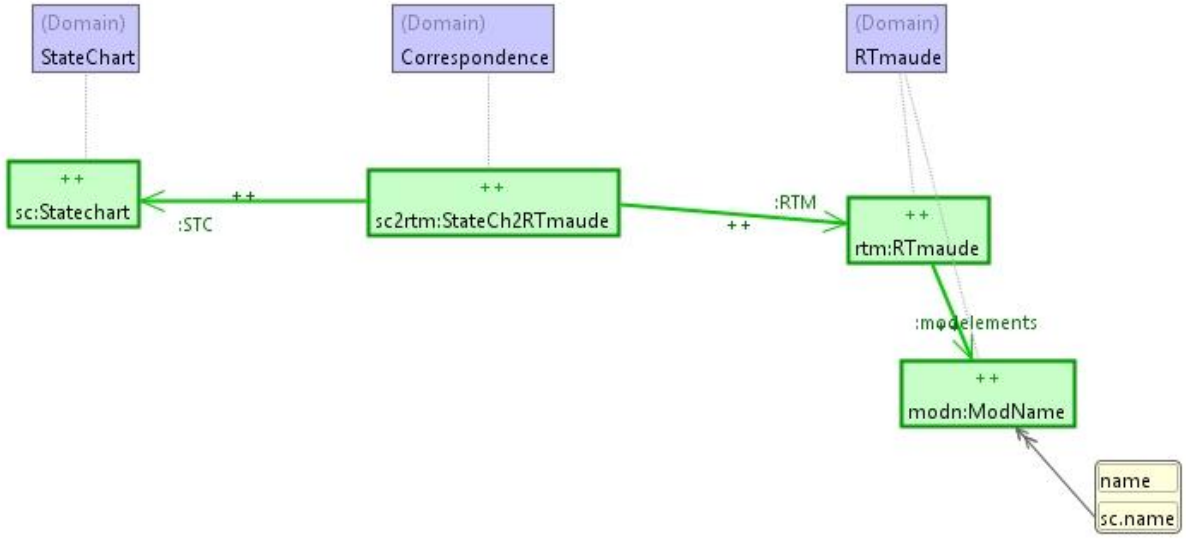


Figure 6.9. Règle *"Statechart to RTmaude"* (Axiome).

2. Règle *"Initial state to Module elements"*

L'état initial d'un diagramme d'états-transitions est formellement généré dans RT-Maude par une règle *InitialSt2ModElt* (figure 6.10) qui définit l'entête d'un module temporisé, qui contient une importation du module prédéfini (NAT-TIME-DOMAIN), et une variable de temps (nommée R). Nous définissons un type nommé *State* pour décrire le type de chaque état du diagramme d'états-transitions. De plus, "*State*" d'un système devrait être représenté par un terme de type *System*.

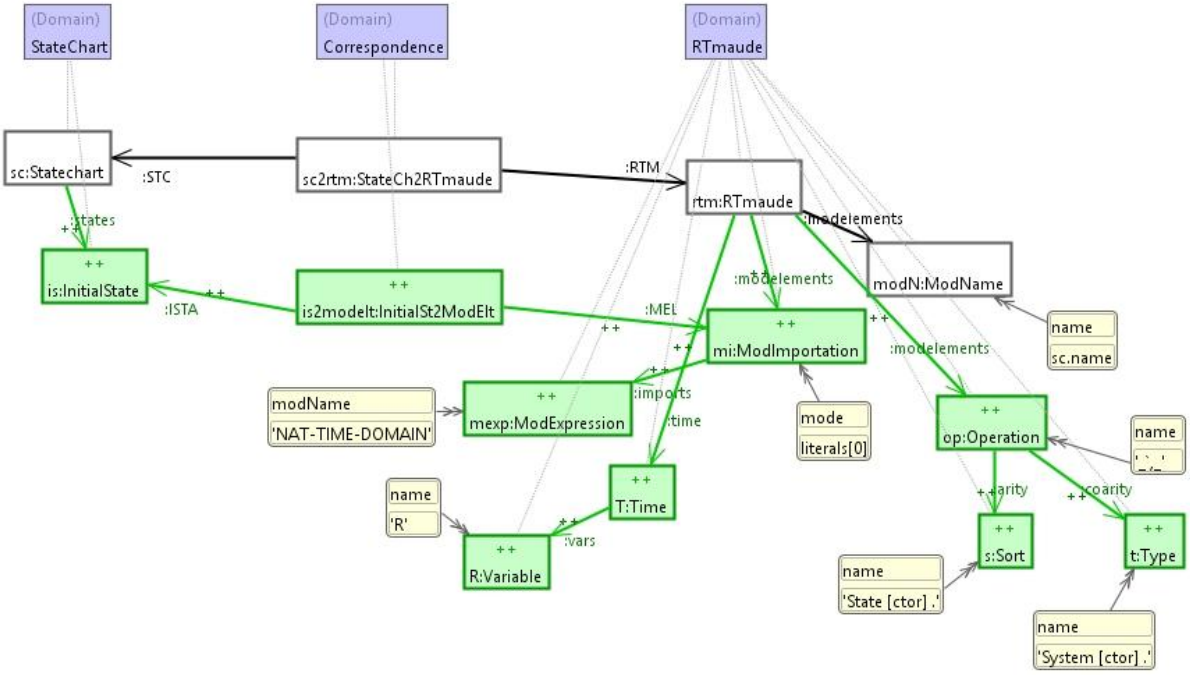


Figure 6.10. Règle *"Initial state to Module elements"*.

3. Règle *"Transition to Conditional rule"*

La règle *Tr2Crule* traduit chaque transition du diagramme d'états-transitions en une règle de réécriture qui spécifie les transitions atomiques du système (figure 6.11). Avant cela, nous créons une règle conditionnelle, pour l'état source *srcState*, dans laquelle la variable de temps

avance d'une unité (devient R + 1) sur l'état pendant que la condition de la transition, qu'elle est représentée par une flèche de l'état source vers l'état cible, n'est pas satisfaite. Cela signifie que les deux côtés LHS (Left Hand Side) et RHS (Right Hand Side) de la règle conditionnelle ont le même nom de l'état source. En outre, dans le côté droite RHS de la règle nous assignons R + 1 comme nom d'étiquette pour la variable de temps R du LHS. Nous utilisons l'opérateur de négation *not* pour exprimer que la condition est insatisfaite.

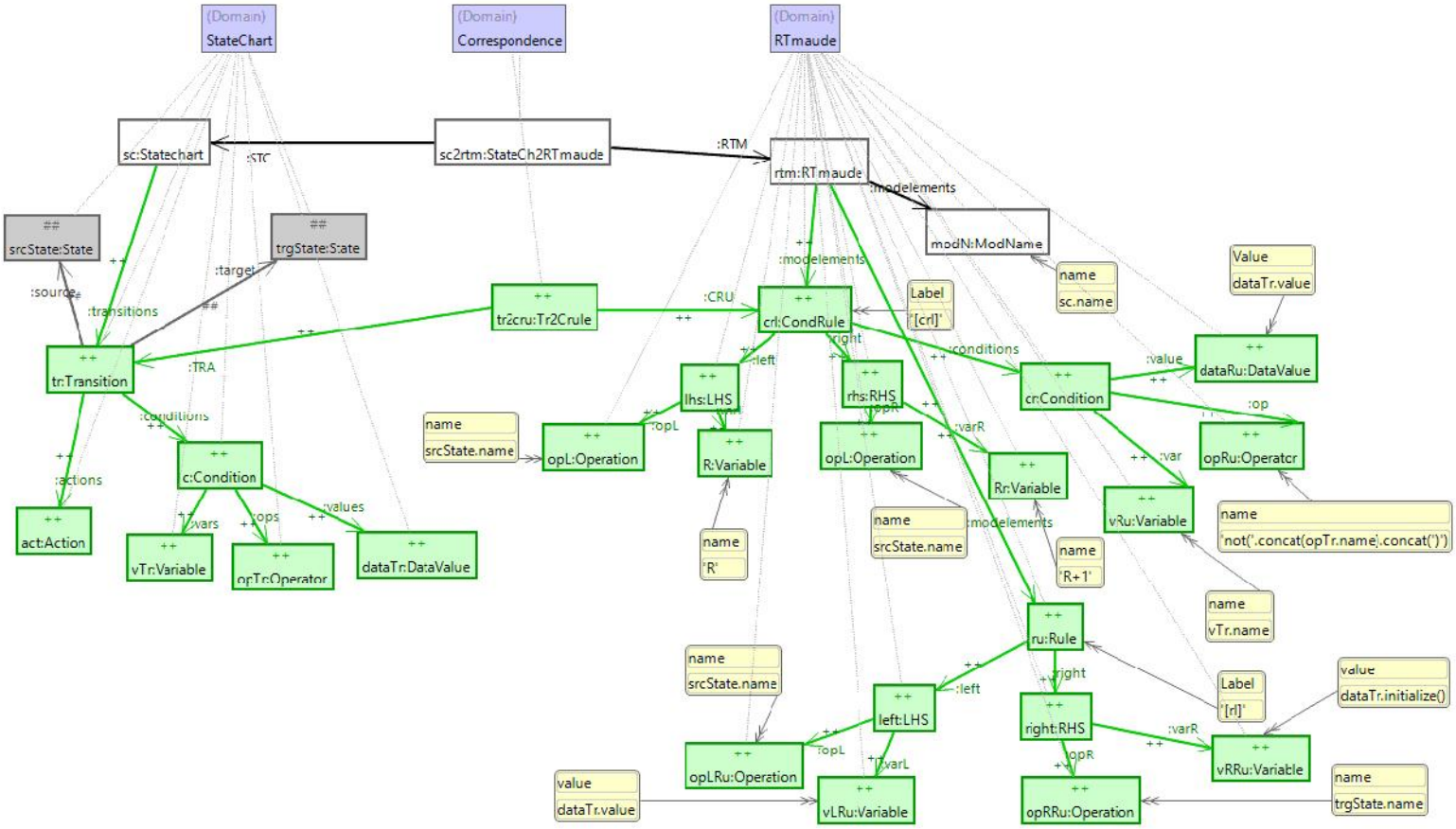


Figure 6.11. Règle "*Transition to Conditional rule*".

Pour chaque transition entre un état source *srcState* et un état cible *trgState*, une règle est générée qui définit dans LHS le nom de l'état de la source (représentant une *opération* dans la spécification RT-Maude) et la valeur de temps actuelle. Dans le côté droite RHS de la règle, nous trouvons le nom de l'état cible avec une valeur de temps initialisée. Notez que dans ce travail, nous traitons le cas particulier où chaque état doit initialiser la valeur de temps dans son entrée, comme une action associée à la condition. Nous utilisons des nœuds réutilisables pour l'état source et cible du diagramme d'états-transitions (nœuds à gauche dans la figure 6.11 en couleur gris) pour éliminer la multiplication de la même transformation du même état, ce qui conduit à une duplication des mêmes règles ou les mêmes règles conditionnelles.

6.3.2 Transformations de modèle à texte (M2T)

Dans le contexte de la génération de code, l'utilisation d'une transformation aurait pour objectif de transformer un modèle de haut-niveau d'abstraction en un second modèle de plus bas niveau d'abstraction (proche du code exécuté). De cette manière, on facilite la conception en épargnant à l'utilisateur les détails d'implémentation, en traduisant automatiquement ce modèle. Nous faisons appel pour cela à des transformations de type M2T que nous appliquons à des modèles

Statechart afin de générer du code compatible avec l'outil RT-Maude. Dans ce cas, on parle d'une transformation M2T endogène (le modèle et le code généré sont conforme au même méta-modèle RT-Maude) et verticale (le modèle et le code généré ne sont pas dans le même niveau d'abstraction).
Le projet Model to Text (M2T) se concentre sur la génération d'artefacts textuels à partir de modèles [265]. Une grande classe de transformations traduit les modèles en texte. Le texte peut être du code généré, d'autres modèles dans la syntaxe textuelle ou d'autres objets textuels tels que des rapports ou des documentations. Le texte est généralement généré à l'aide de modèles, une technique extrêmement bien établie (par exemple dans le développement web). Un *template* peut être considéré comme le texte cible avec des trous pour les parties variables. Les trous contiennent du méta-code (donc du code créant du code), qui est exécuté au moment de l'instanciation du Template pour calculer les parties variables (figure 6.12).

```
«IMPORT rtmaude»
«IMPORT xpand2 »
«IMPORT emf »
«EXTENSION template::GeneratorExtensions»
"---------------------------------- main -----------------------------------"
«DEFINE main FOR RTmaude»
«file("Code")»
«EXPAND Element FOREACH eContents»
«SaveString("endtm)")»
«ENDDEFINE»

«DEFINE Element FOR EObject»
Not Defined: «metaType.name»
«ENDDEFINE»

«DEFINE Element FOR ModName»
«SaveString("(tmod "+this.name+" is")»
«ENDDEFINE»

«DEFINE Element FOR ModImportation»
«SaveString (mode.toString()+" "+ imports.modName+" .")»
«SaveString("sort State . ")»
«ENDDEFINE»

«DEFINE Element FOR Operation»
«SaveString("op "+this.name+" : State Time -> System [ctor] .") »
«ENDDEFINE»

«DEFINE Element FOR Time»
«FOREACH vars AS c SEPARATOR " "»
«SaveString("var "+ c.name+" : Time .")»
«ENDFOREACH»
«ENDDEFINE»

«DEFINE Element FOR CondRule»
«FOREACH {}.add(this.left.opL.name).intersect({}.add(this.right.opR.name)) AS c
SEPARATOR " "»
         «SaveString("op "+ c.toString()+" : -> State [ctor] .")»
«ENDFOREACH»
   «SaveString("crl [crl] : {"+this.left.opL.name+", "+this.left.varL.name+"} =>
{"+this.right.opR.name+", "+
   this.right.varR.name+"} in time 1 if
"+this.conditions.var.name.first()+(switch(this.conditions.op.name.first()){case
'not(>=)' : ' < '
   case 'not(<=)' : ' > '  case 'not(>)' : ' <= '  case 'not(<)' : ' >= '  case
'not(=)' : ' <> '  case 'not(<>)' : ' = '
    default : 'unknown'})+this.conditions.value.Value.first()+" .")»
«ENDDEFINE»

«DEFINE Element FOR Rule»
«SaveString("rl [rl] : {"+ this.left.opL.name+", "+this.left.varL.value+"} =>
{"+this.right.opR.name+", "+this.right.varR.value+"} .")»
«ENDDEFINE»
```

(a) Le template de génération de code.

```
package template;
import java.io.*;
public class MyClasse {
      private static  String filename = "c://test/code.rtmaude";
      private static boolean existed = true;
      public static PrintWriter print1;
            public static void SaveString(String aString) {
                  try {
      print1 = new PrintWriter(new BufferedWriter (new FileWriter (filename,existed)));
      print1.println();
      print1.println(aString);
      print1.close();
            } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                                    }
                  System.out.println(aString);
            }
      public static void file(String name) {
               filename="c://test/"+name+".rtmaude";
            }
}
```

(b) La classe Java appelée.

Figure 6.12. Template Xpand pour la génération de code RT-Maude.

La prochaine étape de ce travail consiste à illustrer la génération de code avec le langage Xpand à partir des modèles EMF [262]. Le processus que nous allons suivre commencera en utilisant le métamodèle RT-Maude (précédemment défini dans la figure 6.7), en écrivant des templates de génération de code dans le fichier **« .xpt »** (figure 6.12(a)), en exécutant le générateur et enfin en ajoutant quelques contraintes (*checks*) . Dans notre cas, nous voulons appeler des méthodes Java (*SaveString* et *file*) à partir d'une expression. Cela peut être fait en fournissant une extension Java via une classe Java (figure 6.12(b)). A la fin de cette étape, nous devrions obtenir une génération de code, qui sera applicable à tout modèle de sortie RT-Maude de la réalisation de la transformation des règles TGG par l'interpréteur TGG de l'étape précédente.
Une fois que nous aurons obtenu la spécification formelle de notre système temps réel dans le module temporisé RT-Maude, la dernière étape consistera à tester l'exactitude (correctness) du système concerné en utilisant une vérification formelle. Cette dernière étape sera réalisée en utilisant l'outil de vérification de modèle Maude LTL model-checker.

6.4 Conclusion

Dans ce chapitre, Nous avons proposé une approche dirigée par les modèles pour la transformation de modèles basée sur l'utilisation des grammaires de graphes. Plus précisément, nous avons proposé une approche automatisée et des outils permettant de transformer le comportement dynamique des systèmes temps-réel exprimés en diagramme d'états-transitions (Statechart) vers leur équivalent en modèle RT-Maude et par la suite générer un code présente des spécifications RT-Maude inséré dans un module temporisé.
L'implantation d'approches IDM nécessite l'utilisation d'environnements dédiés tels que la plateforme Eclipse Modeling Framework (EMF). Cette plateforme nous a permis de mettre en œuvre les méta-modèles, les modèles et les transformations. Ensuite, nous avons eu recours à l'outil TGG Interpreter, intégré dans l'environnement EMF, pour définir les règles de transformation TGG. La phase finale de la génération de code est réalisée à l'aide du langage Xpand.
Pour valider notre approche, il est nécessaire de montrer les résultats d'application à travers des études de cas sur le plan d'implémentation. Ceci fera l'objet du chapitre suivant.

CHAPITRE 7

Études de cas

Sommaire

7.1 Introduction

Dans ce chapitre, nous allons appliquer notre approche présentée dans le chapitre précédent sur des systèmes modélisée par des diagrammes d'états-transitions. Premièrement, la transformation des diagrammes d'états-transitions vers des modèles RT-Maude se déroulera sous l'environnement de développement Eclipse en utilisant l'outil TGG interpreter. Ensuite nous avons eu recours au langage Xpand pour générer des modules temporisés correspondant à chaque système. Finalement, nous lancerons un processus de vérification formelle des spécifications générés à l'aide de propriétés du système, de nature désirables et non désirables, énoncées à l'aide de la logique temporelle linéaire (LTL).
Ce processus, aux premiers abords, peut paraître abstrait et difficile à comprendre. Afin de mieux l'illustrer, trois études de cas sont présentées dans ce chapitre : un feu de circulation tricolore, un thermostat et un robot industriel intégré dans une usine. Pour chacun de ces exemples, la même structure sera suivie afin d'illustrer le processus de translation et de vérification dans son ensemble.

7.2 Études de cas

Pour démontrer l'applicabilité de l'approche proposée, nous montrons maintenant comment des exemples de systèmes temps réel sont transformés en une description RT-Maude à vérifier ultérieurement en utilisant Maude LTL Model-Checker. Notons que la mise en œuvre de notre approche et toutes les expérimentations réalisées sont menées par un laptop Toshiba avec processeur Intel (R) Core (TM) i7 CPU 2.30GHz et 4Go de RAM et sous un système d'exploitation Windows 8 (64 bit).

7.2.1 Feu de circulation tricolore

7.2.1.1 Modélisation

Considérons un système de feu de circulation tricolores [263] composé de trois lumières colorées : rouge, vert et jaune. Une seule des couleurs est active dans un instant donné. Ce système fonctionne indéfiniment dans le temps et le couleur de feu change suivant le cycle : les couleurs s'allument dans l'ordre rouge, vert, jaune, puis rouge à nouveau, et ainsi de suite. Nous utilisons le diagramme de l'états-transitions pour modéliser le comportement de notre système en fonction des différents états qu'un système peut avoir lorsqu'il réagit aux stimuli.

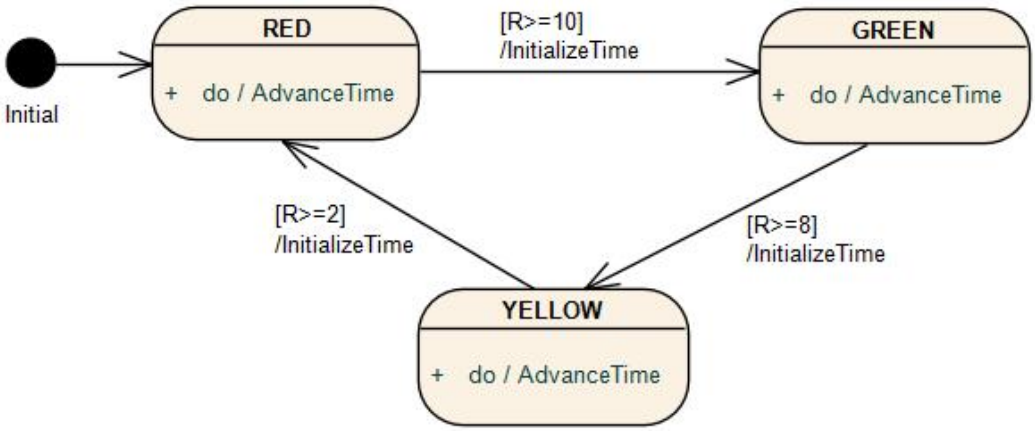


Figure 7.1. Diagramme d'états-transitions décrivant le comportement d'un feu de circulation tricolore.

Comme illustré dans la figure 7.1, le système a trois états. Chaque état a une action *Do* appelée *AdvanceTime* pour exprimer le temps qui passe. Le système passe d'abord en état *ROUGE*. Il passe de l'état *ROUGE* à l'état *VERT* lorsque le déclencheur, associé à la transition entre les

deux états, est actionné et que la garde est satisfaite (temps supérieur ou égal à 10). Cette transition inclut également un effet qui est une action *InitializeTime* (réinitialisation du temps). L'effet est exécuté lorsque la transition se déclenche. Pour les autres transitions, la même description peut être vue. Lorsque le déclencheur, en transition entre l'état *JAUNE* et l'état *ROUGE*, est activé, le système revient au premier état *ROUGE*.

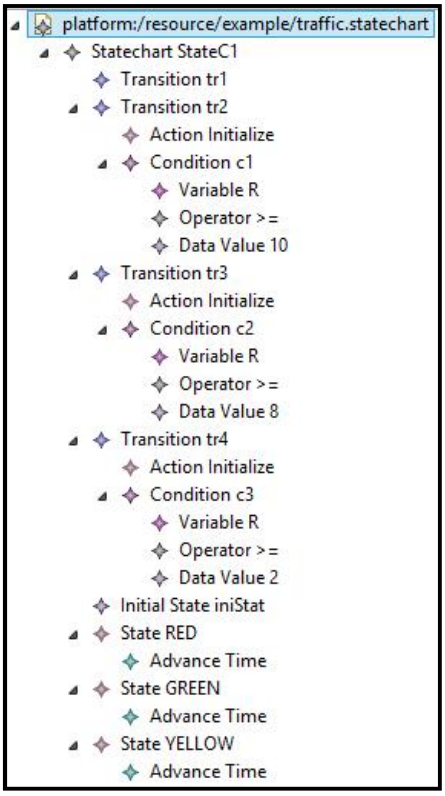


Figure 7.2. Modèle EMF d'un feu de circulation tricolore entré par l'utilisateur.

7.2.1.2 L'exécution de grammaire

La phase d'exécution dans TGG Interpreter se résume à créer une configuration du programme de transformation. Une fois cette dernière est créée, il est exécuté sur le modèle source afin d'obtenir le modèle cible et une trace de l'exécution des règles est générée.
D'abord, un modèle d'entrée correspond à notre système doit être élaboré. Ce modèle présente la syntaxe concrète du système et doit être entré par l'utilisateur (figure 7.2).

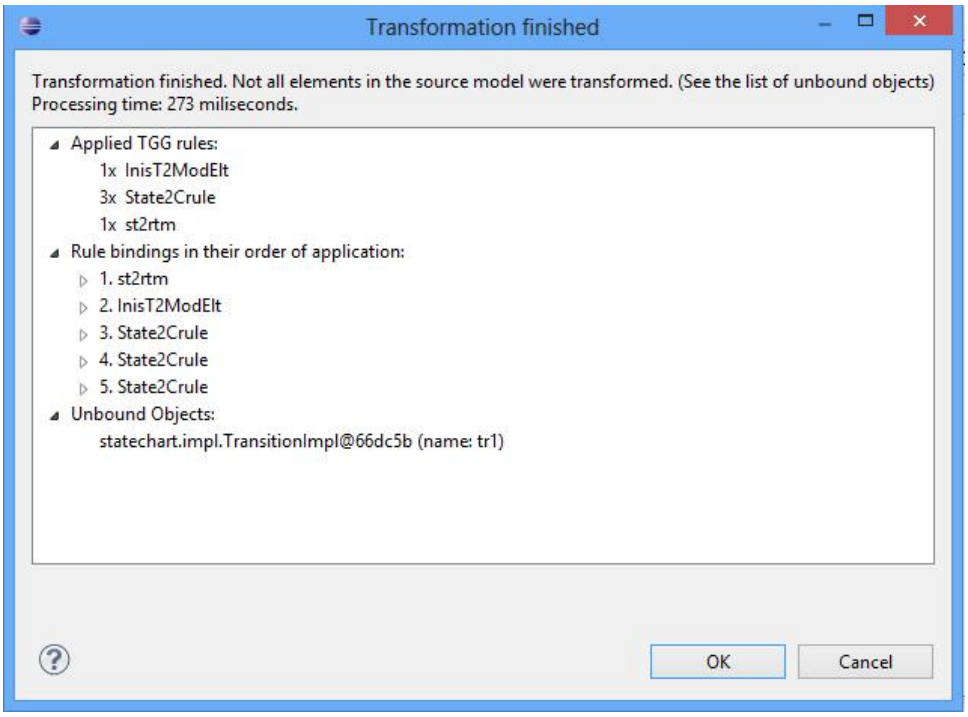


Figure 7.3. Résultat de l'exécution de la transformation.

Après avoir modélisé notre exemple de système de feux de circulation, nous sommes en mesure d'automatiser la génération du modèle de spécifications RT-Maude en exécutant notre transformation TGG. Le modèle présenté à la figure 7.2 est utilisé comme fichier de modèle source. La transformation peut être exécutée par une simple clic-droit sur un fichier de configuration de TGG Interpreter. Deux fichiers sont générés en tant que modèles de sortie, fichier intermédiaire « .xmi » et le fichier de modèle cible avec l'extension « .rtmaude » (figure 7.4). L'exécution de la transformation résultante est illustrée à la figure 7.3. Tous les éléments du modèle source ont été transformés. Selon notre contexte d'exemple, la règle *State2Crule* a été appliquée trois fois. La transition initiale a été considérée comme un objet non lié (*unbound object*). Notez que, dans cette première étude de cas, nous considérons que le code RT-Maude est sensible aux espaces vierges.

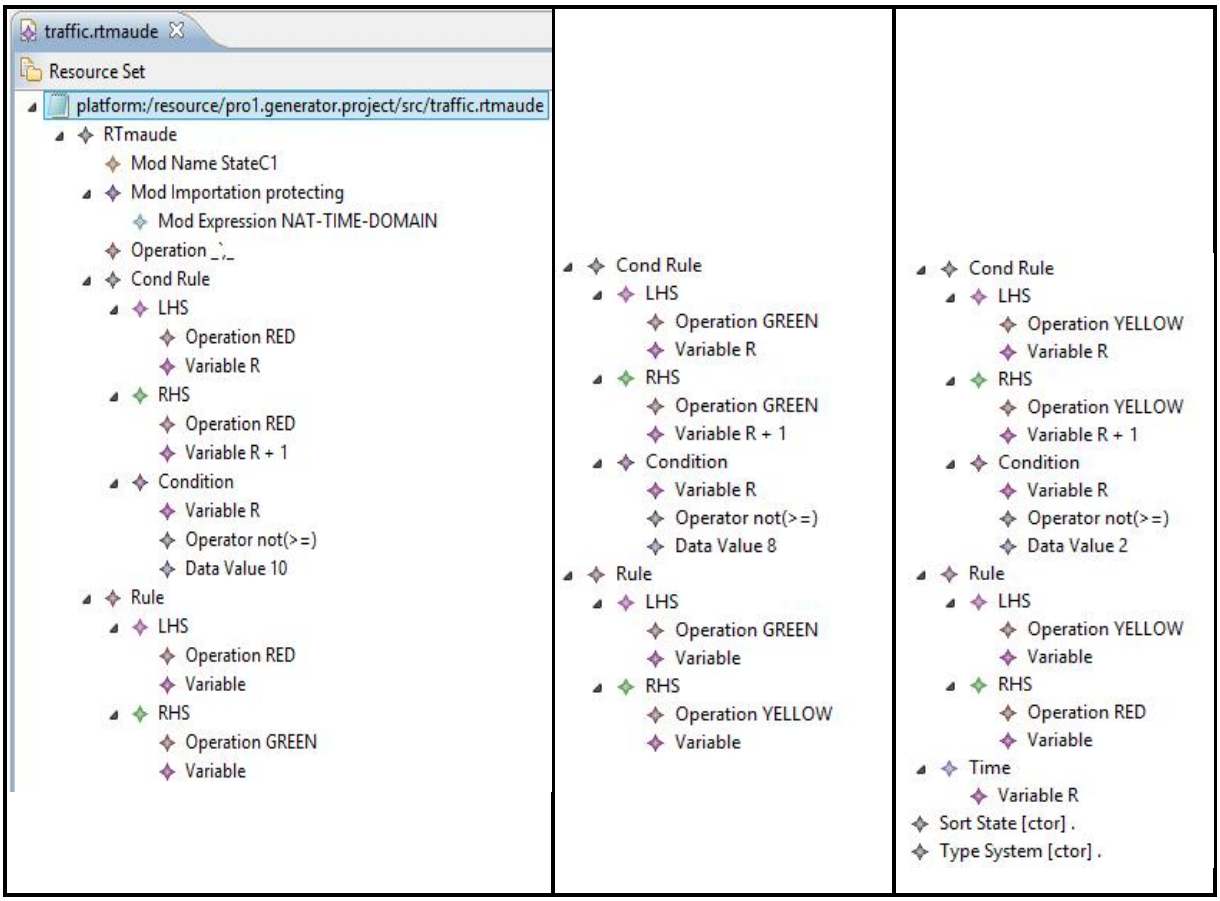


Figure 7.4. Modèle de sortie de la transformation.

7.2.1.3 Génération de code

La deuxième transformation est une transformation Model-to-Text, qui prend un modèle RT-Maude et génère un code au format RT-Maude (inséré dans un module temporisé). Cette étape peut être accomplie en utilisant le template Xpand, décrit dans le chapitre précédent, à partir d'un Workflow. Après l'exécution du Workflow, le code généré devrait être acceptable par l'outil RT-Maude (figure 7.5). Comme remarque concernant le code généré, il faut noter que la déclaration de la variable de temps R à la fin du module temporisé est autorisée dans RT-Maude.

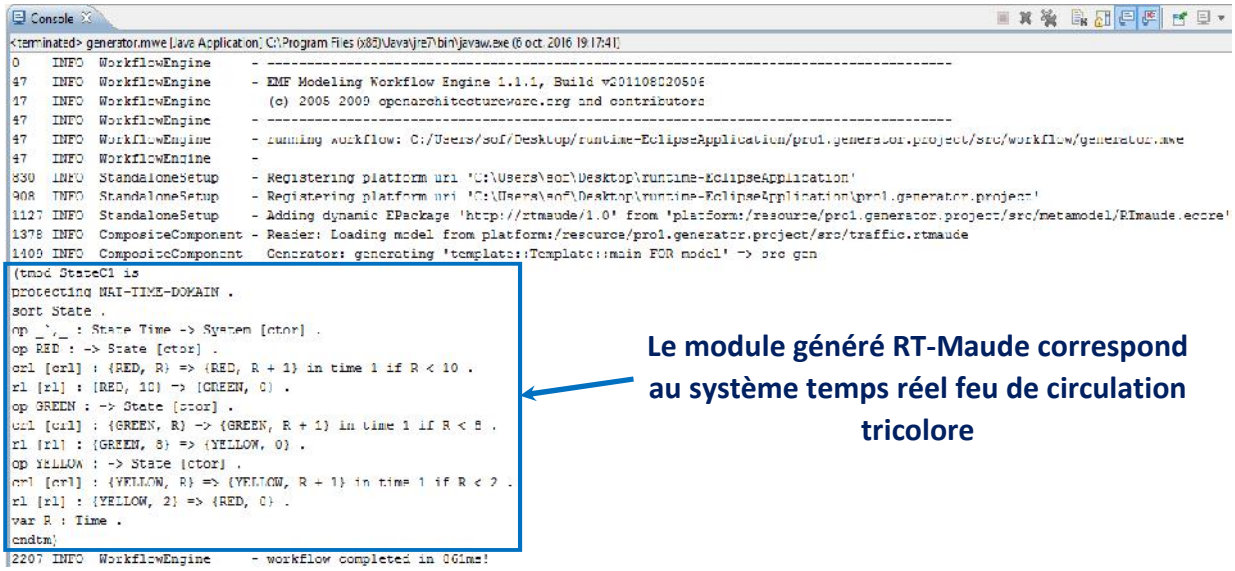


Figure 7.5. Le code final généré (module temporisé RT-Maude).

7.2.2 Thermostat

Notre deuxième étude de cas consiste en un modèle simplifié d'un thermostat fonctionne en allumant et éteignant un appareil de chauffage afin de maintenir une température entre 62 et 74 degrés [266]. Dans le mode de contrôle OFF (chauffage éteint), l'appareil de chauffage est non fonctionnel et la température diminue d'un degré par unité de temps. Dans le mode de contrôle ON (chauffage allumé), l'appareil de chauffage est fonctionnel et la température augmente de deux degrés par unité de temps. Initialement le thermostat est désactivé et la température égale à 68 degrés. La figure 7.6 décrit un tel thermostat par un diagramme d'états-transitions.

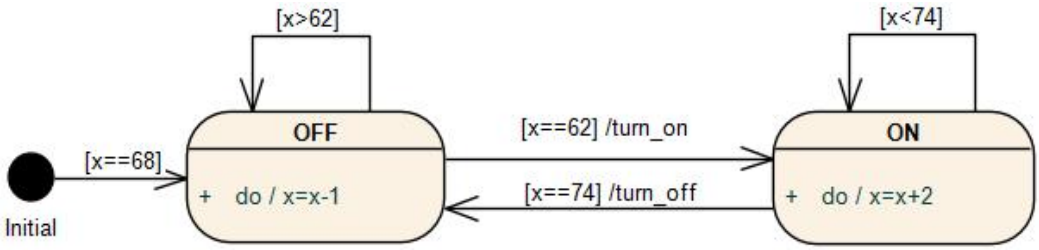


Figure 7.6. Diagramme d'états-transition du Thermostat.

Pour suivre notre approche, il faut d'abord modéliser graphiquement notre système par un diagramme d'états-transitions (figure 7.6). Ensuite, après l'application des étapes de transformation et de génération de code, une spécification en RT-Maude de thermostat est généré dans le module temporisé suivant :

```
(tmod THERMOSTAT is
protecting POSRAT-TIME-DOMAIN .
sort ThermoState .
ops on off : -> ThermoState [ctor] .
op _‘,_ : ThermoState NNegRat -> System [ctor] .
rl [turn-on] : off , 62 => on , 62 .
rl [turn-off] : on , 74 => off , 74 .
vars R R’ : Time .
crl [tick-on] :
```

```
    {on, R} => {on, R + (2 * R')} in time R' if R' <= ((74 - R) / 2) [nonexec] .
crl [tick-off] :
    {off, R} => {off, R - R'} in time R' if R' <= (R - 62) [nonexec] .
endtm)
```

7.2.3 Robot industriel

Nous considérons ici une simple usine de fabrication automatique, ce cas est inspiré de Daws et Yovine [267]. L'usine de fabrication que nous considérons se compose d'un tapis roulant qui se déplace de gauche à droite, une zone de traitement (processing station), et deux robots qui font déplacer les boîtes entre la station et le tapis roulant. Le premier robot appelé D-Robot prend une boîte de la station, chaque 10 secondes, et la place à l'extrémité gauche de tapis roulant. La deuxième robot appelé G-Robot prend la boîte de l'extrémité droite du tapis roulant et la transfère à la station à traiter (figure 7.7).

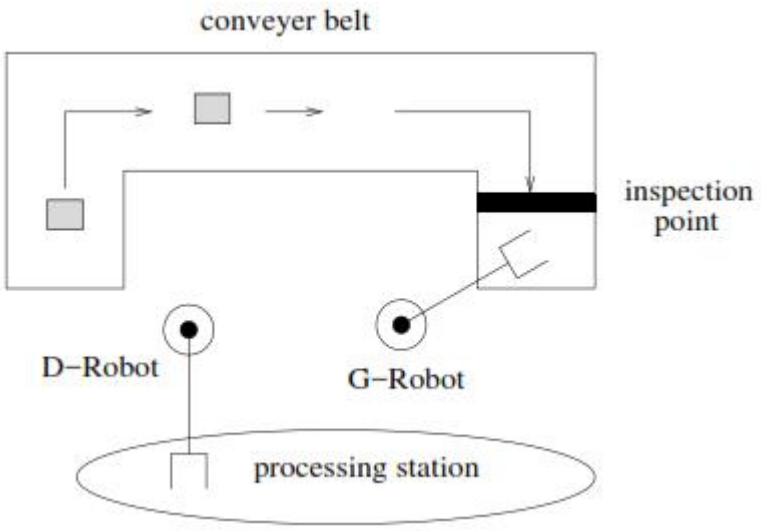


Figure 7.7. Une usine simple.

En particulier, nous nous sommes intéressés de comportement du D-Robot. Initialement, le robot attend (WAIT) jusqu'à ce qu'une boîte soit prête après 10 secondes. Ensuite, il prend la boîte (PICK), tourne à droite (TURN@R) et met la boîte (PUT) sur le tapis roulant. A la fin, le robot tourne à gauche (TURN@L) et revient à sa position initiale. Le temps de déplacement vers la droite et vers la gauche est considéré le même est égale à 6 secondes. La même chose pour les opérations de prise et de remise d'une boite qui ont effectué dans une durée de 2 secondes.

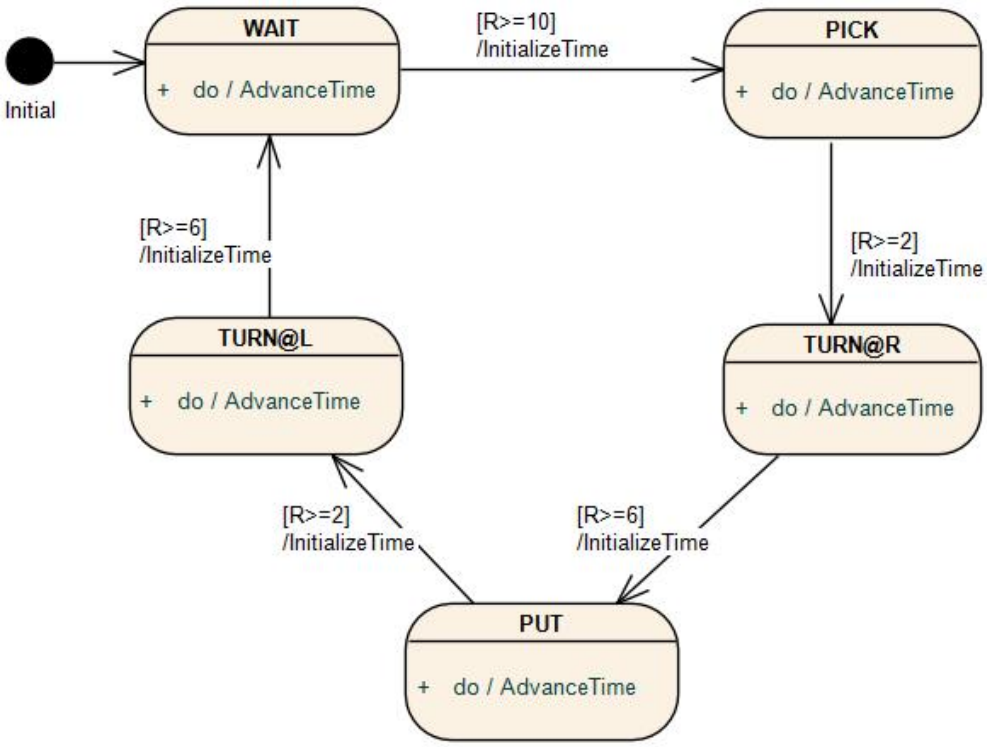


Figure 7.8. Diagramme d'états-transition du robot industriel.

Le but de la modélisation de notre système par un diagramme d'états-transitions, comme il est illustré dans la figure 7.8, et de préparer un modèle source pour le transformer vers un modèle cible RT-Maude. La phase finale est d'obtenir un module temporisé acceptable par l'outil RT-Maude et décrivant le comportement du robot industriel. Un extrait de telles spécifications est représenté dans le module généré suivant :

```
(tmod Robot is
protecting NAT-TIME-DOMAIN .
sort State .
op _`,_ : State Time -> System [ctor] .
var R : Time .
op WAIT : -> State [ctor] .
crl [crl] : {WAIT, R} => {WAIT, R + 1} in time 1 if R < 10 .
rl [rl] : {WAIT, 10} => {PICK, 0} .
op PICK : -> State [ctor] .
crl [crl] : {PICK, R} => {PICK, R + 1} in time 1 if R < 2 .
rl [rl] : {PICK, 2} => {TURN@R, 0} .
op TURN@R : -> State [ctor] .
crl [crl] : {TURN@R, R} => {TURN@R, R + 1} in time 1 if R < 6 .
rl [rl] : {TURN@R, 6} => {PUT, 0} .
op PUT : -> State [ctor] .
crl [crl] : {PUT, R} => {PUT, R + 1} in time 1 if R < 2 .
rl [rl] : {PUT, 2} => {TURN@L, 0} .
op TURN@L : -> State [ctor] .
crl [crl] : {TURN@L, R} => {TURN@L, R + 1} in time 1 if R < 6 .
rl [rl] : {TURN@L, 6} => {WAIT, 0} .
endtm)
```

7.3 Analyse et vérification formelle

L'analyse et la vérification formelle est une étape critique pour fiabiliser le comportement de systèmes étudiés. Notre approche de développement dirigée par les modèles intègre donc l'utilisation de méthodes formelles dès l'étape de spécification afin de prouver que celle-ci est correcte vis-à-vis du comportement de système attendu. LTL Model checker de RT-Maude est un model-checker assez puissant, expressif et qui permet de vérifier certaines propriétés temporisées que d'autres model-checkers ne le permettent pas. Nous utilisons Maude version 2.6 (pour Windows) et Real-Time Maude version 2.3 (voir figure 7.9) pour analyser et vérifier les systèmes décrits dans la section précédente.

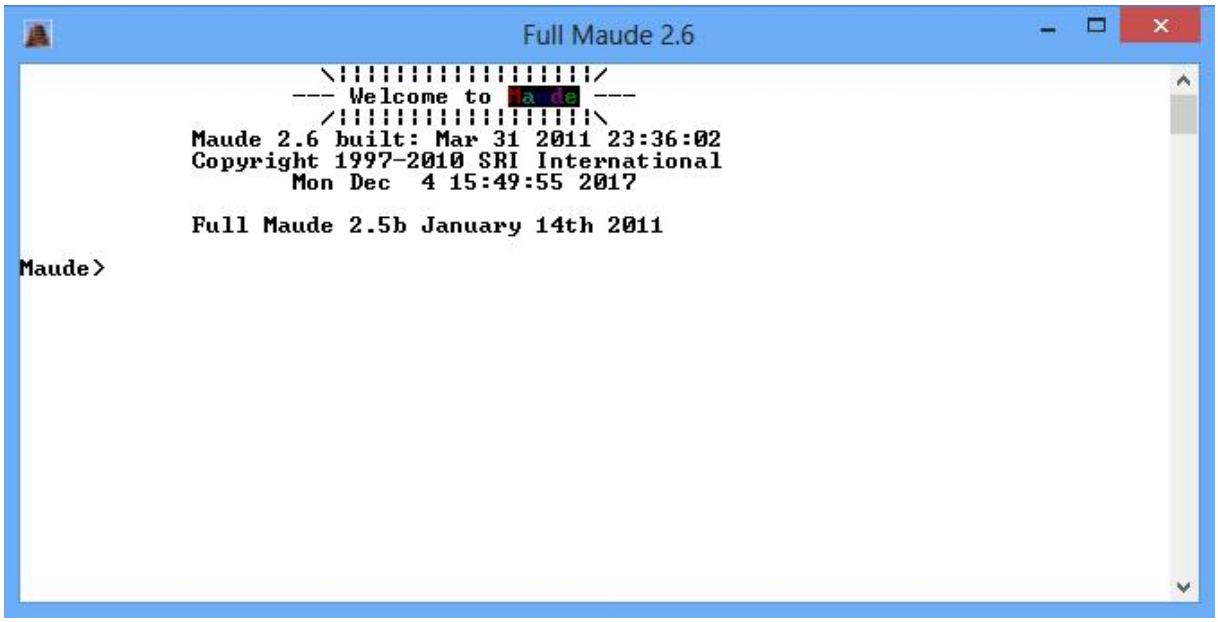


Figure 7.9. Exécution de Maude.

7.3.1 Feu de circulation tricolore

7.3.1.1 Réécriture temporisée (Timed Rewriting)

Les commandes *timed rewrite* et *timed fair rewrite* de RT-Maude peuvent être utilisées pour simuler un comportement (c'est-à-dire, une séquence de pas de réécriture) d'un système commençant par un état initial donné. La réécriture peut être bornée dans le temps, simuler un comportement jusqu'à une certaine durée et / ou être limitée par le nombre de pas de réécriture à effectuer. Les commandes de réécriture temporisée utilisent le mode *tick* actuel pour traiter les règles de *tick* avec une augmentation de temps non déterministe. Le résultat d'une commande de réécriture temporisée est le dernier état du comportement simulé avec un indice indiquant la durée totale de la séquence de réécriture [136]. Nous devons définir une stratégie pour avancer le temps d'exécution du système. Nous pouvons choisir d'augmenter le temps d'une unité de temps, puis de simuler le système. Les commandes entrées par l'utilisateur sont précisées par un texte en italique.

Maude> (trew {RED, 0} in time < 100 .)
rewrites: 3784 in 6286706272ms cpu (36ms real) (0 rewrites/second)
Timed rewrite {RED,0} in StateC1 with mode deterministic time increase in time < 100
Result ClockedSystem :
{YELLOW,1} in time 99

7.3.1.2 Recherche temporisée (Timed Search)

Si nous cherchons, par exemple, les trois premières occasions où les états avec la valeur de temps 2 peuvent être atteints à partir de l'état {RED, 0}, nous avons obtenu trois solutions aux temps 2,12 et 20. Pour chaque temps écoulé, les états du système sont RED, GREEN et YELLOW respectivement :

```
Maude> (tsearch [3] { RED, 0 } =>* { S:State, 2} with no time limit .)
rewrites: 5578 in 6286706272ms cpu (39ms real) (0 rewrites/second)
Timed search [3] in StateC1
      {RED,0} =>* {S:State,2}
with no time limit and with mode deterministic time increase :
Solution 1
S:State --> RED ; TIME_ELAPSED:Time --> 2
Solution 2
S:State --> GREEN ; TIME_ELAPSED:Time --> 12
Solution 3
S:State --> YELLOW ; TIME_ELAPSED:Time --> 20
```

7.3.1.3 Recherche non temporisée (Untimed Search)

Nous pouvons utiliser la recherche non temporisée pour vérifier les propriétés sans limite de temps. En utilisant la commande de recherche non temporisée, nous pouvons montrer que notre système ne se bloque jamais en recherchant des états qui ne peuvent pas être réécrits encore (absence de blocage) :

```
Maude> (utsearch {RED, 0} =>! GS:GlobalSystem .)
rewrites: 2475 in 6286706272ms cpu (41ms real) (0 rewrites/second)
Untimed search in StateC1
      {RED,0} =>! GS:GlobalSystem
with mode deterministic time increase :
No solution
```

En outre, nous pouvons garantir que, dans notre système, nous n'aurons jamais le cas où l'état GREEN est suivi directement par l'état RED :

```
Maude> (utsearch {GREEN, N:Nat} =>1 {RED, N:Nat} .)
rewrites: 2515 in 6286706272ms cpu (25ms real) (0 rewrites/second)
Untimed search in StateC1
      {GREEN,N:Nat} =>1 {RED,N:Nat}
with mode deterministic time increase :
No solution
```

7.3.1.4 Vérification via le model-checker LTL

L'utilisation de la vérification de modèle LTL model-checker a pour objectif de prouver qu'il y aura toujours un état non-mauvais (quelque chose de mauvais n'arrive jamais). Le module suivant définit la proposition *'badState'* qui s'applique dans tous les états où le système est en cas de dysfonctionnement :

```
(tmod CHECK-TRAFFIC is
  including TIMED-MODEL-CHECKER .
  protecting StateC1 .
  op badState : -> Prop .
  eq {S:State, T:Nat} |= badState =
      (T:Nat < 0) or ((S:State == RED) and (T:Nat > 10)) or
((S:State == GREEN) and (T:Nat > 8)) or ((S:State == YELLOW) and
(T:Nat > 2)).
endtm)
```

Tout d'abord, nous vérifions si le système fonctionne toujours correctement, au moins, dans le temps 100 :

Maude> (mc {RED, 0} |=t [] ~ badState in time < 100 .)
rewrites: 7857 in 6286706272ms cpu (117ms real) (0 rewrites/second)
Model check{RED,0} |=t[]~ badState in CHECK-TRAFFIC in time < 100 with mode deterministic time increase
Result Bool :
true

Une commande de vérification de modèle non bornée dans le temps garantit que le comportement de système est toujours convenable :

Maude> (mc {RED, 0} |=u [] ~ badState .)
rewrites: 4897 in 6286706272ms cpu (42ms real) (0 rewrites/second)
Untimed model check {RED,0} |=u []~ badState in CHECK-TRAFFIC with mode deterministic time increase
Result Bool :
true

La commande suivante vérifie si le système atteindra l'état {GREEN, 3} dans chaque comportement possible commençant par l'état {RED, 0}, le résultat sera que la propriété est satisfaite :

Maude> (check {RED, 0} |= <> {GREEN, 3} with no time limit .)
rewrites: 5500 in 6286706272ms cpu (21ms real) (0 rewrites/second)
Check {RED,0} |= <> {GREEN,3} in CHECK-TRAFFIC with no time limit with mode
deterministic time increase :
Result: the property holds.

Aussi, La commande suivante vérifie si le système augmente le temps depuis l'état initial {RED, 0} jusqu'à ce qu'il atteigne la valeur de temps 12, le résultat est que la propriété n'est pas satisfaite car la valeur maximale de temps à atteindre dans notre système est 10, un contre-exemple soit généré :

Maude> (check {RED, 0} |= {S:State, R:Time} until {S:State, 12} in time < 100 .)
rewrites: 13705 in 6286706272ms cpu (184ms real) (0 rewrites/second)
Check {RED,0} |= {S:State,R:Time} until {S:State,12} in CHECK-TRAFFIC in time <
100 with mode deterministic time increase :

Result: the property does not hold. Counterexample:
{YELLOW,2} in time 100

7.3.2 Thermostat

7.3.2.1 Réécriture temporisée (Timed Rewriting)

Dans notre étude de cas, nous pouvons tester le système de thermostat en définissant d'abord un mode de *tick*, disons en mode "par défaut" dans lequel le temps est toujours avancé par quatre unités de temps, par exemple, puis on simule le comportement du thermostat avant 100 unités de temps :

Maude> (set tick def 4 .)
Maude> (trew {on, 64} in time < 100 .)
rewrites: 4157 in 6286706272ms cpu (56ms real) (0 rewrites/second)
Timed rewrite {on,64} in THERMOSTAT with mode default time increase 4 in time
< 100
Result ClockedSystem :
{off,70} in time 99

Ce qui donne une température dans la gamme désirée. La durée totale du comportement simulé n'est pas un multiple de 4, car la durée maximale possible de passage de temps (*time elapse*) peut avoir été inférieur à 4 dans certains cas.

7.3.2.2 Recherche temporisée (Timed Search)

Pour analyser le système de thermostat, un mode de *tick* doit être entré en raison de son augmentation de temps non déterministe. La commande suivante augmentera le temps de 1 à chaque pas de réécriture :

Maude> (set tick def 1 .)

Bien que la température puisse réellement atteindre 71,5 degrés, par exemple, ceci ne sera pas trouvé lors d'une recherche car cet état n'est pas rencontré lorsque le temps avance toujours d'une unité de temps :

Maude> (tsearch {on, 66} => {X:ThermoState, 143/2} in time < 1000 .)*
rewrites: 25320 in 6286706272ms cpu (92ms real) (0 rewrites/second)
Timed search in THERMOSTAT
{on,66} =>* {X:ThermoState,143/2}
in time < 1000 and with mode default time increase 1 :
No solution

Cependant, si nous définissons l'avancement de temps par défaut par 1/2 unité de temps, l'état décrit ci-dessus doit être rencontré :

Maude> (set tick def 1/2 .)
rewrites: 280 in 6286706272ms cpu (3ms real) (0 rewrites/second)
Tick mode set to default mode

Maude> (tsearch [2] {on, 66} => {X:ThermoState, 143/2} in time < 1000 .)*
rewrites: 8222 in 6286706272ms cpu (64ms real) (0 rewrites/second)

```
Timed search [2] in THERMOSTAT
      {on,66} =>* {X:ThermoState,143/2}
in time < 1000 and with mode default time increase 1/2 :
Solution 1
TIME_ELAPSED:Time --> 13/2 ; X:ThermoState --> off
Solution 2
TIME_ELAPSED:Time --> 49/2 ; X:ThermoState --> off
```

Par la commande suivante, on doit s'assuré que la température de thermostat n'atteindra pas une température indésirable (hors de l'intervalle [62,74]) dans une durée de temps égale à 1000 :

```
Maude> (tsearch [1] {on, 66} =>* {X:ThermoState, R:Time}
> such that R:Time < 62 or R:Time > 74
> in time <= 1000 .)
rewrites: 41440 in 6286706272ms cpu (166ms real) (0 rewrites/second)
Timed search [1] in THERMOSTAT
      {on,66} =>* {X:ThermoState,R:Time}
in time <= 1000 and with mode default time increase 1 :
No solution
```

7.3.2.3 Recherche non temporisée (Untimed Search)

En utilisant la commande de recherche non temporisée, nous pouvons montrer que notre système ne se bloque jamais en recherchant des états qui ne peuvent pas être réécrits encore (*No deadlock*) :

```
Maude> (utsearch {X:ThermoState, R:Time} =>! GS:GlobalSystem .)
rewrites: 2739 in 6286706272ms cpu (37ms real) (0 rewrites/second)
Untimed search in THERMOSTAT
      {X:ThermoState,R:Time} =>! GS:GlobalSystem
with mode deterministic time increase :
No solution
```

D'après la description du thermostat, le changement entre les états ON et OFF est conditionné par la mesure de température actuelle. La commande suivante nous permet de connaitre qu'il n'y aura pas un changement direct entre les états de thermostat par un seul pas de réécriture :

```
Maude> (utsearch {off, N:Nat} =>1 {on, N:Nat} .)
rewrites: 3096 in 6286706272ms cpu (23ms real) (0 rewrites/second)
Untimed search in THERMOSTAT
      {off,N:Nat} =>1 {on,N:Nat}
with mode default time increase 1 :
No solution
```

7.3.2.4 Vérification via le model-checker LTL

RT-Maude fournit une procédure semi-décisionnelle pour les systèmes à temps discret utilisant la recherche en largeur (*breadth-first search*). Ces propriétés sont [136] :

• Les propriétés de vivacité (*Liveness*) de la forme : <> *pattern* **such that** *cond*, où *pattern* est une entité de rechercher (éventuellement temporisé). Cette propriété est satisfaite si un état

correspondant à l'entité peut être atteint dans chaque comportement possible à partir de l'état initial (avec la stratégie choisie pour l'avancement du temps).

• Les propriétés "*Until*" de la forme : (*pattern1* **such that** *cond1*) **until** (*pattern2* **such that** *cond2*), où *pattern1* et *pattern2* sont (éventuellement temporisée) des entités de recherche. Une telle propriété est satisfaite si dans chaque comportement de l'état initial:

- un état correspondant à *pattern2* et satisfaisant *cond2* sera atteint; et
- chaque état dans le calcul correspond à *pattern1* et satisfaisant *cond1*, jusqu'à ce que le premier *pattern* correspondant au *pattern2* et satisfaisant *cond2* soit rencontré.

• Propriétés "*Until-stable*" de la forme : (*pattern1* **such that** *cond1*) **untilStable** (*pattern2* **such that** *cond2*). Une telle propriété est satisfaite si (*pattern1* **such that** *cond1*) **until** (*pattern2* **such that** *cond2*) soit maintenu, et *pattern2* **such that** *cond2* soit stable dans le sens où une fois qu'il se maintient dans un comportement, il continuera à tenir dans tous les états suivants.

Pour vérifier notre système, nous illustrons l'utilité de la commande intégrée *check* dans le module THERMOSTAT. La commande suivante vérifie si le thermostat atteindra l'état {off, 63} dans chaque comportement possible commençant de l'état {on, 62} :

```
Maude> (check {on, 62}  |= <> {off, 63} with no time limit .)
rewrites: 4341 in 6286706272ms cpu (36ms real) (0 rewrites/second)
Check {on,62} |= <> {off,63} in THERMOSTAT with no time limit with mode default
   time increase 1 :
Result: the property holds.
```

La commande suivante vérifie si le thermostat atteint une température supérieure à 74 degrés lorsque le chauffage est allumé, le résultat est que la propriété n'est pas satisfaite et un contre-exemple soit généré :

```
Maude> (check {on, 62}  |= <> {X:ThermoState, R:Time} such that X:ThermoState = on /\
R:Time > 74 in time <= 100 .)
rewrites: 13093 in 6280078279ms cpu (108ms real) (0 rewrites/second)
Check {on,62} |= <> {X:ThermoState,R:Time} in THERMOSTAT in time <= 100 with
   mode default time increase 1 :
Result: the property does not hold. Counterexample:
     {off,69} in time 101
```

La propriété suivante montre que le thermostat augmente la température de l'état initial {on, 62} jusqu'à ce qu'il atteigne 75 degrés, la propriété ne sera pas vérifiée car la température maximale à atteindre dans le thermostat est 74 degrés. Comme pour toutes les propriétés non vérifiées, un contre-exemple sera généré :

```
Maude> (check {on, 62}  |= {X:ThermoState, R:Time} until {on, 75} in time < 100 .)
rewrites: 12403 in 6286706272ms cpu (202ms real) (0 rewrites/second)
Check {on,62} |= {X:ThermoState,R:Time} until {on,75} in THERMOSTAT in time <
   100 with mode default time increase 1 :
Result: the property does not hold. Counterexample:
     {off,70} in time 100
```

Il ne devrait pas être possible que la température du thermostat reste à 74 degrés après que cette température est atteinte car l'appareil de chauffage s'éteint automatiquement après avoir atteint 74 degrés, un contre-exemple soit généré :

Maude> (check {on, 62} |= {X:ThermoState, R:Time} untilStable {on, 74} in time < 100 .)
rewrites: 5220 in 6286706272ms cpu (25ms real) (0 rewrites/second)
Check {on,62} |= {X:ThermoState,R:Time} untilStable {on,74} in THERMOSTAT in
time < 100 with mode default time increase 1 :
Result: the property does not hold. Counterexample:
{off,74} in time 6

7.3.3 Robot industriel

7.3.3.1 Réécriture temporisée (Timed Rewriting)

A partir des commandes de réécriture temporisée, nous pouvons simuler le comportement de notre système dans un intervalle de temps limité. Le résultat sera l'état final de système correspond à la borne finale de l'intervalle de temps. Par exemple, la commande suivante montre l'état de système associe après le passage d'environ 100 unités de temps et ce à partir l'état initial {WAIT, 0} :

Maude> (trew {WAIT, 0} in time < 100 .)
rewrites: 4134 in 6286706272ms cpu (59ms real) (0 rewrites/second)
Timed rewrite {WAIT,0} in Robot with mode deterministic time increase in time
< 100
Result ClockedSystem :
{TURN@L,1} in time 99

7.3.3.2 Recherche temporisée (Timed Search)

Maintenant, nous montrons par un exemple comment chercher les trois premières occasions où les états avec la valeur de temps 2 peuvent être atteints à partir de l'état {WAIT, 0}, nous avons obtenu trois solutions aux temps 2,12 et 14. Pour chaque temps écoulé, les états du système sont WAIT, PICK et TURN@R respectivement :

Maude> (tsearch [3] { WAIT, 0 } => { S:State, 2} with no time limit .)*
rewrites: 6476 in 6286706272ms cpu (113ms real) (0 rewrites/second)
Timed search [3] in Robot
{WAIT,0} =>* {S:State,2}
with no time limit and with mode deterministic time increase :
Solution 1
S:State --> WAIT ; TIME_ELAPSED:Time --> 2
Solution 2
S:State --> PICK ; TIME_ELAPSED:Time --> 12
Solution 3
S:State --> TURN@R ; TIME_ELAPSED:Time --> 14

7.3.3.3 Recherche non temporisée (Untimed Search)

L'absence de deadlock est une propriété de sûreté particulière qui exprime que le système ne se trouvera jamais dans une situation où il ne peut plus progresser. Pour notre système, la commande suivante permet d'assurer qu'il n'y aura jamais d'échec dans le futur :

Maude> (utsearch {WAIT, 0} =>! GS:GlobalSystem .)
rewrites: 2602 in 6286706272ms cpu (20ms real) (0 rewrites/second)
Untimed search in Robot
{WAIT,0} =>! GS:GlobalSystem
with mode deterministic time increase :
No solution

De plus, nous pouvons garantir que, dans notre système, nous n'aurons jamais le cas où l'état TURN@L est suivi directement (un pas de réécriture) par l'état TURN@R car le robot doit remettre la boite (état PUT) avant de tourner à gauche :

Maude> (utsearch {TURN@L, N:Nat} =>1 {TURN@R, N:Nat} .)
rewrites: 2620 in 6286706272ms cpu (20ms real) (0 rewrites/second)
Untimed search in Robot
{TURN@L,N:Nat} =>1 {TURN@R,N:Nat}
with mode deterministic time increase :
No solution

7.3.3.4 Vérification via le model-checker LTL

Pour vérifier notre système, on peut définir dans un module CHECK-ROBOT la proposition atomique *badState*, qui est destiné à tenir dans les états où le robot ne fonctionne plus en raison de problèmes d'ordonnancement des états, dépassement de temps pour un état donné ou l'arrêt indésirable de robot :

```
(tmod CHECK-ROBOT is
  including TIMED-MODEL-CHECKER .
  protecting Robot .
  op badState : -> Prop .
  eq {S:State, T:Nat} |= badState =
      (T:Nat < 0) or ((S:State == WAIT) and (T:Nat > 10)) or
((S:State == PICK) and (T:Nat > 2)) or ((S:State == TURN@R) and
(T:Nat > 6)) or ((S:State == PUT) and (T:Nat > 2)) or ((S:State
== TURN@L) and (T:Nat > 6)).
endtm)
```

Pour commencer la vérification de modèle du robot, nous vérifions si le système fonctionne toujours correctement, au moins, dans le temps 100 :

Maude> (mc {WAIT, 0} |=t [] ~ badState in time < 100 .)
rewrites: 11614 in 6286706272ms cpu (150ms real) (0 rewrites/second)
Model check{WAIT,0} |=t[]~ badState in CHECK-ROBOT in time < 100 with mode
deterministic time increase
Result Bool :
true

Nous pouvons aussi vérifier le système sans limiter le temps que le comportement correct de système sera toujours maintenu dans le futur :

Maude> (mc {WAIT, 0} |=u [] ~ badState .)
rewrites: 5949 in 6286706272ms cpu (49ms real) (0 rewrites/second)

```
Untimed model check {WAIT,0}  |=u []~ badState in CHECK-ROBOT with mode
   deterministic time increase
Result Bool :
 true
```

La commande suivante vérifie si le robot passe 3 seconde quand se tourne à gauche {TURN@R, 3} chaque fois que le robot démarre de l'état {WAIT, 0}, le résultat sera que la propriété est satisfaite :

```
Maude> (check {WAIT, 0}  |= <> {TURN@R, 3} with no time limit .)
rewrites: 5902 in 6286706272ms cpu (61ms real) (0 rewrites/second)
Check {WAIT,0} |= <> {TURN@R,3} in CHECK-ROBOT with no time limit with mode
   deterministic time increase :
Result: the property holds.
```

On peut savoir la valeur maximale à atteindre dans notre système. La commande suivante vérifie si le temps augmente dans le système à partir de l'état initial {WAIT, 0} jusqu'à ce qu'il atteigne la valeur de temps 12, le résultat est que la propriété n'est pas satisfaite car la valeur maximale de temps à atteindre dans notre système est 10, un contre-exemple soit généré :

```
Maude> (check {WAIT, 0}  |= {S:State, R:Time} until {S:State, 12} in time < 100 .)
rewrites: 14265 in 6286706272ms cpu (195ms real) (0 rewrites/second)
Check {WAIT,0} |= {S:State,R:Time} until {S:State,12} in CHECK-ROBOT in time <
   100 with mode deterministic time increase :
Result: the property does not hold. Counterexample:
     {TURN@L,2} in time 100
```

7.4 Expérimentations et Résultats

Cette section est consacrée pour l'analyse des résultats et des expérimentations réalisées en appliquant notre approche sur les trois études de cas précédemment décrites. Deux types d'évaluation sont effectués, la première diagnostique est liée au processus de transformation modèle à modèle (M2M) et modèle à texte (M2T). Ensuite, une deuxième analyse comparative concerne les commandes de vérification, des spécifications générés des systèmes, sera abordé. Ces analyses font l'objectif d'éprouver que l'approche proposée soit performante et que leur implémentation soit réalisable et efficace et permet de savoir quels sont les facteurs, liés à un système étudié, devant prendre en compte pour mesurer l'influence de ceux-ci sur l'exécution et l'amélioration de notre approche.

Tableau 7.1. Tableau comparatif des résultats concernant le processus de transformation.

	Nombre d'états	**Nombre de transitions**	**Nombre des règles appliquées**			**Temps exécution TGG (ms)**	**Temps génération code (ms)**
			Sc2stm (axiome)	***InitialSt2ModElt***	***Tr2Crule***		
Feu de circulation	3	3	1	1	3	273	861
Thermostat	2	2	1	1	2	192	583
Robot industriel	5	5	1	1	5	465	1446

Le tableau 7.1 présente les résultats numériques que nous avons obtenus lors de l'application de processus de transformation de la phase d'exécution des règles TGG jusqu'à la génération des modules temporisés associés à chaque étude de cas (feu de circulation, thermostat et robot industriel respectivement). Deux considérations sont donc jugées plus satisfaisant Pour analyser les données du tableau 7.1. La première partie est illustrée graphiquement par la figure 7.10, d'où on constate que les deux règles *Sc2stm (axiome)* et *InitialSt2ModElt* sont appliquées un seule fois pour chaque étude de cas. Cela démontre que, pour générer un modèle RT-Maude pour un système étudié, les règles *Sc2stm* et *InitialSt2ModElt* devrons être appliqué et cela indépendamment le degré de complexité de diagramme d'états-transitions associé à ce système (le nombre d'états ou de transitions).

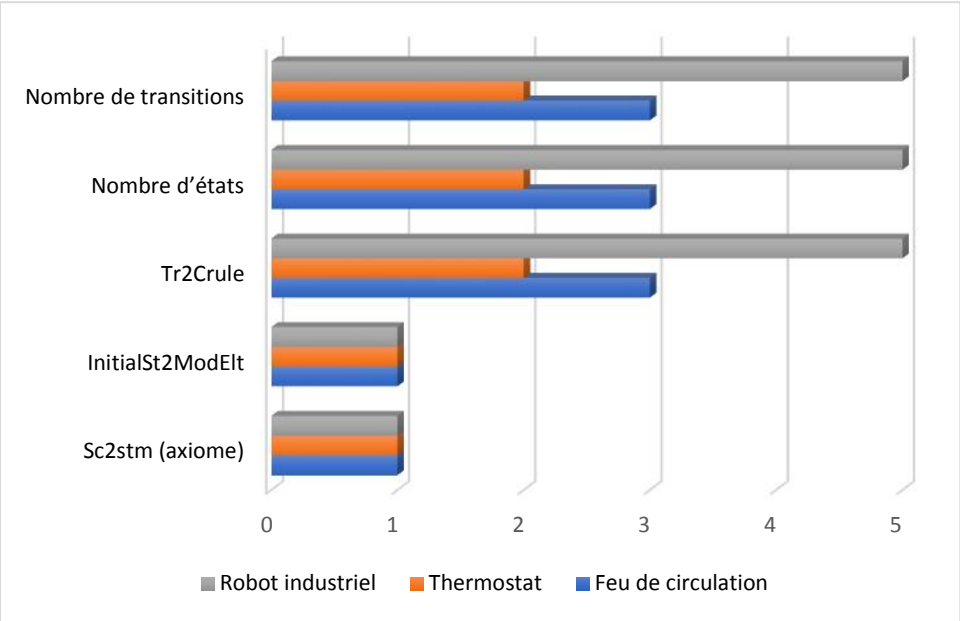


Figure 7.10. Etude comparative des règles à appliquer par rapport les composants des diagrammes d'états-transitions.

La courbe bleue de la figure 7.11 montre le temps d'exécution des règles TGG, calculé en millisecondes, pour chaque étude de cas. D'après cette figure, force est de constater la dépendance entre le temps d'exécution des règles de TGG et le nombre d'états et le nombre de transitions constituant le système étudié.

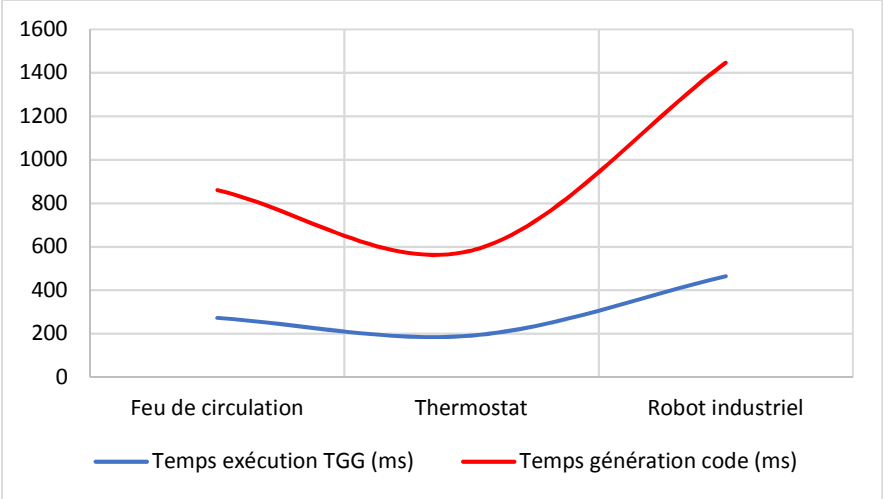


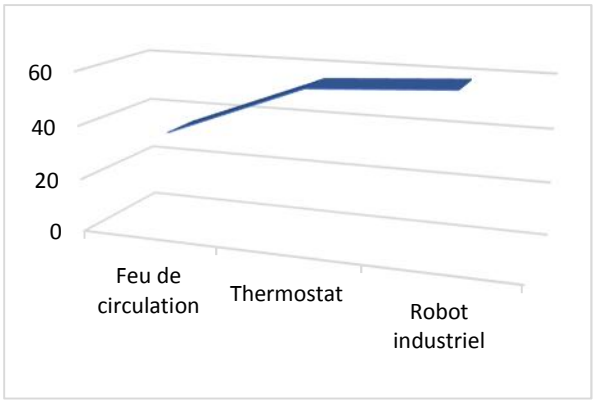
Figure 7.11. Etude comparative de temps d'exécution des règles TGG et de génération de code.

Le même constat peut être fait concernant la courbe rouge d'où il y a une relation directe entre le temps de génération de code et le degré de complexité de système étudié c-à-d plus le système est compliqué, plus nous aurons avoir un temps de génération de code important.

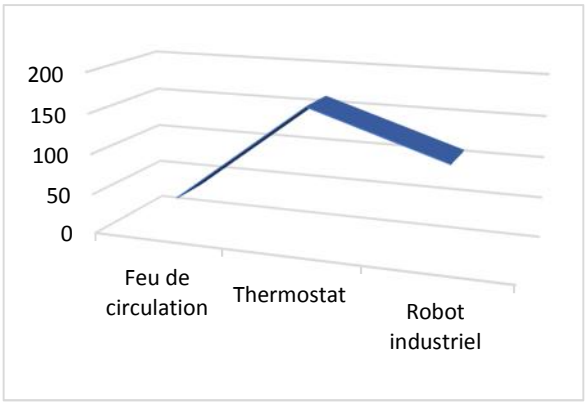
Tableau 7.2. Tableau comparatif de temps d'exécution des commandes de vérification.

	Timed Rewriting (ms)	Timed Search (ms)	Untimed Search (ms)		Model-checking (ms)			
Feu de circulation	36	39	41	25	117	42	21	184
Thermostat	56	166	37	23	108	36	25	202
Robot industriel	59	113	20	20	150	61	49	195

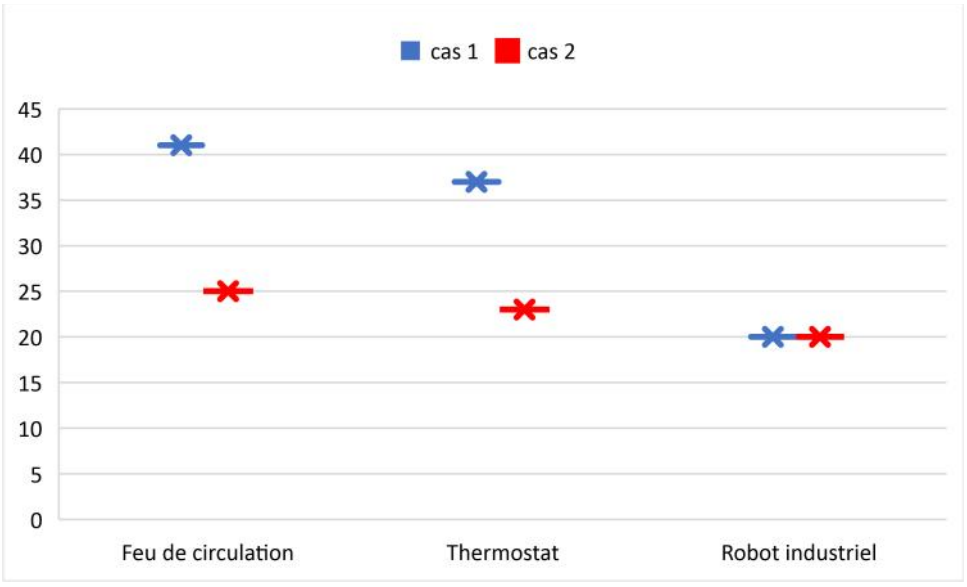
Des mesures à propos le temps d'exécution, des commandes de vérification utilisés par LTL model-checker de RT-Maude pour chaque étude de cas, sont regroupés dans le tableau 7.2. Ces données sont en partie représentées graphiquement sur les figures 7.12 et 7.13 ci-après. Des interprétations des résultats sont mentionnées pour chaque graphique comme suit :



(a) Timed Rewriting (ms)



(b) Timed Search (ms)



(c) Untimed Search (ms)

Figure 7.12. Temps d'exécution des commandes, pour chaque étude de cas, de vérification de LTL Model-checker.

1. Timed Rewriting: le temps d'exécution de la commande lié à la réécriture temporisé montré dans la figure 7.12(a) indique que le système ayant un nombre grand d'états et de transitions prend un temps grand lors d'une opération de réécriture par rapport un système détenant moins de complexité. Cela peut être expliqué par le nombre d'états exploité durant la réécriture.

2. Timed Search: la courbe de la figure 7.12(b) affirme que le thermostat prend un temps plus long lors de la recherche temporisée, par rapport les deux autres systèmes. Malgré que le diagramme d'états-transitions associé au thermostat contient seulement deux états et deux transitions. On peut expliquer cette situation par deux causes, la première et que thermostat possède deux variables de temps. Ainsi que la longueur de l'intervalle de temps consacrée à la recherche temporisé influe considérablement sur le temps d'exécution de la commande.

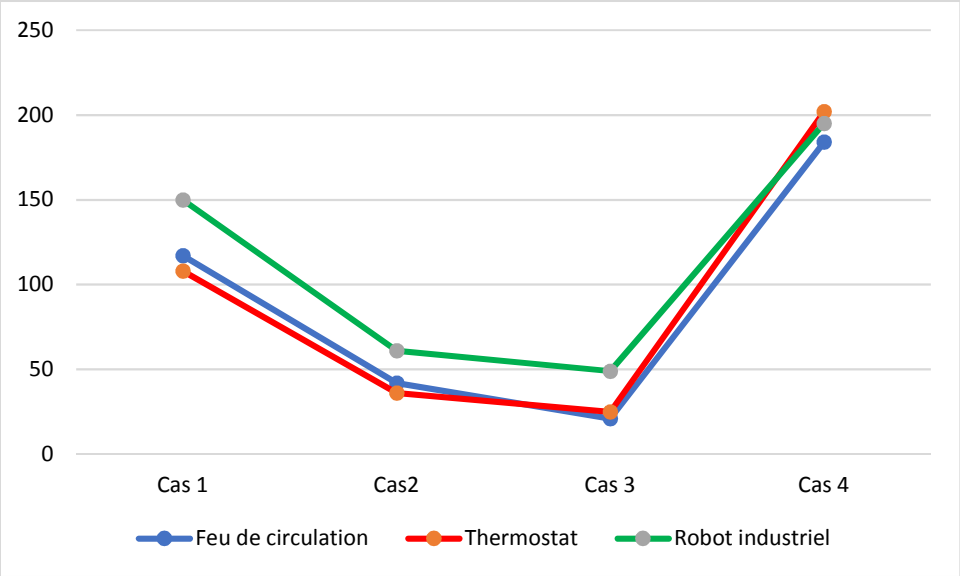


Figure 7.13. Temps d'exécution des commandes de Model-checking (en millisecondes).

3. Untimed Search: d'après la figure 7.12(c), on peut conclure qu'il y a une différence entre le temps d'exécution des commandes de deux cas pour une recherche non temporisée. Cela peut être exprimé par la différence des paramètres de chaque commande. Aussi, on peut remarquer que pour la deuxième commande, le temps d'exécution est convergeant pour les trois systèmes (feu de circulation, thermostat et robot industriel).

4. Model-checking: En observant la figure 7.13, le temps d'exécution pour les quatre commandes pour chaque système peut être visualisé graphiquement par des courbes. En examinant ces résultats, d'une manière comparative, on peut constater que la plupart des commandes de vérification prend un temps plus grand quand on vérifie une propriété de système robot industriel, représenté par une courbe verte, par rapport des autres systèmes (feu de circulation et thermostat). Donc, la complexité de système joue un rôle important dans le temps d'exécution des commandes de vérification (Model-checking). Un système comprend un grand nombre d'états et de transitions peut engendrer des états de recherche et des pas de réécriture important durant la phase de vérification formelle.

Il faut noter que durant les expérimentations effectuées ci-dessus, nous remarquons qu'il est possible que l'exécution multiple d'une même commande puisse mener à un temps d'exécution différent. Ceci due à la configuration de l'ordinateur durant l'exécution des commandes (disponibilité processeur, espace RAM libre). Pour obtenir des résultats exacts ou pour

améliorer les expérimentations, il est fortement conseillé que l'exécution de toutes les commandes seront effectuées dans la même session Maude.

7.5 Conclusion

Dans ce chapitre, nous avons présenté un support outillé de simulation et d'analyse et de vérification formelle des systèmes temps-réel. Ce Framework est basé sur un couplage judicieux entre les systèmes temps réel et le langage RT-Maude permettant de profiter de leurs avantages complémentaires. Nous avons proposé également des études de cas où nous avons utilisé notre outil pour permettre de bien illustrer l'approche de modélisation, de transformation (M2M), de transformation (M2T) et de vérification formelle des systèmes temps-réel présentée dans le chapitre précédent.

Cette approche a été validé par un Model-checker LTL de RT-Maude. Les simulations faites par LTL Model-checker de RT-Maude en testant les propriétés évoquées sur différents modules temporisées ont montré la performance de cette méthode de vérification des modèles. En particulier, la notion du contre-exemple généré par RT-Maude est d'une grande importance vue qu'elle nous aide à trouver rapidement la cause de l'erreur et par la suite la corriger. Finalement, nous avons montré l'efficacité de notre approche à travers une étude comparative des résultats obtenus durant l'application de notre approche sur les études de cas.

Conclusion générale

Les défis que pose le développement des Systèmes Embarqués Temps-Réel (SETR) s'accentuent, du fait qu'ils sont caractérisés par une complexité croissante. Ces systèmes sont en même temps soumis à de plus fortes exigences en matière de leur bon fonctionnement logique ou temporel.
La notion de transformation se base sur un ensemble de techniques, langages et d'outils dont l'objectif final est de faciliter le concept de programmation et de minimiser le coût du développement.
Dans une activité classique d'Ingénierie Dirigée par les Modèles (IDM), les systèmes sont modélisés à l'aide d'une notation semi-formelle et sont par la suite validés puis implantés. L'étape de validation, basée sur ces modèles, est particulièrement cruciale pour les Systèmes Embarqués Temps-Réel (SETR), afin de s'assurer de leur bon fonctionnement.
Cependant, une démarche IDM reste insuffisante dans le sens où elle n'indique pas comment utiliser les modèles pour appliquer l'analyse. Face à cette situation, l'intégration de méthodes formelles dans les cycles de développement de ces systèmes est devenue primordiale. Ces méthodes sont depuis longtemps reconnues afin d'aider au développement de systèmes fiables, en raison de leurs fondements mathématiques, réputés rigoureux sur l'exhaustivité de la vérification formelle qu'ils permettent de l'activer.

Nous avons présenté dans cette thèse une démarche pour l'intégration des méthodes formelles aux démarches de développement suivant les bases de l'ingénierie dirigée par les modèles, pour la spécification et la vérification formelle de SETR. Notre but est de contribuer à une meilleure fiabilité des systèmes. Plus spécifiquement, en renforçant la qualité de leurs spécifications comportementales en termes de contraintes de temps logique. C'est dans ce contexte que se situe cette thèse. Le travail réalisé est divisé en trois parties :

- Dans la première partie, nous avons proposé des méta-modèles pour manipuler les modèles source (Statechart) et les modèles cibles (RT-Maude). Notre outil basé sur la méta-modélisation et une grammaire de graphe assure la transformation automatique des diagrammes d'états-transitions UML vers des modèles contenant des descriptions équivalentes en RT-Maude ;
- Dans la deuxième partie, nous avons exploité le langage de génération de code Xpand pour générer des spécifications formelles de système à partir des modèles RT-Maude ;
- Enfin, dans la dernière partie, nous avons aborder une étape de vérification approfondie des systèmes spécifiés précédemment afin d'analyser et de vérifier les propriétés comportementales des systèmes. L'outil RT-Maude LTL Model-Checker est alors utilisé pour l'analyse des propriétés du système modélisé.

L'approche proposée a été validée à l'aide de trois études de cas différents. Ces trois études de cas étaient, dans l'ordre : un feu de circulation tricolores, un système de thermostat et un robot industriel intégré dans une usine. Une étude comparative, basée sur ces cas, en termes de performance et de complexité a été réalisée.
Nous avons pu à travers ce travail atteindre nos objectifs fixés, cependant un certain nombre d'améliorations peuvent être envisagés. Tout d'abord, il pourrait être envisageable de réaliser une implémentation de notre approche en utilisant d'autres outils de transformation de graphes tels que AGG et VIATRA2 dans un but de comparaison des performances. Aussi, Il serait possible d'étendre cette étude pour prendre en compte la transformation des autres diagrammes UML (diagramme de séquence, diagramme d'activité, etc.). Il serait intéressant de proposer l'utilisation des approches afin d'assurer la qualité des modèles ce qui présenterait un avantage supplémentaire pour le développeur. Une perspective intéressante de ce travail consisterait en

l'obtention d'étendre cette approche de spécification et de vérification sur d'autres types de systèmes tels que les Systèmes Multi-Agents (SMA). De plus, il serait fort intéressant que d'autres formalismes plus éloignés des formalismes basés sur les états-transitions, pourront pris en considération, tels que les automates autorisés et les réseaux de Petri et d'autres outils de vérification pourront être ciblé tel que UPPAAL. Conjointement, Il serait alors possible d'enrichir cette étude par l'exploitation des modules orientés objet intégré dans le langage Maude.

Finalement, la dernière perspective réside au niveau de l'approche graphique ou textuelle choisie pour la transformation de modèles. Pour répondre à cette problématique, il serait supportable d'étudier l'utilisation des autres approches de transformation de graphes présentées dans ce document et autres, pour en faire l'objet d'expérimentation dans le contexte de transformation de modèles.

Bibliographie

[1] A. Evans et T. Clark. Foundations of the Unified Modeling Language. Dans Proc. Of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, 23-24 September 1997, 1997.
[2] Schnoebelen, P., Bérard, B., Bidoit, M., Laroussinie, F., Petit, A. (1999). *Vérification de logiciels : Techniques et outils de model-checking*, Vuibert, Paris.
[3] Kent Stuart : Model driven engineering. Integrated Formal Methods (IFM), 2002 Turku (Finland) Springer, 286-298.
[4] Gérard S, Terrier F, Dubois H, Mraidha C, and Baudry B. L'ingénierie des modéles pour les systémes temps-rèel. As CNRS sur le mda(model driven architecture), pages 76–95, Avril 2003.
[5] P. C. Ölveczky et J. Meseguer : Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation, 20(1-2):161–196, 2007.
[6] A. Berger. *Embedded System Design-An Introduction to Processes, Tools, and Techniques.* CMP Books, ISBN: 1578200733, Lawrence, Kansas, USA 2002.
[7] *Ariane 5 – flight 501 failure*, 1996. Report by the Inquiry Board, chairman : J-L Lions.
[8] Nancy G. Leveson and Clark S. Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7) :18–41, 1993.
[9] Pierre-Alain Reynier. Vérification de systèmes temporisés et distribués : modèles, algorithmes et implémentabilité. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, June 2007.
[10] *The machinery of democracy : Protecting elections in an electronic world, 2006*. Report by Brennan Center Task Force on Voting System Security, Lawrence Norden, Chair.
[11] J. Stankovic and K. Ramamritham. *Tutorial : hard real-time systems*. IEEE Computer Society Press Los Alamitos, CA, USA, 1989.
[12] Herman Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
[13] Pierre Ficheux. *Linux Embarqué*. édition Eyrolles, 2002.
[14] Syntec Informatique, RNT Logiciel. *Livre Blanc des premières Assises Françaises du Logiciel Embarqué*. Number 6. Collection ThémaTIC, Mars 2007.
[15] Yvon Trinquet and Jean-Pierre Elloy. Systèmes d'exploitation temps réel. *Techniques de l'ingénieur, Traité Contrôle et Mesures*, Mars 1999.
[16] Jean-Pierre Elloy. *Le temps réel*. Rapport établi par le Groupe de réflexion du CNRSSPI, TSI 7(5), Hermes, 1988.
[17] Yvon Trinquet. *Encylcopédie de l'informatique et des systèmes d'information*, chapitre Systèmes temps réel - Introduction, pages 719–722. Vuibert, 2006.
[18] Rania Mzid. Rétro-ingénierie des plateformes pour le déploiement des applications temps réel. Système d'exploitation [cs.OS]. Thèse de doctorat, Ecole Nationale des Ingénieurs de Sfax; Université de Bretagne Occidentale, 2014.
[19] Asma Mehiaoui. Techniques d'analyse et d'optimisation pour la synthèse architecturale de systèmes temps réel embarqués distribués : problèmes de placement, de partitionnement et d'ordonnancement. Ingénierie assistée par ordinateur. Thèse de doctorat, Université de Bretagne occidentale Brest, 2014.
[20] J. Xu and D. L. Parnas. *On Satisfying Timing Constraints in Hard-Real-Time Systems*. IEEE Trans. Softw. Eng., 19 :70–84, January 1993.
[21] Pascal Chevochot and Isabelle Puaut. *An Approach for Fault-Tolerance in Hard Real-Time Distributed Systems*. In Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, SRDS '99, pages 292–, Washington, DC, USA, 1999. IEEE Computer Society.

[22] Pascal Chevochot and Isabelle Puaut. *Tolérance aux fautes dans les systèmes répartis temps-réel strict*. Techniques et Sciences Informatiques (TSI), 1999.
[23] Emmanuel Grolleau. “Ordonnancement temps réel hors-ligne optimal à l’aide de réseaux de Petri en environnement monoprocesseur et multiprocesseur”. PhD thesis, LISI-ENSMA, 1999.
[24] Mohamed-Lamine Boukhanoufa. Adaptabilité et reconfiguration des systèmes temps-réel embarqués. Autre [cs.OH]. Thèse de doctorat, Université Paris Sud - Paris XI, 2012.
[25] E.Hull, K. Jackson, J. Dick, “Requirement Engineering – Second Edition,” Springer, ISBN 1-85233-879-2, 2004.
[26] T. Wahl, K. Rotherme, “Representing Time in Multimedia Systems.” International Conference on Multimedia Computing and Systems (ICMCS’94), pp.538-543, Boston, USA, 1994.
[27] T. Vallius, J. Röning. Embedded Object Architecture. In Proceedings of the 8th Euromicro Conference on Digital System Design (DSD'05), IEEE Computer Society 2005.
[28] Edsger W. Dijkstra. Notes on Structured Programming. 1970.
[29] Nouha Abid. Verification of Real Time Properties in Fiacre Language. Thèse de doctorat, LAAS-CNRS, Toulouse, France, Décembre 2012.
[30] Baier C. et Katoen, J. P. (2008). *Principles of Model Checking*. ISBN-10: 0-26202649-X ISBN-13: 978-0-262-02649-9.
[31] P. H. Feiler, D. P. Gluch, et J. J. Hudak. The Architecture Analysis & Design Language (AADL) : An Introduction. Rapport Technique, Software Engineering Institute (SEI), http ://www.sei.cmu.edu/publications/documents/06.reports/06tn011.html, 2006.
[32] Airbus. IRIS, Basic 3D Design User’s Manual, 2000.
[33] DASSAULT Systems. CATIA, Basic 3D Design User’s Manual, 1985.
[34] A. Arnold, G. Point, A. Griffault, et A. Rauzy. The altarica formalism for describing concurrent systems. Fundamenta Informaticae, 40 :109–124, 2000.
[35] P. Pettersson et K. G. Larsen. UPPAAL. Bulletin of the European Association for Theoretical Computer Science, 70 :40–44, 2000. http://www.uppaal.com/ (consulté le 09 Décembre 2017).
[36] J. Lyle Peterson. Petri Net Theory and the Modeling of Systems. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
[37] Laurent Sagaspe. Allocation sûre dans les systèmes aéronautiques : Modélisation, Vérification et Génération. Modélisation et simulation. Thèse de doctorat, Université Sciences et Technologies - Bordeaux I, 2008.
[38] Z. Mammeri. Systèmes temps réel et embarqués : Concepts de base, expression des contraintes temporelles. Cours M2P GLRE Génie Logiciel, logiciels Répartis et Embarqués. Université Paul Sabatier, Toulouse III. Disponible à l'adresse : https://www.irit.fr/~Zoubir.Mammeri/Cours/M2PConcSTR/Chap2STRbases.pdf (accédé le 09 septembre 2017)
[39] Cook, S. (1971). *The complexity of theorem-proving procedures*. In 3rd Annual ACM. Symposium on Theory of Computing, pages 151–158. ACM Press.
[40] Dingel, J. et Filkorn, T. (1995). *Model Checking for infinite state systems using data abstraction, assumption commitment style reasoning and theorem proving.* In 7th International Conference on Computer Aided Verification (CAV), volume 939 of Lecture Notes in Computer Science, pages 54–69. Springer-Verlag.
[41] Raida ElMansouri. Modélisation et Vérification des processus métiers dans les entreprises virtuelles : Une approche basée sur la transformation de graphes. Thèse de doctorat, Université Mentouri Constantine – Algérie, 2009.
[42] Clarke, E. M., Grumberg, O. et Peled, D. A. (1999). *Model checking*. MIT Press.

[43] Bardin, S. (2008). *Introduction au Model Checking*. École Nationale Supérieure de Techniques Avancées.
[44] Amira Methni. Méthode de conception de logiciel système critique couplée à une démarche de vérification formelle. Logique en informatique [cs.LO]. Thèse de doctorat, Conservatoire national des arts et métiers - CNAM, 2016.
[45] Ahmed Mekki. Contribution à la Spécification et à la Vérification des Exigences Temporelles : Proposition d'une extension des SRS d'ERTMS niveau 2. Automatique, Génie informatique, Traitement du Signal et Images. Thèse de doctorat, école centrale de lille, 2012.
[46] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. Computer, vol. 23(9):8-23, 1990.
[47] Dehimi Nardjess Tissilia. Un Cadre Formel pour La Modélisation et L'analyse Des Agents Mobiles. Thèse de doctorat, Université Constantine2, Constantine – Algérie, 2013.
[48] Meunier F.: Modélisation des ressources linguistiques d'une application industrielle. Conférence TALN 1999, Cargèse, 12-17 juillet 1999.
[49] Saksena M., Karvelas P.: Designing for Schedulability : Integrating Schedulability Analysis with Object-Oriented Design, In Proc., Euromicro Conference on Real-Time Systems, June 2000, pp 101 – 108.
[50] Sommerville I. Le génie Logiciel et ses applications. Paris: InterEditions, Dunod, 1988, 330 p. ISBN 2729601805.
[51] Royce W. W.: Managing the Development of Large Software Systems: Concepts and Techniques. In Proc. of the 9th international conference on Software Engineering, 1987, pp. 328 – 338. ISBN 0-89791-216-0
[52] Goumaa H., Scott D.: Prototyping as a tool in the Specification of User Requirements. Proc. 5th IEEE Int'l Conf. Software Eng, 1981, pp. 333-342.
[53] Giddings R. V.: Accommodating Uncertainty in Software Design. Comm.ACM, 1984, vol. 27, n°. 5, pp. 428-434, 1984.
[54] Hirsh E.: Evolutionary Acquisition of Command and Control Systems. Program Manager, 1985, pp. 18-22.
[55] Karen Godary. Validation temporelle de réseaux embarqués critiques et fiables pour l'automobile. Thèse de Doctorat de l'Institut National des Sciences Appliquées de Lyon, 2004. (Cité en pages 14, 120, 121 et 129.)
[56] Cédrick Lelionnais. Contribution à la considération du comportement des plates-formes d'exécution logicielles temps réel. Génie logiciel [cs.SE]. Thèse de doctorat, Ecole Centrale de Nantes, 2014.
[57] Joost-Pieter Katoen. Principles of model checking. MIT Press, 2008. (Cité en pages 14, 121 et 122.)
[58] Olfa Mosbahi. Développement formel des systèmes automatisés. Thèse de Doctorat de l'Institut National Polytechnique de Lorraine ; l'Université Tunis-El Manar, 2008.
[59] P. Behm, P. Benoit, and J. M. Meynadier. Meteor : A Successful Application of B in a Large Project. In FM 99 — World Conference on Formal Methods in the Development of Computing Systems, number 1708 in LNCS, pages 369–387. Springer Verlag, 1999.
[60] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, April 1986.
[61] Wolfgang Reisig. *Petri Nets : An Introduction*, volume 4 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1985.
[62] C. A. Richard Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
[63] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[64] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, April 1994.
[65] Cyril Dumont. Système d'agents mobiles pour les architectures de calculs auto-adaptatifs. Informatique [cs]. Thèse de doctorat, Université Paris-Est, 2014.
[66] R. Alur and D. Dill. Automata for modeling real-time systems. In *Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.
[67] R. Alur and D. Dill. A theory of timed automata. *TCS*, 126 :183–235, 1994.
[68] Houda Bel Mokadem. Vérication des propriétés temporisées des automates programmables industriels. Software Engineering. École normale supérieure de Cachan - ENS Cachan, 2006.
[69] G. Behrmann, A. David, and K. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 33–35. Springer Berlin / Heidelberg, 2004.
[70] T. Murata, "Petri nets : Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, April 1989.
[71] R. David and H. Alla, *Discrete, Continuous, and Hybrid Petri Nets*. Springer, 1ed., 23 November 2004.
[72] C. Girault et R. Valk, editors. Petri Nets for System Engineering. Springer, 2003.
[73] Pradeepa Thomas Benjamin. Le suivi de l'apprenant dans le cadre du serious gaming. Informatique et théorie des jeux [cs.GT]. Thèse de doctorat, Université Pierre et Marie Curie - Paris VI, 2015.
[74] Ladet, P. (1989). Réseaux de petri. Techniques de l'Ingénieur, R 7252 :1–17.
[75] Kurt JENSEN : Coloured Petri nets : A high level language for system design and analysis. Springer, 1991.
[76] J. Lamp, "Using petri nets to model weltanschauung alternatives in soft systems methodology," Oct. 27 1998.
[77] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1974. Project MAC Report MAC-TR-120.
[78] P.M. Merlin. A study of the recoverability of computing systems. PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA, 1974.
[79] Parosh Aziz Abdulla and Aletta Nylen. Timed petri nets and bqos. In 22nd International Conference on Application and Theory of Petri Nets (ICATPN'01), volume 2075 of Lecture Notes in Computer Science, pages 53–70, Newcastle upon Tyne, United Kingdom, june 2001. Springer-Verlag.
[80] David de Frutos Escrig, Valentín Valero Ruiz, and Olga Marroquín Alonso. Decidability of properties of timed-arc petri nets. In 21st International Conference on Application and Theory of Petri Nets (ICATPN'00), volume 1825 of Lecture Notes in Computer Science, pages 187–206, Aarhus, Denmark, june 2000. Springer-Verlag.
[81] Wael Khanza. Réseau de Petri P-Temporels. Contribution à l'étude des systèmes à événements discrets. PhD thesis, Université de Savoie, 1992.
[82] Marc Boyer and Olivier (H.) Roux. Comparison of the expressiveness of arc, place and transition time Petri nets. In 28th International Conference on Application and Theory of Petri Nets and other models of concurrency (ICATPN'07), volume 4546 of Lecture Notes in Computer Science, pages 63–82, Siedlce, Poland, jun 2007. Springer-Verlag.
[83] Morgan MAGNIN. Réseaux de Petri à chronomètres : Temps dense et temps discret. Thèse de doctorat, Université de Nantes, 2007.
[84] Sifakis J., Le controle des systèmes asynchrones : concepts, propriétés, analyse statique. Thèse de Docteur en Sciences Mathématiques, Institut Polytechnique de Grenoble, 1977.

[85] David R., Alla H., Du grafcet aux réseaux de Petri, Paris, Edition Hermés, 1992.
[86] Bassem Sammoud. Contribution à la modélisation et à la commande des feux de signalisation par réseaux de Petri hybrides. Autre. Thèse de doctorat, Université de Technologie de Belfort-Montbeliard, 2015.
[87] Rakkay, H. (2009). *Approches formelles pour la modélisation et la vérification du contrôle d'accès et des contraintes temporelles dans les systèmes d'information* (Thèse de doctorat, École Polytechnique de Montréal). Tiré de http://publications.polymtl.ca/123/ (consulté le 05 Décembre 2017).
[88] Bérard B., Bidoit M., Finkel A., Laroussinie F., Petit A., Petrucci L. et Schnoebelen P., *Systems and software verification - model-checking techniques and tools*, Springer-Verlag, Berlin (Allemagne) : 190 p., 2001.
[89] Amir Pnueli. The temporal logic of programs. In Foundations of Computer Science, 1977., 18th Annual Symposium on, pages 46-57, 1977.
[90] A. Pnueli "*The Temporal Semantics of Concurrent Programs*" Theoretical Computer Science, 13, 45–60, Elsevier Science, 1981.
[91] M. Clarke, E. A. Emerson "*Design and synthesis of synchronous skeletons using branching-time temporal logic*" in: D. Kozen (Ed.), Proceedings of the 3rd Workshop on Logics of Programs (LOP'81), Vol. 131 of LNCS, Springer-Verlag, 52-71, 1981.
[92] E Allen Emerson and Jai Srinivasan. Branching time temporal logic. In Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, pages 123,172. 1989.
[93] R. Koymans "*Specifying real-time properties with metric temporal logic*" Real-Time Systems, 2 (4), 255–299, 1990.
[94] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *This paper appears in : Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425, Philadelphia, PA, USA, 1990. IEEE Computer Society. 26, 28, 29, 36.
[95] Rajeev Alur and Thomas Henzinger. Back to the Future: Towards a Theory of Timed Regular Languages. In Proc. 33rd IEEE Symposium on Foundations of Computer Science (FOCS'92), pages 177–186. IEEE Computer Society Press, 1992.
[96] Rajeev Alur and David Dill. A Theory of Timed Automata. Theoretical Computer Science (TCS), 126(2) :183–235, 1994.
[97] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In J.W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, Real-Time: Theory in Practice. REX Workshop Proceedings, volume 600 of Lecture Notes in Computer Science, pages 74–106, Mook, Netherlands, June 1991. SpringerVerlag.
[98] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-Checking in Dense Real-Time. Information and Computation, 104(1) :2–34, 1993.
[99] *Thomas. A. Henzinger*, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193-244,1994.
[100] Robert Abo. Approches formelles pour l'analyse de la performabilité des systèmes communicants mobiles : Applications aux réseaux de capteurs sans fil. Autre [cs.OH]. Conservatoire national des arts et metiers - CNAM, 2011.
[101] Mahdi Gueffaz. ScaleSem : model checking et web sémantique. Calcul formel [cs.SC]. Thèse de doctorat, Université de Bourgogne, 2012.
[102] J. Meseguer, "Conditional Rewriting Logic as an Unified Model of concurrency". Theoretical Computer Science, 96(1) :73–155, 1992.
[103] J. Meseguer. "Rewriting Logic as a Semantic Framework of Concurrency: a Progress Report". Seventh International Conference on Concurrency Theory (CONCUR'96), Volume 1119 of LNCS, Springer Verlag, p. 331-372, 1996.

[104] J. Meseguer. “Rewriting logic and Maude: a Wide-Spectrum Semantic Framework for Object-based Distributed Systems”. In S. Smith and C.L. Talcott, editors, Formal Methods for Open Object-based Distributed Systems, (FMOODS’2000), p. 89-117. Kluwer, 2000.
[105] M. Clavel, F. Duran, S. Ecker, P. Lincoln, N. Marti-Oliet, J. Meseguer, C. Talcott. “The Maude 2.0 System”. In Proc. Rewriting Techniques and Applications (RTA), Volume 2706 of LNCS, Spring-Verlag , p. 76-87., 2003.
[106] Meseguer J., “Membership algebra as a logical framework for equational specification”, in F. Parisi-Presicce, (ed), Proc. WADT'97, LNCS, pp. 18-61, 1998.
[107] 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.
[108] Noura Boudiaf, "Développement des Outils Basés Maude pour les ECATNets. Domaine d’Application : Analyse des Programmes Ada". Thèse de doctorat, université de Constantine, 2007.
[109] Malika Benammar, Une Approche Basée Architecture pour la Spécification Formelle des Systèmes Embarqués, Thèse de doctorat, université de Constantine, 2011.
[110] N. Marti-Oliet, J. Meseguer. Rewriting Logic: Roamap and Bibliography, In Theoretical Computer Science, June 2001.
[111] J. Meseguer. Rewriting logic as a unified model of concurrency. In Lecture Notes in Computer Science, Volume 458, pages 384-400. August 1990.
[112] P. C. Ölveczky, J. Meseguer. Specifying Real-Time Systems in Rewriting Logic. 1st International Workshop on Rewriting Logic and its Applications (WRLA’96), In Lecture Notes in Computer Science, editor, Vol. 4. Springer Verlag 1996.
[113] M. Clavel, J. Meseguer. Reflection and Strategies in Rewriting Logic, Electronic Notes in Theoretical Computer Science 4, Elsevier Science B. V. 1997.
[114] 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. (Consulté le 09 Décembre 2017).
[115] 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. (Consulté le 09 Décembre 2017).
[116] 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.
[117] Futatsugi K. et Diaconescu R.., “CafeOBJ Report”, World Scientific, AMAST Series, 1998.
[118] G. J. Holzmann. The Spin Model Checker – Primer and Reference Manual. Addison-Wesley, Boston, Massachusetts, USA, 2004.
[119] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of Computer Aided Verification (CAV 02)*, 2002.
[120] Nadia Menad, Modélisation des Systèmes embarqués Temps-Réel : Vers une ingénierie dirigée par les méthodes formelles. Université Mohamed Boudiaf des sciences et de la technologie d'Oran, 2015.
[121] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer et J. F. Quesada : Maude : specification and programming in rewriting logic. Theor. Comput. Sci., 285(2):187–243, 2002. (Cited on pages 97 et 124.)
[122] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Marti-Oliet, José Meseguer et Carolyn Talcott : Maude manual (version 2.6). http://maude.cs.uiuc.edu/maude2manual/maude-manual.pdf, 2011. Accédé : Décembre 2013. (Cited on pages 124 et 131.)

[123] José Meseguer : A logical theory of concurrent objects. OOPSLA/ECOOP'*90* Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications, 25(10):101–115, 1990. (Cited on page 124.)
[124] Manuel Clavel, Francisco Durán, Joe Hendrix, Salvador Lucas, José Meseguer et Peter Ölveczky : The maude formal tool environment. In Till Mossakowski, Ugo Montanari et Magne Haveraaen, éditeurs : Algebra and Coalgebra in Computer Science, volume 4624, pages 173–178. 2007. (Cited on page 125.)
[125] Steven Eker, José Meseguer et Ambarish Sridharanarayanan : The maude {LTL} model checker. Electronic Notes in Theoretical Computer Science, 71:162 – 187, 2004. (Cited on page 125.)
[126] Narciso Martí-Oliet et José Meseguer : Rewriting logic as a logical and semantic framework. In J. Meseguer, éditeur : Handbook of Philosophical Logic, volume 9 de Handbook of Philosophical Logic, pages 1–87. Elsevier Science Publishers, 2002. (Cited on page 125.)
[127] José Raúl Romero, José Eduardo Rivera, Francisco Durán et Antonio Vallecillo : Formal and tool support for model driven engineering with maude. Journal of Object Technology, 6(9):187–207, 2007. (Cited on pages 125 et 127.)
[128] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, CONCUR'96, volume 165 of F, pages 347—398. Springer, 1997.
[129] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, C. Talcott. "Maude Manual (Version 2.2) ". December 2005.
[130] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer et Carolyn Talcott. All about maude – a high-performance logical framework : How to specify, program and verify systems in rewriting logic. Springer-Verlag, Berlin, Heidelberg, 2007. (Cité en pages 47,60,66et98.)
[131] M.Clavel F. Duran, S. Eker, P. Lincoln, N. Marti'-Oliet, J. Meseguer, C.Talcott. MAUDE MANUAL (Version 2.3), SRI Inter, 2007, web http://maude.cs.uiuc.edu/maude1/manual/. (Consulté le 02 Décembre 2017).
[132] F. Duran, J. Meseguer. The Maude specification of Full-Maude. Technical report, SRI International, Computer Science Laboratory, February 1999.
[133] José Meseguer, Carolyn Talcott. Maude manual (Version 2.5), June 2010. Disponible sur: http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf. (Accédé le 05 Octobre 2017).
[134] Michael HUTH et Mark RYAN : Logic in Computer Science : Modelling and reasoning about systems. Cambridge University Press, 2004.
[135] Olveczky, P. C., «Real-Time Maude 2.2 Manual». Department of Informatics. University of Oslo, 2006.
[136] Peter Csaba Ölveczky, « Real-Time Maude 2.3 Manual », Department of Informatics, University of Oslo, August 8, 2007.
[137] P. C. Ölveczky, J. Meseguer. Specification and Analysis of Real-Time Systems using Real-Time Maude. Fundamental Aspects of Software Engineering (FASE'2004), In Springer, editor, Vol. 2984 of "Lecture Notes in Computer Science", 2004.
[138] Manuel Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Marti-Oliet, José Meseguer et Carolyn Talcott. Maude Manual (Version 2.6)(January 2011), 2011. (Cité en pages 47,60,66,112et116.)
[139] Maria, A. Introduction to modeling and simulation. Pages 7-13, 1997.
[140] H. Kadima. MDA, une conception orientée objet guidée par les modèles. DUNOD 2005.
[141] J. Celso, J. Freire, J. Giraudin, *Agnès Front Atelier MODSI: Un Outil de Méta-Modélisation et de Multi-Modélisation*. Laboratoire de Logiciels et Systèmes Réseaux, IMAG B.P. 72 - 38402 - Saint Martin d'Hères Cedex – France.

[142] Bahri, M. R. *Une approche intégrée Mobile-UML/Réseaux de Petri pour l'Analyse des systèmes distribués à base d'agents mobiles*. Thèse de doctorat, Université de Mentouri, Constantine, 2011.
[143] S. Cook, J. Daniels, Designing Object Systems - Object-Oriented Modelling with Syntropy. Prentice-Hall, 1994.
[144] J. P. Bowen, M. G. Hinchey, *Seven more myths of formal methods*. IEEE Software, pp. 34–41, july 1995.
[145] Audibert, L. (november 2007-2008). Uml 2.0, institut universitaire de technologie de villetaneuse, département informatique, adresse du document : http://laurentaudibert.developpez.com/cours-uml/html/cours-uml.html. (Consulté le 10 Septembre 2017).
[146] Booch G. Object-Oriented Analysis and Design with Applications. The Benjamin/Cummings Publishing Company Inc., Redwood City, California, 1994.
[147] Jacobson I., Chirterson M., Jonsson P., et Overgaard G. Object-Oriented Software Engineering, Addison-Wesly Publishing, Reading, Massachusetts, 1992.
[148] Rumbaugh J., Blaha M., Permerlani W., Eddy F. et Lorensen W., Object-Oriented Modeling and Design, Practice-Hall, Englewood Cliffs, New Jersey, ISBN: 0136298419, 1991.
[149][OMG UML] Object Management Group, *Unified Modeling Language™, UML® Resource Page,* http://www.uml.org/, 2017. (Consulté le 05 Octobre 2017).
[150] Warmer J., Kleppe A., The Object Constraint Language: Precise Modeling with UML, Addison-Wesly Publishing, Reading, Massachusetts, 1998.
[151] OMG. *OMG Unified Modeling Language™ (OMG UML), Infrastructure* Version 2.3, http ://www.omg.org/spec/UML/2.3/, Mai 2010. (Accédé le 04 Octobre 2017).
[152] OMG. *OMG Unified Modeling Language™ (OMG UML), Superstructure* Version 2.3, http ://www.omg.org/spec/UML/2.3/, Mai 2010. (Accédé le 04 Octobre 2017).
[153] Crampes J.B., *Génie logiciel - Méthode orientée-objet intégrale MACAO, Démarche participative pour l'analyse, la conception et la réalisation de logiciels*, Ellipses (Eds.), 314 pages, 2003.
[154] Thomas Collonvillé. Elaboration de processus de développements logiciels spécifiques et orientés modèles : application aux systèmes à évenements discrets. Informatique [cs]. Université de Haute Alsace - Mulhouse, 2010.
[155] Sekou Kangoye. Elaboration d'une approche de vérification et de validation de logiciel embarqué automobile, basée sur la génération automatique de cas de test. Systèmes embarqués. Université d'Angers, 2016.
[156] Hutchinson, J., Whittle, J. & Rouncefield, M. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming,* 89**,** 144-161, 2014.
[157] H.-E. Eriksson, M. Penker, B. Lyons, and D. Fado, *UML2 Toolkit*. John Wiley & Sons, 2004.
[158] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Inc, 1998.
[159] P.A Muller, N. Geartner. Modélisation objet avec UML, Eyrolles, 2éme édition 2000, Deuxième tirage 2001.
[160] Fabien Chiron. Contribution à la flexibilité et à la rapidité de conception des systèmes automatisés avec l'utilisation d'UML. Informatique mobile. Université Blaise Pascal- Clermont-Ferrand II, 2008.
[161] Kruchten, P. The 4+1 view model of architecture. IEEE Software, 12(6):42-50, 1995. (Cité dans la page 62.)
[162] Muchandi, V. Applying 4+1 view architecture with UML2. White paper. Sparx Systems, 2007. (Cité dans les pages vii and 63.)

[163] Fowler M., *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition*, Addison Wesley Professional (Eds.), Addison-Wesley Object (Series), 208 pages, 2003.
[164] Laurent Debrauwer et Fien Van Der Heyde. UML 2 : initiation, exemples et exercices corrigés. Éditions ENI, Nantes, France, 2008. (Cité en pages 145,160,163 et 167.)
[165] David Harel. *Statecharts : A visual formalism for complex systems*. Sci. Comput. Program., vol. 8, no. 3, pages 231–274, Juin 1987. (Cité en page 160.)
[166] Rumbaugh, I. Jacobson, G. Booch, UML 2.0 GUIDE DE REFERENCE, Campus Press, 2005.
[167] D. Pilone, N. Pitman, UML 2.0 in a Nutshell, O'Reilly, 2005.
[168] REBAIAIA Mohamed-Larbi. Spécification et Vérification des Systèmes Critiques : Extension de l'Environnement VALID pour la Prise en Charge du Temps Réel. Thèse de doctorat, université de Batna, 2011.
[169] OMG. *UML 2.1.2 Superstructure*. pages 1–738, 2007. (Cité en pages 20, 30, 67, 74, 76, 77, 99, 117)
[170] G Booch, J Rumbaugh et I Jacobson. The Unified Modeling Language User Guide. Addison-Wesley Longman Inc, 1999. 19, 29, 30, 70, 115
[171] Booch, G., Rumbaugh, J. & Jacobson. Guide de l'utilisateur UML. 2éme édition. Eyrolles .2000.
[172] Jacobson. I, Booch. G., Rumbaugh, J. The Unified Software Development Process. Addison Wesley, 1999.
[173] Philippe Kruchten. The Rational Unified Process: An introduction. 3éme edition. Addison-Wesley Professional.1999.
[174] Pascal Roques and Franck Vallée. *UML 2 en action : De l'analyse des besoins à la conception J2EE*. Eyrolles, Juin 2004.
[175] Frédérick Seyler. *UGATZE : Méta-modélisation pour la réutilisation de composants hétérogènes distribues*. Phd thesis, L'Université de Pau et des pays de l'Adour, Décembre 2004.
[176] J. Bézivin. "*In Search of a Basic Principle for Model Driven Engineering*", The European Journal for the Informatics Professional, vol. 2, pp. 21-24, 2004.
[177] Jean-Marc Jezequel, Benoit Combemale et Didier Vojtisek. Ingénierie dirigée par les modèles : des concepts à la pratique. Editions ellipses, Paris, 2012. (Cité en pages 17,19,24,26,28,35,44,63,73,105 et 137.)
[178] J. Bézivin. On the unification power of models. Software & Systems Modeling, 4(2) :171–188, 2005.
[179] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. Software, IEEE, 20(5) :36–41, 2003.
[180] E. Seidewitz, *What Models Mean*, IEEE Software, Vol. 20, Num. 5, Sept. 2003, Page(s): 26 - 32.
[181] J.-M. Favre. Towards a basic theory to model model driven engineering. In 3rd Workshop in Software Model Engineering, WiSME. Citeseer, 2004.
[182] Jean Bézivin : Sur les principes de base de l'ingénierie des modèles. RSTI-L'Objet, 10(4) :145-157, 2004.
[183] J. Bézivin, M. Blay, M. Bouzhegoub, J. Estublier, J.M. Favre, S. Gérard, and J.M. Jezequel. *Rapport de Synthèse de l'AS CNRS sur le MDA (Model Driven Architecture)*. Rapport CNRS, janvier 2005.
[184] A. D. Ionita, J. Estublier, and G. Vega. *Domaines réutilisables dirigés par les modèles*. In Proc. Of Ingénierie dirigée par les modèles, IDM05, Paris, 2005.
[185] OMG. Meta Object Facility (MOF) Core Specification version 2.0. OMG Document, January 2006.

[186] OMG. Uml 2 metamodel. ptc/04-10-05 (UML 2.0 Superstructure FTF Rose model containing the UML 2 metamodel), 2004.
[187] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick et T.J. Grose, *Eclipse Modeling Framework: A Developer's Guide*. Addison-Wesley Pub Co, 1st edition, August 2003.
[188] T.-L.-A. Dinh, O. Gerbé, and H. Sahraoui. *Un méta-méta-modèle pour la gestion de modèles*. In IDM 06 Actes des 2èmes Journées sur l'Ingénierie Dirigée par les Modèles, Lille, France, 2006.
[189] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, and J. Sprinkle. *Composing domain-specific design environments*. Computer Networks and ISDN Systems, pages 44–51, 2001.
[190] Marvin L. Minsky. *Semantic Information Processing*. The MIT Press, 1969.
[191] V. Chapulrat, B.Kamsu-Foguen,F. Pruner, "*Aformal verification framework and associated tools for enterprise modelling: Application to UEML*", computers in industry, Novembre 2005.
[192] Bran Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5) :19–25, 2003.
[193] J.Montmain, J-M. Penalva, V.Chapurlat, A-L.Courbis, B.Vayssade, M.Vinches, "*Le management du risque*", Rapport interne, Audit Risque du 21 Juin 2006.
[194] OMG. MDA Guide Version 1.0.1. OMG omg/2003-06-01, June 2003.
[195] Rania Mzid. Rétro-ingénierie des plateformes pour le déploiement des applications temps réel. Système d'exploitation [cs.OS]. Ecole Nationale des Ingénieurs de Sfax; Université de Bretagne Occidentale, 2014.
[196] M. Mohamed, M. Romdhani, and K. Ghedira. Classification des approches de refactorisation des modèles. IDM 2008, page 169, 2008.
[197] OMG. MOF QVT final adopted specification. Novembre 2005.
[198] F. Jouault. Contribution à l'étude des langages de transformation de modèles. Thèse de doctorat, Université de Nantes, France, 2006.
[199] K. Czarnecki and S.Simon Helsen, "Classification of Model Transformation Approaches", OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.
[200] Jonathan Musset, Etienne Juliot et Stéphane Lacrampe : Acceleo™ 2.2 : Guide utilisateur. Obeo, http://www.eclipse.org/acceleo/, Avril 2008. (Cité pages 17 et 129.) (Consulté le 05 Octobre 2017).
[201] Markus Völter et Thomas Stahl : Model-Driven Software Development : Technology, Engineering, Management. Numéro 978-0-470-025703. John Wiley & Sons, Juin 2006. (Cité page 17.)
[202] L. Bondé. Transformations de Modèles et Interopérabilité dans la Conception de Systèmes Hétérogènes sur Puce à Base d'IP. Thèse de doctorat, Université des Sciences et Technologies de Lille, 2006.
[203] **"***Object Management Group (OMG), Model Driven Architecture (MDA)"*, site Internet, http://www.omg.org/mda, (Consulté : Novembre 2017).
[204] Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0, avril 2008. Online : http://www.omg.org/spec/QVT/1.0/, (Consulté : Novembre 2017).
[205] « OptimalJ 3.1, Using OptimalJ : Tutorials ». Téléchargeable online sur : www.uio.no/studier/emner/matnat/ifi/INF5120/v04/verktoy/OptimalJTutorials.pdf, (accédé : Novembre 2017).
[206] D. Balasubramanian, A. Narayanan, C. vanBuskirk and G. Karsai, "*The Graph Rewriting and Transformation Language: GReAT*", In Proceedings of the 3rd International Workshop on Graph Based Tools, Brazil, 2006.

[207] "*AGG*", site web : http://tfs.cs.tu-berlin.de/agg/, (Consulté : Novembre 2017).
[208] J. De Lara and H. Vangheluwe, "*AToM ^{3} : A Tool for Multi-Formalism Modelling and Meta-Modelling*", LNCS, Springer, vol. 2306, pp. 174-188, 2002.
[209] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev and P. Valduriez, "*ATL: a QVT Like Transformation Language*", In Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, USA, pp. 719– 720, 2006.
[210] P.A. Muller, F. Fleurey and J.M. Jezequel, "*Weaving Executability into Object-Oriented Meta-Languages*", LNCS, Springer, vol. 3713, pp. 264–278, 2005.
[211] C. Dumoulin, "*ModTransf : A Model to Model Transformation Engine*", 2004. Disponible online sur : www.lifl.fr/~dumoulin/mdaTransf/doc/modTransf.pdf, (Accédé : Novembre 2017).
[212] Czarnecki K. and Helsen S. 2003. Classification of Model Transformation Approaches. *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*.
[213] Hamidou Togo. Parallélisation de simulateur DEVS par métamodélisation et transformation de modèle. Autre. Université Blaise Pascal - Clermont-Ferrand II, 2015.
[214] Chris Raistrick, Paul Francis, John Wright, Colin Carter et Ian Wilkie. 2004. *Model Driven Architecture with Executable UML*. Cambridge.
[215] THE MIDDLEWARE COMPANY. 2003. *Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach: Productivity Analysis*. Object Management Group.
[216] Object Management Group. 2010. « Success Stories ». Online : http://www.omg.org/mda/products_success.htm, (accédé : Novembre 2017).
[217] Interactive Objects Software GmbH. « ABB Uses ArcStyler to Web-Enable High-End Resources for Cost-Effective Access via the Intranet ». Online : http://www.omg.org/mda/mda_files/SuccessStory_ABB.pdf, (Accédé : Novembre 2017).
[218] Mohagheghi, Parastoo, Vegard Dehlen et Tor Neple. 2009. « Definitions and approaches to model quality in model-based software development - A review of literature ». *Information and Software Technology*, vol. 51, nº 12, p. 1646-1669.
[219] Parastoo Mohagheghi, et Vegard Dehlen. 2008. « Where is the Proof? - A Review of Experiences from Applying MDE in Industry ». In *4th European Conference on Model Driven Architecture Foundations and Applications (ECMDA'08)*, Vol. 5095, p. 432-443.
[220] Mellor Stephen J, Scott Kendall, Uhl Axel et Weise Dirk. 2004. *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional.
[221] Joaquin Miller and Jishnu Mukerji. Model driven architecture (MDA). Draft ormsc/200107-01, Architecture Board ORMSC, July 2001.
[222] A. Clave. D'uml à mda en passant par les méta modèles. Technical report, La Lettre d'ADELI n 56, pp. 120, 2004.
[223] P. Parrend. Introduction à MDA : Principe. http://pparrend.developpez.com/tutoriel/mda-intro/, pp. 120-124, 2006, (accédé : Novembre 2017).
[224] P.A. Caron, F. Hoogstoel, X. Le Pallec, and B. Warin. Construire des dispositifs sur la plateforme moodle - application de l'ingénierie bricoles. In MoodleMoot-2007, Castres, France, pp. 109-121, 14 - 15 Juin 2007.
[225] Jean Bézivin, Xavier Blanc. MDA : Vers un important changement de paradigme en génie logiciel, Juillet 2002. Téléchargeable online sur : http://mfworld42.free.fr/cnam/nfe115-informatique.../MDA.Partie1.JBXB.Last.prn.pdf, (accédé : Novembre 2017).
[226] Combemale, Benoit. *Approche de métamodélisation pour la simulation et la vérification de modèle. Application à l'ingénierie des procédés.* PhD, Institut National Polytechnique de Toulouse, 2008.

[227] Chemma Sofiane. Une approche de composition de services Web à l'aide des Réseaux de Petri orientés objet. Thèse de doctorat, Université de Mentouri, Constantine, 2014.
[228] Object Management Group. MOF 2.5.1 Specification. Technical report, 2016. Site web : http://www.omg.org/spec/MOF/2.5.1, (Accédé : Novembre 2017).
[229] X. Blanc, MDA en action, Eyrolles, 2005.
[230] OMG, "Common Warehouse Metamodel (CWM)", version 1.1, Disponible online : http://www.omg.org/spec/CWM/About-CWM/, (accédé: Novembre 2017).
[231] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1 - January 2011, Disponible online : http://www.omg.org/spec/QVT/1.1/PDF/, (accédé : Novembre 2017).
[232] TCS (Tata Consulting Services), Modelmorf qvt/relation implementation, 2007.
[233] France Telecom, SmartQVT : An open source model transformation tool implementing the MOF 2.0 QVTOperational language, 2007.
[234] James Davis, "GME: the generic modeling environment", In OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 82–83, New York, NY, USA, ACM, ISBN 1-58113-751-6, 2003.
[235] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent and John Crupi, "Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools", Wiley & Sons, 2004.
[236] Case, A. F.: *Computer-aided software engineering (CASE): technology for improving software development productivity*. SIGMIS Database, 17(1):35–43, 1985, ISSN 0095-0033.
[237] Montgomery, S. L.: *AD/Cycle: IBM's Framework for Application Development and Case*. Van Nostrand Reinhold, 1991.
[238] Carnegie Mellon Software Engineering Institute: *Computer-Aided Software Engineering (CASE) Environments*. Online : http://www.sei.cmu.edu/legacy/case/case_whatis.html, (accédé : Novembre 2017).
[239] Brown, A. W., D. J. Carney, E. J. Morris, D. B. Smith, and P. F. Zarrella: *Principles of CASE tool integration*. Oxford University Press, Inc., New York, NY, USA, 1994, ISBN 0-19-509478-6.
[240] Sztipanovits, J. and G. Karsai: *Model-integrated computing*. IEEE Computer, 30(4):110–111, 1997, ISSN 0018-9162.
[241] Gabor Karsai and Aditya Agrawal, "Graph Transformations in OMG's Model-Driven Architecture", Lecture Notes in Computer Science, Vol 3062, 243-259, Springer Berlin /Heidelberg, juillet 2004.
[242] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jorg Kreowski, Sabine Kuske, Detlef Pump, Andy Schürr and Gabriele Taentzer, "Graph transformation for specification and programming", Science of Computer programming, vol 34, NO°1, pages 1-54, Avril 1999.
[243] Grzegorz Rozenberg, "Handbook of Graph Grammars and Computing by Graph Transformation", Vol.1. World Scientific, 1999.
[244] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L.Wendehals, and A. Zündorf, "*Tool Integration at the Meta-Model Level within the Fujaba Tool Suite*", International Journal on Software Tools for Technology Transfer, vol. 6(3), pp. 203-218, 2004.
[245] A. Konigs, "*Model Transformations with Triple Graph Grammars*", in Model Transformations in Practice Workshop at MoDELS, Jamaica, 2005.
[246] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, D. Pataricza, A.and Varro. "*Viatra -Visual Automated Transformations for Formal Verification and Validation of UML Models*", In Proceedings of the 17th IEEE International Conference on Automated Software Engineering, UK, pp. 267-270, 2002.

[247] "*Eclipse Modeling Galileo*", Online: http://www.eclipse.org/downloads/packages/release/galileo/, (consulté : Novembre 2017).
[248] A. Schürr, A. J. Winter and A. Zündorf, "*The PROGRES Approach: Language and Environment*", World Scientific Publishing, vol. 2, pp. 487-550, 1999.
[249] "Python", Online : http://www.python.org, (accédé : Novembre 2017).
[250] Voormann, H. *Luna Rising* [Online]. 2014. Disponible : http://eclipsehowl.wordpress.com/2014/06/25/luna-rising/, (Consulté Novembre 2017).
[251] Dave Steinberg, Frank Budinsky, Ed Merks et Marcelo Paternostro. Emf : eclipse modeling framework. Pearson Education, 2008. (Cité en page 124.)
[252] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. Eclipse Modeling Framework 2nd Edition, Addison Wesley, 2009.
[253] Andy Schürr. Specification of graph translators with triple graph grammars. In Graph-Theoretic Concepts in Computer Science, pages 151–163. Springer, 1995.
[254] Ekkart Kindler and Robert Wagne. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Technical Report, University of Paderborn, D-33098 Paderborn, Germany, 2007.
[255] Alexander Konigs. Model transformation with triple graph grammar. Septembre, 2005.
[256] TGG interpreter. Site web : http://www-old.cs.uni-paderborn.de/en/research-group/software-engineering/research/projects/tgg-interpreter.html, (Consulté Novembre 2017).
[257] Joel Greenyer. A study of model transformation technologies: Reconciling TGGs with QVT. Master's thesis, University of Paderborn, 2006.
[258] Joel Greenyer and Jan Rieke. Applying advanced tgg concepts for a complex transformation of sequence diagram specifications to timed game automata. In Applications of Graph Transformations with Industrial Relevance, pages 222–237. Springer, 2012.
[259] Klatt, B. Xpand : A Closer Look at the model2text Transformation Language. Chair of Software Design and Quality at the University of Karlsruhe, 2007, (Cité page 18).
[260] Oldevik, J. MOFScript Eclipse Plug-In : Metamodel-Based Code Generation. Eclipse Technology eXchange workshop (eTX), Nantes, France, 2006, (Cité page 18).
[261] JET. Java Emission Template, 2004. Tutoriel disponible sur : https://www.eclipse.org/articles/Article-JET/jet_tutorial1.html, (Consulté Novembre 2017).
[262] Xpand Documentation, 2014. Téléchargeable sur : http://git.eclipse.org/c/m2t/org.eclipse.xpand.git/plain/doc/org.eclipse.xpand.doc/manual/xpand_reference.pdf, (Accédé Novembre 2017).
[263] Messaoud Bendiaf, Mustapha Bourahla, Malika Boudia and Seidali Rehab. A Model Transformation Approach for Specifying Real-Time Systems and Its Verification Using RT-Maude. International Journal of Information Technology and Web Engineering (IJITWE), volume 12 – issue 4. October-December 2017.
[264] Rivera, J. E., Francisco, D., & Antonio V. (2008). *A Metamodel For Maude* (Tech. rep.). University of Málaga, Spain (cit. on pp.170, 176).
[265] Eclipse Modeling M2T. Site web: https://eclipse.org/modeling/m2t/, (Consulté Novembre 2017).
[266] Messaoud Bendiaf, Mustapha Bourahla, Abdelhak Boubetra and Makhlouf Naili. Analyzing and Verification of Real-Time and Hybrid Systems Using the Maude LTL Model Checker. International Conference of Computing for Engineering and Sciences (ICCES'2015). Istanbul, Turkey, July 29 – August 2, 2015.
[267] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In Proceedings of the 16th IEEE RealTime Systems Symposium, RTSS '95. IEEE Computer Society, 1995.