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

Meliouh Amel

THÈME

UML et Model-Checking pour la Modélisation et la Vérification des Systèmes Embarqués

Soutenue le: 11/03/2021

Devant le jury composé de :

Pr.Kazar Okba	Université de Biskra	Président
Pr.Chaoui Allaoua	Université de Constantine2	Rapporteur
Pr.Bennoui Hammadi	Université de Biskra	Examinateur
Pr.Kahloul Laid	Université de Biskra	Examinateur
Dr. Kerkouche Elhilali	Université de Jijel	Examinateur
Dr. Maarouk Toufik	Université de Khenchela	Examinateur

DEDICACE

Je dédie ce travail à :

La mémoire de mon père ;

Ma chère mère ;

Mon Mari et mon petit prince Ahmed Younes ;

Mes Chers frères et sœurs ;

Mes beaux-frères ;

Mes nièces et neveux ;

La famille de mon mari.

REMERCIEMENTS

Mon premier remerciement va également à mon dieu tout puissant pour m'avoir aidé à achever ce travail.

Mes remerciements vont aussi à mon encadreur Pr. Allaoua Chaoui, d'avoir assuré l'encadrement de cette thèse ainsi pour tous ses conseils, sa patience, sa confiance, ses encouragements, tout au long de ce travail.

Je remercie Mon mari pour son aide précieuse.

Un grand Merci au Pr. Kazar Okba pour son encouragement et son soutien moral

Je tiens également à exprimer toute ma grande gratitude aux membres de jury :

Pr.Kazar Okba, président de jury
Pr.Bennoui Hammadi
Pr.Kahloul Laid
Dr. Kerkouche Elhilali
Dr. Maarouk Toufik

D'avoir accepté de juger ce travail.

ملخـــــص

يقترح هذا العمل منهجًا للنمذجة والتحقق الرسمي من سير عمل الأنظمة المدمجة. يستند هذا المنهج على إنشاء نماذج لتمثيل السلوك الزمني للنظام وإستخراج أوتوماتيكيا مواصفاته. العمل الرئيسي يرتكز على الاستخدام المشترك لمجموعة من المخططات للغة UML المضاف إليها مجموعة من المفاهيم الزمنية (Real-Time Statechart, Real-Time Collaboration diagrams) لنمذجة سلوك النظام و لغة Maude للتحقق من التشغيل الزمني السليم للنظام. كأول مرحلة، تم تطوير أداة للنمذجة بلغةUML، ثم تم تنفيذ قواعد الرسم البياني لتحويل النماذج المنجزة إلى مواصفات مكافئة مكتوبة بلغة Maude. من خلال هذه المواصفات للنظام المتحصل عليها، يمكن استخدام أداة مراقبة للنماذج (model checking) للتحقق من مدى صحة بعض الخصائص الزمنية المكتوبة في Real Time Temporal Logic(MITL).

الكلمات المفتاحية:

Real-Time Statechart diagram, Real-Time Collaboration diagram, UML, Meta-modeling, Maude **لغة**, ATOM^3, MITL, إستخراج مواصفات مكافئة, قواعد الرسم البياني

Abstract

This work proposes a formal approach for modeling and verification of embedded systems. The approach relies on an automated modeling and code generation based on the systems' temporal behavior. The key concept is the combined use of a set of UML behavior diagrams extended with timing annotations (Real-Time Statechart and Real-Time Collaboration diagrams) for system modeling and Maude language for verification. First, UML modeling tool is developed. Then, a graph grammar is executed to generate automatically an equivalent Maude specification. The approach is based on code generation. This is why it is possible to use an available model checking tool to verify certain timed properties represented in Real Time Temporal Logic (MITL).

Keywords:

UML, Real-Time Statechart diagram, Real-Time Collaboration diagram, Code Generation, Meta-modeling, Graph Grammar, ATOM^3, MITL, Model checking, Maude language.

Résumé

Ce travail présente une approche formelle de modélisation et de vérification des systèmes embarqués. L'approche repose sur une modélisation du comportement temporel du système et une génération automatique de code. Le concept clé est l'utilisation combinée d'un ensemble de diagrammes comportementaux d'UML étendu par des annotations temporelles (diagrammes Statechart temps réel et de collaboration temps réel) pour la modélisation du système et le langage Maude pour la vérification. Tout d'abord, un outil de modélisation UML est développé, ensuite, une grammaire de graphes est exécutée pour générer automatiquement des spécifications en Maude équivalentes. L'approche repose sur la génération de code, donc il est possible d'utiliser un outil de vérification de modèles (Model-cheking) pour vérifier certaines propriétés temporelles représentées dans la logique temporelle temps réel (MITL).

Mots clés :

UML, Real-Time Statechart diagram, Real-Time Collaboration diagram, génération de code, Méta-modélisation, Grammaire de graphes, ATOM^3, MITL, Model checking, language Maude.

Table des matières

Table des figures

Liste des tableaux

Introduction Générale

Contexte général

Les nouvelles technologies s'accompagnant toujours de performances plus grandes permettent la conception de systèmes novateurs qui suscitent l'engouement du grand public ainsi qu'une demande croissante de sa part. Ce cercle sans fin justifie la durée de vie de plus en plus courte des systèmes actuels et se retrouve dans de nombreux domaines.

Aujourd'hui, des systèmes de composants mélangeant électronique et programmes informatiques se trouvent dans de nombreux domaines : le transport (avionique, espace, automobile, ferroviaire), dans les appareils électriques et électroniques (appareils photo, jouets, postes de télévision, électroménager, systèmes audio, téléphones portables), dans la distribution d'énergie, dans l'automatisation, …, etc.

On appelle systèmes embarqués les systèmes qui sont destinés à être intégrés dans de tels appareils et qui offrent des fonctionnalités particulières. Un système embarqué est un système complexe qui intègre du logiciel et du matériel conçus ensemble afin de fournir des fonctionnalités données [Cse17]. Il contient généralement un ou plusieurs microprocesseurs destinés à exécuter un ensemble de programmes définis lors de la conception et stockés dans des mémoires.

Parmi ces systèmes embarqués, certains sont qualifiés de critiques lorsque de leur bon fonctionnement dépendent la vie d'êtres humains ou d'énormes sommes d'argent. On distingue deux types de dysfonctionnement des systèmes. Il y a d'une part les problèmes matériels, qui se produisent lorsqu'un composant connaît un endommagement physique. D'autre part, il y a les problèmes conceptuels, appelés bugs par abus de langage, qui surviennent lorsqu'un composant ne réalise pas précisément les fonctionnalités pour lesquelles il a été conçu [Aya10]. En outre, l'évolution rapide et continue des systèmes embarqués modernes a provoqué de nouveaux défis. Par exemple, la conception des processus complexes qui causent des retards dans le temps de commercialisation et la conséquente augmentation des coûts globaux. Ces systèmes sont plus enclins aux erreurs et par conséquence il devient prioritaire de fournir aux concepteurs des outils effectifs et efficaces pour les aider à surmonter les difficultés liées à la conception des systèmes globales, pour la vérification et pour la validation.

Obéissant donc à une intégration de plus en plus forte, la conception de ces systèmes devient également de plus en plus complexe. Tout logiciel d'un système embarqué étant constitué de fonctions, également appelées tâches, le problème revient à vérifier que celles-ci seront toujours exécutées en temps et en heure. Or, dès lors qu'une nouvelle fonctionnalité est intégrée à un système, celle-ci s'accompagne de nouvelles tâches qui affectent le comportement de celle déjà présentés dans son application et remet donc en question leur vérification. Prenons

pour exemple une automobile dotée, entre autres choses, d'un système ABS. A chaque ajout d'un nouveau dispositif d'assistance électronique, le constructeur doit vérifier que celui-ci fonctionnera correctement et n'entraînera aucune défaillance des systèmes déjà présents dans le véhicule. En conséquence, le constructeur doit s'assurer que l'ensemble des tâches, y compris les anciennes comme celles liées à la fonction ABS, respecteront toujours leur échéance. Il a donc fallu trouver des façons de développer des systèmes sûrs.

Problématique

Les travaux de recherche sur les systèmes embarqués visent le développement de modèles, de langages et d'outils pour la conception de systèmes embarqués. Ces derniers sont construits non seulement de manière modulaire par un grand nombre de composants mais aussi par la diversité de ces composants. Ceci implique que ces composants sont Modélisés dans différents langages de modélisation ou formalismes [Ker11]**.** La vérification des propriétés de ces systèmes est reconnue comme un problème difficile et se heurte à plusieurs problèmes. Un premier obstacle apparaît au niveau du choix des techniques de modélisation ou langages de modélisation utilisés pour modéliser les différents composants du système. Si le langage de modélisation est trop expressif, alors on ne peut pas, mathématiquement, l'analyser de manière automatique. Un second obstacle est lié à la vérification du comportement global du système. Les modèles qui représentent un tel système sont modélisés avec des langages ou des formalismes différents, ce qui rend tout raisonnement global sur le système difficile.

L'objectif principal dans ce domaine est d'étudier des modèles formels permettant de décrire ces systèmes et leurs contraintes (conception, spécification), de les construire (programmation, simulation, synthèse, exécution), et de les analyser (validation, vérification).

La vérification représente un souci majeur pour un grand nombre de systèmes embarqués. D'où l'importance de la validation de ces systèmes, c'est-à-dire test, vérification et certification. Il y a donc un besoin réel et pressant de développer des méthodes et outils efficaces pour la vérification des systèmes embarqués. Car la moindre faille sera exploitée par des malveillants potentiels.

Les phases de vérification et de validation permettent de contrôler que le système satisfait les propriétés attendues. Il en résulte que tout problème lié au comportement qualitatif (Sûreté, équité, absence de blocage, etc.…), comme au comportement quantitatif (perte de messages, vitesse moyenne de transmission, etc.…) est détecté et corrigé très tôt dans le cycle de développement. Une des pistes prometteuses pour la réduction de ces coûts de vérification

est l'utilisation des méthodes formelles. Ces méthodes s'appuient sur des fondements mathématiques et permettent d'effectuer des tâches de vérification à forte valeur ajoutée au cours du développement [Fer14]. Pour pouvoir être réalisée, la vérification impose une description formelle du système, ainsi qu'une spécification formelle de ses propriétés.

Des approches d'ingénierie guidée par des modèles (IDM) sont de plus en plus utilisées dans l'industrie dans l'objectif de maîtriser cette complexité au niveau de la conception [Chk10].

L'Architecture Dirigée par les Modèles (Le Model Driven Architecture (MDA) est une démarche de développement proposée par l'Object Management Group (OMG) [OMG99]. Elle permet de séparer les spécifications fonctionnelles d'un système des spécifications de son implémentation sur une plate-forme donnée. L'approche MDA permet de réaliser le même modèle sur plusieurs plateformes grâce à des projections standardisées.

La mise en œuvre du MDA est entièrement basée sur les modèles et leurs transformations. Cette approche consiste à manipuler différents modèles de l'application à produire, depuis une description très abstraite jusqu'à une représentation correspondant à l'implantation effective du système [Poo01].

L'approche MDA peut être instanciée en utilisant différents langages. C'est pourquoi un certain nombre de langages permettant cette description sont apparus et offrent à ce jour un certain nombre de fonctionnalités. La plupart des approches guidées par des modèles se basent soit sur :

- Le langage de modélisation unifié (*Unified Modeling Language*, UML) [UML20], qui est un langage de modélisation à usage général et ses profiles comme MARTE (*Modeling and Analysis of Real-Time and Embedded*) [Mar03]. Si UML permet une grande expressivité, il ne transporte par en lui-même une sémantique précise pour décrire les architectures [UML20].
- Des langages de description d'architecture (*Architecture Description Language*, ADL), qui sont des langages propres à des domaines particuliers. Un ADL est un langage qui permet la modélisation d'une architecture conceptuelle d'un système logiciel et/ou matériel. Il fournit une syntaxe concrète et une structure générique (Framework) conceptuelle pour caractériser les architectures.
- Des langages spécifiques, conçus pour répondre à des besoins particuliers de modélisation et d'analyse, par exemple Behavior Interaction Priority (BIP) [Ana06], Fractal [Bru06] et Ptolemy [Pto14].

UML (Unified Modeling Language), le langage unifié de modélisation, est un excellent exemple d'unification des notations utilisées dans les méthodologies d'analyse et de conception orientés objet [UML20]. Il est devenu depuis quelques années un standard incontournable dans la modélisation des systèmes. Bien qu'UML est un langage riche, doté d'une notation ouverte et largement utilisé, UML souffre d'un manque de sémantique formelle [Gag07]. Les modèles UML développés pourront donc contenir des incohérences ou des inconsistances difficiles (voire impossibles dans le cas de certains systèmes complexes) à détecter manuellement [Dar02]. Par conséquence, Cela affecte l'analyse formelle et la vérification de la conception des systèmes. Donc UML a besoin d'une base sémantique bien définie pour ses notations. Ce fait est dû que UML est un langage graphique et semi-formelle et sa sémantique n'est pas formellement spécifiée.

D'autre part, les méthodes formelles offrent une solution intéressante à ce problème. Les spécifications formelles auront pour effet d'éliminer les ambigüités au niveau de l'interprétation des modèles UML. Une combinaison appropriée des techniques orientées objets et des méthodes formelles peut rendre le développement logiciel plus rigoureux.

Une difficulté se présente, en raison de la variété des modèles formels servant à représenter les caractéristiques non fonctionnelles des composants, et les exigences sur le système complet. De nombreux formalismes de spécification dédiés aux systèmes embarqués ont été proposés en littérature tels que les automates communicants, Systèmes de transition, algèbres de processus CCS, CSP, LOTOS, Estelle, Promela, etc.….

Le langage formel Maude, basé sur la logique de réécriture, possède une sémantique mathématique. Il a été conçu spécifiquement pour prendre en considération les particularités des systèmes concurrents [Mcc03].

La logique de réécriture de Maude est considérée comme l'une des langages puissants dans la spécification et la vérification des systèmes concurrents [Mcc03]. Le model checking intégré dans l'environnement Maude permet de valider le comportement d'un système, utilisant des propriétés exprimées dans une logique donnée (les logiques temporelles, etc.) [Gag07].

L'application des techniques de vérification automatiques de modèles (model checking) dans le cycle de développement des systèmes embarqués, fournissent des preuves de propriétés telles que l'absence d'erreurs l'exécution ou de satisfiabilité de propriétés temporelles.

Model checking consiste à construire un modèle fini du système analysé et à vérifier les propriétés souhaitées de ce modèle. La vérification demande une exploration complète ou partielle du modèle. Les avantages principaux de cette technique sont : la possibilité de rendre

l'exploration du modèle automatique et de faciliter la production de contre-exemples, lorsque la propriété est violée.

Etude des travaux réalisés

Dans ce contexte, plusieurs travaux ont été réalisés. Dans [Cha18], Les diagrammes d'interaction d'UML 2.0 (IOD) sont transformés vers la logique de réécriture du langage Maude, dans le but de vérifier certaines propriétés pour les systèmes embarqués. Il existe également des travaux similaires, où des définitions sémantiques formelles sont ajoutées aux diagrammes UML pour les transformer systématiquement en un langages de spécification appropriés pour un but de vérification. Dans [Dar02], les auteurs ont présenté une méthode de vérification des diagrammes Statechart d'UML générés lors du processus de développement d'un système embarqué. Ce travail utilise l'outil RATIONAL ROSE pour la modélisation et l'outil de transformation de graphes VIATRA pour transformer les statecharts en Extended Hierarchical Automata (EHA), en raison de vérification. Dans [Vas18], ils ont proposé une analyse statique basée sur des interprétations abstraites, sur SMT-based software Model checking et sur des Preuve déductive du logiciel embarqué pour les automobiles. [Bou09] a proposé une approche d'analyse des systèmes embarqués temps réel, en se basant sur la vérification de la planification des tâches et des exigences pour l'ensemble du système, en utilisant le formalisme des automates temporisés étendus et l'outil TIMES pour la verification. Dans le travail de [Lui01], un modèle formel de calcul pour les systèmes embarqués temps réel basés sur des réseaux de Petri a été défini. Ensuite, une procédure systématique pour transformer le modèle réseaux de Petri vers des automates temporisés pour pouvoir utiliser des outils de vérification disponibles. Dans [Fat09], Les diagrammes d'activité avec CGA (Coarse Grained Actions) et les statecharts d'UML 2.0 sont utilisés pour modéliser les Data/Control Driven Embedded Systems avec l'outil Rhapsody. Les deux diagrammes ont été transformés vers une spécification en langage Maude, à partir de laquelle certaines propriétés peuvent être vérifiées par le model checker.

Contribution

Vérifier un système embarqué consiste à démontrer que l'exécution de n'importe laquelle de ses tâches se termine toujours avant l'échéance, aussi appelée contrainte temporelle, associée à celle-ci.

A la lumière de ces constats, nous avons choisi d'étudier au travers de cette thèse la possibilité de modélisation et de vérification d'un système embarqué par la combinaison des méthodes formelles avec les techniques développées en MDA.

Cette thèse s'intéresse au dimensionnement temps réel de systèmes embarqués monoprocesseur. Notre objectif consiste à mettre au point un outil pour garantir, avant son déploiement, qu'un système embarqué respectera toujours ses contraintes temporelles lorsque celles-ci seront exécutées sur une architecture monoprocesseur donnée.

Ce travail vise à illustrer l'efficacité d'une stratégie basée sur la combinaison d'UML et le langage Maude. Une première partie consiste à proposer un environnement de modélisation capable d'assister les développeurs dans leurs modélisations des systèmes embarqués avec des modèles graphiques. Cet environnement utilise deux diagrammes d'UML étendus par des horloges logiques (diagrammes d'état-transition et de collaboration temps réel). Cette étape est précédée par une étape de méta-modélisation. La Méta-modélisation apporte donc la flexibilité nécessaire à la fourniture de moyens adaptés à la diversité des aspects/composants des systèmes complexes. UML manque d'une sémantique opérationnelle rigoureuse. Nous abordons ce problème en fournissant une génération automatique d'un code en langage Maude depuis les diagrammes UML. Le code généré est le résultat d'exécution d'une grammaire de transformation bien définie, la notion de transformation de modèles joue un rôle fondamental afin de rendre opérationnels les modèles (pour la génération de code, de documentation et de test, la validation, la vérification, l'exécution, etc.) [Ker11]. Une deuxième partie consiste à définir un ensemble de propriétés temporelles exprimées en logique temporelle temps réel (MITL). Ces propriétés seront vérifiées par Maude model-checker par rapport aux spécifications du système.

Structure du manuscrit

Notre thèse est organisée en quatre chapitres. : Dans le premier chapitre, nous présentons les concepts essentiels ainsi que la terminologie que nous utiliserons tout au long de ce manuscrit, concernant les systèmes embarqués. Nous abordons dans le deuxième chapitre les principes clés de l'ingénierie dirigée par les modèles IDM et les différentes variantes d'ingénierie centrées sur les modèles. Dans le troisième chapitre nous présentons un panorama sur la logique de réécriture et le langage Maude. Le quatrième chapitre est consacré à la présentation de notre contribution, qui a pour but de développer un outil de vérification des systèmes embarqués temps réel en utilisant UML et le langage Maude. Cet outil est basé sur la Méta-modélisation et une Grammaire de Graphe qui assure la génération automatique des

spécifications formelles en langage Maude depuis les digrammes d'états-transitions temps réel et le diagramme de collaboration temps réel en exploitant les facilités de l'outil ATOM^3 : un outil de méta-modélisation et transformation de modèles.

Enfin, La conclusion résume les points essentiels de ce travail et présente les perspectives de recherches suggérées par ce travail.

Chapitre 01 :

Modélisation et vérification des systèmes embarqués

1. Introduction

Les systèmes embarqués sont des systèmes électroniques intégrant du logiciel et du matériel, inclus dans des objets ou des systèmes qui ne sont pas perçus comme des ordinateurs par leurs utilisateurs [Chk10]. Actuellement, les systèmes embarqués sont présents dans des applications de plus en plus nombreuses, par exemple les cartes à puce, les systèmes mobiles communicants (tels les téléphones mobiles, les mobiles dans les réseaux ad hoc), l'automobile, l'avionique, les capteurs intelligents, la santé et l'électronique grand public

La complexité croissante de la technologie de l'embarqué fait que le processus de développement des systèmes embarqués doit suivre une démarche rigoureuse basée sur une sémantique formelle pour consolider la description structurelle des composants de ces systèmes et évaluer au plus tôt leur comportement.

L'objectif de ce chapitre est de présenter un état de l'art sur les approches de modélisation et de vérification des systèmes embarqués. D'abord, nous présentons les notions de base sur les systèmes embarqués, ensuite les différentes étapes du cycle de développement classique de tels systèmes en se concentrant particulièrement sur la conception orientée modèles.

Nous énumérons ensuite les approches de modélisation de ces systèmes, à savoir les profils et les langages de modélisation.

Deux approches de base ont été alors utilisées pour la vérification des systèmes. Une première technique de vérification consiste en l'élaboration d'un prototype du système, qui est testé dans son environnement. Il s'agit là de la simulation du comportement du système : le comportement simulé est alors comparé au comportement attendu. S'il y a adéquation, le système est vérifié. D'autres techniques de vérification ont été étudiées : elles consistent à remplacer la vérification expérimentale du système par une preuve de sa correction. On cherche à montrer formellement que le système est en adéquation avec sa spécification, ou qu'il vérifie un ensemble de propriétés décrivant partiellement la spécification.

2. Les systèmes embarqués

2.1 Définitions

Selon la langue de Molière, le mot embarqué, est dérivé du verbe « embarquer » qui Désigne le fait de « mettre quelque chose à bord d'un navire, un avion ou d'un véhicule ». En informatique, parlant un système embarqué (appelé aussi un système enfui) peut être défini comme : « Un système électronique et informatique autonome, qui est dédié à une tâche bien précise » [Chk10]. Un système embarqué (SE) est un système informatisé spécialisé qui

constitue une partie Intégrante d'un système plus large ou une machine. Typiquement, c'est un système sur un seul processeur et dont les programmes sont stockés en ROM. A priori, tous les systèmes qui ont des interfaces digitales (i.e. montre, caméra, voiture...) peuvent être considérés comme des SE. Certains SE ont un système d'exploitation et d'autres non car toute leur logique peut être implantée en un seul programme [Mam10].

- Un système embarqué est une combinaison de logiciel et matériel, avec des capacités fixes ou programmables, qui est spécialement conçu pour un type d'application particulier. Les distributeurs automatiques de boissons, les automobiles, les équipements médicaux, les caméras, les avions, les jouets, et les téléphones portables sont des exemples de systèmes qui abritent des SE. Les SE programmables sont dotés d'interfaces de programmation et leur programmation est une activité spécialisée [Mam10].
- On peut distinguer deux catégories de systèmes embarqués : les systèmes autonomes Et les systèmes enfouis :
  - Un système autonome correspond à un équipement autonome contenant une intelligence qui lui permet d'être en interaction directe avec l'environnement dans lequel il est placé. Il s'agit des téléphones portables, agendas personnels électroniques ou GPS.
  - Un système enfoui (souvent invisible à l'utilisateur) est un ensemble cohérent de constituants informatiques (matériel et logiciel), d'un équipement auquel il donne la capacité de remplir un ensemble de missions spécifiques. Il s'agit d'un système physique sous-jacent avec lequel le logiciel interagit et qu'il contrôle [Mar03].
- En quoi les systèmes embarqués diffèrent-ils de notre PC ? Les systèmes embarqués sont dédiés à des tâches spécifiques, tandis que les PC ont des plateformes génériques. Ces systèmes ont de loin moins de ressources que les PC, et doivent souvent fonctionner dans des conditions environnementales extrêmes [Ben11].
- Les systèmes embarqués ont souvent des contraintes d'énergie et des contraintes temps-réel. 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). C'est pourquoi, un système embarqué doit utiliser un système d'exploitation temps-réel RTOS, plutôt que windows, unix, etc [Ber02].
- Pourquoi le développement des systèmes embarqués est différent ? Généralement, les systèmes embarqués sont supportés par une grande sélection de processeurs et architectures de processeurs, et stockent souvent tout leur code objet dans la ROM. Aussi, les défaillances logicielles sont beaucoup plus graves pour les

systèmes embarqués que pour les applications sur PC. Pour cela, les systèmes embarqués exigent des outils spécialisés et des méthodes de conception plus efficaces [Ber02].

2.2 Caractéristiques d'un système embarqué

Un système embarqué est caractérisé par un ensemble de caractéristiques à savoir :

- **Criticité :** si la fonction du système dans son environnement est jugée critique, c'est-à-dire si ces défaillances sont catastrophiques, alors le système de contrôle informatique est également critique ; c'est le cas par exemple d'un système de régulation d'une réaction nucléaire ou d'un système de commande de vol du type X-by-wire. Notons que pour les systèmes dont la durée de vie (ou de mission) est courte, on peut parfois privilégier l'exigence de disponibilité opérationnelle ou de réalisation de la mission, à savoir la capacité du système à pouvoir répondre à tout moment à une sollicitation. C'est par exemple le cas des systèmes militaires ou des lanceurs spatiaux.

- **Temps-réel** : si leur environnement est dynamique, voire instable, alors le système de contrôle informatique est soumis à des exigences de temps de réponse, c'est-à-dire des exigences portant sur les délais introduits par le système informatique entre des stimuli et des actions correspondantes. Ces délais peuvent être provoqués par les temps d'exécution des tâches et les temps de transmission des messages (incluant les temps d'accès aux ressources CPU et réseaux). Le non-respect de ces exigences peut rendre la boucle de contrôle instable et provoquer des défaillances catastrophiques.

- **Réactivité** : un système embarqué doit par définition surveiller et contrôler un environnement. Si cet environnement présente lui-même un comportement changeant ou de type événementiel (ont dit aussi "sautillant" par opposition à des environnements plus réguliers), il est nécessaire que le système informatique soit capable de s'adapter à ces bouffées d'événements et mettant en œuvre des calculs et des communications également événementiels.

- **Robustesse** : un système embarqué est autonome vis à vis de l'humain : il n'est généralement pas prévu qu'un opérateur intervienne sur le logiciel si celui- ci est perdu. Par conséquent, il est vital que le système soit capable de réagir de façon satisfaisante (et déterministe) même si son environnement se comporte de manière imprévue, improbable ou fantaisiste. Le concepteur du système ne peut en aucun cas faire d'hypothèse sur le comportement de l'environnement [Wol06].

- **Faible encombrement**, faible poids :

– Electronique « pocket PC », applications portables où l'on doit minimiser la consommation électrique (bio instrumentation...).

– Difficulté pour réaliser le packaging afin de faire cohabiter sur une faible surface électronique analogique, électronique numérique, RF sans interférences.

- **Faible consommation** :

– Batterie de 8 heures et plus (PC portable : 2 heures).

- **Environnement** :

– Température, vibrations, chocs, variations d'alimentation, interférences RF, corrosion, eau, feu, radiations.

– Le système n'évolue pas dans un environnement contrôlé.

– Prise en compte des évolutions des caractéristiques des composants en fonction de la température, des radiations

- **Fonctionnement critique pour la sécurité des personnes. Sûreté** :

– Le système doit toujours fonctionner correctement.

– Sûreté à faible coût avec une redondance minimale.

– Sûreté de fonctionnement du logiciel

– Système opérationnel même quand un composant électronique lâche.

– Choix entre un design tout électronique ou électromécanique

- Beaucoup de systèmes embarqués sont fabriqués en grande série et doivent avoir des prix de revient extrêmement faibles, ce qui induit :

– Une faible capacité mémoire.

– Un petit processeur (4 bits). Petit mais en grand nombre

- La consommation est un point critique pour les systèmes avec autonomie.

– Une consommation excessive augmente le prix de revient du système embarqué car il faut alors des batteries de forte capacité.

- **Faible coût** :

– Optimisation du prix de revient [Kad16].

2.3 Architectures des systèmes embarqués

On peut distinguer trois types d'architectures embarquées : les architectures centralisées que l'on trouve plus fréquemment dans des engins tels que les missiles ou les lanceurs, les architectures fédérées plus communes aux aéronefs civils et que l'on peut interpréter comme étant des architectures localement centralisées et globalement asynchrones, et enfin les architectures réparties plutôt mises en œuvre dans les aéronefs militaires.

2.3.1 Les architectures centralisées

A l'époque des premières machines informatiques embarquées, le nombre et la complexité des fonctions mises en œuvre étaient relativement restreints. Un mini-calculateur suffisait à centraliser l'acquisition, le traitement et la mémorisation des informations. Les avantages de ce type d'architecture sont nombreux :

- Le calculateur est généralement situé dans un endroit accessible, facilitant ainsi l'intégration puis la maintenance du système embarqué.
- Le calculateur est généralement situé dans un endroit protégé, à l'abri des perturbations (interférences, risques d'explosion . . .) pouvant être provoquées par les autres composantes de l'engin, simplifiant ainsi l'analyse des pannes de l'avionique et, par suite, sa certification.
- Le système étant centralisé, son architecture matérielle en est simplifiée.

En revanche, les inconvénients inhérents à une architecture centralisée sont également importants, et souvent rédhibitoires :

- Un système centralisé est plus vulnérable aux défaillances.
- Le système de traitement étant localisé en un endroit unique, il est par conséquent loin des capteurs ou des actionneurs (voire des deux). Une architecture centralisée nécessite donc un très grand nombre de bus analogiques de grande longueur.

2.3.2 Les architectures réparties

Dans de telles architectures, les fonctions ne sont plus réalisées par un calculateur, mais par un ensemble d'équipements informatiques communicants. On parle alors souvent de "chaîne fonctionnelle" (bien qu'une telle architecture ne constitue pas une chaîne à proprement parler mais un réseau de processus).

Une telle répartition est rendue nécessaire souvent par la géographie de l'application, mais aussi par l'utilisation de capteurs et actionneurs offrant localement des capacités informatiques (on parle de capteurs et actionneurs "intelligents").

2.3.3 Les architectures fédérées

Une architecture fédérée est généralement constituée d'un ensemble de sous-systèmes réalisant chacun une fonction embarquée (commandes de vol, pilote automatique), pilotant des actionneurs, ou élaborant des informations numériques à partir d'informations produites par des capteurs. Ces sous-systèmes sont localisés en divers endroits de l'engin embarquant (un aéronef par exemple), généralement à proximité des capteurs ou des

actionneurs, et peuvent être considérés chacun pris séparément comme des sous-systèmes centralisés. En ce sens, une architecture fédérée peut être assimilée à un assemblage de sous-architectures centralisées

Les communications entre sous-systèmes sont assurées principalement par des flots de données asynchrones, la plupart du temps périodiques, et sont supportées par des canaux à diffusion et des mécanismes de files d'attentes en entrée de chaque sous-système.

- L'intérêt de ce type d'architecture est qu'elle permet naturellement la conception et le développement séparé de chaque sous-système (après une première spécification globale), puis l'évolution de l'architecture par ajouts, transformations ou suppressions de sous-systèmes [Wol06].

2.4 Structure générale d'un système embarqué

Un système embarqué est composé de plusieurs couches. La couche la plus abstraite est le logiciel spécifique de l'application. Cette couche communique avec une deuxième couche s'appelant le système d'exploitation qui contient deux parties : la première partie de la gestion des ressources et la deuxième partie de communication logicielle, suivie par la couche de réseau de communication matérielle embarquée, et à la fin on trouve la couche des composants matériels qui est la plus basse.

- La couche basse : comprend les composants matériels du système, parmi ces composants, on trouve les composants standards : (DSP, MCU, etc).

- La couche de réseau de communication matérielle embarquée : contient les dispositifs nécessaires à la communication entre les composants.

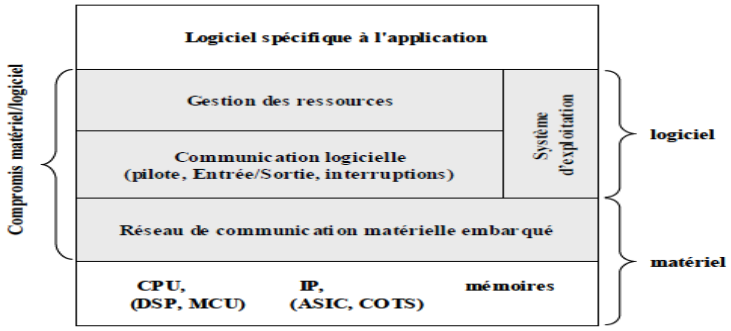


Figure 1.1 : structure d'un système embarqué [Bou09]

- Le système d'exploitation comprend deux couches :

  - Une couche de communication logicielle : composée des pilotes d'entrées/sorties, des interruptions, ainsi que les contrôleurs qui sont utilisées pour

contrôler le matériel. Le logiciel de cette couche et le matériel sont intimement liés. Le logiciel et le matériel sont séparés grâce à cette couche.

- Une couche de gestion des ressources : qui utilise des structures de données particulières et des accès non standards que l'on ne trouve pas des OS standards, par contre, les couches de gestion des ressources des OS standards sont souvent volumineuses afin d'être embarquées. La couche de gestion des ressources permet d'isoler l'application de l'architecture.

- La couche de logiciel spécifique de l'application : est la couche la plus abstraite, elle communique avec la couche de plus bas niveau qui est la couche système d'exploitation.

2.5 Les systèmes embarqués et le temps réel

Généralement, un système embarqué doit respecter :

- Des contraintes temporelles fortes *(*Hard Real Time).
- On y trouve enfoui un système d'exploitation ou un noyau Temps Réel (Real Time Operating System, RTOS).

Le Temps Réel est un concept un peu vague. On pourrait le définir comme : "Un système est dit Temps Réel lorsque l'information après acquisition et traitement reste encore pertinente" [Kad16].

Cela veut dire que dans le cas d'une information arrivant de façon périodique (sous forme d'une interruption périodique du système), les temps d'acquisition et de traitement doivent rester inférieurs à la période de rafraîchissement de cette information.

Pour cela, il faut que le noyau ou le système Temps Réel soit déterministe et préemptif pour toujours donner la main durant le prochain tick à la tâche de plus forte priorité prête.

Suivant les exigences temporelles d'un système embarqué, on peut distinguer deux classes de ces systèmes :

- Systèmes embarqués à contraintes temps réel strictes : Ces systèmes doivent impérativement respecter les contraintes de temps imposées par l'environnement, et le non-respect d'une contrainte temporelle peut avoir des conséquences catastrophiques. Le système de contrôle de trafic aérien et de conduite de missile sont deux exemples de ces systèmes.
- Systèmes embarqués à contraintes temps réel souples : 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 peut être acceptable par le système [Hoc13].

2.6 Les exigences de développement des systèmes embarqués

Les systèmes embarqués, principalement avioniques et automobiles, doivent également répondre à des exigences liées à leur processus de développement, et notamment des exigences ayant pour but de réduire leur temps de développement, de modification et de validation. On peut regrouper ces exigences en quatre catégories :

2.6.1Besoins de modèles à la fois locaux et globaux

Un système embarqué est souvent constitué de plusieurs entités communicantes (ne serait-ce que pour des besoins de redondance). Une spécification d'un système embarqué doit donc mettre en œuvre ces deux niveaux de description : un niveau local correspondant par exemple à une tâche ou un ensemble de tâches d'un sous-système (le pilote automatique par exemple), et un niveau global décrivant l'interaction entre ces composants (par exemple au niveau d'un avion complet).

Le niveau global est en pratique assez peu décrit, ou du moins ne fait l'objet bien souvent que de descriptions textuelles ou graphiques informelles. Ce constat peut s'expliquer par le fait que ce niveau n'est pas directement opérationnel. Le code généré automatiquement l'est à partir des spécifications des composants locaux, alors que le niveau global ne conduit généralement pas à un codage ou une réalisation automatique. Seuls les points de vue physique et intégration (notamment pour la description de l'utilisation des moyens de communication lorsque ceux-ci sont partagés) peuvent faire l'objet d'une description globale et détaillée. Pour autant, dès lors que l'on souhaite valider le fonctionnement du système (en considérant donc à la fois son comportement fonctionnel et ses performances de temps et ses occurrences de pannes), l'utilisation d'un modèle global, de simulation ou de vérification, est incontournable [Wol06].

2.6.2 Déterminisme

Une des exigences minimales imposée pour la certification d'un système embarqué critique est son déterminisme. Il doit afficher un comportement prévisible, et toujours identique, lorsqu'il est soumis plusieurs fois à un même scénario. Or, on sait qu'un tel niveau de déterminisme est très difficile à atteindre. Plusieurs sources de non-déterminisme peuvent en effet affecter un système : d'une part les imprécisions liées à ses organes physiques

(capteurs et actionneurs), et d'autre part les retards et éventuels décalages dus à l'asynchronisme du système (dans le cas d'une architecture globalement asynchrone bien entendu). En conséquence, le non déterminisme est souvent inhérent aux systèmes informatiques pour peu qu'ils soient un peu étendus et qu'ils contiennent des organes imprécis. L'exigence est donc, plutôt que de vouloir proscrire globalement un tel non-déterminisme, d'une part de l'interdire au niveau de chaque unité d'exécution élémentaire, puis d'autre part de montrer que le système dans sa globalité est stable ou tolérant à son indéterminisme intrinsèque [Wol06].

2.6.3 Vérification

L'exigence de validation est essentielle dès lors que l'on considère un système critique. Pour autant, cette exigence n'est pas toujours aisée. En pratique, elle requiert des moyens de simulation, de test sur des maquettes ou sur des prototypes (par exemple des essais en vol). Il est rare en revanche que l'on puisse utiliser des techniques de vérification automatique sur un système global principalement à cause du problème bien connu de l'explosion combinatoire. C'est là une des difficultés persistantes des méthodes formelles pour la conception de systèmes embarqués.

Néanmoins, cette vérification peut être plus ou moins facilitée selon la nature des modèles développés, selon qu'ils soient synchrones ou asynchrones, temporisés ou discrets, par exemple. En réalité, modéliser, c'est déjà préparer la vérification. Et inversement, exiger de pouvoir vérifier c'est d'abord exiger des modèles se prêtant à cette activité [Wol06].

2.6.4 Le problème de la modification par "delta" :

La dernière exigence que l'on mentionnera est liée au processus industriel. Certains systèmes sont parfois difficiles à spécifier précisément du premier coup, de part la complexité des fonctions mises en œuvre, mais aussi du nombre important de données d'environnement à traiter. C'est par exemple le cas d'un pilote automatique d'aéronef. De telles difficultés se traduisent concrètement par un nombre important de modifications et de réglages de la spécification du début du développement jusqu'à la phase de certification. D'autres systèmes peuvent être appelés à être modifiés en phase opérationnelle. C'est par exemple le cas des systèmes militaires auxquels on veut pouvoir demander de s'adapter en quelques jours sur le champ de bataille à une nouvelle situation tactique. Ces modifications peuvent aller de l'adaptation des moyens de communication d'un aéronef que l'on intègre dans un dispositif allié, jusqu'à l'embarquement d'un nouveau capteur ou d'une nouvelle

arme. Dans ce cas, il doit être possible de modifier le système embarqué (le plus souvent sa partie logicielle) sans que cela nécessite une longue phase de rectification.

Les modèles, support du développement et de la validation du système embarqué, doivent donc être suffisamment modulaires pour permettre de telles modifications locales [Wol06].

3. Approches de Développement des Systèmes Embarqués

Le développement des systèmes embarqués possède des caractéristiques et des besoins qui les distinguent particulièrement des autres systèmes. Significativement, les systèmes embarqués diffèrent des autres systèmes dans plusieurs aspects d'après, en exigeant :

- Un haut degré d'intégration des composants software et hardware.
- Un besoin temps-réel rigide.
- Un comportement très complexe, caractérisé par de nombreux scénarios compliqués.
- Un besoin en fiabilité et sécurité résistant, surtout pour les systèmes avec des missions critiques (safety-critical and/or mission-critical).
- Un traitement parallèle et souvent distribué.

Egalement, le développement de tels systèmes inclus l'utilisation de :

- Méthodes formelles pour la spécification, conception, vérification et test.
- Outils et méthodes de communication, synchronisation, etc.
- Co-conception (co-design) hardware/software.
- Langages de programmation spécialisés.
- Outils (des environnements) de développement spécialisé.
- Contraintes d'optimisation.

3.1Développement Classique

3.1.1 Principe générale

A 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 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 [Ber02].

Le développement dans ce cas se fait selon les phases suivantes :

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 [Val05].

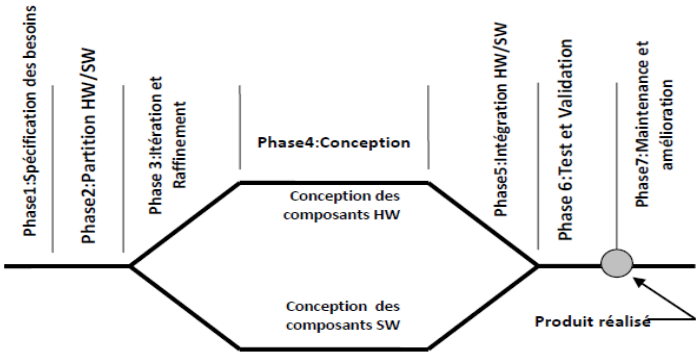


Figure 1.2 Cycle de vie de conception embarquée [Ber02].

▪ **Spécification des besoins :** Souvent, cette phase est établi selon le résultat d'un compromis entre le fabriquant du produit (système embarqué) et les concepteurs. En tenant compte d'une part des considérations de conception de tels produits, d'autre part des contraintes requises par les systèmes embarqués. Un facteur commun pour les produits réussis est que l'équipe de conception partage une vision commune du produit conçu. Beaucoup de produits échouent parce qu'il n'y avait pas une articulation cohérente des objectifs du projet [Ber02].

▪ **Partitionnement :** Puisque la conception embarquée implique les composants hardware et software, alors quelqu'un doit décider quelle partie du problème sera résolue dans le hardware et laquelle dans le software. Ce choix s'appelle la "décision du partitionnement". Décider comment partitionner la conception des fonctionnalités qui sont représentées par le hardware et le software est une partie principale de création d'un système embarqué. Le choix du partitionnement a un impact significatif sur le coût du projet, le temps du développement et le risque.

▪ **Itération et Implémentation :** Cette phase du processus de développement représente un secteur brouillé entre l'implémentation et le partitionnement hardware/software dans lequel le chemin du hardware et celui du software divergent. La conception dans cette phase est délicate, bien que les principales parties à concevoir soient partitionnées entre les composants hardware et les composants software.

- **Conception conjointe hardware/software (Co-design) : Ces** dernières années, la conception conjointe software/hardware "**Co-design**" émerge avec un développement constant de la technologie de conception. D'abord, une analyse globale du système est nécessaire, ensuite, elle est décrite dans un langage de programmation spécifique (généralement les langages C, C++ ou Java) malgré que chaque partie est implémentée dans le software ou le hardware et la simulation est exigé pour le partitionnement HW/SW approprié. Si la partition ne satisfait pas les besoins, une répartition et une simulation peuvent être exécuté pour obtenir une meilleure partition [Den05].

- **Intégration software/hardware :** Le processus d'intégration software et hardware embarqués est un exercice (souvent compliqué) de débogage, afin de prendre en charge les problèmes d'ambiguïté qui peuvent surgir. Dans certain cas, la conception doit combiner le premier prototype hardware, l'application software, le code du driver, et le logiciel système d'exploitation pour pouvoir tester cette intégration. Dans le cas des systèmes temps réel, ce scénario est si peu probable. La nature des systèmes embarqués temps réel mène à un comportement fortement complexe et non déterministe qui ne peut pas être analysé dans cette phase. Souvent on fait recours aux techniques de simulation pour simuler le comportement du système [Ber02].

- **Test et Validation :** Cette phase est généralement reléguée à une équipe distincte (les testeurs). Les conditions de test et de fiabilité pour les systèmes embarqués sont beaucoup plus important que la grande majorité des applications du bureau surtout s'il s'agit des systèmes de performances critiques. En plus, tester un système embarqué est plus que s'assurer qu'il est performant, il doit aussi respecter les marges extrêmement serrées de conception en matière de coût et de contraintes de capacités optimales [Den05].

- **Maintenance et Amélioration :** On trouve souvent une documentation bien rédigée pour les produits les plus commercialement développés. Mais, il n'y a pas un développement d'outils spécifiques aux produits déjà en service. Cependant, la majorité des concepteurs de systèmes embarqués maintiennent et améliorent les produits existants, plutôt que concevoir les nouveaux produits. Ils utilisent dans ce cas la documentation existante, et le vieux produit pour le maintenir et l'améliorer [Val05].

3.1.2 Discussion

L'étude des différentes phases de ce cycle de développement nous a précisé qu'il suit une démarche informelle basée sur le prototypage et le débogage, et que certaines de ces phases sont brouillées, cas de la phase itération implémentation. Ils indiquent aussi qu'il y a une quantité considérable d'itération et de raffinement d'optimisation qui se produit dans les phases et entre les phases sans qu'on puisse détecter la méthode exacte de ces optimisations. Même si des développeurs arrivent à fabriquer certains produits en faisant recours à un prototypage et une simulation dans les phases de partitionnement pour confirmer le choix des composants, cette démarche reste très couteuse étant donné qu'elle n'est pas fondée sur des bases solides [Ben11]. La phase de test et validation occupe la majeure partie du coût de développement. Le fait de la déporter en dernier lieu rend ce coût exponentiellement élevé surtout s'il s'agit d'un débogage tardif dans le cycle de développement.

3.2 Développement niveau système

Les systèmes embarqués sont généralement dédiés à une utilisation spécifique prévue dès les premières phases du processus de développement. Donc, le concepteur doit avoir une vue globale et unifiée de l'architecture matérielle et des composants logiciels. La complexité du processus de conception de tels systèmes est déterminée par la sémantique et la syntaxe du langage de conception niveau système SLDL (System Level Design Language) adoptée pour véhiculer l'implémentation.

En général un SLDL exige deux attributs essentiels :

1) Il devrait supporter la modélisation à tous les niveaux d'abstraction.
2) Les modèles doivent être exécutables et simulables, de sorte que les fonctionnalités et les contraintes peuvent être validées.

L'objectif de la conception niveau système est de générer l'implémentation du système à partir de son comportement. Les approches de conception niveau système peuvent être classifiées d'après dans quatre groupes :

−La synthèse niveau système : cette approche concerne beaucoup plus les systèmes enfouis comme les systèmes en chip (SoC) ou cartes à puces.

−La conception basée plateforme : elle concerne essentiellement les systèmes embarqués autonomes comme par exemple les téléphones portables, les agendas électroniques, etc.

−La conception basée modèle.

−La conception basée architecture [Tal05].

3.2.1 Synthèse niveau système

Dans la synthèse niveau système, la conception commence par décrire le comportement du système dans un langage haut niveau comme C, C++, SDL. Cette description est ensuite transformée vers une description structurelle sous un format qualifié par transferts de registres RTL (Register Transfer Level). Le niveau RTL utilise des langages de description de matériels (Hardware Description Langages ou HDL) comme VHDL, Verilog ou HardwareC [Fra01]. Ces langages de description intègrent la capacité d'exprimer le temps, par une ou plusieurs horloges, et la notion de concurrence matérielle.

Les deux langages de conception niveau système ou SLDL les plus utilisés dans l'ingénierie des systèmes embarqués sont :SystemC et SpecC.

Quel que soit le langage utilisé, la conception dans ce cas doit essentiellement combiner la construction d'une architecture pour les calculs, les communications et le stockage du code et des données et la compilation de cette architecture, où la compilation sous-entend la synthèse d'un matériel dédié [Fra01].

3.2.2 Conception basée plateforme

Dans l'approche de la conception basée plateforme, le comportement du système est mappé sur une architecture prédéfinie du système, au lieu d'être généré à partir du comportement comme dans l'approche de synthèse niveau système. Cette approche est adoptée, dans l'industrie des systèmes embarqués, par Sun Microsystems et son langage Java. Sun Microsystems propose l'environnement de spécification Java pour les systèmes embarqués, où la machine JVM (Java Virtual Machine) est adaptée aux besoins de chaque plateforme par le développeur de systèmes [Ish05].

Ce processus de développement doit être suivi, comme le montre la figure 1.3, par plusieurs phases de compilation, simulation, débogage avant d'être déployé en ROM (ou Romization) [Cou06].

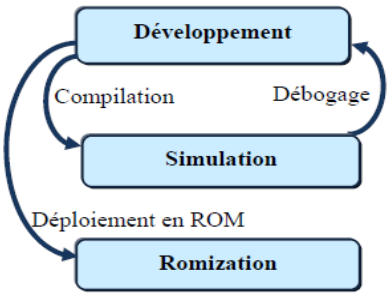


Figure 1.3: Cycle de vie de conception embarquée [Cou06].

La romization est le processus par lequel un système logiciel est pré-déployé par un outil spécifique le « romizer » utilisé pour instancier le programme initial d'un dispositif. Le romizer crée une image mémoire appropriée pour le dispositif cible qui contient le système avec les composants déployés dessus. L'image mémoire produite par le romizer est Complètement prête à fonctionner et mappable à la mémoire physique du dispositif [Cou05].

3.2.3 Conception basée modèle

L'approche basée modèle, qualifie aussi par le développement dirigé par les modèles MDD (Model Driven Development), place le modèle au centre du processus de développement des systèmes en général. Cette approche est beaucoup plus orientée vers la modélisation fournie par UML [OMG99]. Comme standard de l'OMG, UML offre un langage de modélisation sous forme d'un ensemble d'annotations graphiques et textuelles permettant de couvrir toutes les étapes du cycle de développement d'un système.

Dans le cas de développement des systèmes embarqués, le profil UML pour la modélisation et l'analyse des systèmes embarqués temps réel MARTE [OMG08] remplace et étend le profil UML pour la spécification de l'ordonnancement, la performance et le temps SPT [OMG05]. Cette extension concerne surtout de nouvelles annotations et des techniques d'analyse permettant la modélisation et l'analyse des aspects matériels et logiciels des systèmes embarqués temps réel. Le profil MARTE est structuré autour d'un noyau nommé TCRM (Time, Concurrence and Ressources Modeling), décrivant des constructions de modélisation pour le temps, les ressources, la concurrence, l'allocation et les propriétés non-fonctionnelles NFPs [Esp06].

3.2.4 Conception basée architecture

L'approche basée architecture ou l'adoption de la description architecturale joue un rôle important dans le développement de logiciels complexes et maintenables à moindre coût et avec respect des délais. Notons que l'architecture logicielle d'un système définit son organisation, à un niveau élevé d'abstraction, comme une collection de composants, de connecteurs, et de contraintes d'interactions avec leurs propriétés additionnelles définissant le comportement prévu. Les méthodes formelles s'imposent pour appréhender la complexité et la fiabilité de ces systèmes. Elles entraînent le développement de techniques d'analyse de systèmes et de génération automatique de tests à partir de leurs spécifications. Cela permet d'avoir une évaluation des comportements possibles d'un système au plus tôt, ce qui réduit à la fois le temps et les coûts de validation, et augmente la fiabilité du système [Fei04].

3.2.5 Discussion

La compilation classique d'un programme est la traduction de sa description écrite dans un premier langage ou langage source en un programme équivalent écrit dans un autre langage ou langage cible. La compilation, dans l'approche de synthèse niveau système, est une transformation du programme en préservant sa sémantique. Le travail du compilateur est de transformer le programme source en un programme optimisé en tenant compte des paramètres de chaque composant de l'architecture cible [Der02].

Le concept de machine virtuelle Java JVM permet la portabilité où le code Java est totalement portable à condition qu'il existe une machine virtuelle sur la cible. Il facilite ainsi le développement, parce que le langage Java permet de développer plus vite une application, et qu'une grande partie du code peut être développée et testée indépendamment de la cible. Cependant, Java présente également des limitations pour le développement d'applications critiques. La portabilité théorique du langage Java se heurte au masquage du matériel et des couches basses (par exemple : le RTOS) issues de fournisseurs différents.

MARTE définit les constructions d'un langage seulement mais ne couvre pas les aspects méthodologiques pour offrir une démarche à suivre pour la modélisation. Le profil souffre de l'absence d'une sémantique formelle des modèles UML décrivant le comportement du système. Il est aussi difficile de disposer d'une vue globale du système à modéliser, il faut faire appel à plusieurs types de diagrammes [Ben11].

4. Langages de modélisation des systèmes embarqués

L´émergence de langages de modélisation se fait de plus en plus dans le milieu industriel, dont les objectifs essentiels sont d'une part d'assister les formateurs et ingénieurs dans la conception de systèmes soumis a de fortes contraintes dans des représentations plus abstraites ; et d'autre part, fournir un formalisme de description des architectures afin de l'intégrer dans une démarche de génération automatique pour la vérification.

Nous citons dans ce qui suit quelques langages et profils de modélisation des systèmes embarqués.

4.1AADL (Architecture Analysis & Design Language)

AADL est un langage de description d'architecture permettant la spécification des systèmes embarqués. Il est un standard international publié par la SAE (Society of Automotive Engineers).

AADL repose principalement sur la notion de composants. Il permet de décrire des systèmes embarqués temps réel en assemblant des blocs développés séparément. Il définit une interface précise pour chaque composant et il sépare l'implantation interne du composant de la description de l'interface. Un composant représente une entité matérielle ou logicielle qui appartient au système. AADL permet aussi de décrire à la fois la partie matérielle et la partie logicielle d'un système. Pour cela, il utilise une syntaxe textuelle aussi bien qu'une représentation graphique.

AADL permet aussi de spécifier les reconfigurations dynamiques des systèmes via des machines à états composées par des modes et des transitions entre eux. Seuls les événements (event ports) définis au niveau des composants peuvent déclencher un changement de mode. La transition d'un mode à un autre nécessite des activations et/ou désactivations de certains sous-composants, connexions, etc [Chk10].

4.2 Profile MARTE (Modeling and Analysis of Real-Time Embedded systems)

MARTE est un profil UML standard de l'OMG, inspiré du profil SPT et assurant la modélisation et l'analyse des systèmes embarqués temps réel. Il permet la modélisation des systèmes embarqués à plusieurs niveaux d'abstraction grâce aux différents paquetages (sous profils) offerts. Ces paquetages permettent la séparation entre les deux parties, logicielle et matérielle, du système. En outre, MARTE aide à la spécification des propriétés non-fonctionnelles comme le temps et l'empreinte mémoire et les contraintes temporelles complexes. Pour cela, il offre le langage VSL (Value Spécification Language) sous la forme d'une extension du langage déclaratif OCL afin de supporter les contraintes comportementales. Il offre aussi une bibliothèque définissant de nouveaux types utiles pour la spécification des propriétés non-fonctionnelles.

Par ailleurs, MARTE permet de spécifier les reconfigurations comportementales des systèmes embarqués temps réel en utilisant une machine à états composée par des modes et des transitions entre eux. Un mode représente un état particulier du système tandis qu'une transition représente un événement qui déclenche une reconfiguration. Les transitions sont déclenchées par des événements. Le diagramme de collaboration est utilisé pour représenter un mode opérationnel tandis que le diagramme d'états-transitions est utilisé pour représenter les transitions entre les modes [Kri13].

4.3 SysML

La communauté de l'ingénierie système a voulu définir un langage commun de modélisation pour les systémiers, adapté à leur problématique, comme UML l'est devenu pour les informaticiens. Ce nouveau langage, nommé SysML, est fortement inspiré de la version 2 d'UML, mais ajoute la possibilité de représenter les exigences du système, les éléments non-logiciels (mécanique, hydraulique, capteur, . . .), les équations physiques, les flux continus (matière, énergie, etc.) et les allocations.

La version 1.0 du langage de modélisation SysML a été adoptée officiellement par l'OMG le 19 septembre 2007. Depuis, deux révisions mineures ont été publiées : SysML 1.1 en décembre 2008, et SysML 1.2 en juin 2010.

SysML s'articule autour de neuf types de diagrammes, que l'OMG a répartis en deux grands groupes

- **Quatre diagrammes comportementaux :**
  - Diagramme d'activité (montre l'enchainement des actions et décisions au sein d'une activité complexe).
  - Diagramme de séquence (montre la séquence verticale des messages passés entre blocs au sein d'une interaction).
  - Diagramme d'états (montre les différents états et transitions possibles des blocs dynamiques).
  - Diagramme de cas d'utilisation (montre les interactions fonctionnelles entre les acteurs et le système à l'étude).
  - Un diagramme transverse : le diagramme d'exigences (montre les exigences du système et leurs relations).
- **Quatre diagrammes structurels :**
  - Diagramme de définition de blocs (montre les briques de base statiques : blocs, compositions, associations, attributs, opérations, généralisations, etc.)
  - Diagramme de bloc interne (montre l'organisation interne d'un élément statique complexe, en termes de parties, ports, connecteurs, etc.)
  - Diagramme paramétrique (représente les contraintes du système, les équations qui le régissent)
  - Diagramme de packages (montre l'organisation logique du modèle et les relations entre packages)

SysML innove par rapport à UML en proposant un diagramme d'exigences qui permet de modéliser les exigences système et surtout de les relier ensuite aux éléments

structurels ou dynamiques de la modélisation, ainsi qu'à des exigences de niveau sous-système ou équipement [Kor13].

4.4 UML

UML est un langage de modélisation défini par l'Object Management Group. Le but d'UML est de pouvoir représenter le comportement, l'architecture, les structures de données et la structure elle-même d'un système embarqué.

Ces différentes représentations s'appuient sur différents diagrammes, chacun proposant de représenter le système suivant un point de vue particulier. UML propose treize diagrammes différents répartis en trois catégories. La première catégorie regroupe les diagrammes structurels avec le diagramme de classe, le diagramme d'objets, le diagramme de composant, le diagramme de structure composite, le diagramme de package et le diagramme de déploiement ; la seconde catégorie concerne les diagrammes comportementaux avec le diagramme d'activité le diagramme de machine à états et le diagramme de cas d'utilisation ; la dernière catégorie rassemble les diagrammes d'interaction avec le diagramme de séquence, le diagramme de communication, le diagramme de temps et le diagramme de collaboration [Pir14] .

Si UML permet une grande expressivité, il ne transporte pas en lui-même une sémantique précise pour décrire les architectures. Pour modéliser un système embarqué, UML est peut-être raffinée par la définition de Profil.

Le profil UML d'ordonnancement de performance et de temps, UML Profile for Schedulability, Performance, and Time, SPT a été proposé pour les concepteurs et développeurs des systèmes embarqués temps réel [Men15].

4.5 Spécification PEARL et le profil UML-RT

La spécification PEARL a été proposée pour assurer la programmation des applications temps réel automatiques. PEARL est étendue par des constructions permettant la description des configurations matérielles et logicielles. Elle définit aussi le mapping des modules logiciels sur des composants matériels. Elle introduit des attributs définissant des informations sur le temps permettant de spécifier des systèmes temps réel. Elle permet de spécifier des communications avec leurs caractéristiques ainsi que des architectures tolérantes aux pannes. De plus, PEARL permet la modélisation graphique. Elle assure aussi la spécification des conditions et des méthodes assurant des reconfigurations dynamiques en énumérant toutes les configurations possibles.

En se basant sur les concepts introduits dans la spécification PEARL, un patron de reconfiguration UML pour des systèmes embarqués a été proposé. Ce modèle de reconfiguration est décrit par un profil UML-RT permettant d'assurer la modélisation et la gestion des reconfigurations des architectures logicielles et matérielles.

UML-RT utilise le diagramme de séquence pour modéliser les différentes reconfigurations et plus précisément les transactions d'une configuration à une autre [Kri13].

4.6 Discussion

Les deux standards AADL et MARTE permettent la spécification des systèmes embarqués temps réels avec leurs propriétés non-fonctionnelles et leurs contraintes temporelles. Ils décrivent les reconfigurations dynamiques des systèmes par des machines à états composées par des modes et des transitions entre eux.

Par contre et contrairement à AADL, le profil MARTE traite seulement les reconfigurations comportementales des systèmes embarqués temps réel par la modification des caractéristiques internes des composants. De plus, AADL spécifie les systèmes embarqués à un niveau concret (threads, processeurs, etc) lié à une plate-forme spécifique [Kri13].

De même, la spécification PEARL permet aussi la description des systèmes embarqués temps réel. Elle assure la transaction d'une configuration à une autre par un ensemble d'actions de reconfiguration. Ces transactions sont déclenchées par des événements.

La spécification PEARL prend en considération la tolérance aux pannes qui nécessite une architecture reconfigurable.

UML-RT est un profil qui définit les concepts de la spécification PEARL. Cependant, la spécification PEARL reste très limitée par rapport à AADL et MARTE pour la spécification des systèmes embarqués temps réel. Elle ne permet pas de spécifier les expressions temporelles complexes.

Notre objectif est donc de proposer une approche dirigée par les modèles étendant le langage UML par des concepts temporels, permettant de spécifier le comportement dynamique d’un système embarqué dans un espace de temps. Pour cela nous avons utilisé les diagrammes statecharts et de collaboration temps réels.

5. Vérification des systèmes embarqués

La vérification cherche à assurer que le système construit effectue correctement les fonctions spécifiées. On entend par techniques de vérification des outils qui permettent de vérifier, dans une certaine mesure, que le système satisfait une spécification. Le but est

d'augmenter la confiance que l'on a sur le système développé, ce qui se traduit en pratique par différentes approches.

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

Deux approches de base ont été alors utilisées pour la vérification des systèmes. Une première technique de vérification consiste en l'élaboration d'un prototype du système, qui est testé dans son environnement. Il s'agit là de la simulation du comportement du système, le comportement simulé est alors comparé au comportement attendu. S'il y a adéquation, le système est vérifié. Malheureusement, la simulation ne peut évaluer tous les comportements possibles d'un système complexe, du fait de l'explosion combinatoire du nombre de comportements possibles de ces systèmes. De ce fait, la simulation ne peut vérifier que le comportement du système sur un jeu de tests extrêmement réduit au regard de tous les comportements possibles du système [Hoc13].

D'autres voies de vérification ont été étudiées : elles consistent à remplacer la vérification expérimentale du système par une preuve de sa correction. On cherche à montrer formellement que le système est en adéquation avec sa spécification, ou qu'il vérifie un ensemble de propriétés décrivant partiellement la spécification.

5.1 Vérification par simulation

Ce type de vérification est basé sur un modèle HDL du système étudié. Ce modèle décrit le comportement global ou partiel du système. Il est stimulé par une série de vecteurs de test et les résultats obtenus (à ses sorties) sont comparés avec des valeurs prévues dans les mêmes conditions de la manipulation.

L'avantage majeur de la vérification par simulation apparaît dans le fait qu'elle ne nécessite pas un prototype matériel et qu'elle est souvent implémentée par des outils de conception assistée par ordinateur. Cependant, cette méthodologie est gourmande en termes de temps. En outre, elle nécessite une phase de génération de stimuli, les appliquer à l'entrée du système et ensuite entamer la tâche de vérification [Por06].

La simulation est un type de modélisation d'un système qui peut être mise en œuvre sur ordinateur. Elle permet de définir la manière dont ce système évolue en fonction du temps, c'est-à-dire le comportement du système, et donc elle peut être vue comme un processus de modélisation dans lequel une réalité dynamique est imitée grâce à des actions sur ordinateur.

La simulation en terme général ne permet pas d'établir une preuve de correction du comportement au sens mathématique ou formel, mais elle facilite la compréhension des

aspects sélectionnés du comportement du modèle, et aussi des relations entre les composants du système modélisé.

L'exécution d'une simulation peut être réitérée plusieurs fois pour gagner en confiance au comportement correct du modèle, en examinant différents modes de travail, différentes valeurs des données d'entrée, ou bien différents états initiaux.

Le comportement réel ou proposé du système est modélisé par le biais d'une description (un modèle) qui peut être exprimée sous la forme de description structurelle ou comportementale, en utilisant des langages de description de matériel (HDL).

La figure suivante présente une vue globale de l'environnement de la simulation dans lequel la description d'un système, avec les vecteurs de stimuli (test), est traité par le simulateur qui offre un moyen de déboguer le code de la description ainsi que de visualiser et de stocker les résultats de la simulation de ces vecteurs de stimuli [Hoc13].

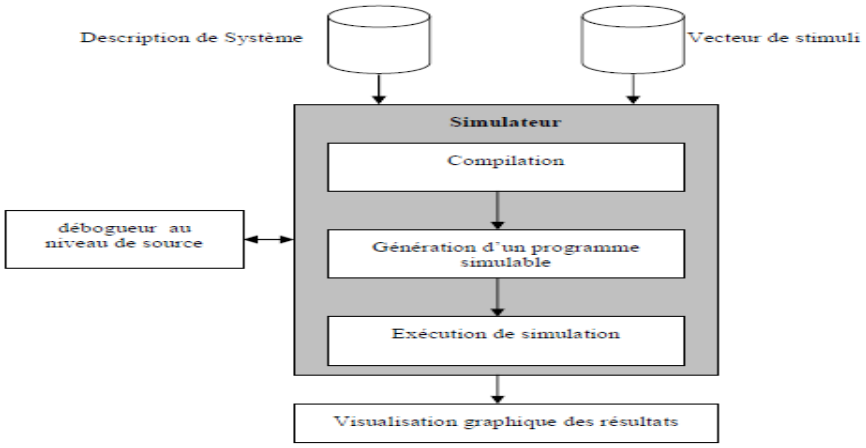


Figure 1.4: Environnement de simulation [Hoc13].

5.2 Vérification formelle

La vérification formelle consiste à démontrer qu'un système va se comporter comme attendu. Cette démonstration est exhaustive par nature et détecte les problèmes même pour des configurations auxquelles le concepteur n'aurait pas pensé [Bou12].

La vérification formelle peut, soit comparer les données informatiques qui vont permettre la réalisation d'un futur système avec sa spécification, soit vérifier que ce futur système répond à certaines contraintes caractérisant son bon fonctionnement.

La vérification formelle, consiste à suggérer l'utilisation des outils de synthèse dans lesquels un futur système est automatiquement construit à partir de sa spécification. Cette approche alléchante n'est pas autant à l'abri des erreurs qu'il n'y paraît à première vue.

Une implémentation représente le modèle qui devrait être vérifié afin de savoir s'il correspond ou pas à sa description initiale (spécification). La spécification représente les propriétés auxquelles se réfère le processus de vérification pour conclure de la conformité d'un système.

Un certain nombre de techniques de descriptions formelles sont apparues. Elles sont basées sur des langages de description formelle et disposent d'outils de vérifications automatiques de ces descriptions. Un langage est formel si sa sémantique est précise et non ambiguë [Hoc13].

5.2.1Techniques de vérification formelle

On entend par techniques de vérification des outils qui permettent de vérifier, dans une certaine mesure, que le système satisfait une spécification. Le but est d'augmenter la confiance que l'on a sur le système développé, ce qui se traduit en pratique par différentes approches.

Parmi les méthodes de vérification formelle, nous pouvons distinguer les trois approches suivantes :

- Les vérifications basées sur les preuves de théorèmes.
- Les vérifications basées sur les équivalences.
- Les vérifications basées sur le model checking.

D'autres classifications sont évidemment proposées dans la littérature.

Pour mettre en place ces approches, on utilise souvent des langages de spécifications formelles tel que :

1. **Les langages fonctionnels** : sont basés sur l'utilisation des principes de programmation fonctionnelle pour décrire d'une manière formelle les systèmes hardware. Ces langages ont l'avantage sur d'autres de permette des descriptions à différents niveaux d'abstraction.

2. **Le langage OBJ3 :** est un langage de spécification algébrique. L'utilisation de ce langage pour la description du hardware a l'avantage de permettre des descriptions hiérarchiques et abstraites.

3. **Le langage LOTOS** : est basé sur l'utilisation des types abstraits de données et les processus pour décrire un comportement d'un contrôleur. Les types abstraits de données sont les plus adaptés pour la modélisation de simples circuits. Pour décrire des systèmes plus complexes, il est préférable d'utiliser des processus. Dans ce cas, la connexion entre

composants matériels peut être décrite par l'utilisation des différents opérateurs de synchronisation de ce langage.

4. **Le langage PROMELA** : est un langage de spécification de systèmes asynchrones. Ce qui veut dire que ce langage permet la description de systèmes concurrents, comme les protocoles de communication. Il autorise la création dynamique de processus. La communication entre ces différents processus peut se faire en partageant les variables globales ou alors en utilisant des canaux de communication. On peut ainsi simuler des communications synchrones ou asynchrones.

5. Les langages de programmation des systèmes réactifs : cette classe de langages est une nouvelle famille de langages synchrones pour la spécification et la vérification des systèmes réactifs, tel qu'ESTEREL, LUSTRE, SIGNAL, SML, STATCHARTS...

6. **Le langage VHDL** : est un langage de description du matériel basé sur une sémantique formelle. Des programmes écrits en VHDL ont été traduits (compilés) en un modèle formel en se basant sur sa sémantique de simulation pour pouvoir appliquer des méthodes de vérification formelles [Hoc13].

7. **Le langage Maude** : Maude est un langage formel de spécification et de programmation déclarative basé sur une théorie mathématique de la logique de réécriture. Maude est un langage simple expressif et performant et il est considéré parmi les meilleurs langages dans le domaine de spécification algébrique et la modélisation des systèmes concurrents [Cla00].

5.2.1.1 Les vérifications basées sur les preuves de théorèmes

Cette approche est basée sur les preuves de théorèmes, a pour principe de poser un ensemble d'axiomes, souvent donnés par le concepteur, puis à prouver un ensemble d'assertions déterminant ainsi la conformité du système. Cette preuve est faite plus ou moins manuellement.

L'aide apportée par les outils tels que les démonstrateurs de théorèmes n'évitent pas totalement la nécessité de l'intervention humaine. Ce type de vérification est rarement employé, il est utilisé essentiellement dans le domaine des spécifications algébriques et logiques [Bou12].

Dans le domaine de la vérification des systèmes embarqués, citons les travaux de [Col09], qui a réalisé une des premières preuves significatives en vérifiant le microprocesseur FM8501. Il permet de postuler le problème de vérification à différents

niveaux d'abstraction. Cette approche a connu des succès significatifs dans la vérification de conception des processeurs. Ces démonstrateurs de théorèmes sont utilisés aussi pour la vérification des systèmes décrits en VHDL [Hoc13].

Cependant, l'inconvénient majeur de cette approche vient du fait que le pouvoir d'expression de la logique du premier ordre et surtout d'ordre supérieur, ne permet pas d'automatiser complètement les preuves. De nos jours, la majorité des systèmes de theorem-proving sont semi-automatiques et exigent plus d'efforts du côté de l'utilisateur afin de développer des spécifications pour chaque composant et à superviser le processus d'inférence.

5.2.1.2 Les vérifications basées sur les équivalences

Cette approche consiste à vérifier l'équivalence (le plus souvent par rapport à une relation de bisimulation) entre le modèle de description de l'implémentation et une spécification qui décrit ce que l'on attend de cette implémentation, les deux utilisant le même formalisme de description. Si les deux modèles sont équivalents cela prouve que l'implémentation est conforme à la conception [Bou12].

5.2.1.3 Les vérifications basées sur le model checking

C'est l'approche la plus utilisée, elle est basée sur les modèles, permet une vérification simple et efficace et elle est complètement automatisables. La vérification basée sur les modèles ou Model checking est surtout applicable pour les systèmes ayant un espace d'états fini.

Les algorithmes de vérification dans cette méthode utilisent l'ensemble des états que le système peut atteindre pour prouver la satisfaction ou la non-satisfaction des propriétés. Cependant, le problème majeur de ce type de vérification est la taille souvent excessive de l'espace d'états. En effet, elle peut être exponentielle par rapport à la taille de la description du système. Une des causes principales de cette explosion combinatoire du nombre d'états et le fait que l'exploration est réalisée en prenant en compte tous les entrelacements possibles d'évènements concurrents.

Pour pallier à ce problème de l'explosion combinatoire, différentes solutions dont l'objectif et de réduire la taille de l'espace d'états, ont été proposées dans la littérature.

Les techniques de vérification de modèles, reposent sur :

- La construction d'un graphe d'états représentant les comportements du système.
- L'expression de la propriété à vérifier dans une logique temporelle.

- Des algorithmes qui permettent de vérifier que le modèle satisfait sa spécification.

Le modèle est donné par une structure de Kripke qui décrit quels sont les états possibles du système, ses évolutions (transitions entre états) et les propriétés atomiques que vérifie chaque état. Les propriétés plus complexes sont données dans des logiques temporelles qui permettent d'exprimer des relations de causalité entre les états. Le model checking couvre l'ensemble de tous les états du système, ce qui en fait une technique de vérification formelle : toutes les exécutions du système sont considérées.

De plus, mise à part la modélisation du système et l'expression de sa spécification, le model checking est une technique entièrement automatique ce qui la rend très utilisée en industrie aujourd'hui [Bou12].

5.2.2 Apports des méthodes formelles

Les méthodes formelles sont basées sur des techniques mathématiques pour la description de propriétés dans un processus de conception de systèmes (matériels/logiciels).

Les méthodes formelles servent pour la spécification, le développement et la vérification des systèmes d'une façon systématique. Il y a plusieurs raisons pour justifier l'adoption des méthodes formelles.

- ✓ **Complétude de la spécification**

  Dans un processus de conception, l'utilisation des méthodes formelles a pour but d'assurer que la spécification est conforme à tout ce qui est décrit dans le cahier des charges du système à concevoir. Il est pratiquement délicat d'assurer qu'une spécification non-formelle contienne tout ce qui a été prévu au départ.

- ✓ **Réduction des taux d'erreurs au niveau de la spécification**

  Dans ce contexte, l'utilisation des méthodes formelles a pour but d'obtenir une spécification qui contient moins d'erreurs.

  Contrairement à la raison précédente, il ne s'agit pas ici d'inclure ou d'exclure des comportements dans la spécification initiale mais plutôt d'être sûr que ce qui est spécifié est correct. Car tout type d'erreur dans la spécification est susceptible d'apparaître dans la phase d'implémentation et que les conséquences impliquées en matière de coût sont considérables.

- ✓ **Réduction des taux d'erreurs au niveau de l'implémentation**

L'implémentation peut contenir des problèmes, et une bonne partie du développement est consacrée au test et au débogage de cette implémentation. Donc, beaucoup de tests et de simulations sont indispensables, les vecteurs de stimuli choisis ne peuvent couvrir l'ensemble des états possibles atteignables du système. Les méthodes formelles représentent un complément important pour éliminer plus de problèmes fonctionnels et logiques dans une implémentation.

- ✓ **Réduction du coût de développement**

Les méthodes formelles sont coûteuses à utiliser et leur utilisation demande beaucoup de temps et d'efforts. Bien que l'utilisation de méthodes formelles prenne beaucoup de temps et coûte de l'argent, son rendement peut être énorme s'il aide à éviter seulement quelques erreurs critiques. En utilisant des méthodes formelles, la phase de spécification du système prend typiquement plus de temps, par contre, avec une spécification plus claire et plus correcte, l'implémentation, l'intégration et la phase de tests peuvent être effectuées beaucoup plus rapidement [Hoc13].

5.3 Discussion

Les approches de vérification non-formelle (simulation et test), ne sont pas très efficaces pour les systèmes complexes. Elles sont très coûteuses, trop lentes et non exhaustives, voir même impossible de tester ou simuler toutes les entrées possibles d'un système dont le nombre d'entrées est excessivement grand [Bou12].

Les besoins de produire de plus en plus vite et à moindre coût sont incompatibles avec des systèmes partiellement vérifiés. La simulation et la vérification par équivalence ne garantissent pas une conception sans erreurs, c'est pourquoi les méthodes formelles sont intégrées au flot de conception.

La vérification formelle d'un système cherche à caractériser le comportement de ce dernier par une analyse mathématique, contrairement aux techniques de simulation qui représentent une analyse expérimentale du système. La vérification formelle cherche à démontrer l'équivalence ou l'inclusion de deux modèles formels, l'un représentant une spécification et l'autre une implémentation du système, ou bien les deux représentant deux implémentations différentes d'un même système.

Dans notre travail, nous avons utilisé une méthode formelle de vérification de modèles basée sur le langage Maude. Afin de vérifier qu'un système embarqué satisfait certaines propriétés temporelles exprimées dans la logique temporelle d'un système embarqué.

6. Conclusion

Dans ce chapitre, nous avons présenté un état de l'art sur les systèmes embarqués ainsi que les différentes approches de modélisation et de vérification de ces systèmes. Pour ce faire, nous avons d'abord commencé par les approches de modélisation, à savoir les profils comme MARTE et UML-RT et les langages comme AADL et UML. Chaque approche de modélisation a sa stratégie et ses caractéristiques. Dans notre travail nous avons utilisé les deux diagrammes comportementaux d'UML : les diagrammes d'état-transition et de collaboration temps réel.

Ensuite, nous avons distingué deux approches de vérification des systèmes embarqués : la vérification par simulation et la vérification formelle. Nous avons tout d'abord montré comment la simulation est incapable de vérifier les systèmes complexes, vu son incapacité de couverture exhaustive de tous les cas possibles. Les méthodes formelles s'appuient sur le raisonnement mathématique pour prouver des propriétés de systèmes.

L'approche de vérification que nous proposons dans ce travail repose sur l'approche formelle en utilisant le langage Maude pour la spécification et le Model checking pour la vérification.

Chapitre 02 :

Ingénierie dirigée par les modèles

1. Introduction

L'ingénierie logicielle s'oriente actuellement vers l'ingénierie des modèles, après l'approche objet, où chaque artefact logiciel est considéré comme un modèle.

L'ingénierie dirigée par les modèles IDM (*Model Driven Engineering*) présente une approche de développement et devenue très populaire dans l'ingénierie logicielle, qui se concentre sur la création et l'exploitation de modèles abstraits. En d'autres termes, c'est une représentation abstraite des connaissances et des activités qui régissent un domaine applicatif particulier facilitant la compréhension du système modélisé [Ait12].

L'approche IDM vise à générer tout ou une partie de l'application à partir des modèles. Ce qui permet en soi d'augmenter la productivité tout en optimisant la compatibilité entre les systèmes grâce à la réutilisation à grande échelle des modèles normalisés.

Plusieurs approches IDM ont vu le jour. La plus populaire de ces approches est l'architecture dirigée par les modèles de l'OMG. Celle-ci a été étendue et ne se limite pas aux modèles (exp UML) mais plutôt met au centre de la démarche en génie logiciel les modèles et non pas les programmes. Il s'agit de l'architecture orientée modèles (MDA).

MDA joue un rôle essentiel dans l'introduction des méthodes formelles dans les activités de développement des systèmes. MDA repose, d'une part, sur la définition de langages par le biais de la méta-modélisation qui permet d'exprimer les différents aspects d'un langage. D'autre part, sur l'utilisation de transformations de modèles pour combiner ces aspects, échanger des informations entre eux afin de générer des modèles dans un langage formel, pour permettre la validation et la vérification par différentes techniques formelles.

Dans ce chapitre, nous présentons les concepts et les principes de MDA, ainsi qu'un panorama sur les méthodes formelles. Ensuite nous abordons l'utilisation et l'intégration de ces méthodes formelles dans l'approche MDA afin d'augmenter la fiabilité et de garantir l'absence d'erreurs de conception.

2. Ingénierie dirigée par les modèles (IDM)

2.1 Présentation

L'Ingénierie Dirigée par les Modèles IDM est une approche de développement mettant à disposition des outils, des concepts et des langages afin de simplifier et de mieux maitriser le processus de développement de systèmes qui ne cessent de croitre en

complexité. D'autre part, elle permet aussi d'augmenter la productivité, la qualité, la réutilisabilité et l'évolution de ces systèmes.

L'IDM est un domaine de recherche en pleine expansion aussi bien dans le monde académique que dans le monde industriel qui considère les modèles comme les éléments de base tout au long du processus de développement [Béz04].

L'IDM raisonne entièrement à un haut niveau d'abstraction et non plus à celui des langages de programmation classiques. Une application sera alors générée en tout ou en partie, automatiquement ou semi-automatiquement à partir de modèles, en utilisant notamment des transformations successives de ces modèles [ker11].

2.2 Notions de Modèle et Méta-modèle

Un modèle est une abstraction et une simplification d'un système qu'il représente. Il offre donc une vision schématique d'un certains nombres d'éléments que l'on décrire sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions que l'on se pose sur lui.

Il faut noter qu'il reste toutefois difficile de répondre à la question "Qu'est-ce qu'un bon modèle ?". Néanmoins un modèle doit être suffisant et nécessaire pour permettre de répondre à certaines questions du système qu'il représente, exactement de la même façon que le système aurait répondue lui-même. Le modèle doit se substituer au système pour permettre d'analyser de manière plus abstraite certaines de ses propriétés [ker11].

Pour qu'un modèle soit productif, il doit pouvoir être manipulé par une machine. Le langage dans lequel ce modèle est exprimé doit donc être clairement défini. De manière naturelle, la définition d'un langage de modélisation a pris la forme d'un modèle, appelé Méta modèle.

Un méta modèle définit la structure que doit avoir tout modèle conforme à ce méta modèle. Autrement dit, tout modèle doit respecter la structure définie par son méta modèle. Par exemple, le méta modèle UML définit que les modèles UML contiennent des packages, leurs packages des classes, leurs classes des attributs et des opérations, etc. La figure 3.1 illustre la relation entre un méta modèle et l'ensemble des modèles qu'il structure [Bla05].

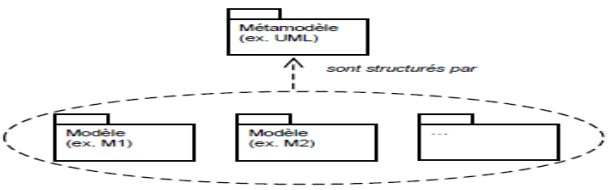


Figure 2.1 : La relation entre un méta modèle et l'ensemble des modèles [Bla05].

Le Méta-modèle à son tour est exprimé dans un langage de méta-modélisation spécifié par le Méta-Méta-modèle. Le langage utilisé au niveau du méta-méta-modèle doit être suffisamment puissant pour spécifier sa propre syntaxe abstraite et ce niveau d'abstraction demeure largement suffisant (méta-circulaire). Chaque élément du modèle est une *instance* d'un élément du méta modèle.

Un modèle est dit *conforme* à un méta-modèle et constitue une représentation d'un système existant ou imaginaire. La relation entre un méta-méta-modèle et un méta-modèle est analogue à la relation entre un méta-modèle et un modèle [ker11].

2.3 Transformation de Modèles

La notion de transformation de modèles constitue l'élément central de la démarche IDM. En effet, cette notion porte sur l'automatisation de l'opération de transformation pendant le cycle de développement qui peut avoir des sémantiques différentes en fonction des utilisations : raffinement, optimisation, génération de code, etc.

La transformation de modèles est une opération qui consiste à générer un ou plusieurs modèles cibles conformément à leur méta-modèle à partir d'un ou de plusieurs modèles sources conformément à leur méta-modèle. Elle est qualifiée d'endogène si les modèles sources et cibles sont conformes au même méta-modèle (source et cible sont dans le même espace technologique), sinon elle est dite exogène et elle se fait entre deux méta-modèles différents (source et cible sont dans deux espaces technologiques différents).

Dans la littérature, on peut distinguer trois types de transformations :

- **Les transformations verticales** : La source et la cible d'une transformation verticale sont définies à différents niveaux d'abstraction. Une transformation qui baisse le niveau d'abstraction est appelée un raffinement. Une transformation qui élève le niveau d'abstraction est appelée une abstraction.
- **Les transformations horizontales** : Une transformation horizontale modifie la représentation source tout en conservant le même niveau d'abstraction. La modification peut être l'ajout, la modification, la suppression ou la restructuration d'informations.
- **Les transformations obliques** : Une transformation oblique combine une transformation horizontale et une verticale. Ce type de transformation est notamment utilisé par les compilateurs, qui effectuent des optimisations du code source avant de générer le code exécutable [ker11].

De manière orthogonale à cette catégorisation, selon [Cza03], il existe deux grandes classes de transformation de modèles :

- Les transformations de type **Modèle vers code** qui sont aujourd'hui relativement matures
- Les transformations de type **modèle vers modèle** qui sont moins maîtrisées.

La transformation se fait par l'intermédiaire d'un ensemble de règles de transformations décrivant la correspondance entre les entités du modèle source et celles du modèle cible. La façon d'exprimer les règles de transformation peut être déclarative, impérative ou hybride. Il est à la charge de l'utilisateur de définir le langage de transformation qui répond le mieux à ses besoins et à ses compétences.

Dans la spécification déclarative, les règles décrivent ce qu'on devrait avoir à l'issue d'un certains nombres d'éléments de modèle source.

Par opposition à la spécification déclarative, la spécification impérative permet de décrire comment le résultat devrait être obtenu en imposant une suite d'actions que la machine doit effectuer.

Enfin, la spécification hybride regroupe à la fois la spécification déclarative et la spécification impérative.

En réalité, la transformation se situe entre les méta-modèles source et cible. La transformation des entités du modèle source se fait en deux étapes :

- La première étape permet d'identifier les correspondances entre les concepts des modèles source et cible au niveau de leurs méta-modèles, ce qui induit l'existence d'une fonction de transformation applicable à toutes les instances du méta-modèle source.
- La seconde étape consiste à appliquer la transformation du modèle source afin de générer automatiquement le modèle cible par un programme appelé moteur de transformation ou d'exécution [ker11].

Une présentation générale des principaux concepts impliqués dans la transformation de modèles est illustrée dans la Figure 2.2.

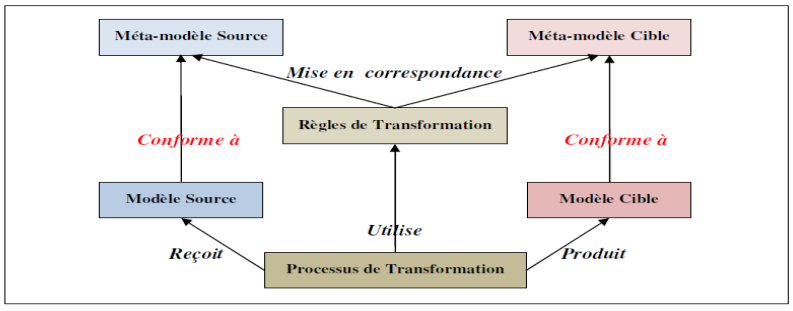


Figure 2.2 : Concepts de base de la transformation de modèles [ker11]**.**

2.4 Manipulations des modèles

Les manipulations des modèles dans le cadre de l'IDM sont nombreuses et apportent chacune un ensemble de problématiques spécifiques. Nous présentons ici une liste non exhaustive de ces manipulations et donnons un aperçu des problématiques qui leur sont liées.

2.4.1 Réalisation de modèles

La réalisation des modèles nécessite non seulement l'expertise technique pour comprendre ou concevoir la partie du système concerné mais aussi une bonne connaissance du langage de modélisation utilisé. Dans le cas de systèmes complexes, les modèles deviennent également complexes et surtout de taille importante. La qualité de l'outillage devient alors essentielle car ce sont les outils qui permettent de mieux visualiser le modèle, de s'affranchir de certains détails ou encore de vérifier automatiquement la syntaxe. Les problématiques liées à l'outillage sont variées : elles concernent la visualisation des modèles, les méthodes d'assistance, le support des langages de modélisation, etc … .

2.4.2 Stockage de modèles

L'informatisation des modèles pose le problème de la gestion de leur persistance puis de leur accès par les utilisateurs. Les problématiques liées à cette activité concernent, par exemple, les formats de stockage, l'organisation du stockage ainsi que la gestion des méta-données concernant les modèles.

2.4.3 Echange de modèles

L'échange de modèles entre différents acteurs d'un projet est une vraie nécessité car elle conditionne la bonne communication entre ces acteurs. Des problèmes de compréhension de modèles entre différents acteurs peuvent avoir des conséquences catastrophiques sur le système implémenté. L'échange de modèles pose notamment des problèmes de format (sérialisation, transport, etc.), de traduction et d'interprétation de la sémantique pour l'interopérabilité (notamment entre les outils de modélisation).

2.4.4 Exécution de modèles

L'exécution de modèles comprend un éventail de tâches différentes allant de la simulation à l'exécution en temps réel en passant par l'exécution symbolique ou la génération de code. Dans ce cadre, les problèmes majeurs concernent l'exécutabilité de la

sémantique des langages de modélisation utilisés. En effet, cette propriété d'exécutabilité conditionne la possibilité de pouvoir calculer, à partir du modèle, un comportement du système, ou même tous les comportements possibles de ce système [Cza03].

2.4.5 Vérification de modèles

La vérification d'un modèle consiste à vérifier les propriétés propres de ce modèle par rapport à ce que l'on attend de lui (correction syntaxique, sémantique, etc.). La vérification de modèle recouvre différents aspects allant de la vérification de la syntaxe à la vérification de la sémantique. Les problématiques les plus complexes concernent bien sûr la vérification de la sémantique. Différentes techniques de vérification existent, avec leurs problématiques particulières : la preuve, le test ou encore le model checking. Les techniques de preuve s'appuient sur l'utilisation de représentations formelles (à base de logique, d'automates, de Réseaux de Petri par exemple) du système. Les techniques de model checking visent à analyser le comportement spécifié par le modèle de manière à vérifier des propriétés comme la sûreté, l'atteignabilité ou la vivacité et l'équité. Les problématiques dans ce contexte sont liées notamment à l'identification avec exhaustivité des états possibles du système (explosion des espaces d'état). Enfin, le test est utilisé en complément du model checking, notamment dans le cas de systèmes pour lesquels le model checking est particulièrement inefficace (lorsque les systèmes sont trop complexes par exemple) [Ait12].

2.4.6 Validation

La validation permet de vérifier que le système implémenté répond aux besoins initiaux qui ont amené à sa conception. Certaines techniques comme le test peuvent être utilisées à la fois pour la vérification et pour la validation. Dans ce cadre, les modèles permettent notamment de générer des scénarios et des vecteurs de test de façon automatique. Par ailleurs, afin de minimiser les risques d'erreur de conception le plus en amont possible, les modèles peuvent être validés les uns par rapport aux autres au cours du cycle de développement. Il s'agit alors, par exemple, de vérifier que certaines propriétés sont préservées d'un modèle à un autre [Fav06].

2.4.7 Gestion de l'évolution des modèles

Les modèles évoluent au cours du cycle de développement du système. Ils peuvent être modifiés dans le cadre de correction d'erreurs ou d'ajout de fonctionnalités

par exemple. Les problématiques dans ce contexte concernent en particulier la répercussion automatique des modifications sur les différents modèles impliqués [ker11].

2.5 Les approches de l'Ingénierie dirigée par les modèles

2.5.1 L'Architecture Dirigée par les Modèles (MDA)

L'architecture MDA est une approche de développement proposée et soutenue par l'OMG (*Object Management Group*) [OMG]. MDA est un moyen de structuration et de gestion de l'architecture logicielle d'une organisation où les modèles sont utilisés à grande échelle. En effet, elle permet de définir une structuration des lignes directives pour les spécifications exprimées en tant que modèles.

Elle est supportée par des outils automatisés et des services pour, à la fois, définir les modèles et faciliter les transformations entre les différents types de modèles. En d'autres termes, MDA permet de séparer la spécification des fonctionnalités du système étudié de la spécification de son application sur sa plateforme d'implémentation. Elle offre la possibilité de concevoir des modèles indépendants de toutes plateformes ou environnements d'implémentation. L'approche MDA fournit un moyen au travers des modèles pour orienter la compréhension, la conception, le déploiement, l'exploitation et la maintenance du système étudié.

Il s'agit donc d'une forme d'ingénierie générative, le code source de l'application n'est plus considéré comme L'élément central d'un logiciel, mais comme un élément dérivé d'éléments de modélisation [ker11].

Dans cette optique, des outils permettant de construire et d'exploiter ces modèles ont été développés. Ces outils sont construits autour du concept de *Méta-modèle* qui permet de définir un langage de modélisation particulier à un domaine ou d'intégrer plus facilement plusieurs types de modèles. Ils intègrent aussi le concept de *transformation de modèles* pour pouvoir, par exemple, relier un modèle à une plate forme technologique spécifique ou générer du code [Bla05].

2.5.2 Standards de l'OMG

L'approche MDA a le mérite d'être élevée au rang de standards de l'OMG (voir la Figure 2.3), dont notamment UML, OCL, MOF, XMI et CWM.

- **UML** (Unified Modeling Language) Un langage visuel permettant de modéliser des systèmes à l'aide de diagrammes et de textes. Il permet aussi de décrire des architectures, des solutions ou des points de vue.

- **OCL** (Object ConstraintLanguage) Un ajout à UML, lui apportant la capacité de formaliser l'expression des contraintes. Il est désormais intégré à UML.
- **MOF** (Meta-Object Facility) Un ensemble d'interfaces standards permettant de définir et de modifier des méta-modèles et leurs modèles correspondants. Le MOF est un standard de méta modélisation pour définir la syntaxe et la sémantique d'un langage de modélisation, il a été créé par l'OMG afin de définir la notation UML.
- **XMI** *(*XML Metadata Interchange*)* Un standard d'échange de métadonnées.
- **CWM** *(*Common Warehouse Metamodel*)* Une interface servant à faciliter les échanges de métadonnées entre outils, plates-formes et bibliothèques de métadonnées dans un environnement hétérogène. Il est basé sur UML, MOF et XMI [ker11].

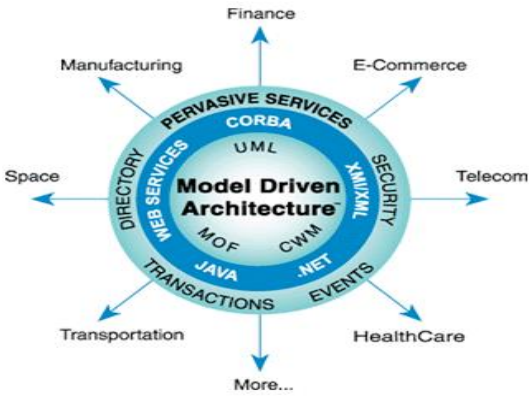


Figure 2.3: Les standards de l'Architecture Dirigée par les Modèles [And04].

La troisième couche contient les services qui permettent de gérer les évènements, la sécurité, les répertoires et les transactions. La dernière couche propose des frameworks spécifiques au domaine d'application (Finance, Télécommunication, Transport, Espace, médecine, commerce électronique, manufacture,…) [And04].

2.5.3 Les différents modèles du MDA

Dans l'approche MDA, l'OMG a défini plusieurs modèles qui vont servir dans un premier temps à modéliser l'application puis, par transformations successives, à générer le code de l'application. Les quatre principaux types de modèles définis dans l'approche MDA sont les suivants :

- **CIM** (Computation Indépendant Model) :

Appelé aussi modèle de domaine ou modèle métier, le CIM capture les exigences en termes de besoins et décrit la situation dans laquelle le système sera utilisé. Le CIM permet la vision du système dans l'environnement où il opèrera, mais sans rentrer dans le détail de la structure du système, ni de son implémentation.

- **PIM** (Platform Indépendant Model) :

Le PIM décrit le système indépendamment de la plate-forme cible sur laquelle il s'exécutera. Il présente donc une vue fonctionnelle détaillée du système, sans détails techniques. Il peut être raffiné progressivement jusqu'à intégrer des détails d'architecture spécifiques à un type de plate-forme (machine virtuelle, système d'exploitation, etc.) mais il doit rester technologiquement neutre.

- **PSM (Platform Spécifique Model)**

Il est dépendant de la plate-forme technique spécifiée par l'architecte. Le PSM sert essentiellement de base à la génération de code exécutable vers la ou les plates-formes techniques. Le PSM décrit comment le système utilisera cette ou ces plates-formes. Il existe plusieurs niveaux de PSM. Le premier, issu de la transformation d'un PIM, se représente par un schéma UML spécifique à une plate-forme. Les autres PSM sont obtenus par transformations successives jusqu'à l'obtention du code dans un langage spécifique (Java, C++, C#, etc.) Un PSM d'implémentation contiendra par exemple des informations comme le code du programme, les types pour l'implémentation, les programmes liés, les descripteurs de déploiement [And04].

- **PDM (Plateforme Description Model)**

Cette notion n'est pas encore bien définie par l'OMG, pour l'instant il s'agit plus d'une piste de recherche. Un PDM contient des informations pour la transformation de modèles vers une plate-forme en particulier et il est spécifique de celle-ci. C'est un modèle de transformation qui va permettre le passage du PIM vers le PSM. Normalement, chaque fournisseur de plate-forme devrait le proposer [And04].

2.5.4 Transformations de modèles dans MDA

La Figure 2.4 donne une vue générale des transformations possibles entre les différents types de modèles.

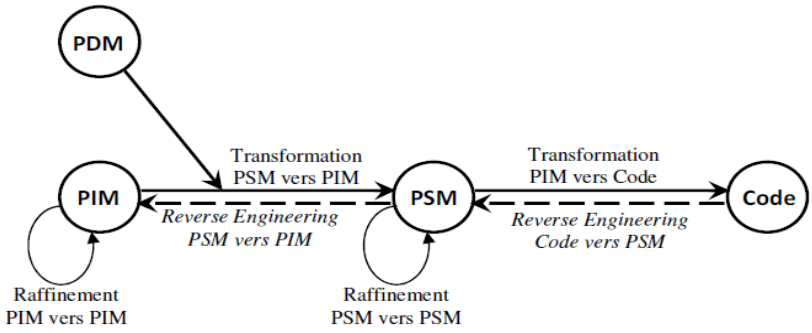


Figure 2.4: Les modèles et les transformations dans l'approche MDA [ker11].

- **Transformations PIM vers PIM et PSM vers PSM :** Les transformations de type PIM vers PIM ou PSM vers PSM visent à enrichir, filtrer ou spécialiser le modèle. Il s'agit de transformations de modèle à modèle.
- **Transformation PIM vers PSM :** La transformation de PIM vers PSM permet de spécialiser le PIM en fonction de la plate-forme cible choisie. Elle n'est effectuée qu'une fois le PIM suffisamment raffiné. Cette transformation de modèle à modèle est réalisée en s'appuyant sur les informations fournies par le PDM.
- **Transformation PSM vers Code :** La transformation de PSM vers l'implémentation (le code) est une transformation de type modèle à texte. Le code est parfois assimilé à un PSM exécutable. Dans la pratique, il n'est généralement pas possible d'obtenir la totalité du code à partir du modèle et il est alors nécessaire de le compléter manuellement [Cza03].
- **Transformations inverses PSM PIM et code PSM :** Ces transformations sont des opérations de rétro-ingénierie (reverse engineering). Ce type de transformations pose de nombreuses difficultés mais il est essentiel pour la réutilisation de l'existant dans le cadre de l'approche MDA [ker11].

2.5.5 Le principe de transformations de modèles

Une transformation de modèle est un ensemble de règles qui s'appliquent aux éléments d'un méta-modèle définissant les modèles sources valides (c'est-à-dire auxquels la transformation peut s'appliquer) et qui définissent pour chacun de ces éléments leur(s) équivalent(s) parmi les éléments d'un méta-modèle cible. Le moteur de transformation (aussi appelé « transformateur de modèles ») lit le modèle source, qui doit donc être conforme au méta-modèle source, et applique les règles définies dans la transformation de modèle afin de créer le modèle cible qui sera conforme au méta-modèle cible. Les méta-modèles source et cible peuvent faire partie de la définition de langages de modélisation

et, dans ce sens, les transformations de modèles peuvent être considérées comme des systèmes de traduction (ou de réécriture) d'un langage vers un autre [Bla05].

- **La structure d'une règle de transformation** : Une règle de transformation est une description de la manière dont une ou plusieurs constructions dans un modèle source peuvent être transformées en une ou plusieurs constructions dans un modèle cible. Une transformation de modèle est notamment caractérisée par la réunion des éléments suivants : des règles de transformation, une relation entre la source et la cible, un ordonnancement des règles, une organisation des règles, une traçabilité et une direction.

- **Règles de transformation** : Une règle de transformation contient deux parties : une partie gauche (left-hand side «LHS») et une partie droite (right-hand side «RHS»). La LHS exprime des accès aux modèles sources, alors que la RHS indique les expansions (création, modification, suppression) dans les modèles cibles. Chacune des deux parties peut être représentée par une combinaison de :
  - **Variables** : une variable contient un élément de modèle (source ou cible) ou une valeur intermédiaire nécessaire à l'expression de la règle.
  - **Patterns** : un pattern désigne un fragment de modèle et peut contenir des variables. Il peut être représenté à l'aide d'une syntaxe abstraite ou concrète dans le langage des modèles correspondants. Cette syntaxe peut être textuelle ou graphique.
  - **Logique** : une logique permet d'exprimer des calculs et des contraintes sur les éléments de modèle. Cette expression peut être non exécutable (expression de relations entre modèles) ou exécutable. Une logique exécutable peut être sous une forme déclarative ou impérative.

- **Relation entre les modèles source et cible** : Pour certains types de transformations, la création d'un nouveau modèle cible est nécessaire. Pour d'autres, la source et la cible sont le même modèle, ce qui revient en fait à une modification de modèle.

- **Organisation des règles** : L'organisation des règles définit comment composer plusieurs règles de transformation. Les règles peuvent être organisées de façon modulaire, avec la notion d'importation. Les règles peuvent également utiliser la réutilisation, par le biais de mécanismes d'héritage entre règles, ou la composition, par le biais d'un ordonnancement explicite. Enfin, les règles peuvent être organisées selon une structure dépendante du modèle source ou du modèle cible.

- **Ordonnancement des règles** : Les mécanismes d'ordonnancement déterminent l'ordre dans lequel les règles sont appliquées. Dans le cas d'un ordonnancement implicite, l'algorithme d'ordonnancement est défini par l'outil de transformation. Dans le cas d'un ordonnancement explicite, des mécanismes permettent de spécifier l'ordre d'exécution des règles. Cet ordre d'exécution peut être défini de manière externe ou interne : tandis qu'un mécanisme externe établit une séparation claire entre les règles et la logique d'ordonnancement, un mécanisme interne permet aux règles d'invoquer d'autres règles. Enfin, l'ordonnancement des règles se repose également sur des conditions, des itérations ou sur une séparation en plusieurs phases, certaines règles ne pouvant être appliquées que dans certaines phases.
- **Traçabilité** : Les transformations peuvent stocker les corrélations entre les éléments des modèles source et cible. Certaines approches fournissent des mécanismes dédiés pour supporter la traçabilité. Dans les autres cas, le développeur doit implémenter la traçabilité de la même manière qu'il crée n'importe quel autre lien dans un modèle.
- **Direction** : Les transformations peuvent être unidirectionnelles ou bidirectionnelles. Dans le premier cas, le modèle cible est calculé ou mis à jour sur la base du modèle source uniquement. Dans le second cas, une synchronisation entre les modèles source et cible est possible [Bla05].

2.5.6 MOF et L'architecture à quatre niveaux

L'OMG, dans le cadre de ses travaux concernant la méta-modélisation, a défini la notion de méta-méta-modèle ainsi que la standardisation d'une architecture générale décrivant les liens entre modèles, méta-modèles et méta-méta-modèles. Cette architecture est hiérarchisée en quatre niveaux comme le montre la Figure 2.5.

Le MDA est composé de plusieurs modèles, « descriptions abstraites d'une entité du monde réel utilisant un formalisme donné » qui vont servir dans un premier temps à modéliser l'application, puis par transformations successives à générer du code.

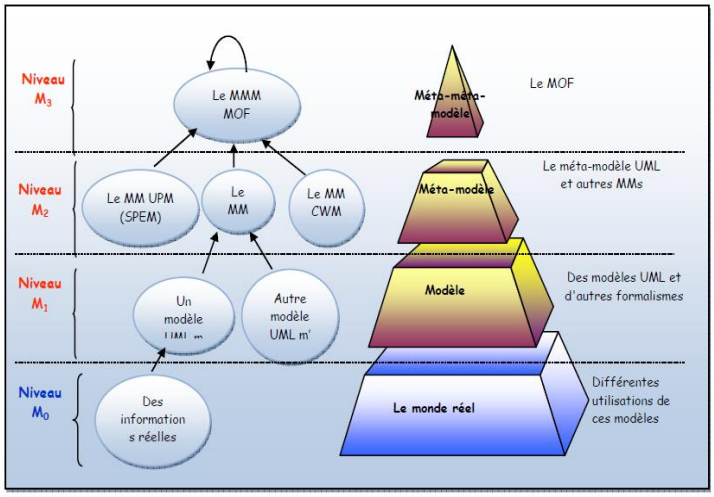


Figure 2.5: Architecture à quatre niveaux [ker11].

- **NiveauM3 (MMM*) :*** Le niveau M3 (ou méta-méta-modèle) est composé d'une seule entité qui s'appelle le MOF (*Meta Object Facility*). Le MOF permet de décrire la structure des méta-modèles, d'étendre ou de modifier les méta-modèles existants. Le MOF est réflexif,

- **Niveau M2 (MM) :** Le niveau M2 (ou méta-modèle) définit le langage de modélisation et la grammaire de représentation des modèles M1. Le méta-modèle UML, qui définit la structure interne des modèles UML suivant le standard UML, fait partie de ce niveau. Les profils UML, qui étendent le méta-modèle UML, appartiennent aussi à ce niveau. Les concepts définis par un méta modèle sont des instances des concepts du MOF [Fav06].

2.6 Processus de Vérification en IDM

Dans le contexte de l'ingénierie dirigée par les modèles, le développement des systèmes complexes fait appel à différentes techniques de modélisation qui dépendent :

- Du domaine du système ou du sous-système.
- Des différentes phases du cycle de développement.
- Des différents niveaux d'abstraction aux quels le système est étudié.
- Des différents aspects spécifiques à analyser et à vérifier lors de la conception.

Dans le but d'analyser le comportement modélisé pour assurer le bon fonctionnement du système et détecter d'éventuels problèmes, différentes techniques telles

que la simulation, le test ou les méthodes formelles sont alors utilisées dès les premières étapes de sa conception.

Dans ces techniques, les modèles peuvent être utilisés comme support pour ces activités de vérification et de validation. Certaines analyses peuvent être conduites directement en utilisant les modèles. Par exemple, des simulations ou des tests sur des modèles comportementaux sont souvent favorisés. En effet, il s'agit d'une méthode simple et relativement peu coûteuse, consistant à générer des cas de tests pertinents pour vérifier les scénarios attendus. En revanche, même pour un système d'une taille raisonnable, l'analyse des comportements possibles ne peut pas être exhaustive et l'on court le risque de ne pas identifier des situations potentiellement dangereuses pour le système.

D'autres analyses nécessitent la transformation des modèles réalisés dans des formalismes formels adaptés à un type de vérification formelle désirée. Les méthodes formelles, telles que la preuve de théorème, les réseaux de Petri ou le Model Checking, s'appuient sur un cadre mathématique précis et non ambigu permettant de modéliser à la fois le système et les propriétés qu'il doit vérifier [ker11].

3. Les méthodes formelles de vérification

Les méthodes formelles sont des techniques basées sur les mathématiques pour décrire des propriétés de systèmes. Elles fournissent un cadre très rigoureux pour spécifier, développer et vérifier des systèmes d'une façon systématique, plutôt que d'une façon ad hoc, afin de démontrer leur validité par rapport à une certaine spécification [Win90].

3.1 Les langages formels

Les méthodes formelles reposent sur l'utilisation de langages formels pour donner une spécification du système que l'on souhaite développer à un niveau de détail désiré.

Un langage formel est en effet un langage doté d'une sémantique mathématique adéquate basée sur des règles d'interprétation et des règles de déduction [Pet99]. Les règles d'interprétation garantissent l'absence d'ambiguïté dans les descriptions produites, contrairement à des descriptions en langage informel ou semi-formel qui peuvent donner lieu à différentes interprétations. Alors que les règles de déduction permettent de raisonner sur les spécifications afin de découvrir de potentielles incomplétudes, inconsistances ou pour prouver des propriétés attendues [ker11].

3.2 Techniques d'analyse

Comme les systèmes deviennent de plus en plus complexes, il est crucial non seulement d'assurer certaines performances, mais également de garantir l'absence de risques de dysfonctionnement ou au moins limiter leur impact. Pour cela, de nombreuses méthodes et techniques ont été développées durant les dernières décennies pour analyser ce type de systèmes.

3.2.1 Vérification

La vérification répond à la question "Construisons-nous correctement le modèle ? La vérification est l'ensemble des actions de revue, inspection, test, simulation, preuve automatique, ou autres techniques appropriées permettant d'établir et de documenter la conformité des artéfacts du développement vis-à-vis des critères préétablis. La vérification est définie comme étant "*la confirmation par examen et apport de preuves tangibles (informations dont la véracité peut être démontrée, fondée sur des faits obtenus par observation, mesures, essais ou autres moyens) que les exigences spécifiées ont été satisfaites*" [ker11].

3.2.2 Validation

La validation cherche à répondre à la question "Construisons nous le bon modèle ? " La validation consiste à évaluer l'adéquation du système développé vis-à-vis des besoins exprimés par ses futurs utilisateurs. Par définition la validation est la "confirmation, par examen et apport de preuves tangibles, que les exigences particulières pour un usage spécifique prévu sont satisfaites. Plusieurs validations peuvent être effectuées s'il y a différents usages prévus" [ker11].

3.2.3 Qualification

La qualification consiste à s'assurer que le modèle peut servir à la communication sans ambiguïtés ni interprétation parasite possible entre les activités du projet et/ou entre les acteurs d'un groupe de travail [ker11].

3.2.4 Certification

Elle consiste à s'assurer que le modèle respecte une norme et peut servir de base à l'établissement d'un référentiel réutilisable et générique à un domaine. Cela sous-entend la nécessaire implication et la responsabilité d'un organisme tiers qui reconnaît la

pertinence, la rigueur, l'intérêt du modèle et garantit ces qualités lors de sa diffusion [ker11].

3.3 Techniques de vérification formelle

De façon générale, on distingue deux grandes catégories de techniques pour vérifier formellement la correction d'un système. La première catégorie de techniques est appelée vérification de modèles, ou *model checking*, qui consiste à construire un modèle à partir d'une description formelle d'un système. La seconde catégorie regroupe 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.

3.3.1 Vérification de Modèle (Model Checking)

Le model checking est une approche automatisée permettant de vérifier qu'un modèle de système est conforme à ses spécifications. Le comportement du système est formellement modélisé, via des automates, réseaux de Petri, algèbres de processus, . . . et les spécifications, exprimant les propriétés attendues du système, sont formellement exprimées par exemple via des formules de logique temporelle [Bou15].

En pratique, les propriétés du système sont souvent classées en deux grandes catégories informelles. Les propriétés de *sûreté* énonce qu'une situation particulière ne peut être atteinte. Les propriétés de *vivacité* énoncent quelque chose de mauvais (ou de bon) qui finira par se produire.

La vérification de modèles 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 [Att07].

La Figure 2.6, donne un aperçu simplifié du cycle d'utilisation du model checking. Le cycle peut se diviser en trois phases :

1. Modélisation formelle du comportement du système

2. Expression formelle des propriétés attendues

3. Si une propriété n'est pas satisfaite, un contre-exemple est produit qui décrit un scénario possible d'erreur (i.e de violation de la propriété).

L'analyse de celui-ci aide à apporter les corrections nécessaires que ce soit sur la modélisation du comportement du système ou sur l'expression formelle des propriétés attendues.

Ce cycle est répété jusqu'à ce que toutes les formules, c'est à dire toutes les spécifications, soient vérifiées.

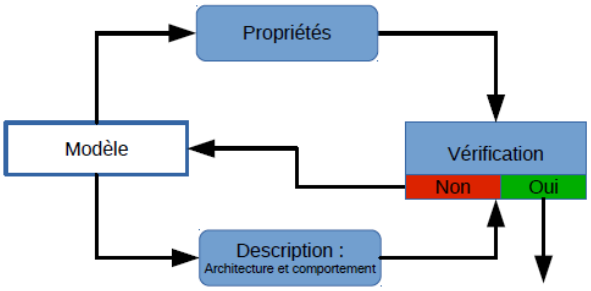


Figure 2.6: Cycle d'utilisation du model checking [Bou15].

3.3.2 Preuve de théorèmes (Theorem proving)

La preuve de théorèmes est une technique où le système et les propriétés recherchées sont exprimés comme des formules dans une logique mathématique. Cette logique est décrite par un système formel qui définit un ensemble d'axiomes et de règles de déduction. La preuve de théorèmes est le processus de recherche de la preuve d'une propriété à partir des axiomes du système. Les étapes pendant la preuve font appel aux axiomes et aux règles, ainsi qu'aux définitions et lemmes qui ont été éventuellement 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 [ker11].

4. Combinaison d'IDM avec les Méthodes Formelles

Aujourd'hui l'utilisation de méthodes formelles (*MFs*) est essentielle pour le développement de systèmes complexes, en particulier pour les systèmes 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 la transformation de modèles.

Chacune des deux approches a des points forts et des points faibles. Dans ce qui suit, nous examinons comment ces deux approches peuvent être combinées en montrant comment les inconvénients des méthodes formelles peuvent être surmontés grâce aux apports de l'IDM et réciproquement [Gar09]. Le tableau 3.1 représente brièvement les avantages et les limites de l'IDM et des MFs.

	Avantages	**limites**
IDM	▪ Notations conviviales et souvent graphiques ▪ Génération automatique d'outils de développement ▪ Transformations automatiques de modèles	▪ Manque de sémantiques formelles ▪ Analyse de modèles impossible
MFs	▪ Fondements mathématiques rigoureux ▪ Analyse formelle de modèles	▪ Notations complexes ▪ Absence d'outils de développement ▪ Manque d'intégration

Tableau 2.1 : Les MFs& IDM [Att07].

➢ **Avantages des MFs**. L'utilisation des méthodes formelles dans l'ingénierie des systèmes devient indispensable, surtout dans les phases amont du développement. Un modèle abstrait du système peut servir de support pour s'assurer que le système en cours de développement répond aux exigences des clients par simulation ou tests, et de garantir certaines propriétés liées à son comportement par l'analyse formelle (validation et vérification).

➢ **Inconvénients des MFs**. Bien qu'il existe plusieurs applications des méthodes formelles dans le monde industriel et avec de bons résultats, les développeurs hésitent encore à adopter les MFs. En plus de leur difficulté d'utilisation et d'apprentissage, ce scepticisme est principalement dû :

1) Aux notations complexes que les techniques formelles utilisent par rapport à d'autres notations intuitives et graphiques utilisées par les langages semi-formels comme le langage UML.

2) A l'absence de support outillé qui permet d'aider et d'assister les développeurs dans leurs démarches de développement de manière transparente.

3) Au manque d'intégration entre les différentes méthodes formelles et leurs outils d'analyse adjacents.

- **Avantages de l'IDM**. L'approche IDM met davantage l'accent sur l'automatisation du processus de développement des systèmes. Cette approche est basée principalement sur des représentations du système à un haut niveau d'abstraction qui sont les modèles. Le processus de développement, dans cette approche, revient à raffiner, maintenir, et éventuellement de transformer en d'autres modèles ou de générer le code exécutable. Il est important à noter que ces différentes activités se font d'une manière automatique. En plus, la méta-modélisation, qui est aussi un concept clé de l'approche IDM, permet de donner à un langage de modélisation une notation abstraite, ce qui permet de générer automatiquement son éditeur. La méta modélisation de langages est de plus en plus adoptée pour des domaines spécifiques afin de réduire la complexité et d'exprimer efficacement les concepts du domaine.

- **Inconvénients de l'IDM**. Bien que la définition d'une syntaxe abstraite d'un langage par un méta-modèle est bien maîtrisée et supportée par de nombreux environnements de méta modélisation, la définition de la sémantique de ces langages reste une question ouverte et cruciale. Actuellement, les environnements de méta-modélisation sont en mesure de faire face à la plupart des problèmes de définition syntaxique, mais ils manquent de tout support rigoureux permettant de fournir la sémantique des méta-modèles. La sémantique est généralement donnée en langage naturel, cela implique que les langages définis par méta modélisation ne sont pas encore aptes à l'analyse formelle de leurs modèles.

L'absence de notations conviviale, d'intégration des déférentes techniques formelles et d'interopérables de leurs outils sont des défis importants pour les MFs. L'IDM permet, par le biais des notions de méta-modèle et de transformation de modèles, de concevoir des supports outillés pour les méthodes formelles tout en assurant leur interopérabilité. En ce sens, l'IDM n'apporte rien de nouveau sur le plan théorique aux concepts d'analyse formelle, mais elle peut leur permettre une meilleure intégration afin de profiter pleinement de leurs avantages [ker11].

5. Conclusion

Dans ce chapitre, nous avons présenté les principaux concepts du domaine de l'ingénierie des modèles. Nous avons mis en évidence le rôle central des modèles et de l'outillage sous-jacent dans le processus de conception des systèmes. Nous avons aussi

identifié et mis en valeur les problématiques actuelles liées à la manipulation des modèles et au besoin de s'assurer de leur qualité tout en réduisant les coûts et le délai de production. Dans le cadre de l'approche MDA, nous avons présenté différentes techniques de traitement des modèles parmi lesquelles nous avons détaillé la méta-modélisation ainsi que les techniques de transformation de modèles. Ces techniques sont aujourd'hui des pivots majeurs pour l'innovation dans les domaines de recherche liés à la modélisation des systèmes complexes.

Une combinaison des techniques de MDA et le système Maude, représente notre but, pour cette raison le langage Maude fera l'objet du prochain chapitre.

Chapitre 03 :

La logique de réécriture et Le Langage Maude

1. Introduction

La logique de réécriture, est une logique qui a été présentée par José Meseguer [Mes 92], comme une conséquence de son travail sur les logiques générales pour décrire les systèmes concurrents. Cette logique permet de réécrire un terme c-à-d le remplacer par un terme équivalent, en vertu des lois de l'algèbre des termes [Mes02]. La réécriture pour cette logique permet de calculer une relation de réécrivabilité entre des termes algébriques. L'idée essentielle de la logique de réécriture est que la sémantique de la réécriture peut être rigoureusement changée d'une manière très fructueuse [Mes92]. Il a été largement démontré qu'elle permet de raisonner parfaitement sur le comportement des systèmes concurrents. Donc la logique de réécriture est un modèle de calcul et un cadre sémantique expressif pour la concurrence, le parallélisme, la communication, et l'interaction. Dans ce contexte, Maude représente un langage formel largement utilisée dans la vérification des systèmes concurrents Maude est un langage formel de spécification algébrique et de programmation déclarative, simple, expressif et performant. Dans ce chapitre, nous allons commencer par introduire la logique de réécriture et sa sémantique et présenter ses concepts de base. Ensuite le langage de la logique de réécriture Maude et ses différents niveaux de spécification.

2. Logique de Réécriture

La logique de réécriture est une logique de changement concurrent qui peut traiter l'état et le calcul des systèmes concurrents. Cette logique a été largement utilisée pour spécifier et analyser des systèmes et langages dans différents domaines d'applications. Donc la logique de réécriture offre un cadre formel nécessaire pour la spécification et l'étude du comportement des systèmes concurrents. En effet, elle permet de raisonner sur des changements complexes possibles correspondant aux actions atomiques axiomatisées par les règles de réécriture. Le point clé de cette logique est que la déduction logique, qui est intrinsèquement concurrente, correspond au calcul dans un système concurrent [Mes92].

Dans la logique de réécriture, un système concurrent est décrit par une théorie de réécriture **R = (Σ, E, L, R) où (Σ, E)** désigne la signature (une théorie équationnelle) définissant la structure algébrique particulière des états du système (multi-ensemble, arbre binaire…) qui sont distribués selon cette même structure. La structure dynamique du système est décrite par les règles de réécriture étiquetées ***R*** (***L***est un ensemble d'étiquettes

de ces règles). Les règles de réécriture précisent quelles sont les transitions élémentaires et locales possibles dans l'état actuel du système concurrent.

Chaque règle (notée ***[t] → [t']***) correspond à une action pouvant survenir en concurrence avec d'autres actions. Donc, la logique de réécriture est une logique qui capture clairement le changement concurrent dans un système.

Définition (Théorie de réécriture étiquetée)

Une théorie de réécriture ***R*** est un 4-uplet ***(Σ, E, L, R)*** tel que :

1. ***Σ*** est un ensemble de symboles de fonctions, et de sortes.
2. ***E*** un ensemble de ***Σ***-équations (l'ensemble des équations entre les ***Σ***-termes).
3. ***L*** est un ensemble d'étiquettes.
4. ***R*** est un ensemble de règles de réécriture, défini ainsi : $R \subseteq L \times (T_{\Sigma,E}(X))^2$

Chaque règle est un couple d'éléments, le premier est une étiquette, le second est une paire de classes d'équivalence de termes $T_{\Sigma,E}(X)$ sur la signature (Σ,E),modulo les équations E, avec $X = \{x_1, \ldots, x_n, \ldots\}$ un ensemble infini et dénombrable de variables [Mes04].

Pour une règle de réécriture de la forme $r([t], [t'], C_1, \ldots, C_k)$, la notation suivante est utilisée, r : [t] → [t'] if C1 ∧ … ∧ C_k, où une règle r exprime que la classe d'équivalence contenant le terme t peut se réécrire en la classe d'équivalence contenant le terme t' si la condition de la règle C1 ∧… ∧C_k est vérifiée. Cette dernière est appelée condition de la règle et peut être abrégée par la lettre C, et la règle de réécriture, dans ce cas, est dite conditionnelle.

La partie conditionnelle d'une règle peut être vide, dans ce cas les règles sont appelées règles de réécriture inconditionnelles et sont notés par r : [t] → [t']

Une règle de réécriture peut être paramétrée par un ensemble de variables {x1, …, xn} qui apparaissent soit dans ***t, t'***ou ***C***, et nous écrivons :

r : [t(x1, …, xn)] → [t'(x1, …, xn)] if C(x1, …, xn) [Mes92].

3. Système Maude

Maude est un langage formel de spécification et de programmation déclarative basé sur une théorie mathématique de la logique de réécriture. La logique de réécriture et le langage Maude sont développés par Jose Meseguar et son groupe dans le laboratoire d'informatique en SRI International. Maude est un langage simple expressif et performant,

il est considéré parmi les meilleurs langages dans le domaine de spécification algébrique et la modélisation des systèmes concurrents [Mcc03].

Maude spécifie des théories de la logique de réécriture, les types de données sont définies algébriquement par des équations et le comportement dynamique du système est définie par des règles de réécritures.

Maude supporte aussi la programmation orientée objet avec l'inclusion de l'héritage multiple et la communication asynchrone par le passage des messages.

Le groupe de Maude ont focalisé leurs efforts aussi sur la performance, la version actuelle de l'interpréteur peut atteindre des millions de réécriture par seconde. Par ce fait, Maude est en concurrence avec les langages de haut niveau en termes d'efficience.

Maude contient aussi un support d'utilisation des sockets pour la programmation réseau ce qui permet non seulement de modéliser, de simuler et d'analyser des systèmes concurrents mais aussi de les programmer.

Pour spécifier un système concurrent, Maude offre trois types de modules: Par conséquent, on différencie trois types de modules: les modules fonctionnels pour implémenter les théories équationnelles. Les modules système implémentent les théories de réécriture et définissent le comportement dynamique d'un système. Les modules orientés-objet implémentent les théories de réécriture orientées objet (ils peuvent être réduits à des modules systèmes) [Cla99].

3.1 Modules fonctionnels

Ces modules définissent les types de données et les opérations qui sont utilisés par les équations. En Maude, une spécification équationnelle est un module fonctionnel qui est représenté par la syntaxe suivante :

fmod MODULENAME is *(le nom du module)**

BODY *(les déclarations de sortes, d'opérations, de variables,**

D'équations, d'axiomes d'appartenance et de commentaires*)*

Endfm.

Où MODULENAME est le nom de module introduit, et **BODY** est l'ensemble de déclarations des sortes, d'opérations, des variables, des équations, des axiomes d'appartenance et des commentaires. Les commentaires commencent par '***' ou '---' et se termine par la fin de la ligne courante ou elles commencent par *** (ou --- (et se termine par l'occurrence de")".

Le corps du module spécifie une théorie ***(Σ, E U A, Φ)*** 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*)***. 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 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.

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** [Cla02].

Un exemple d'un module fonctionnel des nombres naturels est le suivant :

```
fmod BASIC-NAT is          ***(le nom du module est : BASIC-NAT)
sort Nat                   ***(declaration de sorte naturelle)
op 0 : -> Nat  [ ctor ] .  *** (declaration d'une opération
<mathématiquement : fonction> qui donne zéro)
op s_ : Nat -> Nat [ ctor  ] .***(déclaration d'une opération)
op _+_ : Nat Nat -> Nat .
op max : Nat Nat -> Nat .
vars N M : Nat .       ***(déclaration des variables naturelles N et M)
eq 0 + N = N .***(équation)
eq s M + N = s (M + N).
eq max(0, M) = M .
eq max(N, 0) = N.
eq max(s N, s M) = s max(N, M).
endfm [Cde99]
```

3.2 Modules Systèmes

Les modules systèmes sont utilisés pour définir le comportement dynamique des systèmes concurrents en enrichissant les modules fonctionnels par un ensemble de règles de réécriture. L'introduction des règles de réécriture permet d'exprimer la concurrence dans les systèmes.

Un module système décrive une théorie de réécriture qui inclue des sortes, des opérations, des variables, des équations, des axiomes d'appartenances (conditionnelles et inconditionnelles) et des règles de réécriture conditionnelles et inconditionnelles.

Une règle de réécriture s'exécute quand sa partie gauche correspond (match) une portion dans l'état global du système et avec la satisfaction de la condition en cas d'une règle conditionnelle [Mcc03].

En Maude, la syntaxe suivante représente le module système :

mod MODULENAME is

BODY

endm.

Les règles de réécriture **R** sont introduites avec les mots clés ***rl*** ou ***crl***. Elles sont spécifiées dans Maude avec la syntaxe :

crl [l] : t => t' if cond .

Si la règle est non conditionnelle, le mot clé ***crl*** est remplacé par ***rl*** et la clause «***if cond***» est omise [Cla02].

Un exemple qui montre l'utilisation de ces modules est le suivant **:**

```
mod CHOICE-INT is***(CHOICE-INT est le nom du module)
including INT          ***(importation  du module INT (le module pour les entiers)
pour utiliser ses opérateurs)
op _?_ : Int Int -> Int***(déclaration d'un opérateur)
vars I J : Int .***(déclaration de deux entiers I et J)
rl [choose_first] : I ? J =>I .
rl [choose_second] : I ? J => J.
endm[Cde99]
```

3.3 Modules orientés objet

Les modules orientés objet sont supportés par le système Full-Maude [Mcc03]. Notons que Full Maude est une extension de Maude (Core Maude) dont le code est écrit en Maude, ce qui enrichit le langage Maude avec un module algébrique très puissant et extensible. Ces modules orientés objet sont introduits par les mots clés :

(omod MODULNAMEis

…..

Endom)

Où le corps du module est une théorie de réécriture $\mathfrak{R}$= (Σ, E U A, Φ, R). 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.

Un objet est représenté par le terme < O : C | a1 : v1, …, an : vn>, où O est le nom de l'objet instance de la classe C, ai, ∈ 1..n, les noms des attributs de l'objet, et vi, leurs valeurs respectives.

La déclaration des classes suit la syntaxe : class C | a1 : s1, …, an : sn. où C est le nom de la classe et si 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 :

crl [r] : M1 … Mn < O1 : F1 \| a1> … < Om : Fm \| am > =>< Oi1 : F'i1 \| a'i1 > …

<Oik : F'ik \| a'ik> M1' … Mp' if Cond .

Où ***r*** est l'étiquette de la règle, ***Ms, s ∈ 1..n, et M'u, u ∈ 1..p***sont des messages, ***Oi, i ∈ 1..m, et Oil, l ∈ 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***.

Un exemple qui illustre l'utilisation des modules orientés objet est le suivant :

```
load full-maude ***(charger de Full Maude)
(omod POPULATION is
protecting NAT .***(utiliser les operation naturelles)
protecting STRING . ***(pour utiliser les operation string)
sortStatus . ***(état)
op single : ->Status [ctor] . ***(état célibataire)
ops engaged married separated :Oid -> Status [ctor] . ***(états : dévorcé, marrié, fiancé)
subsort String <Oid . **(déclaration d'une sous-sorte de String)
class Person | age : Nat, status : Status . ***(déclaration d'une class nommée personne, age ;de type naturel, état ; de type état)
vars N N' : Nat . vars X X' : String .
crl [birthday] :< X : Person | age : N > =>< X : Person | age : N+ 1 > if N < 999 .
crl [engagement] :< X : Person | age : N, status : single >< X' : Person | age : N', status : single > =>< X : Person | status : engaged(X') >< X' : Person | status : engaged(X) > if N > 15 or N' > 15 .
op initState : -> Configuration .        *** etat initial
eq initState = < "Peter" : Person | age : 37, status : single >< "Lizzie" : Person | age : 34, status : single >< "Sam the Snake" : Person | age : 40, status : single > .
endom) [Cde99].
```

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 [Cla02].

Les modules systèmes et fonctionnels peuvent être utilisés en Full Maude. Il faut juste les mètre entre deux parenthèses ().

- **Règles de réécriture pour l'objet** : l'associativité et la commutativité d'une configuration multi ensembles rend la dernière plus flexible. Elle est vue comme une soupe dans laquelle les objets et les messages se flottent. Ces derniers peuvent être ensemble à chaque instant pour participer à une transaction concurrente.

En général, une règle de réécriture dans ***R*** qui décrit le dynamique d'un système orienté objet peut avoir la forme :

$$
\begin{aligned}
&r : M_1 \dots M_n \ < O_1 : F_1 \mid atts_1 > \cdots < O_m : F_m \mid atts_m > \\
&\rightarrow < O_{i1} : F'_{i1} \mid atts'_{i1} > \cdots \ < O_{ik} : F'_{ik} \mid atts'_{ik} > \\
&\quad < Q_1 : D_1 \mid atts''_1 > \cdots < Q_p : D_p \mid atts''_p > \\
&\quad M'_1 \dots M'_q \\
&\quad \Leftarrow C
\end{aligned}
$$

Où : ***r*** est l'étiquette de la règle, ***Ms*** sont des messages, $i_1 \ldots i_k$, sont les différents numéros des objets, et ***C*** c'est la condition [Mcc03].

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 [Ben11].

5. Exécution et analyse formelle sous 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). Maude nous offre la possibilité de simuler l'exécution de telles 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 .

La commande de réécriture ***rewrite*** (abrégée par ***rew***) et la commande de réécriture équitable ***frewrite***(abrégée par ***frew***) exécutent une seule séquence de réécriture (parmi plusieurs séquences possibles) à partir d'un terme initialement donné suivant la syntaxe :

rewrite {in module : } term .

frewrite {in module : } term .

Ces commandes permettent de réécrire un terme initial en utilisant les règles, les équations et les axiomes d'adhésion dans le module spécifié [Cla16].

6. Analyse formelle et vérification des propriétés

La vérification formelle de modèles sur les systèmes spécifiés en Maude s'effectue à l'aide d'outils disponibles autour du système Maude. Parmi ces outils, on distingue un outil d'analyse d'accessibilité (la commande ***search***) et un module de vérification par *model checking* de propriétés exprimées en logique linéaire temporelle (LTL) [Eke02]. Pour analyser toutes les séquences de réécritures possibles à partir d'un état (terme) initial t_0, on utilise la commande ***search***. Celle-ci recherche si des états correspondants à des patterns donnés et satisfaisant certaines conditions, peuvent être accessibles à partir de t_0. L'exécution de cette commande effectue un parcours en profondeur de l'arbre de calcul (arbre d'accessibilité), généré lors de cette recherche, afin de détecter les violations d'invariants dans les systèmes à états infinis [Cla07].

L'analyse formelle des systèmes par Model Checking est un ensemble de techniques pour vérifier automatiquement des propriétés temporelles relatives au comportement des systèmes [Cla01]. Le Model Checking permet de vérifier la satisfiabilité

d'une propriété donnée dans un état ou un ensemble d'états, en faisant une exploration exhaustive de l'ensemble des états accessibles à partir d'un état initial. Il reçoit en entrée une abstraction du comportement du système (un système de transitions), représentée par une structure Kripke ***K***, et une propriété ϕ de ce système, formulée dans une certaine logique temporelle, et répond si l'abstraction satisfait ou non la formule ϕ c'est-à-dire, si $K \models \phi$. L'intérêt du Model Checking est qu'il retourne une trace d'exécution du système violant la propriété lorsque cette dernière est non valide.

6.1 Logique temporelle linéaire (LTL)

La logique temporelle linéaire (LTL) est une extension de la logique classique avec des opérateurs temporels, tels que ***G*** et ***F*** (représentant respectivement " globalement ", " finalement ou fatalement ") qui permettent d'exprimer des propriétés portant sur l'exécution d'une séquence d'états. Elle est dite temporelle car elle décrit le séquencement d'évènements observés dans un système. Cette logique permet de spécifier des propriétés intéressantes pour les systèmes concurrents notamment les propriétés de **sûreté**, d'**accessibilité**, de **vivacité** et d'**équité**.

- **Sûreté** : quelque chose de mauvais n'arrive jamais.
- **Accessibilité** : une certaine situation peut être atteinte.
- **Vivacité** : quelque chose de bon est toujours possible.
- **Equité** : quelque chose se répètera infiniment souvent.

Les formules LTL sont construites à partir de variables propositionnelles appelées propositions atomiques, d'opérateurs booléens **(¬, ∧, ∨, ⇒, ⇔),** et d'opérateurs temporels, ***F*** **(Futur),** ***G*** **(Global),** ***U*** **(Until),** ***X*** **(neXt)**.

Les propriétés atomiques permettent de décrire les états du système : un état ***t*** est dit étiqueté par une proposition atomique ϕ si ϕ est vraie dans ***t***. Les opérateurs temporels permettent de relier des états du système au sein d'une séquence d'exécution.

- La formule ***Xf*** indique que la formule LTL ***f*** doit être vérifiée à l'instant suivant immédiat le long de l'exécution (***neXt***).
- La formule ***f U g*** indique que ***f*** est toujours vrai jusqu'à un état où ***g*** est vrai (*Until*).
- La formule ***Gf*** signifie ***f*** est toujours vérifiée (*Generally*) dans toute l'exécution.

• La formule ***Ff*** *indique* que ***f*** doit être vérifiée plus tard au moins dans un état de l'exécution (*Fatalement*) [Ben11].

6.2 Model Checker LTL de Maude

L'outil de model checking qu'offre Maude fait appel à la logique LTL. Il exige que le système à vérifier, décrit par une théorie de réécriture $\mathcal{R} = (\Sigma, E, \Phi, R)$, soit à état fini, c'est-à-dire qu'à partir d'un état initial [t_0], l'ensemble des états accessibles défini par $\{[u] \in T_{\Sigma,E} \mid \mathcal{R} \vdash [t_0] \rightarrow [u]\}$ soit fini. Donc, le LTL model checker de Maude prend en entrée une théorie de réécriture $\mathcal{R}$ dont il génère la structure Kripke $K(\mathcal{R}, State)$ sous-jacente ainsi qu'une formule temporelle LTL ϕ et vérifie si cette structure $K(\mathcal{R}, State)$ satisfait la formule ϕ. Si cette formule n'est pas valide, le LTL model checker retourne un contre-exemple qui montre une séquence d'états menant à la violation de cette formule. Cet outil est implémenté sous la forme d'un module, spécifié en termes de la logique de réécriture, dans le langage Maude. Ce dernier importe d'autres modules prédéfinis (spécifiés également dans le langage Maude) :

• Le module ***LTL-SIMPLIFIER*** implémente des procédures formelles pour simplifier la formule LTL exprimant une propriété.

• Le module ***SATISFACTION*** spécifie la syntaxe et la sémantique de l'opérateur de satisfaction (|=) indiquant si une formule donnée est vraie ou fausse dans un certain état.

• Le module ***SAT-SOLVER*** permet de vérifier la satisfiabilité et la topologie d'une formule spécifiée en logique LTL [Eke02].

Dans ce qui suit, nous trouvons une partie des opérateurs LTL dans la syntaxe de Maude :

```
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.

L'état State est générique. Après avoir spécifié le comportement de son système dans un module système 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*** [Cla16].

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 est affiché. Le contre-exemple concerne l'état dans lequel la formule n'est pas valide :

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

7. Exécution de Maude

Nous pouvons commencer une session avec Maude sous Windows en cliquant sur le fichier **full Maude 2.6**. La principale fenêtre de Maude s'ouvre comme le montre la figure 3.1.

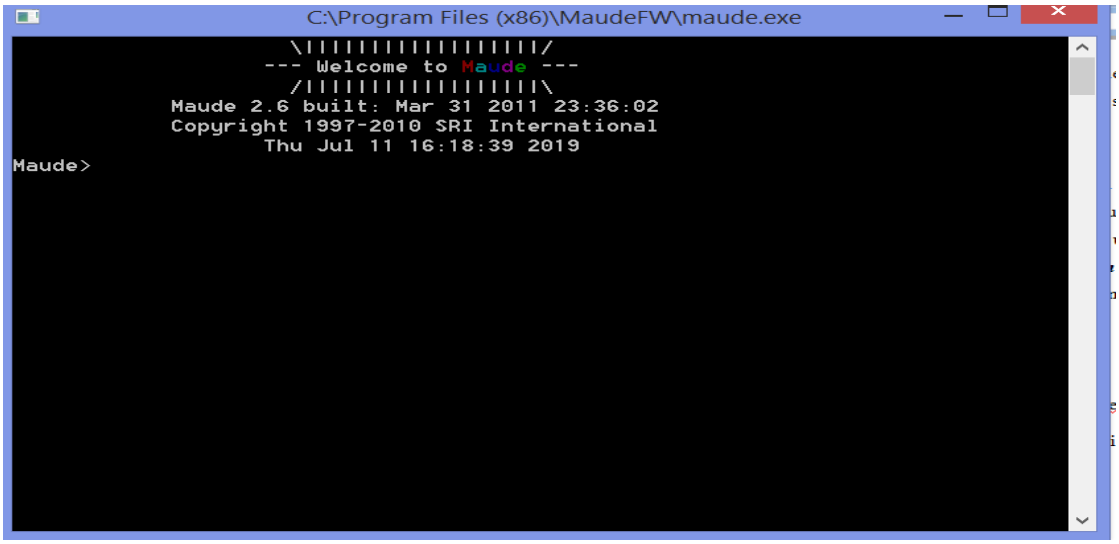


Figure 3.1. Exécution de Maude

Le système Maude est maintenant prêt pour accepter des commandes ou des modules. Durant une session l'utilisateur interagit avec le système en saisissant sa requête au 'Maude prompt'. Par exemple, si on veut quitter Maude

Maude>quit

'***q***' peut être utilisé comme abréviation pour la commande '***quit***'. On peut aussi saisir des modules et utiliser autres commandes. Ce n'est pas vraiment pratique de saisir un module dans le 'prompt', plutôt l'utilisateur peut écrire un ou plusieurs modules dans un fichier texte et faire entrer le fichier dans le système en faisant appel à la commande '***in***' ou la commande '***load***' [Cla16]. Supposant le fichier **my-*nat.maude*** contenant le module NAT décrit comme suit :

```
fmodSIMPLE-NAT is
sort Nat .
opzero : -> Nat .
op s_ : Nat -> Nat .
op _+_ : Nat Nat -> Nat .
vars N M : Nat .
eq zero + N = N .
eq s N + M = s (N + M) .
endfm
```

Nous pouvons l'introduire au système en faisant la commande suivante :

Maude> in my-nat.maude

Après entrer le module NAT nous pouvons par exemple, réduire le terme s szero + s sszero (qui correspond à 2 + 3 dans la notation Peano) dans de tel module. Le lancement et la validation de la commande 'reduce' retourne un résultat comme suit :

Maude> reduce in NAT: s s zero + s sszero.

reduce in NAT: s s zero + s sszero.

rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)

result Nat: s ssss zero

Ce n'est pas nécessaire de donner le nom du module explicitement dans lequel on réduit le terme. Toutes les commandes dont un module a besoin référent au module en cours par défaut, sinon il faut donner explicitement le nom du module. Le module en cours est en général le dernier introduit ou utilisé, ou bien nous pouvons utiliser la commande **reduce**

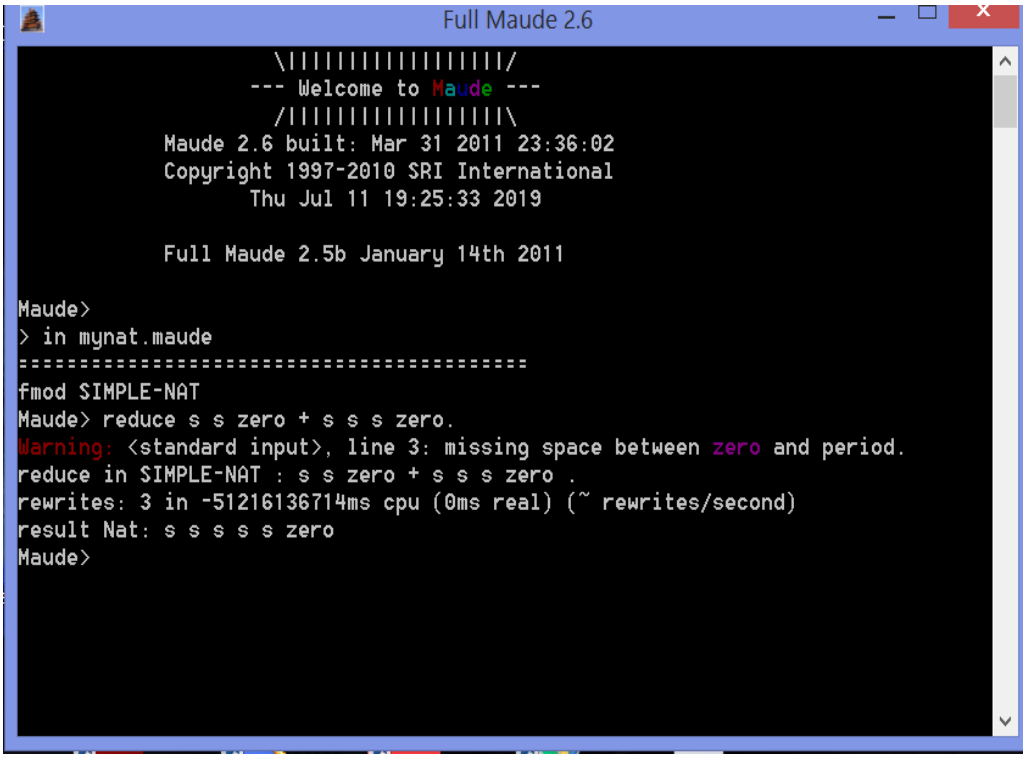


Figure 3.2. Exécution de la commande reduce

8. Conclusion

Dans ce chapitre, nous avons présenté les concepts de base de la logique de réécriture ainsi que le système Maude. Nous avons mis l'accent sur la vérification avec le système Maude et la logique temporelle, qui sont utilisés dans notre travail. Plus précisément nous allons utiliser une extension de la logique temporelle linéaire qui est la logique temporelle temps réel qui sera détaillée dans le prochain chapitre.

Chapitre 04 :

Une approche et un outil de modélisation et de vérification des systèmes embarqués

1. Introduction

Dans le chapitre précédent, nous avons mentionné que MDA est une approche de développement des systèmes complexes dans laquelle les modèles et les transformations de modèles sont au cœur. Cependant, une démarche MDA 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, réputes rigoureux sur l'exhaustivité de la vérification formelle qu'ils permettent d'activer.

Notre travail s'intéresse à l'utilisation des techniques offertes par MDA pour la modélisation du comportement d'un système embarqué.

La modélisation du comportement d'un système embarqué est réalisée par l'association implicite de la notion de temps à un modèle. C-à-d, un modèle de temps logique a été proposé pour enrichir les deux diagrammes d'UML (statecharts et le diagramme de communication) et permettre la description et l'analyse de contraintes temporelles.

Ce modèle de temps consiste en l'utilisation des diagrammes d'états-transitions et de communication temps réel (**RTST** : Real Time Statecharts et **RTC** : Real Time communication diagram).

Une fois un système embarqué est modélisé, la difficulté est de pouvoir exprimer des propriétés temporelles pertinentes et les vérifier formellement avec un outil de vérification tel que model checking de Maude.

Dans ce chapitre, nous présentons notre contribution qui consiste au développement d'un outil basé sur la méta-modélisation et les grammaires de graphes pour la manipulation et la simulation des RTST et RTC en utilisant l'outil AToM^3.

Le principe est de proposer un outil à l'utilisateur, qui lui permet de faire une édition graphique d'un modèle UML illustré par les RTST et RTC, modélisant le (comportement d'un système embarqué et de générer automatiquement via une grammaire de graphes, sa description équivalente dans Maude. Enfin, le model checking du système Maude est appelé pour vérifier certaines propriétés exprimées dans la logique temporelle temps réel (MITL).

2. Comportement temporel d'un système

Dans le monde réel, les opérations des entités physiques consomment beaucoup de temps. Quand elles sont exécutées sur un nombre limité de périphériques, leur durée maximale est appelée (**WCET** : worst case execution time, en français c'est le plus long temps d'exécution) [Sve03].

Les systèmes embarqués doivent satisfaire des contraintes temporelles strictes, qui sont dérivées des systèmes qu'ils contrôlent. En général, le calcul du **WCET** est nécessaire pour montrer la satisfaction de ces contraintes. L'obtention systématique des limites supérieures des temps d'exécution (Deadlines), des programmes pourrait complètement résoudre le problème de l'arrêt du programme.

Des timers (clocks/compteurs) dans les systèmes embarqués sont souvent disponibles. Ils pourront être utilisés pour obtenir des mesures très fines et très précises des temps d'exécutions de petits segments de code. Le temps d'exécution du segment de code est obtenu en calculant la différence entre les valeurs lues dans le clocks au début et à la fin de l'exécution [Cot08].

Typiquement le formalisme des automates temporisés peut servir à modéliser (ou spécifier) un système temps réel. On peut alors vérifier des propriétés sur ce modèle ou générer du code pour l'implanter. Un automate temporisé est un automate étendu avec des horloges modélisant des contraintes et comportements temporels. Initialement toutes les horloges commencent à 0 et augmentent au même rythme**.** Sur une transition, on peut soit tester la valeur d'une (ou de plusieurs) horloge(s) ; soit remettre une (ou plusieurs) horloge(s) à zéro. Les automates temporisés sont implémentés dans les systèmes embarqués par une tâche de contrôle, qui vérifie périodiquement l'activation des transitions. Donc, une transition activée n'est pas déclenchée au point d'activation, mais plutôt pendant la période de la tâche de contrôle. Cela entraîne un retard, causé par la limitation du périphérique [Sve03].

La notion de temps employée dans un système embarqué est généralement limitée par la valeur maximale de l'horloge du RTOS (Real Time Operating System) ou du matériel. Par conséquent, les valeurs d'horloge que nous pouvons comparer dans un programme peuvent toujours être encodées par un ensemble de points dans le temps. Au lieu d'une horloge absolue seules les différences de temps relatives sont utilisées, une extension directe d'un modèle d'automate vers le temps discret existe, où les différentes valeurs d'horloge sont utilisées pour déterminer un comportement temporel. Cependant,

lorsqu'on utilise ces valeurs d'horloge, cela n'inclut généralement pas l'hypothèse que le temps ne s'écoule que par étapes discrètes. Une valeur d'horloge ***n*** est supposée représente l'intervalle **[n, n+1]** plutôt qu'un point discret dans le temps. Par conséquent, la lecture et la comparaison des valeurs d'horloge vérifient si les limites supérieures ou inférieures se maintiennent effectivement.

3. Les diagrammes d'états -transition et de communication temps réel

3.1 Les diagrammes d'états-transitions temps réel

Un diagramme d'état-transition temps réel (**RTST**: Real Time Statechart) est un diagramme d'état-transition simple, étendu par de annotation de temps ,tel que des horloges, des invariants et des gardes de temps [Mir05].

Dans un RTST, à chaque transition **t** est associé un intervalle de temps {L, U} qui représente la valeur minimale et maximale du temps d'exécution de **t**. A chaque état est associée une horloge globale [Kes05].

Un RTST est un quadruplet (**S, E, C, Tr**), où :

- **S** est un ensemble fini d'états ;
- **E** est un ensemble fini d'événements ;
- **C'**est un ensemble fini d'horloges ;
- **Tr =Tr1 U Tr2** est un ensemble fini de transitions**, Tr1** représente les transitions immédiates, qui sont franchies par des entrées, bien que **Tr2** représente les transitions temporelles qui dépendent dans leurs franchissements du temps que des entrées [Sve03].

Il y a une horloge globale **t** dans **C**, qui représente une horloge principale, et a une valeur positive.

- Chaque transition **tr** est un 5-tuplet (**si, e, c, a, sj**), où :
  - **si** est un état d'entrée à une transition.
  - **e** est un événement déclencheur d'une transition.
  - **c** est une contrainte ou une condition sur le franchissement d'une transition**.**
  - Une transition **tr** est devenue franchissable lorsqu'un événement **e** se produit, elle comprend une contrainte fonctionnelle (**cf)** et une contrainte de temps (**ct**). En particulier, pour chaque transition **tr ∈Tr2**, **ct** appartient à un intervalle de nombres réels **[L, U]**, qui spécifient les retards minimaux (**L**) et maximaux (**U**) de la transition et satisfont **L ≤ U** ; tandis que pour chaque transition **tr ∈ Tr1, ct = 0** ;

- **a** est l'action exécutée lors du franchissement de la transition. L'action **a** a un temps d'exécution **w<=** valeur d'horloge globale ;
- **sj** est un état de sortie pour la transition **tr**.
- Si une transition **tr ∈ Tr1** est activée par une horloge globale **t ∈ C**, elle doit être franchie à l'instant **t**.
- Si une transition **tr ∈ Tr2** est activée à l'horloge globale **t ∈ C** et que son intervalle de temps est **[L, U]**, elle doit être franchie dans le temps **[L + t, U + t]**.

▪ Chaque état **s** est un 5-tuple (**n, gc, entry, do, exit**), où :

- **n** est le nom de l'état
- **gc** est une horloge globale ∈ C pour l'état, chaque état a un invariant temporel **t**, où **gc<= t**.
- **entry** est l'action d'entrée dans l'état qui a un worst case time **WCET <= t**
- **do** est une action exécutée dans l'état et a un worst case time **WCET <= t** ou un intervalle de temps
- **exit** est une action de sortie de l'état qui a un worst case time **WCET** <= t
- L'horloge globale **gc** peut être réinitialisée lors de l'entrée et de la sortie des états.
- Comme l'opération **do** est exécutée périodiquement, un intervalle est spécifié à partir duquel une période fixe est choisie [Sve03].

3.2 Les diagrammes de collaboration temps réel

Un diagramme de collaboration temps réel (**RTC** : Real Time Collaboration diagram) est un diagramme de collaboration simple, étendu par de annotation de temps, tel que des horloges.

Un RTC est une collection d'objets nommés avec des liens les reliant. Formellement un RTC est un triplet (**CO, CL, C**), où :

- **CO** est un ensemble fini d'objets de collaboration ;
- **CL** est un ensemble fini de liens de collaboration ;
- **C**'est un ensemble fini d'horloges.

▪ Chaque objet de collaboration **co** est un quadruple **(n, Te, Tt, Tr)**, où :

- **n** est le nom de l'objet.
- **Te** est le temps d'envoi d'un message qui doit être <= à une horloge globale ∈ **C**.

- **Tt** est un temps de transmission d'un message entre deux objets, qui doit être <= à une horloge globale ∈ **C**.

- **Tr** est un temps de réception d'un message qui doit être <= à une horloge globale ∈ **C**.

▪ Chaque lien de collaboration **cl** est un 2-tuple (**event, Ttr**), où :

- **event** est une liste de messages qui sont échangés entre deux objets, chaque message a un ordre et un nom.

- **Ttr** est un temps de transmission global d'un message qui doit être <= à une horloge globale ∈ C [Kes05].

4. Méta-modélisation des RTSTs et RTCs

Pour développer un outil de modélisation des systèmes embarques avec les RTCs et les RTSTs avec ATOM^3 construire, il faut définir deux méta-modèles pour les deux diagrammes.

Le méta-formalisme utilisé dans l'étape de méta-modélisation est le modèle de diagrammes de classes d'UML et les contraintes sont exprimées en code Python.

4.1 Méta-modèle des RTSTs

Pour méta-modéliser les RTSTs dans l'outil AToM^3, nous avons définis cinq classes reliées par huit associations comme le montre la Figure 4.1. Ce méta-modèle permet de spécifier les attributs, les contraintes, les relations, ainsi que l'apparence graphique.

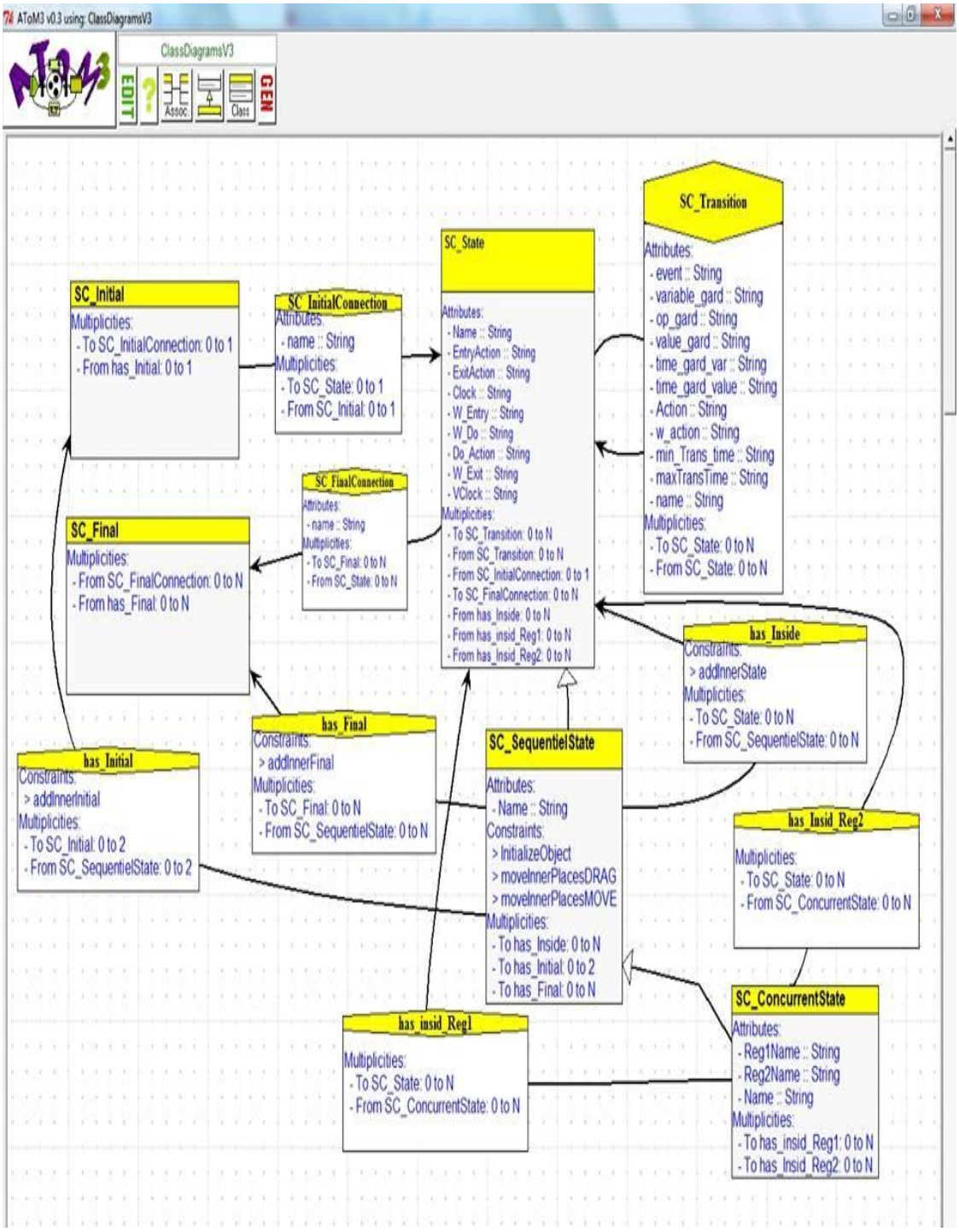


Figure 4.1: Méta-modèle des RTSTs

4.1.1 Les associations

Les associations contenues dans le méta-modèle sont :

- **SC_InitialConnection** : représente une transition entre un état initial et un état simple ou composite. Elle possède un seul attribut : Name, qui représente le nom d'une transition.

▪ **SC_Transition** : représente une transition entre deux états. Elle possède douze attributs, Nous citons ici quelques attributs :

- name : le nom d'une transition.
- event : l'évènement qui déclenche la transition.
- min_Trans_time et maxTransTime : la durée maximale et minimale d'une transition [min_Trans_time, maxTransTime].
- Time_gard_value :le temps d'évaluation de garde d'une transition.

▪ **has_Initial** : cette association permet de représenter l'appartenance d'un état initial à un état composite concurrent ou séquentiel.

▪ **has_Final** : cette association représente l'appartenance d'un état final à un état composite concurrent ou séquentiel.

▪ **has_Inside** : cette association permettant de représenter l'appartenance d'un état simple à un état composite concurrent ou séquentiel.

▪ **has_Inside_Reg1** : permettant d'illustrer l'appartenance d'un état simple à la première région d'un état concurrent.

▪ **has_Inside_Reg2** : permettant d'illustrer l'appartenance d'un état simple à la deuxième région d'un état concurrent.

▪ **SC_FinalConnection** : représente une transition entre un état simple ou composite et un état final. Elle possède un seul attribut : Name qui est le nom d'une transition.

4.1.2 Les classes

Les classes contenues dans le méta-modèle sont :

▪ **SC_Initial** : représente le début d'un diagramme RTST (état initial).

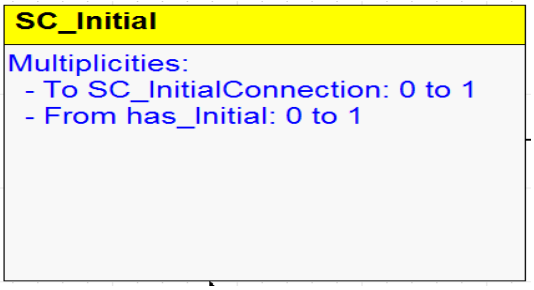


Figure 4.2: La classe SC_Initial.

▪ **SC_State** : représente un état simple dans un RTST. Elle possède neuf attributs :

- Name : le nom d'état.

- EntryAction : l'activité d'entré dans un état.
- ExitAction : l'activité de sortie d'un état.
- Do_Action : l'activité exécutée dans un état.
- Clock : c'est le nom d'une horloge associée à unétat.
- VClock : le temps consommé par un état.
- W_Entry : le WCET de l'action "Entry".
- W_Do : le WCET de l'activité "Do".
- W_ Exit : le WCET de l'action "Exit".

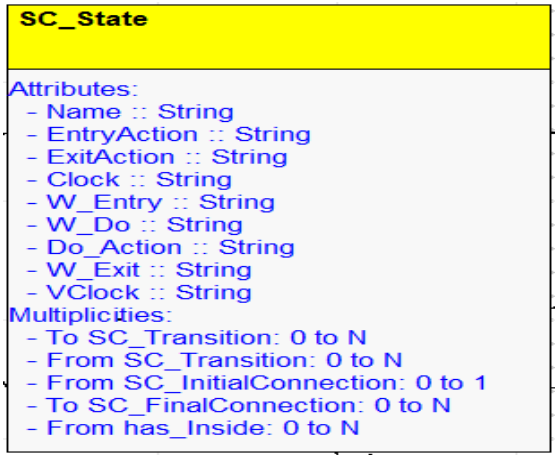


Figure 4.3: La classe SC_State .

▪ **SC_SequentielState** : représente un état Séquentiel (composite). Elle possède un seul attribut Name, qui est le nom d'un état Séquentiel.

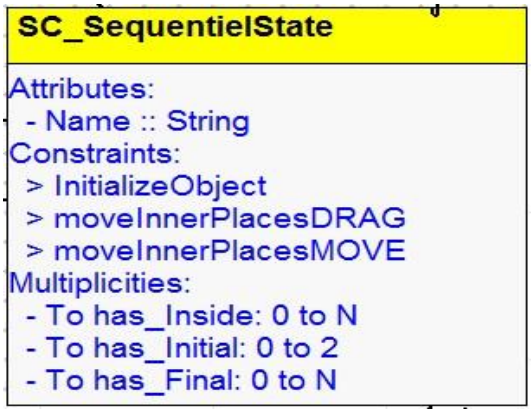


Figure 4.4: La classe SC_SequentielState .

▪ **SC_ConcurrentState** : représente un état Concurrent (orthogonal) contenant deux régions, chaque région représente un flot d'exécution. Cet état possède trois attributs :

- Name : le nom d'état Concurrent.

- Reg1Name : le nom de la première région.
- Reg2Name : le nom de la deuxième région.

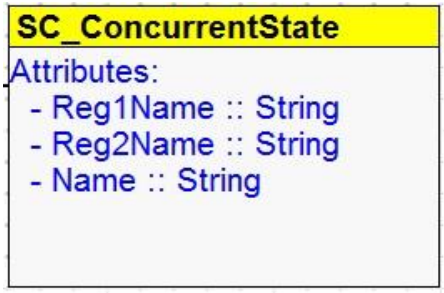


Figure 4.5: La classe SC_SequentielState .

- **SC_Final** : représente l'état final d'un RTST.

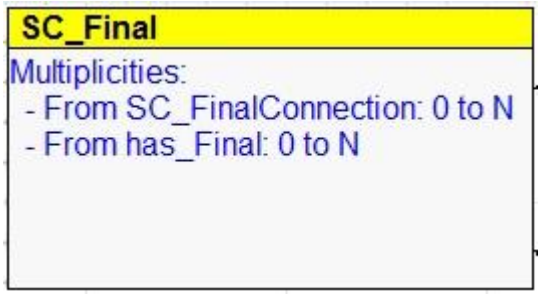


Figure 4.6: La classe SC_Final.

Le méta-modèle de la figure 4.1, nous a permis de générer sous ATOM^3 un outil de modélisation assurant la manipulation des RTSTs .

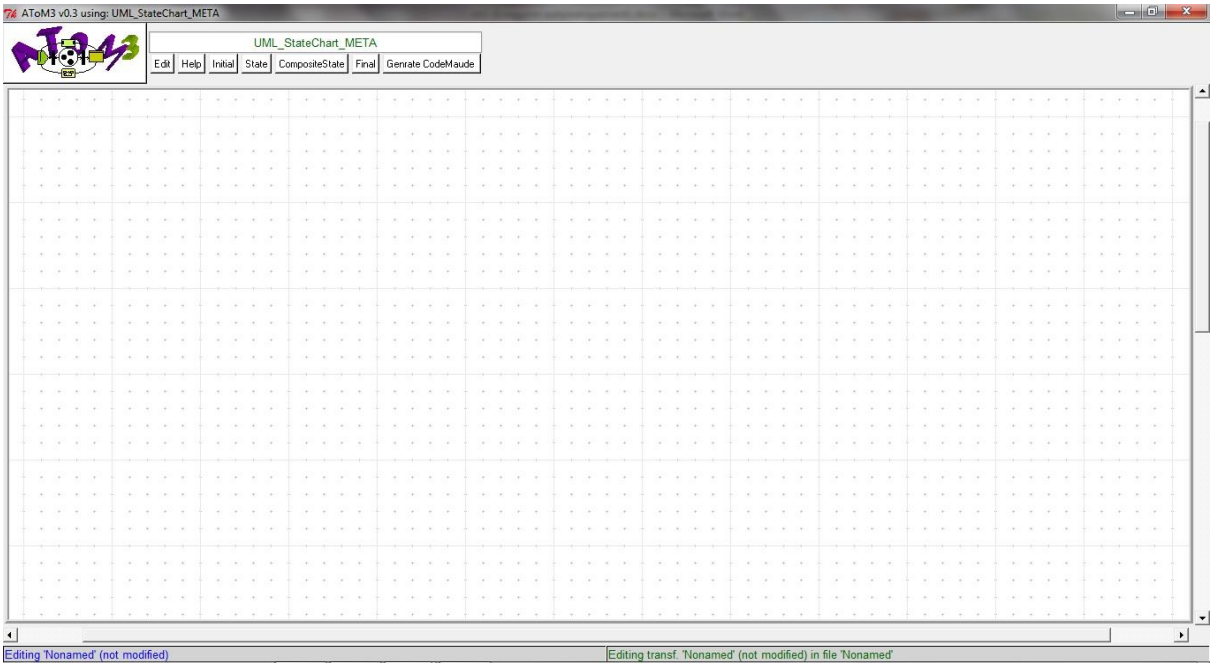


Figure 4.7: Outil de manipulation des RTSTs.

La figure 4.8 illustre l'apparence graphique des éléments d'un diagramme RTST. De gauche vers la droite nous trouvons : un état initial, un état simple, un état composite séquentiel, un état composite concurrent et un état final.

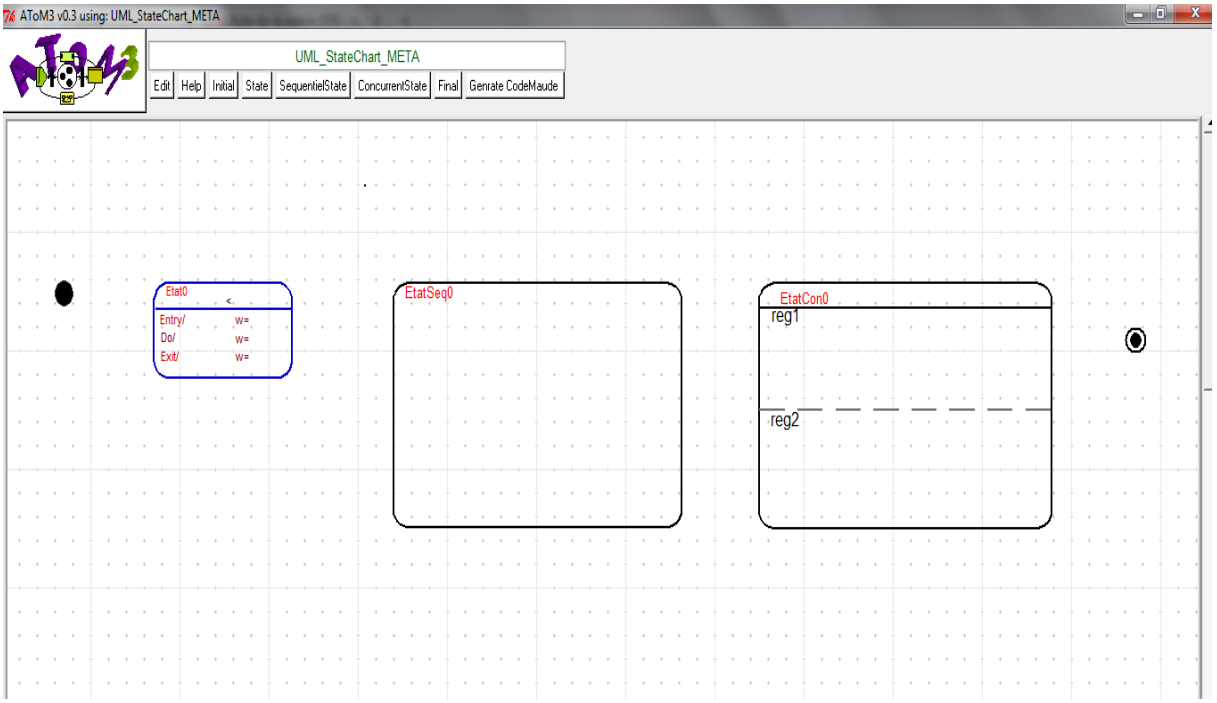


Figure 4.8: Representation graphique des éléments d'un diagrammeRTST.

4.2 Méta-modèle des RTCs

Pour méta-modéliser les RTCs dans l'outil AToM^3, nous avons définis deux classes reliées par une association comme le montre la Figure 4.9. Ce méta-modèle permet de spécifier les attributs, les contraintes, les relations, ainsi que l'apparence graphique.

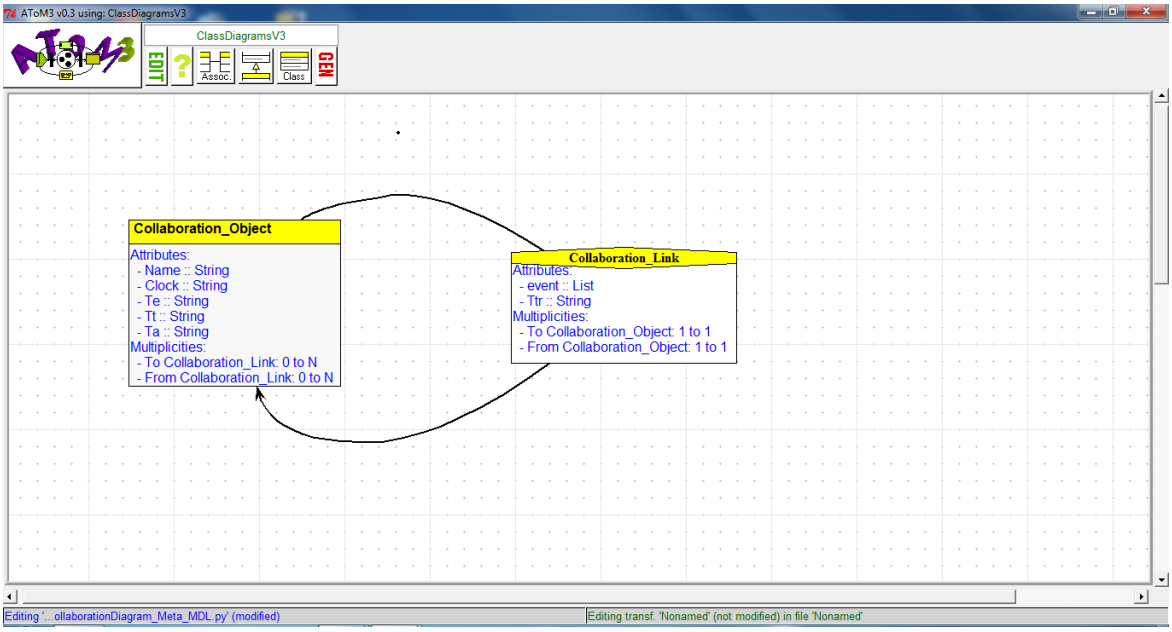


Figure 4.9: Méta-modèle des RTCs

Le méta_modèle est composé de :

- **Collaboration_Object** : Une méta-classe pour les objets, chaque objet a cinq attributs
  - **Name :** représente le nom d'un objet du diagramme
  - **Clock:** représente une horloge globale .
  - **Ta :** représente le temps de réception d'un message.
  - **Te :** représente le temps d'envoi d'un message.
  - **Tt :** représente le temps de transmissioni d'un message.
- **Collaboration_Link**:Une méta-classe pour les communications entre les objets . Chaque lien a deux attributs
  - **event**. Représente la liste des messages échangés, chaque message est composé de deux attributs :
    - **order:** représente l 'ordre d'un message
    - **name :** représente le nom d'un messge .
    - **Ttr :** représente le temps total d'envoi d'un message .

A partir de ce méta-modèle, ATOM^3 génère un outil de manipulation des RTCs. Cet outil est illustré dans la figure suivante :

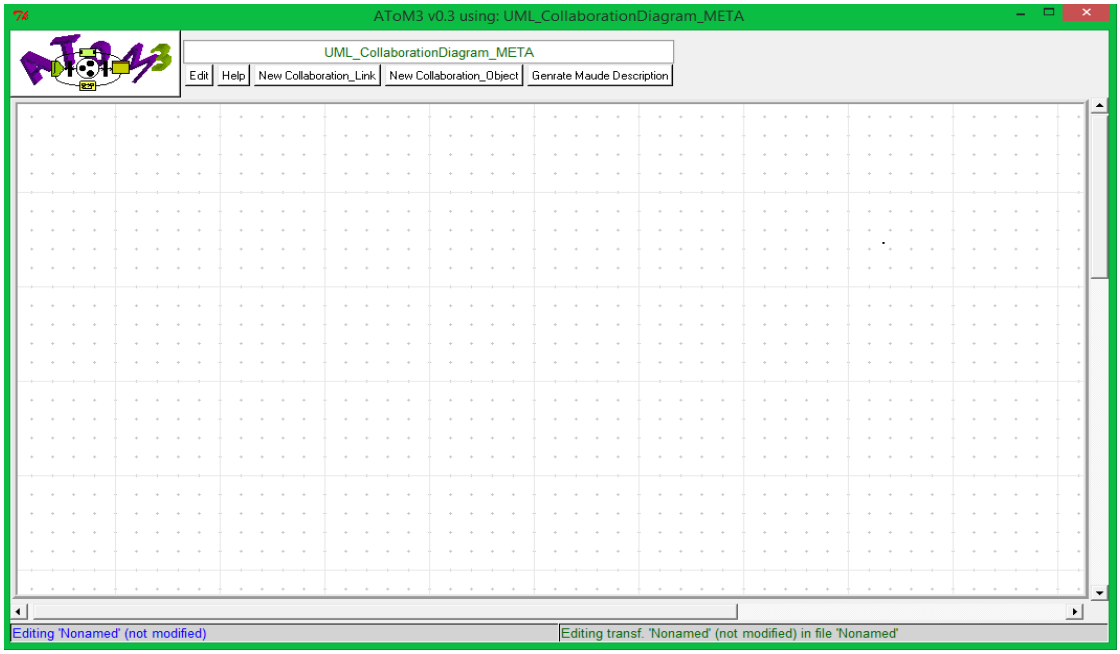


Figure 4.10 Outil de manipulation des RTCs

5. Génération du code Maude équivalent

Pour générer automatiquement une spécification en langage Maude équivalente aux deux diagrammes RTC et RTST respectivement, nous avons définis une grammaire de quarante règles qui seront exécutées dans un ordre ascendant de leurs priorités par le système de réécriture de graphes de l'outil AToM^3.

Lors de l'application de cette grammaire sur les deux diagrammes, un fichier d'extension " *.maude"* qui contient une description textuelle équivalente en Maude sera généré.

Dans notre travail l'application de la grammaire va parcourir les deux diagrammes sans les changer, en générant la spécification Maude équivalente. Cela signifie que dans toutes les règles de notre grammaire, la partie gauche est identique à la partie droite.

5.1 La grammaire de transformation

La grammaire que nous avons définie est composée de quarante règles en plus d'une action initiale et une action finale

- L'action initiale qui contient la création du fichier Maude en plus de l'initialisation des variables temporaires utilisées dans les règles. La figure 4.11 illustre une partie de l'action initiale.

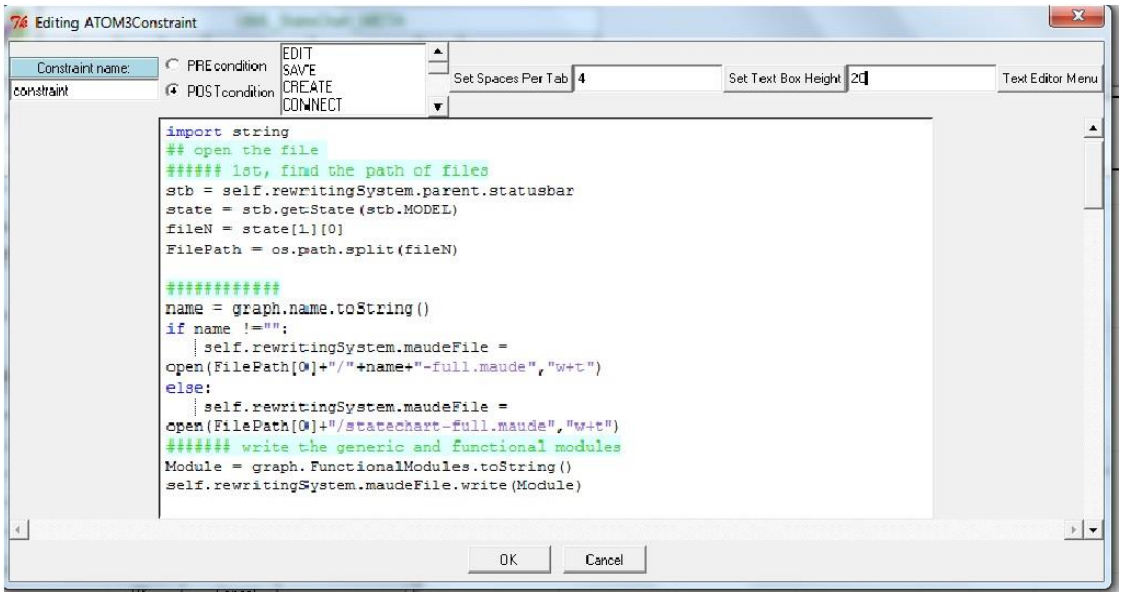


Figure 4.11 Action finale de la grammaire.

➢ L'action finale contient la fermeture du fichier crée ainsi que la libération des variables temporaires utilisées. La figure 4.12 illustre une partie de l'action initiale.

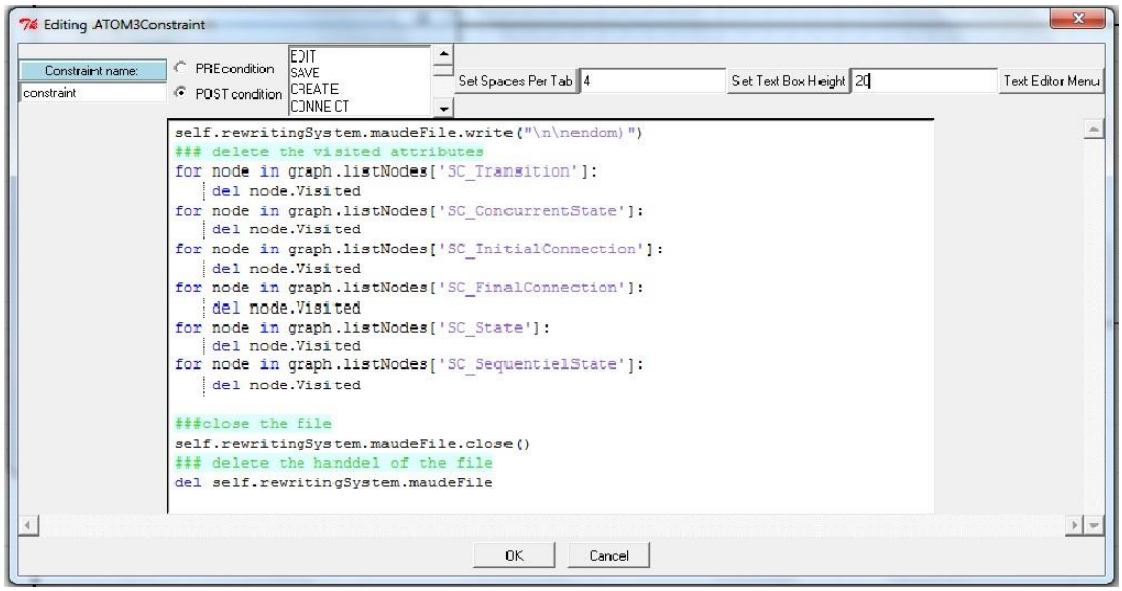


Figure 4.12 Action finale de la grammaire

➢ Chaque règle de la grammaire. Contient deux parties gauche et droite et est caractérisée par un nom et une priorité d'exécution. Dans ce manuscrit, nous allons présenter les règles principales.

- **SimpleState(priorité 1)**

Cette règle est appliquée pour localiser un état simple non encore traité, qui sera ensuite marqué comme visité.

Pour appliquer cette règle, il faut vérifier que l'état n'est pas traité auparavant. Pour cela nous avons définis une condition pour cette règle :

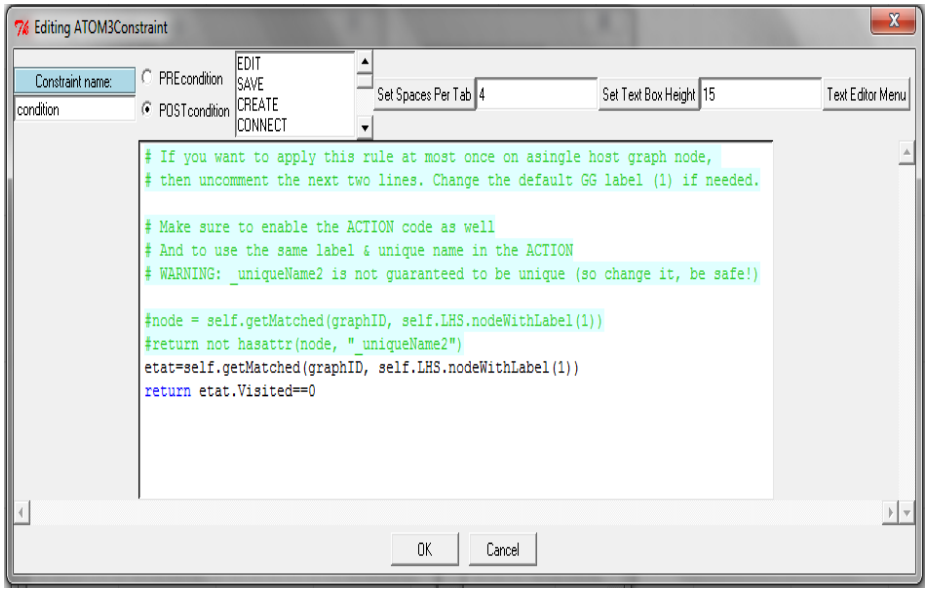


Figure 4.13 : Condition d'application de la règle SimpleState.

Après l'application de la règle, une action est exécutée permettant de générer le code Maude correspondant à tous les états simples et marquer tous les termes de temps correspondants.

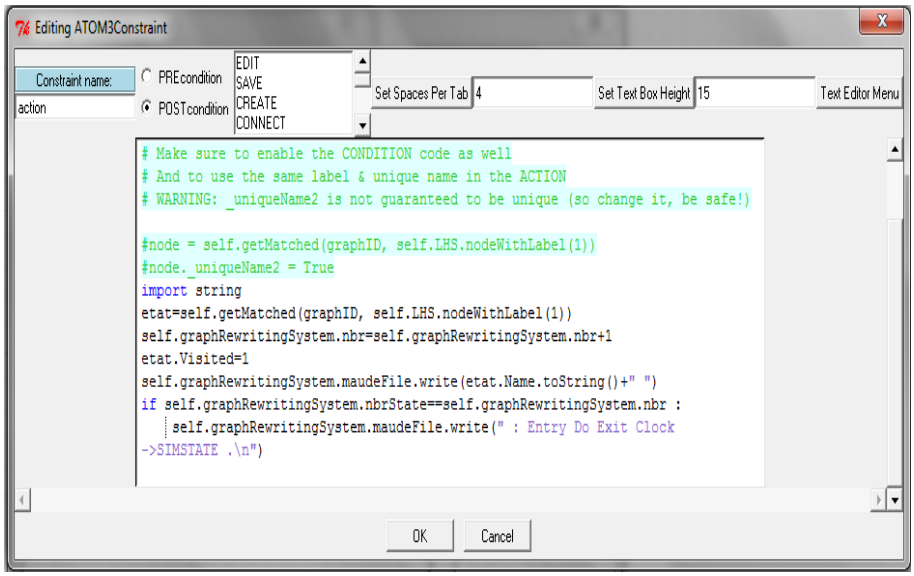


Figure 4.14: Action de la règle SimpleState.

- **Con_SimpleToFinal_R1 (priorité 10)**

Cette règle permet de marquer les transitions qui existent entre un état simple et un état final dans un état composite concurrent (orthogonal), exactement dans la première région de l'état.

Pour appliquer cette règle, il faut vérifier que la transition n'est pas traitée auparavant. Pour cela nous avons définis la condition suivante :

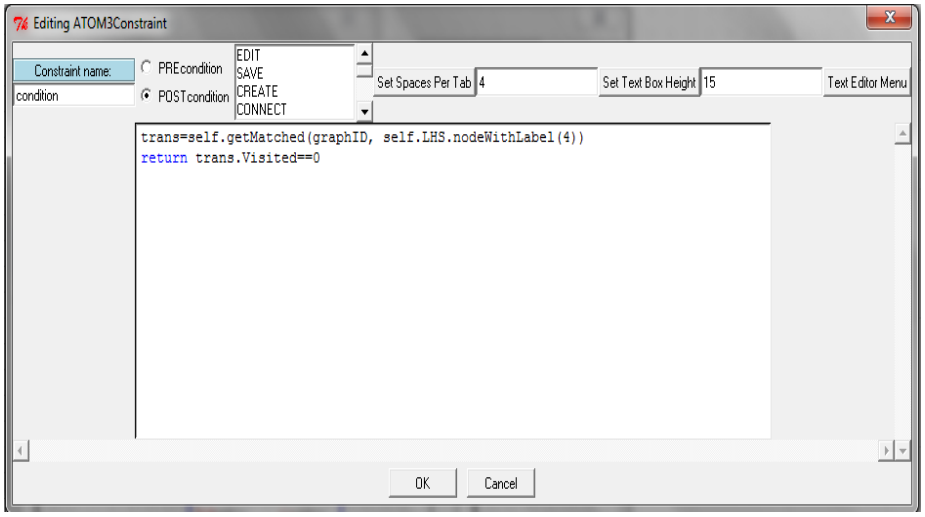


Figure 4.15: Condition d'application de la règle Con_SimpleToFinal_R1.

Une fois que la règle est appliquée, une règle de réécriture est générée dans le fichier Maude.

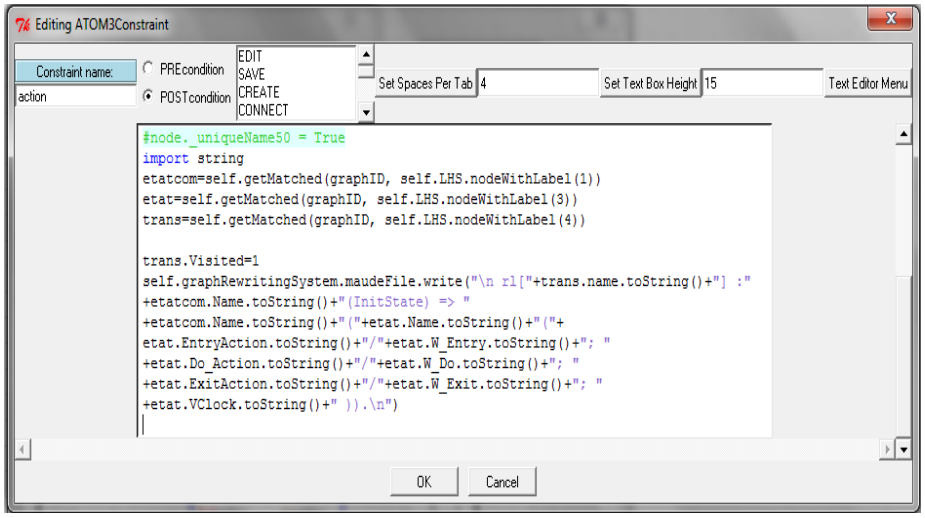


Figure 4.16: Action de la règle Con_SimpleToFinal_R1.

- **SimpleToSimple (priorité 22)**

Cette règle est appliquée pour marquer les transitions entre deux états simples.

Pour appliquer cette règle, il faut vérifier que la transition n'est pas traitée auparavant pour générer la règle de réécriture équivalente en Maude. Pour cela nous avons définis une condition pour cette règle :

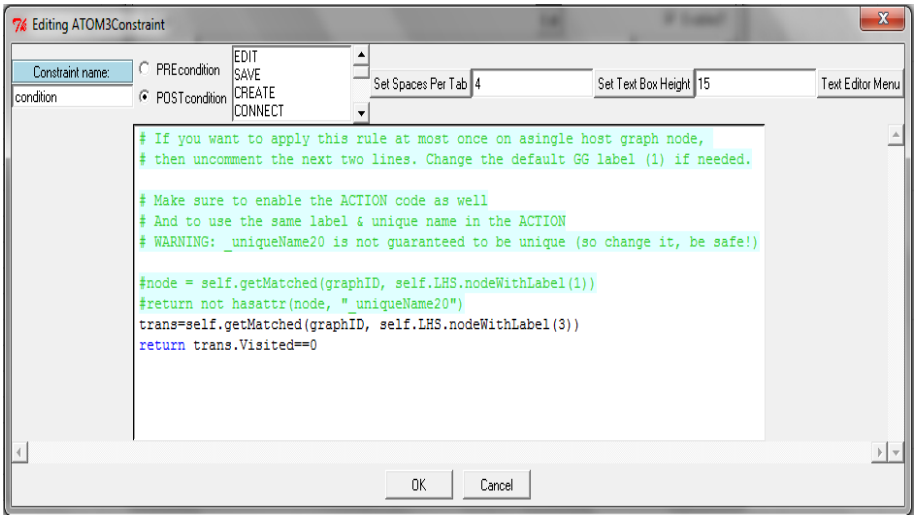


Figure 4.16: Condition d'application de la règle SimpleToSimple.

Après l'application de la règle, une action est exécutée permettant de générer tous les transitions entre les états simples dans le fichier Maude.

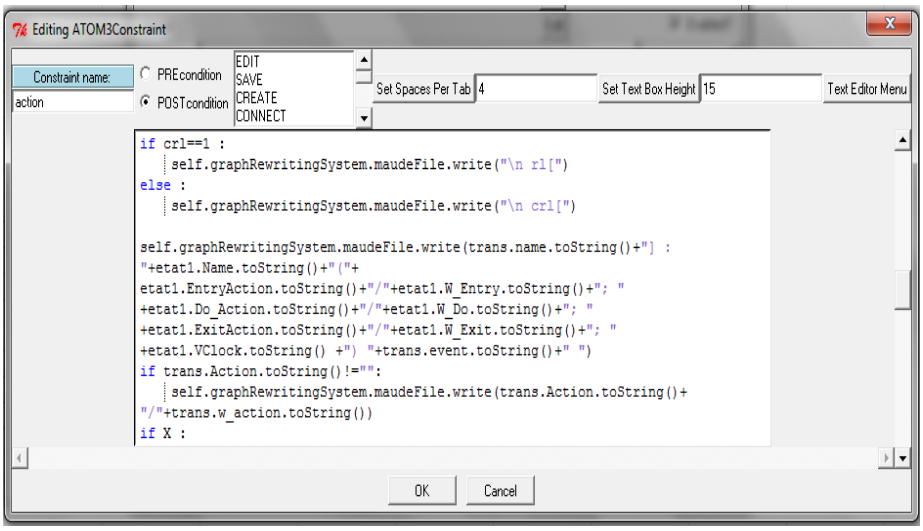


Figure 4.17: Action de la règle SimpleToSimple.

- **ConcurrentToFinal (priorité 35)**

Cette règle est appliquée pour marquer les transitions entre un état concurrent et un état final.

La condition d'application de cette règle, est qu'il faut vérifier que la transition n'est pas traitée auparavant.

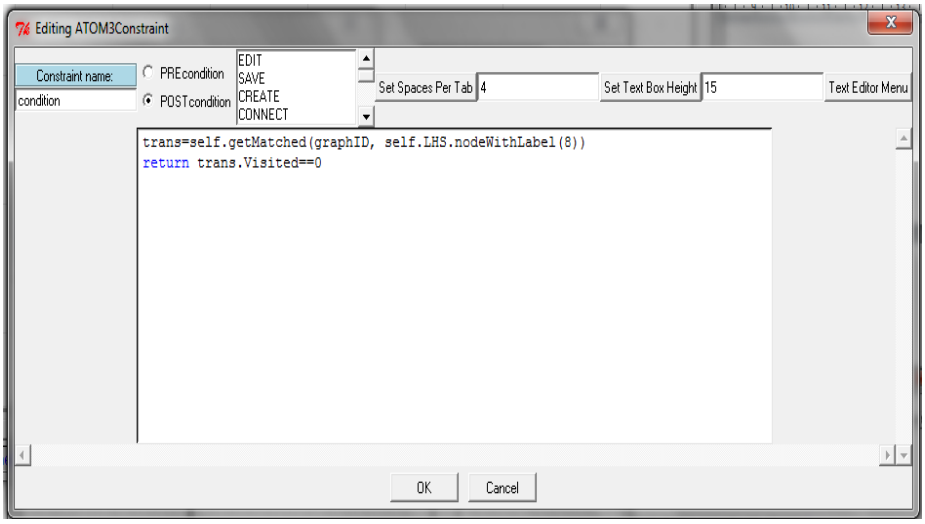


Figure 4.18: Condition d'application de la règle ConcerrentToFinal.

L'action correspondante à cette règle permet de :

- Générer tous les transitions entre les états concurrents et finaux dans le fichier Maude.

- Marquer que ces transitions sont traitées (visitées).

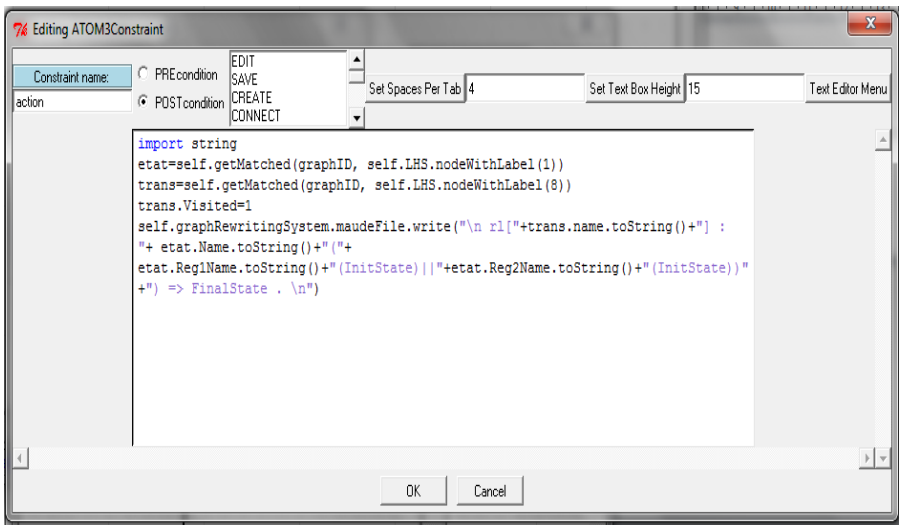


Figure 4.19: Action de la règle ConcerrentToFinal.

- **Objects2Maude (priorité 36)**

Cette règle permet de localiser un objet du diagramme de collaboration non encore traité, qui sera ensuite marqué comme Visité.

Pour l'application de cette règle, il faut vérifier que l'objet n'est pas traité auparavant.

Une fois que la règle est appliquée, son action permet de générer tous les objets du diagramme dans le fichier Maude.

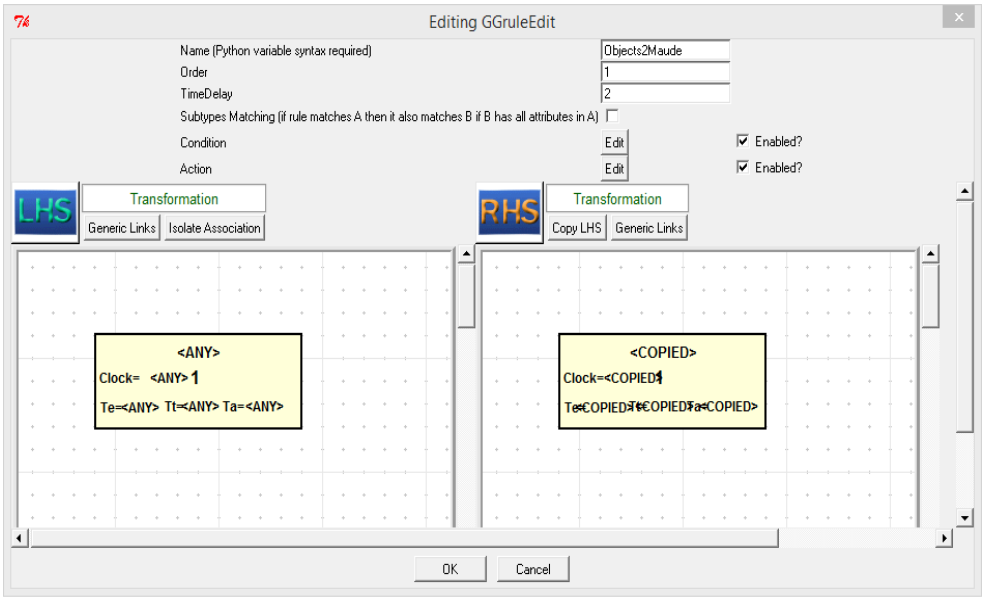


Figure 4.20: La règle Objects2Maude.

- **Link2Maude (priorité 37)**

Pour l'application de cette règle, il faut vérifier que le lien entre deux objets différents n'est pas traité.

L'application de la règle permet degénérer tous les liens de collaboration entre deux objets diffèrent dans le fichier Maude.

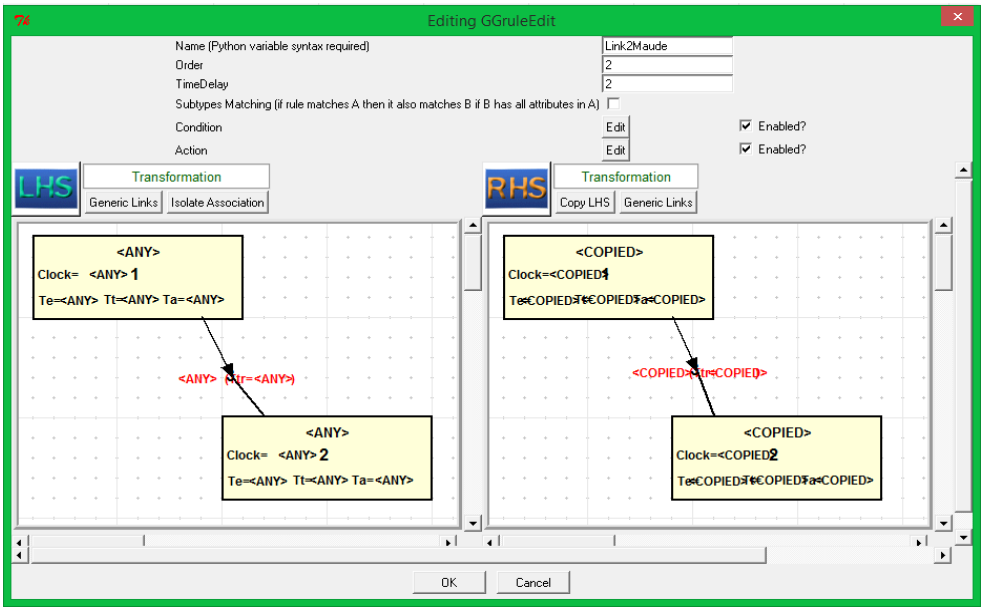


Figure 4.21 La règle Link2Maude

- **Linkboucle_2Maude**

Cette règle est appliquée pour localiser les liens formant une boucle.

L'application de la règle permet de générer le code Maude pour tous les liens formant des boucles.

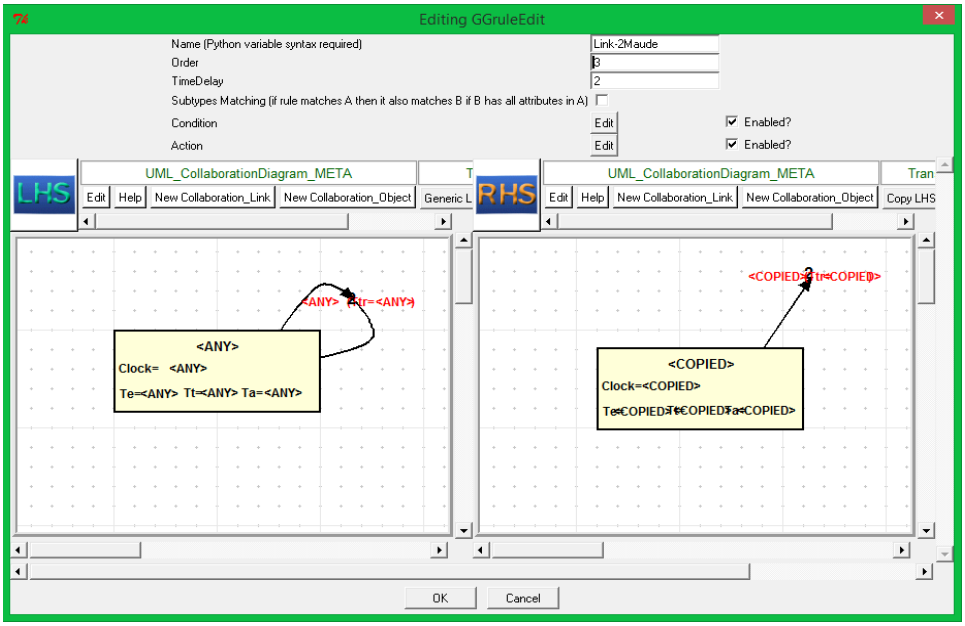


Figure 4.22 La règle Linkboucle2Maude

Pour exécuter la grammaire, nous avons ajouté un bouton « GenerateCodeMaude » dans l'interface utilisateur d'ATOM^3.

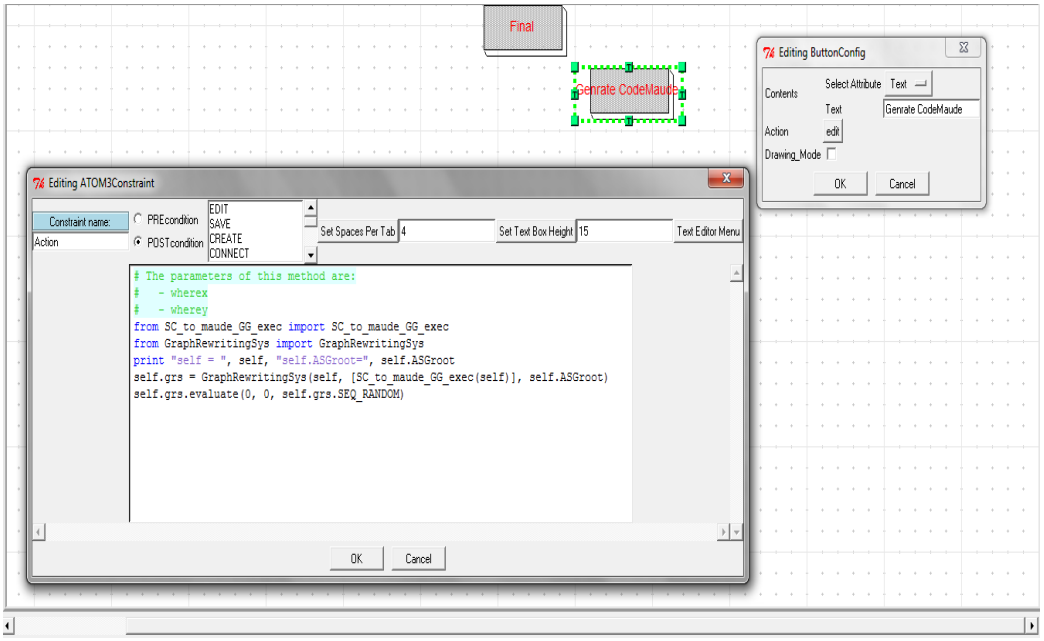


Figure 4.23: Bouton « GenerateCodeMaude » et son action.

6. Vérification des propriétés temporelles d'un système embarqué

Pour procéder à la vérification de certaines propriétés temporelles dans un système embarqué, nous devons passer par quatre étapes :

- La modélisation du système par les deux diagrammes RTST et RTC.
- La génération d'un code Maude depuis les deux diagrammes
- La définition d'une propriété temporelle dans la logique temporelle LTL
- La vérification de cette propriété par Model checking du Maude.

Pour bien illustrer cette vérification, nous allons appliquer les quatre étapes sur un exemple.

L'exemple du système embarqué concerne une montre à cadran numérique avec trois boutons, comme le montre la figure 4.24

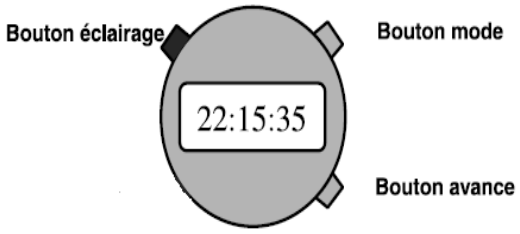


Figure 4.24: Montre à cadran numérique simplifiée.

- Le mode initial est le mode « Affichage ». Quand nous appuyons une fois sur le bouton mode, la montre passe en « modification de l'heure ». Chaque pression sur le bouton incrémente l'heure d'une unité.
- Quand nous appuyons à nouveau sur le bouton mode, la montre passe en « modification minute ». Chaque pression sur le bouton incrémente les minutes d'une unité.
- Quand nous appuyons à nouveau sur le bouton mode, la montre passe en mode « Affichage ».

Il y a ainsi un bouton éclairage, en le pressant nous éclairons le cadran de la montre, jusqu'à ce que nous le relâchions.

Nous sommes clairement en présence de deux comportements concurrents :

- La gestion de l'affichage.
- La gestion de l'éclairage.

6.1 Modélisation avec ATOM^3

La figure 4.25 montre le RTST et Le RTC qui concerne l'exemple de la montre. Le RTST contient deux régions concurrentes.

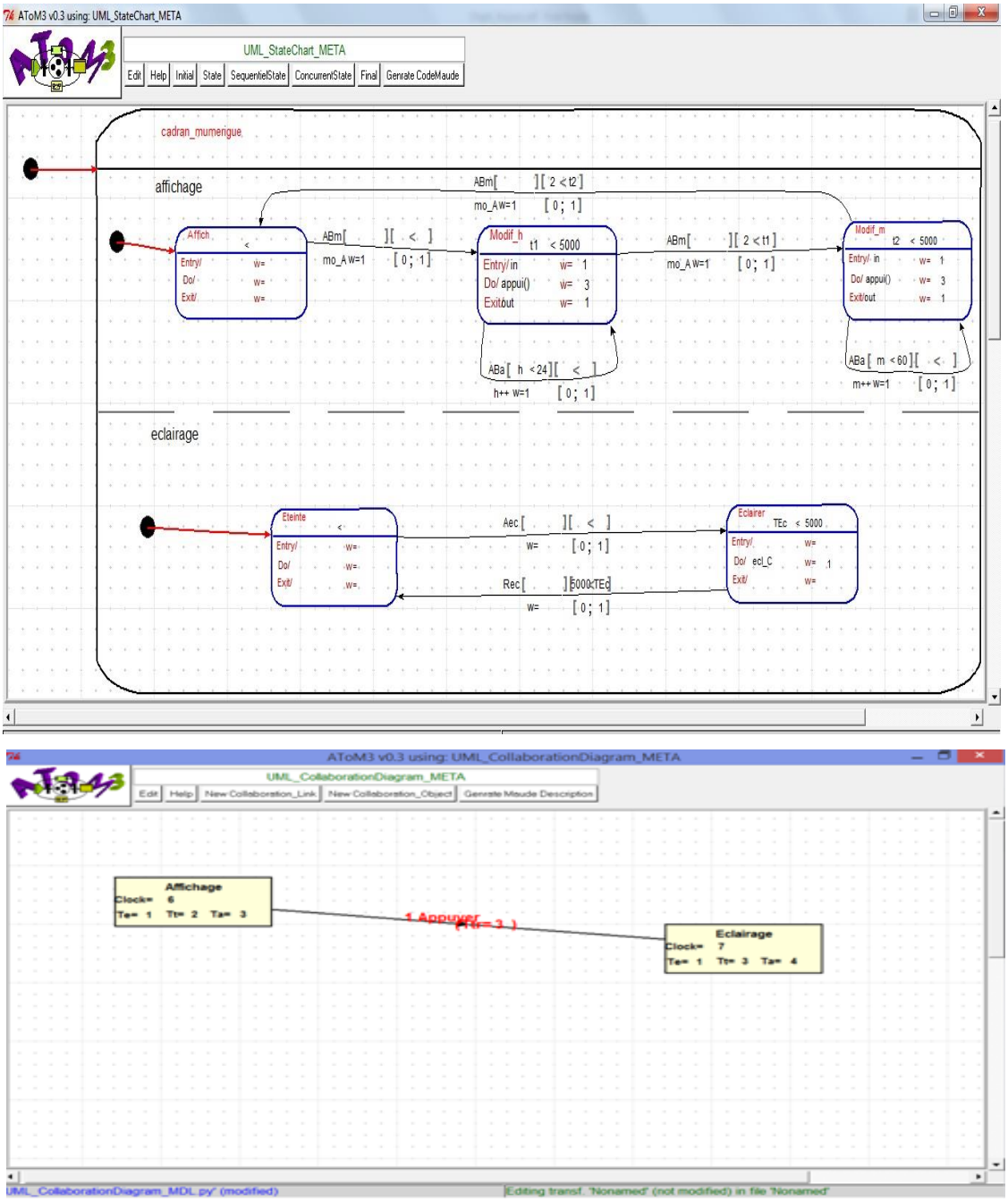
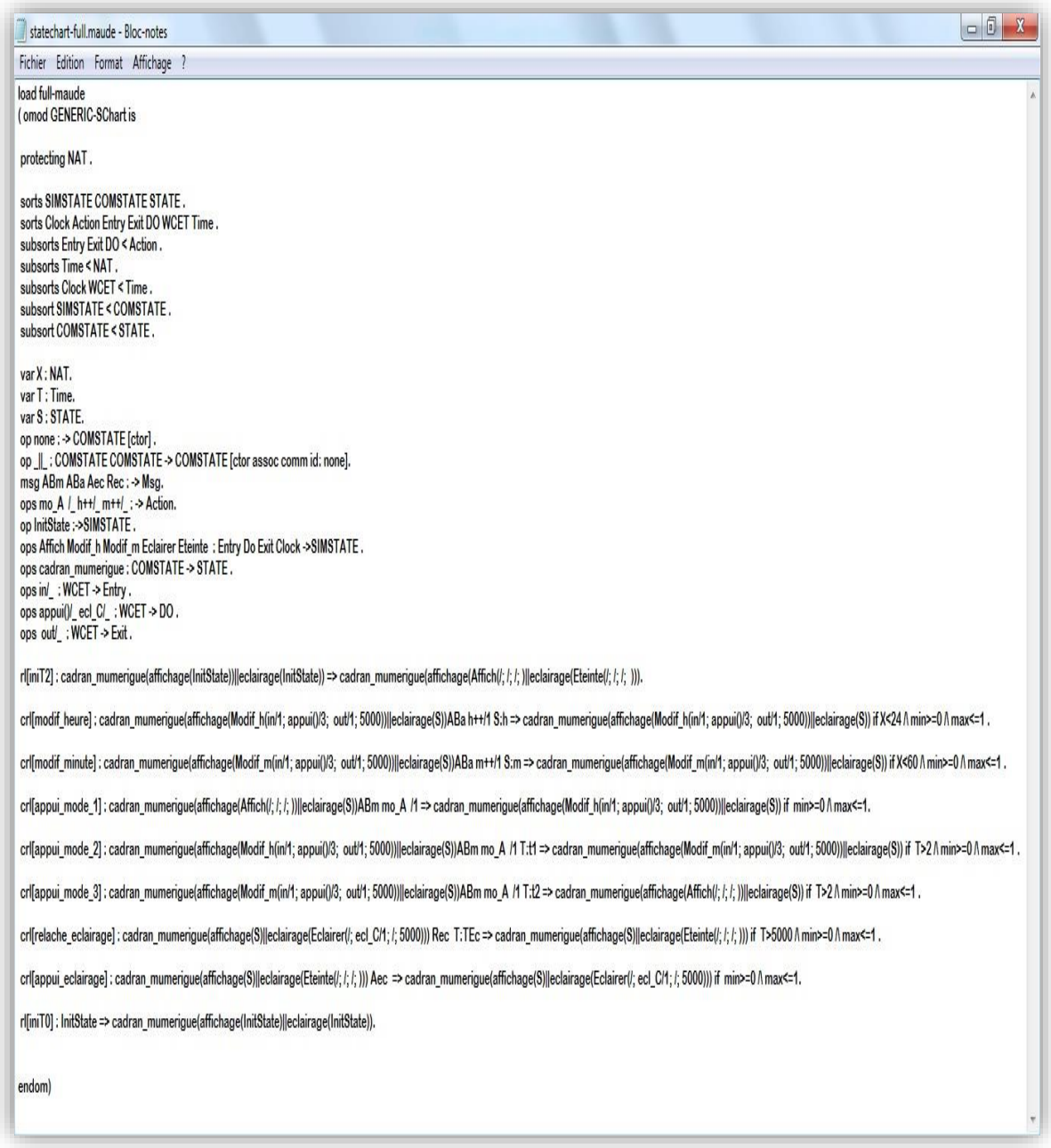


Figure 4.25: Modèles d'une partie de la Montre à cadran numérique.

6.2 Génération du code en Langage Maude

La deuxième étape dans la vérification est la génération d'un code en langage Maude à partir des modèles présentés dans la figure 4.25. Le code généré est obtenu par l'application de la grammaire de transformation. Après l'exécution de cette grammaire, nous avons obtenu le fichier "SChart.Maude" de la Figure 4.26.



statechart-full.maude - Bloc-notes

Fichier Edition Format Affichage ?

```
load full-maude
( omod GENERIC-SChart is

protecting NAT .

sorts SIMSTATE COMSTATE STATE .
sorts Clock Action Entry Exit DO WCET Time .
subsorts Entry Exit DO < Action .
subsorts Time < NAT .
subsorts Clock WCET < Time .
subsort SIMSTATE < COMSTATE .
subsort COMSTATE < STATE .

var X : NAT.
var T : Time.
var S : STATE.
op none : -> COMSTATE [ctor] .
op _||_ : COMSTATE COMSTATE -> COMSTATE [ctor assoc comm id: none].
msg ABm ABa Aec Rec : -> Msg.
ops mo_A /_h++/_m++/_ : -> Action.
op InitState :->SIMSTATE .
ops Affich Modif_h Modif_m Eclairer Eteinte : Entry Do Exit Clock ->SIMSTATE .
ops cadran_mumerigue : COMSTATE -> STATE .
ops in/_ : WCET -> Entry .
ops appui()/_ ecl_C/_ : WCET -> DO .
ops out/_ : WCET -> Exit .

rl[iniT2] : cadran_mumerigue(affichage(InitState)||eclairage(InitState)) => cadran_mumerigue(affichage(Affich(/; /; /; ))||eclairage(Eteinte(/; /; /; ))).

crl[modif_heure] : cadran_mumerigue(affichage(Modif_h(in/1; appui()/3; out/1; 5000))||eclairage(S))ABa h++/1 S:h => cadran_mumerigue(affichage(Modif_h(in/1; appui()/3; out/1; 5000))||eclairage(S)) if X<24 /\ min>=0 /\ max<=1 .

crl[modif_minute] : cadran_mumerigue(affichage(Modif_m(in/1; appui()/3; out/1; 5000))||eclairage(S))ABa m++/1 S:m => cadran_mumerigue(affichage(Modif_m(in/1; appui()/3; out/1; 5000))||eclairage(S)) if X<60 /\ min>=0 /\ max<=1 .

crl[appui_mode_1] : cadran_mumerigue(affichage(Affich(/; /; /; ))||eclairage(S))ABm mo_A /1 => cadran_mumerigue(affichage(Modif_h(in/1; appui()/3; out/1; 5000))||eclairage(S)) if min>=0 /\ max<=1.

crl[appui_mode_2] : cadran_mumerigue(affichage(Modif_h(in/1; appui()/3; out/1; 5000))||eclairage(S))ABm mo_A /1 T:t1 => cadran_mumerigue(affichage(Modif_m(in/1; appui()/3; out/1; 5000))||eclairage(S)) if T>2 /\ min>=0 /\ max<=1 .

crl[appui_mode_3] : cadran_mumerigue(affichage(Modif_m(in/1; appui()/3; out/1; 5000))||eclairage(S))ABm mo_A /1 T:t2 => cadran_mumerigue(affichage(Affich(/; /; /; ))||eclairage(S)) if T>2 /\ min>=0 /\ max<=1 .

crl[relache_eclairage] : cadran_mumerigue(affichage(S)||eclairage(Eclairer(/; ecl_C/1; /; 5000))) Rec T:TEc => cadran_mumerigue(affichage(S)||eclairage(Eteinte(/; /; /; ))) if T>5000 /\ min>=0 /\ max<=1 .

crl[appui_eclairage] : cadran_mumerigue(affichage(S)||eclairage(Eteinte(/; /; /; ))) Aec => cadran_mumerigue(affichage(S)||eclairage(Eclairer(/; ecl_C/1; /; 5000))) if min>=0 /\ max<=1.

rl[iniT0] : InitState => cadran_mumerigue(affichage(InitState)||eclairage(InitState)).


endom)
```

Figure 4.26: Le code Maude généré

6.3 Vérification d'une propriété temporelle

Model Checking est un ensemble de techniques pour vérifier automatiquement des propriétés temporelles relatives au comportement des systèmes.

Le Model Checking permet de vérifier la satisfiabilité d'une propriété donnée dans un état ou un ensemble d'états, en faisant une exploration exhaustive de l'ensemble des états accessibles à partir d'un état initial. Il reçoit en entrée une abstraction du comportement du système (un système de transitions), représentée par une structure Kripke

K, et une propriété **φ** de ce système, formulée dans une certaine logique temporelle, et répond si l'abstraction satisfait ou non la formule **φ** c'est-à-dire, si **K |= φ** [Ben11].

L'intérêt du Model Checking est qu'il retourne une trace d'exécution du système violant la propriété lorsque cette dernière est non valide.). Maude intègre un LTL Model-Checker, qui est spécifié aussi en Maude.

Le Model-Checker contient un module appelé LTL qui définit tous les opérateurs qui permettent de construire les formules en logique temporelle linéaire.

6.3.1 Logique temporelle Temps réel

La logique temporelle peut être utilisée pour exprimer des propriétés temporelles liées au comportement d'un système. De telles propriétés ne sont que qualitatives, il n'est pas possible d'exprimer des limites temporelles quantitatives sur des événements [Mar02]. Ceci est cependant crucial pour exprimer les propriétés des systèmes en temps réel.

La logique temporelle temps réel MITL est une extension de la logique temporelle linéaire avec des contraintes temporelles quantitatives.

Les Contraintes quantitatives peuvent être exprimées à l'aide de prédicats de la forme **~ k, avec ~ Є {=, <, ≤, ≥, >} et k Є T,** où **T**est le domaine du temps. Ces prédicats sont introduits dans les opérateurs temporels. Ils permettent l'expression des délais bornés. Par exemple, la formule $\varphi_1 U_{<k} \varphi_2$ **,** est satisfaite sur une séquence d'états, si et seulement s'il existe un état **q** situé à moins de **k** unités de temps de l'état initial, et vérifiant φ_2 et tel que tous les états précédents le long de cette séquence satisfont φ_1. [Mil07]. Les contraintes quantitatives peuvent aussi être définies par des intervalles temporels, par exemple de la forme **[a, b].**

Les formules de logique temporelle sont interprétées sur des séquences d'états. Par contre dans MITL, les observations doivent être étendues par des contraintes sur leurs temps. Cela se fait en représentant une séquence d'observations comme une séquence d'états temporisée.

Les formules de la logique MITL sont construites à partir des propositions atomiques sur un ensemble de variables d'états. L'opérateur logique **Until** est indicé par un intervalle **I** qui est de l'une des formes suivantes :

[**a, b], [a, b[, [a, ∞[,]a, b],]a, b[,]a, ∞[où a <b** pour **a, b** ∈ **R+** (**R+** ensemble des réels positifs).

Les bornes inférieure et supérieure de **I** sont représentées respectivement par **l(I)** et ***r(I)*.** La dimension de cet intervalle est définie par **|I| = r(I) − l(I)** qui peut être infinie si **I** est non borné.

Les formules MITL sont définies inductivement par la grammaire suivante sur un ensemble de propositions **P** :

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi U_I \varphi$$

Où **I** est un intervalle non singulier et φ une proposition de ***P*.** L'hypothèse de singularité permet de garantir la décidabilité de la satisfiabilité d'une formule MITL [Mil07].

Les formules de la logique MITL sont interprétées sur des exécutions temporisées de la forme **τ = (q, I)** où **q** est une séquence d'états **q= q0q1q2. . .** et **I** une séquence d'intervalles I= **I0 I1 I2 . . .** telle que:

- pour tout **i**, les intervalles **Ii** et **Ii+1** sont adjacents et disjoints,
- tout instant **t** ∈ **R+** appartient à un intervalle **Ii**.

Une exécution temporisée **(q, I)** exprime que dans l'intervalle $\mathbf{I_k}$, le système se trouve dans l'état $\mathbf{q_k}$ (un changement d'état implique un changement de date).

Pour une formule φ de la logique MITL et une séquence d'exécution temporisée **τ = (q, I)**, la relation de satisfaction, **τ** satisfait φ notée **τ** $\models$ φ, est sémantiquement définie par :

- $\tau \models \varphi$ ssi $(\tau, 0) \models \varphi$,
- $(\tau, t) \models \top$,
- $(\tau, t) \models$ p ssi $\exists k$ tel que $t \in I_k$ et $p \in \rho(q_k)$, où ρ est la fonction de valuation qui associe à chaque état un ensemble de propositions,
- $(\tau, t) \models \neg\varphi$ ssi $(\tau, t) \not\models \varphi$,
- $(\tau, t) \models \varphi_1 \wedge \varphi_2$ ssi $(\tau, t) \models \varphi_1$ et $(\tau, t) \models \varphi_2$,
- $(\tau, t) \models \varphi_1 \, U_I \, \varphi_2$ ssi $\exists t' \in I + t$ † tel que $(\tau, t') \models \varphi_2$ et $\forall t'' \in [t, t'[, (\tau, t'') \models \varphi_1$.

Autres Opérateurs Les opérateurs unaires $\Diamond_I$ (finalement) et $\Box_I$ (toujours) peuvent être définis à partir de l'opérateur UI

- $\Diamond_I \varphi = \top U_I \varphi$.
- $\Box_I \varphi = \neg \Diamond_I \neg \varphi$.

Une séquence d'exécution τ satisfait la formule $\Box_I \varphi$ (respectivement $\Diamond_I \varphi$) à l'instant t, si et seulement si φ est satisfaite à tout instant (respectivement, à un certain instant) de l'intervalle t + I.

Nous pouvons également représenter les opérateurs de la logique LTL, $\Diamond, \Box$ et U en utilisant les opérateurs de MITL :

- $\Diamond \varphi = \Diamond_{[0..\infty[} \varphi$.
- $\Box \varphi = \Box_{[0..\infty[} \varphi$.
- $\varphi_1 \, U \, \varphi_2 = \varphi_1 \, U_{[0..\infty[} \, \varphi_2$.

L'opérateur $\rightsquigarrow_I$ (leads to) peut être défini à l'aide des opérateurs **toujours** et **finalement**, par exemple :

$$\varphi_1 \rightsquigarrow_I \varphi_2 = \Box(\varphi_1 \Rightarrow \Diamond_I \varphi_2)$$

Cette formule indique que si φ_1 est satisfaite, alors durant l'intervalle **I,** la formule φ_2 deviendra vraie inévitablement.

Exemple Considérons la propriété de vivacité bornée "Si la section est restrictive alors le train doit freiner dans les *k* unités de temps". Nous exprimons cette propriété en logique MITL par [Mil07].

$$\Box(section\_restrictive \Rightarrow \Diamond_{[0,k]} train\_freiné)$$

Maude LTL Model-Checker propose le module MODEL-CHECKER qui offre la fonction Model-check. L'utilisateur peut invoquer cette fonction en la donnant un état initial et la formule que nous souhaitons vérifier. Le Model-Checker va vérifier si cette formule est satisfaite dans l'espace d'états de système ou non. La fonction Model-check retourne True si la propriété est satisfaite. La syntaxe du module MODEL-CHECKER est la suivante :

fmod MODEL-CHECKER is

protecting QID .

including SATISFACTION .

```
including LTL .
subsort Prop <Formula .
*** transitions and results
sorts RuleName Transition TransitionListModelCheckResult .
subsortQid<RuleName .
subsort Transition <TransitionList .
subsort Bool <ModelCheckResult .
ops unlabeled deadlock: ->RuleName .
op {_,_} : State RuleName -> Transition [ctor] .
op nil : ->TransitionList [ctor] .
op __ :TransitionListTransitionList ->TransitionList
[ctorassoc id: nil] .
op counterexample:
TransitionListTransitionList ->ModelCheckResult [ctor].
op modelCheck : State Formula ~>ModelCheckResult
[special (...)].
Endfm
```

6.3.2 Définition d'une propriété temporelle

Dans notre travail, nous avons utilisé la logique MITL pour spécifier les propriétés temporelles, cela pour la raison que nous avons besoin de spécifier des contraintes de temps quantitatives.

Dans l'exemple du quadrant numérique, nous allons vérifier que le temps d'exécution de chaque état doit satisfait qu'il doit être dans l'intervalle [0, Clock] , càd que chaque état par lequel est passé le quadrant numérique doit consommer un temps inférieure à la valeur de chaque clock qui est égale à 5000 ms

Nous allons commencer par la déclaration des propositions suivantes :

```
ops step stepstepstep : String -> Prop .
eq <state :Modif_h | OPS : Entry Do Exit Clock , BLOCKED : false > CCF |=
step(state) + step(state) + step(state) < step(state) = true.
ops step stepstepstep : String -> Prop .
eq <state :Modif_m | OPS : Entry Do Exit Clock , BLOCKED : false > CCF |=
step (state) + step(state) + step(state) < step(state) = true.
```

ops step step : String -> Prop .

eq <state :Eclaire | OPS : ecl_C Clock , BLOCKED : false > CCF |=

step(state) < step(state) = true.

□ ~ (Entry(Modif_h) + Do(Modif_h) + Exit(Modif_h) < Clock(Modif_h))

□ ~ (Entry(Modif_m) + Do(Modif_m) + Exit(Modif_m) < Clock(Modif_m))

□ ~ (ecl_C(Eclaire) <Clock(Eclaire))

Apres l'ouverture d'une session Maude, les commandes suivantes sont introduites afin de vérifier les propriétés définies en dessus.

(red modelCheck(initState, [] ~ ((Entry(Modif_h) + Do(Modif_h) + Exit(Modif_h) < Clock(Modif_h))) .)

(red modelCheck(initState, [] ~ ((Entry(Modif_m) + Do(Modif_m) + Exit(Modif_m) < Clock(Modif_m))) .)

(red modelCheck(initState, [] ~ ((red ecl_C(Eclaire) < Clock(Eclaire)) .)

L'exécution du Model Checking s'effectue par l'ouverture d'une session Maude et l'introduction du module MCHECK. Puis l'introduction des commandes de vérification citées en dessus. Le résultat de vérification du code est présenté dans la figure suivante :

```
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.6 built: Mar 31 2011 23:36:02
            Copyright 1997-2010 SRI International
                  Wed May 27 20:56:36 2015
Maude> red modelCheck(initState, [] ~ ((Entry(Modif_h) + Do(Modif_h) + Exit(Modif_h)
< Clock(Modif_h))) .
reduce in MCHECK : modelCheck(initState, [] ~ (Entry(Modif_h) + Do(Modif_h) +
Exit(Modif_h) < Clock(Modif_h))) .
rewrites: 1003 in 5ms cpu (5ms real) (201 rewrites/second)
result Bool: true

Maude> red modelCheck(initState, [] ~ ((Entry(Modif_m) + Do(Modif_m) + Exit(Modif_m)
< Clock(Modif_m))) .
reduce in MCHECK : modelCheck(initState, [] ~ (Entry(Modif_m) + Do(Modif_m) +
Exit(Modif_m) < Clock(Modif_m))) .
rewrites: 1006 in 5ms cpu (5ms real) (202 rewrites/second)
result Bool: true

Maude> red modelCheck(initState, [] ~ ((red Do(Eclaire) < Clock(Eclaire)) .
reduce in MCHECK : modelCheck(initState, [] ~ ((red Do(Eclaire) < Clock(Eclaire)) .
rewrites: 1002 in 1ms cpu (1ms real) (1002 rewrites/second)
result Bool: true
```

Figure 4.27 Résultat de vérification par Maude Model-cheking.

Nous remarquons que tous les états par les qu'elles le quadran numérique est passé ont consommé un temps inferieur a la valeur du clock assigné pour chaque état.

7. Conclusion

Nous avons présenté dans ce chapitre notre contribution qui consiste à proposer une approche de vérification d'un modèle temporisé d'un système embarqué. L'approche est basée sur la méta-modélisation et la génération de code. Nous avons ainsi, détaillé la démarche suivie pour implémenter cette approche et faire la vérification formelle avec Model-cheking de Maude. Un exemple illustrant un système embarqué est modélisé par notre outil de modélisation et certaines propriétés temporelles sont vérifiées.

Conclusion générale

Le but de ce travail est de développer une approche de modélisation et de vérification automatique des systèmes embarqués. Cette approche s'intéresse à modéliser un comportement temporel du système et se compose en deux étapes :

- La première consiste au Développement d'un outil de modélisation d'un comportement temporel pour un système embarqué en se basant sur les concepts de l'ingénierie dirigée par les modèles. Pour se faire, nous avons utilisé les deux diagrammes d'UML Real Time Statechart (RTST) et Real Time Communication (RTC), qui sont des diagrammes traditionnels enrichis par des notations temporelles illustrant l'aspect temps réel d'un système embarqué. Pour développer cet outil, nous avons définis deux méta-modèles pour les deux digrammes d'UML. Ces méta-modèles forment une entrée pour l'outil ATOM^{3,} qui génère à son tour l'outil de modélisation et nous offre la possibilité de définir une grammaire de graphes pour assurer la génération automatique des descriptions équivalentes en langage Maude.
- La deuxième étape, concerne la vérification de certaines propriétés temporelles. Dans ce contexte, nous avons utilisé la logique temporelle temps réel (MITL) pour la raison que la logique temporelle (LTL), ne permet pas d'exprimer des contraintes temporelles quantitatives. Pour faire la vérification, nous avons utilisé model-checker de Maude qui prend en entrée la spécification générée dans la première étape et une propriété à vérifier exprimée en MITL, et répond si cette propriété est satisfaite ou non dans le système.

Notre approche peut être utilisée pour éviter certaines erreurs conceptuelles dans une phase de développement avancée avant d'implémenter un système embarqué. Dans un travail futur ; nous proposons d'étendre cette approche pour corriger les erreurs détectées et indiquer leurs sources. Nous prévoyons également d'utiliser d'autres outils de transformation de modèle tels que Triple Graph Grammar (TGG), Attributed Graph Grammar system (AGG)…etc. et comparer les résultats dans le but d'étudier leurs performances. Ainsi, nous proposons de nous occuper de la vérification de la transformation elle-même pour garantir formellement que la transformation est correcte.

Bibliographie

[AGG06] AGG home page. http : //tfs.cs.tu-berlin.de/agg, Octobre 2006.

[Ait12] Ait-Cheik-Bihi Wafa, Approche orientée modèles pour la vérification et l'évaluation des performances de l'interopérabilité et l'interaction des services, thèse de doctorat, université de Technologie de Belfort-Montbeliard , France, 2012.

[Ana06] Ananda Basu, Marius Bozga, Joseph Sifakis. Modeling Heterogeneous Real-Time Components in BIP. Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06), Sep 2006, Pune, India. IEEE Computer Society Press, pp.3-12, 2006.

[And04] Andre Sylvain, MDA (Model Driven Architecture) principes et états de l'art, mémoire d'ingéniera, Conservatoire National des Arts et Metiers, Lyon, France, 2004.

[AToM^3] AToM^3 home page. http://atom3.cs.mcgill.ca/, 2006.

[Att07] Attiogbé Christian Jérémie, Contributions aux approches formelles de développement de logiciels : Intégration de méthodes formelles et analyse multifacette, thèse de doctorat, Université de Nantes Atlantique, 13 septembre 2007.

[Aya10] Ayache Nicolas, Vérification Formelle Compositionnelle et Automatique de Systèmes de Composants, Université de Paris-sud 11 Centre d'Orsay, 5 janvier 2010.

[Ben11] Benammar Malika, Une Approche Basée Architecture pour la Spécification Formelle des Systèmes Embarqués, Thèse Doctorat en Sciences, Université Mentouri de Constantine, 2011.

[Ber02] Berger Arnold, Embedded System Design-An Introduction to Processes Tools and Techniques, CMP Books, ISBN: 1578200733, Lawrence, Kansas, USA 2002.

[Béz04] Bézivin Jean, Sur les principes de base de l'ingénierie des modèles, DBLP-Objet, 10(4) :145–157, Décembre 2004.

[Bla05] Blanc Xavier, MDA en action, Ingénierie logicielle guidée par les modèles, Edition Eyrolles, ISBN : 2-212-11539-3, 2005.

[Boul09] Boulkroune Ramzi, Les systèmes embarqués, Mémoire Online Informatique et Télécommunications, département d'électronique, université d'Annaba Algérie, 2009.

[Bou09] Bourahla Mustafa, Modeling and Verification of Real-Time Embedded Systems, Proceedings of the Third international conference on Innovation and Information and Communication Technology (ISIICT), December 2009.

[Bou12] Boulanger Jean Louis, Formal Methods, British Library Cataloguing-in-Publication Data, ISBN : 978-1-84821-362-3, 2012.

[Bou15] Bourdil Pierre-alain, Contribution à la modélisation et la vérification formelle par model checking - Symétries pour les Réseaux de Petri temporels, thèse de doctorat, Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse), 03 Décembre 2015.

[Bru06] Bruneton Eric, Coupaye Thierry, Leclercq Matthieu, Quéma Vivien, Stefani. Jean-Bernard, The Fractal Component Model and its Support in Java, Software - Practice and Experience (SP&E), 36(11-12):1257- 1284, September 2006. special issue on "Experiences with Auto-adaptive and Reconfigurable Systems".

[Cla01] Clarke Edmund M, Grumberg Orna, Peled Doron, Model Checking. MIT Press, Cambrigde, Massachusetts United States, ISBN: 978-0-262-03270-4, 2001.

[Cha18] Chafika DJaoui, Kerkouche Elhilali, Chaoui Allaoua, Khaled Khalfaoui, A Graph Transformation Approach to Generate Analysable Maude Specifications from UML Interaction Overview Diagrams, IRI Conférence: International Conference on Information Reuse and Integration : 10.1109/IRI.2018.00081, 2018.

[Chk10] Chkouri Mohamed Yacine, Modélisation des systèmes temps-réel embarqués en utilisant AADL pour la génération automatique d'applications formellement vérifiée**s,** thèse de doctorat, université Joseph Fourier, France, 7 Avril 2010.

[Cla00] Clavel Manuel, Duran Francisco, Eker Steven, Lincoln Patrick, MartiOliet Narciso, Meseguer José et Quesada José Francisco, A tutorial on Maude. SRI International, http://maude.cs.uiuc.edu/maude1/tutorial, March 2000.

[Cla02] Clavel Manuel, Francisco Duran, Steven Eker, Narciso Marti-Oliet, Patrick Lincoln, José Meseguer, Carolyn Talcott. Maude Manual. Version 2, 2002.

[Cla07] Clavel Manuel, Durán Francisco, Eker Steven, Lincoln Patrick, Marti-Oliet Narciso, Meseguer José, Carolyn Talcott, All About Maude – A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic. Lecture Notes in Computer Science, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Cla16] Clavel Manuel, Francisco Duran, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso MartiOliet, Meseguer José, Carolyn Talcott, Maude Manual (Version 2.7.1), July 2016.

[Cla99] Clavel Manuel, Duran Francisco, Eker Steven, Martı-Oliet Narciso, Lincoln Patrick, Meseguer José et Quesada José Francisco, Maude: Specification and Programming in Rewriting Logic, SRI International Lab, http://maude.csl.sri.com, 1999.

[Col09] Collavizza Hélène, Contribution à la vérification formelle et programmation par contraintes, Habilitation à Diriger des Recherches, Université de Nice-Sophia Antipolis, 2009.

[Cot08] Cottet Francis, Sébastien Gérard. Emmanuel Grolleau, Jérôme Hugues, Yassine Ouhammou, Sara Tucci-Piergiovanni, Systèmes Temps Réel Embarqués, Spécification, conception, implémentation et validation temporelle, 2eme édition,Dunod,ISBN 978-2-10-071331-8, 2008.

[Cou05] Courbot Alexandre, Pavlova Mariela, Gilles Grimaud, Vandewalle Jean-Jacques**,** Romization: Early Deployment and Customization Java Systems for Restrained Devices. Rapport de recherché INRIA N°5629 – Juillet 2005.

[Cou06] Courbot Alexandre, Pavlova Mariela, Gilles Grimaud, Vandewalle Jean-Jacques**,** A Low-Footprint Java-to-Native Compilation Scheme Using Formal Methods,In Seventh Smart Card Research And Advanced Application IFIP Conference (CARDIS'06), Tarragona, Spain, Avril 2006.

[Cse17] Cours sur les systèmes embarqués, www.technologuepro.com,2017.

[Cza03] Czarnecki Krzysztof, Simon Helsen, Classification of Model Transformation Approaches, OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[Dar02] Darvas Ádám, István Majzik , Balázs Benyó, Verification of statechart UML models of Embeeded sytems, University of Technology and Economics, Department of Measurement and Information Systems, 2002.

[Den05] Deng Qingxu, Xu Hai, Shuisheng Wei , Han Yu, Ge Yu, An Embedded SOPC System Using Automation Design. In Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'05), IEEE Computer Society, 2005.

[Der02] Derrien Steven, Guillou Anne-Claire, Patrice Quinton, Risset Tanguy et Wagner Charles, Automatic Synthesis of Efficient Interfaces for Compiled Regular architectures, In International Samos Workshop on Systems, Architectures, Modeling and Simulation (Samos), Samos, Grece, 2002.

[Eke02] Eker Steven, José Meseguer, Ambarish Sridharanarayanan, The Maude LTL Model Checker, In Fourth Workshop on Rewriting Logic and its Applications (WRLA'02), Vol 71 of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[Esp06] Espinoza Huascar, Medina Julio, Dubois Hurbert, Gerard Sébastien. FrançoisTerrier, Towards a UML based Modeling Standard for Schedulability Analysis of Embedded Systems, MARTES Workshop at MODELS Conference, Pages 79-90, 2006.

[Fat09] Fateh Boutekkouk, UML Modeling and Formal Verification of control/data driven Embedded Systems, UML AADL Workshop Potsdam, Germany,2009.

[Fav06] Favre Marie Jean, Estublier Jacky et Blay-Fornarino Mireille, L'Ingénierie Dirigée par les Modèles : au-delà du MDA, Traité IC2 – Information – Commande – Communication, Hermès – Lavoisier, 2006.

[Fei04] Feiler Peter H, John Hudak, Bruce Lewis, David P Gluch, Pattern-Based Analysis of an Embedded Real-Time System Architecture, World Computer Congress, Workshop on Architecture Description Languages (WADL04), Toulouse, August 2004.

[Fer14] Fernandes Pires.A, Amélioration des processus de vérification de programmes par combinaison des méthodes formelles avec l'Ingénierie Dirigée par les Modèles, Thèse de doctorat Sureté de logiciel et calcul de haute performance, Université de Toulouse, 26 juin 2014.

[Fra01] Fraboulet Antoine, Optimisation de la Mémoire et de la Consommation des Systèmes Multimédia Embarqués. Thèse de doctorat de l'université Lyon I, 2001.

[Gag07] Gagnon Patrice, Vérification Formelle de Diagrammes UML : Une Approche Basée sur la Logique de Réécriture, Mémoire de maitrise en mathématiques et informatique appliquées, Université du Quebec, 2007.

[Gar09] Gargantini Angelo, Elvinia Riccobene , Patrizia Scandurra, Integrating Formal Methods with Model-driven Engineering, In Proceedings of the Fourth

International Conference on Software Engineering Advances, Porto, Portugal , ISBN: 978-0-7695-3777-1, 2009.

[Hoc13] Hocine Riadh, Une méthodologie de conception automatique de modèles formels pour des descriptions System C, thèse de doctorat, Université de Batna, 2013.

[Ish05] Ishikawa Hiroo, Nakajima Tatsuo, EarlGray: A Component-Based Java Virtual Machine for Embedded Systems. In Proceeding of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), IEEE Computer Society, 2005.

[Kad16]Kadionik Patrice, Introduction Les Systèmes Embarqués, cours, http://www.enseirb.fr/~kadionik,14-Oct-2016.

[Ker11] Kerkouche Elhilali, Modélisation Multi-Paradigme: Une Approche Basée sur la Transformation de Graphes, thèse de doctorat , Université de Mentouri Constantine,2011.

[Kes05] Kesteney Yonit, Pnueli Amir, Timed and Hybrid statecharts and their textual representation, Formal Techniques in Real-Time and Fault-Tolerant Systems, Volume 571 of the series Lecture Notes in Computer Science pp 591-620, Springer Berlin Heidelberg, Online ISBN978-3-540-46692-5, 2005.

[Kor13] Kordon Fabrice, Hugues Jérome, Canals Agusti, Dohet Alain, Modélisation et analyse de systèmes embarqués, Édition Hermès/Lavoisier, 2013.

[Kri13] Krichen Fatma, Architectures logicielles à composants reconfigurables pour les systèmes Temps Réel Répartis Embarqués, thèse de doctorat, Université Toulouse II - Le Mirail et L'Université de Sfax ,2013.

[Lui01] Luis Alejandro Cortés, A Petri Net based Modeling and Verification Technique for Real-Time Embedded Systems, Thesis No. 919, ISBN 91-7373-228-1, Département d'informatique, Linköpings University, Sweden, 2001.

[Mam10] Mammeri Zoubir, Introduction aux systèmes embarqués et temps réel, Cours – Module ASTRE IRIT - UPS – Toulouse, 2010.

[Mar02] Marc Constantijn Willem Geilen, Techniques for Verification of Complex Real-Time Systems, Thèse de doctorat, Technische Universiteit Eindhoven, 2002.

[Mar03] Maraninchi Florence, Caspi Paul, La place de l'informatique dans l'enseignement des Logiciels et Systèmes Embarqués. Laboratoire Verimag, http://www-verimag.imag.fr/, Avril 2003.

[Mcc03] McCombs Theodore, Maude 2.0 Premier Version 1.0, August 2003.

[Men15] Menad Nadia, Modélisation des Systèmes embarques Temps-Réel : Vers une ingénierie dirigée par les méthodes formelles, thèse de doctorat, Université des Sciences et de la Technologie d'Oran (MB), Algérie, 2015.

[Mes04] Meseguer José, Roşu Grigore, Rewriting logic semantics: From language specifications to formal analysis tools, International Joint Conference on Automated Reasoning IJCAR 2004, vol. 3097, pp. 1–44. Springer, Heidelberg, 2004.

[Mes92] Meseguer José, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science, Volume 96(1), pages 73-155, 1992.

[Mes96] Meseguer José, Rewriting logic as a semantic framework for concurrency, In Lecture Notes in Computer Science, editor, 7th International Conference on Concurrency Theory, volume 1119. Springer Verlag, August 1996.

[Mil07] Miloud Rached, Spécification et vérification des systèmes temps réel réactifs en B, thèse de doctorat, Institut de Recherche en Informatique de Toulouse France, 2007.

[Mir05] Mirko Loghi, Tiziana Margaria, Graziano Pravadelli, Bernhard Steffen , Dynamic and formal verification of Embedded Systems: A Comparative Survey, International Journal of Parallel Programming, Vol. 33, No. 6, 2005.

[OMG05]OMG, UML Profile for Schedulability, Performance, and Time Specification, OMG document number: formal/05-01-02 (v1.1), January 2005.

[OMG99] OMG Unified Modeling Language Specification. Version 1.3, June 1999.

[Pet99] Petit Michael, Formal requirements engineering of manufacturing systems: a multiformalism and component-based approach, Thèse de doctorat de l'Université de Namur, Belgique, Octobre 1999.

[Pir14] Pires A Fernandes, Amélioration des processus de vérification de programmes par combinaison des méthodes formelles avec l'Ingénierie Dirigée par les Modèles, thèse de doctorat, Institut Supérieur de l'Aéronautique et de l'Espace (ISAE) Université Toulouse II, 2014.

[Poo01] Poole John D, Model-driven architecture : Vision, standards and emerging technologies, In In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models, 2001.

[Por06] Portolan Michele, Conception d'un Système Embarqué Sûr et Sécurisé, thèse de doctorat, Labotatoire TIMA, Institut National Polytechnique de Grenoble, France, 2006.

[Pto14] System Design, Modeling, and Simulation using Ptolemy II, First Edition, Version 1.02 ISBN: 978-1-304-42106-7, Claudius Ptolemaeus, Editor , 2014.

[Sus06] Susanne Graf, Sébastien Gérard, Oystein Haugen, Iulian Ober, Bran Selic, Modeling Standard for Schedulability Analysis of Real-Time Systems, International Workshop on Modeling and Analysis of Real-Time and Embedded Systems, MARTES at MoDELS, Research Rapport 343, ISBN 82-7368-299-4, ISSN 0806-3036, October 2006.

[Sve03] Sven Burmester, Holger Giese, Real-Time Statechart Semantics, Software Engineering Group, University of Paderborn, Germany, 2003.

[Tal05] Talarico Claudio, Gupta Aseem, Peter Ebenezer, Rozenblit Jerzy W, Embedded System Engineering Using C/C++ Based Design Methodologies, In Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05), IEEE Computer Society ,2005.

[UML20] https://laurent-audibert.developpez.com/Cours-UML/?page=introduction-modelisation-objet

[Val05] Vallius Tero, Röning Juha, Embedded Object Architecture. In Proceedings of the 8th Euromicro Conference on Digital System Design (DSD'05), IEEE Computer Society, 2005.

[Vas18] Vassil Todorov, Frédéric Boulanger, Safouan Taha, Formal verification of automotive embedded software, FORMALISE: 6th International Conference on Formal Methods in Software Engineering, Gothenburg, Sweden, 2018.

[Win90] Wing Jeannette .M, A Specifier's Introduction to Formal Methods, Computer, vol.23 (9):8 -23, 1990.

[Wol06] Wolfgang Theurer, Une méthodologie de modélisation multi-modèle distribuée par métier pour les systèmes embarqués, thèse de doctorat, École Nationale Supérieure des Ingénieurs des Études et Techniques d'Armement, France, 2006.