Une approche basée sur l'ingénierie dirigée par les modèles pour la vérification des descriptions AADL.

Mohamed Elkamel Hamdane

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université AbdElhamid Mehri Constantine 2
Faculté des Nouvelles Technologies de l'Information et de la Communication
Département d'Informatique Fondamentale et ses Applications

Année : ...../2018

n° d'ordre : ...../2018

Série : ...../2018

THÈSE

Pour l'obtention du diplôme
Docteur En Science
Spécialité : Informatique

Une approche basée sur l'ingénierie dirigée par les modèles pour la vérification des descriptions AADL

Par
Mohamed ElKamel HAMDANE
Soutenue le 14 Mai 2018, Devant le **JURY**

Faiza BELALA	Prof.	Université AbdElhamid Mehri Constantine 2	Président
Allaoua CHAOUI	Prof.	Université AbdElhamid Mehri Constantine 2	Rapporteur
Salah HAMRI	MCA	Université Larbi Ben M'hidi d'Oum El Bouaghi	Examinateur
Elkamel MERAH	MCA.	Université Abbas Laghrour Khenchela	Examinateur
Salah MERNIZ	MCA	Université AbdElhamid Mehri Constantine 2	Examinateur

à la lumière de ma vie, ma chère mère Farida

à la fleur de ma vie, ma fille Dhouha

à la plus proche de mon cœur, ma fille Nour Elyakin

à la femme que j'ai trouvé à côté de moi , ma femme Zineb

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

Je dédie cette thèse.

Remerciements

Je tiens à remercier toutes les personnes qui ont rendu cette thèse possible par leur aide et leurs contributions.

Je tiens d'abord à remercier mon encadreur de thèse Mr Allaoua Chaoui, professeur à l'université de Constantine 2 et fondateur du laboratoire MISC. Son sourire, son aide et ses conseils ont été d'une valeur inestimable.

Je voudrai tout particulièrement remercier Mr Martin Strecker, Maitre de conférences à l'Université de Toulouse 3 pour avoir suivi mes travaux durant ces années de thèse. Son expérience et les discussions que nous avons eues ; m'ont été des clés pour ouvrir le chemin de la recherche scientifique.

Je remercie Mme Faiza Belala, professeur à l'université de Constantine 2, pour m'avoir fait l'honneur de présider ce jury.

Je remercie Mr Salah Merniz, Docteur à l'université de Constantine 2 ; et Mr. Salah Hamri, Docteur à l'université de Oum El Bouaghi ainsi que Mr. Elkamel Merah, Docteur à l'université de Khenchela pour avoir accepté d'être les examinateurs de cette thèse.

Enfin, je voudrai remercier tous ceux qui ont contribué à ce travail.

ملخص :يَهْدِف هَذَا ٱلْعَمَل لِاقتِرَاح طَرِيقَةٌ تَسْمَحُ بِالتَأَكُدِ وَ ٱلتَحَقُقِ مِنْ تَصَامِيمِ AADL وَ هِيَ فِي طَورِ ٱلتَصْمِيمِ. الطَرِيقَةُ المُقْتَرَحَةُ تَسْتَخدِمُ ٱلمَنهَجِيةِ المُوَجَهَةُ بِالنَمَاذِجِ وَ التِي تَمَ تَدْعِيمُهَا بِسِلْسِلَةٍ مِنَ الوَسَايِلِ. أَوَّلاً: اِمْتِثَالاً لِمَنهَجِيَةِ ٱلتَصْمِيمِ القَائِمَةِ عَلَى مَفْهُومِ ٱلنَمَاذِجِ، قُمْنَا بِتَعْرِيفِ مِيتَا-نَمُوذَجٌ لِـ AADL وَ أَخَرٌ لِـ timed automata. ثَانِيًا اقْتَرَحْنَا حُزْمَةٌ مِنَ القَوَاعِدِ التِّي تُمَكِنُ مِنْ تَحْوِيلِ تَصْمِيمٌ لِنَمُوذَجِ AADL إِلى نَمُوذَجِ timed automata وَ هَذَّا عَلَى مَرحَلَتَينِ: الأُولَى تَعْمَلْ عَلَى اسَتِخْرَاجِ نَمُوذَجِ timed automata مِنْ نَمُوذَجْ AADL، الثَّانِيَةَ تَسْمَحَ بِتَّحْوِيلِ ٱلنُمُوذَجِ ٱلمُسْتَخْرَجِ إِلَى مِلَّفٍ وِفْقَ لُّغَةِ ta-format. إِنْطِلاقًا مِنْ Uppaal وَ اِسْتِنَادًا بِهَذَّا المِلَفِ يُمْكِنُ مُبَاشَرَةَ عَمَلِيَةِ ٱلتَحَقُقِ مِنْ خَصَايِصِ ٱلنُمُوذَجِ ٱلمُسْتَخْرَجِ. وَ لِمَعْرِفَةِ نَجَّاعَةِ ٱلطَرِيّقَةِ المُقْتَرَحَةِ قُمْنَّا بِتَطْبِيقِهَا عَلَى دِرَاسَةَ حَالَةٍ.

Abstarct, Résumé

Abstract. In this work, we propose an approach for the verification of the AADL (Architecture and Analysis Design Language) description. This approach is based in Model Driven Engineering (MDE) and assisted by a toolchain. Indeed, we define a source meta-model for AADL and a target meta-model for the timed automata formalism ; we define a transformation process in two steps : the first is a Model2Model transformation which takes an AADL Model and produces the corresponding timed automata model. The second transformation is a Model2Text transformation which takes a timed automata model and generates a text in ta-format code. This code is accepted by the Uppaal toolbox. The goal of this effort is to insure some properties of AADL models using the Uppaal model checker. A case study has been developed to show the feasibility and validity of the proposed approach.
Keywords : AADL, Timed Automata, Model Transformation, Verification, Uppaal

Résumé. Dans ce travail, nous proposons une approche pour la vérification des descriptions AADL (Architecture and Analysis Design Language). Cette approche est basée sur l'ingénierie dirigée par les modèles (IDM) et assistée par un ensemble d'outils. En effet, nous proposons un méta-modèle source pour AADL et un méta-modèle cible pour le formalisme des automates temporisés. Puis, nous définissons un processus de transformation en deux étapes ; la première est une transformation de type Model2Model qui prend un modèle AADL en entrée et produit le modèle d'automates temporisés correspondant en sortie. La deuxième étape est une transformation de type Model2Text qui prend le modèle généré dans la première étape et le transforme en texte dans le code "ta-format". Ce code est une représentation textuelle interne du model-checker UPPAAL. L'objectif de cette initiative est d'assurer certaines propriétés des modèles AADL comme l'absence de l'interblocage, la sûreté et la vivacité. Une étude de cas a été développé pour démontrer la faisabilité et la validité de l'approche proposée.
Mots Clés :AADL, Automate temporisé, Transformation de modèle, Vérification, Uppaal

TABLE DES MATIÈRES

TABLE DES FIGURES

Introduction Générale

INTRODUCTION GÉNÉRALE

Présentation du contexte de l'étude

La tâche de développement consiste aujourd'hui à modéliser des systèmes de plus en plus complexes qui doivent répondre à des exigences de fiabilité et de sécurité de plus en plus fortes. Cette situation est justifiée car la demande à l'innovation et le développement accéléré de nouvelles technologies de l'information et de la communications ne cessent de s'accroître.

La complexité de tels systèmes, dont les comportements deviennent de plus en plus difficiles à appréhender, est une problématique pour leur conception, leur développement et leur maintenance, comme le soulignent les récents rapports. Une problématique, a conduit à une révision approfondie sur les méthodes et les principes de modélisation afin de cerner cette complexité par des démarches de développement propres à ce type de systèmes y compris les systèmes développés par le langage AADL (Architectur and Analysis design language).

L'intégration des méthodes formelles dans le développement des systèmes complexes est une approche depuis longtemps reconnue pour remédier à ce problème. Plusieurs travaux récents ont encore souligné la nécessité d'utiliser les approches formelles pour répondre aux exigences croissantes de fiabilité car elles sont plus efficaces et permettent de réduire la possibilité d'introduction d'erreurs dans les premières phases de développement de ces systèmes.

C'est dans ce cadre que nous proposons dans cette thèse une approche pour la vérification des systèmes AADL. Cette proposition s'appuie sur deux axes complémentaires, le premier

est l'utilisation d'une démarche de transformation dans le cadre d'une approche de développement dite d'Ingénierie Dirigée par les Modèles (IDM). Le deuxième est relatif à une démarche de vérification formelle basée sur la technique du model-checking.

Dans notre cas, nous nous appuyons sur les automates temporisés comme modèles formels servant de support à la vérification. En plus de leurs capacités de modéliser les contraintes temporelles, l'utilisation des automates temporisés permet d'atteindre un bon niveau d'équilibrage entre l'expressivité de la modélisation formelle des systèmes étudiés et la puissance d'analyse par des outils stables de model-checking, grâce auxquels beaucoup de résultats sont prouvés.

Objectifs

L'objectif principal de cette thèse est d'étudier la possibilité d'utiliser la transformation de modèles dans le cadre de vérifications des propriétés et en particulier les propriétés temporelles sur des descriptions AADL. Pour le faire, il est nécessaire de pouvoir décrire le comportement des modèles étudiés. Le but préalable à la vérification devient alors l'extraction d'une description formelle sous forme d'automates temporisés représentant le comportement de système AADL en cours de développement.

Organisation du manuscrit

La première partie de ce manuscrit est consacrée au "contexte et état de l'art". Elle est divisée en quatre chapitres. Dans le premier chapitre, nous présentons une étude détaillée sur le langage AADL. Nous décrivons les éléments de base du langage accompagnés par des exemples, ainsi qu'une description détaillée sur le mécanisme de l'annexe comportementale.

Le deuxième chapitre est une présentation non exhaustive, mais assez complète sur les concepts de l'ingénierie dirigée par les modèles. Dans ce chapitre nous montrons comment cette ingénierie contribue à la maîtrise de la modélisation des systèmes complexe. Puis, on montre qu'il est nécessaire d'accompagner ce type de modélisation par une approche de vérification formelle.

Dans le troisième chapitre, on commence par présenter les différents aspects de la vérification par model-checking ainsi que les avantages qu'elle apporte pour la vérification des

systèmes complexes. Puis, on présentera en détail le formalisme des automates temporisés.

Dans le quatrième chapitre, on exposera différents travaux de transformation autour du langage AADL ; qui ont précédé et accompagné cette thèse et qui portent directement sur notre problématique. Ensuite, nous présenterons une étude comparative entre ces travaux.

Dans la seconde partie de ce manuscrit, nous nous focalisons sur notre “proposition et sa mise en œuvre”. Cette partie est divisée en deux chapitres. Dans le premier chapitre nous détaillons comment nous transformons une description AADL en formalisme des automates temporisés. Nous y exploitons les concepts introduits dans les chapitres précédents pour traduire les descriptions AADL en automates temporisés qui seront exploité directement par le model-checker UPPAAL. Les problématiques que nous abordons ici sont l’interprétation des éléments AADL en terme de construction logicielle, ainsi que l’intégration et la traduction de descriptions comportementales définies par une annexe comportementale au sein de la description AADL.

Le deuxième chapitre constitue une mise en application de notre approche. Nous présenterons d’abord les outils utilisés pour valider notre approche. Par la suite, Nous présenterons deux études de cas "Système de chauffage" et "Régulateur de température". Puis, nous préciserons en détail la chaîne de transformation que nous avons implémenté. Enfin, on donnera des exemples de vérification des propriétés à l’aide du model-checker UPPAAL.

Nous terminerons par une conclusion générale dans laquelle nous résumerons nos principales contributions et ferons le point sur un ensemble de perspectives.

Parie 1 : Contexte et état de l’art

CHAPITRE 1

PRÉSENTATION DE AADL

« Celui à qui on communique un enseignement est quelque fois plus apte à le comprendre que celui qui l'a entendu. » *Prophète Mouhamed (Paix et salut d'Allah sur lui)*

Sommaire

Introduction

AADL [1] (Architecture and Analysis design language) est un langage destiné à la modélisation des architectures logicielles et matérielles des systèmes embarqués temps réel. Ce langage est un dérivé du MetaH [2], normalisé par le standard SAE [3] (Society Of Automotive Engineers) pour décrire les systèmes embarqués temps réel.

Ce langage était destiné à ses débuts aux systèmes avioniques et spatiaux, ce qui explique son ancien nom (Avionics Architecture Description Language). Grâce à sa grande capacité d'expression qui dépasse largement le domaine de l'avionique il s'avère par la suite qu'il est extensible à d'autres systèmes logiciels et matériels.

La première version AADL 1.0 a été publiée en 2004, et le langage évolue régulièrement depuis. Une deuxième version du standard AADL 2.0 a été publiée en 2006 et révisée en 2008. En septembre 2012, le standard a publié la dernière version AADL 2.1 [4]. Vis à vis de ses qualités de modélisation, AADL joue un rôle central au sein de plusieurs projets Européen tels que, ASSERT [5] (Automated proof-based System Software Engineering for Real-Time systems), OpenEMBeDD [6],SPICES [7]...

Dans la littérature, plusieurs outils sont disponibles autour d'AADL, parmi eux nous pouvons citer :

— OSATE (SEI-AADL-Team, 2006) (Open Source AADL2 Tool Environment) : Outil de modélisation textuelle d'AADL (un modèle AADL peut être écrit dans un chier basé sur XML) en environnement Eclipse, développé par le SEI (Software Engineering Institute) . Il est indispensable pour commencer à faire de l'AADL.

— TOPCASED (Pi *et al.*, 2009) (Toolkit in Open-source for Critical Application Systems) : Outil de modélisation graphique d'AADL en environnement Eclipse, développé par le projet Topcased piloté par Airbus. Cet outil est fondé sur OSATE et le complète.

— CHEDDAR (Singhoff *et al.*, 2004) : Cheddar est un outil de simulation permettant de calculer différents critères de performance (contraintes temporelles, dimensionnement de res-

1. http://www.aadl.info/
2. http://www.htc.honeywell.com/metah/
3. http://www.sae.org/
4. http://www.aadl.info/aadl/currentsite/ [consulté 21/03/2018]
5. http://www.assert-project.net/.
6. https://openembedd.org
7. http://www.spices-itea.org/public/news.php

sources). Il permet, entre autre, de tester le respect des contraintes temporelles d'un jeu de tâches modélisant une application/un système temps réel. Cheddar peut être employé comme plug-in avec STOOD ou avec TOPCASED.

— ADeS (Architecture Description Simulation)(Geensyde, 2002) : Outil de simulation du comportement d'architectures décrites en AADL. Cet outil, développé par Axlog, repose sur OSATE.

— Ocarina (Ocarina, 2005) : Développé par Télécom-Paris-Tech Ocarina est un compilateur AADL écrit en Ada pour manipuler des modèles AADL. Cet outil propose la manipulation des modèles AADL, la génération des modèles formels, et la génération des applications distribuées.

— Versa/Furness toolset (Sokolsky *et al.*, 2006) : il permet de rassembler les principaux outils AADL open-source en un seul plugin sous l'environnement Eclipse.

— STOOD (Dissaux, 2004) : est un outil de modélisation et de conception de logiciels embarqué développé par la société Ellidiss. Il support trois notations graphiques, Hood, UML et AADL

— . . . etc.

Ces outils permettent : (i) d'éditer des modèles AADL (Stood, OSATE, TOPCASED, . . .); (ii) de générer le code logiciel du système à partir d'un modèle AADL (Stood, Ocarina,. . .); (iii) Conduire diverses analyses (OSATE, Versa/Furness, Cheddar,. . .).Par contre, la plupart de ces outils font juste la simuation. Et par définition la simulation permet d'explore une exécution possible du système soumis aux caractéristiques/restrictions de l'environnement. Par contre, il est souhaitable de faire ***une vérification complète du système*** afin d'explorer tous les comportements possibles et assure ainsi certains propriétés dès la phase de modélisation.

1.1 Principes du langage

Du point de vue modélisation, AADL peut être exprimé selon différentes syntaxes. Le standard en définit trois : en texte brut, en XML, ainsi qu'une représentation graphique(voir Figure 1.1). Par cette multiplicité des syntaxes possibles, AADL peut être utilisé par de nombreux outils [8] commerciaux ou non.

8. Pour plus de détails le lecteur peut s'orienter vers : https://wiki.sei.cmu.edu/aadl/index.php/AADL_tools [consulté le 14/03/2018]

Remarque :
Il faut noter que la syntaxe de référence dans AADL est ***la syntaxe textuelle***.

AADL offre aussi la possibilité de décrire des architectures complètes ; la partie matérielle et la partie logicielle du système. Plus précisément, un modèle AADL est présenté comme un assemblage de composants mappé sur une plate-forme d'exécution. AADL décrit les interfaces fonctionnelles des composants (tels que les données en entrée et en sortie), comment les composants sont combinés (c'est-à-dire comment les données en entrée et en sortie sont connectées ou bien comment les composants logiciels sont alloués aux composants matériels) (Benammar, 2011). Il décrit également les mécanismes de flux de données et des flux de contrôles utilisés dans les systèmes embarqués et les aspects non fonctionnels tels que les conditions de synchronisation, les comportements par défaut ou sur erreur, le partitionnement de temps et d'espace et les propriétés de sûreté, ...etc. Par contre, il est souhaitable de

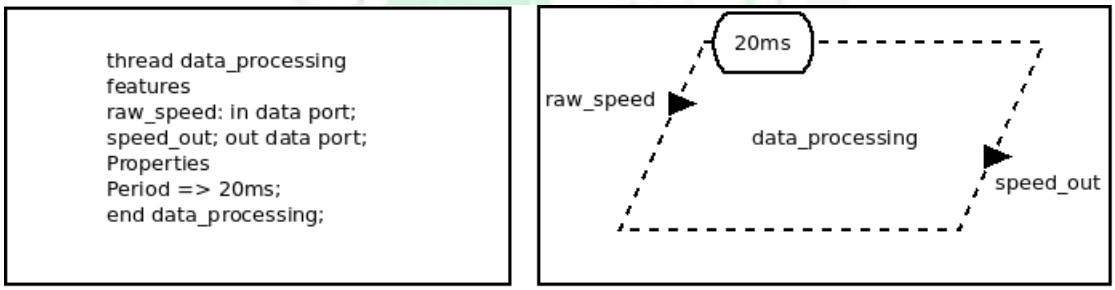


FIGURE 1.1: Représentation textuelle et graphique dans AADL

faire une vérification afin d'explorer tous les comportements possibles du système et assure ainsi certaines propriétés dés la phase de modélisation.

1.2 Notion de composant

Le langage AADL modélise l'architecture d'un système embarqué par une hiérarchie de composants, dont l'interaction est représentée par des connexions. Ce que fait que les composants sont les éléments de base d'une telle architecture . Un composant AADL, selon le

standard, représente une entité matérielle ou logicielle qui appartient au système en cours de modélisation (Feiler *et al.*, 2003).

Un composant possède un type (*component type*), qui spécifie son interface, et une ou plusieurs implantations (*component implementation*). Le type d'un composant est constitué de trois parties : les éléments d'interface (*features*), les flux (*flows*) et les propriétés (*properties*). Cette spécification est utilisée par les autres composants du système pour interagir avec ce composant. Une implantation, par contre, décrit la structure interne du composant en termes de sous composants (*subcomponents*), connexions (*connections*) entre les éléments d'interface de ces sous-composants, et les propriétés (*properties*), et les annexes . Dans ce cas, le composant est généralement décrit sous forme d'un assemblage de sous-composants qui sont des instances de types ou d'implantation d'autres composants. La spécification d'une implantation englobe aussi les connexions qui lient les sous-composants, les flux (*flows*) traversant les sous-composants, les modes (*modes*) pour représenter les états opérationnels, et les propriétés qui spécifient les caractéristiques structurelles et/ou comportementales de ce composant. Dans la figure 1.2 nous proposons une représentation simple qui résume les parties d'un composant AADL :

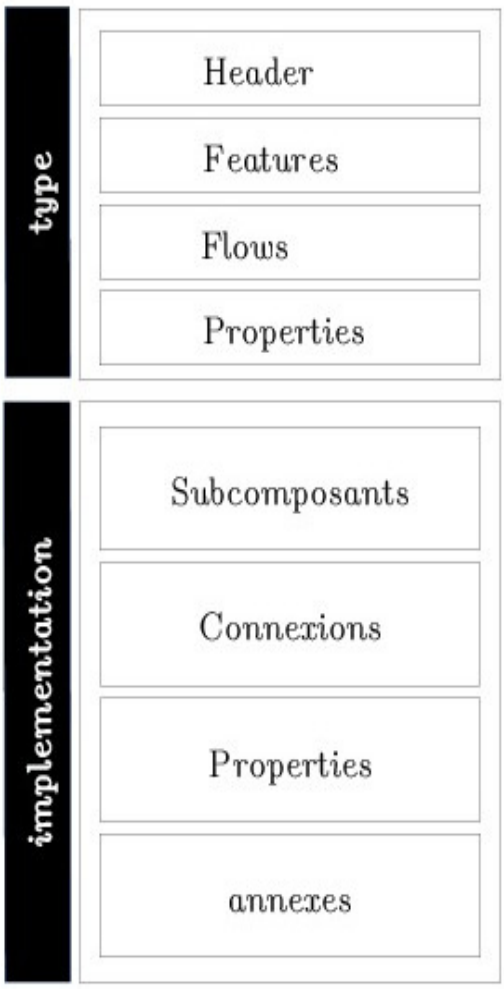


FIGURE 1.2: Architecture abstraite d'un composant AADL

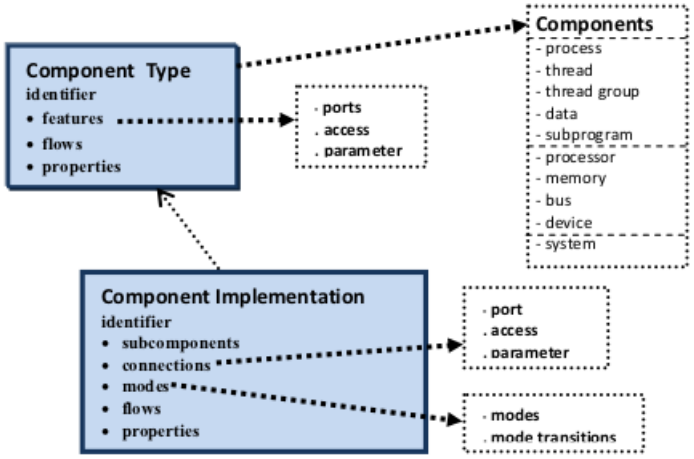


FIGURE 1.3: Description des éléments architecturaux dans AADL

En AADL, les ports ce sont des éléments permettant à un composant d'échanger les données et les signaux avec l'extérieur. En effet, ils correspondent à la principale façon de décrire les échanges d'information entre composants. Ils sont déclarés en entrée, en sortie ou en entrée/sortie. On distingue trois types de ports : les ports d'événement (servent principalement à transmettre des signaux, la réception d'un événement peut déclencher l'activation d'un thread), les ports de donnée (servent à transmettre les donnée mais contrairement aux ports d'événements, ils ne déclenchent rien à la réception), les ports d'événement/donnée (servent à transmettre et stocker des données. Ils disposent d'une fille pour stocker les données arrivant et ont le même comportement que les ports *event*).

Du point de vue élément de modélisation, AADL définit trois catégories principales de composants, les composants logiciels (*data, subprogram, thread, thread group, process*), les composants matériels (*memory bus, processor, device*), et le composant system. Ce dernier, ne représente pas une entité concrète mais il est utilisé pour créer des blocs qui aident à structurer l'application.

1.2.1 Composants matériels

L'ensemble de ces composants matériels permettent de définir l'architecture matérielle d'une application. En effet, AADL offre quatre catégories de composants matériels : les processeurs, les mémoires, les bus et les périphériques.

Les processeurs sont décrits en utilisant le composant processor. Celui-ci modélise un micro-contrôleur et un noyau (système d'exploitation réduit au strict minimum : ordonnanceur, pilotes. . .).

Un processeur exécute les processus légers du système, il peut contenir des mémoires comme sous-composants. Un processeur peut " accéder " à un bus.

Les mémoires sont modélisées avec le composant *memory*. Il représente une mémoire physique quelconque (disque dur, mémoire vive. . .). Les mémoires servent à stocker les données et le code et sont accessibles par les processus légers en cours d'exécution.

Les bus sont modélisés par l'intermédiaire du composant bus. Ils fournissent l'accès entre les processeurs, les périphériques et les mémoires. Les connexions entre les entités logicielles d'un système peuvent être associées à un bus donné auquel cas le flot de données (ou de contrôle) passe par le bus en question.

Le périphériques sont décrits à l'aide des composants *device*. Ils modélisent une large variété de matériel allant des simples capteurs jusqu'aux appareils les plus complexes. Dans tous les cas, un périphérique est vu comme une boite noire et sa structure interne ne peut être décrite en AADL. Seule l'interface d'un périphérique est visible par les autres composants du système.

1.2.2 Composants software

Il existe cinq catégories de composants logiciels : les processus lourds, les processus légers (tâches), les groupes de processus légers, les sous-programmes et les données. Les processus lourds sont modélisés en utilisant le composant *process*. Un processus AADL est un espace mémoire qui sert à contenir les processus légers ainsi que les données partagées entre eux. Par conséquent, pour effectuer un comportement quelconque, un processus lourd doit contenir au moins un processus léger. Les processus légers sont modélisés grâce aux composants *thread*. Ils modélisent les fils d'exécution qui constituent la partie active de l'application (comme les *threads POSIX* par exemple). Le comportement effectué par un processus léger est spécifié par l'une des manières suivantes :

— En utilisant les propriétés et en pointant vers du code fourni par l'utilisateur,

— En utilisant les annexes pour décrire le comportement au sein même du modèle,

— En appelant des sous-programmes AADL.

Dans le cas où de nombreux processus légers d'un système possèdent des caractéristiques proches et pour éviter la duplication de code, AADL introduit les groupes de processus légers. Ils décrivent des tâches partageant un nombre de propriétés. Ces groupes servent aussi à introduire une hiérarchie entre les processus légers. Ils sont décrits en utilisant le composant *thread group*.

Les sous-programmes sont modélisés avec le composant *subprogram*. Ils décrivent des procédures comme en C ou en ADA. Comme pour les processus légers, le comportement des sous-programmes est spécifié de plusieurs manières (propriétés, annexes, appel à d'autres sous-programmes AADL).

Enfin, les données sont modélisées en utilisant les composants data. Les données représentent des types de données, lorsqu'elles sont déclarées sous la forme de composants ou bien quand elles sont utilisées dans les éléments d'interfaces. Elles représentent des variables partagées lorsqu'elles sont instanciées sous la forme de sous-composants.

La notion de composant est hiérarchique. Une implantation peut contenir des sous composants. Un sous composant est une instance d'une implantation. L'implantation d'un composant peut contenir une déclaration de modes, chaque mode représente une configuration alternative de sous composants.

La figure 1.5 représente l'implantation d'un processus AADL. L'interface de ce processus est constitué de deux ports p1 et p2. Ce processus contient deux instances de *threads*, chacun de ces *threads* dispose aussi de deux ports. Les lignes entre les ports représentent les connexions entre les différentes interfaces.

1.3 Connexion entre composants

Une description AADL est un assemblage de sous-composants connectés entre eux au moyen de connexions (Vergnaud, 2006). Les connexions sont un autre élément fondamental du langage AADL, car elles permettent d'assembler les composants présents dans une architecture.

Une connexion est orientée et peut éventuellement être nommée. Une connexion AADL est un lien qui représente la communication de données et de contrôle entre les composants, plus précisément, entre les ports de différents *threads* ou entre les *threads* et les processeurs ou les périphériques. La source et la destination finales d'une connexion sont soit un *thread*, un

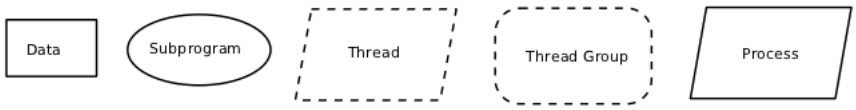


FIGURE 1.4: Représentation graphique des éléments logiciels

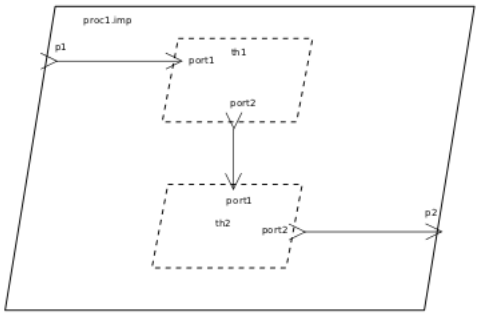


FIGURE 1.5: Représentation graphique d'une implémentation de processus

processeur, ou un périphérique. Dans le cas d'une connexion entre deux ports data ou deux ports *event data*, les connexions sont typées, on ne peut établir une connexion qu'entre ports transmettant un type de données compatibles.

L'exemple de Listing 1.1 illustre la méthode d'utilisation des connexions. En effet, cet exemple définit un *thread PR* et son interface composée de 6 ports, un de champs type. La figure 1.5 montre la version graphique de cette spécification.

```
1  thread PR
2  features
3  Port1 : in data port ;
4  Port2 : in event port
5  {Queue_Size => 5;
6  Dequeue_Protocol => OneItem};
7  Port3: in event data port
8  {Queue_Size => 10;
9  Dequeue_Protocol => AllItem};
10 Port4: out data port;
11 Port5: out event port;
12 Port6: out event data port;
13 end PF;
14
15 process proc1
16 features
17 p1: in event port;
18 p2: out event port;
19 end proc1;
20
```

```
process implementation proc1.imp
subcomponents
th1: thread th1.imp;
th2: thread th2.imp;
connections
connection1: event port p1->th1.port1;
connection2: event port th1.port2->th2.port2;
connection1: event port th2.port2->p2;
endproc1.imp
```

Listing 1.1: Connexions entre composants

1.4 Les propriétés

Les propriétés constituent aussi un aspect fondamental d'AADL car elles permettent d'enrichir de manière spécifique la description d'un composant (Feiler *et al.*, 2003). Cette notion permette d'associer des informations aux composants AADL. Il est ainsi possible de décrire les contraintes s'appliquant à l'architecture. Par exemple, les propriétés sont utilisées pour spécifier le temps d'exécution théorique d'un sous-programme, la période d'un *thread*, le protocole de fille d'attente utilisé pour un port d'événement/donnée,...etc.

Les déclarations de propriétés sont regroupées dans des ensembles de propriétés (property sets), semblables aux paquetages. Une propriété se définit par un nom, un type, la liste des éléments auxquels elle peut s'appliquer, et éventuellement une valeur par défaut. Nous en décrivons ici quelques unes, qui sont utilisées dans le cadre de ce travail.

(A) Propriétés liés aux threads : elles permettent de renseigner les politiques de déclenchement des *threads*, la concurrence, et le passage d'un mode AADL à un autre. On y trouve :

— *Dispatch Protocol* : Les informations liées au déclenchement de *threads* (apériodique, sporadique, périodique, etc.)

— *Dequeue Protocol* : Les informations liées aux politiques de traitement des messages reçus (traite un, plusieurs ou tous les messages)

— *Compute_exection_time* : représente le temps d'exécution d'un *thread* ou d'un *sousprogramme*.

(B) Les propriétés liées aux ports : Contrairement aux data ports, les events ports peuvent être mis en file d'attente. L'information envoyée sur un port data peut écraser la précédente.

La longueur de la file peut être spécifiée par l'intermédiaire d'une propriété AADL. Si la file d'attente est pleine, une des politiques suivantes peut être adoptée :

— *DropOldest* : On Supprime le message le plus ancien,

— *DropNewest* : On Supprime le message le plus récent,

— La politique de dépilage est spécifiée par la propriété *Dequeue–Protocol*.

(C) Propriétés temporelles : Il existe plusieurs propriétés pour la notion de temps :

— *X_Deadline* : délai maximal autorisé pour effectuer l'opération X.

— *X_Execution_Time* : temps estimé pour exécuter l'opération X.

— *Period* : qui donne le délai minimal entre deux déclenchements d'un *thread*.

1.5 Les annexes

Les annexes sont un autre moyen d'incorporer des informations aux éléments d'une description AADL. Contrairement aux propriétés, elles ne peuvent être associées qu'aux déclarations de composants et permettent d'insérer des informations exprimées dans une syntaxe différente de celle d'AADL (Kawtharany, 2007). En effet, elles permettent d'étendre la syntaxe AADL tout en utilisant des outils existants. La principale utilisation de ce mécanisme est de permettre de développer facilement de nouvelles analyses basées sur des notations ou des langages spécifiques.

Plusieurs annexes autour de AADL, utilisant différents formalismes, ont été approuvées et publiées par le standard. On peut citer par exemple :

— ***Annexe des notations graphiques*** : Les diagrammes graphiques d'AADL apportent plus de clarté l'utilisation et de compréhension au plan architectural d'un système embarqué. En effet, elle définit un ensemble de symboles graphiques pouvant être utilisés pour exprimer des relations entre composants, dispositifs, et connexions dans un modèle AADL.

— ***Annexe des formats d'échange XMI*** : elle contient la définition du méta modèle AADL et les formats d'échange basés-XML pour les modèles AADL(SEI-AADL-Team, 2006) . Le méta modèle AADL décrit la structure des modèles AADL, c'est-à-dire, une représentation objet des spécifications AADL qui correspond sémantiquement à un arbre syntaxique abstrait.

— ***Annexe d'Interface de Programme d'Application*** : Cette annexe fournit des directives aux utilisateurs permettant d'assurer la transition entre les modèles AADL et le texte source d'un langage de programmation tels que ADA ou C.

— ***Annexe du modèle d'erreur*** : Elle permet également des évaluations qualitatives et quantitatives des propriétés du système telles que la sûreté, la fiabilité, l'intégrité, la disponibilité, et la maintenabilité.

— ***Annexe comportementale*** : le but de cette annexe est de permettre la description du comportement interne de composant AADL. En effet, cette annexe permet d'établir des relations entre les entrées d'un *thread* et ses sorties via un système de transition.

Dans ce qui suit, nous allons donner plus de détail concernant l'annexe comportementale car elle est considéré comme une partie clé pour les travaux réalisés dans le cadre de cette thèse.

1.5.1 Étude de l'annexe comportementale

L'annexe comportementale est décrite par un système de transition comme le montre le listing 1.2 :

```
1 annex specification_comportement
2 {**
3 <state_variables >
4 <initialisation >
5 <states >
6 <transitions >
7 **};
```

Listing 1.2: Structure générale de l'annexe comportementale

Elle est définie par les éléments suivants :

— Les états (States) : les états de cet automate peuvent être déclarés localement ou faire référence aux états de l'automate de mode du composant. L'état interne du composant peut aussi être décrit à l'aide de variables.

— Les Transitions (Transition) : Cet élément permet de définir les transitions de l'état source vers l'état de destination. La transition peut avoir une garde (une condition booléenne par exemple) et une action à exécuter si la garde est vraie.

1. ***Les états dans annexe comportementale*** : L'annexe comportementale décrit les systèmes de transitions qui définissent le modèle d'états. Ce modèle d'états est décrit par les options de l'automate (Kawtharany, 2007).
   — *Initial* : Correspond à l'état dans lequel se trouve le composant après initialisation
   — *Final* : Correspond à l'état dans lequel se trouve le composant après finalisation (retour d'un appel de fonction ou finalisation d'une tâche)
   — *Complete* : utilisable pour les *threads* seulement : correspond à un état dans lequel le *thread* n'utilise pas la ressource d'exécution (préempté, attente passive,...etc).
2. ***Les transitions dans annexe comportementale*** : Chaque transition est associée :
   (a) Un état source
   (b) Un état cible
   (c) Une garde : est une expression décrivant le moment où la transition pourra être exécutée. La contrainte de temps est représentée dans l'annexe par la clause <*guard*> qui contient une réception facultative d'événement ou de données d'événement. La garde d'annexe comportementale peut prendre trois valeurs :
      — la garde est toujours vraie
      — la garde sous forme d'une expression.
      — la garde sous forme d'une expression et d'un port (*event*).
   (d) Un ensemble d'actions : Les actions peuvent être des appels à des sous programmes,des modifications des variables locales, ou des émissions de données ou d'événement sur les ports du composant.
3. ***Communication des événements (Envoi/Réception) et des messages :***
   (a) Réception d'événements : les messages sont reçus par des ports d'événement, ou ports de données ou ports de données/événement. Les ports d'événements et les ports de données/événement sont associés aux files d'attente. En déclenchant, zéro, un ou tous les éléments de la file d'attente sont transférés au *thread*, selon la valeur de la propriété de Dequeue_Protocol. Pour les ports de données, zéro ou un message seront transférés. Des données simples sont lues en utilisant le nom du port. Si "*p*" est un *Event Port* ou un *Event Data Port* :

— $p.count$: est une fonction qui retourne le nombre d'évènements dans la fille interne du *thread*.

— $p?$: retire un évènement de la file de messages.

— $p?x$: retire un évènement de la file de messages et stocke la donnée du message dans la variable x.

— $p.refresh$: est une fonction qui retourne vrai si le port a été modélisé depuis le dernier transfert.

(b) Envoi de messages : les messages sont envoyés par des «output» d'événements de données ou ports d'événement de données. Les données peuvent être stockées dans des ports de données et dans des ports de données d'événement en utilisant le nom « port » comme variable.

— Si "p" est un *Event Port* ou un *Event Data* Port, la clause "p!" correspond à l'envoi immédiat de l'évènement.

— Si "p" est un *Event Data Port* ou un *Data Port*, la clause "p := d" correspond à l'écriture de la valeur de d sur le port.

— Si "p" est un *Event Data Port*, la clause "p!d" correspond à l'appel système et l'envoi immédiatement de l'évènement avec la valeur de la donnée d.

4. ***Mécanismes de synchronisation dans l'annexe comportementale*** : Les actions de délai et de calcul (computation) indiquent des intervalles non déterministes de temps d'attente et de temps de calcul. La contrainte booléenne d'arrêt (*timeout*) peut être employée à l'intérieur des gardes.

— *Computation (minimum, maximum)* : exprime l'utilisation de l'unité centrale de traitement pendant une période non déterministe entre le minimum et le maximum.

— *Delay (minimum, maximum)* : le délai entre l'émission et la réception du message. Cette fonction exprime une suspension pendant une période non déterministe entre le minimum et le maximum.

Le Listing 1.3 montre un exemple d'utilisation de l'annexe comportementale. Ce modèle AADL représente le comportement d'un *thread* nommé *Producer*. C'est un *thread* périodique avec une période de 10ms. Ce *thread* contient une annexe comportementale, qui défini une variable c de type entier avec une valeur initial $c = 0$. L'état initial de l'annexe comportementale

est l'état s_0 à partir duquel il peut se déplacer vers l'état s_1 à travers une interaction sur le port *Data_Source*. Lors de la transition la valeur de c est incrément.

```
thread Producer
feature
Data_Source: in out event data port Behavior :: integer;
end Producer;
thread implementation Producer.imp
properties
Dispatch_Protocol => Periodic;
Period => 10ms;
annex behavior_specification {**
state variables c : Behavior::integer;
initial c:=0;
states
s0: initial complete state;
transitions
s0-[Data_Source!(c)]-> s0{c:=c+1};
**}
```

Listing 1.3: Exemple descriptif montre l'utilisation de l'annexe comportementale

1.6 AADL vs UML, SysML et MARTE

Dans cette section, nous allons tenter de proposer une petite comparaison entre AADL et UML (Unified Modeling Language) (Booch *et al.*, 1994). Cette comparaison apparaît incontournable vu la place importante du langage UML dans le domaine de la modélisation des systèmes logiciels.

AADL est considère comme un des langages de description d'architecture logicielle et matérielle, il offre un véritable langage textuel et graphique. Il est utilisé dans plusieurs domaines en particulier dans les systèmes embarqués et temps réel. Par ailleurs, UML est né au milieu des années 90, standardisé par l'OMG issu et fruit d'un large travail d'experts reconnus, de nombreux acteurs industriels ont adopté UML et participent à son développement. Depuis les premières versions, le langage connait un succès croissant. Actuellement UML est dans la version [9] 2.5 qui apparaît aujourd'hui comme un langage très puissant pour la modélisation des systèmes logiciels (Idiri, 2009).

AADL définit non seulement, la représentation textuelle et graphique de l'architecture logicielle et matérielle, mais permet aussi de préciser formellement (à certains niveaux)

9. depuis septembre 2013

quelques aspects de la syntaxe et la sémantique (Idiri, 2009). La description AADL peut être simuler par des analyseurs syntaxique et sémantique du langage pour assurer la cohérence (ex : un thread ne peut pas contenir un processus). Par ailleurs, UML repose sur un ensemble de diagrammes pour représenter un système (des diagrammes structurels décrivant l'architecture du système et des diagrammes comportementaux décrivant le comportement du système). De même que AADL repose sur la notion de composant pour décrire l'architecture logicielle et matérielle, UML aussi depuis la version 2.0, a intégré la notion de composant (diagramme de composant), mais uniquement pour modéliser l'architecture logicielle pas matérielle. Pour concurrencer le langage AADL, le consortium OMG a proposé une extension du standard UML nommée, MARTE (UML Profile for Modeling and Analysis of Real-Time and Embedded Systems) (Thomas *et al.*, 2007) pour modéliser les systèmes embarqués temps réel (la concurrence, les contraintes temporelles, les contraintes d'embarquement (taille de la mémoire), les supports d'exécution qu'ils soient matériels ou logiciels). En outre, AADL et UML permettent une modélisation semi-formelle ce qui engendre une possibilité d'avoir une ambiguïtés sémantiques en particulier lorsque le système est d'une complexité importante.

Le tableau 1.1 suivant résume une comparaison entre AADL et les profiles UML.

TABLE 1.1: Comparaison entre AADL, UML, SysML et MARTE (Evensen et Weiss, 2010)

	AADL	**UML**	**SysML**	**MARTE**
Utilisation	Système embarque temps réel et matériel aéronautique	Modélisation des systèmes orientés objets	Système d'engineering	Système embarque temps réel et matériel aéronautique
Formalisme	• Sémantique stricte des connecteurs de composants • Les relations entre les entités sont renforcées par un langage spécifique	• Centré diagramme • Formalismes sous-jacentes définis par le méta-modèle UML	• Centré diagramme • Formalismes sous-jacentes définis par le méta-modèle SysML	• Centré diagramme • Formalismes sous-jacentes définis par le méta-modèle MARTE • Profil UML pour AADL • Guide de modélisation pour chaque niveau d'abstraction
Architecture	• Exécution des entités software • Composants matériel temps réel • Représentation statique du logiciel	• Implémentation, exécution, et compilation des entités temporelles. • Représentation statique/dynamique du logiciel. • Du haut niveau d'abstraction jusqu'aux détails de la conception.	• Vue conceptuelle et système • Représentation statique/dynamique du système	• Implémentation, exécution, et compilation des entités temporelles. • Composants matériel temps réel • Du haut niveau d'abstraction jusqu'aux détails de la conception.
Outils	Open Source AADL Tool Environment(OSATE) , Ocarina	MagicDraw, Visual Paradigm,Papyrus UML	MagicDraw, Visual Paradigm,Papyrus UML	MagicDraw, Visual Paradigm,Papyrus UML
Extensibilité	• configure un ensemble de propriétés • Libraires d'annexes • configure un plug-ins pour OSATE	• Stereotypes • Meta-models • Profiles	• Stereotypes - Meta-models	• Stereotypes • Meta-models
Limitation	• abstraction sur les composants temps réel du système d'exploitation • Pas de modélisation du comportement	• Nombre important des diagrammes • Formalisme non disponible	Utilisation uniquement pour les systèmes d'engineering	• Nombre important des diagrammes. • Méta-modèle sous-jacent complexe

1.7 Synthèse sur AADL

AADL comme langage de description d'architecture est conçu particulièrement pour modéliser et analyser les systèmes embarqués temps réel. Cependant, le standard AADL se focalise sur la description des aspects architecturaux tels que les composants et leurs connexions, mais ne traite pas directement leur implantation comportementale, ni de la sémantique des données manipulées.

Le composant *thread* est considéré comme la brique de base d'une modélisation en langage AADL. En effet, le standard donne la possibilité de spécifier les conditions d'exécution des *threads* par la déclaration des propriétés tels que le *deadline*, la politique d'expédition (*Dispatch_Protocol*), la période,...etc. Le standard définit aussi, pour chaque *thread*, une sémantique dynamique à travers l'ajout de l'annexe comportementale écrit à base d'automate décrivant ces états et les conditions de transition de ses états. On remarque que l'annexe comportementale offre la possibilité d'associer à certaine action des unités temporelles ce qui reflète l'importance qu'a donné le standard pour la prise en charge de l'aspect temporel.

Conclusion

Dans ce chapitre introductif nous avons rappelé les éléments du langage AADL et avons retracé les évolutions qu'il a lancé depuis sa création jusqu'aux recommandations et normalisations les plus récentes. En effet, tous les travaux étudiés considèrent que AADL est un langage très riche en terme d'expressivité car il permet de couvrir un grand nombre d'applications avec des comportements variés et complexes. De plus la notion de l'annexe comportementale intègre les notions de base pour un système temps réel, comme la synchronisation des événements et des messages, l'invocation de sous-programmes avec des protocoles temps réel, etc., ce qui nous permet de modéliser les parties synchrones et asynchrones d'un système AADL.

Il est clair qu'AADL a une longueur d'avance sur ses concurrents grâce à sa richesse d'expressivité. Par contre, cette richesse doit être structurée dans une approche de modélisation afin de maîtriser la complexité croissante des systèmes. En effet, la nouvelle approche de modélisation IDM (Lämmel *et al.*, 2006) s'inscrit dans ce cadre.

Dans le chapitre suivant, nous allons présenter cette approche qui est considérée comme un pilier dans le domaine de la modélisation des systèmes complexes.

CHAPITRE 2

INGÉNIERIE DIRIGÉE PAR LES MODÈLES ET LA TECHNOLOGIE DE TRANSFORMATION

Si tu veux savoir qui est le modèle de tous les modèles ; sans doute c'est le *Prophète Mouhamed (Paix et salut d'Allah sur lui)*
Hamdane Mohamed Elkamel

Sommaire

Introduction

Suite à l'approche objet des années 80 et de son principe du « tout est objet », l'ingénierie du logiciel s'oriente aujourd'hui vers le principe du « tout est modèle ». Elle œuvre à fournir un cadre de développement dans lequel les modèles passent de l'état contemplatif à l'état productif et deviennent les éléments de première classe dans le processus de développement. En effet, l'approche vise ainsi à améliorer la portabilité et la séparation des concepts, en particulier la séparation de la technologie des concepts métiers. En pratique, pour un projet cela permet d'accélérer le développement et de réduire les coûts de conception.

Par conséquence, l'ingénierie dirigée par les modèles (IDM), ou Model Driven Engineering (MDE) en anglais (Lämmel *et al.*, 2006) a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique (voir figure 2.1) (Combemale, 2008).

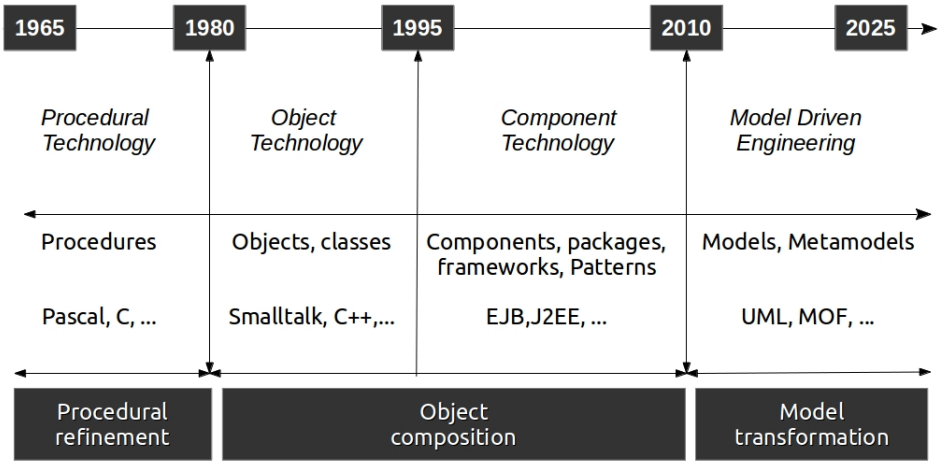


FIGURE 2.1: Évolution chronologique des paradigmes/artefacts (Bézivin, 2014)

Nous proposons dans ce chapitre une présentation des principes clés de cette nouvelle

ingénierie. Nous introduisons dans un premier temps la notion de modèle à travers les travaux de normalisation de l'OMG. Nous détaillons ensuite les deux axes principaux de l'IDM à savoir la méta-modélisation et la transformation de modèle. Par la suite, on va présenter les outils actuellement disponibles pour la transformation de modèle. Enfin, Nous concluons par une discussion sur la combinaison entre les techniques formelles et l'approche IDM.

2.1 Les principes généraux de l'IDM

Cette méthodologie est encore récente et fait l'objet de plusieurs propositions. On peut la retrouver, avec de légères variations, sous de nombreux noms : Model-Driven Architecture (MDA [1]), Model-Driven Development (MDD), Model Integrated Computing (MIC), Model-Driven Software Development (MDSD), etc.

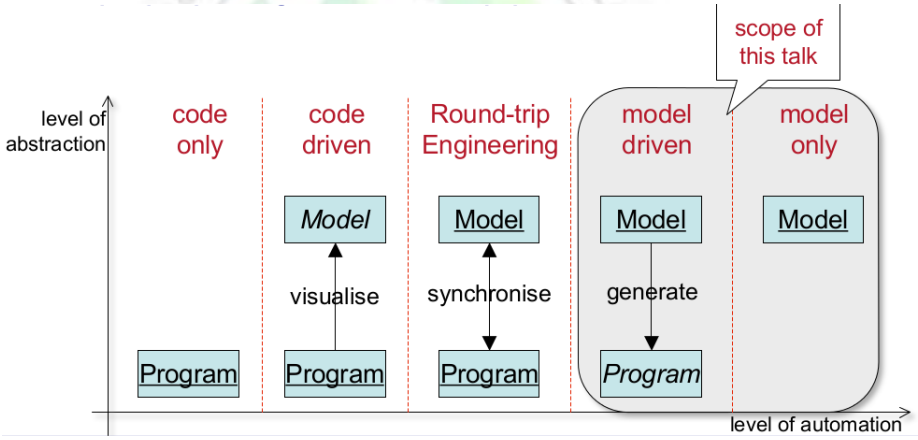


FIGURE 2.2: Contexte de l'IDM (MENS, 2008)

Cette vision de développement basée sur « tout est modèle » est encadrée par quelques concepts. En effet, les principaux travaux de modélisation et méta-modélisation ont été effectués par les groupes de travail des différents organismes de normalisation (OMG (**?**), UML (Booch *et al.*, 1994), graphes conceptuels (Bézivin et Gerbée, 2001)) et par les groupes intéressés par l'échange de modèle (Bézivin, 1998), ou la transformation de modèles (Bézivin *et al.*, 1995). Les besoins de langage commun et les besoins d'échange d'information entre les outils de modélisation ont conduit à une réflexion sur les objets de modélisation et aussi sur

1. http://www.omg.org/ [consulté le 15/03/2018]

leurs méta-modèles. Les différentes définitions présentées dans cette section sont établies en se référant principalement aux travaux de (Bézivin et Gerbée, 2001; Kleppe *et al.*, 2003) et évidemment la spécification MDA (Diaw *et al.*, 2010; Favre *et al.*, 2006) de l'OMG. Nous nous référons également aux travaux de synthèse sur l'IDM proposés par (Combemale, 2008) et (Diaw *et al.*, 2010).

Depuis plusieurs années, il y a consensus (G., 1997; Booch *et al.*, 1994; Bézivin, 1998) sur une architecture à quatre niveaux de modélisation :

— **M0 : donnée**

— **M1 : modèle**

— **M2 : méta-modèle**

— **M3 : méta-métamodèle**

— **le méta-métamodèle (M3)** représente le niveau le plus abstrait, il représente le langage de définition des méta-modèles. Il définit des notions de base qui vont permettre la représentation de tous les autres niveaux ainsi que lui-même. Des exemples sont ***Méta-class, Méta-attribut, Méta-Opération*** dans le cas de UML ; MétaEntité avec (méta)attribut et méta-Relation avec (méta)attribut pour CDIF ; et concept, relation conceptuelle et graphe pour les graphes conceptuels.

— **Le métamodèle(M2)** définit le langage de représentation ou formalisme des modèles. Des exemples sont Classe, Attribut, Opération pour UML ; Entité, Attribut, Relation pour le formalisme Entité-relation ; et Type de concept, et Type de relation pour les graphe conceptuels.

— **Le modèle(M1)** définit le langage de représentation du domaine étudié. Des exemples sont employés , organisation, Client, Mission.

Exemple de compréhension :

Afin de mieux comprendre la signification de ces 4 niveaux d'abstraction nous allons prendre l'exemple simple d'une carte topographique (voir la figure 2.4). La vue aérienne de ENS de Constantine est une vision réelle de l'école (M0) à un instant donné. C'est le système à modéliser. Il faut maintenant établir le modèle (M1). Ici le choix s'est porté sur un modèle type carte topographique, en Ingénierie Dirigé par les modèles on parle de choix du formalisme.

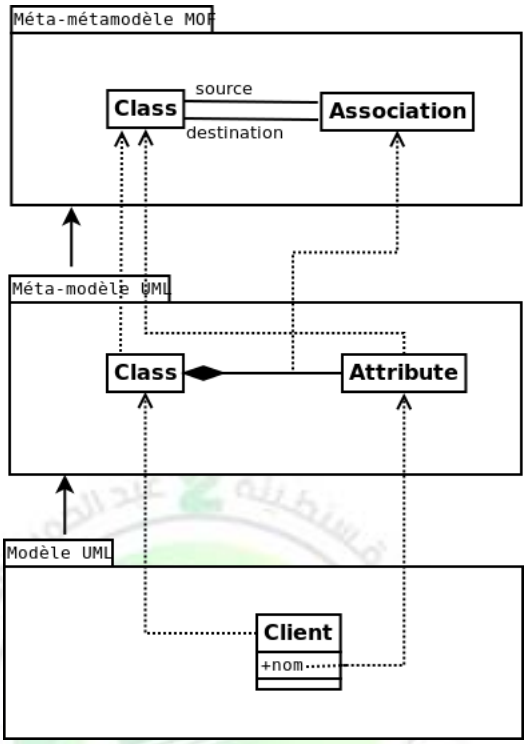


FIGURE 2.3: Relation entre modèle, méta-modèle et méta-métamodèle

Le modèle aurait put être une carte thermique ou géologique. C'est donc une des façons de modéliser l'école. Cette carte, pour être comprise par tous se doit d'être conforme à sa légende (M2). C'est le méta-modèle des cartes topographiques. L'ensemble des cartes topographiques sont conformes à une seule et même légende. Tous les concepts d'une carte topographique sont présents dans la légende (le méta-modèle). Si la légende est modifiée, l'ensemble des instances du méta-modèle (l'ensemble des modèles, l'ensemble des cartes topographiques) est modifié. Imaginons maintenant une carte thermique de l'école, il est évident que la carte ne sera pas la même, la légende non plus, pourtant les deux légendes auront des points communs. C'est le méta-méta-modèle (M3), Ce lui qui décrit les concepts clef présents dans les méta-modèles. Pour construire une légende il faut : du texte,des couleurs,...Toutes les légendes sont conformes à ce méta-méta-modèle. Il est indispensable que ce niveau M3 soit conforme à lui-même, afin d'éviter un empilement des niveaux. Ici du texte permet de définir le méta-méta-modèle. Il est donc conforme à lui même.

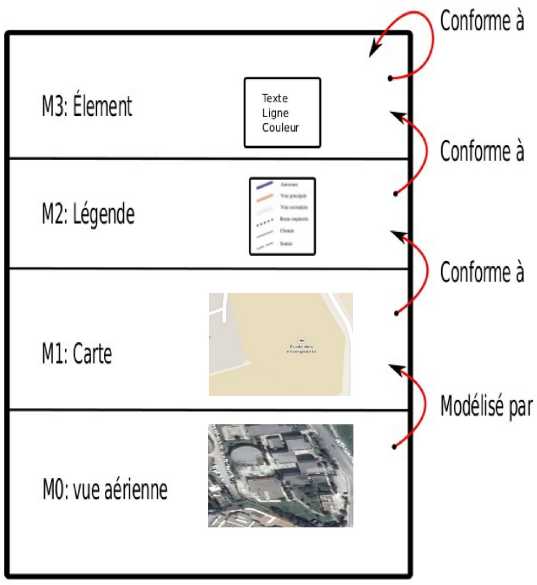


FIGURE 2.4: Exemple de modélisation

2.1.1 Concept de modèle

Un modèle est une représentation abstraite et simplifiée(i.e. qui exclut certains détails) d'une entité qui peut représenter un processus ou un système du monde réel en vue de le décrire, de l'expliquer ou de le prévoir (Kleppe *et al.*, 2003).

Un modèle est compris aussi comme une abstraction d'un système (Pelfresne, 2003) dans le sens où il reflète un ou plusieurs aspects [2] d'un système ou une partie de ce dernier.

En d'autre terme, un modèle reflète en quelques sorts ce que le concepteur voit important pour la compréhension et la prédiction du future système. De plus, un modèle abstrait du système rend la possibilité de faire ***une réflexion*** sur celui-ci.

Maintenant et d'après ces définitions, il est claire d'identifier la première relation majeure de l'IDM, entre le modèle et le système qu'il représente, appelée « ***représentation De*** ». La figure 2.5 explique cette relation avec un exemple.

On peut classer les modèles selon trois catégories et cela dû au langage de modélisation utilisé pour les représenter :

2. Par exemple le comportement du système peut être : discret, stochastique, temporel,... etc.

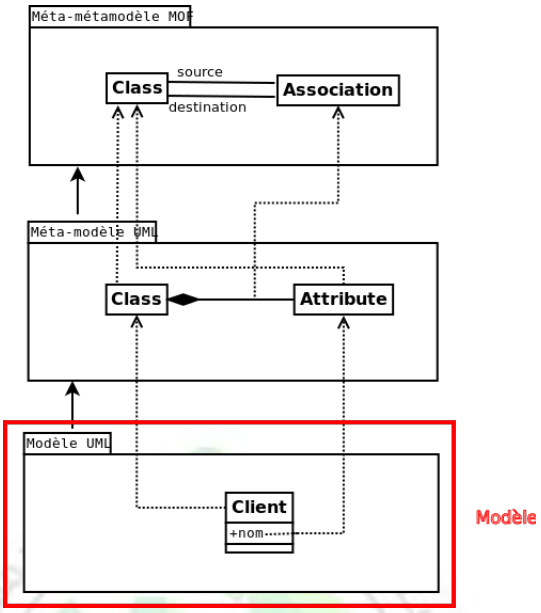


FIGURE 2.5: Concept de modèle

1. ***Modèle formelle*** : il est représenté par des expressions mathématique. Généralement ce type est décrit par : les logiques (première ordre, d'ordre supérieur,...), les automates (automates états transitions, réseau de pétri,...), les algèbres, ...etc.
2. ***Modèle semi-formelle*** : il est représenté par des diagrammes, graphes et texte. Par exemple, ce type est décrit par les méthodes/langages comme : UML, Merise, DFD, Entité-Association, SADT, ...etc.
3. ***Modèle informelle*** : il est représenté par des phrases de la langue courant. Par exemple : cahier de charge en langue arabe.

2.1.2 Concept de Méta-modèle

Un métamodèle est un langage de modélisation. Il définit les concepts ainsi que les relations entre concepts nécessaires à l' expression de modèles.

La notion de métamodèle conduit à l' identification d' une seconde relation, liant le modèle et le langage utilisé pour le construire, appelée « conformeA » . Ces deux relations permettent ainsi de bien distinguer le langage qui joue le rôle de système, du (ou des) métamodèle(s) qui jouent le rôle de modèle(s) de ce langage (Piel, 2007). La figure 2.6 explique ce concept.

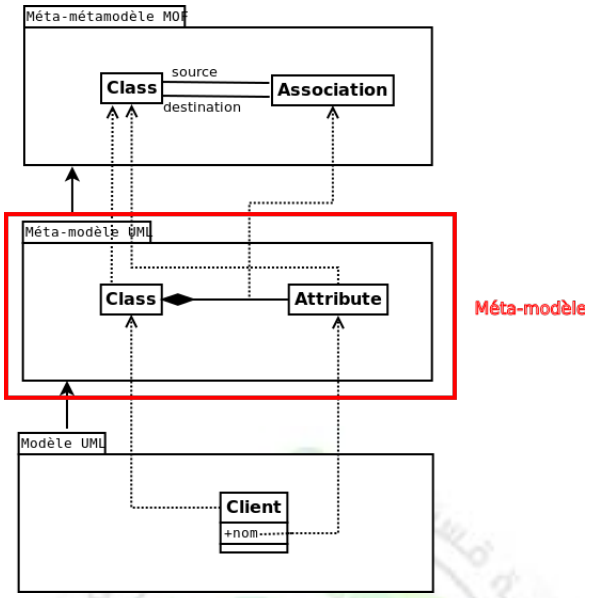


FIGURE 2.6: Concept de méta-modèle

Selon (Beugnard *et al.*, 2014) les liens entre le modèle et son métamodèle peuvent être classer en 11 situations et cela en rapport avec l'ordre dans lequel les modèles apparaissent ou sont reliés dans la démarche (Voir figure 2.7).

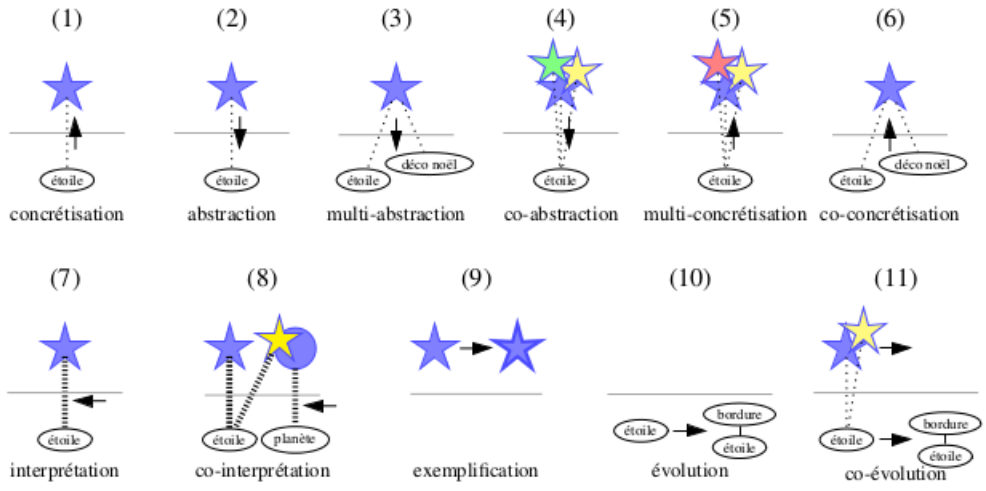


FIGURE 2.7: Situations de modélisation (haut : niveau instance, et bas : niveau méta) (Beugnard *et al.*, 2014)

1. Un méta-modèle existe, on cherche produire un modèle [***concrétisation***].
2. Un modèle existe, le travail consiste à trouver un méta-modèle [***abstraction***].
3. Un modèle existe, il faut trouver plusieurs méta-modèles [***multi-abstraction***].
4. Des modèles existent, il faut élaborer un méta-modèle [***co-abstraction***].
5. Un méta-modèle existe, le travail consiste à construire plusieurs modèles [***multi-concrétisation***].
6. Des méta-modèles existent, le travail consiste à construire un modèle [***co-concrétisation***].
7. Un modèle et un méta-modèle existent, il faut les relier [***interprétation***].
8. Des modèles existent, des méta-modèles existent, le travail consiste à les relier[***co-interprétation***].
9. Un modèle existe, le travail consiste à construire un autre modèle (sans aucun méta-modèle) [***exemplification/extension***].
10. Un méta-modèle existe, le travail consiste à construire un autre méta-modèle (sans aucun modèle) [***évolution/extension***].
11. Un méta-modèle existe avec plusieurs de ses modèles conformes, le travail consiste à faire évoluer le méta-modèle (cas précédent) en adaptant (ou non) ses modèles [***co-évolution***].

2.1.2.1 Langage de contraintes

Afin de vérifier la conformité des modèles à leurs métamodèles, les contraintes OCL (Object Constraint Langage) sont considérés comme un moyen efficace pour la concrétiser. Cette conformité permet d'exprimer toutes les questions relatives à la préservation de la sémantique du modèle traduit.

Il faut noter, pour toute transformation ; c'est au concepteur des règles de transformation d'imposer n'importe quelle condition qui doit être respectée.

Pratiquement, le langage OCL est disponible dans la plateforme EMF/Eclipse à travers l'outil ***OclInEcore***[3]. Dans le cadre de cette thèse, l'utilisation de ce langage ne constituée pas une priorité.

3. https://wiki.eclipse.org/OCL/OCLinEcore [consulté le 15/03/2018]

2.1.3 Concept de Méta-métamodèle

Un méta-métamodèle est un langage de métamodélisation utilisé pour exprimer les méta-modèles. C'est le cas par exemple pour la proposition de MDA (Model Driven Architecture) (Favre *et al.*, 2006; Diaw *et al.*, 2010).

La métamodélisation est considérée comme l'activité qui consiste à représenter les langages de modélisation par le biais des modèles. En effet, cette activité a permis l'émergence de la notion de DSMLs (Domain Specific Modeling Languages, littéralement langages de modélisation spécifiques à un domaine).

Nous disposons à ce jour de nombreux langages et environnements pour la métamodélisation, on peut citer par exemple : MOF (OMG, 2006) (Meta Object Facilities) et ses variations (EMOF : pour Essential MOF, CMOF -pour Complete MOF) (Steinberg *et al.*, 2008), Eclipse-EMF/Ecore (BUDINSKY *et al.*, 2003), GME/MetaGME (LEDECZI *et al.*, 2001), AMMA/KM3 (JOUAULT et Bézivin, 2006; MAIER, 2005), XMF-Mosaic/Xcore (CLARK *et al.*, 2004), le standard de l'OMG MOF ou Kermeta (MULLER *et al.*, 2005).

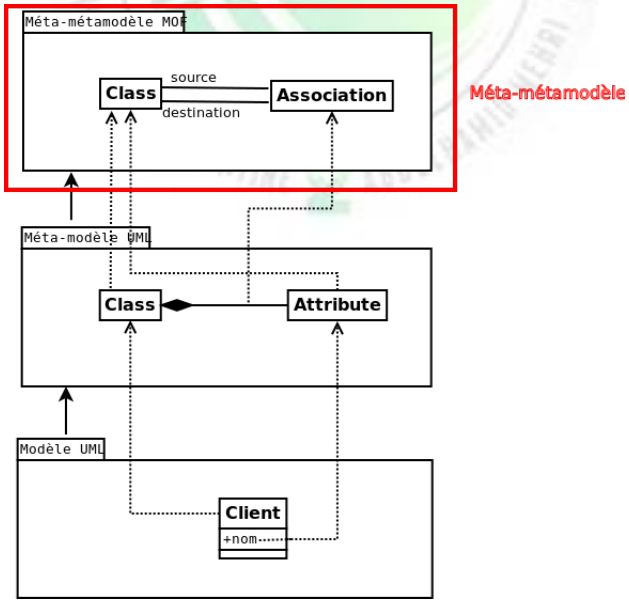


FIGURE 2.8: Concept de Méta-métamodèle

2.1.4 Concept de transformation de modèle

Selon (MENS *et al.*, 2005) : « Une transformation est la génération automatique d'un modèle cible à partir d'un modèle source, selon une définition de transformation. Une définition de transformation est un ensemble de règles de transformation qui décrivent comment un modèle exprimé dans le langage source peut être transformé en un modèle du langage cible. Une règle de transformation est la description de comment un ou plusieurs éléments du langage source peuvent être transformés en un ou plusieurs éléments du langage cible. »

Donc, la transformation de modèle consiste à manipuler les modèles à travers des transformations, ces transformations assurent le passage d'un ou plusieurs modèles sources (représenté(s) dans un niveau d'abstraction donné) à un ou plusieurs modèles cibles (représenté(s) dans un nouveau échelle d'abstraction). Le modèle transformé est appelé modèle source et le modèle résultant de la transformation est appelé modèle cible .

La transformation de modèle se réalise donc selon trois principes de base :

— un méta-modèle source qui définit la syntaxe des modèles instances du formalisme source ;

— un méta-modèle cible qui définit la syntaxe des modèles instances du formalisme cible ;

— une définition de transformation, qui précise les règles de mise en correspondance syntaxique des éléments de modèle du méta-modèle source vers ceux du méta-modèle cible.

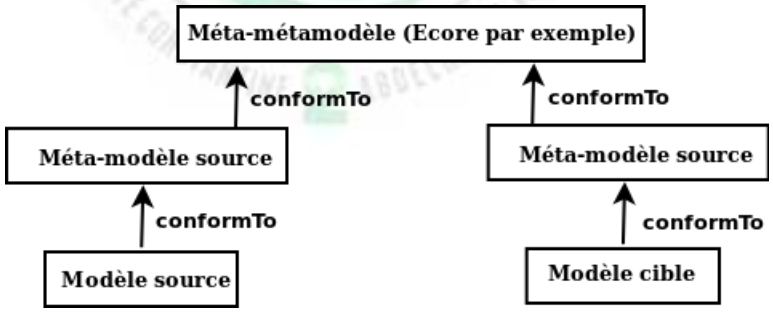


FIGURE 2.9: Concept de transformation de modèle

Historiquement, les travaux réalisés dans le domaine de la transformation ne sont pas récents et peuvent être chronologiquement classés selon plusieurs générations en fonction de la structure de donnée utilisée pour représenter le modèle (Bézivin, 2003) :

— **Génération 1** : Transformation de structures séquentielles d'enregistrement.Dans ce cas un script spécifie comment un fichier d'entrée est réécrit en un fichier de sortie (p. ex. des

scripts Unix, AWK ou Perl). Bien que ces systèmes soient plus lisibles et maintenables que d'autres systèmes de transformation, ils nécessitent une analyse grammaticale du texte d'entrée et une adaptation du texte de sortie (GERBER *et al.*, 2002; Bézivin, 2003).

— **Génération 2** : Transformation d'arbres. Ces méthodes permettent le parcours d'un arbre d'entrée au cours duquel sont générés les fragments de l'arbre de sortie. Ces méthodes se basent généralement sur des documents au format XML et l'utilisation de XSLT [4] ou XQuery [5].

— **Génération 3** : Transformation de graphes. Avec ces méthodes, un modèle en entrée (graphe orienté étiqueté) est transformé en un modèle en sortie. Ces approches visent à considérer « l'opération » de transformation comme un autre modèle conforme à son propre métamodèle (lui-même défini à l'aide d'un langage de métamodélisation, par exemple le MOF).

2.1.5 Classification des approches de transformation

Les approches de transformation de modèle ont été classées selon plusieurs axes. Chaque axe mène à une classification particulière. La classification des approches de transformation proposée par ***Czarnecki et Helsen*** (Czarnecki et Helsen, 2003) se base sur les techniques de transformation utilisées dans les approches et les facettes qui les caractérisent. Selon cette classification on peut distinguer deux types de transformation de modèles : les transformations de type modèle vers code et les transformations de type modèle vers modèles. Généralement, le premier type de transformation peut être vu comme un cas particulier du deuxième type, nous avons seulement besoin de fournir un méta-modèle pour le langage de programmation cible.

Cependant, pour des raisons pratiques de réutilisation de la technologie des compilateurs existants, souvent, le code est simplement généré en tant que texte, qui est ensuite introduit dans un compilateur. Pour cette raison, nous distinguons entre la transformation de type modèle vers code et la transformation de type modèle vers modèle.

4. XSL (eXtensible StyleSheet Language) Transformation, cf. `http://www.w3.org/TR/xslt` [consulté le 15/03/2018]
5. An XML Query Language, cf. `http://www.w3.org/TR/xquery/` [consulté le 15/03/2018]

2.1.5.1 Transformations de type modèle vers modèle (M2M)

Les transformations de type modèle vers modèle (c-à-d d'une syntaxe abstraite vers une syntaxe abstraite) consistent à transformer un modèle source en un modèle cible. Ces modèles peuvent être des instances de différents méta- modèles. Elles offrent des transformations plus modulaires et faciles à maintenir. Dans les cas de MDA par exemple où on trouve un grand espace d'abstraction entre PIMs (Platform Independent Model) et PSMs (Platform Specific Model), il est plus facile de générer des modèles intermédiaires qu'aller directement vers le PSM cible. Les modèles intermédiaires peuvent être utiles pour l'optimisation ou bien pour des fins d'enlever les bogues. De plus, les transformations de type modèle vers modèle sont utiles pour le calcul des différentes vues du système et leurs synchronisation.

Dans cette catégorie on trouve généralement trois types de transformation :

(a) ***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.

(b) ***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.

(c) ***Les transformations hybrides*** : elles combinent entre transformation horizontale et verticale. Ce type de transformation est notamment utilisé par les compilateurs, qui effectuent des optimisations du code source avant de générer le code exécutable.

Si les deux méta-modèles (source et cible) correspondant respectivement aux modèles (source et cible) sont ***identiques***, on parle d'une transformation endogène. Dans le cas contraire (méta-modèles différents) la transformation est exogène (GERBER *et al.*, 2002). La figure 2.10 montre les principaux types de transformations dans cette catégorie.

Dans cette catégorie, on distingue plusieurs approches qui offrent des mécanismes pour sa mise en œuvre :

— **Les approches manipulant directement les modèles** : Dans ces approches, une API permet de manipuler la représentation interne des modèles. Elles sont en général implémentées comme un ***framework*** orienté objet qui fournit une infrastructure pour organiser les

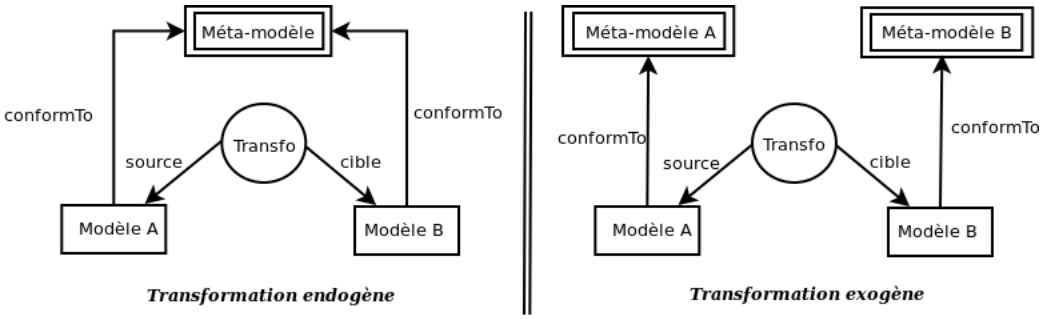


FIGURE 2.10: Les principaux types de transformations M2M

transformations. La combinaison JMI (Java Metadata Interface) et Java est souvent utilisée dans la mise en œuvre de cette approche.

— **Les approches relationnelles** : Le principe de base de cette approche consiste à établir une relation entre les éléments des modèles sources et cibles. Ces relations seront spécifiées à l'aide de contraintes. Elles sont purement déclaratives et leur spécification n'est pas exécutable. La norme ***QVT Relational*** répond à cette approche implantée par quelques outils comme ***medini QVT***.

— **Les approches basées sur la transformation de graphe** : Les modèles et les méta-modèles possèdent souvent une représentation graphique apparentée à un graphe. Un modèle, dans ce cas, peut être considéré comme un graphe étiqueté, contraint par des règles de cohérence essentiellement définies comme un méta méta-modèle MOF. Les techniques de ré-écriture de graphes et de transformation de graphes peuvent être appliquées pour des transformations de modèles. D'une manière générale, les systèmes de ré-écriture de graphes combinent une notation graphique et une notation textuelle afin d'exprimer ces transformations. Cette catégorie d'approches est mise en œuvre dans : VIATRA, UMLX et BOTL.

— **Les approches dirigées par la structure** : Dans ces approches, la transformation se réalise en deux phases : la première crée la structure hiérarchique du modèle cible et la seconde vient compléter le modèle en définissant les valeurs des attributs et des références. Comme exemples d'approches de cette catégorie, on cite OptimalJ, Interactive Objects and Project Technology (IOPT).

— **Les approches hybrides** : Elles combinent plusieurs approches précédentes. Le langage de transformation de règles est une combinaison d'approches déclarative et impérative.

ATLAS Transformation Language (ATL), Kermeta et ModTransf sont des exemples de cette catégorie.

2.1.5.2 Transformations de type Modèle vers code (M2C)

Dans cette catégorie on cherche à balayer le modèle source pour générer en sortie un code qui est généralement décrit dans un langage de programmation ou de structuration. Ici, on parle de transformation d'une syntaxe abstraite vers une syntaxe concrète. Dans ce type, on distingue deux approches : les approches basées sur le principe du visiteur (Visitor-based approach) et celles basées sur le principe des patrons (Template-based approach).

(a) ***Approche basée sur le visiteur (Visitor-based)*** : approche de base pour la génération de code, elle consiste à fournir un mécanisme de visiteur pour traverser la représentation interne d'un modèle et créer le code. On peut citer comme exemple le framework Jamda qui fournit un ensemble de classes pour représenter les modèles UML, une API pour manipuler les modèles, et un mécanisme de visiteur pour générer le code.

(b) ***Approche basée sur les templates (Template-based)*** : Actuellement, la majorité des outils MDA disponibles supportent cette approche. La structure d'un ***template*** ressemble au code à générer, dans un ***template*** il n'y a pas de séparation syntaxique entre le LHS (Left Hand Sides) et le RHS (Right Hand Sides). Le LHS utilise une logique exécutable pour accéder au modèle source, le RHS combine des patrons non typés et une logique exécutable (la logique : code ou requêtes déclaratives). Parmi les outils basés sur ce principe, on peut citer : JET, Codagen Architect , ArcStyler, AndroMDA, OptimalJ et XDE (les deux derniers outils fournissent aussi la transformation modèle vers modèle).

Remarque :

la transformation « modèle-vers-code » s'effectue généralement d'une façon semi-automatique et assistée ; c'est à dire la génération se produit progressivement de manière partielle afin d'arriver à une génération complète du code.

2.1.6 Standards et langages pour la transformation de modèle

De nombreux langages sont à ce jour disponibles pour écrire des transformations de modèle de génération 3. On retrouve d'abord les langages généralistes qui s'appuient directe-

ment sur la représentation abstraite du modèle. On citera par exemple l'API d'EMF (BUDINSKY *et al.*, 2003) qui, couplée au langage Java, permet de manipuler un modèle sous la forme d'un graphe. Dans ce cas, c'est à la charge du programmeur de faire la recherche d'information dans le modèle, d'expliciter l'ordre d'application des règles, de gérer les éléments cibles construits, etc.

Afin d'abstraire la définition des transformations de modèle et rendre transparent les détails de mise en œuvre, l'idée a été de définir des DSML (Domain Specific Modeling Language) dédiés à la transformation de modèle (JOUAULT et KURTEV, 2005). Cette approche repose alors sur la définition d'un méta-modèle dédié à la transformation de modèle et d'outils permettant d'exécuter les modèles de transformation. Nous citerons par exemple ATL que nous utilisons tout au long de cette thèse. Il s'agit d'un langage hybride (déclaratif et impératif) qui permet de définir une transformation de modèle à modèle (appelée Module) sous la forme d'un ensemble de règle. Il permet également de définir des transformations de type modèle vers texte (appelée Query). Une transformation prend en entrée un ensemble de modèles (décrits à partir de méta-modèles en Ecore ou en KM3).

Afin de donner un cadre normatif pour l'implantation des différents langages dédiés à la transformation de modèle, l'OMG a défini le standard QVT (Query/- View/Transformation) (OMG, 2008). Le méta-modèle de QVT est conforme à MOF et OCL (G, 2003) et utilisé pour la navigation dans les modèles. Le méta-modèle fait apparaître trois sous-langages pour la transformation de modèles (cf. figure 2.11), caractérisés par le paradigme mis en œuvre pour la définition des transformations (déclaratif, impératif et hybride).

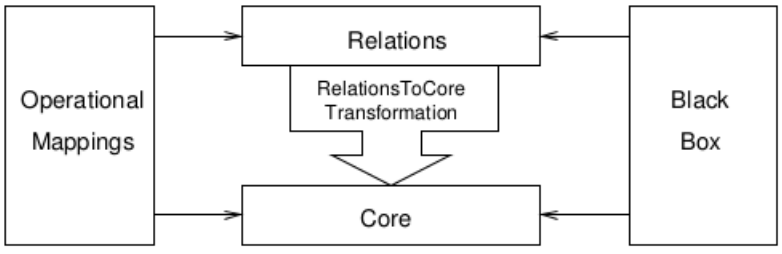


FIGURE 2.11: Architecture du standard QVT (OMG, 2008)

2.1.6.1 Preuve de la transformation

Dans le contexte de la transformation de modèle une question importante est souvent posée *"Est ce que la transformation est correcte ? !"* c'est à dire est ce la transformation produira comme résultat le modèle attendu ?. Par exemple : dans le scénario classique de la transformation [6] d'une Class-UML vers une Table ; la transformation est considérée correcte lorsque tous les attributs de la classe seront transformés en des colonnes dans la table correspondante. En d'autre terme, il n'aura pas une perte d'information lors de l'exécution de la transformation.

Comme l'indique (Rahim et Whittle, 2015) les techniques de la vérification formelle sont sollicité pour répondre à la question précédente. En effet, ces techniques sont classé en trois catégories (Amrani *et al.*, 2015) : approches basées sur le ***test***, approches basées sur le ***model-checking*** et approches basées sur la ***preuve de théorème***.

Nous devons noter que, c'est difficile de trouver une définition de référence qui précise le concept de la correction d'une transformation. Cependant, (Lucio *et al.*, 2016) on définit plusieurs niveaux sur lesquels la transformation doit être prouvée :

(i) ***niveau syntaxique*** : préservation structurelle, vérification du type, etc ;

(ii) ***niveau sémantique*** : équivalence, cohérence, etc ;

(iii) ***niveau d'exécution*** : terminaison, confluence, etc.

Remarque :

Nous estimons que la démarche la plus efficace pour vérifier une transformation c'est celle qui couvre les trois niveaux : syntaxique, sémantique et d'exécution. Par conséquent, cette efficacité a une influence directe sur la complexité de la preuve.

Exemple de compréhension

Pour exploiter la technique du *preuve de théorème* dans le cadre la preuve de transformation, nous proposons un framework [7] qui offre la possibilité d'assister les transformation de type modèle-vers-modèle par l'assistant de preuve Coq [8] :

6. https://www.eclipse.org/atl/atlTransformations/ [consulté le 15/03/2018]
7. un article soumis à la revue ***Science of Computer Programming -Elsevier*** en janvier 20118, M.E Hamdane, K. Berramla, A. Chaoui *"Using Coq to prove correctness of M2M transformations : ER2REL case study"*
8. http://coq.inria.fr [consulté le 15/03/2018]

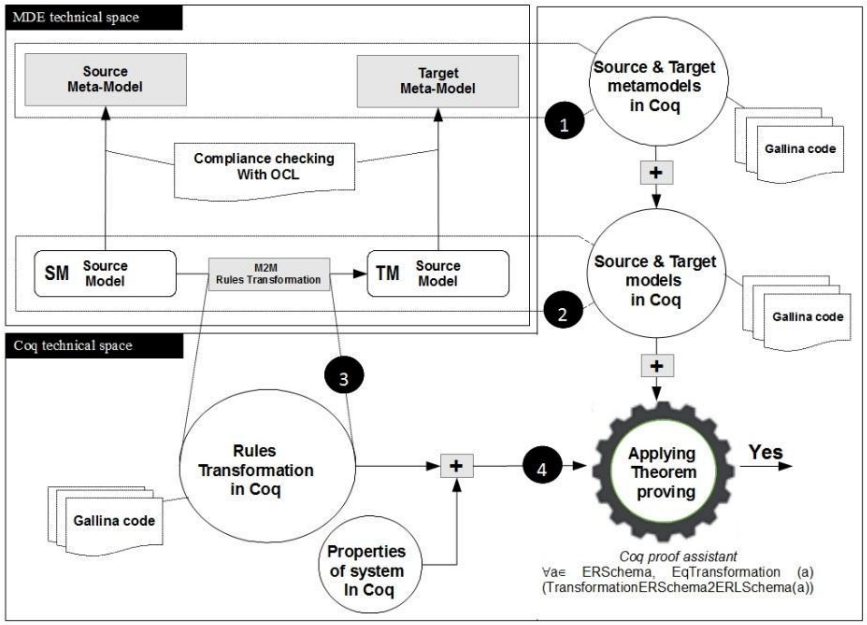


FIGURE 2.12: Assistance des transformations par la preuve de théorème

Le framework est structuré en deux espaces : l'espace technique MDE et l'espace technique Coq. Le premier représente une transformation M2M classique. Il est composé de deux méta-modèles : source et cible. À partir du méta-modèle source, nous pouvons ensuite instancier un modèle source (SM). Puis, après une exécution des règles de transformation, le système génère automatiquement le modèle cible (TM) conforme au méta-modèle cible. La relation entre le méta-modèle et leur modèle peut être contrôlée par un ensemble de conditions exprimées dans le langage des contraintes OCL.

Le deuxième espace concerne la traduction de tout le processus de transformation dans l'assistant de preuve Coq. En effet, la première étape ; traduire les méta-modèles source et cible dans l'outil Coq à travers le langage "Gallina". La même tâche est effectuée au niveau des modèles et des règles de transformation (étapes 2 et 3). L'étape 4, permet de définir un ensemble de propriétés qui expriment des exigences spécifiques à vérifier. Ces propriétés sont combinées avec les étapes précédentes afin de construire un système de preuve en "Coq". Ce système doit répondre à la question suivante : $\forall a \in InputModel, EqTransformation(a)(TransformationInputModel2OutputModel(a))$

La formule proposée signifie : " y a-t-il une équivalence sémantique entre le modèle source et cible ?". Pour répondre à cette question en Coq, il faut définir une démonstration en se basant sur des tactiques, de propositions et d'axiomes (Voir Appendix D).

2.1.7 Discussion : Transformation de modèle où Transformation de graphe ?

Historiquement, la transformation de graphes a été introduite en informatique à partir de la fin des années 60 et début des années 70 (Pfaltz et Rosenfeld, 1969; Montanari, 1970; Pratt, 1971). Le principe de base est la modification de graphes par des règles de transformation. Chaque application d'une règle sur un graphe mène à une transformation de ce graphe. Rigoureusement, le terme « transformation de graphes » désigne une séquence d'application de règles de transformation sur un graphe donné. Notons que dans la littérature, on distingue parfois les notions de transformation de graphes et de grammaire de graphes. Les grammaires de graphes sont proches des grammaires de ***Chomsky*** sur les chaînes, il s'agit d'un couple muni d'un ensemble de règles et d'un graphe de départ. Cette technique est basée principalement sur les grammaires de graphes. On peut citer quelques outils qui s'inscrivent dans cette technique : PROGRES (Andy Schürr, 1999), AGG(Taentzer, 2003) , Fujaba (Burmester *et al.*, 2004), Moflon(Amelunxen *et al.*, 2008) , ATOM3 (de Lara et Vangheluwe, 2002), VIATRA2 (Varró *et al.*, 2006), GreAT (Balasubramanian *et al.*, 2006),... etc.

Par contre l'IDM et la notion de transformation de modèle ont été introduits par l'organisme de standardisation OMG lors quelle a rendu publique son initiative MDA (Model Driven Architecture) (Favre *et al.*, 2006), qui peut être vue comme une restriction de l'IDM à la gestion de l'aspect particulier de dépendance d'un logiciel à une plateforme d'exécution. Cette technique est basée principalement sur les techniques de métamodélisation comme (Steinberg *et al.*, 2008; Jouault et Kurtev, 2006) : MOF, UML, BPMN, EMF et GEF, QVT , ATL XML,XMI,XSLT,... etc. Dans le tableau suivant, nous essayons de tracer une comparaison entre les deux techniques :

TABLE 2.1: Comparaison entre Transformation de Graphes et Transformation de Modèles

	Transformation de graphe	**Transformation de modèles**
Principe	basée sur la théorie des graphes	basée sur la théorie de l'objet et précisément UML
Concepts	Graphe, re-écriture, grammaire	modèle, transformation, modèle-vers-modèle , modèle-vers-code
Outils	— AGG + Tiger Eclipse plug-ins, — Fujaba, Moflon, ATOM3, VIATRA2, GrGen.Net, — MoTMoT, GReAT, GROOVE, ...	— OMG : MOF, UML, OCL, BPMN, — Eclipse : Ecore, EMF, GEF, — W3C : XML, XMI, XSLT, ...
Avantage	très expressive	très pratique

La combinaison entre ces deux techniques est-elle possible ? la réponse est *"**oui**"* du fait que la transformation de graphes est considère comme une technique pour réaliser la transformation de modèle (Ehrig *et al.*, 2005) où tout simplement la transformation de graphe est un cas spécial de la transformation de modèle.

2.2 Intégration des Méthodes formelles à IDM

La question de l'utilisation des techniques formelles dans l'approche IDM fait l'objet de plusieurs discussions (DESEL, 2002; SCHMIDT, 2006; Hillah, 2009; Kerkouche, 2011). En effet, l'utilisation des techniques de vérification au moment de la phase de modélisation est essentielle pour le développement de systèmes complexes, en particulier pour les systèmes critiques où les questions liées à la sûreté/fiabilité sont fondamentales. En d'autre terme, l'utilisation des méthodes formelles dans l'ingénierie des systèmes devient indispensable, surtout dans les phases amont du développement de ce type de systèmes.

Pour le développement dirigé par les modèles, trois critères de qualité prévalent (Hillah, 2009) :

— le premier concerne *la séparation des préoccupations*. La conception des modèles de développement est de ce point de vue caractérisée selon deux types de préoccupations : la définition des modèles indépendants de toute plate-forme technologique d'une part **(Platform Independent Model)**, et des modèles spécifiques à des plate-formes d'exécution particulières **(Platform-Specific Model)** d'autre part.

  Les modèles sont caractérisés selon les aspects architecturaux et comportementaux du système à réaliser. Ces aspects sont traités dans plusieurs modèles (ou points de vue) du

même système, à différents niveaux d'abstraction, selon la finalité de la phase de développement en cours : analyse, génération de code, tests, déploiement, etc. ;

— le deuxième critère concerne *la satisfaction des propriétés comportementales attendues*, décrites à travers la spécification du système. La détection de comportements inattendus, potentiellement indésirables est très bénéfique dans ce contexte. La vérification formelle fournit les moyens de satisfaire ce critère ;

— enfin, le troisième critère concerne *l'automatisation du processus de développement du système*, si possible sur l'ensemble de son cycle de vie. Les démarches d'ingénierie dirigées par les modèles, axées sur les meilleures pratiques de développement et d'outillage de niveau industriel, satisfont ce critère.

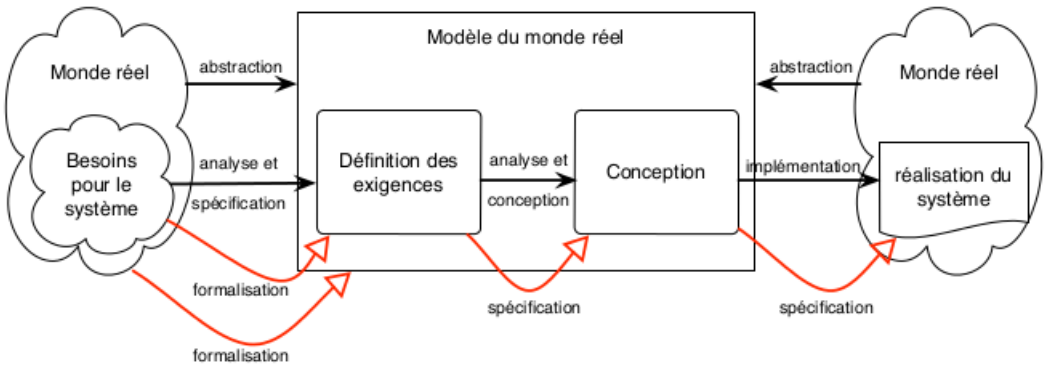


FIGURE 2.13: Aperçu sur le développement dirigé par les modèles (DESEL, 2002)

La figure 2.13 illustre la possibilité d'intégration des méthodes formelles au sein de IDM. Il faut lire cette figure de gauche à droite. Le modèle du monde réel traduit une abstraction de celui-ci. Le système à réaliser (à droite) devra au final tourner dans le monde réel. Pour y arriver il faut d'abord, à travers une activité de formalisation, spécifier les exigences du système. La conception du système intervient ensuite. Cette étape est alimentée par la spécification des exigences. Elle donne lieu à un modèle qui servira de spécification pour la réalisation du système. Abstraction est faite des détails d'implémentation dans ce modèle.

De la même façon, pour analyser la fiabilité du comportement des systèmes AADL, il faut disposer de spécifications précises et cohérentes. Or, dans ce type de projet :

— les multiples modèles de développement, définis et préconisés ne sont souvent pas mis en œuvre de manière appropriée, et subissent des entorses ;

— les spécifications sont souvent incomplètes et ambiguës ;

— ces spécifications sont souvent décrites sous forme de construction qui ne disposent pas de véritable sémantique formelle, c-à-d. dont la sémantique n'est pas strictement mathématiquement fondée de manière à éviter les ambiguïtés dans leur interprétation ;

— les techniques de test et de simulation manquent d'exhaustivité.

Donc, est il clair que cette combinaison est possible et rentable du faite que la modélisation par l'approche IDM vise à produire des modèles et les techniques formelles sont utilisées comme support pour s'assurer que les modèles en cours de développement répondent aux exigences attendus et garantir ainsi que certaines propriétés liées à son comportement sont validés. Par conséquent, l'intégration des techniques formelles dans le développement de ce type de systèmes permettent de superviser le processus de développement et d'offrir ainsi un seuil de garantie sur la qualité des modèles en cours de développement grâce à leurs vérification automatique.

Conclusion

Dans ce chapitre, nous avons présenté les principes de l'ingénierie dirigée par les modèles et nous avons montré sa vision pour aboutir à une modélisation qui cadre tous les aspects d'un système complexes tout en offrant des outils pour maitriser cette complexité. L'idée innovante de l'IDM est pas vraiment la notion de modèle ou de transformation de modèles car ces notions existent déjà avant l'apparition de cette approche. Cependant, c'est grâce à la modélisation par plusieurs niveaux d'abstraction (via la notion de méta-modélisation) toute en offrant la possibilité d'automatiser le passage entre ces niveaux ; qui rend cette vision bien adaptée pour cerner la complexité des systèmes actuels.

Comme l'indique R.F. Paige et al. (Paige *et al.*, 2017), *"IDM a été étudiée et appliquée pendant de nombreuses années, et elle a évoluée vers un état où elle a été utilisée avec succès dans une variété de projets importants. Elle est maintenant à un stade de maturité"*. D'un autre côté, l'intégration des techniques formelles à l'IDM permet de donner à celle-ci une dimension mathématique rigoureuse facilitant ainsi l'analyse comportementale des modèles en cours de développement.

Dans le troisième chapitre nous abordons une technique d'analyse formelle de type model-checking ainsi que le modèle d'analyse des automates temporisés.

CHAPITRE 3

MODEL-CHECKING ET AUTOMATE TEMPORISÉ

Les enfants ont plus besoin de modèles que de critiques." *Joseph Joubert, Pensées*

Sommaire

Introduction

Afin de pouvoir exploiter la technique de la vérification formelle par model-checking dans le cadre de notre travail, on a besoin de comprendre ses principes ainsi que sa relation avec les automates temporisés. En effet, dans un premier temps, on décrira les différents éléments sur lequel le processus de vérification par model-checking est construit. Cette description est faite de manière progressive, on commence par décrire la notion de modèle, puis la notion de spécification et enfin les logiques utilisées dans le cadre de la vérification formelle par model-checking. On présentera ensuite le formalisme des automates temporisés d'un point de vue syntaxique, puis sémantique. Avant de conclure on présentera une classification des propriétés vérifiables dans un automate temporisé.

3.1 Éléments de la vérification formelle

Le Model-checking et la vérification formelle en générale est une approche s'appuyant sur un raisonnement mathématique qui permet de prouver que la description formelle d'un système satisfait certaines propriétés souhaitées. Plus spécifiquement, la vérification formelle comporte trois étapes globales :

— la modélisation du système,

— la spécification des propriétés attendues du système

— et finalement la preuve que le système modélisé possède bien les propriétés attendues.

Nous définissons la première étape indispensable pour toute vérification d'un système qui n'est autre que sa modélisation formelle. Cette étape vise à décrire clairement et sans ambiguïté, au moyen d'un modèle ou d'un langage, le comportement d'un système.

La vérification de modèles consiste à construire un modèle fini d'un système et vérifier qu'une propriété cherchée est vraie dans ce modèle. Généralement, il y a deux façons 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.

Donc, afin de vérifier un tel système, le concepteur doit (Schnoebelen *et al.*, 1999) :

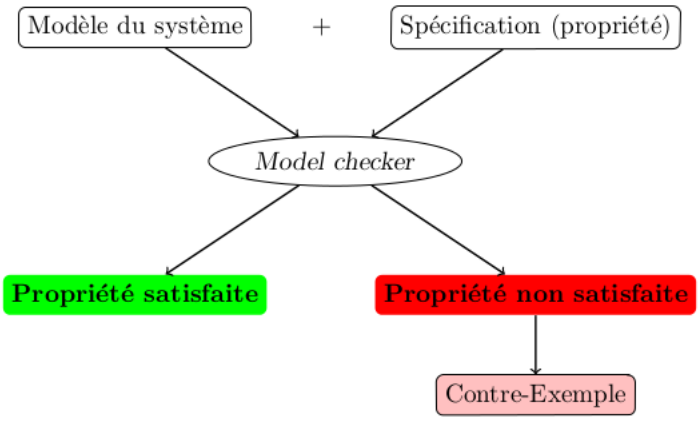


FIGURE 3.1: Principe de model-cheking

— Réaliser un modèle du système qu'il souhaite vérifier. Cette tâche est délicate puisque la plupart du temps, le système est décrit en langue naturelle ou sous forme de programme incompatible avec le formalisme des modèles et il est alors nécessaire de traduire, le plus souvent à la main, cette description.
— Spécifier formellement, par exemple dans un langage logique, les propriétés attendues du système. Nous donnons quelques exemples de propriétés :
  — Le système peut-il tombé dans une situation de blocage ?
  — Après une panne, le signal d'alarme se déclenche-t-il ?
  — Le café est servi dans au plus de 3 secondes à partir de l'introduction d'une pièce de monnaie ?
  — Après une pression sur le bouton d'appel, l'ascenseur arrive-t-il un jour ?
  — Au début d'un choc sur une voiture, l'airbag s'ouvre automatiquement ?
  — . . . etc.

De façon générale, on distingue deux grandes catégories de techniques pour vérifier formellement un système complexe. La catégorie première regroupe les techniques de preuve de théorème ou *"**theorem proving**"* qui sont des démonstrations mathématiques au sens classique du terme où la vérification des propriétés est effectuée par déduction à partir d'un ensemble d'axiomes et de règles d'inférences. La seconde 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 preuve automatique de théorème, consiste à laisser l'ordinateur prouver les propriétés automatiquement, étant données une description du système, un ensemble d'axiomes et un ensemble de règles d'inférences. Cependant, la recherche de preuve est connue pour être un problème non décidable en général (en logique classique), c'est-à-dire que l'on sait qu'il n'existe (et n'existera jamais) aucun algorithme permettant de décider en temps fini si une propriété est vraie ou fausse. Au contraire, le model-checking est complètement automatique et rapide (Schnoebelen *et al.*, 1999). Il produit aussi des contre-exemples qui représentent des erreurs subtiles dans la conception et ainsi il peut être utilisé pour aider le débogage. Dans le cadre de cette thèse, on s'intéresse beaucoup à la catégorie de model-checking.

Le Model-checking est complètement automatique et rapide. De plus, Il produit aussi des contre-exemples qui représentent des erreurs subtiles dans la conception et ainsi il peut être utilisé pour aider le débogage (Hutzler et Klaudel, 2004).

3.2 Principe de Model-checking

L'objectif de la vérification par model-checking est de contrôler certaines propriétés du modèle en développement (Hutzler et Klaudel, 2004). En effet, l'analyse formelle des systèmes par 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 un modèle, 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 $K| = \Phi$. L'intérêt du Model-checking est qu'il retourne une trace d'exécution du système violant la propriété lorsque cette dernière est non valide (voir figure 3.2).

Le Model-checking est une technique de vérification automatisée pour systèmes dynamiques c'est-à -dire, il s'agit ici de vérifier algorithmiquement si un modèle donné satisfait une propriété, souvent cette propriété est formulée dans une logique.

Le Model-checking utilise des outils appelés "model-checker". Un Model-checker est un outil qui agit de la façon suivante : il prend en entrée un modèle exprimé dans un formalisme imposé, et une spécification qui exprime une propriété que doivent vérifier certaines données

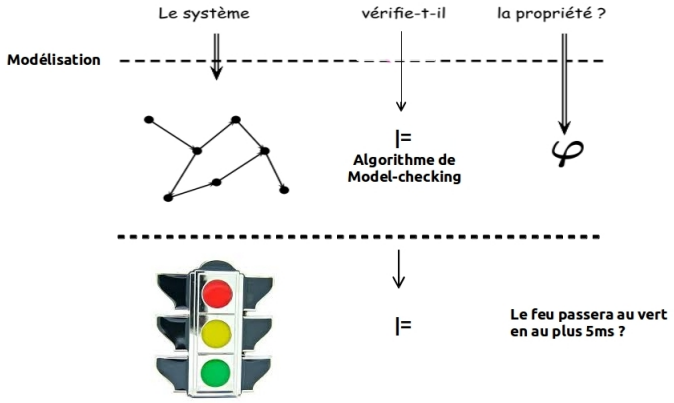


FIGURE 3.2: Fondement formelle du Model-checking

du modèle. Il effectue ensuite un calcul à partir de ces données, et peut produire deux résultats différents : soit toutes les exécutions du modèle satisfont la spécification, et le résultat est positif, soit au moins une exécution du modèle ne satisfait pas la spécification, et dans ce cas le résultat est négatif et le Model-Checker donne cette exécution ou une simplification de celle-ci comme un contre-exemple d'exécution non satisfaisante. A partir de ce contre-exemple, l'utilisateur peut essayer de corriger la source du problème puis effectuer une nouvelle vérification du modèle. La source du problème peut être soit une erreur de l'application ou du cahier des charges, soit une erreur de modélisation, soit enfin une erreur de spécification (voir la figure 3.3).

En contrepartie, l'effectivité du Model-checking est limitée par la taille des systèmes qu'on peut analyser en pratique. En effet, les modèles ont, même pour des systèmes relativement simples, des tailles importantes et on parle du problème de l'explosion combinatoire. L'analyse du modèle est alors impossible en raison de la complexité importante des calculs nécessaire. Dans de telles situations, l'utilisateur doit simplifier le modèle et proposer une abstraction qui préserve les propriétés à vérifier (Zennou, 2004).

3.2.1 Notion de Modèle

Comme nous l'avons déjà vu, un outil de model-checking ne travaille pas sur une implémentation d'un système, mais sur une modélisation de celui-ci. La modélisation consiste à construire, dans un cadre formel précis et imposé par le model-checker, un objet symbolique

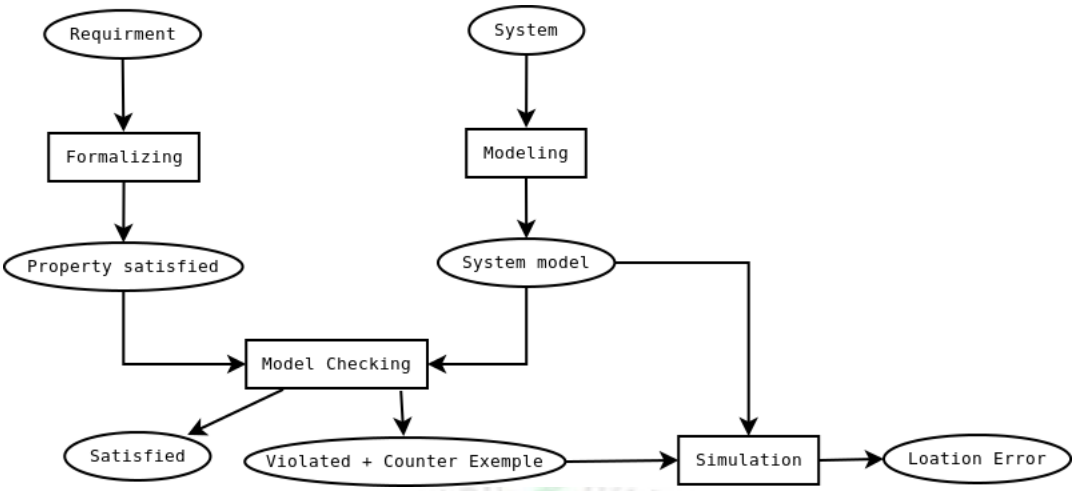


FIGURE 3.3: Cycle d'étapes du Model-checking (Zennou, 2004)

dont le comportement est aussi proche que possible de celui du système, et qui est particulièrement fidèle dès que des données ayant un lien avec la spécification sont en cause (Oddoux, 2003).

Un modèle peut aussi être construit à partir d'un simple cahier des charges qui décrit le fonctionnement du système. La vérification de ce modèle permettra ainsi de vérifier la validité des contraintes du cahier des charges, avant de commencer l'implémentation du système.

Suivant la nature du système et le model-checker utilisé, le modèle pourra être un simple système de transitions, ou aura une forme plus évoluée, permettant de modéliser la communication entre processus par exemple (Oddoux, 2003). Un modèle peut éventuellement être infini, si son nombre de configurations n'est pas borné.

3.2.2 Notion de Spécification

La spécification exprime, une fois encore dans un contexte formel imposé par le model-checker, une ou plusieurs propriétés que le modèle doit vérifier. Le langage d'expression d'une spécification se doit en général d'inclure les prédicats de la logique (Oddoux, 2003). Cependant il est courant de devoir exprimer des propriétés incluant des contraintes temporelles. Inclure des contraintes en temps réel complique fortement le Model-checking.

C'est pourquoi on s'intéresse la plupart du temps à l'ordre dans lequel les événements s'enchaînent : on parle alors de logique temporelle. Celle-ci permet d'exprimer des contraintes naturelles comme les situations suivantes :

— « Jamais il n'y aura une erreur dans le système de contrôle d'un téléphérique »

— ou encore « toute demande sera suivie d'une réponse » .

— . . . etc.

Pour décrire les propriétés de bon fonctionnement des systèmes, les logiques temporelles sont des formalismes bien adaptés (David *et al.*, 2002), notamment par leur capacité à exprimer l'ordonnancement des actions (événements) dans le temps.

La logique temporelle est une forme de logique spécialisée dans les énoncés et raisonnements faisant intervenir la notion d'ordonnancement dans le temps. Les notations de la logique temporelle sont plus claires et plus simples, ses opérateurs sont calqués sur des constructions linguistiques (les adverbes **« toujours »**, **« tant que »**, etc. . . , les temps de la conjugaison des verbes) de sorte que les énonces en langue naturelle et leur formalisation en logique temporelle sont assez proches. Enfin, la logique temporelle est livrée avec une sémantique formelle, équipement indispensable pour un langage de spécification(Alur *et al.*, 1990).

Il est remarquable que la description de propriétés en logique temporelle présentent deux qualités importantes : elles sont abstraites, c'est-à-dire indépendantes des détails d'implémentation de l'application (par exemple, la propriété qu'au plus un processus peut utiliser une ressource à un instant donné doit être satisfaite par tous les protocoles d'exclusion mutuelle) et elles sont modulaires dans le sens modifiable localement (le rajout, le changement ou la suppression d'une propriété ne remet pas en cause la validité des autres).

3.2.3 Logiques utilisés dans le model-checking

Deux familles de logiques temporelles sont couramment utilisées : les premières sont des logiques temporelles linéaires (Linear-time Temporal Logic, abrégé LTL) (Pnueli, 1977), et les secondes sont des logiques temporelles arborescentes (Computation Tree Logic, CTL, CTL*) (CLARKE et EMERSON, 1981; Mateescu, 2003). En effet, la logique LTL permettant d'exprimer des propriétés portant sur des chemins individuels (issus de l'état initial) du modèle. Par

ailleurs, la logique CTL permettant d'exprimer des propriétés portant sur les arbres d'exécution (issus de l'état initial) du modèle.

(A) La logique LTL : Un Aperçu

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'événements 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.

la logique LTL est définie par :

— un ensemble de propositions atomiques AP ou variables de propositions, i.e, des expressions booléennes portant sur des variables, des constantes et des prédicats

— des connecteurs logiques tels que AND, OR, NOT, $\Longrightarrow$

— des modalités :

  — **X** : ne**X**t (demain) : Xp signifie que p est vrai dans l'état suivant le long de l'exécution

  — **F** : Finally (un jour) : Fp signifie que p est vrai plus tard au moins dans un état de l'exécution

  — **G** : Globally (toujours) : Gp signifie que p est vrai dans toute l'exécution

  — **U** : Until (jusqu'à) : p**U**q signifie que p est toujours vrai jusqu'à un état ou q est vrai

Exemple 1. La propriété d'invariance " durant toute l'exécution, x est différent de 0 " se traduit en logique LTL par G$\rceil$(x = 0).

Exemple 2. G(p $\Longrightarrow$ Fp) signifie "pendant toute l'exécution, si p est vrai à un état donné, alors q sera vrai plus tard au moins dans un état suivant".

(B) La logique CTL : Un Aperçu

La logique temporelle CTL (Computational Tree Logic) est un sous-ensemble du modèle arborescent de la logique temporelle. Dans CTL, les opérateurs temporels apparaissent uniquement par paire, composés par des opérateurs A ou E suivies par G, U ou X. Nous présentons dans ce qui suit la syntaxe et la sémantique de CTL, suivie d'un exemple et d'une synthèse.

Syntaxiquement, les formules CTL sont construites comme suit :

(a.) Chaque proposition atomique est une formule CTL.

(b.) Si f et g sont des formules, alors il est de même pour :

$\neg$ f, (f $\wedge$ g) , A**X**f , E**X**f , A(f **U** g), E(f **U** g)

Les opérateurs utilisés sont supposés être dérivés à partir des lois suivantes :

— vrai = $\neg$ faux

— f $\vee$ g = $\neg$ ($\neg$ f $\wedge$ $\neg$ g)

— AF g = A (vrai U g)

— EF g = E(vrai U g)

— AG f = $\neg$ E (vrai U $\neg$ f) = $\neg$ EF $\neg$ f

— AG f = $\neg$ A (vrai U $\neg$ f) = $\neg$ AF $\neg$ f

- EXf signifie qu'il existe un état, successeur immédiat, atteignable à partir d'un état qui satisfait f.

- E(f U g) signifie que, pour quelques chemins de calcul, f reste vrai jusqu'à ce que g le devienne.

- A(f U g) signifie que la propriété f U g mentionnée apparaît sur tous les chemins de calcul.

Remarque :

Les opérateurs utilisés dans CTL peuvent apparaître avec une autre notation, selon la correspondance suivante :

E $\equiv \exists$	A $\equiv \forall$	F $\equiv \diamond$	G $\equiv$ II	X $\equiv$ 0

Coté sémantique, la vérité d'une formule est définie en respectant un modèle d'automate qui est le triplet (S, R, L), où S est l'ensemble des états, R est la relation de transition et L est la fonction de valuation. La relation de transition est l'ensemble des paires(s, t) sachant que t est le successeur immédiat de s. Un modèle de branchement (ou bien un arbre de calcul) est obtenu en commençant par l'état désigné et en déroulant le graphe (S, R) sur un arbre infini.

Un chemin d'un modèle $K = (S, R, L)$ est une séquence infinie d'états $(s_0, s_1, s_2, ...)$, sachant que chaque paire successive d'états (s_i, s_{i+1}) est un élément de R. Chaque chemin est un sous-ensemble ordonné, linéairement maximal, de la structure arborescente déroulée à partir de s_0.

Pour plus de détails et de précision concernant la technique de vérification par model-checking, le lecteur peut s'orienter vers (Clarke *et al.*, 2018).

3.3 Automate temporisé comme modèle d'analyse

Dans notre travail, nous avons choisi le formalisme des automates temporisés comme outil de base pour la formalisation des architectures AADL. Ce choix est justifié par la généralité qu'offrent les automates temporisés, leur capacité à représenter les contraintes temporelles liées à l'occurrence des événements, leur pouvoir d'expression et les possibilités de modélisation qu'ils donnent (Alur et Dill, 1994).

Nous présentons dans cette section le formalisme des automates temporisés. Nous faisons un rappel de l'aspect syntaxique et sémantique de ce formalisme de modélisation.

3.3.1 Définition

Un automate temporisé est un automate à états finis muni d'un ensemble de variables réelles positives appelées horloges (voir figure 3.4). Les horloges sont incrémentées simultanément, avec une dynamique égale à 1 et peuvent être remises à zéro. Les horloges sont des variables fictives dans le sens où elles ne sont pas des variables du système, mais elles sont utilisées pour mesurer le temps et définir les contraintes temporelles sur le franchissement des transitions (Alur et Dill, 1994).

Une transition, entre deux sommets d'un automate temporisé, est franchie suite à l'occurrence d'un événement en entrée et la satisfaction d'une contrainte de franchissement, appelée garde. Ce franchissement est instantané et peut déterminer la mise à zéro d'un ensemble d'horloges.

Avant de définir formellement le modèle automate temporisé, nous introduisons, dans ce qui suit, un exemple de compréhension. Soit le scénario suivant : « **Si j'appuie sur le bouton la lumière s'allume. Si j'appuie deux fois (rapidement) sur le bouton, la lumière s'allume plus fort. Si j'appuie sur le bouton, la lumière s'éteint** ».

3.3.2 Syntaxe

Un automate temporisé est défini par un sept-uplet : $A = (Q, X, \Sigma, E, q_0, Q_f, I)$

- Q est un ensemble fini de sommets (localité) ;
- $X = \{x_1, ..., x_n\}$ est un ensemble fini d'horloges ;
- Σ est un ensemble fini d'événements (ou d'actions ou de symboles) ;

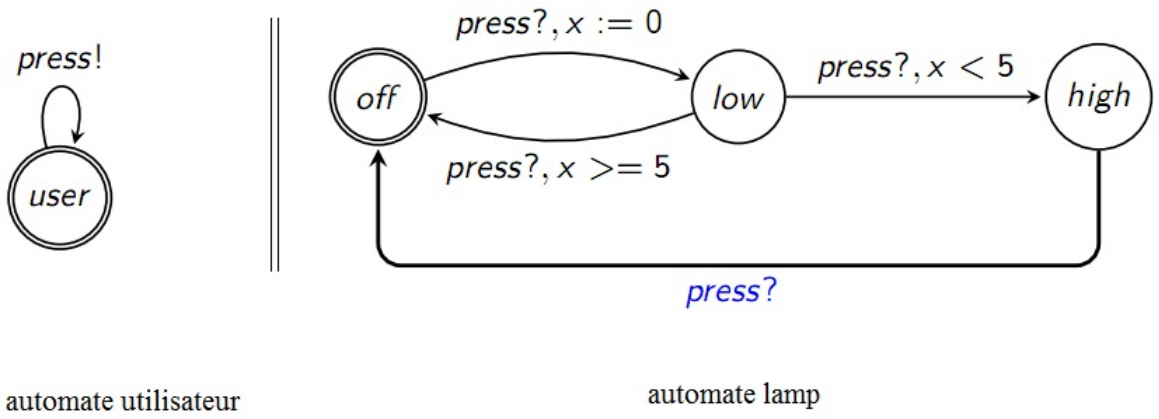


FIGURE 3.4: Exemple d'un automate temporisé

— E est une relation de transition entre les sommets ;

— $q_0 \in Q$ est le sommet initial ;

— $Q_f \in Q$ est un sous-ensemble de sommets finaux qui n'admettent pas des transitions de sortie vers d'autres sommets (sommets puits) ;

— $I : Q \rightarrow G(X)$ est une fonction qui associe pour chaque sommet $q \in Q$, une contrainte $I(q)$ dans $G(X)$, appelée invariant du sommet q.

Chaque transition $e \in E$ est définie par un quintuplet $e = (q, g, a, r, q')$ et est noté aussi par :

$$q \overset{g,a,r}{\rightarrow} q'$$

— $q$et q désignent, respectivement, le sommet de départ et le sommet d'arrivée de la transition ;

— $g \in G(X)$ est une contrainte sur l'ensemble des horloges, appelée garde de franchissement. Cette contrainte doit être satisfaite par les valeurs des horloges pour permettre le franchissement de la transition ;

— $a \in \Sigma$ correspond à l'événement déclenchant le franchissement de la transition ;

— $r \subseteq X$ est un sous-ensemble d'horloges remises à zéro lors du franchissement de la transition, appelé l'ensemble de ré-initialisation.

3.3.3 Sémantique

Dans ce formalisme, le système part de la configuration initiale (état de contrôle l_0 et toutes les horloges à zéro) ; puis effectue alternativement deux types de transition : transition d'action si la valeur courante des horloges le permet (ce type de transition s'effectue de manière instantanée et certain horloges peuvent alors être remises à zéro) et transitions de temps qui augmente toutes les horloges d'une même durée c'est à dire, les horloges sont synchrones en respectant l'invariant associé à la localité courante.

Définition 1.
Soit X un ensemble fini d'horloges. Une valuation des horloges est une fonction $v : X \to R^+$, qui associe un nombre réel positif à chaque horloge de X (Alur et Dill, 1994).
Étant donnée une valuation v sur X et un réel positif $d \in R^+$, alors la valuation $v + d$ correspond à l'écoulement d'une durée de d unités de temps (u.t.) depuis la valuation v :

$$(v + d)(x) = v(x) + d, \forall x \in X$$

Définition 2.
Un état d'un automate temporisé est un couple (q, v), où q est un sommet et v une valuation d'horloges. La configuration initiale d'un automate temporisé est (q_0, v_0) où $v_0(x) = 0$ pour toute horloge $x \in X$.
La notation $v \models g$, où $v \in R_n^+$ et $g \in G(X)$, signifie que la valuation v vérifie la garde g (Alur et Dill, 1994).

Définition 3.
Un chemin P dans un automate temporisé A est une séquence finie ou infinie de transitions de E de la forme (Alur et Dill, 1994) :

$$q_0 \overset{g1,a1,r1}{\to} q_1 \to q_2 \overset{g2,a2,r2}{\to} q_3 \to q_4 \overset{g3,a3,r3}{\to} q_6 \to q_6 \overset{g4,a4,r4}{\to} q_7 \to \dots$$

où q_0 désigne le sommet initial.

D'un point de vue exécution, l'automate temporisé admet deux types d'évolution possibles :

— *les transitions de temps* : l'automate séjourne dans le même état en laissant le temps s'écouler. En effet, les valeurs des variables progressent d'une manière synchrone avec la même durée, en respectant la contrainte de l'invariant du sommet. Formellement :

$$(q, v)d \rightarrow (q, v + d) \text{ pour } d \in R^+, si \forall 0 \preceq t \preceq d, v + t \models I(q)$$

— *les transitions d'actions* : Ces transitions font évoluer l'automate d'une état à un autre. Une transition d'action est possible lorsque la valuation courante des horloges satisfait la contrainte de garde et un événement est réalisé. Ces transitions sont aussi instantanées. En plus, le franchissement d'une transition peut remettre à zéro un sous-ensemble d'horloges.

$$(q, v) \overset{g,a,r}{\rightarrow} (q', v') \text{ s'il existe } q \rightarrow q' \in E, \text{ tel que}$$
$$v \models g, v = v[Y \leftarrow 0] et v \models I(q)$$

- Une exécution est un chemin dans le système de transition temporisé où alternent les pas de temps et les transitions discrètes.
- Une exécution génère un mot temporisé : une suite d'actions couplées avec l'instant auquel elle se produisent

Exemple : Une exécution possible de l'automate temporisé de la figure 3.4 est définit par la séquence suivante :

$(m, [x = 0, y = 0], []) \overset{3}{\rightarrow} (m, [x = 3, y = 3]) \overset{a}{\rightarrow} (n, [x = 0, y = 3],) \overset{a}{\rightarrow} (n, [x = 0, y = 0],) \overset{3}{\rightarrow}$
$(n, [x = 3, y = 3],) \overset{a}{\rightarrow} (n, [x = 3, y = 0],) \overset{1}{\rightarrow} (n, [x = 4, y = 1],) \overset{b}{\rightarrow} (m, [x = 4, y = 0],) \rightarrow \ldots$

3.3.4 Model-checker pour les automates temporisés

Dans le milieu académique, une famille de model-checker (outil de model-checking) ont été développés on se basant sur la théorie des automates temporisés. On trouve par exemple : TIMES , Stratego , ***UPPAAL*** [1] (Larsen *et al.*, 1997), CORA , TRON , TIGA , COVER , PORT , PROHytech (Henzinger *et al.*, 1995), Kronos(Daws *et al.*, 1996) . Ces outil qui ont été utilisés

1. http://www.uppaal.org/ [consulté le 15/03/2018]

avec succès pour la spécification et la vérification des applications temps-réel industrielles (BENGTSSON *et al.*, 1996; LINDAHL *et al.*, 1998).

Uppaal est à l'heure actuelle (version 4.0.13 stable) le plus complet des outils. Dans ce travail nous avons choisi l'outil Uppaal car il permet à la fois la conception, la simulation et la vérification. De plus, Uppaal est régulièrement mis à jour et offre une interface graphique conviviale qui assure une prise en main aisée de l'outil. Les modèles manipulés par Uppaal sont une variante équivalente du modèle des automates temporisés. Dans le dernier chapitre on va essayer de faire une description sur les fonctionnalités de base de cet outil.

3.4 Propriétés vérifiables

Nous distinguons dans la littérature plusieurs types de propriétés, qui peuvent être classées différemment. Nous présentons dans ce paragraphe une classification des propriétés temporelles souvent rencontrées dans les travaux de spécification par les automates temporisés sur les systèmes temps réel (Schnoebelen, 1999).

- Propriétés d'atteignabilité (*reachability*)

Une propriété d'atteignabilité énonce qu'un certain état peut être atteint ou non.
Exemples :

(A1) "On peut entrer dans une section critique"

(A2) "On peut revenir à l'état initial"

(A3) "On ne peut pas atteindre l'état Crash"

Parfois c'est la négation de la propriété qui est recherchée. un état d'exception ne peut pas être atteint.

- Propriétés de sûreté (*safety property*)

Une propriété de sûreté énonce que, sous certaines conditions, quelque chose de mauvais ne se produit jamais.
Exemple :

(S1) "Les deux systèmes ne seront jamais simultanément en section critique ",

(S2) "Il n'y aura jamais de débordement mémoire" ,

(S3) "Durant toute l'exécution, x est différent de 0",

(S4) "Quelque chose non souhaité ne se produit jamais".

Les propriétés de sûreté peuvent être exprimées en d'autres formes, entre autre la non atteignabilité.

- Propriétés de vivacité (*liveness*)

Une propriété de vivacité énonce, que sous certaines conditions, quelque chose de bien finira par avoir lieu. Nous distinguons dans la littérature plusieurs formes de vivacité, entre autre, la vivacité de progrès (ou simple) et la vivacité bornée.

1. *Propriétés de vivacité simple* : Ce type de propriété énonce qu'un état est toujours atteignable.
   Exemples :

(VS1) "Une requête est satisfaite un jour" ,

(VS2) "Le système peut toujours retourner à l'état initial" ,

2. *Propriétés de vivacité bornée (ou réponse bornée)*. Ce type de propriété impose un délai maximal avant lequel la situation souhaitée finira par avoir lieu.
   Exemples :

(VB1) "Toute requête finira par être satisfaite en au moins de 5 mn" ,

(VB2) "Le feu passera au vert en au plus 3mn " ,

(VB3) "Le programme se termine en au plus 10s".

- Propriétés d'équité ou de vivacité générale (*fairness*)

Cette propriété énonce que, sous certaines conditions, quelque chose aura lieu (ou n'aura pas lieu) un nombre infini de fois.
Exemples :

(E1) "La barrière sera levée infiniment souvent",

(E2) "Si l'on demande l'accès à une section critique un nombre infini de fois, alors il sera accordé un nombre infini de fois ".

- Propriétés absence de blocage (*deadlock*)

une situation à partir de laquelle le système ne peut plus progresser, ou bien (livelock, inter-blocage ou étreinte éternelle) des processus se bloquent mutuellement, on l'apparente souvent à une propriété de sûreté.

Toutes ces propriétés peuvent se vérifier sur un système AADL en cours de développement ***si et seulement si nous pouvons établir une représentation formelle*** (sous forme d'automates temporisés par exemple). Dans ce cas, la vérification concerne plusieurs axés :

- La structure de la description comme la validité des connexions par exemple.
- Le respect des contraintes exprimées dans les propriétés AADL telles que : la politique d'ordonnancement ou bien le temps d'exécution des ***threads***, ...
- ...etc

Conclusion

La vérification *a priori* est une phase importante dans la conception des systèmes complexes de type temps réel, car elle permet de s'assurer d'un certain nombre de caractéristiques de l'architecture considérées avant sa construction effective. Pour cela, il est nécessaire de pouvoir extraire une description formelle du système.

Dans ce chapitre nous avons montré tous les éléments utilisés dans un processus de vérification par model-checking. Ensuite, nous avons présenté les automates temporisés car ils sont classés parmi les puissants formalismes pour la spécification des systèmes temps réel et ceci grâce à leur lisibilité et puissance d'analyse. D'ailleurs, plusieurs outils sont recensés autours de ce formalisme.

Dans le chapitre suivant on va présenter un ensemble de travaux qui s'intéressent à la formalisation de AADL dont le but est la vérification.

CHAPITRE 4

TRAVAUX DE TRANSFORMATION & VÉRIFICATION AUTOUR DE AADL

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

Sommaire

Introduction

Dans ce chapitre, nous dresserons un état de l'art sur quelques travaux autour de transformation & vérification d'AADL qui ont précédé notre contribution, ainsi que quelques travaux qui ont été publiés après. Dans les sections de 1 jusqu'au 6 on présentera en ordre chronologique une série de travaux qui portent directement sur la problématique de la transformation & vérification d'AADL.

Avant de conclure nous tracerons, une étude comparative basée sur quelques critères d'évaluation dont le but de tirer les avantages et ainsi les manques de chaque travail.

4.1 Travail de T. Vergnaud et al. 2006

L'objectif de cette transformation (Vergnaud, 2006) de vérifier la description AADL d'un système réparti à l'aide de RdPc (Réseau de Pérti coloré) (Rozenberg et Engelfriet, 1998). Le formalisme est considéré comme une notation formelle pour la modélisation et la vérification de systèmes concurrents. Les auteurs ont essayé de vérifier des propriétés de l'architecture tel que :

— L'assemblage des composants n'engendre pas un inter-blocage.

— La structure de l'architecture n'engendre pas un débordement du fil d'attente.

— Les données utilisées dans les sous programmes sont définis d'une façon déterministe.

Dans ce travail, les auteurs ont proposé que la transformation prend en considération les instances des composants et non pas leurs déclaration. De plus, les composant de base concerné par cette étude sont : les ***threads***, les sous programmes et les données. Les composants décrivant la plate-forme d'exécution sont ignorés. Les autres composants logiciels (groupes de ***threads*** et processus) et les systèmes sont considérés comme des simples conteneurs.

L'idée générale de cette transformation est basé sur le principe qu'une description AADL contient des composants qui acceptent des données en entrée et produisent d'autres en sortie. Le processus de transformation est guidé part huit règles de transformation, on peut citer par exemple :

— **Règle 1** : Chaque flux de données est traduit en jetons, les transitions modélisent les composants actifs qui consomment des données et produisent d'autres données et finalement les places qui modélisent les entités permettant de stocker les données.

— **Règle2** : Un réseau de Pétri généré à partir d'une description AADL comprend deux classes de couleurs :

  — Les jetons représentant les flux de données appartiennent à une classe nommée Value qui contient deux couleurs : U pour les données indéfinies (undefined) et D pour les données définies (defined)

  — Les jetons de contrôle des ***threads*** appartiennent à une classe nommée "Control" qui contient autant de couleurs que d'instances de ***threads*** dans la description AADL.

— **Règle 3** : Modélisation haut niveau : une instance de composants de donnée et traduit par une place. Les composants « actifs » de haut niveau (***system, process, threads***) d'une architecture sont traduits par une transition (représentant le composant lui-même) entourée de places modélisant ses éléments d'interface. La figure 4.1 montre la représentation graphique des composants hauts niveau : Les composants « actifs » de haut niveau (systèmes, processus, ***threads***) d'une architecture sont traduits par une transition.

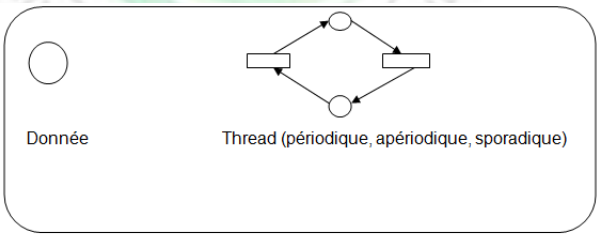


FIGURE 4.1: Traduction du composant donnée

4.2 Travail de J. Champeau et al., 2008

AADL a un grand nombre des types composants qui peuvent être utilisés pour construire des modèles hiérarchiques. Le but de cette transformation (Champeau *et al.*, 2008) est de valider le comportement des composants logiciels de l'architecture de système.

À l'origine IF (Champeau *et al.*, 2008) est conçu pour être un environnement complet (langage et une boîte à outils) utilisé comme moyen de vérification dans les domaines des protocoles de télécommunication, des systèmes embarqués ou des systèmes temps-réels. Pour cela, une spécification IF doit être développée. Cette spécification est exprimée d'un coté en terme d'éléments de type système, processus et éléments de communication (signaux et routes de communication). D'autre côté, la spécification par le comportement qui est exprimée en termes d'états, transitions et d'actions ainsi que les données(Champeau *et al.*, 2008).

Système : un système est composé d'un ensemble d'instances de processus actifs qui s'exécutent en parallèle et qui communiquent de manière asynchrone par échange des signaux via des routes de communication (*signalroute*) ou par adressage direct. La description d'un système contient la définition des éléments suivants : types de données, constantes, variables partagées, signaux de communication asynchrones (signal) et processus.

Processus : un processus est défini comme un automate temporisé étendu. Chaque instance est associée à un identificateur unique. Un processus possède une file d'attente et est constitué respectivement d'un ensemble de variables locales typées, incluant des horloges, d'un ensemble d'états de contrôle et d'un ensemble de transitions entre ces états.

Signal : un signal (signal) est utilisé dans la communication entre processus. Chaque processus sauvegarde les signaux qui lui ont été envoyés dans sa propre file d'attente. La déclaration du type d'un signal contient son nom et la liste de ses paramètres éventuelles.

Route de communication : les routes de communication (signalroute) assurent la réalisation de la communication asynchrone entre processus ou entre processus et l'environnement. Chaque instance d'une route a un identificateur unique. La description d'une route de communication contient le nom du *signalroute*, sa source et sa destination (processus ou environnement) et l'ensemble des signaux transportés. On distingue plusieurs types de *signalroutes* : des routes ordonnées ou non ordonnées (FIFO ou multiset), avec ou sans perte (reliable, lossy), urgentes ou non urgentes (urgent, delay).

Transition : un processus peut changer d'état en exécutant une transition. Chaque transition peut être associée à une échéance (eager, lazy ou delayable) qui détermine son niveau d'urgence. Par défaut, toute transition est définie comme étant urgente (eager). Le corps d'une transition est constitué d'un ensemble d'actions élémentaires. Chaque transition se termine soit par une action *nextstate*spécifiant son état destination ou par une action stop détruisant l'instance courante du processus

Dans ce travail, les auteurs définissent le principe de transformation en quatre règles :

— **Règle 1 : Le système**
système qui représente la racine composante des modèles AADL est transformé dans le langage IF par "system", l'élément racine dans le modèle "IF".

— ***Règle 2 : Les données***
Les données "dataType" sont transformées à " IF" Type.

— **Règle 3 : Les processus et les *threads***
Les processus et les ***threads*** sont les concepts qui exigent l'attention détaillée parce qu'ils mettent en capsule le comportement des composant dans une architecture AADL. Les processus doivent contenir au moins un ***thread*** ou ThreadGroup. Les processus et les ***threads*** dans AADL sont transformés dans "IF Process".

— **Règle 4 : Ports dans AADL** Les Ports est le mécanisme utilisé pour assurer la communication entre les ***threads*** dans une architecture AADL. Ce type de communication est transformé dans le langage "IF" par le concept de Signal et ils doivent être gérer par un ***"Process IF"***.

4.3 Travail de L. Pi et al., 2009

La traduction d'AADL vers le langage FIACRE (Intermediate Format for the Architectures of Embedded Distributed Components) (Pi *et al.*, 2009) a été mise en œuvre au sein de l'atelier TOPCASED[1] en utilisant les techniques de l'IDM. Le langage FIACRE a été défini sémantiquement et est intégré à deux boîtes de vérification (TINA (Berthomieu *et al.*, 2004) ou CADP (Garavel *et al.*, 2011)).

Le langage Fiacre est construit autour de deux notions principales qui sont les processus et les composants. Un processus est défini comme un ensemble d'états de contrôle, chacun associé à un programme impliquant des constructions déterministes classiques ; des constructions non-déterministes ; des événements d'interactions sur des ports de communication typés et orientés (in/out). Par contre, un composant est défini comme une composition parallèle de composants et/ou de processus interagissant par des portes et/ou des variables partagées. La chaîne de transformation proposée est décrite dans la figure 4.2 :

Les principales règles de transformation sont :

1. http ://www.topcased.org/

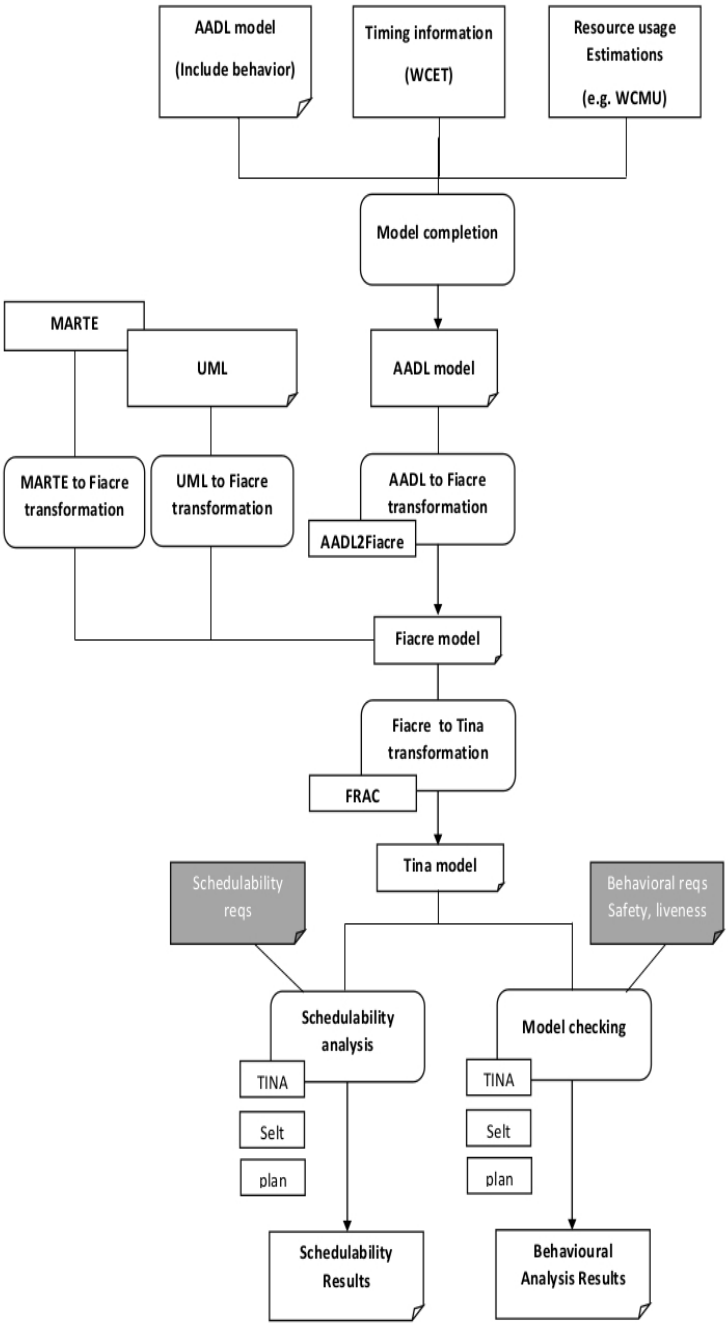


FIGURE 4.2: Transformation AADL2FIACRE (Pi *et al.*, 2009)

— **Règle 1 : Composants matériels :**
Les composants matériels d'AADL (bus, mémoires, processeurs, . . .) ne seront pas traduits vers FIACRE, car il n' a pas d'équivalent pour ces éléments dans ce dernier langage. Probablement, il faudra tenir compte de certaines propriétés, des bus de communication (telles que le délai de propagation, ou le temps de transmission) pour les appliquer aux connexions qu'ils véhiculent.

— **Règle 2 : *threads***
Les ***threads*** sont les composants AADL les plus importants, car ils sont les composants actifs. La seule possibilité est de les traduire vers des processus FIACRE (Il ne faut pas confondre les processus AADL, qui sont essentiellement des conteneurs des ***threads***, avec les processus FIACRE, qui sont les éléments actifs du langage.). En effet, cette traduction est naturelle car les concepts représentés par ces deux éléments sont essentiellement les mêmes : ce sont des composants actifs, qui présentent un comportement qui peut être décrit par une machine à états (Directement dans le cas des processus FIACRE, et au travers de l'annexe comportementale dans le cas des ***threads*** AADL) , qui peuvent envoyer et recevoir des données et/ou des événements sur des ports, etc.

— **Règle 3 : Processus et systèmes :**
Les processus AADL peuvent être vus comme des conteneurs de ***threads***. De la même façon, la fonction principale des systèmes AADL est de contenir des processus et d'autres systèmes. C'est-à-dire, que ces deux composants organisent hiérarchiquement l'architecture logicielle des systèmes modélisés avec AADL. La façon la plus naturelle de traduire ces composants est, donc, de le faire comme des composants FIACRE (Il ne faut pas confondre les composants AADL, qui est le terme générique pour désigner les ***threads***, systèmes, processus, mémoires, processeurs, etc., avec les composants FIACRE, qui servent à composer les états de plusieurs processus FIACRE.) , car ils peuvent contenir des processus (qui sont les équivalents des ***threads AADL***), et ils peuvent organiser ces processus et d'autres composants de façon hiérarchique. Les composants FIACRE peuvent spécifier également les connexions entre les processus et composants qu'ils contiennent, tout comme les systèmes et processus AADL ils spécifient la manière dont les ***threads*** sont connectés à leur intérieur.

— **Règle 4 : Connexions :**
Les connexions AADL peuvent être traduites directement vers des connexions FIACRE.

Dans les deux cas, on a la notion de ports qui sont des caractéristiques de l'élément considéré et qui peuvent être connectés sur un port d'un autre élément. Pour connecter des éléments qui n'ont pas une relation de contenance directe (dans la hiérarchie de composants) il faut découper la connexion en plusieurs tranches.

4.4 Travail de P. Hladik et al., 2010

L'objectif de ce travail(Hladik *et al.*, 2010) est l'intégration du langage Pola (langage formel de vérification) dans le processus de conception des systèmes temps réel critiques en se basant sur le langage AADL. À l'aide d'une chaîne de traduction/vérification automatique, un modèle Pola est traduit dans un langage adapté au Model-checking, puis la vérification proprement dite est effectuée à partir de cette traduction.

La figure 4.3 présente l'ensemble de la chaîne de vérification qui peut être mise en œuvre à partir d'une description d'une architecture temps réel faite en Pola.

Un modèle Pola est composé de systèmes de tâches, de ressources, de politiques d'ordonnancement et d'allocations de ressources. Le système de tâche est l'élément de plus haut niveau de la description. L'utilisateur peut aussi décrire les comportements à l'aide d'un Rdp (réseau de Pétri) dans une partie dédiée .

— **Les tâches** : sont les entités élémentaires prises en compte par l'ordonnanceur. Elles peuvent être caractérisées par des informations de période, de temps d'exécution, de date d'échéance maximale, etc. Il est également possible de décomposer les différentes étapes d'une tâche par un enchaînement d'actions

— **Les ressources** : une ressource est la représentation abstraite des différents supports d'exécutions qu'une tâche peut avoir besoin d'utiliser. Elle ne porte aucune autre information que d'être libre ou utilisée. Une tâche ne pourra s'exécuter que lorsque toutes les ressources nécessaire pour son exécution lui sont attribuées

— **Politiques d'ordonnancements** : Une politique d'ordonnancement est définie par une fonction d'ordre (min pour un ordre croissant ou max pour un ordre décroissant) entre les tâches dont le critère est évalué par une combinaison linéaire des caractéristiques des tâches. Ainsi l'ordre défini par la politique représente la priorité sur laquelle l'ordonnanceur s'appuie pour déterminer à quelle tâche donner les ressources.

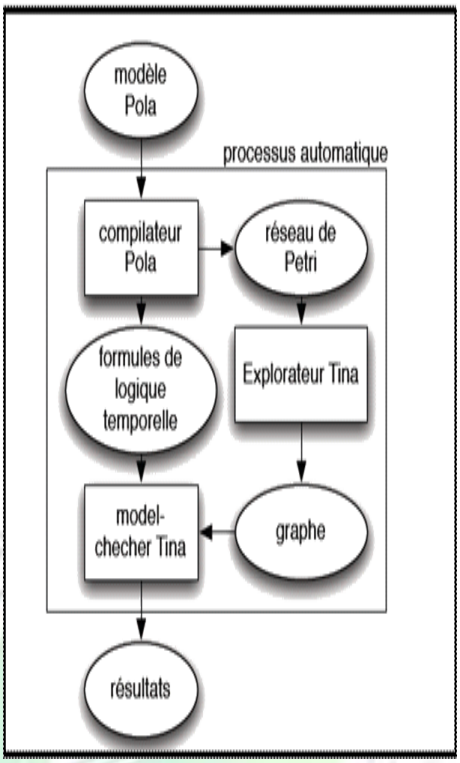


FIGURE 4.3: Processus de vérification avec POLA (Hladik *et al.*, 2010)

— **Politique d'allocation** : Les allocations permettent de caractériser les ressources allouées aux tâches et indiquent à l'ordonnanceur les choix possibles. Il est également possible d'avoir plusieurs allocations pour une même tâche, auquel cas, le choix de l'une ou l'autre allocation sera indéterminé.

Le Listing 4.1 présente un exemple de système modélisé en Pola avec deux ressources (lignes 6 et 7) et trois tâches (lignes 9_14, 15_21 et 22_26). Chaque tâche est composée d'une seule action qui porte sa durée d'exécution et la manière dont les ressources lui sont allouées (lignes 10,16 et 23).

La ressource data est uniquement partagée par les tâches "T1" et "T2", alors que la ressource "proc" est partagée entre toutes taches (les allocations "alloc1" et "alloc2"). Les deux tâches "T1" et "T2" sont périodiques (lignes 11 et 17), tandis que "T3" est apériodique. L'activation de T3 est spécifiée dans la partie "behavior" (lignes 2_5) et se produit sur la terminaison de "T1". La politique d'ordonnancement est spécifiée ligne 8 et donne la priorité à la tâche avec l'échéance

la plus petite.

```
system sys is
    behavior is
             tr t1end -> t3begin
             lb sys.T1.act1 t1end
             lb sys.T3.released t3begin
    res proc is preemptable
    res data is not preemptable
    policy DM is min D
    task T1 is
         action act1 in [1,1] with alloc1
         period [2,2]
         deadline 2
         policy DM
    end
    task T2 is
         action act1 in [2,2] with alloc1
         period [3,3]
         deadline 3
         offset 1
         policy DM
    end
    task T3 is
         action act1 in [1,1] with alloc2
         deadline 1
         policy DM
    end
    allocation alloc1 is
         resources proc, data
         tasks T1, T2
    allocation alloc2 is
         resources proc
         tasks T3
end
```

Listing 4.1: Exemple de modèle en POLA (Hladik *et al.*, 2010)

Principe de la transformation de AADL vers Pola est expliqué par les règles suivantes :

— **Règle 1 *threads* :**
Le comportement d'une application est principalement capturé en AADL par les ***threads***, qui se traduisent en Pola par la notion de tâche (task). Le comportement temporel des ***threads*** est aussi lié à l'ordonnancement spécifié par les propriétés *Scheduling_Protocol* et *Preemptive_Scheduler* des composants "processor". Au niveau Pola, cela va se traduire par une politique d'ordonnancement (policy) et des allocations (alloc).

— **Règle 2 Les sous programmes** :
Un subprogram en AADL décrit une partie de code qui peut être appelée par des ***threads***. Il a donc une influence sur l'exécution et le comportement de ces derniers. Cette notion se traduit en Pola sous la forme d'actions (action).

— **Règle 3 : Les Données :**
En AADL, les données sont modélisées à l'aide du composant data et la propriété associée *Concurrency_Control_Protocol* qui spécifie leur politique d'accès. En Pola, cela se traduit par une ressource (res).

— etc.

4.5 Travail de M. Benammar et al., 2010

(Benammar et Belala, 2010) ont proposé une annexe nommée ABAReL (AADL Behavioral Annex based on generalized Rewriting Logic). Cette annexe permet de décrire formellement la structure statique et le comportement d'un composant AADL en se basant sur la logique de réécriture. Ils ont utilisé l'outil RT-Maude tool[2] pour vérifier la description AADL transformée. La figure 4.4 présente une équivalence entre les composants AADL et les concepts de la logique de réécriture révisée.

4.6 Travail de Y. Zhang et al., 2011

La description d'un système embarqué par le langage AADL utilise une hiérarchie relationnelle entre les composants et leurs sous composants. L'algorithme de la transformation (Zhang *et al.*, 2011) se résume dans l'organigramme de la Figure 4.5 :

La transformation proposée par (Zhang *et al.*, 2011) est guidée par huit règles. On peut citer par exemple :

— **Règle 1 :** Utiliser le fichier de la description AADL du système pour construire l'arbre des composants. La racine de l'arbre est le composant system, le deuxième niveau représente les sous composants du system (process, data, sous système,...etc) et ainsi de suite jusqu'à l'arrivée à la fin l'arbre (un niveau qui n'a pas de sous niveau). Chaque nœud de l'arbre

2. http://heim.ifi.uio.no/peterol/RealTimeMaude/

Elément Architectural AADL	Concept orienté objet temps réel de la logique de réécriture révisée
Configuration AADL ***A***	Théorie de réécriture orientée objet temps réel $\mathbf{T}_A$
Composant ***C*** (de la configuration ***A***)	Classe *Component* ***C***
Une instance de composant ***O***	Un objet ***O*** de la classe ***C***
Catégorie de composant	Attribut *Category* de la classe ***C***
Interfaces fonctionnelles ou *Features* d'une instance de composant	Attributs de ***C***: *IPortN* , *OPortN*
Flux ***F***	Message ***F***
Interaction entre composants	Règle de réécriture Rl : m < $O_1 : C_1$ > $\rightarrow$ < $O_2 : C_2$ >
Connexion ***Cx***	Classe Connection ***Cx***

FIGURE 4.4: Équivalence sémantique entre AADL et la logique de réécriture révisée (Benammar et Belala, 2010)

contient des informations sur les composants tels que : le nom du composant, le mode courant, des informations pour passer d'un mode à un autre, . . . etc.

— **Règle 2 :** Utiliser la méthode de parcours par largeur [breadth_first] pour parcourir l'arbre des composants, stocker le mode courant dans l'ensemble des états de l'automate, le mode initial est stocké dans un état initial. A partir de cette étape on peut extraire tous les états de l'automate ainsi que les états initiaux avec l'initialisation de l'ensemble de transition.

— **Règle 3 :** À partir de la règle 2 l'ensemble des états est construit, le mode vecteur est obtenu par le produit cartésien du mode dans le nœud parent et le mode dans le nœud fils. L'ensemble des transitions a été initialisé.

— **Règle 4 :** l'arbre obtenu dans la règle 1 est parcouru une autre fois avec la même méthode pour extraire les événements qui ont causé les transitions et les stocker dans l'ensemble de transitions.

— etc.

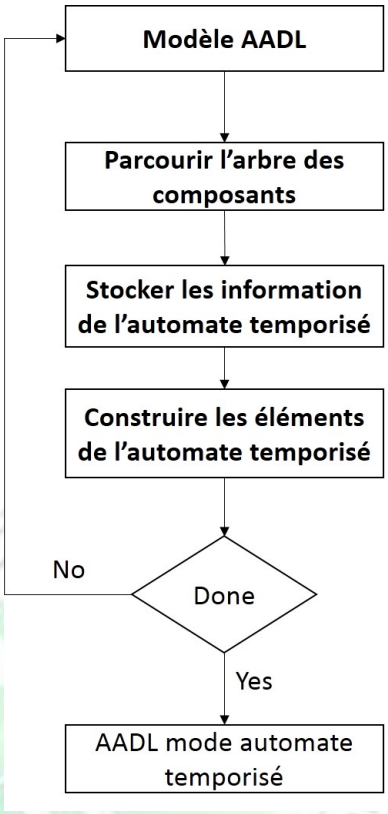


FIGURE 4.5: Transformation des modes AADL vers les automates temporisées (Zhang *et al.*, 2011)

4.7 Travail de Z. Yang et al., 2014

Le travail de (Yang *et al.*, 2014) propose une méthodologie qui permet de transformer des modèles AADL vers le modèle ***Timed Abstract State Machines*** (TASM). Premièrement, les auteurs cible un sous-ensemble des composants AADL qui sera impliqué dans la transformation :

— *threads* periodique,

— *ports* de communication de type donnée,

— *modes* de changement,

— *l'annexe* comportementale.

Deuxièmement, ils ont proposé une démarche de transformation des modèles AADL vers le modèle TASM afin de vérifier deux propriétés : la consommation de ressources et la synchronisation. La figure 4.6 présente le cadre général du travail de(Yang *et al.*, 2014) :

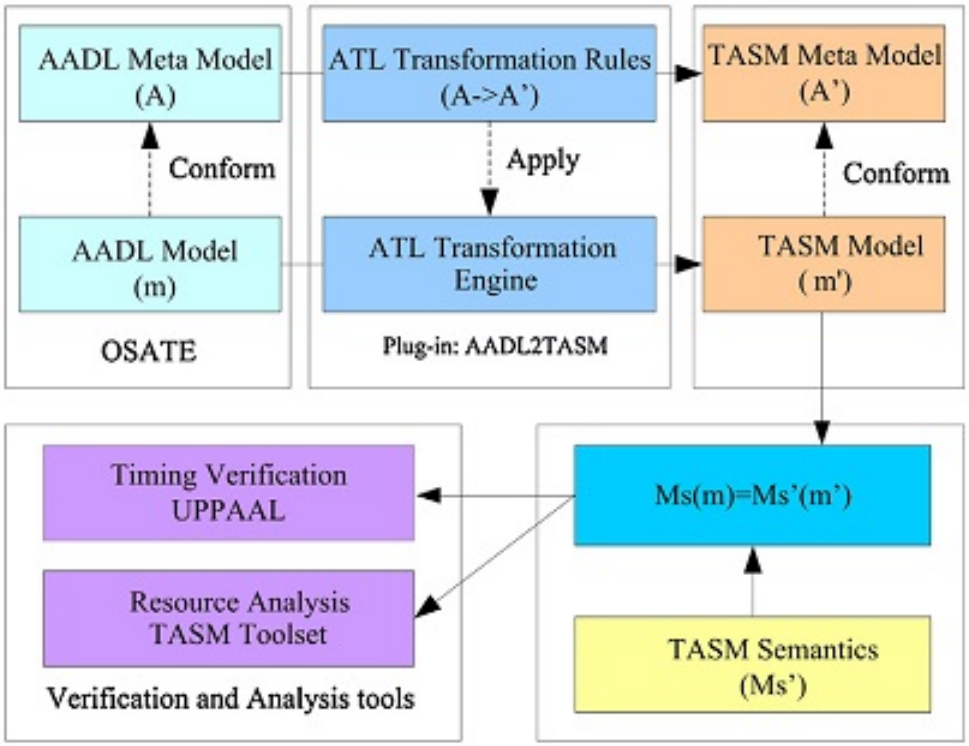


FIGURE 4.6: Prototype de l'outil de la transformation AADL2TASM (Yang *et al.*, 2014)

Troisièmement, les auteurs présentent une preuve dont le but est de montrer que l'approche proposée préservera la sémantique entre le modèle source et cible. Ils ont utilisé l'assistant de preuve Coq [3] pour réaliser la démonstration.

4.8 Travail de P. Gracy et al., 2018

(Gracy et Meenakshi, 2018) s'intéressent à la problématique de valider la sécurité d'un système embarqué temps réel. Pour cela, ils utilisent le langage AADL et cela dû à sa capacité

3. https://coq.inria.fr/

de modéliser le comportement correct et incorrect d'un système. Tout d'abord, ils fixent cinq catégories d'erreurs :

1. Erreurs liées aux services,
2. Erreurs liées aux temps,
3. Erreurs liées aux valeurs,
4. Erreurs liées à la réplication,
5. Erreurs liées à la concurrence.

Dans ce travail, les auteurs utilisent l'annexe de modèle d'erreur d'AADL et l'annexe BLESS pour décrire les paramètres spécifiques à la validation de la sécurité. La solution proposée est basée sur la détection des défauts/erreurs, l'isolation et la re-configuration du système.

Les auteurs proposent aussi d'intégrer leur approche à une démarche basée sur *le teste "testing"* pour valider les modèles AADL (cf. figure 4.7).

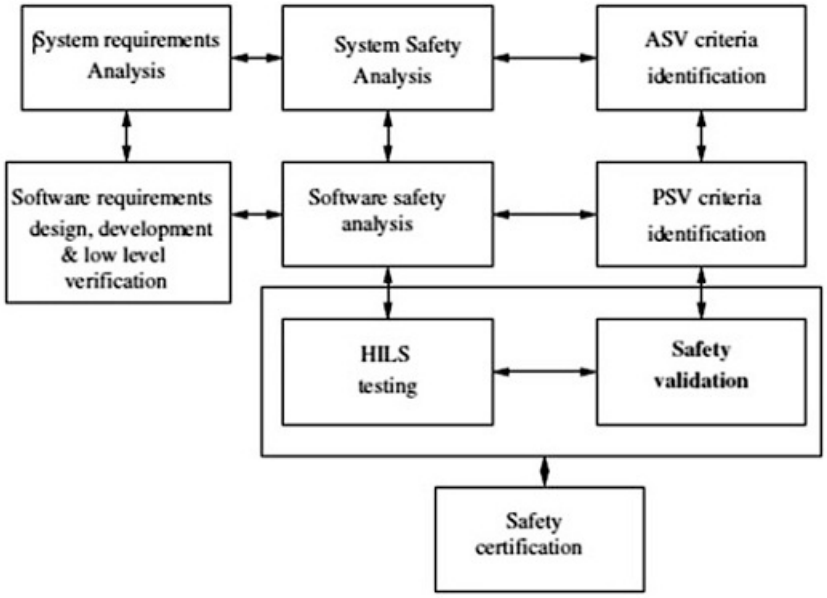


FIGURE 4.7: intégration de l'analyse de sécurité dans un système de testing (Gracy et Meenakshi, 2018)

Pour montrer l'utilité de la contribution , (Gracy et Meenakshi, 2018) présentent une étude de cas impliquant la validation de la sécurité d'un système de commandes de vol automatiques (AFCS).

4.9 Discussion

D'après une analyse approfondie des différents travaux mentionnés dans la section précédente, on peut les classer selon leur démarche de vérification en deux grandes directions :

La première consiste à focaliser l'étude de AADL sur un point particulier du comportement du système, par exemple l'ordonnancement ou l'occupation des *"buffers"* de communication. Pour cela, seules les informations en relation avec ce point sont extrait du modèle AADL afin d'être exploitées par la suite par un outil dédié à l'analyse. C'est par exemple la démarche adoptée pour faire de la vérification de modèle AADL avec l'outil Cheddar ou MAST.

La deuxième direction considérée beaucoup plus générique, consiste à transformer un modèle AADL vers un modèle formel ayant une sémantique bien définie. Cette approche est généralement exhaustive du point de vue comportemental et permet de couvrir un ensemble important de comportements AADL. Par contre, cette approche nécessite une connaissance préalable sur les techniques de vérification.

Nous avons essayé de faire une petite évaluation de ces travaux sachant que chaque travail sera évalué en fonction des points suivants :

- Le modèle cible de la transformation
- L'objectif de la transformation
- Les composants AADL concernés par la transformation
- Les outils utilisés pour la transformation/analyse
- Prise en charge de l'aspect sémantique de AADL
- Vérification par Model-checking
- Transformation manuelle / automatique

Les différents points de comparaison cités ci-dessus ne se veulent pas exhaustifs, mais permettent d'illustrer les points en différences ou en communs entre ces travaux.

Travail	modèle cible	L'objectif	composants AADL	outils	aspect sémantique	Vér. par Model-checking	Tran. manuelle/automatique
T. Vergnaud et al. 2006	Rdp coloré	Vérifier l'interblocage et la circulation des données	Tous les composants architecturaux	Ocarina	non	non	manuelle
J. Champeau et al. 2008	Langage IF	Vérifier les protocoles de communication	system, process, thread , data, port	CADP, Kronos, TGV	oui	non	manuelle
L. Pi et al. 2009	Langage Fiacre	Réduire le niveau d'abstraction entre AADL et outils de vérification	system,process, thread, connexion	TopCased	oui	non	manuelle
P. Hladik et al. 2010	Pola	Vérifier l'ordonnancement des thread	system, process, thread , data, subprogram, thread behavior	ATL, Acceleo, TINA	non	oui	manuelle
P. Benammar et al. 2010	Logique de réécriture	vérifier propriétés temporellles	system, process, thread , data, subprogram	RT-Maude	oui	oui	manuelle
Y. Zhang et al. 2011	automates temporisées	Vérifier la configuration de l'architecture	Tous les composants architecturaux	non spécifier/Uppaal	Non	oui	manuelle
Z. Yang et al. 2014	TASM	consommation de ressources, synchronisation	thread, port, mode, annex	OSAT, TASM toolset, Uppaal	oui	oui	manuelle
P. Gracy et al. 2018	-	vérifier la sécurité	Annexe comportementale et BLESS	-	oui	non	manuelle

TABLE 4.1: Une comparaison récapitulative entre quelques travaux sur AADL

Il faut juste noter que le travail (Pi *et al.*, 2009) prend en compte le comportement global de l'architecture AADL dans le but de vérifier les protocoles de communication mis en place dans l'architecture AADL. Par ailleurs, le travail (Hladik *et al.*, 2010) cherche à formaliser efficacement l'ordonnancement entre ***threads*** dans une architecture AADL. Dans le travail

(Vergnaud, 2006) et (Zhang *et al.*, 2011) montrent que la transformation vers les formalismes de type automate (les réseau de pétrie coloré ou les automates temporisés,) est justifiée par le fait que ces deux modèles permettant d'étudier l'architecture globale vise-à-vis des interactions entre les différents nœuds applicatifs, par exemple le réseau de Pétri généré dans le travail (Vergnaud, 2006) permet à partir d'une description AADL de modéliser la circulation des données et les flux d'exécution. Par ailleurs, le travail (Zhang *et al.*, 2011) cherche à modéliser via les automates temporisés le changement de mode d'un système AADL lors de la réception d'un événement. Une autre remarque est que le résultat de la transformation reste en général lié à l'environnement de développement et cela est justifié puisque la plupart des outils utilisés sont des outils à caractère académique.

Ce qui est remarquable aussi d'après cette étude est que la vérification concerne toujours un sous ensemble de AADL et cela à cause du rapport richesse/complexité de AADL. Une autre remarque qu'on peut signaler est le manque d'intérêt envers les contraintes temporelles malgré qu'AADL offre un ensemble de mécanismes qui prennent en charge cet aspect. La même chose pour l'annexe comportementale malgré que ce dernier à une grande importance puisqu'il permet de décrire le comportement interne du composant *"**thread**"* ou sous programme.

Conclusion

Au niveau de la vérification du comportement, l'ensemble des méthodes et approches essayent de transformer les modèles AADL vers des modèles spécifiques. Ces derniers, sont liés directement à des outils basés sur des sémantiques fortes (comme : algebra ACSR, Lustre, BIP, TLA+, TPN, GSPN, IF, Fiacre, RT-Maude, Compass, etc.) ou bien la cible soit un modèle d'analyse classique tel que : les réseaux de Pétri, les machines à état, etc.

A notre connaissance, aucun des travaux étudiés n'a fourni une démarche standard pour automatiser le processus de transformation d'AADL vers le modèle formel souhaité (Voir la dernière colonne du tableau 4.1). Pour cela, nous estimons qu'une approche basée sur l'IDM permettra de contourner cette limite. Cette vision sera discutée dans le chapitre suivant.

Partie 2 : Approche de transformation de AADL vers les automates temporisés

CHAPITRE 5

PRÉSENTATION DE NOTRE APPROCHE

" Galilée : La science ne connaît qu'une loi : la contribution scientifique."
Bertolt Brecht – La Vie de Galilée

Sommaire

Introduction

Dans ce chapitre, nous allons présenter une approche permettant d'automatiser la transformation & vérification des descriptions AADL. Notre proposition s'appuie sur deux points essentiels : une démarche de transformation de modèle et une démarche de vérification basée sur le Model-checking.

Avant de présenter notre approche, nous justifions d'abord la tendance de formalisation dans le cycle de développement des architectures AADL. Ensuite, nous préciserons l'objectif de la vérification. Puis, nous détaillerons les différentes étapes de base sur lesquelles repose notre proposition avant de conclure.

5.1 Vers un nouveau cycle de développement de AADL

Le développement d'un système embarqué temps-réel nécessite la vérification régulière du respect des spécifications tout au long de son développement. Une approche de développement itérative permet une rétroaction entre l'application et ses spécifications initiales. Cette approche consiste en une démarche de conception pas à pas permettant la validation de l'architecture. Il est ainsi possible de détecter les problèmes relativement tôt, ce qui permet d'éviter de coûteuses modifications sur l'architecture finale (CHKOURI, 2010). Les modèles AADL sont exploités de différentes manières, comme montre la Figure 5.1 :

— Différentes opérations peuvent être effectuées pour vérifier la cohérence des contraintes exprimées sur l'architecture ;

— Un système exécutable peut être généré afin de tester et déboguer la conformité vis-à-vis des contraintes temporelles et spatiales ;

— Des représentations formelles peuvent être extraites du modèle AADL afin de valider le comportement des entités ;

Nous avons vu aux chapitres précédents qu'AADL se focalise essentiellement sur les descriptions architecturales, et permet de modéliser les architectures avec une représentation centrale. L'usage des propriétés et des annexes permet de spécifier les descriptions comportementales des composants.

La syntaxe d'AADL permet une grande souplesse : il n'est pas nécessaire de fournir tous les détails d'une architecture pour pouvoir en exploiter la description. Il est ainsi possible de

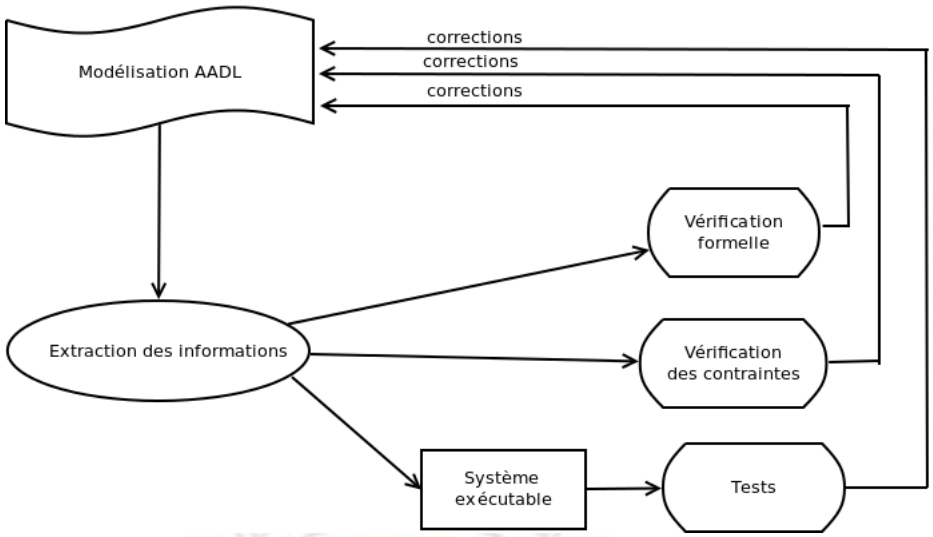


FIGURE 5.1: Cycle de développement (CHKOURI, 2010)

ne spécifier que les informations qui sont pertinentes pour une exploitation donnée de la modélisation. En contrepartie, cette liberté entraîne une dispersion des éléments de description qui limite les moyens de vérification et d'analyse de l'architecture (CHKOURI, 2010).

Notre objectif final est de décrire tous les éléments architecturaux d'une application avec AADL pour la transformer automatiquement vers le formalisme des automates temporisés, ayant une sémantique opérationnelle formellement définie.

5.2 Objectif de la vérification

Précédemment, nous avons vu que la vérification d'une architecture AADL peut porter sur la structure de la description, comme la validité des connexions ; elle peut aussi concerner le respect des contraintes exprimées dans les propriétés AADL, telles que les temps d'exécution et les politiques d'ordonnancement (Singhoff *et al.*, 2005). De la même façon que la génération de code source à partir d'une description AADL a pour objectif la production d'un système exécutable, la production d'une représentation formelle doit correspondre à un objectif de vérification défini (Hamdane *et al.*, 2013a).

Dans notre travail nous nous focalisons sur la vérification du comportement de la description architecturale de AADL. En effet, nous exploitons la description du comportement

interne de chaque composant AADL de type ***thread*** et plus précisément sur la partie annexe comportementale afin d'extraire une représentation formelle sous forme d'automates temporisés qui expriment le comportement global de l'architecture. Notre objectif est de vérifier des propriétés telles que :

— Les communications entre les composants utilisés sont définis de façon déterministe,

— Les contraintes temporelles sont bien respectées,

— La structure de l'architecture n'engendre pas systématiquement un inter-blocage,

— ...etc.

5.3 Présentation de l'approche proposée

L'approche que nous avons mise en œuvre dans ce travail est l'exploitation des techniques liés à la technologie IDM pour vérifier une description AADL. En effet, à travers cette approche l'objectif est de transformer un modèle AADL vers un modèle d'analyse de type automate temporisé qui sera interprété/analysé par le modèle checker UPPAAL. La Figure 5.2 décrit notre première solution :

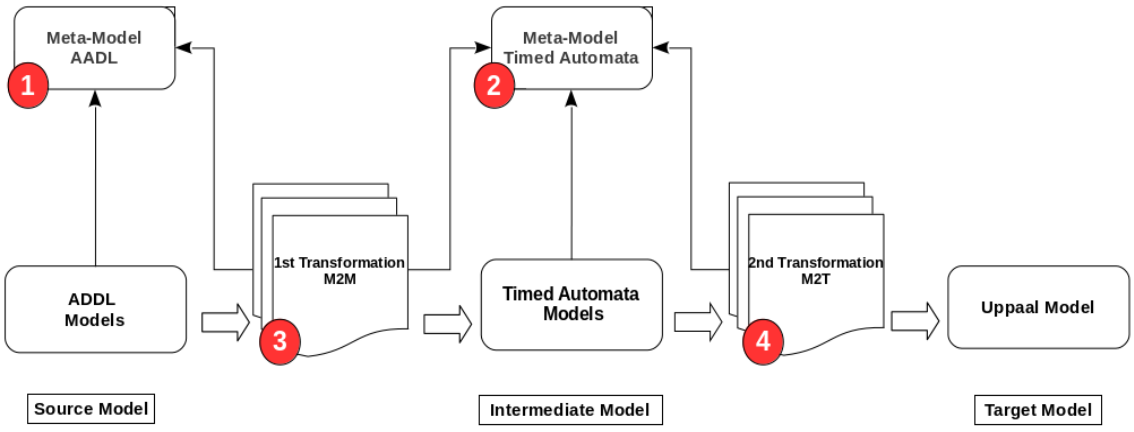


FIGURE 5.2: Première solution(Hamdane *et al.*, 2013a)

Dans le but de structurer la transformation nous avons amélioré la solution proposée. La figure 5.3 présente une approche de transformation & de vérification des descriptions AADL :

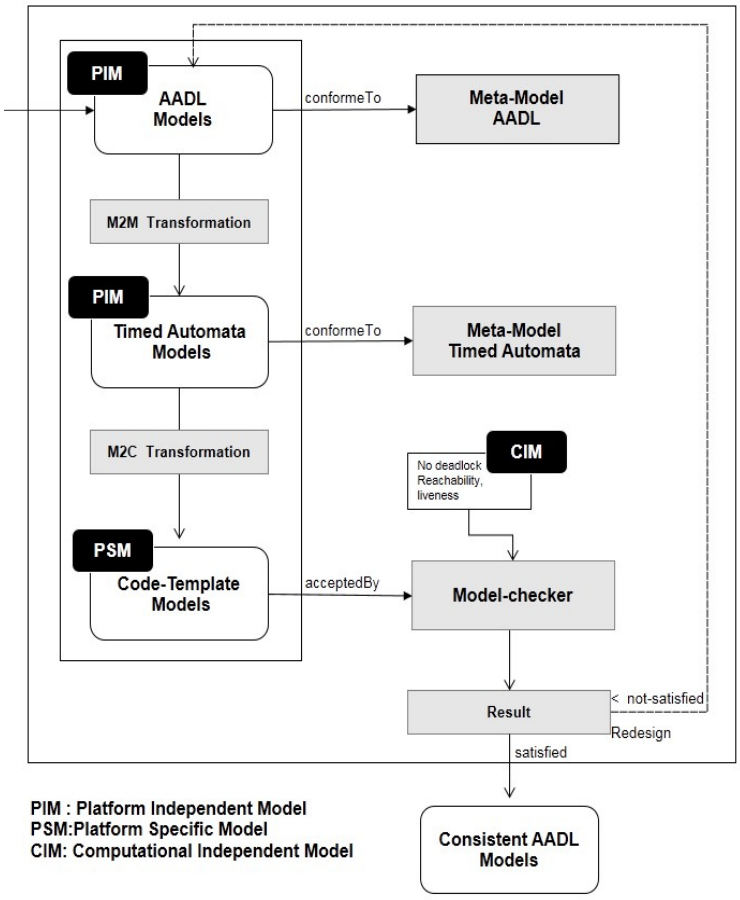


FIGURE 5.3: Cadre général de l'approche proposée (Hamdane *et al.*, 2017)

Selon la figure ci-dessus, l'entrée de l'approche est un modèle PIM-AADL. Ensuite, trois phases principales doivent être accomplies :

— **La première étape** : nous avons proposé en premier temps deux méta-modèles. Le premier est le méta-modèle PIM-AADL qui est le méta-modèle source et le second est celui des automates temporisés qui est considéré comme le méta-modèle cible (PIM-TimedAutomata) dans la première transformation. Nous avons utilisé comme langage de métamodélisation le langage **Ecore**, défini par IBM [1] et utilisé dans le framework de modélisation d'Eclipse Modeling Framework(EMF (Steinberg *et al.*, 2008)), la définition de ce langage ainsi que les concepts de bases et un exemple démonstratif seront détaillées dans le chapitre suivant.

1. http://www.ibm.com

— **La deuxième étape** : Nous allons définir un processus de transformation en deux phases :

- *La première phase* : consiste à prendre le modèle **PIM-AADL** et de le transformer vers un modèle intermédiaire de type automates temporisés (PIM-TimedAutomata). Cette transformation est de type modèle-vers-modèle. Le choix des automates temporisés est justifié par le fait que ce formalisme constitue un modèle formel pour décrire les architectures à caractère temporel et par conséquent il fournit un support formel à leur analyse. De plus, il y a autant d'outils de vérification qui sont développés autour de ce formalisme. Nous avons utilisé comme langage de transformation le langage **ATL** (ATLAS Transformation Language) (Jouault et Kurtev, 2006) qui se conforme au standard **QVT** (Jouault et Kurtev, 2006) définit par **l'OMG** [2] (Object Management Group). Ainsi, ce langage est un plug-in dans le framework **Eclipse**. Dans le chapitre suivant on va donner un exemple démonstratif pour comprendre le principe du langage **ATL**.
- *La deuxième phase* : ce processus est une transformation de type ***Model2Text***. En effet, on cherche ici à générer à partir du modèle automate temporisé produit par la phase précédente (PIM-TimedAutomata) un modèle spécifique à l'outil d'analyse UPPAAL. Le format accepté par cet outil est *"ta-format"* (Larsen *et al.*, 1997). Nous avons utilisé comme langage de transformation dans cette phase le langage **Xpand** [3]. Ce langage est une technologie IDM pour réaliser les transformations ***model2Text***. De plus, ce langage est un **plug-in** dans le framework Eclipse.

— **La troisième étape** : cette étape consiste à faire transmettre la description générée vers l'outil **UPPAAL** (Larsen *et al.*, 1997). Dès que la description sera acceptée on peut exploiter tous les services offerts par le vérificateur de modèle **UPPAAL** et particulièrement l'évaluation de certaines propriétés du modèle comme : l'absence du blocage, l'atteignabilité, la sûreté, la vivacité, etc. Une présentation introductive sur l'outil **UPPAAL** sera donnée dans le chapitre suivant.

2. http://www.omg.org
3. http://wiki.eclipse.org/Xpand

Remarque :
Il est très intéressant de noter que la transformation proposée ne doit pas être interprétée sémantiquement dans un sens mathématique $A \Leftrightarrow B$, mais plutôt : B peut être déduit de A (Hamdane *et al.*, 2017).

5.3.1 Étape 1 : Métamodélisation

Le premier méta-modèle que nous avons proposé, représente le méta-modèle source du langage AADL. Ce dernier permet de créer des modèles conformes au langage AADL. Un modèle AADL est composée d'un composant système (system), un ou plusieurs processus (***process***), un ou plusieurs ***threads***, des sous programmes (***subprogram***) ou bien des données (***data***).

Afin de réduire la complexité d'abstraction entre le modèle AADL et les automates temporisés d'un côté et de tenter de maîtriser la richesse et la complexité du AADL d'un autre côté, le méta-modèle proposé doit respecter les hypothèses suivantes :

— Le méta-modèle prend en considération uniquement les composants logiciels,

— Seules les propriétés liées à l'aspect comportemental sont pris en considération,

— Nous exploitons seulement deux types de ***feature*** : (i) type port et (ii) le type accès à un sous programme,

— Le nombre des données partagé entre les composants égal à 1.

(A) Méta-modèle de AADL La Figure 5.4 montre le Méta-modèle de AADL.

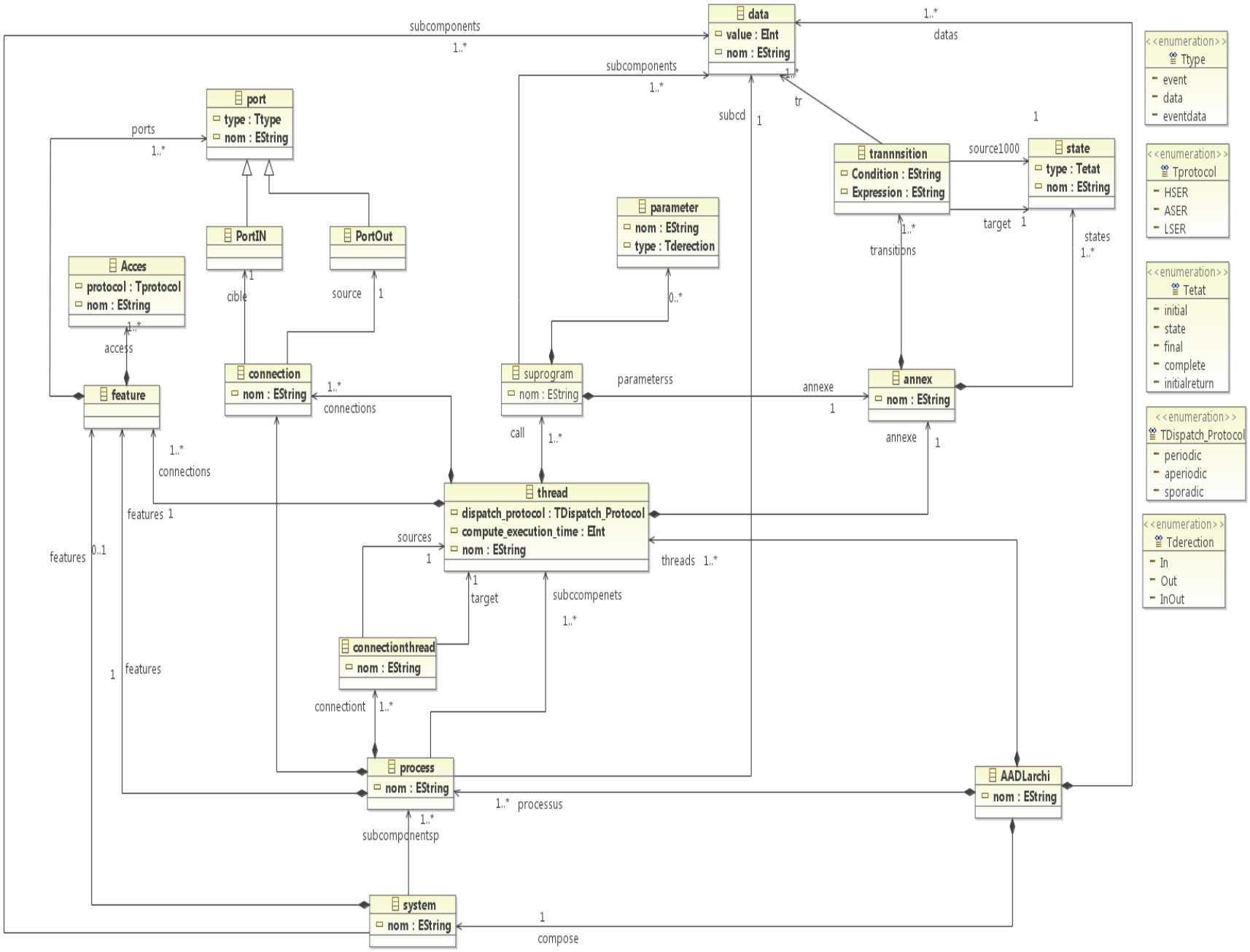


FIGURE 5.4: Méta-modèle proposé pour un sous ensemble AADL (Hamdane *et al.*, 2013b)

(B) Méta-modèle des automates temporisés

Le deuxième méta-modèle que nous proposons (Hamdane *et al.*, 2013b; Hamdane et Chaoui, 2011) est celui des automates temporisés. D'une façon générale, un automate temporisé est composé de plusieurs états, horloge , transitions, chaque transition a un état source, un état cible et un ensemble d'horloges qui doivent être remises à 0. Le méta-modèle proposé est représenté sur la Figure 5.5 :

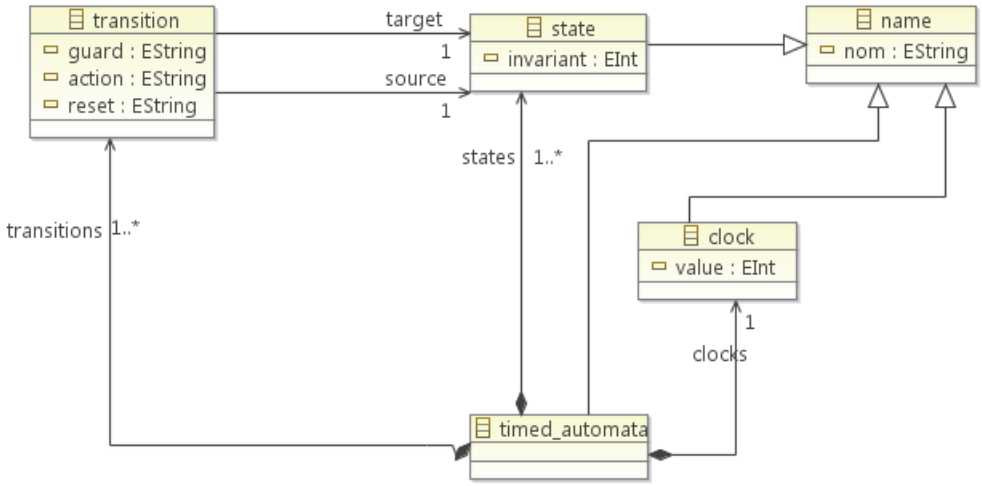


FIGURE 5.5: Méta-modèle proposé pour les automates temporisés (Hamdane et Chaoui, 2011)

5.3.2 Étape 2 : Processus de Transformation

Ce processus est réalisé en deux phases, il permet de réduire le niveau d'abstraction entre AADL considère comme étant un langage de modélisation de haut niveau et le formalisme des automates temporisés considère comme langage de modélisation de bas niveau. Le tableau suivant propose le principe de la transformation :

Concept AADL	**Concept automate temporisé**
process	automate temporisé
thread	état
data	horloge
Connexion entre deux threads Dans l'annexe comportementale du thread : à partir de la transition attachée à l'état final; elle est de la forme : $s_m - [condition] -> s_n\{expression\}$	***Transition entre deux états*** — $guard =$ condition, — $action$ = l'information envoyée aux portes de sortie, — $reset =$ expression.

TABLE 5.1: Concepts AADL transformable dans les automates temporisés

Comme nous l'avons montré au chapitre précédent, l'annexe comportementale dans une description AADL est définie pour indiquer le comportement interne des ***threads*** et les ***sous-programmes*** sous forme de machine à états. Certains éléments de l'annexe comportementale peuvent être traduits directement par des éléments dans l'automate temporisé alors que d'autres leur traduction est compliquée.

— **(A) la première phase** :

Elle a pour objectif de traduire des modèles AADL, conformes au méta-modèle AADL (Figure 5.4) en des modèles automates temporisés, conformes au méta-modèle automate temporisé(voir Figure 5.5).

Les deux méta-modèles sont représentés dans le langage Ecore (Steinberg *et al.*, 2008). Les règles de transformation font appel aux constructions ou méta entités des méta-modèles source et cible. Ces règles sont exprimées dans le langage ATL (Jouault et Kurtev, 2006) qui est un langage standard de transformation recommandé par l'OMG. Nous avons développé cinq règles de transformation pour assurer le passage d'un modèle AADL vers un modèle automate temporisé :

▷ ***Règle 1 : Process vers Automate Temporisé***

La première règle consiste à transformer toute occurrence d'un composant *process* dans une description AADL vers un automate temporisé tel que :

— Le nom de l'automate temporisé reçoit le nom du process

— La référence ***subcComponentt*** (cf. Figure 5.4) qui fait référencer un ***thread*** dans le méta-modèle AADL sera traduite par une référence ***states*** qui fait référencer un état dans le méta-modèle automate temporisé

— La référence ***subcd*** dans le méta-modèle AADL sera transformée par une référence de type ***clocks*** qui pointe une horloge dans le méta-modèle d'automate temporisé.

Concept AADL Source	*Concept automate temporisé cible*

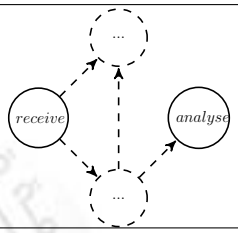
`process prc`
`end processing;`
`process implementation prc.others`
`subcomponents`
`receive : thread receiver.impl;`
`analyse : thread analyser.impl;`
`. . .`
`end prc.others;`
	

TABLE 5.2: Exemple de règle : Process2timedAutomaton

▷ ***Règle 2 : Thread vers State***

Dans cette règle la transformation consiste à traduire chaque composant ***thread*** au niveau du méta-modèle AADL par un état dans le méta-modèle automate temporisé tel que :

— L'état de l'automate prend le même nom d'un thread AADL

— L'invariant de l'état dans l'automate temporisé reçoit la valeur de la propriété ***"compute_execution_time" ou bien "period"***.

Concept AADL Source	*Concept automate temporisé cible*

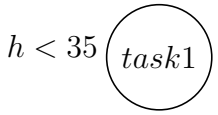
`thread task1`
`features`
`. . . . . .`
`properties`
`period --> 35`
`. . . . . .`
`end task1;`
`thread implementation task1.impl`
`. . . . . .`
`end task1.impl;`
	

TABLE 5.3: Exemple de la règle Thread2State

À partir de cette étape, on a pu extraire tous les états de l'automate ainsi que la valeur de leurs invariants c'est à dire le résultat de cette règle est les ensembles Q et I (cf. section 3.3.2, chapitre 3).

▷ ***Règle 3 : Data vers clock***

Le rôle de cette règle est de récupérer l'ensemble d'horloge. En effet, dans la description AADL nous avons pris seulement la donnée implémentée par un processus ; ces derniers se traduisent par une horloge au niveau de l'automate temporisé qui a le même type et le même nom.

Concept AADL Source	*Concept automate temporisé cible*

`data h`
`end h;`
`data implementation h.imp`
`..........`
`end h.imp;`
	`clock h ;`

TABLE 5.4: Exemple de la règle Data2clock

▷ ***Règle 4 : Connection vers Transition***

Cette règle permet de construire l'ensemble E des transitions. En effet, les connexions entre les ***Threads*** sont projetées sous forme de transitions entre les états dans l'automate temporisé.

Les informations attachées à la transition sont récupérées à partir de l'implémentation et l'annexe comportementale.

Nous définissons une transition T_k entre S_i et S_j ; s'il y a une connexion entre deux ***threads*** tels que :

(a) S_i est l'état source de la transition,

(b) S_j est l'état cible de la transition,

(c) Dans l'annexe comportementale de S_i , à partir de la transition attachée à l'état final ; elle est de la forme : $s_m - [condition] - > s_n\{expression\}$:

— $guard =$ condition,

— $action =$ l'information envoyée aux portes de sortie,

— $reset =$ expression,

Concept AADL Source	*Concept automate temporisé cible*

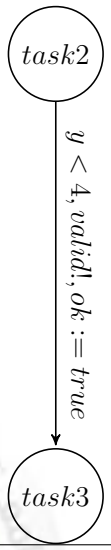
`thread task2`
`features`
`PortOut1 in event data port`
`Behavior::integer;`
`...`
`properties`
`...`
`end task2;`
`thread implementation task2.impl`
`connections`
`C1: h task.PortOut1->task3.PortIn1;`
`......`
`annex behavior_specification{**`
`.......`
`s3-[y<4]`
`-> S4{valid!, ok:=true, PortOut1!y;}`
`**}`
`end task2.impl`
	

TABLE 5.5: Exemple de la règle Connection2Transition

À cette étape nous avons terminé la construction de l'ensemble des transitions E.

Donc, le résultat de l'application de ces règles est la construction de tous les éléments nécessaires pour la production d'un modèle automate temporisé à partir d'un modèle AADL.

Remarque :

Il faut juste noter que dans notre travail et par hypothèse ; l'approche générera un automate temporisé qui doit être déterministe ; mais si le cas contraire se produit c'est-à-dire dans un état donnée ; un événement peut déclencher plusieurs transitions, alors l'utilisateur est sollicité pour enlever l'indéterminisme.

— **(B) La deuxième phase :**

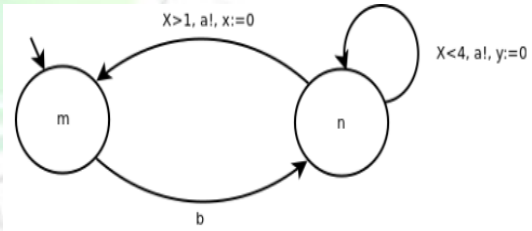
La génération de code **"ta_format "** à partir d'un automate temporisé a pour but la production d'un modèle manipulable par le model-checker UPPAAL, et la production d'une représentation formelle qui doit correspondre à un objectif de vérification défini. Concrètement, cette étape prend en entrée l'automate produit par la phase précédente et produit

automatiquement en sortie une structure textuelle exploitable par l'outil UPPAAL sous forme de code ta-format (voir Appendix-A qui décrit la ***Forme Normale de Backus*** de ce langage).

A fin de comprendre la structure du code cible qui est le résultat de cette deuxième phase dans le processus de transformation, le Listing 5.1 montre la correspondance dans UPPAAL entre la description textuelle d'un automate temporisé sous forme de code **"ta_format "** et sa représentation graphique :

```
1  // global declaration
2  clock x,y;
3  chan a,b;
4  // process description
5  process P1{
6  state m,n;
7  init m;
8  trans n-->m{
9  guard x>1;
10 sync a!;
11 assign x==0;
12 };
13 trans n-->n{
14 guard x<4;
15 sync a!;
16 assign y==0;
17 };
18 trans m-->n{
19 sync b?;
20 };
21 }
22 // system configuration
23 system exemple P1;
```

Listing 5.1: Correspondance ente le code "ta-format" et l'annotation graphique d'automate temporisé



On remarque que la description textuelle sous ta-format est subdivisée en trois compartiments : le premier (**// global declaration**) va contenir les déclarations des variables en terme d'horloges et de canal de communication. Le deuxième compartiment (**// process description**) va contenir les états de l'automate ainsi que les transitions entre états. Le dernier compartiment (**// system configuration**) va contenir la configuration de l'automate.

Pour assurer le passage entre l'automate produit par la première phase du processus de transformation et la description en **"ta-formt"**, nous proposons les règles de transformation :

— ***Règle 1*** : Cette règle est appliquée pour créer un fichier avec l'extension «.ta » et garder le ouvert en mode écriture. Par la suite, cette règle permet aussi de récupérer le nom de l'automate et de diviser (décorer) le fichier en trois compartiments à savoir : **(i)** la partie

définie par // *GlobalDeclaration*, **(ii)** la partie définie par // *ProcessDescription*, **(iii)** la partie définie par // *SystemConfiguration*.

— ***Règle 2 : Concerne les états de l'automate***

L'application de cette règle permet de récupérer pour chaque état de l'automate les informations suivantes :le nom et l'invariant. Par la suite, on va remplir la partie *ProcessDescription* par les informations récupérées.

— ***Règle 3 : concerne les horloges de l'automate***

Cette règle est appliquée pour récupérer les horloges de l'automate et les placer dans la partie *GlobalDeclaration*.

— ***Règle 4 : concerne l'état initial***

Cette règle permet de récupérer le nom et l'invariant de l'état initial de l'automate temporisé.

— ***Règle 5 : concerne les transitions de l'automate***

L'application de cette règle permet de récupérer pour chaque transition : l'état source et cible, la garde, l'action et le reset. Ces informations seront placées dans la partie *ProcessDescription*.

La dernière action après une exécution correct de ces règles est la configuration de l'automate au niveau de la partie ***SystemConfiguration*** et de fermer ainsi le fichier de sortie.

A ce point de l'approche nous arrivons à générer à partir d'un modèle AADL une description équivalente et interprétable par UPPAAL.

5.3.3 Étape 3 : Analyse et Vérification

Le résultat de l'étape précédente est l'entrée de celle-ci. En effet, le modèle ***PSM-UPPAAL*** de l'automate temporisé est maintenant prêt à être exploiter par le model-checker UPPAAL. Dans cette étape, nous pouvons exploiter le modèle des automates temporisés de deux manières :

(a) **Simulation** : elle permet de constater plusieurs comportements du modèle AADL décrit par son automate temporisé en identifiant les chemins accessibles et les transactions fran-

chissables selon les contraintes d'horloges. Par exemple en laissant le choix aux utilisateurs de choisir l'interaction à exécuter ou de laisser le choix au simulateur.

(b) **Vérification** : la vérification d'une architecture AADL peut porter sur la structure de la description (comme la validité des connexions par exemple) où bien sur le comportement. L'avantage de cette technique est qu'elle donne la possibilité ***d'interroger le modèle*** en vue de tester des propriétés comme : l'absence de l'inter-blocage, l'atteignabilité, la vivacité, etc.

Formuler les propriétés dans la logique temporelle :

Au niveau de cette étape et en plus du modèle ***PIM-UPPAAL*** généré à l'étape 2, une intervention de l'utilisateur est nécessaire car nous avons besoin de **formuler les propriétés** qu'on cherche à vérifier sur le modèle AADL. En effet, des propriétés de sûreté et de vivacité peuvent alors être vérifiées. Ces propriétés sont exprimées généralement sous une forme de logique modale, par exemple la logique : **LTL, CTL** (cf. section 3.2.3, chapitre 3).

Conclusion

Dans ce chapitre, nous avons proposé une approche basée sur l'ingénierie dirigée par les modèles et la technique de Model-checking pour la transformation & la vérification des descriptions AADL. L'avantage de cette proposition est quelle offre plus d'automatisme dans le processus de transformation.

Cette approche vise à offrir plus de garantie dans le cycle de développement des applications AADL et ouvrir ainsi une fenêtre vers les outils de vérification formelle à base des automates temporisés. En suivant les principes de l'ingénierie dirigée par les modèles, nous avons proposé deux méta-modèles l'un pour AADL et l'autre pour le formalisme des automates temporisés. De plus, un processus de transformation a été proposé afin d'assurer le passage de AADL vers les automates temporisés. Ce formalisme et grâce à sa richesse permet de retranscrire la plu part des éléments qu'on trouve dans une description AADL.

Dans le cadre de la mise en œuvre de l'approche proposée, il est essentiel de disposer d'outils complémentaires et compatibles qui peuvent être regroupées dans un seul cadre afin

de former une chaîne d'outils qui supporte notre approche. Cela, fait d'objet du prochain chapitre.

CHAPITRE 6

MISE EN ŒUVRE DE L'APPROCHE

" Experience is a hard teacher because she gives the test first, the lessons afterwards. "

Vernan Law

Sommaire

Introduction

Dans de chapitre nous allons montrer la validation de l'ensemble des idées exposées dans le chapitre précédant sur deux exemples réels.

Il est organisé comme suit : nous commençons d'abord par introduire les outils utilisés pour implémenter l'approche proposée. Après avoir présenté les études de cas, nous l'appliquerons aux étapes de transformation pour élaborer l'automate temporisé à partir d'un modèle AADL. Ensuite, nous exposerons l'automate temporisé généré sur le model-checker UPPAAL et vérifierons ainsi certaines propriétés à savoir : l'absence de blocage, l'atteignabilité et la sûreté. Nous terminerons par une conclusion .

6.1 Environement et choix techniques

Dans cette section on va introduire brièvement les outils utilisés afin de mettre en œuvre notre démarche.

Etant donné que l'approche proposée est dirigée par les modèles, nous avons utilisé un cadre technique qui permet de mettre en œuvre les différents aspects liés à l'IDM, à savoir : les modèles, les méta-modèles, les transformations et la vérification. Dans le paragraphe suivant, nous expliquerons nos besoins techniques. Ensuite, nous décrirons les différents outils que nous avons utilisés afin de développer un système AADL.

6.1.1 Besoins

Pour arriver à réaliser l'approche proposée, nous avons besoin de langages et d'outils qui fournissent :

1. Un environnement convenable pour créer les méta-modèles, les modèles et de garantir ainsi la relation de conformité d'un modèle avec son métamodèle,
2. Des langages de transformation (M2M et M2C) facilitant le passage d'un modèle source vers un modèle cible, conformes respectivement à un métamodèle source et cible,
3. Un environnement standard permettant d'intégrer l'ensemble des techniques liées à l'IDM telles que : la méta-modélisation, l'instanciation et la transformation, et qui permet également de tenir compte des standards souvent utilisés,

4. Un model-checker académique stable qui permet l'analyse et la vérification des automates temporisés.

6.1.2 Outils pour la mise en œuvre de notre approche

Il existe plusieurs logiciels pour implanter une approche IDM (Diaw *et al.*, 2010). Cependant, certains outils se contentent de fournir un environnement restreint qui permet de répondre uniquement à un aspect particulier de modélisation ou de transformation (Atigui, 2013). Notre objectif est de trouver un cadre technique convenable pour la modélisation, la méta-modélisation ainsi que la transformation, y compris de modèle vers modèle et de modèle vers code.

Nous avons eu recours à la plateforme Eclipse Modeling Framework [1] (EMF) qui présente un environnement complet pour la mise en œuvre de notre démarche IDM. La figure 6.1 résume l'ensemble des outils utilisés pour implémenter l'approche proposée.

6.1.2.1 Eclipse Modeling Framework (EMF)

EMF (BUDINSKY *et al.*, 2003) est le framework de modélisation intégré dans l'atelier de développement Eclipse. Eclipse [2] est un système logiciel intégré initié par IBM [3] sous licence **"Open source"** et dont les caractéristiques sont : l'extensibilité, l'universalité et la polyvalence. Cet environnement permet de créer des projets de développement mettant en œuvre n'importe quel langage de programmation. La spécificité d'Eclipse vient du fait de son architecture totalement développée autour de la notion de plug-in, toutes les fonctionnalités de cet atelier logiciel sont développées en tant que plug-in.

EMF présente un cadre permettant l'élaboration d'applications basée sur les modèles. Il présente un environnement technique assez complet couramment utilisé dans le développement dirigé par les modèles. EMF repose sur trois technologies : **Java, XML et UML**. Il permet ainsi de décrire un modèle sous forme d'un diagramme **UML** ou d'un schéma XML ou ainsi en utilisant le langage Java. Il est possible d'utiliser l'une de ces représentations et de générer les deux autres (Atigui, 2013).

1. http://www.eclipse.org/modeling/emf/
2. http://www.eclipse.org
3. http://www.ibm.com

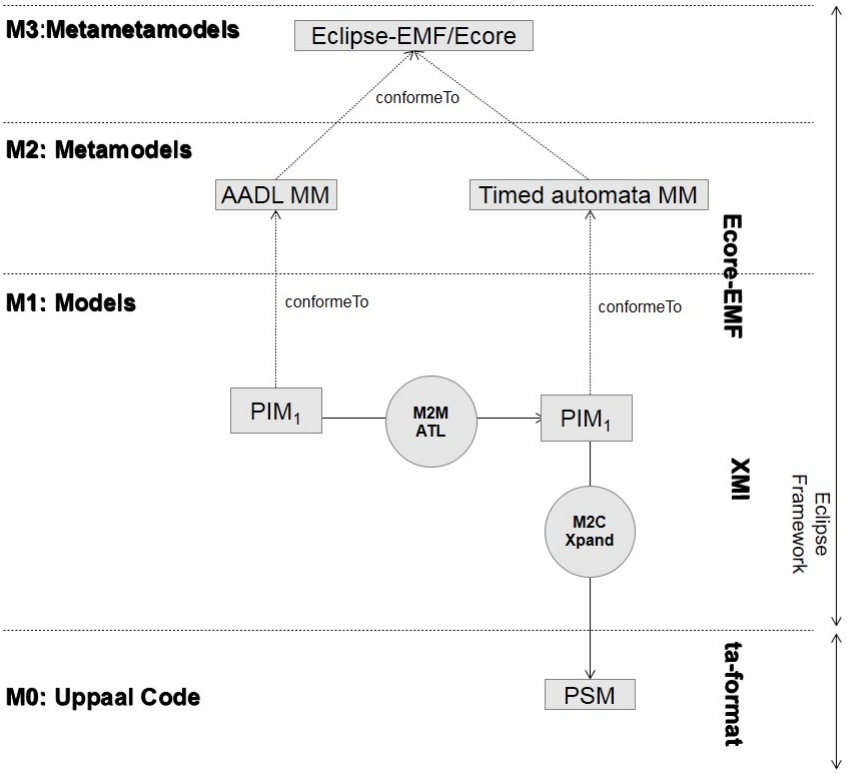


FIGURE 6.1: Outils utilisés pour implémenter l'approche proposée

EMF est conforme à la spécification minimale du **MOF** (Essential MOF, cf. section 2.1.3 dans le chapitre 2). Le langage de définition des méta-modèles d'EMF est **Ecore**. Ce langage définit toutes les entités apparaissant dans un modèle conforme au MOF. EMF permet de spécifier uniquement des éléments de structure, et ne permet pas de spécifier des aspects comportementaux. A l'origine, l'objectif d'EMF est de modéliser des programmes Java et travailler au niveau modèle en plus du code. Pour cela, il offre une gamme d'outils complète permettant de créer des méta-modèles, générer les éditeurs de modèles, et les API Java pour la manipulation de modèles, et la génération de code Java grâce notamment à la technologie **EMF/Jet**. Néanmoins, la technologie **EMF** peut s'intégrer avec d'autres outils, plus généralistes, grâce au mécanisme de plug-in de la plateforme Eclipse.

6.1.2.2 Mise en œuvre des méta-modèles (Ecore)

Ecore (Steinberg *et al.*, 2008; JET, 2004) est un langage de méta-modélisation définissant les concepts de manipulation des modèle dans EMF. Ces concepts, toujours préfixés par un "E", le méta-modèle Ecore contient les informations sur les classes définies. Sa forme simplifiée est représentée par les éléments suivants (Steinberg *et al.*, 2008) :

— ***EClass*** : représente une classe, avec zéro ou plusieurs attributs et zéro ou plusieurs références.

— ***EAttribute*** : représente un attribut qui a un nom et un type.

— ***EReference*** : représente une extrémité d' une association entre deux classes. Il a un drapeau pour indiquer si la relation est utilisée dans la modélisation ou non, et un référence de classe vers lequel il pointe.

— ***EDataType*** : représente le type d'attribut, par exemple, int, float ou tout simplement l'ensemble de types de base de la bibliothèque de java.util.*.

La Figure 6.2 représente le méta-modèle **Ecore** simplifié :

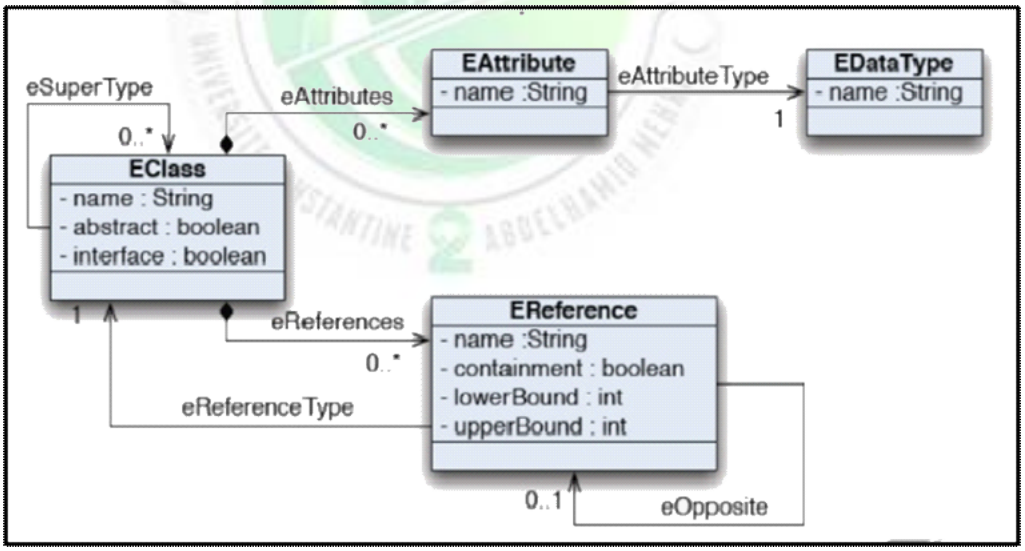


FIGURE 6.2: Métaméta-modèle Ecore simplifié

6.1.2.3 Prise en main de Ecore

Pour créer un méta-modèle utilisant EMF il faut suivre les étapes suivantes :

1. Lancer Eclipse EMF,
2. Aller à file → new → Project, la fenêtre suivante (cf. Figure 6.3) :
3. Créer un projet Eclipse Modeling Framework vide (Empty EMF Project) et précise son nom et l'emplacement d'enregistrement,

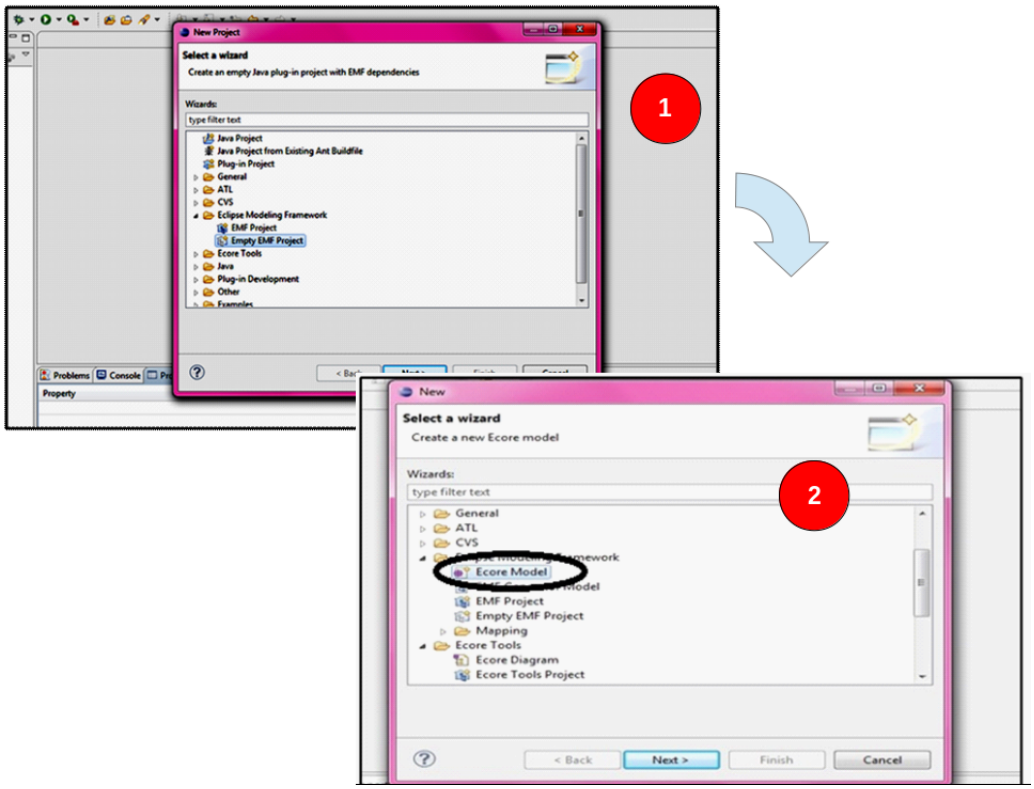


FIGURE 6.3: Premières étapes pour créer un méta-modèle avec Ecore

4. Pour créer un méta-modèle **Ecore**, sélectionner le projet que vous avez créé précédemment. Et sélectionner en suite : new→ **other**,
5. Lorsque vous cliquez sur **Other** une fenêtre s'affiche. Elle sert à sélectionner le type de fichier à créer : choisir **Eclipse Modeling Framework**, puis **Ecore model**. La figure **??** montre le résultat de cette action.

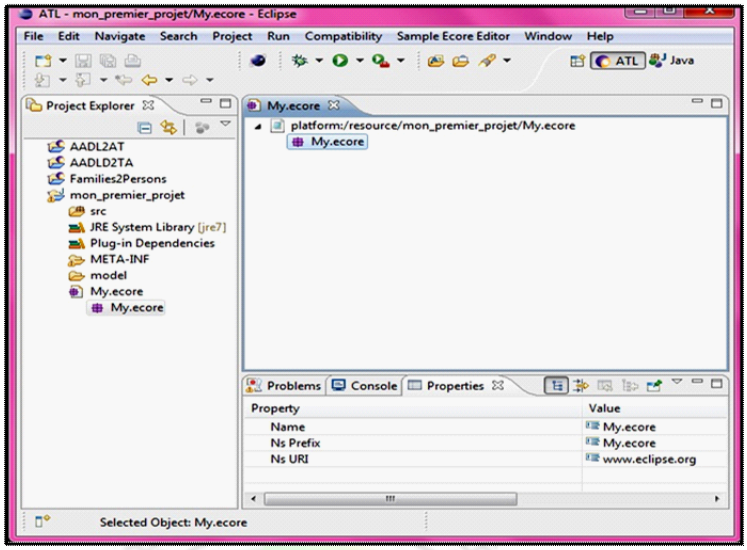


FIGURE 6.4: Métaméta-modèle Ecore simplifié

Remarque :

- Lorsqu'on définit un méta-modèle avec Ecore (voir la zone Proporties dans la Figure 6.4), il faut que les informations suivantes soient correctes : **(i)** Name : contient le même nom que fichier Ecore, **(ii)** Ns Prefix : contient le même nom que le fichier Ecore.**(iii)** Ns URI : contient une adresse internet quelconque exemple : www.eclipse.org.

- Ces trois informations sont nécessaires pour pouvoir par la suite créer des instances du méta-modèle ou bien pour initialiser le **Ecore Diagram**.

6. Pour créer les classes du méta-modèle, les attributs de ces classes ainsi que les références entre ces dernières, suivez les étapes suivantes :
   (a) Sélectionner la racine du fichier **Ecore** (dans notre exemple My.ecore) puis cliquer avec bouton droit et choisir **new child** → **Eclass**
   (b) De la même façon on peut créer autant de classe selon les besoins.
   (c) Chaque classe est caractérisée par un ensemble d'attributs, on peut ajouter un attribut par la sélection de la classe → cliquer bouton droit → **new child** → **EAttribute** . Lorsqu'on ajoute un attribut, un ensemble de propriétés lui a été lui associé, aller sur la propriété **Name** et donner un nom à ce dernier ainsi que son type.

(d) Pour ajouter une référence entre les classes, commencer par le choix de la classe source → bouton droit → **new child** → **EReference**. Puis donner un nom à la référence, un type (le nom de la classe cible), la propriété *Iscontainement* reçoit la valeur *True*. Préciser les cardinalités (exemple de 1 à *). La Figure 6.5 résume ces étapes.

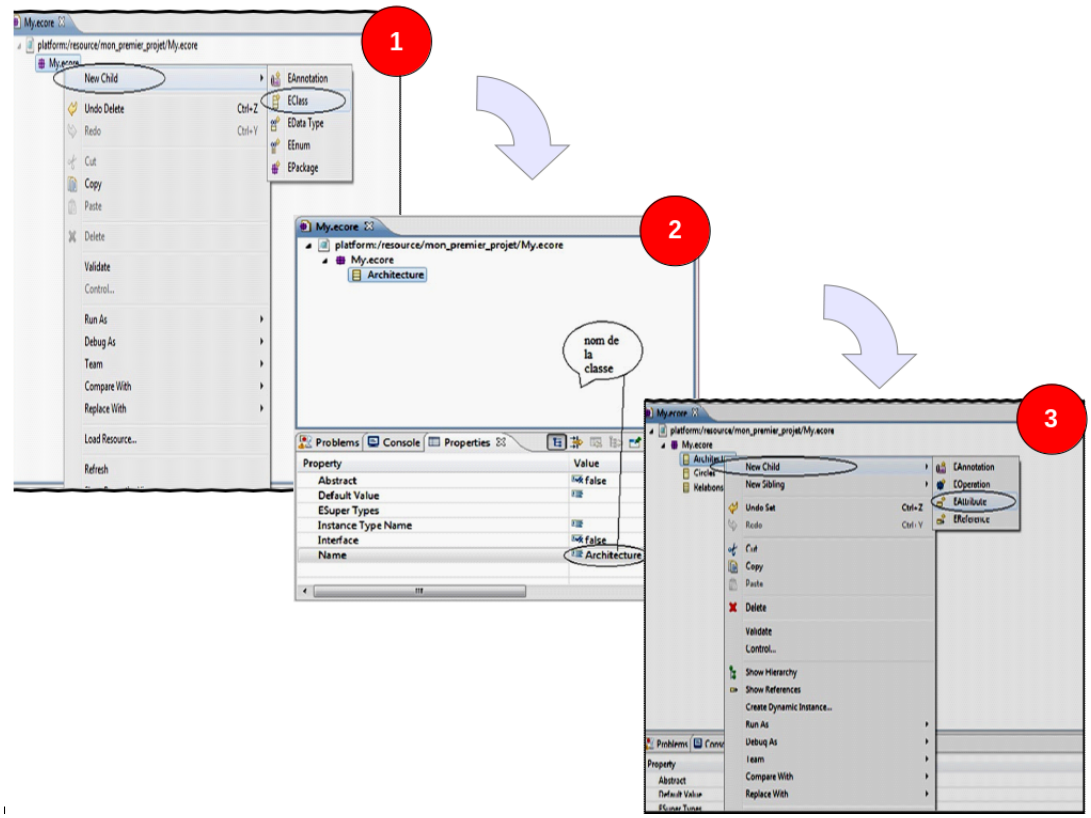


FIGURE 6.5: Création de classe, Attributs et références dans Ecore

6.1.2.4 ATL : ATLAS Transformation Language

ATL est un langage spécifié pour assurer la transformation de modèles dans le cadre de l'approche IDM. Il est inclus dans un plug-in à l'environnement Eclipse en fournissant ainsi un éditeur coloré et un compilateur de règles.

Ce langage est officiellement parti d'une tentative d'implémentation du QVT (OMG, 2008) «Request For Proposal» de l'OMG, en reprenant en particulier les diverses façons déclaratives

et impératives de spécifier une transformation. Il est développé sur la plateforme Eclipse et plus particulièrement sur sa branche EMF.

ATL est inspiré par les exigences OMG QVT (O.M.G, 2002) et s'appuie sur le formalisme OCL (G, 2003). Leur choix est motivé par sa large adoption dans l'approche MDE et le fait que c'est un langage standard pris en charge par l'OMG et les principaux fournisseurs d'outils de modélisation.

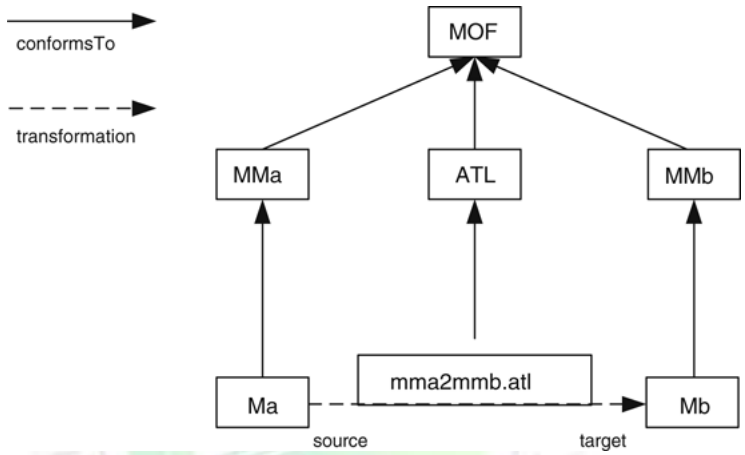


FIGURE 6.6: Aperçu sur le principe de transformation par ATL (Jouault *et al.*, 2008)

Comme l'indique la Figure 6.6, ATL est appliquée dans le cadre d'une transformation de type modèle vers modèle. Dans ce cas, un modèle source Ma est transformé en un modèle Mb cible selon une définition de transformation mma2mmb.atl écrit dans le langage ATL. La définition de transformation est un modèle conforme au méta-modèle de ATL. Tous les méta-modèles sont conformes à la MOF (Jouault *et al.*, 2008).

ATL est un langage de transformation hybride. En effet, il est possible de mélanger entre le style déclaratif et impératif. Généralement, le style déclaratif de transformation est le plus utilisé car il présente un certain nombre d'avantages. Il est généralement basé sur la spécification des relations entre les modèles sources et les modèles cibles et il a tendance à être plus proche de la façon dont les développeurs perçoivent intuitivement une transformation. Ce style met aussi l'accent sur le codage de ces relations et masque les détails relatifs à la sélection des éléments à la source. Cependant, il est parfois difficile de fournir une solution complètement déclarative pour un problème de transformation donné. Dans ce cas, les développeurs peuvent recourir au style impératif du langage ATL.

Les transformations ATL sont unidirectionnels c'est-à-dire que ATL fait uniquement des lectures sur les modèles sources et produit en écriture seule les modèles cibles. Lors de l'exécution d'une transformation, le modèle source peut être navigué mais les changements sur lui ne sont pas autorisés par contre le modèle cible ne peut pas être parcouru. Les modèles source et cible pour ATL peuvent être exprimés dans le format de sérialisation XMI/OMG. Pour les méta-modèles source et cible, ils peuvent être exprimées en Ecore ou bien aussi en XMI (Steinberg *et al.*, 2008) ou dans une annotation plus pratique comme KM3. (Jouault *et al.*, 2008).

6.1.2.5 Syntaxe d'une règle ATL

L'opération élémentaire dans une transformation ATL est nommée « ***règle ATL*** » . Un programme de transformation écrit en ATL est composé de règles qui spécifient comment les éléments du modèle source sont reconnus et parcourus pour créer et initialiser les éléments du modèle cible. Ces règles sont de la forme générale décrite par la figure suivante :

```
1 rule ForExample {
2 from
3 i : InputMetamodel !InputElement
4 to
5 o : OutputMetamodel !OutputElement (
6 attributeA <- i.attributeB,
7 attributeB <- i.attributeC + i.attributeD
8 )
9 }
```

- *ForExample* est le nom de la règle de transformation,
- *i (resp. o)* est le nom de la variable ; qui dans le corps de la règle va représenter l'élément source identifié (resp. l'élément cible créé),
- *InputMetaModel* (resp. OutputMetaModel) est le méta-modèle auquel le modèle source (resp. le modèle cible) de la transformation est conforme,
- *InputElement* désigne la métaclasse des éléments du modèle source auxquels cette règle va s'appliquer,
- *OutputElement* désigne la métaclasse à partir de laquelle la règle va instanciée les éléments cibles,

- *Le point d'exclamation* «!» permet de spécifier à quel méta-modèle appartient une méta-classe en cas d'homonymie,
- *attributeA et attributeB* sont des attributs de la métaclasse OutputElement. Leur valeur est initialisée à l'aide des valeurs des attributs *i.attributeB, i.attributeC* et *i.attributeD* de la méta-classe *InputElement*.

Voici comment on pourrait formuler la fonction de cette règle en langage naturel. La règle *"ForExample"*, pour chaque élément **"i"** de type *"InputElement"* qu'elle identifie dans le modèle source, crée dans un modèle cible un élément o de type *"OutputElement"*, et initialise les valeurs des attributs *"attributeA"* et *"attributeB"* de **"o"** avec les valeurs des attributs *"attributeB","attributeC" et "attributeD"* de **"i"**. Dans cet exemple simple, on voit l'utilisation du «.», issu de la spécification OCL et repris par ATL, qui permet dans les expressions *"i.attributeB"*, *"i.attributeC"* et *"i.attributeD"* de **'naviguer'** dans le modèle source **"i"**.

En plus des règles, le langage ATL dispose du mot clé « **helper**»[4] , qui permet de définir des macros à l'extérieur des règles pour factoriser des parties de code souvent utilisées. Un programme ATL, appelé un module, est essentiellement un groupement de règles et de ***helpers***.

En dehors du module lui-même, les éléments fixes de la traduction sont donc les deux méta-modèles source et cible. Le modèle source peut être vu comme le paramètre de la transformation, et le modèle cible son résultat.

C'est à travers la configuration de l'environnement d'exécution du programme que l'on spécifie concrètement dans quels fichiers le moteur de traduction doit chercher les méta-modèles, le modèle source, le fichier programme, et dans quel fichier on attend qu'il écrive le modèle résultat.

6.1.2.6 Exemple démonstratif

Nous avons vu dans la section précédente la méthode pour créer un méta-modèle Ecore. Dans cette section on va voir la méthode de création d'une transformation avec ATL.

Pour créer un ensemble de règles de transformation et dans le but d'organiser le projet et éviter les problèmes de dépendance il est conseillé de créer un projet ATL comme suit :

1. File new→ project → ATL Project → donner le nom du projet→ finich.

4. équivalent à la notion de fonction ou procédure dans le paradigme impératif

2. Créer trois répertoires dans le projet. le premier pour mettre les deux méta-modèles, le deuxième pour les modèles source et cible de la transformation. Le troisième est réservé au fichier de la transformation.

3. Passer maintenant à la création des méta-modèles, il faut créer deux méta-modèles : Le premier sera le méta-modèle source et le deuxième est le méta-modèle cible (cf. Section 2.1.2). Les Figures 6.7 et 6.8 présentent respectivement le méta-modèle source et cible nécessaires à la transformation :

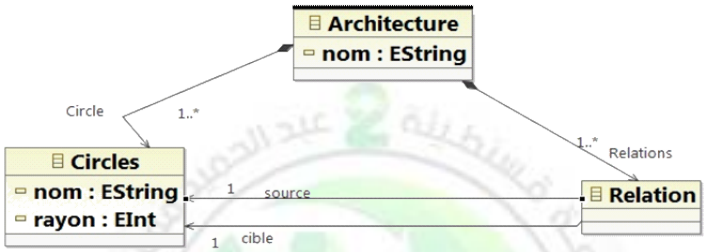


FIGURE 6.7: Exemple de Méta-modèle source nommé "Architecture"

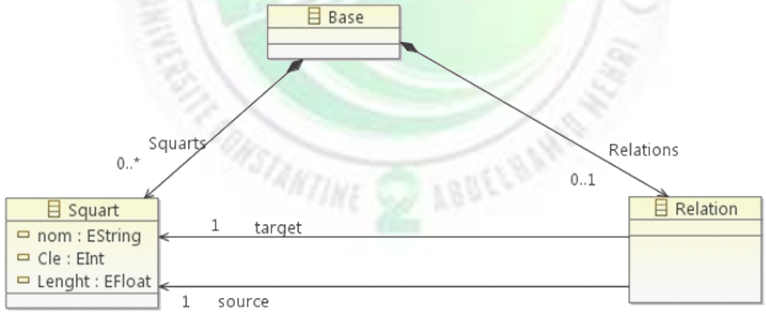


FIGURE 6.8: Exemple de Méta-modèle cible nommé "Base"

4. Créer un ficher ATL dans le dossier transformation. Donner un nom à ce denier ainsi que les chemins des méta-modèles source et cible (pour créer automatiquement l'entête). Définir l'ensemble des règles qui consistent à transformer une architecture vers une base. Dans cet exemple, on va traduire un cercle vers un carré (qui ont le même nom et la longueur du carré reçoit la valeur du rayon du cercle). En plus, on va transformer les relations entre cercles vers des relations entre carrés. Le Listing suivant représente le fichier ATL adéquate :

```
1  - - @ path MM==/CrilesToSquarts/metamodels/Circle_metamodel.ecore
2  - - @ path MM1==/CrilesToSquarts/metamodels/Squart_metamodel.ecore
3
4  module MyRules;
5  create OUT: Squart_metamodel from IN: Circle_metamodel;
6
7  rule ArchitBase{
8  from
9  Arch : Circle_metamodel!Architecture
10 to
11 Ba: Squart_metamodel!Base(
12 Relations<-Arch.Relations,
13 Squarts<-Arch.Circles
14 )
15 }
16
17 rule themaintransformation{
18 from
19 C: Circle_metamodel!Circle_metamodel
20 to
21 S: Squart_metamodel!Squart(
22 Clec<-C.Cle,
23 Color<-C.Color,
24 Lenght<-C.Rayon
25 )
26 }
27 rule Relation2Relation{
28 from
29 RC: Circle_metamodel!Relation
30 to
31 RS:Squart_metamodel!Relation(
32 target<-RC.target,
33 source<-RC.source
34 )
35 }
```

Listing 6.1: Exemple de règles de transformation en ATL

5. ***Exécution de la transformation*** : Nous avons vu précédemment la méthode pour créer un méta-modèle avec le framework EMF , on va maintenant voir comment créer un modèle à partir d'un méta-modèle. Pour créer un modèle (instance de méta-modèle) il faut suivre les étapes suivantes :
   — Aller sur le projet qui contient le méta-modèle a instancié (dans cet exemple *"Circle-ToSquart"*).
   — Ouvrir le méta-modèle se format Ecore (le méta-modèle source "Achitecture").

— Cliquer droit sur la racine du méta-modèle puis choisir l'option ***"Create Dynamic Instance"*** (dans cet exemple la racine est nommée Architecture). En suite, donner un nom à votre modèle, préciser et valider le chemin pour enregistrer ce modèle (enregistre le dans le dossier "Models").

Vous pouvez créer maintenant les classes de type Circle ou Relation ainsi que la possibilité de donner des valeurs aux attributs d'une classe (nom et rayon). Toutes ces étapes sont résumées dans la Figure 6.9 :

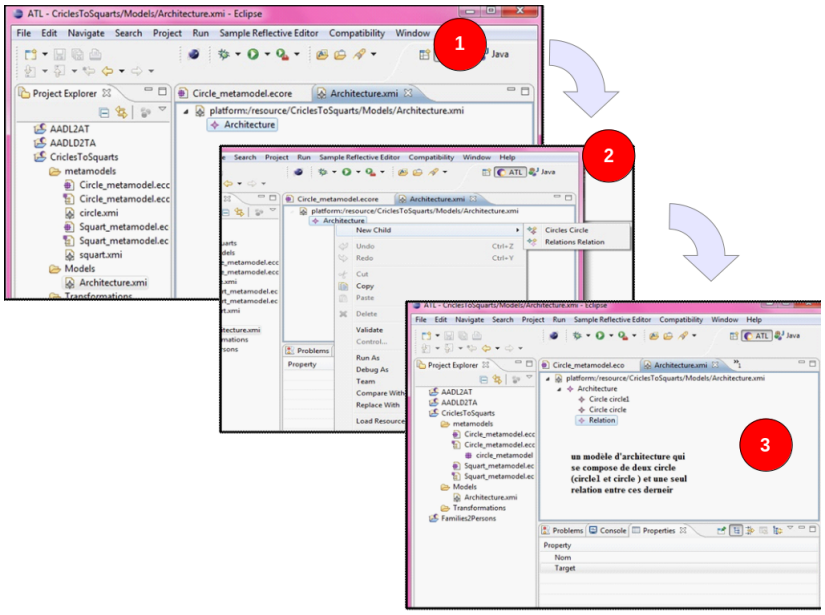


FIGURE 6.9: Instanciation de modèle à partir d'un méta-modèle

6. Pour exécuter la transformation, il faut ouvrir le fichier ATL qui contient les règles de transformation : aller dans la barre d'outil → run → une fenêtre suivante s'affiche qui permet de configurer la compilation du fichier ATL (cf. Figure 6.10) :

7. Après l'exécution des règles ATL, le compilateur ATL génère le modèle cible c'est-à-dire un fichier du nom « **Base.xmi** » dans le dossier "Models". Ce fichier contient un modèle de «**Base**» qui se compose de deux carrés qui ont les mêmes noms que le cercle et une relation entre ces derniers. La Figure suivante montre le résultat de cette transformation :

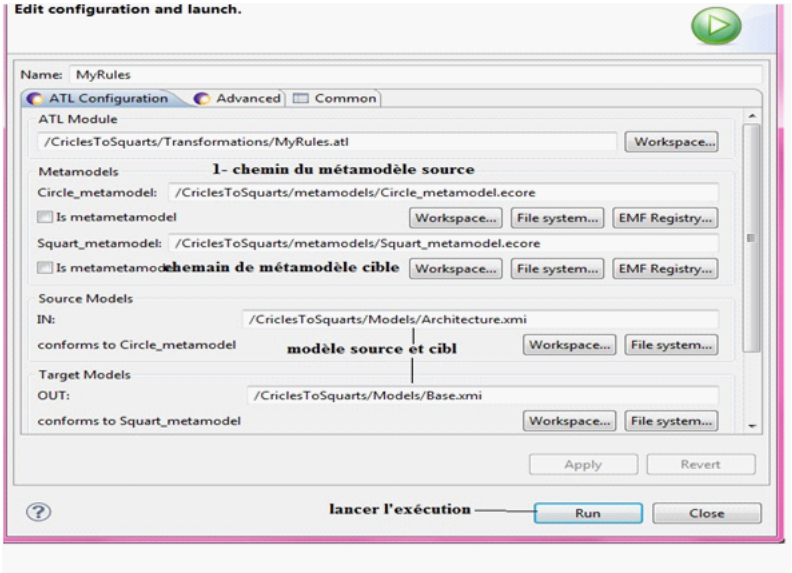


FIGURE 6.10: Configuration de Run pour ATL

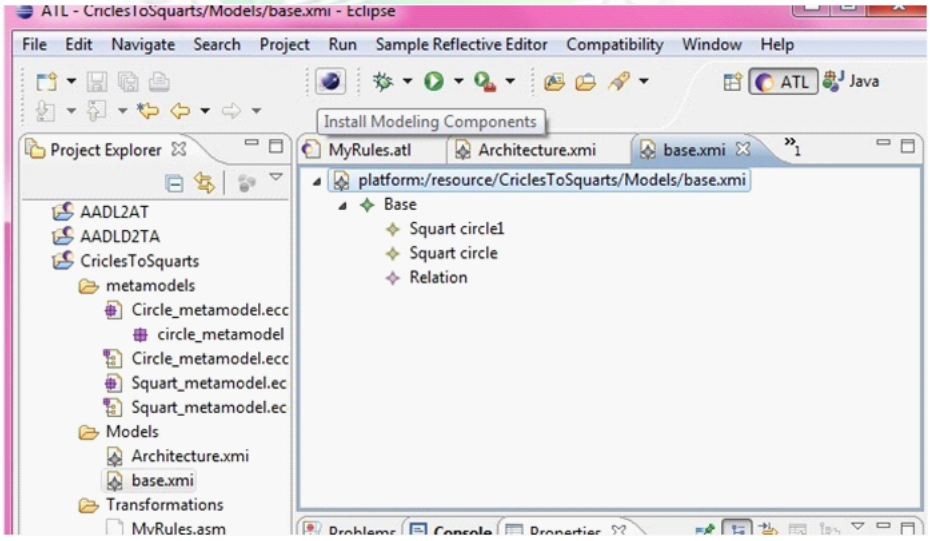


FIGURE 6.11: Résultat de l'application des règles ATL

6.1.2.7 Xpand : Langage de génération de code

Dans cette section nous allons présenter le langage des ***templates Xpand*** (Klatt, 2008) du projet **oAW**[5] intégré dans le framework de modélisation d'Eclipse EMF. Dans un projet de type **Xpand** nous avons besoin de définir quatre éléments (cf. Figure 6.12) :

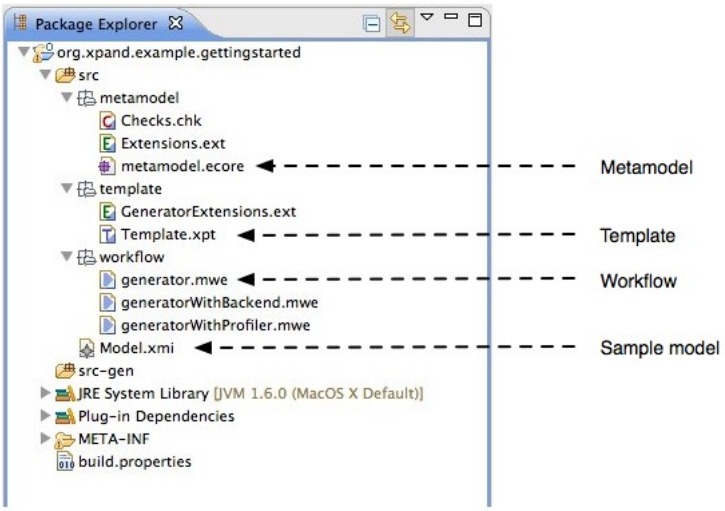


FIGURE 6.12: Éléments de la transformation par Xpand

— ***MetaModel*** : le Meta-Modele exprimé en Ecore

— ***Model*** : un modèle conforme au méta-modèle qui va subir la transformation

— ***Template*** : l'endroit pour définir les règles de transformations

— ***Workflow*** : fichier XML qui permet d'harmoniser le processus de transformation entre "Template" et le "Workflow".

La partie la plus importante est le *Template*, qu'on va décrire brièvement dans la section suivante.

6.1.2.8 Structure générale d'un template Xpand

La structure générale d'un *template Xpand* (Klatt, 2008) permet le contrôle de la génération du code correspondant à un modèle. Le modèle doit être conforme à un méta-modèle donné

5. http://www.openarchitectureware.org [consulté le 16/03/2018]

exprimé en Ecore. Le *template* est stocké dans un fichier ayant l'extension **" .xpt"**. Un fichier *template* se compose d'une ou de plusieurs instructions **"IMPORT"** afin d'importer les méta-modèles, de zéro ou plusieurs **"EXTENSION"** et d'un ou plusieurs blocs **"DEFINE"**.

(A) **Le bloc DEFINE**

Les blocs *"DEFINE"*, aussi appelés *templates*, constituent le concept central du langage Xpand. C'est la plus petite unité du fichier *template*. La balise blocs*"DEFINE"* se compose d'un nom, une liste optionnelle de paramètres et du nom de la méta-classe pour laquelle le template est défini. Les *templates* peuvent être polymorphes, ils ont le format suivant :

```
1 << DEFINE templateName(formalParameterList) FOR MetaClass >>
2 a sequence of statements
3 << ENDDEFINE >>
```

(B) **L'instruction FILE**

L'instruction *"FILE"* redirige la sortie produite, à partir des instructions de son corps, vers la cible spécifiée. La cible est un fichier dont le nom est spécifié par expression. Le format de l'instruction *"FILE"* se présente comme suit :

```
1 << FILE expression >>
2 a sequence of statements
3 << ENDFILE >>
```

(C)) **L'instruction EXPAND**

L'instruction *"EXPAND"* appelle un bloc *"DEFINE"* et insère sa production *"output"* à son emplacement. Il s'agit d'un concept similaire à un appel de sous-routine (méthode). Le format de l'instruction *"EXPAND"* se présente comme suit :

```
1 << EXPAND definitionName [(parameterList)]
2 [ FOR expression  FOREACH expression  [ SEPARATOR expression ] ]»
```

— *"definitionName"* est le nom du bloc *"DEFINE"* appelé. Si *"FOR"* ou *"FOREACH"* est omise l'autre **template** est appelé pour l'instance courante ***"this"***.

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

```
1  << DEFINE main FOR Standard >>
2  <<FILE "res/rdplifip.pnml" >>
3  <?xml version ="1.0"encoding = "ISO-8859-1"?>
4  <pnml xmlns="http://www.laas.fr/tina/TPN.rng">
5  <net id="rdp1" type="http://www.laas.fr/tina/TPN.rng">
6  <name> <text>rdplifip</text> </name>
7
8  <<FOREACH containsPlaces AS Places >>
9    <<EXPAND place FOR Place>>
10 <<ENDFOREACH>>
11 <<FOREACH containsTransitions AS Transitions >>
12   <<EXPAND transition FOR Transition>>
13 <<ENDFOREACH>>
14 <<FOREACH containsArcs AS Arc >>
15   <<EXPAND arc FOR Arc>>
16 <<ENDFOREACH>>
17 </net>
18 </pnml>
19 <<ENDFILE>>
20 <<File "res/rdplifip.net">>
21
22 net rdp1
23 <<ENDFILE>>
24 <<ENDDEFINE>>
```

Listing 6.2: Exemple démonstratif des règles en Xpand

Le listing 6.2 permet de générer à partir de réseau de Pétir un code en **PNML** (Petri Net Markup Langage) (Jungel *et al.*, 2000) . En effet, à partir d'un modèle de réseau de pétri en **XMI** on arrive à générer automatiquement un code en **PNML** qui sert comme entrée à l'outil de vérification **TINA** [6].

6.1.2.9 UPPAAL Model-checker : Un aperçu

UPPAAL (Larsen *et al.*, 1997) est un outil pour modéliser, simuler et vérifier des systèmes temps réel et spécifier en formalisme des automates temporisés. Depuis la dernière version, le

6. pour plus de précisions sur cet outil, le lecteur peut s'orienter vers (Berthomieu *et al.*, 2004).

client graphique a été programmé en **Java 2** pour le rendre exécutable sur plus grand nombre de plate-formes. Le moteur de vérification, lui, a été amélioré il est toujours disponible pour les plate-formes : Linux, SunOS et MS Windows.

UPPAAL comporte essentiellement trois modes, on peut accéder à ces modes par les onglets en haut au milieu de son interface graphique (cf. Figure 6.13) :

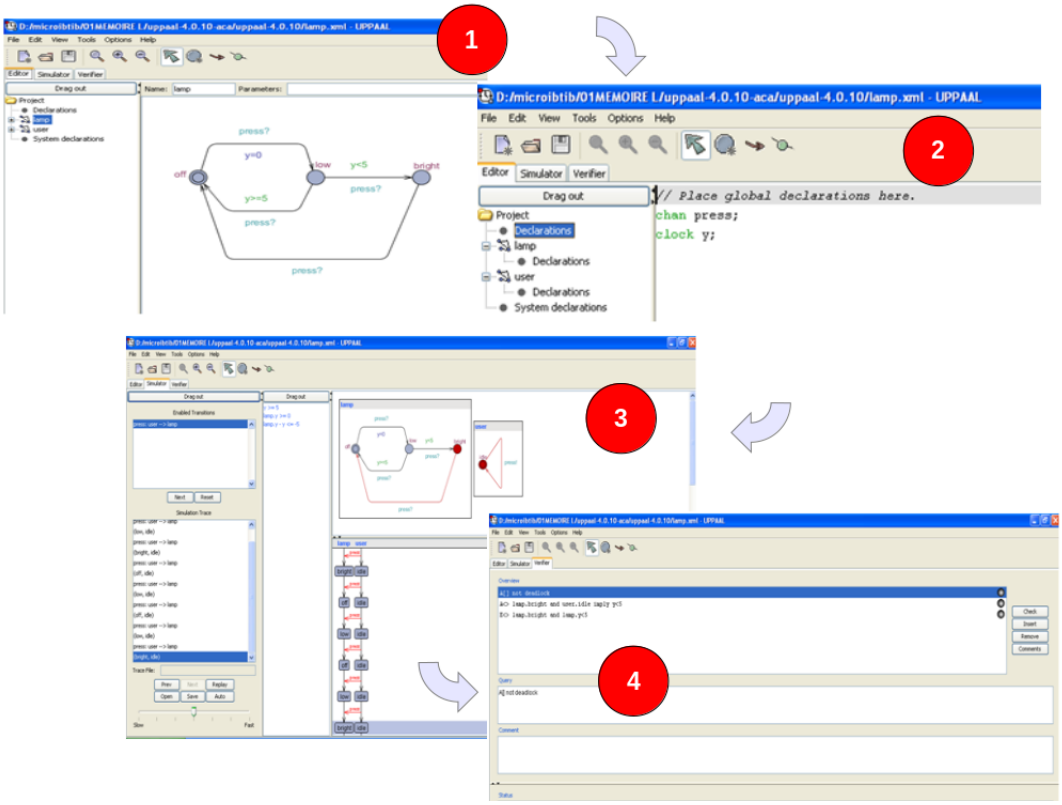


FIGURE 6.13: Modes d'utilisation de UPPAAL

1. **Édition** : ce mode par défaut permet de spécifier le système à analyser comme étant un produit synchronisé d'automates temporisés. On peut également spécifier des variables globales ou locales ; représentées dans la Figure 6.13 par l'étape 1 et 2.
2. **Simulation** : Lors de la construction d'un modèle, la simulation est une des approches les plus pratiques et les plus fructueuses pour se convaincre de sa validité. A ce titre, la procédure de simulation par UPPAAL est un des points forts de cet outil. La simulation

permet, à l'aide d'une interface graphique, de visualiser une configuration du système et les différentes transitions possibles à partir de cette configuration.

Dans une simulation pas-à-pas, on peut choisir une transition particulière, obtenir la nouvelle configuration et recommencer, ce qui permet de manipuler le réseau et d'observer son comportement. Les exécutions obtenues ainsi peuvent être sauvegardées, afin d'être rejouées ultérieurement.

UPPAAL permet également de simuler des exécutions aléatoires : à chaque étape où plusieurs transitions sont possibles, l'une d'entre elles est choisie au hasard. Il suffit dans ce cas de fixer la longueur souhaitée pour l'exécution et la vitesse de simulation pour observer le comportement du réseau (Bellaa, 2012). Il est représenté dans la Figure 6.13 par l'étape 3.

3. **Vérification** : Dans ce mode, on peut spécifier des propriétés d'accessibilité qui sont alors automatiquement vérifiées par l'outil. Si une propriété est fausse, l'outil génère un contre-exemple sous forme de trace à exécuter dans le simulateur (dans Options/diagnostic trace/shortest). Le model-checker permet aussi de vérifier des propriétés temporelles basées sur la logique temporelle CTL (***et plus précisément la logique TCTL***) ou des propriétés de vivacité, l'inter-blocage, ..etc.

Sous Uppaal et en mode vérification les propriétés à vérifier sont insérées dans le champ " ***query*** " puis on fait appel au model-checker à travers l'option **"check"** pour vérifier la satisfaction de propriété, si elle est satisfaite UPPAAL affiche le statut en vert ***"property is satisfied"*** sinon il affiche en rouge ***"property not satisfied"***. Représentée dans la Figure 6.13 par l'étape 4.

Généralement, l'étude d'un système spécifié par les automates temporisés est basée sur l'analyse de l'atteignabilité des états de l'automate. Par contre, UPPAAL permet d'élargir les propriétés vérifiables sur le modèle vers d'autres types de propriétés.

Exemple de propriétés :

— Propriété de l'absence de blocage

- Propriété sous la logique CTL : $AG \rceil$ (interblocage)
- Propriété sous UPPAAL : $A[]$ not deadlock

— On peut aussi vérifier des propriétés qui prennent en considération l'écoulement de temps :

- Propriété sous la logique CTL
  $AG(braillant \wedge press) \Longrightarrow AF(\langle 5) \; braillant$
- Propriété sous UPPAAL :
  $A <> lamp.bright \; and \; user.idle \; imply \; y < 5$

6.2 Études expérimentales

6.2.1 Étude de cas 1 : Système de chauffage

6.2.1.1 Description du système

Description[7] : Le système de chauffage choisi comporte deux vannes **"V1"** et **"V2"**, un bac, un thermostat et deux capteurs (cf. Figure 6.14) de niveau : le capteur **"S1"** qui surveille le niveau maximal et le capteur **"S2"** qui surveille le niveau minimal. Le système commence par une phase de remplissage, en utilisant la vanne **"V1"**.

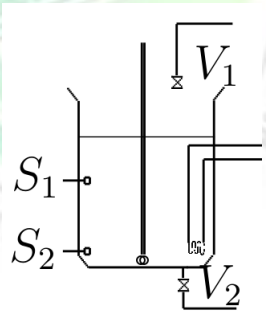


FIGURE 6.14: Système de chauffage

Dès que le niveau du liquide atteint le niveau maximal, après une intervalle de [40,50] u.t. (unité de temps), un capteur de niveau émet l'événement rempli, la vanne **"V1"** passe en position fermée et le système commence la phase de chauffage qui dure 60 u.t. Ensuite, le contrôleur commande l'évacuation du liquide présent dans un bac en ouvrant la vanne **"V2"**. La phase d'évacuation dure entre 20 et 25 u.t.. Elle s'achève lorsque le capteur de niveau **"S2"** détecte que le bac est vide et alerte le contrôleur par l'émission de l'événement vide.

7. étude de cas adaptée de (Derbel, 2009)

A l'interception de cet événement, le contrôleur ferme la vanne **"V2"**. Un nouveau cycle du système commence.

6.2.1.2 méta-modélisation

La description de ce système dans le langage AADL est donnée dans **Appendix B**. Deuxièmement nous avons développé le modèle système de chauffage à partir du méta-modèle proposé (cf. section 4.1, chapitre 5). Ce modèle est le modèle source dans l'approche proposée. Il est représenté dans la Figure 6.15 :

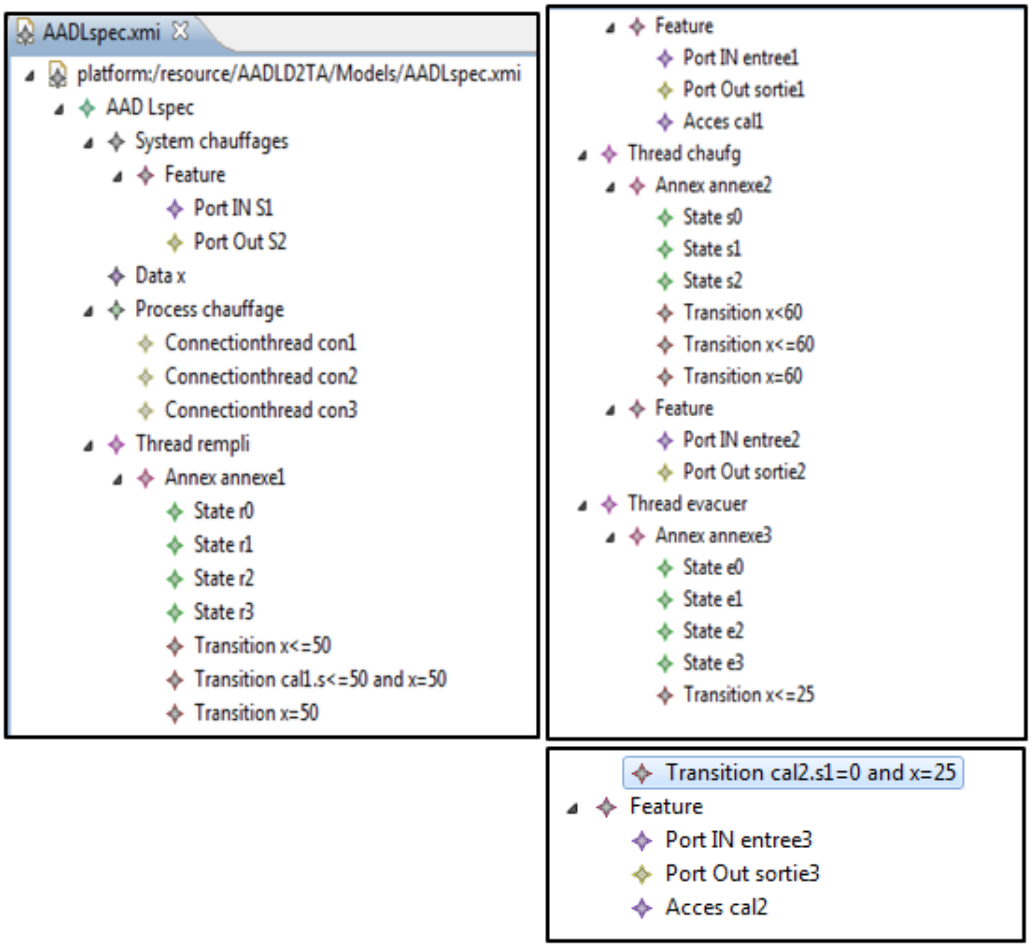


FIGURE 6.15: Modèle de système de chauffage instancié

6.2.1.3 Illustration de la transformation

Dans cette partie on va implémenter les règles de transformation proposées dans le chapitre précédent par le standard **ATL** pour la première étape de l'approche et par **Xpand** pour la deuxième transformation spécifiée dans l'approche :

(A) Les règles de la première transformation (Maadl2Mta) sont exprimées en ATL

```
1
2  @pathMM = /AADL2TA/meta-models/AADLdesr.ecore
3  @path MM1 = /AADL2TA/meta-models/TA.ecore
4  module AADL2TA;
5  create OUT: TA from IN : AADLdesr;
6  rule Process2TA{
7  from
8  M3:AADLdesr!process
9  to
10 M4:TA!timed_automata(
11 nom < -M3.nom,
12 states < -M3.subccompenets,
13 clocks < -M3.subcd,
14 transitions<-M3.connectiont)
15 }
16
17 rule data2clock{
18 from
19 M1:AADLdesr!data
20 to
21 M2:TA!clock(n
22 om < - M1.nom,
23 value <- M1.value)
24 }
25
26 rule thred2State{
27 from
28 M5:AADLdesr!thread
29 to
30 M6:TA!state(nom <- M5.nom,
31 Invariant <-M5.compute_execution_time)
32 }
33 helper context AADLdesr!transition def: IsFinal(): Boolean=
34 if self.etat = 'final' then
35   true
36 else
37   false
38 endif;
39
40 rule transition2trnsition{
41 from
```

```
42  M7:AADLdesr!transition(M7.IsFinal())
43  To M8:TA!transition(gard <- M7.gard,
44  Action <- M7.action,
45  reset <- M7.modifie)
46  }
47
48  rule Connection2Transition{
49  from
50  M9: AADLdesr!connectionthread
51  To M10:TA!transition(
52  source <- M9.sources,
53  target <- M9.target )
54  }
```

Listing 6.3: règle de transformation en ATL

Donc, on va appliquer l'ensemble des règles de la transformation pour produire l'automate temporisé correspondant sous forme de modèle dans **EMF**. Les figures suivantes montrent en détails l'application des règles de transformation pas à pas sur le modèle du système de chauffage décrit précédemment.

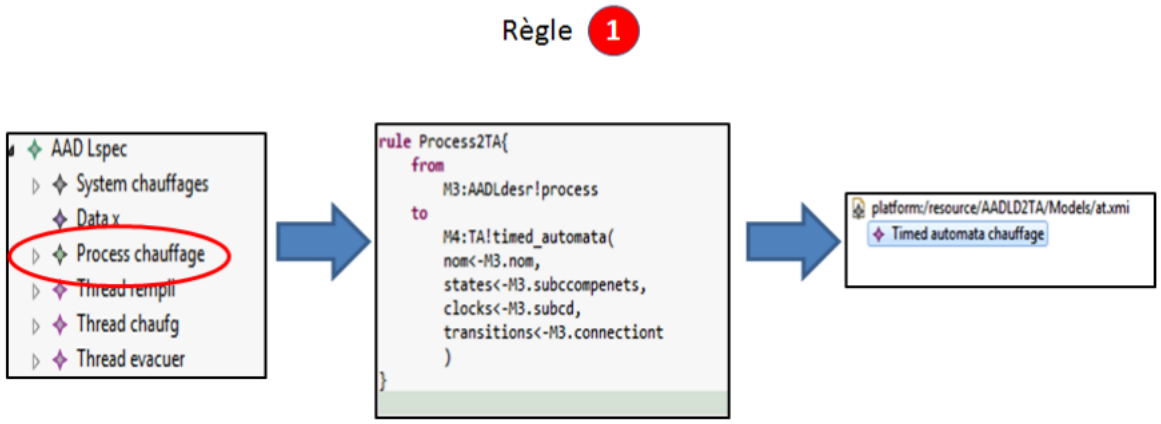


FIGURE 6.16: Démonstration de l'application de la règle 1

Le résultat de cette première transformation est exprimé par l'automate temporisé de la Figure 6.19 :

(B) Les règles de la deuxième transformation (Mta2Cta) exprimés par Xpand :

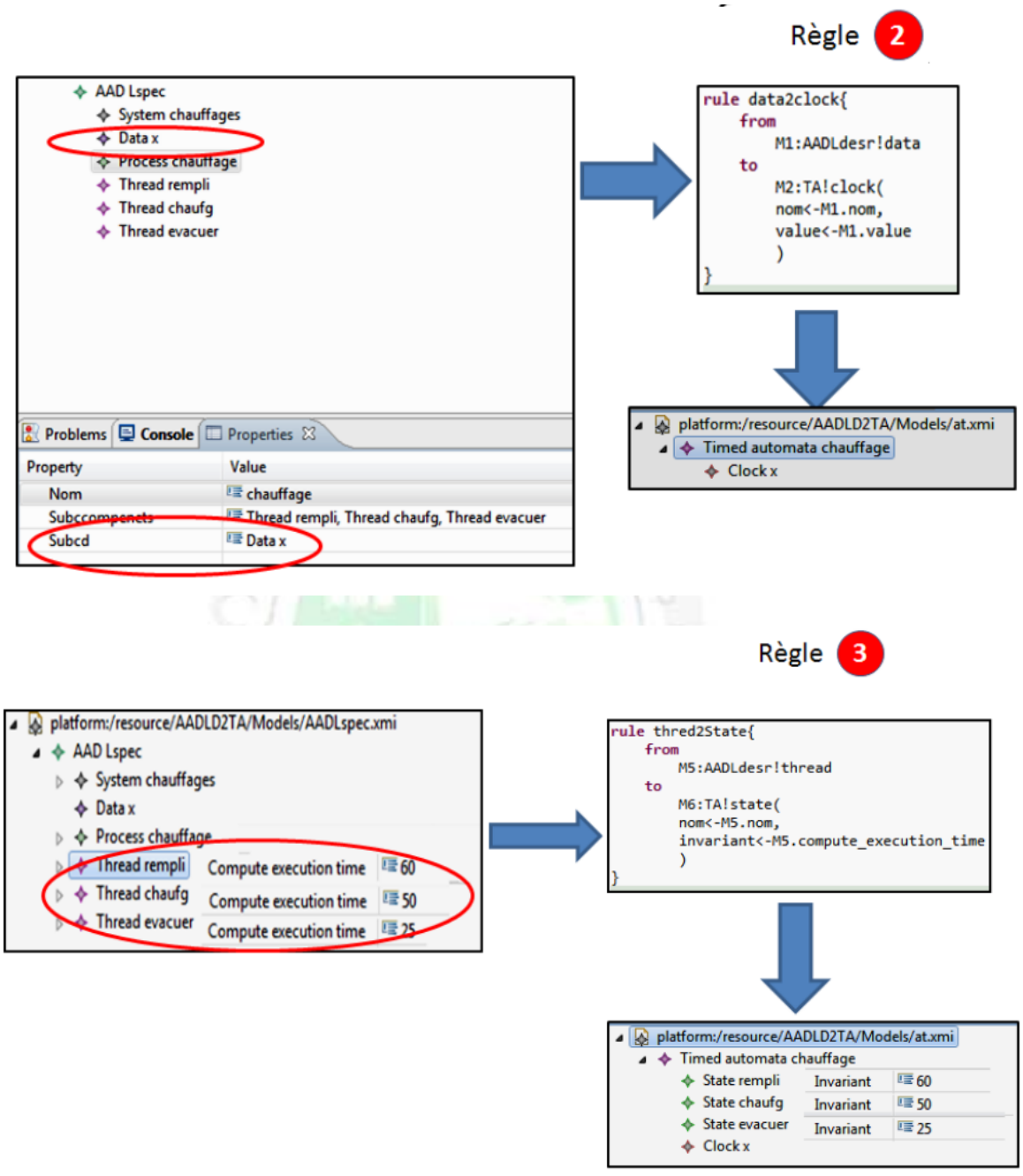


FIGURE 6.17: Démonstration de l'application de la règle 3

Règle 4

```
helper context AADLdesr!transition def : IsFinal():Boolean=
    if self.etat='final' then
        true
    else
        false
    endif;
rule transition2trnsition{
    from
    M7:AADLdesr!transition(M7.IsFinal())
    to
    M8:TA!transition(
    gard<-M7.gard,
    action<-M7.action,
    reset<-M7.modifie
    )
}
```

Règle 5

```
rule Connection2Transition{
    from
        M9:AADLdesr!connectionthread
    to
        M10:TA!transition(
        source<-M9.sources,
        target<-M9.target
        )
}
```

FIGURE 6.18: Démonstration de l'application des règles 4 et 5

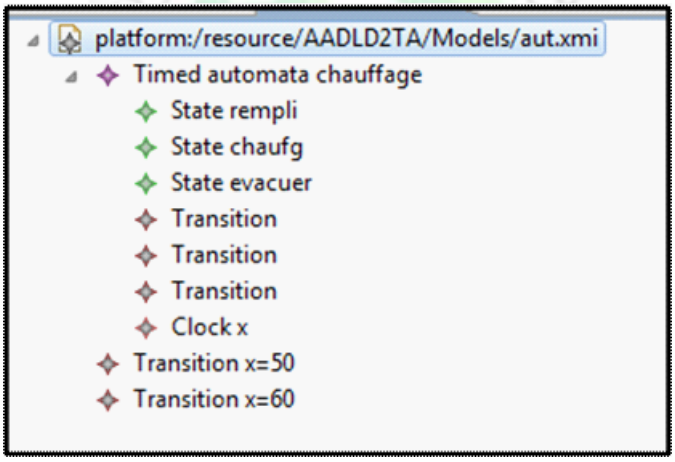


FIGURE 6.19: Résultat de la première phase de transformation

Dans cette étape on va exposer le modèle de la Figure 6.19 à une autre transformation qui permet de produire une description textuelle acceptée par **"UPPAAL"**. Les règles de la deuxième phase en **"Xpand"** sont données par le listing suivant :

```
1  "IMPORT␣meta-model"
2  "DEFINE␣main␣FOR␣Automate"
3  "FILE␣this.name+".ta"
4  clock␣" EXPAND clock2text FOREACH clocks
5  SEPARATOR ',' ";
6  chan␣" EXPAND chan2text FOREACH transitions
```

```
SEPARATOR ',' ";
process␣<<this.name>>␣{
state␣<<EXPAND␣state2text␣FOREACH␣states␣SEPARATOR␣','>>;
<<EXPAND␣intialState2text␣FOREACH␣states>>
trans␣<<EXPAND␣transition2text␣FOREACH␣transitions
SEPARATOR␣','>>␣;}
process␣user␣{
state␣idl;
init␣idl;
trans␣idl->idl␣{sync␣next?;␣};
}
system␣<<this.name>>,␣user;
<<ENDFILE>>
<<ENDDEFINE>>
<<DEFINE␣clock␣2␣text␣FOR␣Clock>>
<<this.name>>
<<ENDDEFINE>>

<<DEFINE␣intial␣State␣2␣text␣FOR␣State>>
<<IF␣this.isInitial==true>>
init␣<<this.name>>;
<<ENDIF>>
<<ENDDEFINE>>
<<DEFINE␣chan␣2␣text␣FOR␣Transition␣>>
<<this.action>>
<<ENDDEFINE>>
<<DEFINE␣state2text␣FOR␣State>>
<<IF␣this.Invariant␣==null>>␣<<this.name>>
<<ELSE>>
<<this.name>>␣{<<this.Invariant>>}
<<ENDIF>>
<<ENDDEFINE>>
<<DEFINE␣transition2text␣FOR␣Transition>>
<<this.source.name>>␣-␣>␣<<this.target.name>>␣{
<<IF␣this.guard␣!␣=␣'null'>>
guard␣<<this.guard>>;
<<ELSE>>
<<ENDIF>>
<<IF␣this.action␣!␣=␣'null'>>
sync␣<<this.action>>!;
<<ELSE>>
<<ENDIF>>
assign␣X:␣=␣0;
}
<<ENDDEFINE>>
```

Listing 6.4: règle de transformation en XPAND

Après l'exécution de ces règles sur le modèle de la Figure 6.19, le résultat de la deuxième phase de transformation est illustré par le Listing 6.5 :

```
// Global Declaration
clock X;
chan next;
//Process Description
process sysChauffage{
state remplissage {X<=50}, chauffage{X<=60}, evacuation{X<=25};
init remplissage;
trans
remplissage -> chauffage{
guard X>=40 and X<=50;
sync next!;
assign X:=0;
},
chauffage -> evacuation{
guard X==60;
sync next!;
assign X:=0;
},
evacuation -> remplissage{
guard X>=20 and X<=25;
sync next!;
assign X:=0;
};

}
process user {
state idl;
init idl;
tran idl->idl{sync next?;};
}
// System Configuration
system sysChauffage,user;
```

Listing 6.5: Résultat de la deuxième phase de transformation

Ce code fait l'entrée de l'outil UPPAAL, la Figure 6.21 montre l'interprétation de ce code par UPPAAL :

6.2.1.4 Vérification avec UPPAAL

Maintenant, le modèle équivalent du modèle source (modèle AADL) est représenté par le modèle d'automates temporisés dans le vérificateur UPPAAL. Le modèle checker UPPAAL

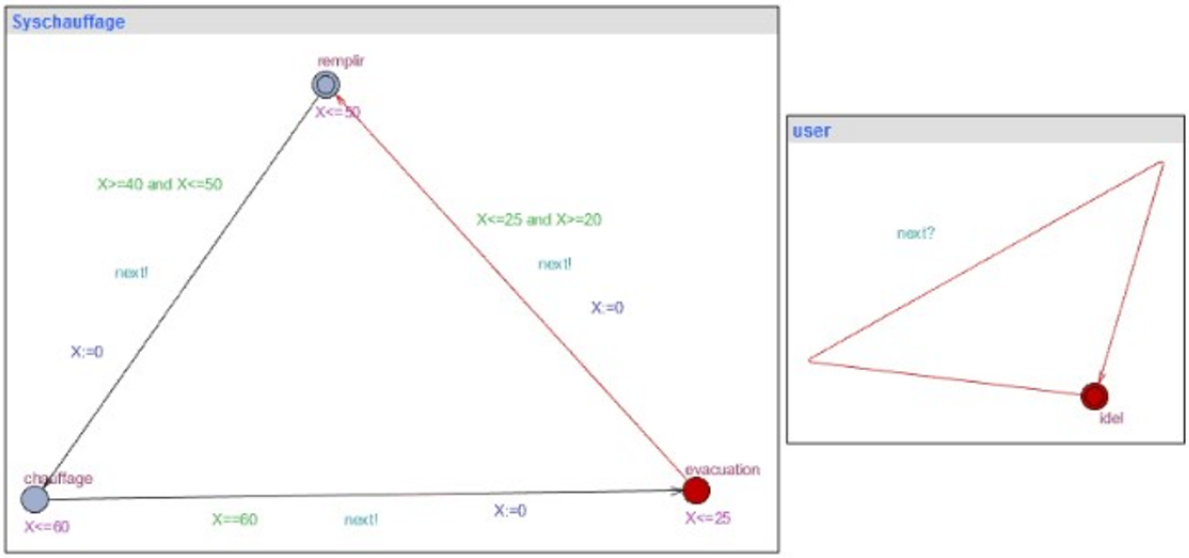


FIGURE 6.20: Interprétation du code généré par UPPAAL

exige que la requête (la propriété à vérifier) soit exprimée dans la logique temporelle TCTL (Timed Computationnal Tree Logic) (Vergnaud, 2006).

Une série de requêtes (Hamdane *et al.*, 2013a) est détaillée ci-dessous pour exprimer la vérification de l'absence de blocage, l'atteignabilité et la sûreté :

(a) ***Vérification de la propriété Absence de blocage : deadlock property***

- Est ce que le système peut avoir une situation de blocage ?
  - **Formalisation en TCTL** : A[] not deadlock
  - **Résultat** : Property is satisfied

(b) ***Vérification de la propriété de l'atteignabilité : reachability property***

- Est ce que le système reviendra à l'état initial (termine le cycle) ?
  - **Formalisation en TCTL** : A<> Syschauffage.remplir
  - **Résultat** : Property is satisfied

- Est ce que le système passera obligatoirement par une phase d'évacuation ?
  - **Formalisation en TCTL** : A<> Syschauffage.evacuation

- **Résultat** : Property is satisfied

- Si le système terminera la phase de chauffage est ce qu'il peut atteindre la phase d'évacuation ?
  - **Formalisation en TCTL** : Syschauffage.chauffage– >Syschauffage.evacuation
  - **Résultat** : Property is satisfied

(c) ***Vérification de la propriété de sûreté : liveness property***

- Est ce que je peux avoir une situation où le système est dans la phase de chauffage mais l'horloge n'a pas dépassé 50 ut ?
  - **Formalisation en TCTL** : Syschauffage.chauffage – > X < 50
  - **Résultat** : Property is not satisfied.
  - **Commentaire** : l'horloge X dans ce cas doit être obligatoirement supérieure à 50 ut selon la description du système.

- Est ce que le système terminera la phase de remplissage dans 30 ut ?
  - **Formalisation en TCTL** : Syschauffage.remplir – > X <= 30
  - **Résultat** : Property is not satisfied,
  - **Commentaire** : puisque cette phase est terminée après l'écoulement de 50 ut.

- Est ce que le système terminera la phase de remplissage et X est supérieur à 50 ut ?
  - **Formalisation en TCTL** : Syschauffage.remplir – > X >= 50
  - **Résultat** : Property is satisfied

6.2.2 Étude de cas 2 : Régulateur de température d'un réacteur d'avion

6.2.2.1 Description du système

Description : [8] Nous décrivons ici le contrôleur de température d'un réacteur d'avion. La température du réacteur est captée périodiquement par un capteur. La tache du contrôleur

8. étude de cas adaptée de (Omar, 2009)

consiste à **réfrigérer** le réacteur lorsqu'il reçoit le signal **"r"** du capteur. L'opération de réfrigération est mise en œuvre au moyen de deux barres (**"b1"** et **"b2"**) qui doivent être utilisées en ordre. Lorsque le contrôleur reçoit le signal **"r"** du capteur, commence l'opération de mouvement de la barre **"b1"** qui prend un certain temps "T1", ensuite il répète l'opération avec la barre **"b2"** qui prend aussi un certain temps "T2".

L'arrivée du signal **"r"** pendant le temps de réfrigération est considéré comme une erreur. De plus, le capteur n'est pas fiable, il peut tomber en panne et ne pas envoyer de signal **"r"** au contrôleur. Dans ce cas, et pour des raisons de sécurité, si le temps écoulé depuis le dernier signal **"r"** reçus est supérieur à **"t_max "**, le contrôleur commence une nouvelle opération de réfrigération (cf. Figure 6.21).

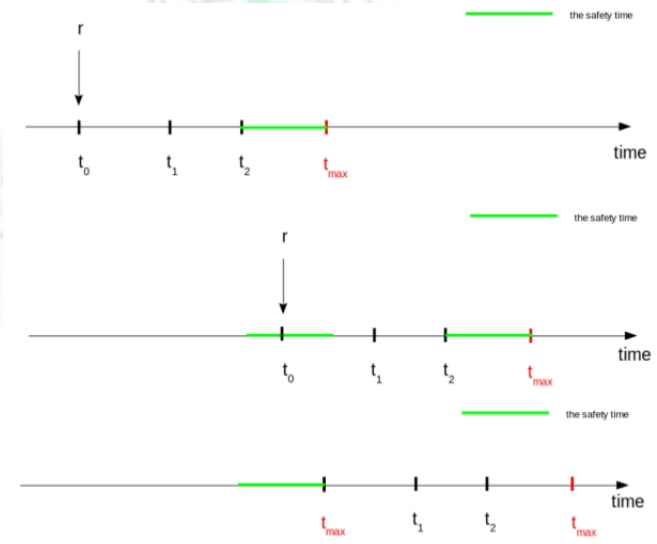


FIGURE 6.21: Fonctionnement de régulateur de température

6.2.2.2 méta-modélisation

La Figure 6.22 présente le modèle du système de régulateur de température en Ecore.

6.2.2.3 Illustration de la transformation

A ce niveau de l'approche, nous procédons à l'application des règles de transformation sur le modèle source du système de régulateur présenté dans la Figure 6.22. Le résultat de la

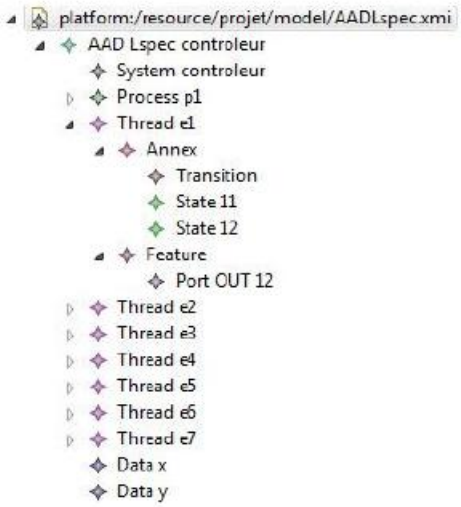


FIGURE 6.22: Capture d'écran sur le modèle source

transformation est présenté dans la Figure 6.23.

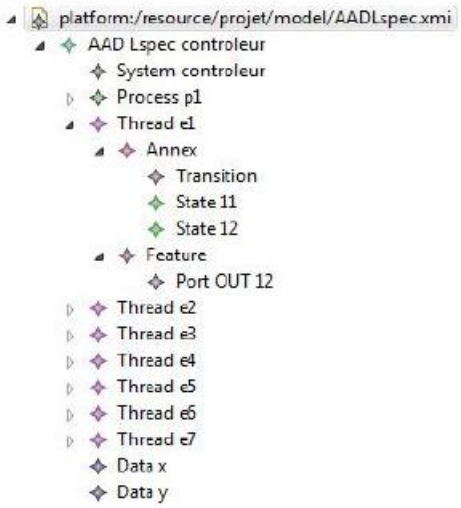


FIGURE 6.23: Capture d'écran sur le modèle généré

Maintenant, pour générer le code **"ta-format"** équivalent à la Figure 6.23 nous procédons

à exécuter le script **XPAND** (c.f Listing C.1) définit dans l'approche proposée. Le résultat est illustré par le Listing 6.6 :

```
\\global declaration
clock x,y;
chan r,b1,b2,exit;
int t1=50;
int t2=30;
int tmax = 120;
 \\processes description
process ctr{state wait,
  te1,te2{y<=t1},te3{y<=t2},te4{x<=tmax};
 init wait;
 trans wait->te1{sync r?;assign x:=0;},
 te1->te2 {sync b1!;assign x:=0,y:=0;},
 te2->te3 {sync b2!;assign y:=0;},
 te3->te4 {sync exit!;},
 te4->te2 {guard x==tmax;sync b1?;assign x:=0,y:=0;},
 te4->te1 {guard x<tmax;sync r?;assign x:=0;};}
process sensor{state start, bar1, bar2, end;
 init start ;
 trans start -> bar1{sync r!;},
 start -> bar2{sync b1!;},
 bar1 -> bar2{sync b1?;},
 bar2 -> end{sync b2?;},
 end -> start{sync exit?;};}
 \\system configuration
system ctr,sensor;
```

Listing 6.6: Résultat de la deuxième phase de transformation sur le modèle de régulateur de température

Ce code fait l'entrée de l'outil UPPAAL, la Figure 6.24 montre l'interprétation de ce code par UPPAAL :

6.2.2.4 Vérification avec UPPAAL

Nous proposons maintenant une série de requêtes (Hamdane *et al.*, 2017) permettant de vérifier le modèle cible présenté dans la Figure 6.24 :

(a) ***Vérification de la propriété Absence de blocage : deadlock property***

- Quelle que soit la situation, le système ne tombe pas dans un état de blocage ?
  - **Formalisation en TCTL** : $A[]!deadlock$

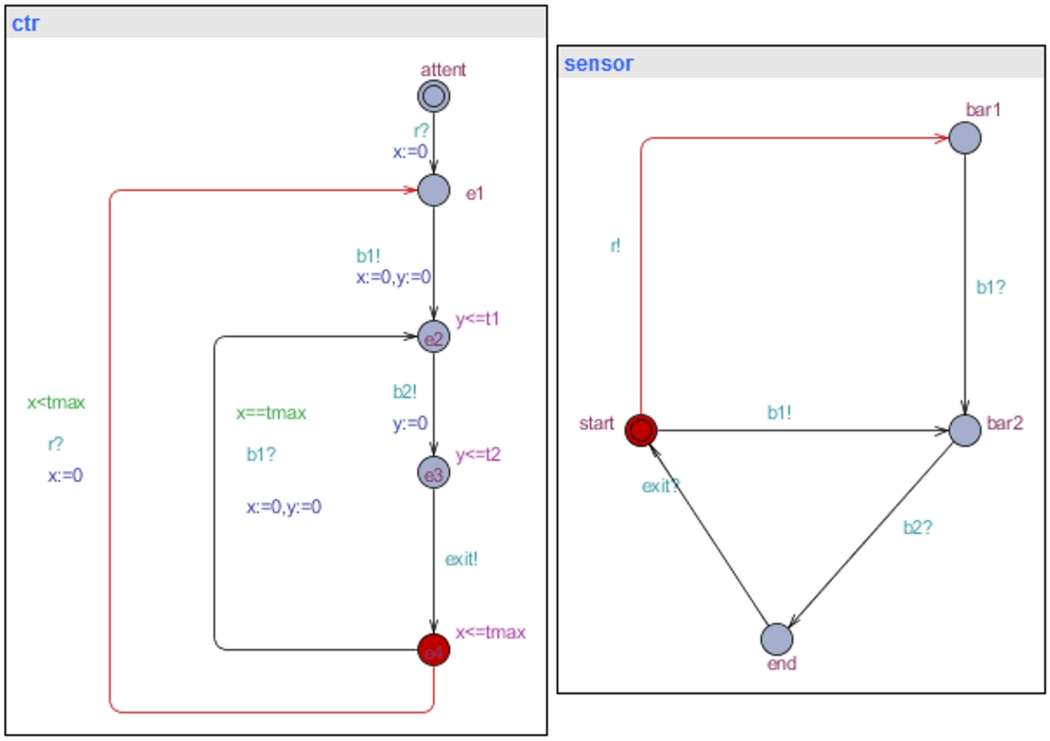


FIGURE 6.24: Interprétation du code généré par UPPAAL

- **Résultat** : property is satisfied !

(b) ***Vérification de la propriété de l'atteignabilité : reachability property***

- Est il possible que le système lance une opération de déplacement des deux barres "$b1$" et "$b2$" en même moment ?
  - **Formalisation en TCTL** : $A[]ctr.te2$ and $ctr.te3$
  - **Résultat** : property is not satisfied
- Lorsque le contrôleur termine le mouvement de la barre "$b1$", va-t-il lancer directement le mouvement de la barre "$b2$"
  - **Formalisation en TCTL** : $ctr.te2 \rightarrow ctr.te3$
  - **Résultat** : property is satisfied
- Dans le cas ou le capteur envoi le signal "x" et le système est dans un état d'attente, est ce que le contrôleur commence immédiatement l'opération de réfrigération ?

- **Formalisation en TCTL** : $sensor.bar1 \rightarrow ctr.te1$
- **Résultat** : property is satisfied

(c) ***Vérification de la propriété de sûreté : liveness property***

- Est ce que le système est toujours capable de terminer le mouvement de la barre "$b1$" dans un délai qui ne dépasse pas 50 u.t. ?
  - **Formalisation en TCTL** : $ctr.te2 \rightarrow y <= 50$
  - **Résultat** : property is satisfied
- Vérifier si la transition a lieu quand l'horloge "y" est compris entre 0 et t2 $\in [0..50]$?
  - **Formalisation en TCTL** : $A[]ctr.te3$ imply $(y >= 0$ and $y <= t1)$
  - **Résultat** : property is satisfied
- Si le système accomplit le mouvement de la barre "$b1$" alors l'horloge est supérieure à 50 u.t. ?
  - **Formalisation en TCTL** : $A[]ctr.te3$ imply $(y >= t2)$
  - **Résultat** : property is not satisfied
- Si le système est en attente, vérifiez si le temps écoulé n'a pas dépassé tmax (tmax = 120 u.t.) depuis le dernier signal "r" ?
  - **Formalisation en TCTL** : $A[]ctr.te4$ imply $(x >= 0$ and $x <= tmax)$
  - **Résultat** : property is satisfied

Les résultats obtenus, montrent que la majorité des requêtes sont satisfaites ce qui signifie que la modélisation proposée du système de régulateur de température avait un haut niveau de cohérence confirment l'intégrité de l'approche proposée.

La première requête permet de s'assurer que le comportement du modèle ne génère pas une situation de blocage. Les requêtes 2 à 4 permettent de vérifier la propriété d'accessibilité. Le résultat explique que : *si un état n'est pas accessible dans les automates temporisés, il ne l'est certainement pas dans le modèle initial de l'AADL*. Cela signifie que l'interaction entre les composants du modèle AADL est garantie.

Par ailleurs, les requêtes de 5 à 8 ont été utilisées pour vérifier la propriété de vivacité. Les résultats de ces requêtes confirment que les contraintes de temps dans ce système sont

respectées, à savoir, la durée du mouvement de la barre 1, la barre 2 et le temps de sécurité en cas de défaillance du capteur.

Enfin, nous devons signaler que les requêtes 3 et 7 ne sont pas satisfaites parce qu'elles sont en contradiction avec la description initiale du système en question. Par contre, leurs négation sont logiquement satisfaites.

Conclusion

Nous avons choisi deux études de cas *"Système de Chauffage" et "Régulateur de température"* pour tester et prouver le bon fonctionnement de notre approche de transformation.

Nous avons utilisé le standard Ecore comme langage de méta-modélisation au sein du framework EMF pour réaliser le méta-modèle de AADL et celui des automates temporisés. À partir du méta-modèle AADL, nous avons instancié un modèle AADL pour décrire deux modèles : le *"Système de Chauffage" et "Régulateur de Température "*. Par la suite, nous nous sommes servi du langage ATL pour assurer la traduction de ces deux modèles vers le modèle des automates temporisés. Par la suite, pour rendre le modèle des automates temporisés exploitable directement par le model-checker UPPAAL, nous avons généré un code équivalant en *ta-format.*

Nous avons effectué des analyses de vérification sur le modèle généré des automates temporisés en se servant de l'outil UPPAAL. Cette vérification est par rapport aux propriétés : d'atteignabilité, l'absence de blocage, la sûreté et la vivacité.

Ainsi, nous avons accompli les objectifs de nos travaux de thèse qui consistent à proposer une approche basée sur l'ingénierie dirigée par les modèles pour la vérification des descriptions AADL.

Conclusion Générale

CONCLUSION GÉNÉRALE

Nous avons présenté dans cette thèse une approche pour la transformation & la vérification des modèles AADL en se basant sur une combinaison entre les principes de l'ingénierie dirigée par les modèles et les principes de la vérification par model-cheking. Notre objectif est de contribuer à une meilleure fiabilité des modèles AADL, en renforçant la qualité de leurs spécifications comportementales à travers une vérification *a priori*. L'avantage de cette proposition est quelle offre plus d'automatisme dans le processus de transformation.

Notre effort était de réduire le niveau d'abstraction entre les modèles AADL et la technique de vérification par model-checking. Pour cela, nous sommes concentrés sur les éléments logiciels du langage AADL et plus particulièrement sur les composants *thread* car nous les estimons primordiale pour tout système AADL. Nous avons mis en place une démarche articulée autour de trois phases :

— *La métamodelisation* : Notre première contribution porte sur la définition d'un méta-modèle pour un sous ensemble du langage AADL (considéré comme méta-modèle source pour les éléments logiciel du langage) et un méta-modèle intermédiaire pour le formalisme les automates temporisés (considéré comme méta-modèle intermédiaire). Cette étape était assuré par le langage de métamodélisation EMF.

— *La transformation* : Dans cette phase nous avons proposé un processus de transformation qui prend en entrée un modèle AADL qui est une instance du méta-modèle source définie

dans la première étape et produit en sortie un fichier en code **"ta-format"**. Cette étape est assurée par la chaine d'outil **EMF-ATL-Xpand**.

— *La vérification* : Nous avons utilisé pour la vérification le model-checker UPPAAL qui est un outil stable pour la vérification des propriétés de sûreté ou de vivacité. Le fichier en "ta-format" généré par l'étape précédente est interprété directement par le model-checker UPPAAL. Il est également nécessaire dans cette étape de formuler dans la logique LTL les propriétés de sûreté et de vivacité qu'on cherche à vérifier sur le modèle des automates temporisés.

Pour évaluer notre approche, nous l'avons appliquée à deux études de cas : *"Système de chauffage"* et *"Régulateur de température"*. Le comportement de ce type de système est caractérisé par un ensemble de contraintes qui s'adaptent bien à notre contexte de travail.

Les perspectives de ce travail peuvent être déclinées de la manière suivante :

Une première perspective concerne l'implémentation. Nous nous sommes servis de plusieurs technologies IDM autour de la plateforme Eclispe comme EMF, ATL et Xpand. Par conséquent, il est intéressant de développer un prototype sous forme d'un plugin Eclipse regroupant l'étape de la métamodélisation et le processus de transformation.

Une deuxième perspective concerne le langage AADL. Notre approche prend principalement certains éléments de AADL comme : thread, process, port et feature. Il serait intéressant d'étendre cela afin de couvrir d'autres types d'éléments tout en préservant un niveau de complexité maîtrisable du méta-modèle AADL.

Une troisième perspective réside au niveau de méta-modèles AADL. Dans notre travail nous avons tenu en compte uniquement une seule donnée partagée entre les composants AADL. Il est envisageable de relâcher cette hypothèse et d'étudier son impact sur le processus de transformation.

La quatrième perspective entre dans le cadre de la transformation de modèle. Dans ce travail, nous considérons la transformation de modèles AADL vers les modèles d'automates temporisés. Pour le renforcer, nous pouvons considérer la transformation inverse (transformation bidirectionnelle (Lano et Tehrani, 2016)) c'est-à-dire à partir des automates temporisés

vers le modèle AADL. Ce type de transformation fait partie en effet des principes de l'IDM et peut s'avérer utile dans certains contextes.

Enfin, la dernière perspective peut être envisagée au niveau des prochaines versions d'AADL. On peut évoquer la possibilité d'ajouter une nouvelle annexe à AADL. Nous proposons comme nom *"OAC Annex" (Open AADL Connectivity)* dont le but est d'assurer une interface directe avec les outils de la vérification formelle.

BIBLIOGRAPHIE

Alur, R., Courcoubetis, C. and Dill, D.L. (1990). Model-checking for real-time systems. *In 5th Symposium on Logic in Computer Science(LICS'90)*, page 414–425.

Alur, R. and Dill, D.L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2): 183–236.

Amelunxen, C. et al (2008). Metamodel based tool integration with moflon. *In 30th International Conference on Software Engineering, New York, ACM Press. Research Demonstration*.

Amrani, M. et al (2015). Formal verication techniques for model transformations : A tridimensional classication. *Journal of Object Technology*, 14(3):13–43.

Andy Schürr, Andreas J. Winter, A.Z. (1999). The progres approach : language and environment. *World Scientific Publishing Co., Inc., River Edge, NJ, USA*, 5:487–550.

Atigui, F. (2013). *Approche dirigée par les modèles pour l'implantation et réduction d'entrepôts de donées*. Thèse de doctorat, Université Paul Sabatier de Toulouse.

Balasubramanian, D. et al (2006). The graph rewriting and transformation language : Great. *Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs)*, 1.

Bellaa, I. (2012). Modélisation et vérification des systèmes temps réel : Par l'outil de model checking uppaal. Rapport technique, Université de Khenchela.

Benammar, M. (2011). *Une Approche Basée Architecture pour la Spécification Formelle des Systèmes Embarqués*. Thèse de doctorat, Université Mentouri de Constantine.

Benammar, M. and Belala, F. (2010). How to make aadl specification more precise. *International Journal of Computer Applications*, 8(10):16–23.

BENGTSSON, J. et al (1996). Verification of an audio protocol with bus collision using uppaal. *Procceding of the 8th International Conference on Computer-Aided Verification*, 1102:244–256.

Berthomieu, B., Ribet, P.O. and Vernadat, F. (2004). The tool tina construction of abstract state spaces for petri nets and time petri nets. *Journal of Production Research*.

Beugnard, A. et al (2014). Des situations de modélisation pour évaluer les outils de modélisation. pages 181–196.

Booch, G., Rumbaugh, J. and Jacobson, I. (1994). Unified modeling language. Rapport technique, Rational Software Corporation.

BUDINSKY, F., STEINBERG, D. and ELLERSICK, R. (2003). *Eclipse Modeling Framework : A Developer's Guide*. Addison-Wesley Professional.

Burmester, S. et al (2004). Tool integration at the meta-model level within the fujaba tool suite. *International Journal on Software Tools for Technology Transfer (STTT)*, 6:203–218.

Bézivin, J. (1998). Who is afraid of ontologies ? *In OOPSLA'98 Workshops, Vancouver, Canada*.

Bézivin, J. (2003). La transformation de modèles. *INRIA-ATLAS & Université de Nantes,école d'été d'informatique CEA-EDF-INRIA*.

Bézivin, J. (july 2014). Software modeling and the future of engineering. *Keynote at 7th international conference on model transformation, York United Kingdom*.

Bézivin, J. and Gerbée, O. (2001). Towards a precise definition of the omg/mda frameword. *the 16th International Conference on Automated Software Engineering, ASE*, page 273–280.

Bézivin, J., Lennon, Y. and Nguyen, C. (1995). From cobol to omt : A reengineering workbench based on semantic networks. *In Tools USA'95, Santa Barbara, CA, USA*, pages 137–152.

Champeau, J. et al (2008). Aadl execution semantics transformation for formal verification. *Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'08)*, pages 263–268.

CHKOURI, M.Y. (2010). *Modélisation des systèmes temps-réel embarqués en utilisant AADL pour la génération automatique d'applications formellement vérifiées*. Thèse de doctorat, Université de Grenoble.

CLARK, T. et al (2004). Applied metamodelling – a foundation for language driven development. version 0.1.

Clarke, E., Henzinger, T. and Veith, H. (2018). *Handbook of Model Checking*. Springer International Publishing.

CLARKE, E.M. and EMERSON, E.A. (1981). Design and synthesis of synchronization skeletons using branching-time temporal logic. *In D. KOZEN, : Logic of Programs*, 131:52–71.

Combemale, B. (2008). Ingénierie dirigée par les modèles (idm) État de l'art. Rapport technique, Institut de Recherche en Informatique de Toulouse (UMR CNRS 5505).

Czarnecki, K. and Helsen, S. (2003). Classification of model transformation approaches. *Workshop on Generative Techniques in the Context of Model- Driven Architecture*.

David, A., Oliver-Moller, M. and Yi, W. (2002). Formal verification of uml statecharts with real-time extensions. *In* LNCS, éditeur : *In 5th International Conference Fundamental Approaches to Software Engineering*, volume 2306, page 218–232. Springer-Verlag.

Daws, C. et al (1996). The tool kronos. *In* LNCS, éditeur : *In Procceding of Workshop Hybrid Systems III : Verification and Control*, volume 1066, page 208–219. Springer-Verlag.

de Lara, J. and Vangheluwe, H. (2002). Atom3 : A tool for multi-formalism and metamodeling. *in FASE'02, Fundamental Approaches to Software Engineering, Grenoble, France*.

Derbel, H. (2009). *Diagnostic à base des systèmes temporisés et d'une sous-classe de systèmes hybrides*. Thèse de doctorat, Université Joseph Fourier - Grobonle 1.

DESEL, J. (2002). Model validation - a theoretical issue. *Application and Theory of Petri Nets*, 2505:393–410.

Diaw, S., Lbath, R. and Coulette, B. (2010). Etat de l'art sur le développement logiciel basé sur les transformations de modèles. *Technique et Science Informatiques, Ingénierie Dirigée par les Modèles*, 29(4-5):505–536.

Dissaux, P. (2004). Using the aadl for mission critical software development. *2nd European Congress ERTS, EMBEDDED REAL TIME SOFTWARE*, Rapport Technique HPL-2004-187.

Ehrig, K. et al (2005). Model transformation by graph transformation : A comparative study. *International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*.

Evensen, K. and Weiss, K. (2010). A comparison and evaluation of real-time software systems modeling languages. *AIAA Infotech Aerospace 2010*, page 3504.

Favre, J.M., Estublier, J. and Blay-Fornarino, M. (2006). L'ingénierie dirigée par les modèles : au-delà du mda. page 273–280.

Feiler, P.H., Bruce, L. and Vestal, S. (May 2003). The sae architecture analysis design language (aadl) standard : A basis for model-based architecture-driven embedded systems engineering. *In RTAS Workshop 2003*, 1:1–10.

G., O.M. (1997). Object managment groupe, meta objet facility (mof) specification. Rapport technique.

G, O.M. (2003). Object management group, inc. object constraint language (ocl), document ptc/03-10-14. Rapport technique.

Garavel, H. et al (2011). Cadp 2010 : A toolbox for the construction and analysis of distributed processes. *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2011)*.

Geensyde, A. (2002). Ades : un simulateur pour aadl. Rapport technique, Agence spatiale européene (ESA), http ://www.axlog.fr/aadl/ades_fr.html.

GERBER, A. et al (2002). Transformation : The missing link of mda. *In Proceedings of the First International Conference on Graph Transformation (ICGT)*, 2505:90–105.

Gracy, P. and Meenakshi, D. (2018). *Model-Based Safety Validation for Embedded Real-Time Systems*, pages 59–71. Springer Singapore.

Hamdane, M.E., Berramla, K. and Chaoui, A. (2017). Using mda with model checking to ensure the development of consistent aadl models. *In Procceding of the 1st IEEE international conference on Embedded Distributed Systems, EDiS'2017, Oran, Algeria*, pages 135–140 available at : doi=DOI:10.13140/RG.2.2.18910.33605.

Hamdane, M.E. and Chaoui, A. (2011). Specification and verification of timed automaton using meta-modeling and graph grammars. *Proceeding of the 4th IEEE International Conference on the Applications of Digital Information and Web Technologies (ICADIWT'11)*, pages 137–142, available at : http://ieeexplore.ieee.org/document/6041415/.

Hamdane, M.E., Chaoui, A. and Strecker, M. (2013a). From aadl to timed automaton - a verification approach. *International Journal of Software Engineering and Its Applications IJSEIA*, Volume 7 (4):115–126, available at : https://www.scopus.com/authid/detail.uri?authorid=53877527700.

Hamdane, M.E., Chaoui, A. and Strecker, M. (2013b). Toolchain based on mde for the transformation of aadl models to timed automata models. *Journal of Software Engineering and Applications*, 6 (3):147–155, available at : doi=10.4236/jsea.2013.63019.

Henzinger, T.A., Ho, P.H. and Wong-Toi, H. (1995). A user guide to hytech. *In* LNCS, éditeur : *In Procceding of the 1st International Workshop Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019, page 41–71. Springer-Verlag.

Hillah, M.L.M. (2009). *intégration des méthodes formelles au développement dirigé par les modèles pour la conception et la vérification des systèmes et applications répartis*. Thèse de doctorat, Université Pierre et MARIE Curie.

Hladik, P.E., Peres, F. and Shi., X. (2010). Analyse d'un modèle aadl à l'aide de pola. *Proceeding of Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2010)*, pages 239–243.

Hutzler, G. and Klaudel, H. (2004). Automates temporisés et systémes multi-agents temps-réel. *Hermés Science*, 1:69–82.

Idiri, R. (2009). Test d'application aadl. Mémoire de D.E.A., Université de Bretagne Occidentale.

JET (2004). Jet tutorial part 1, http ://www.eclipse.org/articles/article-jet/jet_tutorial1.html.

Jouault, F. et al (2008). Atl : A model transformation tool. *Science of Computer Programming*, 72(1–2):31–39.

JOUAULT, F. and Bézivin, J. (2006). Km3 : a dsl for metamodel specification. *In Proceedings of the IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 4037:171–185.

JOUAULT, F. and KURTEV, I. (2005). Transforming models with atl. *In* LNCS, éditeur : *In Satellite Events at the MoDELS 2005 Conference*, volume 3844, page 128–138. Springer.

Jouault, F. and Kurtev, I. (2006). On the architectural alignment of atl and qvt. *Proceedings of the SAC'06 ACM Symposium on Applied Computing*, pages 1188–1195.

Jungel, M., Kindler, E. and Weber, M. (2000). The petri net markup language. *In Procceding of the 7th Workshop Algorithmen und Werkzeuge für PetriNetze (AWPN), Fachberichte Informatik Universität Koblenz -Landau*, page 47–52.

Kawtharany, M. (2007). Transcription de patterns d'utilisation métier en aadl. Rapport technique, Université de bretagne occidentale.

Kerkouche, E. (2011). *Modélisation multi-paradigme : une approche basée sur la transformation de graphes*. Thèse de doctorat, Université de Constantine.

Klatt, B. (2008). Xpand : A closer look at the model2text transformation language. *12th European Conference on Software Maintenance and Reengineering*.

Kleppe, A.G., Warmer, J. and Bast, W. (2003). Mda explained : The model driven architecture : Practice and promise.

Lano, K. and Tehrani, S.Y. (2016). Verified bidirectional transformations by construction. *In PAME/VOLT@ MODELS*, pages 28–37.

Larsen, K.G., Pettersson, P. and Yi, W. (1997). Uppaal in a nut shell. *International Journal on Software Tools for Technology Transfer*, 1:134–152.

LEDECZI, A. et al (2001). Generic modeling environment. *In Proceedings of the IEEE Workshop on Intelligent Signal Processing (WISP), Budapest, Hungary*.

LINDAHL, M., PETTERSSON, P. and YIW (1998). Formal design and analysis of a gear-box controller. *Procceding of the 4th Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1384:281–297.

Lämmel, R., Saraiva, J. and Visser., J. (2006). Generative and transformational techniques in software engineering. 4143:36–64.

Lucio, L. et al (2016). Model transformation intents and their properties. *Software & systems modeling*, 15 (3):647–684.

MAIER, M.W. (2005). Km3 : Kernel metametamodel. Rapport technique, LINA INRIA ATLAS, Nantes.

Mateescu, R. (2003). Logiques temporelles basées sur actions pour la vérification des systémes asynchrones. Rapport technique, Institut National De Recherche En Informatique Et En Automatique, france.

MENS, T. (18 janvier 2008). A taxonomy of model transformations. Rapport technique, GPL - Journée sur les transformations de modèles, LIP6, Paris, France.

MENS, T., CZARNECKI, K. and GORP, P.V. (2005). A taxonomy of model transformations. *In Dagstuhl Seminar Proceedings of the internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI)*, page 171–185.

Montanari, U. (1970). Separable graphs, planar graphs and web grammars. *Information and Control*, 16:243–267.

MULLER, P.A., FLEUREY, F. and JÉZÉQUEL, J.M. (2005). Weaving executability into object-oriented meta-languages. volume 3713, page 264–278. Springer.

Ocarina (2005). Ocarina : http ://aadl.enst.fr/ocarina/.

Oddoux, D. (2003). *Utilisation des automates alternants pour un model-checking : efficace des logiques temporelles linéaire*. Thèse de doctorat, Université Paris 7.

Omar, I.K. (2009). modélisation d'un contrôleur de température d'un réacteur d'avion par uppaal. Rapport technique.

O.M.G (2002). Omg/rfp/qvt mof 2.0 query/views/transformations rfp. Rapport technique, OMG document ad/2002-04-10. Available from : www.omg.org.

OMG (2006). Object management group, inc. meta object facility (mof) 2.0 core specification. final adopted specification.

OMG (2008). Object management group, inc. meta object facility (mof) 2.0 query/view/transformation (qvt) specification.

Paige, R., Zolotas, A. and Kolovos, D. (2017). *The Changing Face of Model-Driven Engineering*, pages 103–118. Springer International Publishing.

Pelfresne, R. (2003). Modéliser, pourquoi ? comment ? *Journée d'étude – ADBS*.

Pfaltz, J.L. and Rosenfeld, A. (1969). Web grammars. *In International Joint Confenence on Artificial Intelligence*, page 609–619.

Pi, L. et al (2009). A comparative study of fiacre and tasm to define aadl real time concepts. *In 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 347–352. IEEE.

Piel, E. (2007). *Ordonnancement de systèmes parallèles temps-réel De la modélisation à la mise en œuvre par l'ingénierie dirigée par les modèles*. Thèse de doctorat, Université des Sciences et Technologies de Lille.

Pnueli, A. (1977). The temporal logic of programs. *In Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77), IEEE Computer Society*, page 46–57.

Pratt, T.W. (1971). Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595.

Rahim, L.A. and Whittle, J. (2015). A survey of approaches for verifying model transformations. *Software System Model*, 14:1003–1028.

Rozenberg, G. and Engelfriet, J. (1998). Elementary net systems. LNCS 1491:12–121.

SCHMIDT, D. (2006). *Introduction : Model-Driven Engineering*, volume 39.

Schnoebelen, P. (1999). Vérification de logicièls : Techniques et outils du model-cheking. *Vuibert*.

Schnoebelen, P. et al (1999). Vérification de logiciels : techniques et outils du model-checking vuibert.

SEI-AADL-Team (June 2006). An extensible open source aadl tool environment (osate). Rapport technique, Software Engineering Institute, Carnegie Mellon University.

Singhoff, F. et al (2004). Cheddar : a flexible real time scheduling framework. *ACM SIGAda Ada Letters*, 124(4).

Singhoff, F. et al (2005). Scheduling and memory requirements analysis with aadl. *Proceedings of the annual ACM SIGAda international conference on Ada (SIGAda'05), ACM Press*, page 1–10.

Sokolsky, O., Lee, I. and Clark., D. (2006). chedulability analysis of aadl models. *International Parallel and Distributed Processing Symposium (IPDPS)*.

Steinberg, D. et al (2008). *EMF : Eclipse Modeling Framework -chapter Ecore Modeling Concepts, Second Edition*. Addison-Wesley Professional.

Taentzer, G. (2003). Agg : A graph transformation environment for modeling and validation of software. *In Proceedings of Applications of Graph Transformations with Industrial Relevance : Second International Workshop*, LNCS 3062:446–453.

Thomas, F., Espinoza, H. and Gérard, S. (2007). Marte : le future standard omg pour le développement dirigé par les modèles des systèmes embarqués temps réel. *Genie Logiciel*, Rapport Technique HPL-2004-187.

Varró, D., Balogh, A. and Pataricza, A. (2006). The viatra2 transformation framework - model transformation by graph transformation. *Eclipse Modeling Symposium*.

Vergnaud, T. (2006). *Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées*. Thèse de doctorat, École nationale supérieure des télécommunications, ENST-Paris.

Yang, Z. et al (2014). From aadl to timed abstract state machines : A verified model transformation. *Journal of Systems and Software*, 93:42–68.

Zennou, S. (2004). *Méthodes d'ordre partiel pour la vérification de systémes concurrents temps réel*. Thèse de doctorat, Université aix marseille I.

Zhang, Y., Dong, Y. and Zhou, W. (2011). A study of the aadl mode based on timed automata. *In IEEE 2nd International Conference on Software Engineering and Service Science (ICSESS)*, pages 224–227. IEEE.

Annexes

ANNEXE A

BNF POUR TA-FORMAT

Dans cette annexe, nous illustrons la Forme Normale de Backus du langage "ta-format" :

```
1   Ita --> VarList ProcList Globals
2   VarList --> Channel VarList Var VarList
3   ProcList --> Proc Proc ProcListGlobals -> system IdList ;Channel -> urgent chan IdList ;
    chan IdList ;
4   Var --> Type IdList ;
5   Proc --> process Id f ProcBody g
6   IdList --> Id Id , IdListProcBody -> StateDecls TransDeclsStateDecls -> state IdList ; commit
    IdList ; init Id ; state IdList ; init Id ;
7   Transdecls --> trans TransList ;
8   TransList --> Trans Trans , TransListTrans -> Id SInv -> Id OpG OpS OpA -> Id f OpG OpS OpA g
9   Sinv --> f InvList gInvList -> Inv Inv , InvList
10  Inv --> Id <= Nat Id < NatOpG -> guard GuardList ;
11  OpS --> Id -> Id?
12  OpA --> assign AssignList ;GuardList -> Guard Guard , GuardList
13  AssignList --> Assign Assign , AssignListType -> clock int
14  Assign --> ClockAssign IntAssignGuard -> Id RelOp Nat Id RelOp Id Op Nat
15  ClockAssign --> Id := Nat
16  IntAssign --> Id := IntExpr
17  IntExpr --> Int * Id Op Nat Id Op Nat Id IntRelOp -> < <= >= > ==Op -> + -
18  Id --> Alpha Id AlphaNumNat -> Num Num Nat
19  Int --> Nat -NatAlpha -> A ... Z a ... z
20  Num --> 0 ... 9
21  AlphaNum --> Alpha Num _
```

Listing A.1: Forme Normale de Backus du langage ta-format

ANNEXE B

CODAGE DU SYSTÈME DE CHAUFFAGE EN AADL

Vous trouvez dans cette annexe le code AADL du système en chauffage :

```
1
2  Thread rempli
3  Features
4  Entree1: in event data port Behavior::integer;
5  Cal1: server subprogram remplir.imp
6  { Behavior_Properties::Server\_Call_Protocol=>HSER;};
7  Sortie1:out event data port Behavior::integer;
8  End rempli;
9  Thread implementation rempli.imp
10 Connections
11 Con1: event data port sortie1->chauffage.entree2 ;
12 Properties
13 Dispatch_protocol=>aperiodic;
14 Period=>50ms;
15 Compute_Execution_Time => 10ms .. 60ms;
16
17 Annex behavior_An2{**
18 States
19 S0:initial state; s1,s2:state;s3:final state;
20 Transitions
21 s0-[entree1?x]->s1{while x<=50 do x:=x+1;cal1?};
22 s1-[on (cal1.s<=50 and x=50) sortie1!(x:=0)]-s2;
23 s2-[on (x=50) and Timeout 50]->s3;
24 connections
25 con_sub:event data port cal.s->sortie1;
26 **};
27 End rempli.imp;
```

```
28
29
30  Thread chaufg
31  Features
32  entree2:in event data port;
33  sortie2:out event data port;
34  End chaufg;
35  Thread implementation chaufg.imp
36  Connections
37  Con2: event data port sortie2-> evacuation.entree3;
38  Properties
39  Dispatch_protocol=>aperiodic;
40  Period=>60ms;
41  Compute_Execution_Time => 10ms .. 70ms;
42
43  Annex behavior_specification {**
44  State
45  s0:initial state;
46  s1:state;s2:final state;
47  transitions
48  s0-[entree2?x]->s1{if x<=60 then entree2:=x+1;};
49  s1-[on x<60]->s0;
50  s1-[on (x=60) ]->s2{x:=0;sortie2!x;};
51  **};
52  End chaufg.imp;package Behavior
53  public
54  data integer end integer;
55  data float end float;
56  data Boolean end boolean;
57  end Behavior;
58  System chauffages
59  end chauffages;
60  System implementation chauffages.imp
61  Subcomponents
62  x:data Behavior::Integer;
63  Proc1:process processus.imp;
64  End chauffages.imp;
65  Process processus
66  Features
67  Pin:in event data port Behavior::integer;
68  End processus;
69  Process implementation processus.imp
70  Subcomponents
71  remplissage:thread rempli.imp;
72  chauffage: thread chaufg.imp;
73  evacuation: thread evacuer.imp;
74  Connections
75  event data port Pin->remplissage.entree1;
76  End processus.imp;
```

```
Subprogram remplir
features
s:in parameter Behavior::integer ;
m: out event data port Behavior::Integer;
end remplir;
subprogram implementation remplir.imp
annex behavior_An1 {**
states
s0:initial return state;
transitions
s0-[]->s0{s:=s+1;m!s;};
**};
End remplir.imp;
Subprogram evacu
Features
s1:in parameter Behavior::integer ;
m1: out event data port Behavior::Integer;
end evacu;
subprogram implementation evacu.imp
annex behavior_An1 {**
states
s0:initial return state;
transitions
s0-[]->s0{s1:=s1-1;m1!s1;};
\\ **};

End evacu.imp;
Thread evacuer
Features
entree3:in event data port Behavior::integer;
sortie3:out event data port Behavior::integer;
cal2: server subprogram evacu.imp
{ Behavior_Properties::Server_Call_Protocol =>HSER;};
End evacuer;
Thread implementation evacuer.imp
Connections
Con3: event data port sortie3->remplissage.entree1 ;
Annex behavior_specification{**
States
s0 :initial state ;
s1,s2 :state ;
s3:final state;
transitions
s0-[entree3?x]->s1{if x<=25 then sortie3:=x+1};
s1-[on x<25]->s0;
s1-[on (x>=20 and x<=25)]->s2{cal2!(s1);};
s2-[on cal2.s1!=0and x<25 ]->s1;\\
s2-[on (cal2.s1=0 and x=25]->s3{x:=0;sortie3!x;};
connections
```

```
126  event data port B2.s1->sortie3 ;
127  **};end evacuer.imp;
```

Listing B.1: Régulateur de temperature en AADL

ANNEXE C

CODAGE DU SYSTÈME DE RÉGULATEUR DE TEMPÉRATURE EN AADL

```
Thread te1
Features
pIn1te1 : in event data port Behavior::integer ;
pIn2te1 : in event data port Behavior::integer ;
pOutte1 : out event data port Behavior::integer ;
End te1;

Thread implementation te1.imp
Connections
Con1: event data port pOutte1->te2.pIn1te2 ;
Properties
Dispatch_protocol => aperiodic ;
Period => 60 ms ;
Compute_execution_Time => 10ms..70ms ;
Annex behavio_te1{**
s0:initial state ;
s1,s2:state ;
s3:final state ;
transitions
s0-[ (pIn1te1?x) or (pIn2te1?x)]->s1 ;
s1-> s2{if y<=60 then pIn1te1:=y+1 ; };
s2-[on y<60]->s1 ;
s2-[on (y=60)]->s3{y:=0 ; pOutte1!y; } ;
**} ;


Thread te2
```

```
Features
pIn1te2 : in event data port Behavior::integer ;
pIn2te2 : in event data port Behavior::integer ;
pOutte2 : out event data port Behavior::integer ;
End te2;

Thread implementation te2.imp
Connections
Con2: event data port pOutte2->te3.pIn1te3 ;
Properties
Dispatch_protocol => aperiodic ;
Period => 50 ms ;
Compute_execution_Time => 10ms..60ms ;
Annex behavio_te2{**
s0:initial state ;
s1:state ;
s2:final state ;
transitions
s0-[pIn1te2?y]->s1{if y<=50 then pIn1te2:=y+1 ; };
s1-[on y<50]->s0 ;
s1-[on (x=50)]->s2{y:=0 ; pOutte2!y; } ;
**} ;

Thread te3
Features
pInte3 : in event data port Behavior::integer ;
pOutte3 : out event data port Behavior::integer ;
End te3;

Thread implementation te3.imp
Connections
Con3: event data port pOutte3->te4.pIn1te4 ;
Properties
Dispatch_protocol => aperiodic ;
Period => 30 ms ;
Compute_execution_Time => 10ms..40ms ;
Annex behavio_te3{**
s0:initial state ;
s1:state ;
s2:final state ;
transitions
s0-[pIn1te3?y]->s1{if y<=30 then pIn1te3:=y+1 ; };
s1-[on y<30]->s0 ;
s1-[on (y=30)]->s2{x:=80 ; pOutte3!y; } ; ** x=t1+t2=50+30=80
**} ;


Thread te4
Features
```

```
pIn1te4 : in event data port Behavior::integer ;
pIn2te4 : in event data port Behavior::integer ;
pIn3te4 : in event data port Behavior::integer ;
pOut1te4 : out event data port Behavior::integer ;
pOut2te4 : out event data port Behavior::integer ;
End te4;
Thread implementation te4.imp
Connections
Con4: event data port pOut1te4->te2.pIn2te2 ;
Con5: event data port pOut2te4->te1.pIn2te1 ;
Properties
Dispatch_protocol => aperiodic ;
Period => 120 ms ;
Compute_execution_Time => 10ms..130ms ;
Annex behavio_te4{**
s0:initial state ;
s1:state ;
s3,s4:final state ;
transitions
s0-[pIn1te4?x]->s1;
s1-->s2{pIn1te4:=x+1; };
s2-[on x<120 and (pIn3te4.count=0)]->s1;
s2-[on ((x<120) and (pIn3te4.count<>0 ))]->s4{x:=0 ; pOut2te4!x;};
s2-[on(x=120)]->s3{x:=0;y:=0; pIn1te2!y } **} ;
Explication
Annex behavio_te1{**
s0:initial state ;
s1:state ;
s3,s4:final state ;
transitions
s0-[pIn1te4?x]->s1;
s1-->s2{pIn1te4:=x+1; };
s2-[on x<120 and (pIn3te4.count=0)]->s1;
s2-[on ((x<120) and (pIn3te4.count<>0 )) ]->s4{x:=0 ; pOut2te4!x;};
s2-[on(x=120)]->s3{x:=0;y:=0; pIn1te2!y } ** si x=120 alors aller  te2 y=0, x=0 **}
```

Listing C.1: Régulateur de temperature en AADL

ANNEXE D

PREUVE DE LA TRANSFORMATION ER2REL EN COQ

Inductive ERSchema : Set := | build_ERSchema (name : string)(Entities : list Entity)(Relships : list Relship)
with Entity : Set :=
| build_Entity (name : string)(IdEntity : string) (belongEnt : list ERAttribute)
with ERAttribute : Set :=
| build_ERAttribute (name : string)(ERtype : string) with Relship : Set :=
| build_Relship (name : string)(ERproperty : string)(Weight : weightType)
(belongRel : list ERAttribute)
(entitySource : Entity)(entitytarget : Entity)
with weightType : Set :=
|m1|mn| a11 | b11 | c11 | other .

...
Inductive RELSchema :
Set := | build_RELSchema (name : string)(Relations : list Relation)
with Relation : Set :=
| build_Relation (name : string)(PKey : string) (belongatt : list RELAttribute)
with RELAttribute : Set :=
| build_RELAttribute (name : string).

```
...
Definition Customer : Customer := build_Customer " add ".
Definition Order : Order := build_Order " valide ".
Definition Article : Article := build_Article "ok".
Definition Request : Request := build_Request "ok".
Definition Category : Category := build_Category "ok".
Definition Provider : Provider := build_Provider "ok".
...
Fixpoint belongEnt2belongatt (a : list ERAttribute ) : list RELAttribute := match (a) with
| nil => nil
|x : : nil => ERAttribute2RELAttribute (x ) : : nil /* **** */
|x : :l=>( ERAttribute2RELAttribute (x ) : : nil )++( belongEnt2belongatt (l)) end .
Definition Entity2Relation (a : Entity ) : Relation := build_Relation
( Entity_name (a)) ( Entity_IdEntity (a)) ( belongEnt2belongatt (
Entity_belongEnt (a ))).
...
Fixpoint relships2Rlations (a : list Relship ) : list Relation :=
match (a) with
| nil => nil
|x : : nil => match weightTypeofmn (x) with
| true => Relschip2Relation (x ) : : nil
| false => match weightTypeofa11 (x) with
| true => Relschip2Relation (x ) : : nil
| false => match weightTypeofother (x) with
| true => nil
| false => nil end
end
end
|x : :l=> match weightTypeofmn (x) with
| true => ( Relschip2Relation (x ) : : nil )++ relships2Rlations (l)
| false => match weightTypeofa11 (x) with
| true => ( Relschip2Relation (x ) : : nil )++ relships2Rlations (l)
| false => match weightTypeofother (x) with
```

```
| true => relships2Rlations (l)
| false => relships2Rlations (l) end
end
end
end .
...
—- rule 1 —-
Definition ERAttribute2RELAttribute (a : ERAttribute ) : RELAttribute :=
build_RELAttribute ( ERAttribute_name (a)) .
—- proof of the rule 1 —
Axiom Axiome3 : forall name : string , build_RELAttribute name =
build_RELAttribute "".
Axiom Axiome2 : build_RELAttribute "" = build_RELAttribute "".
Axiom Axiome1 : forall name : string , forall a :ascii ,
build_RELAttribute ( String a name ) = build_RELAttribute "".
—- formalization of the proof —-
Lemma Prov01 : ( forall a : ERAttribute , forall b : RELAttribute ,
( ERAttribute2RELAttribute (a) ) =b).
Proof . intros a. intros b. unfold ERAttribute2RELAttribute .
unfold ERAttribute_name .
induction a. induction b. induction name0 . induction name . simpl; auto .
rewrite ( Axiome1 name ). simpl; auto . rewrite ( Axiome1 name0 ).
simpl; auto .
rewrite ( Axiome3 name ). simpl; auto .
....
```