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

Université Mohamed Sadik Benyahia de Jijel
Faculté des Sciences Exactes et informatique
Département d'Informatique

Mémoire de fin d'étude

pour obtention du diplôme Master de Recherche en Informatique

Option : *Systèmes d'Information et Aide à la Décision*

Thème

Modélisation et vérification des systèmes temps réel dynamiquement reconfigurables : Application sur Autovision.

Présenté par :

LEFOUILI Hadjiba.

KAABAR Houda.

Encadré par :

Mr.ALLIOUCHE Abdelaziz

Promotion : 2017/2018.

⁎ *Remerciements* ⁎

Nous remercions le grand dieu le tout puissant, de nous avoir donné le bon sens et la grande volonté pour réaliser ce travail.
Nous remercions du fonds du coeur, notre encadreur : Mr Alliouche Abdelaziz pour nous avoir aidé et guidé en toute sincérité tout au long de notre préparation du mémoire. Nous remercions aussi tous nos professeurs de l'université pour leur contribution à notre formation scientifique.
Nous tenons à remercier très sincèrement l'ensemble des membres du jury qui me font le grand honneur d'accepté de juger mon travail.
Enfin, j'adresse mes plus sincères remerciements à tous ceux qui nous ont aidés, de près ou de loin, à réaliser ce travail.

⁂ Dédicaces ⁂

Je dédie ce modeste travail à :

⁂ *Mes très chers parents ;*

⁂ *Toutes ma famille ;*

Table des matières

Table des figures

Liste des abréviations

DEVS Discret Event System Specification

DSDEVS Dynamic Structue Discret Event System Specification

RecDEVS Reconfigurable Discret Event System Specification

IDM Ingénierie Dirigée par les Modèles

MDA Model Driven Architecture

CIM Computational Independant Model

PIM Platform Independent Model

PSM Platform Specific Model

ATOM3 A Tool for Multi-formalism and Meta-modelling

MDSL Modelling, Simulation and Design Lab

Introduction générale

Il est nécessaire d'insister aujourd'hui sur la présence des systèmes embarqués dans notre quotidien. Ces systèmes souvent critiques, mettent en jeu des coûts importants. Ils sont de plus en plus autonomes, complexes et interagissent directement avec l'environnement. Ils existent dans les transports, l'électroménager, l'informatique, l'autonomique, les équipements médicaux,...etc.

Donc ces systèmes embarqués sont souvent utilisés par le public dans la vie de tous les jours sans même qu'on ne s'en rende compte, par exemple dans les systèmes de freinage d'une voiture, le contrôle de vol d'un avion,...etc.En général, un système embarqué est un système complexe qui intègre du matériel et du logiciel conçus ensemble afin de fournir des fonctionnalités données. Il contient généralement plusieurs microprocesseurs destinés à exécuter un ensemble de programmes stockés dans la mémoire. Les systèmes embarqués fonctionnent généralement en temps réel. Un système temps réel se déstingue des autres systèmes informatiques par le fait qu'il est soumis à des contraintes temporelles fortes qui lui permettent d'interagir avec l'environnement d'un procédé à contrôler.

La spécification d'un système revient à définir celui-ci à l'aide d'un ou plusieurs modèles, selon un formalisme de modélisation. Pour cela ce genre de systèmes peut être représenté par un modèle RecDEVS, afin de pouvoir simuler et vérifier leurs comportements à travers un outil particulier (UPPAAL dans notre cas). Cependant une telle opération n'est pas possible directement du fait que l'outil de vérification a son propre modèle de représentation et ne supporte pas les modèles RecDEVS, ainsi une transformation de modèle est fortement exigée qui permet de transformer un modèle RecDEVS à un modèle UPPAAL équivalent. Notre travail se concentre sur un système à temps réel Autovision qui est un aide-conducteur, devenu très indispensable de nos jours pour prévenir le conducteur d'un véhicule à propos des différents dangers sur la route et lui aider à prendre la bonne décision en temps réel. Actuellement ce système vise en principe la détection des tunnels à l'aide d'un ensemble de composants dynamiquement reconfigurables.

Problématique et objectifs du travail

L'objectif de notre travail se résume en la proposition d'un méta-modèle pour les modèles RecDevs, la modélisation du système Autovision, la transformation du modèle résultat en un modèle UPPAAL équivalent à travers une grammaire de graphe, et à la fin la vérification du comportement de ce système.

Organisation du mémoire

Notre mémoire est organisé comme suit :

- ✓ Le chapitre 1 est consacré aux modèles DEVS et RecDEVS,dans lequel nous expliquons le principe de ces modèles à travers quelques exemples.
- ✓ Dans le chapitre 2 nous expliquons le principe de l'outil de vérification UPPAAL, et nous donnons des exemples sur leurs modèles et la structure des fichiers modèles.
- ✓ Dans le troisième chapitre nous présentons le système Autovision et ses différents composants statiques et dynamiques, et nous donnons le rôle de chaque composant et le principe général de son fonctionnement.
- ✓ Le quatrième chapitre sera consacré à la transformation des modèles à travers les grammaires de graphes. Cette approche permet l'automatisation des processus de modélisation, depuis les phases de développement jusqu'à celles des tests,en passant par la génération de code. Nous présentons par la suite une énumération de quelques types de transformation de modèles, suivie d'une classification de certaines approches de transformation. Sur un autre plan, nous mettons l'accent sur les transformations de graphes qui représentent une des approches de transformation de modèles et cela après avoir fait un bref aperçu sur la notion de graphe. Enfin, nous présentons l'outil AToM3, l'outil de transformation utilisé dans notre travail.
- ✓ Dans le dernier chapitre nous présentons notre approche de transformation de modèles RecDEVS élaborés en Atom3 vers les modèles UPPAAL. Cette tâche est totalement faite à travers l'outil Atom3 qui permet d'une part la modélisation et de l'autre part la transformation de graphe à l'aide d'une grammaire de graphe. Ainsi nous présentons notre méta-modèle proposé pour les modèles RecDEVS, le modèle correspondant au système Autovision, et puis notre grammaire de graphe qui permet de transformer un modèle RecDEVS vers un modèle UPPAAL. A la fin de ce chapitre nous donnons quelques facteurs qui peuvent servir à la vérification du fonctionnement de ce système.

Enfin, nous terminons notre mémoire par une conclusion générale dans laquelle nous donnons quelques points de perspectives pour une ultérieure amélioration.

Chapitre 1

Formalisme DEVS et RecDEVS

1.1 Introduction

Le formalisme DEVS est un formalisme modulaire et hiérarchique pour l'analyse, la modélisation et la simulation des systèmes complexes. Ce formalisme de modélisation et de simulation, est issu des mathématiques discrètes, pour définir des modèles complexes et permettre leurs hiérarchisations. Il est basé sur les événements discrets pour la modélisation des systèmes discrets et continus. Les modèles sont en interaction via l'échange des événements. Aujourd'hui, il ya beaucoup des travaux scientifiques sur le formalisme DEVS dans les différents domaines d'application comme les systèmes industriels, la biologie, l'épidémiologie ou l'agronomie.
Nous commençons ce chapitre par une présentation de formalisme DEVS, son historique, et ses extensions surtout nous intéressons à son extension RecDEVS, et nous essayons d'expliquer ses modèles de base : composant, exécutif réseau et système de bus.

1.2 La théorie de la modélisation et la simulation

Une architecture conceptuelle pour la modélisation et la simulation a été proposée des systèmes particulièrement adaptés au formalisme DEVS. Comme le montre la Figure 1.1, cette architecture présente trois entités :

1.2.1 Le système

C'est le phénomène observé dans un environnement donné. L'environnement donne les spécifications des conditions dans lesquelles évolue le système et permet son expérimentation et sa validation[1].

1.2.1.1 Système à évènement discret

Les systèmes à événements discrets (SED) sont des systèmes dynamiques dont l'espace des états est discret et dont l'évolution est conforme à l'occurrence d'événements physiques à des intervalles de temps éventuellement irréguliers ou inconnus [2].

1.2.2 Le modèle

Un modèle est une approximation, une abstraction de la réalité. Il peut être exprimé par des relations mathématiques, des représentations graphiques, des symboles, des lettres,...[2]

1.2.3 Le simulateur

C'est une entité qui est responsable de l'interprétation du modèle(exécuter ces instructions) pour générer son comportement.
Ces entités sont reliées par deux relations comme représenter dans la figure 1.1 :

1. **La relation de modélisation :** elle est composée des règles de construction et de validation du modèle.
2. **La relation de simulation :** elle est composée des règles d'exécution du modèle qui permettent au simulateur de générer correctement le comportement attendu du système [1].

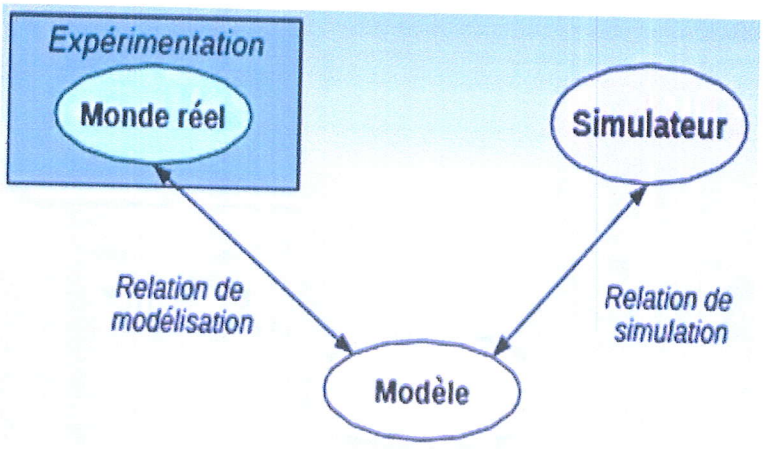

FIGURE 1.1 – Modélisation et simulation

1.3 Le formalisme DEVS

Le formalisme DEVS a été introduit à la fin des années 70 par le professeur B.P Zeigler qui avait pour objectif de donner un socle mathématique solide à la modélisation et la simulation (MS) des systèmes à événements discrets. Un système à événements discrets est un système pouvant être décrit à partir d'un ensemble d'états et de règles de transition entre ces états. Le formalisme DEVS permet la représentation d'un système comme un modèle ou un ensemble de modèles possédant des états et des transitions. De plus, il donne la possibilité de définir de manière distincte la structure d'un système [1].

1.3.1 Modélisation DEVS

Le formalisme DEVS repose sur la définition de deux types de modèles : les modèles atomiques et les modèles couplés.

1.3.1.1 Modèle atomique

Il décrit l'aspect comportemental des systèmes. Il est non décomposable et constitue l'objet fondamental pour la modélisation DEVS. Il est défini par un ensemble d'états et de transition entre ces états.

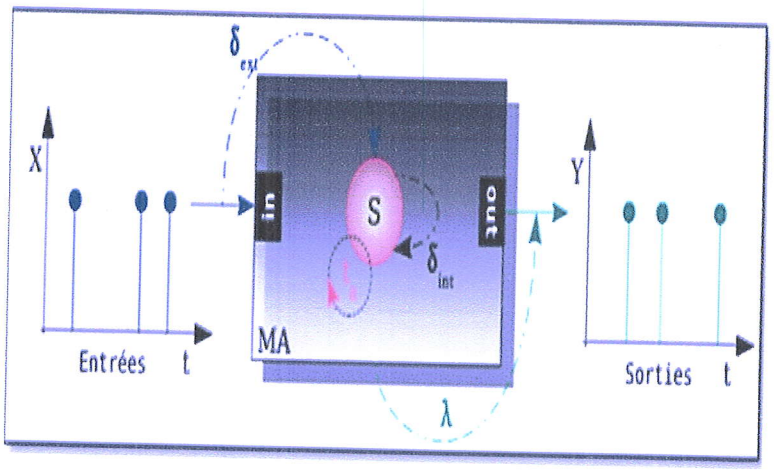

FIGURE 1.2 – Description d'un modèle atomique DEVS

Les modèles atomiques sont définis comme ci-dessous :
MA :< X ; Y ; S ; ta ; δ_{ext} ; δ_{int} ; λ >

- ✓ X : est l'ensemble des valeurs d'entrée du modèle caractérisé par les informations port, valeur.
- ✓ Y : est l'ensemble des valeurs des évènements sortants caractérisé par les informations port, valeur.
- ✓ S : est l'ensemble des états partiels ou séquentiels incluant les variables d'états.
- ✓ ta : S $\rightarrow$ R+ : est la fonction d'avancement du temps qui est utilisée pour déterminer la durée de vie de l'état.
- ✓ δ_{ext} : Q $\rightarrow$ X $\rightarrow$ S : est la fonction de transition externe qui définit comment un message d'entrée X peut changer l'état du système, où :
  - ➤ Q = (s,te) | s $\in$ S, 0 < te < ta(Si) est l'ensemble des états
  - ➤ te est le temps écoulé depuis le dernier évènement
- ✓ δ_{int} : S $\rightarrow$ S : est la fonction de transition interne qui définit comment un état du système change de façon interne quand le temps écoulé atteint la durée de vie de l'état.
- ✓ λ :S $\rightarrow$ Y : est la fonction de sortie qui définit comment un état du système génère des évènements de sortie quand le temps écoulé atteint la durée de vie de l'état [3].

Les modèles atomiques réagissent à deux types d'évènements externes ou internes. Un évènement externe provient d'un autre modèle, il déclenche la fonction de transition externe (δ_{ext}) et met à jour le temps de vie de l'état (ta(Si)). Un évènement interne entraîne un changement d'état du modèle. Il déclenche les fonctions de transition interne (δ_{int}) et de sortie (λ). Le modèle calcule ensuite avec la fonction d'avancement du temps (ta)la date du prochain évènement interne [4].

1.3.1.2 Modèle couplé

Il représente l'aspect structurel. Il est possible pour un modèle couplé de comprendre un ou plusieurs modèles atomiques ou couplés. Il peut aussi être représenté par un modèle atomique unique.

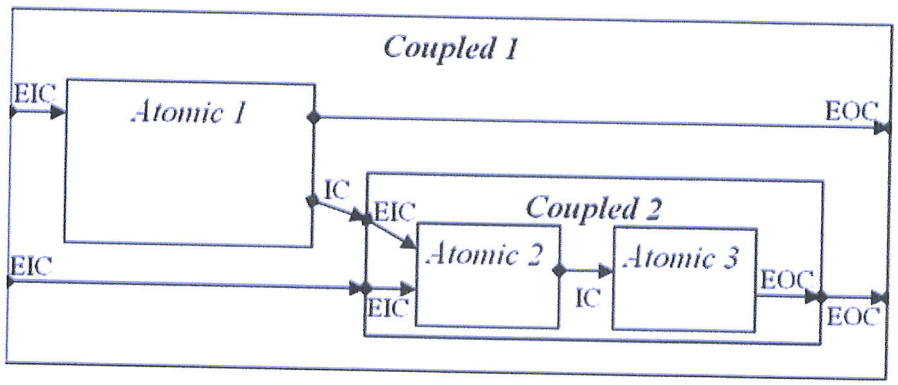

FIGURE 1.3 – Exemple d'un modèle couplé DEVS

C'est un exemple simple des niveaux de hiérarchie que DEVS est capable de décrire. Les fonctions de couplage (EOC, EIC et IC) sont marquées, et les ports sont représentés par des losanges noirs. Par exemple, le modèle couplé 1 possède deux ports d'entrée, et deux ports de sortie. Le modèle atomique 2 possède un port d'entrée, et un port de sortie. Il a été prouvé que DEVS était ≪fermé sous couplage≫, ce qui signifie qu'un modèle couplé (quel que soit le nombre de ses sous modèles) peut être transformé en modèle atomique unique, qui en est l'exact l'équivalent.

Les modèles couplés sont définis comme ci-dessous :

{**CM :< XM, YM, CM, EIC, EOC, IC, Select >** }, avec :

- ✓ XM : tous les ports d'entrée.
- ✓ YM : tous les ports de sortie.
- ✓ CM : la liste des modèles formant le modèle couplé.
- ✓ EIC : tous les couplages qui connectent les entrées du modèle couplé à ses composants.
- ✓ EOC : tous les couplages qui connectent les sorties du modèle couplé à ses composants.
- ✓ IC : tous les couplages internes qui connectent les composants entre eux.
- ✓ Select : définit une priorité entre événements simultanés destinés à des composants différents [3].

1.3.2 Extensions DEVS

Le formalisme DEVS a bénéficié au fil du temps de nombreuses extensions, chacune permettant de résoudre un type de problème particulier ou de mieux s'adapter à des domaines spécifiques. Nous allons concentrer notre étude sur l'extension RecDEVS (combinaison de PDEVS et DSDEVS).

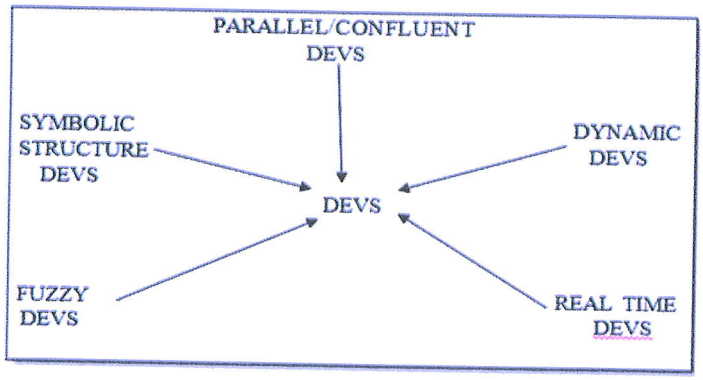

FIGURE 1.4 – Extensions de DEVS

1.3.2.1 DEVS parallèle

Dans cette version, la notion de «event bag» à été ajoutée. Cette notion intègre le fait que plusieurs événements intervenant au même instant peuvent être regroupés dans un bag. L'ensemble est noté X^b. De même manière, un modèle atomique peut envoyer plusieurs événements en sortie au même instant. L'ensemble de sortie est noté alors Y^b. La fonction δ_{con} (S $\times X^b$)est introduite afin de résoudre le conflit d'exécution entre les fonctions de transition interne et externe d'un modèle atomique.
L'association de ces deux notions (bag et δ_{con}) permet de gérer les collisions entre les fonctions de transition interne et externe et en même temps de traiter plusieurs événements arrivants au même instant sur un modèle atomique et générer un troisième type de transition (transition confluence). Dans le formalisme DEVS classique, à chaque arrivée d'un événement, la fonction de transition externe est invoquée. Il y avait donc autant d'invocations que de messages simultanés sur les ports d'un modèle atomique. Avec cette extension, les événements simultanés sont disponibles dans l'unique invocation de la fonction de transition. Cette extension permet donc d'améliorer les performances du simulateur [1].

1.3.2.2 DEVS à structure dynamique (DSDEVS)

Bien que le formalisme Devs n'est pas ciblé à l'origine aux architectures dynamiquement reconfigurables, DSDEVS a été amélioré et donc adopté à des fins de reconfiguration matérielle. Dans ce but, DSDEVS a introduit un composant de système spécial, l'exécutif du réseau C_x, qui fait partie de chaque système DSDEVS.

1.3.2.3 C'est quoi une reconfiguration

C'est l'action de configurer à nouveau, ainsi c'est le changement de spécialisation d'un système par changement ou ajout d'équipements. Il existe plusieurs méthodes de reconfiguration :

- **La reconfiguration statique :** La méthode statique permet de modifier la configuration matérielle du système. Elle est dite statique car son action n'intervient pas durant la phase d'exécution du circuit, le système doit être stoppé par l'utilisateur avant qu'une nouvelle configuration puisse être téléchargée.
- **La reconfiguration partielle :** Dans l'architecture partiellement reconfigurable une petite partie de l'ensemble de matériel est reconfigurée. Dans ce cas la partie restante de matériel restent tel qu'il est.
- **La reconfiguration dynamique :** C'est un cas particulier de la reconfiguration partielle. Tandis que certains composants doivent être reconfigurés, les autres composants continuent leur exécution. Cette situation est décrite dans la figure 1.6 [6]

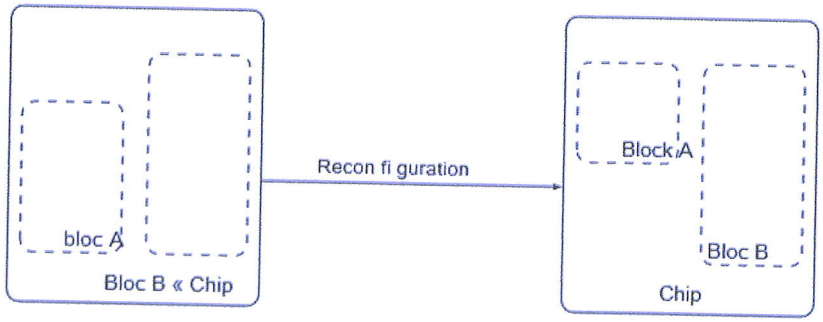

FIGURE 1.5 – Reconfiguration partielle dynamique

1.3.2.4 La définition de la Reconfiguration DEVS (RecDEVS)

L'extension RecDEVS représente les différentes propriétés particulières des architectures matérielles reconfigurables. Son concept le plus fondamental est la représentation des blocs matériels reconfigurables comme les composants du DEVS. Ainsi, quelques types des fonctions dynamiques ont été introduits pour modéliser la reconfiguration au sein de RecDEVS.
L'exécutif dédié réseau C_x est l'élément responsable de la reconfiguration du système.

$$S^{RecDevs} = < X_i, Y_i, D, C_x >$$

D = Ensemble de tous les composants disponibles DEVS.

Une liste des noms disponibles de composants nommée D a été ajoutée à la description du système mondial RecDEVS. Cette liste peut être comparée à une liste de type de

composants disponibles.Ainsi, il est possible d'ajouter plusieurs composants DEVS du même type à un seul système.
RecDEVS repose sur une communication basée sur les messages. Chaque instance de composant est définie par son type d ∈ D et un identifiant unique ID ∈N. Ainsi, l'ensemble I = {C (ID, d) | d ∈ D, ID ∈ N} peut être utilisé pour répondre à tous les composants instanciés dans une configuration.
Une communication entre deux composants est réalisée par l'envoi d'un message. Chaque message se compose d'un triplet (expéditeur, destinataire, donnée), où l'expéditeur et le récepteur appartiennent à I [6].

1.3.2.5 Reconfiguration en RecDEVS

Un des objectifs de RecDEVS est de modéliser le système de telle sorte que l'exécutif du réseau reste transparent pour un développeur de système. Un processus de reconfiguration doit être complètement encapsulé dans les composants DEVS participant, où la reconfiguration dépend de l'état actuel de l'exécutif du réseau. D'autres composants ne peuvent pas déclencher une reconfiguration directement, à savoir, ils ne peuvent pas modifier cet état. Par conséquent, la tâche de reconfiguration se déplace du domaine fonctionnel au domaine de communication en définissant un ensemble dédié de messages de reconfiguration.
La reconfiguration est maintenant déclenchée par l'envoi des messages correspondant à l'exécutif du réseau. Ceci nécessite qu'une modification de la fonction de sortie locale λ, un changement de C_x est très nécessaire.
La reconfiguration du composant RecDEVS se compose d'une séquence fixe de messages comme suit :

➢ **L'ajout d'un composant :**

➤ Si le composant $C_{d'}^{i'}$ veut créer un nouveau composant de type d'' ∈ D, il envoie un message ($C_{d'}^{i'}$, d', C_x, (nouveau d'')) à l'exécutif du réseau.

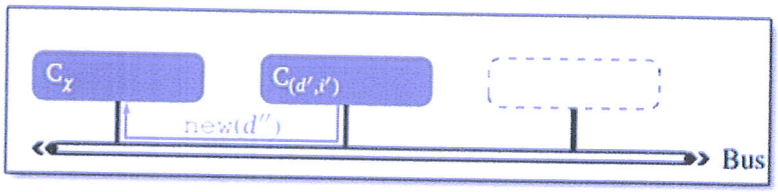

FIGURE 1.6 – Envoie de message de création (new(d'')) à C_x.

➤ C_x reçoit le message et exécute une transition externe δ_{ext}. Cela va créer une nouvelle RecDEVS composante $C_{d''}^{i''}$ et l'ajouter à la liste des composants instanciés.

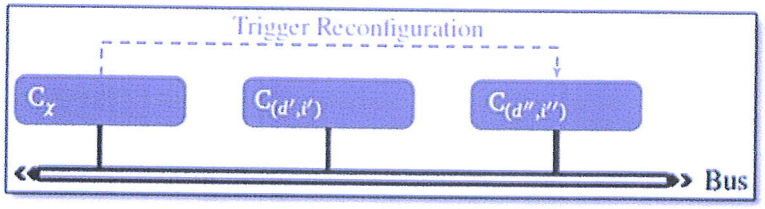

FIGURE 1.7 – Création d'un nouvel composant RecDEVS

➤ Un message de confirmation (C_x,$C^{i'}_{d'}$, (confirmer, $C^{i''}_{d''}$) avec l'adresse du nouveau composant est ensuite envoyé à l'expéditeur.

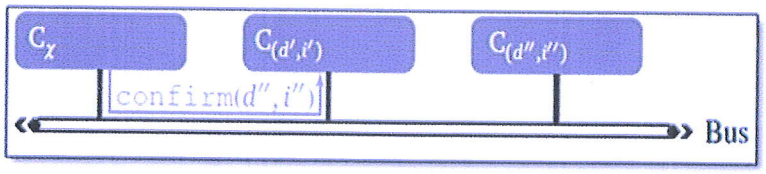

FIGURE 1.8 – Message de confirmation

➤ A la réception du message de confirmation à l'origine peut traiter le composant nouvellement créé.

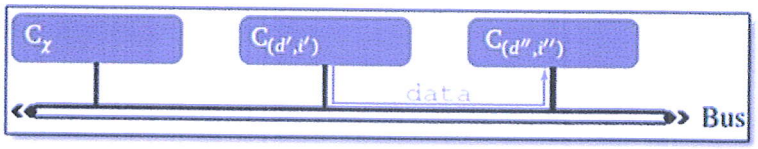

FIGURE 1.9 – Échange des données avec le nouveau composant

➤ Si plusieurs composants envoient des messages en même temps, tous les messages peuvent se trouver sur le système de bus en même temps. Il appartient à une mise en oeuvre effective de RecDEVS pour limiter le nombre de messages simultanés.

➢ **La suppression d'un composant :**La suppression des composants est légèrement différente. Dans RecDEVS chaque composant peut être supprimé que lorsqu'il atteint un état cohérent avant qu'il ne cesse de fonctionner.

➤ Pour se supprimer, un composant $C^{i'}_{d'}$ envoie le message ($C^{i'}_{d'}$, C_x, (del)) à l'exécutif du réseau.

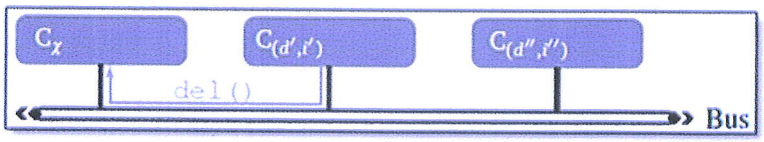

FIGURE 1.10 – Envoie de message de suppression (del ()) à C_x.

➤ C_x Reçoit le message et libère les ressources de $C_{d'}^{i'}$ Cependant, il n'est pas obligatoire de supprimer ce composant immédiatement. La suppression peut avoir lieu après que l'exécutif du réseau a reçoive le message correspondant.

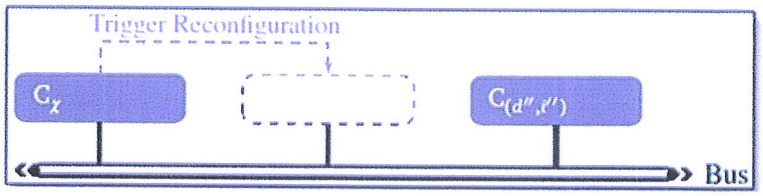

FIGURE 1.11 – Suppression de composant.

Un composant est supprimé d'un certain temps après qu'il a été complètement retiré de la structure de communication. Dans ce cas, il est nécessaire d'adapter la structure de communication lors de la suppression du composant. Il appartient au système conçu pour veiller à ce qu'un composant ne réagit pas aux messages entrants après l'émission d'un message de suppression [6].

1.4 Conclusion

Nous avons présenté dans ce chapitre les notions théoriques de la modélisation et de la simulation de système, et en particulier nous avons détaillé le formalisme DEVS. En effet, ce dernier peut être défini comme une méthodologie universelle et générale qui fournit des outils pour modéliser et simuler des systèmes dont le comportement repose sur la notion d'évènements. Ce formalisme est basé sur la théorie générale des systèmes, la notion de modèle, et permet la spécification de systèmes complexes à évènements discrets sous forme modulaire et hiérarchique. Ainsi nous avons motivé et décrit les caractéristiques spécifiques qui sont nécessaires pour les systèmes reconfigurables dynamiquement. Enfin, nous avons décrit le mécanisme de reconfiguration dans RecDEVS au moyen d'un directeur de réseau dédié et le système général de message sur la base de communication approprié pour les systèmes matériels reconfigurables.

Chapitre 2

UPPAAL

2.1 Introduction

La vérification des systèmes réactifs, systèmes critiques ou des systèmes embarqués est un problème très important aujourd'hui. Les approches par model checking se sont largement développées ces dernières années. Cela suppose de modéliser le comportement du système à vérifier et d'énoncer la propriété de correction attendue. Ensuite on peut utiliser un model checker afin de vérifier si le modèle satisfait ou non la propriété.
Le model checker UPPAAL est une boîte à outils pour la vérification des systèmes en temps réel développée conjointement par l'université d'Uppsala et l'université d'Aalborg. Il dispose d'une interface utilisateur Java et d'un moteur de vérification écrit en C++. Dans cette section nous allons voir en détail cet outil.

2.2 Vérification des systèmes temps réel

2.2.1 Systèmes à temps réel

Les systèmes temps réels réactifs sont définis par leur capacité à réagir aux sollicitations de leur environnement en se conformant à un certain nombre de contraintes temporelles.En un temps limité, le système doit acquérir et traiter les données et les événements caractérisant l'évolution temporelle de cet environnement, prendre les décisions appropriées et les transformer en actions [7].

2.2.2 Vérification formelle

La vérification 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.
Le model checking représente l'une des techniques possibles de la dernière étape de la

vérification formelle. Son rôle illustré dans la figure 2.1 est de vérifier qu'une propriété donnée représentée par une formule logique est satisfaite par le modèle établi du système [8].

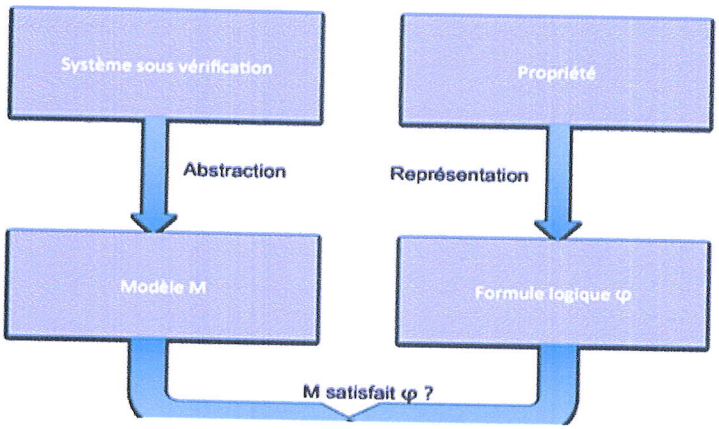

FIGURE 2.1 – Vérification d'un système par model checking

2.2.3 Le model checking

Le model checking est une approche automatisée permettant de vérifier qu'un modèle de système est conforme à ses spécifications. Le comportement du système est formellement modélisé, via des automates, réseaux de Petri...ect, les spécifications exprimant les propriétés attendues du système, sont formellement exprimées par exemple via des formules de logiques temporelles. En pratique, les propriétés du système sont souvent classées en deux grandes catégories informelles. Les propriétés de sûreté énonce qu'une situation particulière ne peut être atteinte. Les propriétés de vivacité énonce quelque chose de mauvais (ou de bon) qui finira par se produire [9].

2.3 L'outil UPPAAL

2.3.1 Définition

UPPAAL est un ensemble d'outils pour la vérification automatique des propriétés de sûreté et de vivacité bornée, des systèmes temps réel. Il est construit avec l'architecture Client/Serveur(Figure 2.2). La machine UPPAAL est le serveur, elle est développée en C++. L'interface graphique utilisateur(GUI) est le client, développé en JAVA. La communication s'effectue via des protocoles internes. Ainsi, la conception d'UPPAAL offre la possibilité d'exécuter le serveur et l'interface GUI sur deux machines différentes. La machine UPPAAL est constituée de plusieurs outils tel que checkta (Syntax Checker) et verifyta (Model Checker). L'interface graphique utilise des outils tel que atg2ta et hs2ta [10].

- **atg2ta :** C'est un programme de transformation de représentation graphique (.atg) d'un réseau d'automates en représentation textuelle en UPPAAL (.ta).
- **hs2ta :** Ce programme est capable de transformer des automates hybrides linéaires en réseaux d'automates temporisés.
- **checkta :** C'est un programme assurant la vérification syntaxique de la modélisation d'un système donnée en format textuel.
- **verifyta :** Ce programme est le noyau de la vérification dans UPPAAL. Il reçoit en entrée la description du système et une propriété à vérifier. Il répond par "oui" ou "non" [11].

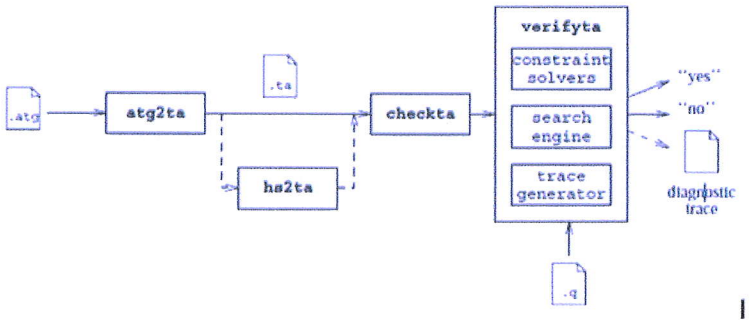

FIGURE 2.2 – Aperçu d'UPPAAL

2.3.2 Environnement UPPAAL

L'idée derrière l'outil est de modéliser un système avec des automates temporisés en utilisant un éditeur graphique, simulez-le pour valider qu'il se comporte comme prévu, et enfin pour vérifier qu'il est correct par rapport à un ensemble de propriétés. L'interface graphique (GUI) du client Java reflète cette idée est divisé en trois principaux parties : l'éditeur, le simulateur et le vérificateur, accessibles via trois "onglets".

2.3.2.1 L'éditeur

Un système est défini comme un réseau d'automates temporisés, appelés processus dans l'outil, mis en parallèle. Un processus est instancié à partir d'un modèle. L'éditeur (figure2.3) est divisé en deux parties : une arborescence pour accéder aux différents modèles et déclarations et un éditeur de dessin [12].

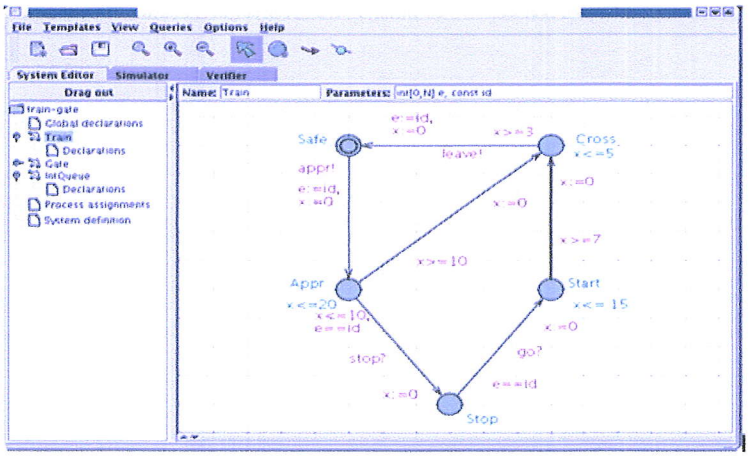

FIGURE 2.3 – L'éditeur UPPAAL

2.3.2.2 Le simulateur

Le simulateur peut être utilisé de trois façons : l'utilisateur peut exécuter le système manuellement et choisir les transitions à prendre, le mode aléatoire peut être basculé pour laisser le système fonctionner seul, ou l'utilisateur peut passer par une trace (enregistré ou importé du vérificateur) pour voir comment certains états sont accessibles. La figure 2.4 montre le simulateur. Elle est divisée en quatre parties [12] :

- **The control part :** permet de choisir de feu activé transitions, passer par une trace, et basculer la simulation aléatoire
- **The variable view :** indique les valeurs des variables entières et des contraintes d'horloges
- **The system view :** montre tous les automates instanciés et les emplacements actifs du l'état actuel.
- **The message sequence chart :** montre les synchronisations entre les différents processus ainsi que les emplacements actifs à chaque étape.

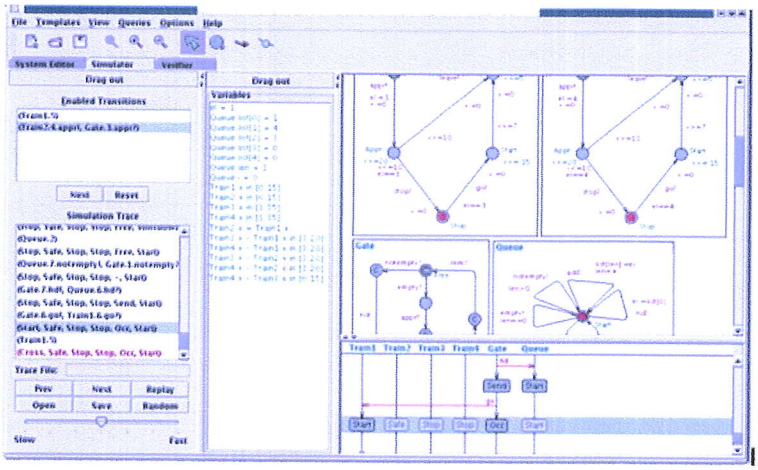

FIGURE 2.4 – Le simulateur UPPAAL

2.3.2.3 Le vérificateur

An niveau de vérificateur, l'utilisateur peut modéliser une ou plusieurs propriétés, insérer ou supprimer des propriétés, et basculer la vue pour voir les propriétés ou les commentaires dans la liste. Quand une propriété est sélectionnée, il est possible de modifier sa définition. Ou des commentaires pour documenter ce que signifie la propriété informelle. Le panneau d'état au bas montre la communication avec le serveur. Lorsque la génération de trace est activée et le model-checker trouve une trace, l'utilisateur est demandé si elle veut l'importer dans le simulateur. Les propriétés satisfaites sont marquées vertes et celles non satisfaites sont marquées rouge [12].

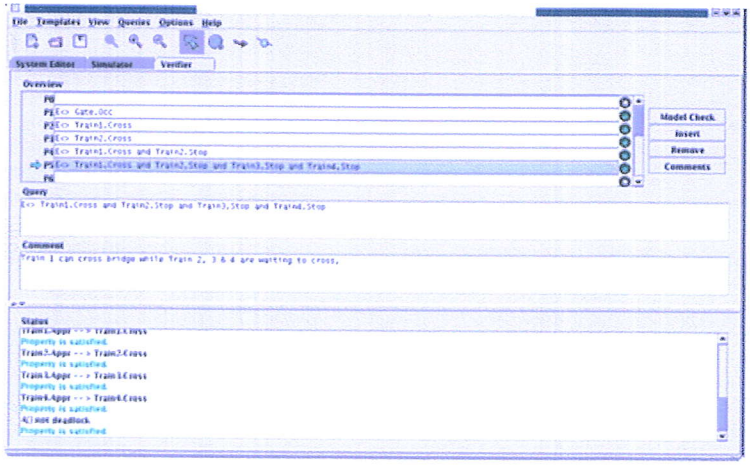

FIGURE 2.5 – Le vérificateur UPPAAL

2.4 Les automates temporisés en UPPAAL

UPPAAL est basé sur une extension des automates temporisés. Un automate temporisé est une machine d'états finie étendue avec des variables d'horloge. Toutes les horloges avancent

en même temps. Un système est modélisé en tant que réseau de tels automates en parallèle. De plus, le modèle est étendu avec les variables bornées entières. Un état du système est défini par les localités des automates, les valeurs des horloges et les valeurs des variables entières. Le système change d'état en prenant une transition qui peut comprendre un arc dans n'importe quel automate.

2.4.1 Syntaxe d'automate temporisé

Un automate temporisé est un 6 uplet (L,l_0, X,Σ, E, Inv) où [13]

- L est un ensemble fini de localité,
- $l_0 \in$ Q est la localité initial,
- X est un ensemble fini d'horloges (à valeur réelle positive),
- Σ est un ensemble fini d'action,
- E $\subseteq$ L $\times C(X) \times \Sigma \times 2^x \times L$ est un ensemble fini de transitions ; e $=< l,\delta,\alpha,\mathcal{R}, \rho, l' >$ $\in$T représente une transition de la localité l vers la place l', δ est la garde (contrainte sur les horloges) associée à e,α est l'étiquette de e et $\mathcal{R}$ est l'ensemble d'horloges devant être remises à zéro et ρ est la fonction d'affectation d'horloge.
- Inv : L$\longrightarrow$C(X) associe un invariant à chaque localité,

Exemple d'un automate temporisé :

La figure 2.6(a) montre un automate temporisé modélisant une lampe simple. La lampe a trois emplacements : éteint, bas et lumineux. Si l'utilisateur appuie sur un bouton, c'est-à-dire synchronise avec press ?, la lampe est allumée. Si l'utilisateur appuie sur bouton à nouveau, la lampe est éteinte. Cependant, si l'utilisateur appuie deux fois sur le bouton rapidement, la lampe s'allume et devient claire.

Le modèle utilisateur est représenté sur la figure 2.6(b). L'utilisateur peut appuyer sur le bouton au hasard à tout moment ou même ne pas appuyer sur le bouton du tout. L'horloge y de la lampe est utilisée pour détecter si l'utilisateur était rapide ($y \leq 5$) ou lent ($y \geq 5$) [12].

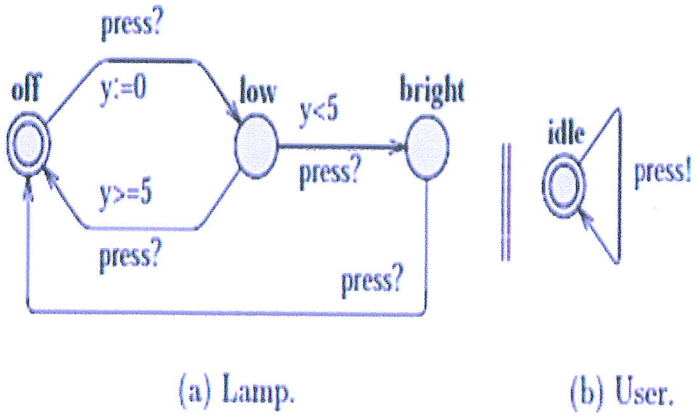

FIGURE 2.6 – Automate temporisé d'une simple lampe

2.4.2 Quelques notions

Les automates temporisés en Uppaal se constituent d'un ensemble d'horloges, invariants, gardes, canaux de synchronisation, transitions, locations urgent et committed.

➢ **Les invariants :**
les invariants expriment des contraintes sur les horloges afin de rester dans un noeud particulier, un invariant c'est une conjonction de conditions de la forme $x \leqslant const$ ou $x \leq const$ où x est une horloge et const est un nombre entier.

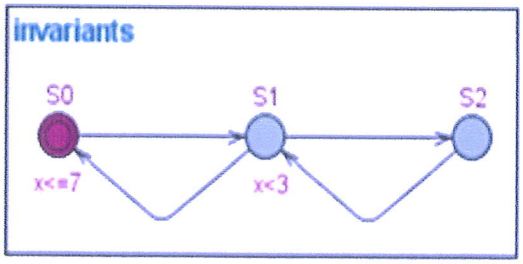

FIGURE 2.7 – Modèle avec invariants

➢ **Les gardes :**
un garde est similaire à un invariant, c'est une expression particulière qui doit être satisfaite pour que la transition soit prise.

➢ **Les types des états :**
On peut distinguer trois différents types d'états en UPPAAL : normal, urgent (U) et Committed (C) sans invariant.

☞ **Etat normal :** Avec ou sans invariant.

☞ **Etat urgent :** Lorsque le système est dans un état urgent, le passage du temps est interdit, l'état doit immédiatement être quitté quand une des transitions sortantes peut être exécutée.

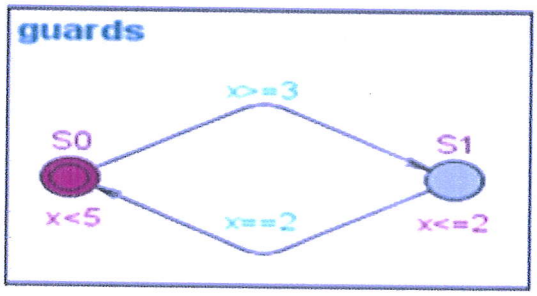

FIGURE 2.8 – Modèle avec gardes

☞ **Etats committed :** L'état committed est le même que d'un état d'urgence, et quand on est dans un tel état, la prochaine transition doit faire sortir de l'état committed.

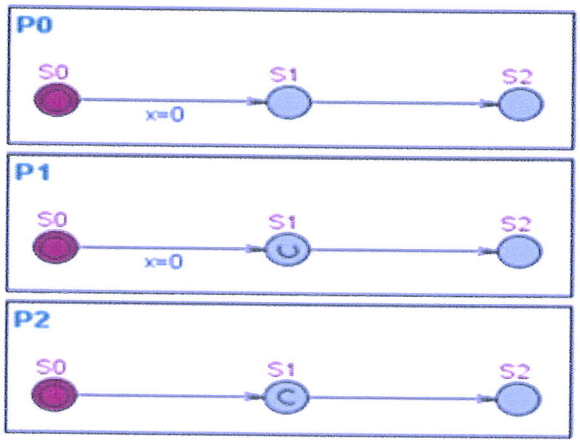

FIGURE 2.9 – Automate avec normal, urgent et committed états

Le temps ne peut pas progresser dans un état urgent, mais les transitions vers les états normaux sont autorisées.

Un état committed est plus restrictif : la seule transition possible est toujours celle qui sort de l'état committed.

➢ **Les canaux de synchronisation :**
Une étiquette de synchronisation a une des deux formes canal! ou bien canal?, ou canal! est du coté émission et canal? est de coté réception. Un canal est lancé lorsque sa transition canal! est effectuée. Les canaux de synchronisation sont utilisés pour permettre la communication entre les composants d'un même système [14].

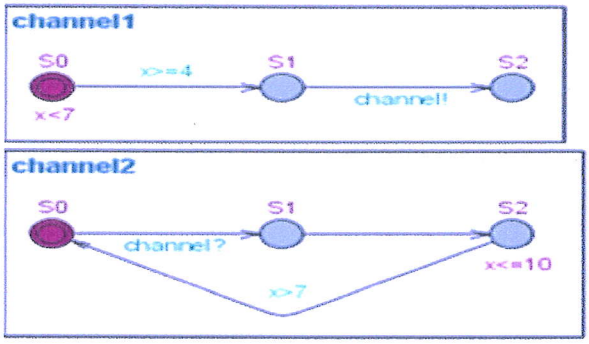

FIGURE 2.10 – Deux automates avec un canal de synchronisation

2.5 Conclusion

UPPAAL Model Checker est l'outil le plus pratique de vérification formelle. Par ailleurs, dans ce deuxième chapitre nous avons représenté la structure générale, les principales caractéristiques d'UPPAAL ainsi que son environnement graphique. Et enfin nous avons terminé par les fonctionnalités principales apportées par cet outil aux automates temporisés.

Chapitre 3

Système Autovision

3.1 Introduction

Dans les futurs systèmes automobiles, une aide à la conduite basée sur la vidéo vous aidera à améliorer la sécurité. Le traitement de la vidéo pour l'assistance du conducteur nécessite la mise en oeuvre en temps réel des algorithmes complexes. Basée sur le matériel disponible dans les environnements automobiles, une implémentation purement logicielle n'offre pas le traitement en temps réel nécessaire. Par conséquent l'accélération matérielle est nécessaire.

Mobileye ont présenté leur puce EyeQ avec deux coeurs ARM et quatre coprocesseurs dédiés pour la classification d'objets, le suivi et la reconnaissance de la voie. Des circuits matériels dédiés (ASIC) peuvent offrir le traitement en temps réel nécessaire, mais ils n'offrent pas la flexibilité nécessaire.

Dans ce chapitre nous allons représenter le concept Autovision suivi par le scénario de conduite typique, les coprocesseurs sont décrits plus en détail.

3.2 Le concept Autovision

Le concept Autovision est basé sur une séparation des opérations au niveau du pixel et le code d'application de haut niveau. Les opérations au niveau des pixels sont accélérées par des coprocesseurs, appelés aussi Moteurs d'accélération matérielle, à la différence du code d'application de haut niveau qui est implémenté entièrement programmable sur les noyaux standards du processeur PowerPC (PPC).

En outre, le code de l'application est capable de déclencher un processus de reconfiguration, d'où les coprocesseurs disponibles sur le système peuvent être reconfigurés dynamiquement, ce qui permet d'avoir un ensemble beaucoup plus vaste de fonctionnalités par rapport à aux celles sur un seul dispositif.

Ce processus utilise les capacités de reconfiguration partielle dynamique de XilinxVir-

tex FPGA, un Virtex-II Pro FPGA avec deux coeurs de processeur embarqués PowerPC (PPC) a été utilisé sur le conseil de développement XUPV2P de Digilent Inc. Sur un de ces coprocesseurs le code d'application de haut niveau pour le traitement d'image est implémenté. Quand il est nécessaire de reconfigurer un coprocesseur, ce PPC lance le processus de reconfiguration et notifie le deuxième PPC (sur lequel le logiciel de gestion de reconfiguration est encours d'exécution) que le processus de reconfiguration devrait être lancé. Ensuite, le premier CPU peut continuer à traiter les applications de niveau supérieur sans attendre que le processus de reconfiguration soit terminé [15].

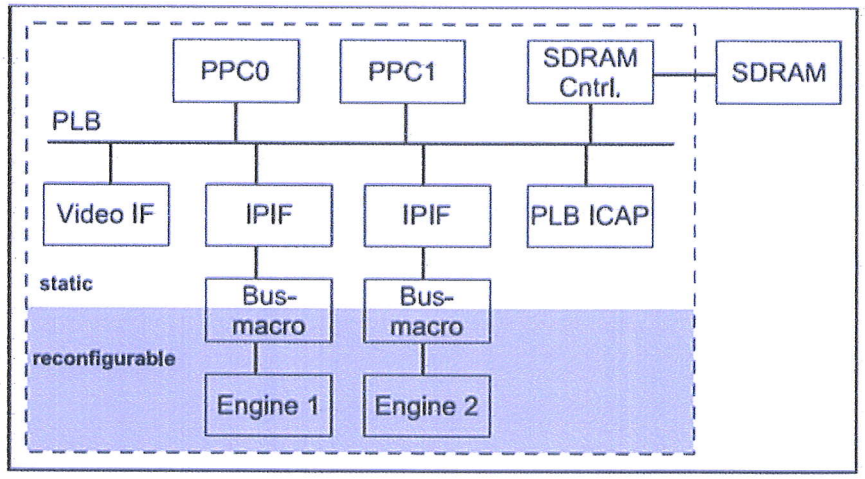

FIGURE 3.1 – Diagramme d'Autovision

Comme il est représenté sur la Figure 3.1 uniquement les moteurs sont reconfigurés sans leurs interfaces IP (IPIF) et les trames entrantes sont stockées dans la mémoire principale (SDRAM).

En utilisant le concept d'Autovision il est possible de traiter le flux vidéo d'entrée en temps réel, dont chaque image doit être traitée au sein de 40ms. La détection de point lumineux (feux de véhicules et de lumières de tunnel) à l'intérieur d'un tunnel nécessite 3.5ms en utilisant le Moteur Feu Arrière. Comparée à une solution purement logicielle qui fonctionne sur l'une des PPC avec 300MHz la solution d'accélération du matériel est 200 fois plus rapide.

Si des mesures supplémentaires doivent être effectuées, il pourrait ne pas être assez de temps pour effectuer le traitement d'une seule image en 40ms dans le cas d'une solution purement logicielle. En outre un système embarqué comme Autovision représente une alternative à faible coût par rapport à une implémentation logicielle pure sur un processeur de haute performance [15].

3.3 Scénario de conduite typique

Dans le système Autovision le scénario de conduite typique suivant est considéré :

Une voiture roule sur une journée ensoleillée sur l'autoroute. Les autres voitures peuvent être suivies par leur forme extérieure.

Cela se fait par un coprocesseur appelé ShapeEngine. Lorsque la voiture se rapproche d'un tunnel, l'entrée du tunnel est détectée et marquée comme une région d'intérêt (ROI) par le Moteur Tunnel (TunnelEngine). Dès que ce ROI est signalé une détection de bord et une amélioration du contraste sont effectuées. A l'intérieur du tunnel ou au moment de la nuit, les voitures sont détectées par leurs feux arrière, ceci est fait par un coprocesseur appelé TailLightEngine [15].

Environment	Shape Engine	Tunnel Engine	Contrast-/ EdgeEng.	Taillight Engine	PPC
Highway	x	x			x
Tunnelentrance		x	x		x
Tunnel inside				x	x

FIGURE 3.2 – Profil d'utilisation de coprocesseurs

Sur l'autoroute le ShapeEngine et TunnelEngine sont actifs. Si la voiture se rapproche d'un tunnel le ShapeEngine est remplacé par le Contrast-/EdgeEngine. Actuellement, le EdgeEngine et le ContrastEngine sont mises en oeuvre en tant que deux coprocesseurs distincts, alors que dans les versions futures, ils seront combinés [15].

3.4 Les moteurs d'accélération matérielle

Les moteurs d'accélération matérielle sont des coprocesseurs sur PLB. Ils peuvent accéder aux données de pixels stockées dans la mémoire principale et donc directement décharger les processeurs [15].

3.4.1 Le moteur d'adressage (AddressEngine)

L'adressage des pixels est un processus très répétitif et peut effectivement nécessiter plus de temps de traitement que le calcul de pixel lui-même. Donc il serait utile d'utiliser un coprocesseur matériel pour accélérer l'adressage des pixels.

Le moteur d'adressage garantit le chargement et le stockage corrects des données de pixels à partir et vers la mémoire principale. Comme il peut demander à lui-même les données de pixels, donc aucun des processeurs PPC est impliqué dans le transfert [15].

3.4.2 Moteur de détection du contour (EdgeEngine)

Le moteur de détection du contour doit détecter des bords horizontaux dans les trames vidéo indiquant les obstacles. Toutefois, le conducteur est intéressé seulement par ces obstacles qui sont dans sa voie.

Ainsi, en plus de ça une détection de voie est effectuée en utilisant une transformation de Hough [15].

La figure 3.3 montre le résultat de ce moteur obtenu immédiatement après l'entrée d'un tunnel .

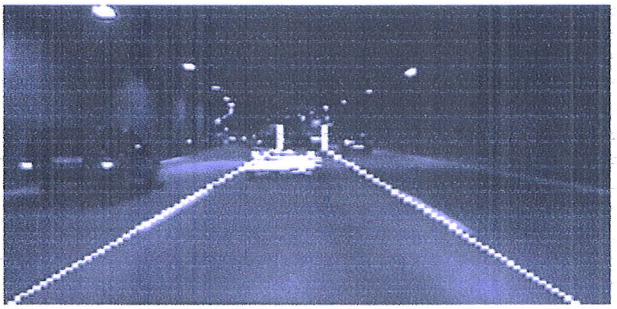

FIGURE 3.3 – Les bords horizontaux indiquant les obstacles

Ce moteur se compose de cinq parties qui sont toutes implémentées dans le matériel. Une première détection de bord est effectuée d'une détection de voie. Ensuite les lignes potentielles qui représentent la voie sont sélectionnées. Les deux marqueurs de voie sélectionnés entourent le ROI. Ils sont marqués dans l'image de sortie dans la quatrième étape. Dans la dernière étape les lignes horizontales apparaissent dans la ROI déterminé précédemment [15].

3.4.3 Moteur de contraste

Le moteur de contraste est conçu dans le but d'augmenter le contraste dans une sous-fenêtre prédéfinie de l'image d'entrée. Cette sous-fenêtre est généralement le ROI détecté par la reconnaissance de l'entrée du tunnel, qui est sombre et a un faible contraste par rapport à la zone extérieure. Le bruit provenant de la caméra sera augmenté lorsqu'une amélioration du contraste est effectuée. Ainsi, le bruit doit être réduit avant d'améliorer le contraste. En conclusion, le moteur de contraste effectue deux opérations sur la fenêtre du ROI : Réduction du bruit et amélioration du contraste [15].

La figure 3.4 montre l'image originale et la sortie résultante provenant du moteur de contraste.

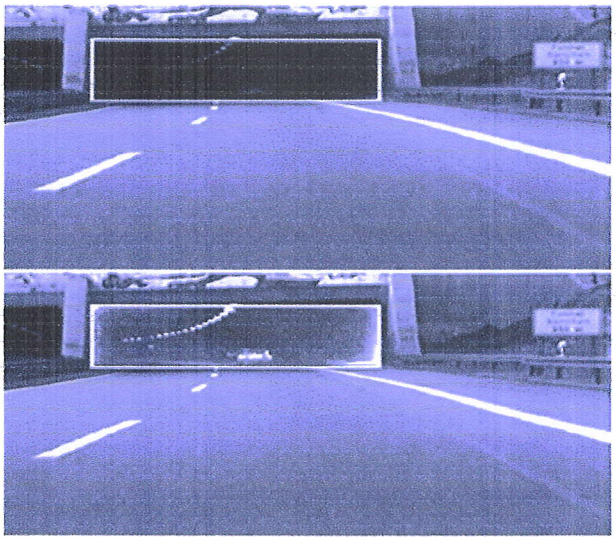

FIGURE 3.4 – Moteur de contraste : le roi marqué (en haut), et le résultat amélioré par le contraste (en bas)

Comme il est montré dans la figure 3.1 il est prévu de combiner le moteur de détection de contour et le moteur d'amélioration de contraste dans un seul coprocesseur.

3.4.4 Moteur de détection des feux arrière

Ce moteur essaie de détecter les voitures par leurs feux arrière ainsi que ses plaques d'immatriculation soit à l'intérieur d'un tunnel ou bien durant la nuit. Comme il peut détecter les feux statiques à l'intérieur du tunnel [15].

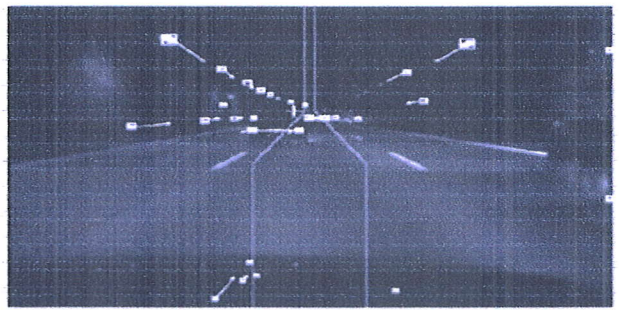

FIGURE 3.5 – TailightEngine : La détection des voitures et des feux statiques

3.4.5 La détection de l'entrée de tunnel

La reconnaissance de l'entrée du Tunnel n'est actuellement implémentée que dans le logiciel. Un propre algorithme a été implémenté pour détecter l'entrée du tunnel. Dans une version future un coprocesseur appelé Moteur de tunnel (TunnelEngine) doit détecter l'entrée du tunnel et la marquer comme un ROI [15].

FIGURE 3.6 – Détection de l'entrée de tunnel

3.4.6 La détection de la voie de Conduite (Driving Tube Detection)

La détection de voie est nécessaire pour plus d'un moteur. Cela conduit à l'idée d'avoir un coprocesseur séparé pour la détection de la limite de voie. Actuellement, le travail se centralise uniquement sur les voitures roulantes sur les routes.

Ainsi, les marqueurs de voie sont plutôt droits. Si la route est plus courbée ce que l'on a appelé la voie de conduite doit être détecté. Le moteur de tunnel et le moteur de détection de contours pourraient alors utiliser les informations délivrées par un coprocesseur appelé Moteur de détection de la voie de conduite [15].

3.5 La reconfiguration des coprocesseurs

La figure 3.6 donne une vue sur l'ensemble des composants de l'Autovision et leur interaction représenté par un diagramme de séquence :

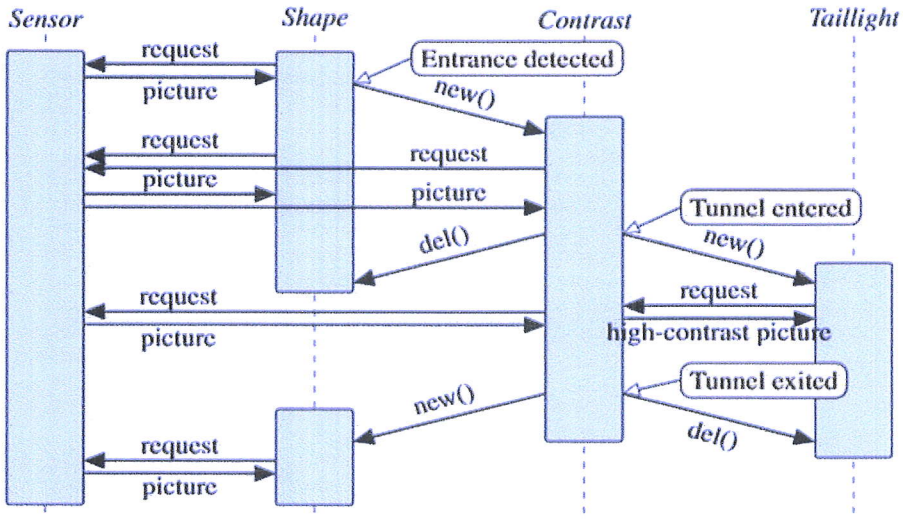

FIGURE 3.7 – Diagramme de séquence d'Autovision

Comme il est montré dans la figure, quand le composant Shape reçoit les photos envoyées par le capteur après une demande, il analyse les résultats afin de trouver les formes importants. Si la forme d'une entrée d'un tunnel est détectée le composant Contrast est invoqué. Ce dernier améliore et éclaire les photos provenant du capteur et lorsque le véhicule entre ou quitte le tunnel il active ou désactive les autre composants : Shape ou Taillight, si le véhicule est à l'intérieur de tunnel et la photo est sombre Contrast suspend le composant Shape et active le composant Taillaight, et quand la photo est éclatante ça veut dire que c'est la fin de tunnel Contrast désactive le composant Taillaight au moment qu'il active le composant Shape [14].

3.6 Conclusion

Dans ce chapitre, nous avons présenté le système Autovision. Nous avons bien expliqué son architecture et son principe basé sur la reconfiguration d'un ensemble de composants appelés moteurs d'accélération matérielle, dont nous avons décrit chaque composant et le rôle qu'il joue pour réaliser un bon fonctionnement du système. Enfin, nous avons donné une vue globale concernant la méthode de reconfiguration et de communication entre ces moteurs d'accélération.

Chapitre 4

Transformation des modèles

4.1 Introduction

Le génie logiciel s'est servi d'un ancien concept, les modèles, mais avec une nouvelle approche, l'ingénierie dirigée par les modèles (IDM). Celle-ci est basée sur les standards de l'OMG. Elle propose une architecture à quatre niveaux, avec les éléments suivants à chaque niveau : méta-méta-modèle, méta-modèle, modèle et information (c'est-à-dire l'implémentation du modèle). Pour atteindre la caractéristique «méta», il ne faut pas raisonner sur le contenu du modèle, c'est-à-dire sur ce qu'il faut modéliser dans le système. Il faut plutôt raisonner sur les concepts (et les relations entre ces concepts) nécessaires à la description des modèles d'un système.

4.2 L'Ingénierie dirigée par les modèles (IDM)

L'IDM est l'appellation générique pour une discipline particulière du génie logiciel qui regroupe plusieurs familles d'approches partageant des pratiques, des méthodes et des techniques communes, souvent implémentées au sein d'outils de développement.

L'émergence de l'IDM vise à se détacher de la machine et du code pour se concentrer sur des concepts plus généraux. Pour cela, l'IDM propose comme solution de regrouper des concepts au sein d'abstractions de système : les modèles.

4.2.1 Définitions autour du modèle

Puisque IDM est basé en principe sur la modélisation ainsi que la méta-modélisation, nous allons expliquer les trois notions de modèle, méta-modèle, méta-méta-modèle et bien sur les liens qui les unissent.

1. **Le modèle**
   Le modèle est une abstraction de la réalité et s'utilise pour représenter un système existant ou virtuel, naturel ou artificiel [5], de telle sorte qu'il soit suffisant pour

répondre à certaines questions en lieu de ce dernier [16].

2. **Le méta modèle**

En IDM le langage de modélisation, dans lequel est exprimé un modèle, est décrit par un méta-modèle. Le méta-modèle a la particularité de contenir tous les concepts nécessaires pour créer des modèles dans un domaine, un contexte particulier : le méta-modèle est au coeur de l'IDM [5].

Les deux notions de modèle et méta-modèle conduit à déduire deux types de relations illustrés dans la figure 4.1 :

✓ Le modèle représente un système.

✓ Le modèle doit être conforme à son méta-modèle.

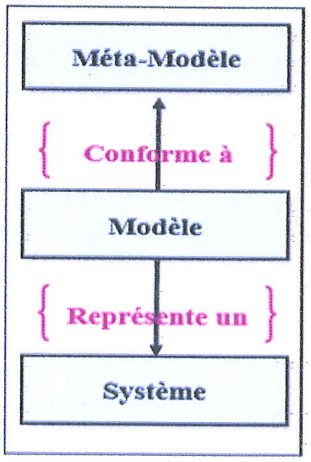

FIGURE 4.1 – Relation entre système, modèle et méta-modèle

En IDM tout est modèle, un méta-modèle est décrit selon un certain formalisme : le méta-méta-modèle.

3. **Le méta-méta-modèle**

Un méta-méta-modèle est un modèle qui décrit un langage de méta-modélisation, c-à-d les éléments de modélisation nécessaires à la définition des langages de modélisation. Il a de plus la capacité de se décrire lui-même modèle [16].
C'est sur ces principes que se base la modélisation à quatre niveaux généralement décrite sous une forme d'une pyramide (figure 3.1). La base de la pyramide représente le monde réel. Les modèles représentant cette réalité constituent le niveau M1. Les méta-modèles permettant la définition de ces modèles constituent le niveau M2. Enfin le méta-méta-modèle est représenté au sommet de la pyramide (niveau M3).

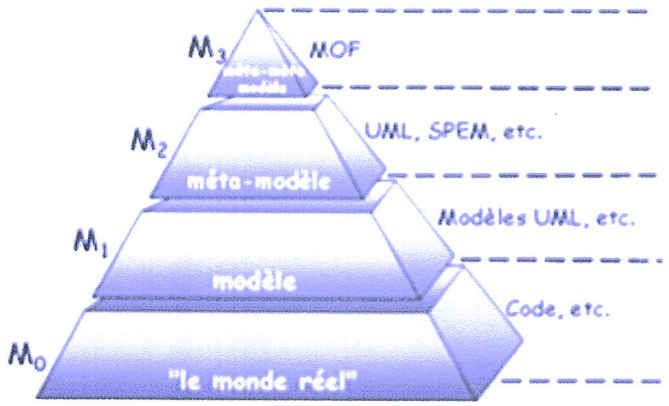

FIGURE 4.2 – La pyramide de modélisation à quatre niveaux

4.3 L'approche MDA

L'OMG a définit l'approche MDA en 2000 afin d'exploiter pleinement les avantages des modèles.

Le principe clé de MDA consiste en l'utilisation de modèles aux différentes phases du cycle de développement d'une application. Plus précisément, MDA préconise l'élaboration de modèles d'exigences (CIM), d'analyse et de conception (PIM) et de code (PSM) [17].

4.3.1 Les différents modèles de l'architecture MDA

1. **Le modèle d'exigences CIM**
   La première chose à faire lors de la construction d'une nouvelle application est bien entendu de spécifier les exigences du client. Un tel modèle doit représenter l'application dans son environnement afin de définir quels sont les services offerts par l'application et quelles sont les autres entités avec lesquelles elle interagit.
   Les modèles d'exigences peuvent même être considérés comme des éléments contractuels, destinés à servir de référence lorsqu'on voudra s'assurer qu'une application est conforme aux demandes du client .

2. **Le modèle d'analyse et de conception abstraite PIM**
   Une fois le modèle d'exigences réalisé, le travail d'analyse et de conception peut commencer, cette phase utilise aussi un modèle appelé PIM.
   Nous ne considérons ici que la conception abstraite, c'est-à-dire celle qui est réalisable sans aucune connaissance des techniques d'implémentation. Pour cela UML est préconisé par l'approche MDA comme étant le langage à utiliser pour réaliser des modèles d'analyse et de conception indépendants des plates-formes d'implémentation .

3. **Le modèle de code ou de conception concrète PSM**
   Une fois les modèles d'analyse et de conception réalisés, le travail de génération de code peut commencer. Cette phase, la plus délicate du MDA, elle doit aussi utiliser des modèles.
   MDA considère que le code d'une application peut être facilement obtenu à partir de modèles de code [17].

4.3.2 Architecture générale de l'approche MDA

La figure 3.2 donne une vue générale de l'approche MDA. La construction d'une nouvelle application commence par l'élaboration d'un ou de plusieurs modèles d'exigences (CIM). Elle se poursuit par l'élaboration des modèles d'analyse et de conception abstraite de l'application (PIM). Pour réaliser concrètement l'application, il faut ensuite construire des modèles spécifiques des plates-formes d'exécution. Les PSM n'ont pas pour vocation d'être pérennes. Leur principale fonction est de faciliter la génération de code. La génération de code à partir des modèles PSM d'ailleurs n'est pas réellement considérée par MDA. Celle-ci s'apparente plutôt à une traduction des PSM dans un formalisme textuel [17].

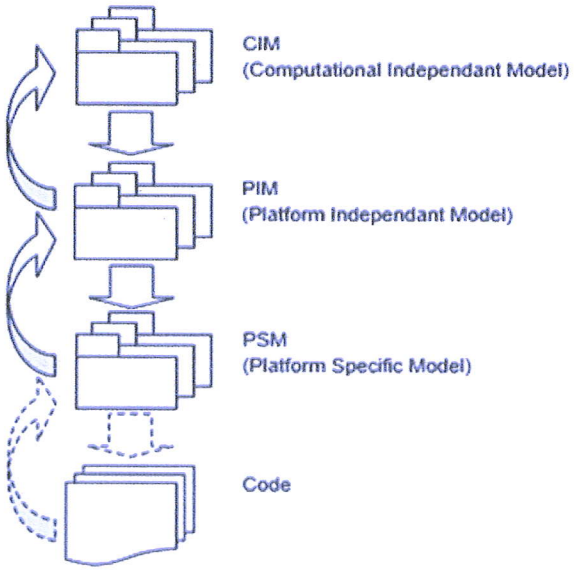

FIGURE 4.3 – Aperçu globale de l'approche MDA

4.4 Transformation de modèles

Une transformation prend un ou plusieurs modèles en entrée et génère un ou plusieurs modèles en sortie. Dans une transformation on distingue deux composantes : la première est la définition de la transformation, la seconde est l'outil de transformation.
Pour réaliser des transformations de modèles, les modèles doivent être exprimés dans un langage de modélisation défini à l'aide d'un méta-modèle [18].

Le niveau d'abstraction est un autre facteur important à prendre en considération dans les transformations. On distingue les transformations horizontales et les transformations verticales.

1. **La transformation verticale :** La source et la cible d'une transformation verticale n'ont pas le même niveau d'abstraction. Une transformation qui baisse le niveau d'abstraction est appelée un raffinement. Par contre, une transformation qui augmente le niveau est appelée une abstraction.

2. **La transformation horizontale :** Une transformation horizontale modifie la représentation source tout en conservant le même niveau d'abstraction. La transformation peut être l'ajout, la modification ou la suppression d'informations abstraction [19].

Si on revient à l'approche MDA on distingue différents transformations entre ses modèles :

- ✓ **Transformations CIM vers PIM :** Cette étape consiste à construire, partiellement, des modèles PIM à partir des CIM. Le but est de retranscrire les informations contenues dans les CIM vers les modèles PIM.
- ✓ **Transformations PIM vers PIM :** Cette étape s'exprime par l'enrichissement des modèles PIM c-à-d leur rajouter de l'information utile et à spécifier leur contenu. Les PIM sont ainsi raffinés et les informations qu'ils contiennent précisées.
- ✓ **Transformations PIM vers PSM :** Cette étape consiste à créer des modèles PSM à partir des informations fournies par les modèles PIM. Un PSM fournit les informations utiles à la génération du code de l'application et est dépendant de la plateforme d'exécution. Il est possible de créer autant de PSM qu'il y a de plateformes cibles.
- ✓ **Transformations PSM vers code :** La transformation des modèles PSM en code source consiste à générer le code source de l'application, de façon totale ou partielle, à partir des modèles PSM de l'application.

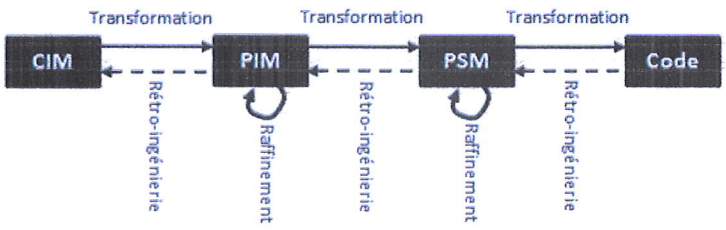

FIGURE 4.4 – Transformation des modèle MDA

4.4.1 Classification des approches de transformation

Selon [20], il existe deux grandes approches de classification de modèles : la transformation de type modèle vers modèle et la transformation de type modèle vers code.

4.4.1.1 La transformation de type « modèle vers code »

Elle est utilisée pour générer du code dans un langage de programmation (Java, C++, XML, HTML, etc.) particulier à partir d'un modèle source. On distingue deux approches de transformations de type modèle vers code : les approches basées sur le principe du visiteur ou celles basées sur le principe des patrons.

1. Les approches basées sur le principe du **visiteur** : transforment le modèle en entrée vers un code écrit dans un langage de programmation en sortie. La différence de sémantique entre le modèle et le langage cible est réduite par l'ajout de visiteurs au modèle d'entrée. Le modèle enrichi par les visiteurs est parcouru pour obtenir Le code cible et créer ainsi un flux de texte en sortie.
2. Les approches basées sur l'utilisation de **templates** : Dans ces approches, un template est un fragment de texte (du code) cible contenant des bouts de métacodes qui permettent :
   (a) d'accéder aux informations du modèle source,
   (b) de sélectionner du texte (code),
   (c) de réaliser des expansions de manière itérative.

4.4.1.2 La transformation de type « modèle vers modèle »

Les transformations de type modèle vers modèle sont aujourd'hui plus maîtrisées, en effet, elles ont beaucoup évolué au cours de ces dernières années, et plus particulièrement depuis l'apparition du MDA. Dans la démarche de l'IDM, la transformation de «modèle à modèle» est utilisée pour réaliser une activité de génération de modèles qui peut être automatisée [18].

4.5 La transformation de graphes

Les modèles et méta-modèles possèdent souvent une représentation graphique apparentée à un graphe. Les techniques de réécriture de graphes et de transformation de graphes peuvent être appliquées pour des transformations de modèles. D'une manière générale, les systèmes de réécriture de graphes combinent une notation graphique et une notation textuelle afin d'exprimer ces transformations. Un programme de transformation essentiellement composé de règles de réécriture va d'abord sélectionner un fragment d'un graphe source identifié grâce à un langage de navigation. L'application d'un filtre sur ce fragment sélectionné permet de le modifier avant de le recopier dans le graphe cible [19].

4.5.1 Grammaire de graphes

La transformation de graphe est spécifiée sous forme d'un modèle de grammaires de graphes. Elles sont composées de règles dont chacune est composée d'un graphe de côté

gauche (LHS) et d'un graphe de côté droit (RHS).
Une grammaire de graphe GG = (P,S,T) est un ensemble P de règles muni d'un graphe initial S et d'un ensemble T de symboles terminaux [18].

4.5.2 Le principe de règles

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

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

4.5.3 Application de règles

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

1. Choisir une occurrence du graphe de côté gauche L dans G.
2. Vérifier les conditions d'application d'après cond.
3. Retirer l'occurrence de L (jusqu'à K) de G ainsi que les arcs pendillés, c-à-d tous les arcs qui ont perdu leurs sources et/ou leurs destinations. Ce qui fournit le graphe de contexte D de L qui a laissé une occurrence de K.
4. Coller le graphe de contexte D et le graphe de côté droit R suivant l'occurrence de K dans D et dans R. c'est la construction de l'union de disjonction de D et R et, pour chaque point dans K, identifier le point correspondant dans D avec le point correspondant dans R.
5. Enfoncer le graphe du côté droit dans le graphe de contexte de L suivant la relation d'enfoncement emb :
   La dérivation directe depuis G vers H à travers r dénotée par G $\Rightarrow$ H, est l'application de la règle r sur un graphe G pour fournir un graphe H.

4.5.4 Système de transformation de graphe

Un système de transformation de graphe (figure 4.5) se défini comme un système de réécriture de graphes qui applique les règles de la Grammaire de Graphes sur son graphe initial jusqu'à ce que plus aucune règle ne soit applicable [18].

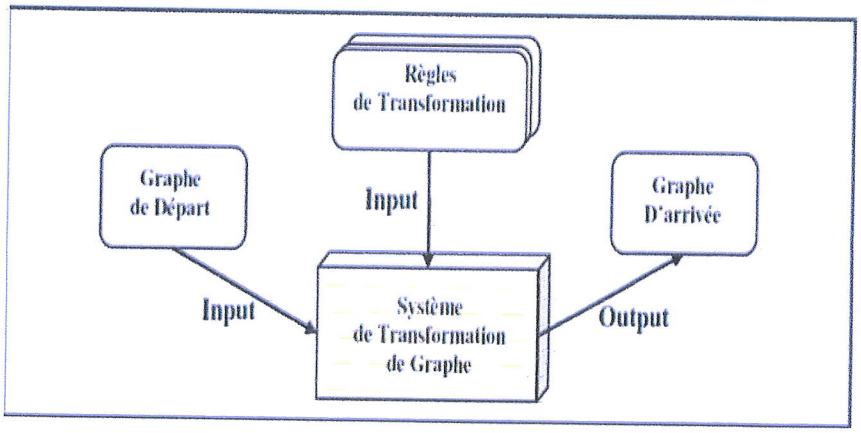

FIGURE 4.5 – Système de transformation de graphe

4.6 ATOM3

AToM3 est un outil de modélisation développé par le laboratoire MSDL à l'université de McGill Montréal, Canada. Cet outil a été conçu en collaboration avec Juan de Lara de Universidad Autónoma de Madrid (UAM), Espagne.
Les deux principales fonctionnalités d'AToM3 sont la méta-modélisation et la transformation de modèles. La méta-modélisation est la description ou la modélisation de différents types de formalismes. La transformation de modèles est une technique consistant à transformer, traduire ou modéliser automatiquement un modèle décrit dans un certain formalisme, vers un autre modèle qui peut être décrit soit dans le même formalisme soit dans un autre.
Dans AToM3 les formalismes et les modèles sont décrits comme des graphes. Cet environnement génère un outil de manipulation graphique des modèles décrits dans un formalisme donné. Les transformations entre modèles sont ensuite effectuées par réécriture des graphes et peuvent être décrites comme ces derniers [19].

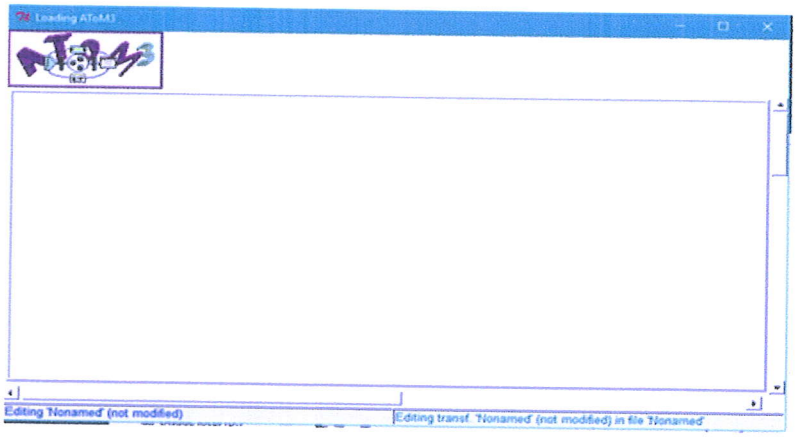

FIGURE 4.6 – Interface d'ATOM3

4.7 Conclusion

Dans ce chapitre nous avons définit en détail les principaux concepts de domaine de l'ingénierie de modèles. Dans le cadre de l'approche IDM, nous avons mis en évidence les différentes techniques de transformations de modèles, par la suite nous avons présenté les concepts fondamentaux de la transformation des graphes. Et enfin, nous avons terminé par présenter AtoM3, l'un des outils les plus utilisés pour la modélisation et la transformation de modèle et celui utilisé dans le cadre de ce mémoire.

Chapitre 5

Modélisation et vérification des système dynamiquement reconfigurable

5.1 Introduction

Les deux modèles RecDEVS et Uppaal ont une structure similaire. Ils sont basés sur la notion des évènements chronométrés. Cette similarité est nécessaire pour permettre un processus de transformation entre les deux modèles.
Dans ce chapitre et basant sur cette transformation, nous avons utilisé l'outil Atom3 pour développer une application complexe décrivant la reconfiguration de système Autovision vu dans le troisième chapitre. On a bien décrit son comportement par un modèle RecDEVS afin de le simuler et de vérifier certaines propriétés de ce système utilisant le model checker Uppaal. Dans ce but nous avons suivi une succession d'étapes.

5.2 Approche proposée

Notre approche est basée sur l'amélioration d'un méta modèle RecDevs décrit en Atom3 [21] par l'ajout d'un troisième type de transition « transition confluence » expliquée dans le premier chapitre (DEVS parallèle).
Nous avons créé un modèle RecDevs de système Autovision. Comme nous avons utilisé une grammaire de graphes proposée dans [21] pour transformer le modèle RecDevs en un fichier texte xml valable pour l'outil UPPAAL afin de le vérifier.

5.2.1 Le méta-modèle de RecDEVS

Comme nous avons noté, le méta-modèle servi (figure 5.1) est composé des classes et des associations suivantes :

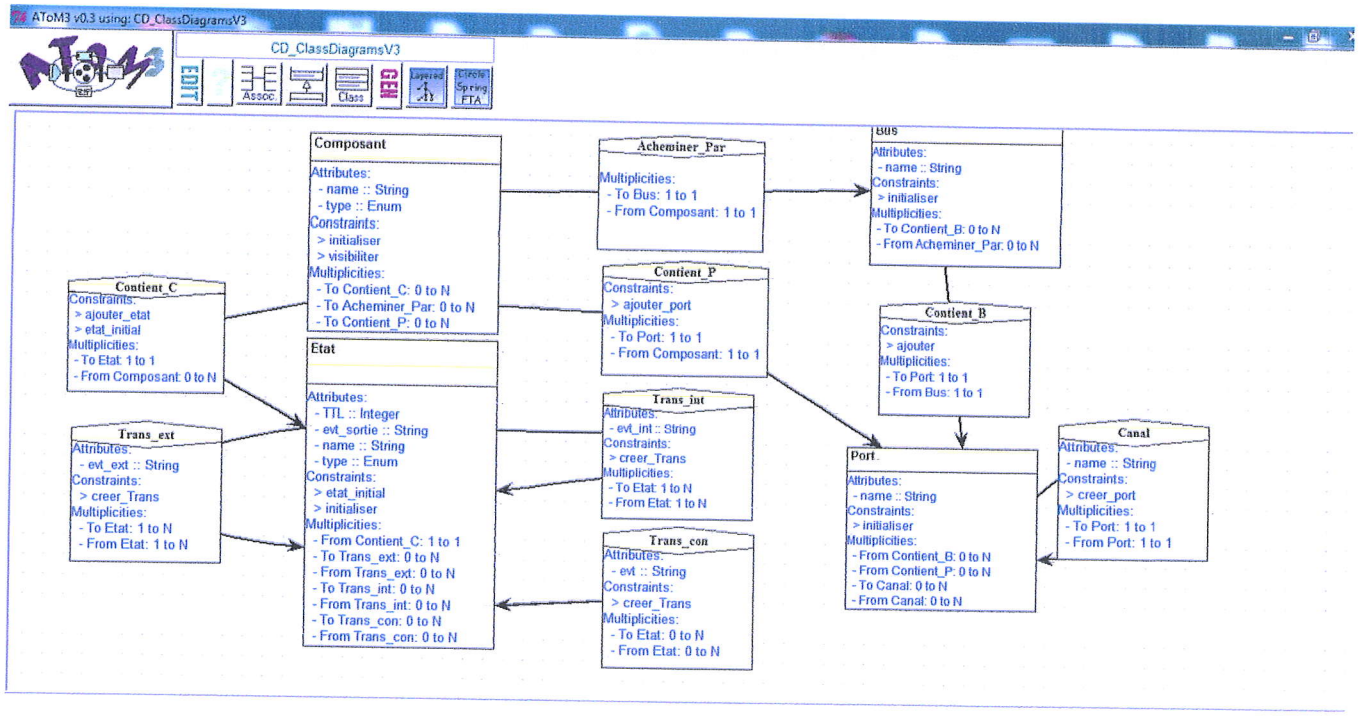

FIGURE 5.1 – méta-modèle RecDevs

1. **Les classes**

   (a) La classe **Composant** : dans ce méta-modèle chaque composant représente un modèle atomique a un nom et un type (permanant ou reconfigurable), ainsi deux méthode 'initialiser','visibiliter' permettent l'initialisation des variables et le type de composant.

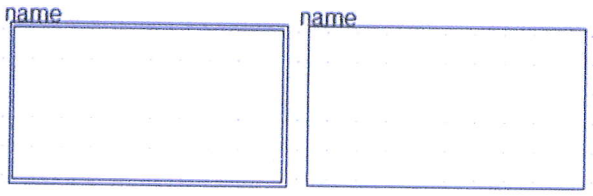

FIGURE 5.2 – Apparence graphique d'un composant permanant (partie gauche) ou reconfigurable (partie droite)

   (b) La classe **Bus** (circuit de message) :La classe Bus possède un seul attribut : 'name' de type string qui détermine le nom de bus, et une seule méthode : 'initialiser' permet d'initialiser les variables. Cette classe et représentée comme suit :

   (c) La classe **Etat** (l'état actuel de modèle atomique) : Concernant cette classe, elle possède l'attribut 'name' pour déterminer le nom d'objet de type état. Et l'attribut TTL qui détermine la durée de vie d'état. Elle possède aussi l'attribut

(d) Un Bus peut contenir un ensemble de ports donc une association « Contient-p » relie les deux classes « Bus » et « ports »

(e) « Canal » : cette association relie deux classes « Port ».

(f) Transition-interne, transition-externe et transition-confluence : ces trois associations relient deux classes « Etat ». Elles possèdent une seule contrainte : 'creer-etat' qui permet d'interdire la relation si un ou deux états n'appartiennent à aucun composant, et aussi d'interdire la relation entre deux états qui n'appartiennent pas au même composant.

A partir de ce méta-modèle il est possible de générer automatiquement l'outil de modélisation de RecDEVS qui apparait dans la figure ci-dessus :

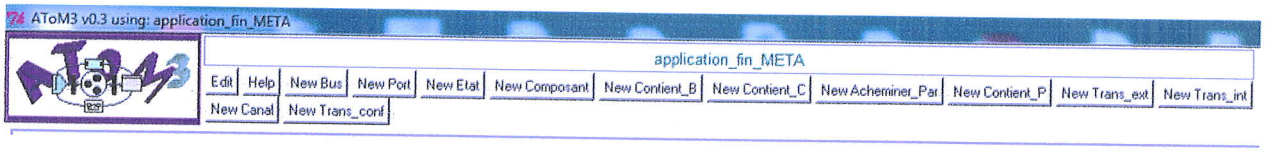

FIGURE 5.6 – L'outil de modélisation de RecDEVS

5.2.2 La grammaire de transformation

Nous avons proposé une grammaire nommée « grammaire » composée d'une action initiale, dix (10) règles de transformation, et d'une action finale.

1. **L'action initiale :**
   Les traitements réalisés dans l'action initiale de cette grammaire sont :

   ✓ Création d'un fichier texte vide nommé « recdevS »

   ✓ Création des variables globales nécessaires.

2. **Les règles :**

   ✓ Règle 1 : Permet de transformer un composant vers une Template.

   ✓ Règle 2 : Permet de transformer un état vers une location.

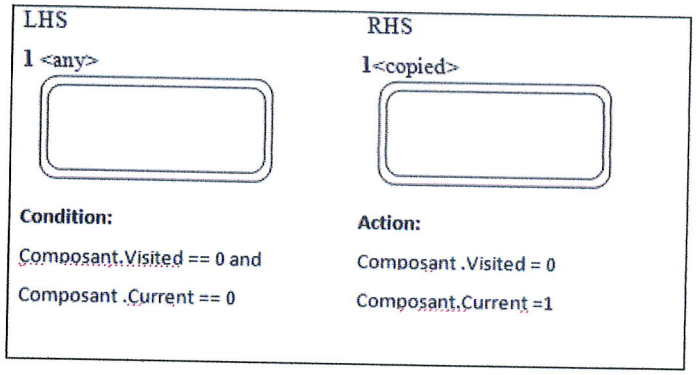

FIGURE 5.7 – Règle 1 « ComposantVersTemplate »

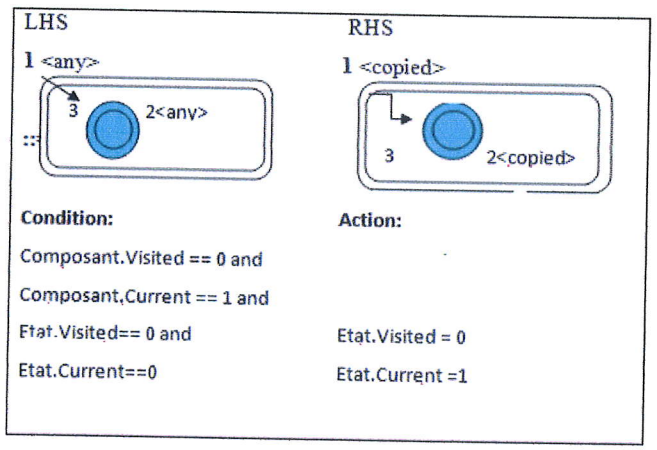

FIGURE 5.8 – Règle 2 « EtatVersLocation »

✓ Règle 3 : Appliquée pour localiser deux états reliés par un arc interne, dont l'état source est en cours de traitement et l'état destination est non encore traité, et générer le code correspondant.

✓ Règle 4 : Appliquée pour localiser deux états reliés par un arc externe, dont l'état source est en cours de traitement et l'état destination est non encore traité, et générer le code correspondant.

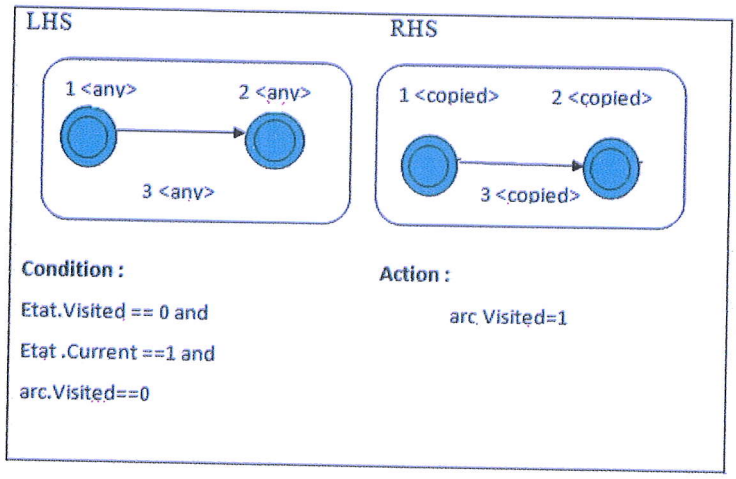

FIGURE 5.9 – Règle 3 «TransitionIntVersTransition »

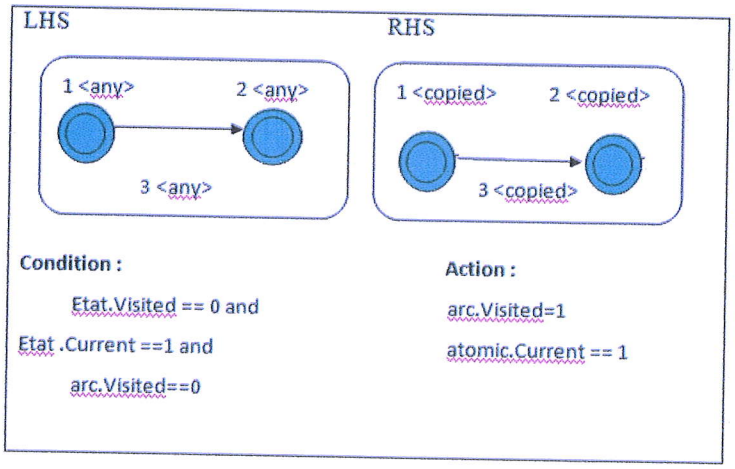

FIGURE 5.10 – Règle 4 « TransitionExtVersTransition »

✓ Règle 5 : Appliquée pour localiser deux états reliés par une transition confluence, dont l'état source est en cours de traitement et l'état destination est non encore traité, et générer le code correspondant.

✓ Règle 6 : Appliquée pour localiser un état en cours de traitement relié par un arc interne non encore traité, et générer le code correspondant.

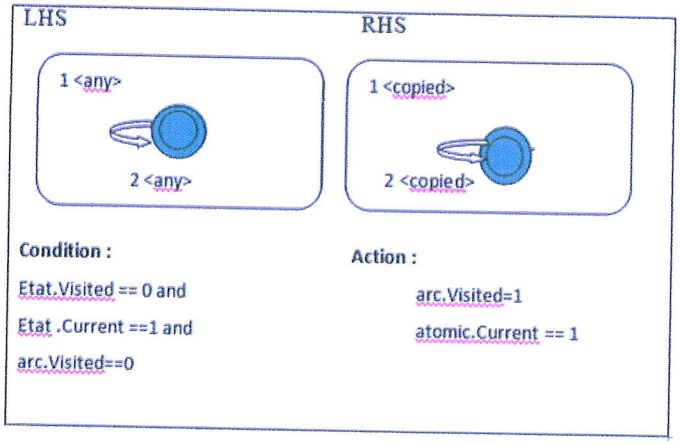

FIGURE 5.12 – Règle 6 « SelfTransitionInt »

✓ Règle7 : Appliquée pour localiser un état en cours de traitement relié par un arc externe non encore traité, et générer le code correspondant.

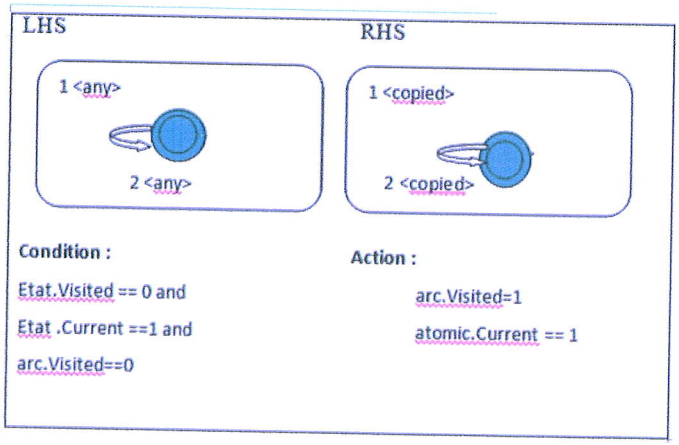

FIGURE 5.13 – Règle 7 « SelfTransitionExt »

✓ Règle 8 : Appliquée pour localiser un état en cours de traitement relié par une transition confluence non encore traitée, et générer le code correspondant.

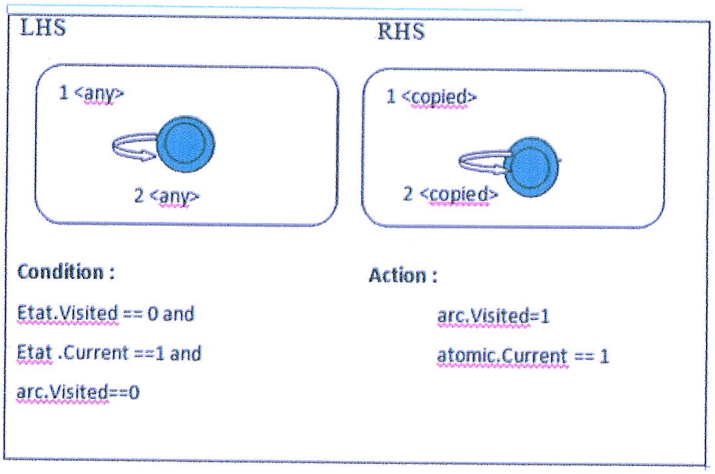

FIGURE 5.13 – Règle 7 « SelfTransitionExt »

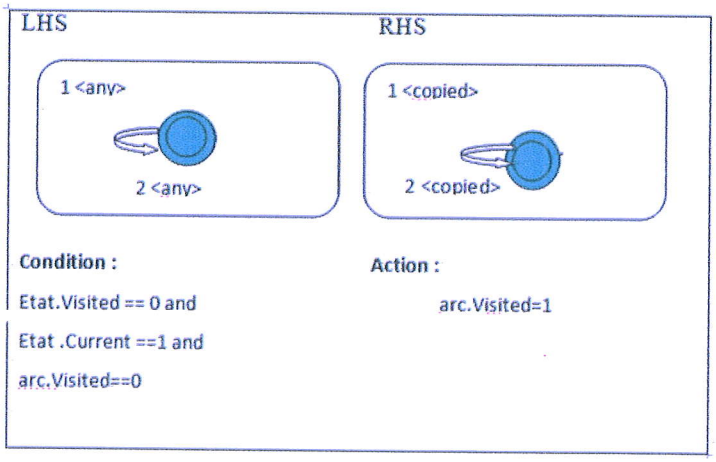

FIGURE 5.14 – Règle 8 « SelfTransitionConf »

✓ Règle 9 : la règle « FinEtat » est appliquée pour interdire la visite de cet etat une autre fois après le traitement de tous les états.

✓ Règle 10 : la règle FinComposant est utilisée pour interdire la visite d'un composant autre fois après le traitement de tous les composants.

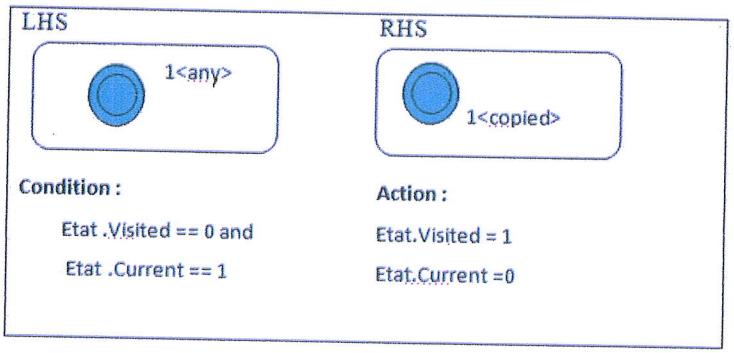

FIGURE 5.15 – Règle 9 « FinEtat »

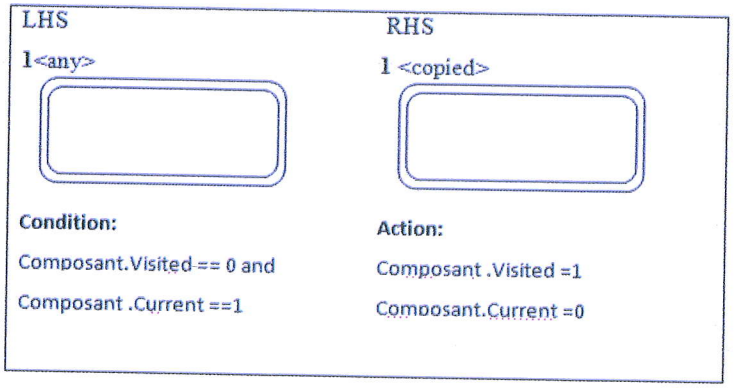

FIGURE 5.16 – Règle 10 « FinComposant »

3. **L'action finale**

   A la fin d'exécution des règles, et dans l'action finale de cette grammaire nous détruisons les attributs temporaires des éléments du modèle et nous fermons le fichier « recdevs ».

5.2.3 Fichier UPPAAL

Un modèle UPPAAL est sauvegardé sous forme d'un fichier ".xml ". La figure suivante représente la structure globale de ce fichier :

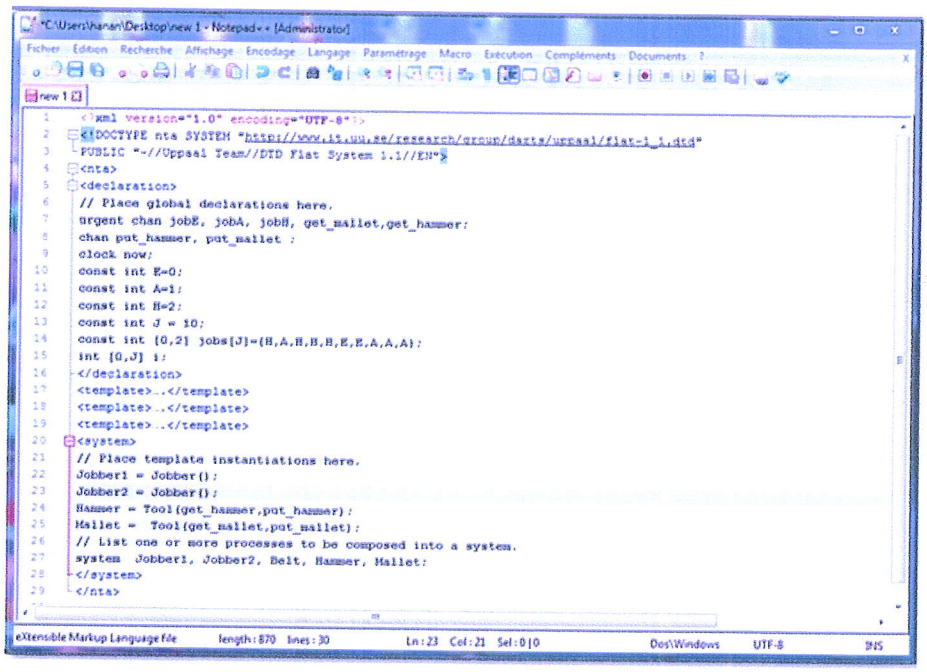

FIGURE 5.17 – Structure globale d'un fichier « xml »

5.3 Étude de cas : Système Autovision

Nous allons traiter dans cette étude de cas le fonctionnement de système Autovision,ou nous avons créé un modèle RecDEVS constitué de tous les composants du système qui interagissent entre eux par des messages pour assurer la reconfiguration dynamique. Nous avons suivi les étapes suivantes :

5.3.0.1 La modélisation

La figure 5.18 représente l'aperçu global de l'ensemble de composants :
Maintenant on va expliquer le fonctionnement de chaque composant d'une façon séparée :

1. **Executive-Cx :**
   Comme on a vu dans le chapitre 1 cet exécutif est le composant responsable sur la reconfiguration dynamique de système qui est basée sur deux messages new et delete. Il est toujours actif donc nous avons l'attribué un type permanent. Il a pour rôle de gérer la communication entre les autres composants. L'état initial de ce composant est Start-Cx, lorsqu'un composant envoie un des messages shape-Cx-cont ?, cont-Cx-tail ? ou bien cont-Cx-shape ? Il change son état pour créer l'un des composants Contrast, Tailight ou Shape puis il revient à son état initial après l'émission de message de création (Cx-cont !, Cx-tail !, Cx-shape !). Et quand il reçoit un message tail-Cx ?, shape-Cx ? ou cont-Cx ? il fait une transition vers l'état delete pour désactiver le composant qui a demandé sa suppression.

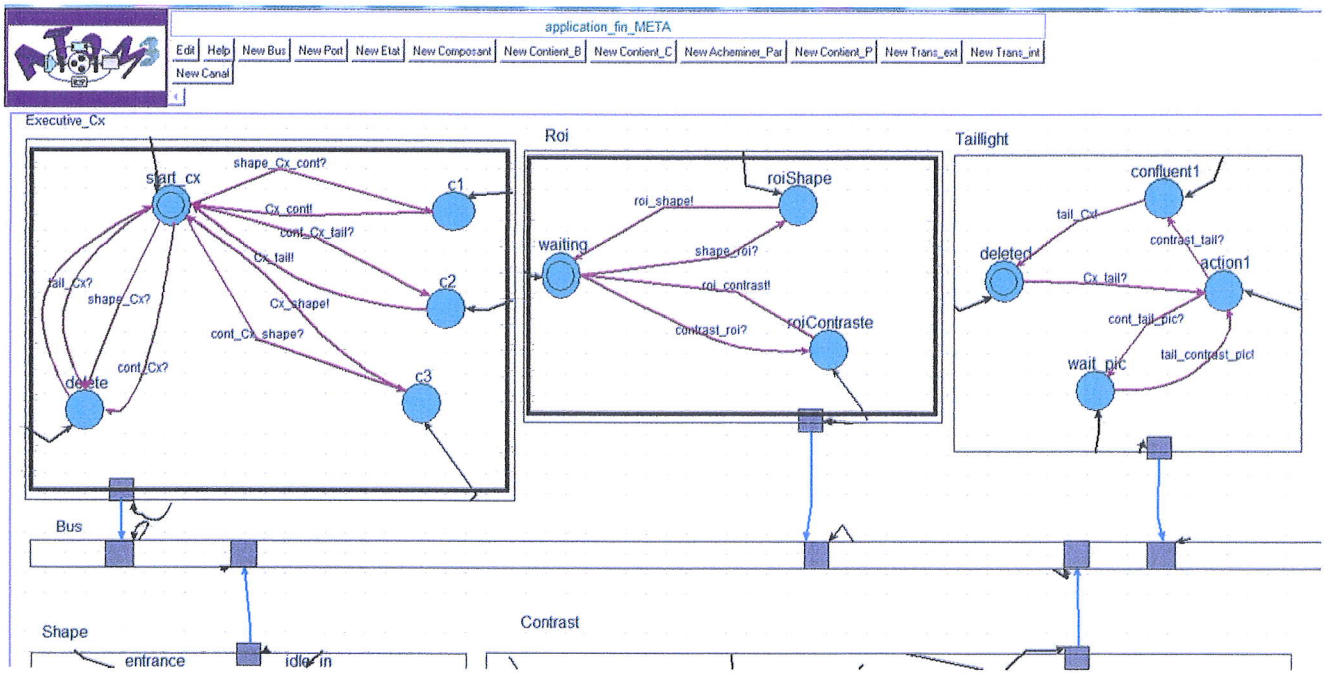

FIGURE 5.18 – Aperçu globale des composants d'Autovision

2. **Roi :**
   Le rôle de ce composant permanent est d'envoyer des images sur la route aux deux autres composants shape et contrast, donc en principe il est toujours en attente des demandes des photos au niveau de son état initial « waiting », au moment qu'il reçoit un message externe shape-roi ? ou contrast-roi ? il change son état, il repend par un message sortant roi-shape ! ouroi-contrast ! envoyé au composant demandeur et il retourne à l'état d'attente.

3. **Shape :**
   Le composant Shape est l'un des composants reconfigurables de système Autovision. Lors de démarrage du système, il est dans l'état « idle-out » c.-à-d. à l'extérieur de tunnel, il effectue une requête d'image (shape-roi !) et lors de la réception de l'image (roi-shape !) il analyse le résultat. S'il ne s'agit pas de l'entrée de tunnel, il retourne à nouveau à son état initial. Sinon il envoie un message à l'exécutif Cx pour demander l'activation du composant Contrast et il reste dans l'état « idle-in » jusqu'à qu'il recevoir un message de suppression (contrast-shape ?), il demande son désactivation (shape-Cx !). Et quand la voiture sort de tunnel ce composant est invoqué (Cx-shape !) pour recommencer son travail.

4. **Contrast :**
   Le contrast aussi est un composant reconfigurable, il est désactivé (l'état « deleted2 »), il sera activé par la transition externe Cx-cont ? Il communique avec le composant Roivia des requêtes de demande et de réception d'images (contrast-roi ! et roi-contrast ?). Le rôle de Contrast est de détecter l'entrée et la sortie de tun-

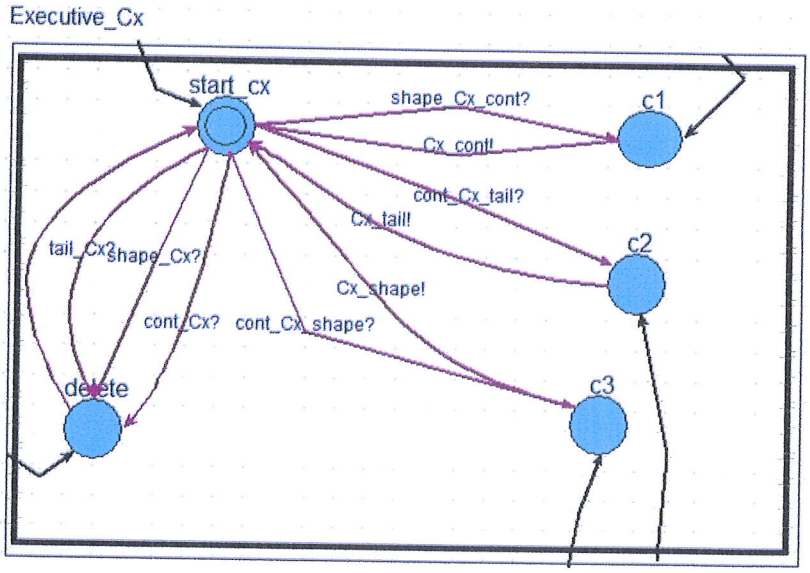

FIGURE 5.19 – Aperçu graphique du composant Executive-Cx

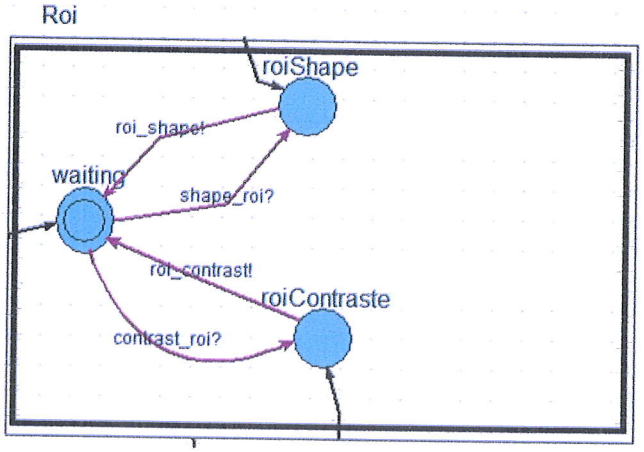

FIGURE 5.20 – Aperçu graphique de composant Roi

nel. Dans le cas de l'entrée, il suspend le Shape(contrast-shape!) puis il émet un message d'activation de composant Tailight (cont-Cx-tail!). Dans l'autre cas (sortie de tunnel), il désactive le Tailight,il active le Shape, et il termine par demander sa suppression. Le contrast peut également repend à une requête d'amélioration des photos envoyé par le Tailight(tail-contrast-pic ?).

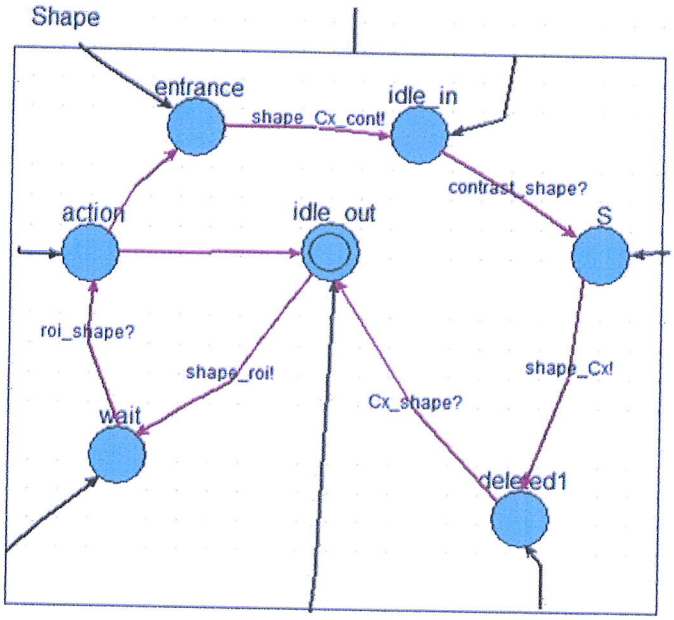

FIGURE 5.21 – Composant Shape

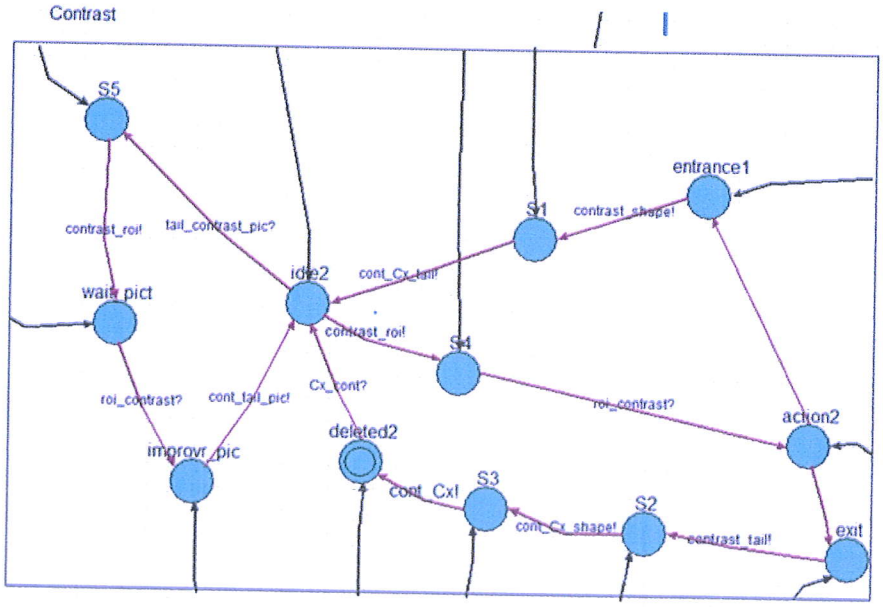

FIGURE 5.22 – Composant Contrast

5. **Taillight :**

   Ce dernier composant fonctionne lorsque la voiture est à l'intérieur de tunnel, il sera activé par Cx (Cx-tail ?), il fait des demandes des images éclairées par le Contrast (tail-contrast-pic !) pour les analyser. On cas ou le Contrast détecte la sortie de tunnel il désactive le taillight (contrast-tail ?).

Après la modélisation de système Autovision, la simulation ainsi que la vérification formelle de notre système devient une étape fondamentale et très importante pour s'assurer sa fiabilité.

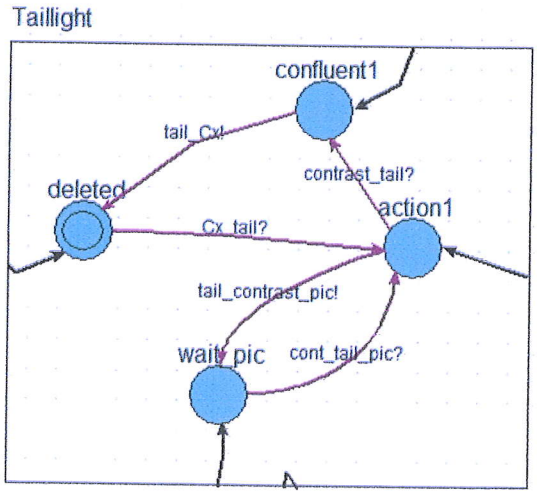

FIGURE 5.23 – Composant Taillight

La figure suivante représente le chargement de la grammaire de transformation qui possède un ensemble de règles précédent.

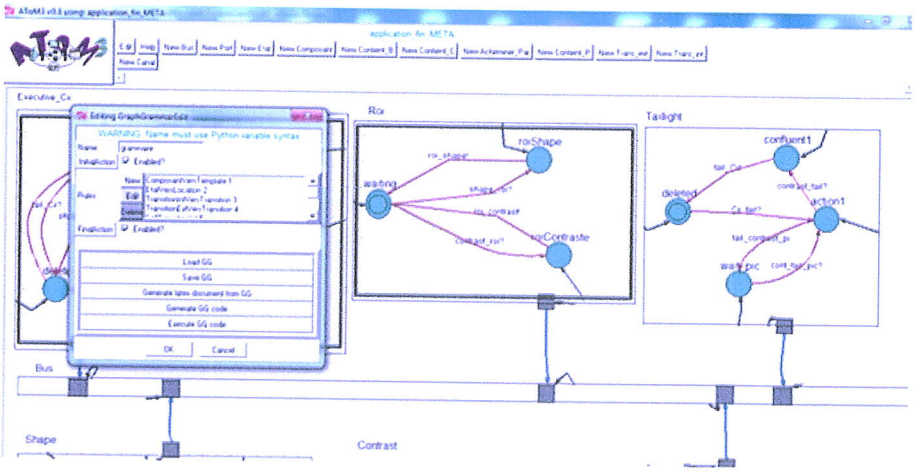

FIGURE 5.24 – Chargement de la grammaire.

La figure suivante représente l'importation de la grammaire de transformation qui possède un ensemble de règles précédent.

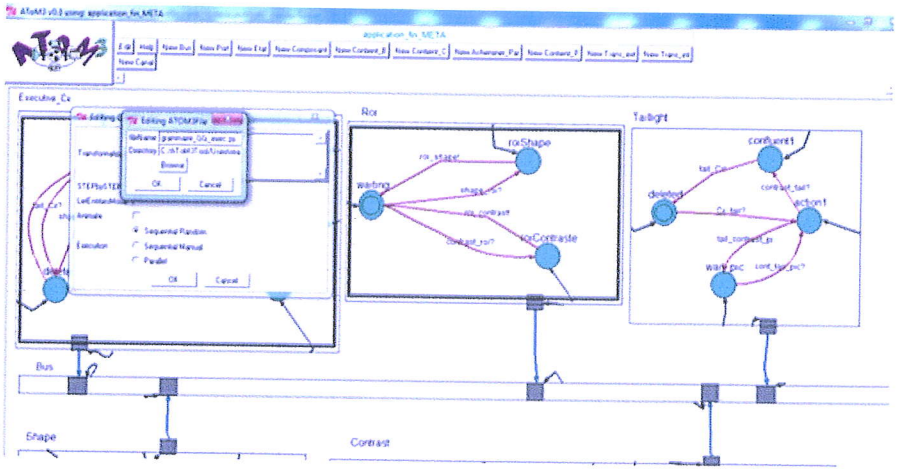

FIGURE 5.25 – Importation de la grammaire.

L'exécution de la grammaire de graphe sur cet exemple est présentée dans la Figure 5.26

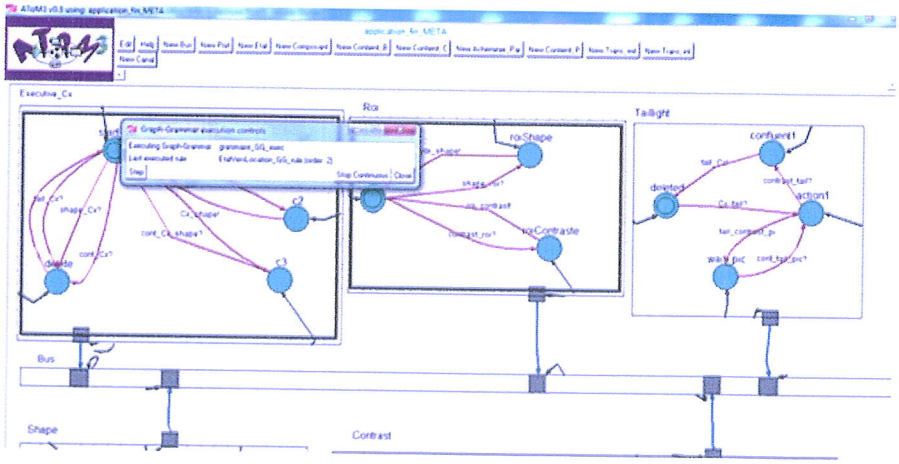

FIGURE 5.26 – Exécution de la grammaire

Finalement, On obtient le code UPPAAL équivalent au modèle RecDEVS présenté dans la Figure 5.27

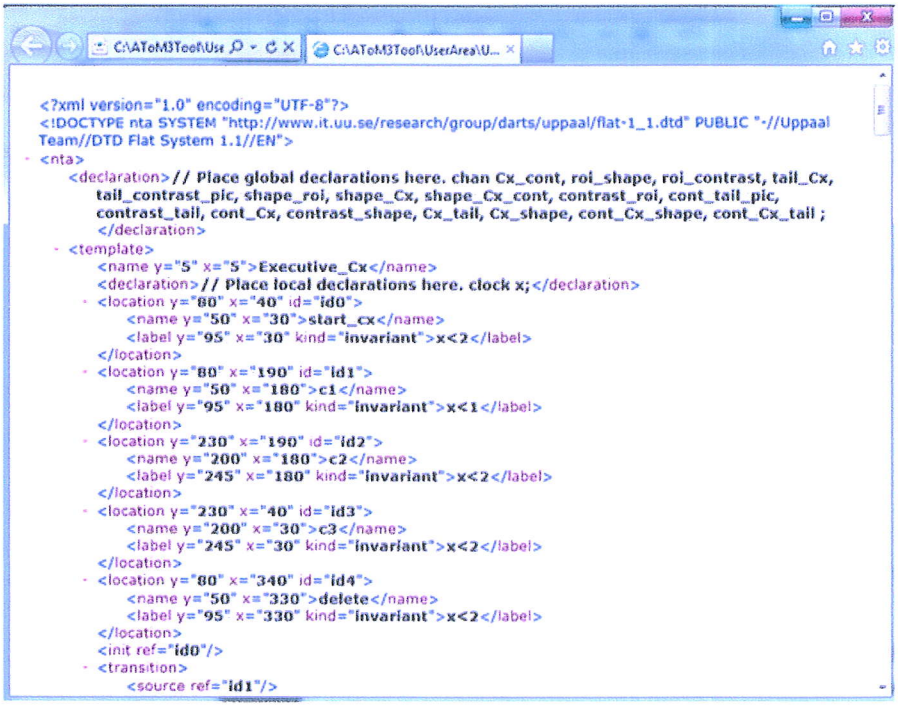

FIGURE 5.27 – Code UPPAAL équivalent au modèle RecDEVS d'Autovision

5.3.1 La simulation

Après l'ouverture de fichier xml par l'outil UPPAAL, le simulateur fait apparaitre les résultats suivants :

5.3.2 La vérification des propriétés

En ce qui concerne la reconfiguration nous avons identifié plusieurs propriétés de vérification à appliquer dans le système Autovision

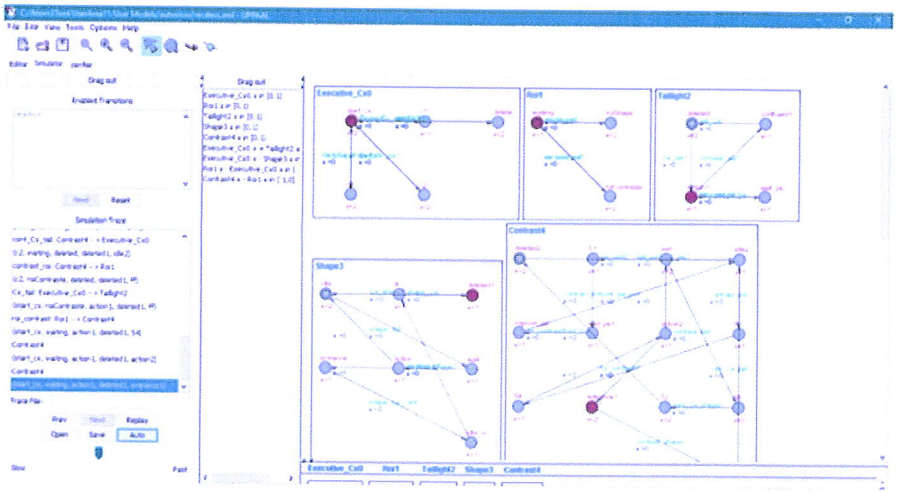

FIGURE 5.28 – Le simulateur UPPAAL (les transitions)

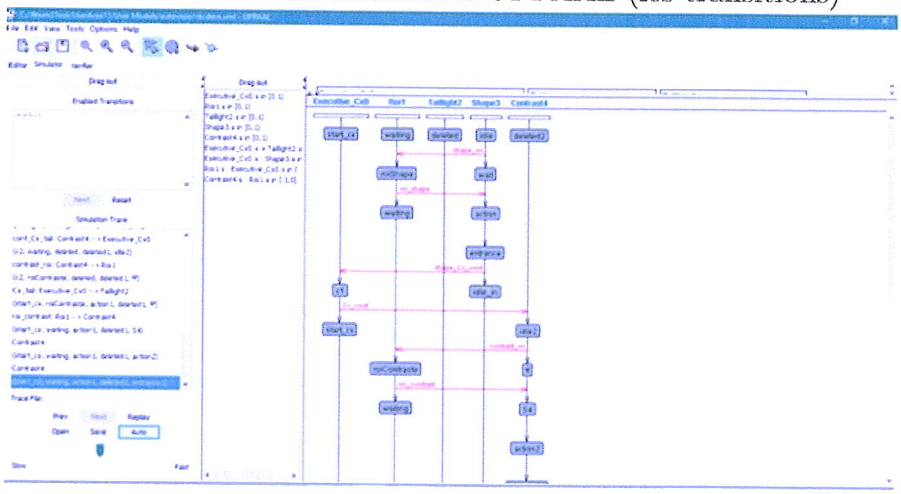

FIGURE 5.29 – Le simulateur UPPAAL (les traces de simulation)

1. **Propriété 1 :** E< >Contrast4 .idle2 imply Contrast4.deleted2
   Cette propriété vérifie que la transition entre Contrast.idle2 et Contrast.deleted est utilisé.
2. **Propriété 2 :** A[ ]not deadlock
   Cette propriété examine que le système n'arrive pas à un cas d'interblocage.
3. **Propriété 3 :** E< >Shape3 .idle-out imply Shape3.wait
   Le rôle de cette propriété est la vérification que le shape effectue des demandes d'images.
4. **Propriété 4 :** E< >Contrast4 .S4 imply Contrast4.idle2
   Cette propriété vérifie que le Contrast demande des images.
5. **Propriété 5 :** E< > (Shape3 .idle-out and Contrast4.deleted2)
   Cette propriété vérifie que cet état est accessible.

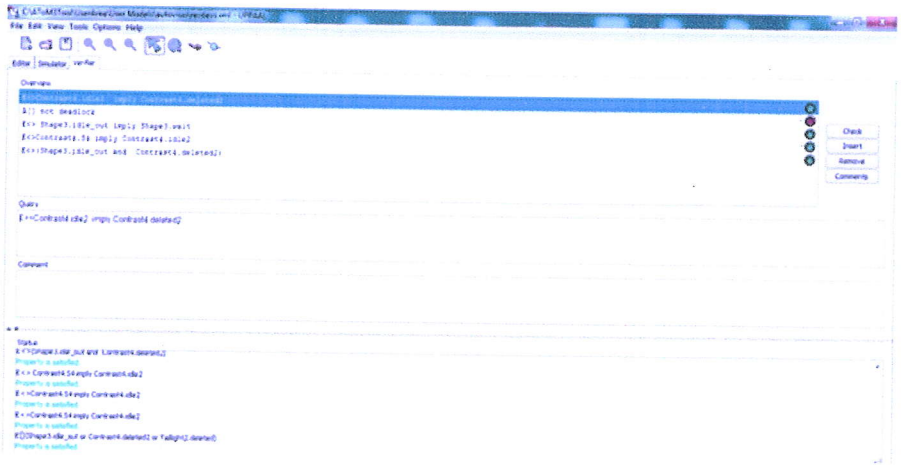

FIGURE 5.30 – Vérificateur UPPAAL

5.4 Conclusion

Dans ce chapitre nous avons présenté une approche de modélisation de système temps réel d'Autovision à l'aide de modèle RecDevs afin de le transformer à une représentation équivalente à l'outil de vérification de modèle UPPAAL, ce dernier nous a permet de tester la validité de certains propriétés de sureté et vivacité de système.

Conclusion générale

Ce mémoire rentre dans le cadre de la modélisation et de la simulation basées sur Rec-DEVS qui offre aujourd'hui une alternative très intéressante quant à la représentation des systèmes temps réel dynamiquement reconfigurable où la notion de vérification est toujours présente. Dont on a bien appliqué ces techniques sur l'exemple d'Autovision.
Nous avons commencé notre travail par représenter la notion de système à évènements discrets dans le premier chapitre, suivi par une explication des modèles « RecDevs ». Ensuite, dans le deuxième chapitre nous avons décrit les aspects fondamentaux de model checker UPPAAL qu'on a utilisé comme outil de simulation et de vérification des propriétés de modèle Autovision qu'on a proposé suivant une succession des étapes, d'abords on a commencé par modéliser dans l'outil ATOM3 les différents composants de système Autovision à l'aide d'un méta modèle de formalisme RecDEVS, dont chaque composant a sa propre apparence graphique. Nous avons également utilisé une grammaire de graphe adaptée pour transformer le modèle source en un code cible en UPPAAL simulé et vérifié par la suite.
Comme il est indiqué dans ce travail, la vérification et la modélisation des systèmes temps réel dynamiquement reconfigurable est une approche très important pour améliorer les performances.
Comme perspectives pour ce travail nous pouvons prévoir l'étude et la modélisation d'autres systèmes dynamiquement reconfigurables, pour pouvoir en appliquer cette approche afin de détecter et d'en améliorer ses points faibles.

Bibliographie

[1] Laurent Capocchi. *DEVSimPy-v2.8.* PhD thesis, University of Corsica., 2014.

[2] Maen ATLI. *Contributions à la synthèse de commande des systèmes à événements discrets : nouvelle modélisation des états interdits et application à un atelier flexible.* PhD thesis, Université de Laurraine, 2012.

[3] Youssef BOUANAN. *Contribution à une architecture de modélisation et de simulation à évènements discrets : Application à la propagation d'information dans les réseaux sauciaux.* PhD thesis, Université de Bordeaux, 2016.

[4] Paul-Antoine Bisgambiglia. *Approche de modélisation approximative pour des systèmes à évènements discrets : Application à l'étude de propagation de feux de forêt. Modélisation et simulation.* PhD thesis, Université Pascal Paoli, 2008.

[5] Stéphane Garredu. *Approche de méta-modélisation et transformations de modèles dans le contexte de la modélisation et simulation à évènements discrets : application au formalisme DEVS.* PhD thesis, Université Pascal Paoli, 2013.

[6] Felix Madlener. *A Model of Computation for Recon ?gurable Systems.* PhD thesis, technische universitat darmstadt, 2013.

[7] H.Klaudel G.Hutzler and D.Yue Wang. Automates temporisés et systèmes multi agents temps-réel. 2004.

[8] Manal Najem. *Model-checking symbolique pour la vérification de systèmes et son application aux tables de décision et aux systèmes d'éditions collaboratives distribuées.* PhD thesis, École polytechnique de Montréal, 2009.

[9] Pierre-Alain Bourdil. *Contribution à la modélisation et la vérification formelle par model checking - Symétries pour les Réseaux de Petri temporels.* PhD thesis, INSA de Toulouse, 2015.

[10] Faïez CHARFI. *Une approche d'interfaçage de CoD à UPPAAL pour la spécification et la vérification des systèmes temps réel.* PhD thesis, Université de Tunis, 2003.

[11] F.Larsson P.Pettersson J.Bengtsson, Kim G. Larsen and Wang.Yi. Uppaal in 1995. 1996.

[12] A.David G.Behrmann and Kim G.Larsen. A tutorial on uppaal. 2004.

[13] Kim G. Larsen P.Pettersson J.Illum Rasmussen Wang.Yi A.DAVID, G.Behrmann and M.Magnin. *Systèmes embarqués communicants Approches formelles.* hermes, 2007.

[14] Julia Weingart. Verification of devs models for reconfigurable systems. 2009.

[15] W.Stechele C.Claus and Andreas Herkersdorf. Autovision a run-time reconfigurable mpsoc architecture for future driver assistance systems. 2007.

[16] Benoît Combemale. Ingénierie dirigée par les modèles (idm) ? État de l ?art. 2008.

[17] O.Salvatori and X.Blanc. *MDA en action : ingénierie logicielle guidée par les modèles.* Ayrolles, Paris, 2005.

[18] Mme DEHIMI NARDJESS TISSILIA. *Un Cadre Formel pour La Modélisation et l'analyse Des Agents Mobiles, Thèse de Doctorat en Sciences en informatique.* PhD thesis, Université Mentouri Constantine, 16 août 2012.

[19] BOUDIA MALIKA. *Transformation des diagrammes d'états-transitions vers Maude, Mémoire pour le diplôme de magistère en Sciences Technologies de l'information et de communication.* PhD thesis, UNIVERSITE DE M'SILA, 2007.

[20] Czarnecki and S.Helsen. Classification of model transformation approaches. 2003.

[21] C.Hanane and D.Bessma. *Approche De Transformation De Modèle RecDEVS Vers UPPAAL.* PhD thesis, UNIVERSITE DE JIJEL, 2016.

RÉSUMÉ

Ces dernières années ont vu une vaste propagation des systèmes informatiques utilisés dans la vie quotidienne sous formes des différents appareilles. Ces systèmes sont souvent en temps réel dynamiquement reconfigurable. Donc pour bien comprendre et améliorer son comportement, et pour qu'ils nous donneront une meilleure production, la modélisation et la vérification sont devenues de plus en plus ces technique importantes afin de permettre une bonne décision en temps réel.
Dans notre mémoire, nous avons approfondi dans l'un des systèmes temps réel Autovision. Dans ce contexte nous avons utilisé une modélisation RecDEVS qui est la plus adaptée à ce genre de systèmes via l'outil ATOM3. Notre approche pour vérifier et simuler le modèle résultant est centralisée sur l'utilisation d'une grammaire de graphes pour la transformation de modèle RecDEVS en un code UPPAAL prêt à être simulé et vérifié.

Mots clés : RecDEVS, UPPAAL, AToM3, Transformation de graphe, Autovision.

ABSTRACT

Recent years have seen a vast spread of computer systems used in our life in the form of different devices. These systems are often dynamically reconfigurable in real time. So to better understand and improve its behavior, modeling and verification have become more and more important techniques to allow a good decision in real time.
In our thesis, we have depended Autovision, one of real time systems. In this context we have used a RecDEVS model which is the most adapted to this kind of systems by ATOM3 tool. Our approach to verify and semulate the resulting model is centralized on the use of a graph grammar for RecDEVS model transformation into an UPPAAL code ready to be simulated and verified.

Key words : RecDEVS, UPPAAL, AToM3, Graph transformation, Autovision.