UNIVERSITÉ DE NANTES

Master Informatique
2014/2015

Rapport de projet de recherche

Module d'initiation à la recherche

Antoine Forgerou Jérémy Bardon Nicolas Bourdin

Sommaire

1 Introduction

Ce document est un rapport rassemblant le travail que nous avons effectué dans le cadre du module d'initiation à la recherche au sein du Master 1 ALMA. Ce module permet d'initier les étudiants au travail de recherche en informatique, en immersion dans une équipe du laboratoire du LINA.

Le sujet de recherche auquel nous avons contribué est le suivant :

Modélisation et analyse des systèmes logiciels hétérogènes

La préoccupation de l'équipe qui nous a accueilli est de trouver un moyen d'assurer la cohésion de modèles d'un système hétérogène lors d'opérations d'ajout et/ou de suppression des modèles le composant. Tout les modèles seront alors capables de communiquer ensembles ; et dans un premier temps, le choix a été fait de s'appuyer sur l'exemple d'automates que l'on voudrait composer.

Au cours de ce rapport, différents points seront abordés. Tout d'abord une présentation du laboratoire LINA ainsi que de l'équipe AeLoS que nous avons intégrée le temps du module de recherche. Nous reviendrons alors plus en détail sur la problématique soulevée par le sujet pour ensuite présenter les 3 solutions proposées par l'équipe. Ensuite, nous expliquerons la solution que nous avons choisie de mettre en place et nous rentrerons au cœur du projet avec la présentation de la solution que nous avons implémentée.

2 Présentation du laboratoire

Acteur central du développement de l'informatique dans la région des Pays de La Loire, le LINA (Laboratoire d'Informatique de Nantes Atlantique) est un laboratoire de recherche en sciences et technologies du logiciel qui est dirigé par Pierre Cointe. Avec ses 180 membres, ce laboratoire est actuellement situé sur deux sites Nantais : la Lombarderie (Faculté des Sciences et Techniques) et la Chantrerie (Ecole des Mines et Polytech' Nantes).

2.1 Les différentes équipes

Le LINA est composé des équipes suivantes :

— AeLoS : **A**rchitectures et **Lo**giciels **S**ûrs
— ASCOLA : **AS**pect and **CO**mposition **LA**nguages
— AtlanMod : **Atlan**tic **Mod**eling
— ComBi : **Com**binatoire et **Bi**oinformatique
— DUKe : **D**ata **U**ser **K**nowledg**e**
— GDD : **G**estion de **D**onnées **D**istribuées
— OPTI : Optimisation globale, optimisation multi-objectifs
— TALN : **T**raitement **A**utomatique du **L**angage **N**aturel
— TASC : Programmation par contraintes

3 L'équipe AeLoS

L'équipe Architectures et Logiciels Sûrs (AeLoS), dirigé par Christian ATTIOGBE, est une équipe de recherche.

Le projet central de cette équipe s'appuie sur les trois thématiques suivantes :

- Architecture :
- Composants logiciels corrects :
- Multiformalisme et analyse multifacette :

3.1 Les membres

Une récente liste des membres est disponible sur le site du LINA, à l'adresse suivante :
`http://www.lina.univ-nantes.fr/spip.php?page=membres&id_equipe=15&lang=fr`

4 Présentation du sujet de recherche

4.1 Problématique

La construction rigoureuse d'un logiciel se fait à partir de son modèle. Pour des logiciels complexes (par exemple ceux qui ont de nombreux composants différents – hétérogènes– et qui communiquent), on dispose de plusieurs modèles (hétérogènes aussi) qui doivent interagir de façon cohérente, pour garantir l'interopérabilité sémantique entre les modèles puis les composants logiciels. Le domaine des systèmes embarqués (et aussi des objets connectés) regorge d'exemples.

Prenons un exemple concret :
un système de surveillance filme et transmet des images sur un moniteur distant comme ceci :

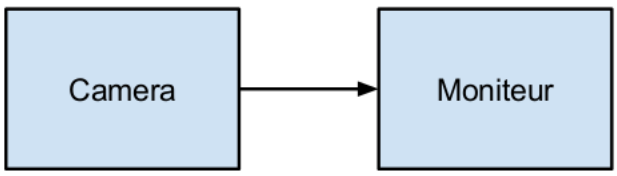

FIGURE 4.1 – Exemple n1 - Camera Moniteur

Que se passerait il si l'on souhaite alors sauvegarder les vidéos de surveillance sur une base de données comme ci-dessous ? Notre rôle est de faire en sorte que les différents modèles qui composent le systèmes se doivent de fonctionner correctement malgré l'ajout de nouveaux composants.

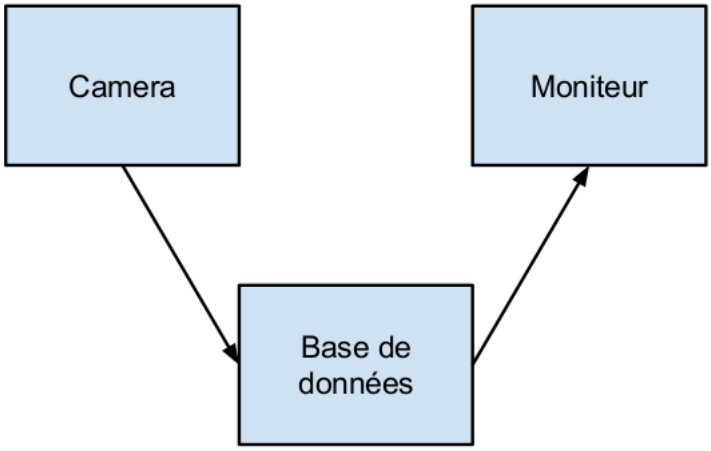

FIGURE 4.2 – Exemple n2 - Camera BDD Moniteur

Autre exemple :
Le logiciel comme YMPD [1] permettant la lecture côté serveur de piste audio, est constitué de deux langages que tout opposent :

— Le langage C (de bas niveau)

— Le langage Javascript (de très haut niveau)

Cependant, le logiciel doit fonctionner correctement malgré l'utilisation de ces langages différents. Ainsi, l'utilisation de modèles rend possible la représentation de systèmes et permet le fonctionnement du logiciel dans un langage commun qui n'est autre que le modèle d'automate. D'où la nécessite de manipuler les modèles.

4.2 Solutions proposées par l'équipe AeLoS

Pour répondre à la problématique, l'équipe AeLoS nous a proposé 3 façons de faire tout en nous laissant la liberté d'en proposer d'autres. Dans le but

1. ympd.org

de simplifier les exemples, nous avons fait le choix de représenter les modèles comme des formes géométriques que l'on essayerait d'emboîter.

La première solution consiste à adapter un des modèles au deuxième quand cela est possible. On peut prendre l'exemple du théorème de Kleene qui assure qu'un automate à états finis peut être écrit sous la forme d'une expression rationnelle et vice et versa. Par conséquent on peu adapter des composants ayant comme modèle les automates ou les expressions régulières.

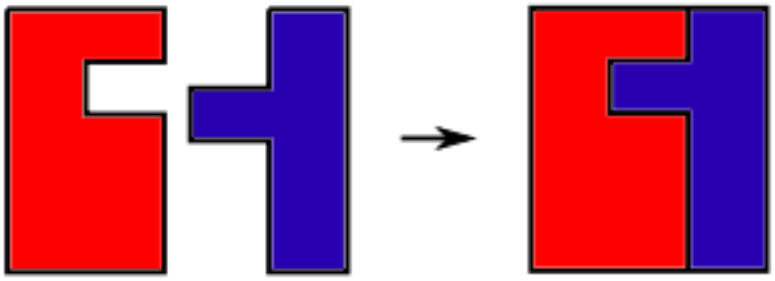

FIGURE 4.3 – Solution 1 - Adaptation d'un modèle

Dans cet exemple c'est le composant de gauche qui s'est adapté – par une légère modification – pour pouvoir interagir avec la partie droite.

L'approche qui parait la plus évidente mais qui peut se révéler complexe consiste à intégrer un troisième modèle dans le système. Ce dernier va alors jouer le rôle de passerelle entre les deux modèles que l'on veut faire communiquer.

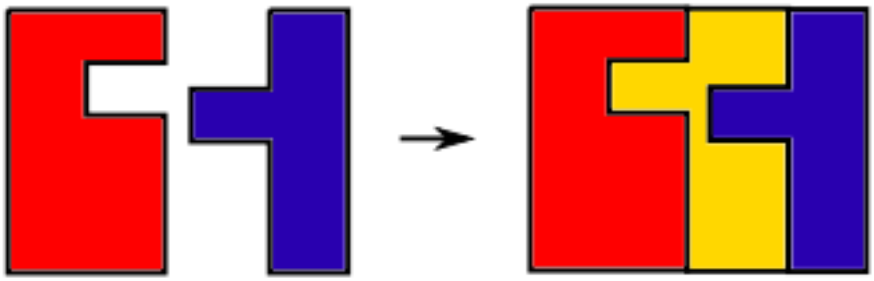

FIGURE 4.4 – Solution 2 - Interface entre les modèles

Le principe est de faire interagir les modèles ensemble puis d'effectuer des vérifications pour déterminer si les modèles peuvent communiquer à 100%, en partie ou pas du tout. Cela rentre dans une approche globale (Plug'N'check) que l'équipe étudie.

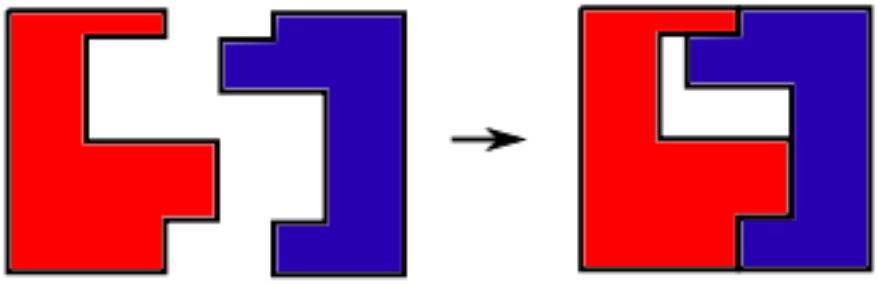

FIGURE 4.5 – Solution 3 - Plug'N'Check

L'exemple montre que certaines parties sont satisfaites mais d'autres parties ne peuvent pas communiquer.

Dans le cadre de notre stage d'initiation à la recherche, nous nous sommes orientés vers la première solution. Nous avons donc entrepris de transformer des modèles dans un langage commun afin de permettre de les interfacer.

Il est admis que les automates sont une spécialisation de graphe, ce qui signifie que n'importe quel automate peut-être représenté sous forme de graphe. C'est la raison qui nous a poussé à utiliser le langage *DOT* dans le projet.

4.3 Rappel sur les Automates

Les automates sur lesquels nous nous appuierons pour ce projet sont les automates à états fini, non-déterministes contenant des ϵ-transitions.

Rappelons la définition de l'automate à états fini non déterministe :

1. un ensemble fini d'états, souvent noté Q
2. un ensemble fini de symboles, appelé alphabet et souvent noté Σ
3. une fonction de transition, souvent nommée δ, qui reçoit comme arguments un état de Q et un symbole de Σ ou la chaîne vide notée ϵ et qui rend comme résultat une partie de Q

$$\delta : Q * (\Sigma \cup \epsilon) \rightarrow P(Q)$$

4. un état initial q0, appartenant à Q
5. un sous-ensemble F de Q constitué des états finals, ou états acceptants.

Voici un exemple d'automate, extrait de notre projet :

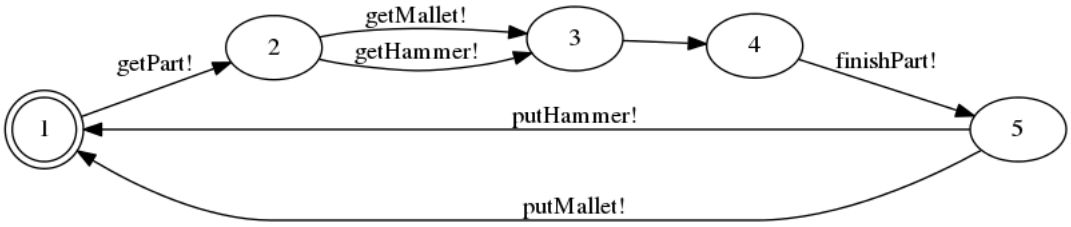

FIGURE 4.6 – Automate du "jobber" qui usine des pièces

En appliquant la définition donnée ci-dessus, on obtient pour notre exemple :

1. l'ensemble d'états Q = {1,2,3,4,5}
2. l'alphabet Σ = {getPart !,finishPart !,getMallet !,putMallet !,getHammer !,putHammer !}
3. l'état initial q0 = 1 et il n'y a pas d'état final F = {}
4. la liste des fonctions de transition :
   (a) δ(1,getPart !) = 2
   (b) δ(2,getHammer !) = 3 et δ(2,getMallet !) = 3

(c) $\delta(3,\epsilon) = 4$
(d) $\delta(4,\text{finishPart}\,!) = 5$
(e) $\delta(5,\text{putMallet}\,!) = 1$ et $\delta(5,\text{putHammer}\,!) = 1$

4.4 UPPAAL : un outil de modélisation des automates

Uppaal est un environnement intégré d'outils pour la modélisation , la validation et la vérification des systèmes temps réel modélisés comme des réseaux d' automates.

Uppaal se compose de trois parties principales :

- un langage de description,
- un simulateur,
- un vérificateur de modèle.

Le langage de description est un langage de commande non-déterministe avec des types de données (par exemple des entiers, des tableaux, etc.) . Il se sert de la modélisation ou du langage de conception pour décrire le comportement du système grâce à des réseaux d'automates étendus avec des variables d'horloge et des données.

Le simulateur est un outil de validation qui permet l'examen de possibles exécutions d'un système au début de la conception(ou la modélisation) et fournit un moyen peu coûteux de détection de défaut avant la vérification par le vérificateur de modèle qui couvre l'ensemble des comportement dynamique du système.

Le modèle peut vérifier les propriétés invariantes et l'accessibilité en explorant l'espace et l'état d'un système, c'est à dire l'analyse de l'accessibilité en termes d'états symboliques représentés par des contraintes.

4.5 Le format DOT

Le langage DOT est un langage de description de graphe dans un format texte. C'est une manière simple de décrire des graphiques que les humains et les programmes informatiques peuvent utiliser. Les graphes DOT sont généralement des fichiers avec pour extension un .gv (ou .dot).

La syntaxe DOT peut aussi bien décrire des graphes non-orientés que des graphes orientés, comme des automates finis. Il est possible de placer des étiquettes sur les transitions pour par exemple leurs donner un nom ou une explication. Le langage possède aussi des attributs permettant une description graphique plus poussée, par exemple choisir la couleur d'un nœud ou de dessiner une transition en pointillé.

4.5.1 Graphe non-orienté

```
graph monGraphe {
    a -- b -- c;
    b -- d;
}
```

Listing 4.1 – Description textuelle en DOT

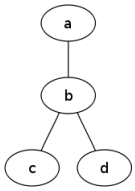

FIGURE 4.7 – Résultat de visualisation

4.5.2 Graphe orienté

```
graph monGraphe {
    a -> b -> c;
    b -> d;
}
```

Listing 4.2 – Description textuelle en DOT

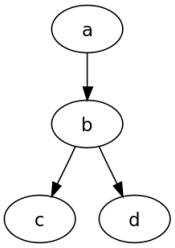

FIGURE 4.8 – Résultat de visualisation

5 Hetersys [1] : notre solution

5.1 Introduction

Notre but est de pouvoir composer n'importe quel type d'automate en utilisant Uppaal. Celui-ci propose de créer des automates et de les rassembler sous forme de projets mais il faut obligatoirement dessiner ces automates avec les outils mis à disposition par Uppaal.

Etant donné que les automates sont un sous-ensemble des graphes, ce module propose d'importer directement un automate – décrit sous forme de graphe – dans un projet Uppaal pré-existant.

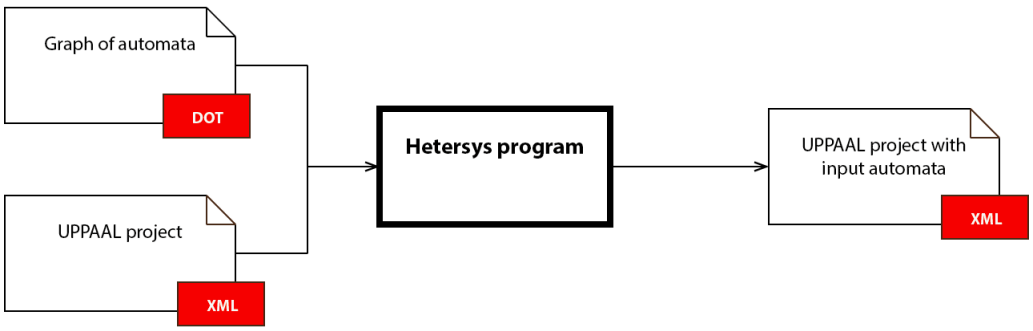

Figure 5.1 – Entrées et sorties du programme

Pour la description du graphe, nous avons choisi de nous appuyer sur le format DOT bien connu qui est selon nous un langage très simple et minimaliste qui permet de construire des graphes très rapidement.

1. Heterogeneous systems : Tiré du nom du sujet de recherche "Modélisation et analyse des systèmes logiciels hétérogènes"

5.2 Conception

Comme évoqué dans l'introduction la fonctionnalité attendu de notre module est de pouvoir intégrer un automate dans un système déjà composé de plusieurs automates.

Dans un premier temps nous avons fait le choix d'utiliser le format DOT pour la description des automates – sous forme de graphe – et le logiciel UPPAAL qui permet de faire de la composition d'automates.

Afin d'assurer l'extensibilité de notre module, nous n'avons pas tout simplement converti le format DOT en format UPPAAL. En effet, l'automate en entrée est mis sous forme de graphe – en mémoire – par un *importeur* qui le donne ensuite à un *exporteur* qui s'occupe de la conversion. Ce type de fonctionnement permet facilement d'intervertir les modules importeur et exporteur ainsi que d'ajouter facilement de nouveaux formats d'entrée et/ou de sortie.

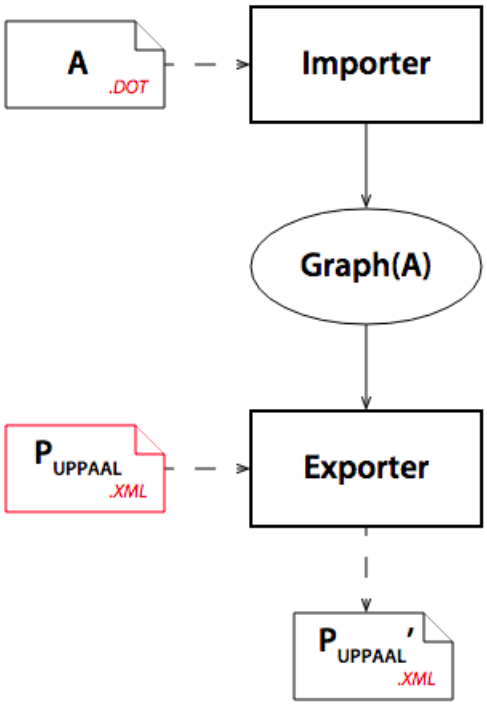

FIGURE 5.2 – Composants du logiciel

Quelque soit le format utilisé pour définir l'automate en entrée et le

système, le fonctionnement reste identique car le but est d'ajouter l'automate dans le système.

$$P'_{UPPAAL} = P_{UPPAAL} + XML(Graph(A)) \tag{5.1}$$

Ce module constitue un noyau d'application mais pour les besoins de la recherche il évoluera pour aussi permettre la suppression d'un automate dans le système.

Le but du sujet de recherche étant d'assurer la cohésion entre les modèles d'un système hétérogène, le module assure un certain nombre de vérifications avant d'effectuer l'action demandée.

Afin d'effectuer ces vérifications, ìl faut aussi importer les automates du système dans le module et on va alors utiliser un second *importeur* qui connait le format des automates dans le système (XML pour Uppaal).

5.3 Architecture logicielle

L'architecture interne du programme a été pensée pour être la plus modulaire possible ce qui signifie que l'on peut facilement importer et/ou exporter dans un autre format sans changer ce qui existe déjà.

En effet, le programme possède une représentation interne de l'automate sous forme de graphe ce qui rend ce genres d'ajustements plus simples. Les parties *importer* et *exporter* constituent donc les points d'extension de notre programme.

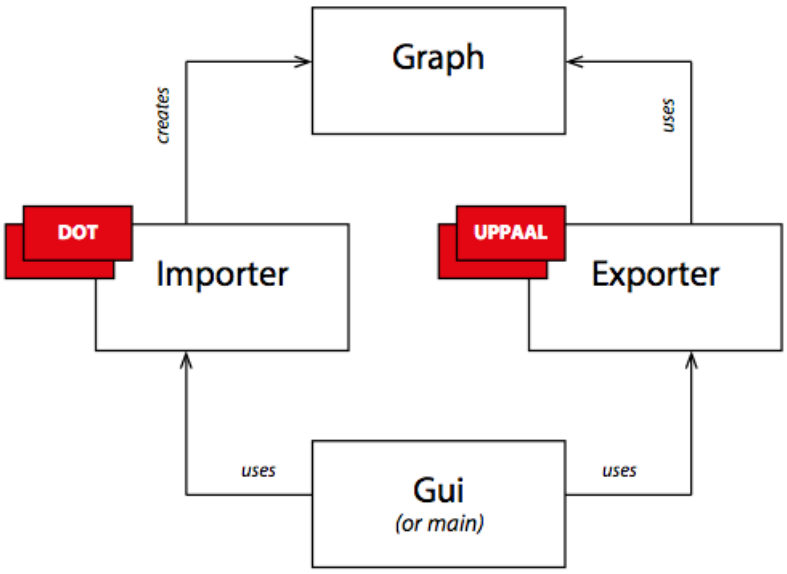

FIGURE 5.3 – Composants du logiciel

Nous avons mis à disposition une interface graphique mais nous l'avons conçue dans l'optique de la découplée le plus possible du moteur de l'application. Il serait alors aisé de la supprimer ou de la remplacer puisqu'elle suit le patron de conception MVC.

Par conséquent, la routine principale du programme se trouve au niveau du *modèle* qui se charge de dérouler le programme tout en interagissant avec l'interface.

5.4 Fonctionnement interne

La partie principale du programme est celle qui connaît et utilise l'importeur et l'exporteur. Le fonctionnement se résume donc à utiliser ces deux composants afin de charger l'automate et de l'ajouter dans le projet UPPAAL tout en procédant à certaines vérifications au préalable.

La ressource la plus importante est le graphe car il est généré par l'importeur puis utilisé par l'exporteur – non représenté mais passé à la création – pour vérifier la compatibilité avec le projet UPPAAL et la conversion dans le bon format.

Par ailleurs, au niveau de l'implémentation – avec l'interface graphique –

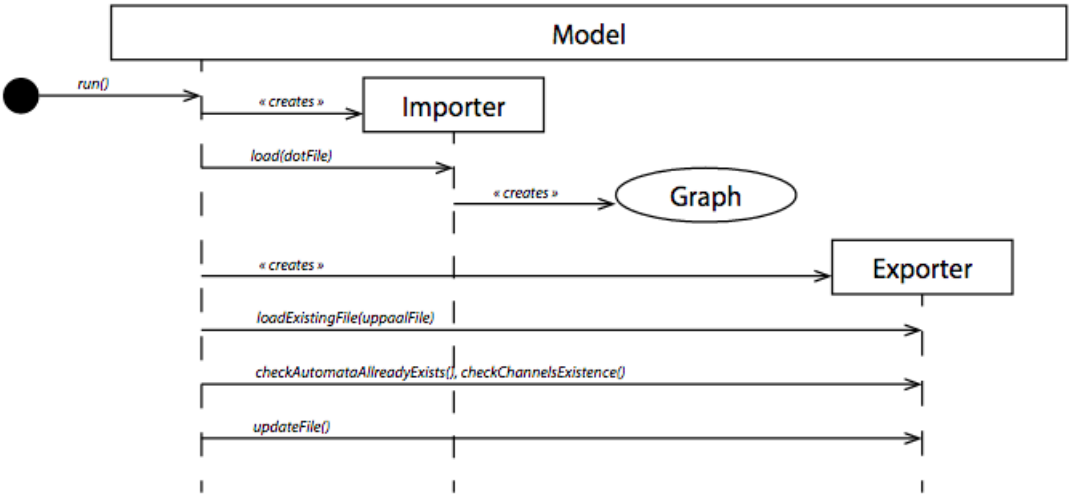

FIGURE 5.4 – Schéma du déroulement du programme (non formalisé)

nous avons découpé cette routine en étapes. Ceci nous a permis de pouvoir reprendre à n'importe quel moment si il y a eu un souci et que l'utilisateur est en moyen de le résoudre via l'interface.

5.5 Choix techniques

Technologie

Nous avons décidé d'utiliser la technologie JAVA pour développer notre solution car elle a été utilisé par les développeurs du projet UPPAAL.

UPPAAL n'est pas open source mais s'il le devient ce sera alors plus facile d'intégrer notre solution d'import directement en interne.

Aspect mis de coté

Afin de nous concentrer sur la conversion du graphe et l'importation dans le projet UPPAAL, le logiciel ne prend pas en compte la disposition graphique des éléments de l'automate.

Il se contente alors de les afficher en ligne afin de pouvoir les identifier facilement mais l'utilisateur devra les réorganiser lui-même sachant que UPPAAL n'intègre pas de fonctions de réorganisation automatique

5.6 Documentation du module Hetersys

5.6.1 Fenêtre principale

Il s'agit du premier élément auquel l'utilisateur est confronté mais l'interface a été pensée pour faciliter l'apprentissage.

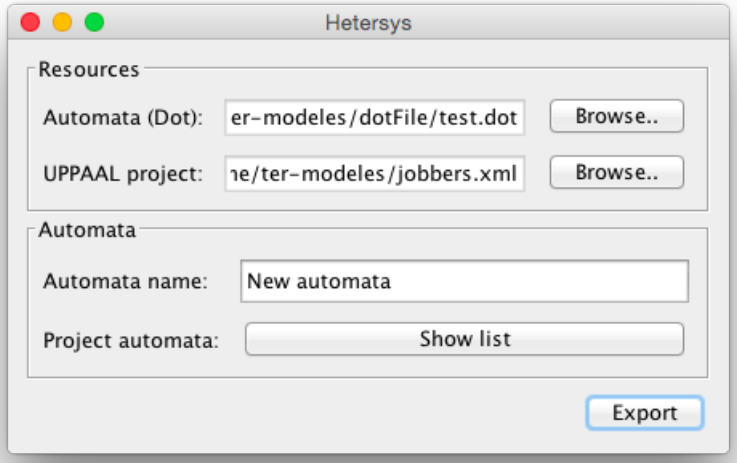

FIGURE 5.5 – Fenêtre principale

En effet elle est décomposée en 2 parties : *Resources* et *Automata*.

La première partie regroupe les fichiers dont le programme à besoin en entrée à savoir un automate – décrit sous forme de graphe en DOT – et un projet UPPAAL pré-existant.

La seconde section s'intéresse à l'automate qui sera ajouté dans le projet. Dans un projet UPPAAL, chaque automate a un nom pour pouvoir l'identifier de manière unique ce qui veut dire que deux automates ne peuvent pas avoir le même nom. C'est pourquoi le bouton « Show list » permet d'afficher le

nom des automates déjà présents dans le projet afin de ne pas les réutiliser.

Un appui sur le bouton export permet simplement d'exporter l'automate – au format DOT – dans le projet Uppaal donné.

5.6.2 Gestion des canaux

L'automate donné en entrée n'ayant pas de contraintes particulières – si ce n'est respecter le format DOT – il n'est pas obligatoire que celui-ci ait des canaux en commun avec les autres automates du projet.

C'est pourquoi, notre module va se charger de scanner les canaux utilisés dans l'automate et dans le projet pour notifier l'utilisateur dans 2 cas particuliers.

Canaux non déclarés

Dans le cas où l'automate en entrée utiliserait des canaux qui ne sont pas déclarés dans le projet Uppaal, notre module demandera à l'utilisateur s'il veux les importer.

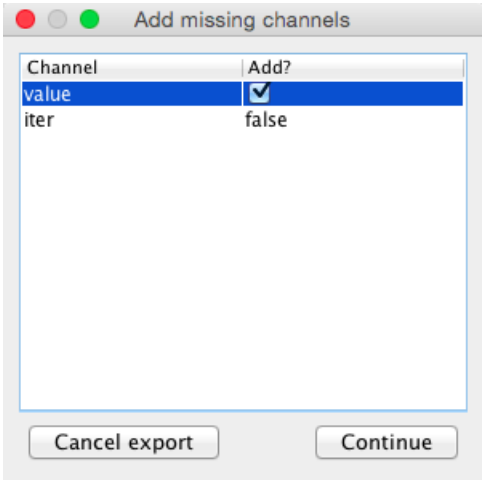

Figure 5.6 – Ajout de canaux

La liste des canaux permet de décider au cas par cas quel canaux il faut importer ou pas.

Aucun canal en commun

Le but de ce logiciel étant de composer les automates entre eux, lorsque l'automate en entrée n'a aucun canal en commun avec ceux dans le projet un message d'alerte apparaît.

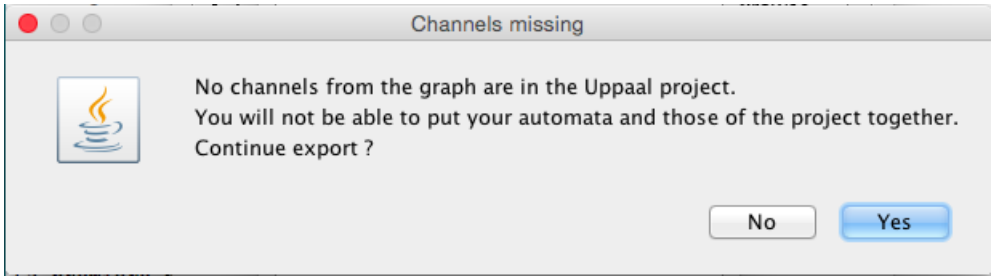

FIGURE 5.7 – Aucun lien entre les automates

Ceci permet à l'utilisateur d'annuler l'export pour changer son automate et utiliser des canaux déjà existants avant de le ré-exporter.

5.6.3 Messages d'information

Automate en double

L'automate à insérer ne doit pas porter le même nom que ceux déjà dans le projet donc une fenêtre notifie l'utilisateur si c'est le cas.

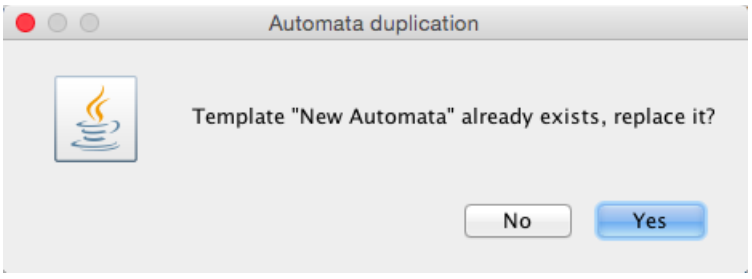

FIGURE 5.8 – Automate en double

Fin de l'export

L'export d'un automate est très rapide mais le logiciel notifie quand même l'utilisateur quand celui-ci est terminé.

Le message rappelle quel projet UPPAAL a été modifié pendant l'opération mais aussi le nom de l'automate qui a été ajouté.

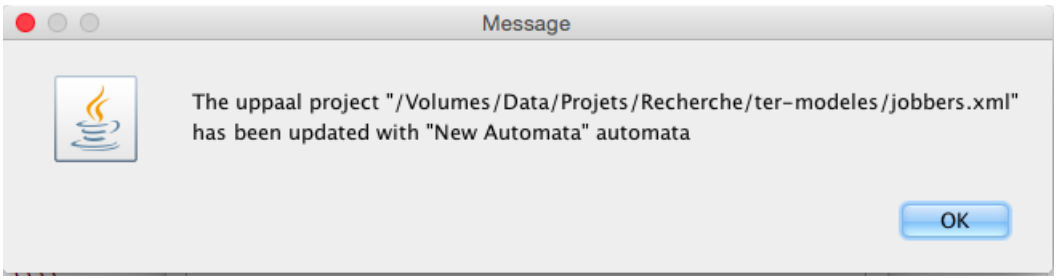

FIGURE 5.9 – Export réussi

6 Expérimentations et validation

Afin d'expliciter les résultats obtenus grâce à notre application, nous allons prendre deux cas concrets. Nous pourrons alors voir à travers eux les extensions possibles de notre programme.

6.1 Cas du robot travailleur

Notre premier exemple est le cas qui à servi d'exemple précédemment : le robot travailleur. Le principe est basique, un travailleur se sert d'outil (un marteau et un maillet) pour usiner des objets.

Pour commencer, nous avons créé le cas le plus simple sous Uppaal : un travailleur, travaillant sur un seul objet. Pour travailler sur cet objet, il peut se servir d'un maillet ou d'un marteau, cependant pour le moment il n'a qu'un marteau à sa disposition.

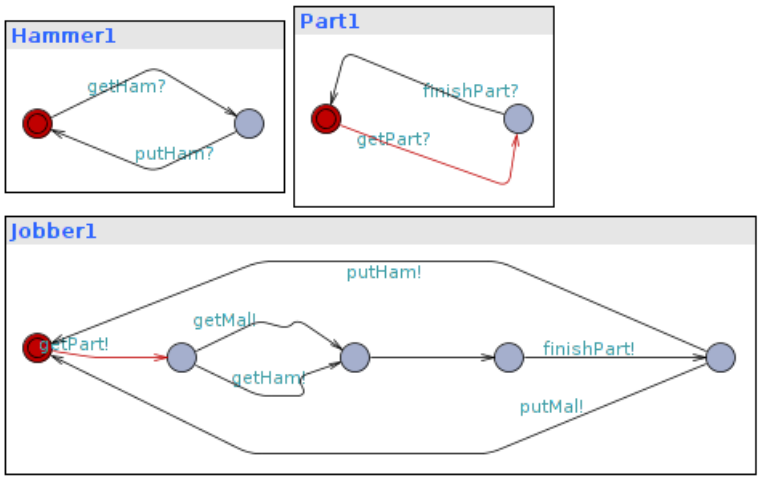

Figure 6.1 – Cas de départ

Le fonctionnement est plutôt simple, le robot travailleur commence par prendre un objet, puis un outil disponible. Une fois qu'il a fini de travailler, il pose son outil, puis l'objet.

Pour enrichir notre système, on souhaite ajouter la possibilité au travailleur de prendre l'outil mallet. Pour ce faire on commence par créer un fichier dot, décrivant le fonctionnement du mallet, puis on lance notre application pour l'ajouter au système déjà existant.

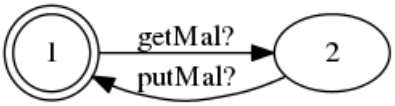

FIGURE 6.2 – L'automate maillet

L'ajout du maillet au système, ne nécessite pas la création de nouveau canaux, car il utilise ceux déjà existant. Après la simulation dans UPPAAL, on constate que l'automate maillet est parfaitement intégré, le travailleur prenant aléatoirement le marteau ou le maillet pour travailler.

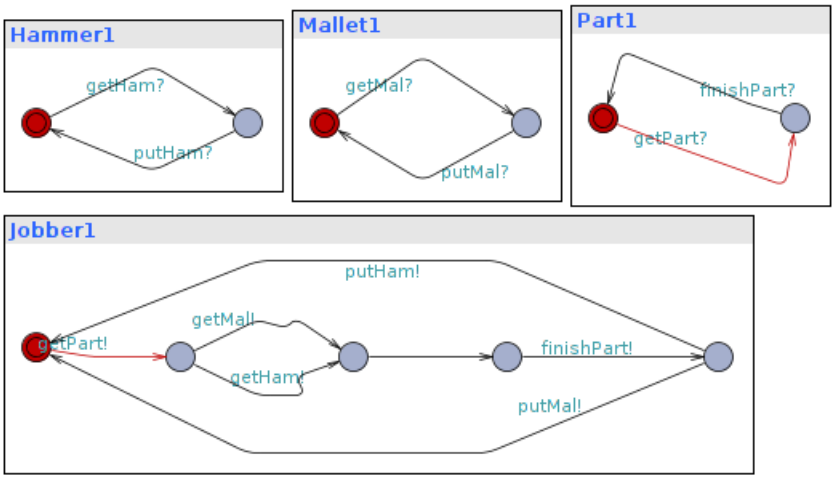

FIGURE 6.3 – Cas après l'ajout de l'automate maillet

6.2 Cas de la surveillance caméra-écran

Pour illustrer le fonctionnement de notre programme, nous avons choisi un second exemple, celui d'un système de surveillance. Cet exemple est un des cas sur lesquels l'équipe AeLoS travaille : une caméra de surveillance filme une pièce et envoie en continue les images à un écran qui les affiche.

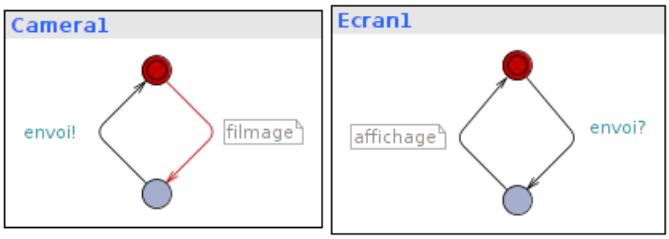

FIGURE 6.4 – Cas de départ

Maintenant, nous souhaitons rajouter une fonction de sauvegarde à notre système. Avant que les images ne soit affichées sur un écran elle doivent être enregistrées. L'automate représentant ce comportement est assez simple : il récupère la vidéo de la caméra, l'enregistre et la renvoie ensuite à l'écran.

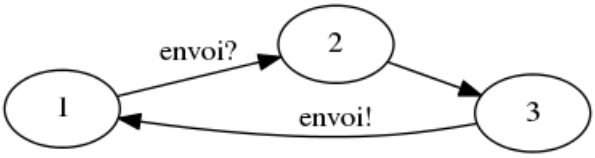

FIGURE 6.5 – L'automate de sauvegarde

L'automate est intégré au système. Cependant le fonctionnement n'est pas celui attendu. En effet, nous souhaitions que systématiquement l'enregistrement passe par la caméra, ce qui n'est pas le cas ici. Lors de l'envoi de l'image par la caméra, la réception sera faite aléatoirement par ceux qui attendent cette enregistrement : le système de sauvegarde ou l'écran.

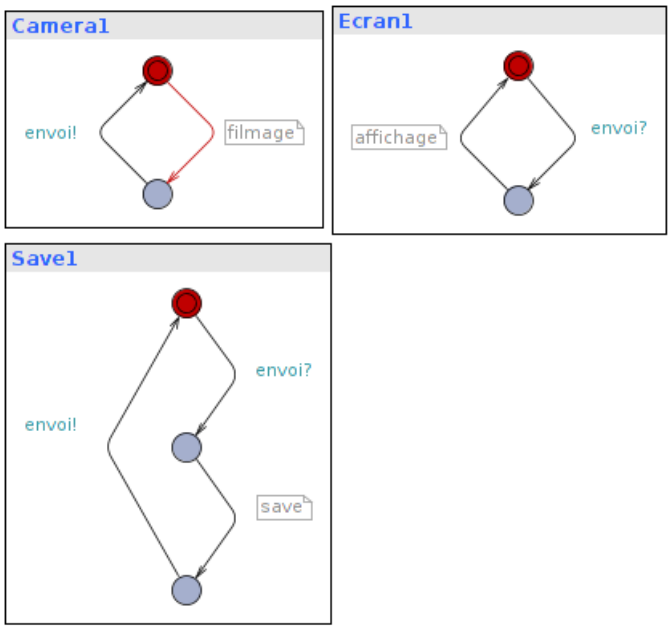

FIGURE 6.6 – Cas après l'ajout de la fonction de sauvegarde

Afin d'obtenir le comportement obtenu, il faudrait empêcher la communication caméra/écran. On pourrait par exemple, créer un canal spécifique entre la sauvegarde et l'écran. De cette manière, la communication se fera toujours comme souhaitée.

6.3 Problèmes connus

Cette partie expose les soucis que peux engendrer notre solution.

En effet, quand l'on ajoute un nouvel automate dans un projet Uppaal et que l'on veux lancer une simulation ce n'est pas possible.

Ceci est lié au fait que pour tout automate présent dans le projet, Uppaal oblige à avoir une "instance" dans le système. Cependant, nous considérons que notre logiciel doit seulement se soucier de la cohérence entres les modèles et non de leur utilisation : c'est donc de la responsabilité de l'utilisateur.

6.4 Conclusion

Nous avons conscience des limites de notre application, cependant nous nous sommes concentrés sur l'ajout d'un automate à un système existant.

La gestion de la cohérence avec le système existant est le coeur du sujet de recherche mais n'en est pour l'instant qu'à ces débuts. Cependant, notre solution intègre déjà les vérifications suivantes :

— Existence d'un automate du même nom

— Indication des canaux communs entre l'automate à insérer et le projet Uppaal

— Ajout à la demande des nouveaux canaux pendant l'insertion

Pour conclure, le but premier de notre outil – qui est rempli – est d'insérer un automate dans un projet Uppaal mais dans le futur ce sera la partie *gestion de la cohérence* qui sera prédominante car il s'agit du sujet de recherche.

7 Bilan

7.1 Résultats

Après avoir étudié les 3 solutions proposées par l'équipe AeLoS (section 4.2) pour composer des systèmes hétérogènes, nous avons choisi de nous concentrer sur la première qui repose sur le principe d'adaptation.

En effet, pour faire communiquer 2 automates de nature différente on peut essayer d'en modifier un pour qu'il s'adapte – dans sa forme – au second. Nous avons donc implémenté une solution reposant sur le logiciel Uppaal qui lui propose de composer des automates entre eux.

Notre solution – appelée Hetersys – permet d'importer un automate décrit sous forme de graphe – au format DOT – dans un projet Uppaal qui contient d'autres automates. L'adaptation de l'automate se fait donc de manière naturelle – par le passage au format graphe – et ce car n'importe quel type d'automate peux être représenté sous cette forme.

7.2 Perspectives

La solution que nous avons implémentée constitue un noyau d'application – fonctionnel de bout en bout – qui a été conçu pour être facilement extensible pour les personnes qui voudront y ajouter des fonctionnalités.

Nous nous sommes concentrés sur le langage DOT pour représenter les graphes – pour accélérer le développement – mais il serait intéressant de pouvoir définir des types de syntaxes. Cela permettrait aux langages dont la syntaxe est proche de ne pas avoir à redéfinir un importeur dont juste les mots (ou les symboles) liés au langage seraient changés.

Si l'on se ressitue au niveau du problème dans sa globalité, notre outil

permet de gérer l'ajout de composants logiciels dans un système hétérogène mais il n'intègre pas encore toutes les éléments liés à cette opération.

Cela s'explique par le fait que la recherche sur le sujet n'est pas finie, elle demande donc plus de travail pour identifier les points qui doivent être vérifiés lors de l'ajout – ou de la suppression – d'un modèle au sein d'un système hétérogène.

Notre travail peut tout de même constituer une base pour tester des hypothèses sur le sujet et éventuellement – au travers d'évolutions – être un support pour démontrer par des expérimentations les résultats de la recherche.