VALIDATION DE SPÉCIFICATIONS DE SYSTÈMES D'INFORMATION AVEC ALLOY

par

Mohammed Ouenzar

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 18 mars 2013

Your file Votre référence
ISBN: 978-0-494-95109-5

Our file Notre référence
ISBN: 978-0-494-95109-5

Le 22 mars 2013

le jury a accepté le mémoire de Monsieur Mohammed Ouenzar dans sa version finale.

Membres du jury

Professeur Marc Frappier
Directeur de recherche
Département d'informatique

Monsieur Benoît Fraikin
Codirecteur
Département d'informatique

Professeur Jean-Louis Lanet
Évaluateur externe
Université de Limoges

Professeur Richard St-Denis
Président rapporteur
Département d'informatique

Sommaire

Le présent mémoire propose une investigation approfondie de l'analyseur ALLOY afin de juger son adaptabilité en tant que vérificateur de modèles. Dans un premier temps, l'étude dresse un tableau comparatif de six vérificateurs de modèles, incluant ALLOY, afin de déterminer lequel d'entre eux est le plus apte à résoudre les problématiques de sécurité fonctionnelle posées par les systèmes d'information. En conclusion de cette première phase, ALLOY émerge comme l'un des analyseurs les plus performants pour vérifier les modèles sur lesquels se fondent les systèmes d'information. Dans un second temps, et sur la base des problématiques rencontrées au cours de cette première phase, l'étude rapporte une série d'idiomes pour, d'une part, présenter une manière optimisée de spécifier des traces et, d'autre part, trouver des recours afin de contourner les limitations imposées par ALLOY. À ces fins, le mémoire propose deux nouveaux cas d'espèce, ceux d'une cuisinière intelligente et d'une boîte noire, afin de déterminer si oui ou non l'analyseur est capable de gérer les systèmes dynamiques possédant de nombreuses entités avec autant d'efficacité que les systèmes qui en possèdent moins. En conclusion, le mémoire rapporte que ALLOY est un bon outil pour vérifier des systèmes dynamiques mais que sa version récente, DYNALLOY, peut être encore mieux adapté pour le faire puisque précisément conçu pour faire face aux spécificités de ce type de système. Le mémoire s'achève sur une présentation sommaire de ce dernier outil.

Mots-clés: ALLOY ; boîte noire ; idiome de spécification ; logique dynamique ; logique du premier ordre ; sécurité ; solutionneurs de satisfiabilité ; spécification formelle ; sûreté ; vérification de modèles.

Remerciements

Dans un premier temps, je tiens à remercier mon directeur, le professeur Marc Frappier, qui m'a proposé d'entamer une recherche en informatique au sein du laboratoire qu'il dirige à l'Université de Sherbrooke. Sa bonne humeur, sa patience, sa minutie et son sens de l'organisation ont été de précieux recours dans la réussite de ce projet. Par delà les connaissances qu'il aura su me transmettre, son fameux mantra, "Il n'y a pas de miracle", aura toujours été pour moi le catalyseur d'une motivation réelle à positiver et à soutenir mes efforts durant les moments de difficulté et de découragement.

Merci au Docteur Benoît Fraikin, mon codirecteur de maîtrise, qui a su partager une partie de ses connaissances scientifiques avec moi. Celles-ci ont été d'une grande utilité pour ma recherche. Malgré son emploi du temps chargé, M. Fraikin a toujours su trouver le temps de répondre à mes questions quand bien même celles-ci n'avaient aucun rapport avec l'objet de ma recherche.

Je suis également reconnaissant à l'égard du Professeur Richard St-Denis, parangon de rigueur et d'honnêteté. Merci à vous pour votre aide et votre contribution.

Un grand merci à Guillaume Bouffard, Michel Embe Jiague, Romain Chossart, Alexis Mehdi Mantrach et à tous mes amis et collègues, qui ont participé, d'une manière ou d'une autre, à la réalisation de ce projet de recherche.

Je remercie également les employés du Département informatique de l'Université de Sherbrooke, et en particulier les techniciens Michel Benoît, Serge Cadorette et Jean-Francois Gauthier qui facilitent au quotidien la vie du personnel au département. Enfin, un merci aux professeurs Jeans-louis Lanet et Benoît Crespin grâce à qui j'ai pu bénéficier du programme de coopération entre l'Université de Limoges et l'Université de Sherbrooke.

À la mémoire de tout enfant mort à cause de l'ignominie humaine.

Abréviations

APIS Automatic Production of Information Systems (production automatique de systèmes d'information)

ADL Activities of Daily Living (activités de la vie quotidienne)

BCG Binary Coded Graph (graphe codé binaire)

BDD Binary Decision Diagram (diagramme de décision binaire)

CNF Conjunctive Normal Form (forme normale conjonctive)

CSP Communicating Sequential Processes (processus séquentiels communicants)

ER Entité-Relation

IS Information System (système d'information)

KSE Kitchen Stove Element (élément composant un four)

LTL Linear Temporal Logic (logique temporelle linéaire)

LTS Labelled Transition System (système de transitions étiquetées)

MDE Model-Driven Engineering (ingénierie dirigée par les modèles)

OOL Object-Oriented Language (langage orienté objet)

PwSN People with Special Needs (personnes avec des besoins spécifiques)

UML Unified Modelling Language (langage de modélisation unifié)

XTL eXtended Temporal Logic (logique temporelle étendue)

Table des matières

Sommaire i

Remerciements ii

Abréviations iii

Table des matières iv

Liste des figures vii

Liste des tableaux viii

Liste des programmes ix

Introduction 1

1 La vérification de modèles et ALLOY 5

1.1 Présentation de la vérification de modèles 5

1.2 Introduction à ALLOY 6

1.3 Le fonctionnement interne de ALLOY 6

1.4 Les outils de ALLOY 7

1.5 Avantages et limitations 8

1.6 Processus du développement d'un modèle en ALLOY 10

2 Comparaison de six vérificateurs de modèles 13

2.1 Introduction 15

2.2 Related Work 16
2.3 Presentation of the Case Study 17
2.4 An Overview of the Model Checkers 19
2.4.1 SPIN 19
2.4.2 NUSMV 20
2.4.3 FDR2 21
2.4.4 CADP 22
2.4.5 ALLOY 23
2.4.6 PROB 23
2.5 Specifying the Model and the Properties 24
2.5.1 SPIN 24
2.5.2 NUSMV 26
2.5.3 FDR2 27
2.5.4 CADP 28
2.5.5 ALLOY 29
2.5.6 PROB 30
2.6 Analysis of the Case Study 31
2.7 Conclusion 33

3 La vérification de modèles et l'informatique ubiquitaire 35
3.1 Introduction 37
3.2 INOVUS Intelligent Oven Project 38
3.2.1 Context 38
3.2.2 Scenario 38
3.2.3 Specifications 39
3.2.4 Approach 40
3.3 Related Work 41
3.3.1 Consistency Issue in Software Development Process 41
3.3.2 Formal Specifications 42
3.3.3 Introducing ALLOY 43
3.3.4 Tests and ALLOY Combination 44
3.4 INOVUS Development Process 44

3.4.1 First Prototype Specifications 45
3.4.2 First Increment Design 45
3.4.3 First Increment Implementation Using ALLOY 47
3.5 INOVUS Model Verification with ALLOY 50
3.5.1 More About ALLOY Syntax 50
3.5.2 Model Analysis 51
3.5.3 Additional Validation Tools in ALLOY 52
3.6 Results and Discussion 52
3.6.1 Relevancy of the Proposed Process 52
3.6.2 Evaluation of ALLOY in Pervasive Computing 53
3.7 Conclusion 54

4 Idiomes **55**
4.1 Revue de littérature 55
4.2 Les traces en ALLOY 56
4.3 Les propriétés de type arborescente 59
4.4 Boîtes noires 63
4.4.1 La vérification de modèles et les boîtes noires 64
4.4.2 Implémentation des boîtes noires 65
4.5 DYNALLOY 67

Conclusion **71**

A Spécification de la bibliothèque en ALLOY **74**

Liste des figures

1.1 Le processus du traitement interne de ALLOY 8
1.2 Exemples de la limitation réelle et de la limitation absolue 10
1.3 Idiome du développement d'un modèle en ALLOY 12

2.1 Requirement class diagram of the library system 17
2.2 Model checking duration in seconds for the properties of the library specification 33

3.1 INOVUS architecture in the home environment 40
3.2 Classical process with the proposed additional step 41
3.3 Coverage difference between tests and model checking 44
3.4 Simplified class diagram 46
3.5 Translation example from a UML class into its corresponding ALLOY signature 48

4.1 Principe des boîtes noires 64
4.2 Axiomes de base et axiomes de réduction 65

Liste des tableaux

3.1 Risks considered in this increment 45

4.1 Tableau récapitulatif du temps d'exécution par rapport aux bornes choisies . 63
4.2 Tableau récapitulatif du temps d'exécution par rapport aux bornes choisies . 66

Liste des programmes

1.1 Exemple illustrant quelque aspects de ALLOY 9
3.1 ALLOY translation of the property *Exceeding the hot threshold temperature for a temperature sensor must be detected as a risk situation* 50
3.2 Trace example in ALLOY 51
4.1 Exemple d'une trace initiée par un prédicat 57
4.2 Exemple d'une trace sans état initial prédéfini (partie 1) 57
4.3 Exemple d'une trace sans état initial prédéfini (partie 2) 58
4.4 Exemple d'une trace sans état initial prédéfini (partie 3) 58
4.5 Exemple de la technique du *stuttering* 60
4.6 L'ensemble des prédicats assurant le bon fonctionnement des actions *Return*, *Leave* et *Cancel* (partie 1) 61
4.7 L'ensemble des prédicats assurant le bon fonctionnement des actions *Return*, *Leave* et *Cancel* (partie 2) 62
4.8 L'ensemble des prédicats assurant le bon fonctionnement des actions *Return*, *Leave* et *Cancel* (partie 3) 63
4.9 Exemple d'une pile 66
4.10 Action Empiler 66
4.11 Propriété R-8 (partie 1) 66
4.12 Propriété R-8 (partie 2) 67
4.13 La spécification de l'action *Leave* avec DYNALLOY (partie 1) 68
4.14 La spécification de l'action *Leave* avec DYNALLOY (partie 2) 68
4.15 La spécification de l'action *Leave* avec ALLOY 69
A.1 La spécification de la bibliothèque (partie 1) 74
A.2 La spécification de la bibliothèque (partie 2) 75

A.3 La spécification de la bibliothèque (partie 3) 75
A.4 La spécification de la bibliothèque (partie 4) 76
A.5 La spécification de la bibliothèque (partie 5) 77
A.6 La spécification de la bibliothèque (partie 6) 79
A.7 La spécification de la bibliothèque (partie 7) 79
A.8 La spécification de la bibliothèque (partie 8) 81
A.9 La spécification de la bibliothèque (partie 9) 81
A.10 La spécification de la bibliothèque (partie 10) 82
A.11 Spécification des propriétés en ALLOY (partie 1) 84
A.12 Spécification des propriétés en ALLOY (partie 2) 84
A.13 Spécification des propriétés en ALLOY (partie 3) 86
A.14 Spécification des propriétés en ALLOY (partie 4) 87
A.15 Spécification des propriétés en ALLOY (partie 5) 88
A.16 La propriété 14 en ALLOY avec un état initial (partie 1) 89
A.17 La propriété 14 en ALLOY avec un état initial (partie 2) 90
A.18 La propriété 14 en ALLOY avec un état initial (partie 3) 90
A.19 La propriété 14 en ALLOY sans un état initial prédéfini (partie 1) 92
A.20 La propriété 14 en ALLOY sans un état initial prédéfini (partie 2) 92
A.21 La propriété 14 en ALLOY sans un état initial prédéfini (partie 3) 93
A.22 La propriété 14 en ALLOY avec un état initial prédéfini (partie 4) 94

Introduction

Contexte

« L'information, c'est le pouvoir »

Dans une économie du savoir, les informations sont devenues de véritables actifs stratégiques pour les entreprises en quête de performance. Depuis plusieurs années déjà, nombreuses sont celles à avoir investi massivement dans l'acquisition de systèmes d'information efficaces afin de mieux collecter, traiter et diffuser les informations dont elles disposent à l'interne. Mais plus que l'efficacité, c'est se doter d'un système d'information fiable, sans failles, qui est devenu un véritable enjeu pour ces entreprises, car de nombreux cas d'instabilité informatique ont prouvé que des brèches au niveau de la sécurité des systèmes pouvaient se traduire par des pertes financières (p.ex. affaire Kerviel en France) et humaines (p.ex. affaire de l'institut contre le cancer à la ville de Panama) aussi colossales que dramatiques.

Le propos de ce mémoire s'inscrit précisément dans le cadre de la problématique de sécurité fonctionnelle généralement posée par les systèmes d'information.

Motivations et problématique

Le laboratoire du GRIL (Groupe de recherche en ingénierie du logiciel), à l'Université de Sherbrooke, a créé pour les systèmes d'information une méthode formelle baptisée APIS (Automatic Production of Information Systems) [22] . Elle utilise la méthode EB^3, elle aussi développée au laboratoire du GRIL, pour modéliser les systèmes d'information. À ce stade, les méthodes de validation doivent être implémentées afin de vérifier si les spécifications

EB^{3}sont conformes aux exigences formulées par le client. Pour ce faire, deux possibilités s'offrent aux chercheurs : soit de développer un nouvel outil *ad hoc* taillé sur mesure ou soit d'utiliser des outils déjà existants. Puisque la première alternative est à l'évidence plus « coûteuse » en ressources (temps et personnes) que la seconde, c'est donc la deuxième voie que nous avons choisie dans le cadre de la présente recherche. Ce faisant, au gré de nos explorations préliminaires de la littérature du domaine de vérification de modèles, nous avons remarqué d'emblée une absence assez notable d'études comparatives portant sur les vérificateurs de modèles, justifiant *ipso facto* la nécessité de mener une étude à la fois pionnière et approfondie sur le sujet. À cette fin, notre groupe de chercheurs a donc choisi de polariser son attention sur la comparaison de six des principaux vérificateurs de modèles : ALLOY, SPIN, NUSMV, PROB, CADP et FDR2.

Le corpus du présent mémoire se scinde en deux phases successives : dans un premier temps, le rapport se concentre sur l'évaluation, puis le développement parallèle, d'idiomes servant à exprimer certaines des propriétés les plus répandues au sein des systèmes d'information ; dans un second temps, et sur la base des résultats obtenus à l'issue de cette évaluation préliminaire, l'étude se focalise sur l'utilisation d'un des vérificateurs jugés les mieux adaptés aux exigences d'un contexte d'application particulier des systèmes d'information, celui de l'informatique ubiquitaire (*pervasive computing*). Cette étape a pour but de faciliter le rajout d'une couche de sécurité au processus de développement habituel du domaine de l'ubiquité numérique. Ce travail est réalisé en collaboration avec le laboratoire DOMUS (domotique et informatique mobile de l'Université de Sherbrooke).

Méthodologie

Tel que susmentionné, la méthodologie adoptée dans le cadre de la présente étude se déroule en deux phases successives : dans une première phase, nous tentons d'établir un profilage technique de l'outil ALLOY afin de mieux comprendre les fonctionnalités qu'il intègre et identifier les atouts et limitations majeurs qu'il possède. La raison qui justifie la nécessité d'identifier les limitations du vérificateur réside dans le fait que celles-ci mettent au clair les propriétés difficilement vérifiables, condition *sine qua non* au développement ultérieur des idiomes de vérification visant à les valider. Une fois complétée, cette étape vise finalement à déterminer qui de ALLOY ou d'un des autres vérificateurs à l'étude dé-

montre la meilleure adaptabilité pour les systèmes d'information. Afin de corroborer ces conclusions, la seconde phase de l'étude se concentre quant à elle sur l'étude de ALLOY dans un contexte de développement bien particulier : celui de l'informatique ubiquitaire. Par l'entremise d'une étude de cas, l'objectif de cette seconde étape vise à évaluer la capacité du vérificateur à répondre aux diverses problématiques rencontrées dans le domaine de l'informatique ubiquitaire. Une fois compilées, les observations collectées à l'issue de cette étape doivent non seulement nous aider à préciser où se situent concrètement les diverses forces et faiblesses de ALLOY, mais aussi et surtout à confirmer (ou infirmer) si le fait de l'avoir privilégié à d'autres vérificateurs dès la première étape était un choix pertinent ou pas.

Résultat

En définitive, l'étude approfondie de ALLOY permet de mieux en comprendre les fonctionnalités diverses en même temps qu'elle facilite l'identification de ses principales forces et faiblesses. Globalement, et en raison de ses atouts majeurs (syntaxe proche de la programmation orientée objet, structure basée sur la logique du premier ordre, analyse automatisée), ALLOY peut être considéré au côté de PROB comme l'un des deux meilleurs vérificateurs qui soient dans le cas des systèmes d'information. On remarque entre autres que le vérificateur permet d'écrire les propriétés de type « préservation d'invariant » ainsi que la plupart de celles de type LTL [7], celles concernant les systèmes d'information, assez facilement, même si cette facilité varie grandement selon que la propriété est en soi difficilement spécifiable ou pas. Dans le cas des propriétés CTL [7] notamment, ALLOY révèle ses limites, la structure de ce genre de propriétés semblant difficilement, voire parfois même carrément non vérifiable : parmi elles, les propriétés de type réinitialisation (largement utilisées dans le cadre des systèmes d'information), exigent par exemple le développement d'un processus complexe en ALLOY afin de pouvoir être vérifiées. Une comparaison entre deux différentes manières de spécifier les traces a d'ailleurs facilité la compréhension des avantages et des inconvénients d'une telle approche. Toutes les propriétés de type LTL ou CTL ont été exprimées en logique du premier ordre en se basant sur les traces ou les arbres d'exécution. Enfin, dans le domaine de l'informatique ubiquitaire, on observe que ALLOY donne de bons résultats à l'exception près que la spécification des actions et des scénarios

d'exécution qui y prend place est souvent longue. Pour contourner ce problème, nous suggérons donc l'utilisation d'une version dynamique de ALLOY, DYNALLOY, en raison de sa meilleure adaptabilité aux contingences propres au contexte de « l'ubiquité numérique ».

Structure du mémoire

Le reste du mémoire est organisé comme suit. Le premier chapitre introduit les principes fondamentaux du fonctionnement de l'analyseur ALLOY. Le deuxième chapitre dresse une comparaison de six vérificateurs de modèles, dont l'analyseur ALLOY. Le troisième chapitre illustre dans une première partie l'usage des vérificateurs de modèles dans le domaine de l'informatique ubiquitaire, ainsi, en deuxième partie, le chapitre brosse un tableau des différents avantages et limitations de l'usage de l'analyseur ALLOY dans ce domaine. Enfin, le dernier chapitre présente un ensemble d'idiomes qui facilitent l'écriture de certains types de propriétés.

Chapitre 1

La vérification de modèles et ALLOY

Ce chapitre a pour but de présenter les principes et la théorie de la vérification de modèles. Ce chapitre commence avec une présentation des principes et la théorie de la vérification de modèles ainsi que le langage/analyseur/vérificateur ALLOY. Après l'introduction, le chapitre explique le fonctionnement interne de cet analyseur en citant ses outils internes et externes. Une fois cette phase achevée, le chapitre continue avec une description d'un idiome de spécification et de vérification de modèles pour finir avec un résumé des limitations de cet outil.

1.1 Présentation de la vérification de modèles

La vérification de modèles est un processus qui assure la satisfaction d'une formule logique dans un modèle abstrait. Autrement dit, elle répond à la question suivante : un système M satisfait-il une propriété F ? En termes logiques : M est-il un modèle de F ? La vérification de modèles consiste donc dans les grandes lignes à explorer exhaustivement l'espace d'états du modèle afin de vérifier qu'il n'existe pas de contre-exemple des formules (propriétés) à vérifier. Edmund M. Clarke et E. Allen Emerson ont introduit les premières notions de vérification de modèles avec des formules de logique temporelle en 1981 [14], suivies par les études de Jean-Pierre Queille et Joseph Sifakis en 1982 [48, 49]. Certes, après trente ans de recherche, le problème de l'explosion de l'espace d'états reste toujours

un défi scientifique. Cependant, au fil des années, l'optimisation et la création de nouvelles techniques ont permis de réduire considérablement le temps de traitement et d'intégrer de plus en plus la vérification de modèles dans le processus de développement des entreprises.

Dans le monde de la vérification de modèles, on peut distinguer deux grandes familles : la vérification de modèles symboliques et la vérification de modèles explicites. La première approche est en grande partie basée sur les diagrammes de décision (BDD) : elle consiste donc à représenter le modèle sous la forme de formules booléennes, ce qui entraîne le problème d'ordonnancement des variables. La seconde approche passe quant à elle par des représentations intermédiaires (p.ex. réseaux de Pétri, automates) afin de réduire la taille du graphe qui représente le modèle.

1.2 Introduction à ALLOY

ALLOY est un vérificateur/analyseur de modèles développé au MIT (*Massachusetts Institute of Technology*) par le professeur Daniel Jackson et ses étudiants du SDG (*Software Design Group*). De sa première version, *Nitpick*, ayant vu le jour en 1995, à sa version actuelle, ALLOY 4.0, l'outil a évolué considérablement, tant dans le fond que de la forme, en conservant toutefois la même base conceptuelle, celle de la logique du premier ordre. En effet, dans ALLOY les relations sont utilisées pour présenter les types de données, les structures et le temps. Pour analyser le modèle, autrement dit, pour vérifier sa cohérence et trouver des contre-exemples à des propriétés à vérifier, ALLOY utilise une des implémentations de la théorie du SAT [37]. L'outil propose plusieurs types et structures de base comme les entiers, les entiers naturels et les booléens. Il offre par ailleurs certaines structures de base sous forme de modules, comme par exemple les séquences, rendant ainsi la tâche de modélisation plus facile, plus rapide et automatisée.

1.3 Le fonctionnement interne de ALLOY

Les axiomes, le modèle et la cohérence sont les éléments clés encadrant le fonctionnement de ALLOY. En effet, ALLOY nécessite à l'entrée une spécification qui se définit comme un ensemble de signatures et de formules appelées axiomes. Une fois la spécification formulée, ALLOY permet dans un premier temps de vérifier si les formules F sont cohérentes

en cherchant un modèle M qui les satisfasse toutes (en d'autres termes, pour une représentation concrète de chaque signature donnée par M, toutes les formules de F sont satisfaites) puis, dans un second temps, de vérifier par approximation si une propriété est satisfaite dans tous les modèles de F. Comme cela est en général impossible à vérifier exhaustivement, ALLOY « inverse » la problématique en cherchant un modèle M « imitant » le modèle F qui arrive à contredire la propriété F'. Dans l'éventualité où il en trouve un, ceci signifie que la propriété F' n'est pas valide dans l'ensemble des modèles. Dans l'éventualité où il n'en trouve pas, puisque l'utilisateur définit la taille maximale du modèle à chercher, on ne peut toujours pas conclure que la propriété soit satisfaite dans tous les modèles. Pour mettre en œuvre ces fonctionnalités, ALLOY offre deux commandes principales :

- `run` : commande permettant de chercher une instance du modèle qui satisfait les contraintes imposées par sa spécification. Cette commande est essentiellement utilisée pour assurer la cohérence du modèle, c'est-à-dire que les formules du modèle ne se contredisent pas.
- `check` : commande permettant de chercher un contre-exemple d'une propriété que l'on désire satisfaire dans tous les modèles de la spécification.

À chaque utilisation de ces deux commandes, ALLOY fait appel à *Kodkod* [56], un outil qui permet la traduction automatique d'un modèle relationnel en modèle booléen dans une forme normale conjonctive (FNC) [37]. Ensuite, ces formules sont passées au SAT-solver qui, à son tour, fait le traitement nécessaire pour renvoyer des solutions sous FNC.Finalement, celles-ci sont traduites encore une fois par *Kodkod* et transférées à l'utilisateur via l'interface de ALLOY. La figure 1.1 résume le fonctionnement interne de ALLOY.

La spécification 1.1 basé sur l'histoire d'*Adam* et d'*Eve* illustre certains aspects de ALLOY.

1.4 Les outils de ALLOY

ALLOY possède deux types d'outils : internes et externes. Les outils internes permettent l'analyse (cohérence du modèle) et la vérification de propriétés sur le modèle et sont totalement transparents à l'utilisateur. Parmi eux, on retrouve, entre autres, les SAT-solvers, *Kodkod* et Unsat-core [16], un outil utilisé pour identifier les formules qui causent la non-satisfaction/incohérence dans le modèle. À l'inverse, les outils externes sont visibles par

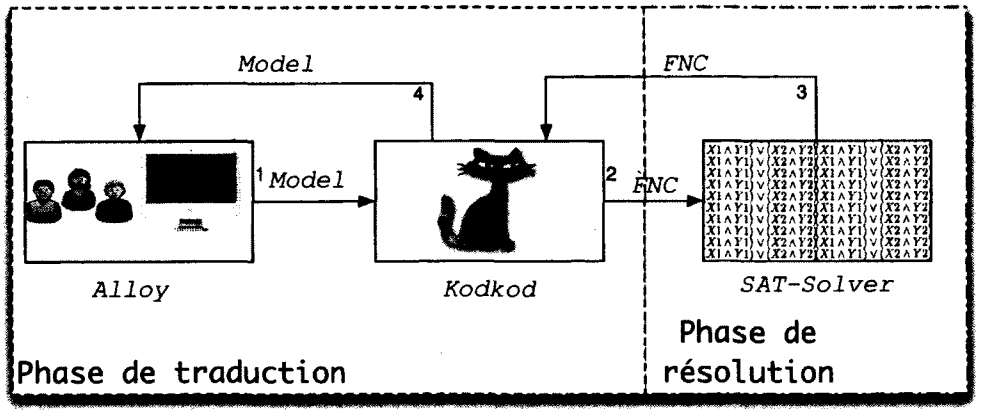


figure 1.1 – Le processus du traitement interne de ALLOY

l'utilisateur, lui permettant d'évaluer les instances du modèle et les contre-exemples des propriétés. Parmi eux, on retrouve notamment *Evaluator* et *dot* [1]. Alors que le premier permet d'évaluer les formules en rapport avec la réponse obtenue après l'évaluation/réponse de l'analyseur (en demandant par exemple la cardinalité d'un ensemble ou l'évaluation d'un prédicat sur une transition du système trouvé), le second offre la possibilité de représenter automatiquement une structure d'information sous la forme d'un graphe (autrement dit, il donne une représentation graphique des instances et des contre-exemples obtenus).

1.5 Avantages et limitations

Parmi les avantages qui distinguent ALLOY des autres vérificateurs de modèles, on peut citer, entre autres, sa notation proche de celle de la plupart des langages orientés objet, son meilleur temps de traitement des opérations de vérification [21] ainsi que son intégration de modules de base et d'outils de visualisation et d'évaluation. Bien que ces caractéristiques représentent de réels atouts pour l'utilisateur, ALLOY souffre de certaines limites qui circonscrivent parfois son champ d'application. Parmi elles, le vérificateur possède deux limitations majeures : une limitation absolue et une limitation réelle. La limitation absolue correspond au fait que le nombre d'atomes, une entité ayant une définition très proche à celle de l'objet, est limité à 2^{32}, ce qui dépend en grande partie de la taille et du nombre de

1. http://www.graphviz.org/Documentation.php

```
-- model entities definition.
sig Person
{
--lone : 0 .. 1
father : lone Men,
mother : lone Women,
--set : 0 .. n
children : set Person,
brother : set Person,
sister : set Person,
}
sig Men extends Person
{
wife : lone Women
}
one sig Adam extends Men {}
sig Women extends Person
{
husband : lone Men
}
one sig Eve extends Women {}
-- axioms
fact
{
some mother.Eve
some father.Adam
-- Eve has no mother and no father
no Eve.mother
no Eve.father
no Adam.mother
no Adam.father
-- Eve's husband is Adam
Eve.husband = Adam
Adam.wife = Eve
...
}
-- generate a model satisfying the signatures and the axioms.
pred show []{}
run show for 4
```

prog 1.1 – Exemple illustrant quelque aspects de ALLOY

produits cartésiens dans le modèle. Par exemple, pour une relation binaire (une signature qui contient un champ unaire), le nombre d'atomes ne peut pas dépasser 2^{16}, et pour une relation ternaire (une signature qui contient un champ binaire), le nombre d'atomes ne peut pas dépasser 2^{10}, et ainsi de suite. La limitation absolue est une limitation posée par la structure interne des données de ALLOY. En revanche, la limitation réelle dépend entièrement de la complexité du modèle. Si le modèle est très complexe, ALLOY risque d'être à court de mémoire et le traitement est alors arrêté. La figure 1.2 présente un exemple illustrant chacune de ces limitations.

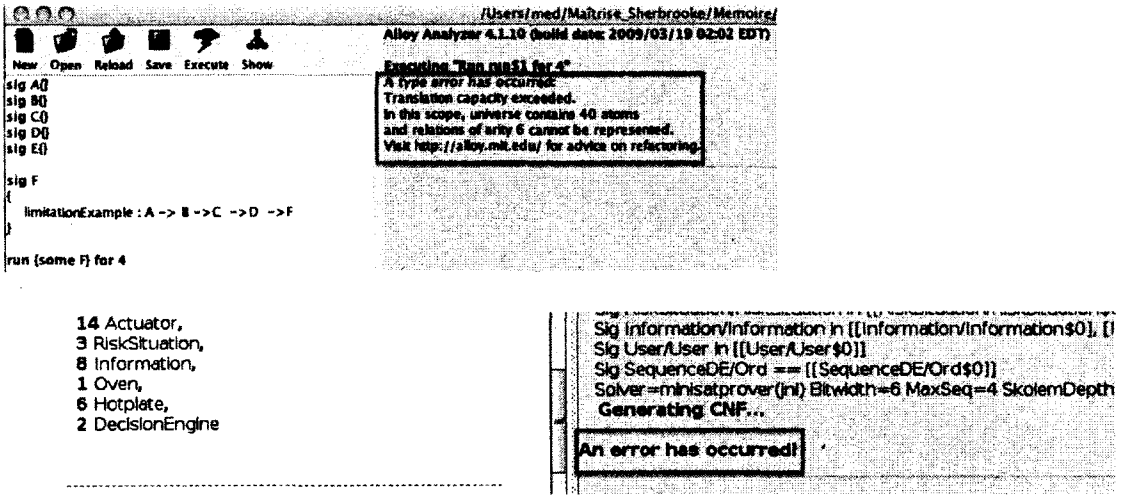


figure 1.2 – Exemples de la limitation réelle et de la limitation absolue

1.6 Processus du développement d'un modèle en ALLOY

Faire le lien entre les langages de programmation et les analyseurs de modèles semble difficile de prime abord en raison du temps de développement du modèle. C'est pourquoi, il appert important de définir un idiome qui décrit la façon optimale de réaliser un développement sûr. Cette technique consiste à écrire le modèle d'une manière incrémentale en s'assurant à chaque fois que sa cohérence reste intacte. La logique de cette approche au « pas-à-pas » repose sur le fait que plus le modèle devient grand, plus le temps de détection des éventuelles incohérences augmente. Dans l'éventualité où une incohérence est détectée, l'utilisateur est alors confronté à deux alternatives possibles : la première est d'utiliser le

Unsat-core et la seconde est d'éliminer les assertions d'une façon itérative, et ce, dans l'optique conjointe de mettre le doigt sur ce qui cause l'incohérence. Une fois que cette phase est achevée et que tous les prédicats sont cohérents, l'utilisateur peut passer à la deuxième étape de l'analyse, à savoir : celle de la vérification des propriétés. La figure 1.3 résume le fonctionnement de ce processus de vérification incrémentale et itérative.

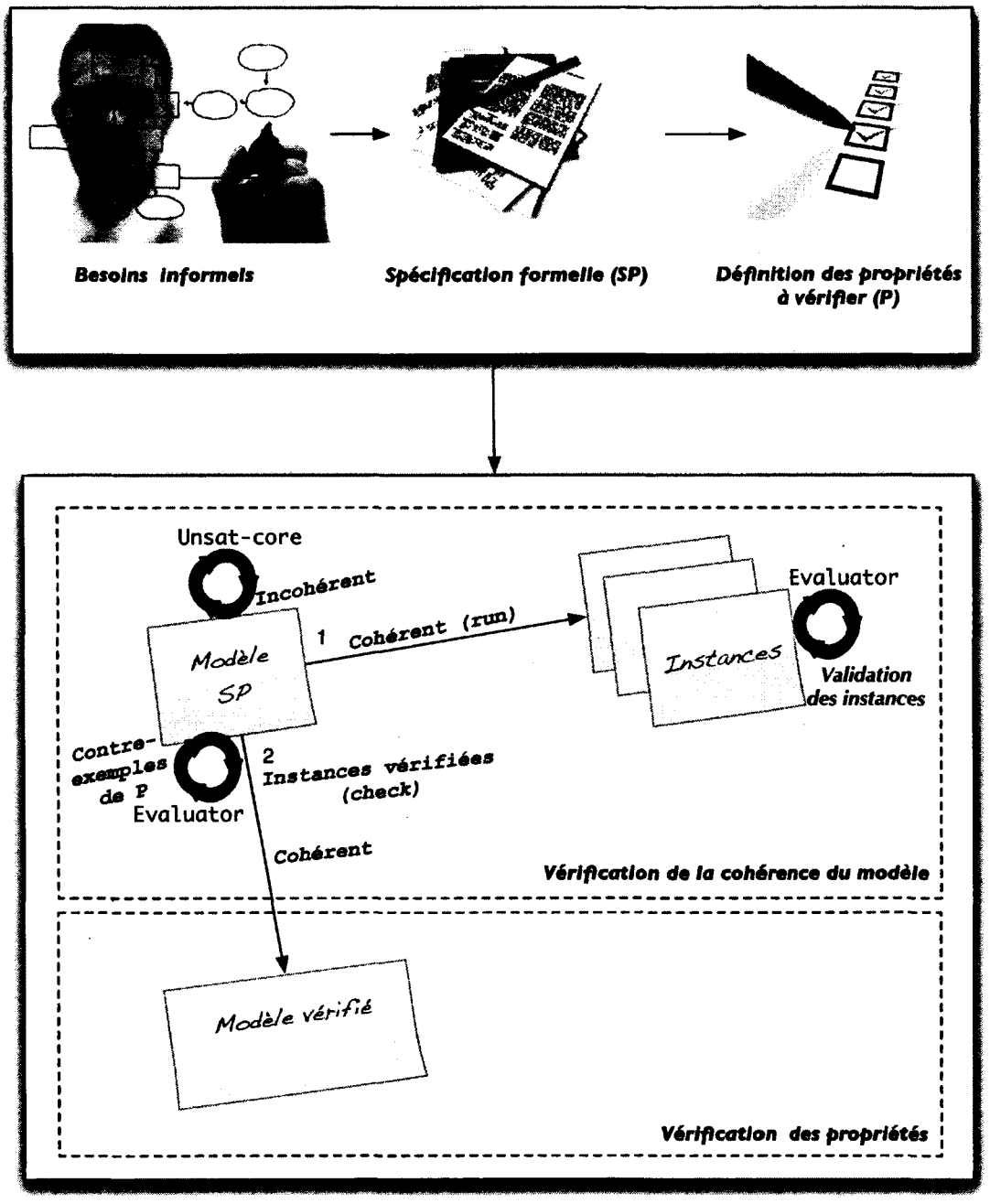


figure 1.3 – Idiome du développement d'un modèle en ALLOY

Chapitre 2

Comparaison de six vérificateurs de modèles

Résumé

Cet article présente une comparaison de six vérificateurs de modèles sur un cas d'étude qui est celui de la bibliothèque. Cette étude vise à mettre en évidence les outils les plus adaptés à la vérification de systèmes d'information dans une approche d'ingénierie dirigée par les méthodes.

Commentaires

Cet article a été publié à la conférence ICFEM 2010 qui a eu lieu à Shanghai en novembre 2010. Ma participation à l'article concerne toutes les parties qui sont en rapport à ALLOY, ainsi que les tests effectués sur le serveur du département pour chacun des six vérificateurs de modèles comparés.

Comparison of Model Checking Tools for Information Systems

Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, Mohammed Ouenzar
Département d'informatique, Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1

Abstract

This paper compares six model checkers (ALLOY, CADP, FDR2, NUSMV, PROB, SPIN) for the validation of information system specifications. The same case study (a library system) is specified using each model checker. Fifteen properties of various types are checked using temporal logics (CTL and LTL), first-order logic and failure-divergence (FDR2). Three characteristics are evaluated: ease of specifying information system i) behavior, ii) properties, and iii) the number of IS entity instances that can be checked. The paper then identifies the most suitable features required to validate information systems using a model checker.

2.1 Introduction

Information systems (IS) now play a prominent role in our society to support business processes and share organisational data. Yet, even if they are one of the early application domains of computing, their development relies mostly upon a manual and informal process. The problem addressed in this paper is the formal *validation* of IS specifications using model checking. Model checking is an interesting technique for IS specification validation for several reasons: it provides broader coverage than simulation or testing, it requires less human interaction than theorem proving, and it has the ability to easily deal with both safety and liveness properties.

The validation of IS specification is of particular interest in model-driven engineering (MDE) and generative programming, which aim at synthesizing an implementation of a system from models (i.e., specifications). Hence, if the synthesis algorithms are correct, one only needs to validate the models to produce correct systems. IS MDE specification languages usually do not have any dedicated model checker. Since developing a model checker is a long process and since several model checkers already exist, it is simpler to choose an existing tool that is maintained by a team specialized in the model checking field. In this paper, we compare six model checkers: SPIN [30], NUSMV [9], FDR2 [50], CADP [24], ALLOY [32] and PROB [35], which are representative of the main classes of model checkers: explicit state, symbolic, bounded and constraint satisfaction. The comparison is based on a single case study which is representative of IS structure and properties. Our comparison aims at answering the following questions.

1. Is the modeling language of the model checker adapted for the specification of IS models? This is especially important in the context of MDE IS, since it must be straightforward to automatically translate an IS MDE specification into the language of the model checker.
2. Is the property specification language adapted to specify IS properties?
3. Is the model checker capable of checking a sufficient number of instances of IS entities?

Our case study focuses on the control part of an IS, which determines the sequences of actions that the IS must accept. Validation of input-output behavior (data queries) and user interactions with graphical user interface are omitted.

This paper is structured as follows. Section 2.2 presents a synthesis of related work on model checking of IS. Section 2.3 presents a description of the case study, a library IS. Section 2.4 provides an overview of the model checkers, comparing relevant points. The modeling and verification process of the IS for each tool is provided in Sect. 2.5. Then, the analysis of processes and the model checking results for the case study are presented in Sect. 3.6. Finally, we conclude in Sect. 2.5.

2.2 Related Work

There is an extensive literature on model checking. This section focuses on comparative studies of model checkers related to IS. Model checking has been extensively applied to business process modelling. Yeung [58] proposes a framework to analyse *suspendible business* transactions modelling with statecharts and CSP [50]. It is applied to a library specification similar to the one studied in this paper. However, the paper does not actually experiment with model checkers to check business process. In [5], a travel agency business process has been modelled with SPIN and PROMELA, and CIA and CSP (LP). Safety properties and deadlocks have been successfully verified with both model checkers; reachability properties have not been tested. The authors propose an extension of CSP with notion of "compensation" (a behavior to compensate a process failure). In [43], business processes are converted from BPEL to automata, but also to Petri nets and CSP and LOTOS [10]. It concludes that process algebras are suitable for verification of the reliability of IS, in the particular case of business process. In [15], the authors study the verification of data-driven applications in the particular case of web-based systems using an ASM-like [52] specification language. The study focuses particularly on reachability properties, but any type of property can be used for modelling. The modelling process is complex and demands significant expertise. Both modelling techniques give an insight on what can be done with these subclasses of IS. In [6], four state-based model modeling techniques with their model checkers (USE, Alloy, ZLive and ProZ) are compared along four criteria: animation, generation of pre and postconditions, execution analysis and expertise. The study mostly checks invariant properties.

Each of these studies provides partial answers to our questions, for a subset of model checkers, using different case studies and properties. This makes it difficult to compare

model checkers and identify the one best suited for IS validation.

2.3 Presentation of the Case Study

This section presents the user requirements of a library system which is used for the formal verification of properties. In order to avoid any confusion, key concepts are first defined. *Lending* a book means that a user borrows a book without reserving it beforehand. *Taking* a book means borrowing a book after having reserved it. *Borrowing* a book denotes either *taking* or *lending* it. In the requirements list, a *member* is a person who has *joined* (and still not *left*) library membership.

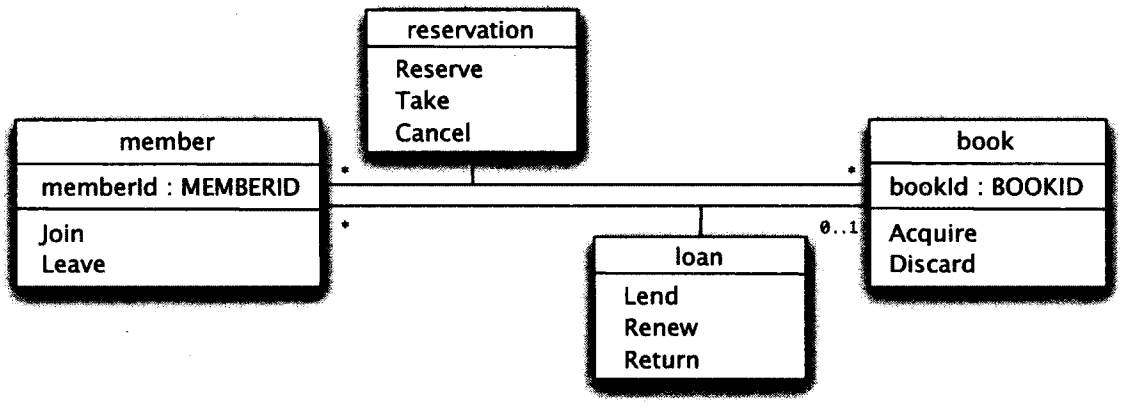


Figure 2.1: Requirement class diagram of the library system

The requirement class diagram corresponding to the model is given in Figure 2.1. Entity attributes are listed in the upper part of each class, while entity actions are listed in the lower part. The library system only contains two entity types: **books** and **members**. **Members** can `Join` and then `Leave` library membership whereas **books** can be `Acquired` and then `Discarded`. **Members** can `Lend`, `Renew` several times and `Return` a **book**. They can also `Reserve` a **book** under certain conditions (e.g. if it cannot be lent at that moment), and then, either `Cancel` the reservation or `Take` the **book**. Hence the library system contains 10 actions.

The following list describes the properties that we verify using the model checkers.

1. A book can always be acquired by the library when it is not currently acquired.

2. A book cannot be acquired by the library if it is already acquired.
3. An acquired book can be discarded only if it is neither borrowed nor reserved.
4. A person must be a member of the library in order to borrow a book.
5. A book can be reserved only if it has been borrowed or already reserved by another member.
6. A book cannot be reserved by the member who is borrowing it.
7. A book cannot be reserved by a member who is reserving it.
8. A book cannot be lent to a member if it is reserved.
9. A member cannot renew a loan if the book is reserved.
10. A member is allowed to take a reserved book only if he owns the oldest reservation.
11. A book can be taken only if it is not borrowed.
12. A member who has reserved a book can cancel the reservation at anytime before he takes it.
13. A member can relinquish library membership only when all his loans have been returned and all his reservations have either been used or canceled.
14. Ultimately, there is always a procedure that enables a member to leave the library.
15. A member cannot borrow more than the loan limit defined at the system level for all users.

In the context of IS, one usually distinguishes two types of properties. The first one is called a *liveness* property. It represents the fact that the system is still alive, i.e. not stuck in a deadlock (the system is blocked in a single state) or a livelock (the system loops in a subset of states considered as non-evolving). They can also express the fact that an action implies a reaction from the system; the latter is however rarely used in information systems, where actions are human-driven (one cannot force a user to trigger specific actions). Properties 1, 12 and 14 are liveness properties. In IS, liveness properties usually describe *sufficient conditions* to *enable* an action (immediately or sometime in the future). For instance, Property 1 states that the library *has the right* to acquire the book (under certain conditions). In other terms, it forces the IS to allow the action, but the user is not forced to invoke this action. The other properties are called *safety* properties. They usually describe *necessary conditions* to enable an action, or what a user is *not allowed* to do with

the system at a given point. The remaining properties are safety properties. A third type of properties is usually distinguished from these two. These are *fairness* properties, but as IS users cannot be forced to do some action, they seldom occur in IS specifications. Fairness is not considered in this study.

2.4 An Overview of the Model Checkers

Four large families of model checkers are considered. *Explicit state* model checkers, like CADP, SPIN and FDR2, use an explicit representation of the transition system associated to a model specification. This transition system is either computed prior to property verification, as in CADP and FDR2, or on-the-fly while checking a property, as in SPIN (also possible in some cases with CADP and FDR2). *Symbolic* model checkers, like NUSMV, represent the transition system as a Boolean formula. *Bounded* model checkers, like NUSMV and ALLOY (indirectly), consider traces, of a maximal length k, of the transition system and represent them using a Boolean formula. *Constraint satisfaction* model checkers, like PROB, use logic programming to verify formula. SPIN, CADP, NUSMV and PROB support temporal languages (LTL [47], CTL [17] and XTL [38]) for property specification. ALLOY and FDR2 use the same language for both model specification and property specification (first-order logic and CSP, respectively).

2.4.1 SPIN

SPIN was one the first model checker developed, starting in 1980. It introduced the classical approach for on-the-fly LTL model checking. Specifications are written in Promela and properties in LTL. An LTL property is compiled in a Promela `never` claim, i.e. a Büchi automaton. SPIN generates the C source code of an on-the-fly verifier. Once compiled, this verifier checks if the property is satisfied by the model. Working on-the-fly means that SPIN avoids the construction of a global state transition graph. However, it implies that transitions are (re-)computed for each property to verify. Hence, if there are n properties to verify, a transition is potentially computed n times, depending on optimizations.

PROMELA, the model specification language of SPIN, is inspired from C. Hence, it is an imperative language, with constructs to handle concurrent processes. State variables can

be global and accessed by any process. PROMELA offers basic types like `char`, `bit`, `int` and arrays of these types. Processes can communicate by writing and reading over a *channel*, either synchronously using a channel of length 0, or asynchronously, using a channel of length greater than 0. Operator `atomic` allows a compound statement to be considered as a single atomic transition, except when this compound contains a blocking statement, such as a guard or a blocking write or read over a channel, in which case the execution of the `atomic` construct can be interrupted and control transferred to another process. Statements can be labeled and these labels can be used in LTL formulae.

SPIN uses propositional LTL, with its traditional operators *always*, *eventually* and *until*. The latter is sometimes referred as the "strong until" operator, as opposed to the "weak until" operator. The *next* operator is not allowed to ensure that partial order reduction can be used during the model checking. An LTL formula can refer to labels and state variable values of a PROMELA specification. SPIN only considers states; there is no notion of event on a transition. An LTL formula holds for a PROMELA specification if and only if it holds for every possible run of the PROMELA model. A run is an execution trace consisting of the sequence of states visited during execution. It can be infinite.

2.4.2 NUSMV

NUSMV is a model checker based on the SMV (Symbolic Model Verifier) software, which was the first implementation of the methodology called *Symbolic Model Checking* described in [39]. This class of model checkers verifies temporal logic properties in finite state systems with "implicit" techniques. NUSMV uses a symbolic representation of the specification in order to check a model against a property. Originally, SMV was a tool for checking CTL properties on a symbolic model. But NUSMV is also able to deal with LTL (+Past) formulae and SAT-based *Bounded Model Checking*. The model checker allows to write properties specification both in CTL or LTL and to choose between BDD-based symbolic model checking and bounded model checking.

NUSMV uses the SMV decription language to specify finite state machines. A specification consists of module declarations and each module may include variable declarations and constraints. System transitions are modelled by assignment contraints or transition constraints, which define next values for declared variables in a module. An assignment

gives explicitly a value for a variable in the next step, while a transition constraint, given by a boolean formula, restricts the set of potential next values. Each module can be instantiated by another one, for example by the main module, as a local variable. In fact, each instance of a module is by default processed synchronously with the others during an execution. But NuSMV can also model interleaving concurrency by using the "process" keyword in module instantiation. To get different instances of module, instantiations can be parameterized. However, the description language is quite low-level. All assignments, parameters or array indexes have to be constant. Thus, specifications may be longer than in PROMELA, because each case has to be explicitly written. As NuSMV modules can declare state variables and input variables, an SMV specification can be both state or event oriented. Input variables are used to label incoming transitions and their values can only be determined by specifying transition constraints.

CTL properties can only be expressed by using state variables, but NuSMV allows to use input variables and state variables in LTL specifications. Moreover, NuSMV can also check invariant properties, which can be written in a temporal manner as *Always p* where *p* is a boolean formula. Invariant specifications are checked by a specialized algorithm during reachability analysis, that gives a result faster than CTL or LTL algorithms.

2.4.3 FDR2

FDR2 is an explicit state model checker for CSP, the well known process algebra. FDR2 can check refinement, deadlocks, livelocks and determinacy of process expressions. It gradually builds the state-transition graph, compressing it using state-space reduction techniques, while checking properties, which also makes it an *implicit* state model checker.

Models are described using a variant of CSP, called CSP_M. It supports classical process algebra operators like prefix, choice, parallel composition with synchronisation, sequential composition and guards. Quantified versions of choice, parallel composition and sequential are supported. FDR2 supports basic data types like integer, boolean, tuples, sets and sequences. Lambda terms can be used to define functions on these types. Expressions are dynamically typed (except for actions, called channels in CSP, which are declared and typed). CSP does not support state variables; however, they can be simulated to some extent by using a recursive process with parameters.

Properties are expressed as CSP processes. They are checked using process refinement. FDR2 supports three refinement relations: $\sqsubseteq_T$ (trace refinement), $\sqsubseteq_F$ (stable-failure refinement) and $\sqsubseteq_{FD}$ (stable-failure-divergence refinement). We say that $P \sqsubseteq_T Q$ iff the traces of Q are included in the traces of P. A trace of a process P is a sequence of visible events that P can execute. We say that $P \sqsubseteq_F Q$ iff the failures of P are included in the failures of Q. A failure (t, S) of a process P denotes the set of events S that P can refuse after executing trace t. Trace refinement is used to check safety properties, while stable failure is used to check liveness (or reachability) properties. Failure-divergence refinement is used to check livelocks (infinite loops on internal actions), which are not relevant for our case study.

2.4.4 CADP

CADP is a rich and modular toolbox. We have selected LOTOS-NT to specify models and XTL to specify properties. The XTL model checker takes as input a labelled transition system (LTS), encoded in the BCG (*Binary Coded Graph*) format. LOTOS-NT is a variant of LOTOS that supports local states variables. A LOTOS-NT specification is translated into a LTS using Caesar. This LTS is minimized into a trace equivalent LTS. Finally, properties written in XTL are checked against this LTS using the XTL model checker.

LOTOS-NT is inspired from LOTOS. A LOTOS-NT specification is divided into two complementary parts: an algebraic specification of the abstract data types and a process expression. LOTOS-NT offers traditional process algebra operators like sequence, choice, loop, guard and parallel synchronization. It supports state variables, which are local to a process and cannot be referred by another process. Assignment statements can be freely mixed with other process expression constructs.

XTL, the property specification language of CADP, is used to express temporal logic properties. XTL provides low-level operators which can be used to implement various temporal logics like HML, CTL, ACTL, LTAC, as well as the modal mu-calculus. XTL formulae are evaluated on a LTS. XTL allows one to refer to transitions (events) and values of their parameters. No LTL library is currently provided. In this paper, the CTL and HML libraries are used.

Since the LTS does not contain any state variable, the difficult part in writing XTL properties for LOTOS-NT models is to characterize states. Indeed, the specifier can only use

action labels to define particular states. The HML library, with its two handy operators *Dia* and *Box*, is used for this purpose. *Box*(a, p) holds in a given state if and only if every action matching action pattern a leads to a state matching state pattern p. On the other hand, *Dia*(a, p) holds for a given state if and only if there exists at least one action matching action pattern a leading to a state matching state pattern p. An XTL formula holds for a LTS if and only if it holds for all states of the LTS.

2.4.5 ALLOY

ALLOY is a symbolic model checker. Its modeling language is first-order logic with relations as the only type of terms. Basic sets and relations are defined using "signatures", a construct similar to classes in object-oriented programming languages, which supports inheritance. ALLOY uses SAT-solvers to verify the satisfiability of axioms defined in a model and to find counterexamples for properties (theorems) which should follow from these axioms.

An ALLOY specification consists of a set of signatures, noted (`sig`), which basically define sets and relations. Constraints, noted `fact`, are formulae which condition the values of sets and relations. The declaration `sig X {r : X -> Y}` declares a set `X` and a ternary relation `r` which is a subset of the Cartesian product $X \times X \times Y$. ALLOY supports usual operations on relations, like union, intersection, difference, join, transitive closure, domain and range restriction. Integer is the only predefined type. Cardinality constraints can be defined on relations (*e.g.*, injections and bijections). Properties are simply written as first-order formulae.

2.4.6 PROB

PROB is a model checking and an animation tool designed for the B Method [1]. Currently it also supports CSP_M, Z, and Event-B. This study uses the B Method and the B language.

B specifications are organized into abstract machines (similar to classes and modules). Each machine encapsulates state variables, an invariant constraining the state variables, and operations on the state variables. The invariant is a predicate in a simplified version of the ZF-set theory, enriched by many relational operators. In an abstract machine, it is

possible to declare abstract sets by giving their name without further details. This allows the actual definition of types to be deferred to implementation. Operations are specified in the Generalized Substitution Language, which is a generalization of Dijkstra's guarded command notation. Hence, operations are defined using substitutions, which are like assignment statements. A substitution provides the means for identifying which variables are modified by the operation, while avoiding mentioning those that are not. The generalization allows the definition of non-deterministic and preconditioning substitutions. The preconditioning substitution is of the form **PRE** P **THEN** S **END**, where P is a predicate and S a substitution. When P holds, the substitution is executed; otherwise, the result is undetermined and the substitution may abort.

Properties in ProB can be written in LTL, past LTL or CTL, hence combining the strengths of each language. In addition, ProB allows for the inclusion of first-order formulae in temporal formulae. It also offers two convenient operators for LTL. The first one, denoted by $e(A)$, checks if the action A is executable in a given state of a sequence. The second one, denoted by $[A]$ checks if A is the next operation in the sequence. Consequently ProB can express properties on either states or events.

2.5 Specifying the Model and the Properties

This section describes how the IS model and properties are specified with each model checker. For sake of conciseness, specifications are omitted. They are available in [13].

2.5.1 Spin

Two styles have been considered for the Spin specification. In the first style, there is only one process which loops over a choice between all actions. It was quickly abandoned, because it blows quite rapidly in terms of number of states. In the second style, there is one process for each instance of each entity and each association. The process describes the entity (or the association) life cycle. Therefore, the Promela specification of the case study contains four process definitions, one for each entity (book and member) and one for each association (loan and reservation). Each process definition is instantiated n_i times to model n_i instances of entity i, and $n_i * n_j$ times to model an association between entity i and j.

We use a producer-modifier-consumer pattern as the basis of a life cycle for an entity and an association. It can be represented by the following regular expression $P.M^*.C$ where P is the producer (for example `Acquire`), M is a modifier and C is a consumer (for example `Discard`). The concatenation operator "." of regular expressions can be represented by a semi-colon ";", the sequential composition operator of PROMELA, or an arrow "->" that denotes the same operator. Some events have a precondition which is not represented in the regular expression. For example, a book cannot be discarded if it is still borrowed. Consequently the execution of an event is guarded by a precondition.

When a member takes a book he has reserved, two associations are involved: the loan association and the reservation association. This leads to ensure that both processes execute the `take` event in an atomic step. It is not obvious and straightforward. To achieve an atomic step, the `take` event is split into two events: one in the reservation association process (as a consumer) and one in the loan reservation (as a producer). A channel with an empty capacity is used to ensure the handshake. This is a classic strategy in PROMELA. Nevertheless, the handshake cannot be made within an `atomic` instruction. This could break the local atomicity in the sender. But it could be used at the end of an `atomic` and at the beginning of another. In this way, no other process can be interleaved with the handshake of the two processes. The result is a pattern described in [13], in which an event is simultaneously the consumer of an association and the producer of another one.

In SPIN, the properties are expressed using LTL. Reachability properties are difficult to express in LTL. Fortunately, since event preconditions are explicitly written via labels in the specification, expressing a property like "a book can be acquired" is straightforward. Consequently, when a property asserts the possibility of an event execution, it is represented by a propositional formula in the LTL formula that uses a label of the process and, sometimes, the precondition of this event. For example, "the book `b0` can be acquired" is expressed as "process book `b0` is at the `discarded` label".

Property 14 is not expressible in LTL, since it is equivalent to a reset property. The reset property is known to be expressible in CTL only. LTL and CTL are complementary languages. The semantics of LTL formulae is defined on traces of the transition system, while the semantics of CTL formulae is defined on the transition system itself, which allows one to refer to the branching structure of the execution. Some properties can only be expressed in either LTL or CTL. For instance, a *reset* property, which states that it is always possible

to go back to some desired state, cannot be expressed in LTL, since this transition to reset does not have to occur in each possible run of the system. Since an LTL formula holds if and only if it holds for every possible run of the system, an LTL property would force this reset transition to occur in every run. Dually, a property of the form "eventually p always holds" cannot be expressed in CTL [19], due to the branching nature of the logic.

Two simple patterns can express almost 70 % of the requirements. In LTL and with the state-oriented paradigm, these patterns are expressed as "if action A can be executed then the state verifies P" or "if P holds then action A can be executed" where P is only true between B and C. Therefore the two main patterns are $\Box(\mathsf{can\_}A \Rightarrow P)$ or $\Box(P \Rightarrow \mathsf{can\_}A)$. Inexpressible properties cannot simply be considered as negligible. This is an important weakness of SPIN that cannot be overcome.

2.5.2 NuSMV

To model the library system example in an SMV specification, we use a systematic method based on the structure of the class diagram. Each class, that represents an object and has attributes, is encoded into a module containing variables and parameterized by a key to identify entities. Then, for each kind of action defined in the system, a new module is created, parameterized by class modules involved in that action. Action modules check that a given precondition is satisfied. Then, if the precondition holds, they modify variables of entities to apply postconditions using assignment constraints.

Properties of the library system can be expressed by CTL formulae on state variables, or by using LTL formulae with state variables or input variables. Specifications on state variables are close to Promela specifications, except that NuSMV can check CTL and LTL properties. This allows to easily express all requirements. Specifications on input variables are event-oriented. However, only LTL can be used to write event-oriented specifications in NuSMV.

Property 1 is a sufficient condition to enable an event. It is easily expressed as follows: `AG (!book1.is_acquired -> EX book1.is_acquired)`. Property 12 is specified in a similar manner, except that it must be repeated for each position in the array representing the reservation queue. Hence, the text of properties may linearly grow with the number of entity instances, an unfortunate limitation due to the restriction to constants in accessing

array positions. Property 14 is also very similar to 1, except that EF is used instead of EX. Properties expressing necessary conditions (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13) can be expressed in two different forms (state, event). For instance, property 2 is expressed using events, saying that a discard must always occur between two acquires:

```
G acquire1.do ->
      X((!acquire1.do U discard1.do) | G !acquire1.do)
```

2.5.3 FDR2

CSP is quite handy to explicitly represent IS entities life cycles. We use a nominal/controller pattern to specify the library case study. Each entity E is represented by a quantified interleave of the form $E = |||k : T \ @ \ E(k)$, where $E(k)$ models the life cycle of an instance k of entity E. The associations to which an entity E participates are represented in a similar manner, and called within $E(k)$. The global behavior is obtained by composing the entities in parallel: $Nominal = E_1 \parallel \ldots \parallel E_n$. Since CSP_M does not directly implement Hoare's parallel operator $\parallel$, we use CSP_M's operator $\overset{\parallel}{_X}$, where X denotes the association actions on which entities must synchronize. Some ordering constraints are represented using recursive processes to simulate state variables. The relevant state variables $\vec{v}$ of an entity E are represented by a recursive process $C_E(k, \vec{v})$ which offers a choice [] between actions to control, in the form $[]_i \ G \ \& \ a_i \rightarrow C_E(k, e(\vec{v}))$, where "&" denotes a guard operator with condition G which tests the values of $\vec{v}$. These control processes are composed in parallel with *Nominal*, synchronizing on a_i.

Safety properties are checked by trace refinement and are relatively easy to specify. Suppose that property p only involves actions a_i. One writes a process P that represents the traces on a_i satisfying p, and checks that the system Q, restricted to actions a_i, trace refines P as follows: $P \sqsubseteq_T Q \setminus (\Sigma \setminus \{a_i\})$, where $\setminus$ is the hiding operator of CSP_M and Σ denotes all actions of the specification and "–" is set difference.

Reachability properties 1, 12 and 14 are checked using stable-failure refinement, and are a bit more tricky to specify. For instance, property 1 states that a book can always be acquired, if it is not currently in the library. This is typically specified in CSP as follows: $P \,|||\, CHAOS(\{b_i\}) \sqsubseteq_F Q \setminus \{c_i\}$. Process P recursively loops over acquire and discard events:

acquire(b)→discard(b)→ $P(b)$. Essentially, P states that discard can never be refused after an acquire, and an acquire can never be refused after a discard. Hidden actions $\{c_i\}$ are those that unavoidably can occur between acquire and discard, *ie*, association actions. The interleave with $CHAOS(\{b_i\})$ states that other actions can occur before or after, but we do not really care about their order. Unfortunately, hiding association actions introduces unstable states, which weakens the specification of the property under stable-failure refinement. To make a short explanation, infinite internal action loops are introduced by hiding; hence some errors in the behavior of association actions are not detected by this form of property specification. To overcome this, we have to check each association in isolation, disabling events from the other associations, which is weaker than property 1. These are very subtle issues which are difficult to master. Reachability properties of IS specifications are far from trivial to specify in CSP.

2.5.4 CADP

The LOTOS-NT specification of the library system is similar in structure to the CSP specification already described. Since there is no quantified interleave operator in LOTOS-NT, one has to hardcode entities and associations interleaves, which means that the number of interleaves to hard code in the specification text grows exponentially with the number of entities, making verification experiments a bit cumbersome.

Safety properties of the case study are defined using two patterns. The first states that an action A should not happen between two actions B and C. For example, a member should not leave the library if he has reserved a book (i.e. between a `Reserve` and a `Take` or `Cancel`). The second pattern expresses the prohibition of an action A outside a sequence delimited by two actions B and C; it is illustrated by the fact that a member should not renew a book if he has not borrowed it yet (i.e. outside a `Lend` or `Take` and a `Return`). In XTL, one can represent these patterns using macros. They are defined using a weak until operator, defined by macro `AW_A_B`. These two patterns, respectively called `no_A_between_B_and_C` and `no_A_outside_B_and_C`, are used for properties (2, 3, 6, 7, 8, 9, 11, 13) and 4, respectively. Liveness properties are written directly with classic ACTL and HML operators, like properties 1, 12 and 14. No correct formulation has been found for properties 5 and 10. For property 5, one must characterize using events the states where

a book is *not borrowed nor reserved.* Property 10 involves a queue, which is as hard to describe using events.

2.5.5 ALLOY

Each IS entity E is represented by a signature E, which models the set of possible entity instances. System states are represented by a signature `sig S { e1 : E1, ..., en : En, a1 : Ei -> Ej, ...}`, where e_i models the active instances of E_i in a state, and a_i models the instances of association A_i. Each action is represented by a predicate $P[s : S, s' : S, p : T]$ relating a before-state s to an after-state s' for input parameters p. We have systematically followed a pattern for these predicates, which is a conjunction of a precondition, a postcondition and a "nochange" predicate that determines which attributes are unchanged by the action.

A property of the form "when condition C holds, action a must be executable" (*e.g.*, Property 1) is written as follows: $\forall s : S, p : T \cdot C \Rightarrow preA[s, p]$, where $preA[s, p]$ is the precondition of action $a(p)$. Similarly, if C is the result of executing an action $b(p)$ that should enable an action $a(p)$ (*e.g.*, Property 12), it can be written as $\forall s, s', p : T : S \cdot b[s, s', p] \Rightarrow preA[s', p]$. A property of the form "action a is executable only when condition C holds" (*e.g.*, Properties 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13) is written as: $\forall s : S \cdot preA[s, p] \Rightarrow C$. These three patterns are approximations of the property, because an action can be executed when its precondition holds and when $\exists s' : S \cdot postA[s, s', p]$ holds. In other words, the precondition must hold and the postcondition must be feasible. However, checking such an existential formula over IS states in ALLOY is usually not possible, due to memory limitations. Luckily, the feasibility of postconditions is rarely an issue. Hence, we safely approximate the executability of an action to its precondition only. Invariant properties I for some action a (*e.g.*, Property 15) are easily expressed as a formula of the form $\forall s, s' : S, p : T \cdot I[s] \wedge a[s, s', p] \Rightarrow I[s']$. Properties expressing the reachability of a state from a condition C (*e.g.*, Property 14) are more tricky to express. We first tried to find a trace in the transition system, but that reveals to be impossible to check due to memory limitations. We then resorted to show the existence of a trace by describing how it can be computed. For property 14, the predicate describing such a trace states that an iteration over actions `return`, `cancel` and `leave` ultimately leads to a state where the member does not

exist. If the property fails, it is either because the bound given for the length of the trace is too small (*i.e.*, the last state of the trace satisfies the precondition of a `return`, a `cancel` or a `leave`) or because the property is false in the model. By looking at the counterexample found, we can determine which case holds and increase accordingly the bounds for the trace and the number of library states. An additional difficulty is to determine the valid library states where C holds. These can be either characterized by a fact, which is error-prone to specify, but more efficient to check, or by executing entity and association producers from the initial state of the system to automatically construct valid library states satisfying C, which is simple to specify, but significantly less efficient to check.

2.5.6 ProB

Each action is specified as an operation defined as a precondition and a postcondition. Therefore, the main difficulty is to translate the ordering constraints (like, for example, "a book must be acquired in order to borrow it") in a precondition and find appropriate updates of state variables, as in SMV and Alloy.

As already mentioned, most of the requirement can be categorized in two patterns: $\Box(\texttt{can\_}A \Rightarrow P)$ and $\Box(P \Rightarrow \texttt{can\_}A)$. In general, a requirement that looks like "A can be executed only if P is true" (the first template) can be seen as an indication that the precondition of A implies P. On the other hand, "A can be executed if P is true" (the second one) means that P implies the precondition of A. In ProB, `can_`A is expressed with the executability operator $e(A)$. However, it denotes the exact condition under which A is executable; it is not an approximation as we have done in Alloy.

Specifying properties is straightforward using LTL and CTL. All properties are expressed in LTL, except 14, expressed in CTL. Property 12 is slightly more difficult to express. It denotes an ordering constraint that depends on both the current state (the book has been reserved by the member) and the previous action (once a `Take` is executed, the executability of the `Cancel` is not needed anymore). Thus, the executability operator, the *next action* operator and the LTL release operator are needed. This property does not not fit in the two described patterns.

Since ProB uses the B notation, it can be used in conjunction with Atelier B. This means that some proofs can be done prior or after using ProB. These tools can work together. For

example, Property 15 is defined as an invariant. Atelier B generates proof obligations for invariants. But when the proof fails, PROB is quite useful to find where the problem is located. On the other hand, most temporal properties are generally not provable in Atelier B because they cannot be easily expressed as an invariant.

2.6 Analysis of the Case Study

In this section, we analyse the results of our case study along several aspects of IS specications which distinguish the salient features of each model checker.

Model specification language: abstraction over entity instances. This feature enables the specifier to parameterize the number of instances for each entity and association (*e.g.*, the number of books). If it is lacking, then the size of the specification text grows exponentially. All model checkers, except NUSMV and LOTOS-NT support this feature. It is worse in NUSMV, where each transition must be hardcoded for a given member and book. In LOTOS-NT, quantification for interleave is missing.

Model specification language: representation of entity and association structures. This is reasonably well supported by all model checkers. Modeling actions that involve several associations, like `take`, is not trivial in SPIN.

Model specification language: representation of IS scenarios. This is also reasonably well supported by all model checkers. IS requirements are often described as scenarios on events, from which event ordering constraints are deduced. These ordering constraints are more explicitly represented in event-based languages like CADP and SPIN. They are encoded as preconditions in state-based languages like SPIN, NUSMV, PROB and ALLOY, which are a little bit more cryptic.

Property specification language: abstraction over entity instances. Similarly, this feature enables the specifier to abstract from entity instances by using quantification on variables. If it is lacking, either the number of properties grows exponentially with the number of instances to check, or, as we did in this case study, a property is hardcoded for a particular instance of each entity, assuming that each entity behaves in a similar fashion (which may not hold in practice). NUSMV, LOTOS-NT and SPIN lack this feature, since it is generally not supported in LTL, CTL and XTL. PROB does not suffer from this limitation in CTL and LTL, because it evaluates properties for all elements of abstract sets when

necessary. Hence, only PROB, FDR2 and ALLOY fully support this feature.

IS property specification. We have identified the following classes of properties for IS:

1. **SCE: Sufficient state condition to enable an event** (*e.g.*, case study properties 1 and 12). These are relatively easy to specify in state-based languages like NUSMV, PROB and SPIN. All of these properties must be approximated in ALLOY, otherwise they require a too large number of atoms to be completely checked. The validity of the approximation relies on the hypothesis that the postcondition of an action is satisfiable when the precondition holds. FDR2 can also handle these properties using stable failure refinement, but sometimes by approximation (property 1).
2. **NCE: Necessary state condition to enable an event** (*e.g.*, case study properties 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13). These are also relatively easy to specify in state-based languages like NUSMV, PROB and SPIN, and with some approximations for ALLOY (similar to SCE). Properties 5 and 10 have not been specified in XTL for CADP, because the states where too difficult to characterize using events only. There were no problem to specify these using trace refinement in FDR2. However, we suspect that there may be cases where characterizing states using only events may be difficult.
3. **SCEF: Sufficient state condition to enable an event on some execution path** (*e.g.*, property 14). This is easy to specify in NUSMV and PROB, thanks to CTL. It is also possible for CADP and FDR2, when the state condition is easy to characterize using events. It can be specified in ALLOY by providing the path of events leading to the desired event, when such a path does not exceed the number of atoms available. It is not possible to specify these properties in SPIN, since they are not supported by LTL.
4. **INV: Invariant state property** (*e.g.*, property 15). All model checkers can handle these without particular problems.

Property specification language: access to states and events. Since most of the properties use both states and events, model checkers that support both, like PROB and ALLOY, are simpler to use, since they can represent properties more explicitly (or directly) than the others. CADP and FDR2 being event oriented, handling states is sometimes cumbersome. SPIN offers limited support for events and we have used it extensively, similarly in NUSMV, but to a lesser extent.

Execution time and number of entity instances. Figure 2.2 shows the execution for the number of instances (number of books and number of members). The average time per property is also provided, since not every model checker can handle all properties. Overall, CADP, NUSMV, PROB and FDR2 cannot check, within reasonable bounds of time and memory, more than 3 instances for each entity for at least one property, although for some properties they can check a few more instances. SPIN can handle up to 5 entity instances. ALLOY is the most efficient model checker for IS for large number of entity instances. FDR2 is the most efficient for 3 instances; it fails due to memory limitations for more than 3 instances per entity. ALLOY is quite impressive: it can handle up to 98 instances for all properties except 14 in less than a minute. Property 14 is checked for 8 members and 8 books in a few minutes. With the library case study, 3 instances is a minimum to check reservation queues of length greater than 1. Note that the latest release of PROB fails for 3 properties, due to some defects which have been reported to authors. This is why we only include the results of 12 properties in Fig. 2.2.

Tools support. Simulators are available in each method, which is very handy to discover specification errors. The simulator in NUSMV is not straightforward to use, because it is sometimes difficult to select the transition to execute.

Step	SPIN		NUSMV	FDR2	CADP	ALLOY		PROB
Nb of Books/Members	3/3	5/5	3/3	3/3	3/3	3/3	8/8	3/3
Check Time	772.52	8645.6	3844.5	77.08	970.19	221.08	288.59	1094.4
Number of properties	14	14	15	15	13	15	15	12
Average (per property)	55.18	617.54	256.3	5.14	74.63	14.74	19.24	91.19

Figure 2.2: Model checking duration in seconds for the properties of the library specification

2.7 Conclusion

We have presented a comparison of six model checkers for the verification of IS. The comparison is based on a case study of a typical IS. The study reveals that a good IS model checker has to be very polyvalent. To conveniently specify IS models and properties, it should support both states and events. Process algebraic operators are desirable to easily expressed IS scenarios, while state variables are handy to streamline specification of

properties. CTL seems sufficient to handle most common properties. LTL is useful, but insufficient (*e.g.*, SCEF properties). A pure first-order logic like ALLOY is sufficient, but less intuitive in the case of SCEF properties. Given these characteristics, PROB seems to be the most polyvalent model checker for IS.

Since these conclusions are drawn from a single example, they must be further validated with additional examples. However, the library case study is sufficiently complex to exhibit a good number of characteristics found in most IS. It only contains two entities and two associations; large IS typically have hundred of entities and associations, but it seems quite reasonable to suppose that the verification of a property can be restricted to the entities and attributes involved. Hence, the properties checked in this case study are representative of typical IS properties.

Additional case studies would certainly find other limitations of these model checkers. For instance, our case study only addresses the sequence of actions that an IS must accept. It does not cover output delivery (*e.g.*, queries) and user interface interactions.

Chapitre 3

Vérification de modèles au service de l'informatique ubiquitaire

Résumé

Cet article illustre une mise en œuvre de la vérification de modèles dans le domaine de l'informatique ubiquitaire. Il décrit une nouvelle méthode de développement qui intègre ALLOY dans son processus pour assurer la sécurité de l'utilisateur. Il présente aussi une étude en profondeur des avantages et des limitations de la méthode en proposant des alternatives pour les contourner ces dernières.

Commentaires

Cet article a été publié à la conférence Pervasive 2011 qui a eu lieu à San Francisco en juin 2011. Les résultats obtenus sont le fruit d'un travail de deux laboratoires : le DOMUS et le GRIL. L'article a été écrit principalement par Thibault de Champs et moi-même.

Pervasive safety application with model checking in smart houses: The INOVUS intelligent oven

Thibault de Champs, Mohammed Ouenzar, Bessam Abdulrazak, Marc Frappier, Helene Pigot, Benoit Fraikin
Département d'informatique, Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1

Abstract

Safety is a major challenge in developing assistive software for people with special needs in smart houses. INOVUS is an ongoing project about safety issues surrounding cooking activities. This paper presents the INOVUS project and highlights lacks in current software development processes to meet safety requirements in pervasive computing. In the INOVUS project, we propose an approach to introduce model checking as a new layer to strengthen the design and the understanding of specifications in the development process of safety related pervasive computing applications. Finally, we illustrate the model checking process and present the initial results of the INOVUS prototype.

3.1 Introduction

Today, pervasive computing has become an active research area in which building smart houses to fulfill basic needs of *Activities of Daily Living* (ADL) is emerging. In smart environments, applications for cognitive assistance aim to provide *People with Special Needs* (PwSN [1]) with adapted help. Safety is a major challenge in developing assistive applications for PwSN in smart houses. This is an important aspect of the INOVUS project, which aims at developing a safety solution for people when performing cooking activities in a smart house. The INOVUS pervasive application ensures PwSN safety based on defined risk contexts, offers adequate actions in case of upcoming or inevitable hazard and provides information about potential current risks to the inhabitant and the caregivers if it is needed.

Human environment interaction is considered as one of the predominant part in pervasive computing. Promoting independent living for PwSN in smart houses requires insuring safe human-machine interaction and safety in general at home, and consequently reliable systems. To achieve this goal, usual software development processes [45, 46] are incomplete in term of software consistency management [44]. This paper proposes an additional layer based on formal specifications to ensure consistency of the software under development, which is a key point for its future reliability. Formal specifications are mathematical software representations, which describe system objectives and behavior early in the design. Additionally, those methods allow to verify/prove the resulting model [7, 2]. In a pervasive computing approach, model checking stands out because of their code development approach.

This paper discusses the INOVUS design, introduces our approach using model checking based on first-order logic and presents the implementation of the solution using ALLOY [32] to develop a software capable of managing the potential risks in a smart kitchen. The goal is to improve the design phase in order to facilitate development of applications, in addition to strengthen the consistency in the development process of safety application. In this paper, we also evaluate the ALLOY model checker whose modeling language is close to object-oriented programming and which has an automatic analysis tool (ALLOY-analyzer) ready to use in pervasive computing. This paper also presents the limits of our approach and alternative improvements.

1. People with Special Needs: Elderly or people with disabilities.

This paper is structured as follows: SECT. 3.2 introduces the INOVUS project and raised issues. SECT. 3.3 presents the related work on consistency checking. SECT. 3.4 discusses the proposed additional layer based on model checking and the INOVUS implementation using ALLOY model checker. SECT. 3.5 presents more details about the implemented model verification, whereas SECT. 3.6 discusses the results and, SECT. 3.7 concludes on the proposed approach.

3.2 INOVUS Intelligent Oven Project

This section presents INOVUS project: the context of the project, an illustration scenario, an overview of the specifications and the adopted approach.

3.2.1 Context

INOVUS is a specific safety related ongoing project in DOMUS laboratory. DOMUS lab aims at research in domotics, mobile computer science, and specialized in the development of assistive technologies for PwSN throughout a smart environment. These environments provide PwSN with assistance in terms of services, medical monitoring, safety, among others.

Safety at home is an emerging issue in pervasive computing. It is related to various ADLs. Studies have shown that this is a major concern for PwSN and their caregivers [25, 55]. These risks are mainly located in bathroom and kitchen [12, 36, 51]. For instance, main risks at home for PwSN are falls, fires and burns [12, 27, 29, 36, 51]. Numerous studies/solutions have addressed the fall problem [8, 34, 42]. However, rare are those addressing fires and burns. The existing solutions address risks after manifestation or propose prevention guides (e.g. [3]). Thus, there is no integrated solution taking in account the major contexts of risk for cooking activities. This justifies the motivation behind the INOVUS project.

3.2.2 Scenario

In order to clarify how to address the aforementioned issues (SECT. 3.2.1), we examine the following scenario. Mr. Smith is a 74 year old man who lives alone. One year ago, he

was diagnosed with probable Alzheimer disease in the early stage. To facilitate his daily living, his daughter (a caregiver) visits him everyday at noon to prepare lunch and dinner meals for him. Two month ago, as his daughter was late, Mr. Smith started preparing food using the oven. Few minutes later, he moved to the living room to watch his favorite TV show which just began, forgetting the meal on the stove. His daughter arrived forty minutes later. As she entered the house, she noticed a dense smoke because of the food burning and her father hardly breathing.

To help Mr. Smith and his daughter, we propose an integrated technological support to ensure safety in cooking activities. In the above case, the INOVUS intelligent oven would have pro-actively replied to the situation. (1) The too long time away from the oven would have been detected. A warning would have been sent to Mr. Smith via TV, screens, and/or speakers of the smart environment. If Mr. Smith would have not reacted in time to this message, the system would have automatically shut down the oven. (2) In the same time, the smoke appearance would have been detected. This critical risk which threatened Mr. Smith would have resulted in an emergency stop of the oven. In addition, his daughter or/and relevant emergency services (e.g. 911) would have been informed about the situation. Such a solution would ensure safety of Mr. Smith and serenity for his daughter.

3.2.3 Specifications

The goal of INOVUS project is to provide an autonomous solution that ensures user safety and takes in consideration major safety issues in cooking activities. INOVUS (architecture in FIG. 3.1) is based on smart environment infrastructure, especially sensors and actuators networks distributed in the kitchen area. Sensors allow the system to infer the activities of the house (for instance movements near the oven or actions on it) or detect changes in the surrounding environment (e.g. smoke, temperature of surfaces, people presence). Actuators allow the system to feedback user through screens, speakers, or flashing lights, or control appliances (shutdown the oven).

The INOVUS *decision engine* observes sensors in order to infer risk contexts, informs the user about existing hazards through actuators in the house (warnings) and alerts caregivers if it is necessary (emergency call). The user has a pre-configured time to react to the danger or inform the *decision engine* that he/she is conscious of the situation. If the user does or

can not react to the warning, the *decision engine* automatically takes control of the oven and plans actions in order to avoid dangerous situations. In case of an unavoidable accident, emergency procedures are initiated.

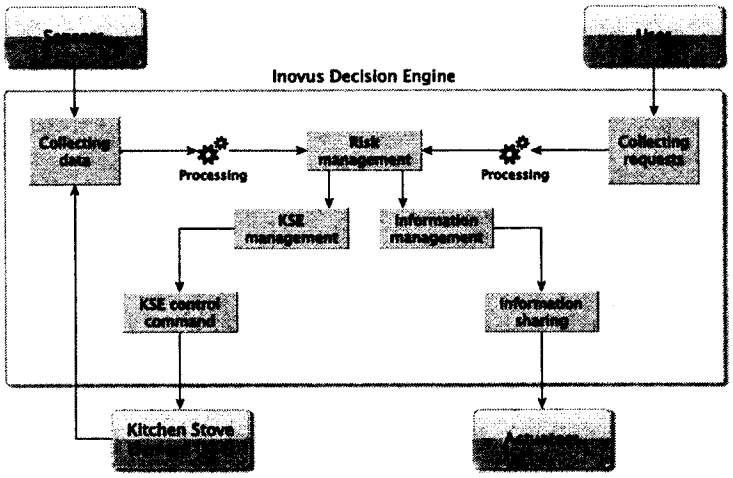


Figure 3.1: INOVUS architecture in the home environment

3.2.4 Approach

Software development process includes several steps with specific objectives for the development of an application. Main steps are requirements collecting, specifications writing, design, implementation and tests. Reliability of the resulting software is evaluated according to verification and validation steps. Despite the relatively complete process, errors or system failures often remain and occur throughout software's life cycle.

Software consistency and reliability have to be ensured when addressing safety of people in ADLs at home. Evolving directly in the occupant's environment and assisting him/her in performing ADLs, the application should not provide erroneous information or generate incorrect reaction that can lead the user to dangerous situations. In order to reduce the risk of introducing faults in software, the INOVUS software development process has been strengthened with formal specifications. To do so, we added in our approach a model checking stage into the development process. The additional stage provides a formal modeling of the specifications to validate the consistency and behavior of the resulting solution. The formal model is jointly developed with the classical software design, creating

a loop between these two steps (FIG. 3.2). This cycle is located after the writing of specification and before the code implementation, which allows better validation of the design. This combination of standard software design and model checking adds a consistency layer to the resulting model and then provides a complete formal model of the application. The model checking approach provides a mathematical verification of the model. Efficiency of this approach has been proven [7], and coupling it with one of the usual tools for software design ensures that the requirements of the project are satisfied in terms of software behavior. This appears to be a key point while designing applications for smart houses.

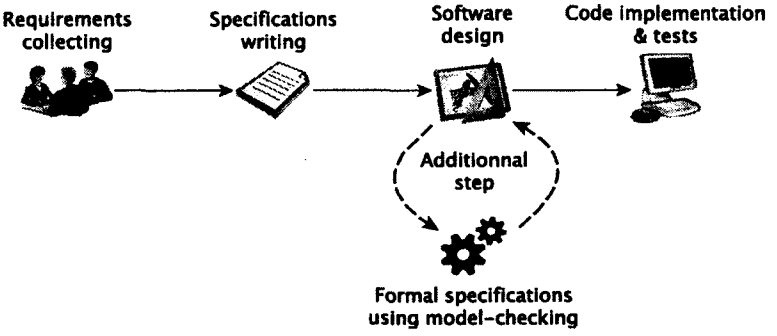


Figure 3.2: Classical process with the proposed additional step

3.3 Related Work

This section highlights more details about formal specifications and consistency checking in software development process.

3.3.1 Consistency Issue in Software Development Process

Several specific approaches can be used in software development process (e.g. waterfall, iterative, spiral among others) [26, 45, 46]. Stages of these processes are quite similar: design, testing, debugging and verification. Software design appears to be an important stage of the process. Experts view design consistency as a lack in the management of software development process [20, 26, 28, 31, 44]. Huzar et al. in [31] define consistency as incorrect relations between the artifacts that represent the aspects of a system. Consistency

can be split in two parts: (1) Intra-model consistency discusses the multiple views of the software architecture and consequently the different abstraction levels, which are only multiple interpretations and concepts representations; (2) Inter-model consistency discusses the connections between model refinements. On the other hand, Nuseibeh et al. in [44] define inconsistency as "*any situation in which a set of descriptions does not obey some relationship that should hold between them.*" Most description tools, especially *Unified Modeling Language* (UML) [11], use multiple views to represent the software architecture according to the specifications with imprecise semantics, and thus suffer inconsistency. Nuseibeh et al. in [44] defend that maintaining consistency all the time is counterproductive, and time and resource-consuming. They proposed a framework to monitor, diagnose, handle, measure and analyze inconsistent elements in order to obtain respectable and manageable inconsistency.

Moreover, several description techniques were proposed [18, 20, 28, 31] where formal approaches remain a recurrent suggestion because of its mathematical basis. A formal specification provides precise models, specifies constraints and checks them. The model checking is the formal approach adopted in the INOVUS project. As explained below, the golden mean managing inconsistencies with formal specifications lies in abstraction.

3.3.2 Formal Specifications

Formal specifications suggest abstraction as the core of software development. The abstraction level is an essential point in having an efficient programming. According to D. Jackson: *"An Abstraction is not a module, or an interface, class, or method; it is a structure, pure and simple—an idea reduced to its essential form. [...] The best abstractions, however, capture their underlying ideas so naturally and convincingly that they seem more like discoveries"* [32].

Several developers come up with improper abstractions that result in inconsistent design and laborious implementation. It is the result of a lack of abstraction and modeling in common software development process. Modeling is often based on *wishful thinking*, a concept taken up by B. Meyer [40] who describes the formation of beliefs and decisions made according to what might be pleasing to imagine instead of by appealing to evidence, rationality or reality. Formal specification has been proposed as a solution to tackle the

lack of abstraction by attacking its design head-on. Some examples [7] show that the use of formal specifications often comes up in very high critical situations and later in the software life cycle, and corroborate its efficiency. The under-usage of formal specifications until now is explained by two facts: complex mathematical logic notation and slow theorem provers development. Model checking [7] is one of the main dimensions in formal specification. Model checkers explore the transition system graph to check the satisfaction of predefined properties.

Numerous model checkers have been developed by organizations for specific purposes, and the difficulty is to find the most appropriate one depending on the needs. Frappier et al. in [21] present a comparison of common model checkers to address this difficulty and find out about the benefits and disadvantages of each model checker.

3.3.3 Introducing ALLOY

ALLOY is a model checker based on first-order logic which includes only types of relations in the language. ALLOY modeling language is close to the one of *Object-Oriented Languages* (OOL) with new features like signatures and the notion of scope. A scope defines the number of each signature in the model. Unlike its predecessor Z [53], which is appropriate for describing structural properties of systems, ALLOY models are automatically analyzable. This analysis is performed by SAT-solvers [37], which are tools used to check model consistency and find counterexamples of assertions. ALLOY representation of systems is based on abstract models. An ALLOY specification consists in signatures that basically define sets and relations. Constraints (denoted `fact`) are formulas conditioning the values of sets and relations. The signature `sig X {r : X -> Y}` declares a set `X` and a ternary relation `r` which is a subset of the Cartesian product $X \times X \times Y$. ALLOY supports usual operations on relations, like union, intersection, difference, join, transitive closure, domain and range restriction. Cardinality constraints can be defined on relations (e.g. injections and bijections). More details about INOVUS implementation with ALLOY are given in SECT. 3.4 and SECT. 3.5.

3.3.4 Tests and ALLOY Combination

In software testing, the cardinality of possible cases set could be great, thus unmanageable. Consequently, tests are conducted only on the cases considered by the software tester as the limit points, which means that there are cases that are not tested, which can be a source of defects. In ALLOY, the space of checked cases is wide and leads to an unattainable coverage degree in testing because of the results of research in constraint-solving technology [37]. ALLOY analysis is not complete since it only examines a subset of the case space according to the scope selected by the user. FIG. 3.3 illustrates the coverage difference between tests and ALLOY, and brings out the relevancy of combining them.

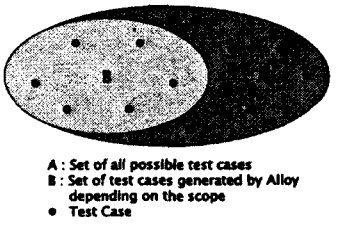


Figure 3.3: Coverage difference between tests and model checking

3.4 INOVUS Development Process

The SECT. 3.2 presents the INOVUS specifications. According to the related work given in previous section, here we describe the INOVUS first model and the corresponding implementation using ALLOY model checker to ensure consistency.

A regular approach in formal specifications is the use of increments. This allows to begin the software design with a high abstraction level in order to validate the main concepts of the desired model. Next increments refine the software design, gradually reducing abstraction levels. Such a technique lets designers to validate step by step the software construction. INOVUS design is based on this method.

This section describes below the INOVUS first increment in terms of specifications, design, and implementation with ALLOY.

3.4.1 First Prototype Specifications

INOVUS first prototype specifications are based on the requirements described in SECT. 3.2. This first version is restricted according to the complete specification of the INOVUS project and will be extended in further increments with a larger cooking hazards collection, according to the model checker validation.

This preliminary prototype follows the architecture described in FIG. 3.1. Stove elements (i.e. hotplates and oven) are distinguished, and commonly denoted as *Kitchen Stove Element* (KSE). Burns, fires and intoxication are the risks integrated in this increment. These are general hazards, but contexts in which they appear may differ. TABLE 3.1 describes the included contexts.

Table 3.1: Risks considered in this increment

Risks	Contexts
Burns	Hot surfaces (oven front or hotplate)
	Opened oven door
Fires	Food forgotten while baking
	No surveillance in home
	People leaving home while food baking
Intoxication	Smoke detection in the kitchen

These elements lead to the design of this first prototype.

3.4.2 First Increment Design

Software design is the basis of software development, thus the choice of the design tool is crucial. For INOVUS project, UML has been chosen because of its object-oriented approach and its multiple software views. Static diagrams concur with the structure of the ALLOY model and dynamic diagrams concur with the dynamic ALLOY feature. Consequently, we have to highlight the entities composing the system and the events that may occur. To do so, relevant diagrams for this step are selected and classes (FIG. 3.4), use cases, and state diagrams were created.

It should be noted that methods and some attributes could be omitted in the class diagram, since they do not contribute to the formal model. Nevertheless, these elements

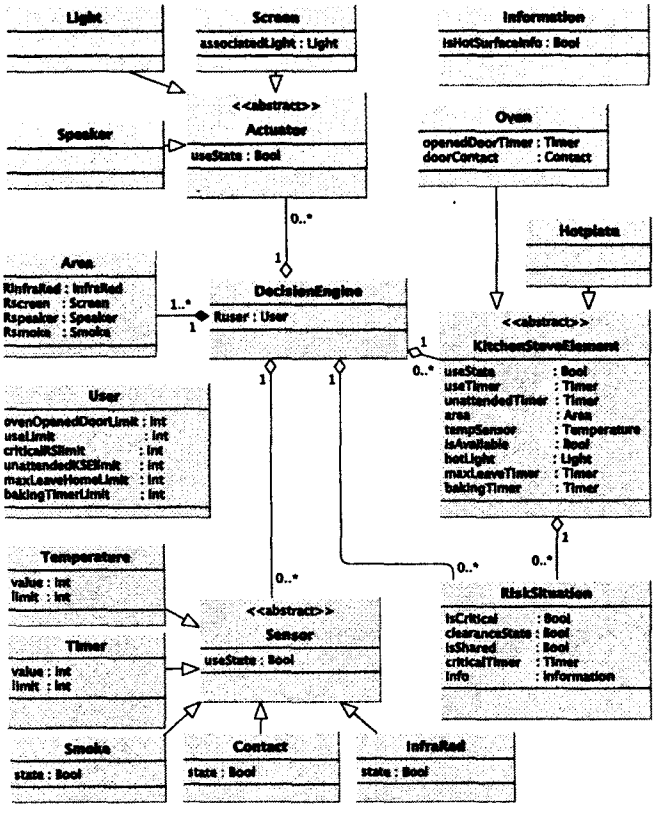


Figure 3.4: Simplified class diagram

are necessary for a code implementation. The main goal is to identify the possible events composing the transition system, their preconditions, and their post-conditions.

Let's have a brief overview of the model representation given by FIG. 3.4. A unique entity (called `DecisionEngine`) is the core of the system. This entity listens to a set of sensors distributed in the home and in the close environment of the kitchen stove. Based on sensors' values, it infers user's activities; sensors returning pertinent messages allow the `DecisionEngine` to perform a planned sequence of actions, whether a risk situation exists or a command is asked by user. If a context of risk appears, a `RiskSituation` is added to the risk situations set of the concerned `KitchenStoveElement` and the user is informed throughout actuators (screens, lights, and speakers). The `criticalTimer` timer specifies the reaction time the user has to notice information about the detected risk. Smoke detection or reaching the `criticalTimer` limit of a `riskSituation` leads to the emergency stop of the associated KSE. To be started again, the user has to acknowledge the risk situation.

Temperature sensors detect hot surfaces, and the associated information is shared only with the `hotLight` light of the associated `KitchenStoveElement`.

User can set the `useTimer` (KSE max use time), the `unattendedTimer` (used to detect activities in the kitchen coupled with infra-red sensors), the `maxLeaveTimer` (used to detect user presence in home coupled with infra-red sensors), and the `bakingTimer` (used to program the baking time). He/She can also notify the system that he/she is aware of the risk and does not want to be informed about it. If a context of risk disappears due to a user reaction, the diffusion of the associated information is stopped and the `riskSituation` is deleted.

These static and dynamic representations of the model are the preliminary steps to the code implementation in ALLOY syntax.

3.4.3 First Increment Implementation Using ALLOY

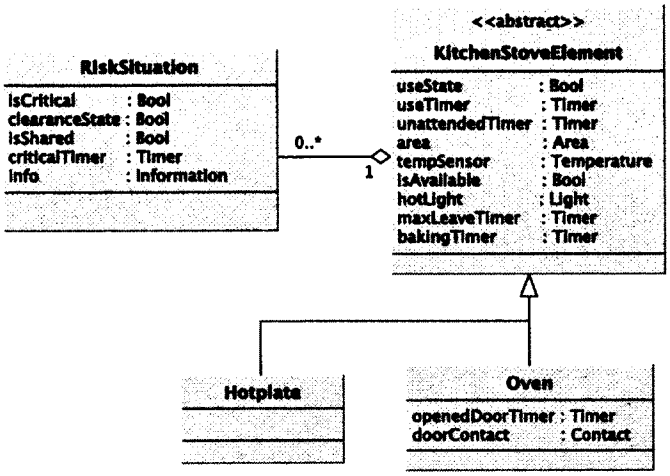
This subsection presents the UML diagrams implementation in ALLOY language.

Diagrams Implementation

Based on the INOVUS design, the transition system can be defined according to the specifications and UML diagrams. The design brings out dynamic parts of the software. Combining these dynamic elements with the desired behavior leads to the definition of transitions between them. In the INOVUS project, these transitions accord with a risk situation appearance, an user action, or to environmental changes (i.e. sensors state evolution: sensors state changes are caused by the environment modification, namely user's actions and their eventual consequences).

Diagrams resulting of the design stage need to be transposed in ALLOY code to generate the corresponding INOVUS transition system. ALLOY seems quite adapted for this task since ALLOY syntax is close to OOL and UML notations. A translation example from a class represented with UML to the corresponding ALLOY code is given in FIG. 3.5. More details about the translation process are given in the following section.

Furthermore, some abstractions have been made. In particular, the way to interact with the user of the kitchen stove has been generalized and the message that the system is sharing with the user is not specified (i.e. only actuators states are checked). The two other needed




```
abstract sig KitchenStoveElement
{
  useState : one Bool,
  useTimer : one Timer,
  unattendedTimer : one Timer,
  area : one Area,
  tempSensor : one Temperature,
  isAvailable : one Bool,
  hotLight : one Light,
  maxLeaveTimer : one Timer,
  bakingTimer : lone Timer,
  rs : set RiskSituation
}
sig Hotplate extends KitchenStoveElement
{
}
sig Oven extends KitchenStoveElement
{
  openedDoorTimer : one Timer,
  doorContact : one Contact
}
```

Figure 3.5: Translation example from a UML class into its corresponding ALLOY signature

abstractions concern time and temperature evolution. These simplify the formal design and do not affect the consistency of the resulting model instance.

On the other hand, UML dynamic diagrams are represented by transition graphs called *traces*. A *trace* is a sequence of states generated by ALLOY according to a defined scope. Those states result from the transition system predicates evaluation (the actions of the transition system). This takes place in ALLOY analysis and is presented in SECT. 3.5

This constitutes the first part of the INOVUS formal model, which ensures the model consistency. In a second time, it can now be completed with the properties to check.

Properties Checking

The model checking approach separates properties checking into three tasks. Thereby, properties to check have been selected and written in natural language. We then transposed them into ALLOY code according to the INOVUS transition system and started the checking process. Properties definition in natural language is crucial, since the translation in corresponding ALLOY code is a strict and logical formulation of these assertions.

Concerning relevant properties selection, they are classified according to the severity/degree of the hazard they represent. INOVUS project focuses on safety for people while cooking, therefore we want to validate the safety properties that the model has to satisfy. For this reason, we need to choose the ones which appear the most relevant to safety issues. Concretely, we want to especially validate detection of risk situations, and corresponding reaction. These assertions represent the system's behavior when a risk situation occurs.

LISTING 3.1 is an example of an included safety property, followed by its ALLOY expression. This example corresponds to the hot surface detection.

To explain this example, the threshold temperature exceeding is represented by the `Env_eThresholdTemperatureExceededHot[d,d',k]` event of the transition system. An occurrence of this event implies a non-critical risk situation for the specified kitchen stove element, which must have been added in the set of risk situations for the second decision engine. The `check` command initiates the checking step. About quantifiers, the assertion must be verified for all reachable decision engine and for the entire set of KSE included in the initial decision engine (denoted `d`).

```
assert PropertyExample
{
  all d,d':DecisionEngine,k:d.RkitchenStoveElement |
    d'=SequenceDE/next[d] and
    Env_eThresholdTemperatureExceededHot[d,d',k]
  =>
  some rs:RiskSituation | rs !in d.RriskSituation and
    rs in d'.RriskSituation and isFalse[rs.isCritical]
}
```

Listing 3.1: ALLOY translation of the property *Exceeding the hot threshold temperature for a temperature sensor must be detected as a risk situation*

3.5 INOVUS Model Verification with ALLOY

This section gives more details about UML translation in ALLOY[2], the analysis driven on the formal model, and some tools brought by ALLOY analyzer.

3.5.1 More About ALLOY Syntax

As mentioned in SECT. 3.4.3, FIG. 3.5 illustrates a translation example from a UML class into corresponding ALLOY code. Translation is almost automatic due to similar syntax as in the object-oriented programming. In ALLOY, each class is represented by a signature (denoted by `sig`) and attributes are signature fields which are written in the same way as in *Object-Oriented Language* (OOL). More exactly, each field represents a Cartesian product between the signature in which it was declared and the signature which defines its type. Keywords one, `lone` and `set` define cardinalities for classes and fields. Aggregation and composition relations in the class diagram are represented by *sets* union in the formal model. Axioms are expressed by the `fact` keyword and represents automatically assumed constraints. Abstract classes are represented by the `abstract` keyword. Inheritance is represented by the `extends` keyword. Actions are specified by predicates (denoted by `pred`) written in the form *preconditions/actions/postconditions* which is the most used pattern in case of dynamic systems. Functions are presented by the `fun` keyword.

2. http://alloy.mit.edu/alloy4/

3.5.2 Model Analysis

ALLOY analysis is focused on two commands:

- run allows to have one model instance, meaning one model satisfying all constraints. It can also be considered as a command used to check the model consistency which means that assertions in the model do not contradict each other.
- check checks a property for all model instances in the scope defined by the user, which can be million of usage scenarios called traces. More exactly, a trace is a totally ordered set: it represents an ordered state sequence of the modeled system. It is defined by a fact and contains all predicates that defines the transition system.

In the INOVUS implementation, every trace begins with an initial state defined by the `init` predicate. Every element of a trace is then a validated predicate (meaning checked as *true*) in a transition system from a state to another. LISTING 3.2 shows a trace sample of the INOVUS implementation which returns a possible predicates sequence after the initialization state. Every `DecisionEngine` entity on a trace represents a KSE state. The first step is to assume that for every trace, the predicate `init` has to be satisfied for the first state on the trace (line 2); lines 3-5 indicate that every `d'` state is the d-subsequent state; third step is to implement predicates that allow to describe the systems' transition from initialization phase (lines 6-7).

```
1 fact {
2   init[SequenceDE/first[]]
3     all d:DecisionEngine
4         - SequenceDE/last[] |
5     let d'=SequenceDE/next[d] |
6       some k:d.RkitchenStoveElement |
7         User_eAskStartKSE[d,d',k]
8         or User_eAskStopKSE[d,d',k]
9 }
```

Listing 3.2: Trace example in ALLOY

ALLOY consistency checking is decidable but very time-consuming if there are some inconsistencies in the model. It guarantees an instance exists for all predicates called in traces. Thereby, the necessity of the consistency checking step appears to be quite obvious and complements properties checking.

3.5.3 Additional Validation Tools in ALLOY

In addition to SAT-solvers, ALLOY provides some tools to evaluate expressions and formulas on models. Unsat core [16] is a tool used to identify the key portions of a model that causes unsatisfiability. However Unsat core is not yet a mature software. *dot*[3] is a tool used to provide graphical representation of a model. *Evaluator* is a tool allowing to evaluate first-order formulas.

3.6 Results and Discussion

The development of the INOVUS first prototype allowed to evaluate the use of ALLOY, and as a consequence model checking, for the development of a safety application in a smart environment. This led us to preliminary results and a discussion concerning the relevancy of the proposed development process and the choice of ALLOY for a use in pervasive computing.

3.6.1 Relevancy of the Proposed Process

Formal model helps understanding the specifications for a better code implementation. The proposed process is iterative. Each cycle of the formal model design highlights mistakes or improper elements of the UML design according to the specifications. This enables adjustments in the initial UML representation and/or the formal model. Nine iterations were necessary to produce a complete and stable instance of the model first increment in INOVUS project. Each iteration results from poorly implemented specifications in the structural model, or from new abstractions which appears as necessary in the formal model.

Choices of elements that can be omitted are fundamentals for a good abstraction. There is no need to represent the system in details but only the elements which need to be checked by ALLOY-analyzer, which has two main advantages. First, the ALLOY model checker workload is substantially reduced whereas a too large model would be time consuming and resource intensive. The model checker may even be an inefficient tool. Second, by reducing the constraints that have to be respected by the desired model, the corresponding code

3. http://www.graphviz.org/Documentation.php

implementation will be facilitated. In fact, a complete representation of the specifications in the formal model is too restrictive for the coding stage. With this point in mind, designers will be able to validate a formal model according to the critical elements of specifications, while enabling flexible handling of the non critical elements in the development stage.

3.6.2 Evaluation of ALLOY in Pervasive Computing

Using the ALLOY analyzer has been an efficient way to improve the INOVUS model understanding and consistency. It also enables to obtain a final functional version of the UML diagrams and to assure better system reliability. It has also been an approach to guarantee that critical requirements are satisfied.

Object-oriented and easily controlled ALLOY syntax are two major arguments to support use of ALLOY in pervasive computing. Nevertheless, some limitations exist with ALLOY. First, the more the number of entities generated by the model is large, the more the complexity is high. This can slow down the project progress. For this reason, developers need to gradually construct the ALLOY model to prevent complex and time-consuming actions. This progressive design also allows to detect inconsistency earlier in the development process. Second, as a first-order language, ALLOY has the *frame problem*: the developer has to indicate what has to be changed (or not) on modified signatures' fields for every action that allows a transition system. This can be costly in terms of development time, especially in the case of dynamic system such as the INOVUS project. DYNALLOY [23] is an extension of ALLOY that describes properties regarding execution traces, in the style of dynamic logic specifications. DYNALLOY simplifies the way to represent dynamic systems and to model in the same way as programming (conditions, loops, ...). It is the future candidate to study in the INOVUS project.

Finally, ALLOY encompass a limitation of reachable atoms. In other words, the more there are relations between signatures, the more the number of reachable atoms is reduced, thus it is better to avoid large relations. Distributed environments implies large entities generation, and consequently large atoms use. INOVUS implementation avoid this restrictive factor by defining relations which are at most ternary ones.

3.7 Conclusion

This paper proposes the introduction of model checking as a new layer to ensure safety for people while developing applications in smart houses. Using this tool, the INOVUS project aims to be an alternative and complete technology to the existing solutions in terms of safety in the kitchen for PwSN.

The model checking approach enhances the design of a software under development and ensures precise respect of requirements. As a consequence, this paper shows how formal specifications can be an asset for pervasive applications development. The INOVUS design reveals critical steps in which formal specifications improved software design for pervasive applications.

Further work on this project is to extend the range of the INOVUS safety application in terms of risks contexts and ability of the decision engine to infer user's activities and danger level. This should be done in next increments. To do so, the choice of the model checker has to be optimized because ALLOY does not seem to be the best one for pervasive applications in smart houses. An evaluation of DYNALLOY should be done.
This will then bring an implementation in our smart environment, in order to validate the proposed safety solution.

Chapitre 4

Idiomes trouvés à partir des problématiques traitées

Ce chapitre présente une synthèse des travaux entrepris dans l'optique de parfaire la compréhension du fonctionnement complexe de ALLOY. Au-delà de son aspect purement descriptif, il vise également à proposer, sous la forme d'idiomes, de nouvelles pistes de solutions à des problématiques usuelles auxquelles les utilisateurs peuvent être confrontés.

4.1 Revue de littérature

La spécification des modèles dynamiques ainsi que les traces en ALLOY ont fait l'objet de nombreuses études scientifiques au cours des dernières années [32, 23, 33, 54]. Ces sujets ont notamment été abordés par Marcelo Frias et son équipe dans une étude ayant abouti à la conception d'une extension dynamique de ALLOY : DYNALLOY. Ce faisant, à l'instar de l'étude de Frias et al., la plupart des recherches encore menées aujourd'hui sur le sujet souffrent d'un caractère trop généraliste et d'une portée limitée qui se circonscrit au seul volet « intégration du dynamisme » à ALLOY. Notre étude se démarque pour sa part de toutes ces publications par son intérêt porté à la comparaison entre deux manières différentes de spécifier des traces et son effort concomitant d'identification des avantages et inconvénients de telles approches. Plus spécifiquement, le présent rapport se concentre sur l'étude de deux types de traces : d'un côté, les traces structurées par des invariants, et de

l'autre, celles qui n'imposent aucune contrainte ; sujet quelque peu éludé dans la littérature scientifique. ALLOY est également appréhendé dans ce rapport sous l'angle des propriétés de type CTL. Pour ce faire, nous nous inspirons des travaux de Williams et al. [57] qui suggèrent une procédure permettant de vérifier les propriétés CTL avec des solutionneurs de satisfiabilité via le développement d'un idiome de vérification approprié.

4.2 Les traces en ALLOY

Une trace est une séquence ordonnée d'états construite à partir des prédicats qui définissent les transitions du système. Cette section se focalise justement sur deux idiomes différents pour spécifier une trace.

Le premier idiome oblige la trace à commencer avec un état initial représentant l'état du système avant le démarrage. À partir de cet état, l'évaluation des prédicats/actions définissant le système de transition est faite d'une façon non déterministe. Cependant, commencer avec l'état initial implique que les premiers états des traces soient toujours les mêmes. Par exemple, en considérant l'exemple de la bibliothèque, la précondition de l'action réservation ne peut être évaluée à vrai que s'il y a au moins deux membres inscrits et que le membre à qui le livre est emprunté est différent de celui qui fait la réservation. C'est pourquoi, pour vérifier des propriétés portant sur la réservation, la trace doit avant tout transiter par cinq états : dans deux états représentant le moment où deux membres sont inscrits, un autre état où un livre est acquis, un autre où le livre est emprunté et enfin un état où ce livre est réservé. Ceci étant, puisque la taille de la séquence est limitée, les états atteints une fois que le livre est réservé sont eux aussi limités. Par exemple, si la taille de la séquence est de sept et que la propriété à vérifier porte sur les réservations, les traces vérifiées différeront juste au niveau des deux états en fin de séquence alors que les cinq premiers états seront toujours les mêmes. La spécification 4.1 expose les grandes lignes de cet idiome en commençant par un état où tous les ensembles sont vides.

À l'inverse du premier idiome, le deuxième idiome offre pour sa part à la trace la possibilité de partir de n'importe quel état pourvu qu'il soit valide, c'est-à-dire accepté par la spécification du modèle. Cet état valide doit être défini par des invariants, ce qui est une tâche complexe et longue à faire. À partir de là, on peut considérer que le premier idiome est facile et plus rapide à écrire que le second mais le nombre d'états vérifiés dans le se-

```
pred Init [L:Lib]
{
-- all sets are empty
no L.books
no L.members
no L.loan
no L.membersReservingOneBook
no L.Renew
}

fact
{
-- initial state
Init[LibSeq1/first[Seq]]
all idx : LibSeq1/inds[Seq] - LibSeq1/lastIdx[Seq]|
some m:Member,b:Book|
Join[m,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
or Acquire[b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
or Lend[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
or Reserve[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
or Take[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
or Renew[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
}
```

prog 4.1 – Exemple d'une trace initiée par un prédicat

cond est largement supérieur à celui du premier idiome. Les spécifications 4.2, 4.3 et 4.4 illustrent le principe de cet idiome.

```
-- trace definition
fact
{
all L:Lib|L in LibSeq/elems[Seq]
-- define valid states
ValideSpec[Seq]
-- define the system transitions.
DynLibSeq[Seq]
```

prog 4.2 – Exemple d'une trace sans état initial prédéfini (partie 1)

```
pred ValideSpec[P:Seq]
{
-- A book can not be reserved in the case
-- that it is neither lent or reserved.
all b : Book |
some (LibSeq/first[P].membersReservingOneBook.b)
=>
((b in LibSeq/first[P].loan.Member) or ((\#LibSeq/first[P].
   membersReservingOneBook.b) > 1))
-- A book can not be reserved and lent to "m".
all m : Member,b : Book|
(m in b.(LibSeq/first[P].loan) => not (m in Int.(LibSeq/first[P].
   membersReservingOneBook.b)))
-- A member can not renew a loan if the
-- book concerned is not already lent to this member.
all b : Book,m : Member|
(b in LibSeq/first[P].Renew.m) =>b in (LibSeq/first[P].loan.m)
}
}
pred DynamicLib[L,L':Lib]
{
  some b:Book,m:Member|
                        Acquire[b,L,L']
                        or Join [m,L,L']
                        or Lend[m,b,L,L']
                        or Take[m,b,L,L']
                        or Leave[m,L,L']
                        or Cancel[m,b,L,L']
                        or Return[m,b,L,L']
                        or Reserve[m,b,L,L']
}
```

prog 4.3 – Exemple d'une trace sans état initial prédéfini (partie 2)

```
pred DynLibSeq[P:Seq]
{
all idx : LibSeq/inds[P]-LibSeq/lastIdx[P] |
DynamicLib[LibSeq/at[P,idx],LibSeq/at[P,idx.LibSeq/ord/next]]
}
```

prog 4.4 – Exemple d'une trace sans état initial prédéfini (partie 3)

4.3 Les propriétés de type arborescente

Parmi les propriétés vérifiées dans le cas de la bibliothèque, il y a une propriété de type arborescente (*reset*) qui correspond de fait à la propriété numéro 14. Pour pouvoir vérifier ce type de propriétés, il faut normalement générer le graphe de transition au complet. Cependant, cette génération n'est pas possible dans ALLOY du fait de la contrainte de limitation absolue imposée par le vérificateur. Aussi, d'autres solutions alternatives doivent être explorées afin de mieux contourner ce problème. Cette section propose justement deux alternatives pour pouvoir exprimer des propriétés de ce type. Qualifiées d'idiomes, celles-ci ont l'avantage de pouvoir être généralisées à toute propriété CTL du type $AG\ \psi EF\phi$.

Basée sur les séquences, la première alternative, connue sous le nom de *stuttering*, consiste à minimiser le traitement en générant des sous-graphes au lieu du graphe au complet. Autrement dit, et pour reprendre l'exemple de la bibliothèque, la propriété à vérifier dans ce premier cas de figure est de la forme suivante : « Pour toute bibliothèque possédant un membre *m*, il existe toujours une séquence définie par les actions *Leave*, *Cancel*, *Return* (LCR) qui permet à ce membre de se désinscrire s'il le souhaite ». Cette séquence peut être complète ou incomplète, c'est-à-dire qu'il est possible que sa taille ne soit pas suffisante pour considérer tous les états permettant à ce membre de se désinscrire de la bibliothèque. Par conséquent, si la séquence est incomplète, il est impératif que son dernier état soit celui où l'une des préconditions des trois actions LCR (*Leave, Cancel, Return)* est évaluée à vrai. En somme, l'intérêt est porté sur le sous-graphe concerné par la propriété en question, ce qui implique subséquemment une minimisation de son traitement par le vérificateur. La spécification 4.5 illustre le principe de cette méthode.

```
assert Prop14
{
all P : SeqLCR|
all m: LibSeq1/last[Seq].members|
TransLCR[m,P]
=>
-- m unsubscribed from the library
(
(not m in LibSeq2/last[P].members)
or
-- another transition can be done (LCR),
-- however the scope defined is too small
(
(
CanLCR[LibSeq2/last[P],m] and
(not LibSeq2/lastIdx[P] = LibSeq2/finalIdx))
)
)
}
check Prop14 for 3 Member,3 Book,7 Lib, 7 SeqIdx,7 SeqIdx1,6 SeqLCR
pred CanLCR[L:Lib,m:Member]
{
CanLeave[m,L] or
some b:Book|CanCancel[m,b,L] or
some b:Book|CanReturn[m,b,L]
}
-- transactions generated by the leave,
-- cancel and return predicats.
pred TransLCR[m:Member,P:SeqLCR]
{
-- all Sequence's index expect the last
all idx : LibSeq2/inds[P]-LibSeq2/lastIdx[P] |
LCR[m,LibSeq2/at[P,idx],LibSeq2/at[P,idx.LibSeq2/ord/next]]
}
pred LCR[m:Member,L,L' : Lib]
{
some b:Book |Cancel[m,b,L,L'] or Return[m,b,L,L'] or Leave[m,L,L']
}
```

prog 4.5 – Exemple de la technique du *stuttering*

Quant à la seconde alternative, celle-ci consiste pour sa part à trouver un programme qui permet d'arriver aux résultats attendus. Une fois ce programme défini, la vérification porte alors sur les actions qu'il exécute en procédant de manière incrémentale. Pour reprendre l'exemple de la bibliothèque, la vérification des actions LCR (c'est-à-dire, celles qui permettent à un membre de se désinscrire de la bibliothèque) consiste donc à s'assurer que celles-ci font bien ce qu'elles étaient supposées faire : pour l'action *Cancel* - qui est une transition d'un état *L* à un autre *L'* - notamment, le but est de vérifier que si la précondition est évaluée à vrai alors le membre avait au moins une réservation à l'état *L* et que cette action transite bien vers un état *L'* lorsque le membre annule sa réservation. L'étape subséquente consiste ensuite à vérifier que si les préconditions des actions *Cancel* et *Return* ont été évaluées à vrai une fois dans la trace et que le membre est toujours inscrit à la bibliothèque, alors la précondition du *Leave* est elle aussi toujours évaluée à vrai. Les programmes 4.6, 4.7, 4.8 illustrent les différentes étapes citées auparavant.

```
-- if a transition from a state L to a state L' satisfies the return
-- predicate, then there is a book which was lent by a member
-- m in the state L and it is no longer in the state L'.
pred H1[m:Member]
{
all b:Book,L,L':Lib
{
Return[m,b,L,L']
=>
(m in b.(L.loan)) and (m !in b.(L'.loan))
}
}
-- if a transition from a state L to a state L'
-- satisfies the cancel predicate, then there is a book which was reserved
-- by a member m in the state L and it is no longer in the state L'.
pred H2[m:Member]
{
```

prog 4.6 – L'ensemble des prédicats assurant le bon fonctionnement des actions *Return*, *Leave* et *Cancel* (partie 1)

```
all b:Book,L,L':Lib
{
Cancel[m,b,L,L']
=>
some i : Int |
m in i.(L.membersReservingOneBook.b) and
m !in i.(L'.membersReservingOneBook.b)
}
}
-- if a registered member can not cancel a reservation
-- and can not return a book,
-- because there is neither reservation nor loan, then he can leave.
pred H3[m:Member]
{
all L : Lib
{
not (some b : Book | CanCancel[m,b,L])
and
not (some b : Book | CanReturn[m,b,L])
and
(m in L.members)
=>
CanLeave[m,L]
}
}
-- if a transition from a state L to a state L' satisfies the
-- leave predicate, then there is a member
-- which was registered and he is no longer.
pred H4[m:Member]
{
all L,L':Lib
```

prog 4.7 – L'ensemble des prédicats assurant le bon fonctionnement des actions *Return*, *Leave* et *Cancel* (partie 2)

```
{
Leave[m,L,L'] => (m !in L'.members)
}
}
```

prog 4.8 – L'ensemble des prédicats assurant le bon fonctionnement des actions *Return*, *Leave* et *Cancel* (partie 3)

Le tableau 4.1 résume le temps d'exécution de chacun des idiomes présentés ci-dessus.

tableau 4.1 – Tableau récapitulatif du temps d'exécution par rapport aux bornes choisies

Idiome	Propriété	Nombre de livres et de membres	Temps de vérification en (s)
Traces sans définition d'état initial	Toutes	5	16,13
Traces sans définition d'état initial	Toutes	10	796
Traces avec un état initial défini	Toutes	5	15,7
Traces avec un état initial défini	Toutes	10	482,12
Propriété de bégaiement (*stuttering*)	Prop 14	5	1,8
Propriété de bégaiement (*stuttering*)	Prop 14	10	68,2
Programme de vérification	Prop 14	5	9,01
Programme de vérification	Prop 14	10	336,13

4.4 Boîtes noires

Dans une étude menée en 1986, Mills et al. [41] décomposent la description des programmes en trois niveaux d'abstraction : les boîtes noires (*Black Box*), les boîtes d'état (*State Box*) et les boîtes transparentes (*Clear Box*). Selon Mills, les boîtes noires, illustrés par la figure 4.1, sont qualifiées des vues extérieures sur le système tandis que les deux autres niveaux d'abstraction sont pour leur part considérés comme parties intégrantes de la conception du système en question.

Cette section du chapitre est précisément consacrée à l'étude des boîtes noires. Comme le souligne le modèle de Mills, la spécification des boîtes noires porte exclusivement sur l'aspect extérieur du système. En d'autres termes, les boîtes noires décrivent le comportement du système dans son environnement en assignant une réponse à chaque stimulus. On

distingue deux cas de figure spécifiques où la réponse assignée est particulière : le premier cas où le système n'est pas supposé donner de réponse et la sortie est par conséquent de type *nulle* ; et le second cas quand le stimulus envoyé est refusé par le système et la réponse assignée de type *illégal*.

figure 4.1 – Principe des boîtes noires

4.4.1 La vérification de modèles et les boîtes noires

La vérification des boîtes noires (ou la vérification des entrées et sorties du système) consiste tout d'abord à modéliser le système dans son ensemble puis ensuite à vérifier si ce dernier répond correctement aux stimuli envoyés [4]. Cette section présente une étude sur la faisabilité et le taux de complexité de la modélisation et vérification des propriétés sur les boîtes noires en ALLOY. Le cas d'une pile possédant les fonctions habituelles (*top*, *pop*, *clear*, *push*) est l'exemple témoin de cette étude. Les axiomes vérifiés ici peuvent être décomposés en deux catégories avec les axiomes de base d'un côté et les axiomes de réduction de l'autre. La différence entre chacun de ces types réside dans le fait que si les axiomes de base vérifient les sorties de l'exécution d'une action, les axiomes de réduction vérifient quant à eux la sortie de l'exécution d'une combinaison d'actions par rapport à une séquence plus courte. La figure 4.2 donne des exemples de chacun des types d'axiomes vérifiés. Par exemple, l'axiome R-8 qui est un axiome de réduction consiste à vérifier la sortie d'une transition de x à x' en exécutant l'action *push* suivie de l'action *pop*. Cette transition doit aboutir à un état x' qui est identique à celui de l'état x. L'action B-5 à son tour consiste à vérifier que la transition définie par l'action *push* suivie de l'action *top* donne en sortie le même élément que celui qui avait été inséré lors de l'action *push*.

No	Condition	Préfixe	Supprimer	Suffixe	Sortie
R-6	$x' \neq \varepsilon$	ε	x.clear	x'	
R-7	$x' \neq \varepsilon$	x	top	x'	
R-8	$x' \neq \varepsilon$	x	(push, t).pop	x'	
R-9	$x' \neq \varepsilon$	ε	pop	x'	

Axiomes de réduction

No	Prémisse	Entrée	Sortie
B-1		x.clear	τ
B-2		x.(push, t)	τ
B-3		x.pop	τ
B-4		top	erreur
B-5		x.(push, t).top	t

Axiomes de base

figure 4.2 – Axiomes de base et axiomes de réduction

4.4.2 Implémentation des boîtes noires

Dans le cas de l'implémentation d'une boîte noire, trois éléments essentiels doivent être modélisés : l'état du système, les réponses du système et la relation de transition du système. Pour reprendre l'exemple de la pile, le système correspond ici à une séquence ordonnée d'éléments, les réponses du système aux signatures représentant les trois types de réponses possibles (*Element*, *Error*, *Tau*) et enfin, le fonctionnement du système aux actions présentes sous la forme de prédicats définissant la transition du système d'un état vers un autre (soulignons par ailleurs que dans le cadre de cet exemple, aucun invariant n'a été nécessaire). Une fois que cette partie est achevée, la deuxième étape consiste alors à implémenter et vérifier les axiomes, une étape qui se réalise facilement grâce aux tables de définition. Les programmes 4.9, 4.10, 4.11 et 4.12 illustrent un exemple de chacune des étapes susmentionnées (définition des entités, définition des actions, définition des propriétés).

Dans notre cas, l'exemple de la pile est implémentée sous la forme d'une séquence. Elle est codée comme une entité héritant de la signature *Seq* qui est elle-même une signature de base dans ALLOY (son but étant, comme son nom l'indique, de faciliter l'utilisation des séquences). Concernant les différents types de sorties possibles, ces dernières sont toutes implémentées sous la forme d'entités héritant de la signature *Output* permettant ainsi à chaque type de sortie d'avoir ses propres règles. Les actions quant à elles sont implémentées sous la forme de prédicats appelant d'autres prédicats déjà définis dans le corps de la signature *Seq*. Finalement, les propriétés traduites sous la forme de prédicats sont vérifiées

lors des appels de la commande *check* comme dans le cas de la propriété B-1 illustrée par le programme 4.11. Le tableau 4.2 résume le temps d'exécution, le nombre d'entités ainsi que les propriétés vérifiées lors de l'implémentation des boîtes noires.

tableau 4.2 – Tableau récapitulatif du temps d'exécution par rapport aux bornes choisies

Propriété	Taille de la pile	Temps de vérification en (s)
Toutes	3	0,421
Toutes	5	0,624
Toutes	8	100,34
Prop B-1	10	5726907

```
sig Stack extends Seq{}
abstract sig Output{}
sig Element extends Output{}
lone sig Tau extends Output{}
lone sig Error extends Output{}
```

prog 4.9 – Exemple d'une pile

```
pred push[s,s': Stack,e : Element
    ,o: Output]
{
-- insert e at the first Index
insert[s,firstIdx,e,s']
and
-- no output
o = Tau
}
```

prog 4.10 – Action Empiler

```
-- delete the first element
pred pop[s,s' : Stack]
{
-- stack not empty
some s.seqElems
all idx : inds[s]- firstIdx |
at[s,idx] = at[s',signatures/Stack/ord/prev[idx]]
-- decrease the cardinality
#s'.inds = #s.inds - 1
}
-- propriété R-8
```

prog 4.11 – Propriété R-8 (partie 1)

```
-- if a push predicate is followed by a pop predicate,
-- then the stack still unchanged.
pred spec_push_pop_A[s,s':Stack,e:Element,o:Output]
{
some s":Stack|
push[s,s',e,o] and pop[s',s",o]
=>
s=s"
}
run spec_push_pop_A for 4
check specPushPop
{
all s,s':Stack,e:Element,o:Output|spec_push_pop_A[s,s',e,o]
} for 8
```

prog 4.12 – Propriété R-8 (partie 2)

4.5 DynAlloy

Pour répondre aux divers problèmes rencontrés au cours du projet Inovus ainsi que la difficulté d'implémentation des idiomes concernant les propriétés de types réinitialisation, de nouvelles avenues de recherche devaient être empruntées, et ce, en veillant à préserver une logique qui soit identique à celle de Alloy. Afin d'atteindre cet objectif, le vérificateur nouvelle génération DynAlloy [23] a donc été utilisé en grande partie pour la couche de dynamisme supplémentaire qu'il intègre par rapport à Alloy : en effet, les notions d'action et de propriété dynamique y ont été ajoutées afin de minimiser le travail de l'utilisateur. Grâce à cette nouvelle couche, l'intention du développeur peut donc désormais entièrement se concentrer sur les entités qui sont changées lors d'une transition, alors qu'originellement, dans Alloy il lui fallait spécifier de façon explicite ce que chaque transition modifiait ou ne modifiait pas, une contrainte qui peut parfois rendre le temps de développement long et fastidieux.

Désormais, les actions ne sont plus présentées comme des prédicats de base mais plutôt

comme des actions exprimées par le mot-clé « act » et écrites sous la forme suivante : « précondition », « postcondition ». Ici, les mots-clés « pre » et « post » définissent cet idiome. En somme, dans DynAlloy, les actions sont donc une version adaptée de prédicats permettant de passer d'une écriture totalement abstraite à une écriture que les développeurs maîtrisent mieux : celle de la programmation. En effet, le vérificateur permet une écriture de spécifications intégrant des notions de programmation de base telles que les conditions ou les boucles. DynAlloy facilite en outre la vérification d'un changement d'état sur un programme en permettant de spécifier des propriétés de la forme : "À partir d'un état E_x et en exécutant un programme P on arrive à un état E_y", ce qu'on appel la correction partielle (*partial correctness*), une spécification qui exigerait plus de temps pour être réalisée en Alloy. Les spécifications 4.13 et 4.14 illustrent la façon d'écrire des actions en DynAlloy, ainsi, elle permet à l'aide du programme 4.15 de mieux comprendre quelques concepts de dynamisme intégrés en DynAlloy. La notion de la programmation est le concept clé qui diffère le langage père du langage fils. En DynAlloy, un programme peut contenir comme dans la plupart des langages de programmation (avec quelques petites notions de plus) des déclarations de variables, des conditions (*if then else*), des boucles (*while-do*), des appels séquentiels ou non déterministes, des sous-programmes, des itérations non déterministes (*repeat*) ainsi que des axiomes (*assume*). En dépit du tableau laudatif dépeint ici, l'étude comparative de DynAlloy et Alloy devrait faire l'objet de plus de recherches dans le futur afin de mieux cerner certaines des problématiques évoquées dans cette étude.

```
--pre-condition
pred Leave_pre[]{}
--post-condition
pred Leave_post[
L:Lib,
loan:Lib->(Book->Member),
members,members':Lib->Member,
membersReservingOneBook:Lib->((Int->Member)->Book)
]
{
```

prog 4.13 – La spécification de l'action *Leave* avec DynAlloy (partie 1)

```
some m : Member
{
m in L.members
no (L.loan.m)
m !in Int.(L.membersReservingOneBook.Book)
-- "members' " is automatically interpreted by Dynalloy as
-- the next state of "members"
members' = members - L->m
}
---action--
act Leave[
L:Lib,
m:Member,
members:Lib->Member,
loan:Lib->(Book->Member),
membersReservingOneBook:
Lib->((Int->Member)->Book)
]
{
-- pre : a key word that includes the preconditions required
-- to satisfy the transition from a state s to a state s'
pre
{
-- same precondition as in the ALLOY specification
Leave_pre[L,m,members,loan,membersReservingOneBook]
}
-- post : a key word that includes the postconditions required to satisfy
-- the transition from a state s to a state s'
post
{
Leave_post[L,m,members,members']
```

prog 4.14 – La spécification de l'action *Leave* avec DynAlloy (partie 2)

```
}
}
pred NoChangeloan[L,L':Lib]
{
L.loan=L'.loan
}
pred Leave[m:Member,L,L':Lib]
{
---checking---
CanLeave[m,L]
---treatement--
L'.members = L.members - m
---no change---
NoChangeloan[L,L']
NochangeRenew[L,L']
NoChangeSeqBook[L,L']
NoChangebooks[L,L']
}
```

prog 4.15 – La spécification de l'action *Leave* avec ALLOY

Conclusion

Après avoir dressé un tableau comparatif entre six vérificateurs de modèles dont ALLOY, la présente étude s'est attelée à démontrer l'efficacité de ce dernier dans le domaine de « l'informatique ubiquitaire ». De cette étude comparative, nous en avons conclu dans un premier temps qu'un bon vérificateur de modèles en était un polyvalent, c'est-à-dire ayant la capacité de supporter à la fois les états et les évènements. Grâce à sa syntaxe proche d'un langage orienté objet et sa logique du premier ordre, ALLOY s'approche assez bien de cet idéal, même si l'on a constaté que certaines de ses limitations intrinsèques avaient la capacité d'en circonscrire le champ d'application parfois. À cet égard, l'exemple de la bibliothèque a représenté un bon cas d'illustration des systèmes d'information, ce qui au-delà d'être une gageure de la validité de cette étude, doit encourager les chercheurs à conduire davantage la recherche dans le futur sur le sujet afin d'identifier d'autres avantages et limitations que ALLOY pourrait avoir. Cette première phase de l'étude ne portant que sur les séquences d'actions qu'un système d'information doit accepter, la seconde phase de l'étude s'est, à l'inverse, concentrée sur la capacité de ALLOY à pouvoir s'adapter à la vérification des sorties produites par les systèmes, et ce, par l'entremise de l'étude d'un autre cas d'espèce : celui des « boîtes noires ». Les résultats collectés à l'issue de cette seconde phase ont démontré l'excellente capacité de ALLOY à répondre à ce type de problématique. Enfin, le dernier pan de l'étude s'est quant à lui concentré sur une nouvelle manière de développer des applications pour les maisons intelligentes en s'assurant que les modèles sur lesquels celles-ci reposent sont bel et bien cohérents. Pour ce faire, nous avons donc introduit en ALLOY une couche de sécurité supplémentaire au processus de vérification des modèles desdites applications afin de s'assurer que celles-ci reposaient bel et bien sur des modèles cohérents. Grâce à la facilité de traduction des diagrammes UML en ALLOY, nous en avons conclu que le vérificateur (ALLOY) était *derechef* un très bon outil pour assurer la

cohérence des modèles ainsi que la vérification des propriétés. Néanmoins, par sa difficulté à spécifier des systèmes dynamiques, nous en avons de nouveau dégagé que son extension dynamique, DYNALLOY, était encore une fois la solution la mieux adaptée des deux pour le faire.

Ainsi que susmentionné, deux écueils majeurs ont surgi au cours de la recherche. Le premier d'entre eux concerne les propriétés de type CTL et, plus spécifiquement, celles de type réinitialisation. En effet, en raison de la limitation absolue intrinsèque à ALLOY, la génération du graphe de transition au complet était impossible au cours de la première étude, celle menée sur le cas de la bibliothèque, ce qui avait pour conséquence d'empêcher l'expression desdites propriétés. Pour résoudre ce problème, nous avons donc dû développer des idiomes *ad hoc* dont la limitation majeure est qu'ils n'étaient pas généralisables à l'ensemble des propriétés de type CTL. Le second écueil rencontré est pour sa part inhérent au *modus operandi* de ALLOY. En effet, une des interrogations majeures qu'il restait à résoudre à l'issue de la première phase de l'analyse consistait à savoir si ALLOY s'adapterait ou non avec autant d'efficacité à un système d'information plus « large » que celui étudié jusque là. Au vu des observations faites et des résultats obtenus à l'issue de cette seconde phase (le système traité ici était une cuisinière), nous avons constaté que ALLOY souffrait d'un manque de concision dans la description du dynamisme limitant de *facto* sa capacité à gérer des systèmes possédant un grand nombre d'identités. Pour contourner ce problème, nous avons donc suggéré l'utilisation d'une version plus récente du vérificateur, DYNALLOY, dont les fonctionnalités dynamiques permettent de mieux gérer les spécificités de tels systèmes.

Dans le futur, et à l'instar de la méthodologie adoptée dans ce mémoire, les recherches en sécurité des systèmes d'information devront donc s'orienter vers une investigation plus poussée et approfondie de DYNALLOY, une meilleure compréhension de ses fonctionnalités, ainsi qu'une identification ciblée de ses forces et faiblesses. Un tel « profilage » pourra ainsi être posé en complément à celui réalisé dans le cadre de cette étude sur ALLOY, permettant de ce fait aux développeurs de savoir à quel contexte l'utilisation d'un de ces deux outils s'apprête le mieux. D'autres recherches devront par ailleurs investir la problématique de conversion des diagrammes UML en ALLOY ou en DYNALLOY dans l'optique d'automatiser le processus de traduction des langages. Une telle automatisation permettrait ainsi de réduire considérablement la tâche des programmeurs, dont une bonne partie du temps

de travail est encore trop souvent consacrée à la traduction des langages de modélisation graphique en langage supporté par les vérificateurs de modèle.

Annexe A

Spécification de la bibliothèque en ALLOY

```
module model
open util/sequniv as ResSq
--Signatures specification
one sig Constants
{
maxNbLoans : Int
}
{
maxNbLoans = 7
}
sig Book{}
sig Member{}
sig Lib
{
members:set Member,
books: set Book ,
loan: (books -> members),
membersReservingOneBook: (seq members)->books,
Renew: (books -> members)
}
```

prog A.1 – La spécification de la bibliothèque (partie 1)

```
-- Nochange predicates are used
-- to describe sets that remains unchanged
-- when a transition from a
-- state s to a state s' is done

pred NoChangebooks[L,L':Lib]
{
L.books =L'.books
}
pred NoChangemembers[L,L':Lib]
{
L.members =L'.members
}
pred NoChangeloan[L,L':Lib]
{
L.loan=L'.loan
}
pred NoChangeSeqBook[L,L':Lib]
{
L.membersReservingOneBook= L'.membersReservingOneBook
}
pred NoChangeRenew[L,L':Lib]
{
L.Renew = L'.Renew
}
/*----------------
   Initialization
------------------*/
pred Init [L:Lib]
{
no L.books
```

prog A.2 – La spécification de la bibliothèque (partie 2)

```
no L.members
no L.loan
no L.membersReservingOneBook
no L.Renew
}
/*--------- ----------
   Acquire
-------------------*/
pred CanBeAcquire[L:Lib,b:Book]
{
-- check that b is acquired by the library
no(b & L.books)
}
pred Acquire[b:Book,L,L':Lib]
{
---Preconditon---
CanBeAcquire[L,b]
---Postcondition---
-- add b in the set of acquired books
L'.books = L.books + b
---No Changes--
NoChangemembers[L,L']
NoChangeloan[L,L']
NoChangeSeqBook[L,L']
NoChangeRenew[L,L']
}
/*--------- ----------
   Join
-------------------*/
pred CanJoin[m:Member,L:Lib]
{
```

prog A.3 – La spécification de la bibliothèque (partie 3)

```
-- check that m is not registered in the library
no (m & L.members)
}
pred Join[m:Member,L,L':Lib]
{
---Precondition--
CanJoin[m,L]
---Postcondition--
-- add m in the set of registered members
L'.members=L.members +m
----No changes--
NoChangebooks[L,L']
NoChangeloan[L,L']
NoChangeSeqBook[L,L']
NoChangeRenew[L,L']
}
/*-------------------
   LEND
-------------------*/
pred CanLend[m:Member,b:Book,L:Lib]
{
(b in L.books) and (m in L.members)
-- the highest number of authorized loans
(#((L.loan).m)<Constants.maxNbLoans)
-- b is not lent
all m':Member|no((L.loan).m' & b)
-- b is not reserved
(no (L.membersReservingOneBook.b))
}
pred Lend[m:Member,b:Book,L,L':Lib]
{
```

prog A.4 – La spécification de la bibliothèque (partie 4)

```
---Precondition----
CanLend[m,b,L]
}
---Postcondition-----
L'.loan=L.loan + (b->m)
---No changes----
NoChangebooks[L,L']
NoChangemembers[L,L']
NoChangeSeqBook[L,L']
NoChangeRenew[L,L']
/*-------------------
    RESERVE
-------------------*/
pred CanReserve[m:Member,b:Book,L:Lib]{
(b in L.books and m in L.members )
-- b is borrowed
one (b & ((L.loan).Member)) or (some (L.membersReservingOneBook.b))
-- b is not lent by m
no (m & b.(L.loan))
-- b is not reserved by m
no (Int.(L.membersReservingOneBook.b) & m)
}
pred Reserve[m:Member,b:Book,L,L':Lib]{
--- Precondition--
CanReserve[m,b,L]
----PostCondition--
L'.membersReservingOneBook.b = ResSq/add[L.membersReservingOneBook.b,m]
---No changes---
all b':Book - b|
L'.membersReservingOneBook.b' = L.membersReservingOneBook.b'
NoChangebooks[L,L']
NoChangemembers[L,L']
```

prog A.5 – La spécification de la bibliothèque (partie 5)

```
NoChangeloan[L,L']
NoChangeRenew[L,L']
}
/*-------------------
    CANCEL
-------------------*/
pred CanCancel[m:Member,b:Book,L:Lib]
{
(b in L.books and m in L.members )
-- b is reserved by m
one (Int->m & (L.membersReservingOneBook.b))
}
pred Cancel[m:Member,b:Book,L,L':Lib]
{
----Preconditon-----
CanCancel[m,b,L]
----Postconditon----
-- delete m from the b reservation list
L'.membersReservingOneBook.b=ResSq/delete[L.membersReservingOneBook.b,ResSq
    /indsOf [L.membersReservingOneBook.b,m]]
----No changes---
all b':Book - b|
L'.membersReservingOneBook.b' = L.membersReservingOneBook.b'
NoChangebooks[L,L']
NoChangemembers[L,L']
NoChangeloan[L,L']
NoChangeRenew[L,L']
}
/*--------- ----------
    RETURN
--------------------*/
```

prog A.6 – La spécification de la bibliothèque (partie 6)

```
pred CanReturn[m:Member,b:Book,L:Lib]
{
(b in L.books and m in L.members )
-- b is already lent by m
one ((L.loan).m & b)
}
pred Return[m:Member,b:Book,L,L':Lib]
{
---Precondition--
CanReturn[m,b,L]
---PostConditon---
-- delete b->m from the set of loans
L'.loan=L.loan - (b ->m)
L'.Renew = L.Renew - (b -> m)
---No changes---
NoChangebooks[L,L']
NoChangemembers[L,L']
NoChangeSeqBook[L,L']
}
/*--------------------
  TAKE
----------------------*/
pred CanTake[m:Member,b:Book,L:Lib]
{
-- b and m are acquired by the library
(b in Lib.books) and (m in L.members)
(#((L.loan).m)<Constants.maxNbLoans)
-- check the position of the member m (the top of the reservation list)
(L.membersReservingOneBook.b) = (0 -> m)
 no (b.(L.loan))
 }
```

prog A.7 – La spécification de la bibliothèque (partie 7)

```
pred Take[m:Member,b:Book,L,L':Lib]
{
---Preconditon---
CanTake[m,b,L]
---PostCondition--
L'.loan=L.loan + (b->m)
-- deletes m from the b reservation list
L'.membersReservingOneBook.b=ResSq/delete[L'.membersReservingOneBook.b,0]
---No changes---
all b':Book - b|
L'.membersReservingOneBook.b' = L.membersReservingOneBook.b'
NoChangebooks[L,L']
NoChangemembers[L,L']
NoChangeRenew[L,L']
}
/*-----------------
  LEAVE
-------------------*/
pred CanLeave[m:Member,L:Lib]
{
m in L.members
-- m is not in the loan list
no (L.loan.m)
-- m has no reservation
no( Int.(L.membersReservingOneBook.Book) & m)
}
pred Leave[m:Member,L,L':Lib]
{
----Preconditon---
CanLeave[m,L]
----Postconditon--
```

prog A.8 – La spécification de la bibliothèque (partie 8)

```
L'.members = L.members - m
----No changes---
NoChangeloan[L,L']
NoChangeRenew[L,L']
NoChangeSeqBook[L,L']
NoChangebooks[L,L']
}
/*-----------------
DISCARD
-------------------*/
pred CanDiscard[b:Book,L:Lib]
{
b in L.books
no (b.(L.loan))
no ((L.membersReservingOneBook.b))
}
pred Discard[b:Book,L,L':Lib]
{
----Precondition---
CanDiscard[b,L]
----Postconditon---
L'.books = L.books - b
---No changes---
NoChangeloan[L,L']
NoChangeSeqBook[L,L']
NoChangemembers[L,L']
NoChangeRenew[L,L']
}
/*--------------------
  RENEW
----------------------*/
```

prog A.9 – La spécification de la bibliothèque (partie 9)

```
pred CanRenew[m:Member,b:Book,L:Lib]
{
-- b is already borrowed by m
one (b.(L.loan) & m)
-- b has no reservation
ResSq/isEmpty [L.membersReservingOneBook.b]
}
pred Renew[m:Member,b:Book,L,L':Lib]
{
----Preconditon---
CanRenew[m,b,L]
---Postcondition---
-- override the old b->m
L'.Renew=L.Renew ++ (b->m)
----No changes--
NoChangebooks[L,L']
NoChangemembers[L,L']
NoChangeloan[L,L']
NoChangeSeqBook[L,L']
}
```

prog A.10 – La spécification de la bibliothèque (partie 10)

```
--All properties written except 14 and 15
open LibSpecification
 --1. A book can always be acquired by the library
 --when it is not currently acquired.
--2. A book cannot be acquired by the library
--if it is already acquired.
assert Prop1And2
{
all b:Book,L:Lib
{
CanBeAcquire[L,b] <=> not (b in L.books)
}
}
check Prop1And2 for 2 Lib,8 Member,8 Book
--3. An acquired book can be discarded only
--if it is neither lent nor reserved.
assert Prop3
{
all b:Book,L:Lib
{
some L':Lib |Discard[b,L,L'] => no (b.(L.loan)) and no ((L.
    membersReservingOneBook.b) )
}
}
check Prop3 for 2 Lib,8 Member,8 Book
--4. A person must be a member of the library
--in order to borrow(lend or take) a book.
assert Prop4
{
all b:Book,m:Member,L:Lib
{
```

prog A.11 – Spécification des propriétés en ALLOY (partie 1)

```
some L':Lib|Lend[m,b,L,L'] => (b in L.books) and (m in L.members)
some L':Lib|Take[m,b,L,L'] => (b in L.books) and (m in L.members)
}
}
check Prop4 for 2 Lib,8 Member,8 Book
--5. A book can be reserved only if it has been borrowed or
--already reserved by another member
assert Prop5
{
all b:Book,m:Member,L:Lib
{
some L':Lib|Reserve[m,b,L,L']
=>
(b in ((L.loan).Member))
or
(some (L.membersReservingOneBook.b)
)
}
}
check Prop5 for 2 Lib,8 Member,8 Book
--6. A book cannot be reserved by the member who is borrowing it.
assert Prop6
{
all b:Book,m:Member,L:Lib
{
some L':Lib|Reserve[m,b,L,L'] => ((m !in b.(L.loan)))
}
}
check Prop6 for 2 Lib,8 Member,8 Book
--7. A book cannot be reserved by a member who is reserving it.
assert Prop7
{
```

prog A.12 – Spécification des propriétés en ALLOY (partie 2)

```
all b:Book,m:Member,L:Lib
{
some L':Lib|Reserve[m,b,L,L'] => ( (m !in Int.(L.membersReservingOneBook.b)
)
)
}
}
check Prop7 for 2 Lib,8 Member,8 Book
--8. A book cannot be lent to a member if it is reserved
assert Prop8
{
all b:Book,m:Member,L:Lib
{
some L':Lib|Lend[m,b,L,L'] => (no (L.membersReservingOneBook.b))
}
}
check Prop8 for 2 Lib,8 Member,8 Book
--9. A member cannot renew a loan if the book is reserved
assert Prop9
{
all b:Book,m:Member,L:Lib
{
some L':Lib|Renew[m,b,L,L'] => (no (L.membersReservingOneBook.b))
}
}
check Prop9 for 2 Lib,8 Member,8 Book
--10. A member is allowed to take a reserved book
--only if he owns the oldest reservation
assert Prop10
{
all b:Book,m:Member,L:Lib
{
```

prog A.13 – Spécification des propriétés en ALLOY (partie 3)

```
some L':Lib|Take[m,b,L,L'] => (L.membersReservingOneBook.b) = (0 -> m)
}
}
check Prop10 for 2 Lib,8 Member,8 Book
--11. A book can be taken only if it is not borrowed
assert Prop11
{
all b:Book,m:Member,L:Lib
{
some L':Lib|Take[m,b,L,L'] => not ( b in (L.loan).Member)
}
}
check Prop11 for 2 Lib,8 Member,8 Book
--12. Anyone who has reserved a book can cancel
--the reservation at anytime before he takes it
assert Prop12
{
all b:Book,L:Lib,m:Member
{
some L':Lib| Take[m,b,L,L'] => CanCancel[m,b,L']
}
}
check Prop12 for 2 Lib,8 Member,8 Book
--13. A member can relinquish library membership only when all his
--loans have been returned and all his reservations
--have either been used or canceled
assert Prop13
{
all m:Member,L:Lib
{
some L':Lib|Leave[m,L,L'] => (no (L.loan.m) and (m !in Int.(L.
   membersReservingOneBook.Book)))
```

prog A.14 – Spécification des propriétés en ALLOY (partie 4)

```
}
}
check Prop13 for 2 Lib,8 Member,8 Book
--15. A member cannot borrow more than the loan
--limit defined at the system level for all users
assert Prop15
{
all L,L':Lib,m:Member,b:Book
{
Take[m,b,L,L'] => #(L'.loan.m)<=7
Lend[m,b,L,L'] => #(L'.loan.m)<=7
}
}
check Prop15 for 2 Lib, 8 Member, 8 Book
```

prog A.15 – Spécification des propriétés en ALLOY (partie 5)

```
open LibSpecification
open util/IJALSequence[Lib] as LibSeq1
open util/LCRSequence[Lib] as LibSeq2
---------TRACES---------
fact
{
Init[LibSeq1/first[Seq]]
all idx : LibSeq1/inds[Seq] - LibSeq1/lastIdx[Seq]|
some m:Member,b:Book|
Join[m,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
or Acquire[b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
or Lend[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
or Reserve[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
or Take[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
or Renew[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
}
--------Prop14(CTL)--------
----L=Leave,C=Cancel,R=Return---
pred LCR[m:Member,L,L' : Lib]
{
some b:Book |
Cancel[m,b,L,L']
or Return[m,b,L,L']
or Leave[m,L,L']
}
---CanLCR : forces the analyzer to end the transition -
pred CanLCR[L:Lib,m:Member]
{
CanLeave[m,L] or
some b:Book|CanCancel[m,b,L] or
some b :Book|CanReturn[m,b,L]
}
```

prog A.16 – La propriété 14 en ALLOY avec un état initial (partie 1)

```
--TransLCR : predicates that the analyzer
--is able to satisfy during the SeqLCR (LCR) building
pred TransLCR[m:Member,P:SeqLCR]
{
-- all Sequence indexs except the last
all idx : LibSeq2/inds[P]-LibSeq2/lastIdx[P] |
-- LCR[L,L.next]
LCR[m,LibSeq2/at[P,idx],LibSeq2/at[P,idx.LibSeq2/ord/next]]
}
pred NegProp14 []
{
one SeqLCR
some P : SeqLCR|
some  m : LibSeq1/last[Seq].members |
TransLCR[m,P]
and m in LibSeq2/last[P].members
and
(
not CanLCR[LibSeq2/last[P],m]
or
--the SeqLCR last index is the final index(max Seq)
LibSeq2/lastIdx[P] = LibSeq2/finalIdx
)
}
run{ NegProp14[]}for 3 Member,3 Book,36 Lib, 13 SeqIdx,13 SeqIdx1,13 SeqLCR
--the last library state in the IAJLR sequence is the
-- first library state in the LCR
fact
{
LibSeq1/last[Seq] = LibSeq2/first[SeqLCR]
}
```

prog A.17 – La propriété 14 en ALLOY avec un état initial (partie 2)

```
--compare the solving time between running a model
-- with predicates and running the same model with assert.
assert Prop14
{
all P : SeqLCR|
all m: LibSeq1/last[Seq].members|
TransLCR[m,P] // OrLCR
=>
(
(
not m in LibSeq2/last[P].members)
or
(
-- incomplete sequence => still possible to do an LCR
( CanLCR[LibSeq2/last[P],m]
and
(not LibSeq2/lastIdx[P] = LibSeq2/finalIdx)
)
)
)
}
check Prop14 for 4 Member,4 Book, 50 Lib, 25 SeqIdx,25 SeqIdx1,25 SeqLCR
```

prog A.18 – La propriété 14 en ALLOY avec un état initial (partie 3)

```
open LibSpecification
open util/sequence[Lib] as LibSeq
--use traces to check property 14.
--traces starts from a valid state,
--defined by a fact
pred BuggyLeave[L,L':Lib]{L=L'}
pred LCR[m:Member,L,L' : Lib]
{
some b:Book |
Cancel[m,b,L,L']
or Return[m,b,L,L']
or Leave[m,L,L']
}
pred TransLCR[m:Member,P:Seq]
{
all idx : LibSeq/inds[P]-LibSeq/lastIdx[P] |
LCR[m,LibSeq/at[P,idx],LibSeq/at[P,idx.LibSeq/ord/next]]
}
pred OrCanLCR[L:Lib,m:Member]
{
CanLeave[m,L]
or some b:Book| CanCancel[m,b,L]
or some b :Book| CanReturn[m,b,L]
}
--use "run" command to check the property 14
pred NegProp14
{
-- start from a valid library state
all P : Seq | Prop14StartLib[LibSeq/first[P]]
some P : Seq |
some m : LibSeq/first[P].members |
```

prog A.19 – La propriété 14 en ALLOY sans un état initial prédéfini (partie 1)

```
TransLCR[m,P]
and m in LibSeq/last[P].members
and (
-- LCR can not be applied
not OrCanLCR[LibSeq/last[P],m]
or
-- reache maximum sequence's length
LibSeq/lastIdx[P] = LibSeq/finalIdx
  )
}
run NegProp14 for 10 but 8 Book, 8 Member
run NegProp14 for 8 but 6 Book, 6 Member
run BuggyLeave for 5
run {} for 3 but 2 Lib
-- use the "check" command to check the property 14.
assert Prop14
{
all P : Seq |
all m : LibSeq/first[P].members |
(
Prop14StartLib[LibSeq/first[P]]
and
TransLCR[m,P]
)
  =>
(
(
not m in LibSeq/last[P].members)
or
-- the sequence is incomplete
(
```

prog A.20 – La propriété 14 en ALLOY sans un état initial prédéfini (partie 2)

```
OrCanLCR[LibSeq/last[P],m]
and
not
LibSeq/lastIdx[P] = LibSeq/finalIdx
)
)
}
check Prop14 for 10 but 8 Book, 8 Member
-- define the first valid library state of the trace
pred Prop14StartLib[L:Lib]
{
all m: Member| (#((L.loan).m)<=Constants.maxNbLoans)
-- books can not be reserved if it's not loaned or reserved
all b : Book|
some (L.membersReservingOneBook.b)
=>
((b in L.loan.Member) or ((#L.membersReservingOneBook.b) > 0))
-- members can not reserve and lend the same book
all m : Member,b : Book|
(m in b.(L.loan) => not (m in Int.(L.membersReservingOneBook.b)))
all m : Member,b : Book|
(m in Int.(L.membersReservingOneBook.b)) => not (m in b.(L.loan))
-- members can not renew a book and he has not lent it before.
all b : Book,m : Member|
(b in L.Renew.m) =>b in (L.loan.m)
-- the relation books -> lone members
all b : Book|
((b in L.books) and (b in L.loan.Member)) => one (b ->Member & (L.loan))
all b : Book|
((b in L.books) and (b in L.Renew.Member)) => one (b ->Member & (L.Renew))
-- valid book reservation sequences.
```

prog A.21 – La propriété 14 en ALLOY sans un état initial prédéfini (partie 3)

```
all m:Member,b:Book |
(m in Int.(Lib.membersReservingOneBook.b))
 =>
one (Int->m & (Lib.membersReservingOneBook.b))
}
```

prog A.22 – La propriété 14 en ALLOY avec un état initial prédéfini (partie 4)

Bibliographie

[1] J.R. Abrial.
The B-Book : Assigning Programs to Meanings.
Cambridge University Press, 1996.

[2] J.R. Abrial.
Modeling in Event-B : System and Software Engineering.
Cambridge University Press, 2010.

[3] M. Ahrens.
« Home Smoke Alarms : The Data as Context for Decision ».
Fire Technology, 44:313–327, 2008.

[4] L. Ben Arfa, M. Frappier, R. Mili, A. Mili et R. Douglas Skuce.
« A process for verification based inspections ».
Dans *SEKE 94, The 6th International Conference on Software Engineering and Knowledge Engineering, June 21-23, 1994, Jurmala, Latvia*, pages 100–107. Knowledge Systems Institute, 1994.

[5] J.C. Augusto, C. Ferreira, A. Gravell, M. Leuschel et M.Y.K. Ng.
« The benefits of rapid modelling for e-business system development ».
ER Workshops, pages 17–28, 2003.

[6] E.G. Aydal, M. Utting et J.Woodcock.
« A comparison of state-based modelling tools for model validation ».
TOOLS-Europe08, Switzerland, 2008.

[7] C. Baier et J.P. Katoen.
Principles of Model Checking (Representation and Mind Series).
The MIT Press, 2008.

[8] A.J. BHARUCHA, V. ANAND, J. FORLIZZI, M.A. DEW, C.F. REYNOLDS, S. STEVENS et H. WACTLAR.
« Intelligent Assistive Technology, Applications to Dementia Care : Current Capabilities, Limitations, and Future Challenges. ».
The American Journal of Geriatric Psychiatry, 17:88–104, 2009.

[9] A. BIERE, E.M. CLARKE, A. CIMATTI et Y. ZHU.
« Symbolic Model Checking without BDDs ».
Dans *Proceeding of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, volume 1579 de *LNCS*, pages 193–207, 1999.

[10] T. BOLOGNESI et Ed. BRINKSMA.
« Introduction to the ISO Specification Language LOTOS ».
Dans P. H. J. van EIJK, C. A. VISSERS et M. DIAZ, éditeurs, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers B.V., 1989.

[11] G. BOOCH, J. RUMBAUGH et I. JACOBSON.
Unified Modeling Language User Guide, The (Addison-Wesley Object Technology Series).
Addison-Wesley Professional, 2005.

[12] S.E. CARTER, E.M. CAMPBELL, R.W. SANSON-FISHER, S. REDMAN et W.J. GILLESPIE.
« Environmental hazards in the homes of older people ».
Age and Ageing, 26(3):195–202, 1997.

[13] R. CHANE-YACK-FA, B. FRAIKIN, M. FRAPPIER, R. CHOSSARD et M. OUENZAR.
« Comparison of Model Checking Tools for Information Systems ».
Rapport Technique 29, Université de Sherbrooke, juin 2010, Disponible from `http://pages.usherbrooke.ca/gril/TR/TR-GRIL-1006-29.pdf`.

[14] E.M. CLARKE et E.A. EMERSON.
« Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic ».
Dans *Logic of Programs, Workshop*, pages 52–71. Springer-Verlag, 1982.

[15] A. DEUTSCH, L. SUI et V. VIANU.
« Specification and verification of data-driven web applications ».
Journal of Computer and System Sciences, 73(3):442–474, 2007.

[16] N. D'IPPOLITO, M. FRIAS, J. GALEOTTI, E. LANZAROTTI et S. MERA.
« Alloy+HotCore : A Fast Approximation to Unsat Core ».
Dans *ASM*, pages 160–173, 2010.

[17] M.C. EDMUND et E. E. ALLEN.
« Synthesis of Synchronization Skeletons for Branching Time Temporal Logic ».
Dans Dexter KOZEN, éditeur, *Logic of Programs Workshop*, volume 131 de *LNCS*, pages 52–71. Springer-Verlag, 1981.

[18] A. EGYED.
« Instant consistency checking for the UML ».
Dans *ICSE '06 : Proceedings of the 28th international conference on Software engineering*, pages 381–390. ACM, 2006.

[19] E.A. EMERSON et J.Y. HALPERN.
« "Sometimes" and "Not Never" revisited : On branching Versus Linear Time Temporal Logic ».
J. ACM, 33(1):151–178, 1986.

[20] P. FRADET, D. MÉTAYER et M. PÉRIN.
« Consistency Checking for Multiple View Software Architectures ».
Dans Oscar NIERSTRASZ et Michel LEMOINE, éditeurs, *Software Engineering — ESEC/FSE '99*, volume 1687 de *Lecture Notes in Computer Science*, pages 410–428. Springer Berlin / Heidelberg, 1999.

[21] M. FRAPPIER, B. FRAIKIN, R. CHOSSART, R. CHANE-YACK-FA et M. OUENZAR.
« Comparison of Model Checking Tools for Information Systems ».
Dans *12th International Conference on Formal Engineering Methods (ICFEM 2010)*. Springer, 2010.

[22] M. FRAPPIER, B. FRAIKIN, F. GERVAIS, R. LALEAU et M. RICHARD.
« Synthesizing Information Systems : the APIS Project ».
Dans *RCIS*, pages 73–84, 2007.

[23] M. FRIAS, J.P. GALEOTTI, C. Lopez POMBO et N. AGUIRRE.
« DynAlloy : upgrading Alloy with actions ».
Dans Gruia-Catalin ROMAN, éditeur, *ICSE 2005 : 27th International Conference on Software Engineering*, pages 442–450. ACM Press, 2005.

[24] H. GARAVEL.
« *Compilation et vérification de programmes LOTOS* ».
Thèse de doctorat, Université Joseph Fourier, Grenoble, November 1989.

[25] H. GILMOUR.
« Living alone with dementia : risk and the professional role. ».
Nursing Older People, 16(9):20–24, 2004.

[26] B. HAILPERN et P. SANTHANAM.
« Software debugging, testing, and verification ».
IBM Systems Journal, 41(1):4–12, 2002.

[27] R. HARPER et W. DICKSON.
« Reducing the burn risk to elderly persons living in residential care ».
Burns, 21(3):205–208, 1995.

[28] C.L. HEITMEYER, R.D. JEFFORDS et B.G. LABAW.
« Automated consistency checking of requirements specifications ».
ACM Transactions on Software Engineering and Methodology, 5(3):231–261, 1996.

[29] A.J. HILL, F. GERMA et J.C. BOYLE.
« Burns in older people—outcomes and risk factors ».
Journal of the American Geriatrics Society, 50(11):1912–1913, 2002.

[30] G.J HOLZMANN.
The Spin Model Checker : Primer and Reference Manual.
Addison-Wesley, 2004.

[31] Z. HUZAR, L. KUZNIARZ, G. REGGIO et J.L. SOURROUILLE.
« Consistency Problems in UML-Based Software Development ».
Dans Nuno NUNES, Bran SELIC, Alberto Rodrigues da SILVA et Ambrosio TOVAL ALVAREZ, éditeurs, *UML Modeling Languages and Applications*, volume 3297 de *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 2005.

[32] D. JACKSON.
Software Abstractions.
MIT Press, 2006.

[33] J.L. JACOB.
« Trace Specifications in Alloy ».
Dans *ASM*, pages 105–117, 2010.

[34] K. KALLIN, J. JENSEN, L. OLSSON, L. NYBERG et Y. GUSTAFSON.
« Why the elderly fall in residential care facilities, and suggested remedies ».
The Journal of Family Practice, 53(1):41–527, 2004.

[35] M. LEUSCHEL et M. BUTLER.
« ProB : A Model Checker for B ».
Dans Keijiro ARAKI, Stefania GNESI et Dino MANDRIOLI, éditeurs, *FME 2003 : Formal Methods*, volume 2805 de *LNCS*, pages 855–874. Springer-Verlag, 2003.

[36] L. MACKENZIE, J. BYLES et N. HIGGINBOTHAM.
« Designing the Home Falls and Accidents Screening Tool (HOME FAST) : selecting the Items ».
The British Journal of Occupational Therapy, 63:260–269, 2000.

[37] S. MALIK et L. ZHANG.
« Boolean satisfiability from theoretical hardness to practical success ».
Commun. ACM, 52(8):76–82, 2009.

[38] R. MATEESCU et H. GARAVEL.
« XTL : A meta-language and tool for temporal logic model-checking ».
Dans *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98*, page 10, juillet 1998.

[39] K.L. MCMILLAN.
« *Symbolic Model Checking* ».
Thèse de doctorat, Carnegie Mellon University, 1993.

[40] B. MEYER.
« On Formalism in Specifications ».
IEEE Software, 2(1):6–26, 1985.

[41] H.D. MILLS, R.C. LINGER et A.R. HEVNER.
Principles of information systems analysis and design.
Academic Press Professional, Inc., 1986.

[42] F.G. MISKELLY.
« Assistive technology in elderly care ».
Age and Ageing, 30(1):455–458, 2001.

[43] S. MORIMOTO.
« A Survey of Formal Verification for Business Process Modeling ».

Dans *Proceedings of the 8th international conference on Computational Science, Part II*, ICCS '08, pages 514–522. Springer-Verlag, 2008.

[44] B. NUSEIBEH, S. EASTERBROOK et A. RUSSO.
« Making inconsistency respectable in software development ».
Journal of Systems and Software, 58(2):171–180, 2001.

[45] M. PEZZÈ et M. YOUNG.
Software Testing and Analysis : Process, Principles, and Techniques.
Wiley, 2008.

[46] S.L. PFLEEGER et J.M. ATLEE.
Software Engineering : Theory and Practice.
Pearson Prentice Hall, 3 édition, 2006.

[47] A. PNUELI.
« The temporal logic of programs ».
Dans *Foundations of Computer Science, 18th Annual Symposium on*, pages 46–57, 1977.

[48] J.P. QUEILLE et J. SIFAKIS.
« Specification and verification of concurrent systems in CESAR ».
Dans *Symposium on Programming*, pages 337–351, 1982.

[49] J.P. QUEILLE et J. SIFAKIS.
« A Temporal Logic to Deal with Fairness in Transition Systems ».
Dans *FOCS*, pages 217–225, 1982.

[50] A.W. ROSCOE.
The Theory and Practice of Concurrency.
Prentice Hall PTR, amended 2005, 3rd édition, 1998.

[51] A. SORCINELLI, L. SHAW, A. FREEMAN et K. COOPER.
« Evaluating The Safe Living Guide : A Home Hazard Checklist for Seniors ».
Canadian Journal on Aging, 26(2):127–137, 2007.

[52] M. SPIELMANN.
« *Abstract state machines : Verification problems and complexity* ».
Thèse de doctorat, Bibliothek der RWTH Aachen, 2000.

[53] J.M. SPIVEY.
Understanding Z : a specification language and its formal semantics.
Cambridge University Press, 1988.

[54] G. THEOPHILOS, D.J. DOUGHERTY, F. KATHI et K. SHRIRAM.
« Towards an Operational Semantics for Alloy ».
Dans *FM '09 : Proceedings of the 2nd World Congress on Formal Methods*, pages 483–498. Springer-Verlag, 2009.

[55] K.M. THOM et S.E. BLAIR.
« Risk in Dementia Assessment and Management : a Literature Review ».
The British Journal of Occupational Therapy, 61:441–447, 1998.

[56] E. TORLAK et D. JACKSON.
« Kodkod : a relational model finder ».
Dans *TACAS'07 : Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, pages 632–647. Springer-Verlag, 2007.

[57] P. WILLIAMS, A. BIERE, E. CLARKE et A. GUPTA.
« Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking ».
Computer Aided Verification, pages 124–138, 2000.

[58] W.L. YEUNG, K. LEUNG, J. WANG et W. DONG.
« Modelling and model checking suspendible business processes via statechart diagrams and CSP ».
Science of Computer Programming, 65(1):14–29, 2007.