Extension événementielle d'une méthode formelle légère et application à l'analyse du protocole distribué Chord

Jeanne Odette Hortense Matiedje Tawa

▶ To cite this version:

Jeanne Odette Hortense Matiedje Tawa. Extension événementielle d'une méthode formelle légère et application à l'analyse du protocole distribué Chord. Réseaux et télécommunications [cs.NI]. Institut Supérieur de l'Aéronautique et de l'Espace (ISAE), 2019. Français. NNT : . tel-02427484

HAL Id: tel-02427484
https://hal.science/tel-02427484

Submitted on 3 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)*

Présentée et soutenue le 04 Octobre 2019 **par :**

MATIEDJE TAWA JEANNE ODETTE HORTENSE

Extension événementielle d'une méthode formelle légère et application à l'analyse du protocole distribué Chord

---

JURY

Sylvain CONCHON	Université Paris-Sud	Examinateur
Virginie WEILS	ONERA	Examinatrice
Régine LALEAU	Université Paris-Est Créteil	Rapporteur
Dominique MERY	Université de Lorraine	Rapporteur
Julien BRUNEL	ONERA	Membre du jury
David CHEMOUIL	ONERA	Membre du jury

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture
Unité de Recherche :
Office national d'études et de recherches aérospatiales (ONERA)
Directeur(s) de Thèse :
David CHEMOUIL et *Julien BRUNEL*
Rapporteurs :
Régine LALEAU et *Dominique MERY*

Dédicaces

Á la mémoire de mon regretté papa TAWA Joseph Bonaparte : ce que je suis, je le tiens de toi.
Á papa BOPDA Athanase : ton amour a su combler le vide que peut créer l'absence d'un père.
Á Mes chères et braves mamans MAGNE Jacqueline et KAKE Henriette : vous êtes ces mains cachées qui ont fait de moi, cette femme que je suis, vous l'avez fait au prix d'énormes sacrifices.
Á Ma famille : votre amour est un repère et un refuge pour moi, et votre soutien, ma véritable source de courage et de réussite.
Á Mes amis : vous êtes pour moi une famille d'un autre continent, d'un autre pays, des frères et sœurs d'une autre mère. Votre soutien morale et physique inconditionnel me donne la force et le courage d'avancer.

Remerciements

Tout d'abord, je tiens à remercier mes encadrants de thèse, David Chemouil et Julien Brunel, pour leur encadrement bienveillant durant ces quatre années de thèse. Ils m'ont offert, chacun à leur manière, un soutien tant sur le plan technique qu'humain qui m'a permis d'avancer dans ce travail d'apprentissage par la recherche.
J'adresse tous mes remerciements à Régine Laleau, ainsi qu'à Dominique Mery de l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de cette thèse. Leurs analyses attentives et leurs remarques constructives, ont contribué à la qualité de manuscrit.
J'exprime ma gratitude à Sylvain Conchon et Virgie Weils , qui ont bien voulu être examinateurs.

Je tiens à remercier Kevin Dermas, Rémy Dermas , Thomas Polaseck pour leur gentillesse, leur disponibilité permanente et pour les nombreux encouragements.

Un grand merci à Claire et Lyne pour vos accompagnement dans mes diverses démarches administratives Un grand merci aux doctorants et anciens doctorants du DTI S avec qui nous avons passé de bons moments autour de discussions enflammées lors des repas ou des frameworks. En particulier à Guillaume Davy et Matthieu Maunoury pour leurs marques d'attention particulières et à Hugo Daigmorte pour son sens de l'humour très singulier. Merci à Quentin Pérras pour ta gentillesse, ton soutient et surtout pour tous ces débats politiques de néophytes.

Je remercie la grande famille Pola Anne et la grande famille Tamo Kamwa, en particulier ma maman, mes frères et mes sœurs de m'avoir encouragé et aidé, pas seulement dans les dernières années d'études qu'on été ce doctorat, mais aussi pour toutes les années avant, depuis mon départ du Cameroun.

Je tiens aussi à remercier tous mes amis de la grande famille IAI-Gabon de la France et d'ailleurs. Vous mes aînés vous êtes des modèles pour moi, et à vous mes contemporains merci de m'accepter tel que je suis.

Un grand merci à tous les membres de l'assemblé HOHFAN SILOE de Toulouse en particulier notre responsable Micheal et les chantres avec qui nous passons de véritables moments de communion fraternelle les mardi et vendredi soir. Un grand merci au membre de l'assemblé ICC de Toulouse, ainsi qu'aux amies et aînés du mouvement AFRICATHO. En particulier Marcus, Gabriel, Nadège Prisca, Maillys.

Un grand merci à mes amis Lylia Sypile, Noëlla Obone, Joëlle Sihon, Asma Toumi, Stéphane Nindjin, Yannick Boumba, Jérémie Loubaky, Ninon Ikonga, Danielle Fokou. Merci à vous tous qui de prêt ou de loin avez participé à la réussite de ce projet de thèse.

Bien des pensées pour vous tous, tout ce qui allongerait trop l'écriture de cette page ou n'y aurait pas sa place. La liste est bien loin d'être exhaustive : les personnes qui comptent ou ont compte pendant ces annees se connaissent et se reconnaissent pour ça, je l'espère.

Résumé

Cette étude concerne l'utilisation de la logique du premier ordre et la logique temporelle linéaire pour la spécification et la vérification des systèmes dynamiques ayant des structures riches. Elle concerne l'étude de la correction de fonctionnement du protocole de recherche distribuée Chord. L'objectif est d'une part d'améliorer la spécification et la vérification des systèmes dans le langage formel Electrum qui est une extension dynamique du langage formel Alloy basé sur la logique du premier ordre temporelle linéaire. Pour ce faire, nous avons développé une couche syntaxique au-dessus d'Electrum. L'objectif de cette couche est de faciliter la spécification du comportement dans Electrum, pour ce faire, nous avons défini une syntaxe permettant de générer automatiquement une partie de la spécification du comportement. Par ailleurs cette couche permet également de réduire les erreurs de spécification en déchargeant les utilisateurs de la spécification de certaines tâches comportementales ardues et sujettes aux erreurs.

D'autre part, l'objectif est d'analyser formellement la correction de la propriété fondamentale de vivacité du protocole Chord. Pour ce faire nous avons spécifié et vérifié le protocole Chord avec l'extension d'Electrum que nous avons développée, puis nous avons prouvé sa correction et montré les avantages de notre méthode d'analyse.

Mots clés : Logique du premier ordre, Logique Temporelle Linéaire, Méthodes Formelles, Spécification et Vérification Formelle, Model-Checking, Electrum, réseau peer-to-peer, protocole de recherche distribuée Chord.

Abstract

This study deals with the use of first-order logic and linear temporal logic for specification and verification of dynamics system with rich structural properties. It also concerns the study of the operating safety of the distributed lookup protocol Chord. The aim is on the one hand to improve the specification and verification of systems whith the formal language Electrum a dynamic extension of the formal language Alloy based on first-order linear temporal logic. To do this, we have developed an action layer above Electrum. The purpose of this layer is to make the specification of the behavior in Electrum easier, to do this, we have defined a syntax to automatically generate part of the specification of the behavior. Moreover, this layer also makes it possible to reduce specification errors by getting rid the users of the specification of certain laborious and error-prone behavioral tasks. On the other hand, the aim is the formal analyzing of the fundamental liveness property of the Chord protocol. To do this, we specified and verified the Chord protocol with the extension of Electrum that we have developed, then we proved its correctness and showed the advantages of our analysis method.

Keywords : Fisrt-Order Logic, Linear Temporal Logic, Formals Methods, Formal Specification and Verification, Model-Checking, Electrum, peer-to-peer Network, Distributed Lookup Protocol, Protocol Chord.

Table des matières

Abréviations **15**

1 Introduction **17**
1.1 Contexte . . . 17
1.1.1 Les méthodes formelles . . . 17
1.1.2 Protocole de recherche distribuées . . . 20
1.2 Problématique et objectifs . . . 21
1.3 Organisation du manuscrit . . . 22

I État de l'art **23**

2 Les logiques **25**
2.1 Logique propositionnelle . . . 25
2.1.1 Syntaxe . . . 26
2.1.2 Sémantique . . . 26
2.1.3 Satisfiabilité . . . 27
2.2 La logique du premier ordre . . . 27
2.2.1 Syntaxe de la logique du premier ordre . . . 27
2.2.2 Sémantique . . . 29
2.3 Logique du premier ordre relationnelle . . . 31
2.3.1 Syntaxe . . . 31
2.3.2 Sémantique . . . 32
2.4 Logique Temporelle Linéaire (LTL) . . . 33
2.4.1 Syntaxe de LTL . . . 33
2.4.2 Sémantique LTL . . . 34
2.5 Logique temporelle linéaire avec passé PLTL . . . 35
2.5.1 Syntaxe de PLTL . . . 35
2.5.2 Sémantique PLTL . . . 35
2.6 La logique du premier ordre temporelle linéaire FOLTL . . . 36
2.6.1 Syntaxe . . . 36
2.6.2 Sémantique . . . 36
2.7 Conclusion . . . 37

3 Les langages Alloy, Dynalloy et TLA+ 39
3.1 Le langage Alloy et Alloy Analyzer . . . 39
3.1.1 Atomes et relations . . . 40
3.1.2 Signatures et champs . . . 41
3.1.3 Formules . . . 44
3.1.4 Prédicats et fonctions . . . 44
3.1.5 Assertion . . . 45
3.1.6 Modélisation du comportement dans Alloy . . . 45
3.1.7 Commandes et scope . . . 49
3.1.8 Vérification . . . 50
3.1.9 L'outil Alloy Analyzer . . . 51
3.2 DynAlloy : Une extension dynamique de Alloy . . . 53
3.2.1 Modélisation de la structure dans DynAlloy . . . 55
3.2.2 Modélisation du comportement dans DynAlloy . . . 55
3.2.3 Assertion des propriétés dynamiques . . . 58
3.2.4 Commandes et scope . . . 59
3.2.5 Syntaxe abstraite et Sémantique de DynAlloy . . . 59
3.2.6 Vérification dans DynAlloy . . . 60
3.3 TLA+ . . . 62
3.3.1 Modélisation de la structure dans TLA+ . . . 62
3.3.2 Terminologie . . . 64
3.3.3 Modélisation du comportement dans TLA+ . . . 65
3.3.4 Spécification des propriétés dynamiques . . . 66
3.3.5 Vérification avec TLC . . . 67
3.4 Conclusion . . . 67

4 Electrum 69
4.1 Introduction . . . 69
4.2 Spécification formelle . . . 71
4.3 Vérification Formelle . . . 75
4.3.1 Syntaxe des commandes . . . 75
4.3.2 Techniques de vérification . . . 78
4.3.3 L'outil Electrum Analyzer . . . 82
4.4 Sémantique . . . 82
4.4.1 Noyau d'Electrum . . . 82
4.4.2 Traduction d'Electrum vers Electrum Kernel . . . 83
4.4.3 Sémantique d'Electrum en FOLTL . . . 83
4.5 Conclusion . . . 84

5 Protocole de recherche distribuée Chord 87
5.1 Contexte . . . 87
5.2 Description du protocole Chord . . . 89
5.2.1 Structure des réseaux Chord . . . 90
5.2.2 Algorithme de recherche de données dans les réseaux Chord . . . 91
5.2.3 Les propriétés caractéristiques des réseaux Chord . . . 92
5.2.4 Les opérations dynamiques de Chord . . . 94

5.3 Conclusion 97

II Contributions **99**

6 Extension d'Electrum avec des actions **101**
6.1 Introduction 101
6.2 Syntaxe de la couche action 102
6.2.1 Actions 103
6.2.2 Les conditions du cadre 103
6.2.3 Modèle temporel 105
6.2.4 Occurrence des actions dans les contraintes 105
6.3 Sémantique 106
6.3.1 Structure des actions et modèle temporel 106
6.3.2 Occurrences des actions 109
6.3.3 Les effets d'une action 110
6.3.4 Les conditions du cadre 110
6.4 Expérimentation 111
6.4.1 Cas d'études 111
6.4.2 Modélisation 113
6.4.3 Vérification 114
6.5 Discussion 115

7 Analyse de la propriété de vivacité fondamentale du protocole Chord **117**
7.1 Motivations 117
7.2 Introduction 118
7.3 Formalisation du protocole Chord 119
7.3.1 Structures de données 119
7.3.2 Propriétés du réseau 121
7.4 Les événements du protocole Chord 123
7.4.1 Opération join 123
7.4.2 Opération fail 125
7.4.3 Opération de stabilisation 125
7.5 Le modèle temporel 130
7.6 La propriété de correction du protocole Chord 130
7.7 L'analyse et l'évaluation des résultats 134
7.8 Discussions 137
7.9 Autres travaux de vérification des protocoles peer-to-peer 138

8 Conclusion **141**
8.1 Synthèse des travaux 141
8.1.1 Electrum Action 141
8.1.2 Protocole de recherche distribuée Chord 141
8.2 Limitation des solutions proposées 142
8.3 Travaux futurs 142

A Modèle complet du protocole Chord **145**

B Modélisation des systèmes classiques d'Alloy dans Electrum ordinaire et Electrum avec action 151
B.1 Modèles de l'élection du leader 151
B.1.1 Modèle Electrum Ordinaire 151
B.1.2 Modèle Electrum Avec Action 152
B.2 Modèles de spantree 154
B.2.1 Modèle Electrum Ordinaire 154
B.2.2 Modèle Electrum Avec Action 156
B.3 Modèles de Firewire 158
B.3.1 Modèle Electrum Ordinaire 158
B.3.2 Modèle Electrum Avec Action 164

Liste des tableaux

7.1 vérification de la cohérence et des différentes branches des actions. 136
7.2 Temps(s.) d'analyse de la correction (bugué et correcte). 136
7.3 Tableau comparatif des travaux d'analyse sur Chord et d'autres protocoles peer-to-peer. 139

Table des figures

3.1 Modèle Alloy de l'Hôtel . . . 44
3.2 Modèle Alloy de l'Hôtel selon l'idiome Event idiome. . . . 47
3.3 Syntaxe concrète du language Alloy . . . 51
3.4 Éditeur des spécifications d'Alloy Analyzer . . . 52
3.5 Visualiseur des instances et contre-exemples d'Alloy Analyzer . . . 52
3.6 Modèle DynAlloy du système Hôtel, extrait du modèle du site de DynAlloy [PGAF] 55
3.7 Modèle TLA+ de l'Hôtel, issue de [MC16] . . . 63

4.1 Example du modèle Alloy de l'hôtel traduit en un modèle Electrum . . . 74
4.2 État initial . . . 76
4.3 L'hôte `Guest1` est enregistré à la réception dans la chambre `Room1` . . . 76
4.4 L'hôte `Guest1` quitte l'hôtel avec sa clé . . . 76
4.5 L'hôte `Guest0` est enregistré à la réception dans la chambre `Room1` . . . 77
4.6 L'hôte `Guest1` peut entrer dans la chambre `Room1` . . . 77
4.7 Syntaxe concrète du langage Electrum, les ajouts à la syntaxe Alloy sont soulignées . 79
4.8 Architecture de Electrum Analyzer . . . 82
4.9 Electrum Kernel Syntaxe abstraite d'Electrum. . . . 83
4.10 Traduction d'Electrum Kernel vers FOLTL (cf.. Def. Chap : 2 section 2.6). . . . 85

5.1 Exemple de réseau Ideal . . . 91
5.2 Exemple de réseau Valid . . . 91
5.3 Structure d'un réseau Chord loopy. Les traits en pointillés représentent la *relation suivant* autour de l'anneau et les traits pleins *la relation successeur*. . . . 92
5.4 Réseau initialement Ideal . . . 95
5.5 Le nœud 4 est dans le réseau. . . . 95
5.6 Le nœud 4 se stabilise et notifie le nœud 6 . . . 97
5.7 Le nœud 6 rectifie en réponse à la notification du nœud 4. . . . 97
5.8 Le nœud 1 se stabilise et notifie le nœud 4 qui rectify dans la foulée. . . . 97

6.1 Modèle Electrum standard de l'Hôtel . . . 102
6.2 Exemples d'actions . . . 103
6.3 Modèle Electrum action de l'Hôtel . . . 104
6.4 Expression de l'effet d'une action . . . 110
6.5 Comparaison en terme de nombre de lignes du modèle . . . 114

6.6 Résumé des tests effectués (20 Time pour les scénarios bornés) ; C désigne la "Cohérence", S la "sécurité" et L la "vivacité". L'unité du temps est la seconde (time-out : 10 m) ; UMC-d est le UMC en mode décomposition(avec au maximum 36 threads).. 116

7.1 Exemple d'un réseau Valid (gauche) et Ideal (droit) 123
7.2 Une occurrence de l'action `join` 124
7.3 Une occurrence de l'action fail 126
7.4 Etape de la séquence de correction du réseau 127
7.5 Une Occurrence de `stabilizeFromFst` programme une action `stabilizeFromFstPrdc` . 129
7.6 Occurence de `rectify` programmée par un `stabilizeFromFstPrdc` 130
7.7 Contre-exemple manifesté par l'exécution infinie de l'action `skip` 131
7.8 Les quatres premiers étapes du contre-exemple. 134
7.9 Sequence responsable du contre-exemple 135

Abréviations

FO : First-Order Logic
FOR : Relational First-Order Logic
LTL : Linear Temporal Logic
PTL : Linear Temporal Logic With PasT
FOLTL : First-Order Linear Temporal Logic
SMV : Symbolic Model Verification
BMC : Bounded Model-Checking
UMC : Unbounded Model-Checking
TLA : Temporal Logic of Actions
SAT : Satisfiable
UNSAT : Unsatisfiable

Chapitre 1

Introduction

1.1 Contexte

À la fin des années soixante éclatait "**la crise des logiciels**" dont quelques exemples historiques tristement célèbres sont : le crash de la fusée *Ariane 5 en 1996* [Lio96], *les anti-missiles Patriote (Guerre du Golfe 1991), les sondes perdues, vers Mars (années 99), vers Venus (années 60)*, *la grande panne du réseau téléphonique américain ATT en 1990.* Cette crise a mis en évidence la nécessité de pouvoir contrôler le processus de construction des systèmes informatiques qu'ils soient matériels ou logiciels. C'est pourquoi de nos jours, le défi majeur en informatique est de pouvoir construire des systèmes de qualité en respectant un certain nombre de contraintes de temps et de budget. La qualité des logiciels est dégradée par la présence de défaillances ou *"bugs"* durant leur fonctionnement. Ces défaillances sont généralement causées par des erreurs que le concepteur n'a pas pu déceler et traiter lors de la phase de conception, malgré des efforts de relecture.

Plusieurs méthodes de détection de *erreurs* ont été proposées. La méthode traditionnelle est le test intensif. Un test consiste à exécuter des scénarios (séquences arbitraires d'actions) en surveillant que le système se comporte comme il se doit. Effectuer un grand nombre de tels tests permettrait de réduit les cas d'erreur, néanmoins il n'est pas possible d'exécuter tous les scénarios, ainsi la plupart du temps le système peut toujours contenir des erreurs à l'issue de cette procédure.

Toutefois, l'informatique occupe une part croissante dans les domaines sensibles tels que l'aviation, la médecine, les télécommunications le nucléaire ce qui impose une confiance absolue en l'absence de toutes sortes de défaillances dans certains logiciels utilisés dans ces domaines. L'une des meilleures approches proposées pour garantir cette confiance sont les méthodes de vérification formelles.

1.1.1 Les méthodes formelles

L'objectif des méthodes formelles est de raisonner rigoureusement sur des systèmes dans le but de démontrer leur correction par rapport à un ensemble d'exigences appelées spécifications. Elles sont basées sur les logiques qui permettent de spécifier et de vérifier des exigences sur des systèmes.

En effet, les logiques répondent aux besoins de rigueur et de précision dans la description du comportement de systèmes, elles offrent également des mécanismes d'abstraction pour simplifier et unifier le raisonnement sur ces systèmes.

Par ailleurs, contrairement aux méthodes traditionnelles de tests, les méthodes formelles permettent de traiter de manière exhaustive tous les scénarios d'exécution possibles d'un système dans un domaine spécifique.

Pour toutes ces raisons elles offrent une très forte assurance sur l'absence de bugs dans un logiciel.

On distingue plusieurs catégories de méthodes formelles :

1. le typage dans la programmation dont le principe est de restreindre des langages de programmation à des classes de programmes dont le comportement statique est assuré. Le typage réduit le pouvoir d'expressivité des langages, mais les permet de produire des programmes sur lesquels il est facile de raisonner et dont d'éliminer les exécutions erronées ;
2. le model checking qui consiste à analyser exhaustivement toutes les évolutions possibles du système afin de démontrer l'absence d'erreurs dans tout ses états accessibles. Il est réalisé par le biais d'algorithmes intelligents permettant l'énumération de l'ensemble des états du système sous une forme symbolique économique ;
3. la vérification déductive qui consiste à considérer un système uniquement d'un point de vue purement logique et sémantique et de raisonner dessus à l'aide de démonstration de théorèmes de manière systématique avec des règles d'inférence d'une logique bien définie ;
4. l'interprétation abstraite.

Toutefois, l'utilisation des méthodes formelles requiert une bonne connaissance en mathématique, ce qui limite leur utilisation ainsi que leur expansion dans le monde de l'industrie.

Néanmoins, contrairement aux autres méthodes, les techniques de model-checking sont beaucoup plus répandues, car ils sont plus facilement automatisables et donc utilisables par des personnes n'ayant pas une grande connaissance en mathématique.

Méthodes de vérification formelle

Les méthodes de vérification formelle sont constituées de trois composants [HR04]

1. Le langage de description qui permet de construire une abstraction ou un modèle du système réel ;
2. Le langage de spécification permettant de construire une formule représentant les propriétés à vérifier ;
3. La méthode de vérification qui permet de vérifier si le modèle (l'abstraction) satisfait la formule.

Ces techniques se distinguent d'une part, par la simplicité, la flexibilité et le pouvoir expressif du langage de spécification, c'est-à-dire l'ensemble des propriétés pouvant être exprimées dans ce langage. Cette expressivité définit la classe des systèmes ainsi que les types de propriétés pouvant être vérifiés avec la méthode. Et d'autre part, par le degré d'automatisation de la méthode de vérification.

Par ailleurs, il est très difficile de conjuguer l'expressivité et l'automatisation. Ainsi, lorsqu'on conçoit une méthode formelle une question importante se pose : faut-il privilégier le pouvoir expressif du langage de spécification ou le degré d'automatisation de la méthode de vérification ?

Méthodes formelles légères

Historiquement, les méthodes formelles ont été introduites comme de simples alternatives aux méthodes de développement traditionnelles. De cette façon leur utilisation dans le domaine de l'industrie exigeait d'énormes changements dans le processus de développement déjà existant dans l'entreprise.

Des approches plus pragmatiques et légères visant à utiliser les méthodes formelles pour compléter voire améliorer le processus de développement, ont été proposées.

La principale tendance actuelle dans les méthodes formelles appliquées est l'approche dite de "*modélisation formelle légère*" [Jac03], [Jac06],[Jac12].

La modélisation formelle légère consiste à construire un petit modèle formel abstrait des concepts clés d'un système logiciel, puis à l'analyser à l'aide d'outils entièrement automatisés («push-button») fonctionnant par énumération exhaustive sur un domaine de possibilités borné.

La raison fondamentale pour laquelle les méthodes formelles légères ont eu du succès est qu'elles ne favorisent ni l'expressivité ni le degré d'automatisation.

En effet, elles fournissent des langages de spécification formels simples et flexibles permettant aux utilisateurs de spécifier des classes de systèmes de complexités différentes à différent niveau d'abstraction. Ces langages sont accompagnés par des analyseurs (model checker) complètement automatisés qui fournissent un retour rapide sur la correction des spécifications.

Alloy [Jac03] est l'une des principales méthodes formelles légères, développée par Daniel Jackson. Son langage de spécification basé sur une extension de la logique du premier ordre (FOR) offre d'une part un niveau d'abstraction assez élevé permettant la modélisation de systèmes très complexes et d'autre part, un moyen très adapté pour la spécification des propriétés structurelles (relations entre composants d'un système, hiérarchie) sur les systèmes.

Il est accompagné d'Alloy Analyzer un outil complètement automatisé qui réalise la vérification des propriétés statiques sur les systèmes.

Il est facile d'utilisation en ce sens que sa syntaxe est très proche de celles des langages orientés objet qui sont connus et appréciés des concepteurs. De plus, Alloy Analyzer offre une interface graphique très conviviale permettant aux utilisateurs de spécifier et d'analyser aisément les modèles représentant les systèmes.

Cependant, le pouvoir expressif d'Alloy ne permet pas une spécification simple et directe des propriétés comportementales (ordre d'exécution des événements, évolution des paramètres variables). De plus, Alloy Analyzer contraint la vérification à s'appliquer sur des horizons de temps bornés.

Plusieurs travaux sur l'introduction du dynamisme dans Alloy ont été effectués, dont l'un des plus célèbres est le langage DynAlloy. Cependant, ce langage impose des schémas de syntaxe pour la spécification du comportement, détériorant ainsi la flexibilité à laquelle sont habitués les utilisateurs d'Alloy. D'autre part, ce langage ne permet pas de spécifier et de vérifier toutes les classes de propriété dynamiques possibles, en l'occurrence les propriétés dites de vivacité ne sont pas modélisable dans DynAlloy.

Le langage SMV basé sur la logique temporelle linéaire (LTL) offre un moyen très adapté pour la spécification des propriétés temporelles. Les model checkers NuSMV [CCGR00],[CCG+02] et Nuxmv [CCD+14] réalisent la vérification des spécification SMV des propriétés temporelles des systèmes sur des horizons de temps tant bornés que non bornés. Néanmoins, le langage SMV n'est

pas adapté pour la spécification des propriétés structurelles (Exigences sur la structure d'un système comme par exemple : les relations entre les différents composants d'un système).

La méthode formelle Electrum [MBC+16, Alca] est une extension d'Alloy avec des aspects dynamiques. Son langage de spécification basé sur la logique du premier ordre temporelle linéaire (FOLTL) permet de spécifier à la fois les propriétés comportementales et les propriétés structurelles sur des systèmes, avec la même flexibilité que le langage Alloy. Il est accompagné d'Electrum Analyzer qui autorise la vérification des propriétés des systèmes sur des horizons de temps tant bornés que non bornés. L'analyse bornée est réalisée par une traduction directe vers Alloy, tandis que l'analyse non bornée consiste en une compilation du modèle Electrum vers un modèle SMV, puis d'une vérification par les model checkers NuSMV et nuXmv.

Un premier prototype d'Electrum a été développé. Son utilisation s'est avérée très avantageuse pour la modélisation et la vérification de systèmes dynamiques dotés des propriétés structurelles riches. Néanmoins, pour plus d'efficacité, des améliorations du point de vue de la performance ou de la facilité de modélisation sont nécessaires.

1.1.2 Protocole de recherche distribuées

Au cours des dernières décennies, l'utilisation d'Internet a connu une grande évolution grâce aux concepts *de réseau peer-to-peer*. Les réseaux *peer-to-peer* sont des systèmes distribués sans organisation hiérarchique ni contrôle centralisé. Contrairement aux systèmes traditionnels, client / serveur, ces systèmes présentent une faible barrière au déploiement, car ils ne requièrent aucun investissement supplémentaire dans du matériel haute performance. Ils offrent une grande évolutivité des nœuds participants et une robustesse face à certains types d'erreurs et d'attaques. Ces caractéristiques ont conduit à une large application des systèmes peer-to-peer dans les technologies actuelles de l'information et de la communication.

Cependant, le problème fondamental que rencontrent les systèmes peer-to-peer est la localisation efficace des données stockées dans le réseau. En effet, le mouvement (allées-venues) des nœuds dans le réseau entraîne une migration de données, rendant ainsi la localisation des données assez complexe. Pour résoudre ce problème, les protocoles de recherches distribués ont été conçus, dont les plus performants implémentent une table de hachage distribuée (*distributed hash table* ou DHT en anglais) *cf* chapitre 5, qui fournit des fonctionnalités indispensables au fonctionnement des systèmes peer-to-peer.

Propriété de correction des protocole peer-to-peer

La localisation correcte et efficace des données dans le réseau suppose que le protocole de recherche fonctionne comme il se doit. Le fonctionnement correct des protocoles de recherches distribuées est caractérisé par une exigence appelée *"propriété de correction du protocole"*.

Ces propriétés sont généralement conçues par des raisonnements informels supposant à tort que les systèmes complexes fonctionneraient exactement comme l'humain le pense. Or, les systèmes peer-to-peer ont tendance à ne pas fonctionner comme on imagine [Zav15b], il y a un écart entre l'intuition de l'humain et la réalité, ce qui induit une certaine probabilité non-négligeable de la présence d'erreurs (subtiles) dans la définition de la propriété de correction.

Avec l'expansion des technologies matérielles et logicielles, d'Internet, ainsi que la taille, les fonctionnalités et l'application omniprésente des systèmes peer-to-peer, il dévient presque urgent de minimiser cette probabilité pour garantir une sûreté aux applications peer-to-peer.

Une façon de s'assurer de la correction de ces protocoles est d'utiliser les méthodes formelles pour spécifier et vérifier leur conception. En effet, les méthodes formelles améliorent considérablement la compréhension de ces systèmes en relevant des incohérences, des ambiguïtés et des omissions n'ayant pas été capturées lors de la description des exigences.

1.2 Problématique et objectifs

En pratique, dans Electrum le comportement d'un système est naturellement spécifié par des actions élémentaires qui font référence à deux instants de temps consécutifs, ce qui nécessite que très peu de pouvoir expressif du langage Electrum.

Cependant, lors de la spécification du comportement dans Electrum, certaines tâches fastidieuses et parfois sujettes aux erreurs telles que la spécification des contraintes sur l'ordre dans lequel les actions sont autorisées à s'exécuter, ou encore la spécification des exigences sur l'évolution des éléments variables du système restent à la charge de l'utilisateur. Une solution à ce problème pourrait être l'introduction dans Electrum d'une syntaxe et d'une sémantique propres aux actions, permettant une génération systématique de telles contraintes, ce qui déchargerait l'utilisateur de ces tâches fastidieuse, et par conséquent simplifierait la modélisation du comportement. Par ailleurs, distinguer explicitement les actions dans les modèles des autres contraintes peut être très avantageux pour une procédure de vérification qui s'appuie sur un model checker.

En effet, le système de transition d'un modèle (et par ricochet son automate) est définit à partir des actions qui font évoluer le système. Or, dans Electrum les actions sont définies par de simples prédicats, on ne peut donc pas les distinguer d'autres prédicats quelconques spécifiant des contraintes sur le système.

Il s'en suit que, lorsqu'on traduit un modèle Electrum, le modèle SMV résultant ne possède pas de système de transition (l'automate représentant le système est maximal). En outre, la formule LTL à vérifier est très grande, car constituée de la spécification de la propriété initiale et de la spécification des actions. Ceci détériore significativement les performances de la vérification.

La distinction des actions des autres prédicats permettrait la construction d'un véritable système de transition (et donc d'un automate plus petit) basé sur les actions, et la formule à vérifier serait simplifiée. Dans cette étude nous proposons une extension d'Electrum à l'aide d'une couche action qui répond à ces besoins.

Le protocole de recherche distribué Chord [SMK+01],[SMLN+03], [LNBK02] basé sur DHT est l'une des meilleures solutions au problème de localisation de données dans un réseau peer-to-peer utilisé dans les applications de partage de fichiers lourds (audios et vidéos) sur Internet, à l'instar de BiTorrent. Le fonctionnement exact de Chord est caractérisé par une propriété de vivacité. Selon ses auteurs, il est correct et sa correction est facilement prouvable. Cependant, aucune preuve formelle de cette correction n'a été proposée par les auteurs, ce qui a suscité des questionnements sur sa validité. C'est dans ce sens que des travaux de spécification et de vérification formelle sur la validité de la correction du protocole Chord ont été réalisés.

L'essentiel de ces travaux ont consisté en :

- une analyse complètement automatisée d'une propriété de sûreté sur Chord [Zav19, Zav11, Zav12, Zav15b]. Il s'agit principalement de recherche et de vérifier avec Alloy un invariant

inductif, utile pour prouver la correction. Cette recherche a été très ardue et au bout du compte, l'invariant trouvé n'a pas été suffisant pour prouver la correction de Chord, qui est une propriété de vivacité. Néanmoins, il a mis en évidence la présence des incohérences et des ambiguïtés dans la description du protocole.

- une analyse de la correction de Chord en tant que propriété de vivacité avec Spin [Zav15a]. La limitation majeure de cette analyse est que le langage de spécification Promela utilisé n'est pas adapté pour la spécification des propriétés structurelles qui caractérisent la topologie d'un réseau Chord. Par conséquent, divers programmes C ont été nécessaires pour la gestion de la structure du réseau ainsi que la visualisation des résultats des analyses.
- une analyse manuelle de la correction de Chord comme propriété de vivacité [Zav17]. Cette analyse présente de petits bugs et des omissions.

En somme, l'analyse avec Alloy fournit un invariant qui ne suffit pas à prouver la correction, l'analyse avec Spin requiert l'utilisation de programmes supplémentaires pour la gestion de la structure et enfin l'analyse manuelle présente des erreurs et omissions, d'où l'objet de notre étude. Nous nous intéressons à la spécification et la vérification formelle complètement automatisée de la propriété fondamentale de vivacité du protocole Chord, avec la méthode formelle légère Electrum qui en effet, est adapté pour la spécification et la vérification des propriétés tant structurelles que comportementales.

1.3 Organisation du manuscrit

Ce mémoire commence par un aperçu de l' état de l'art dans le domaine des méthodes formelles. On y approfondit les logiques (chapitre 2), en vue de définir des langages de spécification et de vérification formels.

Cette présentation est prolongée par une introduction des méthodes formelles définies sur ces logiques ou sur des variantes (chapitre 3). Il s'agit plus précisément d'Alloy, de son extension dynamique DynAlloy et de TLA+. Un bilan critique des atouts et des limitations de ces méthodes formelles y est également dressé. Nous poursuivons cet état de l'art avec une exposée sur la méthode formelle Electrum (chapitre 4) dont nous voulons développer une extension. Nous clôturons cette partie avec la présentation du protocole de recherche distribué Chord (chapitre 5) dont l'analyse de la propriété de correction fait partie des contributions de cette thèse.

Notre apport consiste d'une part à développer une extension d'Electrum avec une couche action permettant d'améliorer la spécification du comportement dans Electrum (chapitre 6). Ces travaux ont été présentés dans [BCT18]. D'autre part notre contribution consiste à analyser avec Electrum muni de cette couche action, le protocole Chord (chapitre 7). Ces autres travaux ont été présentés dans [BCT18].

Dans ce mémoire nous illustrons également l'ensemble des concepts présentés à travers des cas d'étude proposés par Daniel Jackson dans [Jac03]. Nous avons fait le choix de présenter de manière exhaustive les différents modèles de ces systèmes dans l'annexe B.

Première partie

État de l'art

Chapitre 2

Les logiques

En informatique, la logique est utilisée pour le développement des langages de modélisation des systèmes réels, de sorte qu'un raisonnement formel sur les modèles obtenus soit possible. Le raisonnement consiste en une spécification et une vérification formelles rigoureuses des propriétés sur les systèmes.

Les propriétés des systèmes sont définies informellement par des phrases déclaratives du langage naturel. La logique est donc une base solide sur laquelle peuvent reposer les langages de spécification et de vérification formels permettant de raisonner sur ces propriétés. Par ailleurs, les propriétés peuvent décrire soit la structure des systèmes (ex : relations ou hiérarchies entre les composantes du système), ou leur comportement (ex : séquencement des événements du système, contraintes sur l'évolution des éléments variables). Selon la classe de propriétés ciblées, les langages de spécification et de vérification formels se baseront sur une logique adéquate.

Le reste du chapitre est présenté de la façon suivante. Dans la section 2.1, nous donnons une présentation de la logique propositionnelle. Dans la section 2.2, nous présentons la logique du premier ordre (FO) puis nous poursuivons dans la section 2.3 avec la logique relationnelle du premier ordre (FOR). Ces logiques sont adaptées pour les langages de spécification et de vérification formelles des propriétés structurelles et relationnelles sur les systèmes.

Nous exposons dans la section 2.4 la logique temporelle linéaire qui est utilisée pour les langages de spécification et de vérification formelles des propriétés dynamiques des systèmes. Nous consacrons la dernière section 2.6 à la logique du premier ordre temporelle linéaire (FOLTL) qui est utilisée pour les langages de spécification et de vérification formelles des systèmes ayant à la fois des propriétés dynamiques et des propriétés structurelles.

2.1 Logique propositionnelle

La logique propositionnelle permet d'exprimer des énoncés ou propositions auxquels on attribue une valeur de vérité. C'est le formalisme logique le plus simple permettant de spécifier et de vérifier formellement les propriétés des systèmes. Dans la logique propositionnelle les énoncés plus complexes sont exprimés en termes de relations entre les propositions grâce aux connecteurs logiques.

2.1.1 Syntaxe

Définition 2.1.1 (**Alphabet**). *Pour la logique propositionnelle ; l'alphabet est donné par :*

— *Un ensemble* $\mathcal{AP} = \{p, q, r....\}$ *des variables propositionnelles ;*

— *les constantes* $\top$ *et* $\bot$ *;*

— *un ensemble de connecteurs logiques : (*$\wedge$ *(et) ,*$\vee$ *(ou),*$\neg$ *(non) ,* $\Rightarrow$ *(implication) ,*$\Leftrightarrow$ *(équivalence))*

Définition 2.1.2 (**Formules**). *L'ensemble des formules propositionnelles est le plus petit ensemble* Γ *défini inductivement comme suit :*

- *les constantes* $\top \in \Gamma$ *et* $\bot \in \Gamma$
- *Les variables propositionnelles (p, q, r, . . .,)* $\in \Gamma$ *;*
- *si* $\phi \in \Gamma$ *alors ,* $\neg\phi \in \Gamma$ *;*

  si $\phi \in \Gamma$ *et* $\psi \in \Gamma$ *alors :*
- $(\phi \wedge \psi) \in \Gamma$ *;*
- $(\phi \vee \psi) \in \Gamma$ *;*
- $(\phi \Rightarrow \psi) \in \Gamma$ *;*
- $(\phi \Leftrightarrow \psi) \in \Gamma$*.*

Définition 2.1.3 (**Syntaxe minimale**). *La syntaxe ci-dessus est très large, on peut obtenir une syntaxe minimale grâce aux équivalences entre connecteurs. L'ensemble(* $\wedge, \neg, \top, \bot$*) constitue une syntaxe minimale si on considère les équivalences suivantes :*

- $(\phi \vee \psi) \equiv \neg(\neg\phi \wedge \neg\psi)$ *;*
- $(\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$ *;*
- $(\phi \Leftrightarrow \psi) \equiv (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$*.*

2.1.2 Sémantique

La syntaxe définit les règles pour construire des formules propositionnelles correctes, mais elle n'attribue aucun sens à celles-ci. La sémantique de la logique propositionnelle consiste à attribuer une valeur à chaque formule dans un de ces contextes possibles.

Définition 2.1.4 (**Interprétation**). *Une interprétation* $\mathcal{M}$ *est un sous-ensemble de* $\mathcal{AP}$ *représentant l'ensemble des propositions atomiques considérées comme vraies.*

La fonction d'interprétation attribue des valeurs aux propositions atomiques, ce qui n'est pas suffisant pour interpréter les formules. En effet, pour définir la sémantique d'une formule, il faut donner un sens aux connecteurs logiques qui constituent cette formule. Cela se fait au travers d'une relation de satisfaction entre une interprétation et l'ensemble des formules évaluées comme vrai par celle-ci.

Définition 2.1.5 (**Satisfaction des formules**). *La relation de satisfaction* $\models$ *(dont la négation se note* $\not\models$*) entre une interprétation* $\mathcal{M}$ *et une formule est définie inductivement comme suit :*

— $\mathcal{M} \models \top$ *;*

— $\mathcal{M} \nvDash \bot$;
— $\mathcal{M} \vDash P \in \mathcal{AP}$ *ssi* $P \in \mathcal{M}$;
— $\mathcal{M} \vDash \neg\varphi$ *ssi* $\mathcal{M} \nvDash \varphi$;
— $\mathcal{M} \vDash \varphi \wedge \psi$ *ssi* $\mathcal{M} \vDash \varphi$ *et* $\mathcal{M} \vDash \psi$.

2.1.3 Satisfiabilité

En pratique beaucoup de problèmes algorithmiques peuvent être ramenés à des problèmes de satisfiabilité de différentes logiques, particulièrement en vérification de systèmes.

Définition 2.1.6 (Problème de Satisfiabilité)**.** *Le problème de satisfiabilité pour la logique propositionnelle (noté SAT) se définit de la manière suivante :*
Étant donnée une formule ϕ *définie sur* $(\mathcal{AP})$*. Existe t-il une interprétation* $\mathcal{M}$ *telle que* $\mathcal{M} \vDash \phi$ *?*

La complexité algorithmique du problème de satisfiabilité est propre à chaque logique.

Théorème 1 (Problème SAT)**.** *Le problème SAT est NP-complet [GJ90].*

La logique propositionnelle traite de manière tout à fait satisfaisante les propriétés contenant les connecteurs (non), (et), (ou) et (implique). Cependant, elle ne permet pas d'exprimer les propriétés ayant des connecteurs tel que (il existe), (pour tout), (parmi) ou encore (seulement) ? Ici, la logique propositionnelle montre des limites claires et le désir d'exprimer des propriétés plus subtiles a conduit à la conception de la logique des prédicats, également appelée logique du premier ordre.

2.2 La logique du premier ordre

La logique du premier ordre est une extension de la logique propositionnelle au moyen de quantificateurs, des symboles de fonctions et de prédicats, ainsi que des variables. Dans ce document, FO désigne la logique du premier ordre dont nous donnons ci-dessous la syntaxe et la sémantique

2.2.1 Syntaxe de la logique du premier ordre

Définition 2.2.1 (**Alphabet**)**.** *Pour la logique premier ordre l'alphabet est un ensemble de symboles défini par :*

— *les constantes booléenne* $(\top, \bot)$ *;*
— *les connecteurs de la logique propositionnel* $(\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg)$ *;*
— *les quantificateurs universel* $\forall$ *et existentiel* $\exists$ *;*
— *les variables spécifiées par un ensemble* $\mathcal{V} = \{x, y, z, ...\}$ *;*
— *les symboles de fonction spécifiés par un ensemble* $\mathcal{F} = \{f, g, h, ...\}$ *sur lequel est définie une fonction* ***arité*** *:* $\mathcal{F} \longrightarrow \mathbb{N}_{\neq 0}$ [1] *, qui renvoie le nombre d'arguments d'une fonction. ;*
— *les symboles de constantes spécifiés par ensemble* $S_{\mathcal{C}} = \{a, b, c, ...\}$
— *les symboles de prédicats spécifiés par l'ensemble* $\mathcal{P} = \{P, Q, R, ...\}$*, le nombre d'arguments d'un prédicat est donné par la fonction* ***arité*** : $\mathcal{P} \longrightarrow \mathbb{N}_{\neq 0}$ *;*

1. L'ensemble des entiers naturels strictement positifs

— *le symbole de prédicat $\doteq$ défini sur l'ensemble des variables.*

La dualité entre les différents connecteurs et quantificateurs permet de construire des minimaux, qui sont des sous-ensembles représentatifs de tous les connecteurs et des quantificateurs. $\{\forall, \neg, \wedge\}$ est un exemple de minimal dont on peut se contenter pour exprimer toutes formules FO. En effet, $\exists y P(y) = \neg(\forall y \neg P(y))$ et $p \vee q = \neg(\neg p \wedge \neg q)$.

Termes et Formules

Il existe deux catégories d'objets dans FO : les termes et les formules. Les termes sont les éléments de base du langage, c'est-à-dire qu'ils ne contiennent aucun opérateur logique.

Définition 2.2.2 (**Termes**). *L'ensemble des termes est le plus petit ensemble τ défini inductivement par [HR04], [CL93] [DNR+ 01] :*

- *Toute variable $x \in \mathcal{V}$, $x \in \tau$;*
- *Toute constante $c \in S_{\mathcal{C}}$, $c \in \tau$;*
- *Pour $t_1, t_2, ..._{,n} \in \tau^n$, et $f \in \mathcal{F}$ un symbole de fonction d'arité n alors $f(t_1, t_2, .., t_n) \in \tau$.*

Les termes n'ont pas de valeurs de vérité, cependant l'application d'un symbole de prédicat sur des termes donne un prédicat dont on peut calculer la valeur de vérité. Ainsi, à partir de cette formule, on peut construire des formules plus complexes grâce aux connecteurs logiques.

Définition 2.2.3 (**Formules**). *L'ensemble des formules FO est le plus petit ensemble Γ défini inductivement sur les termes et les symboles de prédicat comme suit :*

- *Si $P \in \mathcal{P}$ est un symbole de prédicat d'arité $n \geq 1$, et si $t_1, ...t_n \in \tau^n$, alors $P(t_1, ...t_n) \in \Gamma$;*
- *si t_1 et t_2 sont des termes alors $t_1 = t_2 \in \Gamma$*
- *Si $\phi \in \Gamma$, alors $\neg\phi \in \Gamma$;*
- *Si $\phi \in \Gamma$ et $\psi \in \Gamma$, alors $(\phi \wedge \psi) \in \Gamma$, $(\phi \vee \psi) \in \Gamma$ et $(\phi \Rightarrow \psi) \in \Gamma$;*
- *Si $\phi \in \Gamma$ et x une variable, alors $\forall x\phi \in \Gamma$ et $\exists x\phi \in \Gamma$.*

Dans le domaine des méthodes formelles, la notion de formules est capitale, en effet, la spécification d'une propriété d'un système consiste à construire dans une logique donnée une formule représentant la propriété, tandis que la vérification concerne l'étude de la validité de cette formule.

Remarque 1. $P(t_1, ...t_n)$ *est appelée formule atomique.*

L'introduction de variables et de quantificateurs permet d'exprimer les notions d'universalité et d'existence. Intuitivement, $\forall x Q(x)$ est vraie si Q est vrai pour toutes les valeurs possibles de x. L'affectation des valeurs à x dépend de sa position relative aux quantificateurs qui sont dans la formule : selon cette position, on va distinguer les variables libres des variables liées.

Variables libres et liées

Définition 2.2.4 (**Occurrences libres et liées**). *Dans une formule de la forme $\forall x\psi$ ou $\exists x\psi$, **la portée de la liaison pour x** est la formule ψ.*

*Soit ϕ une formule de la logique du premier ordre, une occurrence de x dans ϕ **est libre** si elle n'est pas dans la portée d'une liaison pour x, sinon elle est dite **liée**.*

Définition 2.2.5 (**Variables libres et liées**)**.** *Soit ϕ une formule de la logique du premier ordre. Une variable x est une* ***variable libre*** *de ϕ si et seulement s'il y a une occurrence libre de x dans ϕ. Une variable x est une* ***variable liée*** *de ϕ si et seulement s'il y a une occurrence liée de x dans ϕ.*

Définition 2.2.6 (**Formules closes**)**.** *Une formule qui ne contient pas de variables libres est dite close*

2.2.2 Sémantique

Pour donner un sens aux objets de FO, on a recours à une sémantique. Cette sémantique s'adosse à une structure d'interprétation et à une fonction de valuation des différentes variables. Ces deux éléments permettent dans un environnent donné de calculer la valeur de vérité d'une formule.

Définition 2.2.7 (**Structure d'interprétation**)**.** *Soit $\mathcal{F}$ un ensemble de symboles de fonctions et $\mathcal{P}$ un ensemble de symboles de prédicats, chaque symbole ayant un nombre d'arguments fixé. Une structure d'interprétation $\mathcal{M}$ est définie par l'ensemble des données suivantes ;*

- *Un ensemble non-vide $\mathcal{D}$ de valeurs concrètes appelé structure de base ou domaine ;*
- *Pour chaque symbole de constante $c \in S_{\mathcal{C}}$, un élément $c^{\mathcal{M}} \in D$;*
- *Pour chaque symbole de fonction d'arité $n \in \mathbb{N}_{\neq 0}$, une fonction concrète $f^{\mathcal{M}} : \mathcal{D}^n \longrightarrow \mathcal{D}$ de tuples de $\mathcal{D}$ dans $\mathcal{D}$;*
- *Pour chaque symbole de prédicat d'arité $n \in \mathbb{N}_{\neq 0}$, un sous-ensemble $P^{\mathcal{M}} \subseteq \mathcal{D}^n$ de tuples d'éléments de $\mathcal{D}$.*

La structure $\mathcal{M}$, basée sur les ensembles $(\mathcal{F}, \mathcal{P})$ n'est pas suffisante pour établir la satisfaction d'une formule FO. En effet, lorsqu'une formule contient des expressions quantifiées tel que $\forall x \phi(x)$, la validité de $\phi(x)$ doit être établie pour toutes les valeurs possibles de x. Ainsi, l'interprétation des formules doit être relative à un environnement où les valeurs sont associées aux variables. Cet environnement est défini par la fonction suivante.

Définition 2.2.8 (**Environnement**)**.** *Un environnement est une distribution de valeurs aux variables, plus précisément, c'est une fonction $\rho : \mathcal{V} \longrightarrow \mathcal{D}$, de l'ensemble des variables dans le domaine $\mathcal{D}$ de la structure d'interprétation. Soit $v \in \mathcal{V}$ et $d \in \mathcal{D}$. L'environnement $\rho\,[v \longrightarrow d]$ est définie par :*

$$\rho\,[v \longrightarrow d]\,(y) = \begin{cases} d & si\ v = y \\ \rho(y) & si\ v \neq y \end{cases} \tag{2.1}$$

Avec la définition d'un environnement, l'interprétation des termes ainsi que des formules est donnée par les définitions suivantes.

Définition 2.2.9 (**Interprétation des termes**)**.** *Soit $\mathcal{M}$ une structure d'interprétation de domaine $\mathcal{D}$ sur $(\mathcal{F}, \mathcal{P})$, soient $\rho : \mathcal{V} \longrightarrow \mathcal{D}$ un environnement et t un terme. L'interprétation de t dans $\mathcal{D}$ selon ρ notée $[\![t]\!]^{\mathcal{D}}_{\rho}$ est définie inductivement par :*

- *si $v \in \mathcal{V}, [\![v]\!]^{\mathcal{D}}_{\rho} = \rho(v)$*
- *si $c \in S_{\mathcal{C}}, [\![c]\!]^{\mathcal{D}}_{\rho} = c^{\mathcal{M}} \in \mathcal{D}$*

— *si* $f(t_1, ...t_n) \in \mathcal{F}$, $[\![f(t_1, ...t_n)]\!]_\rho^\mathcal{D} = f^\mathcal{M}[[\![t_1]\!]_\rho^\mathcal{D}, ...[\![t_n]\!]_\rho^\mathcal{D}]$

Les symboles de constantes et de fonctions sont évalués par la fonction de l'interprétation tandis que les variables sont évaluées par la fonction de valuation. Une fois les variables évaluées et les termes interprétés, on peut vérifier la satisfaction des formules.

Définition 2.2.10 (**Satisfaction des formules**). *Soit $\mathcal{M}$ une structure d'interprétation sur $(\mathcal{F}, \mathcal{P})$ et ρ un environnement. La relation de satisfaction $\mathcal{M}, \rho \vDash \phi$ pour toute formule FO ϕ est définie par induction :*

- $\mathcal{M}, \rho \models x \doteq y$ *si* $\rho(x) = \rho(y)$ *avec* $\doteq$ *l'égalité entre deux symboles de variables.*
- $\mathcal{M}, \rho \vDash P(t_1, t_2, ..., t_n)$ *iff* $([\![t_1]\!]^\mathcal{D}, [\![t_2]\!]^\mathcal{D}, ..., [\![t_n]\!]^\mathcal{D}) \in P^{\mathcal{M},\rho}$ *;*
- $\mathcal{M}, \rho \vDash t_1 = t_2$ *iff* $[\![t_1]\!]^\mathcal{D} = [\![t_2]\!]^\mathcal{D}$ *;*
- $\mathcal{M}, \rho \vDash \forall x \phi$ *iff* $\mathcal{M}, \rho \vDash_{\rho[x \longrightarrow a]} \phi$ *pour tout* $a \in \mathcal{D}$ *;*
- $\mathcal{M}, \rho \vDash \exists x \phi$ *iff* $\mathcal{M}, \rho \vDash_{\rho[x \longrightarrow a]} \phi$ *pour au moins un* $a \in \mathcal{D}$ *;*
- $\mathcal{M}, \rho \vDash \neg \phi$ *iff* $\mathcal{M}, \rho \nvDash \phi$ *;*
- $\mathcal{M}, \rho \vDash \phi \wedge \psi$ *iff* $\mathcal{M}, \rho \vDash \phi$ *et* $\mathcal{M}, \rho \vDash \psi$ *;*
- $\mathcal{M}, \rho \vDash \phi \vee \psi$ *iff* $\mathcal{M}, \rho \vDash \phi$ *ou* $\mathcal{M}, \rho \vDash \psi$ *;*
- $\mathcal{M}, \rho \vDash \phi \Rightarrow \psi$ *iff* $\mathcal{M}, \rho \nvDash \phi$ *ou* $\mathcal{M}, \rho \vDash \psi$ *.*

Remarque 2. *Lors de l'interprétation des formules seules les variables libres sont valuées.*

Les formules résultent de la spécification formelle des propriétés des systèmes. Il faut rappeler que le but de la logique n'est pas seulement de spécifier les propriétés, mais aussi et surtout de vérifier s'il existe des modèles qui satisfont ces propriétés.

Sans perte de généralité, on s'intéresse seulement aux formules closes. Étant donné que ces formules ne contiennent pas de variables libres, la fonction de valuation est égale à l'ensemble vide.

Définition 2.2.11 (**Modèle de formule**). *Soient ϕ une formule close et $\mathcal{M}$ une structure d'interprétation de domaine $\mathcal{D}$.*

$\mathcal{M}$ est un modèle de ϕ si $\mathcal{M} \vDash_\emptyset \phi$. On note : $\mathcal{M} \models \phi$.

Définition 2.2.12 (**Validité et satisfiabilité**).

1. *Une formule close ϕ est satisfiable si et seulement s'il existe une structure d'interprétation $\mathcal{M}$ tel que : $\mathcal{M} \models \phi$*
2. *Une formule ϕ est valide si et seulement si pour tout modèle $\mathcal{M}$, $\mathcal{M} \models \phi$, on écrit tout simplement : $\vDash \phi$*

Définition 2.2.13 (**Conséquence logique et cohérence**). *Soient $\Gamma = \{\phi_1, \phi_2, ...\phi_n\}$ un ensemble de formules et ψ une formule FO. On dit que ψ est* ***une conséquence logique*** *de Γ et on note $\Gamma \vDash \psi$ si et seulement si, pour toute structure d'interprétation $\mathcal{M}$, si $\mathcal{M} \vDash \phi$, pour toute formule $\phi \in \Gamma$, alors $\mathcal{M} \vDash \psi$.*

Par ailleurs, Γ est dit ***cohérent ou satisfiable*** *s'il existe un modèle $\mathcal{M}$ tel que $\mathcal{M} \vDash \phi$ pour toute formule $\phi \in \Gamma$*

Dans la modélisation formelle, la cohérence est utilisée pour s'assurer qu'il n'y a pas de contradiction dans la spécification de l'ensemble des éléments du système.

Décidabilité et expressivité

Théorème 2 (**Décidabilité**). *La satisfiabilité dans FO est un problème* ***semi-décidable****. [HR04], [DNR+ 01]*

Grâce à l'utilisation des symboles de prédicats et de fonctions, des variables ainsi que des quantificateurs, FO est beaucoup plus expressif que la logique propositionnelle. Néanmoins, FO ne permet pas d'exprimer toute les propriétés possibles. Les systèmes peuvent posséder des propriétés relationnelles dont la spécification nécessite des connecteurs ensemblistes et relationnels. Bien que la plupart de ces connecteurs soient exprimables en FO, cette dernière n'est pas capable d'exprimer la fermeture transitive d'une relation. FO montre ses limites et le désir d'exprimer des propriétés ayant des connecteurs ensemblistes et relationnelles à conduit à la conception de la logique relationnelle du premier ordre.

2.3 Logique du premier ordre relationnelle

La logique relationnelle du premier ordre (FOR) qui est une extension de FO par des connecteurs ensemblistes et relationnels permet de spécifier propriétés contenant des connecteurs de la théorie ensembliste et relationnelle.

2.3.1 Syntaxe

Définition 2.3.1 (**Logique du premier ordre relationnelle**). *La logique du premier ordre relationnelle est une extension de la FO par des éléments de la théorie ensembliste et relationnelle. L'alphabet d'un langage FOR s'obtient à partir de celui de FO le façon suivante : Les symboles de fonctions sont éliminés et un ensemble de symboles de relation $\mathcal{R}$ sur lesquelles sont appliqués les opérateurs ensemblistes et relationnelles suivants est ajouté.*

- $\subseteq$ *inclusion ;*
- $+$ *union ;*
- & *intersection ;*
- $-$ *exclusion ;*
- $\times$ *produit cartésien ;*
- $'.'$ *jointure ;*
- $\hat{}$ *fermeture transitive ;*
- $\sim$ *transposition.*

Remarque 3. *Les opérateurs relationnels de fermeture transitive et de transposition sont appliqués uniquement sur les relations binaires.*

Termes et formules

Définition 2.3.2 (**Termes**). *Tout comme en FO, l'ensemble des termes de FOR est le plus petit ensemble τ défini inductivement sur $(\mathcal{V}, \mathcal{R})$ comme suit :*

- *Pour toute constante $c \in S_{\mathcal{C}}$, $c \in \tau$;*
- *Pour toute variable $x \in \mathcal{V}$, $x \in \tau$;*

- *si $t \in \tau$, alors $\sim t \in \tau$ et $\hat{} t \in \tau$;*
- *Si $r \in \mathcal{R}$ est un symbole de relation d'arité n et $t_1, t_2, ...t_n \in \tau^n$ alors, $r(t_1, t_2, ...t_n) \in \tau$.*
- *si $t_1 \in \tau$ et $t_2 \in \tau$: $t_1 \times t_2 \in \tau$, $t_1.t_2 \in \tau$, $t_1 + t_2 \in \tau$, $t_1 \& t_2 \in \tau$ et $t_1 - t_2 \in \tau$.*

Définition 2.3.3 (Formules). *L'ensemble des formules FOR est le plus petit ensemble Γ défini inductivement sur τ comme suit :*

— *Si $P \in \mathcal{P}$ est un symbole de prédicat d'arité $n \geq 1$, et si $t_1, ...t_n \in \tau^n$, alors $P(t_1, ...t_n) \in \Gamma$;*

— *si $t_1 \in \tau$ et $t_2 \in \tau$ alors, $t_1 \subseteq t_2 \in \Gamma$;*

— *si $\phi \in \Gamma$ alors, $\neg\phi \in \Gamma$;*

— *si $\phi \in \Gamma$ et $\psi \in \Gamma$ alors $\phi \wedge \psi \in \Gamma$, $\phi \vee \psi \in \Gamma$ et $\phi \Rightarrow \psi \in \Gamma$;*

— *si $\phi \in \Gamma$ et t une variable alors : $\forall t\phi \in \Gamma$ et $\exists t\phi \in \Gamma$;*

2.3.2 Sémantique

Soit $\mathcal{D}$ une structure de base ou domaine des variables du premier ordre. Pour donner un sens aux termes et aux formules, une fonction de valuation $\rho : \mathcal{R} \cup \mathcal{V} \cup S_{\mathcal{C}} \longrightarrow \mathcal{D}$ qui attribue les valeurs aux relations et aux variables est définie comme suit :

$$\rho(x) = \begin{cases} \{c_{\mathcal{D}}\} \text{ avec } c_{\mathcal{D}} \in \mathcal{D} \text{ si } x \text{ est une variable} \\ \{\langle s_0, s_1, ...s_n \rangle\} \text{ avec } s_i \in \mathcal{D} \ \ \forall i = 0, 1, ..., n \text{ si } x \text{ est un symbole de relation d'arité } n \end{cases} \tag{2.2}$$

La sémantique des termes sur $(\mathcal{D}, \mathcal{R})$ avec la valuation ρ est définie inductivement comme suit :

— $[\![r]\!]_\rho = \rho(r)$ $r \in \mathcal{R}$;

— $[\![v]\!]_\rho = \rho(v)$ $v \in \mathcal{V}$;

— $[\![p + q]\!]_\rho = \{x \mid x \in [\![p]\!]_\rho$ ou $x \in [\![q]\!]_\rho\}$

— $[\![p \ \& \ q]\!]_\rho = \{x \mid x \in [\![p]\!]_\rho$ et $x \in [\![q]\!]_\rho\}$

— $[\![p - q]\!]_\rho = \{x \mid x \in [\![p]\!]_\rho$ et x $\notin [\![q]\!]_\rho\}$

— $[\![p.q]\!]_\rho = \{\langle p_1, p_2, ..., p_{n-1}, q_2, ...q_m \rangle \mid \langle p_1, p_2, ..., p_n \rangle \in [\![p]\!]_\rho \ \langle q_1, q_2, ..., q_m \rangle \in [\![q]\!]_\rho$ *et* $p_n = q_1\}$;

— $[\![p \times q]\!]_\rho = \{\langle p_1, p_2, ..., p_n, q_1, q_2, ...q_m \rangle \mid \langle p_1, p_2, ..., p_n \rangle \in [\![p]\!]_\rho \ \langle q_1, q_2, ..., q_m \rangle \in [\![q]\!]_\rho\}$;

— $[\![\sim p]\!]_\rho = \{\langle p_1, p_2 \rangle \mid \{\langle p_2, p_1 \rangle \in [\![p]\!]_\rho\}$;

— $[\![\hat{} p]\!] = \{\langle x, y \rangle \mid \exists \langle p_1, ..., p_n \rangle \mid \langle x, p_1 \rangle, \langle p_1, p_2 \rangle, ..., \langle p_n, y \rangle \in [\![p]\!]_\rho\}$.

La satisfaction des formules est également définie par l'induction suivante :

— $\mathcal{M}, \rho \vDash t_1 \subseteq t_2$ iff $[\![t_1]\!]_\rho$ est incluse dans $[\![t_2]\!]_\rho$;

— $\mathcal{M}, \rho \vDash \neg\phi$ iff $\mathcal{M}, \rho \nvDash \phi$;

— $\mathcal{M}, \rho \vDash \phi \wedge \psi$ iff $\mathcal{M}, \rho \vDash \phi$ et $\mathcal{M}, \rho \vDash \psi$;

— $\mathcal{M}, \rho \vDash \phi \vee \psi$ iif $\mathcal{M}, \rho \vDash \phi$ ou $\mathcal{M}, \rho \vDash \psi$;

- $\mathcal{M}, \rho \vDash \ \forall x\phi$ iff $\mathcal{M}, \rho \vDash_{\rho[x \longrightarrow \{a\}]} \phi$ pour tout $a \in \mathcal{D}$;
- $\mathcal{M}, \rho \vDash \ \exists x\phi$ iff $\mathcal{M}, \rho \vDash_{\rho[x \longrightarrow \{a\}]} \phi$ s'il existe a $a \in \mathcal{D}$;

Théorème 3. *La satisfiabilité dans FOR est* ***semi-décidable****. Les opérateurs relationnels et particulièrement la fermeture transitive confère à FOR un pouvoir expressif plus grand que celui de FO. [HR04], [TD06],[TJ07],[Jac03], [Jac12].*

La satisfaction des formules propositionnelles, FO et FOR est évaluée statiquement, c'est-à-dire dans un système à un état. Toutefois, les propriétés qui caractérisent les systèmes dynamiques doivent être exprimées avec des formules dont la satisfaction varie avec le temps. Les logiques FO et FOR ne sont pas adaptées pour exprimer de telles formules . La nécessité d'exprimer des propriétés ayant des connecteurs temporelles à conduit à la conception des logiques temporelles dont la logique temporelle linéaire du futur et du passé que nous présentons dans cette thèse.

2.4 Logique Temporelle Linéaire (LTL)

La logique temporelle est une extension de la logique propositionnelle à l'aide des connecteurs temporels permettant de décrire l'évolution d'un système. Elle permet de décrire des propriétés sur une succession d'états. Sa particularité parmi les logiques temporelles est que le temps est considéré comme linéaire, c'est -à-dire que les instants se succèdent et qu'il n'y a qu'un seul futur.

2.4.1 Syntaxe de LTL

Définition 2.4.1 (**Alphabet du langage LTL**)**.** *Pour LTL l'alphabet est un ensemble défini par :*

— *les constantes* $(\top, \bot)$ *;*
— *les connecteurs de logique propositionnelle* $(\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg, =)$ *;*
— *les connecteurs de la logique temporelle linéaire* $(\mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{R}, \mathbf{W})$ *;*
— *l'ensemble des propositions atomiques* $A_{\mathcal{P}} = (p, q, r, , ,)$*.*

$\mathbf{X}$ *se lit (next),* $\mathbf{G}$ *(globally) ,* $\mathbf{F}$ *(future),*$\mathbf{U}$ *(until),* $\mathbf{W}$ *(weak until) et* $\mathbf{R}$ *(release).*

Définition 2.4.2. ***Formules***
L'ensemble des formules LTL est le plus petit ensemble Γ *défini par l'induction suivante :*

- $\top \in \Gamma$, $\bot \in \Gamma$, *pour tout* $P \in A_{\mathcal{P}}$, $P \in \Gamma$ *;*
- *si* $\phi \in \Gamma$ *alors,* $\neg\phi \in \Gamma$ *;*
- *si* $\phi \in \Gamma$ *et* $\psi \in \Gamma$ *alors,* $\phi \wedge \psi \in \Gamma$, $\phi \vee \psi \in \Gamma$ *et* $\phi \Rightarrow \psi \in \Gamma$ *;*
- *si* $\phi \in \Gamma$ *alors,* $(\mathbf{G}\,\phi) \in \Gamma$, $(\mathbf{F}\,\phi) \in \Gamma$, $(\mathbf{X}\,\phi) \in \Gamma$ *;*
- *si* $\phi \in \Gamma$ *et* $\psi \in \Gamma$ $\phi\,\mathbf{U}\,\psi \in \Gamma$, $\phi\,\mathbf{W}\,\psi \in \Gamma$ *et* $\phi\,\mathbf{R}\,\psi \in \Gamma$.

Exemple 1 (**Propriétés LTL classiques**)**.** ***Accessibilité*** *: Il existe un état ou une propriété* ϕ *sera vraie, elle s'écrit* $(\mathbf{F}\,\phi)$ *;*

Invariance *: Une propriété* ϕ *n'arrivera jamais, elle s'écrit* $(\mathbf{G}\,\phi)$ *;*

Vivacité *: A tout instant une propriété* ψ *sera vraie à partir d'une certaine étape de l'exécution où* ϕ *est vraie, elle s'écrit* $(\mathbf{G}(\phi \Rightarrow \mathbf{F}\,\psi))$ *;*

Equité forte *: Si infiniment souvent une cause* ϕ *a lieu alors, infiniment souvent son effet* ψ *sera observé, elle s'écrit* $(\mathbf{G}\,\mathbf{F}\,\phi \Rightarrow \mathbf{G}\,\mathbf{F}\,\psi)$ *;*

Equité faible *: Si une cause* ϕ *à souvent lieu infiniment alors, on finira par observer son effet* ψ*, elle s'écrit* $(\mathbf{F}\,\mathbf{G}\,\phi \Rightarrow \mathbf{G}\,\mathbf{F}\,\psi)$

2.4.2 Sémantique LTL

La satisfaction d'une formule est définie par rapport à une structure d'interprétation. Pour LTL, la structure d'interprétation est une suite indexée par les entiers naturels, appelée *trace d'exécution* qui permet de donner une valeur de vérité aux atomes à chaque instant du temps.

Définition 2.4.3 (**Trace d'exécution**). *Une trace d'exécution est un couple* $\pi = (\mathbb{N}, V)$ *où*

- *l'ensemble* $\mathbb{N}$ *des nombres naturels munit de la relation d'ordre strict représente les instants (ou états) ; le successeur de l'instant* i *est noté* $i + 1$ *;*
- $V : \mathbb{N} \longrightarrow 2^{\mathcal{P}}$ *est une fonction de valuation qui associe chaque instant à l'ensemble de propositions atomiques qui sont vraies à cet instant.*

Définition 2.4.4 (**Satisfaction des formules**). *On définit la relation de satisfaction entre une interprétation* π *, un instant* i *et une formule* ϕ *comme suit :*

— $\pi, i \not\models \bot$

— $\pi, i \models p$ *ssi* $p \in V(i)$

— $\pi, i \models \neg\phi$ *ssi* $\pi, i \not\models \phi$

— $\pi, i \models \phi \wedge \psi$ *ssi* $\pi, i \models \phi$ *et* $\pi, i \models \psi$

— $\pi, i \models \phi \vee \psi$ *ssi* $\pi, i \models \phi$ *ou* $\pi, i \models \psi$

— $\pi, i \models \phi \rightarrow \psi$ *ssi* $\pi, i \not\models \phi$ *ou* $\pi, i \models \psi$ *;*

— $\pi, i \models \mathbf{X}\,\phi$ *ssi* $\pi, i + 1 \models \phi$ *;*

— $\pi, i \models \mathbf{G}\,\phi$ *ssi pour tout* $j \geq i$ $\pi, j \models \phi$ *;*

— $\pi, i \models \mathbf{F}\,\phi$ *ssi il existe* $j \geq i$ *tel que* $\pi, j \models \phi$

— $\pi, i \models \phi\,\mathbf{U}\,\psi$ *ssi il existe* $j \geq i$ *tel que* $\pi, j \models \psi$ *et pour tous* $i \leq k \leq j - 1$ $\pi, k \models \phi$

— $\pi, i \models \phi\,\mathbf{W}\,\psi$ *ssi il existe* $j \geq i$ *tel que* $\pi, j \models \psi$ *et pour tous* $i \leq k \leq j - 1$ $\pi, k \models \phi$ *ou pour tout* $\ell \geq i$ $\pi, \ell \models \phi$

— $\pi, i \models \phi\,\mathbf{R}\,\psi$ *ssi il existe* $j \geq 1$ *tel que* $\pi, j \models \phi$ *et pour tous* $i \leq k \leq j - 1$ $\pi, k \models \psi$.

Satisfiabilité

Définition 2.4.5 (**Satisfiabilité et modèle**). *Une formule* ϕ *est dite satisfaite par une trace* π *(noté* $\pi \models \phi$*) si elle est satisfaite par son premier état. Une formule est dite valide si toute trace* π *le satisfait on parle de modèle de la formule et on note* $\models \phi$.

Le problème de satisfiabilité est le même que pour la logique propositionnelle à la différence que la structure d'interprétation est une trace d'exécution.

Définition 2.4.6 (**Problème de satisfiabilité de LTL**). *Le problème de satisfiabilité pour la logique linéaire temporelle se définit de la manière suivante : étant donnée une formule* ϕ*, existe-t-il une trace* π *telle que* $\pi \models \phi$ *?*

Théorème 4 (**Complexité de LTL**). *Dans la logique temporelle linéaire, la satisfiabilité est un problème décidable et PSPACE-complet. [HR04], [DNR*$^{+}$*01]*

2.5 Logique temporelle linéaire avec passé PLTL

Comme vue précédemment, les logiques temporelles linéaires **pur futur** ($\mathbf{F}, \mathbf{G}, \mathbf{X}, \mathbf{U}, \mathbf{W}$) expriment des propriétés à vérifier sur des traces exécutions qui partent d'un état **s**. Les logiques temporelles linéaires avec passé (PLTL) s'intéressent aussi aux traces exécutions qui arrivent jusqu'à un état **s**. La logique PLTL est une extension de LTL avec les deux opérateurs du passé **(S,Y)**.

2.5.1 Syntaxe de PLTL

Définition 2.5.1 (**Alphabet du langage PLTL**). *Pour la logique temporelle linéaire avec passé l'alphabet est un ensemble défini par :*

— *les constantes* $(\top, \bot)$ *;*

— *les connecteurs de la logique propositionnelle* $(\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg, \top, \bot)$ *;*

— *les connecteurs de la logique temporelle linéaire (du futur)* $(\mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{R}, \mathbf{W})$ *;*

— *les connecteurs de la logique temporelle linéaire du passé* $(\mathbf{S}, \mathbf{Y})$ *;*

— *l'ensemble des propositions atomiques* $\mathcal{P} = \{p, q, r, , , \}$

- **Y** *se lit* **yesterday** *et signifie : pour tout formule* ϕ*,* $\mathbf{Y}\phi$ *est satisfaite dans un état d'une trace, si dans l'état précédent dans cette trace* ϕ *était satisfaite ;*
- **S** *se lit* **since** *et signifie : pour* ϕ *et* ψ *deux formules,* $\phi\, \mathbf{S}\, \psi$ *est satisfaite dans un état d'une trace, si* ϕ *est restée vrai dans le passé depuis l'instant où* ψ *était satisfait.*

La BNF des formules de PLTL est la suivante : $\phi ::= \top \mid \bot \mid p \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \neg\phi \mid (\phi \Rightarrow \phi) \mid (\mathbf{G}\, \phi) \mid (\mathbf{F}\, \phi) \mid (\mathbf{X}\, \phi) \mid (\phi\, \mathbf{U}\, \phi) \mid (\phi\, \mathbf{W}\, \phi) \mid (\phi\, \mathbf{R}\, \phi) \mid (\mathbf{Y}\phi) \mid (\phi\, \mathbf{S}\, \phi)$ avec $P \in \mathcal{P}$

2.5.2 Sémantique PLTL

La structure d'interprétation dans PLTL est une trace d'exécution. La satisfaction est vérifiée sur des traces autant finies qu'infinies . Cependant, dans cette thèse, nous ne présenterons que le PLTL sur les traces infinies car, les langages définis sur PLTL qui nous intéressent, opèrent uniquement sur horizon de temps infini. La sémantique des connecteurs LTL reste la même. La sémantique des opérateurs du passé est donnée par la définition suivante.

Définition 2.5.2 (**Satisfaction des formules PLTL**). *On définit la relation de satisfaction entre une interprétation* π *, un instant* i *et une formule* ϕ *comme suite :*

$$\begin{array}{l} \pi, i \models \mathbf{Y}\phi \textit{ si } i > 0 \textit{ et } \pi, i-1 \models \phi \\ \pi, i \models \phi\, \mathbf{S}\, \psi \textit{ s'il existe } j \leq i \textit{ tel que } \pi, j \models \psi \\ \quad \textit{et pour tout } j \leq k \leq i, \pi, k \models \phi \end{array}$$

Dans PLTL, l'utilisation des connecteurs du passé n'augmente pas son expressivité par rapport à LTL. PLTL a l'avantage d'être naturelle en ce sens que les connecteurs du passé facilitent l'expression des spécifications.

Théorème 5 (**Décidabilité, complexité**). *Le problème de satisfiabilité dans PLTL est décidable et PSPACE-complet. PLTL possède le même degré d'expressivité que LTL, cependant PLTL est plus concise (la meilleure traduction d'une formule PLTL vers LTL donnant une formule équivalente de taille triplement exponentielle) [Mar03].*

De nombreux systèmes réels possèdent des propriétés structurelles et comportementales. Pour ce type de systèmes, on souhaite modéliser ces deux aspects à la fois. Cette modélisation doit s'adosser sur une logique capable d'exprimer ces deux types de propriétés. Les logiques présentées jusqu'ici expriment exclusivement l'un ou l'autre type de propriétés. Les limites de ces logiques et la nécessité de modéliser les systèmes dynamiques dotées de propriétés structurelles riches à conduit à s'intéresser à la logique du premier ordre temporelle linéaire.

2.6 La logique du premier ordre temporelle linéaire FOLTL

2.6.1 Syntaxe

Définition 2.6.1 (**Alphabet**). *Pour la logique du premier ordre temporelle linéaire, l'alphabet est un ensemble défini par :*

- *les connecteurs et les constantes de la logique propositionnelle* ($\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg, \top, \bot, \forall, \exists$)
- *les quantificateurs de la logique du premier ordre* ($\forall$ *et* $\exists$)*;*
- *l'ensemble des variables* $\mathcal{V} = \{x, y, z, ...\}$ *;*
- *les opérateurs temporels linéaires (LTL et PLTL)* ($\mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{R}, \mathbf{W}, \mathbf{S}, \mathbf{Y}$)
- *l'ensemble des symboles de prédicats* $\mathcal{P} = (P, Q, R, , ,)$
- *le prédicat* $\doteq$ *défini sur l'ensemble des variables.*

Les termes dans FOLTL sont les termes de FOR. La syntaxe des formules FOLTL est donnée par la BNF :

$$\phi ::= P(x_1, \ldots, x_k) \mid x_1 \doteq x_2 \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall x.\phi \mid \exists x.\phi \mid \phi \Rightarrow \phi \\ \mid \mathbf{X}\,\phi \mid \mathbf{Y}\phi \mid \mathbf{G}\,\phi \mid \mathbf{F}\,\phi \mid \phi\,\mathbf{U}\,\phi \mid \phi\,\mathbf{R}\,\phi \mid \phi\,\mathbf{S}\,\phi$$

avec $x_i \in \mathcal{V}$ *et* $P \in \mathcal{P}$ *(* $P(x_1, \ldots, x_k)$ *est une formule seulement si l'arité de* P *est* k*).*

2.6.2 Sémantique

Définition 2.6.2 (**Structure d'interprétation**). *Une structure d'interprétation pour FOLTL est un couple* $\mathcal{M} = (\mathcal{D}, \rho)$ *avec :*

- $\mathcal{D}$ *un domaine des variables du premier ordre ;*
- ρ *une fonction qui associe à chaque prédicat* $P \in \mathcal{P}$ *à chaque instant* $i \in \mathbb{N}$ *une relation* $\rho(P, i) \subseteq \mathcal{D}^k$ *où* k *est l'arité de* P *[MBC+16].*

Tout comme dans FO la sémantique des formules est relative à un environnement qui associe chaque variable libre x à un élément du domaine $\mathcal{D}$. Ainsi, la satisfaction des formules est donnée par la définition suivante.

Définition 2.6.3 (**Satisfaction des formules**). *On définit la relation de satisfaction entre une structure $\mathcal{M}$, un instant $i \in \mathbb{N}$, un environnement σ et une formule ϕ de la façon suivante :*

$$\begin{array}{l}
\mathcal{M}, \sigma, i \models x \doteq y \textit{ si } \sigma(x) = \sigma(y) \\
\mathcal{M}, \sigma, i \models P(x_1, \ldots, x_n) \textit{ if } (\sigma(x_1), \ldots, \sigma(x_n)) \in \rho(P, i) \\
\mathcal{M}, \sigma, i \models \neg\phi \textit{ if } \mathcal{M}, \sigma, i \not\models \phi \\
\mathcal{M}, \sigma, i \models \phi \vee \psi \textit{ si } \mathcal{M}, \sigma, i \models \phi \textit{ or } \mathcal{M}, \sigma, i \models \psi \\
\mathcal{M}, \sigma, i \models \forall x.\phi \textit{ if for all } a \in \mathcal{D}, \mathcal{M}, \sigma[x \mapsto a], i \models \varphi \\
\mathcal{M}, \sigma, i \models \mathbf{X}\,\phi \textit{ si } \mathcal{M}, \sigma, i+1 \models \phi \\
\mathcal{M}, \sigma, i \models \mathbf{G}\,\phi \textit{ si for each } j \geq i, \mathcal{M}, \sigma, j \models \phi \\
\mathcal{M}, \sigma, i \models \mathbf{F}\,\phi \textit{ s'il existe } j \geq i \textit{ tel que } \mathcal{M}, \sigma, j \models \phi \\
\mathcal{M}, \sigma, i \models \phi\,\mathbf{U}\,\psi \textit{ s'il existe } j \geq i \textit{ tel que } \mathcal{M}, \sigma, j \models \psi, \\
\qquad\qquad\qquad\qquad \textit{et pour tout } i \leq k < j, \mathcal{M}, \sigma, k \models \phi.
\end{array}$$

FOLTL est enrichie avec les connecteurs du passé **Y** *et* **S** *dont la sémantique est :*

$$\begin{array}{l}
\mathcal{M}, \sigma, i \models \mathbf{Y}\phi \textit{ si } \mathcal{M}, \sigma, i-1 \models \phi \textit{ avec } i \geq 1\,^{2}; \\
\mathcal{M}, \sigma, i \models \phi\,\mathbf{S}\,\psi \textit{ s'il existe } 1 \leq j \leq i \textit{ tel que} \\
\mathcal{M}, \sigma, j \models \psi \textit{ et pour tout } j \leq k \leq i, \mathcal{M}, \sigma, k \models \phi
\end{array}$$

Une formule ϕ close est satisfaite si et seulement s'il existe un modèle $\mathcal{M}$ tel que $\mathcal{M}, \emptyset, 0 \models \phi$, ce qu'on note simplement $\mathcal{M} \models \phi$.

Théorème 6 (**Décidabilité et expressivité**). *La satisfiabilité dans FOLTL est indécidable.*

2.7 Conclusion

La logique est à la base des langages de modélisation et de vérification formels des systèmes. Les systèmes sont représentés par des modèles abstraits sur lesquels on peut raisonner formellement, tandis que les propriétés sont spécifiées par des formules d'une logique définie. Le raisonnement concerne :

- soit le problème de model checking qui consiste à prouver qu'étant donnés un modèle $\mathcal{M}$ et une formule ϕ, on a $\mathcal{M} \vDash \phi$;
- soit le problème de satisfiabilité qui consiste à vérifier s'il existe un modèle $\mathcal{M}$ tel que $\mathcal{M} \vDash \phi$, pour une formule ϕ donnée.

La logique FOR est suffisante pour le développement des langages de spécification et de vérification des systèmes statiques complexes, tandis que, la logique LTL est très adéquate pour la conception des langages de modélisation des systèmes dynamiques. La logique FOLTL est adaptée à la conception des langages de spécification et de vérification les systèmes dynamiques dotés de propriétés structurelles riches.

Il existe plusieurs techniques de vérification formelle qui opèrent de plusieurs manières : la vérification peut être automatisée (model-checking), semi-automatisée (assistant de preuve) ou manuelle. Lorsqu'il s'agit des propriétés temporelles la vérification peut se faire sur des traces bornées ou non bornées.

Dans le chapitre suivant nous introduisons d'abord Alloy le langage de spécification et de vérification formelle basé sur FOR, qui est très adapté à analyse des propriétés structurelles sur des systèmes complexes, mais qui repose sur des techniques ad'hoc pour spécifier et analyser les propriétés temporelles et ceci sur des horizons de temps bornés. Ensuite nous présentons DynAlloy une extension qui facilite la spécification des propriétés temporelles des systèmes dans Alloy. En parallèle, nous présentons, le langage TLA+ basé sur une variante de LTL(Logique Temporelle pour les Actions TLA.) Il réalise la vérification des propriétés temporelles sur des horizons de temps non bornés. Nous rappelons que le but de l'étude est de développer des langages permettant de spécifier et de vérifier les systèmes dynamiques avec de propriétés structurelles riches.

Chapitre 3

Les langages Alloy, Dynalloy et TLA+

La logique sert à développer les langages de spécification et de vérification permettant d'analyser les systèmes réels. Cependant, la sémantique d'un langage de spécification formel doit être donnée dans une logique dont le pouvoir expressif permet de spécifier le type de propriétés ciblées (ex : la sémantique des langages adaptés à la spécification des propriétés structurelles sur les systèmes peut être donnée dans FOR). En effet, le choix et l'utilité d'une méthode formelle sont dictés par : la complexité du système et le type de propriétés (statiques, dynamiques) que l'on souhaite analyser avec la technique ; son degré d'automatisation ainsi que les outils de vérification qui l'accompagnent ; le niveau d'abstraction qu'offre son langage de modélisation.

Le model checking est une technique de vérification formelle dont l'objectif est de vérifier si, étant donné une abstraction (un modèle) d'un système et une spécification (formule d'une logique définie) d'une propriété, le modèle satisfait la formule.

Historiquement, le model checking est conçu principalement pour la vérification des interactions entre systèmes concurrents, mais n'est pas très adapté pour la vérification des propriétés structurelles des systèmes. Pour la vérification de telles propriétés sur des systèmes dynamiques, le langage Alloy a été conçu en intégrant la technique de vérification complètement automatisée du model checking à un langage de spécification des propriétés structurelles des systèmes : le langage Z [Bow01], [Spi92], [PTS96].

3.1 Le langage Alloy et Alloy Analyzer

Alloy [Jac06],[Jac12], [Jac03],[HR04] est un langage de spécification formelle très expressif orienté objet, qui permet de décrire et d'explorer des modèles sur lesquels des propriétés de la logique relationnelle du premier ordre sont vérifiées. Il est accompagné d'un analyseur complètement automatisé basé sur *KodKod*[TD06], [TJ07] permettant la vérification des modèles Alloy par une compilation vers un solveur standard SAT [SE05], [SE09], [LBP10]. L'automatisation de la vérification permet un retour immédiat sur les modèles, cependant la vérification n'est pas complète en ce sens qu'elle est réalisée sur un domaine fini (espace des cas). Néanmoins, grâce aux avancées dans les technologies SAT, le domaine de vérification couvert est assez grande(des milliards de cas).

L'utilisation d'Alloy concerne de nombreux domaines comme : l'analyse des protocoles réseau et web [CGKP06], [GMB06], [PKPZ07], [Zav19], [Zav11], [GPCR12], [Kum12], [Zav15b], [ME15], [Zav15a], [Zav17], l'analyse des modèles de transformation et de raffinement [ABK07], [Bol05], [PSK+11], la spécification de l'architecture des systèmes logiciels [MS08].

Dans la suite pour des raisons de précision et de clarté, la présentation des concepts s'appuiera sur le système de verrouillage des chambres d'hôtel par des clés électroniques jetables. Dans cette thèse, ce système sera référencé par ***Hôtel***.

Le système de verrouillage des portes d'hôtels *a été introduit par Jackson dans [Jac06]. Il traite de l'utilisation des cartes contenant des clés jetables pour l'ouverture des portes dans les hôtels.*

À l'arrivée de chaque hôte le système lui attribue une clé qu'il pourra éventuellement garder à son départ. Le système doit pouvoir empêcher un hôte de rentrer dans sa chambre après que celle-ci a été attribuée à quelqu'un d'autre. L'idée est que l'hôtel attribue à chaque nouvel occupant, une nouvelle clé qui recode la serrure de sorte que les clés précédemment codées dans cette serrure ne fonctionnent plus.

La serrure est une unité autonome ayant une mémoire contenant la clé en cours d'utilisation. Les clés sont émises par un générateur de nombres pseudo-aléatoires. Une serrure est déverrouillée soit par la clé en cours d'utilisation, soit par le successeur de la clé en cours d'utilisation. Le codage du successeur de la clé en cours d'utilisation dans la serrure rend cette dernière obsolète. Aucune communication entre la réception et les serrures n'est requise. En utilisant le même générateur pseudo-aléatoire et en synchronisant initialement la réception et les serrures des portes, la réception peut conserver ses enregistrements de la clé en cours d'utilisation en phase avec les portes elles-mêmes.

3.1.1 Atomes et relations

Dans Alloy tous, les éléments sont construits à partir de relations entre atomes.

Définition 3.1.1 (**Atome**)**.** *Les atomes sont les entités de base dans Alloy. Ils sont indivisibles et non interprétés.*

Les éléments plus complexes sont construis à l'aide des relations entre les atomes

Définition 3.1.2 (**Relation**)**.** *Une relation d'arité n est une structure qui relie les atomes. Elle consiste en un ensemble de n-uples d'atomes. Par analogie aux bases de données, une relation peut être vue comme une table d'atomes dans laquelle l'ordre des colonnes importe et chaque ligne possède obligatoirement une entrée pour chacune de ses colonnes.*

Toutes les structures en Alloy sont représentées par des relations, en particulier :

— Un ensemble est une relation unaire (une colonne) ;

— Un scalaire est un singleton (une ligne une colonne).

L'intérêt de cette représentation est d'uniformiser l'application des opérateurs relationnels sur les structures.

Dans Alloy, les relations sont dites plates ou du premier ordre [Jac03], c'est-à-dire qu'elles contiennent uniquement des atomes et pas d'autres relations. Cette restriction aux relations du premier ordre rend la logique plus facile à analyser. En effet,les relations plates sont suffisamment expressives pour de nombreuses applications.

Cependant, elle entraîne une perte de pouvoir expressif du langage, même s'il est généralement possible de contourner le problème en reformulant en une relation plate une situation qui semble faire appel à une relation d'ordre supérieur.

Un modèle [1] Alloy est un ensemble de déclarations de signatures, de champs et de formules.

3.1.2 Signatures et champs

Définition 3.1.3 (**Signature**). *Dans Alloy, une structure est introduite par une déclaration de* signature *avec le mot-clé `sig`. La sémantique d'une* signature *est un ensemble d'atomes.*

Une signature peut être introduite comme sous-signature d'une autre signature. Ainsi, la déclaration des signatures établit une classification hiérarchique. Cette hiérarchie peut être introduite de plusieurs manières.

Une signature peut être introduite comme sous-signature d'une autre signature. Ainsi, la déclaration des signatures établit une classification hiérarchique. Cette hiérarchie peut être introduite de plusieurs manières

Définition 3.1.4 (**Extension de signature**). *Une hiérarchie par extension est introduite par le mot-clé `extends`, sémantiquement ceci signifie que les sous-signatures de la signature englobante sont toutes disjointes.*

Définition 3.1.5 (**Inclusion de signature**). *Une hiérarchie par inclusion est introduite par le mot-clé `in` sa sémantique est simplement que les sous-signatures sont incluses dans la signature englobante (et peuvent donc se chevaucher).*

Définition 3.1.6 (**Signature top-level**). *Une signature est top-level si elle est introduite indépendamment de toutes autres signatures.*

Définition 3.1.7 (**Signature abstraite**). *Les signatures abstraites sont introduites par le mot-clé `abstract`. Sémantiquement les atomes de la signature appartiennent alors uniquement à ses sous-signatures concrètes par extension.*

Définition 3.1.8 (**Champ**). *Les relations entre signatures sont introduites par des déclarations de* champs *de signatures. La sémantique d'un champ est une relation dont chaque tuple contient les atomes des signatures concernées.*

Définition 3.1.9 (**Champ variable**). *Les champs dont la sémantique peut changer avec le temps sont dits* variables*.*

L'accès aux champs d'une signature se fait par le biais de l'opérateur relationnel de jointure `<<.>>` ainsi, si r a pour sémantique un atome de la signature `Room`, `r.currentKey` est la clé actuellement encodée dans sa serrure, tandis que `r.keys` est l'ensemble des clés qui lui sont attribuées. Il en est de même pour l'accès aux champs des autres signatures.

1. Ici il ne s'agit pas d'un modèle au sens logique, mais seulement de l'ensemble des spécifications du système

Définition 3.1.10 (**Instance**). *Une instance est une affectation des valeurs aux variables relationnelles, qui satisfait toutes les contraintes du modèle (cf définition 3.1.12). Ces variables relationnelles, auxquelles on attribue des valeurs, sont les signatures et les champs. L'affectation est effectuée selon les règles suivantes :*

À une signature, on affecte un ensemble d'atomes et

à un champ, on affecte une relation.

On peut chercher des instances spécifiques qui satisfont une contrainte particulière, dans ce cas, on affectera des arguments aux variables relationnelles contenues dans la contrainte.

Exemple 2. *Dans le système de l'Hôtel dont le modèle est représenté à la figure 3.1, les clés, les chambres, les hôtes, et la réception sont introduites respectivement par les signatures* top-level *et non abstraites : Keys [l :3] ; Room [l :5] ; Guest [l : 16] ; FD [l : 12] .*

L'ensemble des clés attribuées à une chambre et la clé actuellement encodée dans sa serrure, sont introduits respectivement par la déclaration des champs `keys` [l : 6] et `currentKey` [l :7] de la signature `Room`. Les champs `lastKey` [l : 14] et `occupant` de la signature `FD`, ainsi que `gkeys` de la signature `Guest` représentent dans cet ordre la dernière clé enregistrée à la réception pour une chambre, l'actuel occupant d'une chambre et le trousseau de clés d'un hôte.
Une instance d'hôtel ayant deux chambres, deux hôtes et trois clés peut être donnée par :

$Room = \{(R1), (R2)\}$

$Guest = \{(G1), (G2)\}$

$Key = \{(K1), (K2), (K3)\}$ *avec*

$atomes = \{R1, R2, G1, G2, K1, K2, K3\}$

Pour les relations :

$\{(R1, G2, K1), (R1, G1, K2), (R2, G2, K3), (R2, G1, K1)\}$ *appartient à* `occupant`

$\{(R1, K1), (R1, K2), (R1, K3), (R2, K1), (R2, K3)\}$ *appartient à* `Currentkey`

$\{(G1, K1), (G1, K2), (G2, K3)\}$ *appartient à Gkeys*

Contraintes de multiplicité

Définition 3.1.11 (**Multiplicité**). *La* multiplicité *impose des contraintes sur la cardinalité des champs et des signatures.*

Pour une signature, la cardinalité est le nombre d'atomes de cette signature dans une instance du modèle. Plus précisément m `sig A` signifie que chaque instance du modèle possède m atomes de A.

Pour une relation donnée $r : S\ m \rightarrow n\ E$, la multiplicité ici indique que chaque atome de S est associé à n atomes de E par la relation r et que chaque atome de E est associé à m atomes de S par la relation r.

Alloy fournit des mot-clés pour spécifier cette multiplicité dont l'analogie aux bases de données est la suivante :

— `lone` : au plus un atome (**(0 - 1)** en base de données)

— `one` : exactement un atome (**(1 - 1)** en base de données)

— `some` : au moins un atome(**(1 - n)** en base de données)

— `set` : zéro ou plusieurs un atome (**(0 - n)** en base de données)

par défaut (si elle n'est pas indiquée) la multiplicité est `one` pour les relations unaire, sinon elle vaut `set` .

Exemple 3. *Dans une instance d'Hôtel, il y a exactement une réception ce qui est spécifié avec la multiplicité* `one` *sur la signature* `FD` *[l : 12]. À tout instant, une unique clé est encodée dans la serrure d'une porte [l : 7] d'où l'utilisation de la contrainte* `one` *sur le champ* `currentkey`*. La réception peut enregistrer au plus une clé par chambre, ceci est spécifié par la contrainte* `lone`*[l : 14].*

```
1  open util/ordering[Time] as to
2  open util/ordering[Key] as ko
3  sig Key {}
4  sig Time {}
5  sig Room {
6    keys: set Key,
7    currentKey: keys one → Time
8  }
9  fact DisjointKeySets {
10   Room<: keys in Room lone→ Key
11 }
12 one sig FD {
13   lastKey: (Room → lone Key) → Time,
14   occupant: (Room → Guest) → Time
15 }
16 sig Guest {
17   gkeys: Key → Time
18   }
19 fun nextKey [k: Key, ks: set Key]:
20   set Key{
21   min [k.nexts & ks]
22 }
23 pred init [t: Time] {
24   no Guest.keys.t
25   no FD.occupant.t
26   all r: Room |
27    FD.lastKey.t [r] = r.currentKey.t
28 }
29 pred noFrontDeskChange [t,
30      t': Time] {
31  FD.lastKey.t = FD.lastKey.t'
32  FD.occupant.t = FD.occupant.t'
33 }
34 pred noRoomChangeExcept[t,
35   t': Time, rs: set Room] {
36   all r: Room − rs |
37      r.currentKey.t = r.currentKey.t'
38 }
```

```
1  pred noGuestChangeExcept [t, t': Time,
2    gs: set Guest] {
3    all g: Guest − gs | g.keys.t = g.keys.t'
4  }
5  pred checkin [t, t': Time, g: Guest,r:
6    Room, k: Key] {
7    g.keys.t' = g.keys.t + k
8    let occ = FD.occupant {
9        no occ.t [r]
10       and occ.t' = occ.t + r → g
11       }
12
13   let lk = FD.lastKey {
14      lk.t' = lk.t ++ r → k
15      and k = nextKey [lk.t [r], r.keys]
16      }
17  }
18 noRoomChangeExcept [t, t', none]
19   noGuestChangeExcept [t, t', g] {
20 }
21 pred entry [t, t': Time, g: Guest,
22   r: Room, k: Key] {
23   k in g.keys.t
24   let ck = r.currentKey {
25    (k = ck.t and ck.t' = ck.t)
26   or (k = nextKey[ck.t, r.keys] and ck.t' = k)
27   }
28   noRoomChangeExcept [t, t', r]
29   noGuestChangeExcept [t, t', none]
30   noFrontDeskChange [t, t']
31 }
32 pred checkout [t, t': Time, g: Guest] {
33   let occ = FD.occupant {
34     some occ.t.g
35     and occ.t' = occ.t − Room →g
36     }
37   FD.lastKey.t = FD.lastKey.t'
38   noRoomChangeExcept [t, t', none]
39   noGuestChangeExcept [t, t', none]
40 }
```

```
1
2  fact traces {
3    init [ first ]
4    all t: Time−last | let t' = t.next {
5      some g: Guest, r: Room, k: Key |
6        entry [t, t', g, r, k]
7        or checkin [t, t', g, r, k]
8        or checkout [t, t', g]
9    }
10 }
11 assert NoBadEntry {
12   all t: Time, r: Room, g: Guest, k: Key {
13     let t' = t.next,
14       o = FD.occupant.t[r]{
15       entry [t, t', g, r, k]
16       and some o ⇒ g in o
17     }
18 }
19 pred consitence {}
20 run consistence for 4 but exactly 4 Key,
21   exactly 3 Room
```

```
1
2  fact NoIntervening {
3    all t: Time−last {
4      let t' = t.next, t'' = t'.next{
5        all g: Guest, r: Room, k: Key {
6          checkin [t, t', g, r, k]
7          ⇒ (entry [t', t'', g, r, k] or no t'')
8            or entry [t, t', g, r, k]
9              and some o ⇒ g in o }}
10   }
11 }
12 check NoBadEntry
13 check NoBadEntry for 5 but exactly 5 Key,
14   10 Time
```

FIGURE 3.1 – Modèle Alloy de l'Hôtel

3.1.3 Formules

Les contraintes sur le modèle sont introduites par des déclarations de formules appelées faits, qui s'ajoutent aux contraintes de multiplicité.

Définition 3.1.12 (Faits). *Les faits(mot clè `fact`) sont des axiomes auxquels chaque instance du modèle doit se conformer.*

Dans Alloy, les faits sont des formules FOR dont la sémantique est donnée au chapitre. 2, section 2.3. Les signatures et les champs jouent le rôle de prédicats. La structure d'interprétation est une instance.

Exemple 4. *Le fait (`fact`) DisjointKeys [l : 9] dans le modèle Hôtel impose que dans une instance de l'Hôtel l'ensemble des clés attribuées aux différentes chambres soient disjointes.*

Définition 3.1.13 (Faits locaux). *Les faits locaux d'une signature sont des axiomes auxquels chaque atome de la signature doit se conformer.*

Du point de vue syntaxique, les faits locaux ne sont pas introduits par le mot-clé `fact`, ils sont spécifiés juste après la déclaration de la signature et de ses champs, entre accolades.

3.1.4 Prédicats et fonctions

Pour factoriser la spécification du modèle, des contraintes (terminologie Alloy pour les formules) et des expressions réutilisables sont introduites par des déclarations de prédicats et de fonctions.

Définition 3.1.14 (Prédicat). *Un prédicat est une déclaration d'un ensemble de contraintes avec ou sans paramètres.*

Exemple 5. *Le prédicat `init`[l : 23] de la figure 3.1 spécifie l'ensemble des contraintes sur l'état initial du système.*

Les prédicats sont utiles à plus d'un titre : ils sont souvent utilisés pour la spécification des événements du système, ainsi que certaines contraintes qui pour des raisons d'analyse ne doivent pas être définies comme des faits.

Définition 3.1.15 (**Fonction**)**.** *Une fonction est une déclaration d'expressions réutilisables avec d'éventuels paramètres.*

Exemple 6. *Dans le modèle de l'Hôtel la fonction `nextkey` [l : 20] de la figure 3.1 calcule la valeur de la prochaine clé qui sera encodée dans la serrure d'une porte à partir de celle en cours d'utilisation. Pour permettre ce calcul, les clés sont totalement ordonnées.*

3.1.5 Assertion

Dans un modèle Alloy, les propriétés à vérifier sont introduites par la déclaration des assertions.

Définition 3.1.16 (**Assertion**)**.** *Une assertion est une contrainte qui doit être impliquée par la conjonction des faits du modèle. Elle est introduite par le mot-clé `assert`.*

Les assertions sont vérifiées par l'analyseur d'Alloy. Si l'assertion n'est pas impliquée par les faits alors il peut y avoir des erreurs soit dans la conception du modèle soit dans la formulation de l'assertion.

Exemple 7. *Dans le modèle de l'Hôtel l'assertion `NoBadEntry` spécifie la contrainte qui stipule que seuls les occupants d'une chambre ont le droit d'y accéder.*

3.1.6 Modélisation du comportement dans Alloy

L'évolution du système est décrite par des séquences d'exécution d'événement élémentaires.

Modélisation du temps

Alloy repose sur FOR dans laquelle les propriétés comportementales ne peuvent être spécifiées de façon native. Pour analyser les systèmes dynamiques Alloy utilise des idiomes très connus [Jac03] qui consistent à introduire dans le modèle une signature `Time` dont les atomes sont ordonnés et représentent les instants de temps possibles. Cet ordre (`util/ordering`) est entièrement défini par la donnée d'un plus petit élément(`first`), d'un plus grand élément (`last`) et d'une fonction partielle `next` qui calcule le successeur d'un atome.

Alloy offre deux idiomes pour la spécification des champs variables, un **idiome** étant un moyen standard d'exprimer une construction courante.

***état global* :** la signature `Time` regroupe tous les champs variables du modèle ;

***état local* :** les champs variables sont dans les mêmes signatures que les champs statiques. Dans ce cas une colonne supplémentaire représentant le temps est ajoutée aux champs variables, la jointure sur cette colonne indique à chaque instant la valeur du champ.

Exemple 8. *Le modèle Hôtel est spécifié selon l'idiome* état local *avec la signature `Time`, ainsi les champs variables `currentKey, lastKey, occupant, gkeys` ont une colonne supplémentaire `Time`.*

Modélisation des événements

La modélisation des événements dans Alloy se fait suivant un idiome, cet idiome n'est pas fixé. En effet grâce à sa flexibilité, le langage Alloy permet aux concepteurs de définir des idiomes à leurs convenances. En Alloy, deux idiomes sont couramment utilisés pour la spécification des événements des systèmes.

Idiome prédicat Dans cet idiome, les événements des systèmes sont spécifiées par des prédicats en relation avec deux instances de temps consécutives. Le système de transition du modèle est défini par une relation entre les gardes les postconditions.

Les gardes des prédicats spécifient l'état avant l'exécution de l'événement (les conditions dans lesquelles l'opération est exécutable), tandis que les postconditions décrivent l'état du système (effets de l' opération) après l'exécution de l'événement.

Exemple 9. *Le modèle de l'Hotel de la figure 3.1 est spécifie selon l'idiome prédicat. En effet, les événements d'enregistrement, d'entrée et de sortie de l'hôtel sont spécifiées respectivement par les prédicats* `checkin`*[l : 6],* `checkout`*[l :32], et* `entry` *[l : 22]*

Idiome Event En dehors de l'idiome prédicat, Alloy offre l'idiome *Event* dans lequel les événements sont spécifiés par des signatures dont les champs sont les paramètres de l'événement. La spécification du système de l'Hôtel selon l'idiome *Event* donne le modèle présenté dans la figure 3.2. Chaque prédicat devient une déclaration d'une signature dont les atomes sont des ensembles d'événements et son corps devient des faits locaux de cette signature. Par exemple la signature `checkout` [l : 9] spécifie l'ensemble des opérations dans lesquelles un hôte quitte l'hôtel.

Contrairement à l'idiome prédicat, cet idiome offre au concepteur le moyen de définir une hiérarchie entre les opérations. Cette hiérarchisation permet le partage des paramètres et des contraintes pour une spécification plus simple et facile à gérer.

Exemple 10. *Dans le modèle de l'Hôtel, les opérations se partagent le paramètre hôte (*`Guest`*). En outre, les conditions du cadre (cf definition 3.1.17) sur* `lastkey` *définies dans la signature FD est partagées par les signatures* `checkout` *et* `entry`*.*

En fonction des paramètres partagés une hiérarchisation des signatures est définie. La signature singleton `Event` *regroupe les signatures qui ont les champs* `Guest` *et deux instants de temps consécutif. Par ailleurs, les signatures* `checking` *et* `entry` *ayant les champs* `room` *et* `key` *en commun, sont regroupées par la signature* `RoomKeyEvent`*. Ainsi la signature* `checkout` *étend directement la signature* `Event`*, tandis que* `checking` *et* `entry` *étendent la sous-signature* `RoomKeyEvent` *de* `Event`*.*

La trace d'exécution (cf : section suivante) et les assertions sont beaucoup plus simples à exprimer que dans l'idiome prédicat.

L'assertion stipule qu'un événement `checkin` est soit le dernier événement soit est directement suivi par un événement `entry` sur la même chambre.

Un autre avantage de l'idiome `Event` vient du fait que les événements sont des atomes, ce qui permet à chaque instant de l'analyse d'identifier les événements qui se sont produits. Néanmoins,

contrairement à l'idiome prédicat, l'espace d'états généré est beaucoup plus grand et croit plus rapidement, ce qui peut avoir de l'influence sur la taille des systèmes vérifiables.

```
abstract sig Event {
  pre, post : Time,
  g: Guest
   }
abstract sig RoomKeyEvent extends Event {
room: Room ,
key: Key ,
 }
sig checkout extends Event{}  {
  let occ = FD.occupant {
    some occ.pre.guest
    occ.post = occ.pre − Room →guest
}
```

```
sig entry extends RoomKeyEvent { } {
  key in guest.keys.pre
  let ck = room.currentKey{
     (k = ck.pre and ck.post = ck.t) or
     (key = nextKey[ck.pre, room.keys] and
            ck.post = key)
}
sig checkin  extends RoomKeyEvent { }{
  guest.keys.post = guest.keys.pre + key
    let occ = FD.occupant {
    no occ.pre [room]
    occ.post = occ.pre + room → guest
  }
  let lk = FD.lastKey {
     lk.post = lk.pre ++ room → key
     key = nextKey [lk.pre [room], room.keys]
 }
 }
```

```
fact traces {
  init [ first ]
  all t: Time−last | let t' = t.next {
    some e: Event {
      e.pre = t and e.post = t'
// Conditions du cadre a la Reiter
      currentKey.t != currentKey.t'
        ⇒ e in entry
      occupant.t != occupant.t'
        ⇒ e in checkin + checkout
      lastKey.t != lastKey.t
        ⇒ e in checkin
   }
}
assert NoBadEntry {
  all e: entry {
    o = FD.occupant.(e.pre) [e.room]|
      some o ⇒  e.guest in o
  }
}

```

```
fact NoIntervening {
  all c: checkin {
     c.post = last
     e.room = c.room
     e.guest = c.guest
     }
}

pred consitence {}
run consistence for 4
  but exactly 4 Key,
  exactly 3 Room , 6 Event
check NoBadEntry for 5
  but exactly 5 Key, 10 Time
```

Figure 3.2 – Modèle Alloy de l'Hôtel selon l'idiome Event idiome.

Trace d'exécution

La spécification de l'évolution du système consiste à définir l'ensemble des états par lesquels il est autorisé à transiter durant son exécution. On parle de ***trace d'exécution***. Dans Alloy une trace est définie par une quantification universelle sur les atomes de la signature représentant le temps, et une contrainte selon laquelle une transition entre états doit être déclenchée par l'exécution d'au moins une opération. Ainsi un état du modèle est soit un état initial (à l'instant `first`), soit un état qui résulte de l'exécution d'au moins une opération.

Dès lors, l'analyse des propriétés temporelles dans Alloy se fait sur des traces bornées par le nombre d'atomes de la signature qui représente le temps. Une conséquence est qu'Alloy est moins adapté pour l'analyse des propriétés sur traces infinies telles que la vivacité. Toutefois, il est très efficace pour l'analyse des propriétés de sûreté.

Néanmoins, [Cun14] propose une technique de vérification du comportement sur des traces non bornées contenant une *boucle laso* représentée par une relation entre le dernier état et un état précédent.

Exemple 11. *Dans le modèle de l'Hôtel de la figure 3.1, les états initiaux satisfont le prédicat* `init` *qui spécifie l'ensemble des contraintes de démarrage à savoir : (1) pas de clé pour chaque hôte, (2) pas d'occupant dans les chambres et (3) pour chaque chambre, la clé actuellement codée dans la serrure est également la dernière enregistrée à la réception. La suite de la trace d'exécution [3.1 : l : 2] spécifie le fait qu'un état résulte de l'exécution d'au moins une des opérations* `checkin`*,* `entry` *et* `checkout`*.*

Conditions du cadre

Il faut contrôler l'évolution des champs variables afin d'assurer la cohérence du modèle avec le système réel. Ce contrôle est réalisé par le biais *des conditions du cadre.*

Définition 3.1.17. *Les conditions du cadre définissent l'impact de l'exécution des opérations sur les champs variables.*

Les conditions du cadre peuvent spécifier l'ensemble des champs variables qui ne seront pas modifiés à la suite d'une exécution d'une opération. Dans ce cas, elles sont intégrées dans le corps de l'opération.

Elles peuvent aussi spécifier les opérations autorisées à modifier un champ variable donné, dans cet autre cas elles sont définies en dehors des opérations grâce aux faits (`fact`) supplémentaires. On parle également de conditions du cadre selon le style ***Reiter***.

Exemple 12. *Dans Hôtel, les conditions du cadre sont intégrées dans le corps des opérations. En effet, les prédicats* `noRoomChangeExcept` *[l : 35] et* `noGuestChangeExcept` *[l : 2 de la figure 3.1 spécifient respectivement les modifications des champs variables contenues dans la signature* `Room` *et dans la signature* `Guest`*. Ces prédicats servent à exprimer les conditions du cadre de ces champs. Dans l'opération* `checking` *les conditions du cadre des paramètres variables sont exprimées par la conjonction de ces deux prédicats. Il en ressort qu'à l'exécution d'une opération* `checking[g, r, k ]`*, aucune chambre ne change de clé dans sa serrure et qu'à l'exception de l'hôte g qui ajoute la clé k à son trousseau de clés, les autres hôtes ne modifient pas leur trousseau de clés. le trousseau de clés de l'hôte g ainsi que la clé courante de la chambre r*

On aurait pu utiliser le style *Reiter* . Dans ce cas, pour chaque champ variable, on spécifierait un fait indiquant l'ensemble des opérations autorisées à modifier celui-ci. Ces conditions du cadre s'écriront :

```
fact FCCurrent{
//conditions du cadre currentKey
 all t,t':Time{
   all r:Room{
    r.current.t != r.current.t'
    implies some g:Guest, k: Key |
    Entry[g,r,k]}}}
fact FCGkeys{
//conditions du cadre gKey
all t,t':Time{
  all g:Key {
   g.gkeys.t != g.gkeys.t' implies
   some r:Room, k: Key |
    Checking[g,r,k]}}}

```

```
fact FCLastkey{
//conditions du cadre lastKey
  all t,t':Time{
   FD.occupant.t != FD.occupant.t'
    implies some g: Geust {(checkout[g] or
     some r:Room, k: Key |
         checkin [g,r,k])}}
       }
fact FCOccupant{
//conditions du cadre occupant
all t,t':Time{
 FD.lastkey.t != FD.lastkey.t'
   implies some g:Guest, r:Room, k: Key |
     Checking[g,r,k] }}

```

Aucun style de condition de cadre n'est meilleur que l'autre de façon absolue. Chacun possède des atouts et des limitations. Contrairement au style **Reiter** l'ajout du nouveau champ variable dans le modèle entraîne la modification du corps de toutes les opérations. Néanmoins, si le modèle a beaucoup de champs variables et très peu d'opérations, le style **Reiter** est moins concis que l'autre. Par contre s'il y a beaucoup d'opérations et très peu de champs variables alors le style **Reiter** est plus concis.

3.1.7 Commandes et scope

Commandes

Les vérifications à effectuer par l'analyseur Alloy sont déclarées à l'aide des commandes `run` et `check`.

La commande `run`

La commande `run` instruit Alloy Analyzer de rechercher une instance satisfaisant une contrainte donnée, de telles instances sont appelées *exemples*. Ces contraintes sont généralement exprimées par des prédicats (formules). Pour une définition plus formelle, étant donné un modèle Alloy si :

— ϕ_{model} est la formule du modèle, définie par la conjonction des contraintes implicites liées à la déclaration des signatures et des champs, des contraintes d'héritage et de multiplicité, ainsi que les faits du modèle ;

— ϕ_{run} la formule de la commande `run`.

Alors, la commande instruit Alloy Analyzer de vérifier la satisfiabilité de la formule $\phi_{model} \wedge \phi_{run}$, c'est à dire vérifier s'il existe une structure d'interprétation $\mathcal{M}$ telle que $\mathcal{M} \vDash \phi_{model} \wedge \phi_{run}$.

La commande `run` sert en particulier à vérifier la cohérence d'un modèle, c'est-à-dire que les faits ne sont pas contradictoires. Ainsi, la cohérence est généralement la première commande exécutée sur un modèle pour s'assurer que celui-ci est instanciable.

Commande `check`

La commande `check` instruit Alloy Analyzer de vérifier si une assertion est valide. Plus précisément Alloy Analyzer cherche des instances qui ne satisferaient pas l'assertion considérée. De telles instances sont appelées *contre-exemples*.

En suivant le raisonnement précédent, la commande `check` est formellement spécifiée comme suit. Étant donné un modèle Alloy si ϕ_{model} est la formule du modèle et ϕ_{check} la formule de la commande `check`, alors la commande demande au model checker de vérifier la validité de formule $\phi_{model} \Rightarrow \phi_{check}$, c'est-à-dire $\mathcal{M} \models \phi_{model} \Rightarrow \phi_{check}$ pour toute structure d'interprétation $\mathcal{M}$. Par négation ceci revient à vérifier qu'il n'existe pas de structure d'interprétation $\mathcal{M}$ telle que $\mathcal{M} \models \phi_{model} \wedge \neg\phi_{check}$

Scope

Au-delà des noms de prédicat ou d'assertion, Alloy Analyzer reçoivent également des scopes pour chaque signature du modèle.

Définition 3.1.18 (**Scope**). *Un scope définit le nombre maximal ou exacts d'atomes qui sera considéré pour chaque signature dans une instance d'un modèle. Il est introduit dans la commande par le mot clé `for` en outre, pour signifier que ce scope est exact, on utilise l'expression `but exactly`.*

Remarque 4 (**Scope par défaut**). *Si aucun scope n'est renseigné Alloy Analyzer attribut un scope par défaut d'une valeur égale à trois à tous les signatures top-level [Jac03].*

Exemple 13. *Dans le modèle de l'Hôtel de la figure 3.1, le prédicat `consistence` [l : 21 spécifie la cohérence qui est vérifiée par la commande `run consistence`. Dans l'exécution de cette commande on considérera au plus quatre hôtes, exactement quatre clés et exactement trois chambres. Par ailleurs pour la commande `check NoBadEntry` Alloy Analyzer considérera trois atomes pour chaque signature.*

3.1.8 Vérification

Alloy est basé sur FOR, qui est une logique indécidable. Il n'existe donc pas de méthode de vérification pouvant prouver la validité d'une assertion. Toutefois, Alloy propose une méthode d'analyse appelée *instance finding* [Jac03], [Jac06],[TD06] ,[Jac12] qui consiste à analyser une assertion de façon exhaustive dans un domaine d'instances fini fixé par les scopes des signatures. Si un contre-exemple est trouvé dans ce domaine alors l'assertion n'est pas valide. Cependant, l'absence de contre-exemples dans ce domaine ne signifie pas que l'assertion est vraie, car elle pourrait être fausse dans un domaine plus grand.

Alloy ne s'intéresse donc pas à la preuve de l'assertion, mais à la recherche de contre-exemples qui invalident l'assertion. Cette technique appelée *d'instance finding* utilisée par Alloy est très adaptée pour l'analyse des assertions invalides, car les contre-exemples sont générés rapidement (dans de plus petits domaines). De façon générale, l'efficacité d'Alloy repose sur l'hypothèse du petit scope suivante :

Hypothèse du petit scope *La plupart des bugs ont un petit contre-exemple. Autrement dit, si une assertion est invalide, alors elle possède probablement un petit contre-exemple.* [Jac03]

Une implication de cette hypothèse est que pour une assertion si on l'analyse sur les tout petits modèles, alors il est très probable que l'on trouve un contre-exemple de cette assertion.

```
1  spec ::= module qualName [ [ name,+ ] ] import* paragraph*
2  import ::= open qualName [ [ qualName,+ ] ] [ as name ]
3  paragraph ::= sigDecl | factDecl | funDecl | predDecl
4      | assertDecl | checkCmd
5  sigDecl ::= [ abstract ] [ mult ] sig name,+ [ sigExt ] { varDecl,* } [ block ]
6  sigExt ::= extends qualName | in qualName [ + qualName ]*
7  mult ::= lone | some | one
8  decl ::= [ disj ] name,+ : [ disj ] expr
9  varDecl ::=  decl
10 factDecl ::= fact [ name ] block
11 assertDecl ::= assert [ name ] block
12 funDecl ::= fun name [ [ decl,* ] ] : expr { expr }
13 predDecl ::= pred name [ [ decl,* ] ] block
14 expr ::= const | qualName | @name | this | unOp expr
15     | expr binOp expr | expr arrowOp expr | expr [ expr,* ]
16     | expr [ ! | not ] compareOp expr
17     | expr ( ⇒ | implies ) expr else expr
18     | quant decl,+ blockOrBar | ( expr ) | block
19     | { decl,+ blockOrBar }
20 const ::= none | univ | iden
21 unOp ::= ! | not | no | mult | set | ~ | * | ^
22 binOp ::= || | or | && | and | <=> | iff | ⇒ | implies
23     | & | + | − | ++ | <: | :> | .
24 arrowOp ::= [ mult | set ] → [ mult | set ]
25 compareOp ::= in | =
26 letDecl ::= name = expr
27 block ::= { expr* }
28 blockOrBar ::= block | / expr
29 quant ::= all | no | mult
30 checkCmd ::= check qualName [ scope ]
31 scope ::= for number [ but typescope,+ ] | for typescope,+
32 typescope ::= [ exactly ] number qualName
33 qualName ::= [ this/ ] ( name/ )* name
```

Figure 3.3 – Syntaxe concrète du language Alloy

3.1.9 L'outil Alloy Analyzer

Alloy Analyzer [Jac03] est un outil qui permet de vérifier les spécifications Alloy. Il s'appuie sur un système d'analyses automatiques et bornées. Alloy Analyzer est implémenté au-dessus de KodKod [TJ07] qui est un solveur de contraintes FOR. La résolution consiste en une compilation des contraintes FOR vers une formule de la logique propositionnelle qui sera résolue par un solveur SAT, SAT étant la satisfaction des formules propositionnelles présentée dans le chapitre de la logique à la section 2.1.

Alloy Analyzer est accompagné de plusieurs solveurs SAT dont *MiniSAT, SAT4J, Charff et Berkmin.* Lors de la compilation de FOR vers la logique propositionnelle, plusieurs optimisations sont appliquées. La plus importante est la *suppression de symétries.* Les atomes étant non-interprétables, les modèles Alloy sont naturellement symétriques. En effet, à partir d'une instance, on peut en obtenir plusieurs autres par une simple permutation des atomes.

Ainsi, l'instance renvoyée par une analyse est le représentant d'une classe d'équivalence. Le domaine est réduit en classe d'équivalence ce qui impacte considérablement l'efficacité de l'analyseur. Kodkod calcule ces contraintes de symétries et l'intègre à la contrainte à vérifier.

Alloy Analyzer offre un éditeur pour la spécification et la modélisation et un visualiseur qui affiche les exemples et les contre-exemples sous forme de graphe. Ce visualiseur permet une gestion et une exploitation rapide des instances, ce qui facilite l'analyse des propriétés.

La figure 3.4 présente l'éditeur des spécifications Alloy tandis que la figure 3.5 présente son visualiseur.

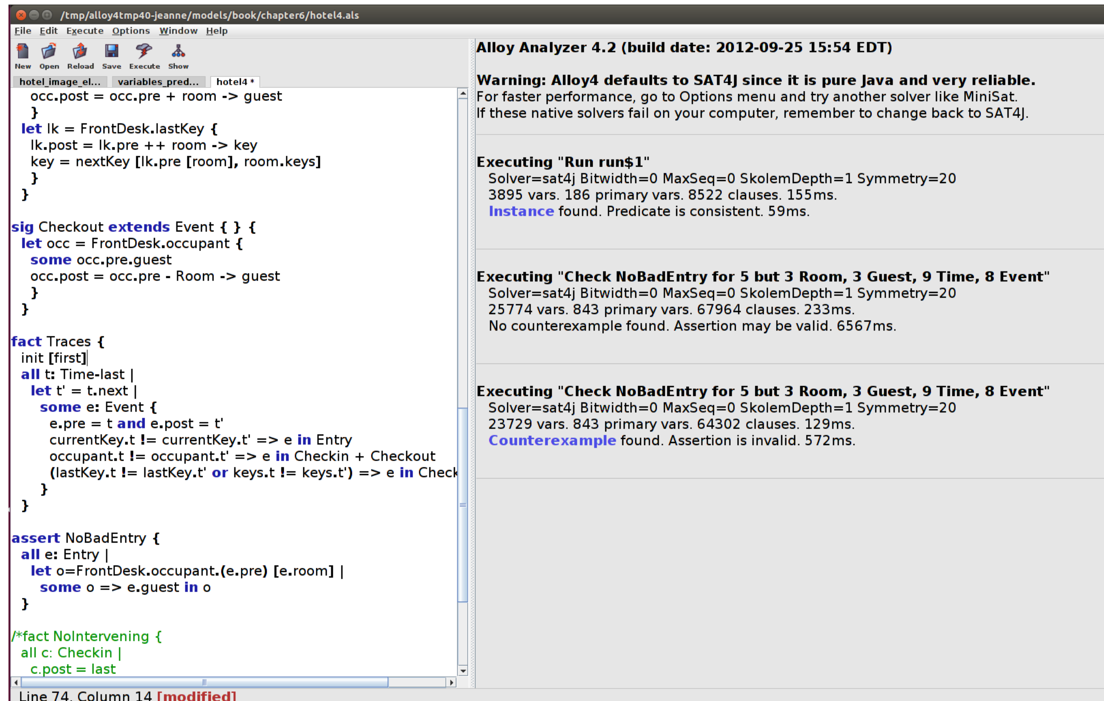


Figure 3.4 – Éditeur des spécifications d'Alloy Analyzer

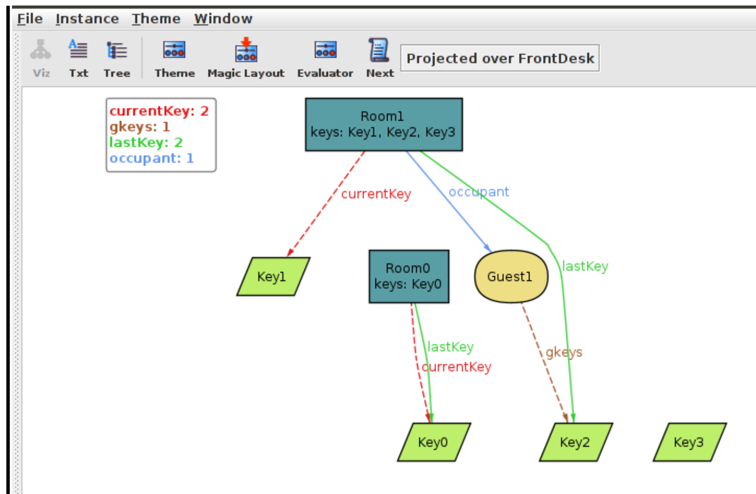


Figure 3.5 – Visualiseur des instances et contre-exemples d'Alloy Analyzer

Les limitations d'Alloy concernent l'analyse des aspects dynamiques des systèmes. En effet, la spécification des traces d'exécution utilise un idiome verbeux et sujet aux erreurs. Par ailleurs cet idiome rend la spécification fastidieuse en obligeant les développeurs à se préoccuper des particularités liées à sa mise en œuvre plutôt que de se concentrer uniquement sur la propriété qu'ils souhaitent réellement vérifier.

La raison pour laquelle Alloy n'est pas adapté pour l'analyse des propriétés dynamiques est la logique utilisée. En effet, la sémantique d'un événement du système est un prédicat FOR ayant éventuellement des paramètres d'entrées et de sorties, le comportement ne peut donc être exprimé de façon native dans Alloy. Par convention, les paramètres de sorties décrivent l'état du système après l'exécution de l'événement avec les paramètres d'entrées. Ainsi, la notion de paramètres variables est purement intentionnelle et ne figure pas dans la sémantique.

Plusieurs travaux de recherche ont été consacrés à l'amélioration des aspects dynamiques ou comportementaux dans Alloy. C'est le cas de DynAlloy [FPB+03, FLPB+05, FGLPA05, FLPGA07] une extension d'Alloy avec des actions, basé sur une logique dynamique pour spécifier le comportement, c'est-à-dire : les événements, la trace d'exécution et les propriétés dynamiques.

3.2 DynAlloy : Une extension dynamique de Alloy

DynAlloy est une extension dynamique d'Alloy dans laquelle les transitions d'états des systèmes sont décrites par des actions. L'introduction d'une sémantique des actions facilite significativement la spécification du comportement dans DynAlloy par rapport à Alloy. DynAlloy permet d'exprimer les propriétés sur les actions au moyen *d'assertions de correction partielle ou triplets de Hoare* [DS90].

DynAlloy est complètement compatible avec les modèles Alloy et étend leur syntaxe avec :

1. des éléments de description des actions ;
2. des programmes qui décrivent le comportement des systèmes. La syntaxe de DynAlloy prend en charge les comportements abstraits, par le biais de choix non-déterministes et d'itérations finies, ainsi que les comportements plus concrets, à l'aide des constructions de la programmation standard telles que les compositions séquentielles et les expressions conditionnelles ;
3. des assertions de correction partielle qui décrivent les propriétés des systèmes.

L'analyse des modèles DynAlloy consiste en une compilation vers des modèles Alloy, à l'aide d'une technique de traduction optimisée [FGLPA05] rendant souvent l'analyse plus efficace que lorsque le comportement est décrit avec l'approche standard d'Alloy.

DynAlloy est accompagné d'un outil d'analyse [RCGB+17, BGG14] qui est une adaptation de Alloy Analyzer permettant la spécification et la vérification complètement automatisée des modèles DynAlloy.

DynAlloy est utilisée dans plusieurs domaines entre autres : dans les langages de programmation orientée objet [GF07, FG06] comme méthode formelle d'analyse ; pour la vérification de l'architecture des systèmes [BG08] dynamiques ; pour l'analyse des propriétés sur les structures de données (les tableaux) [AFM+09].

```
1   open util/ordering[Key] as ko
2   open util/boolean
3   sig Key {}
4   sig Room {keys: set Key }
5   sig Guest {}
6   fact DisjointKeySets {
7     Room<:keys  in Room lone-> Key}
8   fun nextKey [k: Key, ks: set Key]:
9     set Key { min [k.nexts & ks]
10  }
11  pred init[currentKey: Room -> one Key,
12    lastKey: Room -> one Key,
13    occupant: Room -> lone Guest,
14    guestKeys: Guest -> set Key]
15     {
16    no guestKeys and no occupant
17      (all r: Room {
18         r. lastKey = r.currentKey
19         r.currentKey in r.keys) }
20  }
21  act entry[g: Guest, r: Room, k: Key,
22  currentKey: Room -> Key,
23  occupant: Room -> Guest,
24  guestKeys: Guest -> set Key,
25  badEntry: Bool]
26  {
27    pre { k in g.guestKeys
28      (k = r.currentKey or
29      k = nextKey[r.currentKey, r.keys])}
30    post {(k = r.currentKey implies
31      currentKey' = currentKey)
32      (k = nextKey[r.currentKey, r.keys]
33      impliescurrentKey' = currentKey
34          ++ (r->k))
35      badEntry' = (not noBadEntry[g, r,
36        occupant] implies badEntry
37          else False )}
38  }
39  act checkin[g: Guest, r: Room, k: Key,
40  lastKey: Room -> Key,
41  occupant: Room -> Guest,
42  guestKeys: Guest -> Key]
43  {
44    pre {no r.occupant
45      k = nextKey[r.lastKey, r.keys] }
46    post {guestKeys' = guestKeys
47            + (g->k)
48      occupant' = occupant + r->g
49      lastKey' = lastKey ++ r->k}
50  }
51  pred noBadEntry[g: Guest, r: Room,
52      occupant: Room -> Guest]
53  { let o = r.occupant |
54      some o => g in o
55  }
56  act checkout[g: Guest, occupant:
57  Room -> Guest]
58  {
59    pre {some occupant.g }
60    post {occupant' = occupant
61            - Room->g}
62  }
63  act nondet[e: univ, s: univ] {
64    pre {some s }
65    post {e' in s }
66  }
67  program hotelActionsProg[
68  g: Guest, r: Room,
69  k: Key,currentKey: Room -> Key,
70  lastKey: Room -> Key,
71  occupant: Room -> Guest,
72  guestKeys: Guest -> Key,
73  badEntry: Bool]
74  {
75  (nondet[g, Guest];
76   nondet[r, Room]; nondet[k, Key];
77    (checkin[g, r, k, lastKey,
78      occupant, guestKeys]
79    + checkout[g, occupant] +
80    entry[g, r, k, currentKey,
81    occupant, guestKeys, badEntry]))*
82  }
83  program traceProg[g: Guest,r: Room,
84  k: Key, currentKey: Room -> Key,
85  lastKey: Room -> Key,
86  occupant: Room -> Guest,
87  guestKeys: Guest -> Key,
88  badEntry: Bool]
89  {
90   assume (init[currentKey, lastKey,
91        occupant, guestKeys]
92      and badEntry = False );
93    call hotelActionsProg[g, r, k,
94    currentKey, lastKey,
95    occupant, guestKeys, badEntry]
96  }
97  run traceProg for 3 but lurs  4
98
99  assertCorrectness assertNoBadEntry
100 [g: Guest, r: Room, k: Key,
101 currentKey: Room -> Key,
102 lastKey: Room -> Key,
103 occupant: Room -> Guest,
104 guestKeys: Guest -> Key,
105 badEntry: Bool]
106 {
107   pre {init[currentKey, lastKey,
108         occupant, guestKeys]
109       badEntry = False }
```

```
110       badEntry = False }
111     program {
112       call hotelActionsProg[g, r, k,
113       currentKey, lastKey,
114       occupant, guestKeys, badEntry]}
115   post {badEntry' = False }
116 }
117 program noBadEntryInterveningProg
118 [g: Guest, r: Room, k: Key,
119 currentKey: Room → Key,
120 lastKey: Room → Key,
121 occupant: Room → Guest,
122 guestKeys: Guest → Key,
123 badEntry: Bool]
124 {
125   ((nondet[g, Guest]; nondet[r, Room];
126      nondet[k, Key];
127   checkin[g, r, k, lastKey, occupant,
128     guestKeys];
129   entry[g, r, k, currentKey, occupant,
130       guestKeys, badEntry]) +
131   (nondet[g, Guest];checkout[g,occupant])
132   (nondet[g, Guest]; nondet[r, Room];
133     + nondet[k, Key];
134   entry[g, r, k, currentKey, occupant,
135       guestKeys, badEntry]))*
136 }
137 assertCorrectness NoBadEntryIntervening
138 [g: Guest,r: Room, k: Key,
139 currentKey: Room → one Key,
140 lastKey: Room → one Key,
141 occupant: Room → lone Guest,
142 guestKeys: Guest → set Key,
143 badEntry: Bool]
144 {
145   pre { init[currentKey, lastKey, occupant,
146          guestKeys]
147     badEntry = False }
148   program {
149     call noBadEntryInterveningProg[
150     g, r, k, currentKey, lastKey, occupant,
151     guestKeys, badEntry]}
152   post { badEntry' = False }
153 }
154 check assertNoBadEntryIntervening
155   for 3 but 2 Room, 2 Guest lurs exactly 4
```

FIGURE 3.6 – Modèle DynAlloy du système Hôtel, extrait du modèle du site de DynAlloy [PGAF]

3.2.1 Modélisation de la structure dans DynAlloy

La modélisation des aspects structurels (qui ne traitent pas du changement d'état) des systèmes dans DynAlloy est définie exactement de la même façon que dans Alloy. La structure est introduite par la déclaration de signatures et de champs, et les contraintes sur la structure sont introduites par des faits explicites ou implicites (multiplicités). Contrairement à Alloy, seules les relations statiques sont introduites par des champs, les relations variables étant considérées comme des paramètres des actions (cf : § 3.2.2) modélisant les événements qui changent leurs états.

Dans le modèle DynAlloy du système Hôtel présenté dans la figure 3.6, les ensembles des chambres, des hôtes, et des clés sont modélisés par les mêmes signatures `Room`[l : 4], `Guest` [l : 5], `Key` [l :3] du modèle Alloy. Cependant, les relations `currentkey, lastkey, gkeys et occupant` ne sont plus des champs de signatures. Par ailleurs, la signature `FD` n'existe plus (devient superflue) car étant un singleton(constant)ne possédant aucune relation avec les autres signatures du modèle.

3.2.2 Modélisation du comportement dans DynAlloy

Modélisation du temps et des événements

Contrairement à Alloy, le langage DynAlloy ne définit pas de signature particulière pour représenter le temps. Les états du système sont implicitement définis dans les paramètres des *actions* qui spécifient les événements.

Définition 3.2.1 (Action). *Les actions (atomiques) sont des prédicats faisant référence à deux instants de temps consécutifs. Elles sont spécifiées complètement par :*

1. *les paramètres, c'est-à-dire ce à quoi elle s'applique ou ce qu'elle change ;*

2. *les conditions requises pour exécuter l'action appelée **garde** et qui est une contrainte Alloy caractérisant les états dans lesquels l'action est exécutable ;*
3. *l'effet de l'action appelé **postcondition** qui est une contrainte Alloy décrivant les états dans lesquels doit se trouver le système après l'exécution de l'action.*

(Syntaxe des actions). Concrètement les mots-clés `act`, `pre`, `post` et l'opérateur **prime (')** sont ajoutés à la syntaxe d'Alloy, ces mot-clés introduisent respectivement le nom d'une action éventuellement avec des paramètres, sa garde et sa postcondition. Les paramètres des actions définissent l'état dont elles modifient, ou encore les relations auxquelles elles s'appliquent. Les versions primées de ces paramètres font référence à leur état après l'exécution de l'action, elles ne doivent pas être considérées comme de nouveaux paramètres. Autrement dit, l'imbrication de l'opérateur **prime (')** est proscrite.

Exemple 14. *Dans le modèle DynAlloy de l'Hôtel, les événements sont modélisés respectivement par les actions : `checkin`[l :39] avec pour paramètres les relations variables `lastkey`, `gkeys` et `occupant` ; l'action `entry` [l : 21] avec la relation variable `currentkey` comme paramètre et l'action `checkout`[l :57] avec la relation variable `occupant` comme paramètre.*

Dans la postcondition de l'action `checkin` les expressions `lastkey'`, `gkeys'` et `occupant'` décrivent respectivement l'état de ces relations après l'exécution d'une occurrence de l'action `checkin`.

Remarque 5. *Contrairement à Alloy, la propriété à vérifier `noBadEntry` est intégrée directement dans la postcondition de l'action `entry` susceptible de la violer.*

En effet, l'action `entry` vérifie qu'aucune entrée non autorisée dans une chambre ne se produit, en utilisant un paramètre booléen `badEntry` qui indique à la fin de l'exécution d'une occurrence d'une action, si celle-ci a violé la propriété `noBadEntry`.

Ainsi, un contre-exemple de cette propriété est une suite d'exécution d'occurrences d'action qui se termine par une occurrence de l'action `entry` qui a attribué la valeur "`True`" au paramètre `badEntry` dans l'instant suivant son exécution.

Le fait que les relations variables ne soient pas spécifiées par des champs variables a comme conséquence, la complexité et le manque de concision dans l'expression des actions et des contraintes.

Pour exemple de l'hôtel, la spécification des contraintes de démarrage du système dans DynAlloy est plus complexe et moins concise que dans Alloy. En effet, dans le modèle Alloy le prédicat `init` tient sur 5 lignes et possède un paramètre (qui est une relation unaire), tandis que dans le modèle DynAlloy le prédicat `init` [l : 11] tient sur 10 lignes et possède 4 paramètres (qui sont des relations binaires).

Modélisation des exécutions dans DynAlloy

Contrairement à Alloy (qui utilise une technique ad hoc basée sur les faits), le dynamisme dans DynAlloy est décrit convenablement par des constructions d'actions dont les éléments de base sont les actions atomiques et dans lesquelles les comportements complexes sont définis par des compositions d'actions atomiques appelées **programmes**.

Définition 3.2.2 (Programmes). *Un programme est une action complexe obtenue à partir de compositions d'actions élémentaires. Syntaxiquement, un programme est introduit dans le modèle*

par le mot-clé `program` ou `prog` avec éventuellement des paramètres représentant les relations sur lesquelles il s'applique.

DynAlloy offre plusieurs opérateurs de compositions possibles :

- *"**;**" la composition séquentielle : pour deux actions Act_1 et Act_2 , Act_1 ;Act_2 signifie que Act_2 s'exécute à la suite de Act_1 ;*
- *"**+**"le choix non-déterministe : pour deux actions Act_1 et Act_2 , Act_1 + Act_2 signifie que l'une des deux actions choisie au hasard est exécutée ;*
- *"*****" l'itération : pour une action Act, Act* signifie que l'action est exécutée un nombre fini de fois ;*
- *"**?**" le test : pour une action Act, Act ? signifie qu'on vérifie si l'exécution de l'action s'est terminée.*
- *la définition des **hypothèses**, c'est-à-dire des tests d'actions que l'on suppose vrais, elles sont introduites dans le modèle DynAlloy par le mot-clé `assume`.*

Remarque 6. *Les opérateurs de choix non-déterministe et de composition séquentielle sont respectivement équivalents aux expressions d'Alloy standard suivantes :*

$Opr_1 + Opr_2(x,z) \equiv$ `some y | ($Opr_1$(x,y) or $Opr_2$(y,z);`

$Opr_1 ;Opr_2(x,z) \equiv$ `some y | ($Opr_1$(x,y)  and $Opr_2$(y,z).`

Ainsi, DynAlloy diffère réellement de Alloy par l'utilisation des itérations, des tests et des hypothèses.

Occurrences des actions et programmes

La garde de l'action est aussi la condition d'exécution de celle-ci : lorsqu'elle n'est pas satisfaite, l'occurrence de l'action ne peut pas être exécutée. Cependant, la garde d'une action peut être vraie sans qu'aucune occurrence de l'action ne soit exécutée. La condition d'exécution d'un programme est une composition des gardes des actions atomiques qui le compose. L'occurrence d'une action atomique correspond à une référence à son nom avec des paramètres réels, tandis que l'occurrence ou l'exécution d'un programme est introduite par le mot-clé `call` suivie du nom du programme et des paramètres réels.

Exemple 15. *Dans le modèle de l'Hôtel le programme `hotelActionsProg` [l : 67] réalise toutes les actions possibles dans le système. Tout d'abord, on réalise un choix d'hôte, de clé et de chambre de manière non-déterministe. Puis de façon itérative, on fait un choix non-déterministe d'une action élémentaire parmi les actions `checkin`, `checkout`, `entry` et on déclenche une occurrence avec pour paramètres réels les éléments précédemment choisis.*

Assertions de correction partielle

Ce que l'on souhaite exprimer à propos des actions ou programmes, est comment leurs exécutions transforment le système. Pour ce faire, DynAlloy utilise les **assertions de correction partielle ou de pré et post condition** traditionnelle écrites sous la forme :

$$\{Pre\}$$
$$\{programme\}$$
$$\{Post\}$$

Qui signifie que chaque fois que le `programme` est exécuté dans un état qui satisfait Pre, s'il se termine alors, il le fait dans un état qui satisfait $Post$.

Trace d'exécution

Contrairement à Alloy, la trace d'exécution dans DynAlloy n'est pas définie par un fait (`fact`), mais par un programme sous forme d'assertion de correction partielle. La version DynAlloy de la trace définie via l'ordre total (`util/ordering`) sur les instants de temps dans Alloy est donnée par la définition suivante.

Définition 3.2.3 (Trace d'exécution d'un système). *Pour un système ayant n actions* $Act_1, Act_2, ...Act_n$ *dont l'état initial satisfait la contrainte **Init** et qui fonctionne tant que la contrainte **LoopBreak** est satisfaite, la trace d'exécution est spécifiée par le programme :*

$$\textbf{\textit{ExecutionTraceProg}} = \quad \begin{array}{c} \{\textbf{\textit{Init}}\} \\ \texttt{call}\ SystemActionProg \\ \{\textbf{\textit{LoopBreak}}\} \end{array}$$

Où

$$\textbf{\textit{SystemActionProg}} = (Act_1 + Act_2 + ... + Act_n)^*$$

est le programme d'exécution des actions du système, c'est une itération d'actions telle qu'à chaque étape l'action exécutée est choisie de façon non déterministe.

La trace d'exécution du modèle de l'Hôtel est donnée par le programme `traceProg` [l : 84]. Il commence en supposant qu'à l'état initial le prédicat `init` est satisfait et qu'il n'y a pas d'entrée non autorisée. Puis il appelle le programme qui réalise les actions du système. Il n'y a aucune contrainte sur la terminaison de l'exécution des actions, autrement dit le programme s'arrête dans un état quelconque.

Remarque 7. *Contrairement à Alloy dont la technique de définition de la trace mêle les propriétés statiques et les propriétés dynamiques, DynAlloy favorise une séparation claire entre la spécification de ces deux catégories de propriétés.*

Conditions du cadre

Dans DynAlloy, la sémantique des actions impose qu'une action doive modifier toutes les relations qu'elle a en paramètre et que toute relation autre que ces paramètres doit garder sa valeur après l'exécution de l'action. La définition de cette sémantique présente tout de même une petite subtilité. En effet, on suppose que les paramètres n'ayant pas de version primée dans la postcondition de l'action gardent la valeur qu'ils avaient avant l'occurrence de l'action. Les conditions du cadre sont générées systématiquement à partir de ces contraintes imposées par la sémantique des actions.

3.2.3 Assertion des propriétés dynamiques

La spécification d'une propriété dynamique $Prop$ sur un système dont le comportement est défini par la trace d'exécution

{**Init**}
SystemActionProg
{**LoopBreak**}

est donnée par l'assertion suivante :

```
pre {Init and Prop}
program { call SystemActionProg}
post {LoopBreak and Prop}
```

Syntaxiquement, pour spécifier les propriétés dynamiques, un mot-clé `assertCorrectness` est ajouté à la syntaxe d'Alloy. Il introduit le nom de l'assertion suivi éventuellement de paramètres.

Dans le modèle de l'Hôtel une entrée non autorisée a lieu si et seulement si, le paramètre `badEntry = True`. Elle est spécifiée par l'assertion de correction partielle `assertNoBadEntry` qui stipule que toutes les entrées sont autorisées.

3.2.4 Commandes et scope

Pour l'analyse des spécifications, DynAlloy fournit deux commandes similaires aux commandes Alloy `run` et `check`, mais qui s'appliquent aux propriétés dynamiques et programmes.

Commande `run`. L'exécution d'un programme est similaire à celle d'un prédicat, c'est-à-dire qu'elle consiste à rechercher une exécution du programme ou encore un modèle qui satisfait le programme. La commande `run traceProg` recherche une trace d'exécution du système Hôtel.

Commande `check`. La vérification d'une propriété dynamique consiste à recherche une trace d'exécution qui se termine dans un état qui ne satisfait pas la propriété.

La commande `check assertNoBadEntry` recherche une exécution aboutissant à une entrée non autorisée.

Scope. DynAlloy Analyzer utilise "en coulisse" Alloy Analyzer pour exécuter un programme ou vérifier une propriété dynamique. Les commandes d'analyse `run` et `check` sont accompagnées des scopes de signatures, en particulier la longueur des traces est fixée par le scope de la signature représentant le temps.

Bien que DynAlloy ne requiert pas de signature particulière pour représenter le temps, les itérations de programmes sont une source de potentiels comportements infinis. Ainsi, pour une analyse, on doit fournir aux commandes, un scope indiquant le nombre maximal d'itération à prendre en compte dans l'analyse des programme. Ce nombre appelé **déroulement de boucle** est ajouté à la commande par le biais du mot-clé `lurs` (loop unrolls).

3.2.5 Syntaxe abstraite et Sémantique de DynAlloy

Définition 3.2.4 (Syntaxe abstraite de DynAlloy). *La syntaxe de DynAlloy étend celle d'Alloy avec les clauses :*

block : :=. . . | { *block* } *program* { *block* }
program : := $\langle block, block\rangle(\vec{x})$ " atomique action "
|*block ?* | *program + program* | *program ; program* | *program**

Dans la construction des actions atomiques $\vec{x}$ représente le vecteur des paramètres formels. Les gardes sont donc des formules dont les variables libres sont dans $\vec{x}$, tandis que les postconditions contiennent la version primée de ses variables représentée par le vecteur $\vec{x'}$. Tout les paramètres de $\vec{x}$ qui n'ont pas changé garde leur valeurs dans $\vec{x'}$.

DynAlloy étend la sémantique d'Alloy avec celles des actions atomiques, des programmes et des assertions partielles. Le domaine d'interprétation est le même que dans Alloy c'est-à-dire un ensemble d'instances du modèle.

Définition 3.2.5 (Sémantique de DynAlloy). *Pour un domaine $\mathcal{D}$ donné la sémantique de DynAlloy est donnée par une extension de la sémantique d'Alloy avec :*

- $[\![\{Pre\}\ p\ \{Post\}]\!]_{\mathcal{D}}$ *ssi* $[\![Pre]\!]_{\mathcal{D}} \Rightarrow (\forall\langle\mathcal{D},\mathcal{D}'\rangle \in [p] \Rightarrow [\![Post]\!]_{\mathcal{D}'})$ *"assertion de correction partielle" ;*
- $\mathcal{A}(\langle pre, post\rangle)(\overline{x}) = \{\langle\mathcal{D},\mathcal{D}'\rangle : [\![pre]\!]_{\mathcal{D}} \wedge [\![post]\!]_{\mathcal{D}'}\}$ *"action atomique"*

$[\ ] : program \Rightarrow \mathcal{P}(\mathcal{D} \times \mathcal{D})$ *"programme"*

- $[\langle pre, post\rangle] = \mathcal{A}\langle pre, post\rangle$ *;*
- $[p?] = \{\langle\mathcal{D},\mathcal{D}'\rangle : [\![pre]\!]_{\mathcal{D}} \wedge (\mathcal{D} = \mathcal{D}')\}$ *;*
- $[p_1 + p_2] = [p_1] \cup [p_2]$ *;*
- $[p_1; p_2] = [p_1]; [p_2]$ *;*
- $[p*] = [p]^*$.

3.2.6 Vérification dans DynAlloy

Alloy Analyzer est étendu pour permettre l'expression et la vérification des spécifications impliquant les actions, les programmes et les assertions de correction partielle. Si DynAlloy convient mieux qu'Alloy pour la spécification des propriétés dynamiques ou comportementales, l'utilisation de signature représentant le temps et les traces tels que définis dans Alloy a pour avantage de permettre l'automatisation de l'analyse des propriétés dynamiques dans Alloy. Par conséquent, l'analyse complètement automatisée des spécifications DynAlloy, repose principalement sur une compilation des assertions de correction partielle vers des formules FOR. Cette traduction basée sur la version bornée du calcul des *plus faibles préconditions libérales* est présentée dans [FGLPA05].

Cette traduction permet, étant donné une borne n du nombre d'itérations de boucle, de compiler un programme DynAlloy vers un prédicat Alloy. D'après les expériences (comme indiqué dans [FGLPA05] et avec d'autres exemples sur le site Web de DynAlloy) cette traduction offre souvent de meilleures performances par rapport à l'approche standard de définition de traces dans Alloy.

Cette traduction est donnée par la fonction suivante $wlp : program \times formula \rightarrow formula$ tel que :

$$
\begin{aligned}
wlp[a(\vec{y}), \phi] &= (pre[\vec{y'}/\vec{x}] \Rightarrow \ \texttt{all}\ \vec{n}(post[\vec{n}/\vec{x'}, \vec{y'}/\vec{x}] \Rightarrow \phi[\vec{n}/\vec{y'}])) \\
wlp[p?, \phi] &= p \Rightarrow \phi \\
wlp[p_1 + p_2, \phi] &= wlp[p_1, \phi] \wedge wlp[p_2, \phi] \\
wlp[p_1; p_2, \phi] &= wlp[p_1, wlp[p_2, \phi]] \\
wlp[p*, \phi] &= \bigwedge_{i=0}^{\infty} wlp[p^i, \phi]
\end{aligned}
$$

Dans cette définition, pour une formule ϕ, $\phi[y/x]$ est la substitution de toutes les occurrences libres de la variable x par la variable fraîche y dans la formule ϕ. Lorsqu'on fait référence à une action atomique $\langle pre, post\rangle(\vec{x})$ dans un programme, les paramètres formelles $\vec{x}$ sont remplacés par des paramètres réels $\vec{y}$ et les paramètres de la formule ϕ sont contenus dans le vecteur $\vec{y'}$ Ainsi, pour une action atomique a la formule obtenue stipule que : pour tout paramètre réel $\vec{y'}$ de a, si la garde de a est satisfaite pour $\vec{y'}$ alors, pour $\vec{y'}$ la postcondition implique la formule.

Pour les opérateurs de construction de programme, cette fonction donne des formules Alloy dans tous les cas, à l'exception de d'itération, où la formule résultante peut aller à l'infini. Pour obtenir une formule Alloy, on fixe une borne pour le nombre d'itérations.

Comparativement à l'approche standard d'analyse de propriétés dynamiques dans Alloy, DynAlloy présente une limitation importante en termes expressivité.

En effet, DynAlloy traite une classe importante de propriétés dynamiques, appelées *propriétés d'invariance*, mais ne peut traiter de propriétés dynamiques plus générales.

En particulier, les propriétés de vivacité ne peuvent pas être spécifiées au moyen d'assertions de correction partielle dans DynAlloy, alors que dans Alloy de telles propriétés sont facilement spécifiées. Néanmoins, le fait que ces propriétés soient exprimables dans Alloy ne signifie nécessairement pas qu'Alloy Analyzer peut être utilisé pour leur validation, car, seules des traces infinies peuvent constituer de véritables contre-exemples de propriétés de vivacité.

Ainsi, comme Alloy Analyzer et DynAlloy Analyzer opèrent sur des traces finies, ils ne peuvent pas être utilisés pour valider les propriétés de vivacité. Cependant, [Alca] propose une extension d'Alloy avec un mécanisme de détection de violation de propriétés de vivacité plus complexes, consistant à rechercher des traces ayant des boucles infinies.

Dans la section suivante, nous introduisons TLA+, un autre langage formel basé sur une variante de la logique temporelle qui utilise un idiome d'action pour la spécification du changement d'état et des propriétés comportementales ou dynamique sur des systèmes. TLA+ permet une vérification complètement automatisée des propriétés dynamiques de façon plus générale, sur des traces d'exécutions infinies, mais contrairement à Alloy et DynAlloy n'est pas adapté pour la spécification des propriétés structurelles ou statiques.

3.3 TLA+

TLA+ est un langage de spécification formelle [Lam02b],[Lam02a] ,[MC16], très expressif construit pour spécifier les systèmes dynamiques (concurrents, distribués) et leurs propriétés. TLA+ est basée sur le langage TLA [Lam94] (la logique temporelle des actions) une variante de la logique temporelle basée sur les actions, et sur la théorie des ensembles de *Zermelo-Fraenkel [Rat03]* [2] [3].

Par ailleurs, l'ensemble des états atteignables du système est implicitement issu des *actions*, qui sont des prédicats particuliers concernant deux états consécutifs. Ainsi, TLA+ est basé sur un fragment très restreint de la logique TLA (celle permettant d'exprimer les actions) dont l'utilisation simplifie la spécification du comportement des systèmes. Il supporte également les propriétés de la logique du premier ordre.

Les utilisations académiques de TLA+ vont de la vérification de Paxos [HT18],[LPR16], aux spécifications et vérifications des protocoles réseau et web [[WLG08],[JLLV04], [MFdJ97],[FZL10], [Mar10], [LMW11],[Lu13a], [Lu15],[LMW12],[LMW10],[Lu13b],[MC16] ; en passant pas la cohérence de caches [LLOR99], et l'allocation de registres [DGFGL13]. Il a été utilisé par quelques industriels tels que Digital [JLM+03], Compaq [BL02], HP [LSTY01], Intel[BL02] et Microsoft [BDH07]. Son plus grand succès industriel récemment réalisé concerne Amazon Web Services (AWS) [NRZ+15] où il a été utilisé pour le développement de services fondamentaux comme DynamoDB.

TLA+ est accompagné du model checker temporel TLC et possède un langage déclaratif de preuves hiérarchiques accompagné de l'assistant de preuves interactifs TLAPS [Lam02a], [CDL+12]. TLAPS opère par décomposition de preuves complexes en petites étapes et utilise le raffinement d'étapes en étapes jusqu'à l'établissement de la preuve. TLAPS ne permet pas de vérifier la preuve des propriétés dynamiques.

TLC est un model checker temporel qui autorise des vérifications sur des horizons de temps non borné. Il interprète les modèles TLA+ sous forme de machines à états finies. C'est un vérificateur à états explicites et a un contrôle plus approfondi sur la façon dont les états sont explorés. Cependant, expressivité de TLA+ n'est pas pleinement supportée par TLC. Des restrictions sont imposées à la spécification des prédicats d'action et de l'état initial.

3.3.1 Modélisation de la structure dans TLA+

Pour des soucis de clarté et de précision, nous reprendrons dans cette section le modèle Alloy du système de verrouillage des chambres d'hôtel.

En TLA+, la structure d'un modèle est définie par des modules possédant des paramètres pouvant être constants ou variables. Les paramètres constants définis avec le mot-clé ***CONSTANT***, sont ceux qui ne changent pas de valeur quand le système évolue, tandis que les paramètres variables définis par le mot-clé ***VARIABLE*** sont ceux susceptibles d'évoluer. Les opérateurs arithmétiques ne sont pas fournis par défaut dans TLA+, néanmoins, ils sont définis dans le module ***Natural*** que le mot-clé ***EXTENDS*** permet d'inclure dans le modèle :

2. La théorie des ensembles de Zermelo-Fraenkel [FBHL73] est une axiomatisation en logique du premier ordre de la théorie des ensembles telle qu'elle avait été développée par Georg Cantor. Elle a été élaborée par plusieurs mathématiciens dont Ernst Zermelo et Abraham Fraenkel avec l'axiome de choix

3. Une axiome de la théorie des ensembles affirmant « qu'il est possible de construire des ensembles en répétant infiniment une action de choix, même non spécifiée explicitement.

EXTENDS Naturals

CONSTANT KEY, ROOM, GUEST, Keys

ASSUME $KEY \in Nat$

VARIABLE keys, currentKey, last, occupant, gkeys, Room, Guest

$Key \triangleq 0..\ KEY - 1$

TypeInvariant $\triangleq$

$\wedge\ Room \in\ SUBSET\ ROOM \wedge\ Guest \in\ SUBSET\ GUEST$

$\wedge\ keys\ \in [Room \longrightarrow\ SUBSET\ Key] \wedge\ last\ \in [Room\ \longrightarrow\ Key]$

$\wedge\ current\ \in [Room \longrightarrow\ \ Key] \wedge\ occupant\ \in\ [Room \longrightarrow SUBSET\ Guest]$

$\wedge\ gkeys\ \in [Guest\ \Rightarrow\ SUBSET\ Key\ ]$

$\wedge\ \forall\ r\ \in\ Room:\ current\,[r]\ \in\ keys\,[r]$

$\wedge\ \ r_1, r_2\ \in\ Room:\ (keys\,[r_1] \cap\ keys\,[r_2]) \neq\ \{\}\ \Rightarrow\ r_1\ =\ r_2$

Init $\triangleq$

$\wedge\ Room \in\ SUBSET\ ROOM \wedge\ Guest \in\ SUBSET\ GUEST$

$\wedge\ keys\ \in [Room \longrightarrow\ SUBSET\ Key] \wedge\ gkeys =\ [g \in\ Guest \longrightarrow \{\}]$

$\wedge\ current\ \in [Room \longrightarrow\ \ Key] \wedge\ occupant =\ [r \in\ Room \longrightarrow \{\}]$

$\wedge\ \forall\ r\ \in\ Room:\ current\,[r]\ \in\ keys\,[r] \wedge last\ =\ current$

$\wedge\ \ r_1, r_2\ \in\ Room:\ (keys\,[r_1] \cap\ keys\,[r_2] \neq \Rightarrow\ r_1\ =\ r_2$

Checkout $\triangleq$

$\wedge \exists\ \ r\ \in\ \ Room:\ g \in\ occupant\,[g]$

$\wedge\ occupant' =\ [r\ \in\ DOMAIN\ occupant \mapsto occupant\,[r] \setminus \{g\}]$

$\wedge\ UNCHANGED\ \langle keys,\ last,\ currentkey,\ gkeys,\ Geust,\ Room\rangle$

Entry $\triangleq$

$\wedge\ k \in gkeys\,[g]$

$\wedge\ (k\ =\ currentkey\,[r]\,|\{k\}\ =\ nextkey\,[currentKey\,[r]\,,\ keys\,[r]]$

$\wedge\ currentkey'\ =\ [currentkey_EXCEPT\ !\,[r]\ =\ k]$

$\wedge\ UNCHANGED\ \langle keys,\ lastkey,\ occupant,\ gkeys,\ Geust,\ Room\rangle$

Checking $\triangleq$

$\wedge\ occupant\,[r]\ =\ \{\} \wedge \{k\}\ =\ nextKey\,[lastkey\,[r]\,, keys\,[r]]$

$\wedge\ (k\ =\ currentkey\,[r]\,|\{k\}\ =\ nextkey\,[currentKey\,[r]\,,\ keys\,[r]]$

$\wedge\ occupant'\ =\ [Occupant\ EXCEPT\ !\,[r]\ =\ g]$

$\wedge\ gkeys'\ =\ [gkeys\ EXCEPT\ !\,[g]\ =\ @ \cup \{g\}]$

$\wedge\ lastkey'\ =\ [lastkey\ EXCEPT\ !\,[r]\ =\ k]$

$\wedge\ UNCHANGED\ \langle keys,\ currentkey,\ Geust,\ Room\rangle$

Post(g,r,k) $\triangleq\ occupant\,[r]\ = \{g\} \wedge k\ \in\ gkeys\,[r] \wedge lastkey\,[r]\ =\ k \wedge currentkey\,[r] \neq\ k$

Next $\triangleq \exists\ g\ \in\ Geust\ :\ Checkout(g)\ \vee\ \exists\ r\ in\ Room\ ,\ k\ \in\ Key\ :\ Entry(g,r,k)\ \vee\ Checking(g,r,k)$

NoIntervening $\triangleq\ \forall g \in Guest, k\ \in Key, r \in Room : Post(g,r,k) \Longrightarrow Entry(g,r,k)$

Spec $\triangleq Init \wedge \Box\,[Next \wedge TypeInv]_{vs}$

NoBadEntry $\triangleq$

$\Box\,[\forall g \in Guest, r \in Room, k \in Key : Entry(g,r,k) \wedge occupant\,[r] \neq \{\} \Longrightarrow g \in occupant\,[r]]_{vs}$

Figure 3.7 – Modèle TLA+ de l'Hôtel, issue de [MC16]

Dans le système Hôtel, les composants statiques sont modélisés par des paramètres constants. Ainsi, l'ensemble des clés, des hôtes et des chambres, ainsi que l'ensemble des clés qui leur sont attribuées, sont modélisés par les ensembles $ROOM, KEY, GUEST$ *et* $Keys$:

Cependant, contrairement à Alloy, une valeur fixe doit être attribuée à chaque paramètre constant avant le déploiement du modèle sur TLC, ce qui diffère du traitement des champs statiques d'Alloy et peut conduire à des résultats imprévisibles pour un utilisateur habitué d'Alloy.

En effet, bien que les champs statiques dans Alloy ne traitent pas du changement d'état, Alloy Analyzer explore le comportement du système pour toutes les affectations valides de valeurs à ces champs. Tandis que, TLC va analyser le comportement du système uniquement pour l'affectation des valeurs aux constantes, effectuée lors du déploiement.

Pour l'exemple du système Hôtel, l'analyse du modèle Alloy avec un scope (nombre maximal d'atomes de signature, mais pas exact) de trois, génère un contre-exemple constitué d'une chambre, de deux hôtes et de trois clés. En revanche, dans TLC une configuration avec trois atomes par signature ne génère pas de contre-exemple. En réalité, le modèle impose à chaque chambre d'avoir au moins une clé (clé encodée dans sa serrure), il faudrait au moins deux clés supplémentaires (pour avoir un enregistrement successif de deux hôtes dans la même chambre) pour obtenir un contre-exemple.

Pour obtenir un comportement similaire à celui d'Alloy, on définit deux univers constants $GUEST$ et $ROOM$ contenant respectivement l'ensemble des hôtes et des chambres et $Guest, Room$ sont spécifiés comme des paramètres variables de leurs univers respectifs. En outre, les champs variables *keys, current, last , occupant, gkeys*, sont spécifiées par les paramètres variables équivalentes ;

Les paramètres constants étant bornés dans TLC, TLA+ fournit le mot-clé ***ASSUME*** qui fixe des contraintes sur les valeurs attribuées aux paramètres constants. Il n'a aucun impact sur le modèle mais, instruit TLC de vérifier si l'affectation des valeurs aux paramètres constants est valide (pris dans l'ensemble défini par la borne). Par exemple, l'ensemble des clés du système est borné et totalement ordonné. Si on considère le paramètre constant KEY représentant le nombre de clés valides, ce dernier doit être un entier naturel non-nul.

Pour modéliser le comportement du système,

3.3.2 Terminologie

Définition 3.3.1. ***Une fonction d'état*** *est une expression ordinaire (une macro) du premier ordre (ne contient ni variables primées, ni opérateurs temporels) pouvant contenir des constants et des variables.*

Définition 3.3.2. *Un* ***prédicat d'état*** *est une fonction d'état à valeurs booléennes.*

Définition 3.3.3. ***Un invariant*** Inv *d'une spécification* Spec *est un prédicat d'état tel que* $Spec \Rightarrow \Box\ Inv.$ *où* $\Box$ *représente le connecteur temporel* **G***.*

Définition 3.3.4. ***Une action*** *est un prédicat particulier concernant deux états consécutifs et contenant à la fois des variables primées et/ou non primées.*

Définition 3.3.5. ***Le bégaiement*** *ou* ***stuttering*** *en anglais, est une action particulière, dont l'exécution ne fait rien et ne change rien.*

Définition 3.3.6. *Une variable v est de type $\mathcal{T}$ dans une spécification Spec si $v \in \mathcal{T}$ est un invariant de Spec : $Spec \Rightarrow \Box(\ v \in \mathcal{T})$.*

TLA+ n'est pas typé. Cependant, en réalité, il est considéré comme étant de bonne pratique de définir un **invariant de type :** ***TypeInvariant*** sur les paramètres variables apparaissant dans tous les états atteignables du système. Par ailleurs, TLA+ ne prend pas nativement en charge les opérations de la logique relationnelle tel que la fermeture transitive, ainsi les contraintes structurelles devant être respectées dans tout les états atteignables du système sont converties en contraintes TLA+ et greffées à l'invariant de type. Dans Hôtel, le fait que les ensembles de clés affectées aux différentes chambres soient disjoints est également inclus dans ***TypeInvariant.***

Pour un ensemble ***A, SUBSET A*** est l'ensemble des parties de ***A***. Dans l'Hôtel, le paramètre **Room** est un sous-ensemble de l'univers *ROOM* et *Guest* un sous-ensemble de *GUEST*. Le paramètre *keys* est une relation qui associe un ensemble de clés valides à une chambre, tandis que *last* est une fonction entre une chambre et l'ensemble des clés valides qui lui sont attribuées. Il en est de même pour le reste de paramètres variables du système *Hotel*.

3.3.3 Modélisation du comportement dans TLA+

La spécification d'un système est en réalité la description de ses différents comportements autorisés, c'est-à-dire les actions qu'il peut effectuer durant son exécution. Formellement, un comportement est une séquence d'états, un état étant une affectation de valeurs aux paramètres variables du système. Une étape désigne une succession entre deux états consécutifs, en particulier, une étape de bégaiement signifie qu'entre deux instants consécutifs, les paramètres variables ont conservé leur valeur.

TLC ne prend pas en charge toute l'expressivité de TLA+. Les spécifications TLA+ vérifiables par TLC sont exprimées par des formules sous le format suivant :

$$\textbf{Init} \wedge \Box\,[\textbf{Next}]_v\, s \wedge \textbf{Fairness} \text{ avec :}$$

- *Init* un prédicat d'état restreignant l'état initial ;
- *Next* fixant des contraintes sur les étapes d'évolution valides du système en utilisant les actions ;
- *Fairness* contenant des contraintes d'équité.

L'expression $\Box\,[Next]_v\, s$ signifie qu'à tout moment soit *Next* se produit, soit une étape de bégaiement est effectuée avec $v' = v$. v étant le vecteur des paramètres variables du modèle.

Le modèle Hôtel ne possède aucune contrainte d'équité. Dans l'état initial défini par le prédicat d'état Init de la figure 3.7, les chambres sont supposées vides et pour chacune d'elles, la dernière clé enregistrée à la réception est également celle actuellement encodée dans sa serrure. Les hôtes n'ont pas de clé . Autrement dit, les paramètres `occupant` ainsi que `gkeys` sont vides et `last = currentKey`. Par ailleurs, les chambres (Room) et les hôtes (Guest) sont inclus dans leurs univers respectif ROOM et GUEST, et les ensembles des clés attribuées aux différentes chambres sont tous disjoints.

Le comportement est modélisé par le prédicat d'action *Act* de la même figure, qui est une disjonction de la spécifications des actions *Checkin, Entry et Checkout*. Autrement dit une étape d'évolution du système correspond au tir d'une ou de plusieurs de ces actions. Une action est dite ***activée*** seulement dans les états où son tir permettrait de franchir une étape. Dans l'Hôtel, l'action *Checkout* est activée pour une hôte si et seulement si elle occupe au moins une chambre dans l'hôtel, tandis que pour un triplet d'hôte, de chambre et de clé, l'action *Checkin* est activée dans un état si et seulement si dans celui-ci la chambre est libre et la clé de l'hôte est valide.

La spécification d'une action commence par la description de sa **garde**, suivie de l'effet de l'action sur le système ou encore **postcondition**. *Le tir d'une action équivaut à la satisfaction de sa garde.* Dans l'Hôtel, la garde de l'action *Checkout* est que l'hôte concerné doit être un occupant d'une chambre. La spécification complète de cette action est donnée par l'expression $Checkout$ dans figure 3.7

Les conditions du cadre

Dans TLA+, la définition des conditions du cadre est facilitée par l'utilisation des sucres syntaxiques :

- — étant donnée une variable x, UNCHANGED x est une abréviation de $x' = \ x$;
- — pour une fonction f : $f' = [fEXCEPT![x] = e]$ signifie que f reste inchangée à l'exception de sa valeur en x, qui est mise à jour avec e. Le caractère @ dénote la valeur de $f[x]$ à l'instant précédent.

3.3.4 Spécification des propriétés dynamiques

TLA+ permet la spécification d'un sous-ensemble de formules LTL contenant les connecteurs temporels globally ($\Box$) et eventually ($\Diamond$). Cependant la classe des formules TLC compatibles (vérifiables par TLC) est beaucoup plus restreinte [[Lam02b] p.236] comme le montre la BNF suivante : $Formule ::= P \mid \Box P \mid \Diamond P \mid \Box\,[A]_{var}\,s \mid \Box(P \Rightarrow \Diamond Q) \mid \Box\Diamond\langle A\rangle_{var} s\ \Diamond\Box\,[A]_{var}\,s \mid Fairness$. Où $P\ et\ Q$ sont des prédicats d'état, *Fairness* une condition d'équité et A un prédicat d'actions. Par ailleurs, les formules de cette classe ne doivent pas contenir des quantificateurs $\exists$.

La propriété **NoBadEntry** de Hôtel est spécifiée dans la figure 3.7 par une formule sous la forme : $\Box\,[A]_{var}\,s$.

TLC prend en entrée le module issu de la spécification TLA+ et un fichier de configuration. Ce fichier contient entre autres le nom de la spécification et le nom de la propriété à vérifier introduites respectivement par ***SPECIFICATION*** et ***PROPERTY***. En absence des contraintes d'équité la spécification peut être renseignée par ***INIT*** et ***NEXT*** indiquant ainsi l'état initial et les étapes d'évolution du système.

Le module Hôtel ne possédant pas de contrainte d'équité, son fichier de configuration pourrait contenir soit

SPECIFICATION Spec
PROPERTY NoBadEntry.
ou

INIT *Init*
NEXT *Act*
PROPERTY NoBadEntry.

L' invariant de Type est identifié par ***INVARIANT***. Lorsque l'instruction PROPERTY est utilisée, l'invariant n'est vérifiable que si la formule est sous la forme $\Box P$ *P étant un prédicat d'état.*

3.3.5 Vérification avec TLC

TLC opère par génération de tous les comportements qui satisfont la spécification. Il possède deux modes de vérification :

le model checking , dans ce mode, un modèle correspond à une affectation des valeurs aux différents paramètres de la spécification. La vérification consiste à rechercher tous les états atteignables, c'est-à-dire ceux qui peuvent faire partir d'un comportement satisfaisant la spécification.

la simulation où TLC génère arbitrairement des comportements, mais sans être nécessairement exhaustif dans la recherche des états atteignables.

Il peut y avoir des contraintes supplémentaires sur l'évolution du système n'ayant pas été capturées dans le prédicat d'action **Next**, dans ce cas elles sont prises en compte dans **CONSTRAINTS**. Formellement, pour une contrainte *Constr* donnée, il s'agit de demander à TLC de vérifier :
$Init \wedge \Box [Next]_{vars} \wedge \Box Constr$ au lieu de $Init \wedge \Box [Next]_{vars}$.

La vérification de NoBadEntry dans TLC génère un contre-exemple qui met en évidence un comportement non valide lorsque l'enregistrement d'une hôte n'est pas suivi de son entrée dans sa chambre.

Pour corriger ce contre-exemple, le modèle dynamique est restreint, en éliminant systématiquement tous les comportements dans lesquels une action *checkin[g,r,k]* n'est pas suiviE d'une action *Entry[g,r,k].*

Dans le modèle Alloy, cette contrainte est spécifiée par *NotIntervening.* Cependant, cette formule n'est pas valide dans TLA+ car elle requiert l'utilisation des variables doublement primées. La solution proposée dans [[MC16]] est de modifier les conditions de tir de l'action *Entry* de sorte qu'elle soit activée dans les états où la postcondition de *Checkin* est respectée. Ces restrictions sont spécifiées dans ***CONSTRAINT*** *NotIntervening* [3.7.

Pour une vérification avec une borne de 4 pour les paramètres constants GUEST, ROOM, KEY, le fichier de configuration TLC contient :

SPECIFICATION Spec

PROPERTY NoBadEntry

CONSTRAINT NotIntervening

CONSTANT Key = 4

GUEST = $\{g1, g2, g3, g4\}$

ROOM = $\{r1, r2, r3, r4\}$

3.4 Conclusion

Dans ce chapitre, nous avons présenté des méthodes formelles permettant la spécification et la vérification des propriétés sur des systèmes dynamiques dotés de propriétés structurelles.

Alloy et TLA+ sont deux langages de spécification formels qui doivent leur popularité grandissante à leur simplicité et leur flexibilité, ainsi qu'à l'efficacité des analyseurs qui les accompagnent, respectivement Alloy Analyzer et TLC. Chacun de ces langages possède des limitations dans la spécification et l'analyse de systèmes riches en propriétés statiques et dynamiques. En poursuivant l'objectif de faciliter la spécification et l'analyse de tels systèmes, nous introduisons dans le chapitre

suivant, (chapitre 4) le langage Electrum qui s'inspire d'Alloy et propose une solution permettant la spécification à la fois des propriétés structurelles et des propriétés temporelles et, autorise une vérification sur des traces tant bornées que non bornées.

Chapitre 4

Electrum

4.1 Introduction

Dans le chapitre précédent, nous avons présenté deux langages de spécification et de vérification formelle très populaires.

Alloy, a un langage de spécification très expressif et son expressivité est entièrement prise en charge par Alloy Analyzer. Il est convenable pour l'analyse des propriétés structurelles, mais le dynamisme doit être explicitement modélisé par l'utilisateur. Alloy Analyzer autorise les vérifications sur des traces bornées uniquement, ce qui entrave la correction de la vérification des propriétés dynamiques.

DynAlloy est une extension dynamique d'Alloy avec des actions conçu pour faciliter la spécification et l'analyse des propriétés dynamiques dans Alloy. Cependant, la syntaxe d'action impose un idiome de spécification particulier ce qui détériore l'expressivité d'Alloy et la flexibilité dont sont habitués ses utilisateurs . En particulier, les propriétés de vivacité, qui comprennent une large classe de propriétés dynamiques ne sont pas exprimables dans DynAlloy.

Bien que l'analyse des spécifications DynAlloy des propriétés dynamiques soit plus efficace que les spécifications Alloy équivalentes, elle reste tout de même réalisée sur des traces d'exécutions bornées comme dans Alloy standard.

TLA+ est un langage plus expressif qu'Alloy, basé sur TLA (la logique temporelle des actions) et qui prend aussi en charge la spécification des propriétés FO contenant éventuellement des opérateurs de la théorie des ensembles et relationnelles, à l'exception de la fermeture transitive. Il est très adapté pour la spécification des propriétés dynamiques.

Cependant, TLC ne prend pas en charge toute l'expressivité de TLA+. En effet, les propriétés de TLA+ vérifiables par TLC sont limitées. Selon [MC16] le dynamisme explicite défini par Alloy est plus expressif que le sous-langage de TLA+ vérifiable par TLC. L'assistant de preuve TLAPS permet de réaliser des preuves manuelles des propriétés statiques à l'aide des opérations de décompositions et de raffinements.

Chacun de ces langages présente des limitations dans la spécification et l'analyse d'un type de propriété (propriété structurelle ou propriété comportementale).

Cependant, l'analyse de la plupart des systèmes réels requiert une spécification et une vérification des propriétés tant structurelles que comportementales. Tel est par exemple le cas de l'analyse des algorithmes de calcul distribué dont les propriétés comportementales doivent généralement être vérifiées pour des topologies de réseau caractérisées par des propriétés structurelles spécifiques. Les travaux [BCT18] que nous avons présentés à FMCAD qui portent sur l'analyse du protocole Chord, et bien d'autres travaux tels que [Zav19, Zav11, Zav12, Zav15a, Zav15b, Zav17] [MLW11],[KDN11],[Gon01],[BG07] sont des exemples d'analyses de tels systèmes.

De façon générale, nous appelons ***configurations*** ces composants de l'état du système à l'exemple de la topologie, qui ne traite pas du changement d'état.

Définition 4.1.1. ***Systèmes dynamiques avec des configurations riches***
Les systèmes dynamiques avec des configurations riches sont une classe de systèmes caractérisée par les exigences suivantes [MBC+16] :

E1 : *une distinction claire entre les configurations et le comportement du système ;*

E2 : *les contraintes sur les configurations sont exprimées par des propriétés structurelles tels que l'héritage, les relations complexes entre les entités, ou les propriétés d'accessibilité.*

E3 : *la spécification du comportement se fait dans un style déclaratif, selon différents idiomes à l'instar de l'idiome prédicat [Jac12] , [Jac06] , [Jac03] ;*

E4 : *il existe des propriétés de sûreté et de vivacité à vérifier sur le système.*

Exemple 16. ***Le système de l'hôtel***
Le système de l'hôtel présenté au chapitre 3 section 3.1 est un exemple typique de cette classe de systèmes.

En effet, dans la modélisation réalisée en Alloy, nous avons fourni une abstraction du système dans laquelle l'ensemble des clés possibles est représenté par un ensemble sur lequel est défini un ordre total. À chaque chambre, est attribué un sous-ensemble disjoint de clés, de sorte que la clé actuellement encodée dans la serrure de la chambre soit toujours la plus petite des clés du sous-ensemble.

Ainsi, l'ensemble des clés, des hôtes et des chambres avec une affectation valide des clés constituent une configuration du système. Par ailleurs, ces configurations se démarquent des composantes dynamiques (par exemple, la clé actuellement codée dans la serrure, l'hôte affecté à une chambre et enregistré à la réception), elles restent constantes lorsque le système évolue dans le temps, le système satisfait donc l'exigence (E1).

Comme présenté ci-dessus, une configuration valide n'est pas arbitraire, mais caractérisée par un ensemble précis de contraintes, l'exigence (E2) est ainsi satisfaite.

Les composantes dynamiques du système progressent à mesure que les hôtes s'enregistrent à la réception, entrent dans une chambre avec une nouvelle clé, mettant à jour la clé actuellement encodée dans la serrure de la chambre, ainsi que lorsqu'ils quittent l'hôtel. Ces opérations peuvent facilement être spécifiées de manière déclarative, en s'appuyant par exemple sur les gardes et postconditions définies dans [Jac03], d'où la satisfaction de l'exigence (E3).

Dans cette spécification, une propriété fondamentale de sûreté (un hôte entre dans une chambre seulement s'il est enregistré comme occupant de cette chambre) doit être préservée, l'exigence (E4) est ainsi satisfaite. Il faut noter que dans cet exemple particulier, il n'y a pas de propriété de vivacité.

Toutefois, à l'exemple d'Alloy et de TLA+, la plupart des langages de spécification ainsi que les outils d'analyse qui les accompagnent, excellent soit dans l'analyse des propriétés structurelles soit dans l'analyse des propriétés dynamiques, limitant ainsi leur capacité à analyser la classe des systèmes définie ci-dessus. Par exemple, la plupart des model checkers standards [CCGR00],[CCG+02],[CCD+14] ne fonctionnent bien qu'avec des configurations fixes.

Ainsi, pour analyser les systèmes de cette classe, le langage de spécification doit être suffisamment riche et flexible pour permettre la spécification à la fois des propriétés structurelles et temporelles. En outre, ce langage devrait être accompagné d'un cadre efficace, permettant la vérification automatique par model checking des propriétés temporelles désirées, pour chaque configuration.

Dans ce chapitre nous, présentons le langage de spécification Electrum, une extension dynamique d'Alloy avec les opérateurs LTL et les variables primées de TLA+, qui préserve toute l'expressivité, la simplicité et la flexibilité d'Alloy. Il facilite la spécification des propriétés tant structurelles que comportementales des systèmes présentant les quatre exigences susmentionnées et offre un cadre efficace permettant la vérification automatique sur des traces tant bornées que non bornées.

4.2 Spécification formelle

Electrum [MBC+16] est un cadre de méthodes formelles légères offrant :(1) un langage de spécification basé sur FOLTL et inspiré d'Alloy [Jac12], (2) deux techniques de model checking : model checking borné (BMC) basé SAT et le model checking non borné (UMC) basé sur une compilation des modèles vers les model checkers nuXmv [CCD+14] et NuSMV [CCGR00],[CCG+02].

La syntaxe d'Electrum étend celle d'Alloy à l'aide des signatures et champs variables, des expressions primées ainsi que des connecteurs LTL.

Définition 4.2.1. ***Signatures et champs variables***
Les signatures et champs variables sont ceux dont les valeurs peuvent évoluer sur une trace d'exécution donnée. Ils sont introduits par le mot-clé `var`.

Pour chaque signature top-level variable `s`, nous définissons implicitement une signature statique `Hull_s`, appelée enveloppe de `s`, dont la sémantique est l'ensemble des atomes qui peuplent `s` durant toute l'exécution du système. On note donc qu'à chaque instant, la sémantique d'une signature variable est un sous-ensemble de son enveloppe. Dans une commande, le scope d'une signature top-level variable représente en fait le scope de son enveloppe. Ceci est en effet nécessaire puisque l'analyse nécessite de considérer un domaine de taille bornée.

Exemple 17. *Dans la figure 4.1, toutes les signatures sont statiques, tandis que les champs `currentKey`, `lastkey`, `gkeys`, `occupant` sont variables.*

Définition 4.2.2. ***Multiplicité des éléments variables***
Pour les éléments variables, les restrictions induites par leurs multiplicités sont appliquées globalement dans le temps

Par exemple, deux signatures variables qui en étendent une autre sont nécessairement disjointes dans chaque état d'une trace, mais peuvent s'échanger des atomes d'un état à l'autre. Par exemple, soient A une signature abstraite, B et C des signatures variables qui étendent A.

```
abstract sig A{}
var sig B extends A{}
var sig C extends A{}
```

Si on suppose que $A = \{A_0, A_1 A_2 A_3\}$. On peut avoir : $B_t = \{A_0, A_1, A_2\}$ et $C_t = \{A_3\}$ à un instant t donné et $B_{t+k} = \{A_3, A_1\}$ et $C_{t+k} = \{A_2, A_0\}$ (il y a un échange d'atome entre B et C) à un autre instant $t + k, \quad k > 0$.

Ainsi Electrum permet une distinction claire entre les configurations (constructions statiques) des systèmes et leur évolution (constructions variables) (exigence E1).

Tous ces éléments sont constitués de formules logiques qui tiennent leur expressivité d'Alloy d'une part, de l'utilisation des expressions primées et des connecteurs de la logique temporelle d'autre part.

C'est pour toutes ces raisons qu'Electrum est très adapté pour la spécification et la vérification des systèmes dynamiques ayant des configurations riches (exigence E2) et offre une flexibilité dans la définition des opérations (exigence E3) .

Définition 4.2.3. ***Expression primée***
L'opérateur «'» exprime la valeur d'un terme à l'instant suivant.

Exemple 18. *L'expression `g.gkeys'= g.gkeys + k` dans l'opération `checkin` de la figure 4.1[l : 54] signifie qu'à l'instant suivant la clé `k` sera ajoutée à la liste des clés de l'hôte `g`.*

Connecteurs LTL

Les aspects dynamiques des systèmes sont spécifiés de façon simple et directe grâce aux connecteurs LTL. Par exemple dans Alloy, les traces sont spécifiées par un mécanisme ad hoc qui consiste à quantifier globalement sur l'ensemble des instants. Dans DynAlloy, ces traces sont spécifiées par des programmes (assertions de correction partielle) qui sont des constructions complexes d'actions définis par des itérations, des choix non-déterministes, des compositions séquentielle et des tests d'actions. Ces traces sont spécifiées facilement à l'aide du connecteur Electrum `always` équivalent au connecteur LTL **Globally**. Plus généralement le dynamisme est exprimée grâce aux les connecteurs LTL :

- `always` équivalent au connecteur LTL **Globally** signifie qu'à chaque instant une formule est vraie ;
- `eventually` équivalent au connecteur LTL **Futur** signifie que dans le futur une formule sera vraie ;
- `after` équivalent au connecteur LTL **Next** signifie qu'une formule est vraie à l'instant suivant.

Exemple 19 (Élection du leader). *Le problème de l'élection du leader consiste à choisir en un temps fini, un unique processus leader (chef) parmi un ensemble de processus dans un réseau structuré en anneau unidirectionnel. Chaque processus a un unique identifiant (Id) et communique dans le réseau avec un successeur (son voisin de gauche lorsqu'on parcourt l'anneau dans le sens des aiguilles d'une montre) de plus, à chaque instant, il se prononce sur sa participation à l'élection. Le leader est le processus ayant le plus grand identifiant.*

L'algorithme de la sélection du leader fonctionne de la façon suivante. Initialement, aucun processus ne participe à l'élection et un processus quelconque démarre une élection en créant un message contenant son Id qu'il envoie à son successeur. Tout processus qui reçoit ou envoie un message signale qu'il est participant.

À la réception d'un message le processus compare l'Id du message à son propre Id. Si cet Id est plus grand que le sien alors, il transfère le message à son successeur. Si l'Id est plus petit, si le processus n'est pas encore participant (n'a pas encore envoyé de messages d'élection) alors, il substitut l'Id du message par le sien et envoie le message à son successeur, sinon il détruit le message.

Le processus élu est celui qui recevra son propre identifiant. Il envoie ensuite un message de fin d'élection contenant l'Id du leader (son Id) à son successeur et se déclare non participant. À la réception d'un tel message chaque processus se déclare non participant et enregistre l'Id du message comme celui du leader.

La propriété à vérifier sur ce système est que ***l'algorithme finira par sélectionner un unique leader****.*

Pour la spécification de cette propriété en Electrum, on considère que : (1) chaque processus en plus de son identifiant (Id) possède une variable tampon (ToSend) contenant le meilleur candidat pour le poste de leader à sa connaissance ; (2) à l'état initial, chaque processus se considère comme le meilleur candidat au poste de leader, il initialise donc ToSend avec son Id ; (3) lorsqu'un processus reçoit un Id plus grand que ToSend, avant de transférer le message, il met à jour ToSend avec cet Id ; (4) à chaque instant on détermine l'ensemble `elected` *des potentiels leaders, un processus étant dans* `elected` *s'il reçoit sont identifiant c'est-à-dire si son Id est dans Tosend dans l'instant suivant alors qu'il n'y est pas à l'instant courant.*

La propriété est finalement spécifiée par une assertion qui stipule que si des processus participent à l'élection, alors l'ensemble de potentiels leaders contiendra dans le futur un unique leader. Les expressions Electrum équivalentes au calcul de potentiels leaders et à l'assertion sont les suivantes.

```
always (elected' = { p: Process | (after ( p.id in p.toSend ))
      and (p.id not in p.toSend) })
assert BadLiveness { some Process ⇒ eventually some elected  }
```

- `always` *exprime les instants de façon globale ;*
- `elected'` *exprime l'ensemble des potentiels leaders à l'instant suivant ;*
- `after` *vérifie si oui ou non le processus a gardé son identifiant à l'instant suivant ;*
  `eventually` *exprime le fait que dans le futur un processus sera leader*

Remarque 8. *Bien que les opérateurs «'» et* `after` *donnent des valeurs relatives à l'instant suivant, «'» exprime la valeur d'un terme à l'instant suivant (la valeur reste un terme), tandis que* `after` *indique la satisfaction d'une formule à l'instant suivant.*

```
1  /module hotel
2  open util/ordering[Key]
3  sig Key {}
4  sig Room {
5    keys: set Key,
6    var currentKey: one keys
7  }
8  fact DisjointKeySets {
9    Room<:keys   in   Room lone→ Key
10 }
11 one sig FD {
12   var lastKey: Room → lone Key,
13   var occupant: Room → Guest
14 }
15 sig Guest {
16   var gkeys: set Key
17 }
18 fun nextKey [k: Key, ks: set Key]: set Key {
19   min [nexts[k] & ks]
20 }
21 pred init {
22   no Guest.gkeys
23   no FD.occupant
24   all r: Room |
25     r.(FD.lastKey) = r.currentKey
26 }
27 pred entry [g: Guest, r: Room, k: Key] {
28   k in g.gkeys
29   ((k = r.currentKey and r.currentKey'
30     = r.currentKey) or
31   (k = nextKey[r.currentKey, r.keys]
32     and r.currentKey' = k))
33   noRoomChangeExcept [r]
34   noGuestChangeExcept [none]
35   noFDChange }
36 pred noFDChange {
37   FD.lastKey = FD.lastKey'
38   FD.occupant = FD.occupant'
39 }
40 pred noRoomChangeExcept [rs: set Room] {
41   all r: Room - rs | r.currentKey = r.currentKey'
42 }
43 pred noGuestChangeExcept [gs: set Guest] {
44   all g: Guest - gs | g.gkeys = g.gkeys'
45 }
46 pred checkout [g: Guest] {
47   some FD.occupant.g
48   FD.occupant' = FD.occupant - Room →g
49   FD.lastKey = FD.lastKey'
50   noRoomChangeExcept [none]
51   noGuestChangeExcept [none]{
52 }
53 pred checkin [g: Guest, r: Room, k: Key] {
54   g.gkeys' = g.gkeys + k
55   no r.(FD.occupant)
56   FD.occupant' = FD.occupant + r → g
57   FD.lastKey' = FD.lastKey ++ r → k
58   k = nextKey [r.(FD.lastKey), r.keys]
59   noRoomChangeExcept [none]
60   noGuestChangeExcept [g]
61 }
62 fact traces {
63   init
64   always {
65     some g: Guest, r: Room, k: Key |
66       entry [g, r, k]
67       or checkin [g, r, k]
68       or checkout [g]}
69 }
70 pred NoBadEntry {
71   always { all r: Room, g: Guest, k: Key |
72     entry [g, r, k] and some r.(FD.occupant)
73      ⇒ g in r.(FD.occupant) }
74   }
75
76 assert BadSafety {
77   NoBadEntry
78 }
79 check BadSafety for 2 but exactly 4 Key,
80 exactly 2 Room, exactly 2 Guest,
81 check BadSafety for 3 but exactly 3 Key
82 check BadSafety for 3 but exactly 4 Key
```

FIGURE 4.1 – Example du modèle Alloy de l'hôtel traduit en un modèle Electrum

La figure 4.1 présente le modèle Electrum du système de verrouillage des portes des hôtels. L'ensemble des chambres, des hôtes et des clés sont spécifiés par les signatures statiques `Room`, `Guest` et `Key` respectivement, traduisant le fait que leurs valuations restent fixes durant l'évolution du système. En outre, la réception est spécifiée par la signature singleton statique `FD`, signifiant qu'il y a une seule réception qui ne change pas quand le système évolue. Le champ statique `keys` représente l'ensemble des clés attribuées à chaque chambre ; cette distribution est contrainte d'être une partition à l'aide du fait DisjointKeys. Les autres champs, représentant la clé actuellement encodée dans la serrure d'une chambre (`currentKey`), le registre des hôtes (`occupant`) et les dernières clés attribuées aux chambres au niveau de la réception (`lastKey`) sont tous variables. Les opérations du système sont définies sous forme de prédicats qui font référence à l'instant suivant par des expressions "primées", par ailleurs dans l'expression des propriétés à vérifier les aspects temporels

sont introduits par des connecteurs LTL. Enfin, le fait (`fact traces`) restreint l'ensemble des états atteignables du système.

4.3 Vérification Formelle

4.3.1 Syntaxe des commandes

Tout comme Alloy les propriétés à vérifier sont spécifiées par des assertions introduites par le mot-clé `assert`. Les commandes de vérification sont intégrées dans le fichier de spécification. Cependant, contrairement à Alloy ces assertions contiennent des connecteurs LTL permettant ainsi la vérification des propriétés temporelles riches (Exigence E4). À titre d'exemple, l'assertion `BadSafety` de la figure 4.1 contient le connecteur temporel ***Globally*** (`always` en Electrum) pour signifier que la propriété est vérifiée à chaque instant. Les instructions d'exécution dans Electrum consistent en deux commandes `run` et `check`, restreintes par des `scopes`

Commande `run`

La commande `run` permet de produire une instance satisfaisant une contrainte donnée. Formellement la commande `run` est spécifiée de la façon suivante. Étant donné un modèle Electrum, si ϕ_{model} représente la formule du modèle (ensemble des faits du modèle et des contraintes implicites de multiplicité) et ϕ_{run} la formule ou contrainte de la commande `run`. La commande instruit le model checker de vérifier la satisfiabilité de la formule $\varphi_{model} \wedge \phi_{run}$. Autrement dit le model checker doit trouver une trace d'exécution π telle que $\pi \vDash \phi_{model} \wedge \varphi_{run}$.

Commande `check`

La commande `check` prend en paramètre une assertion ainsi que le scope des signatures (statiques et variables) et cherche à prouver la validité de l'assertion. Plus précisément le model checker cherche un contre-exemple qui ne satisferait pas la contrainte considérée. Tout comme la commande `run`, la commande `check` est formellement spécifiée comme suit.

Étant donné un modèle Electrum, si ϕ_{model} est la formule du modèle et ϕ_{check} la formule ou contrainte de la commande `check`, alors la commande `check` instruit le model checker de vérifier la validité de formule $\phi_{model} \Rightarrow \phi_{check}$ (toute trace de ϕ_{model} satisfait ϕ_{check}). Avec un raisonnement par négation ceci revient à vérifier la satisfiabilité de la formule $\phi_{model} \wedge \neg\phi_{check}$ (il existe une trace de ϕ_{model} qui ne satisfait pas ϕ_{check}). Autrement dit le model checker est chargé de trouver une trace π tel que : $\pi \vDash \phi_{model} \wedge \neg\phi_{check}$. Si le model checker échoue, la conclusion sur la validité de la formule dépend de la technique de vérification utilisée. Dans le cas d'une analyse bornée, la conclusion est identique à celle donnée dans Alloy, c'est-à-dire que l'absence du contre-exemple ne signifie pas la validité de la formule, tandis que pour une analyse non bornée, on conclut que la formule est valide.

Exemple 20. *L'assertion `BadSafety` du modèle Hôtel (figure 4.1) n'est pas valide, un contre-exemple de la commande `check BadSafety for 2 but exactly 4 Key, exactly 2 Room, exactly 2 Guest` est donné par le scénario constitué des figures[4.2, 4.3 , 4.5 , 4.6].*

Ces figures présentent une succession de cinq étapes, chacune décrivant l'état des chambres, des hôtes et des clés. Un carré représente une chambre et contient le sous ensemble de clés qui lui

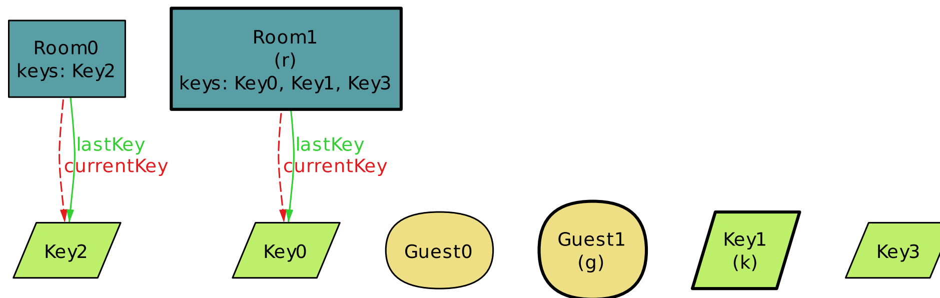


FIGURE 4.2 – État initial

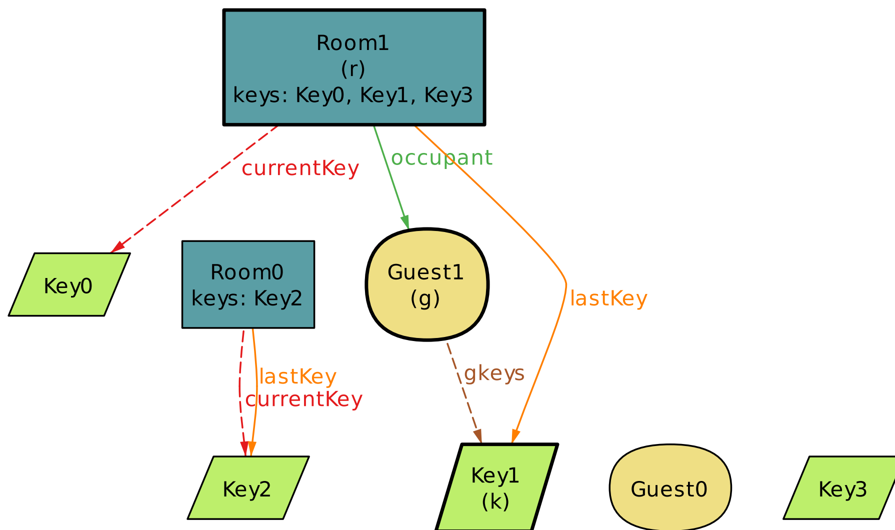


FIGURE 4.3 – L'hôte `Guest1` est enregistré à la réception dans la chambre `Room1`

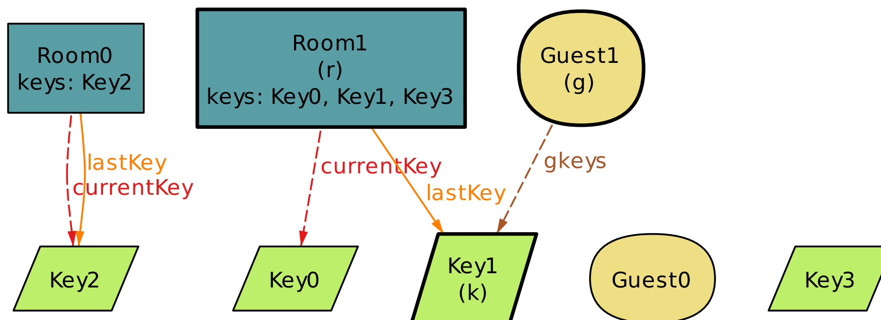


FIGURE 4.4 – L'hôte `Guest1` quitte l'hôtel avec sa clé

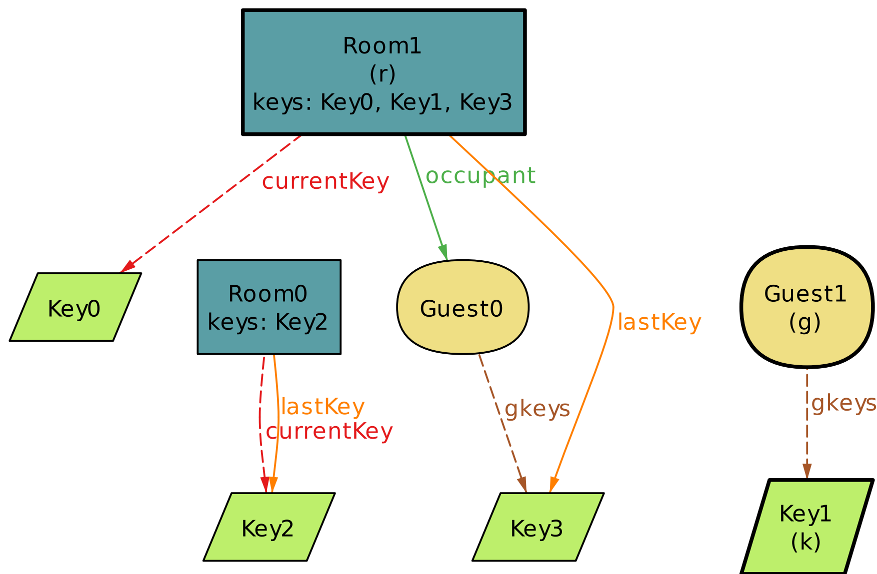


FIGURE 4.5 – L'hôte `Guest0` est enregistré à la réception dans la chambre `Room1`

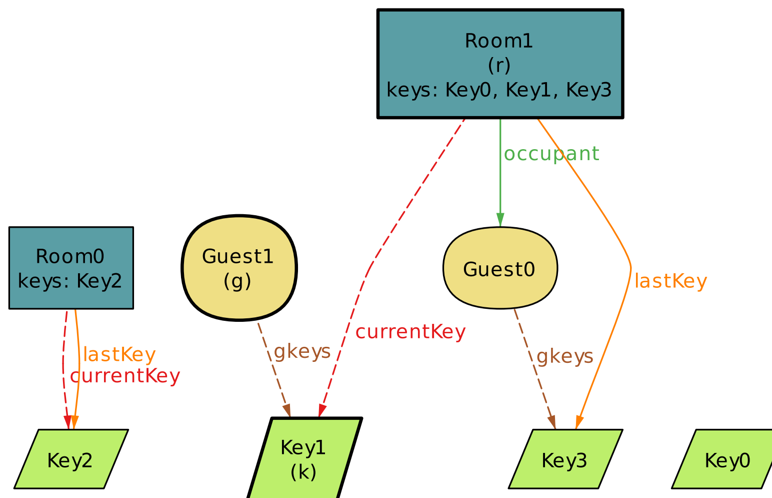


FIGURE 4.6 – L'hôte `Guest1` peut entrer dans la chambre `Room1`

est attribué. Un losange représente une clé tandis qu'un cercle représente un hôte. L'ensemble des figures géométriques constitue les configurations du système. Les champs variables sont représentés par des flèches étiquetées par leur nom

Dans cet exemple particulier, la configuration est l'ensemble constitué de chambres disponibles (Room0 et Room1), des clés (de Key0 à Key3, avec l'ordre total naturel) et d'hôtes (Guest0 et Geust1). Par ailleurs, les clés Key0 à Key3 sont attribuées à la chambre Room1 et Key4 est attribuée

à la chambre Room2.

Dans l'état initial, les hôtes n'ont pas de clé et la clé actuellement encodée dans la serrure d'une chambre est aussi la dernière clé enregistrée pour cette chambre à la réception.

Suite à l'opération `checkin[Guest1, Room1, Key2]` *l'hôte* $Guest1$ *est enregistré comme l'occupant de la chambre* $Room1$ *et reçoit la clé* $Key2$*. Puis avec une opération* `checkout[Guest1]` *il quitte l'hôtel en gardant sa clé sans avoir eu au préalable un accès à sa chambre.*

Avec l'opération `checkin[Geust0, Room1, Key3]` *l'hôte* $Guest0$ *est enregistré dans la chambre* $Room1$ *avec la clé* $Key3$*. Cependant, la serrure de la chambre* $Room1$ *n'ayant pas été encodée par la clé de l'hôte* $Guest1$*, ce dernier peut encore accéder à la chambre* $Room1$ *avec la clé* $Key2$ *malgré qu'il ne soit plus enregistré comme occupant. Ainsi, cette situation viole l'assertion* `Badsafety`*.*

La syntaxe concrète d'Electrum est présentée à la figure 4.7.

4.3.2 Techniques de vérification

Bien qu'Electrum prenne en charge à la fois la modélisation des systèmes et la spécification des propriétés à vérifier sur ces modèles, il n'est utile que s'il est accompagné de techniques de vérification efficaces. Pour répondre à la double nature du problème posé, deux approches distinctes de vérification des spécifications Electrum ont été explorées : une technique bornée et une autre non bornée. La différence entre les deux techniques de vérification est la manière dont le temps est géré. La technique bornée est héritée d'Alloy et la technique non bornée opère sur les traces infinies.

Model Checking borné

La sémantique bornée de FOLTL peut directement être compilée en Alloy lui-même, comme décrit dans [EJT04], en introduisant explicitement une signature représentant le temps et sur laquelle un ordre total est imposé afin de représenter les traces. Une boucle est représentée par une relation entre le dernier instant et un instant antérieur [Cun14]. Le model checking borné d'Electrum est implémenté en utilisant cet encodage alternatif, et déployé comme une version d'Alloy Analyzer, afin de minimiser le temps de prise en main par les habitués d'Alloy. Cependant, contrairement à Alloy ce model checker ne génère pas un seul contre-exemple, mais il offre à l'utilisateur la possibilité d'itérer sur l'ensemble des contre-exemples possibles (dans une borne fixée) qui violent la propriété spécifiée, fournissant ainsi à l'utilisateur une perception plus large des problèmes potentiels dans la spécification.

Model Checking non borné

Cette technique repose sur un encodage direct dans l'outil nuXmv [CCD+14], qui implémente une variété d'algorithmes performants pour le model checking non borné. L'algorithme utilisé dans Electrum est celui nommé k-liveness. Son prédécesseur NuSMV peut aussi être utilisé, cependant, il paraît beaucoup moins efficace que nuXmv pour les exemples étudiés.

nuXmv s'attend à une description d'un système de transition et une formule à vérifier sur ce dernier, le modèle SMV généré selon le schéma de traduction suivant.

```
1  spec ::= module qualName [ [ name,+ ] ] import* paragraph*
2  import ::= open qualName [ [ qualName,+ ] ] [ as name ]
3  paragraph ::= sigDecl | factDecl | funDecl | predDecl
4      | assertDecl | checkCmd
5  sigDecl ::= [ var ] [ abstract ] [ mult ] sig name,+
6                      [ sigExt ] { varDecl,* } [ block ]
7  sigExt ::= extends qualName | in qualName [ + qualName ]*
8  mult ::= lone | some | one
9  decl ::= [ disj ] name,+ : [ disj ] expr
10 varDecl ::= [ var ] decl
11 factDecl ::= fact [ name ] block
12 assertDecl ::= assert [ name ] block
13 funDecl ::= fun name [ [ decl,* ] ] : expr { expr }
14 predDecl ::= pred name [ [ decl,* ] ] block
15 expr ::= const | qualName | @name | this | unOp expr
16     | expr binOp expr | expr arrowOp expr | expr [ expr,* ]
17     | expr [ ! | not ] compareOp expr
18     | expr ( ⇒ | implies ) expr else expr
19     | quant decl,+ blockOrBar | ( expr ) | block
20     | { decl,+ blockOrBar } | expr'
21 const ::= none | univ | iden
22 unOp ::= ! | not | no | mult | set | ~ | * | ^
23     | eventually | always | after
24 binOp ::= || | or | && | and | <=> | iff | ⇒ | implies
25     | & | + | - | ++ | <: | :> | . | until| weakly until | release
26 arrowOp ::= [ mult | set ] → [ mult | set ]
27 compareOp ::= in | =
28 letDecl ::= name = expr
29 block ::= { expr* }
30 blockOrBar ::= block | | expr
31 quant ::= all | no | mult
32 checkCmd ::= check qualName [ scope ]
33 scope ::= for number [ but typescope,+ ] | for typescope,+
34 typescope ::= [ exactly ] number qualName
35 qualName ::= [ this/ ] ( name/ )* name
```

FIGURE 4.7 – Syntaxe concrète du langage Electrum, les ajouts à la syntaxe Alloy sont soulignées

Traduction des champs et des signatures. En fonction de leur statut (static ou variable), les signatures et les champs génèrent des variables booléennes simples ou "Frozen", une variable étant "frozen" si sa valeur est fixée de façon aléatoire à l'état initial, mais reste constante durant l'exécution ;

1. Pour une signature A dont le scope dans la commande à vérifier vaut n.
   - Si A est statique, alors chaque atome de sa sémantique est traduit en une Frozen VAR, pour tous les atomes de A l'ensemble des Frozen Variables $(A\$_i)_{i=1}^{i=n}$ est donc généré ;
   - Si A est variable alors deux groupes distincts de variables booléennes sont générées. Chaque atome du $Hull$ (sémantique d'une signature variable) de A est traduit en une

Frozen VAR, l'ensemble $(_Hull_A\$_i)_{i=1}^{i=n}$ est généré pour les atomes du $Hull$ de A.

- A chaque instant pour un atome $_Hull_A\$_i$ une variable $_Hull_A\$_i_is_in_A$ permet d'indiquer si l'atome $A\$_i$ appartient à la sémantique de A à cet instant. Pour tout les atomes du $Hull$ de A l'ensemble des variables simples $(_Hull_A\$_i_is_in_A)_{i=1}^{i=n}$ est généré

2. Pour une relation $r : A \rightarrow B$ donnée, on définit $r_A\$_i_B\$_j$ une variable booléenne indiquant si $(A\$_i, B\$_j)$ appartient à r. Dans une analyse où les scopes des signatures A et B sont respectivement n et m l'ensemble des variables booléennes suivant est généré.

$$\prod_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}} r_A\$_i_B\$_j$$

Si r est statique alors on a un ensemble de FROZENVAR, sinon on a de simples variables booléennes. Le même schéma de traduction s'applique aux relations d'arité supérieur à deux.

Traduction de reste des éléments du modèle La traduction des différents événements du modèle constitue le système de transition du modèle SMV, il est définit pas des section TRANS et les contraintes sur l'état initial sont spécifiées par une formule constituant la section INIT du modèle SMV. Les faits explicites, les contraintes de typage, de multiplicité et de hiérarchie des signatures et des champs sont traduits en termes invariant, indiqué dans une ou plusieurs des sections INVAR du fichier SMV. La commande Electrum est traduit en une formule LTL sur les variables du modèle SMV et constitue la section LTLSPEC du fichier SMV.

Exemple 21. *Génération du modèle SMV Nous donnons ici un exemple de fichier SMV d'un modèle Electrum. Dans ce petit modèle il n'y a pas d'événement et nous vérifions simplement la cohérence du modèle. Nous mettons en avant la génération des variables.*

Modèle Electrum

```
var sig A{}
var sig B extends A{}
sig C{
  var r:B;
  s: A
}
run{}
```

Modèle SMV

MODULE MAIN

FROZENVAR

//relation variable s :C -> A

s_C$3___Hull_A$3 : boolean ;

s_C$3___Hull_A$2 : boolean ;

s_C$3___Hull_A$1 : boolean ;

s_C$2___Hull_A$3 : boolean ;

s_C$2___Hull_A$2 : boolean ;

s_C$2___Hull_A$1 : boolean ;

s_C$1___Hull_A$3 : boolean ;

s_C$1___Hull_A$2 : boolean ;

s_C$1___Hull_A$1 : boolean ;

//Hull var sig A

_Hull_A$3 : boolean ;

_Hull_A$2 : boolean ;

_Hull_A$1 : boolean ;

//sig C

C$3 : boolean ;

C$2 : boolean ;

C$1 : boolean ;

VAR

//relation variable r :C -> B (extends A)

r_C$3___Hull_A$3 : boolean ;

r_C$3___Hull_A$2 : boolean ;

r_C$3___Hull_A$1 : boolean ;

r_C$2___Hull_A$3 : boolean ;

r_C$2___Hull_A$2 : boolean ;

r_C$2___Hull_A$1 : boolean ;

r_C$1___Hull_A$3 : boolean ;

r_C$1___Hull_A$2 : boolean ;

r_C$1___Hull_A$1 : boolean ;

// var B extends A

_Hull_A$3_is_in_B : boolean ;

_Hull_A$2_is_in_B : boolean ;

_Hull_A$1_is_in_B : boolean ;

//atomes A

_Hull_A$2_is_in_A : boolean ;

_Hull_A$3_is_in_A : boolean ;

_Hull_A$1_is_in_A : boolean ;

// invariant

INVAR

Il est constitué des formules relatives à :

- *la multiplicité des relations r et s (`one`)*
- *hiérarchisation entre les signatures A et B*
- *l'inclusion de la sémantique A dans son Hull.*

//état initial

INIT

TRUE

//système de transition

TRANS

TRUE

//propriété à vérifier

LTLSPEC

TRUE

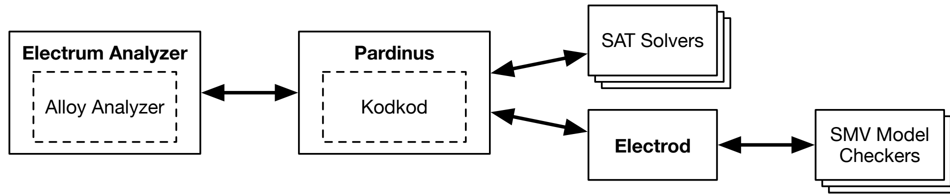


FIGURE 4.8 – Architecture de Electrum Analyzer

4.3.3 L'outil Electrum Analyzer

Electrum Analyzer [Alca] est un outil libre qui permet de vérifier les spécifications Electrum. Il réalise l'analyse automatique bornée et non bornée. Le fichier exécutable ainsi que le manuel d'installation sont disponibles sur le gits et les sites d'Electrum [1] [2] [3].

Les traces d'instances et de contre-exemples sont présentées à l'utilisateur sur une adaptation du visualiseur d'Alloy Analyzer qui possède une fonctionnalité supplémentaire de navigation dans les traces infinies à travers des états avec des boucles.

L'architecture d'Electrum Analyzer est présentée à la figure 4.8. Electrum Analyzer repose sur Pardinus [4] [Alcb] une extension temporelle autonome de Kodkod chargée de l'analyse des modèles issue de Electrum Analyzer. Si l'analyse est réalisée par le model checking borné, alors Pardinus se comporte comme Kodkod, c'est-à-dire qu'il compile l'analyse en un problème SAT et le confie à un solveur. Lorsqu'il s'agit d'une analyse par model checking non bornée le problème est confié à Electrod qui se charge de la compilation vers SMV.

Electrod convertit la couche relationnelle vers la logique simple de premier ordre, puis vers LTL (propositions) en fonction des évaluations possibles des ensembles (signatures) et des relations (champs) déduites du modèle Electrum et des bornes des domaines du premier ordre définies par les scopes.

Par ailleurs, Electrod permet une homogénéisation de l'analyse des traces générées à la fois par le model checking borné et le model checking non borné.

4.4 Sémantique

4.4.1 Noyau d'Electrum

Conformément à l'approche [[Jac12] App.C], la définition de la sémantique d'Electrum est donnée par un sous-langage simplifié nommé *Electrum Kernel*, basé uniquement sur des formules FOLTL présentée dans le chapitre 2. La syntaxe abstraite d'Electrum Kernel est présentée dans la figure 4.9. En ce qui concerne les contraintes et les expressions relationnelles, la traduction d'Electrum vers Electrum Kernel est relativement simple et suit celle d'Alloy.

1. https ://github.com/haslab/Electrum
2. ttps ://forge.onera.fr/projects/electrum,
3. ttps ://forge.onera.fr/projects/electrum
4. Il est disponible sur le git https ://github.com/haslab/Pardinus

```
formula ::= not formula / after formula /  always formula
          / eventually formula /  formula until formula
          / formula and formula /  formula release formula
          / formula or formula / formula implies formula
          / formula weakly until formula/  term in term
          / all decl | formula / some decl | formula
          / one decl | formula /  term = term  /  term + term /term − term
   term ::= x ∈ Var / r ∈ R / ^term  / ~term
          / term & term / term × term / term . term
          / term' / { decl+ | formula }
   decl ::= x : term
```

Figure 4.9 – Electrum Kernel Syntaxe abstraite d'Electrum.

4.4.2 Traduction d'Electrum vers Electrum Kernel

Les signatures et champs dans Electrum sont traduits comme des relations dans le Noyau Electrum *Kernel*. En particulier, les signatures sont des relations unaires. On suppose également l'existence d'un ensemble Var de variables du premier-ordre. Les informations supplémentaires déclarées dans les signatures telles que l'héritage, la multiplicité, les faits locaux, ainsi que le fait que les signatures et champs soient statiques doivent être spécifiées par des formules dans Electrum Kernel. Ci-dessus un exemple qui illustre cette traduction :

Exemple 22.

```
abstract sig A{r:some A}
var sig B,C extends A{}

```

La spécification Electrum Kernel contient :

Relation : *A(1), B(1), C(1), r(2) ;*

Signature non variable *: le fait que A ne soit pas une signature variable est exprimée par la formule :*`always A = A';`

Hiérarchie entre signatures *: le fait que B et C soient des extensions de la signature A est présentée par la formule* `always ((A = B + C ) and not( B & C))`

Le type et les multiplicités des relations : *le type et la multiplicité de la relation r s'expriment par les formules.*

```
always r in A → A
always all a: A |some a.r
```

Les faits locaux sont encadrés par le connecteur `always` *après la traduction, forçant ainsi le* fait *a être satisfait à chaque instant*

4.4.3 Sémantique d'Electrum en FOLTL

L'essence de la compilation d'Electrum Kernel vers FOLTL est de se débarrasser des termes relationnels pour garder uniquement les formules FOLTL. Cette approche standard [EJT04] est la

même que celle utilisée pour la compilation des termes Alloy vers FOR. De ce fait, la principale opération consiste à remplacer les relations d'appartenance et d'inclusion dans les formules Electrum bien formées, par des sous-formules FOLTL équivalentes. Ainsi, la sémantique (notée $[\![.]\!]$) repose sur une fonction $[.]$, qui étant donnés un tuple de variables et un terme, renvoie une formule indiquant que le premier est membre du second.

Dans la suite par soucis de lisibilité, les tuples sont représentés par des vecteurs, leur concaténation par une juxtaposition et leur longueur par $|\,.\,|$.

Définition 4.4.1. *La sémantique des formules d'Electrum Kernel bien formées dans FOLTL est définie dans la section 2.6 du chapitre 2*

4.5 Conclusion

Electrum est un langage très adapté pour la spécification et la vérification des systèmes dynamiques riches en structure. Il offre deux outils de model checking : UBM et BMC. L'analyse avec BMC basé-SAT est très efficace pour la vérification des propriétés de sûreté(Safety) et la taille des modèles vérifiables est la même que dans Alloy. L'analyse UBM basée LTL est très adaptée pour la vérification des propriétés de type vivacité (liveness). Cependant, la méthode de compilation d'Electrum vers SMV est assez naïve, par conséquent, le système de transition du modèle SMV généré, ainsi que la propriété à vérifier sur le modèle sont très grands, ce qui explique que la vérification ne soit possible que pour de tout petits modèles de système. Par ailleurs, contrairement à DynAlloy, la spécification des propriétés dynamiques est simple et directe, grâce à l'utilisation des connecteurs LTL.

Electrum Analyzer est un prototype qui pourra être étendu par une amélioration des performances de l'analyse avec UBM, ainsi que la facilitation de la spécification du comportement en déchargeant d'avantage les utilisateurs des tâches ardues ou sujettes aux erreurs.

$$
\begin{aligned}
[\![\texttt{not}\ f]\!] &= \neg[\![f]\!] \\
[\![\texttt{after}\ f]\!] &= \mathbf{X}[\![f]\!] \\
[\![\texttt{always}\ f]\!] &= \mathbf{G}[\![f]\!] \\
[\![\texttt{eventually} f]\!] &= \mathbf{F}[\![f]\!] \\
[\![f_1\ \texttt{until}\ f_2]\!] &= [\![f_1]\!]\ \mathbf{U}\ [\![f_2]\!] \\
[\![f_1\ \texttt{and}\ f_2]\!] &= [\![f_1]\!] \wedge [\![f_2]\!] \\
[\![t_1\ \texttt{in}\ t_2]\!] &= \forall\vec{x}[\vec{x} \in t_1] \Rightarrow [\vec{x} \in t_2] \\
&\quad \text{avec } \vec{x} \text{ sont des variables } \textit{fraîche} \\
[\![\texttt{all}\ x : t\ \texttt{|}\ f]\!] &= \forall x.[x \in t] \Rightarrow [\![f]\!]
\end{aligned}
$$

$$
\begin{aligned}
[x \in y] &= x \doteq y \\
[x \in s] &= \exists y : s.x \doteq y \\
[\vec{x} \in r] &= \overline{r}(\vec{x})
\end{aligned}
$$

$$
\begin{aligned}
[\langle x_1, x_2\rangle \in \hat{\ } t] &= \text{il existe } y_1, \ldots, y_n \text{ tel que} \\
&\qquad y_1 \doteq x_1 \wedge y_n \doteq x_2 \wedge \bigwedge_{i<n} [\langle y_i, y_{i+1}\rangle \in t] \\
[\langle x_1, x_2\rangle \in\sim t] &= [\langle x_2, x_1\rangle \in t] \\
[\vec{x} \in t_1\ \texttt{\&}\ t_2] &= [\vec{x} \in t_1] \wedge [\vec{x} \in t_2] \\
[\vec{x} \in t_1 \times t_2] &= [\vec{y} \in t_1] \wedge [\vec{z} \in t_2] \\
&\quad \text{avec } \vec{x} = \vec{y}\vec{z}; \\
[\vec{x} \in t_1.t_2] &= \exists u.[\vec{y}u \in t_1] \wedge [u\vec{z} \in t_2] \\
&\quad \text{avec } \vec{x} = \vec{y}\vec{z}, \text{ où } u \text{ est une variable } \textit{fraîche} \\
[\vec{x} \in t'] &= \mathbf{X}[\vec{x} \in t] \\
[\vec{x} \in \{\vec{y} : \vec{t} \mid f\}] &= \Big(\bigwedge_{1\leqslant i\leqslant|\vec{x}|} [x_i \in t_i]\Big) \wedge [\![f\{\vec{y} \leftarrow \vec{x}\}]\!] \\
&\quad \text{où } f\{\vec{y} \leftarrow \vec{x}\} \text{ est la substitution usuelle.}
\end{aligned}
$$

Figure 4.10 – Traduction d'Electrum Kernel vers FOLTL (cf.. Def. Chap : 2 section 2.6).

Chapitre 5

Protocole de recherche distribuée Chord

Le protocole Chord [SMK+01] [SMLN+03] [LNBK02] est un algorithme peer-to-peer qui réalise une table de hachage distribuée [RFH+01, RM06, RGR+04] sur le support d'un réseau de recouvrement structuré en un anneau virtuel de nœuds. Il combine les structures de données assez complexes, la communication asynchrone, le parallélisme et de la tolérance aux pannes, ce qui fait de lui une cible idéale pour la spécification et la vérification formelle. Nous nous intéressons en particulier ici à la partie relative à la maintenance.

Chord est un système dynamique avec des propriétés structurelles riches, cependant des précédents travaux d'analyse réalisés sur lui concernent soit la vérification automatique de ses propriétés structurelles soit une preuve manuelle de l'ensemble de ses propriétés. Dans cette thèse, nous réaliserons une analyse automatique de l'intégralité des propriétés de Chord.

5.1 Contexte

Les réseaux *peer-to-peer* sont des systèmes distribués sans organisation hiérarchique ni contrôle centralisé. En chaque nœud du système, les logiciels fournissent les mêmes fonctionnalités. Les systèmes peer-to-peer offrent plusieurs avantages [SMLN+03] :

1. la coordination du réseau ne requiert aucun investissement supplémentaire dans les matériels haute performance ;
2. l'augmentation de la robustesse face à certains types de pannes les rend bien adaptés pour le stockage de données à long terme ;
3. la redondance de stockage et la cohérence des données ;
4. la recherche distribuée ;
5. l'anonymat et la sélection des données ;
6. l'authentification et la hiérarchisation des noms.

.

Fort de ces caractéristiques, les systèmes peer-to-peer sont un moyen très efficace pour l'agrégation et l'utilisation des ressources de calcul et de stockage distribués dans un réseau afin d'optimiser leur rentabilité.

Cependant, le goulot d'étranglement dans ces systèmes est la localisation efficace des données stockées dans le réseau. Les nœuds vont et viennent de façon arbitraire, ce mouvement engendre une migration des données rendant leur localisation complexe et leur valeur éventuellement non cohérente. Une solution à ces problèmes fut la conception d'une structure de données particulière appelée tables de hachage distribué [RFH+01, RM06, RGR+04].

Définition 5.1.1 (**Table de Hachage Distribuée (DHT en anglais)**). *Une DHT est une technologie permettant la construction d'une table de hachage dans un système réparti. Dans une table de hachage distribuée, chaque donnée est associée à une clé et est distribuée sur le réseau.*

Les DHT possèdent plusieurs propriétés indispensables au fonctionnement des systèmes peer-to-peer. En effet, elles offrent une fonction de hachage cohérente et des algorithmes efficaces de localisation du nœud responsable d'une paire (clé, valeur) donnée. La maintenance des tables de routage communément utilisées pour le stockage des informations sur l'évolution d'un réseau est malheureusement très peu réaliste dans un environnement distribué.

Dans la plupart des réseaux peer-to-peer qui implémentent une DHT, cette table est substituée par la création d'une structure *"d'overlay" ou réseau virtuel* au-dessus du réseau initial, réduisant ainsi la taille de la table en chaque nœud (un nœud est au courant d'un minimum d'autres nœuds du réseau) tout en augmentant considérablement l'efficacité de l'algorithme de recherche. Cet Overlay constitue la structure ou topologie du réseau.

Définition 5.1.2 (**Topologie du réseau**). *Une topologie de réseau informatique est l'architecture physique ou logique définissant les liaisons entre les nœuds du réseau et une hiérarchie éventuelle entre eux.*

Bien que chaque protocole basé sur une DHT ait une topologie qui lui est propre, la technique de construction de l'overlay reste la même pour tous les protocoles [LNBK02].

Définition 5.1.3 (**Construction d'Overlay**). *La construction d'un overlay sur un réseau se fait en trois étapes :*

- *La définition d'une topologie dite "idéale" dans laquelle la résolution de toute requête aboutit à un résultat satisfaisant, et ceci de façon efficace. Plus précisément, on considère que tous les nœuds sont mutuellement atteignables.*
- *La description des opérations d'arrivée et de départ des nœuds du réseau.*
- *La définition d'un protocole de maintenance qui répare périodiquement les perturbations sur la topologie du réseau. En effet, les arrivées et départs des nœuds dégradent la topologie idéale.*

Le protocole Chord est l'une des solutions proposées pour la résolution du problème de localisation de données. Il fournit une fonction de hachage distribuée rapide et implémente une table de hachage cohérente. Dans un N-réseau (réseau ayant N nœuds), chaque nœud a connaissance de log_N autres nœuds. De même, l'algorithme de recherche est en $O(log_N)$. La fonction de hachage distribuée est équitable et les changements sont locaux aux nœuds. Les auteurs de Chord affirment que

trois caractéristiques majeures le distinguent des autres protocoles : sa simplicité, sa performance prouvable et sa correction facilement prouvable [SMK+01] [SMLN+03] [LNBK02].

Chord n'est pas le premier protocole qui propose une solution au problème de localisation de données dans un réseau peer-to-peer. Ses prédécesseurs à l'instar du DNS [MD88] permettaient déjà la localisation des données dans le réseau en attribuant à chaque donnée un nom. Chord offre tous ces services fournis par ses prédécesseurs et par d'autres protocoles qui comme lui implémentent des tables de hachage distribuées : Freenet [C+99], [Hon00], CAN, Pastry [RGR+04]

5.2 Description du protocole Chord

Le protocole de recherche distribuée Chord a été présenté pour la première fois dans [SMK+01]. Ces travaux ont présenté les caractéristiques de Chord et décrivent ses propriétés ainsi que ses opérations sous forme de pseudo-code. Puis [LNBK02] a énoncé l'ensemble des invariants qui garantissent la correction du protocole et enfin,[SMLN+03] a présenté une preuve manuellement de cette correction. De nombreux travaux se sont ensuite intéressés à l'analyse du protocole Chord.

Bien que certains parmi eux [LNBK02], [SMLN+03] aient montré ses bonnes performances, pour la maintenance, les travaux [Zav19], [Zav11], [Zav12], [Zav15b], [Zav15a], [Zav17] menés par P.Zave ont prouvé d'une part que Chord n'est pas correct et d'autre part que la preuve de sa correction n'est pas aussi facile que l'affirment ses auteurs. P. Zave a proposé une version de Chord présumée correcte, laquelle est utilisée dans cette thèse.

Le protocole Chord ne prend en charge qu'une seule opération : étant donnée une clé, il mappe la clé à un nœud. En fonction de l'application, ce nœud peut être responsable d'une donnée associée à la clé. Contrairement à certains protocoles implémentant une DHT, chaque nœud Chord n'a besoin d'information de routage que de quelques autres nœuds. La table de routage étant distribuée, les recherches se font par une communication entre les nœuds. Dans l'état idéal (toutes les données sont accessibles), d'un N-réseau, chaque nœud conserve les informations de log_N autres nœuds pour un routage efficace. Une seule information correcte par nœud suffit pour que Chord soit capable d'effectuer un routage correct des requêtes. Chord possède un algorithme simple pour conserver ces informations dans un environnement dynamique.

Dans Chord, chaque nœud a un identifiant dans un espace d'identifiants à m bits, et peut atteindre d'autres nœuds à l'aide de pointeurs vers d'autres identifiants. Les nœuds et leurs pointeurs forment une topologie essentielle pour assurer la localisation correcte des données sur le réseau.

Cette topologie du réseau est en constante évolution, car des nœuds autonomes peuvent rejoindre ou quitter le réseau (ou être défaillants) à tout moment. Un aspect essentiel du protocole Chord consiste à définir des opérations de maintenance chargées de réparer la topologie du réseau, de sorte que les données stockées dans tout nœud restent accessibles de tout autre nœud, malgré les pannes, les arrivées et les départs.

Ainsi, la propriété de correction de base de Chord est la garantie que les nœuds du réseau finiront par être mutuellement atteignables, autrement dit les données stockées en chaque nœud seront accessibles de tous : *si, à partir d'un instant donné, il n'y a plus de séquence d'arrivées, de départs ou de pannes, le réseau est alors assuré de récupérer une topologie en anneau et de la conserver.* Si le protocole n'est pas correct dans ce sens, certains nœuds d'un réseau Chord seront définitivement inatteignables par d'autres nœuds.

5.2.1 Structure des réseaux Chord

Espace des identifiants

Dans un réseau Chord, chaque nœud possède un identifiant qui est son adresses IP haché en entiers sur m bits. Les paires de clés et données associées sont stockées dans des nœuds. Dans la suite, les notions de nœud et de son identifiant sont confondues. L'espace des identifiants est structuré comme un graphe orienté en forme d'anneau. Intuitivement, les identifiants des nœuds sont ordonnés selon l'ordre usuel des nombres naturels (noté $<$ dans ce qui suit) et pour fermer l'anneau, le plus grand identifiant est rattaché au plus petit. Du fait de cette forme, la localisation d'un identifiant est avantageusement modélisée en vérifiant s'il se situe entre deux autres identifiants.

Définition 5.2.1 (**Ordre sur l'espace des identifiants**)**.** *Soit NODE l'ensemble des nœuds dans un réseau Chord. Étant donnés $n_1 \in NODE$ et $n_2 \in NODE$, l'ensemble des identifiants situés entre n_1 et n_2 est défini par :*

$$\boldsymbol{between}[n_1, n_2] \triangleq \left\{ n \in NODE \;\middle|\; \begin{array}{ll} n_1 < n < n_2 & si\ n_1 < n_2 \\ n_1 < n\ ou\ n < n_2 & sinon \end{array} \right\}$$

Pour chaque nœud n, nous notons ***n.suivant****, le nœud qui suit n selon l'ordre between, c'est-à-dire tel que between[n, n.suivant] = ∅*

Affectation des clés

Les clés sont hachées avec la même fonction de hachage utilisée pour les adresses IP de nœuds et sont mappées dans le même espace d'identifiants que les nœuds. De cette façon, les identifiants des clés et des nœuds sont comparables. De même, nous utiliserons le terme « clé » pour désigner à la fois la clé d'origine et son image par la fonction de hachage.

Selon [LNBK02, SMK+01, SMLN+03], l'attribution des clés se fait selon les règles suivantes :

1. Les identifiants sont ordonnés dans un cercle modulo 2^m.
2. Une clé k est affectée au nœud **nk** tel que **nk = k.suivant**

Pour préserver la cohérence des tables de hachage, les opérations d'arrivée et de départ doivent générer le moins de perturbations possibles. Pour ce faire, lorsqu'un nœud n arrive dans le réseau, il récupère les clés plus petites que son identifiant et plus grandes que l'identifiant du prédécesseur du nœud qu'il suit dans le réseau . De même lorsque n quitte le réseau ou est en panne, ses clés sont attribuées à son successeur dans le réseau.

La Figure 5.1 illustre un réseau Chord dont les identifiants sont codés sur $m = 6$ bits et qui possède huit nœuds et cinq clés. Le nœud suivant la clé k70 est le nœud 10, donc la clé 70 est attribuée au nœud 10. De même les clés 25 et 30 sont attribuées au nœud 30. L'espace des identifiants étant circulaire, à chaque nœud sont associés les clés situées entre lui et son prédécesseur.

Réseaux Chord

Un réseau Chord est construit sur un espace d'identifiants. Dans le but de fournir une procédure de recherche efficace, le réseau est idéalement structuré en anneau. Pour des raisons de tolérance aux pannes, chaque nœud n maintient un pointeur vers une liste de nœuds appelée *liste de successeurs de* n, dont le premier nœud vivant est le *successeur* de n (noté $n.successeur$) dans le réseau. Durant

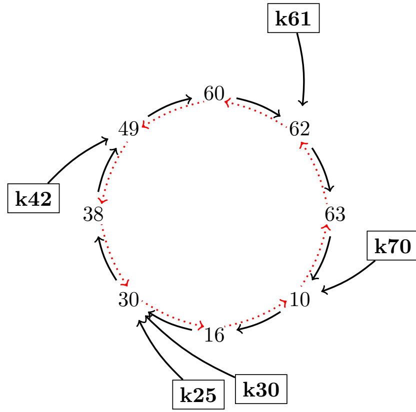


FIGURE 5.1 – Exemple de réseau Ideal

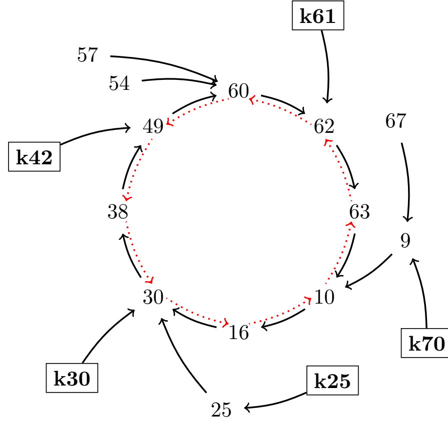


FIGURE 5.2 – Exemple de réseau Valid

l'exécution du protocole, les nœuds acquièrent et mettent à jour un pointeur sur leur *prédécesseur* dans le réseau [SMK+01].

Lorsqu'un réseau est structuré en anneau selon la relation induite par les pointeurs successeurs et que l'ordre des *pointeurs suivants* coïncide avec l'ordre *des pointeurs successeurs* alors, chaque nœud est atteignable depuis n'importe quel autre nœud, autrement dit toutes les données sont accessibles depuis n'importe quel nœud. Un tel réseau est dans un ***état idéal*** (cf : [LNBK02]) ou stable (cf : [SMK+01] et [SMK+01]). Dans cette thèse, nous choisissons l'appellation ***Ideal*** . Les nœuds pouvant rejoindre et quitter le réseau à tout moment, la structure en anneau ne peut pas être garantie en permanence. Par exemple, les nœuds rejoignant un anneau créent un appendice. Les opérations de maintenance visent à récupérer une structure en anneau, malgré ces arrivées et ces départs.

5.2.2 Algorithme de recherche de données dans les réseaux Chord

Le but fondamental de Chord est la recherche d'une clé associée à une valeur donnée. La recherche d'une clé (k) se fait par le biais des requêtes passées autour de l'anneau formé par les pointeurs successeurs. Une requête passe de nœud en nœud jusqu'au nœud n tel que $k \in between[n, n.successeur]$. Pour renforcer les performances du protocole de recherche, chaque nœud doit maintenir une table d'index appelée *Finger Table*.

Définition 5.2.2 (Table d'index). *Soit m le nombre de bits des identifiants de nœuds (resp des clés). Une table d'index est une structure de donnée définie telle que, pour tout nœud n, le $i-ieme$ élément de la table d'index de n (noté $n.finger$) soit égale au successeur de la clé $(n+2^{i-1})$ mod 2^m on note : $n.finger[i] = successeur[(n+2^{i-1})]$ mod 2^m*

Pour tout nœud n, $n.finger[1] = n.successeur$. Lorsqu'un nœud n recherche une clé k, n envoie la requête non pas à son successeur dans le cercle mais au nœud $s = max\{fi \in n.fingers \mid k \in between[n, fi]\}$, le processus se répète jusqu'à l'obtention d'un nœud m tel que $m.successeur = k.suivant$.

Remarque 9. *La table d'index n'a aucun impact sur la définition de l'état idéal d'un réseau Chord, qui ne fait appel qu'à la notion de successeur.*

5.2.3 Les propriétés caractéristiques des réseaux Chord

Les propriétés d'un réseau Chord caractérisent les états dans lesquels une requête donne toujours une réponse exacte. Une meilleure compréhension de ces propriétés requiert l'identification des différentes sources pouvant produire une réponse erronée pour une requête donnée. Deux problèmes majeurs ont été identifiés : (1) la subdivision du réseau en plusieurs composantes non connectées. En effet, dans de telles configurations, les données stockées dans un nœud d'une composante resteront inaccessibles par les nœuds des autres composantes ; (2) l'ordre autour du cercle n'est pas conforme à celui des identifiants des nœuds. Dans de tels réseaux, les résultats des requêtes seront incohérents et erronés. Une illustration de ce problème est donnée dans la Figure 5.3 . La recherche du successeur de la clé 18 donne le noeud 32 ou le nœud 21, selon qu'elle est initiée respectivement par les nœuds 1 et 14 ou le nœud 8.

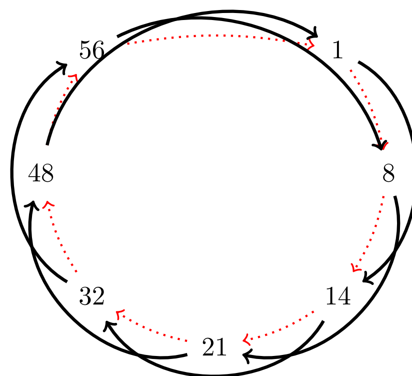


FIGURE 5.3 – Structure d'un réseau Chord loopy. Les traits en pointillés représentent la *relation suivant* autour de l'anneau et les traits pleins *la relation successeur.*

Les auteurs de Chord ont fourni des propriétés explicites d'un réseau garantissant une transmission correcte des données [LNBK02]. Ils définissent notamment l'état *idéal* d'un réseau que nous avons présenté de manière informelle dans la section précédente et un état temporaire imparfait, que nous appelons un état *valide* suivant P. Zave [Zav17]. Comme notre étude ne traitera que de correction du protocole, nous ne présentons pas les propriétés quantitatives et probabilistes mentionnées dans les articles originaux de Chord.

Pour introduire les propriétés du réseau, quelques définitions sont indispensables.

Définition 5.2.3 (**Cohérence locale**)**.** *Un réseau Chord est* localement cohérent *si, pour tout nœud* u*,* $(u.successeur).predecesseur = u$ *.*

Définition 5.2.4 (Cohérence globale). *Un réseau Chord est* globalement cohérent *si, pour chaque nœud u de l'anneau, between*$[u, u.successeur] = \emptyset$.

Le réseau de la figure 5.1 est localement et globalement cohérent.

Définition 5.2.5 (Réseau loopy). *Un réseau Chord est* loopy *s'il est localement cohérent mais globalement incohérent.*

Dans de tels réseaux, la recherche du successeur d'une clé initiée à partir de deux nœuds différents peut renvoyer deux résultats différents. Ainsi, certaines données disponibles sur le réseau peuvent paraître inaccessibles. Le réseau de la Figure 5.3 est *loopy.* Les travaux présentés dans l'article [LNBK02] montrent que seuls les réseaux non-loopy peuvent être corrigés par le protocole de maintenance de Chord.

Définition 5.2.6. ***Réseau Idéal***
Un réseau Chord est dit idéal [LNBK02] [Zav17], ssi :

unicité de l'anneau : *la relation successeur forme un seul anneau de nœuds, et tous les nœuds sont dans cet anneau ;*

non-loopiness : *le réseau est localement et globalement cohérent ;*

validité de la liste des successeurs : *la liste des successeurs de chaque nœud n contient les* r [1] *premiers nœuds qui le suivent dans l'anneau.*

La Figure 5.1 présente un réseau Chord dans un état idéal avec huit nœuds et quatre clés. Chaque clé est stockée dans le nœud ayant le plus petit identifiant parmi les nœuds dont l'identifiant est supérieur à celui de la clé. Par exemple, la clé K12 est stockée dans le nœud 16.

L'unicité de l'anneau et la validité de la liste des successeurs résolvent le problème de subdivision du réseau énoncé plus haut. La propriété *non-loopiness* garantit que le réseau sera toujours bien ordonné. C'est pourquoi dans un réseau Chord *idéal*, toutes les requêtes renvoient des résultats exacts.

Comme expliqués ci-dessus, les arrivées, les pannes et les départs de nœuds forcent le réseau dans un état non idéal. Bien que les opérations de maintenance de Chord visent à réparer de tels états non idéaux, entre l'instant où un nœud rejoint le réseau et l'instant où il est intégré dans l'anneau, les requêtes peuvent être erronées.

Néanmoins [LNBK02] définit une propriété (que nous nommons *valide*), plus faible qu' *idéal.* Cette propriété est satisfaite durant l'exécution du protocole, mais il n'est pas assez forte pour prouver la correction du protocole. Chord est correct en ce sens que cet état est transitoire et peut toujours être réparé par le protocole de maintenance.

Dans un état valide, certains nœuds peuvent ne pas être dans l'anneau, mais dans les appendices de celui-ci. Pour un nœud n de l'anneau, il peut exister une arborescence non vide de nœuds enracinés en n, composés de nœuds ayant récemment rejoint le réseau et ne se trouvant pas encore dans l'anneau. Nous appelons cet arbre l'appendice de n et le désignons par A_n.

Un réseau valide peut contenir des appendices et des pointeurs successeurs et prédécesseurs potentiellement obsolètes, mais qui peuvent encore être mis à jour par le protocole.

1. Les auteurs de Chord affirment que dans un N-réseau Chord la longueur maximale des listes de successeurs est $r \simeq log_N$

Définition 5.2.7. *Un réseau Chord est dit* valide *si :*

connexité *un sous-ensemble de nœuds forme un anneau suivant la relation successeur ; autrement dit, il existe exactement un anneau et les autres nœuds constituent des appendices connectés à l'anneau ;*

non-loopiness

- *l'anneau est non-loopy ;*
- *Pour chaque nœud $n^{'}$ dans un appendice A_n, le chemin de successeurs de $n^{'}$ vers n est constitué de nœuds ordonnés selon l'ordre croissant des identifiants ;*

validité de la liste successeur

- *si n est dans l'anneau, alors $n.successeur$ est égale à $n.suivant$;*
- *si $n^{'}$ est dans un appendice A_n, alors n est le premier nœud vivant de l'anneau qui suit $n^{'}$;*
- *si la liste de successeurs de n passe au-dessus d'un nœud vivant $n^{'}$, alors $n^{'}$ n'est pas dans la liste de successeurs de n.*

La Figure 5.2 présente un réseau Chord dans un état *valide*.

Avec la définition de l'état idéal, la correction du protocole Chord peut être exprimée comme suit :

Théorème 7 (**Correction du protocole Chord**)**.** *Partant d'un réseau initialement* idéal*, quel que soit l'état d'exécution, si ultérieurement il n'y a plus d'opérations d'arrivées ou de départs, alors le réseau finira par devenir idéal et restera comme tel.*

5.2.4 Les opérations dynamiques de Chord

Le comportement du protocole Chord est spécifié par quatre opérations à savoir : (1) les opérations *join* et *fail* décrivant respectivement comment un nœud arrive dans le réseau et comment il quitte le réseau ; (2) les opérations *stabilize* et *rectify* constituent le protocole de maintenance destiné à réparer les perturbations causées par les opérations *join* et *fail*. Chaque opération change l'état d'au plus un nœud.

Opération join

Lorsqu'un nœud n veut intégrer le réseau, il demande à un membre quelconque de trouver le nœud m tel que $m < n < m.successeur$. Le nœud n fixe sa liste de successeurs avec celle de m et prend m comme prédécesseur. En outre, m n'informe aucun autre nœud du réseau de l'existence de n. Cette opération est est résumée par le pseudo-code suivant :

```
n.findSuccList() **recherche de la liste de successeur du nœud n
renvoie
 m | n ∈ between[m, m.SuccList[1]];
n.join()
        m = n.findSuccList();
        n.Predecesseur = m;
        n.SuccList = m.SuccList ;
        ** n.SuccList représente
        ** la liste des successeurs du nœud n
```

Les Figure 5.4 et Figure 5.5 montrent les pointeurs successeurs et prédécesseurs lorsque le nœud 4 rejoint le réseau en contactant le nœud 1.

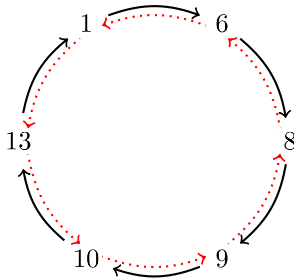


FIGURE 5.4 – Réseau initialement Ideal

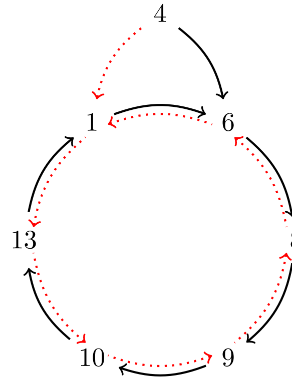


FIGURE 5.5 – Le nœud 4 est dans le réseau.

Opération fail

Un nœud peut quitter le réseau de façon volontaire ou suite à une panne, à l'aide de l'opération *fail*. Ce nœud cesse d'être membre du réseau et est désormais appelé «*nœud mort*». Évidemment, il n'informe aucun autre nœud de son départ et apparaît donc toujours dans la liste de successeurs des autres nœuds.

Par ailleurs, les nœuds peuvent quitter involontairement le réseau, à cause d'un départ inattendu. Dans ce cas, le nœud est aussi considéré *mort*.

Les hypothèses de tolérance aux pannes

Chord repose sur une hypothèse importante selon laquelle chaque membre a toujours au moins un successeur vivant. En pratique, cela dépend de la taille de la liste des successeurs (cette liste ne doit pas contenir de doublons) et du rapport entre les occurrences des opérations de maintenance et les occurrences des défaillances (pannes, départs). Par exemple, supposons qu'un nœud vivant n donné ait une liste de successeurs de taille 3 et que les trois successeurs de n soient morts avant toute occurrence de stabilisation ou de rectification, le réseau n'est plus dans un état valide (la structure en anneau est cassée, ou un appendice est déconnecté) et le protocole ne peut pas se remettre d'une telle situation. Ces contraintes constituent *l'hypothèse de tolérance aux pannes* énoncée comme suit :

Définition 5.2.8. *Les listes de successeurs des nœuds sont suffisamment longues, et les opérations* fail *sont suffisamment peu fréquentes, de sorte qu'aucun membre ne soit jamais laissé sans nœud vivant dans sa liste de successeurs [LNBK02].*

Une autre hypothèse est *la communication parfaite entre un nœud et son successeur*, en ce sens que chaque nœud répond nécessairement à une requête dans un délai fixé. Cela permet une détection parfaite des défaillances (un successeur qui ne répond pas à une requête avant un délai donné est considéré comme mort).

Opération stabilize

Les opération *join* et *fail* créent des perturbations pouvant casser la propriété *idéale* du réseau. Pour corriger ces perturbations, chaque nœud exécute de façon périodique l'opération *stabilize*. C'est ainsi qu'il fait connaissance de nouveaux nœuds qui viennent d'intégrer le réseau. Le but de l'opération est de mettre à jour la liste de successeurs des nœuds.

Lorsqu'un nœud n se stabilise il contacte son successeur (*n.successeur*), si ce dernier est en vie, alors n lui demande l'identifiant de son prédécesseur ((*n.successeur*).*predecesseur*). Si l'identifiant du prédécesseur est un meilleur candidat pour être son successeur que son successeur actuel (selon l'ordre des identifiants des nœuds), alors il prend ce prédécesseur comme son nouveau successeur. Si (*n.successeur*) est mort, alors n prend pour successeur le premier nœud vivant de sa liste de successeur. Après une stabilisation, le nœud stabilisé notifie (demande de rectification) son identité à son successeur. Le pseudo-code est donné par :

```
** verifie periodiquement si le successeur de n est en vie
**              et met a jour
n.stabilize()
Si EstVivant(n.successeur) Alors{
  newsucc =
                (n.successeur).predecesseur;
                ** recherche le meilleurs successeur
        Si newsucc ∈ between[n,(n.successeur)] Alors{
         Si Estvivant(newsucc) Alors
           n.SuccList =
                 append (newsucc, butLast(newsucc.SuccList));
        ** le nouveau successeur est en vie
        Sinon le nouveau successeur est mort et rien ne change
        }
        Sinon la stabilisation est complète
}
Sinon
        n.SuccList =
                append (tail(n.SuccList), (last(n.SuccList)).suivant);
```

∗ ∗ ***append*** est une fonction qui concatène deux listes ;
∗ ∗ ***tail*** renvoie une liste sans son premier élément ;
∗ ∗ ***butLast*** renvoie une liste sans son dernier élément.
∗ ∗ ***last*** renvoie le dernier élément d'une liste

Le nœud 4 de la Figure 5.6 se stabilise et contacte son successeur nœud 6. Il apprend que ce dernier est son meilleur successeur et lui notifie son identité. De même, dans la Figure 5.8 le nœud 1 stabilise juste après la rectification du nœud 6, apprenant que le prédécesseur de son successeur actuel nœud 4, est meilleur que son successeur actuel nœud 6, il adopte nœud 4 comme successeur.

Opération rectify

Suite à une opération de stabilisation, le nœud notifié exécute une opération *rectify*. *rectify* est une opération qui est toujours déclenchée par une stabilisation, elle est chargée de mettre à jour les pointeurs prédécesseurs des nœuds. Un membre notifié n doit adopter le membre notifiant p comme son nouveau prédécesseur si son prédécesseur actuel est mort ou si le membre notifiant est plus proche de lui que son prédécesseur actuel, formellement si $n.predecesseur \in between[p, n]$. Cet algorithme est résumé par le pseudo-code :

```
            ** m  demande a être le prédécesseur de n
n.rectify(newPrdc)
Si (n.Predecesseur = nil  ou
                    newPrdc ∈ between[n.Predecesseur, n])
          Alors n.Predecesseur = newPrdc ;
Sinon rien ne change.
```

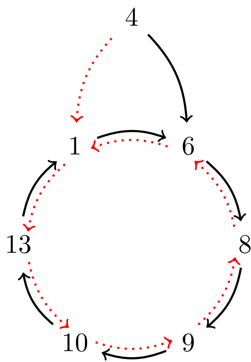


Figure 5.6 – Le nœud 4 se stabilise et notifie le nœud 6

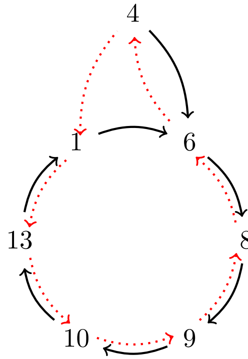


Figure 5.7 – Le nœud 6 rectifie en réponse à la notification du nœud 4.

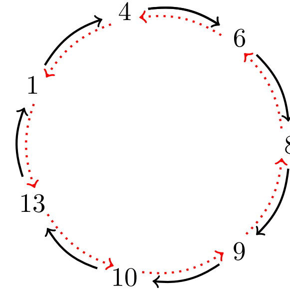


Figure 5.8 – Le nœud 1 se stabilise et notifie le nœud 4 qui rectify dans la foulée.

Les Figure 5.6 et Figure 5.7 illustrent une opération de rectification effectuée par le nœud 6 en réponse à la notification faite par le nœud 4.

5.3 Conclusion

Le protocole de recherche distribuée Chord est un système dynamique avec des propriétés structurelles riches. Sa correction stipule que si la maintenance fonctionne correctement, alors il sera toujours possible pour deux nœuds quelconques du réseau de communiquer. C'est donc une propriété de vivacité. L'une des contributions de cette thèse est l'analyse de la propriété de correction du protocole Chord, avec une méthode de spécification et de vérification formelle adaptée à l'analyse des propriétés de type vivacité. Les résultats de cette analyse sont détaillés dans le chapitre 6.

Deuxième partie

Contributions

Chapitre 6

Extension d'Electrum avec des actions

6.1 Introduction

Dans le chapitre 3, nous avons introduit Electrum, une extension temporelle d'Alloy. Electrum simplifie la spécification des systèmes dynamiques ayant des propriétés structurelles riches et permet la vérification automatique sur horizon temporel borné et non-borné. En pratique, il est souvent naturel de spécifier le comportement d'un système à l'aide de ses actions élémentaires. Dans de nombreux modèles Alloy et Electrum, la description du comportement se base déjà sur des actions élémentaires. Celles-ci sont soit spécifiées par des prédicats soit par des signatures (cf différents idiomes au chapitre 3 section 3.1.6 présentés dans [Jac06]). Elles correspondent aux transitions possibles du système, et nécessitent donc peu d'expressivité en termes de logique temporelle. Faire apparaître explicitement ces actions dans la spécification présente plusieurs avantages :

1. une partie des contraintes comportementales, telle que les conditions du cadre, ou les traces d'exécution, peut être spécifiée de manière systématique ;
2. une procédure de vérification qui s'appuie sur un model checker peut tirer parti d'une telle description de l'évolution du système.

Par conséquent, nous proposons dans ce chapitre une extension d'Electrum avec une couche action, qui est constituée des éléments suivants :

- un langage pour décrire les actions, une action étant un énoncé qui fait référence à deux instants consécutifs : l'instant courant (garde) et son successeur immédiat (post-condition) ;
- ce langage est simplement du sucre syntaxique. La sémantique est définie par une traduction dans Electrum standard ;
- les conditions du cadre sont générées automatiquement ;
- les contraintes sur le séquencement des actions sont générées automatiquement. Elles imposent de suivre un modèle temporel par entrelacement ;
- la syntaxe des contraintes est étendue pour permettre de raisonner sur les occurrences des actions.

```
// STRUCTURE
open util/ordering[Key]
sig Key {}
sig Room {
  keys: set Key,
  var current: one keys
}
fact DisjointKeySets {
  Room <: keys in Room lone-> Key
}
one sig FD {
  var lastKey: Room -> lone Key,
  var occupant: Room -> Guest
}
sig Guest { var gkeys: set Key
}
fun nextKey[k: Key, ks: set Key]:
  set Key {
  min[nexts[k] & ks] }
// Predicat pour les condition du cadre
pred noRoomChangeExcept
  [rs:set Room] {
  all r: Room - rs |
      r.current = r.current'
}
pred noGuestChangeExcept
  [gs: set Guest] {
  all g: Guest - gs |
      g.gkeys = g.gkeys'
}
//Actions
pred checkin[g: Guest, r: Room,
    k: Key] {
  k = nextKey[r.(FD.lastKey), r.keys]
  no r.(FD.occupant)
  gkeys' = gkeys + g->k
  occupant'= occupant + FD->r->g
  lastKey' = lastKey ++ FD->r->k
  noRoomChangeExcept[none]
}

pred checkout[g: Guest] {{
  some FD.occupant.g
  occupant' = occupant - FD->Room->g
  lastKey' = lastKey
  noRoomChangeExcept[none]
  noGuestChangeExcept[none]
}
pred entry [g: Guest, r: Room,
  k: Key] {
  k in g.gkeys and
  ((k = r.current
  r.current' = r.current) or
  (k = nextKey[r.current, r.keys]
    and r.current' = k))
  noRoomChangeExcept [r]
  noGuestChangeExcept [none]
  noDeskChange
}
 // Etat initial
fact init {
  no Guest.gkeys
  no FD.occupant
  all r: Room |
    r.(FD.lastKey) = r.current
}
// Traces
fact traces {
  always (
    some g: Guest, r: Room,
      k: Key {
      entry[g, r, k]
      or checkin[g, r, k]
      or checkout[g] })
}
// Commandes
pred consistent {}
run consistent for 4
  but 10 Time
assert BadSafety  {
  always (
    all r: Room, g: Guest,
      k: Key {
        entry[g, r, k] and
        some r.(FD.occupant)
           implies g in r.(FD.occupant) )
}
check BadSafety for 4 but 10 Time
```

Figure 6.1 – Modèle Electrum standard de l'Hôtel

Le reste de ce chapitre s'articule de la manière suivante. Dans la section 6.2, nous définissons la syntaxe de la couche action, et fournissons un exemple illustratif. Nous expliquons également sa sémantique par une compilation vers Electrum ordinaire dans la section 6.3 et, dans la section 6.4, nous expérimentons son impact sur l'efficacité des analyses. Enfin nous donnons une conclusion dans la section 6.5.

6.2 Syntaxe de la couche action

Dans cette section, nous définissons la syntaxe du langage des actions.

6.2.1 Actions

La syntaxe des actions étend celle d'Electrum standard. Concrètement, nous ajoutons le mot-clé `act` qui introduit le nom d'une action, éventuellement avec des paramètres. Les actions s'exécutent de manière atomique et concernent deux instants consécutifs. Ainsi, la syntaxe autorisée au sein des actions est la suivante : (1) aucune restriction sur l'utilisation des connecteurs de la FOR ; (2) les seuls constructeurs temporels autorisés sont : le connecteur LTL X (`after`) et l'opérateur *prime(')* ; (3) l'imbrication de ces deux constructeurs temporels est proscrite.

La figure 6.3 présente le modèle Electrum Action du système de l'Hôtel. Les événements d'enregistrement, d'entrée et de sortie sont modélisés respectivement par les actions `checkin`, `entryWithRecoding`, `entryWithoutRecoding` et `checkout`. Toutes ces actions ont une syntaxe valide. En effet, l'action `entryWithoutRecoding` contient uniquement des opérateurs de la FOR, tandis que les trois autres actions contiennent uniquement l'opérateur *prime(')* sans aucune imbrication.

En revanche, dans l'exemple présenté à la figure 6.2 : les syntaxes de `BadAction1` (l. 1) et de `BadAction2` (l. 13) ne sont pas valides. En effet, le premier contient des imbrications entre les constructeurs `after` et *prime(')*, tandis que le deuxième contient le connecteur LTL F (`eventually`).

```
1  act BadAction1[r: Room]{
2    some k: Key {
3      r.currentkey'= k --syntaxe correcte
4        after(r.currentkey')= k)
5      //syntaxe incorrecte:
6      //imbrication after et prime
7      after(after((kin r.keys)
8        and (r.currentkey= k)))
9      //syntaxe incorrecte:
10       //imbrication de after
11       }
12 }
```

```
act BadAction2[r: Room]{                13
  some k: Key {                         14
    eventually(r.currentkey'            15
        in r.currentkey+k)              16
//syntaxe incorrecte:                   17
  //connecteur eventually               18
 }                                      19
}                                       20
```

FIGURE 6.2 – Exemples d'actions

6.2.2 Les conditions du cadre

Le déclenchement d'une action modifie la valeur de certaines variables du système. Ainsi, dans la définition d'une action, on indique les variables qui sont modifiées par l'exécution de l'action , ceci est utilisé pour générer les conditions du cadre.

Une action est spécifiée avec une clause `modifies` accompagnée de signatures et de champs variables qu'elle contrôle (le déclenchement de l'action les affecte). En pratique, cette clause est utilisée pour générer les conditions du cadre suivant une règle simple qui dit que : *les signatures et champs variables non contrôlés par une action ne changent pas quand une occurrence de celle-ci a lieu.*

Exemple 23. *Par exemple dans la figure 6.3, l'action* `checkout` *contrôle uniquement le champ* occupant*, ainsi :*

checkout[g] $\Rightarrow$

```
 // structure
open util/ordering[Key]
sig Key {}
sig Room {
  keys: set Key,
  var current: one keys }
fact DisjointKeySets {
  Room<:keys in Room lone→ Key
}
one sig FD {
  var lastKey: Room → lone Key,
  var occupant: Room → Guest }
sig Guest { var gkeys: set Key }
...
fun nextKey[k: Key, ks: set Key]:
    set Key {
 min[nexts[k] & ks] }

// Actions
act checkin[g: Guest, r: Room, k: Key]
  modifies gkeys, occupant, lastKey {
  no r.(FD.occupant)
  k = nextKey[r.(FD.lastKey), r.keys]
  gkeys' = gkeys + g → k
  occupant' = occupant + FD→r→g
  lastKey' = lastKey ++ FD→r→k }

act checkout[g: Guest]
  modifies occupant {
  some FD.occupant.g
  occupant' = occupant − FD→Room→g
}

act entryWithRecoding [g: Guest, r: Room,
   k: Key] modifies current {
  k in g.gkeys
  k = nextKey[r.current, r.keys]
  current' = current ++ r → k
}
act entryWithoutRecoding [g: Guest,
  r: Room, k: Key] {
  k in g.gkeys
  k = r.current
}
//etat initial
fact init {
  no Guest.gkeys
  no FD.occupant
  all r: Room |
    r.(FD.lastKey) = r.current }

// COMMANDES
pred consistent {}
run consistent for 4 but 10 Time

assert BadSafety {
  always (
    all r: Room, g: Guest, k: Key{
   entry[g, r, k] and
    some r.(FD.occupant)
    ⇒ g in r.(FD.occupant) })
}

check BadSafety for 4 but 10 Time
```

FIGURE 6.3 – Modèle Electrum action de l'Hôtel

$\wedge$ *currentkey'= currentkey*

$\wedge$ *lastkey'= lastkey*

$\wedge$ *keys'= keys*

D'autres formalismes se servent de clauses similaires pour générer les conditions du cadre. Notamment TLA+ [Lam02b] utilise la clause ***UNCHANGED***. Contrairement à ce dernier, nous avons préféré indiquer ce qui peut changer par le déclenchement d'une occurrence d'une action, plutôt qu'indiquer ce qui ne peut pas changer. Nous pensons que notre approche réduit le nombre d'erreurs de spécification, lorsqu'il faut introduire de nouvelles signatures et champs variables dans un modèle contenant des actions. Dans ce cas, le déclenchement de l'action ne changera que les signatures et champs variables qu'elle contrôle, à moins que la clause `modifies` ne soit explicitement mise à jour par l'utilisateur. Toutefois, d'après notre expérience, les nouveaux champs et signatures variables sont très souvent modifiés par de nouvelles actions. À ce titre nous privilégions un processus qui préserve les parties pré-existantes du modèle.

6.2.3 Modèle temporel

L'évolution d'un système dans Electrum est régie par un modèle temporel ou trace d'exécution qui fixe les contraintes sur la façon dont les actions doivent se déclencher. Dans cette extension d'Electrum des contraintes supplémentaires liées à la définition des actions sont ajoutées au modèle temporel.

La contrainte sémantique la plus importante imposée par la couche action est un modèle temporel par entrelacement qui dit que : ***exactement une action se produit à chaque instant.***

Ce modèle temporel peut parfois exiger la définition d'une action de bégaiement (cf TLA+ 3.3). Le modèle temporel n'intègre pas par défaut cette notion de bégaiement. En effet, il est souvent requis pour des opérations (décomposition, raffinement de spécification comme dans TLA+ par exemple) [Lam02b] qui ne relèvent pas réellement du domaine des méthodes formelles légères comme Electrum, où l'accent est plutôt mis sur la spécification de petits modèles abstraits de systèmes.

Néanmoins, la sémantique d'Electrum n'autorise pas les traces finies. Dès lors, certains modèles (comme ceux de systèmes distribués) peuvent sembler inconsistants uniquement parce que dans certains états, aucune action ne peut être déclenchée (par exemple, le système a atteint l'état souhaité) et les seules traces possibles seraient finies. Tel est le cas du protocole Chord qui possède un état dit idéal dans lequel aucune action de maintenance ne peut être déclenchée. Pour obtenir les traces infinies, on autorise un état de bégaiement en introduisant une action silencieuse `skip`. Ainsi, lorsque l'état idéal est atteint, le système est capable de boucler indéfiniment dans cet état.

Cependant, introduire une action de bégaiement impose généralement la spécification de contraintes d'équité (voir ci-dessous). Le but est de prohiber des segments de transitions de bégaiement infinis, en forçant le système à exécuter l'action `skip` infiniment si et seulement si aucune action du système n'est déclenchable.

6.2.4 Occurrence des actions dans les contraintes

Une occurrence d'une action peut être référencée dans une contrainte avec ou sans paramètres réels (ex : de entry[g, r, k] dans la figure 6.3 [l.60)]. Le dernier cas équivaut à une quantification existentielle implicite sur l'ensemble des paramètres possibles de l'action. Par exemple, `after` checkout signifie réellement : `after` (`some` g : Guest | checkout [g]).

En outre, il faut noter que la sémantique du référencement d'une action dans une contrainte est en réalité le déclenchement d'une occurrence de celle-ci. Autrement dit, le déclenchement d'une occurrence d'une action implique la satisfaction de son corps, tandis que, le corps d'une action peut être satisfait même si aucune occurrence n'est pas déclenchée.

Exemple 24 (référence et déclenchement d'une action). *Dans le système de l'Hôtel :*

checkout ⇒

```
(some g: Guest|some Desk.occupant.g and
occupant' = occupant−Desk→Room→g)
```

alors qu'on peut avoir :

```
(some g: Guest | some Desk.occupant.g and
occupant' = occupant − Desk→Room→g) ∧ (no checkout)
```

Ceci est dû à la sémantique que nous avons donnée aux actions, elle est décrite dans la section 6.3. On pouvait également utiliser d'autres techniques comparables à celle que nous avons choisie. Par exemple, celle utilisée dans TLA+ [Lam02b] qui considère que l'occurrence d'une action est équivalente à la satisfaction de son corps. Pour l'exemple ci-dessus, on aurait eu :

```
checkout ⇔
    (some g: Guest|some Desk.occupant.g and  occupant' = occupant−Desk→Room→g)
```

Contrairement à TLA+, notre technique permet de distinguer un effet observé causé par une action du même effet causé par une autre action. Nos expériences ont montré qu'il est effectivement intéressant de faire une telle distinction.

6.3 Sémantique

La syntaxe de tous les éléments de la couche action étant définie, pour donner un sens à ces éléments, nous présentons dans cette section une sémantique formelle d'action. La couche action constitue un sucre syntaxique au-dessus d'Electrum standard. Nous décrivons maintenant sa sémantique formelle en termes de traduction vers Electrum standard. Nous illustrerons cette traduction à l'aide de l'exemple Hotel présenté à la figure 6.3.

6.3.1 Structure des actions et modèle temporel

Comme nous avons indiqué précédemment, la stratégie actuelle de traduction ne consiste pas à compiler les actions vers les prédicats, mais plutôt vers une structure de signatures et de champs, codant tous les événements possibles (les occurrences des actions). Pour ce faire, nous introduisons d'abord une énumération `_Action` de tous les noms d'actions. Pour le modèle Hôtel, on a :

```
enum _Action { checkin, checkout, entrywithRecoding, entryWithoutRecoding}
```

Ensuite, nous ajoutons une relation codant tous les événements possibles, en prenant l'union de toutes les valuations possibles des actions. Comme les actions peuvent différer en arité, nous les étendons le cas échéant à la plus haute arité à l'aide d'une signature fictive nommée `_Dummy`. Cette signature ne coûte rien à la vérification, car elle est statique et `one`. En parallèle, nous spécifions le modèle temporel en forçant exactement un événement à se produire à chaque instant. Avec l'Hôtel, on obtient :

```
one sig _Dummy {}
one sig _E { // signature qui englobe toutes les actions possibles.
 var _event :  (checkin → Guest → Room → Key)
        + (checkout → Guest → _Dummy → _Dummy)
        + (entryWithRecoding → Guest → Room → Key)
        + (entryWithoutRecoding → Guest → Room → Key]
 } { one _event } // modele temporel
```

Remarque 10. ***A propos de la traduction des actions***
Le lecteur peut s'interroger sur le choix de la stratégie de traduction adoptée, car plusieurs autres stratégies seraient envisageables :

Solution 1 (**Compilation vers l'idiome prédicat**). *Une solution serait de compiler les actions vers les prédicats, tels que définis dans Alloy ou Electrum standard. Cependant, le principal inconvénient de cette solution est son inadéquation face à la visualisation d'instances ou de contre-exemples, comparativement à une visualisation utilisant des signatures.*

En effet, l'instance d'une signature (représentant l'événement) contenue dans un contre-exemple ou une instance du modèle est toujours facilement identifiable. Tandis qu'un prédicat est une formule et qu'il est laborieux de vérifier si elle est satisfaite par l'instance ou le contre-exemple. Par ailleurs, avec l'utilisation des prédicats, il est très difficile d'exprimer le fait qu'il y a une unique action à chaque instant, il n'est donc pas possible d'utiliser un modèle temporel par entrelacement.

Si les signatures ou les champs sont utilisés, alors d'autres solutions sont possibles :

Solution 2 (**Compilation vers l'idiome Event**). *On pourrait aussi compiler les actions vers l'idiome `Event` tel que définis dans Alloy ou dans Electrum standard. Pour les raisons évoquées ci-dessus (utilisation des signatures et champs), cette solution est très efficace pour la visualisation des contre-exemples contrairement aux prédicats. Néanmoins, pour une compilation vers le format SMV, les occurrences des actions sont introduites à l'aide de variables booléennes supplémentaires, consommant ainsi énormément d'espace mémoire.*

Solution 3 (**Compilation vers une structure de signatures et de champs**). *C'est la solution que nous avons choisie, elle force exactement un événement à se produire à chaque instant. Elle peut être intéressante en ce qui concerne la compilation dans un format SMV. En effet, dans ce cas le champ `_event` pourrait être traduit en une seule variable SMV plus précisément une énumération de labels (chaque label représente l'occurrence d'une action), consommant ainsi moins de mémoire qu'un encodage "plat"(chaque label est encodé par une variable SMV). En réalité,* ***n*** *labels seront encodés par une énumération utilisant $log_2(n)$ bits. Néanmoins, le décodage d'une énumération par le model checker nuXmv (ou NuSMV) a aussi un coût non négligeable.*

Tout compte fait, des évaluations supplémentaires seront nécessaires (notamment sur des modèles complexes) pour choisir la stratégie à adopter. Cette stratégie aura un impact sur la sémantique. En effet, la stratégie que nous avons choisie exprime très simplement, à l'aide d'un argument de cardinalité, qu'exactement un événement se produit à chaque instant (`var _event` contient l'ensemble de tous les occurrences possibles des événements et le fait `one _event` associé à la signature `_E`) impose un événement à chaque instant). Tandis qu'une solution basée sur des prédicats privilégierait un modèle temporel dans lequel les actions peuvent se produire simultanément. En effet, il n'y aurait pas un ensemble d'événements sur lequel on pourrait appliquer une cardinalité.

Exemple 25 (**Stratégies de traduction**). *On considère un modèle restreint du système de l'Hôtel constitué des signatures et de l'action `checkin`. Ici, on ne s'intéresse qu'aux variables SMV générées par chacune des stratégies.*

```
open util/ordering[Key]
sig Key {}
sig Room {
  keys: set Key,
  var current: one keys
}
fact DisjointKeySets {
  Room<:keys in Room lone→ Key
}%divide by = {1000},%
     % fixed zerofill,%
      %empty cells with={\timeout},%
one sig Desk {
  var lastKey: Room → lone Key,
  var occupant: Room → Guest
}
```

```
sig Guest {
  var gkeys: set Key
}
act checkin[g: Guest, r: Room, k: Key]
    modifies gkeys, occupant, lastKey {
  no r.(Desk.occupant)
  k = nextKey[r.(Desk.lastKey), r.keys]
  gkeys' = gkeys + g → k
  occupant' = occupant + Desk→r→g
  lastKey' = lastKey ++ Desk→r→k
 }
```

Si on fixe un scope de 2 pour chaque signature alors, en fonction de la stratégie de traduction utilisée le modèle SMV contiendra les variables suivantes :

Dans le cas d'une compilation vers les prédicats, l'ensemble des variables du modèle est égale à l'ensemble des variables du modèle Electrum standard c'est-à-dire :

VAR
currentkey_Room1_Key1 : boolean
currentkey_Room1_Key2 : boolean
currentkey_Room2_Key1 : boolean
currentkey_Room2_Key2 : boolean
lastkey_Room1_Key1 : boolean
lastkey_Room1_Key2 : boolean
lastkey_Room2_Key1 : boolean
lastkey_Room2_Key2 : boolean
occupant_Room1_Guest1 : boolean
occupant_Room1_Guest2 : boolean
occupant_Room2_Guest1 : boolean
occupant_Room2_Guest2 : boolean
gkeys_Guest1_Key1 : boolean
gkeys_Guest1_Key2 : boolean
gkeys_Guest2_Key1 : boolean
gkeys_Guest2_Key2 : boolean

Variables du modèle SMV dans le cas d'une compilation vers les prédicats.

Dans le cas d'une compilation vers l'idiome Event, les variables suivantes qui représentent les occurrences des actions sont ajoutées au modèle SMV d'Electrum standard :

VAR
Hull_Checkin2_is_in_Checkin : boolean
Hull_Checkin1_is_in_Checkin : boolean
r___Hull_Checkin2_Room2 : boolean
r___Hull_Checkin2_Room1 : boolean
r___Hull_Checkin1_Room2 : boolean
r___Hull_Checkin1_Room1 : boolean
k___Hull_Checkin2_Key2 : boolean
k___Hull_Checkin2_Key1 : boolean
k___Hull_Checkin1_Key2 : boolean
k___Hull_Checkin1_Key1 : boolean
g___Hull_Checkin2_Guest2 : boolean
g___Hull_Checkin2_Guest1 : boolean
g___Hull_Checkin1_Guest2 : boolean
g___Hull_Checkin1_Guest1 : boolean

Variables d'occurrences d'actions dans le d'une compilation vers les événements

Dans la stratégie actuelle les variables suivantes qui représentent les occurrences des actions sont ajoutées au modèle SMV d'Electrum standard :

```
VAR
_event_checkin_Guest1_Room1_Key1 : boolean
_event_checkin_Guest1_Room1_Key2 : boolean
_event_checkin_Guest2_Room1_Key1 : boolean
_event_checkin_Guest2_Room1_Key2 : boolean
_event_checkin_Guest1_Room2_Key1 : boolean
_event_checkin_Guest1_Room2_Key2 : boolean
_event_checkin_Guest2_Room2_Key1 : boolean
_event_checkin_Guest2_Room2_Key2 : boolean
```

Où _event_checkin_Guest1_Room1_Key1 indique le fait que l'action `checkin` *ait pour paramètres $Guest1, Room1$ et $Key1$.*

Variables d'occurrences d'actions dans la stratégie actuelle.

L'optimisation du modèle SMV consiste donc à considérer les variables booléennes précédentes comme des labels de l'énumération suivante :

```
enum _event = { checkin_Guest1_Room1_Key1, checkin_Guest1_Room1_Key2,
    checkin_Guest2_Room1_Key1, checkin_Guest2_Room1_Key2,
    checkin_Guest1_Room2_Key1, checkin_Guest1_Room2_Key2,
    checkin_Guest2_Room2_Key1, checkin_Guest2_Room2_Key2 }
```

Énumération d'occurrence pour une optimisation dans le modèle SMV.

6.3.2 Occurrences des actions

Pour pouvoir référencer les actions dans d'autres contraintes, nous générons un prédicat `fired` indiquant si une action est effectivement déclenchée. Puisque les actions peuvent prendre différents types de paramètres, les arguments du prédicat `fired` sont pris dans l'union de tous ces types. Une signature spéciale `_Arg` est introduite pour représenter cette union de types. Encore une fois l'arité du prédicat `fired` est la plus grande arité des actions. Plus précisément, `fired` a pour paramètres : le nom de l'action pris dans l'énumération `_Action`, et les paramètres de cette action de type `_Arg`. Pour l'Hôtel, les expressions de `_Arg` et `fired` sont données par :

```
var sig _Arg = _Dummy + Guest + Room + Key {}
pred fired [a : _Action, x1, x2, x3 : _Arg] { // si max arite = 3
  a→x1→x2→x3 in _E._event }
```

Ainsi, si un utilisateur fait référence à une action dans une contrainte, alors elle est traduite. en un appel au prédicat `fired` avec les paramètres correspondants. Par exemple `entry[g,r,k]` (Fig. 6.3 [l. 60]) est traduit par `fired[entry,g,r,k]`, qui équivaut à :

```
some g: Guest, r: Room, k: Key | fired[entry,g,r,k].
```

Le déclenchement d'une occurrence d'une action engendre des modifications des variables qui sont sous son contrôle, faisant ainsi passer le système dans un nouvel état. Ces modifications constituent les effets de l'action.

6.3.3 Les effets d'une action

Les effets d'une action doivent être conformes aux contraintes définies dans son corps. Autrement dit, si une action est déclenchée alors son corps est systématiquement satisfait. Contrôler le déclenchement des actions peut être utile à plus d'un titre, par exemple, on peut interdire ou forcer certains comportements du système.

Pour faciliter l'expression des contraintes sur l'effet des actions, nous définissons pour une action `a` le prédicat `body_a` de même arité que `a` qui fait référence au corps (garde et postcondition) de l'actions `a`. Les effets des actions sont spécifiés comme indiquer dans la figure 6.4

```
1  fact Effet_Action {
2    always (all x1: Guest, x2: Room, x3: Key |
3      fired[checkin,x1,x2,x3] implies
4         body_checkin[x1,x2,x3])
5    always (all x1: Guest, x2: _Dummy, x3: _Dummy |
6      fired[checkout,x1,x2,x3] implies
7        body_checkout[x1,x2,x3])
8    always (all x1: Guest, x2: Room, x3: Key |
9      fired[entryWithRecoding,x1,x2,x3] implies
10       body_entryWithRecoding[x1,x2,x3])
11   always (all x1: Guest, x2: Room, x3: Key |
12     fired[entryWithoutRecoding,x1,x2,x3] implies
13       body_entryWithoutRecoding[x1,x2,x3])
14    }
```

FIGURE 6.4 – Expression de l'effet d'une action

Dans cette spécification, `body_checkin[x1,x2,x3]` est un prédicat contenant le corps de l'occurrence `checkin [x1,x2,x3]` de l'action `checkin`. `fired[checkin,x1,x2,x3]` signifie que cette occurrence est déclenchée. Ainsi, la figure 6.4.[l.4] spécifie l'effet de l'action `checkin` à savoir : si l'occurrence `checkin [x1,x2,x3]` est déclenchée, alors le corps (les gardes et postconditions) de l'action est satisfait pour ces paramètres. Pour chaque action une expression similaire de son effet est donnée. La contrainte `fact Effet_Action` de la figure 6.4 contrôle ainsi les effets des différentes actions tout au long de l'exécution du système.

6.3.4 Les conditions du cadre

Comme nous l'avons annoncé précédemment, chaque action s'occupe des conditions du cadre des signatures et des champs variables qu'elle modifie (dans la clause `modifies`). D'autre part, les signatures et les champs variables ne figurant pas dans cette clause sont gérés par des conditions du cadre générées automatiquement.

Le schéma retenu génère des conditions du cadre suivant le style "Reiter" [Jac06, BMR95] c'est-à-dire que : pour chaque signature ou champ variable (contenu dans la clause `modifies` d'une action), une contrainte indique que si elle a été modifiée au cours d'une transition, c'est grâce à l'une des actions qui la modifie.

Exemple 26. *Dans l'Hôtel, le champ `currentKey` est modifiable par l'action `EntryWhitdRecoding`, le champ `occupant` est modifiable par les actions `checkin` et `checkout`, et enfin les champs `gkeys` et*

`lastKey` sont modifiables par l'action `checkin`. Les conditions du cadre pour les champs variables du modèle sont données par le fait suivant.

```
fact {
  always { occupant' != occupant implies
    ((some x1: Guest, x2: Room, x3: Key |
      fired[checkin,x1,x2,x3]) or
    (some x1: Guest, x2: _Dummy, x3: _Dummy |
      fired[checkout,x1,x2,x3])) }
  always { gkeys' != gkeys implies
    (some x1: Guest, x2: Room, x3: Key |
      fired[checkin,x1,x2,x3] }
  always { lastKey' != lastKey implies
    (some x1: Guest, x2: Room, x3: Key|
      fired[checkin,x1,x2,x3])
  always { currentKey' != currentKey implies
    (some x1: Guest, x2: Room, x3: Key|
      fired[ EntryWithRecoding,x1,x2,x3])}
}
```

Remarque 11. *L'action EntryWithoutRecoding ne modifie aucun champ variable, elle n'apparaît donc pas dans l'expression des conditions du cadre de ces champs. Par ailleurs, une signature ou un champ déclaré variable et qui n'est contrôlé par aucune action (ne figure dans la clause `modifies` d'aucune action) est systématiquement défini comme invariant, à l'aide du fait `fact {always v' = v }`.*

6.4 Expérimentation

6.4.1 Cas d'études

Les expériences ont été réalisées sur des modèles classiques d'Alloy [TD06] présentés dans les paragraphes suivants.

SpanTree

Description du système Il s'agit d'un algorithme distribué simple de construction d'arbre couvrant pour des réseaux de topologie arbitraires(mais connectée). Chaque nœud de l'arbre est situé à un niveau(profondeur dans l'arbre) et possède un nœud parent (un nœud du niveau $n-1$ pour un nœud de profondeur n). Dans cet algorithme un nœud racine (ne possède pas de parent) spécifique éventuellement élu au préalable commence par s'affecter le niveau 0. Les nœuds auxquels un niveau a été déjà attribué (c'est-à-dire qui se trouvent déjà dans l'arbre couvrant) diffusent leur niveau aux voisins.

Lorsqu'un nœud reçoit un tel message, s'il n'appartient pas encore à l'arbre, alors il fixe son propre niveau à un plus le niveau de l' émetteur, puis enregistre ce dernier comme son nœud parent. Si le nœud est déjà dans l'arbre alors, il ignore les messages reçu ultérieurement.

Les détails sur la transmission des messages sont bien résumés dans la spécification de cet algorithme. Ils stipulent que le système évolue en sélectionnant à chaque instant un nœud arbitraire

pour exécuter l'algorithme, parmi ceux ne faisant pas partie de l'arbre, mais ayant déjà des voisins qui y sont déjà. Puis il choisit arbitrairement l'un de ses voisins comme parent.

Dans cette spécification, deux propriétés sont a vérifié : La première est une propriété de sûreté qui stipule que : l'algorithme n'introduit jamais de cycle dans la relation parent. La deuxième est une propriété de vivacité qui stipule que : l'algorithme finira toujours par calculé un arbre couvrant de tout réseau.

Modélisation avec Electrum ordinaire et Electrum avec action . Les modèles complets sont présentés dans l'annexe B.3. En ce qui concerne la modélisation dans Electrum ordinaire. Chaque signature représente un type de base et des relations dont la première colonne est dudit type. Ainsi :

- la signature abstraite `Msg` représente un type de message qui peut-être exclusivement une requête ou un accusé de réception. `Req` définit les messages de type requête, tandis que `Ack` définit les messages de type accusé de réception. Ces deux signatures sont des singletons(mot-clé `one`) car dans un modèle il n'y a qu'une seule instance de type, et des sous-signature de la signature `Msg` ;
- la signature `Node` représente les nœuds du réseau. Les relations `to` et `from` représentent respectivement l'ensemble de liens entrants et sortants du nœud. Par ailleurs, les signatures variables t non disjoint `Waiting`, `Active`, `Contending`, `Elected` représentent respectivement le sous-ensembles des nœud ayant les statuts équivalents.
- la signature `Link` représente l'ensemble des liens. Elle possède les relations `target` et `source` qui associent à un lien ses nœuds d'extrémité ; la relation `reverse` représente son lien dual. Des faits supplémentaires tant dans `Node` que ans `Link` assurent la cohérence des différentes relations. La signature `ParentLink` définit des liens particuliers considérés comme parent ;
- la file d'attente de messages des différent nœuds est modélisée par la signature `Queue`. Chaque file contient potentiellement un slot sur lequel les messages sont enregistrés, il est modélisé par la relation `slot`. Par ailleurs, la variable `Overflow` indique le dépassement du slot d'une file.
- la topologie du réseau est un graphe non orienté modélisée par la signature singleton `Tree`. Elle possède une fonction partielle `tree` qui permet de construire l'intégralité de l'arbre à partir de la racine.

  Les exigences sur la topologie du réseau sont modélisées par des faits. A titre illustratif, le fait locale de la signature `Tree`
- la signature abstraite `Op` introduit un ensemble de noms pour les opérations du protocole. Ceci est simplement pour des fin de commodité, en effet elle nous permet de demander une exécution dans laquelle des opérations spécifiques se produisent. Les signatures singletons `Init, AssignParent, ReadReqOrAck, Elect, WriteReqOrAck, ResolveContention, Stutter` représentent respectivement les opérations : d'initialisation ; d'affectation d'un parent à un nœud ; de lecture de message ; d'élection du leader, d'écriture de message, de résolution de conflit, et une opération de saut.

Firewire Ce modèle décrit un protocole d'élection de leader utilisé dans Firewire, un logiciel IEEE.(norme de connexion des appareils électroniques grand public)

Le réseau est supposé consister en un ensemble de nœuds connectés par des liens. Chaque lien entre une paire de nœuds (lien entrant) est associé à un lien dual(lien sortant). En considérant un lien et son dual comme un seul arrêt non orienté, la topologie du réseau serait un arbre.

Le but du protocole est de construire un tel arbre. Dans le modèle, cela est réalisé en marquant un sous-ensemble de liens en tant que liens parent (chacun pointant d'un nœud vers son parent) et en marquant un seul nœud comme racine.

Le protocole fonctionne de la façon suivante : Lorsqu'il nœud s'aperçoit que tous ses liens entrants (ou tous sauf un) ont été marqués comme des liens parents,il envoie un message sur chacun de ses liens sortants. Soit un accusé de réception (indiquant sa volonté d'agir en tant que parent), soit une requête (indiquant son désir d'être un enfant), selon que le dual du lien sortant a été marqué ou non.

Les nœuds feuilles (avec un seul lien entrant) peuvent ainsi initialiser le protocole en envoyant des requêtes aux nœuds adjacents. Cette action modifie le statut d'un nœud de *en attente* à *actif*. Un nœud en attente qui reçoit un message sur un lien peut étiqueter ce-dernier comme un lien parent. Une fois actif, si un nœud reçoit un accusé de réception sur un lien il peut également marquer le lien, mais s'il reçoit une requête, son statut devient par *en conflit*.

La résolution des conflits est modélisée simplement par une seule action qui identifie de manière arbitraire l'un des deux liens ayant une paire de nœuds en conflit. Enfin, un nœud dont tous les liens entrants sont des liens parent se désigne comme une racine. Les trois propriétés à vérifier sur le système sont : (1) dans chaque état d'exécution, au plus un nœud qui a été élu ; (2) il existe un état dans lequel un nœud a été élu (le leader finira pas être élu) ; (3)aucune file d'attente ne déborde.

Élection du leader ce système à été déjà décrit dans le chapitre 4. Le modèle de ce système est nommé ring, juste pour signifier que la topologie du réseau doit être un anneau.

Hôtel ce modèle sert de fil rouge dans la description des différents langages abordés dans cette thèse.

Pour évaluer besoins l'impact de la couche action sur Electrum Nous avons effectué des expériences pour une première évaluation de l'impact de l'utilisation et de la mise en œuvre des actions telles qu'elles sont définies en l'état actuel. Ces expériences ont été réalisées sur des modèles classiques d'Alloy présentés dans la figure 6.5. Les modèles Electrum standard et Electrum action sont disponibles en annexes [B.1, B.2,B.3] Elles ont consisté à comparer leur modélisation temporelle à la fois avec Electrum standard et Electrum avec actions. En tout état de cause, il ne s'agit que d'une première étape, des évaluations supplémentaires seront nécessaires à long terme. Les expériences ont deux principaux objectifs :

1. évaluer l'impact de la couche action sur la modélisation pour l'utilisateur final ;
2. évaluer la performance de la sémantique implémentée (en s'appuyant sur les signatures et les relations variables décrites à la section section 6.3).

6.4.2 Modélisation

Pour l'instant, l'impact (voir Figure 6.5) sur la modélisation n'a été évalué que par nous-mêmes et quelques collègues déjà familiarisés avec Electrum standard, ou tout au moins avec la spécification formelle. Ainsi, la question de connaître l'impact de la couche action sur la modélisation par un débutant est laissé pour des travaux futurs.

Modèle	Electrum	Electrum Action
ring	71	67
hotel	65	57
spantree	111	101
firewire	302	257

FIGURE 6.5 – Comparaison en terme de nombre de lignes du modèle

Néanmoins, avec la couche action les utilisateurs expérimentés ont la latitude de se focaliser uniquement sur les aspects concernant leurs modèles. Il serait utile que des prédicats ad hoc soient généré pour spécifier des contraintes d'équité fortes et faibles. En outre, le développement de la couche action s'est avéré avantageux pour la visualisation des instances et des contre-exemples de modèle, car à chaque instant l'action déclenchée est matérialisée graphiquement.

6.4.3 Vérification

En termes de performances, les résultats ont été plus mitigés (voir Fig 6.6). Nous rapportons des tests effectués sur une machine utilisant un processeur Intel Xeon E5-2699 dual fournissant 36 threads et 512 Go de RAM. Nous avons évalué diverses formes de propriétés, certaines où le problème devrait être SAT et certaines où il devrait être UNSAT :

- la cohérence de chaque modèle est vérifiée à l'aide de la commande `run` ;
- une propriété de sûreté (safety) ou de vivacité (liveness) présumée erronée (c'est-à-dire SAT) est vérifiée avec une commande `check` ;
- certaines propriétés de sûreté ou de vivacité présumées vraies (UNSAT) sont vérifiées avec la même commande `check`. Il faut noter que les propriétés de vivacité sont réputées difficiles à vérifier.

BMC

Les analyses de BMC sur le Xeon ont dû s'appuyer sur SAT4J en raison d'une limitation de la plate-forme. En effet, les résultats sont meilleurs lorsqu'on utilise Minisat ou Glucose sur le Core i5, avec un facteur de 2 à 3. Nous avons comparé pour chaque modèle des versions avec et sans actions. Pour des raisons d'évaluation, les bornes de toutes les signatures ont été fixées respectivement à 3, 4 et 5, ce qui n'est pas toujours réaliste. Néanmoins, dans certains cas, l'utilisateur sait qu'attribuer une petite borne à certaines signatures est suffisant pour plus d'assurance dans la conception. Pour la vérification BMC, nous avions fixé le `scope` de la signature `Time` à vingt (20), c'est-à-dire qu'une trace d'exécution aura au plus vingt états, ce qui peut sembler élevé par rapport aux modèles classiques Alloy et Electrum standard. Cependant, nous faisons ce choix pour prendre en compte la sémantique d'entrelacement (établir la satisfiabilité d'une propriété peut nécessiter plus d'instants).

UMC

La vérification UMC est basée sur une compilation vers SMV et s'appuie sur le model checker nuXmv avec l'algorithme k-liveness. Nous avons vérifié deux versions : la première est basée sur un seul modèle SMV généré. La seconde utilise la fonctionnalité de décomposition [MCP17] de

Pardinus, pour générer plusieurs modèles SMV, un pour chaque configuration (toutes les instanciations possibles du modèle statique modulo les symétries). En principe, cette fonctionnalité pourrait également être utilisée avec l'analyse BMC basée SAT. Cependant, elle a toujours été non efficace dans nos expériences, car l'heuristique de décomposition de Pardinus n'a pas encore été optimisée pour Electrum. Néanmoins, elle reste utile pour UMC en raison de la complexité de ces problèmes (à défaut d'être efficace, elle est un apport considérable pour essayer de réduire la complexité du problème).

UMC vs BMC

En comparant BMC et UMC (décomposés ou non), le BMC est comme on peut s'y attendre généralement beaucoup plus rapide, en particulier pour les modèles SAT. En outre, de nombreux problèmes UNSAT peuvent être résolus dans un délai raisonnable même avec UMC. Enfin, les actions implémentées en tant que signatures et champs sont avantageuses relativement à la visualisation, mais sont assez coûteuse par rapport aux solutions classiques reposant sur des prédicats, en particulier pour BMC (cela varie pour les UMC).

6.5 Discussion

Dans ce chapitre nous avons implémenté une couche action au-dessus de Electrum standard afin de faciliter la spécification du comportement des systèmes dynamiques ayant des propriétés structurelles riches.

Electrum et *sa couche action* en particulier est inspiré du langage de spécification TLA+. Cependant, outre les différences significatives entre les langages TLA+ et Electrum standard soulignées dans [Alca], la sémantique des actions diffère d'un langage à l'autre.

En effet, dans TLA+ il y a une équivalence entre les notions de déclenchement de l'occurrence d'une action et de satisfaction du corps de l'action, tandis que dans Electrum action, on a une simple implication entre la première et la seconde. Notre sémantique permet de distinguer deux occurrences d'action ayant le même effet. Par ailleurs, notre modèle temporel n'a pas la même sémantique que celui de TLA+, car contrairement à TLA+, on impose qu'une seule action soit exécutée à chaque instant.

DynAlloy propose une syntaxe des actions, similaire à celle décrit dans ce chapitre. Cependant, tout comme TLA+ les sémantiques diffères au niveau des modèles temporels et du déclenchement des actions. Le modèle temporel dans DynAlloy est défini par des constructions de langages de programmation impératifs . Contrairement à DynAlloy, Electrum action préserve dans une certaine mesure la flexibilité d'Alloy.

Nous avons réalisé des expérimentations sur des systèmes classiques (cf tableau6.5 : systèmes typiquement académiques) d'Alloy, en comparant (cf tableau6.6) les modèles Electrum standard et les modèles Electrum action. De cette expérimentation il en ressort qu'avec la couche action l'efficacité des analyses est souvent réduite pour des propriétés SAT.

Dans la deuxième contribution de cette thèse (chapitre 7) nous effectuons une analyse formelle de la correction du protocole de recherche distribuée Chord, avec Electrum munit de la couche action. Ces travaux sans être considérés comme de simples études de cas d'Electrum action, ont permis

Specification	Bound	Kind	Expect	Plain BMC	Plain UMC	Plain UMC-d	Using actions BMC	Using actions UMC	Using actions UMC-d
ring-0	3	C	SAT	0.12	0.13	0.16	0.13	0.11	0.17
ring-1	4	C	SAT	0.12	0.12	0.17	0.14	0.13	0.20
ring-2	5	C	SAT	0.15	0.14	0.28	0.13	0.14	0.29
ring-3	3	L	SAT	0.12	0.12	0.15	0.13	0.12	0.15
ring-4	4	L	SAT	0.13	0.14	0.18	0.13	0.12	0.17
ring-5	5	L	SAT	0.13	0.14	0.25	0.15	0.14	0.30
ring-6	3	L	UNSAT	6.99	6.74	3.89	3.30	3.25	4.40
ring-7	4	L	UNSAT	57.08	55.62	17.82	48.55	48.43	21.14
ring-8	5	L	UNSAT	451.25	446.17	t/o	t/o	t/o	419.53
ring-9	3	S	UNSAT	3.90	3.81	3.51	2.78	2.77	3.82
ring-10	4	S	UNSAT	9.07	8.97	5.43	2.98	3.03	5.53
ring-11	5	S	UNSAT	27.24	27.70	21.44	3.29	3.06	9.19
hotel-0	3	C	SAT	0.20	0.23	0.55	0.37	0.33	0.69
hotel-1	4	C	SAT	0.29	0.26	0.49	0.80	0.90	1.04
hotel-2	5	C	SAT	0.32	0.30	0.66	2.58	2.54	1.37
hotel-3	3	S	SAT	0.34	0.34	1.04	0.97	0.91	1.55
hotel-4	4	S	SAT	0.47	0.51	1.41	6.24	6.16	1.64
hotel-5	5	S	SAT	0.58	0.58	1.16	29.39	29.72	1.94
hotel-6	3	S	UNSAT	7.55	6.89	20.82	14.20	14.09	34.67
hotel-7	4	S	UNSAT	71.15	68.66	165.45	277.69	277.56	264.65
hotel-8	5	S	UNSAT	366.81	396.44	t/o	t/o	t/o	t/o
spantree-0	3	C	SAT	0.13	0.13	0.16	0.13	0.12	0.15
spantree-1	4	C	SAT	0.16	0.13	0.21	0.13	0.12	0.19
spantree-2	5	C	SAT	0.16	0.14	0.21	0.14	0.14	0.27
spantree-3	3	L	SAT	0.13	0.12	0.16	0.13	0.12	0.16
spantree-4	4	L	SAT	0.13	0.13	0.17	0.13	0.15	0.18
spantree-5	5	L	SAT	0.14	0.16	0.23	0.14	0.13	0.26
spantree-6	3	L	UNSAT	9.24	1.51	0.71	8.10	2.99	1.30
spantree-7	4	L	UNSAT	116.55	1.79	2.56	59.56	5.34	5.20
spantree-8	5	L	UNSAT	t/o	5.08	2.59	t/o	50.91	10.03
spantree-9	3	S	UNSAT	5.23	5.12	4.69	5.44	5.40	5.13
spantree-10	4	S	UNSAT	31.95	31.00	29.31	35.94	35.82	25.24
spantree-11	5	S	UNSAT	123.12	124.12	t/o	155.73	153.72	t/o
firewire-0	3	C	SAT	0.20	0.19	0.29	0.18	0.17	0.31
firewire-1	4	C	SAT	0.23	0.20	0.34	0.18	0.19	0.33
firewire-2	5	C	SAT	0.23	0.25	0.39	0.21	0.21	0.40
firewire-3	3	S	UNSAT	29.07	28.43	65.77	51.72	50.65	133.33
firewire-4	4	S	UNSAT	t/o	t/o	t/o	t/o	t/o	t/o
firewire-5	5	S	UNSAT	t/o	t/o	t/o	t/o	t/o	t/o
firewire-6	3	L	SAT	0.20	0.19	0.30	0.17	0.17	0.30
firewire-7	4	L	SAT	0.22	0.21	0.34	0.16	0.20	0.33
firewire-8	5	L	SAT	0.25	0.25	0.32	0.21	0.22	0.39
firewire-9	3	L	UNSAT	131.54	129.51	216.75	43.03	41.13	93.28
firewire-10	4	L	UNSAT	519.33	527.71	t/o	t/o	t/o	t/o
firewire-11	5	L	UNSAT	t/o	t/o	t/o	t/o	t/o	t/o

FIGURE 6.6 – Résumé des tests effectués (20 Time pour les scénarios bornés) ; C désigne la "Cohérence", S la "sécurité" et L la "vivacité". L'unité du temps est la seconde (time-out : 10 m) ; UMC-d est le UMC en mode décomposition(avec au maximum 36 threads)..

d'avoir une première appréciation sur l'utilisation d'Electrum action pour l'analyse de systèmes réels.

Chapitre 7

Analyse de la propriété de vivacité fondamentale du protocole Chord

7.1 Motivations

L'application Bittorrent est utilisée par plusieurs milliers d'internautes à travers le monde, pour le téléchargement des fichiers volumineux. Le dysfonctionnement d'un tel service peut engendrer des désagréments considérables, par exemple : le retard du démarrage ou le ralentissement du téléchargement des fichiers , l'indisponibilité des fichiers supposés accessibles, la corruption des fichiers téléchargés. Dans [Vil] certaines origines du problème, sont énoncées notamment : la saturation des serveurs engendrée par une distribution déséquilibrée des données sur l'ensemble des nœuds du réseau, une mauvaise gestion du départ des nœuds du réseau, l'incohérence des données stockées dans ces nœuds. À l'instar de cette application plusieurs autres applications peer-to-peer sont victimes de problèmes similaires, ce qui peut susciter les interrogations suivantes :

— quels sont les problèmes fondamentaux dans la conception des protocoles peer-to-peer responsables du dysfonctionnement répétés des applications ?

— comment prouver la correction du fonctionnement de ces protocoles ?

— les méthodes d'analyse utilisées sont-elles adaptées à ces problèmes ? Prennent-elles en compte toutes les configurations et les comportements possibles du système ?

Les problèmes fondamentaux dans la conception des protocoles peer-to-peer sont : (1) la spécification de sa *propriété de correction*. Il s'agit de définir une propriété qui garantit l'accessibilité des données sous certaines hypothèses fixées. Par exemple, la correction de Chord est l'éventuelle accessibilité sous l'hypothèse qu'il n'y a pas de fracture du réseau en plusieurs composantes ; (2) le choix d'une structure de donnée garantissant la cohérence des données et l'évolutivité tout en facilitant les recherches ; (3) la preuve de la correction du fonctionnement du protocole.

Le fonctionnement correct est assuré par une preuve formelle ou tout au moins une vérification. Néanmoins, les auteurs des protocoles peer-to-peer effectuent trop souvent des preuves par des méthodes numériques (simulation) ou des preuves manuelles qui peuvent être sujettes aux erreurs. Les méthodes formelles [CGL94] fournissent des techniques systématiques et rigoureuses pour spécifier et vérifier la conception d'un système logiciel. Elles peuvent mettre en évidence des problèmes d'inconsistance, d'ambiguïté ou d'incomplétude au sein des spécifications d'un système. Étant donnée la

complexité réaliste et la spécification quasi-informelle des protocoles peer-to-peer, une analyse par des méthodes formelles est très adaptée, en outre elle augmenterait considérablement la confiance en sa correction.

Pour ce qui est de l'efficacité de la méthode d'analyse, le choix peut se baser sur les critères suivants :

1. la nature du système : statique, dynamique ou dynamique avec des propriétés structurelles ;
2. le type de la propriété de correction : sûreté, vivacité, hybride ;
3. le degré d'automatisation : manuel, semi-automatique ou complètement automatique ;
4. l'horizon temporel : statique, borné, non borné ;
5. le domaine d'analyse : fini ou infini

Par exemple, Chord est un protocole dynamique possédant de riches propriétés structurelles, sa correction est une propriété de vivacité, ainsi, une méthode d'analyse idéale serait une vérification complètement automatisée sur horizon de temps non borné sur un domaine non borné. Cette analyse requiert donc un langage de spécification des propriétés tant structurelles que temporelles, idéalement accompagné d'un outil de vérification complètement automatisé, sur horizon de temps non borné.

Dans ce chapitre, nous présentons une analyse de la correction du protocole Chord présenté au chapitre 5 avec Electrum muni de sa couche action présentée au chapitre 6.

7.2 Introduction

Dans le chapitre 5, nous avons présenté Chord, un protocole réalisant une table de hachage distribuée évolutive sur un réseau peer-to-peer sous-jacent.

Selon ses auteurs, Chord se distingue par trois principales caractéristiques à savoir : sa simplicité, sa performance et sa correction facilement prouvable. Prouver la correction de Chord s'est avéré être une tâche beaucoup plus difficile, comme le montre P. Zave dans ses différents travaux. [Zav19, Zav11, Zav12, Zav15b, Zav15a, Zav17].

En effet, la correction du protocole Chord concerne à la fois la topologie du réseau et son évolution temporelle. Cette double nature est l'une des raisons qui justifie la difficulté à prouver la correction de Chord.

En ce qui concerne la spécification et la vérification du protocole Chord, les travaux antérieurs ont principalement porté sur des preuves automatiques des propriétés de sûreté [Zav19],[Zav11], [Zav12], [Zav15b], [Zav15a] ou des preuves manuelles de la correction complète du protocole [Zav17] [KEAAH05](une propriété de vivacité). Dans ce chapitre, nous rapportons une analyse automatique de la correction du protocole Chord avec le langage Electrum muni de sa couche action sur de petits exemples de réseaux. En particulier, nous avons trouvé différents bugs dans les travaux antérieurs et montré que le protocole tel qu'il était décrit dans ces travaux n'était pas correct. Nous avons corrigé tous ces problèmes et fourni une version du protocole pour laquelle notre approche n'a produit aucun contre-exemple. Nous présentons les avantages de cette approche à savoir :

— la capacité d'Electrum à gérer les aspects structurels facilite considérablement la spécification de la topologie du réseau ;

— la capacité d'Electrum à traiter les aspects temporels cadre bien avec la spécification de l'évolution du réseau (tout au long de l'exécution des opérations de maintenance) et facilite la spécification de la propriété de correction, qui est une propriété de vivacité ;
— la vérification automatique de la correction complète est effectuée pour la première fois (toutefois, pour un nombre limité de nœuds) ;
— grâce au retour rapide d'Electrum Analyzer, nous avons pu détecter plusieurs bugs dans les formalisations antérieures du protocole. Nous avons aussi identifié précisément les hypothèses temporelles sur l'ordre des opérations de maintenance nécessaires pour garantir la correction.

Le reste de ce chapitre est structuré de la façon suivante. Dans la section 7.3, nous allons fournir une spécification de la structure et des propriétés d'un réseau Chord. Les événements du protocole Chord seront présentés dans la section 7.4, le modèle temporel dans la section 7.5 et l'analyse de la correction du protocole dans la section 7.6. Nous poursuivrons en section 7.7 avec une évaluation des résultats de la vérification formelle de notre modèle. Dans la section 7.8, nous allons mettre en évidence les aspects importants de notre étude en le comparant à d'autres travaux connexes.

7.3 Formalisation du protocole Chord

Les éléments fondamentaux d'un réseau Chord identifiés au chapitre 5 sont : (1) sa topologie (un anneau dans l'état idéal) formée de nœuds possédant un identifiant, une liste de successeurs et éventuellement un prédécesseur ; (2) les propriétés qui caractérisent ses deux états acceptables notamment l'état "idéal" et l'état "valide" ; (3) les événements qui font évoluer le réseau, notamment : `join` pour rejoindre le réseau, `fail` pour quitter le réseau, `stabilize` et `rectify` pour la maintenance de la topologie du réseau ; (4) l'hypothèse de tolérance aux pannes garantissant que chaque membre du réseau a toujours au moins un successeur ; (5) la propriété de correction du protocole.

Dès à présent, nous allons formaliser ces aspects fondamentaux du protocole Chord en nous inspirant à la fois de la description du protocole Chord dans [LNBK02] (appelé PODC pour le reste du chapitre) et les travaux récents de P. Zave [Zav17].

7.3.1 Structures de données

Spécification des nœuds d'un réseau Chord

Dans un réseau Chord, chaque nœud possède une liste de successeurs. Pour que le réseau soit toujours connecté, la taille de cette liste doit être au moins égale à deux.

Dans cette thèse, nous considérons les listes à deux éléments. Ce choix a peu d'impacts sur la modélisation (la spécification de la stabilisation est indépendante de la taille réelle de la liste des successeurs des nœuds), mais diminue la complexité de l'analyse des modèles.

Les opérations de maintenance requièrent un prédécesseur `prdc` pour chaque nœud . Ces trois relations sont variables, chacune d'elle désigne une fonction partielle dans l'ensemble des nœuds, spécifiée par la multiplicité `lone`.

Remarque 12. *On rappelle qu' en Electrum, une* ***relation*** *entre ensembles est spécifiée par un* ***champ*** *dans une signature. Dans la suite, nous utiliserons indifféremment les deux termes.*

Les opérations `join` et `fail` sont instantanées et ne peuvent donc être mises en attente.

En revanche, à cause de la simultanéité des opérations, un nœud peut être obligé d'interrompre une opération de maintenance pour la reprendre ultérieurement. Cette opération ainsi que ses paramètres (le type d'opération, le nœud concerné) sont stockés dans une boîte aux lettres attribuée à chaque nœud. Cette boîte aux lettres est modélisée par le champ variable `todo`.

Un statut permet d'indiquer l'opération mise en attente, il vaut : `Stabilizing` s'il s'agit d'une opération de stabilisation et `Rectifying` pour une opération de rectification. On distingue un seul statut ayant deux types de valeurs possibles, il équivaut donc à une énumération de `Stabilizing` et `Rectifying`, ils sont modélisés par les signatures `Status`, `Stabilizing` et, `Rectifying`.

En somme, la signature des nœuds est donnée par l'expression :

```
sig Node {
  var fst, snd, prdc: lone Node,
  var todo: Status → Node }
```

```
abstract sig Status {}
one sig Stabilizing,
one Rectifying extends Status {}
```

Remarque 13. *Le champ `todo` est un mécanisme de modélisation, il n'est pas partie prenante de la description du protocole par ses auteurs [LMP06],[SMK+ 01],[SMLN+ 03].*

Spécification des membres d'un réseau Chord

Dans cette modélisation, nous avons choisi de définir un membre du réseau comme étant un nœud dont chacun des deux successeurs `fst` et `snd` pointe effectivement vers un autre nœud. En effet, nous avons privilégié l'utilisation des champs `fst` et `snd` pour tirer profit de l'opération de départ (il met `fst` et `snd` à null) dont l'exécution met automatiquement à jour l'ensemble des membres du réseau.

```
var sig members in Node {}
fact membersDef {
  always members = { n : Node |
   some n.fst and some n.snd } }
```

Chaque nœud possède une liste à deux successeurs (`fst`, `snd` dans ce cas). Cependant, à chaque instant un seul est utilisé pour communiquer dans le réseau. Nous avons nommé ce nœud "*successeur actuel ou successeur tout simplement*". D'après les auteurs Chord [SMK+01],[SMLN+03], [LNBK02], le successeur d'un nœud est le premier membre de sa liste de successeurs. Nous l'avons modélisé par la fonction partielle `succ`. Explicitement, elle stipule que : le successeur d'un nœud est son premier successeur `fst` si celui-ci est membre, et son second successeur `snd` sinon.

Remarque 14. *La restriction de cette fonction à l'ensemble des membres donne une fonction totale. En effet, le contraire signifierait l'existence d'un membre du réseau qui n'a pas de successeur membre du réseau, ce qui violerait l'hypothèse de tolérance aux défaillances définie par Chord [LNBK02].*

Spécification des membres de l'anneau d'un réseau Chord

La définition de la fonction successeur (`succ`) permet de spécifier l'ensemble des membres de l'anneau à chaque instant, qui est en réalité un sous-ensemble des membres du réseau dont les nœuds sont mutuellement atteignables à travers la relation successeur (`succ`). Autrement dit, un membre de l'anneau est accessible via la fermeture transitive de `succ` à partir de lui-même. Il est donc introduit par la signature variable `ring` qui est incluse dans la signature `members`.

```
fun succ: Node → lone Node {
  {m1, m2: members | m1.fst in
    members implies m2 = m1.fst
      else m2 = m1.snd } }
```

```
var sig ring in members {}
fact ringDef {
  always ring = { m : members |
    m in m.^succ } }
```

Spécification des appendices dans un réseau Chord

Lorsque la topologie du réseau n'est pas encore idéale, on peut avoir des membres hors de l'anneau, mais qui seront insérés ultérieurement par les opérations de maintenance. L'ensemble de ces membres constituent des *appendices*. Dans cette modélisation, ils sont introduits par la fonction `appendages`

```
fun appendages: set Node { members − ring }
```

Dans la section suivant nous allons spécifier les propriétés qui caractérisent un réseau Chord bien formé. Nous allons définir l'ordre sur l'espace des identifiants des nœuds du réseau, cet ordre sera utilisé pour spécifier les propriétés caractéristiques des réseaux Chord.

7.3.2 Propriétés du réseau

L'ordre sur l'espace des identifiants des nœuds

Dans notre modélisation, nous assimilons les nœuds à leurs identifiants, au sein de la signature `Node`. Pour modéliser la structure de l'espace des identifiants décrit dans le chapitre 5, nous avons définit le prédicat ternaire `between` et la fonction `nextNode` en se reposant sur un ordre total sur la signature `Node`, défini dans le module `util/ordering`.

```
open util/ordering[Node] // ordre(binaire) total
pred between [n1, nb, n2: Node] {
  lt[n1, n2] implies (lt[n1, nb] and lt[nb, n2])
    else (lt[n1, nb] or lt[nb, n2]) }

fun nextNode: Node → Node {
  { n, m: Node | no next[n]
      implies m = first else m = next[n] } }
```

`between` étend l'ordre total `util/ordering` qui est intégré dans le modèle à l'aide de la primitive `open`. Il est défini par une expression conditionnelle. Un nœud est situé entre le plus petit et le plus grand identifiant dans l'un des deux cas suivants :

1. soit le nœud est plus grand que le plus grand identifiant actuel, auquel cas, il dévient le plus grand identifiant et prend pour successeur le plus petit identifiant ;
2. soit il est plus petit que le plus petit identifiant actuel, il devient donc, le plus petit identifiant et le prédécesseur du plus grand identifiant.

Dans les autres cas, le nœud doit être plus grand que le plus petit et plus petit que le grand.

De la même manière, nous aurons souvent besoin de comparer les nœuds en vérifiant si un nœud se situe entre deux autres, ceci est défini par l'ordre sur l'ensemble des nœuds exprimé par la fonction totale `nextNode`.

Propriétés caractéristiques des réseaux Chord

Les propriétés d'un réseau Chord caractérisent ses deux états acceptables à savoir `Valid` et `Ideal`. L'état `Valid` est caractérisé par une conjonction de cinq propriétés indiquant respectivement que : (1) il y a au moins un anneau ; (2) il y a au plus un anneau ; (3) tout nœud de l'appendice peut atteindre un membre de l'anneau en suivant les pointeurs successeur ; (4) non-bouclage : entre un membre de l'anneau et son successeur, aucun autre membre de l'anneau ne doit figurer. Autrement dit, la relation successeur respecte l'ordre des identifiants de nœuds de l'anneau ; (5) validité de la liste de successeurs : le premier successeur d'un membre est situé entre le membre lui-même et son second successeur. Notez que la conjonction de (1) et (2) correspond à la notion de connectivité (le réseau n'est pas fracturé en plusieurs composants).

Ces propriétés sont modélisées respectivement par les prédicats : `atLeastOneRing`, `atMostOneRing`, `connectedAppendages`, `orderedRing`, et `orderedSuccessors` définis comme suit

```
1  pred valid { atLeastOneRing and atMostOneRing and
2    orderedRing and connectedAppendages and
3    orderedSuccessors }
4
5  pred atLeastOneRing { some ring }
6
7  pred atMostOneRing { all m1, m2: ring | m1 in m2.^succ }
8
9  pred orderedRing { // = non−bouclage
10   all disj m1, m2, mb: ring |
11   // 'disj' = 'all different'
12     m2 = m1.succ implies not between[m1, mb, m2] }
13
14 pred connectedAppendages {
15   all m1: appendages | some m2: ring | m2 in m1.^succ }
16
17 pred orderedSuccessors { // validite de la liste de successeurs
18   all m: members | between[m, m.fst, m.snd] }
```

La Figure 7.1 présente un réseau `Valid` (à gauche) ayant six membres et un appendice, tandis que la figure de (à droite) présente un réseau `Ideal`, un anneau ayant cinq membres

Un réseau idéal est un réseau `Valid` tel que : (1) les membres sont tous dans l'anneau, c'est-à-dire qu'il n'y a pas d'appendice ; (2) les fonctions `fst` et `prdc` sont réciproques (c'est la cohérence locale) ; (3) la liste des successeurs de tout membre contient les deux premiers nœuds qui le suivent dans l'anneau. La modélisation de la propriété `Ideal` donne :

```
1  pred ideal {
2    valid
3    no appendages
4    fst = ~prdc
5    all m: members {
6      m.snd + m.fst in members
7        m.snd = m.fst.fst
8    }
9  }
```

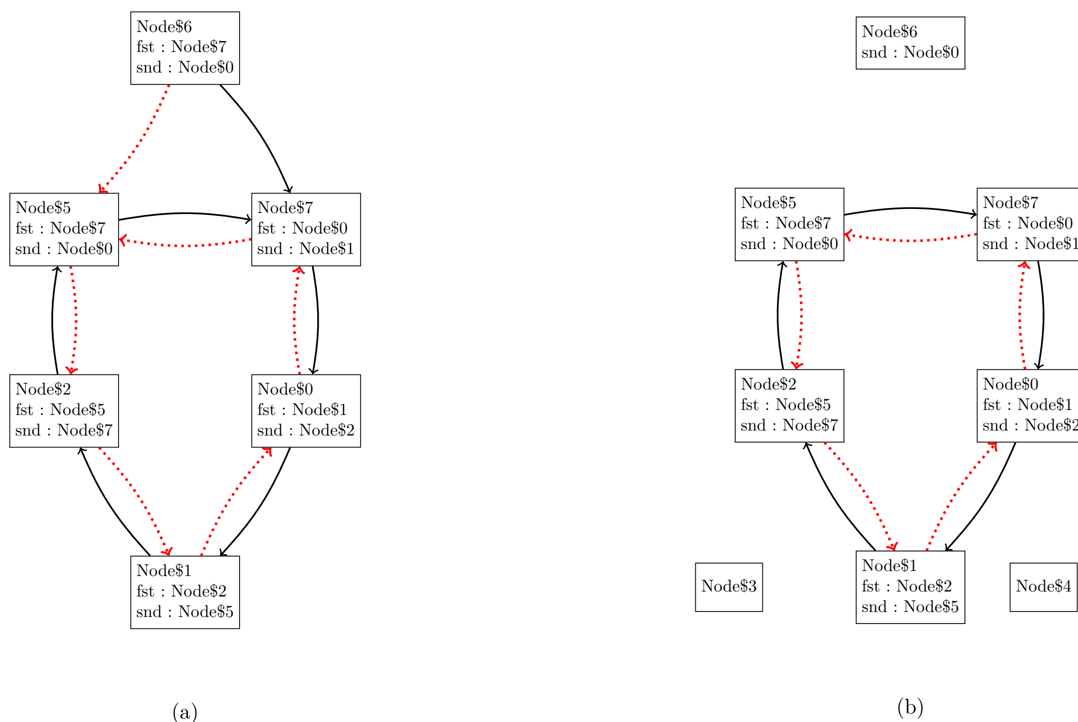


FIGURE 7.1 – Exemple d'un réseau Valid (gauche) et Ideal (droit)

Remarque 15. *Pour des raisons de lisibilité les champs `fst` et `snd` sont affichés à l'intérieur du nœud. Au niveau de l'abstraction, nous avons décidé qu'aucune contrainte ne sera imposée sur les nœuds qui sont hors du réseau. Par exemple dans la figure 7.1 malgré la présence des nœuds non membre $Node\$3$, $Node\$4$, $Node\$6$ et le fait que $Node\$6$ possède un second successeur le réseau est `Ideal`.*

Dans la section suivante, nous allons utiliser Electrum muni de sa couche action (chapitre 6) pour modéliser les événements du protocole Chord. En effet, cette couche rend la spécification des transitions du système facile et assez claire.

7.4 Les événements du protocole Chord

Notre modélisation s'inspire de celle de P. Zave, nous avons donc tiré profit de ses hypothèses de modélisation. Précisément, nous nous assurons que toutes les opérations sont atomiques, en autorisant la communication avec un seul nœud par opération.

7.4.1 Opération join

L'arrivée d'un nœud dans le réseau est modélisée par l'action `join`. Elle modifie les champs `fst`, `snd`, `prdc` et les signatures variables `members` et `ring`. Dans une action `join`, le nœud `new` qui rejoint le

réseau ne doit pas être membre du réseau. Dans PODC, la description informelle de cet événement déclare que le nouveau nœud `new` contacte un nœud quelconque du réseau et lui demande d'initier une requête pour localiser le nœud `m` tel que `new` soit entre `m` et son premier successeur (`m.fst`).

Dans notre modélisation, nous faisons abstraction de cette requête, en supposant qu'il existe un oracle qui détermine ce nœud `m`. Cette abstraction n'affecte pas la correction du protocole.

Pour terminer l'action, le nœud `new` obtient ses pointeurs fst et snd du nœud m qu'il garde comme prédécesseur.

```
act join [new: Node] modifies fst, snd, prdc, members, ring {
 new not in members
 some m: members {
    between[m, new, m.fst] and fst' = fst ++ new→m.fst
    snd' = snd ++ new→m.snd and prdc' = prdc ++ new→m}}
```

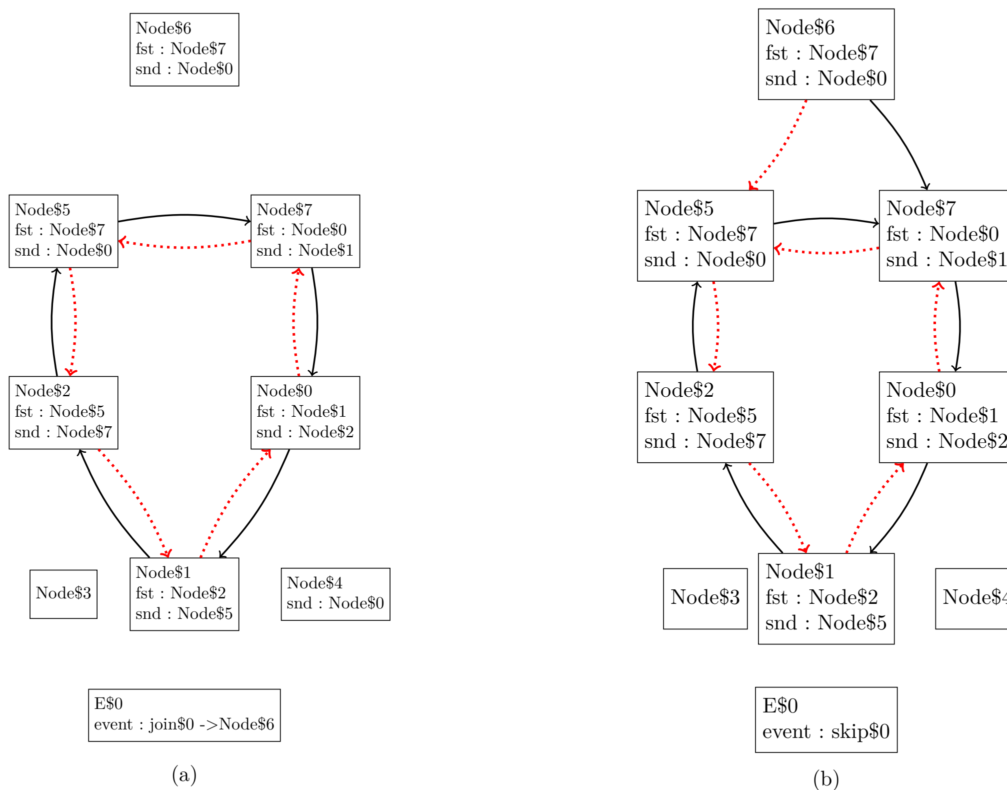


FIGURE 7.2 – Une occurrence de l'action `join`

La figure 7.2 illustre une occurrence de l'action `join`, indiquée dans le champ *event* de la signature événement `E`$0 [1]. Puisque le nœud Node$6 n'est pas encore membre, il déclenche une occurrence de l'action *join*. À l'instant d'après il s'insère ($between[Node\$5, Node\$6, Node\$7]$) entre le nœud

1. Dans un système complexe tel que Chord, connaître l'action exécutée facilite l'analyse du modèle.

Node$5 et son premier successeur Node$7. Il fixe sa liste de successeurs avec celle de Node$5 et garde ce dernier comme prédécesseur (`prdc`). Les champs `fst`, `snd` et `prdc` sont modifiés, ainsi que l'ensemble des membres du réseau `members`(Node$6 devient membre).

7.4.2 Opération fail

Les défaillances ou les départs dans le réseau sont modélisés par l'action `fail`. Dans PODC, une hypothèse déclare qu'un nœud quitte le réseau sans informer son voisinage de son départ. Ainsi, des champs `fst`, `snd` et `prdc` peuvent continuer de pointer sur ce nœud, qui est désormais considéré comme **"mort"**, par opposition aux membres considérés comme des nœuds *"vivants"* .

Par ailleurs PODC n'offre aucun processus systématique (comme pour le join, le stabilize ...) permettant à un nœud de quitter le réseau. Dans notre modélisation, lorsqu'un membre quitte le réseau, ses pointeurs `fst`, `snd` et `prdc` sont supprimés. Ainsi, l'action `fail` modifie les champs `fst`, `snd`, `prdc`, `todo` et les signatures variables `members` et `ring`.

Il faut bien noter que l'hypothèse relative à la tolérance aux pannes qui stipule que tout nœud a au moins un successeur vivant (parmi les nœuds pointés par `fst` et `snd`) peut rendre le départ d'un nœud impossible. Cette hypothèse est simplement spécifiée comme un fait en Electrum :

```
1 fact OperatingAssumption {
2   always {all n: members | some (n.(fst+snd) & members)}
3 }
```

L'action `join` est définie ainsi :

```
act fail [f: Node] modifies fst, snd, prdc, todo, members, ring {
  f in members
  fst' = fst−f→ Node
  snd' = snd−f→ Node
  prdc'= prdc−f→ Node
  todo'= todo−f→ Status→ Node }
```

La Figure 7.3 décrit le scénario de départ du nœud Node$1. Au départ de Node$1, l'hypothèse de tolérance aux pannes est respectée, car le seul nœud Node$0 dont il est le premier successeur possède un second successeur Node$2 en vie. Ainsi, au déclenchement de l'occurrence de `fail`, Node$1 supprime ses pointeurs et quitte le réseau. Il apparaît encore dans la liste de successeurs de Node$0 et demeure le prédécesseur de Node$2 (sera supprimé par les opérations de maintenance).

7.4.3 Opération de stabilisation

Les opérations `join` et `fail` créent des perturbations dans la topologie du réseau, lesquelles sont corrigées par les opérations de maintenance. La stabilisation est l'opération de maintenance chargée de la mise à jour du premier successeur des nœuds.

Inspirée de [Zav17], la stabilisation est divisée ici en deux actions, la première est exécutée périodiquement par chaque nœud pour contrôler l'état de son successeur. La deuxième est exécutée lorsque dans une stabilisation antérieure un nœud a appris que le prédécesseur de son successeur est un meilleur candidat pour être son successeur que son successeur actuel. Le nœud ne pouvant pas communiquer avec ce prédécesseur (il a déjà communiqué avec son successeur dans l'opération), l'opération de modification du successeur est mise en attente et enregistrée dans son champ `todo`.

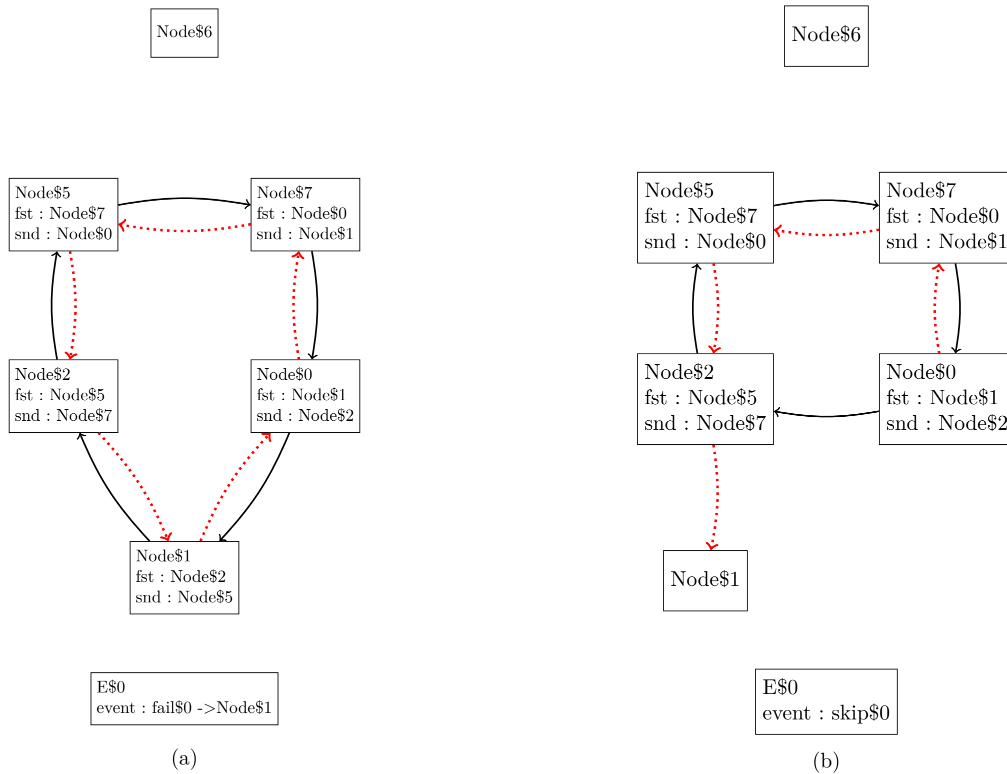


FIGURE 7.3 – Une occurrence de l'action fail

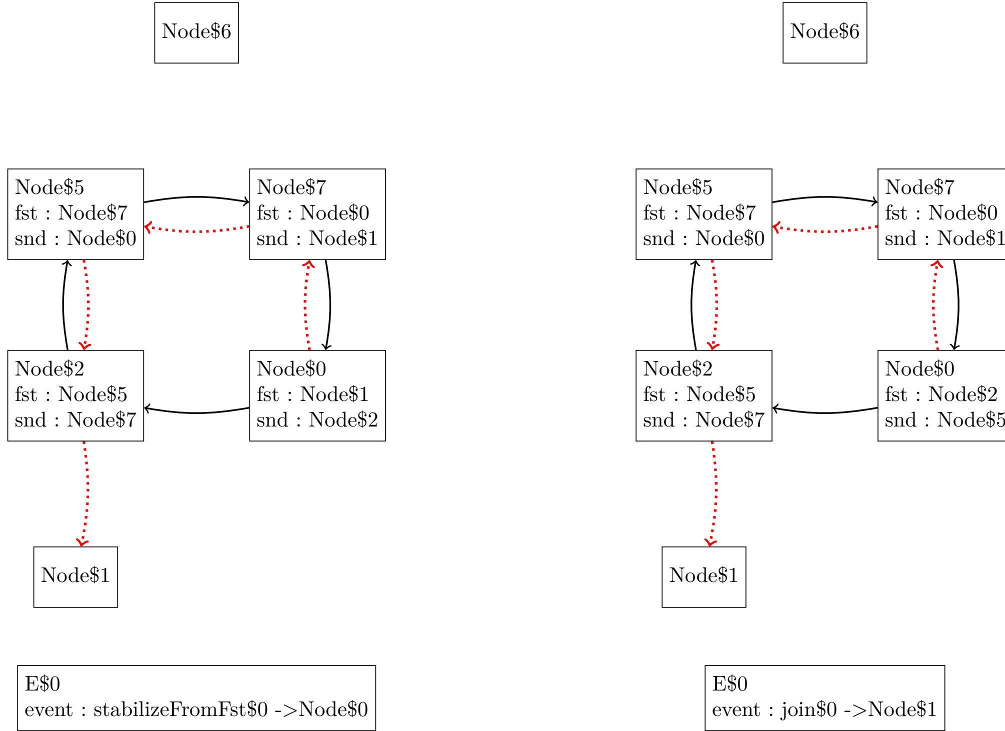
Opération stabilizeFromFst

Lorsqu'un nœud `m` ne possède pas d'opérations en attente, il peut contacter son premier successeur `fst`. Si ce dernier est mort, alors `m` met à jour son premier successeur avec son second successeur `snd`, lequel devra également se mettre à jour. Pour éviter de contacter un autre nœud, on remplace le `snd` par son successeur immédiat dans l'ordre des identifiants (`nextNode`). S'il ne correspond à aucun nœud vivant, alors il sera corrigé ultérieurement par d'autres événements.

Si le premier successeur `fst` est membre, alors sa valeur ne change pas, mais nous mettons à jour le second successeur `snd` (une petite optimisation de la maintenance). Par ailleurs, nous vérifions si le prédécesseur `prdc` du premier successeur `fst` de `m` n'est pas nul et s'il est meilleur (plus proche selon l'ordre between) que le successeur actuel de `m` : si tel est le cas, alors la deuxième partie de la stabilisation `stabilizeFromFstPrdc` est programmée. Sinon, `m` demande à son premier successeur de programmer une rectification avec lui-même : en d'autres termes, le premier successeur s'assure que `m` est bien son prédécesseur. Cette partie de la stabilisation est modélisée par l'expression suivante :

```
act stabilizeFromFst[m: Node] modifies fst, snd, todo, members, ring {
  m in members
  no m.todo.Node // pas d'operation en attente
  m.fst not in members implies {
```

```
    todo' = todo and fst' = fst ++ m->m.snd
    snd' = snd ++ m->nextNode[m.snd] }
  else {fst' = fst and snd' = snd ++ m->m.fst.fst
    (some m.fst.prdc and between[m, m.fst.prdc, m.fst])
      implies todo' = todo + m->Stabilizing->m.fst.prdc
    else todo' = todo + m.fst->Rectifying->m } }
```



(a) Occurrence de `stabilizeFromFst` : le premier successeur de Node$0 est mort.

(b)Occurrence de `join` : le nœud Node$1 rejoint le réseau.

FIGURE 7.4 – Etape de la séquence de correction du réseau

Dans la figure 7.4(a), le nœud Node$1 quitte le réseau et Node$0 dont il était premier successeur, se stabilise : puisqu'il n'a aucune opération en attente, il déclenche une occurrence de `stabilizeFromFst` et découvre que Node$1 est mort. Ainsi, il supprime Node$1 de sa liste de successeurs et met celle-ci à jour avec son second successeur Node$2 et le premier successeur de ce dernier, le nœud Node$5.

Une autre branche de la stabilisation est déclenchée après le retour du nœud Node$1 dans le réseau. Le nœud Node$0 stabilise à nouveau, il contacte son premier successeur Node$2 et découvre que Node$1 serait meilleur premier successeur, il programme la seconde forme de la stabilisation avec Node$1 et l'enregistre dans son champs `todo`.

Opération stabilizeFromFstPdrc

Cette seconde forme de stabilisation intervient lorsqu'une opération de stabilisation a été programmée. Il s'agit d'un cas où un meilleur candidat a été trouvé pour le premier successeur `fst` du nœud `m` pendant une opération `stabilizeFromFst`. La première chose à faire est de vérifier si le candidat ferait encore un meilleur premier successeur `fst`. Par ailleurs, si le candidat n'est plus membre, l'opération est annulée. Autrement, le premier successeur `fst` est mis à jour avec le candidat, et le second successeur `snd` avec le premier successeur `fst` du candidat. Enfin, ce candidat programme une rectification avec `m`, pour modifier ultérieurement son prédécesseur.

Cette action est modélisée ainsi :

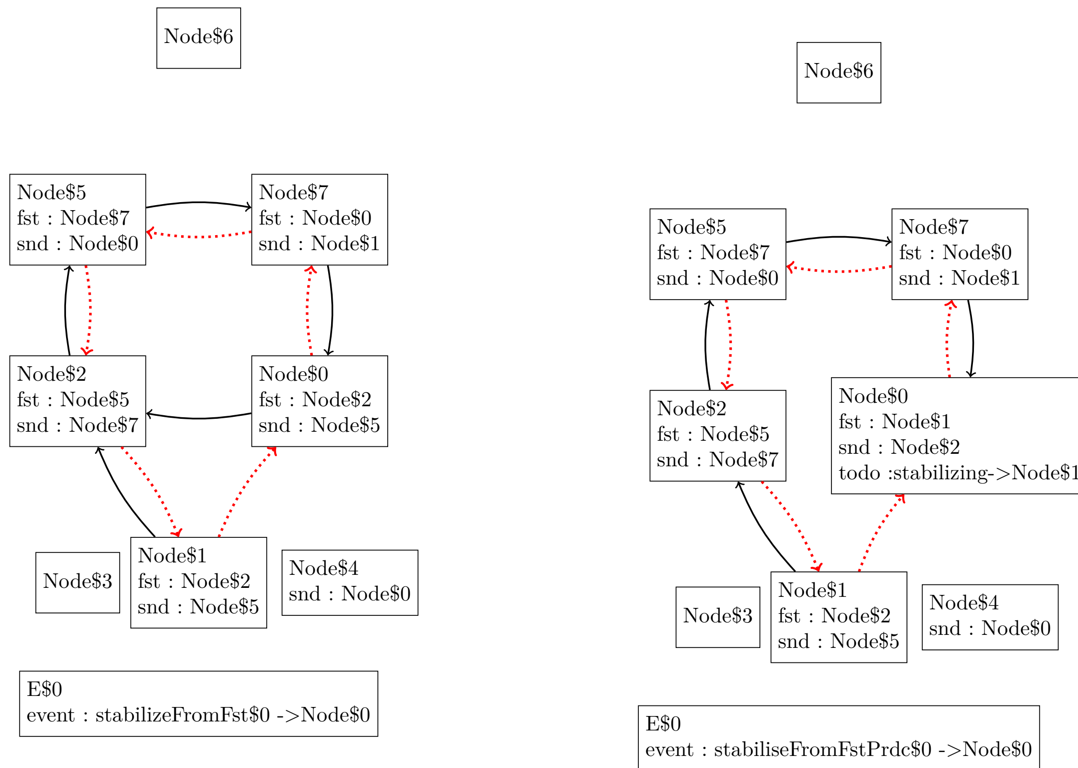
```
1  act stabilizeFromFstPrdc [m, newFst: Node]
2  modifies fst, snd, todo, members, ring {
3    m in members
4    m → Stabilizing → newFst in todo
5    // check that newFst is still better
6    between[m, newFst, m.fst]
7    (newFst not in members) implies {
8      todo' = todo − m → Stabilizing → newFst
9      fst' = fst
10     snd' = snd }
11     else {fst' = fst ++ m → newFst
12     snd' = snd ++ m → newFst.fst
13     todo' = todo − (m → Stabilizing → newFst)
14       + (newFst → Rectifying → m) } }
```

Dans la Figure 7.5, Node$0 termine sa stabilisation en déclenchant `stabilizeFromFstPdrc` avec Node$1, qui en plus d'être membre, demeure le meilleur candidat pour le premier successeur de Node$0, ainsi, il met son premier successeur à jour avec Node$1 qui en retour lui demande de programmer une rectification ultérieure.

Opération de rectification

La rectification est l'opération de maintenance chargée de corriger les pointeurs prédécesseurs des nœuds. Elle est programmée uniquement à la demande d'une stabilisation (`stabilizeFromFstPdrc`, `stabilizeFromFst`). Une rectification d'un nœud `m` est tourjours programmée avec un candidat `newPrdc` comme potentiel nouveau prédécesseur pour `m`. La rectification est modélisée par l'action `rectify`. Elle peut s'exécuter dans exactement trois situations étant donné un nœud `m` : (1) si le prédécesseur `prdc` de `m` est nul ou si le nouveau candidat est meilleur que le prédécesseur actuel, alors le prédécesseur est mis à jour avec le candidat ; (2) sinon, si le prédécesseur actuel n'est pas membre, alors il est également mis à jour ; (3) autrement, le prédécesseur de `m` est laissé tel quel. La rectification modifie le `m.prdc` et le champ `m.todo`, ainsi que l'ensemble des membres de l'anneau.

```
1  act rectify [m, newPrdc: Node] modifies prdc, todo, members, ring {
2    m in members
3    m → Rectifying → newPrdc in todo
4    todo' = todo − m → Rectifying → newPrdc
5    (no m.prdc or between [m.prdc, newPrdc, m]) implies {
6      prdc' = prdc ++ m → newPrdc}
7    else {
```



(a) Occurrence de `stabilizeFromFst` : le premier successeur de Node$0 est membre et le prédécesseur de ce dernier serait un meilleur successeur pour Node$0.

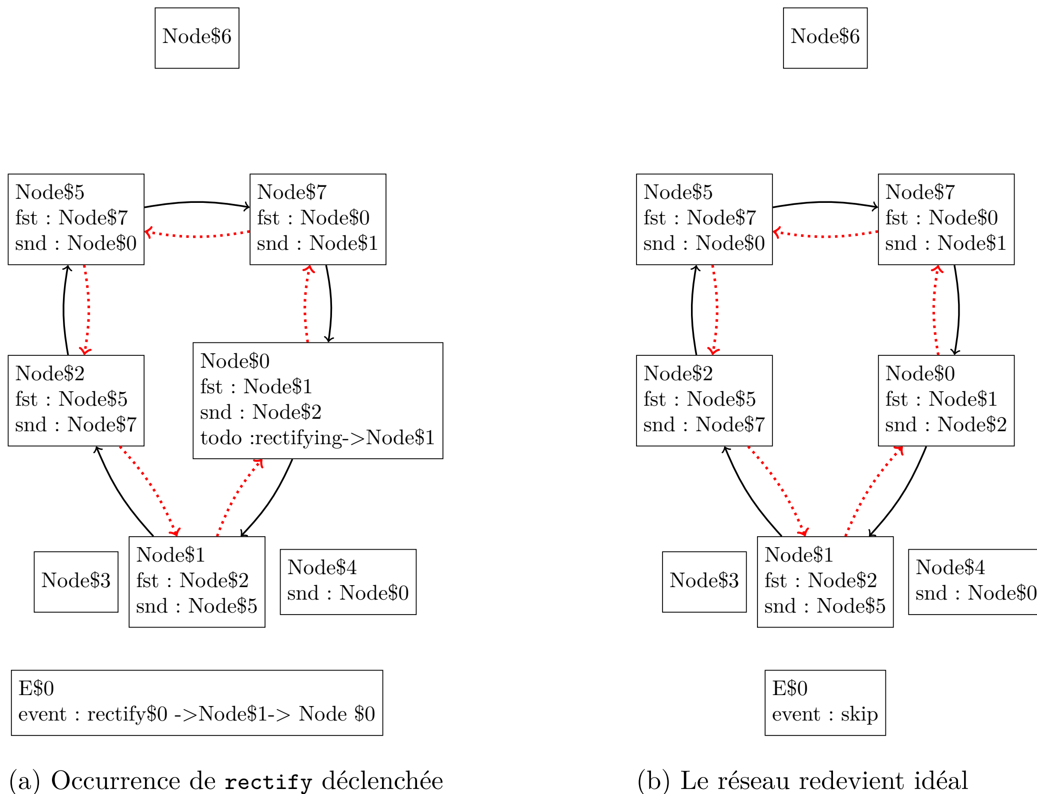
(b) `stabilizeFromFstPrdc` programmée et enregistrée dans `todo`

FIGURE 7.5 – Une Occurrence de `stabilizeFromFst` programme une action `stabilizeFromFstPrdc`

```
8      m.prdc in members implies prdc' = prdc
9        else prdc' = prdc ++ m→newPrdc } }
10
```

Le nœud Node$0 déclenche l'occurrence de la rectification en attente avec Node$1 qu'il a enregistré à l'étape précédente figure 7.6. Node$1 n'est pas meilleur candidat que le prédécesseur actuel. Ainsi, Node$0 ne change pas son prédécesseur. Le réseau redevient idéal l'action `skip` continue de s'exécuter pour maintenir le système actif.

Dans la section suivante, nous allons spécifier le modèle temporel du système, c'est-à-dire les contraintes sur l'ordre à respecter pour l'exécution des opérations (décrites dans cette section) durant l'évolution du système.



(a) Occurrence de `rectify` déclenchée

(b) Le réseau redevient idéal

FIGURE 7.6 – Occurence de `rectify` programmée par un `stabilizeFromFstPrdc`

7.5 Le modèle temporel

Comme présenté au chapitre 6, la forme des traces est automatiquement définie grâce à la couche action d'Electrum. Á tout instant, exactement un événement se produit. Nous spécifions simplement que, dans l'état initial : (1)la topologie du réseau est idéale ; (2) les nœuds ne possèdent pas d'opérations en attente ; (3) les nœuds non-membres n'ont pas de prédécesseur. Les opérations de maintenance du protocole Chord s'exécutent de manière périodique indéfiniment. Néanmoins, lorsque le réseau devient idéal, s'il n'y a plus d'actions `join` ou `fail`, alors plus aucune opération n'est activable. Pour rester conforme à la sémantique d'Electrum, qui comportent des traces d'exécutions infinies, nous ajoutons également une action silencieuse `skip`, qui ne change rien.

```
act skip {} // ne fait rien, ne change rien
```

7.6 La propriété de correction du protocole Chord

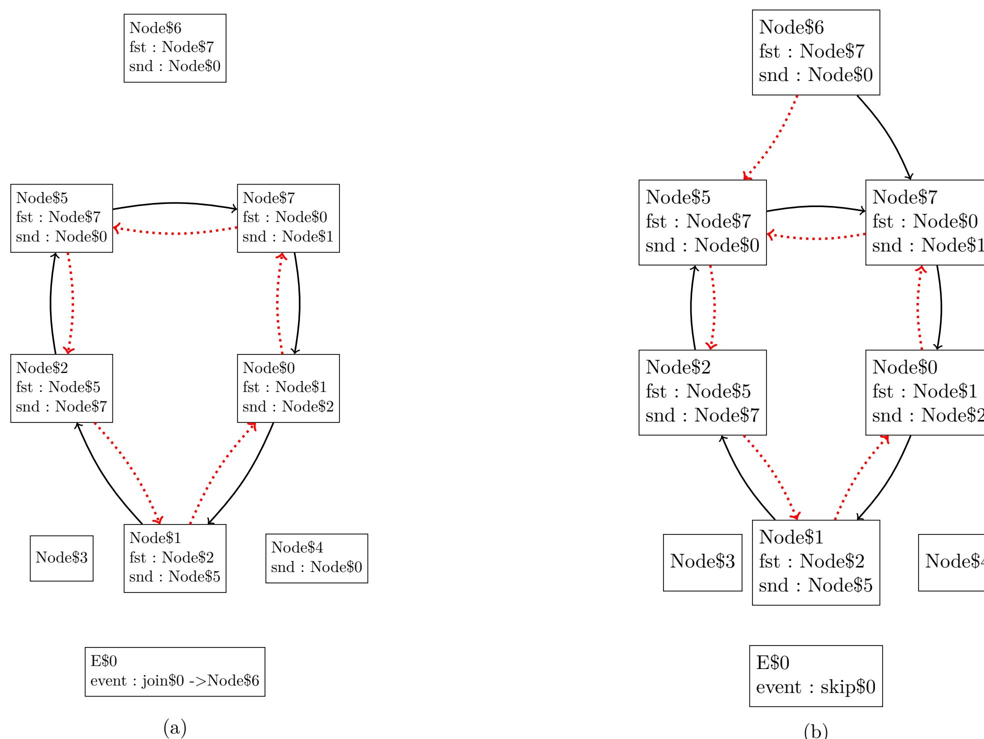
Avec Electrum Analyzer, nous avons vérifié que la spécification est cohérente, c'est-à-dire qu'elle admet un modèle, dans le sens logique. En outre, nous avons vérifié que toutes les branches de toutes

les actions sont réalisables. La propriété de correction de Chord est une propriété de *vivacité*. La traduction de la propriété telle que définie dans PODC est simple, grâce à LTL : elle déclare que si dans le futur, il n'y aura plus jamais d'événements d'arrivée ou de départ de nœuds dans le réseau, alors dans le futur le réseau deviendra idéal et restera ainsi. Ceci est spécifié par l'assertion Electrum :

```
assert correctness {
  (eventually always not (join or fail))
  implies eventually always ideal }
```

Violation de la correction de Chord : absence d'équité

La vérification de cette assertion avec Electrum Analyzer produit le contre-exemple de la figure 7.7 manifesté par la répétition infinie de l'action `skip` dans un état non idéal. Ceci est dû au fait que toute action dont la garde est vraie peut s'exécuter, même si cela crée une situation de famine pour d'autres actions.



(a) (b)

FIGURE 7.7 – Contre-exemple manifesté par l'exécution infinie de l'action `skip`

Classiquement, la solution au problème de famine est d'ajouter des contraintes d'équité aux actions. Il existe deux principales catégories d'équité :

— l'équité forte, qui impose qu'une action qui est infiniment souvent exécutable soit infiniment souvent exécutée. Une telle contrainte empêche la situation où une action est exécutable par intermittence, sans jamais être exécutée ;

— l'équité faible, qui impose qu'une action qui reste toujours exécutable à partir d'un état soit infiniment souvent exécutée. Une telle contrainte empêche la situation où une action reste toujours exécutable sans jamais être exécutée.

Ici, l'action `stabilizeFromFst` est exécutable (sa garde est vraie) si aucune action (`rectify` ou `stabilizeFromPrdc`) n'est en attente pour le même noeud. Elle peut donc être exécutable par intermittence pour un noeud n, si par exemple le schéma suivant se répète : un prédécesseur de n se stabilise et lui demande de faire une rectification, puis que n exécute cette rectification. Pour cette raison, l'action `stabilizeFromFst` nécessite une contrainte d'équité forte.

En revanche, pour les actions `rectify` et `stabilizeFromPrdc`, dans la version modélisée dans le présent manuscrit, une contrainte d'équité faible pourrait suffire puisque ces opérations ne peuvent pas être exécutables par intermittence (une fois exécutables, elles le restent jusqu'à leur exécution). Les contraintes d'équité du modèle de Chord présenté dans le manuscrit pourraient donc être affaiblies dans ce sens.

Pour chaque action on défini un prédicat qui indique si elle est exécutable et un fait qui définit sa contrainte d'équité. Par exemple la contrainte d'équité sur l'action `stabilizeFromFrst`, est définie grâce au prédicat `StabilizeFFromFstPrdcEnabled` et au fait `stabilizeFromFstPrdcSF`.

```
pred rectifyEnabled [ m, n: Node] {
  m in members
  m → Rectifying → n in todo }

fact rectifySF {
  all n, m : Node |
    (always eventually rectifyEnabled[n,m])
      implies (always eventually rectify[n,m]) }

pred StabilizeFFromFstPrdcEnabled [m, n: Node] {
m → Stabilizing → n in todo }

fact stabilizeFromFstPrdcSF{
  all n, m : Node |
    always eventually StabilizeFFromFstPrdcEnabled[n,m]
      implies (always eventually stabilizeFromFstPrdc[n,m]

pred StabilizeFromFstEnabled [m: Node] {
  m in members
  no m.todo.Node
  let succ1 = m.fst |
  (succ1 not in members
  || (succ1 in members  && m.snd != succ1.fst)
  || succ1.prdc != m) }

fact stabilizeFromFstSF
  all n: Node {
```

```
(always eventually StabilizeFromFstEnabled [n])
  implies (always eventually stabilizeFromFst[n])
```

Insatisfaction de la correction de Chord : rectification incomplète

Nous avons ajouté de telles contraintes pour toutes les actions de stabilisation et de rectification, ce qui résout le problème de famine décrit ci-dessus. En outre, cela répond à une exigence de PODC qui stipule que les nœuds doivent effectuer « périodiquement » les actions de maintenance. Cependant, la propriété de correction reste non satisfaite : la vérification de l'assertion avec Electrum Analyzer produit une trace avec six instants, le dernier rebouclant vers son prédécesseur (nous rappelons que les traces sont infinies et représentées sous forme de traces finies avec une boucle de retour du dernier état vers un état précédent). Nous présentons ce contre-exemple dans la figure 7.8 et la figure 7.9.

étape 0 : à l'état initial, le réseau est Ideal ;

étape 1 : l'action `join`[Node$6] est exécutée, Node$6 rejoint le réseau en adoptant Node$5 comme prédécesseur et en copiant également sa liste de successeurs. Les opérations de maintenance sont alors déclenchées pour remettre le réseau à l'état idéal ;

étape 2 : l'action `stabilizeFromFst`[Node$6] est exécutée, le nouveau nœud Node$6 contacte son premier successeur Node$7, découvre qu'il serait meilleur prédécesseur de Node$7 que le prédécesseur courant de Node$7. Il demande donc à Node$7 de programmer une rectification avec lui-même ;

étape 3 : l'action `rectify`[Node$7, node$6] enregistrée précédemment est exécutée, Node$7 apprend que Node$6 est candidat pour être son prédécesseur et met à jour son pointeur prédécesseur avec celui-ci ;

étape 4 : ensuite avec un déclenchement de l'action `fail`[Node$6] le nouveau nœud Node$6 quitte le réseau ;

étape 5 : l'action `stabilizeFromFst`[Node$5] est exécutée, Node$5 contacte son premier successeur Node$7 et apprend que le prédécesseur de Node$7, *i.e.*, Node$6, serait un meilleur premier successeur pour lui-même. Ainsi, il programme une action `stabilizeFromFstPrdc` pour mettre à jour son prédécesseur vers Node$6 (il n'a pas de moyen de savoir que Node$6 est mort à ce stade, cela nécessiterait une communication supplémentaire vers Node$6) ;

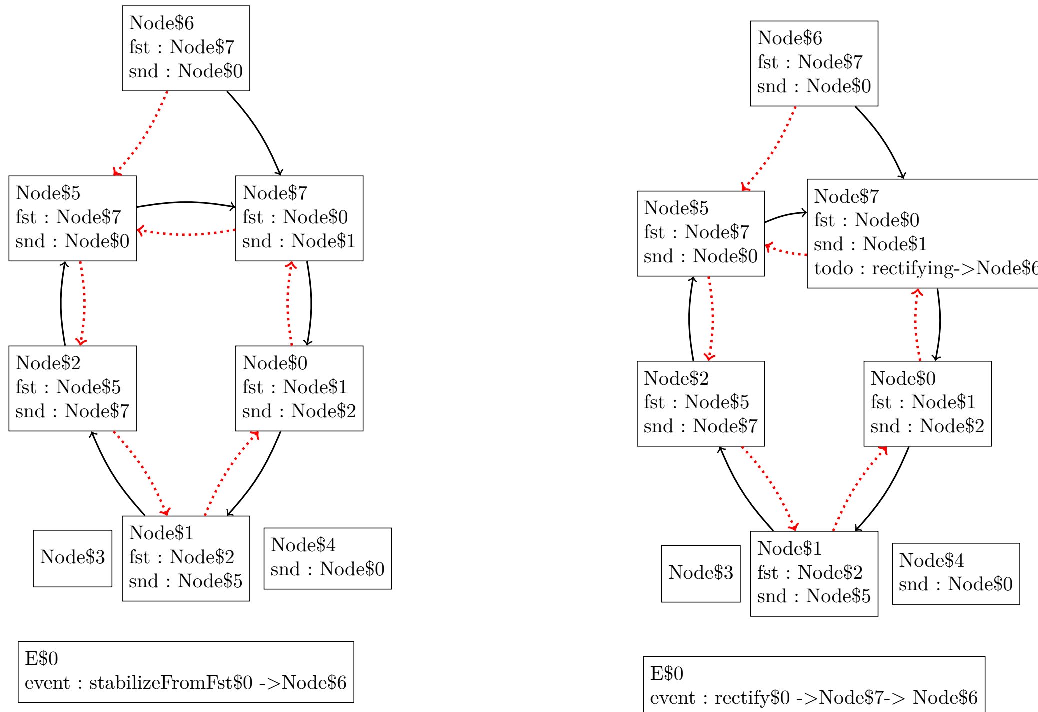
étape 6 : l'action `stabilizeFromFstPrdc`[Node$5, Node$6] est exécutée, mais comme Node$6 n'est plus membre du réseau, cette action n'a aucun effet.

Une rectification du nœud Node$7 serait nécessaire pour mettre à jour son prédécesseur avec le nœud Node$5, mais elle ne peut avoir lieu d'après la description du protocole présentée ci-dessus.

Vers une version correcte du protocole Chord

Une solution possible à ce problème de rectification consiste à ajouter une autre action de rectification ne nécessitant pas de stabilisation pour se déclencher, mais étant exécutée *«périodiquement»* par les nœuds. Nous ajoutons également une contrainte d'équité forte pour cette nouvelle action. Puisque les nœuds ne peuvent pas deviner celui qui sera leur nouveau prédécesseur, l'action indique qu'ils doivent mettre simplement leur pointeur prédécesseur à nul, si ce dernier pointe vers un nœud

mort. L'enjeu ici étant que par d'autres opérations les nœuds finissent par trouver un prédécesseur correct.



(a) Après son arrivée dans le réseau le Node$6 stabililise `stabilizeFromFst`

(b) Le Node$7 programme une rectification avec Node$6

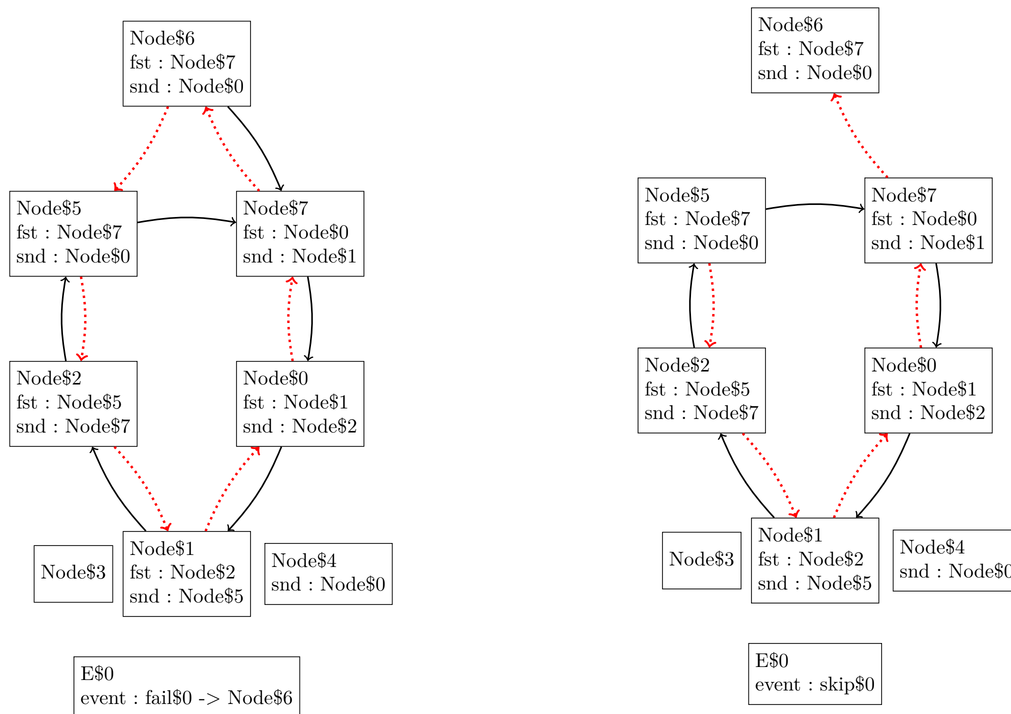
FIGURE 7.8 – Les quatres premiers étapes du contre-exemple.

```
act rectifyNull[m: Node] modifies prdc, members, ring {
 m in members
 m.prdc not in members
  implies prdc' = prdc - m->m.prdc else prdc' = prdc }

fact rectifyNullSF {
  all n {
    always eventually rectifyNullEnabled [n]
      implies (always eventually rectifyNull[n])
```

7.7 L'analyse et l'évaluation des résultats

Cette section présente l'évaluation de la version finale de la propriété de correction du protocole Chord avec Electrum Analyzer. La vérification est exécutée sur un processeur Intel Xeon E5-2699



(a) Départ du nœud Node$6

(b) Problème de rectification entre Node$7 et Node$5

FIGURE 7.9 – Sequence responsable du contre-exemple

fournissant 512 Go de RAM. Le délai d'attente a été fixé à 5h. Nous comparons les techniques de model checking borné par traduction vers un problème SAT en utilisant Minisat (colonnes M dans la table 7.2), de model checking borné par traduction vers LTL en utilisant le model checking borné de nuXmv (colonnes XB de la table 7.2) et de model checking non borné en utilisant une combinaison de IC3 et klive fournie par nuXmv (colonnes XU de la table 7.2). Nous présentons le résultat des techniques bornées pour des horizons temporels de longueur 10 et 15.

Au préalable, nous avons vérifié la cohérence du modèle, (c'est-à-dire qu'il possède effectivement une instance) et que toutes les branches des actions sont exécutables. Toutes ces analyses réalisées en mode BMC d'Electrum Analyzer avec des traces de longueur 10 se sont terminées positivement, comme indiquent les résultats de la table 7.1, à la colonne "modèle action"(la deuxième colonne sera présentée plus tard).

Table 7.1 – vérification de la cohérence et des différentes branches des actions.

Ation.	**modèle action**	**modèle action + Valid comme invariant**
cohérence	0.5	0.3
join	0.6	0.5
fail	0.7	0.6
stabilizeFromFst1	118.8	2 128.8
stabilizeFromFst2	4 039.5	4 409.4
stabilizeFromFst3	0.6	4 584.4
stabilizeFromFstPrdc	305.8	4 594.4
rectify	1 074.0	8 780.5
rectifyNull	4 669.0	4 670.0

légende :

stabilizeFromFst1 : la branche de l'action `stabilizeFromFst` quand le premier successeur est mort.

stabilizeFromFst2 : la branche de l'action `stabilizeFromFst` quand le premier successeur est en vie mais, le second successeur n'est pas à jour.

stabilizeFromFst3 : la branche de l'action `stabilizeFromFst` quand il existe un meilleur candidat (le prédécesseur du premier successeur) pour le premier successeur.

Pour le reste, les noms correspondent aux actions.
N.B, nous ne vérifions que les branches qui modifient le réseau.

Table 7.2 – Temps(s.) d'analyse de la correction (bugué et correcte).

Prop.	**Scope**	**M 10**	**M 15**	**XB 10**	**XB 15**	**XU**
buggy	4	5	5	9	9	36
fixed	4	69	1 316	21	260	558
fixed	5	1 060	t/o	549	t/o	t/o
fixed	6	11 506	t/o	6 930	t/o	t/o
fixed	7	t/o	t/o	t/o	t/o	t/o

Selon les résultats du tableau 7.2, en mode borné Electrum Analyzer peut vérifier la correction pour les réseaux de 4 à 6 membres avec un délai de 10, et exactement 4 membres pour un délai de 15, alors qu'en mode non borné Electrum Analyzer produit un résultat uniquement pour les réseaux à 4 membres.

Nous avons réalisé des travaux supplémentaires visant à améliorer ces résultats, en particulier ceux avec la technique non bornée. Après plusieurs analyses, une piste envisageable a consisté à utiliser des contraintes supplémentaires susceptibles de réduire la taille de l'espace d'états à explorer. En l'occurrence, nous avons exploité le fait que la propriété `Valid` soit soit toujours satisfaite lorsqu'on pas d'un état initial Ideal. Ainsi, la colonne *"modèle action + Valid comme invariant"* du tableau 7.1 correspond au fait d'ajouter le fait suivant au modèle Electrum : `fact always Valid`. Les résultats ne sont malheureusement pas concluant dans ce cas.

Nous soulignons que ces résultats sont intéressants parce qu'ils sont, à notre connaissance, les premiers à aborder la correction de Chord, qui est une propriété de vivacité, automatiquement. En

effet, en ce qui concerne la vérification automatique les travaux antérieurs ont été focalisés sur les propriétés de sûreté. En particulier, [Zav12, Zav15b, Zav17], qui reposent principalement sur Alloy, mais aussi sur Spin, se sont concentrés sur l'analyse d'un invariant inductif (dont la découverte a été une tâche ardue). Cet invariant peut en fait être vérifié dans Electrum, bien qu'Electrum élimine précisément la nécessité d'effectuer une telle recherche. Toutefois, les invariants sont également utiles pour d'autres raisons. [Zav17] concerne également la correction complète de Chord, en s'appuyant sur une preuve manuelle.

7.8 Discussions

Dans cette section, nous soulignons quelques aspects importants de ce travail.

Tout d'abord, nous avons modélisé le protocole Chord de manière très simple grâce à la facilité d'utilisation d'Electrum et sa couche action. Par rapport aux travaux antérieurs, la logique relationnelle de premier ordre, la logique temporelle et les actions (avec la gestion automatique des conditions du cadre et l'entrelacement), associées à l'approche *"push-button"*(complètement automatisée) et au retour visuel d'Electrum Analyzer, nous ont permis de tester différentes approches. Notre modèle s'inspire d'importants travaux de P. Zave, en particulier de [Zav17], dans la mesure où il implémente essentiellement le même algorithme. Cependant, nous soutenons que notre modèle est plus simple, en particulier parce que nous n'avons pas à nous soucier des détails de la présentation des états et aussi parce que l'expression des formules temporelles complexes sur des traces infinies est immédiate.

Par rapport à l'analyse avec Spin [Zav15a], notre modélisation est également simple et immédiate, car P. Zave a dû recourir à divers programmes C pour gérer les notions de graphes présentes dans Chord ainsi que la visualisation.

Deuxièmement, en raison de notre approche basée sur l'analyse de formules de LTL, nous n'avons pas eu besoin d'un invariant inductif pour étudier le protocole. La recherche dudit invariant a été très ardue [Zav19], [Zav11],
[Zav12],[Zav15b],[Zav17], mais, bien sûr, elle est également éclairante. En effet, un invariant est non seulement un moyen de vérification, mais également un moyen pour mieux comprendre le protocole et fournir des indications aux développeurs. À cet égard, nous pensons qu'Electrum pourrait par exemple être utilisé dans une analyse préliminaire et aiderait à trouver ledit invariant.

Troisièmement, nous avons pu trouver quelques bugs dans le modèle Alloy ainsi que la preuve manuelle de [Zav17], confirmés par P. Zave, comme l'ajout de l'opération `rectifyNull` ou l'explicitation des hypothèses d'équité.

Quatrièmement, on prétend souvent que la vivacité «dans l'abstrait» n'est pas très importante, car ce à quoi on aspire est une vivacité bornée, qui est en réalité une propriété de sûreté. Nos travaux montrent que la spécification temporelle directe dans LTL et l'analyse complète de la vivacité étaient utiles pour trouver divers problèmes.

7.9 Autres travaux de vérification des protocoles peer-to-peer

Nous avons mis l'accent dans ce chapitre sur les travaux de P. Zave, qui ont été très importants, car ils ont servi de base à de nombreux autres travaux. Ceci étant, dans la littérature, on trouve plusieurs autres travaux sur l'analyse de la correction du protocole Chord et bien d'autres protocoles peer-to-peer. Dans la suite, nous comparons notre approche à ces autres travaux.

Les travaux présentés dans [LS04] concernent une preuve manuelle d'une propriété de sûreté sur Multichord (une version de Chord avec un mécanisme de communication supplémentaire s'exécutant en arrière-plan) avec la théorie des automates. Il établit que l'algorithme de recherche de MultiChord est parfaitement correct, c'est-à-dire que dans un état idéal hypothétique toute requête renvoie un résultat cohérent.

[LMP04] présente une analyse complètement automatisée de la correction d'une version active de Chord qui est une variante de Chord dont les opérations de maintenance mettent immédiatement le réseau dans un état cohérent après l'arrivée ou le départ d'un nœud. Cette analyse a été réalisée avec une méthode assertionnelle, elle se concentre sur une recherche d'invariant dans l'optique de prouver une propriété de vivacité. Il a démontré que même si la topologie peut être temporairement perturbée pendant la mise à jour des relations successeurs et prédécesseurs, si dans le futur les messages de modification des relations sont tous livrés et qu'il n'y a plus de nouvelle modification initiée, alors dans le futur le protocole restaurera la topologie du réseau

On a également [RRM05] qui concerne une preuve formelle semi-automatisée basée sur Event B et Atelier-B d'une propriété de sûreté sur le protocole FTAR(Fault Tolerant Actif Ring). C'est une extension des travaux réalisés dans [LMP04] qui inclut la tolérance des pannes. Ces travaux prouvent que l'algorithme FTAR maintient une continuité de la topologie durant les modifications de l'état idéal, il garantit la cohérence et la convergence vers l'état idéal en présence des défaillances.

[BG07] propose un modèle abstrait pour l'analyse des réseaux peer-to-peer structuré avec une topologie en anneau. Pour l'étude de cas, il choisit une variante très simple de Chord (sans pannes) et s'appuie sur le $\pi - calculus$ et sur la bisimulation pour analyser sa correction. Pendant que [WWP$^+$15] utilise Verdi, un outil d'implémentation et de vérification formelle des réseaux peer-to-peer basé sur Coq, pour démontrer manuellement une propriété de sécurité sur Chord. Par ailleurs, [KEAAH05] utilise une approche de type "master-equation-based" [Gs96], couplé à la simulation et prouve une propriété de sûreté (cohérence et performance) sur le protocole Chord.

[MLW11] repose sur TLA+, son model checker TLC et son assistant de preuve TLAPS pour prouver la correction du protocole Pastry, qui est une propriété de sûreté. Ces travaux consistaient à la modélisation de l'algorithme de routage et du protocole de communication de Pastry en TLA+, de la vérification du modèle avec TLC et dans une moindre mesure avec TLAPS. Ils ont permis de réduire la propriété de la correction globale en un invariant sur une structure de données sous-jacente.

[Bon12] présente une preuve automatique de la correction du protocole peer-to-peer CAN avec l'assistance de preuve Isabelle.

Dans [BNOG04] la preuve de la correction d'une implémentation de DHT : (RDK) est abordée par une méthode basée sur [Gho06] la bisimulation. D'autres travaux présentés dans [KKPB07] traitent d'une preuve manuelle de la correction des algorithmes fondamentaux de transmission de

messages dans les communications réseaux, basées sur la logique de *Hoare* et la logique temporelle. Le tableau 7.3 synthétise ces différents travaux.

TABLE 7.3 – Tableau comparatif des travaux d'analyse sur Chord et d'autres protocoles peer-to-peer.

Auteurs	**Protocoles**	**Preuve**	**Propriété**	**Outils**
[BCT18]	Chord	A	V	Electum action
[Zav19] [Zav11] [Zav12] [Zav15a]	Chord	A	S	Alloy
[Zav17]	Chord	SA	S+V	Alloy+spin et M
[LMP06]	Chord actif	A	S et V	Méthode assertiomnelle + Invariant
[BG07]	Pure join Chord	SA	S	π-calcul + bisimulation
[WWP+15]	Chord	M	S	Coq
[KEAAH05]	Chord	M	S	master-equation + bisimulation
[LS04]	MultiChord	M	S	théorie des automates
[RRM05]	FTAR	SA	S	B-Event et Atelier B
[LMW11]	Pastry	A	S	TLC et TLAPS
[Bon12]	CAN	A	S	Isabelle
[BNOG04]	DHT	A	S	bisimulation
[KKPB07]	Algorithme de transmission	SA	S	logique temporel+ logique de hoare

Légende :
M = Manuelle ; A = Automatisée ; SA = Semi-Automatisée ; V = Vivacité ; S = Sûreté

Chapitre 8

Conclusion

Dans cette thèse nous avons abordé d'une part, la conception du langage de spécification et vérification formelle que nous avons nommé *Electrum action* et d'autre part, l'analyse formelle de la propriété fondamentale de vivacité du protocole de recherche distribuée Chord.

8.1 Synthèse des travaux

8.1.1 Electrum Action

Electrum est un langage de spécification et de vérification très adapté pour l'analyse des systèmes dynamiques ayant des propriétés structurelles riches. Pour faciliter la spécification du comportement de tels systèmes, nous avons implémenté une couche action au dessus d'Electrum standard.

Cette extension est un sucre syntaxique, c'est-à-dire que la syntaxe supplémentaire et les contraintes sémantiques sont traduites en Electrum standard.

Les expérimentations réalisées ont montré que l'utilisation de la couche action rend la spécification du comportement simple et moins sujette aux erreurs. En effet, la spécification des actions est assez simple et le raisonnement sur les occurrences de celles-ci est assez naturel. La couche action s'est avéré avantageuse pour la visualisation des instances et des contre-exemples.

En outre, le risque d'erreurs est minimisé parce qu'une partie de la spécification du comportement est générée automatiquement. Nous avons expérimenté la couche action sur les exemples classiques tiré de la littérature d'Alloy ainsi que sur un système réel : le protocole de recherche distribuée Chord. Par rapport à Electrum ordinaire, la concision est améliorée et l'efficacité des analyses est raisonnablement réduite en général.

8.1.2 Protocole de recherche distribuée Chord

Dans cette étude nous avons également présenté la spécification et la vérification du protocole distribué Chord. Nous avons mis en évidence l'utilité d'une méthode formelle légère permettant de modéliser et de vérifier des systèmes dynamiques dotés de propriétés structurelles riches, comme le montre Electrum (en particulier avec sa couche d'action).

Grâce la logique du premier ordre temporelle linéaire et aux actions (avec la gestion automatique des conditions du cadre et l'entrelacement) Electrum a permis une modélisation simple et directe des propriétés structurelles et temporelles de Chord, avec un niveau d'abstraction assez élevé et sans perdre les concepts clés du protocole.

L'analyse du modèle Chord avec Electrum est entièrement automatisée et le retour visuel d'Electrum Analyzer facilite l'interprétation des résultats de l'analyse. Ainsi, l'effort de modélisation par rapport aux résultats obtenus est plutôt inférieur à celui de nombreuses autres méthodes formelles. Nous avons également pu tester différentes approches d'analyses.

Même avec des petits réseaux et une analyse avec le model checking borné (BMC) nous avons pu trouver divers bugs dans le modèle de P. Zave (dont nous nous sommes inspirés). Ce qui confirme l'intérêt de travailler avec de petites instances de réseaux. Nous avons pu les résoudre entièrement, en particulier, bien qu'elle ne soit pas une solution totalement satisfaisant, nous avons dû spécifier une nouvelle opération pour résoudre les problèmes non triviaux.

À notre connaissance, il s'agit des premiers travaux d'analyse complètement automatisée de la correction de Chord en tant qu'une propriété de vivacité.

8.2 Limitation des solutions proposées

L'analyse de la correction du protocole Chord est plutôt limitée par la taille des réseaux, en particulier pour l'analyse avec le model checking non borné (UMC) d'Electrum Analyzer. Nous nous attendions à ce résultat. En effet, l'une des raisons qui ont motivé l'étude de Chord, était, pour nous, d'obtenir un banc de test stimulant pour challenger les performances de l'analyse par le model checking non borné (UMC) dans Electrum Analyzer.

En ce qui concerne Electrum action l'impact (voir Figure 6.5) sur la modélisation n'a été évalué que par des utilisateurs familiers d'Electrum ou tout au moins de la spécification formelle. Ainsi l'impact de la couche action sur la modélisation par un débutant n'est pas évaluée, néanmoins, elle offre des avantages pour les utilisateurs expérimentés en les déchargeant de certaines tâches.

8.3 Travaux futurs

Voici plusieurs idées pouvant être explorées dans la continuité de cette thèse. En ce qui concerne Electrum Action, notre étude demeure essentiellement exploratoire, elle peux être étendue, complétée ou généralisée. Elle ouvre de nombreuses perspectives notamment :

1. améliorer l'efficacité des analyses sur horizon non borné dans Electrum avec la couche d'action ;
2. exploiter le modèle temporel (qui impose qu'exactement une action soit exécutée à chaque instant) pour produire un codage SMV optimisé ;
3. réaliser une étude comparative entre la version de la couche action dont la sémantique est traduit en Electrum standard selon les idiomes prédicat et événements avec la version actuelle ;
4. réaliser des expérimentations avec des débutants qui ne sont pas familiers avec Electrum ou avec la spécification et la vérification formelle ;
5. on pourra également étudier des techniques de vérification plus intelligentes pour des analyses à la fois bornée et non bornée indépendamment de la couche action, en particulier, on pourra optimiser la gestion des symétries.

Pour l'analyse de Chord il serait intéressant d'étudier la correction du protocole dans une version affinée, dans laquelle les opérations actuelles de Chord seraient décomposées en actions atomiques plus fines : consultation du nœud successeur, traitement local, programmation d'une opération future, envoi d'un message, etc.

Annexe A

Modèle complet du protocole Chord

```
/*
A model in Electrum of the Chord protocol
Authors: Julien Brunel (ONERA), David Chemouil (ONERA), Jeanne Tawa (ONERA).

SPDX-License-Identifier: ISC

Copyright (c) 2018 ONERA

Permission to use, copy, modify, and/or distribute this software for
any purpose with or without fee is hereby granted, provided that the
above copyright notice and this permission notice appear in all
copies.

THE SOFTWARE IS PROVIDED "AS IS" AND ONERA DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL ONERA BE LIABLE FOR ANY
SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/

open util/ordering[Node]

// CYCLIC ORDERING

pred between [n1, nb, n2: Node] {
  lt[n1, n2] implies (lt[n1, nb] and lt[nb, n2])
  else (lt[n1, nb] or lt[nb, n2]) }

fun nextNode: Node → Node {
```

```
32    { n, m: Node |
33          no next[n] implies m = first // wrap around
34          else m = next[n] } }
35
36  // DOMAIN
37
38  abstract sig Status {}
39  one sig Stabilizing, Rectifying extends Status {}
40
41  sig Node {
42    var fst, snd, prdc: lone Node,
43    var todo: Status -> Node }
44
45  fun succ: Node -> lone Node {
46    { m1, m2: members |
47      m1.fst in members implies m2 = m1.fst
48      else m2 = m1.snd } }
49
50  var sig members in Node {}
51  fact membersDef {
52    always members = { n : Node | some n.fst and some n.snd } }
53
54  var sig ring in members {}
55  fact ringDef {
56    always ring = { m : members | m in m.^succ } }
57
58  fun nonMembers : set Node { Node - members }
59
60  fun appendages : set Node { members - ring }
61
62  // STRUCTURAL PROPERTIES
63
64  pred atLeastOneRing { some ring }
65
66  pred atMostOneRing { all m1, m2: ring | m1 in m2.^succ }
67
68  pred connectedAppendages {
69    all m1: appendages | some m2: ring | m2 in m1.^succ }
70
71  pred orderedRing { // non-loopiness
72    all disj m1, m2, mb: ring |
73        m2 = m1.succ implies not between[m1, mb, m2] }
74
75  pred orderedSuccessors { // successor list validity
76    all m: members | between[m, m.fst, m.snd] }
77
78  pred valid {
79    atLeastOneRing
80    atMostOneRing
81    orderedRing
```

```
82    connectedAppendages
83    orderedSuccessors }
84
85  pred ideal {
86    valid
87    no appendages
88    fst = ~prdc
89    all m: members {
90      m.snd = m.fst.fst
91      m.snd + m.fst in members } }
92
93  fact atLeast3Members { // <=> always (#members >= 3)
94    always some disj base1, base2, base3: Node |
95      base1 + base2 + base3 in members }
96
97  fact init {
98    no todo
99    no nonMembers.prdc
100   ideal }
101
102 // ACTIONS
103
104 act skip {}
105
106 act fail [f: Node]
107 modifies fst, snd, prdc, todo, members, ring {
108   f in members
109   all m : succ.f | m.fst + m.snd - f in members // failure assumption
110   fst' = fst - f -> Node
111   snd' = snd - f -> Node
112   prdc' = prdc - f -> Node
113   todo' = todo - f -> Status -> Node }
114
115 act join [new: Node]
116 modifies fst, snd, prdc, members, ring {
117   new not in members
118   some m: members {
119     between[m, new, m.fst]
120     fst' = fst ++ new -> m.fst
121     snd' = snd ++ new -> m.snd
122     prdc' = prdc ++ new -> m } }
123
124 act stabilizeFromFst [m: Node]
125 modifies fst, snd, todo, members, ring {
126   m in members
127   no m.todo
128   m.fst not in members implies {
129     fst' = fst ++ m -> m.snd
130     snd' = snd ++ m -> nextNode[m.snd]
131     todo' = todo }
```

```
132   else {
133     fst' = fst
134     snd' = snd ++ m → m.fst.fst
135     (some m.fst.prdc and between[m, m.fst.prdc, m.fst]) implies
136       todo' = todo + m → Stabilizing → m.fst.prdc
137     else m != m.fst.prdc implies
138       todo' = todo + m.fst → Rectifying → m
139     else todo' = todo } }
140
141 act stabilizeFromFstPrdc [m, newFst: Node]
142 modifies fst, snd, todo, members, ring {
143   m in members
144   m → Stabilizing → newFst in todo
145   between[m, newFst, m.fst]
146   newFst not in members implies {
147     todo' = todo − m → Stabilizing → newFst
148     fst' = fst
149     snd' = snd }
150   else {
151     fst' = fst ++ m → newFst
152     snd' = snd ++ m →  newFst.fst
153     todo' = todo − (m → Stabilizing → newFst)
154             + (newFst → Rectifying → m) } }
155
156 act rectify [m, newPrdc: Node]
157 modifies prdc, todo, members, ring {
158   m in members
159   m → Rectifying → newPrdc in todo
160   todo' = todo − m → Rectifying → newPrdc
161   (no m.prdc or between[m.prdc, newPrdc, m] or m.prdc not in members) implies {
162     prdc' = prdc ++ m → newPrdc}
163   else
164     prdc' = prdc }
165
166 act rectifyNull [m: Node] modifies prdc, members, ring {
167   m in members
168   m.prdc not in members
169   prdc' = prdc − m → Node }
170
171 // FAIRNESS
172
173 pred stabilizeFromFstEnabled [m: Node] {
174   m in members
175   no m.todo }
176
177 pred stabilizeFromFstPrdcEnabled [m, n: Node] {
178   m in members
179   m → Stabilizing → n in todo }
180
181 pred rectifyEnabled [m, n: Node] {
```

```
182   m in members
183   m → Rectifying → n in todo }
184
185 pred rectifyNullEnabled [n: Node] {
186   n in members
187   n.prdc not in members }
188
189 let strongFairness [enabled, fired] {
190   (always eventually enabled) implies (always eventually fired) }
191
192 fact fairness {
193   all n, m : Node {
194     strongFairness[rectifyNullEnabled[n], rectifyNull[n]]
195     strongFairness[stabilizeFromFstEnabled[n], stabilizeFromFst[n]]
196     strongFairness[stabilizeFromFstPrdcEnabled[n, m], stabilizeFromFstPrdc[n, m]]
197     strongFairness[rectifyEnabled[n, m], rectify[n, m]] } }
198
199 // CORRECTNESS
200
201 consistent: run {} for 5 expect 1
202
203 assert correctness {
204   (eventually always not (join or fail))
205     implies eventually always ideal }
206
207 check correctness for 4 but 10 Time expect 0
208 check correctness for 5 but 10 Time expect 0
209 check correctness for 6 but 10 Time expect 0
210
211 assert invariant{
212   always valid }
213 check invariant for 4 expect 0
214 check invariant for 5 expect 0
215 check invariant for 6 expect 0
216
217 //BENCHMARK  PREDICATE
218 pred  StabilizeFromFstNullFst[m: Node]{
219 eventually{
220   m in members
221   no m.todo.Node
222   m.fst not in members
223  stabilizeFromFst [m]
224 }
225 }
226 pred StabilizeFromFstWithFstCorredSnd[m:Node]{
227 eventually{ m in members
228   no m.todo.Node
229   m.fst  in members
230   m.snd != m.fst.fst
231 stabilizeFromFst [m]
```

```
232 }
233 }
234 pred StabilizeFromFstWithFstBetterFst[m:Node]{
235 eventually{ m in members
236   no m.todo.Node
237   m.fst  in members
238   some m.fst.prdc
239   m.fst.prdc != m
240 stabilizeFromFst [m]
241 }
242 }
243 pred RectifyWihTodo[m,n:Node]{
244 eventually {m in members
245   m → Rectifying → n in todo
246   rectify [m, n]}
247 }
248
249 pred RectifyNullNomemberPrdc [n:Node]{
250   eventually{n in members
251   n.prdc not in members
252 rectifyNull [n]}
253  }
254
255 run StabilizeFromFstNullFst for 4  expect 1
256 run StabilizeFromFstNullFst for 5 expect 1
257 run StabilizeFromFstNullFst for 6 expect 1
258
259 run StabilizeFromFstWithFstCorredSnd for 4 expect 1
260 run StabilizeFromFstWithFstCorredSnd for 5 expect 1
261 run StabilizeFromFstWithFstCorredSnd for 6 expect 1
262
263 run StabilizeFromFstWithFstBetterFst for 4 expect 1
264 run StabilizeFromFstWithFstBetterFst for 5 expect 1
265 run StabilizeFromFstWithFstBetterFst for 6 expect 1
266
267 run RectifyNullNomemberPrdc for 4 expect 1
268 run RectifyNullNomemberPrdc for 5 expect 1
269 run RectifyNullNomemberPrdc for 6 expect 1
270
271 run  RectifyWihTodo for 4 expect 1
272 run  RectifyWihTodo for 5 expect 1
273 run  RectifyWihTodo for 6 expect 1
274
275 run{} for 4 expect 0
```

Annexe B

Modélisation des systèmes classiques d'Alloy dans Electrum ordinaire et Electrum avec action

B.1 Modèles de l'élection du leader

B.1.1 Modèle Electrum Ordinaire

```
1  module ring
2  open util/ordering[Id]
3  sig Id {}
4  sig Process {
5    succ: Process,
6    var queue: set Id,
7    id : Id
8  } {@id in Process lone -> Id}
9  var sig elected in Process {}
10 fact ring {
11   all p: Process | Process in p.^succ
12   }
13 pred init  {
14   all p: Process | p.queue = p.id
15   }
16 pred trans [p: Process] {
17   some sent: p.queue {
18     p.queue' = p.queue' - sent
19     p.succ.queue' = p.succ.queue +
20       (sent- prevs[p.succ.id])
21     }
22   all other : Process - (p + p.succ) |
23     other.queue' = other.queue
24   }
```

```
fact defineElected {
  no elected
  always { elected' = {p: Process |
    (after { p.id in p.queue }) and
         p.id not in p.queue} }
}
fact traces {
  init
  always { all p: Process | trans[p]
    or trans [p.~succ]
    or skip [p] }
  }
pred skip [p: Process] {
  p.queue' = p.queue
  }
assert GoodSafety {
  always { all x : elected |
    always { all y : elected | x = y }}
 }
pred Progress  {
  always {some Process.queue implies
    after { some p: Process | not skip [p]} }
  }
assert BadLiveness { some Process implies
    eventually { some elected } }
assert GoodLiveness {
      some Process && Progress implies
    eventually { some elected } }
pred consistent {}
run consistent for 3 but 20 Time expect 1
run consistent for 4 but 20 Time expect 1
run consistent for 5 but 20 Time expect 1

// Ring (1) scenario
check BadLiveness for 3 but 20 Time expect 1
check BadLiveness for 4 but 20 Time expect 1
check BadLiveness for 5 but 20 Time expect 1

// Ring (2) scenario
check GoodLiveness for 3 but 20 Time expect 0
check GoodLiveness for 4 but 20 Time expect 0
check GoodLiveness for 5 but 20 Time expect 0

// Ring (3) scenario
check GoodSafety for 3 but 20 Time expect 0
check GoodSafety for 4 but 20 Time expect 0
check GoodSafety for 5 but 20 Time expect 0
```

B.1.2 Modèle Electrum Avec Action

```
1
```

```
2   open util/ordering[Id]
3   sig Id {}
4   sig Process {
5     succ: Process,
6     var queue: set Id,
7     id : Id
8   } {@id in Process lone → Id}
9   var sig elected in Process {}
10  pred init  {
11    all p: Process | p.queue = p.id
12    }
13  fact ring {
14    all p: Process | Process in p.^succ
15    }
16  act skip {}
17  act trans [p : Process] modifies queue {
18    some sent : p.queue {
19      p.queue' = p.queue − sent
20      p.succ.queue' = p.succ.queue +
21       (sent − prevs[p.succ.id])
22    }
23    all other : Process − (p + p.succ) |
24      other.queue' = other.queue
25  }
26  fact defineElected {
27    no elected
28    always { elected' = {p: Process |
29      (after { p.id in p.queue })
30      p.id not in p.queue} }
31  }
32  fact traces {
33    init
34    }
35  assert GoodSafety {
36    always { all x : elected |
37      always { all y : elected | x = y }}
38   }
39  pred Progress  {
40    always {some Process.queue implies
41       after { some p: Process | trans [p]} }
42    }
43  assert BadLiveness { some Process implies
44    eventually { some elected } }
45  assert GoodLiveness {
46        some Process && Progress implies
47    eventually { some elected } }
48
49  pred consistent {}
50  run consistent for 3 but 20 Time expect 1
51  run consistent for 4 but 20 Time expect 1
```

```
52 run consistent for 5 but 20 Time expect 1
53
54 // Ring (1) scenario
55 check BadLiveness for 3 but 20 Time expect 1
56 check BadLiveness for 4 but 20 Time expect 1
57 check BadLiveness for 5 but 20 Time expect 1
58
59 // Ring (2) scenario
60 check GoodLiveness for 3 but 20 Time expect 0
61 check GoodLiveness for 4 but 20 Time expect 0
62 check GoodLiveness for 5 but 20 Time expect 0
63
64 // Ring (3) scenario
65 check GoodSafety for 3 but 20 Time expect 0
66 check GoodSafety for 4 but 20 Time expect 0
67 check GoodSafety for 5 but 20 Time expect 0
```

B.2 Modèles de spantree

B.2.1 Modèle Electrum Ordinaire

```
1
2 open util/ordering[Lvl]
3 open util/graph[Process]
4
5 -----Processus et graphe de processus
6 sig Lvl {}
7
8 sig Process {
9   adj : set Process,
10  var lvl: lone Lvl,
11  var parent: lone Process
12 }
13
14 fact processGraph {
15   noSelfLoops[adj]
16   undirected[adj]
17   weaklyConnected[adj] -- Process in Root.*adj
18 }
19
20 one sig Root in Process {}
21
22 ----- initialisation
23
24 pred Init {
25   no lvl
26   no parent
27 }
28
29 -----predicats et operations
```

```
30
31 pred Nop {}
32
33 -- when p is the Root
34 pred ActRoot[p : Process] { --modifies lvl
35   p = Root
36   no p.lvl
37   lvl' = lvl + p->first
38 }
39
40 -- when p is not the Root
41 pred ActNotRoot[p : Process] { --modifies lvl, parent
42   p != Root
43   no p.lvl
44     some adjProc: p.adj {
45        some adjProc.lvl
46     lvl' = lvl + p->(adjProc.lvl).next
47     parent' = parent + p->adjProc
48   }
49 }
50
51 Condition du cadre style reiter
52
53 fact {
54   always { parent != parent' implies
55     some p:Process | ActNotRoot[p] }
56   always { lvl != lvl' => some p:Process |
57      (ActRoot[p] or ActNotRoot[p])}
58 }
59
60
61 pred Trans {
62   Nop or some p:Process | (ActRoot[p] or ActNotRoot[p])
63   !(Nop and some p:Process | (ActRoot[p] or ActNotRoot[p]) )
64 }
65
66 fact Trace {
67   Init
68   always Trans
69 }
70
71 -----equite forte
72
73 pred Fairness {
74   always { (some p : Process | no p.lvl) implies
75        eventually { some p : Process | (ActNotRoot[p] or ActRoot[p]) }  }
76 }
77
78 pred IsSpanTree {
79    tree[~parent]
```

```
80  }
81
82  assert BadLiveness {
83    eventually { IsSpanTree }
84  }
85
86  assert GoodLiveness {
87    Fairness => eventually { IsSpanTree }
88  }
89
90  assert GoodSafety {
91    always { no p : Process | p in p.^(parent) }
92  }
93
94
95  pred consistent {}
96  run consistent for 3 but 20 Time expect 1
97  run consistent for 4 but 20 Time expect 1
98  run consistent for 5 but 20 Time expect 1
99
100 // Span (1) scenario
101 check BadLiveness for 3 but 20 Time expect 1
102 check BadLiveness for 4 but 20 Time expect 1
103 check BadLiveness for 5 but 20 Time expect 1
104 // Span (2) scenario
105 check GoodLiveness for 3 but 20 Time expect 0
106 check GoodLiveness for 4 but 20 Time expect 0
107 check GoodLiveness for 5 but 20 Time expect 0
108 // Span (3) scenario
109 check GoodSafety for 3 but 20 Time expect 0
110 check GoodSafety for 4 but 20 Time expect 0
111 check GoodSafety for 5 but 20 Time expect 0
```

B.2.2 Modèle Electrum Avec Action

```
1
2  open util/ordering[Lvl]
3  open util/graph[Process]
4  -----Processus et Graphe de processus
5  sig Lvl {}
6
7  sig Process {
8    adj : set Process,
9    var lvl: lone Lvl,
10   var parent: lone Process
11 }
12
13 fact processGraph {
14   noSelfLoops[adj]
15   undirected[adj]
16   weaklyConnected[adj] -- Process in Root.*adj
```

```
17  }
18
19  one sig Root in Process {}
20
21  ----initialisation
22
23  fact init {
24    no lvl
25    no parent
26  }
27
28  -----3 Actions
29
30  act Nop {}
31
32  -- when p is the Root
33  act ActRoot[p : Process] modifies lvl {
34    p = Root
35    no p.lvl
36    lvl' = lvl + p->first
37  }
38
39  ------Si p est la racine
40  act ActNotRoot[p : Process] modifies lvl, parent {
41    p != Root
42    no p.lvl
43      some adjProc: p.adj {
44         some adjProc.lvl
45      lvl' = lvl + p->(adjProc.lvl).next
46      parent' = parent + p->adjProc
47    }
48  }
49
50
51  -----equite forte
52
53  let SF [pre, post] { ((always eventually pre)
54        implies (always eventually post)) }
55
56  pred strong_fairness {
57    all p : Process {
58      SF[pre_ActRoot[p], post_ActRoot[p]]
59      SF[pre_ActNotRoot[p], post_ActNotRoot[p]]
60    }
61  }
62
63
64  pred IsSpanTree {
65     tree[~parent]
66  }
```

```
67
68 assert BadLiveness {
69   eventually { IsSpanTree }
70 }
71
72 assert GoodLiveness {
73   strong_fairness => eventually { IsSpanTree }
74 }
75
76 assert GoodSafety {
77   always { no p : Process | p in p.^(parent) }
78 }
79
80
81 pred consistent {}
82 run consistent for 3 but 20 Time expect 1
83 run consistent for 4 but 20 Time expect 1
84 run consistent for 5 but 20 Time expect 1
85
86 // Span (1) scenario
87 check BadLiveness for 3 but 20 Time expect 1
88 check BadLiveness for 4 but 20 Time expect 1
89 check BadLiveness for 5 but 20 Time expect 1
90 // Span (2) scenario
91 check GoodLiveness for 3 but 20 Time expect 0
92 check GoodLiveness for 4 but 20 Time expect 0
93 check GoodLiveness for 5 but 20 Time expect 0
94 // Span (3) scenario
95 check GoodSafety for 3 but 20 Time expect 0
96 check GoodSafety for 4 but 20 Time expect 0
97 check GoodSafety for 5 but 20 Time expect 0
```

B.3 Modèles de Firewire

B.3.1 Modèle Electrum Ordinaire

```
1
2 ---------------   operations
3
4 sig Op {}
5 one sig AssignParentOp, ReadReqOrAckOp, ElectOp, WriteReqOrAckOp, ResolveContentionOp, StutterOp extends Op
6 one sig StateOp {  var op: one Op  }
7
8
9 ---------------   nodes
10
11 abstract sig Node {}
12
13 fun from_node : Node -> Link {  {n: Node, x: Link | x.source = n } }
14 fun to_node : Node -> Link {  {n: Node, x: Link | x.target = n } }
```

```
15
16  var sig Waiting, Active, Contending, Elected extends Node {}
17
18  −−−−−−−−−−−−−−−   messages
19
20  abstract sig Msg {}
21  one sig Req, Ack extends Msg {}
22
23  −−−−−−−−−−−−−−−   queues
24
25  sig Queue {var slot: lone Msg, var overflow: lone Msg}
26
27  pred QueuesUnchanged [xs: set Link, next_slot: Queue→lone Msg, next_overflow: Queue→lone Msg] {
28    all x: xs | x.queue.next_slot = x.queue.slot && x.queue.next_overflow = x.queue.overflow
29  }
30
31  pred ReadQueue [x: Link, next_slot: Queue→lone Msg, next_overflow: Queue→lone Msg] {
32      no x.queue.(next_slot + next_overflow)
33    QueuesUnchanged [Link − x, next_slot, next_overflow]
34  }
35
36  pred PeekQueue [x: Link, m: Msg] { m = x.queue.slot }
37
38  pred IsEmptyQueue [x: Link] { no x.queue.(slot + overflow)  }
39
40
41  −−−−−−−−−−−−−−−   links
42
43  sig Link {
44      source, target: Node,
45      queue : Queue
46  }
47
48  var sig ParentLinks in Link {}
49
50  fun reverse : Link → Link {  {x: Link, y: Link | y.target = x.source && y.source = x.target} }
51
52  −− each Link has its reverse:
53  fact { { all x: Link | reverse[x] in Link } }
54
55  −− each Queue corresponds to one Link:
56  fact {queue in Link one → Queue}
57
58  −− at most one link between a pair of nodes in a given direction:
59  fact {no disj x,y: Link | x.source = y.source && x.target = y.target}
60
61  −− topology is tree−like: acyclic when viewed as an undirected graph:
62  one sig Tree{  tree: Node→Node  }
63  {
64    some root: Node {
```

```
65      tree in Node lone -> Node
66      Node in root.*tree
67      no ^tree & iden
68      tree + ~tree = ~source.target
69      }
70  }
71
72  ---------------    initialization and execution
73
74
75  pred Init {
76    Node in Waiting
77    no ParentLinks
78    all x: Link | IsEmptyQueue [x]
79  }
80
81  fact traces {
82    Init
83    always Trans
84  }
85
86
87
88  ---------------    6 predicates corresponding to the operations
89
90  pred Stutter { StateOp.op = StutterOp }
91
92  pred AssignParent [x : Link] { -- modifies ParentLinks, slot, overflow
93    StateOp.op = AssignParentOp
94    x.target in Waiting
95    ! IsEmptyQueue [x]
96    ReadQueue [x,slot',overflow'] -- preserves all the other queues
97    ParentLinks' = ParentLinks + x
98  }
99
100 pred ReadReqOrAck [x : Link] { -- modifies Active, Contending, slot, overflow
101   StateOp.op = ReadReqOrAckOp
102   x.target in (Active + Contending) & (PeekQueue [x, Ack] => Active' else Contending') -- correct version
103   NoNodeChangeExceptAt [x.target]
104   ! IsEmptyQueue [x]
105   ReadQueue [x,slot',overflow']  -- preserves all the other queues
106 }
107
108
109
110
111 pred WriteReqOrAck [n : Node] { -- modifies Waiting, Active, slot, overflow
112   StateOp.op = WriteReqOrAckOp
113   n in Waiting
114   lone to_node[n] - ParentLinks
```

```
115
116   all x: from_node[n] |
117
118     no x.queue.slot => (  (x.queue.slot' = (reverse[x] in ParentLinks => Ack else Req))
119                           && (no x.queue.overflow')
120                          )
121       else {
122         x.queue.slot' = x.queue.slot
123         some x.queue.overflow'
124       }
125
126   QueuesUnchanged [Link - from_node[n], slot', overflow']
127
128   Waiting' = Waiting - n
129   Active' = Active + n
130 }
131
132 pred ResolveContention [x : Link] { --modifies Contending, Active, ParentLinks
133   StateOp.op = ResolveContentionOp
134   x.(source + target) in Contending
135   Contending' = Contending -  x.(source + target)
136   Active' = Active + x.(source + target)
137   ParentLinks' = ParentLinks + x
138 }
139
140 pred Elect [n : Node] { --modifies Active, Elected
141   StateOp.op = ElectOp
142   n in Active
143   to_node[n] in ParentLinks
144   Active' = Active - n
145   Elected' = Elected + n
146 }
147
148 pred Trans {
149
150   Stutter
151   or
152   (some x: Link | AssignParent[x] or ReadReqOrAck[x] or ResolveContention[x])
153   or
154   (some n:Node | WriteReqOrAck[n] or Elect[n])
155
156 }
157
158 ---------------   Reiter style
159
160 fact {
161   always { slot != slot' => (some x:Link | AssignParent[x] or ReadReqOrAck[x])
162     or some n: Node | WriteReqOrAck [n] }
163   always { overflow != overflow' => (some x:Link | AssignParent[x] or ReadReqOrAck[x])
164     or some n: Node | WriteReqOrAck [n] }
```

```
165   always { Waiting != Waiting' ⇒  some n: Node | WriteReqOrAck[n] }
166   always { Active != Active' ⇒
167     (some x:Link | ReadReqOrAck [x] or ResolveContention [x])
168     or
169     some n:Node | Elect [n] or WriteReqOrAck [n]
170   }
171   always { Contending != Contending' ⇒  some x:Link | ReadReqOrAck [x] or ResolveContention [x] }
172   always { Elected != Elected' ⇒ some n:Node| Elect [n] }
173   always { ParentLinks != ParentLinks' ⇒ some x:Link | AssignParent[x] or ResolveContention[x] }
174 }
175
176
177
178
179 pred NoNodeChangeExceptAt [nodes: set Node] {
180   (Node − nodes) & Waiting = (Node − nodes) & Waiting'
181   (Node − nodes) & Active = (Node − nodes) & Active'
182   (Node − nodes) & Contending = (Node − nodes) & Contending'
183   (Node − nodes) & Elected = (Node − nodes) & Elected'
184 }
185
186
187
188 −−−−−−−−−−−−−−−   fairness
189
190
191 pred Fairness {
192   always { (no Elected) ⇒ eventually { StateOp.op != StutterOp } }
193 }
194
195
196 /*
197 let SF [pre, post] { ((always eventually pre) implies (always eventually post)) }
198
199 pred strong_fairness {
200
201   all x : Link {
202     SF[pre_AssignParent[x], post_AssignParent[x]]
203     SF[pre_ReadReqOrAck[x], post_ReadReqOrAck[x]]
204     SF[pre_ResolveContention[x], post_ResolveContention[x]]
205   }
206   all n : Node {
207     SF[pre_Elect[n], post_Elect[n]]
208     SF[pre_WriteReqOrAck[n], post_WriteReqOrAck[n]]
209   }
210 }
211 */
212
213
214 −−−−−−−−−−−−−−−   properties and run / check
```

```
215
216
217
218 -- in every state of the trace, there is at most one elected node
219 assert GoodSafety {
220       always { lone Elected }
221 }
222
223 -- no queue overflows
224 assert GoodSafety2 {  always { no overflow } }
225
226 -- there is some state in which a node has been elected
227 assert BadLiveness {
228   eventually { some Elected }
229 }
230
231 -- there is some state in which a node has been elected, under the hypothesis of fairness
232
233 assert BadLiveness2 {
234   Fairness ⇒  eventually some Elected
235 }
236
237
238
239 pred noBadRead {
240   always {
241     !(Node in (Active+Contending) && no slot && StateOp.op != ResolveContentionOp) --no {x:Link | pre_ResolveConte
242   }
243 }
244
245 -- there is some state in which a node has been elected,
246 -- assuming strong fairness and the non-occurrence of a state in which
247 -- each node is either Active or Contending while all queues are empty
248 -- and no contention can be solved.
249
250 assert GoodLiveness {
251   Fairness && noBadRead ⇒  eventually some Elected
252 }
253
254
255 pred consistent {}
256
257 run consistent for 3 but 20 Time expect 1
258 run consistent for 4 but 20 Time expect 1
259 run consistent for 5 but 20 Time expect 1
260
261 // Firewire (0) scenario:  in every state of the trace, there is at most one node that has been elected
262 check GoodSafety for 3 but 20 Time expect 0
263 check GoodSafety for 4 but 20 Time expect 0
264 check GoodSafety for 5 but 20 Time expect 0
```

```
265
266 // Firewire (1) scenario: no queue overflows
267 /*
268 check GoodSafety2 for 3 but 20 Time expect 0
269 check GoodSafety2 for 4 but 20 Time expect 0
270 check GoodSafety2 for 5 but 20 Time expect 0
271 */
272
273 // Firewire (2) scenario: there is some state in which a node has been elected
274 check BadLiveness for 3 but 20 Time expect 1
275 check BadLiveness for 4 but 20 Time expect 1
276 check BadLiveness for 5 but 20 Time expect 1
277
278 // Firewire (3) scenario: there is some state in which a node has been elected
279 /*
280 check BadLiveness2 for 3 but 20 Time expect 1
281 check BadLiveness2 for 4 but 20 Time expect 1
282 check BadLiveness2 for 5 but 20 Time expect 1
283 */
284
285 // Firewire (4) scenario:  there is some state in which a node has been elected, under several assumptions.
286
287 check GoodLiveness for 3 but 20 Time expect 0
288 check GoodLiveness for 4 but 20 Time expect 0
289 check GoodLiveness for 5 but 20 Time expect 0
290
291
292
293
294
295 // DEFINED VARIABLES
296 // Defined variables are uncalled, no-argument functions.
297 // They are helpful for getting good visualization.
298 fun queued: Link -> Msg {
299     {L: Link, m: Msg | m in L.(queue).slot}
300 }
```

B.3.2 Modèle Electrum Avec Action

```
1
2 -----------nodes
3
4 abstract sig Node {}
5
6 fun from_node : Node -> Link {
7   {n: Node, x: Link | x.source = n }
8 }
9 fun to_node : Node -> Link {
10   {n: Node, x: Link | x.target = n } }
11
12 var sig Waiting, Active, Contending, Elected extends Node {}
```

```
13
14 ---------------   messages
15 abstract sig Msg {}
16 one sig Req, Ack extends Msg {}
17
18 ---------------   queues
19
20 sig Queue {var slot: lone Msg, var overflow: lone Msg}
21
22 pred QueuesUnchanged [xs: set Link, next_slot: Queue->lone Msg,
23   next_overflow: Queue->lone Msg] {
24   all x: xs |
25    x.queue.next_slot = x.queue.slot &&
26    x.queue.next_overflow = x.queue.overflow
27 }
28 pred ReadQueue [x: Link, next_slot: Queue->lone Msg,
29   next_overflow: Queue->lone Msg] {
30     no x.queue.(next_slot + next_overflow)
31   QueuesUnchanged [Link - x, next_slot, next_overflow]
32 }
33
34 pred PeekQueue [x: Link, m: Msg] {
35   m = x.queue.slot }
36 pred IsEmptyQueue [x: Link] {
37   no x.queue.(slot + overflow)
38 }
39 ---------------   links
40 sig Link {
41     source, target: Node,
42     queue : Queue
43 }
44 var sig ParentLinks in Link {}
45
46 fun reverse : Link -> Link {
47   {x: Link, y: Link |
48     y.target = x.source && y.source = x.target} }
49 -- each Link has its reverse:
50 fact { { all x: Link | reverse[x] in Link
51 } }
52 -- each Queue corresponds to one Link:
53 fact {queue in Link one -> Queue}
54
55 -- at most one link between a pair of nodes in a given direction:
56 fact {no disj x,y: Link | x.source = y.source && x.target = y.target}
57
58 -- topology is tree-like: acyclic when viewed as an undirected graph:
59 one sig Tree{  tree: Node->Node  }
60 {some root: Node {
61     tree in Node lone -> Node
62     Node in root.*tree
```

```
63      no ^tree & iden
64      tree + ~tree = ~source.target}
65  }
66  --------------   initialization
67  fact Initialization {
68      Node in Waiting
69      no ParentLinks
70      all x: Link | IsEmptyQueue [x]
71  }
72  --------------   9 actions
73  act Stutter {}
74
75  act AssignParent [x : Link] modifies ParentLinks, slot, overflow {
76      x.target in Waiting
77      !IsEmptyQueue [x]
78      ReadQueue [x,slot',overflow']
79      ParentLinks' = ParentLinks + x
80  }
81  -- now in Active, after in Contending
82  act ReadReqOrAck_AtoC [x : Link] modifies Active, Contending, slot, overflow {
83      x.target in Active
84      !PeekQueue [x, Ack]
85      !IsEmptyQueue [x]
86      ReadQueue [x,slot',overflow']
87      Active' = Active - x.target
88      Contending' = Contending + x.target
89  }
90  -- now in Contending, after in Active
91  act ReadReqOrAck_CtoA [x : Link] modifies Active, Contending, slot, overflow {
92      x.target in Contending
93      PeekQueue [x, Ack]
94      !IsEmptyQueue [x]
95      ReadQueue [x,slot',overflow']
96      Contending' = Contending - x.target
97      Active' = Active + x.target
98  }
99  -- now in Active, after in Active
100 act ReadReqOrAck_AtoA [x : Link] modifies  slot, overflow {
101     x.target in Active
102     PeekQueue [x, Ack]
103     !IsEmptyQueue [x]
104     ReadQueue [x,slot',overflow']
105 }
106 -- now in Contending, after in Contending
107 act ReadReqOrAck_CtoC [x : Link] modifies  slot, overflow {
108     x.target in Contending
109     !PeekQueue [x, Ack]
110     ! IsEmptyQueue [x]
111     ReadQueue [x,slot',overflow']
112 }
```

```
113 act Elect [n : Node] modifies Active, Elected {
114     n in Active
115     to_node[n] in ParentLinks
116     Active' = Active - n
117     Elected' = Elected + n
118 }
119 act WriteReqOrAck [n : Node] modifies Waiting, Active, slot, overflow {
120
121   n in Waiting
122   lone to_node[n] - ParentLinks
123
124   all x: from_node[n] |
125
126     no x.queue.slot =>
127     (  (x.queue.slot' = (reverse[x] in ParentLinks => Ack else Req))
128                            && (no x.queue.overflow')
129                          )
130       else {
131         x.queue.slot' = x.queue.slot
132         some x.queue.overflow'}
133
134   QueuesUnchanged [Link - from_node[n], slot', overflow']
135   Waiting' = Waiting - n
136   Active' = Active + n
137 }
138 act ResolveContention [x : Link] modifies Contending, Active, ParentLinks {
139     x.(source + target) in Contending
140     Contending' = Contending -  x.(source + target)
141     Active' = Active +  x.(source + target)
142     ParentLinks' = ParentLinks + x
143 }
144 ---------------   strong fairness
145 let SF [pre, post] { ((always eventually pre) implies (always eventually post)) }
146
147 pred strong_fairness {
148   all x : Link {
149     SF[pre_AssignParent[x], post_AssignParent[x]]
150     SF[pre_ReadReqOrAck_AtoC[x], post_ReadReqOrAck_AtoC[x]]
151     SF[pre_ReadReqOrAck_CtoA[x], post_ReadReqOrAck_CtoA[x]]
152     SF[pre_ReadReqOrAck_AtoA[x], post_ReadReqOrAck_AtoA[x]]
153     SF[pre_ReadReqOrAck_CtoC[x], post_ReadReqOrAck_CtoC[x]]
154     SF[pre_ResolveContention[x], post_ResolveContention[x]]
155   }
156   all n : Node {
157     SF[pre_Elect[n], post_Elect[n]]
158     SF[pre_WriteReqOrAck[n], post_WriteReqOrAck[n]]
159   }
160 }
161 ---------------   properties and run / check
162 -- in every state of the trace, there is at most one elected node
```

```
163 assert GoodSafety {
164       always { lone Elected }
165 }
166 -- no queue overflows
167 assert GoodSafety2 {  always { no overflow } }
168 -- there is some state in which a node has been elected
169 assert BadLiveness {
170   eventually { some Elected }
171 }
172 -- there is some state in which a node has been elected,
173 // under the hypothesis of strong fairness
174 assert BadLiveness2 {
175   strong_fairness => eventually some Elected
176 }
177 pred noBadRead {
178   always {
179     !(Node in (Active+Contending) && no slot && no {x:Link |
180       pre_ResolveContention[x]}}
181 }
182
183 -- there is some state in which a node has been elected,
184 -- assuming strong fairness and the non-occurrence of a state in which
185 -- each node is either Active or Contending while all queues are empty
186 -- and no contention can be solved.
187 assert GoodLiveness {
188   strong_fairness && noBadRead =>eventually some Elected
189 }
190
191 pred consistent {}
192
193 run consistent for 3 but 20 Time expect 1
194 run consistent for 4 but 20 Time expect 1
195 run consistent for 5 but 20 Time expect 1
196
197 // Firewire (0) scenario:  in every state of the trace,
198 // there is at most one node that has been elected
199 check GoodSafety for 3 but 20 Time expect 0
200 check GoodSafety for 4 but 20 Time expect 0
201 check GoodSafety for 5 but 20 Time expect 0
202
203 // Firewire (1) scenario: no queue overflows
204 /*
205 check GoodSafety2 for 3 but 20 Time expect 0
206 check GoodSafety2 for 4 but 20 Time expect 0
207 check GoodSafety2 for 5 but 20 Time expect 0
208
209 // Firewire (2) scenario: there is some state
210 //in which a node has been elected
211 check BadLiveness for 3 but 20 Time expect 1
212 check BadLiveness for 4 but 20 Time expect 1
```

```
213  check BadLiveness for 5 but 20 Time expect 1
214
215  // Firewire (3) scenario: there is some state
216  //in which a node has been elected
217  /*
218  check BadLiveness2 for 3 but 20 Time expect 1
219  check BadLiveness2 for 4 but 20 Time expect 1
220  check BadLiveness2 for 5 but 20 Time expect 1
221  */
222
223  // Firewire (4) scenario:  there is some state
224  //in which a node has been elected, under several assumptions.
225  check GoodLiveness for 3 but 20 Time expect 0
226  check GoodLiveness for 4 but 20 Time expect 0
227  check GoodLiveness for 5 but 20 Time expect 0
228
229  // DEFINED VARIABLES
230  // Defined variables are uncalled, no-argument functions.
231  // They are helpful for getting good visualization.
232  fun queued: Link → Msg {
233      {L: Link, m: Msg | m in L.(queue).slot}
234  }
```

Bibliographie

[ABK07] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M Küster. Analysis of model transformations via alloy. In *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.

[AFM+09] Nazareno M Aguirre, Marcelo F Frias, Mariano M Moscato, Thomas SE Maibaum, and Alan Wassyng. Describing and analyzing behaviours over tabular specifications using (dyn) alloy. In *International Conference on Fundamental Approaches to Software Engineering*, pages 155–170. Springer, 2009.

[Alca] Cunha Alcino. INESC TEC. Pardinus,V1.0, Available under the MIT License at. `https://github.com/haslab/pardinus/releases/tag/v1.0`. Accessed : 2018-07-24.

[Alcb] Cunha Alcino. INESC TEC. Pardinus,V1.0, Available under the MIT License at. `https://github.com/haslab/pardinus/releases/tag/v1.0`. Accessed : 2018-07-24.

[BCT18] Julien Brunel, David Chemouil, and Jeanne Tawa. Analyzing the fundamental liveness property of the chord protocol. In *Formal Methods in Computer-Aided Design*, 2018.

[BDH07] William J Bolosky, John R Douceur, and Jon Howell. The farsite project : a retrospective. *ACM SIGOPS Operating Systems Review*, 41(2) :17–26, 2007.

[BG07] Rana Bakhshi and Dilian Gurov. Verification of peer-to-peer algorithms : A case study. *Electronic Notes in Theoretical Computer Science*, 181 :35–47, 2007.

[BG08] Antonio Bucchiarone and Juan P Galeotti. Dynamic software architectures verification using dynalloy. *Electronic Communications of the EASST*, 10, 2008.

[BGG14] Pablo Bendersky, Juan Pablo Galeotti, and Diego Garbervetsky. The dynalloy visualizer. *arXiv preprint arXiv :1401.0973*, 2014.

[BL02] Brannon Batson and Leslie Lamport. High-level specifications : Lessons from industry. In *International Symposium on Formal Methods for Components and Objects*, pages 242–261. Springer, 2002.

[BMR95] Alexander Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10) :785–798, 1995.

[BNOG04] Johannes Borgström, Uwe Nestmann, Luc Onana, and Dilian Gurov. Verifying a structured peer-to-peer overlay network : The static case. In *International Workshop on Global Computing*, pages 250–265. Springer, 2004.

[Bol05] Christie Bolton. Using the alloy analyzer to verify data refinement in z. *Electronic Notes in Theoretical Computer Science*, 137(2) :23–44, 2005.

[Bon12] Francesco Bongiovanni. *Design, formalization and implementation of overlay networks : application to RDF data storage*. PhD thesis, Nice, 2012.

[Bow01] Jonathan P Bowen. Z : A formal specification notation. In *Software specification methods*, pages 3–19. Springer, 2001.

[C+99] Ian Clarke et al. *A distributed decentralised information storage and retrieval system*. PhD thesis, Master's thesis, University of Edinburgh, 1999.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2 : An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[CCGR00] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4) :410–425, 2000.

[CDL+12] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. Tla+ proofs. In *International Symposium on Formal Methods*, pages 147–154. Springer, 2012.

[CGKP06] C Chen, P Grisham, Sarfraz Khurshid, and D Perry. Design and validation of a general security model with the alloy analyzer. In *Proceedings of the ACM SIGSOFT First Alloy Workshop*, pages 38–47. Citeseer, 2006.

[CGL94] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5) :1512–1542, 1994.

[CL93] René Cori and Daniel Lascar. *Logique mathématique : cours et exercices. Calcul propositionnel, algèbres de Boole, calcul des prédicats*. Masson, 1993.

[Cun14] Alcino Cunha. Bounded model checking of temporal formulas with Alloy. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 303–308. Springer, 2014.

[DGFGL13] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Leslie Lamport. Adaptive register allocation with a linear number of registers. In *International Symposium on Distributed Computing*, pages 269–283. Springer, 2013.

[DNR+01] René David, Karim Nour, Christophe Raffalli, et al. *Introduction à la logique : théorie de la démonstration : cours et exercices corrigés*. Dunod, 2001.

[DS90] EW Dijkstra and CS Scholten. Predicate calculus and programming semantics. *New York : Spring-Verlag*, pages 1–80, 1990.

[EJT04] Jonathan Edwards, Daniel Jackson, and Emina Torlak. A type system for object models. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 189–199. ACM, 2004.

[FBHL73] Abraham Adolf Fraenkel, Yehoshua Bar-Hillel, and Azriel Levy. *Foundations of set theory*, volume 67. Elsevier, 1973.

[FG06] Marcelo F Frias and Juan P Galeotti. Faster sat-based analysis of oo-programs by separation of mutant and non mutant objects. In *First Alloy Workshop*, 2006.

[FGLPA05] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. Dynalloy : Upgrading alloy with actions. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 442–451. ACM, 2005.

[FLPB+05] Marcelo F Frias, Carlos G López Pombo, Gabriel A Baum, Nazareno M Aguirre, and Thomas SE Maibaum. Reasoning about static and dynamic properties in alloy : A purely relational approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(4) :478–526, 2005.

[FLPGA07] Marcelo F Frias, Carlos G Lopez Pombo, Juan P Galeotti, and Nazareno M Aguirre. Efficient analysis of dynalloy specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(1) :4, 2007.

[FPB+03] Marcelo F Frias, Carlos G López Pombo, Gabriel A Baum, Nazareno M Aguirre, and Tom Maibaum. Taking alloy to the movies. In *International Symposium of Formal Methods Europe*, pages 678–697. Springer, 2003.

[FZL10] Xu Fei, Ai Ming Zhang, and Wan Liang. Formalizing and checking set protocol based on tla. In *E-Product E-Service and E-Entertainment (ICEEE), 2010 International Conference on*, pages 1–3. IEEE, 2010.

[GF07] Juan P. Galeotti and Marcelo F. Frias. Dynalloy as a formal method for the analysis of java programs. In Krzysztof Sacha, editor, *Software Engineering Techniques : Design for Quality*, pages 249–260, Boston, MA, 2007. Springer US.

[Gho06] Ali Ghodsi. *Distributed k-ary system : Algorithms for distributed hash tables.* PhD thesis, KTH-Royal Institute of Technology, 2006.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[GMB06] Rohit Gheyi, Tiago Massoni, and Paulo Borba. A theory for feature models in alloy. In *First alloy workshop*, pages 71–80. Citeseer, 2006.

[Gon01] Li Gong. JXTA : A network programming environment. *IEEE Internet Computing*, 5(3) :88–95, 2001.

[GPCR12] Ana Garis, Ana CR Paiva, Alcino Cunha, and Daniel Riesco. Specifying uml protocol state machines in alloy. In *International Conference on Integrated Formal Methods*, pages 312–326. Springer, 2012.

[Gs96] Bernard Geveau and LS schulman. Master equation based formulation of nonequibrium statiscal. *journal of Mathematical Physique*, 37(8) :3897–3932, 1996.

[Hon00] Theodore Hong. A distributed anonymous information storage and retrieval system. In *ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science : Modelling and reasoning about systems.* Cambridge University Press, 2004.

[HT18] Amy House and Peiyi Tang. A tla+ module for asynchronous message-passing systems. In *SoutheastCon 2018*, pages 1–7. IEEE, 2018.

[Jac03] Daniel Jackson. Alloy : A logical modelling language. *ZB*, 2651 :1, 2003.

[Jac06] Daniel Jackson. *Software abstractions*, volume 2. MIT press Cambridge, 2006.

[Jac12] Daniel Jackson. *Software Abstractions : logic, language, and analysis.* MIT press, 2012.

[JLLV04] James E Johnson, David E Langworthy, Leslie Lamport, and Friedrich H Vogt. Formal specification of a web services protocol. *Electronic Notes in Theoretical Computer Science*, 105 :147–158, 2004.

[JLM+03] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, and Yuan Yu. Checking cache-coherence protocols with tla+. *Formal Methods in System Design*, 22(2) :125–131, 2003.

[KDN11] Yannick L Konga, Karim Djouani, and Guillaume Noel. Modelling and verification of JXTA peer-topeer network protocols. In *Proceedings of the Fifth international conference on Verification and Evaluation of Computer and Communication Systems*, pages 96–107. British Computer Society, 2011.

[KEAAH05] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi. A statistical theory of chord under churn. In *International Workshop on Peer-to-Peer Systems*, pages 93–103. Springer, 2005.

[KKPB07] Martin Karsten, S. Keshav, Sanjiva Prasad, and Mirza Beg. An axiomatic basis for communication. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 217–228. ACM, 2007.

[Kum12] Apurva Kumar. Using automated model analysis for reasoning about security of web protocols. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 289–298. ACM, 2012.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3) :872–923, 1994.

[Lam02a] Leslie Lamport. Specifying systems, 2002.

[Lam02b] Leslie Lamport. *Specifying systems : the TLA^+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[LBP10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7 :59–64, 2010.

[Lio96] J.L et al Lions. Ariane 5. flight 501 failure. report by the inquiry board, 1996. *Retrievable via : http ://www. esa. int*, 1996.

[LLOR99] Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy caching in tla. *Distributed Computing*, 12(2-3) :151–174, 1999.

[LMP04] Xiaozhou Li, Jayadev Misra, and C Greg Plaxton. Active and concurrent topology maintenance. In *International Symposium on Distributed Computing*, pages 320–334. Springer, 2004.

[LMP06] Xiaozhou Li, Jayadev Misra, and C Greg Plaxton. Concurrent maintenance of rings. *Distributed Computing*, 19(2) :126–148, 2006.

[LMW10] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Model Checking the Pastry Routing Protocol. In Jens Bendisposto, Michael Leuschel, and Markus Roggenbach, editors, *10th International Workshop Automated Verification of Critical Systems*, 10th International Workshop Automated Verification of Critical Systems, pages 19–21, Düsseldorf, Germany, September 2010. Universität Düsseldorf. short communication.

[LMW11] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Towards verification of the pastry protocol using tla+. In *Formal Techniques for Distributed Systems*, pages 244–258. Springer, 2011.

[LMW12] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Formal Verification Of Pastry Using TLA+. In Leslie Lamport and Stephan Merz, editors, *International Workshop on the TLA+ Method and Tools*, Paris, France, August 2012.

[LNBK02] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 233–242. ACM, 2002.

[LPR16] Giuliano Losa, Sebastiano Peluso, and Binoy Ravindran. Brief announcement : A family of leaderless generalized-consensus algorithms. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 345–347. ACM, 2016.

[LS04] N Lynch and I Stoica. MultiChord : A resilient namespace management algorithm. Technical report, Technical Memo MIT-LCS-TR-936 2004, 2004.

[LSTY01] Leslie Lamport, Madhu Sharma, Mark Tuttle, and Yuan Yu. The wildfire challenge problem, 2001.

[Lu13a] Tianxiang Lu. *Formal verification of the pastry protocol.* PhD thesis, Université de Lorraine, 2013.

[Lu13b] Tianxiang Lu. *Formal verification of the Pastry protocol.* Theses, Université de Lorraine, November 2013.

[Lu15] Tianxiang Lu. "formal verification of the pastry protocol using tla+. In *International Symposium on Dependable Software Engineering : Theories, Tools, and Applications*, pages 284–299. Springer, 2015.

[Mar03] Nicolas Markey. *Logiques temporelles pour la vérification : expressivité, complexité, algorithmes.* PhD thesis, Orléans, 2003.

[Mar10] Jerzy Martyna. Linking simulation with formal verification and modeling of wireless sensor network in tla+. In *International Conference on Computer Networks*, pages 131–140. Springer, 2010.

[MBC+16] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations. In *Foundations of Software Engineering*, 2016.

[MC16] Nuno Macedo and Alcino Cunha. Alloy meets tla+ : An exploratory study. *arXiv preprint arXiv :1603.03599*, 2016.

[MD88] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Symposium Proceedings on Communications Architectures and Protocols*, pages 123–133. ACM, 1988.

[ME15] Saber Mirzaei and Flavio Esposito. An alloy verification model for consensus-based auction protocols. In *Distributed Computing Systems Workshops (ICDCSW), 2015 IEEE 35th International Conference on*, pages 17–22. IEEE, 2015.

[MFdJ97] Abdelillah Mokkedem, Michael J Ferguson, and Robert de Johnston. A tla solution to the specification and verification of the rlp1 retransmission protocol. In *International Symposium of Formal Methods Europe*, pages 398–417. Springer, 1997.

[MLW11] Stephan Merz, Tianxiang Lu, and Christoph Weidenbach. Towards Verification of the Pastry Protocol using TLA^+. In *31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, volume 6722, 2011.

[MS08] Philippe Merle and Jean-Bernard Stefani. *A formal specification of the Fractal component model in Alloy.* PhD thesis, INRIA, 2008.

[NRZ+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4) :66–73, 2015.

[PGAF] CG Lopez Pombo, JP Galeotti, NM Aguirre, and MF Frias. Demo DynAlloy. `https://sites.google.com/site/dynalloy/home/fse-2017---demo`. Accessed : 2019-06-04.

[PKPZ07] Rodion Podorozhny, Sarfraz Khurshid, Dewayne Perry, and Xiaoqin Zhang. Verification of multi-agent negotiations using the alloy analyzer. In *International Conference on Integrated Formal Methods*, pages 501–517. Springer, 2007.

[PSK+11] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M Pai, and Sanjay Singh. Formal verification of oauth 2.0 using alloy framework. In *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*, pages 655–659. IEEE, 2011.

[PTS96] Ben Potter, David Till, and Jane Sinclair. *An introduction to formal specification and Z.* Prentice Hall PTR, 1996.

[Rat03] Michael Rathjen. Realizability for constructive zermelo-fraenkel set theory. In *Logic Colloquium*, volume 3, pages 282–314, 2003.

[RCGB+17] Germán Regis, César Cornejo, Simón Gutiérrez Brida, Mariano Politano, Fernando Raverta, Pablo Ponzio, Nazareno Aguirre, Juan Pablo Galeotti, and Marcelo Frias. Dynalloy analyzer : a tool for the specification and analysis of alloy models with dynamic behaviour. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 969–973. ACM, 2017.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4) :161–172, August 2001.

[RGR+04] Sean Rhea, Dennis Geels, Timothy Roscoe, John Kubiatowicz, et al. Handling churn in a dht. In *Proceedings of the USENIX Annual Technical Conference*, volume 6, pages 127–140. Boston, MA, USA, 2004.

[RM06] John Risson and Tim Moors. Survey of research towards robust peer-to-peer networks : Search methods. *Computer networks*, 50(17) :3485–3521, 2006.

[RRM05] John Risson, Ken Robinson, and Tim Moors. Fault tolerant active rings for structured peer-to-peer overlays. In *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*, pages 18–25. IEEE, 2005.

[SE05] Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53) :1–2, 2005.

[SE09] Niklas Sörensson and Niklas Eén. Minisat 2.1 and minisat++ 1.0-sat race 2008 editions. *SAT*, page 31, 2009.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4) :149–160, 2001.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord : a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1) :17–32, 2003.

[Spi92] J. M. Spivey. *The Z Notation : A Reference Manual.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[TD06] Emina Torlak and Greg Dennis. Kodkod for alloy users. In *First ACM Alloy Workshop, Portland, Oregon*, 2006.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod : A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[Vil] Arak Villu. What happened on august 16.Blog,2007. `http://heartbeat.skype.com/2007/08/what_happened_on_august_16.html`. Accessed : 2018-08-30.

[WLG08] Hongbing Wang, Hui Liu, and Xiaohui Guo. Specify and compose web services by tla. In *Web Services, 2008. ICWS'08. IEEE International Conference on*, pages 766–767. IEEE, 2008.

[WWP+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi : A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368. ACM, 2015.

[Zav11] Pamela Zave. Why the Chord ring-maintenance protocol is not correct. Technical report, AT&T Research, Tech. Rep, 2011.

[Zav12] Pamela Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2) :49–57, 2012.

[Zav15a] Pamela Zave. A practical comparison of Alloy and SPIN. *Formal Aspects of Computing*, 27(2) :239, 2015.

[Zav15b] Pamela Zave. How to Make Chord Correct. *arXiv preprint arXiv :1502.06461*, 2015.

[Zav17] Pamela Zave. Reasoning about identifier spaces : How to make Chord correct. *arXiv preprint arXiv :1610.01140*, 2017.

[Zav19] Pamela Zave. Lightweight modeling of network protocols in alloy. 06 2019.