UNIVERSITÉ JOSEPH FOURIER – GRENOBLE I

THÈSE

pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Spécialité : Informatique

préparée au laboratoire VÉRIMAG

dans le cadre de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

présentée et soutenue publiquement par

Yliès C. FALCONE

le 09 Novembre 2009

ÉTUDE ET MISE EN ŒUVRE DE TECHNIQUES DE VALIDATION À L'EXÉCUTION

Composition du Jury

Howard BARRINGER *Manchester University*	Examinateur
Ahmed BOUAJJANI *LIAFA – Université de Paris 7*	Examinateur
Jean-Claude FERNANDEZ VÉRIMAG – *Université Joseph Fourier – Grenoble*	Directeur
Klaus HAVELUND *JPL, NASA, USA – California Institute of Technology, CA, USA*	Rapporteur
Thierry JÉRON *IRISA – INRIA de Rennes*	Rapporteur
Jean-François MÉHAUT *LIG – Université Joseph Fourier – Grenoble*	Examinateur
Laurent MOUNIER VÉRIMAG – *Université Joseph Fourier – Grenoble*	Co-Directeur
Jean-Luc RICHIER *LIG – CNRS – Grenoble*	Co-Directeur

Étude et mise en œuvre de techniques de validation à l'exécution

Study and implementation of runtime validation techniques

Résumé

L'étude de cette thèse porte sur trois méthodes de validation dynamiques : les méthodes de vérification, d'enforcement (mise en application), et de test de propriétés lors de l'exécution des systèmes. Nous nous intéresserons à ces approches en l'absence de spécification comportementale du système à valider. Pour notre étude, nous nous plaçons dans la classification *Safety-Progress* des propriétés. Ce cadre offre plusieurs avantages pour la spécification de propriétés sur les systèmes. Nous avons adapté les résultats de cette classification, initialement développés pour les séquences infinies, pour prendre en compte de manière uniforme les séquences finies. Ces dernières peuvent être vues comme une représentation abstraite de l'exécution d'un système. Se basant sur ce cadre général, nous nous sommes intéressés à l'applicabilité des méthodes de validation dynamiques. Nous avons caractérisé les classes de propriétés vérifiables, enforçables, et testables. Nous avons ensuite proposé trois approches génériques de vérification, d'enforcement, et de test de propriétés operant à l'exécution des systèmes. Nous avons montré comment il était possible d'obtenir, à partir d'une propriété exprimée dans le cadre de la classification Safety-Progress, des mécanismes de vérification, d'enforcement, ou de test dédiés à la propriété considérée. Enfin, nous proposons également les outils J-VETO et J-POST mettant en œuvre l'ensemble des résultats proposés sur les programmes Java.

Mots clés : vérification à l'exécution, enforcement à l'exécution, test sans spécification comportementale, propriété, classification safety-progress, applicabilité de techniques, J-VETO, J-POST.

Abstract

This thesis deals with three dynamic validation techniques : runtime verification (monitoring), runtime enforcement, and testing from property. We consider these approaches in the absence of complete behavioral specification of the system under scrutiny. Our study is done in the context of the *Safety-Progress* classification of properties. This framework offers several advantages for specifying properties on systems. We adapt the results on this classification, initially dedicated to infinite sequences, to take into account finite sequences. Those sequences may be considered as abstract representations of a system execution. Relying on this general framework, we study the applicability of dynamic validation methods. We characterize the classes of monitorable, enforceable, and testable properties. Then, we proposed three generic approaches for runtime verification, enforcement, and testing. We show how it is possible to obtain, from a property expressed in the *Safety-Progress* framework, some verification, enforcement, and testing mechanisms for the property under consideration. Finally, we propose the tools J-VETO and J-POST implementing all the aforementioned results on Java programs.

Key words : runtime verification, runtime enforcement, test without behavioral specification, property, safety-progress classification, applicability of techniques, J-VETO, J-POST.

Articles associés

Certaines parties de cette thèse sont apparues à plusieurs endroits sous forme d'articles ou de rapports techniques VÉRIMAG.

Certains résultats du Chapitre 3 Sections 3.6 et 3.2 sont apparus dans [FFM08] lors de la conférence ICISS (International Conference on Information and System Security) et dans [FFM09a] lors de la conférence SAC-SVT (Symposium on Applied Computing - Software Verification and Testing).

Les résultats du Chapitre 4 des Sections 4.2 et 4.3 sur l'applicabilité des méthodes de vérification et d'enforcement sont apparus dans [FFM09b] lors du workshop RV'09 (Runtime Verification).

Une partie des résultats du Chapitre 6 ont été soumis au journal Formal Methods in System Design [FFMR08].

Certaines parties des résultats du Chapitre 7 ont été publiées dans [FFMR06] lors du workshop FATES/RV'06 (Formal Approaches to TESting/Runtime Verification) et dans [FFMR07a] lors de la conférence TestCOM/Fates'07 (TESTing of COMmunicating Systems/Formal Approaches to TESting).

L'outil j-POST a été présenté précédemment dans [FMFR08b] au workshop MBT'08 (Model-Based Testing). Également, un rapport technique [FMFR08a] est disponible. Celui-ci contient plus d'information sur l'utilisation de la chaîne d'outils j-POST.

Table des matières

Introduction **1**

1 Notations **11**
1.1 À propos des séquences, et séquences d'exécution . . . 11
1.2 Expressions régulières et omega-régulières . . . 12
1.3 Propriétés, propriétés finitaires, propriétés infinitaires . . . 13

2 État de l'art **15**
2.1 Vérification de propriétés à l'exécution . . . 17
2.1.1 Notion de moniteur . . . 18
2.1.2 Les autres saveurs de la vérification à l'exécution . . . 21
2.1.3 Caractérisations des propriétés vérifiables à l'exécution . . . 21
2.1.4 Perspectives et futurs challenges . . . 22
2.2 Enforcement de propriétés à l'exécution . . . 23
2.2.1 Bref historique des approches d'enforcement par moniteur . . . 23
2.2.2 Pouvoir calculatoire des mécanismes d'enforcement . . . 23
2.2.3 Caractérisations de l'ensemble des propriétés enforçables avec des moniteurs d'exécution . . . 24
2.2.4 Synthèse de mécanismes d'enforcement à l'exécution . . . 25
2.2.5 Quelques approches d'enforcement . . . 25
2.3 Test de propriétés . . . 25
2.3.1 Caractérisation des propriétés testables . . . 26
2.3.2 Approches pour le test de propriétés . . . 26
2.3.3 Test de propriétés sans spécification comportementale . . . 27

I Cadre général **29**

3 Spécification dans la classification Safety-Progress **31**
3.1 Introduction . . . 33
3.1.1 La classification Safety-Liveness . . . 33
3.1.2 La classification Safety-Progress (résultats précédents) . . . 35
3.1.3 Motivations et contributions . . . 35
3.2 Description informelle de la classification . . . 36
3.3 r-propriétés : Runtime properties . . . 38
3.4 La vue langage des r-propriétés . . . 39
3.4.1 Construction de propriétés finitaires et infinitaires . . . 39
3.4.2 La hiérarchie des langages . . . 42
3.5 La vue logique temporelle des r-propriétés . . . 44

3.5.1 Syntaxe et sémantique . . . 44
3.5.2 Hiérarchie des formules temporelles . . . 46
3.6 La vue automates des r-propriétés . . . 49
3.6.1 Automates de Streett . . . 49
3.6.2 La hiérarchie des automates . . . 50
3.6.3 Synthèse d'automates de Streett à partir de DFAs . . . 51
3.7 La vue topologique des r-propriétés (Remarque) . . . 59
3.8 Connexions entre les différentes vues . . . 59
3.9 Conclusion et perspectives . . . 60

4 Applicabilité des méthodes de validation à l'exécution 63
4.1 Introduction . . . 65
4.2 Monitorabilité dans la classification Safety-Progress . . . 65
4.2.1 Selon la définition classique du monitoring . . . 66
4.2.2 Une définition alternative du monitoring . . . 69
4.3 Enforçabilité par rapport à la classification *Safety-Progress* . . . 73
4.3.1 Critères d'enforcement . . . 73
4.3.2 Caractérisation des propriétés enforçables . . . 75
4.4 Testabilité dans la classification *Safety-Progress* . . . 78
4.4.1 Notion de testabilité . . . 78
4.4.2 Caractérisation des propriétés testables . . . 80
4.4.3 Raffinement de la notion de verdict . . . 85
4.4.4 Introduction de la notion de quiescence . . . 86
4.5 Conclusion et perspectives . . . 87

II Approches pour la validation à l'exécution 91

5 Préliminaires aux approches 93
5.1 À propos des états d'un automate de Streett . . . 93
5.2 Retour sur les notions de monitorabilité et testabilité . . . 95
5.3 Notion de moniteur . . . 96

6 Approches Vérification et Enforcement de propriétés 97
6.1 Introduction . . . 99
6.2 Vérification par moniteur . . . 99
6.3 Enforcement par moniteur . . . 100
6.3.1 Notion de moniteur d'enforcement générique . . . 100
6.3.2 Enforcement de propriétés par un moniteur. . . . 102
6.3.3 Instantiation des moniteurs d'enforcement génériques . . . 103
6.3.4 Propriétés des moniteurs d'enforcement instanciés . . . 104
6.4 Operations de composition sur les moniteurs d'enforcement . . . 105
6.4.1 Négation . . . 106
6.4.2 Union et intersection . . . 108
6.5 Synthèse de moniteurs d'enforcement . . . 110
6.5.1 Transformations spécifiques aux classes de propriétés . . . 111
6.5.2 Correction des transformations . . . 115
6.5.3 Transformation générale . . . 118
6.6 Comparaison avec d'autres approches d'enforcement . . . 119
6.6.1 Comparaison avec les mécanismes d'enforcement . . . 119
6.6.2 Comparaison avec les moniteurs d'enforcement . . . 119
6.7 Conclusion . . . 120

7 Approche test **121**
7.1 Introduction . . . 123
7.2 Principe de l'approche . . . 124
7.3 Représentation algébrique des processus de test . . . 125
7.4 Phase de génération de tests . . . 127
7.5 Phase d'instanciation des tests . . . 128
7.6 Phase de sélection et d'exécution des tests . . . 130
7.7 Approche compositionnelle dirigée par la syntaxe . . . 131
7.7.1 Principe général . . . 131
7.7.2 Génération des tests . . . 132
7.8 Conclusion et perspectives . . . 133

III Mise en œuvre pratique **135**

8 j-VETO : a Java Verification and Enforcement TOolbox **137**
8.1 Introduction . . . 139
8.2 Spécification de la propriété par l'utilisateur . . . 139
8.3 Vue d'ensemble . . . 141
8.4 Synthèse de moniteurs . . . 142
8.4.1 DFA2Streett : synthèse d'automates de Streett depuis des DFAs et des motifs . . . 142
8.4.2 Streett2Monitor : synthèse de moniteurs depuis les automates de Streett. . . . 144
8.5 Intégration du moniteur . . . 147
8.5.1 Mapping2Aspect : Génération d'aspects à partir de mappings . . . 147
8.5.2 MonitorTranslator : traduction de moniteurs en Java . . . 148
8.5.3 Bibliothèque pour le slicing . . . 150
8.6 Modèle et utilitaires communs pour les automates . . . 152
8.6.1 GraphMaker : des graphes pour les DFAs, automates de Streett, et moniteurs . . . 152
8.6.2 Monitor Composer : Composition de moniteurs . . . 153
8.6.3 Modèle pour les objets de type automate . . . 153
8.7 Éxécution du programme avec son moniteur . . . 153
8.7.1 Expérimentations en vérification à l'exécution . . . 154
8.7.2 Expérimentations en enforcement à l'exécution . . . 155
8.8 Conclusion et Perspectives . . . 155

9 j-POST : Property-Oriented Software Testing **157**
9.1 Introduction . . . 159
9.2 Conception des tests . . . 160
9.2.1 Exemple . . . 161
9.3 Génération des tests . . . 162
9.3.1 Vue d'ensemble de la génération de test . . . 162
9.3.2 Analyse syntaxique de l'exigence . . . 162
9.3.3 Implantation de la fonction de génération de test . . . 165
9.3.4 Exemple . . . 167
9.4 Exécution des tests . . . 167
9.4.1 Vue d'ensemble . . . 167
9.4.2 Principe de fonctionnement . . . 169
9.4.3 Concrétisation des interactions avec l'IUT . . . 170
9.4.4 Exemple . . . 173
9.5 Conclusion et perspectives . . . 174

IV Conclusions et Perspectives **175**

Conclusion et Perspectives **177**

Bibliographie **191**

V Annexes 193

A Preuves 195
A.1 Preuves du Chapitre 3 . . . 195
A.1.1 Propriétés des opérateurs Ef et E . . . 195
A.2 Preuves du Chapitre 4 . . . 196
A.3 Preuves du Chapitre 5 . . . 196
A.4 Preuves du Chapitre 6 . . . 197
A.4.1 Preuve de la transformation spécifique aux r-propriétés de guarantee . . . 197
A.4.2 Preuve de la transformation spécifique aux r-propriétés de response . . . 198

B Usage of j-VETO and j-POST 201
B.1 Usage of j-VETO . . . 201
B.1.1 DFA2Streett . . . 201
B.1.2 streett2Monitor . . . 201
B.1.3 compositionEM . . . 202
B.2 Usage of j-POST . . . 202
B.2.1 Test generator . . . 202
B.2.2 Test engine . . . 203

C Autres travaux 205
C.1 A Test Calculus Framework Applied to Network Security Policies . . . 205
Bibliographie . . . 221
C.2 A Compositional Testing Framework Driven by Partial Specifications . . . 221
Bibliographie . . . 238
C.3 j-POST : A Java Toolchain for Property-Oriented Software Testing . . . 238

D Liste des définitions, théorèmes, lemmes, propositions, et exemples 253
D.1 Liste des définitions . . . 253
D.2 Liste des Lemmes, Théorèmes, et Propositions . . . 254
D.3 Liste des Exemples . . . 254

Introduction

Les dernières décennies ont vu l'émergence d'un monde où les systèmes informatiques sont omniprésents. Bien que science assez jeune, l'informatique a bénéficié d'un développement considérable. Nous allons aujourd'hui vers un monde où il n'existera vraisemblablement plus une simple tâche sans système informatique ou numérique impliqué. De l'application pour téléphone portable aux systèmes avioniques, les systèmes informatiques et les logiciels qui y sont enfouis nous rendent de précieux services au quotidien. L'accroissement de la diffusion de ces systèmes n'a d'égal que notre besoin de confiance dans leur bon fonctionnement. Nous souhaitons des systèmes toujours plus fiables, réalisant toujours plus de fonctionnalités, et de manière sûre.

Malheureusement, les systèmes informatiques ont maintes fois montré des défaillances, avec des conséquences plus ou moins catastrophiques. Ces conséquences peuvent être économiques, notamment lorsqu'un fabricant doit rappeler un produit soit parce qu'une partie du logiciel est défectueux ou soit parce qu'un logiciel bogué contrôle une plate-forme matérielle coûteuse. Des exemples célèbres existent : la panne du réseau téléphonique aux États Unis ou le crash de la fusée Ariane 5. Pour certains systèmes dits *systèmes critiques*, une défaillance logicielle peut avoir des conséquences inacceptables (*e.g.*, pertes humaines).

Les défaillances des systèmes informatiques dans l'histoire ont montré les limites des méthodes d'ingénierie existantes. Dès lors, la volonté de fournir des garanties fortes sur le bon fonctionnement des systèmes informatisés a fait émerger une communauté dite des *méthodes formelles* [Sif96, MS97, Sai98, CW96, Hei98]. Les efforts de recherche se sont focalisés sur la conception de modèles mathématiques et la vérification de propriétés sur ces modèles de systèmes informatiques. Même si ces méthodes ne peuvent garantir de manière absolue le comportement du système, ces techniques permettent de mettre en évidence les erreurs et les ambiguïtés lors de la conception de ces systèmes. Ainsi, les efforts de recherche ont permis d'améliorer l'assurance dans le fait que les systèmes se comportent de manière attendue. Une partie de cette communauté se concentre sur la partie logicielle des systèmes et à leur vérification.

La problématique de la vérification de logiciels

Dans cette thèse nous nous plaçons dans le contexte de la vérification de logiciels. En première approximation, nous pouvons dire que vérifier un logiciel consiste à s'assurer que celui-ci satisfera un ensemble d'exigences désirées lors de sa mise en service.

La vérification pour prévenir les fautes sur les programmes

Une *défaillance logicielle* est une déviation entre le comportement produit et le comportement requis du système logiciel. Une *faute* est définie comme étant une déviation entre le comportement courant et le comportement attendu. Ceci est identifié typiquement par une déviation entre l'état courant et l'état attendu du système. Une faute peut éventuellement conduire à une défaillance. Prévenir les fautes est donc un moyen de s'assurer que le système n'aura pas de défaillance. Pour découvrir et éliminer les

éventuelles fautes d'un logiciel, il est possible de le *vérifier*. La définition de l'activité de vérification donnée par l'IEEE est la suivante :

Définition 1 (Vérification [IEE05]). La vérification comprend toutes les techniques appropriées pour montrer qu'un système satisfait sa spécification. L'ensemble des techniques de vérification traditionnelles comprend le theorem-proving [BC04], le model-checking [CGP99], et le test [BJK+05].

A propos des méthodes de vérification

Pour déterminer si le programme satisfait sa spécification, les méthodes de vérification utilisent de manière sous-jacente une technique d'analyse. Rappelons tout d'abord que le problème de la vérification de programme est indécidable en général [Tur36]. Ainsi, pour être automatisables, les méthodes de validation doivent se restreindre à utiliser des techniques d'analyse où les propriétés sont décidables.

Les techniques d'analyse sont dites *statiques* lorsqu'elles ne nécessitent pas une exécution du programme pour opérer. L'avantage de ces méthodes est leur exhaustivité : la spécification est vérifiée sur l'intégralité du modèle ou du programme. Les techniques les plus communes de cette catégorie sont le theorem proving (preuve de théorème) appliqué à la preuve de programmes, le model-checking (vérification de modèles), et l'interprétation abstraite (static analysis). Parmi ces techniques, nous pouvons distinguer le theorem-proving, qui est une technique semi-automatique, des techniques d'interprétation abstraite et de model-checking qui sont complètement automatiques.

Une méthode semi-automatique : le theorem-proving. Le *theorem-proving* [Hoa69, Flo67] consiste à guider un assistant de preuve interactif pour montrer qu'une certaine propriété est vérifiée sur un programme. L'utilité de cette technique vient du fait qu'elle permet de décharger l'utilisateur de certaines obligations de preuves. De plus, l'espace des propriété qu'il est possible de traiter avec les assistants de preuve n'est pas réduit au domaine des propriétés décidables. Cependant, cette technique nécessite l'interaction d'un utilisateur dont la responsabilité est de fournir la propriété ou l'invariant à vérifier. De plus, l'applicabilité de cette méthode peut être délicate lorsque la taille du programme devient conséquente. À noter que, pour certaines propriétés décidables, des "stratégies" peuvent rendre la démonstration de propriétés automatique.

Deux méthodes automatiques : l'interprétation abstraite et le model checking. Le point commun des techniques automatiques est leur exhaustivité. En effet, ces techniques ne se basent pas sur une exécution concrète pour opérer, mais sur une représentation abstraite de l'intégralité des séquences d'exécution d'un programme. Dans le cas de l'interprétation abstraite, elle se base sur le code (source ou objet) du programme. Dans le cas du model-checking, c'est un modèle de l'ensemble des séquences d'exécution, exprimé dans une représentation de plus haut niveau, qui est utilisé.

La technique d'*interprétation abstraite* [CC92] est une technique d'approximation qui permet d'obtenir une représentation de l'ensemble des exécutions d'un programme. Cette technique peut être également utilisée pour réduire l'espace d'états du programme à vérifier et permet, par des approximations, de se ramener à des modèles où les théories sont décidables. Des problèmes spécifiques comme l'initialisation de variables, les violations de bornes de tableau peuvent même être traités dans certains cas uniquement en utilisant l'interprétation abstraite. Les erreurs de concurrence, assez complexes par nature, peuvent être également analysées statiquement et de manière efficace ; comme par exemple récemment pour la détection de deadlock [BBNS09]. En revanche, du fait que cette technique utilise des méthodes de sur-approximation pour faire face, par exemple, à l'expressivité des langages de programmation des programmes vérifiés, elle souffre de plusieurs limitations. Une première limitation est la production de *faux-positifs*. C'est-à-dire que ces techniques détectent des erreurs qui ne se produisent pas réellement. Une deuxième limitation est que la technique d'interprétation abstraite reste cantonnée à la vérification de propriétés spécifiques, laissant de coté les spécifications de taille arbitrairement complexe. De plus, l'applicabilité de cette méthode peut être limitée lorsque, après sa mise en service, le système est placé dans un environnement. Pour les *systèmes réactifs*, *i.e.*, les systèmes qui maintiennent une interaction continue avec leur environnement, les abstractions utilisées peuvent porter également sur l'environnement. Les abstractions de ce dernier sont en général assez difficiles à produire du fait de la difficulté de reproduire les conditions réelles dans lesquelles

un système réactif est placé. Les modèles du système et de son environnement sont donc approchés car ils reposent sur des abstractions.

Le *model checking* [CGP99] trouve son origine dans les travaux concurrents de Queille et Sifakis [Sif82], et de Clarke et Emerson [EC80]. Cette technique consiste à vérifier si le modèle du programme satisfait une propriété donnée. Lorsque le modèle du programme utilisé est fini et peut être exploré facilement, cette exploration peut devenir exhaustive. Cependant, lorsque la taille du système à vérifier ou de la propriété considérée devient conséquente, l'espace d'états à explorer devient trop grand pour être analysé, voire infini. C'est le problème de l'*explosion d'états*. Toutefois, des techniques d'approximation peuvent être utilisées pour extraire une représentation finie ou de taille réduite de celui-ci, *e.g.*, dans l'outil PathFinder [HP00]. Cependant un problème inhérent à cette méthode est un problème de passage à l'échelle. De plus, la technique de model checking repose sur la disponibilité d'une spécification formelle du système à vérifier. En pratique, celle-ci peut ne pas l'être ou être incomplète. Également, en model checking, la vérification porte sur un modèle de programme. À supposer que le modèle ou la spécification d'un programme soit complètement vérifié et ne comporte pas de faute, rien ne garantit qu'il en soit de même pour l'implémentation réelle produite à partir de la spécification ou du modèle.

De la nécessité de méthodes complémentaires. Bien que dans l'idéal, il soit souhaitable de pouvoir vérifier un programme avant son exécution, les méthodes de vérification statiques rencontrent des problèmes limitant leur utilisation dans des applications réelles d'envergure et dans certaines situations spécifiques. De plus, ces méthodes sont souvent liées à la phase de conception du système. Lorsque le système évolue ou est mis à jour, les spécifications doivent également évoluer. Ainsi, lorsque le système est en service, ces méthodes de validation sont difficilement applicables à posteriori. Ce qui rend ces méthodes moins adaptées pour la maintenance des systèmes ou dans le cas de systèmes reconfigurables.

En conséquence, les limitations de ces méthodes montrent que l'élimination de certaines fautes et la vérification de certaines propriétés doivent être réalisées de *manière complémentaire* à l'aide de méthodes d'analyse dynamique. Ces méthodes dites incomplètes opèrent sur *une* exécution du système. Au prix de la perte d'exhaustivité, ces méthodes peuvent faire face aux limitations des méthodes décrites précédemment.

L'étude de cette thèse porte sur trois méthodes d'analyse dynamique de programme. Nous nous intéressons à *la vérification à l'exécution, à l'enforcement à l'exécution, et au test de spécifications sur les programmes exprimées comme un ensemble d'exigences formellement définies*. Les spécifications auxquelles nous nous intéressons sont complémentaires des spécifications comportementales utilisées pour la construction du système. En effet, nous avons vu que ces spécifications sont souvent incomplètes ou peuvent parfois être inexistantes. Ce qui est notamment le cas lorsque l'on utilise des approches de type "boite noire" ou "boite grise". Nous nous intéresserons donc aux méthodes qui cherchent à valider d'autres types de spécifications. Les spécifications auxquelles nous nous intéresserons seront exprimées comme des propriétés ciblant des fonctionnalités précises du système. Les facteurs communs des méthodes étudiées sont d'une part l'*absence de spécification comportementale complète* du système à analyser et d'autre part le *caractère dynamique* de ces méthodes.

Méthodes de vérification à l'étude et leur principe

Nous introduisons brièvement le principe des méthodes que nous allons étudier dans cette thèse. Dans le Chapitre 2, nous proposerons un bref rappel des travaux existants pour chacune de ces méthodes.

"Runtime Verification" - Vérification à l'exécution

La vérification à l'exécution [Run09, HG08, BLS09, CM05] est une technique efficace pour s'assurer lors de l'exécution d'un système que celui-ci satisfait un ensemble d'exigences. Par l'expression d'exigences spécifiques, il est possible de prévenir l'apparition de fautes pouvant mener à une défaillance logicielle. Ces exigences peuvent être utilisées pour représenter les comportements désirés du programme ou pour mettre en évidence les comportements fautifs de celui-ci. Les comportements désirés sont souvent exprimés dans une spécification formelle de haut niveau fournie par l'utilisateur (*e.g.*, une formule de logique temporelle, ou un automate). Les comportements fautifs peuvent venir d'erreurs courantes de programmation (*e.g.*, introduction de deadlock en programmation concurrente, variables non initialisées,...). L'analyse réalisée en vérification à l'exécution porte sur *une exécution* du programme. Pour déterminer la relation entre l'exécution produite et l'exigence exprimée, un ensemble d'événements intéressants est observé à l'exécution.

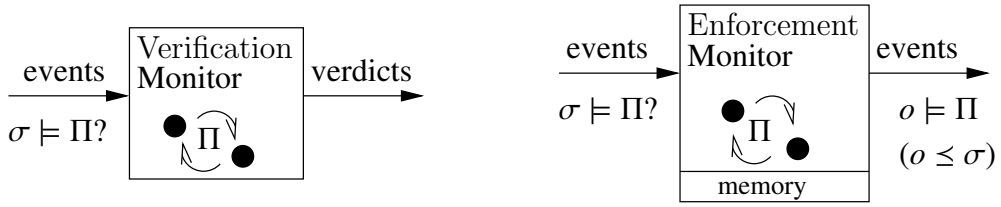


FIGURE 1 – Moniteurs de vérification (gauche) et d'enforcement (droite)

Ces événements peuvent être des instructions particulières (*e.g.*, affectations de variables de sécurité, appels de procédures critiques) ou bien des sorties du programme. De manière générale la vérification à l'exécution requiert de pouvoir observer, lors de l'exécution du programme, l'ensemble des événements ayant une influence sur la satisfaction de l'exigence exprimée.

En pratique, la vérification se fait de la manière suivante (voir Fig. 1) : De l'exécution du système est extraite une représentation abstraite : une trace σ. Cette trace est fournie à une procédure de décision : un moniteur. La trace peut être envoyée au moniteur de manière incrémentale, *i.e.*, événement par événement, ou bien mémorisée entièrement avant utilisation. Un moniteur est un oracle qui détermine par analyse de la trace si la propriété Π considérée est satisfaite sur cette *exécution particulière* du programme ($\sigma \models \Pi$?). Le moniteur produit ainsi un *verdict* déclarant que l'exigence est satisfaite ou non.

La vérification à l'exécution est seulement impliquée dans la détection des violations ou validation de propriétés de correction. Par conséquent, même si une violation a été détectée, l'exécution du programme n'est pas modifiée ou influencée. En revanche, le cadre commun défini par la méthode de vérification à l'exécution est extensible pour pouvoir réagir lorsque des fautes apparaissent. Ce qui est l'objet par exemple des techniques de runtime reflection [Bau08] et runtime enforcement.

"Runtime Enforcement" - Mise en application à l'exécution

La technique de mise en application ou d'enforcement [1] à l'exécution [Sch00, LBW05, Ham06, FFM08] vise à circonvenir les violations de propriétés produites par une séquence d'événements. Cette séquence d'événements peut être générée par un programme ou par son environnement (*e.g.*, un protocole, ou un utilisateur). Cette technique est une extension de la vérification à l'exécution permettant d'aller plus loin que la détection des mauvais comportements d'un programme. L'enforcement de propriétés à l'exécution propose notamment de répondre aux deux questions suivantes :

Que se passe-t-il lorsque la propriété désirée est violée par la séquence d'exécution ?

Est-il possible de prévenir ou corriger le mauvais comportement de cette séquence d'exécution ?

Le principe de l'enforcement à l'exécution est similaire à celui de la vérification. La technique d'enforcement peut donc se voir comme une extension de la technique de vérification à l'exécution où l'on se propose de donner une réponse aux deux questions précédentes soulevées lors de l'apparition de mauvais comportements potentiels.

Le mécanisme utilisé de manière sous-jacente en enforcement de propriétés est un moniteur d'enforcement (voir Fig. 1). Celui-ci possède les caractéristiques décrites précédemment pour les moniteurs de vérification. Notamment, il est toujours dédié à une propriété Π. En revanche, un moniteur d'enforcement possède en plus une *mémoire* utilisée pour mémoriser certains événements qui lui sont fournis en entrée. Le moniteur produit en sortie une séquence d'événements o (au lieu de verdicts). La séquence de sortie qui est produite par le moniteur est un préfixe de la séquence d'entrée. En effet, celui-ci peut "retenir" en mémoire certains événements qui lui ont été fournis en entrée. De plus, la relation entre la séquence d'entrée et la séquence de sortie du moniteur doit respecter deux contraintes qui ont été formulées dans les travaux [Sch00, LBW05] de Schneider, Bauer, Ligatti, et Walker :

soundness (correction) : les séquences d'exécution produites par le moniteur d'enforcement sont correctes vis-à-vis de la propriété considérée ;

transparency (transparence) : les séquences d'exécution originales correctes vis-à-vis de la propriété sont préservées.

1. Dans la suite de cette thèse, nous utiliserons le terme d'"enforcement" pour désigner la mise en application.

Test de propriétés sans spécification comportementale complète

Généralités sur le test. Le test (voir par exemple [Mye04]) est une technique de validation reconnue dont le but est essentiellement de trouver les défauts sur une implémentation, soit durant son développement, soit une fois que la version finale de l'implémentation est produite. Même si beaucoup de travaux ont été réalisés dans ce domaine, améliorer l'efficacité de la phase de test tout en réduisant son coût financier et en terme de temps reste un défi important. Cette demande reste soutenue par une forte demande industrielle.

D'un point de vue pratique, une campagne de test se réalise en deux phases. La première, qui est la *génération de test*, consiste à produire une suite de tests. La seconde, qui est la *phase d'exécution*, consiste à exécuter la suite de tests sur un système cible. Automatiser la phase de génération se fait en produisant la suite de tests depuis une description initiale du *système sous test*. Une suite de test est composée d'un ensemble de *cas de test*. Un cas de test est un ensemble de séquences d'interactions avec le système sous test qui sont exécutées par un testeur externe. Chaque exécution de cas de test doit permettre de produire un verdict, indiquant si le système a réussi ou non sur ce test particulier (ou bien si le résultat du test n'est pas concluant).

La description initiale du système utilisée pour produire la suite de test peut être par exemple le code source du système, certaines hypothèses sur les entrées qu'il peut recevoir (profils utilisateur), ou bien encore certaines exigences sur les propriétés que celui-ci doit vérifier à l'exécution.

Énormément de travaux ont vu le jour autour du test de logiciels. Ces travaux peuvent être par exemple classifiés selon plusieurs critères. Tout d'abord, une première classification [Bei90] peut être établie en fonction de l'objectif des test produits.

- Dans le test de conformité, l'objectif est d'examiner la conformité d'un système par rapport à son modèle vis-à-vis d'exigences fixées.
- Dans le test de robustesse, l'objectif est d'étudier le comportement du système dans un environnement dont les conditions ne sont pas prévues dans sa spécification.
- Dans le test d'interopérabilité, l'objectif est d'évaluer la capacité de plusieurs composants à échanger et utiliser de l'information.
- Dans le test de performance, l'objectif est de mesurer certains paramètres de performance du système.

Une deuxième classification [Bei90] peut être établie en fonction du degré d'observabilité et d'accessibilité du système à tester.

- Dans le test boîte noire, uniquement l'extérieur du système est connu. Et seuls les événements observables échangés entre le système et son environnement sont pris en compte.
- Dans le test boîte blanche, le comportement et la structure interne du système sont connus par le testeur, et peuvent être utilisés par le testeur.
- Dans le test boîte grise, on est à mi-chemin entre test boite noire et boite blanche. La structure modulaire du système peut être connue, mais pas le code de chaque module. On peut également accéder à différentes communications intermédiaires du système.

Le test de conformité boite noire et ses limitations. Usuellement, deux approches pour le test basé sur un modèle formel du comportement sont différenciées. La première est l'approche basée sur les machines à états finis (FSM - Finite State Machine), voir [LY96] pour une vue d'ensemble de cette approche. La seconde est celle basée sur les systèmes de transitions (voir par exemple [BT01]) ; nos travaux se placent dans cette catégorie. Le test basé sur les systèmes de transition a été introduit par les travaux de Rocco de Nicola et Matthew Henessy [NH83]. Brinksma [Bri88] a ensuite utilisé ces travaux pour déterminer des algorithmes pour générer des tests automatiquement. Parmi les nombreuses extensions de ces travaux, Tretmans [Tre90] a proposé une théorie du test de la conformité des entrées et des sorties du programme testé.

Présenter toutes les variations du test, et les différentes approches proposées dans la littérature est largement hors du cadre de cette thèse. Nos travaux coté test sont inspirés du *test de conformité boite noire* [Tre96, BALT90] (black-box conformance testing), qui est une forme de test basée sur les modèles (model-based testing). En effet, dans cette approche, la description du système utilisée peut consister en une caractérisation de ces séquences d'exécution (in)correctes. Pour vérifier la correction vis-à-vis d'exigences comportementales, cette technique propose une théorie complète pour la génération automatique de tests. La génération de tests basée sur les modèles a beaucoup été utilisée dans le domaine

des protocoles de communication [ISO96]. Le succès de cette approche est notament dû à sa capacité à pouvoir faire face au non déterminisme du système testé. Cette technique a été implémentée dans de nombreux outils (*e.g.*, [JJ05, TB03]) ; voir [Har02] ou [BFS04] pour un état de l'art.

Cependant cette technique présente certaines limitations pour son utilisation dans d'autres domaines d'application. Tout d'abord, cette technique repose fortement sur la *disponibilité d'une spécification* du système. Ce qui n'est pas toujours le cas en pratique. De plus, lorsque celle-ci existe, cette spécification doit être assez complète pour assurer la pertinence de la suite de tests produite. Finalement, il est assez probable que cette spécification ne *puisse pas prendre en compte tous les détails* de l'implémentation et reste cantonnée à un niveau d'abstraction. En conséquence, pour pouvoir devenir exécutables, les cas de test produits doivent être *raffinés* en des séquences d'interactions plus concrètes. Automatiser ce processus reste un problème ambitieux [vdBRT05]. De plus, lorsque celui-ci est réalisé à la main, la correction du résultat produit ne peut être garantie complètement.

De la nécessité de bien spécifier

La phase de spécification joue un rôle crucial lors de la vérification et la validation de systèmes informatique. En première approximation, spécifier un système revient à décrire ce que nous attendons de notre système pour son bon fonctionnement. Cette description peut être de différentes sortes : *e.g.*, description du comportement global, description d'un ensemble de propriétés.

Un des points communs entre les méthodes de validation opérant à l'exécution qui ont été décrites précédemment est que la spécification utilisée n'est pas une représentation du comportement global du système. Celle-ci s'exprime comme une liste de propriétés. Ces propriétés peuvent exprimer des comportements recherchés sur le programme ou de mauvais comportements que l'on cherche à éviter. Ces spécifications jouent un rôle d'autant plus prépondérant que les méthodes que nous utiliserons se baseront uniquement sur cette liste de propriétés.

Une spécification comme une liste de propriétés [CMP92a]. Lorsque l'on exprime la spécification du système comme une liste de propriétés, nous bénéficions de deux avantages intéressants. Premièrement, se reposant sur le formalisme de spécification utilisé, il est possible de *s'abstraire de l'implémentation* réelle sous-jacente. En effet, le formalisme permet de décrire ce qui est attendu du système sans faire référence directement à l'état ou aux évènements du programme directement dans son langage d'expression. Plus précisément, les propriétés et le programme peuvent s'exprimer dans des vocabulaires différents. Ainsi le concepteur du système peut spécifier les propriétés sans connaître les détails exacts de l'implémentation, *e.g.*, paradigme utilisé, langage de programmation, *etc.* L'abstraction des détails d'implémentation facilite le raisonnement du concepteur. Il est alors plus facile de détecter les erreurs et incohérences de conception. Un deuxième bénéfice est la *modularité* de la spécification. En effet, chaque propriété peut être étudiée et validée sur le système de façon séparée. Ainsi, il est possible lors de la phase de conception d'ajouter ou supprimer des exigences sur le système.

Cependant, un des risques sous-jacents à exprimer la spécification de notre système comme un ensemble de propriétés est la *sous-spécification*. Comment être sûr que nous avons exprimé assez de propriétés pour garantir un comportement final correct et attendu ? Un exemple bien connu de sous-spécification consiste à exprimer une propriété de correction (partielle) pour un algorithme sans exprimer une propriété de terminaison.

Une solution partielle possible pour éviter le problème de sous-spécification est d'utiliser les distinctions qu'il est possible d'établir entre les propriétés dans le formalisme. S'il est possible d'établir des "classes" de propriétés, celles-ci exprimeraient une certaine complémentarité dans l'espace de l'ensemble des propriétés expressibles. Alors, une bonne pratique est d'exprimer "assez" de propriétés dans chacune des classes.

Il existe plusieurs classifications de propriétés (voir Chapitre 3). La plus connue, et la plus utilisée, est la classification Safety-Liveness [Lam77, AS85]. Deux classes de propriétés y figurent. Les propriétés de safety (sûreté) expriment informellement que "rien de mauvais ne doit arriver". Les propriétés de liveness (vivacité) expriment que "quelque chose de bon arrivera". Une autre classification est la hiérarchie Safety-Progress [CMP92a]. Dans cette classification, le nombre de classes est plus important ce qui fournit une classification plus fine des propriétés. La nécessité de bien spécifier nous amène ainsi à utiliser cette dernière comme cadre de spécification.

À propos du formalisme de spécification. Un formalisme de spécification, pour être utile, se doit d'être compréhensible et reconnu parmi les utilisateurs. De plus ce formalisme doit être assez expressif pour permettre d'exprimer l'ensemble des exigences exprimées lors de la phase de conception. L'expressivité du formalisme doit être choisie de manière adéquate pour que les procédures de décision entourant les propriétés restent décidables et efficaces. Les logiques temporelles sont des formalismes utilisés couramment dans le domaine de la vérification. Il est usuel de distinguer les logiques arborescentes des logiques linéaires. Comme les techniques de notre étude opèrent sur une exécution du système sous-jacent, les logiques temporelles linéaires nous semblent plus appropriées. La logique temporelle linéaire [Pnu77, PM91] (LTL, Linear Temporal Logic) introduite par Pnueli dans le monde de la vérification de logiciels est un formalisme bien accepté, et permet d'exprimer de manière assez concise les spécifications. Son utilisation en model-checking ces dernières années a permis d'obtenir des procédures de décisions toujours plus efficaces. Le formalisme des expressions régulières [Kle51] et ω-régulières [Sta97] a été introduit par Kleene en informatique. C'est un formalisme légèrement plus expressif que LTL qui a l'avantage d'être bien compris des ingénieurs.

Raisonner sur les traces finies. La popularité de LTL dans le domaine du model-checking a motivé son utilisation dans les méthodes d'analyse dynamique. Cependant, la sémantique de LTL est définie de manière rigoureuse et précise sur les traces infinies et non pas sur des préfixes de séquences infinies. Le passage à une sémantique sur les traces finies (*e.g.*, dans [JJ89]) a donné lieu à plusieurs interprétations personnalisées. La plupart de ces approches se basent sur la sémantique *faible* et *forte* de LTL [EFH+03]. Dans la vue forte, une formule est satisfaite sur une trace finie seulement s'il est possible d'évaluer la formule comme vraie sur cette trace finie. Dans la vue faible, une formule est satisfaite si rien n'a montré que la formule n'est pas satisfaite. Il existe également une vue *neutre* de LTL assez similaire à la vue classique. Nous renvoyons au survey de Eisner et ses coauteurs [EFH+03] pour plus de détails sur les avantages et inconvénients de chacune des vues.

Une des spécificités des méthodes de vérification par analyse dynamique d'un programme réside dans le fait que des procédures de décision spécifiques sont nécessaires. Celles-ci doivent permettre de déterminer si la spécification est satisfaite sur une trace finie du programme. Par exemple, dans le domaine de la vérification à l'exécution, l'exécution du programme peut être abstraite par une séquence d'événements lue par le moniteur de manière incrémentale. Ainsi, un des rôles du moniteur est de décider à chaque lecture d'événement si la spécification est satisfaite ou non. Par contre, en model checking, les séquences d'exécution infinies du système sont analysées. Dans ce cas, une interprétation des séquences finies n'est pas nécessaire.

Contributions de cette thèse

Nous présentons de manière succincte les différentes contributions amenées par cette thèse.

- Nous proposons une vision de la classification *Safety-Progress* des propriétés comme cadre de spécification pour les méthodes de validation à l'exécution. Ce qui nécessite d'étendre uniformément les résultats existants sur les séquences infinies aux séquences finies.
- Nous proposons une vue uniforme de l'applicabilité des trois méthodes de validation lors de l'exécution des systèmes décrites précédemment (vérification, enforcement, et test de propriétés). Pour chaque technique de validation, nous précisons l'espace des propriétés pour lequel la technique peut être mise en œuvre.
- Nous proposons une notion de moniteur de vérification paramétré par un domaine de vérité. Dans leur forme la plus fine, ces moniteurs utilisent un domaine de vérité à quatre valeurs permettant de distinguer plus finement les situations de satisfaction d'une propriété par l'exécution d'un programme.
- Nous introduisons une notion de moniteur d'enforcement générique basé sur un nombre fini d'états de contrôle et une mémoire finie mais non bornée. Nous étudions le problème de la composition de ces moniteurs d'enforcement. Aussi, nous montrons une technique de synthèse simple pour obtenir ces moniteurs d'enforcement à partir d'automates de Streett.
- Nous proposons une boîte à outils dédiée à la vérification et à l'enforcement de propriétés sur les programmes Java. Cette boîte à outils prend en charge la synthèse de moniteurs ainsi que leur intégration dans un programme Java.

– Nous proposons une chaîne d'outils dédiée au test de propriétés sur les logiciels Java. Cette chaîne d'outils prend en compte un large spectre des phases du test : de la conception des cas de test jusqu'à leur exécution.

Visite guidée de cette thèse

Nous présentons ici un bref résumé de chacun des chapitres de cette thèse.

Chapitre 1 : Notations. Ce chapitre introduit les notations de base qui seront utilisées dans cette thèse. Comme nous nous intéressons au mécanisme de validation de propriétés opérant à l'exécution des systèmes, nous rappelons les notions de séquences d'exécution de programme. Nous rappelons également les notions d'expression régulières et ω-régulières que nous utiliserons pour décrire les séquences d'exécution de manière concise. Puis nous introduisons la notion de propriété considérée dans cette thèse.

Chapitre 2 : État de l'art. Ce chapitre propose un aperçu des travaux existants autour des méthodes de validation étudiées dans cette thèse. Nous présentons un bref historique des travaux menés dans ces disciplines. Également, pour chaque technique nous rappelons les travaux reliés à ceux proposés dans cette thèse.

Première partie : Cadre général

La première partie de cette thèse est consacrée à la définition d'un cadre général pour l'étude des méthodes de validation à l'exécution. Nous revisitons la classification *Safety-Progress* qui étudie l'espace des propriétés ω-régulières, leur spécification, et leur représentation. Ensuite, nous y étudions les classes de propriétés pour lesquelles il est possible d'appliquer les méthodes de validation.

Chapitre 3 : Spécification de propriétés dans la classification *Safety-Progress* pour la validation à l'exécution. Dans ce chapitre nous présentons un cadre général de spécification de propriétés. Nous nous basons pour cela sur la classification *Safety-Progress* des propriétés introduite par Chang, Manna, et Pnueli [CMP92b]. Nous rappelons les résultats connus, et y intégrons les nôtres. L'extension que nous proposons est motivée par son utilisation comme cadre de spécification pour la validation à l'exécution. Nous montrons dans ce chapitre comment il est possible d'étendre cette classification pour prendre en compte les séquences finies et infinies de manière cohérente. De plus, nous croyons que cette classification générale est une bonne base pour spécifier les propriétés d'un programme.

Chapitre 4 : Applicabilité des méthodes de validation à l'exécution. Ce chapitre est consacré à l'étude de l'applicabilité des méthodes de vérification, d'enforcement, et de test de propriétés à l'exécution. Dans la classification introduite au chapitre précédent nous délimitons les classes de propriétés qu'il est possible de traiter avec les méthodes mentionnées. De façon indirecte, ce chapitre propose donc une comparaison entre ces différentes techniques en terme de validation de propriétés. Ainsi, étant donnée une propriété exprimée dans le cadre de la classification *Safety-Progress*, il est possible de savoir quelles méthodes de validation à l'exécution peuvent être mises en œuvre.

Deuxième partie : Approches pour la validation à l'exécution

La deuxième partie de cette thèse présente trois approches pour la validation de propriétés lors de l'exécution des systèmes. Nous nous basons sur le cadre défini dans la première partie.

Chapitre 5 : Préliminaires aux approches proposées. Ce chapitre introduit quelques concepts sur les automates de Streett. Forts de ces concepts, nous revenons également brièvement sur les notions de monitorabilité et de testabilité. Nous serons capables de les caractériser sur un automate de Streett.

Chapitre 6 : Approches vérification et enforcement de propriétés à l'exécution. Dans ce chapitre nous proposons une méthode de vérification et d'enforcement de propriétés. Nous proposons des notions de moniteurs de vérification et d'enforcement. Nous étudions le problème de leur composition par des opérations booléennes. Puis nous montrons comment il est possible de les synthétiser facilement depuis les automates de Streett.

Chapitre 7 : Approche test de propriétés sans spécification comportementale complète. Ce chapitre propose une approche dédiée au test de propriétés sans spécification comportementale complète. Nous proposons une méthode pour générer un ensemble de tests communicants permettant d'évaluer si une certaine relation est vérifiée entre les séquences d'exécution produites par un programme et l'ensemble des séquences décrites par une propriété.

Troisième partie : Mise en œuvre sur les programmes Java

Chapitre 8 : La boite à outils j-VETO. Ce chapitre présente la boite à outils prototype j-VETO (Java Verification and Enforcement TOolbox). Nous présentons les différents outils la composant ainsi que leur utilisation.

Chapitre 9 : La chaîne d'outils j-POST. Ce chapitre présente la chaîne d'outils j-POST. Cet outil approfondit les résultats présentés dans le Chapitre 7.

Quatrième partie : Conclusion et perspectives

Conclusion et perspectives. Ce chapitre clôt le manuscrit en récapitulant les éléments essentiels de cette thèse et décrivant les perspectives de recherche ouvertes.

Annexes

Annexe A : Preuves. Afin d'alléger la lecture de certaines parties de cette thèse, certaines preuves sont présentées de manière succincte. Cette annexe regroupe les versions complètes de ces preuves.

Annexe B : Utilisation des différents outils. Ce chapitre regroupe les modes d'emploi des différents outils présents dans j-VETO et j-POST.

Annexe C : Travaux connexes. Ce chapitre regroupe deux articles [FFMR06, FFMR07a] correspondant à des travaux précédents coté test. Nous proposons un résumé de ces travaux à la fin du Chapitre 7.

CHAPITRE 1

Notations

Sommaire

1.1 À propos des séquences, et séquences d'exécution 11
1.2 Expressions régulières et omega-régulières 12
1.3 Propriétés, propriétés finitaires, propriétés infinitaires 13

Ce chapitre précise quelques notations et préliminaires, plus particulièrement les notions de *séquences d'exécution d'un programme*, d'*expressions ω-régulières* et de *propriétés de programme*.

1.1 À propos des séquences, et séquences d'exécution

A partir d'un ensemble fini d'éléments E nous définissons les notations relatives aux éléments appartenant à E. Une séquence σ contenant des éléments de E est formellement définie par une fonction totale $\sigma : I \to E$ où I est soit l'intervalle d'entiers $[0, n]$ pour un $n \in \mathbb{N}$ dans le cas des séquences finies, soit $\mathbb{N}$ lui même (l'ensemble des entiers naturels) dans le cas des séquences infinies. Nous notons E^* l'ensemble des séquences finies sur E, par E^+ l'ensemble des séquences finies non vides sur E, et par E^ω l'ensemble des séquences infinies sur E. L'ensemble $E^\infty = E^* \cup E^\omega$ est l'ensemble de toutes les séquences définies sur E. La séquence vide de E est dénotée par ϵ_E ou ϵ lorsque le contexte est clair. La longueur (nombre d'éléments) d'une séquence finie σ est notée $|\sigma|$ et le $(i+1)$-ème élément de σ est noté σ_i. La longueur d'une séquence infinie est notée ∞, avec $\forall i, i < \infty$. Pour deux séquences $\sigma \in E^*, \sigma' \in E^\infty$, nous utilisons $\sigma \cdot \sigma'$ pour noter la concaténation de σ et σ', et $\sigma \prec \sigma'$ pour noter le fait que σ est un préfixe strict de σ'. Nous disons aussi que σ' est une continuation stricte de σ. La séquence σ est un préfixe strict de $\sigma' \in \Sigma^\infty$ lorsque $\forall i \in \{0, \ldots, |\sigma| - 1\} \cdot \sigma_i = \sigma'_i$ et $|\sigma| < |\sigma'|$. Lorsque $\sigma' \in E^*$, nous notons $\sigma \preceq \sigma' \stackrel{def}{=} \sigma \prec \sigma' \vee \sigma = \sigma'$. L'ensemble $pref(\sigma)$ des préfixes d'une séquence $\sigma \in \Sigma^\infty$ est défini comme suit. Si $\sigma \in \Sigma^*$, alors $pref(\sigma) = \{\sigma' \in \Sigma^* \mid \sigma' \preceq \sigma\}$. Si $\sigma \in \Sigma^\omega$, alors $pref(\sigma) = \{\sigma' \in \Sigma^* \mid \sigma' \prec \sigma\}$. L'ensemble $Pref(X)$ des préfixes d'un ensemble de séquences X est l'union de l'ensemble des préfixes des séquences de X : $Pref(X) = \bigcup_{\sigma \in X} pref(\sigma)$. Les différents ensembles des continuations possibles d'une séquence $\sigma \in \Sigma^*$ sont définis comme suit :

- $cont^*(\sigma) = \{\sigma' \in \Sigma^* \mid \exists \sigma'' \in \Sigma^+, \sigma' = \sigma \cdot \sigma''\}$ est l'ensemble des continuations finies de σ ;
- $cont^\omega(\sigma) = \{\sigma' \in \Sigma^\omega \mid \exists \sigma'' \in \Sigma^\omega, \sigma' = \sigma \cdot \sigma''\}$ est l'ensemble des continuations infinies de σ ;
- $cont^\infty(\sigma) = cont^*(\sigma) \cup cont^\omega(\sigma)$ est l'ensemble des continuations finies et infinies de σ.

Pour $\sigma \in E^\infty$ et $n \in \mathbb{N}$, $\sigma_{\ldots n}$ est la sous-séquence contenant les $n+1$ premiers éléments de σ. Aussi, lorsque $|\sigma| > n$, la sous-séquence $\sigma_{n\ldots}$ est la séquence contenant tous les éléments de σ sans les n premiers. Pour $i, j \in \mathbb{N}$ avec $i \leq j$, nous notons $\sigma_{i\ldots j}$ la sous-séquence de σ contenant du $(i+1)$-ème au $(j+1)$-ème éléments (inclus).

Les programmes comme générateurs de séquences d'exécution. Un programme $\mathcal{P}$ est considéré comme un générateur de séquences d'exécution. Nous sommes intéressé par un ensemble restreint des opérations qu'un programme peut produire. Ces opérations influencent la valeur de vérité des propriétés que le programme est supposé satisfaire. De telles séquences d'exécution peuvent être faites des événements d'accès sur un système sécurisé à ses ressources, ou des opérations du noyau d'un système d'exploitation. Dans un contexte logiciel, ces événements peuvent être des abstractions des instructions importantes telles les modifications de variables ou les appels de procédures. Nous représentons ces opérations par un ensemble fini d'*événements*, à savoir un vocabulaire Σ. Nous notons $\mathcal{P}_\Sigma$ un programme pour lequel le vocabulaire est Σ. L'ensemble des séquences d'exécution de $\mathcal{P}_\Sigma$ est noté $Exec(\mathcal{P}_\Sigma) \subseteq \Sigma^\infty$. Cet ensemble est *préfixe-clos*, c'est-à-dire $\forall \sigma \in Exec(\mathcal{P}_\Sigma), \forall \sigma' \in \Sigma^* : \sigma' \preceq \sigma \Rightarrow \sigma' \in Exec(\mathcal{P}_\Sigma)$, ou autrement dit $\forall \sigma \in Exec(\mathcal{P}_\Sigma) : pref(\sigma) \subseteq Exec(\mathcal{P}_\Sigma)$. Dans la suite de cette thèse nous considérerons un vocabulaire Σ.

1.2 Expressions régulières et ω-régulières

Nous rappelons les notions basiques des expressions régulières [Kle51] et ω-régulières [KHKT03, Sta97]. Nous les utiliserons dans la suite de cette thèse pour la description de propriétés. Nous rappelons la syntaxe puis la sémantique de ces expressions.

Syntaxe

Considérant un ensemble non vide X, les expressions ω-régulières sur X sont des descriptions concises des sous-ensembles de X^ω. L'ensemble des expressions ω-régulières sur X est formellement défini comme le plus petit ensemble de séquences finies et infinies construit à partir des éléments de X, des parenthèses "(", ")", des symboles $+$, *, $^\omega$, et fermé par un nombre d'applications fini des règles syntaxiques suivantes :

- Chaque élément de X est une expression régulière.
- Si α et β sont des expressions régulières, alors $\alpha + \beta$, $\alpha \cdot \beta$, et α^* le sont également.
- Si α est une expression régulière et β une expression ω-régulière, alors α^ω et $(\alpha \cdot \beta)$ sont des expressions ω-régulières.
- Si α et β sont des expressions ω-régulières, alors $\alpha + \beta$ en est une également.

Sémantique

Comme mentionné ci-dessus, une expression régulière sur X dénote un sous-ensemble de X^ω, *i.e.*, un ensemble de mots. Considérant une expression ω-régulière α nous notons par $\mathcal{L}(\alpha)$ l'ensemble des mots définis par α. Formellement, $\mathcal{L}$ est défini pour les expressions régulières et ω-régulières comme suit :

- $\mathcal{L}(x) \stackrel{def}{=} \{s : \{0\} \to X \mid s_0 = x\}$ pour tout $x \in X$, *i.e.*, l'ensemble singleton contenant la séquence unique de longueur 1 avec α comme premier élément.
- $\mathcal{L}(\alpha + \beta) \stackrel{def}{=} \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$ où α et β sont soit toutes les deux des expressions régulières ou ω-régulières.
- $\mathcal{L}(\alpha \cdot \beta) \stackrel{def}{=} \{s \cdot t \mid s \in \mathcal{L}(\alpha), t \in \mathcal{L}(\beta)\}$ pour toute expression régulière α et toute expression régulière ou ω-régulière β.
- $\mathcal{L}(\alpha^*) \stackrel{def}{=} \{\epsilon_X\} \cup \bigcup_{i \in \mathbb{N} \setminus \{0\}} \{s_1 \cdot s_2 \cdots s_i \mid \forall j \in [1, i] : s_j \in \mathcal{L}(\alpha)\}$ pour toute expression régulière α ; c'est l'ensemble des séquences obtenues par concaténation finie de séquences dans $\mathcal{L}(\alpha)$.
- $\mathcal{L}(\alpha^\omega) \stackrel{def}{=} \{s_1 \cdot s_2 \cdots s_i \cdots \mid \forall i \in \mathbb{N} : s_i \in \mathcal{L}(\alpha) \setminus \{\epsilon_X\}\}$ pour toute expression régulière α non vide ; c'est l'ensemble de séquences infinies obtenues par concaténation infinie de séquences non vides de $\mathcal{L}(\alpha)$.

Dans la suite, pour simplifier les notations, nous discuterons indistinctement des expressions ω-régulières et de l'ensemble qu'elles décrivent. De plus, nous omettrons les parenthèses lorsque le contexte sera clair et en fixant la précédence de $+, \cdot, ^*, ^\omega$ *t.q.* * et $^\omega$ ont la priorité sur $\cdot$ qui a la priorité sur $+$. Aussi, lorsque l'on fera référence aux expressions ω-régulières d'un ensemble X, nous pourrons noter ϵ pour ϵ_X lorsque le contexte sera clair. Également, dans la suite, nous parlerons indistinctement de mot ou de séquence.

1.3 Propriétés, propriétés finitaires, propriétés infinitaires

Les propriétés que nous considérerons dans la suite sont des ensembles de séquences d'exécution. Cet ensemble représente l'intégralité des "bonnes séquences", *i.e.*, les comportements que l'on autorise sur le programme.

Définition 2 (Propriétés et ensembles de séquences d'exécution). Nous définissons les différents types de propriétés comme suit :
- Une *propriété finitaire* est un sous-ensemble de Σ^*.
- Une *propriété infinitaire* est un sous-ensemble de Σ^ω.
- Une *propriété* est un sous-ensemble de Σ^∞.

Étant donnée une séquence d'exécution finie (resp. infinie, finie ou infinie) σ et une propriété finitaire (resp. propriété infinitaire, propriété) ϕ (resp. φ, θ), lorsque $\sigma \in \phi$, ce qui est noté $\phi(\sigma)$ (resp. $\sigma \in \varphi$, noté $\varphi(\sigma)$, $\sigma \in \theta$, noté $\theta(\sigma)$), nous disons que σ *satisfait* ϕ (resp. φ, θ). Une conséquence de cette définition est que les propriétés que nous considérerons seront restreintes à des ensembles de séquences d'exécution simples [1], en excluant les propriétés spécifiques définies sur des ensembles d'ensembles (powersets) de séquences d'exécution, comme par exemple l'équité (fairness).

1. C'est la distinction faite par Schneider [Sch00], entre les propriétés et les politiques (générales). L'ensemble des propriétés (définies sur les séquences d'exécution simples) est un sous-ensemble de l'ensemble des politiques (une politique est définie comme un ensemble d'ensemble de séquences d'exécution).

CHAPITRE 2

État de l'art

Sommaire

2.1 Vérification de propriétés à l'exécution 17
2.1.1 Notion de moniteur 18
2.1.2 Les autres saveurs de la vérification à l'exécution 21
2.1.3 Caractérisations des propriétés vérifiables à l'exécution 21
2.1.4 Perspectives et futurs challenges 22
2.2 Enforcement de propriétés à l'exécution 23
2.2.1 Bref historique des approches d'enforcement par moniteur 23
2.2.2 Pouvoir calculatoire des mécanismes d'enforcement 23
2.2.3 Caractérisations de l'ensemble des propriétés enforçables avec des moniteurs d'exécution 24
2.2.4 Synthèse de mécanismes d'enforcement à l'exécution 25
2.2.5 Quelques approches d'enforcement 25
2.3 Test de propriétés 25
2.3.1 Caractérisation des propriétés testables 26
2.3.2 Approches pour le test de propriétés 26
2.3.3 Test de propriétés sans spécification comportementale 27

Chapter abstract

An overview of three property-validation methods operating at runtime of systems is proposed. We start first by presenting the runtime verification technique, and then one of its extensions : the runtime enforcement technique. Then, we focus on methods for software testing from properties. We present some notions for these techniques. Moreover, we recall existing characterizations of the property spaces for which those techniques apply.

Résumé du chapitre

Une vue d'ensemble de trois méthodes opérant à l'exécution pour la validation de propriétés sur les systèmes est proposée. Nous commençons par donner une présentation de la technique de vérification à l'exécution puis d'une de ses extensions : la technique d'enforcement à l'exécution. Ensuite, nous nous intéressons aux méthodes de test de logiciels à partir de propriétés. Nous présentons les principes de base de ces approches. De plus, nous revenons sur les caractérisations des espaces de propriétés pour lesquels ces techniques peuvent être appliquées.

2.1 Vérification de propriétés à l'exécution

Origines. L'idée d'utiliser un observateur comme moniteur ou surveillant de programme remonte au moins aux début des années 1960 avec l'apparition des premiers débogueurs. Ensuite, le manque de fiabilité dans les programmes et l'émergence de nouvelles applications distribuées a donné lieu à d'importants efforts de recherche pour donner la possibilité aux utilisateurs de pouvoir surveiller des fonctionnalités spécifiques des systèmes. Différentes approches liées à des domaines spécifiques ont émergé (runtime checking, software fault analysis,...). Cependant le point commun des moniteurs utilisés dans ces approches est que l'on pouvait les caractériser comme des observateurs externes du programme collectant dynamiquement des événements, les analysant, et effectuant des actions en conséquence. Un historique des utilisations du monitoring et des outils dédiés peut être trouvé dans [Sch95, GVS94, PN81, DGR04].

Les idées sous-jacentes à la vérification à l'exécution ne donc sont pas nouvelles. Certains problèmes auxquels cette technique s'intéresse ont fait l'objet de travaux antérieurs. Par exemple dans le domaine des *systèmes synchrones* [HCRP91, CPHP87], des observateurs [HLR94, HFB93] sont développés en parallèle des systèmes pour vérifier leur comportement. Dans le domaine du test [DRP99] également, des observateurs sont utilisés pour vérifier en parallèle une propriété sur les exécutions des tests, comme par exemple dans [RMJ04].

Il est clair que la vérification à l'exécution a également ses origines dans le model-checking. Dans ce domaine, plusieurs travaux se sont intéressés dès le milieu des années 90 à la vérification de traces finies sur les systèmes distribués [JJJR94, FJJR94, HBUP03]. L'idée était de vérifier la correction des calculs de processus distribués par rapport à une propriété globale exprimant par exemple un ordre entre les événements observables des différents calculs distribués.

Une discipline à part entière. Le terme de "vérification à l'exécution" et son émergence comme une discipline scientifique à part entière a son origine dans les travaux de Havelund et Rosu [Hav00, HR01, HR02] et de Lee, Kim, Sokolsky, Viswanathan [LKK+99].

Nous identifions plusieurs raisons qui ont conduit à ce que la vérification à l'exécution soit une discipline à part entière.

- Tout d'abord, la technique de vérification à l'exécution se veut être une technique facile à mettre en œuvre sur les programmes : de manière automatique et indépendamment de la taille des programmes [1]. Pour être une technique reconnue et utilisée largement, l'impact sur le programme du moniteur doit se faire ressentir de manière minimale. Ainsi, il est de première importance qu'un système logiciel, augmenté du code nécessaire à sa vérification lors de son exécution, ait des performances sensiblement identiques. Aussi, l'instrumentation du programme afin de pouvoir en observer ses événements importants doit pouvoir être faite de manière aisée, automatique et sans erreur. Ces problématiques sont spécifiques au fait de vérifier le système lors de son exécution.
- Également, appliquer des algorithmes de vérification existants en model checking ne pouvait se faire de manière directe. En effet, comme vu dans l'introduction, pour la logique linéaire temporelle, les algorithmes existants ne prenaient en compte que les traces infinies. Ainsi, vérifier des propriétés de logique temporelle lors de l'exécution des systèmes a nécessité une adaptation des méthodes existantes. Une première catégorie d'approches a consisté à effectuer une réécriture dynamique des formules vérifiées afin de séparer le passé des obligations futures à satisfaire. Pour cela, ces méthodes reposent sur les caractérisations en terme de point fixe de la logique considérée. Par exemple dans [GH01], Havelund et Giannakopoulou modifient la construction d'automates de Büchi pour en faire des observateurs de formules LTL.

Ce qui a distingué la vérification à l'exécution d'autres techniques de vérification était la nécessité d'adapter ces dernières. En effet, nous avons vu que des problèmes spécifiques sont mis en jeu lors de la vérification à l'exécution. Aujourd'hui, la vérification à l'exécution est un domaine de recherche à part entière représentée principalement par le workshop Runtime Verification [2]. Une définition générale de la vérification à l'exécution est donnée dans [LS08] :

Définition 3 (Runtime Verification - Vérification à l'exécution ([LS08])). La vérification à l'exécution est la discipline de l'informatique qui s'intéresse à l'étude, au développement, et à l'application

1. Par opposition au model-checking qui devient impraticable sur des programmes de taille arbitraire.
2. La première édition de ce workshop a été initiée en 2001 à Paris, France. http://www.runtime-verification.org/.

des techniques de vérification qui permettent d'examiner si *une exécution* d'un système examiné satisfait ou viole une propriété donnée de correction.

En pratique, comme décrit dans l'introduction, c'est le travail d'un moniteur d'examiner l'exécution du système et de déterminer la satisfaction de celle-ci par rapport à la propriété. Le moniteur joue donc un rôle central en vérification à l'exécution.

2.1.1 Notion de moniteur

Une définition informelle et générale de moniteur pourrait être la suivante :

DÉFINITION 4 (MONITEUR). Un moniteur est un dispositif déterminant un verdict par lecture d'une trace finie. Le verdict du moniteur est une évaluation par rapport au respect d'une exigence donnée de la trace lue.

Ainsi, il est possible de voir un moniteur comme une procédure de décision pour le problème suivant :

"est-ce que la trace produite par le programme appartient à l'ensemble des traces déclarées satisfiant l'exigence ?"

Mathématiquement ce problème est celui de la reconnaissance du mot, celui-ci est relativement simple par rapport aux problèmes sous-jacents à d'autres techniques de vérification. Par exemple, en model-checking, le problème est celui de l'inclusion de langage, plus complexe. Ainsi en vérification à l'exécution, on s'intéressera uniquement à des propriétés pour lesquelles il est possible de déterminer un verdict sur une trace d'exécution.

La définition générale de moniteur que nous avons donnée se concrétise en pratique de différentes manières. Nous décrivons brièvement certaines caractéristiques des moniteurs.

Instrumentation du programme à analyser et localisations : Les différents emplacements dans le système où le moniteur intervient sont appelés les *localisations*. Déterminer ces localisations repose sur une analyse du programme. Celle-ci peut être manuelle ou automatique. L'instrumentation manuelle du programme, comme son nom l'indique, consiste en l'ajout manuel du code du moniteur aux points intéressants. Ceci est évidemment fastidieux et source d'erreurs. Lorsque les localisations du programme sont déterminées automatiquement, la technique d'analyse peut être statique, dynamique, ou une combinaison des deux. L'analyse est dite dynamique lorsqu'elle repose sur certaines données du programme évaluables uniquement lors de l'exécution.

Étant donné que les méthodes de vérification à l'exécution ou d'enforcement reposent sur l'instrumentation du système sous-jacent, il paraît naturel que différentes approches pour l'instrumentation du système aient été envisagées. L'instrumentation peut être réalisée au niveau du code source du programme vérifié ou bien au niveau du code objet. Le but commun de toutes ces approches est de réaliser une instrumentation simple, efficace, de telle manière que les performances du programme initial ne soit pas dégradées.

- Quelques premières approches en vérification à l'exécution utilisaient leur propre technique ad-hoc pour instrumenter le programme. Par exemple dans JavaPathExplorer (JPaX) [HR01], le bytecode est instrumenté par des scripts s'aidant d'une bibliothèque de manipulation de bytecode.
- AOP (Aspect-Oriented Programming) [KLM+97]. Un *aspect* regroupe un ensemble de joinpoints, pointcuts, et codes advices. Un *joinpoint* est un point dans le programme où peuvent agir différents aspects. Des exemples de joinpoints sont les appels de méthodes ou de constructeurs, les modifications de variables de classes... Les *pointcuts* sont des éléments liés au flot d'exécution du programme autour desquels on peut greffer l'action des aspects. De manière abstraite, un pointcut définit un ensemble de joinpoints, une "coupe". Le *code advice* définit le code greffé par l'aspect dans l'application d'origine. Dans le cas de code advice de type *before* (resp. *after*, *around*), celui-ci sera intégré avant (resp. après, autour) le pointcut de l'aspect. La programmation par aspect bénéficie ces dernières années d'une bonne popularité dans le domaine de la vérification à l'exécution. C'est

notamment la technique utilisée dans les outils MOP [3] [CR07] et TraceMatches [ABH+06] pour la vérification de propriétés fournies par un utilisateur. Également, la technologie de programmation par aspect a été utilisée pour la détection de data races récemment dans l'outil RacerAJ [BH08]. L'efficacité de la programmation par aspect semble rendre obsolètes les techniques ad hoc précédemment utilisées.

- BCEL (Byte Code Engineering Library) [The09]. La bibliothèque BCEL permet d'analyser, de créer, et manipuler des classes Java contenant du binaire. Les classes sont représentées par des objets représentant toute l'information symbolique de cette classe : méthodes, champs, et instructions de bytecode. Une application possible de BCEL est de créer des objets représentant des classes lues depuis un programme existant, de les transformer, et générer un nouveau programme à partir de ces objets modifiés.
  La technologie BCEL est celle qui est utilisée dans l'outil Polymer [JLD07] pour l'enforcement de propriétés à l'exécution (voir Section 2.2).
- Valgrind [NS07] est une suite d'outils pour détecter les erreurs de gestion de mémoire dans les programmes C. Valgrind permet également d'écrire des outils d'instrumentation pour du code objet.

Emplacement du moniteur : La procédure de décision implantant le moniteur peut être située à différents emplacements par rapport au programme vérifié [DGR04]. Le cas le plus commun est celui où le moniteur s'exécute dans le même espace mémoire que le programme. On parle alors de placement *inline*. Dans ce cas, le code du moniteur peut être inséré dans le code de l'application aux points d'observation ou bien par appel à une routine. Dans le deuxième cas, le moniteur est placé dans un autre espace d'adressage. On parle de mode *outline*. Cela inclut les cas où le moniteur est placé dans un thread ou un processus différent.

Il est également possible de distinguer les cas où le moniteur analyse le programme événement par événement ou bien *aposteriori*. Dans le cas où les événements sont fournis au moniteur de façon incrémentale, un verdict sur la satisfaction de la propriété doit être décidé à chaque pas. Dans ce cas, on parle d'analyse *online* ; ce qui est le cas le plus commun. Si l'analyse du moniteur se fait une fois l'exécution du programme terminée, l'analyse est dite *offline* ; comme par exemple dans RuleR [BRH07, BHRG07]. L'analyse est alors souvent réalisée sur une trace qui est produite pendant l'exécution du programme et consultée par le moniteur une fois celle-ci entièrement produite. Ce dernier cas est alors similaire à ce qu'on appelle parfois le *passive testing* (voir par exemple [HBUP03] ou [MWC08]). L'une des différences du mode offline, qui est aussi un avantage, est que dans ce cas le moniteur peut aller d'avant en arrière sur la trace. Il s'en suit que les algorithmes d'analyse sont souvent plus simples.

Plateforme pour le moniteur : Les moniteurs peuvent être réalisés sur différents types de plateformes d'implantations : logicielles ou matérielles. Dans le cas d'une implantation logicielle, du code est utilisé ; qui peut être dans le langage d'origine du programme analysé ou non. Un moniteur matériel peut être réalisé par un composant ou processeur dédié qui observe le système à travers ses bus pour collecter les données et événements importants.

Différentes tendances dans la vérification à l'exécution

Il est usuel de distinguer deux types d'approches dans la vérification à l'exécution. La première se focalise sur la détection d'erreurs en programmation concurrente. La deuxième est la vérification de propriétés fournies par un utilisateur.

Détection d'erreurs courantes en programmation concurrente : Pour les systèmes multi-thread, les techniques de déboguage classiques sont difficiles à mettre en œuvre. Le programme considéré peut exhiber un nombre de comportements différents assez important, et retrouver une causalité entre les différents événements du système est une tâche difficile. Des méthodes de vérification à l'exécution se sont intéressées à détecter les erreurs communes de concurrence. Ces techniques ont pour objectif d'extraire le maximum d'information possible lors d'une exécution du système pour déterminer si de telles erreurs peuvent arriver potentiellement lors d'autres exécutions ; et ce même si sur l'exécution courante aucune erreur n'a été exhibée. Les erreurs de concurrence fréquemment retrouvées dans les programmes sont les

3. MOP peut se voir d'une certaine manière comme un générateur d'aspects à partir de spécifications.

blocages de processus (*deadlocks*), la famine de processus et les data races. Nous présentons brièvement certaines de ces approches.

Détection de data races. L'outil jPredictor [CŞR08] extrait une relation de causalité de l'exécution du programme. Cette relation est ensuite découpée par une analyse statique et raffinée en utilisant l'information sur l'atomicité des verrouillages.

Dans [BH08, BH09], Bodden et Havelund proposent d'étendre le langage AspectJ avec de nouvelles constructions dédiées à la détection des erreurs de concurrence. Ils introduisent une version modifiée de l'implantation de l'algorithme Eraser pour détecter les race conditions dans les programmes Java.

Artho et ses coauteurs proposent dans [AHB03] de s'intéresser aux data races de haut niveau. Ces types de data races peuvent survenir lors de l'exécution d'un programme, alors qu'aucune data race "classique" n'est présente dans le programme. Les data races de haut niveau sont des inconsistances lors de la lecture/écriture par des méthodes de plusieurs champs partagés.

ConTest [KLT+07, KLT+09] est un outil visant également à la détection d'erreurs de concurrence. En plus de pouvoir détecter les data races, cet outil peut également détecter des violations d'atomicité. L'une des différences majeures de ConTest est son utilisation de "bruit" lors de l'exécution du programme vérifié. Ce "bruit" consiste à insérer des changements de contexte entre threads ou à leur assigner du retard.

Détection de deadlocks. La vérification à l'exécution peut être utilisée également pour détecter des deadlocks potentiels lors de l'exécution des programmes. L'une des premières approches a été celle proposée dans l'outil VisualThread [Har00] maintenu par HP.

Plus tard, Bensalem et Havelund [BH05] ont amélioré l'algorithme présent dans VisualThread dans le but de limiter le nombre de "faux positifs". Cet algorithme a été intégré à l'outil JPaX, et fonctionne en analysant une trace d'exécution du programme.

Ensuite Bensalem et ses coauteurs [BFHM06] ont étendu l'approche présentée dans [BH05]. En effet, l'un des problème de l'approche précédente était son incorrection et incomplétude. L'approche propose de forcer l'apparition de deadlocks détectés par l'algorithme de l'approche précédente. Ce qui est réalisé par l'ajout au programme d'un observateur détectant le deadlock sur le programme, et d'un contrôleur qui essaye d'amener le programme en deadlock.

Erreur d'atomicité. Même lorsque le système est sans deadlock et qu'aucune data race potentielle n'existe, d'autres types d'erreurs peuvent apparaître : les erreurs d'atomicité. La définition d'atomicité pour un bloc de code est la suivante [FQ03] : *"un bloc de code est atomique si pour chaque exécution entrelacée du programme dans lequel le bloc de code est exécuté, il y a une exécution équivalente du programme où le bloc de code est exécuté séquentielement (sans entrelacement avec les autres threads)"*. Plusieurs approches dynamiques pour tester l'atomicité des programmes sur une exécution existent : *e.g.*, les outils Atomizer [FF04] ou jPredictor [CŞR08].

Vérification de spécifications fournies par un utilisateur. La vérification de spécification fournie par un utilisateur consiste à surveiller l'exécution du programme pour déterminer si la spécification est satisfaite ou non. De nombreuses approches ont été proposées depuis la vérification d'assertions simples à des points précis du programme jusqu'à la vérification d'assertions temporelles à différents points du programme. Selon [HG08], il est possible de classer ces techniques selon quatre dimensions :

Quantification sur les localisations : Est-ce que le formalisme de spécification permet de quantifier sur les localisations du programme. On oppose dans ce cas les spécifications où la propriété est vérifiée à des points précis du programme ou bien de manière générale (*e.g.*, dans un contexte orienté objet : "chaque appel à une méthode du package `org.security` doit être précédé d'une évaluation de la propriété").

Quantification temporelle : Est-ce que le formalisme autorise l'expression d'une relation temporelle. Cette relation peut décrire un ordre souhaité d'apparition des événements ou bien exprimer des contraintes temps réel.

Quantification sur les données : Est-ce que le formalisme autorise la liaison et la référence aux valeurs à travers les états. Par exemple : "pour tout objet `i` de type itérateur, un appel à la méthode `hasNext()` sur l'objet `i` doit précéder un appel à la méthode `next()` sur ce *même* objet". Évidemment la quantification sur les données suppose la quantification temporelle.

Spécification de données abstraites : Est-ce qu'il est possible de définir une fonction de mapping qui relie les états concrets du programme aux états abstraits de la spécification.

Dans [HG08], différentes catégories d'approches sont examinées selon ces quatre dimensions.

Synthèse de moniteurs de vérification

Le problème de la synthèse de moniteurs de vérification se pose lorsque l'on cherche à vérifier une spécification fournie par un utilisateur. Généralement, les moniteurs de vérification sont générés depuis des spécifications basées sur LTL, comme par exemple récemment dans [BLS07b, CR07]. De façon alternative, les expressions ω-régulières ont été utilisées pour la génération de moniteurs, comme par exemple dans [dR05]. Nous renvoyons à [Run09, LS08, HG08] pour une liste plus exhaustive.

2.1.2 Les autres saveurs de la vérification à l'exécution

Nous présentons brièvement quelques approches dérivées de ou reposant sur la vérification à l'exécution. Le point commun entre ces approches est qu'elles utilisent la vérification à l'exécution comme base pour pouvoir réagir lors de l'exécution du système.

Runtime Reflection [Bau08]. La technique de runtime reflection est une extension de la vérification à l'exécution, réalisant également une analyse dynamique de l'état du système. Cette technique repose également sur une phase de diagnostic utilisée lorsque la phase de vérification a détectée une faute. Cette technique est ainsi capable de dire lorsqu'une faute est apparue, et d'en donner la raison. Lorsque ce résultat est assez détaillé et non-ambigu, il peut être utilisé pour produire des commandes de contrôle vers le système, *i.e.*, effectuer une reconfiguration dynamique pour rétablir un état du système correct.

Specification/Property learning [RKF+08, EPG+07]. Il peut arriver qu'aucune spécification ne soit disponible pour être vérifiée sur le système. Dans ce cas, il est possible d'utiliser une technique pour apprendre certains aspects du système s'exécutant (*e.g.*, un certain type de comportement, une relation entre ses entrées et ses sorties). Une spécification du système peut ainsi être produite puis vérifiée. Notamment, cette approche se révéle utile lorsque le système évolue. Par exemple l'outil Daikon [EPG+07] est un détecteur d'invariants. Cet outil exécute le programme examiné, observe ses sorties, et examine les invariants qui sont restés vrais durant l'exécution. L'outil DySy [CTS08] proposé plus tard utilise une exécution symbolique pour améliorer la pertinence des invariants produits.

Property Enforcement. La technique d'enforcement de propriétés [Sch00, LBW09, LBW05] est une extension de la technique de vérification visant principalement à circonvenir l'apparition des fautes. Nous détaillerons le principe de l'enforcement de propriétés dans la Section 2.2.

Autres approches D'autres approches existent pour "réagir" à l'exécution, *e.g.*, les techniques de contract enforcement [PS07, KPS08] et FDIR (Fault Detection Identification and Recovery) [CR94].

2.1.3 Caractérisations des propriétés vérifiables à l'exécution

À la connaissance de l'auteur, deux caractérisations des propriétés vérifiables à l'exécution (ou propriétés monitorables) ont été données à ce jour. Nous présentons ces deux caractérisations.

Les propriétés de safety décidables de Σ^ω : [VK04]. La première caractérisation des propriétés vérifiables à l'exécution a été fournie par Viswanatan et Kim dans [VK04]. Les propriétés monitorables ont été caractérisées comme étant un sous-ensemble strict des propriétés de safety. Les auteurs montrent que, du fait de l'indécidabilité de certains problèmes, un moniteur de vérification est limité par des limites calculatoires. Les propriétés monitorables sont précisément définies comme étant les propriétés de safety décidables [4]. Ils montrent également que cet espace de propriétés correspond à la classe Π^0_1 de la hiérarchie arithmétique qui est la classe des propriétés co-récursivement énumérables.

4. De façon simplifiée, une propriété de safety non-décidable est une propriété de safety pour laquelle le test d'appartenance d'une séquence à la propriété nécessite de tester si la séquence appartient à un ensemble non-décidable. Voir [VK04] pour plus de détails.

Définition de la monitorabilité au sens de [PZ06] : Pnueli et ses coauteurs donnent une notion de propriété monitorable reposant sur la détermination de verdict pour une séquence infinie. Plus précisément, considérant une séquence finie $\sigma \in \Sigma^*$, une propriété $\theta \subseteq \Sigma^\infty$ est déterminée négativement (resp. déterminée positivement) par une séquence d'exécution σ si $\neg\theta(\sigma)$ (resp. $\theta(\sigma)$) et toutes les extensions de σ ne satisfont pas (resp. satisfont) θ. Ensuite, θ est dite être σ-monitorable si σ a une extension *t.q.* θ est négativement ou positivement déterminée par cette extension. Finalement, θ est monitorable si elle est σ-monitorable pour chaque σ. Dans le Chapitre 4 Section 4.2, nous donnons une définition formelle pour ces notions dans le contexte des r-propriétés de la classification *Safety-Progress*.

L'idée intuitive derrière la notion de détermination positive, est qu'il n'est plus nécessaire de continuer l'exécution d'un moniteur dédié à une propriété θ après avoir lu σ si θ n'est pas σ-monitorable. L'un des buts de [PZ06] était de caractériser les situations pour lesquelles il était intéressant de monitorer une propriété.

Caractérisation [BLS07b] des propriétés monitorables dans le sens de [PZ06] : Bauer et ses coauteurs se sont inspirés de la définition de monitorabilité de Pnueli pour en proposer une légèrement différente basée sur la notion de préfixe "good" et "bad" introduite dans le model-checking [KV01] par Kupferman et Vardi. L'idée intuitive est qu'avec les propriétés monitorables, il est possible de détecter des violations ou des validations de propriétés infinitaires avec des séquences finies.

Plus précisement, leur définition de propriété monitorable s'explique de la manière suivante. Considérant une propriété infinitaire $\varphi \subseteq \Sigma^\omega$, un préfixe σ est dit être un "bad" préfixe, ce qui est noté $bad_prefix(\sigma, \varphi)$ (resp. "good" préfixe, noté $good_prefix(\sigma, \varphi)$) de φ si $\forall w \in \Sigma^\omega : \neg\varphi(\sigma \cdot w)$ (resp. $\forall w \in \Sigma^\omega : \varphi(\sigma \cdot w)$). Ensuite, un préfixe σ est dit être "ugly" s'il n'a pas de continuation good ou bad, *i.e.*, $\nexists v \in \Sigma^\omega : bad_prefix(\sigma \cdot v, \varphi) \vee good_prefix(\sigma \cdot v, \varphi)$. Finalement, une propriété est dite être monitorable si elle n'a pas de préfixe "ugly", formellement : $\forall \sigma \in \Sigma^*, \exists v \in \Sigma^\omega : bad_prefix(\sigma \cdot v, \varphi) \vee good_prefix(\sigma \cdot v, \varphi)$.

À propos des caractérisations des propriétés vérifiables à l'exécution. La première caractérisation des propriétés monitorables, donnée dans [VK04], peut sembler arbitraire. Elle caractérise l'espace des propriétés monitorables directement comme une classe de propriétés. Elle n'est pas reliée aux contraintes pratiques auxquelles doit faire face un moniteur d'exécution. En revanche, nous pouvons remarquer que les deux dernières définitions de monitorabilité semblent être données suivant l'intuition qu'un moniteur peut détecter de bonnes ou de mauvaises choses à partir d'une observation finie. Il paraît raisonnable qu'un moniteur de vérification soit aussi utilisé pour détecter la satisfaction d'une propriété désirée sur le système vérifié. De plus, ces deux définitions sont équivalentes sur Σ^∞. Nous parlerons donc de la définition donnée dans [PZ06] car elle est plus générale et a été formulée en premier. Bauer et ses coauteurs ont montré que selon leur définition, l'ensemble des propriétés monitorables est un sur-ensemble strict de l'ensemble des propriétés de safety et de co-safety [5]. Ces classes sont prises depuis la classification Safety-Liveness des propriétés [Lam77, AS85]. Les auteurs donnent également un exemple de propriété de type request/acknowledge qui n'est pas monitorable. Nous reviendrons sur cette propriété [6] dans le Chapitre 4 lorsque nous caractériserons l'espace des propriétés monitorables.

2.1.4 Perspectives et futurs challenges

L'une des motivations principales, qui est un but sous-jacent aux perspectives énoncées ci-après, est de pouvoir arriver à intégrer la technique de vérification à l'exécution dans le cycle de développement des logiciels et systèmes. Amener la technique à un niveau de maturité tel qu'elle puisse s'intégrer comme le test dans les habitudes de développement est assurément un défi ambitieux.

Efficacité du moniteur. L'efficacité du moniteur sera toujours un souci majeur en vérification à l'exécution. Même si beaucoup de progrès ont été faits ces dernières années, réduire l'impact de l'action du moniteur en terme de performance sur le programme initial reste un défi important. Pour que la vérification à l'exécution soit une technique acceptée par les industriels, il faut que la dégradation de performance soit acceptable dans le domaine spécifique des applications vérifiées.

5. Dans la classification *Safety-Progress*, la classe des propriétés de co-safety est la classe des propriétés de guarantee. Voir Chapitre 3.

6. Dans la classification *Safety-Progress*, cette propriété est une response. Voir Chapitre 4, Section 4.2, Exemple 10.

Autour du formalisme de spécification. Plusieurs défis liés au formalisme de spécification des propriétés à vérifier restent à relever. Tout d'abord, dans un souci de faire accepter cette technique par les équipes de développement de logiciels, les formalismes de spécification se doivent d'être compréhensibles et répandus. Définir des approches de vérification à l'exécution dans des formalismes largement utilisés dans le domaine du génie logiciel (comme UML par exemple) semble être un pas dans cette direction. Les sous-ensembles de UML dotés d'une sémantique formelle nous semblent de bons candidats pour cette direction. Factoriser les formalismes de spécification permettrait surement de réduire les ambiguïtés entre les spécifications, et permettrait d'identifier les erreurs de manière plus précise. Ce qui faciliterait également la correction des fautes.

Spécifications paramétriques. Récemment dans [CR09], une approche de vérification à l'exécution pour des propriétés paramétriques a été définie. Cette approche permet d'utiliser des événements paramétrés dans les spécifications. Ces paramètres correspondent à des entités du programmes. Toutefois, même si cette première approche est un pas dans la bonne direction pour ce type de propriétés très utiles et expressives, plusieurs limitations et défis restent à relever. Tout d'abord, aucun verdict clair n'est mis en avant dans ces approches. L'espace d'états du moniteur devient assez large, car celui-ci dépend des différentes entités apparaissant à l'exécution du programme. Cette méthode ne permet pas d'interpréter l'espace d'état du moniteur pour définir un verdict. De plus, les propriétés considérées prennent en compte uniquement les spécifications où les paramètres sont quantifiés de manière universelle. Proposer d'autres types de quantification plus fines semble intéressant ; surtout dans un contexte orienté objet.

Correction par routines correctes. Lorsque l'on vérifie un programme lors de son exécution, et que des fautes apparaissent, on peut vouloir réagir. Une manière possible de réagir est alors de lancer des routines de corrections. Ces routines peuvent avoir été prouvées correctes par d'autres techniques de vérification.

2.2 Enforcement de propriétés à l'exécution

Cette section propose un aperçu des travaux reliés au domaine de l'enforcement à l'exécution. Aussi, nous faisons référence aux travaux portant sur la comparaison des différents mécanismes d'enforcement fournie dans [HMS06] car ces travaux situent l'enforcement à l'exécution par rapport à d'autres mécanismes d'un point de vue calculatoire.

2.2.1 Bref historique des approches d'enforcement par moniteur

L'enforcement à l'exécution a été initié par les travaux de Schneider [Sch00] sur ce qui était appelé les *security automata* (automates de sécurité). Plus tard, Viswanathan [Vis00] a remarqué que la classe des propriétés enforçables est en fait limitée par la capacité de calcul du mécanisme utilisé. Comme le mécanisme utilisé ne peut implanter plus que les fonctions calculables, les propriétés enforçables sont incluses dans les propriétés décidables. Plus récemment, Ligatti et ses coauteurs [LBW09, LBW05] ont montré qu'il est possible d'enforcer à l'exécution plus que les propriétés de safety. En utilisant des mécanismes appelés les *edit-automata*, il est possible d'enforcer la classe plus large des propriétés dites *infinite renewal*. Pour mieux cerner les contraintes pratiques auxquelles peuvent faire face les moniteurs d'enforcement, Fong [Fon04] a étudié l'effet de certaines limitations mémoire pour les moniteurs d'enforcement. Les *Shallow History Automata*, ou automates à mémoire superficielle, peuvent uniquement connaître l'occurrence d'événements passés, et ne gardent ainsi aucune information à propos de l'ordre de leurs arrivées.

L'approche d'enforcement de propriétés a été implantée dans plusieurs outils, *e.g.*, [ES00a, ES00b]. La plupart d'entre eux sont basés plus ou moins sur les automates de sécurité de Schneider. Alors que Polymer [JLD07] introduit un cadre plus expressif basé sur les edit-automata.

2.2.2 Pouvoir calculatoire des mécanismes d'enforcement

Hamlen, Morisett et Schneider ont proposé [Ham06, HMS06] une classification des propriétés enforçables en considérant un programme comme une machine de Turing. Leur but était de délimiter l'ensemble

des propriétés enforçables en fonction du mécanisme utilisé. Les propriétés sont classées en fonction de la modification que le mécanisme d'enforcement peut réaliser sur le programme sous-jacent. Notamment chaque mécanisme utilisé correspond à une certaine classe (en calculabilité) de propriétés :

- *Les propriétés enforçables par analyse statique du programme sous-jacent.* Ce sont les propriétés décidables sur le programme sous-jacent.
- *Les propriétés enforçables par des moniteurs d'exécution.* Ce sont les propriétés co-récursivement énumérables.
- *Les propriétés enforçables par réécriture.* Ici la classe de propriétés dépend de la relation d'équivalence utilisée entre les programmes.

Nous nous focaliserons sur les propriétés enforçables par des moniteurs d'exécution.

2.2.3 Caractérisations de l'ensemble des propriétés enforçables avec des moniteurs d'exécution

Les automates de sécurité et les propriétés de safety décidables. Schneider a introduit les automates de sécurité (security automata) comme les premiers mécanismes d'exécution pour enforcer des propriétés. Dans [Sch00], Schneider définit une variante des automates de Büchi qui s'exécutent en parallèle avec le programme sous-jacent. Ces automates sont dotés du pouvoir d'arrêter le programme dès que l'automate de sécurité détecte une violation de la propriété examinée. Schneider a annoncé que l'ensemble des propriétés enforçables avec ce type de mécanisme est l'ensemble des propriétés de safety. Ensuite dans [HMS06] Schneider, Hamlen, et Morisett ont raffiné l'ensemble des propriétés enforçables par ce mécanisme en se basant sur les travaux de Viswanathan [Vis00] : la classe des propriétés enforçables est impactée par la puissance de calcul du moniteur d'enforcement. Comme le mécanisme d'enforcement ne peut implanter plus que les fonctions calculables, les propriétés enforçables sont incluses dans l'ensemble des propriétés décidables. De là, ils ont montré que l'ensemble des propriétés de safety était une limite supérieure stricte du pouvoir des moniteurs d'enforcement travaillant à l'exécution et définis comme des automates de sécurité.

Les Edit-automata et les propriétés de renewal infinies. Ligatti et ses coauteurs [LBW09, LBW05, Lig06, JLD07] ont introduit les *edit-automata* (et leurs variantes) comme mécanismes d'enforcement à l'exécution. Ils ont remarqué que, en arrêtant uniquement le programme, les security automata définis par Schneider étaient trop limités. En fonction de l'entrée courante (produite par le programme) et son état de contrôle courant, un edit-automata peut soit insérer une nouvelle action en remplaçant l'entrée, soit supprimer l'entrée courante (possiblement mémorisée dans l'état courant pour plus tard). Des variantes des edit automata ont également été définies : insertion automata (ne peuvent qu'inserer de nouvelles entrées), suppression automata (ne peuvent que supprimer des entrées).

Les propriétés enforcées par les edit-automata sont appelées les propriétés de *infinite renewal*. Elles ont été définies comme les propriétés pour lesquelles toute séquence infinie valide a un nombre infini de préfixes valides [LBW09]. L'ensemble des propriétés de renewal est un sur-ensemble des propriétés de safety, qui contient certaines liveness, mais pas toutes. Formellement, considérant un alphabet commun Σ, l'espace des propriétés considérées dans [LBW05, LBW09, Lig06] est 2^{Σ^∞}. Ainsi, une propriété θ est dite être une propriété de infinite renewal si et seulement si $\forall\sigma \in \Sigma^\infty : \theta(\sigma) \Rightarrow \forall\sigma' \in \Sigma^* : \sigma' \prec \sigma \Rightarrow \exists\sigma'' : \sigma' \leq \sigma'' \prec \sigma \wedge \theta(\sigma'')$. Dans la classification *Safety-Progress* des r-propriétés (cf. Chapitre 3), les propriétés d'infinie renewal peuvent être vues comme des propriétés de response.

Les Shallow History Automata et un treillis informatif des propriétés enforçables [Fon04]. Fong a étudié l'effet de restreindre la capacité des moniteurs d'enforcement à l'exécution et l'impact sur leurs pouvoir d'enforcement. Les Shallow History Automata (SHA) gardent en mémoire une histoire des événements d'accès qu'un programme sous-jacent a fait. Fong a montré que ces automates pouvaient enforcer un ensemble de propriétés strictement inclus dans l'ensemble des propriétés enforçables par les automates de Schneider. Le résultat a été généralisé en utilisant des mécanismes d'abstraction sur une variante (équivalente) des automates de Schneider. Ce qui a donné lieu à un treillis des propriétés enforçables basé sur l'information que pouvait mémoriser le moniteur. En haut de ce treillis est placé l'ensemble des propriétés enforçables par les security automata. Dans ce cas, ces SHA gardent l'historique de tous les événements. Au bas de ce treillis on retrouve l'ensemble des politiques interdisant un ensemble d'événements particulier. Dans ce cas, les SHA ne distinguent pas les préfixes faits des mêmes événements.

La classification définie par Fong a un intérêt pratique, dans le sens où elle étudie l'effet de limites pratiques de programmation (comme une mémoire limitée). Elle montre également que certaines politiques de sécurité classiques restent enforçables en utilisant de tels shallow automata.

2.2.4 Synthèse de mécanismes d'enforcement à l'exécution

Il y a eu assez peu d'efforts de recherche dédiés à la synthèse de mécanismes d'enforcement à l'exécution. Dans [MM07] Martinelli et Matteucci s'intéressent à la synthèse de mécanismes d'enforcement comme définis par Ligatti. Plus généralement, les auteurs considèrent les security-automata et les diverses formes d'edit-automata. Le moniteur est modélisé par un opérateur algébrique exprimé en CCS. Le programme sous-jacent est alors un terme $Y \rhd_K X$ où X est le programme cible, Y le programme contrôleur et $\rhd_K$ l'opérateur modélisant le moniteur où K est le type de moniteur (security automata, insertion, suppression ou edit). La propriété désirée sur le système sous-jacent est formalisée en utilisant le μ-calcul. Dans [Mat07] Matteucci étend l'approche dans le contexte des systèmes temps-réel.

2.2.5 Quelques approches d'enforcement

Dans [CCBR06] Cuppens et ses coauteurs dérivent des aspects abstraits exprimés en AspectJ depuis des exigences exprimées en Nomad. Nomad [CCBS05] est une logique temporelle métrique. Ils s'intéressent aux propriétés de disponibilité : politiques de temps d'exécution et d'attente maximales. L'utilisation qui en est faite est de prévenir les attaques par SYN flooding dans le protocole TCP/IP.

La technique d'enforcement a souvent été utilisée en pratique pour la mise en place de politiques de contrôle d'accès [AH07, TS07, dOWKK07]. Par exemple, dans [dOWKK07] de Oliveira et ses coauteurs proposent une approche d'enforcement des politiques de sécurité de contrôle d'accès basées sur la réécriture. Leur approche est également basée sur AspectJ. Les politiques de contrôle d'accès sont formalisées en utilisant des systèmes de réécriture. Les politiques sont tissées dans le programme sous-jacent en utilisant Tom, une extension du langage Java pour définir des systèmes de réécriture.

Monitoring Oriented Programming (MOP) [CDR05, CR07] peut être vu comme un paradigme de programmation dans lequel la synthèse de moniteur est réalisée depuis une spécification donnée par l'utilisateur. Cette spécification peut s'exprimer dans divers formalismes, ce qui est l'une des forces de cette approche. En effet, les spécifications disponibles pour l'utilisateur sont variées : LTL, ERE (Extended Regular Expression), CFG (Context Free Grammar),... Cet environnement est implanté dans un outil dédié au langage Java : JavaMOP [CR05]. Dans cette instantiation du framework MOP, le langage d'expression d'aspects AspectJ est utilisé également de manière sous-jacente. À noter que récemment le framework MOP a été implanté dans une version dédiée aux BUS matériels : BusMOP [PMCR08]. Le framework MOP, bien que dédié à la vérification de propriétés, peut être vu comme un système d'enforcement de propriétés. En effet, grâce à l'utilisation de handlers, l'utilisateur peut déclarer du code à exécuter lorsque la propriété spécifiée est violée ou validée. En revanche, aucun lien n'est précisé entre le code à exécuter dans le handler et la propriété vérifiée.

Dans [PMMD05] on présente un système pour traduire des "role slices", *i.e.*, un ensemble de règles RBAC, en aspects. Cette approche est basée sur la modélisation fonctionnelle d'un programme orienté objet ainsi que d'un aspect.

Polymer [BLW05, Lig06] est un langage et un système permettant de spécifier et d'enforcer des politiques de sécurité pour les applications Java. La politique de sécurité est écrite dans une extension du langage Java. Celle-ci est ensuite compilée en bytecode puis intégrée dans le programme original à l'aide de l'outil BCEL.

2.3 Test de propriétés

Le test de logiciel est un sujet très vaste, bénéficiant d'une popularité reconnue. Nous nous focalisons dans cette section sur les approches reliées au test qui sont connexes aux travaux que nous proposerons dans le Chapitre 4 Section 4.4 et Chapitre 7. Nous commençons par rappeler les travaux existants sur la testabilité des propriétés. Puis nous présentons quelques approches liées visant le test de propriétés. Enfin, comme une partie de l'approche de test de propriétés que nous proposons dans cette thèse s'inscrit dans un travail d'équipe, nous la restituons dans le temps en mentionnant les autres travaux complémentaires de nos équipes.

2.3.1 Caractérisation des propriétés testables

Dans [NGH93, Gra94], Nahm, Grabowski, et Hogrefe se sont intéressés à l'ensemble des propriétés temporelles qu'il était possible de tester sur une implémentation. La notion de test considérée utilise une notion de conformité qui est une relation entre les séquences d'exécution décrites par une propriété et l'ensemble des séquences d'exécution que peut produire le programme testé. Les relations considérées entre les séquences de la propriété et celles du programme sont des relations ensemblistes. Une propriété est dite testable s'il est possible de déterminer si une telle relation est vérifiée. Dans leurs travaux, les auteurs étudient la testabilité des propriétés par rapport à la classification *Safety-Progress* des propriétés ([CMP92b] et Chapitre 3). Les ensembles de propriétés testables annoncés sont les ensembles des propriétés de safety et de guarantee. Dans le Chapitre 4 Section 4.4, nous reviendrons sur les travaux présentés dans [NGH93, Gra94] et étendrons ces résultats.

2.3.2 Approches pour le test de propriétés

Nous présentons quelques approches connexes à celle qui sera présentée dans cette thèse autour du test de propriétés.

Test orienté par les propriétés pour la génération d'objectifs de test

L'une des limites du test de conformité est, qu'à partir d'une spécification complète, la taille de la suite de test générée peut être infinie ou non réalisable pratiquement. Des approches de *test orientées par les propriétés* proposent de faire face à cette limitation en testant des propriétés particulières ayant un coté critique. Dans ce cas, des propriétés sont utilisées en plus de la spécification pour générer des objectifs de test qui sont utilisés pour la conduite des cas de test. Le but des objectifs de test est de selectionner un sous ensemble de cas de test à exécuter. Ainsi, ces objectifs de test permettent d'évaluer des fonctionnalités précises du système sous test. Une fois les objectifs de test générés, une sélection des cas de test peut être effectuée en utilisant les techniques classiques définies sur les systèmes de transitions [JJ05, dV01]. Par exemple dans [FMP03], Fernandez et ses coauteurs présentent une approche permettant de générer des cas de test en utilisant des formules de LTL comme objectifs de test. Pour une présentation (non exhaustive) de quelques approches générales, nous renvoyons à [MSM07].

Test de politiques de sécurité

Ces dernières années plusieurs approches ont visé le test de politiques de sécurité en étendant les approches classiques de test de conformité.

Par exemple dans [TMB07, PMT08], le modèle fonctionnel du système est défini en utilisant des contrats qui sont des cas d'utilisation (use cases) augmentés de pré et post conditions. La politique de sécurité est exprimée dans un modèle de contrôle d'accès. Des cas de test sont dérivés pour tester la sécurité et le coté fonctionnel du système de manière complémentaire.

Dans [MOC+07], les auteurs proposent d'intégrer une politique de type contrôle d'accès spécifiée en OrBAC [ABB+03] à une spécification exprimée par une machine à états finis étendue (EFSM).

Dans [MMC08b] les auteurs s'intéressent au test d'une politique de sécurité pour services Web exprimée en Nomad, qui est une forme de logique mi-déontique mi-temporelle. L'approche consiste en une intégration des formules de logique dans une spécification du service Web basée sur des automates communicants.

Des approches de test passif (*e.g.*, [MMC08a, MWC08]), similaires à l'approche de vérification à l'exécution en mode offline, ont été utilisées également pour vérifier la conformité du traffic d'un réseau par rapport à des exigences fonctionnelles ou de sécurité.

Test basé sur les exigences - "Requirement-Based testing"

Dans le test basé sur les exigences, le but est de générer une suite de tests à partir d'un ensemble d'exigences formelles ou informelles. Par exemple dans [RWH07, WRHM06, PRB09], des cas de test sont générés à partir de formules LTL en utilisant un model-checker. Ces approches s'intéressent à définir une notion syntaxique de couverture de l'exigence testée.

2.3.3 Test de propriétés sans spécification comportementale

Certains travaux des équipes Vérimag DCS, et LIG Vasco se sont intéressés à la définition d'une approche de test de propriétés sans spécification comportementale. La motivation originale était le test de politiques de sécurité réseau [DFG+05]. Il s'agissait d'adapter les méthodes de test de conformité de protocole pour le test de politiques de sécurité. Plusieurs problèmes [DFG+05] se posent lors du test de politiques de sécurité. Tout d'abord, plusieurs niveaux d'abstraction sont mis en jeu par la définition de la politique de sécurité. Les événements décrits par la politique n'apparaissent pas forcément à un niveau d'interface précis entre le testeur et le système. De plus, les interactions que doit mettre en œuvre un testeur se situent à différents niveaux. L'étude réalisée dans [DFG+05] a mis en lumière les limitations du test de conformité classique pour son application au test de politiques de sécurité. Cependant, l'étude montre également que le test de conformité pourrait être étendu afin de prendre en compte la complexité du modèle et des interactions mises en jeu. Ensuite, les travaux de nos équipes ont donné lieu à la définition d'une approche de test par tessons (tile-based testing). La méthode de génération de test est décrite dans [DFG+06], puis formalisée dans [FFMR06] à l'aide d'une algèbre de processus. Nous avons ensuite montré qu'il était possible de générer les tests depuis plusieurs formalismes, et d'exécuter ceux-ci de manière répartie dans [FFMR07a]. Ensuite, une application au test de politiques de sécurité réseaux a été présentée dans [DRG08].

Première partie

Cadre général

CHAPITRE 3

Spécification de propriétés dans la classification *Safety-Progress* pour la validation à l'exécution

Sommaire

3.1 Introduction **33**
3.1.1 La classification Safety-Liveness 33
3.1.2 La classification Safety-Progress (résultats précédents) 35
3.1.3 Motivations et contributions 35
3.2 Description informelle de la classification **36**
3.3 r-propriétés : Runtime properties **38**
3.4 La vue langage des r-propriétés **39**
3.4.1 Construction de propriétés finitaires et infinitaires 39
3.4.2 La hiérarchie des langages 42
3.5 La vue logique temporelle des r-propriétés **44**
3.5.1 Syntaxe et sémantique 44
3.5.2 Hiérarchie des formules temporelles 46
3.6 La vue automates des r-propriétés **49**
3.6.1 Automates de Streett 49
3.6.2 La hiérarchie des automates 50
3.6.3 Synthèse d'automates de Streett à partir de DFAs 51
3.7 La vue topologique des r-propriétés (Remarque) **59**
3.8 Connexions entre les différentes vues **59**
3.9 Conclusion et perspectives **60**

Chapter abstract

When analyzing a system at runtime, the underlying property, its definition and representation play a major role. Having a suitable and convenient framework to express properties is thus a concern.

In this chapter we revisit and extend the existing results about the *Safety-Progress* classification of ω-regular properties, introduced by Chang, Manna, and Pnueli [CMP92a, CMP92b, MP90b]. Our proposed extensions are motivated by runtime analysis. Hence, we believe that this general classification is a suitable basis to specify program properties. When analyzing a system at runtime, an important feature is the interest for finite execution sequences and their evaluation with respect to the considered properties.

In this chapter, we recall the previously established results and integrate our contributions. This chapter is an extended version of [FFM09c]. We provide extra results and proofs. Moreover, the reasoning is more intuitive, and some additional examples are provided.

Chapter organization. The remainder of this chapter is organized as follows. Section 3.1 introduces this chapter : we give a brief overview of the Safety-Liveness classification, then we summarize existing results in the *Safety-Progress* classification, and finally delineate our contributions. Then in Section 3.2 we introduce informally the hierarchy. The language-theoretic view is studied in Section 3.4. The Section 3.5 is dedicated to the temporal logic view. Next, in Section 3.6, we deal with the automata view. In Section 3.7, we recall briefly existing results on the topological view of infinitary properties ; we do not develop this view for finitary properties. Then, in Section 3.8, we present some connections between the views introduced so far. Finally, in Section 3.9 we give some concluding remarks. The presentation of this chapter follows mostly the one used in [CMP92a].

Résumé du chapitre

Lorsqu'on analyse un système à l'exécution, la propriété sous-jacente, sa définition, et sa représentation jouent un rôle majeur. Avoir un cadre de raisonnement adapté et pratique est donc une motivation.

Dans ce chapitre nous revisitons et étendons les résultats connus à propos de la classification *Safety-Progress* des propriétés ω-régulières introduite par Chang, Manna, et Pnueli [CMP92a, CMP92b, MP90b]. L'extension que nous proposons est motivée par la vérification à l'exécution. Nous montrons dans ce chapitre comment il est possible d'étendre cette classification pour prendre en compte les séquences finies et infinies de manière cohérente. Aussi nous croyons que cette classification générale est une bonne base pour spécifier les propriétés d'un programme. En vérification à l'exécution, une caractéristique importante est l'intérêt pour les séquences d'exécution finies et leur évaluation vis-à-vis des propriétés considérées.

Nous rappelons dans ce chapitre les résultats précédemment établis en y intégrant nos contributions. Ce chapitre est une version étendue de [FFM09c]. Nous y apportons des résultats et preuves supplémentaires. Des exemples additionnels sont également fournis ainsi qu'un raisonnement plus intuitif.

Organisation du chapitre. La suite de ce chapitre est organisée comme suit. La Section 3.1 introduit ce chapitre : nous rappelons quelques résultats sur la classification Safety-Liveness, puis résumons les résultats précédents de la classification *Safety-Progress*, et enfin mettons en évidence nos contributions. Dans la Section 3.2 nous introduisons informellement la hiérarchie. La vue langage est étudiée dans la Section 3.4. Nous étudions la vue logique temporelle dans la Section 3.5, et dans la Section 3.6 la vue automates. Dans la Section 3.7 nous résumons les résultats existants pour la vue topologique des propriétés infinitaires ; nous ne développerons pas de vue topologique pour les propriétés finitaires. Dans la Section 3.8, nous présentons les connexions entre les différentes vues. Enfin, en Section 3.9 nous tirons les conclusions de ce chapitre. Pour la structure de la présentation de ce chapitre, nous suivons principalement [CMP92a].

3.1 Introduction

Nous commençons par présenter brièvement dans cette introduction la classification *Safety-Liveness* des propriétés. C'est l'une des classifications les plus connues des propriétés utilisées en vérification formelle. Ensuite, nous rappelons brièvement les résultats précédents amenés par les travaux autour de la classification *Safety-Progress* des propriétés. Enfin, nous exposons nos motivations pour étendre cette classification afin de valider des propriétés sur les programmes lors de leur exécution.

3.1.1 La classification Safety-Liveness

Dans [Lam77], Lamport suggère que deux classes de propriétés peuvent être distinguées utilement : les safety et les liveness.

Les propriétés de **safety** déclarent que quelque chose de mauvais ne peut jamais arriver.

Les propriétés de **liveness** déclarent que quelque chose de bon arrivera inévitablement.

Dans [Lam77], la distinction entre les deux classes de propriétés se justifie par le fait qu'elles amènent des preuves différentes lorsque l'on souhaite donner une preuve formelle qu'un programme donné satisfait une propriété.

La classe des propriétés de Safety [Lam77, Lam83] ("some bad thing does not happen"). La preuve de satisfaction par un programme se fait par une induction calculatoire [1] (inductional computation) qui repose sur un argument d'invariance (cf. [MP84] ou [Smi91] pour plus de détails).

Des exemples de propriétés de safety typiques sont fondamentales pour la construction de systèmes : exclusion mutuelle (mutual exclusion), absence de blocage (deadlock freedom), correction partielle (partial correction), premier arrivé/premier servi (fifo ordering).

La classe des propriétés de Liveness [AS85] ("a good thing eventually happens"). La preuve de satisfaction par un programme se fait par induction structurelle [2] qui repose sur un argument de "bien fondé". (cf. [OL82]). Des exemples de liveness typiques sont l'absence de famine, la terminaison, *etc*

Il est possible de distinguer deux types particuliers de liveness [Sis85] :

- liveness uniforme : il y a une exécution qui peut compléter toute exécution (partielle) de manière à ce que la séquence complétée satisfasse la propriété,
- liveness absolue : toute séquence d'exécution satisfaisant la propriété peut être utilisée pour compléter toute exécution (partielle) de manière à ce que la séquence complétée satisfasse la propriété.

Nous pouvons remarquer que toute liveness absolue est uniforme, et toute liveness uniforme est une liveness.

Les auteurs de [Sis85] avancent que les notions de liveness uniforme et absolue ne capturent pas la notion intuitive de liveness. Pour cela, ils présentent deux exemples de liveness (plus intuitives).

- Leads-to properties : Toute occurrence d'évènements de type E_1 est un jour (eventually) suivie par une occurrence d'un évènement de type E_2. Ce n'est pas une liveness absolue.
- Prédictive : Si A est vrai initialement alors après une exécution partielle, B est tout le temps vrai. Ce n'est pas une liveness uniforme.

A propos de cette classification. Beaucoup de propriétés ne peuvent s'exprimer uniquement comme des liveness ou des safety. En revanche, toute propriété peut s'exprimer comme l'intersection d'une liveness et d'une safety, ce qui est montré dans [AS85] en utilisant un argument topologique. Voici deux exemples :

- propriété de type Until : Un jour un évènement de type E_2 arrivera et tous les évènements précédents sont de type E_1. La partie safety stipule que $\neg E_1$ n'arrive pas avant E_2, et la partie liveness garantie que E_2 arrivera.

1. Une induction calculatoire [Smi91] est utilisée principalement pour montrer des résultats de correction partielle sur un programme. Elle se fait en la supposant vraie au début du programme, et en montrant qu'elle est préservée par les actions de celui-ci. Si le programme termine et est déterministe, alors le programme peut être vu comme un calcul en n pas (toujours les mêmes). Dans ce cas l'induction calculatoire revient à faire une induction sur la longueur des calculs.

2. La preuve d'une propriété par induction structurelle sur un programme se fait en deux étapes. La première consiste à prouver que la propriété est vraie sur les catégories syntaxiques de base. Ensuite, il faut prouver que la propriété est vraie pour tous les éléments composés. Ce qui se fait en supposant d'abord la propriété sur les constituants de l'élément composé concerné (hypothèse d'induction), puis en la prouvant pour l'élément lui même.

– propriété de correction totale : s'exprime comme correction et terminaison. La correction est une propriété de safety, alors que la propriété de terminaison est une liveness.

Notons qu'une seule propriété est à la fois une safety et une liveness : la propriété satisfaite par toute séquence d'exécution.

Plotkin a montré que toute propriété exprimable en logique temporelle peut s'écrire comme la conjonction de deux propriétés de liveness exprimables en logique temporelle. Ce résultat est en fait plus général [AS85] : dès qu'un système a plus de deux états, alors toute propriété sur le système peut s'exprimer comme la conjonction de deux liveness.

Selon [CMP92a], une bonne spécification inclut à la fois des propriétés de safety mais aussi des propriétés de liveness. Toutefois utiliser un langage capable d'exprimer les deux types de propriétés a un surcoût. Si l'on veut utiliser uniquement des safety, on peut se restreindre à un langage de prédicats sur des comportements finis. La seule justification pour utiliser des langages comme la logique temporelle (ou un autre langage équivalent) est sa capacité à exprimer des prédicats sur des comportements infinis à l'aide des propriétés de liveness [3]. Une raison pour étudier les classifications des propriétés est d'identifier un compromis entre expressivité d'un langage de spécification et sa complexité.

Caractérisation des safety et liveness [BK08]. Considérons une propriété infinitaire φ. Informellement, il est possible de caractériser les propriétés de safety et de liveness comme suit :

Si pour chaque séquence infinie $\sigma \in \Sigma^\omega$, il est possible de déterminer si $\sigma \in \varphi$ en examinant seulement les préfixes (finis) de σ, alors la propriété φ est une propriété de safety.

Si pour chaque séquence $\sigma \in \Sigma^\omega$, il n'est pas possible de déterminer si $\sigma \in \varphi$ en examinant les préfixes (finis) de σ, alors la propriété φ est une propriété de liveness.

Ce qui peut être formalisé comme suit :

Définition 5 (Limite de propriétés). Considérons une séquence infinie $\sigma \in \Sigma^\omega$.

La séquence σ est une limite de la propriété finitaire $\phi \subseteq \Sigma^*$ si $pref(\sigma) \subseteq \phi$.

La séquence σ est une limite de la propriété infinitaire $\varphi \subseteq \Sigma^\omega$ si σ est une limite de $Pref(\varphi)$.

Autrement dit, $\sigma \in \Sigma^\omega$ est une limite de $\varphi \subseteq \Sigma^\omega$ si chaque préfixe de σ peut être continué en un mot de φ.

Définition 6 (Propriété infinitaire limite-close). La propriété infinitaire φ est limite-close si elle contient toutes ses limites :

$$\forall \sigma \in \Sigma^\omega : pref(\sigma) \subseteq Pref(\varphi) \Rightarrow \sigma \in \varphi$$

C'est-à-dire, si chaque préfixe de σ peut être étendu en une séquence de φ, alors σ est aussi dans φ. Une caractérisation des propriétés en safety ou liveness peut être donnée pour les propriétés infinitaires.

Définition 7 (Safety/Liveness). Étant donnés un vocabulaire Σ, et $\varphi \subseteq \Sigma^\omega$ une propriété infinitaire définie sur Σ. La propriété φ est dite être :

une safety si elle est limite-close,

une liveness si $Pref(\varphi) = \Sigma^*$.

---

3. Bien que les approches de vérification à l'exécution se soient principalement concentrées sur les propriétés de safety, les approches basées sur LTL sont majoritaires. Ceci peut s'expliquer par le fait que les techniques de vérification à l'exécution ont leur origine dans les techniques de model-checking. Les travaux sur le model-checking se sont longuement intéressés (entre autres) à la génération d'automates de Büchi à partir de formules LTL. L'utilisation d'un tel automate pour vérifier "en ligne" une propriété est donc naturelle.

3.1.2 La classification *Safety-Progress* (résultats précédents)

La classification *Safety-Progress* est une alternative à la dichotomie et classique Safety-Liveness. Au contraire de cette dernière, la classification *Safety-Progress* est une hiérarchie. Un de ses attraits est de fournir une classification plus fine des propriétés. Cette classification dans sa définition originale adresse uniquement les propriétés ω-régulières (infinitaires). De plus, les propriétés de chaque classe sont caractérisées selon 4 *vues* [CMP92a] :

Vue langage. Dans cette vue les propriétés sont des ensembles de séquences infinies d'exécution. Elle décrit la hiérarchie selon la manière dont chaque classe peut être construite à partir d'ensembles de séquences finies. Elle définit ainsi une hiérarchie de langages.

Vue logique. Dans la vue logique, les propriétés sont décrites par des formules de logique linéaire temporelle (LTL). Chaque classe de la hiérarchie est caractérisée syntaxiquement par un motif de formule.

Vue topologique. Dans cette vue, les propriétés sont des ensembles ouverts ou fermés.

Vue automate. Dans cette vue, les propriétés sont représentées par des automates de Streett [Str81]. Chaque classe de propriétés est donnée par des restrictions syntaxiques sur la forme des automates de Streett. Pour les automates de Streett, une condition d'acceptation était définie pour les séquences infinies.

Dans [CP03], les auteurs classifient les propriétés linéaires temporelles selon la complexité de leur vérification. Leur motivation est d'étudier le problème du langage vide, utilisé en model-checking, selon les différentes classes. Les auteurs introduisent deux nouvelles vues à cette classification.

La première repose sur des automates à mots infinis où la condition d'acceptation et la fonction de transitions changent. Pour un type de fonction de transition et une condition d'acceptation donnée, les auteurs déterminent la classe des propriétés reconnue par de tels automates. Les différentes possibilités étudiées sont les suivantes :

- condition d'acceptation : Büchi, co-Büchi, et Streett (occurrences infinies ou occurrence simple) ;
- fonction de transition : déterministe, non-déterministe, universelle, et alternante.

Une seconde caractérisation existante consiste en une hiérarchie Until-Release : une vue logique (LTL) dans laquelle les propriétés sont classées en fonction de la profondeur d'alternance des opérateurs Until et Release.

3.1.3 Motivations et contributions

Motivations. Dans le but de valider des propriétés à l'exécution de systèmes, nous choisissons la classification *Safety-Progress* des propriétés ω-régulières [4]. Plusieurs raisons ont justifié notre choix.

- Tout d'abord, comme dit plus haut, ce cadre offre une classification plus fine des propriétés. Ainsi il est possible de distinguer plusieurs types de propriétés temporelles pour les systèmes à valider. Également, cette classification permet d'obtenir une meilleure compréhension de l'espace des propriétés régulières. En effet, les propriétés de liveness y sont raffinées en plusieurs classes hiérarchiques. De plus, il est possible d'établir des liens entre les différentes classes.
- Aussi, raisonner dans ce cadre est un atout pour spécifier des propriétés sur les systèmes que l'on veut valider. En effet (cf. Chapitre 4), nous verrons que les différentes méthodes de validation à l'exécution ne s'appliquent pas à toutes les classes de propriétés. Les méthodes de validation s'appliquent à certaines classes (soit exactement, soit pour un sur/sous ensemble).
- Également, ce cadre offre quatre vues différentes pour les propriétés, fournissant ainsi quatre moyens d'exprimer une propriété désirée sur le système à valider.
- En outre, dans chaque vue, il est possible de déterminer syntaxiquement quelle est la classe d'une propriété donnée. Ainsi après avoir spécifié une propriété, il est possible de déterminer immédiatement quelle technique de validation peut être mise en oeuvre pour cette propriété.

Dans ce chapitre nous revisitons cette classification dans une optique de vérification à l'exécution. Les mots usuellement considérés sont les séquences d'exécution produites par un programme sous-jacent. Aussi les propriétés sont utilisées pour décrire la spécification du programme : une propriété spécifiée dans cette classification doit être satisfaite dans le programme sous-jacent. Nous préciserons dans les chapitres

4. Dans la suite de cette thèse, le terme de propriété désignera une propriété ω-régulière.

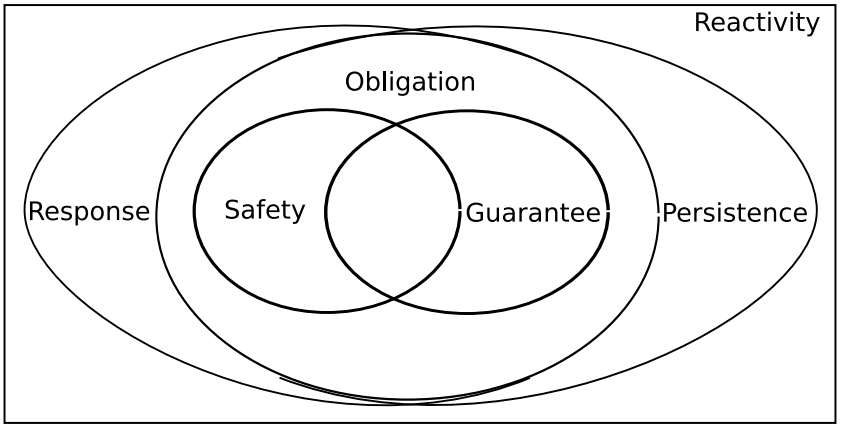


FIGURE 3.1 – Les classes de la classification *Safety-Progress* en représentation ensembliste

suivants le lien précis entre le programme et sa spécification. Pour pouvoir traiter les propriétés finitaires et infinitaires, nous utiliserons la notion de r-propriétés. Notre étude de la classification *Safety-Progress*, dans laquelle nous prenons en compte les séquences finies, concernera uniquement les vues langage, logique, et automate.

Contributions. Les contributions amenées par ce chapitre sont les suivantes :

- Pour la vue langage : nous donnons une définition formelle pour l'ensemble des opérateurs introduits dans [CMP92b]. De plus, nous introduisons deux opérateurs finitaires, un pour les propriétés de response, l'autre pour celle de persistence. Les définitions de ces opérateurs sont consistantes avec leur équivalents infinitaires.
- Pour la vue automate : nous introduisons un critère d'acceptation des automates de Streett pour les séquences finies. Ce critère est compatible avec les opérateurs produisant des propriétés finitaires de la vision langage. De plus, nous introduisons une transformation sur les automates d'états finis (DFA) [HMU79] afin de produire un automate de Streett étant donné un motif de répétition souhaité pour la propriété finitaire. Ces transformations sont les représentants dans la vue automate des opérateurs de construction de propriétés dans la vue langage. Similairement, les transformations sont spécifiques à chaque classe de propriétés.
- Nous montrons que les extensions apportées à la classification *Safety-Progress* sont cohérentes et que l'uniformité des vues est préservée.
- Enfin, nous complétons les connexions possibles entre les différentes vues de la classification.

3.2 Description informelle de la classification

Cette Section introduit informellement la structure de la classification *Safety-Progress* des propriétés. La classification *Safety-Progress* est constituée de quatre classes de base de propriétés définies sur les séquences d'exécution. Cette description informelle des classes de propriétés a été introduite initialement uniquement pour les séquences infinies dans [CMP92a]. Les classes sont définies informellement comme suit, une définition plus précise sera donnée dans les sections suivantes :

- Les propriétés de *safety* (sûreté) sont les propriétés pour lesquelles lorsqu'une séquence satisfait une telle propriété *tous ses préfixes* la satisfont.
- Les propriétés de *guarantee* (garantie) sont les propriétés pour lesquelles lorsqu'une séquence satisfait une telle propriété *il existe (au moins) un* préfixe qui la satisfait.
- Les propriétés de *response* (réponse) sont les propriétés pour lesquelles lorsqu'une séquence satisfait une telle propriété *un nombre infini* de ses préfixes la satisfont.
- Les propriétés de *persistence* (persistance) sont les propriétés pour lesquelles lorsqu'une séquence satisfait une telle propriété *tous ses préfixes à partir d'un certain rang* la satisfont. Autrement dit, seul un nombre fini de ses préfixes ne la satisfont pas.

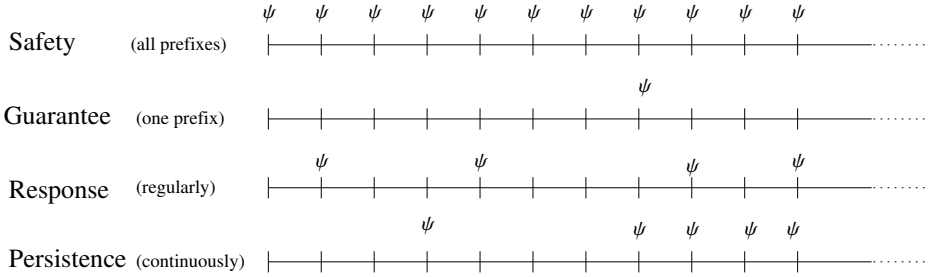


FIGURE 3.3 – Illustration informelle des différences entre les classes (de base) de propriétés

Deux classes additionnelles peuvent être définies par combinaison booléennes (union et intersection) finies des propriétés des classes de base.

- La classe des propriétés *d'obligation* peut être définie comme la classe obtenue par combinaison booléenne de propriétés de safety et de guarantee.
- La classe des propriétés de *reactivity* (réactivité) peut être définie comme la classe obtenue par combinaison booléenne de propriétés de response et de persistence. Cette classe est la plus générale (*i.e.*, toutes les autres classes y sont incluses) et contient toutes les propriétés linéaires temporelles [CMP92a].

Plus précisément, les propriétés des classes de base sont construites à partir de propriétés finitaires. D'une certaine manière, il est possible de voir la construction des propriétés des différentes classes comme l'application de différents "motifs de satisfaction" à une propriété finitaire. Nous utilisons ce principe dans l'outil j-VETO (Chapitre 8) pour obtenir des r-propriétés des différentes classes (puis ensuite les moniteurs qui leurs sont associés).

Nous illustrons maintenant l'organisation hiérarchique de la classification ainsi que la construction de propriétés pour les classes de base.

Organisation hiérarchique. L'organisation hiérarchique de la classification est représentée sur la Fig. 3.2.

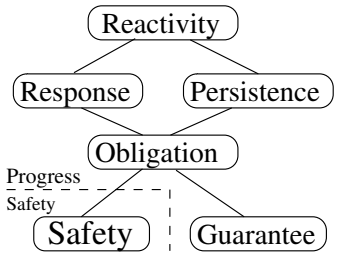


FIGURE 3.2 – Hiérarchie des classes

Pour deux classes reliées, la classe située en dessous est une sous-classe de celle située au dessus. Par exemple, les classes des propriétés de safety et de guarantee sont les deux plus petites classes. Elles sont contenues dans la classe des propriétés d'obligation. Les classes des propriétés de response et de persistence contiennent les propriétés d'obligation et sont contenues dans la classe des propriétés de reactivity. De plus, la classe des propriétés d'obligation est l'intersection de la classe des propriétés de response et de la classe des propriétés de persistence. Enfin, la classe des propriétés de reactivity est la classe la plus générale contenant toutes les autres. Nous allons dans les sections suivantes compléter ce schéma avec les différentes vues de la classification. Une représentation ensembliste de la classification est montrée sur la Fig. 3.1 page précédente. Dans la suite, nous dirons qu'une propriété d'une classe donnée est *pure* si elle ne peut s'exprimer comme une propriété des classes inférieures. Par exemple, nous dirons qu'une propriété est une pure obligation si elle ne peut s'exprimer comme une propriété de safety ni de guarantee.

Illustration informelle de la construction de propriétés. Pour donner l'intuition de chaque classe et mettre en évidence les différences les caractérisant, nous pouvons examiner la Fig. 3.3. Pour la construction d'une classe donnée de la hiérarchie, nous considérons une propriété finitaire ψ. Ensuite, les différentes classes sont construites en considérant les séquences d'exécution vérifiant un certain "motif" sur l'appartenance à ψ. Une séquence appartient à une propriété de safety construite à partir de ψ si tous ses préfixes appartiennent à ψ. Pour les séquences appartenant à des propriétés de guarantee, il faut que ces séquences aient au moins un préfixe appartenant à ψ. Pour les séquences appartenant à des propriétés de response, les préfixes doivent régulièrement ("infiniment souvent") appartenir à ψ. Pour les propriétés de persistence, les préfixes doivent tous appartenir à ψ à partir d'un certain rang.

L'exemple suivant introduit informellement des propriétés pour chaque classe mentionnée ci-dessus. Dans l'Exemple 3 page 41, nous formalisons ces propriétés informelles comme des r-propriétés.

Exemple 1 (Propriétés informelles pour les classes de base) Considérons un système d'exploitation dans lequel une opération donnée *op* est autorisée uniquement lorsqu'une autorisation d'accès *auth* a été accordée auparavant. Le système d'exploitation est muni de trois primitives liées à l'authentification de l'utilisateur : *r_auth* (requête d'authentification), *g_auth* (granted authentication, authentification accordée), *d_auth* (denying authentication, authentification refusée). La déconnection de l'utilisateur est symbolisée par l'événement *disco* (disconnection, déconnection). Le log d'événement dans le système est représenté par l'événement *log* et l'arrêt du système par l'événement *end*. Le vocabulaire global pour cet exemple est donc : $\Sigma = \{op, r_auth, g_auth, d_auth, disco, log, end\}$. Ensuite,

- la propriété Π_1 déclarant que "chaque occurrence de l'opération *op* doit être précédée par au moins une occurrence distincte de *g_auth*" est une propriété de *safety* ; le vocabulaire de cette propriété est donc : $\Sigma = \{op, g_auth\}$;
- la propriété Π_2 déclarant que "lors de cette session, l'utilisateur doit réaliser une demande d'authentification (*r_auth*) qui doit être ensuite autorisée (*g_auth*) ou refusée (*d_auth*)" est une propriété de *guarantee* ; le vocabulaire pour cette propriété est donc : $\Sigma = \{r_auth, g_auth, d_auth\}$;
- la propriété Π_3 déclarant que "le système ne doit pas s'arrêter de s'exécuter, à moins qu'un événement *d_auth* apparaisse, ensuite l'utilisateur doit être déconnecté (*disco*) et le système doit s'arrêter (*end*)" est une propriété d'*obligation* ; le vocabulaire pour cette propriété est donc : $\Sigma = \{d_auth, end, disco\}$;
- la propriété Π_4 déclarant que "chaque occurrence de *r_auth* doit être d'abord écrite dans un log (*log*) et ensuite acquittée soit avec un *g_auth* ou un *d_auth* sans aucune occurrence de *op* pendant ce temps" est une propriété de *response* ; le vocabulaire pour cette propriété est donc : $\Sigma = \{r_auth, log, g_auth, d_auth, op\}$;
- la propriété Π_5 déclarant que "après un *d_auth*, une utilisation (interdite) de l'opération *op* doit entraîner, à un moment, que tout appel à *r_auth* doit toujours résulter en une réponse *d_auth*" est une propriété de *persistence* ; le vocabulaire pour cette propriété est donc : $\Sigma = \{r_auth, d_auth, op\}$.

3.3 r-propriétés : Runtime properties

Runtime properties. Dans cette thèse nous souhaitons valider des propriétés lors de l'exécution d'un système. Comme dit précédemment nous nous intéressons aux séquences finies et infinies (qu'un programme peut potentiellement produire). Les propriétés traitant de l'exécution d'un programme doivent exprimer la satisfaction des séquences finies et infinies de manière uniforme. Ainsi nous introduisons les r-propriétés (runtime properties) comme des couples constitués d'une propriété finitaire et d'une propriété infinitaire.

DÉFINITION 8 (r-PROPRIÉTÉS). Une r-propriété est un couple $(\phi, \varphi) \subseteq \Sigma^* \times \Sigma^\omega$. La propriété ϕ est appelée la *partie finitaire* de la r-propriété, tandis que φ est appelée la *partie infinitaire* de la propriété.

Intuitivement, la propriété finitaire ϕ représente le comportement désiré du point de vue des séquences d'exécution finies, alors que la propriété infinitaire φ est le comportement désiré pour les séquences infinies. La définition de la négation d'une r-propriété suit la définition de la négation des propriétés finitaires et infinitaires. Pour une r-propriété (ϕ, φ), nous définissons $\overline{(\phi, \varphi)}$ comme étant $(\overline{\phi}, \overline{\varphi})$. Les combinaisons booléennes de r-propriétés sont définies de manière naturelle. Pour $* \in \{\cup, \cap\}$, $(\phi_1, \varphi_1) * (\phi_2, \varphi_2) = (\phi_1 * \phi_2, \varphi_1 * \varphi_2)$. Considérant une séquence d'exécution $\sigma \in Exec(\mathcal{P}_\Sigma)$, nous disons que σ satisfait (ϕ, φ) lorsque $(\sigma \in \Sigma^* \wedge \phi(\sigma)) \vee (\sigma \in \Sigma^\omega \wedge \varphi(\sigma))$. Pour une r-propriété $\Pi = (\phi, \varphi)$, nous notons $\Pi(\sigma)$ (resp. $\neg\Pi(\sigma)$) lorsque σ satisfait (resp. ne satisfait pas) (ϕ, φ). Pour une r-propriété $\Pi = (\phi, \varphi)$, un ensemble de séquences finies $E \subseteq \Sigma^*$ (resp. infinies $W \subseteq \Sigma^\omega$), nous notons $\Pi \cap E$ pour $\phi \cap E$ (resp. $\Pi \cap W$ pour $\varphi \cap W$).

Évaluation des r-propriétés. La monitorabilité, l'enforçabilité, la testabilité, et la synthèse de moniteur sont basées sur l'évaluation des r-propriétés. Évaluer une séquence d'exécution σ par rapport à une r-propriété consiste à produire un verdict vis-à-vis de la satisfaction de la séquence courante de σ ou des satisfactions futures des σ-continuations possibles (*i.e.*, $\{\sigma' \in \Sigma^\infty \mid \sigma \prec \sigma'\}$). Les verdicts considérés ici ne sont pas les valeurs booléennes usuelles : ce sont des valeurs prises dans un domaine de vérité.

Un domaine de vérité est un treillis, *i.e.*,un ensemble partiellement ordonné avec une borne supérieure et inférieure. Considérant une domaine de vérité $\mathbb{B}$, une r-propriété Π et une séquence d'exécution σ, l'évaluation de $\sigma \in \Sigma^*$ par rapport à Π dans $\mathbb{B}$, notée $[\![\Pi]\!]_{\mathbb{B}}(\sigma)$, est un élément de $\mathbb{B}$ dépendant de $\Pi(\sigma)$ et de la satisfaction des σ-continuations par rapport à Π.

Nous verrons que l'ensemble des propriétés monitorables et enforçables (Chapitre 4, Section 4.2 et 4.3) dépendra directement du domaine de vérité considéré et de la fonction d'évaluation choisie.

3.4 La vue langage des r-propriétés

Dans la vue langage de la hiérarchie, les r-propriétés sont des couples d'ensembles de séquences d'exécution. Nous commençons par définir formellement les différentes classes de la hiérarchie. Ces classes diffèrent par la manière dont, étant donnée une propriété finitaire, est construit un couple constitué d'une propriété finitaire et d'une propriété infinitaire. Nous étudions ensuite la vue langage des r-propriétés.

3.4.1 Construction de propriétés finitaires et infinitaires

La vue langage de la classification *Safety-Progress* est basée sur la construction de propriétés infinitaires et finitaires à partir de propriétés finitaires. Elle repose sur l'utilisation de quatre opérateurs A, E, R, P (pour la construction de propriétés infinitaires) et quatre opérateurs A_f, E_f, R_f, P_f (pour la construction de propriétés finitaires) s'appliquant à des propriétés finitaires. Dans la classification originale de Manna et Pnueli, les opérateurs A, E, R, P, A_f, E_f ont été introduits (sans définition formelle). Dans ce chapitre, nous définissons également les opérateurs R_f et P_f et donnons une définition formelle pour tous les opérateurs.

Dans les définitions suivantes, ψ est une propriété finitaire définie sur l'alphabet Σ.

Définition 9 (Opérateurs A, E, R, P). Pour $\psi \subseteq \Sigma^*$, les opérateurs A, E, R, P sont définis comme suit :
- $A(\psi) = \{\sigma \in \Sigma^\omega \mid \forall \sigma' \in \Sigma^* : \sigma' \prec \sigma \Rightarrow \psi(\sigma')\}$.
- $E(\psi) = \{\sigma \in \Sigma^\omega \mid \exists \sigma' \in \Sigma^* : \sigma' \prec \sigma \wedge \psi(\sigma')\}$.
- $R(\psi) = \{\sigma \in \Sigma^\omega \mid \forall \sigma' \in \Sigma^*, \exists \sigma'' \in \Sigma^* : \sigma' \prec \sigma'' \prec \sigma \wedge \psi(\sigma'')\}$.
- $P(\psi) = \{\sigma \in \Sigma^\omega \mid \exists \sigma' \in \Sigma^*, \forall \sigma'' \in \Sigma^* : \sigma' \prec \sigma'' \prec \sigma \Rightarrow \psi(\sigma'')\}$.

$A(\psi)$ consiste en tous les mots infinis σ tels que *tous* les préfixes de σ appartiennent à ψ. $E(\psi)$ consiste en tous les mots infinis σ tels que *certains* préfixes de σ appartiennent à ψ. $R(\psi)$ consiste en tous les mots infinis σ tels que *un nombre infinis* de préfixes de σ appartiennent à ψ. $P(\psi)$ consiste en tous les mots infinis σ tels que *tous sauf un nombre fini* de préfixes de σ appartiennent à ψ.

Il existe un lien entre les différents opérateurs introduits. Les opérateurs A et E sont duaux, et il en est de même des opérateurs R et P. C'est-à-dire :

$$\overline{A(\psi)} = E(\overline{\psi})$$
$$\overline{R(\psi)} = P(\overline{\psi})$$

où la complémentation est prise par rapport à Σ^+ pour ψ, et par rapport à Σ^ω pour $A(\psi)$ et $R(\psi)$.

Construction de propriétés finitaires. Similairement, les opérateurs A_f, E_f, R_f, P_f construisent les propriétés finitaires à partir d'autres propriétés finitaires.

Définition 10 (Opérateurs A_f, E_f, R_f, P_f). Pour $\psi \subseteq \Sigma^*$, les opérateurs A_f, E_f, R_f, P_f sont définis comme suit :
- $A_f(\psi) = \{\sigma \in \Sigma^* \mid \forall \sigma' \in \Sigma^* : \sigma' \preceq \sigma \Rightarrow \psi(\sigma')\}$.
- $E_f(\psi) = \{\sigma \in \Sigma^* \mid \exists \sigma' \in \Sigma^* : \sigma' \preceq \sigma \wedge \psi(\sigma')\}$.
- $R_f(\psi) = \{\sigma \in \Sigma^* \mid \psi(\sigma) \wedge \exists \sigma' \in \Sigma^* : \sigma \prec \sigma' \wedge \psi(\sigma')\}$.
- $P_f(\psi) = \{\sigma \in \Sigma^* \mid \psi(\sigma) \wedge \exists \sigma' \in \Sigma^*, \exists \mu \in \Sigma^+ : (\sigma \preceq \sigma' \ \wedge \ (\sigma' \cdot \mu^* \cdot pref(\mu)) \subseteq \psi)\}$.

$A_f(\psi)$ consiste en tous les mots finis σ tels que *tous* les préfixes de σ appartiennent à ψ. L'opérateur A_f peut se voir comme un "filtre" "selectionnant" le sous-language prefixe-clos de ψ. $E_f(\psi)$ consiste en tous les mots finis σ tels que *quelques* préfixes (au moins un) de σ appartiennent à ψ. L'opérateur E_f produit la fermeture par continuation de ψ. $R_f(\psi)$ consiste en tous les mots finis σ tels que $\psi(\sigma)$ et il existe une continuation σ' de σ qui appartient aussi à ψ. L'opérateur R_f "selectionne" les séquences de ψ qui permettent de potentiellement avoir ψ régulièrement. $P_f(\psi)$ consiste en tous les mots finis σ appartenant à ψ tels qu'il existe une continuation σ' de σ à partir de laquelle on peut trouver une extension (aussi grande que l'on veut, de la forme $\sigma' \cdot \mu^*$) dont tous les préfixes plus grand que σ' appartiennent à ψ. L'opérateur P_f "selectionne" les séquences de ψ qui permettent de satisfaire ψ de manière persistente ; ce sont les séquences σ de ψ qui possèdent une extension σ' pour laquelle on peut trouver un mot non vide μ telle que l'on puisse obtenir des mots de ψ en concaténant un nombre fini de fois μ plus l'un de ses préfixes à σ'.

Remarque 1 Nous pouvons observer que :
- $A_f(\psi) = \psi$ si ψ est préfixe-clos et $\emptyset$ sinon.
- $E_f(\psi) = \psi \cdot \Sigma^*$.
- $R_f(\psi) \subseteq \psi$.
- $P_f(\psi) \subseteq \psi$. *

Se basant sur ces opérateurs, chaque classe peut être définie depuis la vue langage.

Définition 11 (r-propriétés des classes de base). Une r-propriété $\Pi = (\phi, \varphi)$ est définie comme :
- Une r-propriété de *safety* si $\Pi = (A_f(\psi), A(\psi))$ pour une propriété finitaire ψ. C'est-à-dire, tous les préfixes d'un mot fini $\sigma \in \phi$ ou d'un mot infini $\sigma \in \varphi$ appartiennent à ψ.
- Une r-propriété de *guarantee* si $\Pi = (E_f(\psi), E(\psi))$ pour une propriété finitaire ψ. C'est-à-dire, chaque mot fini $\sigma \in \phi$ ou infini $\sigma \in \varphi$ est garanti d'avoir (au moins) un préfixe appartenant à ψ.
- Une r-propriété de *response* si $\Pi = (R_f(\psi), R(\psi))$ pour une propriété finitaire ψ. C'est-à-dire, chaque mot infini $\sigma \in \varphi$ a de façon récurrente (un nombre infini) de préfixes appartenant à ψ.
- Une r-propriété de *persistence* si $\Pi = (P_f(\psi), P(\psi))$ pour une propriété finitaire ψ. C'est-à-dire, chaque mot infini $\sigma \in \varphi$ a de façon persistante (continuement à partir d'un certain point) ses préfixes appartenant à ψ.

Dans tous les cas, nous disons que Π est construite à partir de ψ. En outre, les r-propriétés d'obligation (resp. de réactivité) sont obtenues à partir de combinaisons booléennes de r-propriétés de safety et guarantee (resp. response et persistence).

Étant donné un ensemble d'événements Σ, nous notons $\mathsf{Safety}(\Sigma)$ (resp. $\mathsf{Guarantee}(\Sigma)$, $\mathsf{Obligation}(\Sigma)$, $\mathsf{Response}(\Sigma)$, $\mathsf{Persistence}(\Sigma)$, $\mathsf{Reactivity}(\Sigma)$) l'ensemble des r-propriétés de safety (resp. guarantee, obligation, response, persistence, réactivité) définies sur Σ.

Intuitivement, cette interprétation indique qu'une propriété de safety (resp. guarantee, response, persistence) exige qu'une "bonne chose" arrive tout le temps (resp. au moins une fois, infiniment souvent, continuement à partir d'une certain point) ; où ψ est le langage représentant l'ensemble des "bonnes choses".

Nous exposons quelques conséquences directes des définitions des r-propriétés de safety et guarantee.

Propriété 1 (Quelques faits sur les r-propriétés) : *Considérons une r-propriété $\Pi = (\phi, \varphi)$, définie sur un alphabet Σ, construite à partir d'une propriété finitaire ψ, nous avons les faits suivants :*
- *(i) Π est une r-propriété de safety ssi tous les préfixes d'une séquence appartenant à Π appartiennent également à Π, i.e., $\forall \sigma \in \Sigma^\infty : \Pi(\sigma) \Rightarrow (\forall \sigma' \in \Sigma^* : \sigma' \prec \sigma \Rightarrow \Pi(\sigma'))$.*
- *(ii) Π est une r-propriété de safety ssi toutes les continuations d'une séquence finie n'appartenant pas à Π n'appartiennent pas à Π non plus. C'est-à-dire, $\neg\Pi(\sigma) \Rightarrow \forall \mu \in \Sigma^\infty : \neg\Pi(\sigma \cdot \mu)$.*
- *(iii) Π est une r-propriété de guarantee ssi toutes les continuations d'une séquence finie appartenant à Π appartiennent également à Π. C'est-à-dire que $\forall \sigma \in \Sigma^* : \Pi(\sigma) \Rightarrow \forall \sigma' \in \Sigma^\infty : \Pi(\sigma \cdot \sigma')$.* ◇

Preuve : Nous prouvons chacun des trois faits successivement.

- Pour (i) :
  (⇒) Considérons $\sigma \in \Sigma^\infty$. Comme Π est une safety, nous savons qu'elle peut s'écrire $(A_f(\psi), A(\psi))$ pour une propriété finitaire $\psi \subset \Sigma^*$. De plus, nous avons soit $\phi(\sigma)$ ou bien $\varphi(\sigma)$, *i.e.*, $A_f(\psi)(\sigma)$ ou bien $A(\psi)(\sigma)$, tous les préfixes σ' de σ appartiennent à ψ. Nécessairement, tous les préfixes σ'' de σ' appartiennent aussi à ψ, c'est-à-dire $\psi(\sigma'')$. Par définition, cela signifie que $\sigma' \in A_f(\psi)$, *i.e.*, $\phi(\sigma')$ et $\Pi(\sigma')$.
  (⇐) Soit Π une r-propriété telle que tous les préfixes d'une séquence appartenant à Π appartiennent également à Π. Pour montrer que Π est une r-propriété de safety, il suffit de trouver $\psi \subseteq \Sigma^*$ telle que Π puisse s'exprimer $(A_f(\psi), A(\psi))$. Pour ψ il nous suffit de prendre

$$\psi = \{\sigma \in \Sigma^* \mid \Pi(\sigma)\} \cup \bigcup_{\{\sigma \in \Sigma^\omega \mid \Pi(\sigma)\}} pref(\sigma)$$

- Pour (ii). Π peut s'écrire $(A_f(\psi), A(\psi))$. Soit $\sigma \in \Sigma^*$, une séquence finie n'appartenant pas à Π. Ce qui veut dire que $\sigma \notin A_f(\psi)$. Ainsi, σ n'a pas tous ses préfixes dans ψ, *i.e.*, $\exists \sigma' \in \Sigma^* : \sigma' \preceq \sigma \wedge \neg\psi(\sigma')$. Soit $\mu \in \Sigma^\infty$. Si $\mu \in \Sigma^*$, alors on a forcément $\neg A_f(\sigma \cdot \mu)$ car $\sigma' \prec \sigma \preceq \sigma \cdot \mu$, σ' est un préfixe de $\sigma \cdot \mu$ qui n'appartient pas à ψ. Il s'en suit que $\neg\Pi(\sigma \cdot \mu)$. Si $\mu \in \Sigma^\omega$, de la même manière, nous avons que $\neg\Pi(\sigma \cdot \mu)$. Dans les deux cas, nous avons $\neg\Pi(\sigma \cdot \mu)$.
- Pour (iii) :
  (⇒) Soit $\sigma \in \Sigma^*$ telle que $\Pi(\sigma)$. Comme Π est une r-propriété de guarantee, nous savons que σ a au moins un préfixe $\sigma_0 \preceq \sigma$ appartenant à ψ : $\sigma \in E_f(\psi)$. Ensuite, toute continuation σ construite en utilisant n'importe quelle séquence finie ou infinie σ' a au moins le même préfixe appartenant à ψ. Si $\sigma' \in \Sigma^*$, nous avons $\sigma_0 \preceq \sigma \preceq \sigma \cdot \sigma'$ et $\sigma \cdot \sigma' \in E_f(\psi)$. Si $\sigma' \in \Sigma^\omega$, nous avons $\sigma_0 \preceq \sigma \prec \sigma \cdot \sigma'$ et $\sigma \cdot \sigma' \in E(\psi)$.
  (⇐) Soit Π une r-propriété telle que toutes les continuations d'une séquence finie appartenant à Π appartiennent également à Π. Pour montrer que Π est une r-propriété de guarantee, il nous suffit de trouver $\psi \subseteq \Sigma^*$ telle que Π puisse s'écrire $(E_f(\psi), E(\psi))$. Pour ψ, il nous suffit de prendre $\psi = \{\sigma \in \Sigma^* \mid \Pi(\sigma)\}$.

■

Nous illustrons dans l'exemple suivant la construction de propriétés finitaires et infinitaires depuis des propriétés finitaires pour chacun des quatre types d'opérateurs.

Exemple 2 (Construction de propriétés finitaires et infinitaires) Nous utilisons les expressions régulières pour définir les r-propriétés considérées.

- Pour la propriété finitaire $\psi = \epsilon + a^+ \cdot b^*$, $A_f(\psi) = \epsilon + a^+ \cdot b^*$, $A(\psi) = a^\omega + a^+ \cdot b^\omega$, $(A_f(\psi), A(\psi))$ est une r-propriété de safety. Ce langage contient tous les mots qui ont soit uniquement des occurrences de a ou bien un nombre fini d'occurrences de a (au moins une) suivie(s) uniquement d'occurrences de b.
- Pour la propriété finitaire $\psi = a^+ \cdot b^*$, $E_f(\psi) = a^+ \cdot b^* \cdot \Sigma^*$, $E(\psi) = a^+ \cdot b^* \cdot \Sigma^\omega$, $(E_f(\psi), E(\psi))$ est une r-propriété de guarantee.
- Pour la propriété finitaire $\psi = \Sigma^* \cdot b$, $R_f(\psi) = (\Sigma^* \cdot b)^+$, $R(\psi) = (\Sigma^* \cdot b)^\omega$, $(R_f(\psi), R(\psi))$ est une r-propriété de response. Ce langage contient tous les mots qui ont un nombre infini d'occurrences de b.
- Pour la propriété finitaire $\psi = \Sigma^* \cdot b$, $P_f(\psi) = \Sigma^* \cdot b^+$, $P(\psi) = \Sigma^* \cdot b^\omega$, $(P_f(\psi), P(\psi))$ est une r-propriété de persistence. Ce langage contient tous les mots qui, à partir d'un certain rang, contiennent uniquement des occurrences de b.

Exemple 3 (r-propriétés) Les propriétés de l'Exemple 1 peuvent être formalisées en r-propriétés comme suit.

- la r-propriété Π_1 peut être exprimée comme une r-propriété de *safety* construite à partir de $\psi_1 = (g_auth^+ \cdot op)^* \cdot g_auth^*$ avec $\Sigma = \{g_auth, op\}$
- la r-propriété Π_2 peut être exprimée comme une r-propriété de *guarantee* construite à partir de $\psi_2 = (\Sigma \setminus \{r_auth\})^* \cdot r_auth \cdot (\Sigma \setminus \{g_auth, d_auth\})^* \cdot (g_auth + d_auth)$ avec $\Sigma = \{g_auth, d_auth, r_auth\}$
- la r-propriété Π_3 peut être exprimée comme une r-propriété d'*obligation* construite à partir de $\psi_3 = (\Sigma \setminus \{d_auth, end\})^*$, et $\psi'_3 = (\Sigma \setminus \{d_auth, end\})^* \cdot d_auth \cdot (\Sigma \setminus \{disco\})^* \cdot disco$ avec $\Sigma = \{end, d_auth, disco\}$; ensuite Π_3 est $(A_f(\psi_3), A(\psi_3)) \vee (E_f(\psi'_3), E(\psi'_3))$.
- la r-propriété Π_4 peut être exprimée comme une r-propriété de *response* construite à partir de $\psi_4 = r_auth \cdot log^+ \cdot (d_auth + g_auth)$ avec $\Sigma = \{g_auth, d_auth, op, log\}$

– la r-propriété Π_5 peut être exprimée comme une r-propriété de *persistence* construite à partir de $\psi_5 = (d_auth \cdot g_auth)^*.d_auth \cdot (\Sigma \setminus \{r_auth, d_auth\})^* \cdot op \cdot \left(g_auth^* + (r_auth \cdot (\Sigma \setminus \{d_auth\})^+)\right)^*$ avec $\Sigma = \{g_auth, d_auth, op\}$

3.4.2 La hiérarchie des langages

Nous étudions maintenant la relation hiérarchique entre les différentes classes de langages définis précédemment. Notons tout d'abord que, en conséquence de la dualité de certains opérateurs de constructions de langages, les classes de bases contruites à partir d'opérateurs duaux sont duales également. Ce qui peut être exprimé comme suit :

– Si Π est une r-propriété de safety, alors $\overline{\Pi}$ est une r-propriété de guarantee.
– Si Π est une r-propriété de response, alors $\overline{\Pi}$ est une r-propriété de persistence.

Nous étudions maintenant les propriétés de chaque classe en étudiant les liens avec les autres classes. Nous verrons que les quatre classes de bases sont fermées par les opérations booléennes positives.

Fermeture des classes de base

Les quatre classes de base sont fermées par les opérations booléennes positives, *i.e.*, union, et intersection.

La classe des r-propriétés de guarantee. Soient $(E_f(\psi_1), E(\psi_1))$ et $(E_f(\psi_2), E(\psi_2))$ deux r-propriétés de guarantee, leur union et leur intersection sont également des r-propriétés de guarantee. Ce qui est dû aux égalités suivantes (prouvées dans l'Annexe A.1) :

$$E(\psi_1) \cup E(\psi_2) = E(\psi_1 \cup \psi_2) \qquad E_f(\psi_1) \cup E_f(\psi_2) = E_f(\psi_1 \cup \psi_2)$$
$$E(\psi_1) \cap E(\psi_2) = E(E_f(\psi_1) \cap E_f(\psi_2)) \qquad E_f(\psi_1) \cap E_f(\psi_2) = E_f(E_f(\psi_1) \cap E_f(\psi_2))$$

Ce qui est basé sur le fait que $\forall \psi \subset \Sigma^+, E(\psi) = \psi \cdot \Sigma^\omega$, et sur les deux égalités suivantes :

$$\psi_1 \cdot \Sigma^\omega \cup \psi_2 \cdot \Sigma^\omega = (\psi_1 \cup \psi_2) \cdot \Sigma^\omega$$
$$\psi_1 \cdot \Sigma^\omega \cap \psi_2 \cdot \Sigma^\omega = (\psi_1 \cdot \Sigma^* \cap \psi_2 \cdot \Sigma^*) \cdot \Sigma^\omega$$

Nous avons alors :

$$\begin{aligned}(E_f(\psi_1), E(\psi_1)) \cap (E_f(\psi_2), E(\psi_2)) &= (E_f(\psi_1) \cap E_f(\psi_2), E(\psi_1) \cap E(\psi_2)) \\ &= \left(E_f(E_f(\psi_1) \cap E_f(\psi_2)), E(E_f(\psi_1) \cap E_f(\psi_2))\right)\end{aligned}$$

La classe des r-propriétés de safety. La fermeture de la classe des propriétés de safety est établie par les égalités ci-dessous qui sont obtenues par dualité avec les égalités mentionnées pour la classe des propriétés de guarantee.

$$A(\psi_1) \cap A(\psi_2) = A(\psi_1 \cap \psi_2) \qquad A_f(\psi_1) \cap A_f(\psi_2) = A_f(\psi_1 \cap \psi_2)$$
$$A(\psi_1) \cup A(\psi_2) = A(A_f(\psi_1) \cup A_f(\psi_2)) \qquad A_f(\psi_1) \cup A_f(\psi_2) = A_f(A_f(\psi_1) \cup A_f(\psi_2))$$

La classe des r-propriétés de response. Définissons tout d'abord la notion d'extension minimale d'un langage par un autre.

Définition 12 (Extension minimale d'un langage par un autre). L'extension minimale du langage ψ_2 par ψ_1, notée $minex(\psi_1, \psi_2)$, est l'ensemble des mots $\sigma_2 \in \psi_2$ *t.q.*

– Il existe un mot $\sigma_1 \in \psi_1$, *t.q.* $\sigma_1 \prec \sigma_2$, *i.e.*, σ_2 est une continuation de σ_1 dans ψ_2
– Il n'y a pas de $\sigma'_2 \in \psi_2$, *t.q.* $\sigma_1 \prec \sigma'_2 \prec \sigma_2$, *i.e.*, σ_2 est une continuation minimale de σ_1 dans ψ_2

Nous avons donc les égalités suivantes établissant la fermeture des r-propriétés de response :

$$R(\psi_1) \cup R(\psi_1) = R(\psi_1 \cup \psi_2)$$
$$R(\psi_1) \cap R(\psi_2) = R(minex(\psi_1, \psi_2))$$

La classe des r-propriétés de persistence. La fermeture de la classe des r-propriétés de persistence est établie par les égalités suivantes :

$$P(\psi_1) \cap P(\psi_2) = P(\psi_1 \cap \psi_2)$$
$$P(\psi_1) \cup P(\psi_2) = \overline{P(minex(\overline{\psi_1}, \overline{\psi_2})}$$

Relations d'inclusion parmi les classes

Comme nous l'avons vu informellement dans la Section 3.2, il existe une hiérarchie entre les classes dans la classification *Safety-Progress*. Ici, nous revenons sur l'inclusion des deux classes de base les plus basses de la hiérarchie (*i.e.*, les safety et les guarantee) dans les deux classes de bases response et persistence. Les résultats pour les propriétés finitaires sont une adaptation directe des résultats donnés pour les propriétés finitaires dans [CMP92a].

La classe des r-propriétés de response contient la classe des safety et guarantee :

$$A(\psi) = R(A_f(\psi)) \qquad E(\psi) = R(E_f(\psi))$$
$$A_f(\psi) = R_f(A_f(\psi)) \qquad E_f(\psi) = R_f(E_f(\psi))$$

La classe des r-propriétés de persistence contient la classe des safety et guarantee :

$$A(\psi) = P(A_f(\psi)) \qquad E(\psi) = P(E_f(\psi))$$
$$A_f(\psi) = P_f(A_f(\psi)) \qquad E_f(\psi) = P_f(E_f(\psi))$$

Caractérisation des classes de safety et guarantee

Les classes de bases ont été définies de manière constructive. Il peut être intéressant d'avoir une caractérisation directe des propriétés. Les deux propriétés suivantes donnent une caractérisation des r-propriétés de safety et de guarantee. La preuve de ces propriétés est une adaptation directe de la preuve donnée dans [CMP92a].

Propriété 2 (Caractérisation des r-propriétés de safety) : *Une r-propriété Π est une r-propriété de safety ssi*

$$\Pi = (A_f(Pref(\Pi)), A(Pref(\Pi)))$$

◊

Propriété 3 (Caractérisation des r-propriétés de guarantee) : *Une r-propriété Π est une r-propriété de guarantee ssi*

$$\Pi = (E_f(\overline{Pref(\overline{\Pi})}), E(\overline{Pref(\overline{\Pi})}))$$

◊

Les classes composées

La classe des r-propriétés d'obligation. La classe des r-propriétés d'obligation peut être définie (de trois manières équivalentes) comme la classe obtenable par :

- combinaisons booléennes non restreinte de r-propriétés de safety,
- combinaisons booléennes non restreinte de r-propriétés de guarantee,
- combinaisons booléennes positive de r-propriétés de safety et guarantee.

Une conséquence immédiate est que les classes des propriétés de safety et guarantee sont des sous classes de la classe des propriétés d'obligation. De plus l'inclusion est stricte comme le montre la propriété d'obligation suivante qui n'est ni une r-propriété de safety ni une r-propriété de guarantee : $(a^* + \Sigma^* \cdot b \cdot \Sigma^*, a^\omega + \Sigma^* \cdot b \cdot \Sigma^\omega)$

Le lemme suivant (inspiré de [CMP92a]) fournit une décomposition de chaque r-propriété d'obligation en forme normale.

Lemme α (Formes normales des r-propriétés d'obligation) : *Toute r-propriété d'obligation peut être exprimée comme l'intersection*

$$\bigcap_{i=1}^{n} (\text{Safety}_i \cup \text{Guarantee}_i)$$

pour un certain $n > 0$, où Safety_i et $\mathsf{Guarantee}_i$ sont respectivement des r-propriétés de safety et guarantee. Nous nommons cette présentation comme la forme normale conjonctive des r-propriétés d'obligation.

De manière symétrique, toute r-propriété d'obligation peut être représentée comme l'union

$$\bigcup_{i=1}^{n}(\mathsf{Safety}_i \cap \mathsf{Guarantee}_i)$$

Cette représentation est nommée la forme normale disjonctive des r-propriétés d'obligation.

PREUVE : Considérant la Définition 11 déclarant que les r-propriétés d'obligation sont des combinaisons booléennes positives de r-propriétés de safety et guarantee, nous pouvons exprimer toute r-propriété d'obligation sous une forme normale conjonctive

$$\bigcap_{i=1}^{n}(\Pi_0^i \cup \cdots \cup \Pi_{k_i-1}^i \cup \cdots \cup \Pi_{m_i-1}^i)$$

où $\Pi_0^i, \ldots, \Pi_{k_i-1}^i$ sont des r-propriétés de safety, et $\Pi_{k_i}^i, \ldots, \Pi_{m_i-1}^i$ sont des r-propriétés de guarantee. En utilisant la fermeture des classes de safety et guarantee sous l'opérateur d'union, nous pouvons exprimer la r-propriété sous la forme mentionnée plus haut. ■

Lorsqu'une r-propriété Π est exprimée comme $\cap_{i=1}^{k}(\mathsf{Safety}_i \cup \mathsf{Guarantee}_i)$, Π est dite être une r-propriété de k-obligation. L'ensemble des r-propriétés de k-obligation sur l'alphabet Σ ($k \geq 1$) est dénotée $\mathit{Obligation}_k(\Sigma)$. Des définitions et propriétés similaires sont valables aussi pour les r-propriétés de réactivité qui sont exprimées comme la combinaison de r-propriétés de response et persistence.

3.5 La vue logique temporelle des r-propriétés

Nous allons maintenant étudier la vue logique de la classification Safety-Progress des r-propriétés. Dans cette vue les r-propriétés sont décrites par des formules de logique linéaire temporelle. Nous présentons une introduction brève du langage de la logique temporelle. Une introduction complète peut être trouvée dans [PM91].

La logique temporelle LTL (Linear Temporal Logic) a été introduite en vérification par Amir Pnueli dans [Pnu77]. Elle permet de spécifier l'évolution souhaitée de l'état du système ou encore l'ordre d'apparition de certains évènements.

Notons que l'ensemble des r-propriétés spécifiables en LTL représente un sous ensemble strict des propriétés ω-régulières [Wol81]. En effet, la logique LTL a le même pouvoir expressif que les automates d'états finis [5] sans compteurs [MP71, Tho97], ou bien encore les langages réguliers sans étoile. Voir [Zuc86] pour plus de détails. Nous donnons la syntaxe et la sémantique de cette logique et étudions ensuite la hiearchie des formules de logique temporelle.

3.5.1 Syntaxe et sémantique

Syntaxe. Les formules de LTL sont construites à partir de propositions atomiques [6] $p \in \mathit{Prop}$ auxquelles s'appliquent les opérateur booléens $\neg, \vee, \wedge$, et des modalités temporelles de base : $\bigcirc$ (next), $\mathcal{U}$ (until), $\ominus$ (previous), $\mathcal{S}$ (since).

$$\mathit{LTL} \ni \Upsilon, \Upsilon' ::= p \mid \neg\Upsilon \mid \bigcirc\Upsilon \mid \ominus\Upsilon \mid \Upsilon\,\mathcal{U}\,\Upsilon' \mid \Upsilon\,\mathcal{S}\,\Upsilon'$$

Les modalités ou opérateurs $\bigcirc$ et $\mathcal{U}$ (resp. $\ominus$ et $\mathcal{S}$) sont dits être des modalités/opérateurs du futur (resp. du passé). Intuitivement, les formules de LTL se comprennent de la manière suivante. La formule $\neg\Upsilon$ est la négation de Υ, $\neg\Upsilon$ est satisfaite lorsque Υ ne l'est pas. $\Upsilon \vee \Upsilon'$ est la disjonction de Υ et Υ', et

5. Cela est à considérer au sens large : automates d'états finis pour les séquences finies et automates de Buchi pour les séquences infinies.

6. Nous considérons la définition de LTL sur les états. Dans ce cas, lorsqu'on souhaite faire le lien entre logique et automate ou langage, l'alphabet considéré est un sous ensemble de 2^{Prop}

est satisfaite dès que Υ ou Υ' l'est. La formule $\bigcirc\Upsilon$, se lit "next Υ", et indique que Υ doit être satisfaite dans l'état suivant. L'opérateur $\ominus$ est le symétrique dans le passé de l'opérateur $\bigcirc$. La formule $\ominus\Upsilon$, se lit "previous Υ", et est satisfaite lorsque Υ l'est à l'état précédent. La formule $\Upsilon\ \mathcal{U}\ \Upsilon'$ se lit "Υ until Υ' et signifie que Υ doit être satisfaite jusqu'à que Υ' le soit. L'opérateur $\mathcal{S}$ est le symétrique dans le passé de l'opérateur $\mathcal{U}$. La formule $\Upsilon\ \mathcal{S}\ \Upsilon'$ se lit "Υ since Υ'", et signifie qu'il y a eu un moment où Υ' était satisfaite dans le passé, et depuis Υ est satisfaite.

Il est également possible de définir des fragments de la logique en ne considérant que des modalités du futur ou du passé. Ces restrictions syntaxiques donnent lieu à de nouvelles logiques. Notons toutefois que ces restrictions syntaxiques n'enlèvent pas de pouvoir expressif à la logique [Kam68]. Cependant l'utilisation des opérateurs du passé apporte de la concision aux formules [Mar03].

Sémantique. Après avoir introduit la syntaxe de LTL, nous introduisons sa sémantique [PM91, Pnu77]. Notons que cette sémantique peut se définir de manière inductive, contrairement aux logiques LTL_3 [BLS06], LTL_-^+ [EFH+03], et *RV-LTL* [BLS07a].

Définition 13 (Sémantique des r-propriétés en vision logique). Soient $\sigma \in \Sigma^\infty$ un mot fini ou infini, $i < |\sigma|$ un entier naturel strictement inférieur à la longueur de σ, Υ et Υ' deux formules LTL représentant des r-propriétés, et p une proposition atomique de *Prop*. La sémantique des formules de LTL représentant des r-propriétés est donnée par la relation $\models$, définie inductivement (sur la syntaxe) comme suit :

$(\sigma, i) \models p$	*ssi*	$p \in \sigma_i$
$(\sigma, i) \models \neg\Upsilon$	*ssi*	$(\sigma, i) \not\models \Upsilon$
$(\sigma, i) \models \Upsilon \vee \Upsilon'$	*ssi*	$(\sigma, i) \models \Upsilon$ ou $(\sigma, i) \models \Upsilon'$
$(\sigma, i) \models \bigcirc\Upsilon$	*ssi*	$\lvert\sigma\rvert > i + 1$ et $(\sigma, i+1) \models \Upsilon$
$(\sigma, i) \models \Upsilon\ \mathcal{U}\ \Upsilon'$	*ssi*	$(\exists i \leq k < \lvert\sigma\rvert : (\sigma, k) \models \Upsilon') \wedge (\forall i \leq j < k : (\sigma, j) \models \Upsilon)$
$(\sigma, i) \models \ominus\Upsilon$	*ssi*	$i > 1$ et $(\sigma, i-1) \models \Upsilon$
$(\sigma, i) \models \Upsilon\ \mathcal{S}\ \Upsilon'$	*ssi*	$\exists k \leq i : (\sigma, k) \models \Upsilon' \wedge \forall k < j \leq i : (\sigma, j) \models \Upsilon$

Des opérateurs additionnels peuvent être définis comme suit :

$\tilde{\bigcirc}\Upsilon = \neg\bigcirc\neg\Upsilon$	(weak next, next faible)
$\diamond\ \Upsilon = \texttt{true}\ \mathcal{U}\ \Upsilon$	(eventually, inévitablement)
$\square\ \Upsilon = \neg\diamond\ \neg\Upsilon$	(always, toujours)
$\Upsilon\ \mathcal{W}\ \Upsilon' = \square\ \Upsilon \vee \Upsilon\ \mathcal{U}\ \Upsilon'$	(waiting for, until faible)
$\diamondminus\Upsilon = \texttt{true}\ \mathcal{S}\ \Upsilon$	(sometimes)
$\boxminus\ \Upsilon = \neg\diamondminus\neg\Upsilon$	(always in the past, toujours dans le passé)
$\Upsilon\ \mathcal{B}\ \Upsilon' = \boxminus\ \Upsilon \vee (\Upsilon\ \mathcal{S}\ \Upsilon')$	(back to (weak since))
$\tilde{\ominus}\ \Upsilon = \neg\ \ominus\ \neg\Upsilon$	(weak previous, précédemment faible)

L'opérateur de next faible est utilisé pour donner une sémantique aux séquences finies. En effet, $\tilde{\bigcirc}\Upsilon$ est équivalent à $|\sigma| > i \wedge (\sigma, i+1) \models \Upsilon \vee |\sigma| \leq i$.

Ainsi, $\tilde{\bigcirc}\Upsilon$ est vrai si soit la position i est la dernière position de la séquence σ ou alors il existe une position suivante et cette position satisfait Υ.

L'opérateur $\diamond$ est utilisé pour dénoter la satisfaction inévitable de la propriété à laquelle il est appliqué. En effet, la formule $\diamond\ \Upsilon$ est équivalente à $\Upsilon \vee \bigcirc\diamond\ \Upsilon$, elle est vraie si la séquence considérée satisfait Υ à la position courante ou bien elle satisfait $\diamond\ \Upsilon$ à la position suivante.

L'opérateur $\square$ est utilisé pour dénoter la satisfaction permanente d'une formule. En effet la formule $\square\ \Upsilon$ est équivalente à $\Upsilon \wedge \tilde{\bigcirc}(\square\ \Upsilon)$, elle est vraie si la séquence considérée satisfait Υ à la position courante et la position suivante satisfait $\square\ \Upsilon$ si cette position existe.

L'opérateur $\mathcal{W}$ est le until faible. Pour $\Upsilon\ \mathcal{W}\ \Upsilon'$ autorise que Υ' n'arrive jamais si Υ est toujours vraie.

L'opérateur $\diamondminus$ (resp. $\boxminus$) est utilisé pour dénoter la satisfaction inévitable (resp. permanente) dans le passé de la propriété à laquelle il est appliqué.

Définition 14 (Fonction Sat d'interprétation des formules). La fonction *Sat* interprète une formule exprimée en LTL en donnant l'ensemble de ses modèles, *i.e.*, la r-propriété qu'elle représente. La fonction $Sat : LTL \rightarrow 2^{\Sigma^*} \times 2^{\Sigma^\omega}$ est définie comme suit :

$$Sat(\Upsilon) = (\phi, \varphi)$$

où
- $\phi = \{\sigma \in \Sigma^* \mid \sigma \models \Upsilon\}$
- $\varphi = \{\sigma \in \Sigma^\omega \mid \sigma \models \Upsilon\}$

Deux formules de LTL Υ, Υ' sont dites équivalentes si elles ont le même modèle, *i.e.*, si $Sat(\Upsilon) = Sat(\Upsilon')$, ce qui est noté $\Upsilon \sim \Upsilon'$.

3.5.2 Hiérarchie des formules temporelles

Nous allons voir que la hiérarchie que nous avons identifiée dans la vue langage se retrouve dans la vue logique des r-propriétés. Rappelons que nous nous intéressons ici à un sous-ensemble des r-propriétés : l'ensemble qui est exprimable en logique temporelle. Les résultats suivants sont des adaptations au cas des r-propriétés des résultats donnés dans [CMP92a]. Les équivalences de formules présentées ci-après sont établies pour LTL sur les séquences infinies [CMP92a] et FLTL sur les séquences finies [Pnu77].

Dans la suite nous considérons une propriété finitaire $\psi \subseteq \Sigma^+$. La propriété finitaire définie par une formule du passé est l'ensemble des séquences qui satisfont p sur leur dernière position. Pour une formule du passé p, nous notons $esat(p)$ la propriété finitaire définie par p.

Exemple 4 (Propriété finitaire, formule du passé, et *esat*()) La propriété finitaire $(a+b)^* \cdot c$ peut être représentée par la formule du passé $c \wedge \boxminus (a \vee b)$ qui déclare que c doit être vrai et que $a \vee b$ était vrai dans toutes les positions précédentes.

Une r-propriété qui peut être spécifiée par une formule de safety (resp. guarantee, response, persistence) est dite être une propriété safety-spécifiable (resp. guarantee-spécifiable, response-spécifiable, persistence-spécifiable).

Formules de Safety

Nous définissons les formules de safety canoniques comme les formules s'exprimant sous la forme $\Box\, p$, où p est une formule du passé. Une telle formule exprime intuitivement que toutes les positions d'une séquence évaluée doivent satisfaire p.

Les formules de safety sont souvent utilisées pour exprimer l'invariance de l'état du système (*e.g.*, la variable x est toujours positive), ou encore des contraintes de précédences entre les événements du systèmes (*e.g.*, pour deux évènements e_1 et e_2 pouvant apparaître dans le système, l'évènement e_1 appairait toujours avant l'évènement e_2).

Fermeture des propriétés safety-spécifiables. La classe des r-propriétés expressibles par des formules de safety est fermée par les opérateurs booléens positifs *i.e.*,intersection, et union. Pour voir cela, nous présentons les équivalences suivantes pour la conjonction et la disjonction des formules de safety. Nous avons les équivalences de formules de safety suivantes :

$$(\Box\, p \wedge \Box\, q) \sim \Box\, (p \wedge q)$$
$$(\Box\, p \vee \Box\, q) \sim \Box\, (\boxminus\, p \vee \boxminus\, q)$$

La classe des propriétés safety-spécifiables n'est pas fermée par négation. En effet, la négation d'une formule de safety est une formule de guarantee. Ces deux classes sont *duales*, l'une peut être obtenue par la négation de l'autre.

Formules de Guarantee

Nous définissons les formules de guarantee canoniques comme les formules s'exprimant sous la forme

$$\Diamond\, p$$

où p est une formule du passé. De telles formules spécifient qu'au moins une position de la séquence considérée satisfait p.

Les formules de guarantee expriment qu'un (bon) évènement va avoir lieu inévitablement, *i.e.*, au moins une fois. Principalement, elles servent à exprimer l'assurance souhaitée quant à l'apparition d'un évènement donné sur le système (*e.g.*, la terminaison du programme ou d'un thread).

Fermeture des propriétés guarantee-spécifiables. Les propriétés de fermeture pour les r-propriétés spécifiables par des formules de guarantee s'obtiennent par dualité avec les propriétés de fermeture que nous avons pour les propriétés spécifiables par les formules de safety. La dualité entre les deux classes s'exprime dans la vue temporelle de la manière suivante :

$$\neg\Box\, p \sim \Diamond\, (\neg p)$$

De manière intuitive, cette équivalence de formule peut se comprendre par le fait que si il existe une position dans une séquence où la formule p n'est pas vérifiée, alors il n'est pas possible que toutes les positions de la séquence satisfassent p ; et inversement.

La classe des r-propriétés spécifiables par des formules de guarantee est donc fermée par les opérations booléennes positives. Nous avons de plus les équivalences suivantes :

$$\Diamond\, p \vee \Diamond\, q \sim \Diamond\, (p \vee q)$$
$$(\Diamond\, p \wedge \Diamond\, q) \sim \Diamond\, (\diamondminus p \wedge \diamondminus q)$$

Tout comme la classe des propriétés spécifiables par des formules de safety, la classe des propriétés guarantee-spécifiables n'est pas fermée par l'opération de complémentation. La négation d'une formule de guarantee est une formule de safety.

Formules d'obligation

Au dessus des propriétés safety-spécifiables et guarantee-spécifiables, nous trouvons la classe des propriétés obligation-spécifiables. En plus de contenir les deux précédentes classes, elle contient toutes les propriétés ne pouvant pas être spécifiées ni par une formule de safety ni par une formule de guarantee, mais par une combinaison booléenne (dénombrable) de formules de safety et guarantee. Le premier type de formules d'obligation sont les formules de simple obligation.

Une formule canonique simple d'obligation est de la forme :

$$\Box\, p \vee \Diamond\, q$$

où p et q sont des formules du passé. Cette formule spécifie que la séquence considérée doit vérifier que toutes les positions satisfont p ou que au moins une position satisfait q. Notons que cette r-propriété ne peut être exprimée comme une safety ou une guarantee pure.

Fermeture des propriétés obligation-spécifiables La classe des propriétés spécifiables par des formules de simple obligation est fermée par union mais pas par intersection. Pour voir qu'effectivement la disjonction de formules de simple obligation est encore une simple obligation, observons l'équivalence évidente suivante :

$$[(\Box\, p_1 \vee \Diamond\, q_1) \vee (\Box\, p_2 \vee \Diamond\, q_2)] \sim [(\Box\, p_1 \vee \Box\, p_2) \vee (\Diamond\, q_1 \vee \Diamond\, q_2)]$$

Puis, la fermeture des classes des formules de safety et guarantee nous donne que la partie droite de l'équation est bien une formule de simple obligation. En revanche, en utilisant la conjonction dans les formules, nous obtenons une classe de formule plus expressive. La notion de m-obligation ($m \in \mathbb{N}$) canonique est définie comme suit :

$$\bigwedge_{i=1}^{m} [\Box\, p_i \vee \Diamond\, q_i]$$

Une propriété spécifiable par une telle formule est appelée une propriété m-obligation spécifiable.

La classe des r-propriétés obligation-spécifiables forme une hiérarchie stricte. La classe des r-propriétés spécifiables par des formules de $m+1$-obligation contient strictement la classe des r-propriétés spécifiables par des formules de m-obligation.

Formules de Response

Les formules de response canoniques sont définies comme les formules s'exprimant sous la forme $\Box \Diamond p$, où p est une formule du passé. De telles formules spécifient pour les séquences infinies qu'une infinité de positions de la séquence considérée satisfait p.

Fermeture des r-propriétés response-spécifiables. La classe des r-propriétés spécifiables par des formules de response est fermée par combinaisons booléennes positives. Ce qui est montré par les deux équivalences suivantes :

$$[\Box \Diamond p \vee \Box \Diamond q] \sim \Box \Diamond (p \vee q)$$
$$[\Box \Diamond p \wedge \Box \Diamond q] \sim \Box \Diamond (q \wedge \ominus((\neg q)\, \mathcal{S}\, p))$$

Formules de Persistence

Nous définissons les formules de persistence canoniques comme les formules s'exprimant sous la forme $\Diamond \Box p$, où p est une formule du passé. De telles formules spécifient pour les séquences infinies qu'à partir d'un certain rang, toutes les positions de la séquence considérée satisfont p.

Fermeture des propriétés persistence-spécifiables. Les propriétés spécifiables par des formules de persistence sont fermées par les opérations booléennes positives. En effet, les équivalences suivantes peuvent être dérivées des équivalences pour les r-propriétés response-spécifiables par dualité :

$$(\Diamond \Box p \wedge \Diamond \Box q) \sim \Diamond \Box (p \wedge q)$$
$$(\Diamond \Box p \vee \Diamond \Box q) \sim \Diamond \Box (q \vee \ominus(p\, \mathcal{S}\, (p \wedge (\neg q))))$$

Formules de Reactivity

Nous définissons les formules de réactivité simples canoniques comme les formules s'exprimant sous la forme

$$\Box \Diamond p \vee \Diamond \Box q$$

où p et q sont des formules du passé. De telles formules spécifient pour les séquences infinies que la séquence contient une infinité de positions satisfaisant p ou que à partir d'un certain rang toutes les positions de la séquence considérée satisfont q.

De manière similaire aux r-propriétés obligation-spécifiables, les r-propriétés m-reactivity-specifiables sont fermées par disjonction, mais pas par conjonction. La forme générale d'une formule de m-reactivity est la suivante :

$$\bigwedge_{i=1}^{m} [\Box \Diamond p_i \vee \Diamond \Box q_i]$$

Ce type de formule représente les formules les plus générales de la classification. Ce qui est assuré par le théorème suivant [CMP92a, MP90a].

Théorème α (La classe des formules de reactivity est la plus générale) : *Toute formule de logique temporelle est une formule de reactivity.*

À propos des formules de logique temporelle

Une formule équivalente à une formule canonique de safety (resp. guarantee, response, persistence) est appelée formule de safety (resp. guarantee, response, persistence).

Théorème β (Correspondance vue logique et vue langage) : *Les r-propriétés obtenues en appliquant les opérateurs sur les langages X_f et X où $X \in \{A, E, R, P\}$ à une propriété finitaire peuvent être représentées en logique temporelle en appliquant les modalités $\Box$, $\Diamond$, $\Box\Diamond$, $\Diamond\Box$ à une formule du passé p. Ce qui est formalisé par les déclarations suivantes :*

- $Sat(\Box\ p) = (A_f(esat(p)), A(esat(p)))$
- $Sat(\Diamond\ p) = (E_f(esat(p)), E(esat(p)))$
- $Sat(\Box\Diamond\ p) = (R_f(esat(p)), R(esat(p)))$
- $Sat(\Diamond\Box\ p) = (P_f(esat(p)), P(esat(p)))$

3.6 La vue automates des r-propriétés

Pour chaque classe de la classification *Safety-Progress*, il est possible de caractériser syntaxiquement un automate reconnaisseur. Nous définissons une variante des automates de Streett déterministes et complets (introduits dans [Str81] et utilisés dans [CMP92a]) pour la reconnaissance de propriétés. Ces automates traitent des événements et donnent un critère de décision pour les r-propriétés.

3.6.1 Automates de Streett

La variante [7] des automates de Streett que nous définissons se distingue par l'ajout aux automates de Streett originaux d'un critère de reconnaissance pour les séquences finies de telle sorte que ces automates reconnaissent uniformément les r-propriétés.

Définition 15 (m-automate de Streett). Un m-automate de Streett déterministe est un 5-tuple $(Q, q_{\text{init}}, \Sigma, \longrightarrow, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ défini relativement à un ensemble d'événements Σ. L'ensemble Q est l'ensemble des états de l'automate, et $q_{\text{init}} \in Q$ est l'état initial. La fonction $\longrightarrow : Q \times \Sigma \to Q$ est la fonction de transition. Dans la suite, pour $q, q' \in Q, e \in \Sigma$ nous notons $q \xrightarrow{e} q'$ pour $\longrightarrow (q, e) = q'$, et $q \longrightarrow q'$ pour $\exists e \in \Sigma : \longrightarrow (q, e) = q'$. L'ensemble $\{(R_1, P_1), \ldots, (R_m, P_m)\}$ est l'ensemble des paires d'acceptation, pour chaque $i \leq m$, les ensembles $R_i \subseteq Q$ sont les ensembles d'états récurrents, et $P_i \subseteq Q$ les ensembles d'états persistants.

Nous désignons un automate avec m paires d'acceptation comme un m-automate. Lorsque $m = 1$, un 1-automate est aussi un *plain*-automate, et nous désignons R_1 et P_1 par R et P. De plus, nous noterons $\xrightarrow{+}$ la fermeture transitive de $\longrightarrow$, et par $\xrightarrow{*}$ la fermeture réflexive et transitive de $\longrightarrow$. Dans la suite, $\mathcal{A} = (Q^{\mathcal{A}}, {q_{\text{init}}}^{\mathcal{A}}, \Sigma, \longrightarrow_{\mathcal{A}}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ désigne un m-automate de Streett.

Pour $q \in Q^{\mathcal{A}}$, $Reach_{\mathcal{A}}(q)$ est l'ensemble des états atteignables à partir de q avec au moins une transition dans $\mathcal{A}$, c'est-à-dire $Reach_{\mathcal{A}}(q) = \{q' \in Q^{\mathcal{A}} \mid q \xrightarrow{+} q'\}$. Pour $\sigma \in \Sigma^{\infty}$, le *run* de σ sur $\mathcal{A}$ est la séquence des états impliqués par l'exécution de σ sur $\mathcal{A}$. Le run est formellement défini par $run(\sigma, \mathcal{A}) = q_0 \cdot q_1 \cdots$ où $\forall i : (q_i \in Q^{\mathcal{A}} \wedge q_i \xrightarrow{\sigma_i}_{\mathcal{A}} q_{i+1}) \wedge q_0 = {q_{\text{init}}}^{\mathcal{A}}$. Lorsque $\sigma \in \Sigma^*$ $t.q.$ $|\sigma| = n$ et $run(\sigma, \mathcal{A}) = q_0 \cdots q_{n-1}$, nous disons que le run de σ sur $\mathcal{A}$ termine dans q_{n-1}. La *trace* résultante de l'exécution de σ sur $\mathcal{A}$ est la séquence unique (finie ou non) des tuples $(q_0, \sigma_0, q_1) \cdot (q_1, \sigma_1, q_2) \cdots$ où $run(\sigma, \mathcal{A}) = q_0 \cdot q_1 \cdots$. L'unicité de la trace est due au fait que nous considérons uniquement les automates de Streett déterministes.

Nous considérons aussi la notion de *visite infinie* d'une séquence d'exécution $\sigma \in \Sigma^{\omega}$ sur un automate de Streett $\mathcal{A}$, dénotée $vinf(\sigma, \mathcal{A})$, comme l'ensemble des états apparaissant infiniment souvent dans $run(\sigma, \mathcal{A})$. Cet ensemble est formellement défini comme suit : $vinf(\sigma, \mathcal{A}) = \{q \in Q^{\mathcal{A}} \mid \forall n \in \mathbb{N}, \exists m \in \mathbb{N} : m > n \wedge q = q_m$ avec $run(\sigma, \mathcal{A}) = q_0 \cdot q_1 \cdots\}$.

Pour un automate de Streett, la notion d'acceptation est définie en utilisant les paires d'acceptation.

---

7. Plusieurs définitions équivalentes des automates de Streett existent. Elles diffèrent principalement par la définition des paires d'acceptation et la condition d'acceptation des séquences. Nous choisissons de suivre la définition utilisée dans [CMP92a].

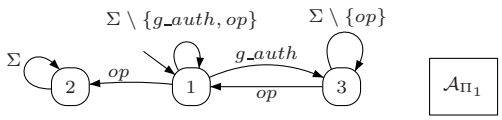


FIGURE 3.4 – Automate reconnaisseur pour la r-propriété de safety Π_1 ($P = \{1, 3\}$)

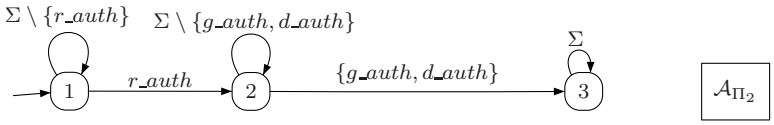


FIGURE 3.5 – Automate reconnaisseur pour la r-propriété de guarantee Π_2 ($R = \{3\}$)

DÉFINITION 16 (CONDITION D'ACCEPTATION (SÉQUENCES INFINIES) [CMP92A]). Pour $\sigma \in \Sigma^\omega$, nous disons que $\mathcal{A}$ accepte σ si :
$\forall i \in [1, m] \cdot vinf(\sigma, \mathcal{A}) \cap R_i \neq \emptyset \vee vinf(\sigma, \mathcal{A}) \subseteq P_i$.

Pour pouvoir traiter les r-propriétés nous avons besoin de définir également un critère d'acceptation pour les séquences *finies*.

DÉFINITION 17 (CONDITION D'ACCEPTATION (SÉQUENCES FINIES)). Pour une séquence d'exécution de longueur finie $\sigma \in \Sigma^*$ telle que $|\sigma| = n$, nous disons que le m-automate $\mathcal{A}$ accepte σ si :
$(\exists q_0, \ldots, q_n \in Q^{\mathcal{A}} : run(\sigma, \mathcal{A}) = q_0 \cdots q_n$ et $\forall i \in [1, m] : q_n \in P_i \cup R_i)$.

Reconnaissance et spécification de r-propriétés par des automates de Streett. Nous disons qu'un automate de Streett $\mathcal{A}$ *spécifie* une r-propriété $(\phi, \varphi) \in \Sigma^* \times \Sigma^\omega$ si et seulement si l'ensemble des séquences d'exécution finies (resp. infinies) acceptées par $\mathcal{A}$ est égal à ϕ (resp. φ), ce que nous notons $specifie(\mathcal{A}, (\phi, \varphi))$. De plus, une r-propriété $(\phi, \varphi) \in \Sigma^* \times \Sigma^\omega$ est dite être *spécifiable* par un automate $\mathcal{A}$ s'il existe un automate de Streett qui spécifie (ϕ, φ).

Exemple 5 (Spécification de r-propriétés par des automates de Streett) Les r-propriétés introduites précédemment dans l'Exemple 1 page 38 peuvent être spécifiées par des automates de Streett.

- La r-propriété Π_1 est spécifiée par le plain-automate $\mathcal{A}_{\Pi_1}$ représenté sur la Fig. 3.4. Son ensemble d'états est $\{1, 2, 3\}$, l'état initial est l'état 1, et les paires d'acceptation sont définies comme suit : $R = \emptyset$ et $P = \{1, 3\}$.
- La r-propriété Π_2 est spécifiée par le plain-automate $\mathcal{A}_{\Pi_2}$ représenté sur la Fig. 3.5. Son ensemble d'états est $\{1, 2, 3\}$, l'état initial est l'état 1, et les paires d'acceptation sont définies comme suit : $P = \emptyset$ et $R = \{3\}$.
- La r-propriété Π_3 est spécifiée par le plain-automate $\mathcal{A}_{\Pi_3}$ représenté sur la Fig. 3.6 page suivante. Son ensemble d'états est $\{1, 2, 3, 4\}$, l'état initial est l'état 1, et les paires d'acceptation sont définies comme suit : $P = \{1\}$ et $R = \{3\}$.
- La r-propriété Π_4 est spécifiée par le plain automate $\mathcal{A}_{\Pi_4}$ représenté sur la Fig. 3.7 page ci-contre. Son ensemble d'états est $\{1, 2, 3, 4\}$, l'état initial est l'état 1, et les paires d'acceptation sont définies comme suit : $P = \emptyset$ et $R = \{1\}$.
- La r-propriété Π_5 est spécifiée par le plain-automate $\mathcal{A}_{\Pi_5}$ représenté sur la Fig. 3.8 page 52. Son ensemble d'états est $\{1, 2, 3, 4\}$, l'état initial est l'état 1, et les paires d'acceptation sont définies comme suit : $P = \{3\}$ et $R = \emptyset$.

3.6.2 La hiérarchie des automates

La hiérarchie entre les différentes classes de propriétés vues dans les sections précédentes se retrouve au niveau de la vue automate. En effet, en appliquant des *restrictions syntaxiques* sur un automate de Streett, nous modifions le type de propriétés reconnues par cet automate.

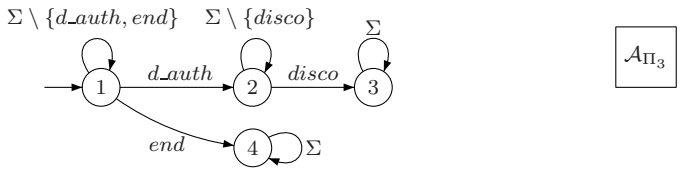


FIGURE 3.6 – Automate reconnaisseur pour la r-propriété de 1-obligation Π_3 ($P = \{1\}$ et $R = \{3\}$)

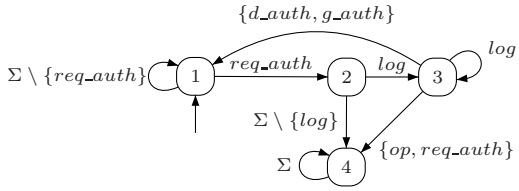


FIGURE 3.7 – Automate reconnaisseur pour la r-propriété de response Π_4 ($R = \{1\}$)

- Un *automate de safety* est un plain-automate tel que $R = \emptyset$ et il n'y a pas de transition depuis un état $q \in \overline{P}$ vers un état $q' \in P$.
- Un *automate de guarantee* est un plain-automate tel que $P = \emptyset$ et il n'y a pas de transition depuis un état $q \in R$ vers un état $q' \in \overline{R}$.
- un *automate de m-obligation* est un m-automate tel que pour chaque i dans $[1, m]$:
  - il n'y a pas de transition depuis $q \in \overline{P_i}$ vers $q' \in P_i$,
  - il n'y a pas de transition depuis $q \in R_i$ vers $q' \in \overline{R_i}$,
- Un *automate de response* est un plain-automate tel que $P = \emptyset$,
- Un *automate de persistence* est un plain-automate tel que $R = \emptyset$,
- Un *automate de réactivité* est n'importe quel automate non contraint.

Exemple 6 (Classes d'automates et restrictions syntaxiques) Sur la Fig. 3.9 page suivante sont représentés les illustrations schématiques pour chaque classe de base d'automates. Les ensembles d'états récurrents et persistants sont schématisés par des carrés. Les transitions entre les états récurrents ou persistants sont représentées par des flèches entre les carrés.

REMARQUE 2 (À PROPOS DE LA HIÉRARCHIE DES AUTOMATES) Il est possible de définir autrement les classes de propriétés. Notamment dans [CP03], les auteurs définissent une version des automates où la condition d'acceptation est définie par la notion d'"occurrence de visite" au lieu de "visite infinie". Les paires d'acceptation sont également utilisées de manière différente. *

Propriétés des automates. Nous exposons une propriété des automates liée à leurs paires d'acceptation. En effet, étant donné un m-automate de Streett (avec m paires d'acceptation), il est possible de caractériser le langage accepté par l'automate résultant en "oubliant" quelques paires d'acceptation sur l'automate initial. Cette propriété est formalisée comme suit.

Lemme β (Oubli de paires d'acceptation pour les propriétés d'obligation) : *Étant donné un m-automate $\mathcal{A}_\Pi = (Q, q_{\text{init}}, \Sigma, \longrightarrow, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ reconnaissant une r-propriété d'obligation Π. Suivant [CMP92a], Π peut être exprimée comme $\bigcap_{i=1}^{m} \Pi_i$ où les Π_i sont des r-propriétés d'obligation.*
Étant donné un sous-ensemble $X \subseteq [1, m]$, l'automate $\mathcal{A}_{\Pi/X} = (Q, q_{\text{init}}, \Sigma, \longrightarrow, \{(R_i, P_i) \mid i \in X\})$ reconnaît la r-propriété $\bigcap_{i \in X} \Pi_i$.

PREUVE : Pour les séquences d'exécution infinies, cette preuve a été faite dans [CMP92a]. Pour les séquences d'exécution finies, la preuve est une adaptation directe. ■

3.6.3 Synthèse d'automates de Streett à partir de DFAs

Nous introduisons maintenant quatre transformations permettant d'obtenir un automate de Streett, étant donnés un automate d'états finis déterministe et un "motif" souhaité pour cette propriété. Ces

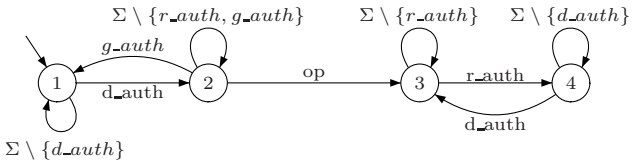


FIGURE 3.8 – Automate reconnaisseur pour la r-propriété de persistence Π_5 ($P = \{3\}$)

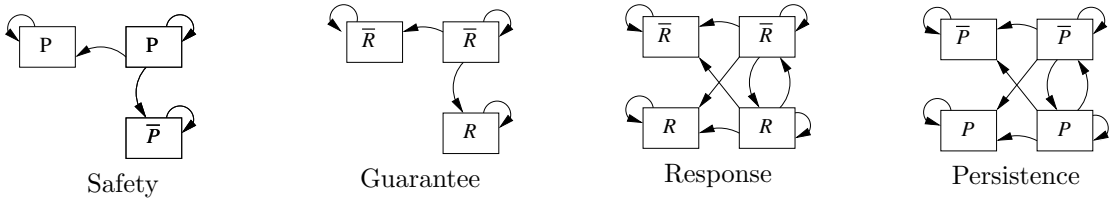


FIGURE 3.9 – Illustrations schématiques de la forme des automates de Streett pour les classes de base

motifs sont inspirés par les différentes classes de la hiérarchie *Safety-Progress*. Ces transformations simples sont les correspondants dans la vue automate des opérateurs de construction de propriétés dans la vue langage et des modalités dans la vue logique [8]. Nous commençons par définir les transformations, puis montrons qu'elles sont correctes.

Un automate d'états finis déterministe (Deterministic Finite-state Automaton, DFA) [HMU79], défini relativement à un alphabet Σ, est formellement défini comme un tuple $(Q, q_{\text{init}}, \longrightarrow, F)$ où Q est un ensemble fini d'états, $q_{\text{init}} \in Q$ est l'état initial, $\rightarrow: Q \times \Sigma \rightarrow Q$ est la fonction de transition, et $F \subseteq Q$ est l'ensemble des états accepteurs.

Définition des transformations

Dans la suite, $\mathcal{A}_\psi = (Q^{\mathcal{A}_\psi}, {q_{\text{init}}}^{\mathcal{A}_\psi}, \longrightarrow_{\mathcal{A}_\psi}, F^{\mathcal{A}_\psi})$ désigne un DFA reconnaissant une propriété finitaire régulière ψ. Nous définissons les transformations pour chaque classe de base de la hiérarchie.

Synthèse d'automates de safety. Pour cette classe de r-propriétés, la transformation est définie comme suit :

DÉFINITION 18 (DFA VERS AUTOMATE DE STREETT DE SAFETY). Nous définissons la transformation d'un DFA vers un automate de Streett de safety comme suit :

DFA2StreettSafety($\mathcal{A}_\psi$) = $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, {q_{\text{init}}}^{\mathcal{A}_\Pi}, \longrightarrow_{\mathcal{A}_\Pi}, \{(\emptyset, P)\})$ tel que :

- $Q^{\mathcal{A}_\Pi} = F \cup \{sink\}$, où $sink \notin Q^{\mathcal{A}_\psi}$,
- ${q_{\text{init}}}^{\mathcal{A}_\Pi} = {q_{\text{init}}}^{\mathcal{A}_\psi}$ si ${q_{\text{init}}}^{\mathcal{A}_\psi} \in F^{\mathcal{A}_\psi}$, et $sink$ sinon,
- $\rightarrow_{\mathcal{A}_\Pi}$ est définie comme la plus petite relation vérifiant :
  - $q \xrightarrow{a}_{\mathcal{A}_\Pi} q'$ si $q \in F \wedge q' \in F \wedge q \xrightarrow{a}_{\mathcal{A}_\psi} q'$ (TSAFE1)
  - $q \xrightarrow{a}_{\mathcal{A}_\Pi} sink$ si $\exists q' \in Q^{\mathcal{A}_\psi} : q' \notin F \wedge q \xrightarrow{a}_{\mathcal{A}_\psi} q'$ (TSAFE2)
  - $sink \xrightarrow{a}_{\mathcal{A}_\Pi} sink$, $\forall a \in \Sigma$ (TSAFE3)
- $P = F$, $(m = 1)$

Nous pouvons remarquer que l'automate résultant est bien un automate de Streett de safety car $R = \emptyset$ et il n'y a pas de transition depuis les états $\overline{P}$ vers les états P. Cette transformation garde uniquement les états accepteurs du DFA de départ et y ajoute un état puits. Ensuite la fonction de transition est modifiée de manière à rediriger les transitions sortant des états accepteurs vers cet état puits. De plus, les transitions du DFA partant d'un état n'appartenant pas à F ont été supprimées. L'ensemble des états persistants est l'ensemble des états accepteurs du DFA.

8. *i.e.*, les opérateurs A, E, R, P (et leurs versions finitaires) de la vue langage et les modalités temporelles $\Box, \Diamond, \Box\Diamond, \Diamond\Box$ de la vue logique.

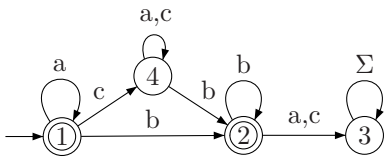


FIGURE 3.10 – DFA reconnaissant la propriété finitaire $a^* \cdot (b^* + c \cdot (c + a)^* \cdot b^+)$

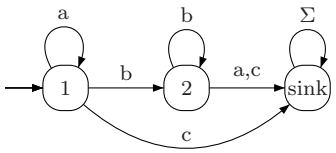


FIGURE 3.11 – DFA2StreettSafety appliquée sur le DFA de la Fig. 3.10 ($P = \{1, 2\}$)

Notons que l'ensemble des états de l'automate de Streett résultant est la plus petite solution X de l'équation : $X = \{q_{\text{init}}\} \cup X \cup post(X) \cap F$. De plus, d'après la forme des automates (de safety) obtenus, pour une séquence $\sigma \in \Sigma^\omega$ et un automate de Streett $\mathcal{A}_\Pi$ obtenu, nous avons les faits suivants : Si $sink \in vinf(\sigma, \mathcal{A}_\Pi)$, alors $vinf(\sigma, \mathcal{A}_\Pi) = \{sink\}$. Sinon, $vinf(\sigma, \mathcal{A}_\Pi) \subseteq P$.

Exemple 7 (DFA vers automate de Streett de safety) Considérons un vocabulaire $\Sigma = \{a, b, c\}$. Le DFA représenté sur la Fig. 3.10 reconnait la propriété finitaire $\psi_1 = a^* \cdot (b^* + c \cdot (c + a)^* \cdot b^+)$. Ses états accepteurs sont les états 1 et 2 (représentés par un double cercle). L'automate de Streett représenté sur la Fig. 3.11 est le résultat de l'application de la transformation DFA2StreettSafety sur le DFA représenté sur la Fig. 3.10. Les états 3 et 4 ont été supprimés (car non-acepteurs), et l'état *sink* a été introduit. Les transitions partant d'un état non supprimé et allant vers un état non-accepteur ont été redirigées vers l'état *sink*. De plus, les états 1 et 2 ont été marqués comme états P.

Synthèse d'automates de guarantee. Pour cette classe de r-propriétés, la transformation est définie comme suit :

DÉFINITION 19 (DFA VERS AUTOMATE DE STREETT DE GUARANTEE). Nous définissons la transformation d'un DFA vers un automate de Streett de guarantee comme suit :

DFA2StreettGuarantee($\mathcal{A}_\psi$) = $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \longrightarrow_{\mathcal{A}_\Pi}, \{(R, \emptyset)\})$ tel que :

- $Q^{\mathcal{A}_\Pi}$ est le plus petit sous-ensemble de $Q^{\mathcal{A}_\psi}$ contenant des états atteignables par $\longrightarrow_{\mathcal{A}_\Pi}$ depuis l'état initial $q_{\text{init}}{}^{\mathcal{A}_\Pi}$,
- $q_{\text{init}}{}^{\mathcal{A}_\Pi} = q_{\text{init}}{}^{\mathcal{A}_\psi}$,
- $\rightarrow_{\mathcal{A}_\Pi}$ est définie comme la plus petite relation vérifiant :
  - $q \xrightarrow{a}_{\mathcal{A}_\Pi} q$ si $\exists q' \in Q^{\mathcal{A}_\psi} : q \xrightarrow{a}_{\mathcal{A}_\psi} q' \wedge q \in F$ (TGUAR1)
  - $q \xrightarrow{a}_{\mathcal{A}_\Pi} q'$ si $q \notin F \wedge q \xrightarrow{a}_{\mathcal{A}_\psi} q'$ (TGUAR2)
- $R = F$, $(m = 1)$

Nous pouvons remarquer que l'automate résultant est bien un automate de Streett de guarantee car $P = \emptyset$ et il n'y a pas de transition depuis les états R vers les états $\overline{R}$. Notons que cet automate n'est pas minimal pour les états R, ceux-ci peuvent être fusionnés. Cette transformation modifie la fonction de transition de la manière suivante : Les transitions sortant des états accepteurs (vers un état accepteur ou non) sont transformées en une boucle sur le même état. L'état initial n'est pas modifié, et l'ensemble des états de l'automate de Streett est défini comme étant le plus petit ensemble d'états que nous pouvons atteindre à partir de l'état initial et la nouvelle fonction de transition.

Exemple 8 (DFA vers automate de Streett de guarantee) Considérons un vocabulaire $\Sigma = \{a, b\}$.

Le DFA représenté sur la Fig. 3.12 reconnait la propriété finitaire $(a \cdot b)^+$. Il possède un seul état accepteur : l'état 3.

Ensuite, l'automate représenté sur la Fig. 3.13 est le résultat de l'application de la transformation DFA2StreettGuarantee sur le DFA représenté sur la Fig. 3.12. On peut voir que les transitions sortant de

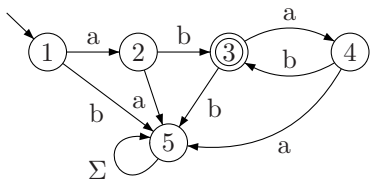


FIGURE 3.12 – DFA $\mathcal{A}_\psi$ reconnaissant la propriété finitaire $\psi = (a \cdot b)^+$

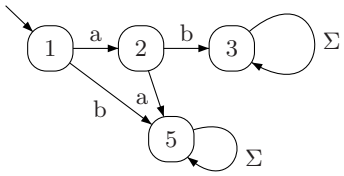


FIGURE 3.13 – DFA2StreettGuarantee appliquée sur le DFA de la Fig. 3.12 ($R = \{3\}$)

l'état accepteur sont redirigées vers celui-ci. De plus, l'état 4 a été supprimé. Et l'état 3 a été marqué comme état R.

Synthèse d'automates de response. Pour cette classe de r-propriétés, la transformation est définie comme suit :

DÉFINITION 20 (DFA VERS AUTOMATE DE STREETT DE RESPONSE). Nous définissons la transformation d'un DFA vers un automate de Streett de response comme suit :

DFA2StreettResponse($\mathcal{A}_\psi$) $= \mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \longrightarrow_{\mathcal{A}_\Pi}, \{(R, \emptyset)\})$ tel que :

- $Q^{\mathcal{A}_\Pi} = Q^{\mathcal{A}_\psi}$,
- $q_{\text{init}}{}^{\mathcal{A}_\Pi} = q_{\text{init}}{}^{\mathcal{A}_\psi}$,
- $\rightarrow_{\mathcal{A}_\Pi}$ est définie comme $\rightarrow_{\mathcal{A}_\psi}$, $R = \{q \in F \mid \exists l \in \mathbb{N}^*, \exists q_0, \ldots, q_l \in Q^{\mathcal{A}_\psi},$ (3.1) $\wedge$ (3.2) $\} \cup \{q \in F \mid q \longrightarrow_{\mathcal{A}_\psi} q\}$

$$\forall j \in [0, l-1] : q_j \longrightarrow_{\mathcal{A}_\psi} q_{j+1} \tag{3.1}$$

$$\exists i \in [0, l], \exists j \in [i, l-1] : q_j \in F \wedge q_i = q_l \wedge q_0 = q \tag{3.2}$$

Nous pouvons remarquer que l'automate résultant est bien un automate de Streett de response car $P = \emptyset$. Cette transformation ne modifie pas l'ensemble des états, ni la fonction de transition. Elle opère un marquage en état récurrent des états accepteurs du DFA initial à partir desquels nous pouvons atteindre un autre état accepteur (ou ce même état) avec au moins une transition.

Exemple 9 (DFA vers automate de Streett de response) L'automate représenté sur la Fig. 3.14 est le résultat de l'application de la transformation DFA2StreettResponse sur le DFA représenté sur la Fig. 3.12. Les transitions de l'automate n'ont pas été modifiées. Et l'état 3 a été marqué comme état R. En effet, il est possible d'atteindre un autre état accepteur (lui même) avec au moins une transition : le chemin $3 \xrightarrow{a} 4 \xrightarrow{b} 3$.

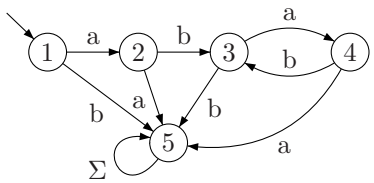


FIGURE 3.14 – DFA2StreettResponse appliquée sur le DFA de la Fig. 3.12 (($R = \{3\}$)

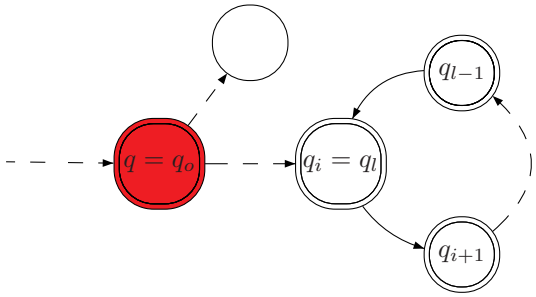


FIGURE 3.15 – Principe du marquage des états persistants pour DFA2StreettPersistence

Synthèse d'automates de persistence. Pour cette classe de r-propriétés, la transformation est définie comme suit :

DÉFINITION 21 (DFA VERS AUTOMATE DE STREETT DE PERSISTENCE). Nous définissons la transformation d'un DFA vers un automate de Streett de persistence comme suit :

DFA2StreettPersistence$(\mathcal{A}_\psi) = \mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, {q_{\text{init}}}^{\mathcal{A}_\Pi}, \longrightarrow_{\mathcal{A}_\Pi}, \{(\emptyset, P)\})$ tel que :

- $Q^{\mathcal{A}_\Pi} = Q^{\mathcal{A}_\psi}$,
- ${q_{\text{init}}}^{\mathcal{A}_\Pi} = {q_{\text{init}}}^{\mathcal{A}_\psi}$,
- $\rightarrow_\Pi$ est définie comme $\rightarrow_{\mathcal{A}_\psi}$,
- $P = \{q \in F \mid \exists l \in \mathbb{N}^*, \exists q_0, \ldots, q_l \in Q^{\mathcal{A}_\psi} : (3.3) \wedge (3.4)\} \cup \{q \in F \mid q \longrightarrow_{\mathcal{A}_\psi} q\}$

$$\forall j \in [0, l-1] : q_j \longrightarrow_{\mathcal{A}_\psi} q_{j+1} \tag{3.3}$$

$$\exists i \in [0, l], \forall j \in [i, l-1] : q_j \in F \wedge q_i = q_l \wedge q_0 = q \tag{3.4}$$

Nous pouvons remarquer que l'automate résultant est bien un automate de Streett de persistence car $R = \emptyset$. Cette transformation ne modifie pas l'ensemble des états, ni la fonction de transition. Elle opère (cf. Fig. 3.15) un marquage en état persistant des états accepteurs du DFA à partir desquels il est possible d'atteindre un état à partir duquel il existe un cycle passant uniquement par des états accepteurs.

Correction des transformations. Étant donnée une propriété ψ, définissant un langage régulier sur un alphabet Σ et spécifiée par un DFA $\mathcal{A}_\psi$, la r-propriété de safety (resp. guarantee, response, persistence) $(X_f(\psi), X(\psi))$ où $X \in \{A, E, R, P\}$ est spécifiée par l'automate de Streett obtenu par la transformation *DFA2Streett* spécifique aux propriétés de safety (resp. guarantee, response, persistence). Ceci est formalisé dans le théorème suivant.

Théorème γ (Correction des transformations *DFA2StreettX*) : *La transformation DFA2StreettSafety (resp. DFA2StreettGuarantee, DFA2StreettResponse, DFA2StreettPersistence) dans la vue automate "correspond" aux opérateurs A_f et A (resp. E_f et E, R_f et R, P_f et P) dans la vue langage. Plus précisément :*

$$(specifie(\mathcal{A}_\psi, \psi) \wedge \mathcal{A}_\Pi = DFA2StreettSafety(\mathcal{A}_\psi)) \Rightarrow specifie(\mathcal{A}_\Pi, (A_f(\psi), A(\psi)))$$
$$(specifie(\mathcal{A}_\psi, \psi) \wedge \mathcal{A}_\Pi = DFA2StreettGuarantee(\mathcal{A}_\psi)) \Rightarrow specifie(\mathcal{A}_\Pi, (E_f(\psi), E(\psi)))$$
$$(specifie(\mathcal{A}_\psi, \psi) \wedge \mathcal{A}_\Pi = DFA2StreettResponse(\mathcal{A}_\psi)) \Rightarrow specifie(\mathcal{A}_\Pi, (R_f(\psi), R(\psi)))$$
$$(specifie(\mathcal{A}_\psi, \psi) \wedge \mathcal{A}_\Pi = DFA2StreettPersistence(\mathcal{A}_\psi)) \Rightarrow specifie(\mathcal{A}_\Pi, (P_f(\psi), P(\psi)))$$

PREUVE : Notre preuve se fait pour chaque classe de propriétés et la transformation qui lui est associée. Pour une séquence d'exécution $\sigma \in \Sigma^*$, nous notons $run(\sigma, \mathcal{A}_\psi) = q_0 \cdots q_i \cdots$ le run de σ sur $\mathcal{A}_\psi$ et pour $\sigma \in \Sigma^\infty$, nous notons $run(\sigma, \mathcal{A}_\Pi) = q'_0 \cdots q'_i \cdots$ le run de σ sur $\mathcal{A}_\Pi$.

Pour les propriétés de Safety. Nous montrons que l'ensemble des séquences acceptées par $\mathcal{A}_\Pi$ obtenu par *DFA2StreettSafety* est exactement $(A_f(\psi), A(\psi))$.

Soit $\sigma \in \Sigma^\infty$ telle que $(A_f(\psi), A(\psi))(\sigma)$, montrons que la séquence σ est acceptée par $\mathcal{A}_\Pi$. Nous avons deux sous-cas : σ est une séquence finie ou non.

- Soit $\sigma \in \Sigma^*$ telle que $|\sigma| = n$, alors par définition des r-propriétés : $\sigma \in A_f(\psi)$, *i.e.*, tous les préfixes de σ appartiennent à ψ. Examinons $run(\sigma, \mathcal{A}_\psi) = q_0 \cdots q_{n-1}$. Comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, nous avons $\forall i \in [0, , n-1] : q_i \in F$. Par définition de la transformation DFA2StreettSafety, nous avons $\forall i \in [0, n-1] : q_i \in P$. D'après (TSafe1), nous avons $run(\sigma, \mathcal{A}_\Pi) = q_0 \cdots q_{n-1}$. D'après le critère d'acceptation des séquences finies, σ est acceptée par $\mathcal{A}_\Pi$.
- Soit $\sigma \in \Sigma^\omega$, alors par définition des r-propriétés : $\sigma \in A(\psi)$, *i.e.*, tous les préfixes finis de σ appartiennent à ψ. Supposons que σ ne soit pas acceptée par $\mathcal{A}_\Pi$. D'après le critère d'acceptation des séquences infinies (Définition 16), nous aurions que $vinf(\sigma, \mathcal{A}_\Pi) \nsubseteq P$ (comme $\mathcal{A}_\Pi$ est un automate de safety, $R = \emptyset$). Par définition de la transformation *DFA2StreettSafety* et la forme de l'automate $\mathcal{A}_\Pi$ obtenu, nous avons que $vinf(\sigma, \mathcal{A}_\Pi) = \{sink\}$. En utilisant (TSafe2), nous savons qu'il existe un plus petit préfixe σ' de σ, tel que le run de σ' sur $\mathcal{A}_\Pi$ atteigne l'état *sink*. Grâce à la définition de *DFA2StreettSafety*, nous pouvons déduire que le run de σ' sur $\mathcal{A}_\psi$ termine dans un état dans $\overline{F}$. Comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, $\sigma' \notin \psi$. Nous obtenons une contradiction avec $\sigma \in A(\psi)$.

Soit σ une séquence acceptée par $\mathcal{A}_\Pi$, montrons que $\sigma \in (A_f(\psi), A(\psi))$. Nous distinguons encore deux sous-cas : σ est une séquence finie ou non.

- Soit $\sigma \in \Sigma^*$ telle que $|\sigma| = n$, alors par définition du critère d'acceptation des séquences finies pour les automates de Streett (Définition 17), nous avons que $q'_{n-1} \in P$. Comme $\mathcal{A}_\Pi$ est un automate de safety, nous pouvons en déduire $\forall i \in [0, n-1] : q'_i \in P$. Grâce à la définition de *DFA2StreettSafety*, nous trouvons que les états visités lors du run de σ sur $\mathcal{A}_\psi$ sont tous dans F : $\forall i \in [0, n-1] : q_i \in F$. Par définition du critère d'acceptation des DFAs, nous en déduisons que tous les préfixes de σ sont acceptés par $\mathcal{A}_\psi$. Comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, nous en déduisons que tous les préfixes de σ appartiennent à ψ, *i.e.*, $\sigma \in A_f(\psi)$.
- Si $\sigma \in \Sigma^\omega$, alors par définition du critère d'acceptation des séquences infinies pour les automates de Streett (Définition 16), nous avons que $vinf(\sigma, \mathcal{A}_\Pi) \subseteq P$. Supposons que $\sigma \notin A(\psi)$, par définition de l'opérateur A (Définition 9), nous aurions l'existence d'un préfixe strict σ' de σ n'appartenant pas à ψ. Soit $n' = |\sigma'|$. Comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, alors le run de σ' sur $\mathcal{A}_\psi$, $run(\sigma', \mathcal{A}_\psi) = q_0 \cdots q_{n'-1}$, vérifierait $q_0 = q_{\text{init}}{}^{\mathcal{A}_\psi} \wedge q_{n'-1} \notin F$. Grâce à la définition de la transformation *DFA2StreettSafety* et de la règle (TSafe2), nous aurions que $q'_{n'-1} = sink \notin P$. Par la suite, en utilisant (TSafe3), toutes les continuations de σ' auraient leurs runs qui terminent dans *sink*. Nous en déduirions que $vinf(\sigma, \mathcal{A}_\Pi) = \{sink\} \nsubseteq \{P\}$. Ce qui contredit l'hypothèse initiale.

Pour les propriétés de Guarantee. Nous montrons que l'ensemble des séquences acceptées par $\mathcal{A}_\Pi$ obtenu par *DFA2StreettGuarantee* est exactement $(E_f(\psi), E(\psi))$.

Soit $\sigma \in \Sigma^\infty$ telle que $(E_f(\psi), E(\psi))(\sigma)$, montrons que la séquence σ est acceptée par $\mathcal{A}_\Pi$. Nous avons deux sous-cas : σ est une séquence finie ou non.

- Soit $\sigma \in \Sigma^*$ telle que $|\sigma| = n$, alors par définition des r-propriétés : $\sigma \in E_f(\psi)$, *i.e.*, σ a au moins un préfixe appartenant à ψ. Considérons $S_{sat} = \{\sigma' \in \Sigma^* \mid \sigma' \preceq \sigma \wedge \sigma' \in \psi\}$, l'ensemble des préfixes de σ qui appartiennent à ψ. Comme $\sigma \in E_f(\psi)$, nous pouvons en déduire que $S_{sat} \neq \emptyset$, S_{sat} a donc un plus petit élément σ_{min}. Soit $n' = |\sigma_{min}|$. Nous avons, par définition de σ_{min}, $\forall \sigma' \in \Sigma^* : \sigma' \prec \sigma_{min} \Rightarrow \sigma' \notin \psi$. Examinons maintenant $run(\sigma_{min}, \mathcal{A}_\psi) = q_0 \cdots q_{n'-1}$. Comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, nous avons $\forall i \in [0, n'-2] : q_i \notin F \wedge q_{n'-1} \in F$. D'après (TGuar2), nous avons $run(\sigma_{min}, \mathcal{A}_\Pi) = q_0 \cdots q_{n'-1}$ avec $\forall i \in [0, n'-2] : q_i \notin R \wedge q_{n'-1} \in R$. D'après (TGuar1), nous avons $\forall i \in [n'-1, n-1] : q_i \in R$. D'après le critère d'acceptation des séquences finies, σ est acceptée par $\mathcal{A}_\Pi$.
- Soit $\sigma \in \Sigma^\omega$, alors par définition des r-propriétés : $\sigma \in E(\psi)$, *i.e.*, (au moins) un préfixe fini de σ appartient à ψ. Supposons que σ ne soit pas acceptée par $\mathcal{A}_\Pi$. D'après le critère d'acceptation des séquences infinies (Définition 16), nous aurions que $vinf(\sigma, \mathcal{A}_\Pi) \cap R = \emptyset$ (comme $\mathcal{A}_\Pi$ est un automate de guarantee, $P = \emptyset$). Autrement dit, nous avons $vinf(\sigma, \mathcal{A}_\Pi) \subseteq \overline{R}$. Comme $\mathcal{A}_\Pi$ est un automate de guarantee, tous les états visités par le run de σ sur $\mathcal{A}_\Pi$ sont dans $\overline{R}$. En effet, d'après la forme de la fonction de transition des automates de guarantee, si un état de R était visité, alors nous aurions $vinf(\sigma, \mathcal{A}_\Pi) \cap R \neq \emptyset$. Considérons maintenant les préfixes de σ, lors du run de ces préfixes sur $\mathcal{A}_\Pi$, aucun d'entre eux ne visite un état R. Il s'en suit que d'après (TGuar2), aucun run de ces préfixes sur $\mathcal{A}_\Pi$ ne visite un état F. Comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, nous en déduirions qu'aucun des préfixes de σ n'appartient à ψ. Nous obtenons une contradiction avec $\sigma \in E(\psi)$, et donc σ est bien acceptée par $\mathcal{A}_\Pi$.

Soit σ une séquence acceptée par $\mathcal{A}_\Pi$, montrons que $\sigma \in (E_f(\psi), E(\psi))$. Nous distinguons encore deux sous-cas : σ est une séquence finie ou non.

- Soit $\sigma \in \Sigma^*$ telle que $|\sigma| = n$, alors par définition du critère d'acceptation des séquences finies pour les automates de Streett (Définition 17), nous avons que $q_{n-1} \in R$. Supposons que $\sigma \notin E_f(\psi)$, *i.e.*, aucun préfixe de σ n'appartiendrait à ψ. Comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, le run de σ sur $\mathcal{A}_\psi$ vérifierait : $\forall i \in [0, n-1] : q_i \notin F$. En partant de $q_{\text{init}}{}^{\mathcal{A}_\psi} = q_{\text{init}}{}^{\mathcal{A}_\Pi} \notin R$, et en utilisant (TGuar2), nous trouverions que $run(\sigma, \mathcal{A}_\Pi) = q_0 \cdots q_{n-1}$ avec $\forall i \in [0, n-1] : q_i \notin R$. Ce qui contredit $q_{n-1} \in R$, et donc $\sigma \in E_f(\psi)$.
- Soit $\sigma \in \Sigma^\omega$, alors par définition du critère d'acceptation des séquences infinies pour les automates de Streett (Définition 16), nous avons que $vinf(\sigma, \mathcal{A}_\Pi) \cap R \neq \emptyset$. Comme $\mathcal{A}_\Pi$ est un automate de guarantee, cela équivaut à $vinf(\sigma, \mathcal{A}_\Pi) \subseteq R$. D'après la forme de la fonction de transition des automates de guarantee, cela signifie qu'il y a un préfixe σ' de σ marquant le passage des états $\overline{R}$ aux états R lors du run de σ sur $\mathcal{A}_\Pi$. Plus formellement, $\exists \sigma' \in \Sigma^* : \sigma' \prec \sigma \wedge |\sigma'| = n' \wedge \forall i \in [0, n'-2] : q_i \in \overline{R} \wedge \forall i > n' : q_i \in R$. Supposons que $\sigma \notin E(\psi)$, *i.e.*, σ n'a aucun préfixe appartenant à ψ. Comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, le run de σ sur $\mathcal{A}_\psi$ vérifierait : $\forall i \in \mathbb{N} : q_i \notin F$. Similairement au cas finitaire, et d'après la transformation DFA2StreettGuarantee (règle (TGuar2)), cela remettrait en cause l'existence de σ'. Nous en déduisons que $\sigma \in E(\psi)$.

Pour les propriétés de Response. Nous montrons que l'ensemble des séquences acceptées par $\mathcal{A}_\Pi$ obtenu par *DFA2StreettResponse* est exactement $(R_f(\psi), R(\psi))$.

Soit $\sigma \in \Sigma^\infty$ telle que $(R_f(\psi), R(\psi))(\sigma)$, montrons que la séquence σ est acceptée par $\mathcal{A}_\Pi$. Nous avons deux sous-cas : σ est une séquence finie ou non.

- Soit $\sigma \in \Sigma^*$ telle que $|\sigma| = n$, alors par définition des r-propriétés : $\sigma \in R_f(\psi)$, *i.e.*, σ vérifie $\psi(\sigma)$ et $\exists \sigma' \in \Sigma^* : \sigma \prec \sigma' \wedge \psi(\sigma')$. Soit $n' = |\sigma'|$. Examinons $run(\sigma, \mathcal{A}_\psi) = q_0 \cdots q_{n-1}$. Comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, nous avons $q_{n-1} \in F$. Examinons maintenant $run(\sigma', \mathcal{A}_\psi) = q_0 \cdots q_{n-1} \cdots q_{n'-1}$. De même, comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, nous avons $q_{n'-1} \in F$. En utilisant la définition de *DFA2StreettResponse* $(\rightarrow_{\mathcal{A}_\Pi} = \rightarrow_{\mathcal{A}_\psi})$, nous avons $run(\sigma', \mathcal{A}_\Pi) = q_0 \cdots q_{n-1} \cdots q_{n'-1}$. De plus, nous pouvons déduire que $q_{n-1} \in R$ car $q_{n-1} \in F$ et $q_{n'-1} \in F$. D'après le critère d'acceptation des séquences finies, σ est acceptée par $\mathcal{A}_\Pi$.
- Soit $\sigma \in \Sigma^\omega$, alors par définition des r-propriétés : $\sigma \in R(\psi)$, *i.e.*, σ vérifie $\forall \sigma' \in \Sigma^*, \exists \sigma'' \in \Sigma^* : \sigma' \prec \sigma'' \prec \sigma \wedge \psi(\sigma'')$. Examinons le run de σ sur $\mathcal{A}_\Pi$, nous allons montrer que ce run visite au moins un état R infiniment souvent. En effet, prenons un préfixe σ' de σ, nous pouvons trouver une extension σ'' de σ', telle que $\psi(\sigma'')$. De plus pour cette extension σ'', nous pouvons encore trouver une extension σ''' telle que $\psi(\sigma''')$. En utilisant $\mathcal{L}(\mathcal{A}_\psi) = \psi$, les runs de l'automate $\mathcal{A}_\psi$ sur σ'' et σ''' s'arrêtent donc sur un état F. Ce qui nous permet de déduire que l'état dans lequel s'arrête le run de $\mathcal{A}_\psi$ sur σ'' est un état R. Ainsi on peut construire une suite $(\sigma_i)_{i \in \mathbb{N}}$ de préfixes de σ (de longueur strictement croissante) telle que le run de chaque σ_i s'arrête dans un état R. Ainsi un nombre infini de préfixes de σ passent par un état de R. Comme $|R| \in \mathbb{N}$, il existe un état de R visité infiniment souvent lors du run de σ sur $\mathcal{A}_\Pi$. D'après le critère d'acceptation des séquences infinies, σ est acceptée par $\mathcal{A}_\Pi$.

Soit σ une séquence acceptée par $\mathcal{A}_\Pi$, montrons que $\sigma \in (R_f(\psi), R(\psi))$. Nous distinguons encore deux sous-cas : σ est une séquence finie ou non.

- Soit $\sigma \in \Sigma^*$ telle que $|\sigma| = n$, alors par définition du critère d'acceptation des séquences finies pour les automates de Streett (Définition 17), nous avons que $q_{n-1} \in R$. D'après la définition de *DFA2StreettResponse*, nous en déduisons que $q_{n-1} \in F$ et $Reach(q_{n-1}) \cap F \neq \emptyset$. Alors, il existe $\sigma' \in \Sigma^*$ telle que $\sigma \prec \sigma' \wedge |\sigma'| = n' \wedge run(\sigma', \mathcal{A}_\psi) = q_0 \cdots q_{n'-1} \wedge q_{n'-1} \in F$. Comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, nous avons $\psi(\sigma)$ et $\psi(\sigma')$. Nous pouvons en déduire que $\sigma \in R_f(\psi)$.
- Soit $\sigma \in \Sigma^\omega$, alors par définition du critère d'acceptation des séquences infinies pour les automates de Streett (Définition 16), nous avons que $vinf(\sigma, \mathcal{A}_\Pi) \cap R \neq \emptyset$. Nous avons que σ possède un nombre infini de préfixes dont le run s'arrête sur un état R. En utilisant la définition de *DFA2StreettResponse*, nous savons que tous ces préfixes sont acceptés par $\mathcal{A}_\psi$ (car par définition l'état terminal de leur run est un état R). En utilisant $\mathcal{L}(\mathcal{A}_\psi) = \psi$, nous savons que tous ces préfixes appartiennent et ont une extension qui appartient également à ψ. Nous pouvons en déduire que $\sigma \in R(\psi)$.

Pour les propriétés de Persistence. Nous montrons que l'ensemble des séquences acceptées par $\mathcal{A}_\Pi$ obtenu par *DFA2StreettPersistence* est exactement $(P_f(\psi), P(\psi))$. Remarquons tout d'abord que d'après la définition de Streett2Persistence (la fonction de transition reste inchangée), nous avons évidemment $\forall j \in [n-1, n'+n''-1] : q_j \longrightarrow_{\mathcal{A}_\Pi} q_{j+1} \wedge q_j \longrightarrow_{\mathcal{A}_\psi} q_{j+1}$. De plus, comme $Q^{\mathcal{A}_\Pi} = Q^{\mathcal{A}_\psi}$, nous pouvons confondre les états q_j et q'_j visités par les runs de σ sur $\mathcal{A}_\psi$ et $\mathcal{A}_\Pi$.

Soit $\sigma \in \Sigma^\infty$ telle que $(P_f(\psi), P(\psi))(\sigma)$, montrons que la séquence σ est acceptée par $\mathcal{A}_\Pi$. Nous avons deux sous-cas : σ est une séquence finie ou non.

- Montrer que σ est acceptée par $\mathcal{A}_\Pi$ revient à montrer que le run de σ sur $\mathcal{A}_\Pi$ termine dans un état P ($q_{n-1} \in P$). Remarquons tout d'abord que $\sigma \in P_f(\psi)$ nous donne $\psi(\sigma)$. De plus, comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, nous en déduisons que $q_{n-1} \in F$.
  Comme $\sigma \in P_f(\psi)$, alors il existe $\sigma', \mu \in \Sigma^*$ tels que (cf. Définition 10) :
  $$\sigma \preceq \sigma' \wedge (\sigma' \cdot \mu^* \cdot pref(\mu)) \subseteq \psi \tag{3.5}$$
  Soient $n' = |\sigma'|$, et $n'' = |\mu|$. Alors, les runs de σ' et $\sigma' \cdot \mu$ sur $\mathcal{A}_\Pi$ peuvent s'exprimer :
  $$run(\sigma', \mathcal{A}_\Pi) = q_0 \cdots q_{n-1} \cdots q_{n'-1}$$
  $$run(\sigma' \cdot \mu, \mathcal{A}_\Pi) = q_0 \cdots q_{n-1} \cdots q_{n'-1} \cdot q_{n'} \cdots q_{n'+n''-1}$$
  D'après (3.5), on a $q_{n'-1} \in F$. Nous pouvons montrer par induction que $run(\sigma \cdot \mu^*, \mathcal{A}_\Pi) = q_0 \cdots q_{n'-1} \cdot (q_{n'} \cdots q_{n'+n''-1})^*$.
  On a de plus $\forall j \in [n', n'+n''-1] : q'_j \in F$. Nous avons également $q_{n'+n''-1} \longrightarrow_{\mathcal{A}_\Pi} q_{n'}$.
  Ainsi, nous pouvons en déduire, d'après la définition de Streett2Persistence, que $q_{n-1} \in P$. Il suffit en effet de prendre $l = n' + n'' - 1 - n$ et $i = n' - n$.
- Pour montrer que $vinf(\sigma, \mathcal{A}_\Pi) \subseteq P$, il suffit de voir que σ peut s'exprimer sous la forme $\sigma' \cdot \mu^\omega$. De là, tous les préfixes de σ plus longs que σ' satisfont ψ, et ont leur run qui s'arrête dans un état F sur $\mathcal{A}_\psi$. Ainsi, on met en évidence une composante fortement connexe d'états F qui sont marqués P par DFA2StreettPersistence. Ainsi, les états visités infiniment souvent lors du run de σ sur $\mathcal{A}_\Pi$ sont les états de cette composante fortement connexe. Ce qui permet de montrer le résultat recherché.

Soit σ une séquence acceptée par $\mathcal{A}_\Pi$, montrons que $\sigma \in (P_f(\psi), P(\psi))$. Nous distinguons encore deux sous-cas : σ est une séquence finie ou non.

- Soit $\sigma \in \Sigma^*$ telle que $|\sigma| = n$, alors par définition du critère d'acceptation des séquences finies pour les automates de Streett (Définition 17), nous avons que $q_{n-1} \in P$. Alors il existe deux possibilités.
  - Dans la première nous avons d'une part que $q_{n-1} \in F$, et d'autre part $\exists n \in \mathbb{N}^*, \exists q_0, \cdots, q_n \in Q^{\mathcal{A}_\psi}$ tels que :
    - $\forall j \in [0, n-1] : q_j \longrightarrow_{\mathcal{A}_\psi} q_{j+1}$, et
    - $\exists i \in [0, n-1], \forall j \in [i, n-1] : q_j \in F \wedge q_i = q_n \wedge q_0 = q_{n-1}$

    Nous avons $\psi(\sigma)$ car $\mathcal{L}(\mathcal{A}_\psi) = \psi$. De plus, il existe $a_0, \ldots, a_{n-1} \in \Sigma$ tels que $\forall j \in [0, n-1] : q_j \xrightarrow{a_j}_{\mathcal{A}_\psi} q_{j+1}$. Nous pouvons en déduire que $\psi(\sigma \cdot a_0 \cdots a_i), \psi(\sigma \cdot a_0 \cdots a_i \cdot a_{i+1}), \ldots, \psi(\sigma \cdot a_0 \cdots a_{n-1})$. Notons $L_p = \sigma' \cdot \Big((a_0 \cdots a_n)^* \cdot a_0 + (a_0 \cdots a_n)^* \cdot a_0 \cdot a_1 + \ldots + (a_0 \cdots a_n)^* \cdot a_0 \cdots a_{n-1}\Big)$. Comme $q_i = q_n$ ($\{q_i, \ldots, q_n\}$ est une composante fortement connexe), nous pouvons montrer par induction que $L_p \subseteq \psi$. De plus, $\forall \sigma' \in \Sigma^* \cap L_p : \sigma \cdot a_0 \cdots a_i \preceq \sigma' \Rightarrow \psi(\sigma')$. Ceci montre que $\sigma \in P_f(\psi)$. En effet, il suffit de prendre $\sigma' = \sigma \cdot a_0 \cdots a_i$, et $\mu = a_{i+1} \cdots a_{n-1}$.
  - Dans la deuxième, nous avons que $q_{n-1} \in F$ et $q_{n-1} \longrightarrow_{\mathcal{A}_\psi} q_{n-1}$. Ainsi, $\exists e \in \Sigma : q_{n-1} \xrightarrow{e}_{\mathcal{A}_\psi} q_{n-1}$. Nous en déduisons que $\psi(\sigma)$ et $\sigma \cdot e^* \subseteq \psi$, car $\mathcal{L}(\mathcal{A}_\psi) = \psi$. Ce qui permet également de déduire facilement que $\sigma \in P_f(\psi)$. ■
- Soit $\sigma \in \Sigma^\omega$, alors par définition du critère d'acceptation des séquences infinies pour les automates de Streett (Définition 16), nous avons que $vinf(\sigma, \mathcal{A}_\Pi) \subseteq P$. C'est-à-dire que tous les préfixes de σ à partir d'un certain rang ont leur run qui termine dans un état P. Comme l'automate $\mathcal{A}_\Pi$ a un nombre fini d'états, cela signifie qu'il existe une composante fortement connexe C, telle que le run de σ sur $\mathcal{A}_\Pi$ "reste dedans". Plus formellement, $\exists n, m \in \mathbb{N} : C = \{q'_0, \ldots, q'_n\} \subseteq Q^{\mathcal{A}_\Pi} \wedge run(\sigma, \mathcal{A}_\Pi) = q_0 \cdots q_m \cdots \wedge \forall i > m : q_i \in C$. De plus, comme $\{q'_0, \ldots q'_n\}$ est une SCC, à partir de tout état de C il est possible d'atteindre n'importequel état de C. Supposons sans perte de généralité que $q'_0 \xrightarrow{a_0}_{\mathcal{A}_\Pi} q'_1 \xrightarrow{a_1}_{\mathcal{A}_\Pi} \cdots \xrightarrow{a_{n-1}}_{\mathcal{A}_\Pi} q'_n \xrightarrow{a_n}_{\mathcal{A}_\Pi} q'_0$, avec $a_0, \ldots, a_n \in \Sigma$. D'après la définition de Streett2Persistence, nous avons les mêmes transitions sur $\mathcal{A}_\psi$, *i.e.*, $q'_0 \xrightarrow{a_0}_{\mathcal{A}_\psi} q'_1 \xrightarrow{a_1}_{\mathcal{A}_\psi} \cdots \xrightarrow{a_{n-1}}_{\mathcal{A}_\psi} q'_n \xrightarrow{a_n}_{\mathcal{A}_\psi} q'_0$.

	Vue Langage	**Vue Logique**	**Vue Automate**
"Brique de base"	$\psi \subseteq \Sigma^+$	p (past formula)	$\mathcal{A}$ (DFA)
r-propriété	$(X_f(\psi), X(\psi))$ $X \in \{A, E, R, P\}$	*Modality* p *Modality* $\in \{\Box, \Diamond, \Box\Diamond, \Diamond\Box\}$	*DFA2StreettX*$(\mathcal{A})$ $X \in \{Saf., Guar., Resp., Persit.\}$

TABLE 3.1 – Différentes manières de spécifier des r-propriétés selon les vues

	Vue Langage	**Vue Logique**	**Vue Automate**
Séquences finies	$\in X_f(\psi)$ (Déf. 9)	$\models$ (Déf. 13)	Critère séquences finies (Déf. 17)
Séquences infinies	$\in X(\psi)$ (Déf. 10)	$\models$ (Déf. 13)	Critère séquences infinies (Déf. 16)

TABLE 3.2 – Différents critères de reconnaissance selon les vues

Notons $L_p = \sigma' \cdot (a_0 \cdots a_n)^* \cdot (a_0 + a_0 \cdot a_1 + \ldots + a_0 \cdots a_{n-1}) = \sigma' \cdot (a_0 \cdots a_n) \cdot pref(a_0 \cdots a_{n-1})$. La séquence σ peut s'exprimer $\sigma' \cdot (a_0 \cdots a_n)^\omega$ avec le fait que pour toute séquence $\sigma'' \in L_p$ qui est une continuation de σ', le run de σ'' termine dans un état P. Ce qui implique que le run de ces mêmes séquences σ'' sur $\mathcal{A}_\psi$ terminent dans un état F. Nous en déduisons, comme $\mathcal{L}(\mathcal{A}_\psi) = \psi$, que $\forall \sigma'' \in L_p : \sigma' \preceq \sigma'' \prec \sigma \Rightarrow \psi(\sigma'')$.
Ce qui permet de déduire, en utilisant la définition de l'opérateur P (Définition 9), que $\sigma \in P(\psi)$.

REMARQUE 3 Ces transformations donnent une traduction effective des r-propriétés exprimées (et exprimables) en LTL appartenant à une classe de base vers leur expression dans la vue automates (cf. Section 3.8). *

REMARQUE 4 Notons que ces transformations engendrent une perte d'information. Il n'est en général pas possible de retrouver le langage finitaire à partir duquel un automate de Streett a été construit. Prenons par exemple l'automate de Streett de guarantee représenté sur la Fig. 3.13. Il existe une infinité de langages finitaires à partir duquel cet automate peut être construit. En effet, pour les obtenir, il suffit d'abord de retransformer cet automate en DFA "minimal" en oubliant les informations sur les paires d'acceptation en transformant l'état R en état accepteur. Ensuite à partir de l'état accepteur, nous pouvons établir arbitrairement les transitions que l'on veut. L'automate constuit sera toujours transformé par *DFA2Streett* en l'automate de la Fig. 3.13. *

3.7 La vue topologique des r-propriétés (Remarque)

La dernière vue de la classification Safety-Progress est une vue topologique. La topologie est basée sur une notion de distance $\mu(\sigma, \sigma') = 2^{-j}$ où j est la longueur de leur plus grand préfixe commun. La distance ainsi définie fait que Σ^ω devient un espace métrique. Alors les différentes classes (de base) peuvent être définies ainsi :

- Safety : ce sont les ensembles fermés.
- Guarantee : ce sont les ensembles ouverts.
- Response : ce sont les ensembles que l'on peut obtenir par intersection dénombrable d'ensembles ouverts.
- Persistence : ce sont les ensembles que l'on peut obtenir par union dénombrable d'ensembles fermés.

À noter que les liveness sont la famille des ensembles denses [9].

Nous ne définirons pas les r-propriétés en vue topologique. En effet, la relation entre les notions de limite sur les séquences finies et infinies n'est pas claire.

3.8 Connexions entre les différentes vues

Dans le tableau 3.1 est représenté, pour chaque "brique de base", *i.e.*, l'élément qui est utilisé pour construire une r-propriété. Dans la vue langage les r-propriétés sont construites à partir d'un langage

9. Dans l'espace topologique Σ^ω ainsi défini, un ensemble E est dense si pour toute séquence $\sigma \in \Sigma^\omega$ et tout $\epsilon > 0$, on peut trouver un élément de E à une distance inférieure de ϵ.

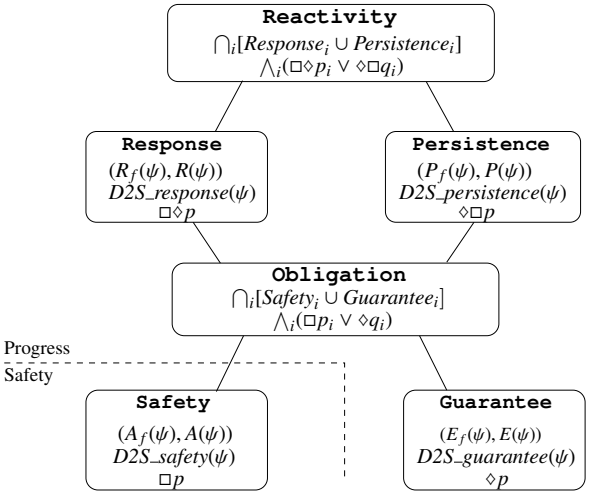


FIGURE 3.16 – Représentation hiérarchique de la classification *Safety-Progress* des r-propriétés

finitaire ψ, et en utilisant les opérateurs X_f, et X, avec $X \in \{A, E, R, P\}$. Dans la vue logique, on applique une modalité $\Box$,$\Diamond$,$\Box\ \Diamond$,$\Diamond\ \Box$ à une formule du passé p. Dans la vue automate, un automate d'états fini $\mathcal{A}_\psi$ est transformé par l'une des transformations DFA2Streett spécifique à une classe de propriétés en automate de Streett qui spécifie $(X_f(\psi), X(\psi))$ en fonction de la classe de propriétés.

Notons que les transformations DFA2Streett donnent une traduction effective pour les classes de propriétés de base. Etant donnée une formule de LTL Υ sur un ensemble de propositions atomiques *Prop*, il suffit de réécrire cette formule sous forme canonique. Ensuite, la formule du passé apparaissant dans la formule atomique peut être transformée en un DFA sur un vocabulaire $\Sigma = 2^{Prop}$ qui reconnait la même propriété.

Nous avons la propriété suivante qui se déduit des résultats présentés dans ce chapitre.

Propriété 4 (Correspondances entre les différentes vues) : *Soit $\psi \subseteq \Sigma^*$, une propriété finitaire régulière, exprimable comme une formule du passé p, et reconnue par un DFA $\mathcal{A}_\psi$.*

$$\begin{aligned}
Sat(\Box\ p) &= (A_f(\psi), A(\psi)) \ \wedge\ specifie(\text{DFA2StreettSafety}(\mathcal{A}_\psi), (A_f(\psi), A(\psi))) \\
Sat(\Diamond\ p) &= (E_f(\psi), E(\psi)) \ \wedge\ specifie(\text{DFA2StreettGuarantee}(\mathcal{A}_\psi), (E_f(\psi), E(\psi))) \\
Sat(\Box\ \Diamond\ p) &= (R_f(\psi), R(\psi)) \ \wedge\ specifie(\text{DFA2StreettResponse}(\mathcal{A}_\psi), (R_f(\psi), R(\psi))) \\
Sat(\Diamond\ \Box\ p) &= (P_f(\psi), P(\psi)) \ \wedge\ specifie(\text{DFA2StreettPersistence}(\mathcal{A}_\psi), (P_f(\psi), P(\psi))) \qquad \Diamond
\end{aligned}$$

3.9 Conclusion du chapitre et perspectives

Conclusion. Nous avons étendu les vues langage et automate de la classification *Safety-Progress* pour pouvoir y intégrer uniformément les séquences de longueur finie. Ces extensions sont consistantes vis-à-vis des différentes vues. La représentation hiérarchique de la classification *Safety-Progress* des r-propriétés est donnée sur la Fig. 3.16. Pour chaque classe, les caractérisations possibles en terme de langages, logique, et automates y sont représentées.

Dans la vue langage (Section 3.4), nous avons introduit de nouveaux opérateurs pour les séquences finies. De plus, nous avons donné une définition formelle à tous les opérateurs permettant de construire des r-propriétés pour les classes de base.

Dans la vue logique (Section 3.5), le cadre que nous avons défini utilise directement les sémantiques de LTL et FLTL. Nous allons voir dans le chapitre suivant, qu'à partir de ce cadre, il est possible de donner une interprétation d'une séquence finie vis-à-vis d'une r-propriété. La cadre que nous avons défini ne modifie pas la sémantique, connue et acceptée, de la logique linéaire temporelle. Ce qui favorise l'utilisation de ce cadre pour la spécification de propriétés.

Dans la vue automate (Section 3.6), nous avons introduit un nouveau critère d'acceptation pour les séquences finies. De plus, nous avons défini des transformations DFA2Streett permettant de passer d'un

DFA vers un automate de Streett pour chaque classe de base. L'ensemble des ces transformations sont implémentées dans l'outil j-VETO (cf. Chapitre 8).

L'une des motivations initiales pour faire la distinction entre propriétés de safety et de liveness était que le type de preuves associé à une propriété dépendait de la classe de celle-ci. Nous nous sommes placés dans le contexte de la classification *Safety-Progress* qui offre une classification plus fine de l'espace des propriétés. Le cadre ainsi défini sera utilisé pour la spécification de propriétés à valider lors de l'exécution. En effet, nous verrons dans le chapitre suivant que l'applicabilité des méthodes de validation pour une propriété dépend de la classe de cette dernière.

Perspectives. Pour les propriétés des classes de response et de persistence, la sémantique finitaire des r-propriétés nous semble mériter d'être raffinée. Par exemple, pour la propriété exprimée par une formule LTL $\Box \Diamond p$, il est peut être souhaitable que la sémantique permette d'exprimer une certaine "densité" de la validité de la formule p, *e.g.*, représenter la "fréquence" où p est satisfaite.

Les connexions entre les différentes vues fournissent une traduction indirecte entre les formules de LTL et les automates de Streett. Également, il nous semble intéressant de trouver, dans le cadre des r-propriétés, une traduction directe depuis une formule LTL quelconque vers l'automate de Streett correspondant. Cette traduction pourrait s'appuyer sur [Zuc86]. Aussi, étudier et comparer les complexités de ces traductions est un travail en suspens.

CHAPITRE 4

Applicabilité des méthodes de validation à l'exécution

Sommaire

4.1 Introduction **65**
4.2 Monitorabilité dans la classification Safety-Progress **65**
4.2.1 Selon la définition classique du monitoring 66
4.2.2 Une définition alternative du monitoring 69
4.3 Enforçabilité par rapport à la classification *Safety-Progress* **73**
4.3.1 Critères d'enforcement 73
4.3.2 Caractérisation des propriétés enforçables 75
4.4 Testabilité dans la classification *Safety-Progress* **78**
4.4.1 Notion de testabilité 78
4.4.2 Caractérisation des propriétés testables 80
4.4.3 Raffinement de la notion de verdict 85
4.4.4 Introduction de la notion de quiescence 86
4.5 Conclusion et perspectives **87**

Chapter abstract

The question we consider in this chapter is the following : what are the classes of properties that can be handled at runtime, and is there a difference between the techniques we are interested in ? This question is not new, but we propose here to address it within a unified framework : the *Safety-Progress* (SP) classification of *r-properties* ([MP90b, CMP92b] and Chapter 3). In this framework, we characterize the sets of monitorable, enforceable, and testable *r-properties* in a unified way.

We start first by revisiting the existing classical definition of monitorability, and characterize the set of monitorable properties according to this definition. Then, we extend this definition to obtain multivalued definitions according to positive and negative determinacy. Moreover, we introduce a new definition of monitorability based on distinguishability of good and bad execution sequences. This definition is weaker than the classical one (based on positive and negative determinacy) and we believe that it better corresponds to practical needs and tool implementations.

Furthermore, we characterize the set of enforceable properties in a way that is independent from any enforcement mechanism. A consequence is that the proposed set of enforceable properties is an upper bound of any enforcement mechanism.

Fig. 4.3 (p. 72) summarizes the results of this chapter for the runtime verification and enforcement sides, depicting the sets of monitorable and enforceable properties wrt. the *Safety-Progress* classification. Some of these results previously appeared in [FFM09b]. This chapter provides additional proofs, examples, and explanations.

Finally, we explore the set of testable properties. Testability is defined according to a relation between the tested system and the *r-property* under scrutiny. So, we characterize the space of testable properties wrt. several relations. For each relation, we give a sufficient condition for a *r-property* to be testable. Then, we study and delineate for each class of *r-properties*, the set of testable *r-properties* in this class.

Résumé du chapitre

La question que nous considérons dans ce chapitre est la suivante : quelles sont les classes de propriétés qui peuvent être utilisées par les techniques de validation opérant à l'exécution, et existe-t-il une distinction entre ces techniques ? Cette question n'est pas originale en soi, mais nous proposons de nous y intéresser ici dans un cadre commun : la classification *Safety-Progress* (SP) des r-propriétés ([MP90b, CMP92b] et Chapitre 3). Dans ce cadre, nous caractérisons les ensembles de r-propriétés monitorables et enforçables de façon uniforme.

Nous commençons par revisiter la définition classique de monitorabilité, et donnons une caractérisation des propriétés monitorables selon cette définition. Ensuite, nous donnons une version de cette définition pour plusieurs domaines de vérité, et donnons une caractérisation des propriétés monitorables selon les différents cas. Puis, nous introduisons une nouvelle définition de monitorabilité basée sur la distinction des séquences d'exécution bonnes et mauvaises. Cette définition est plus faible que la définition classique basée sur la détermination positive et négative. Nous pensons que cette définition correspond mieux aux besoins pratiques et aux implantations d'outils de vérification à l'exécution.

De plus, nous caractérisons l'ensemble des propriétés enforçables d'une façon qui est indépendante de tout mécanisme d'enforcement. Ainsi l'ensemble des propriétés enforçables que nous proposons est une borne supérieure au pouvoir d'enforcement de tout mécanisme.

La Fig. 4.3 (p. 72) résume les résultats apportés par ce chapitre coté vérification et enforcement, décrivant les ensembles de propriétés monitorables et enforçables par rapport à la classification *Safety-Progress*. Certains résultats préliminaires de ce chapitre ont été présentés dans [FFM09b]. Nous y apportons des résultats et preuves supplémentaires.

Enfin, nous explorons l'espace des propriétés testables. La notion de testabilité est définie selon la relation examinée entre le système testé et la r-propriété. Ainsi, nous caractérisons l'espace des propriétés testables selon plusieurs relations. Pour chaque relation, nous donnons une condition suffisante pour qu'une r-propriété soit testable. Puis nous étudions et délimitons pour chaque classe de r-propriétés, l'ensemble des r-propriétés testables de cette classe.

4.1 Introduction

Motivations. L'objectif premier de ce chapitre est de donner une caractérisation des propriétés qui peuvent être vérifiées, enforcées, et testées lors de l'exécution d'un système. Le fait de pouvoir vérifier, enforcer ou tester une propriété à l'exécution dépend de l'évaluation de cette propriété productible par un moniteur à partir d'une séquence que celui-ci a lue. Cette évaluation dépend de la manière dont on interprète la séquence courante par rapport à la propriété (si celle-ci est satisfaite de manière finitaire). Cette évaluation peut aussi dépendre des continuations possibles de la séquence courante. Ainsi un moniteur ou un testeur doit produire une évaluation à chaque pas (à chaque lecture d'événement) de la séquence vue jusqu'à présent. Cette évaluation est une valeur d'un domaine de vérité [1]. Lorsqu'une évaluation définitive est produite par le moniteur celle-ci ne doit pas pouvoir être remise en cause par une exécution future du système. Nous considérons plusieurs domaines de vérité qui seront discrets, ce qui nous permet d'obtenir une caractérisation multivaluée de la notion de monitorabilité.

De façon indirecte, ce chapitre permet également de comprendre comment l'approche d'enforcement est reliée à l'approche (plus générique) de vérification. A notre connaissance, il n'existe pas de caractérisation exacte des propriétés que l'on souhaite valider lors de l'exécution d'un système. Les travaux précédents dans les domaines de la vérification ou l'enforcement de propriétés fournissent des bornes inférieures pour les espaces de propriétés que l'on peut traiter avec ces techniques.

L'intérêt de connaître "l'applicabilité" de ces méthodes nous semble intéressant dans le sens suivant. Étant donnée une propriété venant de la phase de spécification du système, selon son type (*i.e.*, la classe de la propriété), il est possible de mettre en œuvre plusieurs techniques pour améliorer la confiance que le système satisfera bien cette propriété lors de l'exécution de celui-ci.

Dans ce chapitre, nous donnons des définitions de monitorabilité, enforçabilité, et testabilité. Ces définitions peuvent sembler indépendantes du système ou programme considéré. Le lien avec ce dernier se fait grâce au vocabulaire commun Σ. Dans les cas de la vérification et d'enforcement à l'exécution, ce vocabulaire d'observation est obtenu (cf. Chapitre 2) par instrumentation du programme. Dans le cas du test, ce vocabulaire permet d'abstraire les séquences d'intéractions du testeur avec le système testé aux points de contrôle et d'observation.

Contributions. Les questions auxquelles nous essayons de répondre dans ce chapitre sont les suivantes :

1. Étant donné un mécanisme de validation de propriétés à l'exécution, quelles sont les propriétés qui peuvent être traitées par ce mécanisme ?
2. Existe-t-il une différence entre ces différents mécanismes ?

Ces questions ne sont pas originales en elles mêmes, mais nous proposons de nous y intéresser ici dans un cadre unifié : la classification *Safety-Progress* (cf. Chapitre 3) des propriétés. Les contributions de ce chapitre sont les suivantes :

- Nous améliorons et intégrons dans le même cadre les résultats existants [PZ06, BLS09, BLS07b] (et rappelés au Chapitre 2) liés à la caractérisation des propriétés monitorables selon la définition classique donnée par Pnueli et Zaks dans [PZ06].
- Nous proposons une définition alternative de monitorabilité tirant parti de la sémantique des séquences finies.
- Nous donnons une caractérisation exacte de l'ensemble des propriétés enforçables, et ceci indépendamment de tout mécanisme d'enforcement.
- Nous donnons une borne inférieure pour l'ensemble des propriétés testables, et ceci indépendamment de tout testeur. Cette caractérisation étend et corrige certains résultats établis précédemment dans [NGH93, Gra94].

4.2 Monitorabilité dans la classification *Safety-Progress*

Dans cette section nous commençons par revisiter la définition de monitorabilité classique dans le cadre de la classification *Safety-Progress*. Ce qui nous permettra de donner une meilleure caractérisation des propriétés monitorables selon cette définition. Ensuite, nous proposons d'étendre cette définition

1. En vérification à l'exécution, le domaine de vérité peut être de différentes sortes. Généralement, le domaine de vérité est le domaine booléen. La propriété est vraie ou fausse. Il peut être également plus riche, tout en restant discret. Mais il peut être également l'intervalle des réels $[0, 1]$ pour représenter une "probabilité de satisfaction".

classique de monitorabilité pour en faire une définition dépendant d'un domaine de vérité. Enfin nous proposons une nouvelle définition de monitorabilité, et déterminons l'espace des propriétés monitorables selon cette définition.

4.2.1 Selon la définition classique du monitoring

Définition classique de monitorabilité

L'objectif principal du monitoring, dans sa définition classique, est d'évaluer des propriétés infinitaires φ sur une séquence d'exécution (possiblement infinie) à partir de l'un de ses préfixes *(finis)*. Intuitivement, l'idée est de pouvoir détecter des verdicts, *i.e.*, trouver une évaluation, vis-à-vis de la satisfaction de propriétés infinitaires par le système à partir d'une observation finie de son comportement. Ce qui est formalisé comme suit :

Définition 22 (Détermination Positive/Négative [PZ06]). Une r-propriété $\Pi \subseteq \Sigma^* \times \Sigma^\omega$ est dite être :
- déterminée négativement par $\sigma \in \Sigma^*$ si $\neg\Pi(\sigma) \wedge \forall \mu \in \Sigma^\infty : \neg\Pi(\sigma \cdot \mu)$;
- déterminée positivement par $\sigma \in \Sigma^*$ si $\Pi(\sigma) \wedge \forall \mu \in \Sigma^\infty : \Pi(\sigma \cdot \mu)$.

Une r-propriété est déterminée négativement (resp. positivement) si la séquence courante ne satisfait pas (resp. satisfait) la propriété et si toutes ses futures continuations possibles ne la satisfont pas (resp. la satisfont). La signification pratique est la suivante : lorsqu'un moniteur observe un système pour vérifier une propriété, si celle-ci est déterminée négativement ou positivement, alors l'observation du système peut s'arrêter. Le verdict obtenu est définitif.

Définition 23 (Monitorabilité [PZ06] r-propriétés). Une r-propriété Π est :
- σ-monitorable, s'il existe un $\mu \in \Sigma^*$ (fini) *t.q.* Π est déterminée positivement ou négativement par $\sigma \cdot \mu$;
- monitorable, si elle est σ-monitorable pour chaque $\sigma \in \Sigma^*$.

Une r-propriété est monitorable si pour n'importe quelle séquence d'exécution, elle peut être déterminée négativement ou positivement par l'une des extensions de cette séquence.

Implicitement, le domaine de vérité sous-jacent est $\mathbb{B}_3 = \{\top, ?, \bot\}$. Les valeurs de vérité se relient à la détermination positive ou négative comme suit :
- La valeur "$\top$" (resp. "$\bot$") est utilisée pour exprimer la satisfaction (resp. violation) de la propriété lorsque celle-ci est déterminée positivement (resp. négativement).
- La valeur "?" est utilisée pour exprimer le fait qu'aucun verdict ne peut être produit (pour le moment).

Le domaine de vérité $\mathbb{B}_3$ est en fait un treillis. Ces éléments sont ordonnés [2] comme suit $\bot \sqsubset ? \sqsubset \top$. Les opérateurs booléens $\vee$ et $\wedge$ sont définis respectivement comme la borne supérieure et inférieure. Cela permet de composer les évaluations.

Remarquons que cette définition de monitorabilité est difficile à déterminer en pratique. Étant donné une r-propriété, la définition ne donne pas de moyen simple pour déterminer si cette propriété est monitorable, *i.e.*, savoir si pour toute séquence la propriété peut être déterminée positivement ou négativement par une des extensions de cette séquence.

Caractérisation des propriétés monitorables dans sa définition classique. Suivant la définition classique de monitoring, nous allons essayer de caractériser l'ensemble des propriétés de la hiérarchie *Safety-Progress* qui sont monitorables. Dans ce contexte, il est possible de montrer que l'ensemble des propriétés monitorables avec $\mathbb{B}_3$ contient strictement l'ensemble des propriétés d'obligation.

Dans la suite, pour un domaine de vérité $\mathbb{B}$, nous notons $MP(\mathbb{B})$ l'espace des propriétés monitorables avec le domaine de vérité $\mathbb{B}$. Nous commençons par donner un lemme dont nous aurons besoin dans la suite. Ce lemme stipule que les propriétés $MP(\mathbb{B}_3)$-monitorables sont fermées par conjonction et disjonction (en vue logique), ou de façon équivalente, par union et intersection (en vue langage).

2. Dans d'autres contextes, nous aurions pu ordonner les valeurs de vérité selon le "degré d'information" qu'elles représentent sur le système : $? \sqsubset \top$ et $? \sqsubset \bot$. Cependant, nous choisissons de les ordonner plutôt selon la "satisfiabilité" de la propriété ; ceci dans le but de composer facilement les évaluations avec les opérations booléennes.

Lemme γ (Combinaison booléennes de propriétés monitorables) : *L'ensemble $MP(\mathbb{B}_3)$ est fermé par les opérations booléennes positives. C'est-à-dire, étant données deux r-propriétés Π_1, Π_2, nous avons :*

$$\Pi_1, \Pi_2 \in MP(\mathbb{B}_3) \Rightarrow \Pi_1 \wedge \Pi_2 \in MP(\mathbb{B}_3)$$
$$\Pi_1, \Pi_2 \in MP(\mathbb{B}_3) \Rightarrow \Pi_1 \vee \Pi_2 \in MP(\mathbb{B}_3) \qquad \diamond$$

Preuve : Considérons deux r-propriétés $\Pi_1, \Pi_2 \in MP(\mathbb{B}_3)$.

- Preuve de $\Pi_1 \wedge \Pi_2 \in MP(\mathbb{B}_3)$. Il s'agit de montrer que $\Pi_1 \wedge \Pi_2$ est σ-monitorable pour toute séquence $\sigma \in \Sigma^*$. Soit $\sigma \in \Sigma^*$, montrons qu'il existe une extension $\mu \in \Sigma^*$ telle que $\Pi_1 \wedge \Pi_2$ soit négativement ou positivement déterminée par $\sigma \cdot \mu$.
  Comme Π_1 est monitorable, il existe μ_1 telle que Π_1 soit négativement ou positivement déterminée par $\sigma \cdot \mu_1$, c'est-à-dire que nous avons les deux possibilités suivantes :
  - $\exists \mu_1 \in \Sigma^*, \forall \mu'_1 \in \Sigma^* : \neg\Pi_1(\sigma \cdot \mu_1 \cdot \mu'_1)$, Π_1 est déterminée négativement par $\sigma \cdot \mu_1$
  - $\exists \mu_1 \in \Sigma^*, \forall \mu'_1 \in \Sigma^* : \Pi_1(\sigma \cdot \mu_1 \cdot \mu'_1)$, Π_1 est déterminée positivement par $\sigma \cdot \mu_1$

  Comme Π_2 est également monitorable, il existe μ_2 telle que Π_2 soit négativement ou positivement déterminée par $\sigma \cdot \mu_1 \cdot \mu_2$, c'est-à-dire que nous avons les deux possibilités suivantes :
  - $\exists \mu_2 \in \Sigma^*, \forall \mu'_2 \in \Sigma^* : \neg\Pi_2(\sigma \cdot \mu_1 \cdot \mu_2 \cdot \mu'_2)$, Π_2 est déterminée négativement par $\sigma \cdot \mu_1 \cdot \mu_2$
  - $\exists \mu_2 \in \Sigma^*, \forall \mu'_2 \in \Sigma^* : \Pi_2(\sigma \cdot \mu_1 \cdot \mu_2 \cdot \mu'_2)$, Π_2 est déterminée positivement par $\sigma \cdot \mu_2$

  Nous avons donc par combinaison, quatre possibilités, que nous regroupons en deux cas :
  - Distinguons le cas où Π_1 est déterminée positivement par $\sigma \cdot \mu_1$ et Π_2 déterminée positivement par $\sigma \cdot \mu_1 \cdot \mu_2$. Alors, en prenant $\mu = \sigma \cdot \mu_1 \cdot \mu_2$, nous avons que $\Pi_1 \wedge \Pi_2$ est déterminée positivement par μ. Ce qui donne le résultat recherché.
  - Dans les autres cas, il suffit de prendre également $\mu = \sigma \cdot \mu_1 \cdot \mu_2$ pour suffit pour montrer que $\Pi_1 \wedge \Pi_2$ est déterminée négativement par μ. Ce qui donne également le résultat recherché.
- La preuve de $\Pi_1 \vee \Pi_2 \in MP(\mathbb{B}_3)$ est similaire. ■

Théorème δ (Obligation(Σ) $\subset MP(\mathbb{B}_3)$) : *Les r-propriétés d'obligation sont strictement contenues dans l'ensemble des propriétés monitorables avec $\mathbb{B}_3$.*

Preuve : Comme vu précédemment dans le Chapitre 3, les r-propriétés d'obligation sont obtenues par combinaison booléennes de r-propriétés de safety et guarantee. Pour $k \in \mathbb{N}$, une r-propriété de k-obligation s'écrit $\bigcap_{i=1}^{k}(\mathsf{Safety}_i \cup \mathsf{Guarantee}_i)$, où Safety_i et $\mathsf{Guarantee}_i$ sont des r-propriétés de safety et guarantee. L'ensemble de toutes les r-propriétés de k-obligation pour $k \in \mathbb{N}$ est l'ensemble des r-propriétés d'obligation.

Soit $\Pi \in \mathsf{Obligation}(\Sigma)$, il existe $k \in \mathbb{N}$ *t.q.* $\Pi \in k\text{–}\mathsf{Obligation}(\Sigma)$. La preuve repose sur une induction simple sur k et utilise les faits suivants :

- Les propriétés de safety et guarantee sont monitorables. Ce que nous montrons ci-dessous [3] :
  - Soit $\Pi = (A_f(\psi), A(\psi))$ une r-propriété de safety, montrons que Π est monitorable. Soit $\sigma \in \Sigma^*$, montrons que Π est σ-monitorable. La preuve se fait en distinguant deux cas : soit il existe une continuation $\sigma' \in \Sigma^*$ de σ telle que $\neg\Pi(\sigma')$, soit il n'en n'existe pas. Dans le premier cas, nous avons que $\neg A_f(\psi)(\sigma')$, *i.e.*, σ' n'a pas tous ses préfixes appartenant à ψ. Alors évidemment toutes les continuations σ'' de σ' n'ont pas tous leurs préfixes dans ψ : $\forall \sigma'' \in \Sigma^* : \sigma' \preceq \sigma'' \Rightarrow \neg A_f(\psi)(\sigma'')$. Il s'en suit que $\forall \sigma'' \in \Sigma^* : \sigma \preceq \sigma' \preceq \sigma'' \Rightarrow \neg\Pi(\sigma'')$. C'est-à-dire que Π est déterminée négativement par σ'. Dans le deuxième cas, toutes les continuations de σ satisfont Π. C'est-à-dire que Π est positivement déterminée par $\sigma \cdot \epsilon_\Sigma$.
  - Soit $\Pi = (E_f(\psi), E(\psi))$ une r-propriété de guarantee, montrons que Π est monitorable. La preuve se conduit de manière similaire à la preuve précédente. Il suffit de considérer $\sigma \in \Sigma^*$ et de montrer qu'il existe une extension qui fait que Π soit négativement ou positivement déterminée par cette extension. Il suffit similairement de distinguer deux cas : il existe (ou non) une extension de σ qui satisfait la propriété.

3. La preuve peut également s'établir en examinant les restrictions syntaxiques d'un automate reconnaissant une propriété de safety ou guarantee : pour tout $\sigma \in \Sigma^*$ il existe une continuation μ *t.q.* cette propriété soit déterminée positivement ou négativement par $\sigma \cdot \mu$. Par exemple pour un automate reconnaissant une r-propriété de safety, quelque soit l'état de l'automate, il existe un chemin qui soit mène vers une composante fortement connexe de bons états ou une composante fortement connexe de mauvais états.

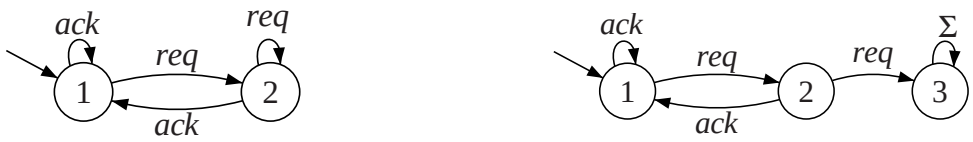


FIGURE 4.1 – r-propriétés de *response* non monitorable (gauche) et monitorable (droite)

- L'union et l'intersection de deux r-propriétés monitorables est monitorable (Lemme γ).
- L'Exemple 11 montre une r-propriété de *response* qui est monitorable, et montre ainsi que l'inclusion est stricte. ■

Nous avons donc étendu le résultat précédent (cf. Chapitre 2 Section 2.1, p. 17) établi par Bauer et ses coauteurs dans [BLS07c] déclarant que $\mathsf{Safety}(\Sigma) \cup \mathsf{Guarantee}(\Sigma) \subset MP(\mathbb{B}_3)$. En effet, dans [BLS07c], la classe des propriétés de guarantee est nommée la classe des propriétés de co-safety. Ce sont bien sur les mêmes classes de propriétés. De plus, l'ensemble des r-propriétés d'obligation est un sur-ensemble strict de l'union des ensembles des propriétés de safety et de guarantee (cf. Chapitre 3). Nous avons donc les inclusions suivantes :

$$\mathsf{Safety}(\Sigma) \cup \mathsf{Guarantee}(\Sigma) \subset \mathsf{Obligation}(\Sigma) \subset MP(\mathbb{B}_3)$$

Au delà des Obligations. Suivant la définition classique de monitorabilité (dans le domaine $\mathbb{B}_3$), il est possible de montrer qu'il existe des propriétés non monitorables et des propriétés monitorables pour les classes supérieures de la classe des r-propriétés d'obligation. Les deux r-propriétés suivantes sont des (pures) *response*[4], l'une n'est pas monitorable, l'autre l'est.

Exemple 10 (Propriété de *response* non monitorable [BLS07c]) La r-propriété de *response* suivante n'est pas monitorable.

"Chaque requête doit être acquittée"

Cette propriété est représentée et formalisée par l'automate de Streett de *response* représenté sur sur la gauche de la Fig. 4.1. L'automate de Streett est défini comme suit : son vocabulaire est $\{req, ack\}$ où *req* (resp. *ack*) représente l'évènement de requête (resp. d'acquittement), et la paire d'acceptation est telle que $R = \{1\}$ et $P = \emptyset$. Pour cette propriété, il existe deux limitations pour pouvoir monitorer avec le domaine de valeur considéré et cette définition de monitorabilité. D'abord, il est impossible de distinguer, dans le domaine $\mathbb{B}_3$, les séquences d'exécution finies correctes (terminant dans l'état 1) et incorrectes (terminant dans l'état 2) : les deux s'évaluent à "?". Deuxièmement, pour toutes les séquences finies, il n'est jamais possible de décider $\top$ ou $\bot$ car chaque séquence finie peut être étendue en une continuation infinie correcte ou incorrecte.

Exemple 11 (Propriété de *response* monitorable) La propriété de *response* suivante est monitorable.

"Chaque requête doit être acquittée et deux requêtes successives (sans acquittement) sont interdites"

Cette propriété peut être formalisée et représentée par l'automate de Streett de *response* représenté sur la droite de la Fig. 4.1 avec $R = \{1\}$, $P = \emptyset$, et le même vocabulaire que l'automate de l'Exemple 10. Intuitivement, étant donnée n'importe quelle séquence d'exécution, cette r-propriété peut toujours être déterminée négativement par l'une de ses extensions. En effet, l'état 3 est accessible depuis n'importe quel état, et lorsqu'une séquence atteint l'état 3 lors de son exécution sur cet automate, la propriété est déterminée négativement. Intuitivement la monitorabilité de cette propriété de response peut s'expliquer par le fait qu'elle contient une partie safety.

4. Nous pouvons nous convaincre assez facilement que cette r-propriété ne peut pas être une obligation. En effet, nous savons d'après [CP03] que les propriétés d'obligation sont exactement reconnues par les automates de Streett où la condition d'acceptation est définie en version "occurrence", au lieu d'une version "visite infinie" comme utilisée dans cette thèse. Étant donné la restriction syntaxique sur les automates de Streett d'obligation stipulant qu'il n'existe pas de transition d'un état $\overline{R_i}$ vers un état R_i, il faudrait un nombre infini d'ensembles R_i pour accepter toutes les séquences spécifiées par ces propriétés.

Au delà des r-propriétés de type obligation, l'espace des r-propriétés monitorables ne peut être caractérisé simplement comme une classe de la hiérarchie. Il existe des r-propriétés de *response* qui sont monitorables et d'autres qui ne le sont pas. Similairement, nous pouvons trouver des r-propriétés de persistence monitorables, et non-monitorables (*e.g.*, la r-propriété représentée sur la Fig. 4.6). Nous verrons cependant plus loin qu'il est possible de donner une caractérisation exacte dans la vue automate de la classification *Safety-Progress*.

Monitorabilité avec $\mathbb{B}_2$. En restreignant $\mathbb{B}_3$ à un domaine de valeurs de vérité de cardinal 2, on s'intéresse uniquement à la détermination positive ou négative. Ce qui réduit ainsi l'ensemble des propriétés monitorables. Cependant, il n'y a pas de caractérisation simple de cet espace de propriétés dans la hiérarchie *Safety-Progress*.

Intuitivement, on pourrait penser que avec $\mathbb{B}_2^{\perp} = \{?, \perp\}$, l'ensemble des r-propriétés monitorables serait l'ensemble des r-propriétés de safety. Mais en fait, il existe plusieurs r-propriétés de safety qui ne peuvent être déterminée négativement. La plus simple d'entre elle est la propriété "toujours vraie" $\mathit{true} = (\Sigma^*, \Sigma^\omega)$ qui ne peut bien évidemment ni être déterminée négativement ni même falsifiée. De plus, toutes les r-propriétés de safety qui sont valides pour les séquences d'exécution plus longues qu'un certain rang k ne sont pas $\sigma - \mathbb{B}_2^{\perp}$-monitorables lorsque $|\sigma| > k$. Pour ce type de r-propriétés, un moniteur produirait uniquement une séquence de " ?" lorsqu'il évaluerait cette propriété sur une séquence donnée.

De façon similaire, il existe pas mal de r-propriétés de guarantee qui ne peuvent être positivement déterminées, et ainsi ne sont pas monitorables avec $\mathbb{B}_2^{\top} = \{?, \top\}$.

Cependant, dans le Chapitre 5 Section 5.1, nous donnons un critère syntaxique sur les automates de Streett pour déterminer si une r-propriété (spécifiée par un automate de Streett donné) est monitorable sous ces conditions.

La notion de détermination négative ou positive nous semble intéressante car elle indique les cas où l'observation des continuations d'une séquence d'exécution n'est plus nécessaire lorsque l'on cherche à connaître la satisfaction vis-à-vis d'une propriété. En revanche l'exigence qu'une propriété doit toujours pouvoir être déterminée négativement ou positivement, exprimée dans la définition de monitorabilité, nous semble trop forte. Nous avions fait ce constat déjà dans [FFM09b]. Ainsi, nous proposons donc une définition alternative de monitoring qui semble mieux correspondre aux besoins d'observation d'un moniteur et aux implémentations existantes d'outils de monitoring. La définition de détermination négative ou positive nous semble toutefois intéressante dans un contexte de test. Ce que nous verrons dans la Section 4.4.

4.2.2 Une définition alternative du monitoring

L'intérêt des définitions précédentes de monitorabilité est due à deux faits : le domaine de vérité considéré est 2-valué ou 3-valué et l'objectif est la détection de verdicts de propriétés infinitaires à partir d'observation finies. Bien qu'il soit possible de donner une sémantique à toutes les propriétés de reactivity avec un domaine de vérité de cardinal 2 ou 3, la question est de savoir si ces valeurs ont du sens pour certaines propriétés lorsque l'on se place dans un contexte de monitoring.

Comme remarqué dans [BLS07c, LS08], il semble intéressant d'examiner plus en détails l'espace des propriétés monitorables, et de répondre plus précisément à la question "quel verdict produire si l'exécution du programme s'arrête ici ?". Ce qui suppose une meilleure distinction pour les séquences finies qui s'évaluent à "?" dans les domaines de vérité de cardinaux 2 ou 3. En d'autres termes, nous appuyons le fait que les séquences finies ont du sens par elles mêmes. Et il nous semble intéressant d'attribuer un verdict à ces séquences même si toutes leurs futures continuations ne s'évaluent pas de la même manière (et rendent la propriété négativement ou positivement déterminée).

Dans [BLS09], les auteurs proposent de considérer une domaine de vérité à quatre valeurs $\mathbb{B}_4 = \{\top, \top_p, \perp_p, \perp\}$. La valeur de vérité $\top_p$ (resp. $\perp_p$) dénote "presumably true" (resp. "presumably false") et elle exprime "la satisfaction (resp. violation) vis-à-vis de Π si l'exécution du programme s'arrête ici". Le domaine de vérité $\mathbb{B}_4$ est également un treillis où les valeurs de vérité sont ordonnées comme suit : $\perp \sqsubset \perp_p \sqsubset \top_p \sqsubset \top$. Les opérateurs $\vee$ et $\wedge$ pour ce domaine de valeur de vérité, comme pour $\mathbb{B}_3$, se traduisent en borne supérieur et inférieure respectivement. Utiliser $\mathbb{B}_4$ amène à une définition alternative de monitoring [5]. Cette définition tire parti de l'évaluation des séquences finies que nous avons définie dans

5. Notons qu'il serait encore possible de raffiner les valeurs $\top_p$ et $\perp_p$. Cette approche peut se généraliser à un treillis booléen. Une motivation serait de définir différents niveaux de satisfaction ou de non-satisfaction de la propriété. Par

Pour $\mathbb{B}_2^{\perp}$:

$[\![\Pi]\!]_{\mathbb{B}_2^{\perp}}(\sigma) = \perp$ si $\neg\Pi(\sigma) \wedge \forall\mu \in \Sigma^{\infty} : \neg\Pi(\sigma \cdot \mu)$,

$[\![\Pi]\!]_{\mathbb{B}_2^{\perp}}(\sigma) = ?$ dans les autres cas.

Pour $\mathbb{B}_2^{\top}$:

$[\![\Pi]\!]_{\mathbb{B}_2^{\top}}(\sigma) = \top$ si $\Pi(\sigma) \wedge \forall\mu \in \Sigma^{\infty} : \Pi(\sigma \cdot \mu)$,

$[\![\Pi]\!]_{\mathbb{B}_2^{\top}}(\sigma) = ?$ dans les autres cas.

Pour $\mathbb{B}_3$:

$[\![\Pi]\!]_{\mathbb{B}_3}(\sigma) = \perp$ si $\neg\Pi(\sigma) \wedge \forall\mu \in \Sigma^{\infty} : \neg\Pi(\sigma \cdot \mu)$,

$[\![\Pi]\!]_{\mathbb{B}_3}(\sigma) = \top$ si $\Pi(\sigma) \wedge \forall\mu \in \Sigma^{\infty} : \Pi(\sigma \cdot \mu)$,

$[\![\Pi]\!]_{\mathbb{B}_3}(\sigma) = ?$ dans les autres cas.

Pour $\mathbb{B}_4$:

$[\![\Pi]\!]_{\mathbb{B}_4}(\sigma) = [\![\Pi]\!]_{\mathbb{B}_3}(\sigma)$ si $[\![\Pi]\!]_{\mathbb{B}_3}(\sigma) = \perp$ ou $[\![\Pi]\!]_{\mathbb{B}_3}(\sigma) = \top$,

$[\![\Pi]\!]_{\mathbb{B}_4}(\sigma) = \top_p$ si $[\![\Pi]\!]_{\mathbb{B}_3}(\sigma) = ?$ et $\Pi(\sigma)$

$[\![\Pi]\!]_{\mathbb{B}_4}(\sigma) = \perp_p$ si $[\![\Pi]\!]_{\mathbb{B}_3}(\sigma) = ?$ et $\neg\Pi(\sigma)$

FIGURE 4.2 – Évaluation d'une r-propriété dans les différents domaines de vérité

le cadre *Safety-Progress* (cf. Chapitre 3).

Évaluation d'une r-propriété dans un domaine de vérité. Nous introduisons tout d'abord, étant donnée une r-propriété, comment nous évaluons une séquence d'exécution dans un domaine de vérité.

DÉFINITION 24 (ÉVALUATION DE PROPRIÉTÉ DANS UN DOMAINE DE VÉRITÉ). Pour chaque domaine de vérité $\mathbb{B}$ que nous avons considéré jusqu'à présent, nous définissons les fonctions d'évaluation $[\![\cdot]\!]_{\mathbb{B}}(\cdot) : (\Sigma^* \times \Sigma^{\omega}) \times \Sigma^* \to \mathbb{B}$ comme exposé sur la Fig. 4.2.

Lors de l'évaluation avec le domaine $\mathbb{B}_2^{\perp}$, on distingue la situation où une séquence ainsi que toutes ses continuations sont fausses (évaluation $\perp$) ; les autres situations sont confondues en produisant l'évaluation "?". Similairement, avec $\mathbb{B}_2^{\top}$, l'évaluation $\top$ est produite lorsqu'une séquence ainsi que toutes ses continuations satisfont la r-propriété et "?" dans les autres cas. L'évaluation avec le domaine $\mathbb{B}_3$ correspond à celle produite dans la définition classique de monitorabilité. L'évaluation avec le domaine $\mathbb{B}_4$ raffine l'évaluation avec $\mathbb{B}_3$ lorsque cette dernière produisait "?".

Notons que l'évaluation dans $\mathbb{B}_3$ correspond à la sémantique de la logique temporelle LTL_3 [BLS07a]. De plus, l'évaluation dans $\mathbb{B}_4$ correspond à la sémantique de logique RV-LTL [BLS09] introduite récemment.

Une définition alternative de la monitorabilité. Intuitivement, la notion de monitorabilité d'une propriété Π repose sur la capacité d'un moniteur donné de distinguer les bonnes et mauvaises séquences d'exécution finies.

DÉFINITION 25 (MONITORABILITÉ). Une r-propriété $\Pi = (\phi, \varphi)$ est dite monitorable sur le domaine de vérité $\mathbb{B}$, ou $\mathbb{B}$-monitorable ssi

$$\forall\sigma_{good} \in \phi, \forall\sigma_{bad} \in \overline{\phi} : [\![\Pi]\!]_{\mathbb{B}}(\sigma_{good}) \neq [\![\Pi]\!]_{\mathbb{B}}(\sigma_{bad})$$

Nous notons $MP^*(\mathbb{B})$, selon cette définition, l'ensemble des propriétés monitorables avec le domaine de vérité $\mathbb{B}$.

Ainsi une r-propriété est monitorable avec le domaine de vérité $\mathbb{B}$ ssi les séquences d'exécution finies mauvaises et bonnes peuvent être évaluées de façons distinctes. Remarquons que la définition ne dépend pas directement de la partie infinitaire de la r-propriété. La partie infinitaire de la propriété vérifiée est utilisée par la fonction d'évaluation de la r-propriété.

Lemme δ ($MP^*(\mathbb{B}_3)$ et propriétés de reactivity qui ne sont pas des safety ou guarantee) : *Il n'y a pas de propriété de reactivity non safety et non guarantee qui soit monitorable au sens de la Définition 25 avec $\mathbb{B}_3$. Formellement :*

$$MP^*(\mathbb{B}_3) \subseteq (\text{Safety}(\Sigma) \cup \text{Guarantee}(\Sigma)) = \emptyset$$

◇

exemple, on pourrait distinguer différentes valeurs "presumably false" pour refléter les différents degrés de panne d'un système.

La preuve de ce théorème est donnée dans l'Annexe A Section A.2.

Théorème ε (Caractérisation multivaluée de la monitorabilité) : *Les ensembles de r-propriétés monitorables selon les domaines de vérité considérés jusqu'à présent sont les suivants :*

(i) $MP^*(\mathbb{B}_2^\perp) = \text{Safety}(\Sigma)$

(ii) $MP^*(\mathbb{B}_2^\top) = \text{Guarantee}(\Sigma)$

(iii) $\text{Safety}(\Sigma) \cup \text{Guarantee}(\Sigma) = MP^*(\mathbb{B}_3)$

(iv) $MP^*(\mathbb{B}_4) = \text{Reactivity}(\Sigma)$

Preuve : Nous prouvons chacun des 4 faits annoncés successivement. Pour les égalités, nous montrons à chaque fois l'inclusion dans les deux sens des ensembles concernés.

Preuve de (i).

- Soit $\Pi = (\phi, \varphi) \in \text{Safety}(\Sigma)$, montrons que $\Pi \in MP^*(\mathbb{B}_2^\perp)$. Comme $\Pi \in \text{Safety}(\Sigma)$, il existe une propriété finitaire $\psi \subseteq \Sigma^*$, *t.q.* $\Pi = (A_f(\psi), A(\psi))$. Considérons $\sigma_{good} \in \phi$ et $\sigma_{bad} \in \overline{\phi}$, il s'agit de montrer que l'évaluation dans $\mathbb{B}_2^\perp$ de ces deux séquences sont différentes. D'une part, nous avons que $\Pi(\sigma_{good})$ (car $\sigma_{good} \in \phi$) et donc $[\![\Pi]\!]_{\mathbb{B}_2^\perp}(\sigma_{good}) =?$. D'autre part, nous avons $\neg\Pi(\sigma_{bad})$ et $\sigma_{bad} \notin A_f(\psi)$ (car $\sigma_{bad} \notin \phi$). En utilisant la Propriété 1 du Chapitre 3, nous avons $\forall \mu \in \Sigma^\infty : \neg\Pi(\sigma_{bad} \cdot \mu)$, *i.e.*, $[\![\Pi]\!]_{\mathbb{B}_2^\perp}(\sigma_{bad}) = \perp$.
- Soit $\Pi \in MP^*(\mathbb{B}_2^\perp)$, montrons que $\Pi \in \text{Safety}(\Sigma)$. D'après la caractérisation des propriétés de safety donnée dans la Propriété 2 (Chapitre 3, Section 3.4), montrer que Π est une r-propriété de safety revient à montrer qu'elle vérifie $\Pi = (A_f(Pref(\Pi)), A(Pref(\Pi)))$. Ce que nous faisons en montrant l'inclusion dans les deux sens.
  - $\Pi \cap \Sigma^* \subseteq A_f(Pref(\Pi))$ est évident car pour tout mot $\sigma \in \Pi$, σ a tous ses préfixes dans $Pref(\Pi)$. Il en est de même pour $\Pi \cap \Sigma^\omega \subseteq A(Pref(\Pi))$.
  - Montrons $A_f(Pref(\Pi)) \subseteq \Pi$. Soit $\sigma \in A_f(Pref(\Pi))$, montrons que $\sigma \in \Pi$. Comme $\sigma \in A_f(Pref(\Pi))$, tous les préfixes de σ appartiennent à $Pref(\Pi)$. C'est-à-dire que tous les préfixes de σ sont les préfixes d'un mot dans Π. Soit σ_{min} le plus petit de ces mots. Nous distinguons deux cas. Soit $\sigma_{min} = \sigma$, ce qui donne $\sigma \in \Pi$. Soit $\sigma \prec \sigma_{min}$. On a forcément $[\![\Pi]\!]_{\mathbb{B}_2^\perp}(\sigma_{min}) =?$, sinon on ne pourrait pas avoir $\sigma_{min} \in \phi$. En utilisant la remarque formulée au début de cette preuve, on obtient $\sigma \in \phi$ et donc $\sigma \in \Pi$. Le même raisonnement peut être conduit pour montrer que $A(Pref(\Pi)) \subseteq \Pi \cap \Sigma^\omega$.

  Au final d'après la définition des r-propriétés (Section 3.3, Définition 8, p. 38), nous avons que Π peut s'écrire $\Pi = (A_f(Pref(\Pi)), A(Pref(\Pi)))$, ce qui donne le résultat attendu.

Preuve de (ii).

- Le raisonnement pour montrer que $\text{Guarantee}(\Sigma) \subseteq MP^*(\mathbb{B}_2^\top)$ est similaire au raisonnement pour montrer que $\text{Safety}(\Sigma) \subseteq MP^*(\mathbb{B}_2^\perp)$. Il suffit de montrer que toutes les mauvaises séquences s'évaluent à "?". De plus, toutes les bonnes séquences s'évaluent à $\top$. En effet, une fois qu'une séquence satisfait une r-propriété de guarantee, toutes ses continuations la satisfont.
- Montrer que $MP^*(\mathbb{B}_2^\top) \subseteq \text{Guarantee}(\Sigma)$ se fait de façon analogue au raisonnement montrant que $MP^*(\mathbb{B}_2^\perp) \subseteq \text{Safety}(\Sigma)$, en montrant que si $\Pi \in MP^*(\mathbb{B}_2^\top)$, alors Π vérifie $\Pi = E(\overline{Pref(\overline{\Pi})})$.

Preuve de (iii)

- La preuve de $\text{Safety}(\Sigma) \cup \text{Guarantee}(\Sigma) \subseteq MP^*(\mathbb{B}_3)$ est évidente en remarquant que $MP^*(\mathbb{B}_2^\perp) \subset MP^*(\mathbb{B}_3)$ et $MP^*(\mathbb{B}_2^\top) \subset MP^*(\mathbb{B}_3)$.
- Le fait que $MP^*(\mathbb{B}_3) \subseteq \text{Safety}(\Sigma) \cup \text{Guarantee}(\Sigma)$ est une conséquence du Lemme δ.

Preuve de (iv). La preuve est évidente en remarquant que toute r-propriété peut être évaluée en distinguant bien les bonnes et les mauvaises séquences. Autrement dit, toute r-propriété de reactivity peut s'évaluer "de manière consistente" avec $\mathbb{B}_4$. En effet, une bonne séquence σ_{good} s'évalue à $\top_p$ ou $\top$ en fonction de ses continuations. Une mauvaise séquence σ_{bad} s'évalue à $\perp_p$ ou $\perp$ en fonction de ses continuations. Ajouter des valeurs de vérité ne fait que raffiner le verdict produit par le moniteur sous-jacent. ∎

L'Exemple 12 illustre une r-propriété d'obligation qui n'est pas $\mathbb{B}_3$-monitorable.

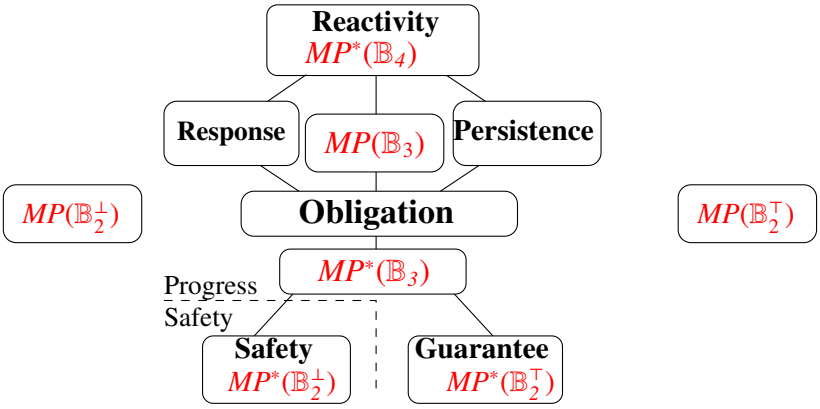


FIGURE 4.3 – r-propriétés monitorables dans la classification *Safety-Progress*

Exemple 12 (Monitoring d'une propriété d'obligation) Considérons la r-propriété exprimée en LTL :

$$\Pi = (\Box\, p) \vee (\Diamond\, q)$$

déclarant que le prédicat d'état p doit *toujours* être vrai ou alors le prédicat d'état q doit inévitablement être vrai. C'est une r-propriété d'obligation, elle ne peut donc pas être monitorée avec $\mathbb{B}_2^{\perp}$ ni $\mathbb{B}_2^{\top}$. Examinons si elle peut être monitorée avec $\mathbb{B}_3$. Considérons les séquences d'exécution suivantes : $\sigma_{good} = \{p\} \cdot \{p\}$ et $\sigma_{bad} = \emptyset \cdot \{p\}$. Dans $\mathbb{B}_3$, nous avons $[\![\Pi]\!]_{\mathbb{B}_3}(\sigma_{good}) = [\![\Pi]\!]_{\mathbb{B}_3}(\sigma_{bad}) =?$. Ainsi, Π n'est pas $\mathbb{B}_3$-monitorable au sens de la Définition 25.

En revanche, Π est $\mathbb{B}_4$-monitorable et $\forall \sigma_{good} \in \Pi, \forall \sigma_{bad} \in \overline{\Pi}, [\![\Pi]\!]_{\mathbb{B}_4}(\sigma_{good}) = \top_p$ et $[\![\Pi]\!]_{\mathbb{B}_4}(\sigma_{bad}) = \perp_p$.

Cette définition de monitorabilité a l'avantage d'identifier les propriétés qui ne devraient pas être monitorées avec un domaine de vérité qui "n'est pas assez fin". En effet, la dernière propriété montre que si l'on construisait un moniteur pour une telle propriété avec le domaine de vérité $_3$, le moniteur produirait une évaluation à "?" pour à la fois des séquences correctes et incorrectes par rapport à la propriété. Ce qui nous semble indésirable.

Nous montrerons dans le Chapitre 5 Section 5.2 que, pour une séquence d'exécution finie donnée σ, $[\![\Pi]\!]_{\mathbb{B}_4}(\sigma)$ peut être obtenu simplement par un calcul sur les états d'un automate de Streett reconnaissant Π.

REMARQUE 5 ("PASSAGE À LA LIMITE POUR LES VERDICTS FAIBLES") Il peut être intéressant de remarquer que l'interprétation des séquences avec des "verdicts faibles" ($\perp_p$, $\top_p$) s'étend aux séquences infinies, selon la classe de propriétés considérée. Considérons une séquence infinie $\sigma \in \Sigma^{\omega}$.

- pour une propriété de safety Π, $(\forall i \in \mathbb{N} : [\![\Pi]\!](\sigma_{\dots i}) = \top_p) \Leftrightarrow \Pi(\sigma)$
- pour une propriété de guarantee Π, $(\forall i \in \mathbb{N} : [\![\Pi]\!](\sigma_{\dots i}) = \perp_p) \Leftrightarrow \neg\Pi(\sigma)$
- pour une propriété de *response* Π, $(\overset{\infty}{\exists}\, i \in \mathbb{N} : [\![\Pi]\!](\sigma_{\dots i}) = \top_p) \Leftrightarrow \Pi(\sigma)$
- pour une propriété de persistence Π, $(\overset{\infty}{\exists}\, i \in \mathbb{N} : [\![\Pi]\!](\sigma_{\dots i}) = \perp_p) \Leftrightarrow \neg\Pi(\sigma)$

où $\overset{\infty}{\exists}$ signifie "il existe une infinité de". *

Les résultats de cette section sont résumés sur la Fig. 4.3 :

- Les classes de propriétés monitorables, dans la définition classique, sont :
  - $MP(\mathbb{B}_2^{\top})$,
  - $MP(\mathbb{B}_2^{\perp})$,
  - et $MP(\mathbb{B}_3)$.

  Les classes $MP(\mathbb{B}_2^{\top})$ et $MP(\mathbb{B}_2^{\perp})$ ne peuvent être comparées directement avec aucune autre classe. La classe $MP(\mathbb{B}_3)$ contient strictement la classe des propriétés d'obligation.
- Les classes de propriétés monitorables, dans la définition que nous avons introduite, sont :
  - $MP^*(\mathbb{B}_2^{\top})$, $MP^*(\mathbb{B}_2^{\perp})$,
  - $MP^*(\mathbb{B}_3)$,
  - et $MP^*(\mathbb{B}_4)$.

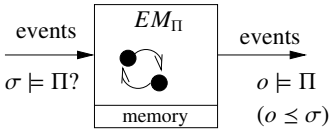


FIGURE 4.4 – Schéma de principe de l'enforcement

La classe $MP^*(\mathbb{B}_2^\top)$ (resp. $MP^*(\mathbb{B}_2^\perp)$) est exactement la classe des r-propriétés de safety (resp. de guarantee). La classe $MP^*(\mathbb{B}_3)$ est strictement contenue dans la classe des r-propriétés d'obligation. Enfin, la classe $MP^*(\mathbb{B}_4)$ est exactement la classe des r-propriétés de reactivity.

4.3 Enforçabilité par rapport à la classification *Safety-Progress*

Précédemment (Chapitre 2, Section 2.2), nous avons vu que les espaces de propriétés enforçables des approches existantes étaient caractérisés selon le mécanisme d'enforcement utilisé. Dans ces approches, les mécanismes utilisés répondent aux contraintes de correction et transparence. Et la classe des propriétés enforçables proposée dans ces approches était celle qui était enforçable à l'aide des mécanismes introduits.

Nous choisissons ici de prendre une approche alternative. En effet, nous avançons le fait qu'il est possible de caractériser l'espace des propriétés enforçables indépendamment du mécanisme utilisé. Nous allons donner une borne supérieure à l'espace des propriétés enforçables par tout mécanisme répondant aux exigences mentionnées plus haut.

4.3.1 Critères d'enforcement

Le problème de l'enforcement peut être représenté sur la Fig. 4.4. Un moniteur d'enforcement lit une séquence d'entrée σ et doit produire une séquence de sortie o. Cette séquence d'entrée peut venir d'un programme, mais aussi représenter les actions d'un utilisateur ou d'un protocole. Il dispose pour cela d'une mémoire finie, mais dont la taille est non bornée avant l'exécution. Cette mémoire lui sert à mémoriser potentiellement des événements de la séquence d'entrée avant des les "relâcher" en sortie. Rappelons les contraintes que nous exigeons sur un tel mécanisme d'enforcement EM_Π dédié à une r-propriété Π. Ces contraintes donnent une relation qui doit toujours être vérifiée entre les séquences d'entrée, de sortie, et la r-propriété :

Soundness (correction) : les séquences de sorties o sont correctes ;

Transparency (transparence) : les séquences d'entrées σ correctes produites par le programme ne doivent pas être modifiées, *i.e.*, la sortie o est équivalente [6] à l'entrée σ.

Une conséquence de la combinaison de la correction et de la transparence est qu'une r-propriété (ϕ, φ) sera considérée comme *enforçable* seulement si chaque séquence incorrecte a un plus grand préfixe correct. Autrement dit, toute séquence incorrecte doit avoir un nombre fini de préfixes corrects [7]. Cette demande de transparence peut se traduire au niveau des r-propriétés. Mais aussi dans la vue automate des r-propriétés, sans même connaître l'espace exact des r-propriétés enforçables. Ainsi, nous donnons un critère d'enforcement sur les r-propriétés. Puis, nous traduisons ce critère en une condition suffisante dans la vue automate des r-propriétés.

DÉFINITION 26 (CRITÈRE D'ENFORCEMENT SUR UNE r-PROPRIÉTÉ). Une r-propriété (ϕ, φ) est dite enforçable *ssi*

$$\forall \sigma \in \Sigma^\omega : \neg\varphi(\sigma) \Rightarrow (\exists \sigma' \in \Sigma^* : \sigma' \prec \sigma, \forall \sigma'' \in \Sigma^* : \sigma' \prec \sigma'' \prec \sigma \Rightarrow \neg\phi(\sigma'')) \tag{4.1}$$

6. Dans le cas général, on peut considérer une relation d'équivalence sur les séquences de Σ, ou en particulier la relation d'égalité. Dans le Chapitre 6, nous considérerons les deux possibilités

7. Notons que ce critère diffère de l'existence d'un bad préfixe (Chapitre 2, Section 2.1). Les bad préfixes sont les séquences qui ne peuvent être étendues vers des séquences (finies ou infinies) correctes.

Ce critère d'enforcement peut se traduire dans la vue automate de la classification *Safety-Progress* comme suit : Une r-propriété Π spécifiée par un automate de Streett $\mathcal{A}_\Pi$ est dite enforçable si chaque composante fortement connexe (SCC, Strongly Connected Component) accessible d'états $\overline{R}$ contient (seulement) soit des états P soit des états $\overline{P}$. Nous notons $\mathcal{S}(\mathcal{A}_\Pi)$ l'ensemble des SCCs accessibles de $\mathcal{A}_\Pi$.

Définition 27 (Critère suffisant d'enforcement dans la vue automate). Étant donné $\mathcal{A}_\Pi$ un m-automate spécifiant Π une r-propriété, Π est dite enforçable *ssi*

$$\forall i \in [1, m], \forall s \in \mathcal{S}(\mathcal{A}_\Pi) : s \subseteq \overline{R_i} \Rightarrow (s \subseteq \overline{P_i} \vee s \subseteq P_i) \tag{4.2}$$

Le critère d'enforcement de la Définition 26 est équivalent (*i.e.*, est également un critère nécessaire) au critère 27 dans la vue automate pour les classes de base, *i.e.*, le critère est aussi nécéssaire. Ceci est déclaré par la propriété exposée ci-dessous.

Propriété 5 (Équivalence entre les critères d'enforcement (pour les classes de bases)) : *Pour une r-propriété $\Pi = (\phi, \varphi)$ d'une classe de base, qui est reconnue par un automate de Streett $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \Sigma, \rightarrow^{\mathcal{A}_\Pi} \{(R, P)\}$, nous avons que :*

$$(4.1) \Leftrightarrow (4.2) \qquad \diamond$$

Et pour les classes de base : $(4.2) \Leftrightarrow \forall s \in \mathcal{S}(\mathcal{A}_\Pi) : s \subseteq \overline{R} \Rightarrow (s \subseteq \overline{P} \vee s \subseteq P)$

Preuve : Cette preuve repose sur le calcul des composantes fortement connexes [Tar72] d'un automate de Streett. La preuve se fait en deux étapes, en prouvant l'implication dans les deux sens.

Preuve de (4.1) $\Rightarrow$ (4.2)**.** Considérons une SCC de $\mathcal{A}_\Pi$ contenant seulement des états $\overline{R}$. Nous allons montrer (4.2) par l'absurde. Supposons qu'il existe deux états q, q' de cette SCC *t.q.* $q \in P$ et $q' \notin P$. Comme q et q' sont dans une SCC, il existe un chemin depuis q vers q' et depuis q' vers q dans $\mathcal{A}_\Pi$. Ainsi, il existerait une séquence d'exécution infinie σ *t.q.* le run de σ sur $\mathcal{A}_\Pi$ contient des occurrences infinies de q et q'. Comme cette SCC est faite d'états $\overline{R}$, σ n'est pas acceptée par $\mathcal{A}_\Pi$ (car $q \in P \wedge q' \notin P \Rightarrow vinf(\sigma, \mathcal{A}_\Pi) \nsubseteq P$), *i.e.*, $\neg\varphi(\sigma)$. Cependant, σ a un nombre infini de "good" préfixes : tous les préfixes dont le run termine dans un état R. Selon (4.1), la r-propriété Π ne serait pas enforçable. Ceci est contradictoire avec notre supposition initiale et nous donne le résultat attendu.

Preuve de (4.2) $\Rightarrow$ (4.1)**.** Considérons $\sigma \in \Sigma^\omega$ *t.q.* $\neg\varphi(\sigma)$. Comme $\mathcal{A}_\Pi$ est un automate d'états fini, le run de σ sur $\mathcal{A}_\Pi$ visite une (dernière) SCC infiniment souvent et peut être exprimée :

$$run(\sigma, \mathcal{A}_\Pi) = q_0 \cdots q_i \cdots = q_0 \cdots q_{k-1} \cdot (q_k \cdots q_{k+l})^n$$

où $q_k, \ldots, q_{k+l}$ sont les états visités infiniment souvent, *i.e.*, $vinf(\sigma, \mathcal{A}_\Pi) = \{q_k, \ldots, q_{k+l}\}$.
Comme $\mathcal{A}_\Pi$ reconnaît Π, σ n'est pas acceptée par $\mathcal{A}_\Pi$, nous avons donc :

$$vinf(\sigma, \mathcal{A}_\Pi) \cap R = \emptyset \wedge vinf(\sigma, \mathcal{A}_\Pi) \nsubseteq P$$

De $vinf(\sigma, \mathcal{A}_\Pi) \cap R = \emptyset$, nous déduisons $vinf(\sigma, \mathcal{A}_\Pi) \subseteq \overline{R}$. Comme $vinf(\sigma, \mathcal{A}_\Pi)$ est une SCC non maximale de $\mathcal{A}_\Pi$, alors en utilisant (4.2), nous en déduisons que soit $vinf(\sigma, \mathcal{A}_\Pi) \subseteq P$ soit $vinf(\sigma, \mathcal{A}_\Pi) \subseteq \overline{P}$. Or, $vinf(\sigma, \mathcal{A}_\Pi) \subseteq P$ est impossible car σ n'est pas acceptée par $\mathcal{A}_\Pi$ (cf. plus haut). Il nous reste donc que $vinf(\sigma, \mathcal{A}_\Pi) \subseteq \overline{P}$. Nous en déduisons pour les états q_i apparaissant dans le run de σ sur $\mathcal{A}_\Pi$ que $\forall i \geq k : q_i \notin P$. Et par suite, en utilisant le critère d'acceptation des séquences finies d'un automate de Streett : $\forall i \geq k : \neg\Pi(\sigma_{\ldots i})$. Ce qui montre que Π est enforçable. ■

Propriété 6 (Comparaison des critères d'enforcement pour les classes composées) : *Considérons une r-propriété $\Pi = (\phi, \varphi)$, spécifiée par un automate de Streett $(Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \Sigma, \rightarrow^{\mathcal{A}_\Pi}, \{(R, P)\})$, nous avons que :*

$$(4.1) \Leftrightarrow (4.2), \text{ pour les } r\text{-propriétés d'obligation}$$
$$(4.1) \Leftarrow (4.2), \text{ pour les } r\text{-propriétés de reactivity} \qquad \diamond$$

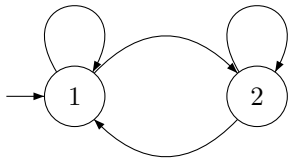


FIGURE 4.5 – Automate de 2-reactivity pour lequel (4.1) $\not\Rightarrow$ (4.2)

PREUVE : Nous donnons une idée de la preuve pour ces classes de propriétés.

Pour les r-propriété d'obligation. De façon similaire à la preuve de la Propriété 5, la preuve repose sur le fait que dans un automate de m-obligation, pour $i \in [1, m]$, il n'y a pas de transition depuis les états R_i vers les états $\overline{R_i}$, et pas de transition depuis les états $\overline{P_i}$ vers les états P_i.

Pour les r-propriétés de reactivity. Considérons $\sigma \in \Sigma^\omega$ *t.q.* $\neg\varphi(\sigma)$. De façon similaire à la preuve de la Propriété 5 (direction $\Leftarrow$), le run de σ sur $\mathcal{A}_\Pi$ visite une SCC infiniment souvent et peut être exprimée :

$run(\sigma, \mathcal{A}_\Pi) = q_0 \cdots q_{k-1} \cdot (q_i + \cdots + q_{i+l})^n$ avec $k \leq i \wedge l \leq |Q|$.

Les états $q_i, \ldots, q_{i+l}$ sont les derniers états visités par le run de σ sur $\mathcal{A}_\Pi$, *i.e.*, ce sont les états visités infiniment souvent. De plus, nous savons que $\forall i \leq j \leq i + l : q_j \in P \vee \forall i \leq j \leq i + l : q_j \in \overline{P}$. Nous avons alors $\forall \sigma' \in \Sigma^* : \sigma_{\ldots k} \preceq \sigma', \neg\Pi(\sigma')$. En effet, cela voudrait dire autrement que $\forall i \in [1, m], \forall j \geq k : q_j \in P_i$. Ce qui nous aurait conduit à avoir $\varphi(\sigma)$ en utilisant le critère d'acceptation des séquences infinies sur les automates de Streett (Définition 16, p. 50). ■

REMARQUE 6 Nous donnons un exemple d'automate de Streett de 2-reactivity pour lequel (4.1) $\not\Rightarrow$ (4.2). Cet automate est représenté sur la Fig. 4.5. Il possède deux états, l'alphabet importe peu, et les paires d'acceptation (R_1, P_1) et (R_2, P_2) sont définies comme suit :

$R_1 = \emptyset,\ P_1 = \emptyset,$

$R_2 = \emptyset,\ P_2 = \{1\}.$

La propriété reconnue par cet automate de Streett vérifie bien la condition (4.1), mais pas la condition (4.2). *

L'ensemble des r-propriétés enforçables est noté EP. Nous allons maintenant caractériser cet ensemble par rapport à la classification *Safety-Progress*. Bien que nous allons montrer que l'ensemble des r-propriétés enforçables est exactement l'ensemble des r-propriétés de *response*, le critère d'enforçabilité sur les automates reste utile. En effet, il fournit une condition suffisante sur les automates de Streett. Car ces automates peuvent ne pas être exprimés "de façon minimale" (en terme d'ensemble de paires d'acceptation). Étant donné n'importe quel automate de Streett, nous avons une procédure de décision syntaxique pour déterminer si la r-propriété reconnue par cet automate est enforçable.

4.3.2 Caractérisation des propriétés enforçables

Utilisant les critères précédemment définis, nous allons caractériser en deux temps l'espace des r-propriétés enforçables.

1. Nous commençons d'abord par prouver que les r-propriétés de *response* sont enforçables. Ce qui donne un sous-ensemble de l'espace des r-propriétés enforçables.
2. Ensuite, nous resserrons cette borne en montrant qu'elle est également une borne supérieure. Pour cela, nous donnons d'abord un exemple de r-propriété de persistence (la classe duale) qui n'est pas enforçable. Ce qui permet de donner l'intuition sur la limite d'enforcement. Puis nous montrons qu'il n'existe pas de r-propriété de persistence pure qui soit enforçable.

Borne inférieure

Théorème ζ (Les r-propriétés de *response* sont enforçables) : *L'ensemble des r-propriétés de response est inclus dans l'ensemble des r-propriétés enforçables*

$$Response(\Sigma) \subseteq EP$$

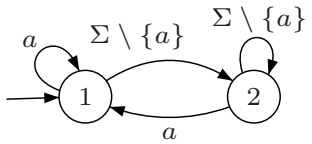


FIGURE 4.6 – Une r-propriété de (pure) persistence non enforçable

PREUVE : Pour montrer cela considérons une r-propriété de *response* $\Pi = (\phi, \varphi)$ et une séquence d'exécution $\sigma \in \Sigma^\omega$. La r-propriété Π peut s'exprimer $(R_f(\psi), R(\psi))$ ($\Pi \in \mathit{Response}(\Sigma)$ dans la vue langage). Supposons que $\neg\varphi(\sigma)$. Cela signifie $\sigma \notin R(\psi)$, *i.e.*, σ n'a pas une infinité de préfixes appartenant à ψ. Elle en a donc un nombre fini. Considérons l'ensemble $S = \{\sigma' \in \Sigma^* \mid \forall\sigma'' \in \Sigma^* : \sigma' \prec \sigma'' \prec \sigma \wedge \neg\psi(\sigma'')\}$ des séquences à partir desquelles toutes les continuations finies ne satisfont pas ψ. Comme $\neg R(\psi)$, cet ensemble n'est pas vide. Notons σ_0 le plus petit élément de S par rapport à l'ordre $\prec$. Nous avons $\forall\sigma' \in \Sigma^* : \sigma_0 \prec \sigma' \Rightarrow \neg\psi(\sigma')$. Comme $\forall\psi \subseteq \Sigma^* : R_f(\psi) \subseteq \psi$ (cf. Définition 10), cela implique que $\forall\sigma' \in \Sigma^* : \sigma_0 \prec \sigma' \Rightarrow \neg\phi(\sigma')$. ■

Une conséquence directe est que les r-propriétés des classes inférieures sont enforçables.

Corollaire α: *Les r-propriétés de safety, guarantee, et d'obligation sont enforçables.* ◇

PREUVE : Ce fait vient évidemment du fait que la classification *Safety-Progress* est une hiérarchie, et les classes des r-propriétés de safety, guarantee, et d'obligation sont toutes incluses dans la classe des r-propriétés de *response*. ■

Borne supérieure

Nous montrons maintenant que la borne inférieure obtenue précédemment est exacte en montrant que l'ensemble des r-propriétés de pure persistence ne sont pas enforçables.

Commençons par donner un exemple de r-propriété de persistence. Nous allons voir qu'elle n'est pas enforçable.

Exemple 13 (Une r-propriété de (pure) persistence non enforçable) Examinons la r-propriété de (pure) persistence suivante :

$$\Pi = (\Sigma^* \cdot a^+, \Sigma^* \cdot a^\omega)$$

Cette r-propriété déclare qu'"il sera inévitablement vrai que a apparaît tout le temps". On peut remarquer que cette r-propriété n'est ni une r-propriété de safety, ni de guarantee, ni d'obligation.

Π est spécifiée par l'automate de Streett $\mathcal{A}_\Pi$ représenté sur la Fig. 4.6. Le critère d'acceptation est $\mathit{vinf}(\sigma, \mathcal{A}_\Pi) \subseteq P$ avec $P = \{1\}$. Nous pouvons comprendre la limite d'enforcement intuitivement avec l'argument suivant : si cette r-propriété était enforçable cela impliquerait qu'un moniteur d'enforcement pourrait décider à partir d'un certain point que le programme sous-jacent produirait toujours l'événement a. Cependant une telle décision ne peut évidemment pas être prise par un moniteur sans mémoriser d'abord la séquence d'exécution toute entière. Ce qui n'est pas réaliste pour une séquence d'exécution infinie.

Plus formellement, comme déclaré dans la section précédente, une r-propriété (ϕ, φ) est enforçable si pour toute séquence d'exécution infinie σ lorsque $\neg\varphi(\sigma)$, le plus long préfixe de σ satisfaisant ϕ existe toujours. Pour la r-propriété considérée, la séquence d'exécution $\sigma'_{bad} = (a \cdot b)^\omega$ exhibe le même problème. En effet, la séquence d'exécution infinie ne satisfait pas la r-propriété alors qu'un nombre infini de ses préfixes satisfont la partie finitaire de la r-propriété (les préfixes qui terminent par l'événement a).

Si nous appliquons les critères d'enforcement (Définitions 26 p. 73 et 27 p. 74) sur les r-propriétés de persistence, il s'avère que les r-propriétés de persistence enforçables sont en fait des r-propriétés de *response*. Ce qui est montré par le théorème suivant.

Théorème η (Les r-propriétés de persistence enforçables sont des *response*) : *L'intersection entre l'ensemble des r-propriétés de persistence et l'ensemble des r-propriétés enforçables est inclus dans l'ensemble des r-propriétés de response*[8] *:*

8. En fait, ce sont mêmes des obligations. Nous savons en effet que Response ∩ Persistence = Obligation

$$\text{Persistence}(\Sigma) \cap EP \subseteq \text{Response}(\Sigma)$$

Preuve : Une r-propriété devient non-enforçable dès qu'il existe une SCC d'états $\overline{R}$ contenant un état P et un état $\overline{P}$ sur son automate reconnaisseur (cf. Définition 27 p. 74). En effet, sur un automate de Streett de persistence, cela permet de reconnaître les séquences infinies invalides avec un nombre infini de préfixes valides. Lorsque l'on retire cette possibilité sur un automate de Streett, l'automate contraint peut être aisément transformé en automate de Streett de *response*. En effet, sur cet automate les états visités infiniment souvent sont soit tous dans P ou soit tous dans $\overline{P}$, c'est-à-dire : $\forall \sigma \in \Sigma^\omega : \mathit{vinf}(\sigma) \cap P \neq \emptyset \Leftrightarrow \mathit{vinf}(\sigma) \subseteq P$. Sur cet automate, il n'y a pas de différence entre les états R et les états P. La conséquence est que en procédant à un nouveau marquage des états P en états R, le nouvel automate spécifie la même propriété. Et le nouvel automate est un automate de *response*. Ce qui permet de conclure. ∎

Corollaire β: *Les r-propriétés de pure persistence ne sont pas enforçables.*

$$(\text{Persistence}(\Sigma) \setminus \text{Response}(\Sigma)) \cap EP = \emptyset \qquad \diamond$$

Preuve : C'est une conséquence directe du Théorème η. ∎

Corollaire γ: *Les r-propriétés de (pure) reactivity ne sont pas enforçables.*

$$\text{Reactivity}(\Sigma) \not\subseteq EP$$
$$\text{Reactivity}(\Sigma) \setminus (\text{Persistence}(\Sigma) \cup \text{Response}(\Sigma)) \cap EP = \emptyset \qquad \diamond$$

Preuve : C'est une conséquence directe du Corollaire β. Une r-propriété générale de reactivity peut être exprimée comme la composition de r-propriétés de *response* et de r-propriétés de persistence. Si la r-propriété de reactivity est pure, elle ne peut se réduire à une simple *response*. Ainsi, la partie persistence de la r-propriété rend la r-propriété de (pure) reactivity non enforçable. ∎

Corollaire δ: *L'ensemble des r-propriétés enforçables est exactement l'ensemble des response :*

$$EP = \text{Response}(\Sigma) \qquad \diamond$$

Preuve : Il reste à prouver que l'ensemble des r-propriétés enforçables est inclus dans l'ensemble des r-propriétés de *response*. Supposons qu'il existe une r-propriété enforçable qui ne soit pas une r-propriété de response. Ainsi, selon la définition de la hiérarchie *Safety-Progress*, cette r-propriété serait une pure persistence ou une pure reactivity. En conséquence, cette r-propriété ne serait pas enforçable. ∎

Remarque 7 (Condition suffisante d'enforçabilité dans les différentes vues) Avec la connaissance de l'espace exacte des r-propriétés enforçables, nous pouvons donner des critères suffisants dans les différentes vues pour savoir si une r-propriété Π est enforçable :

- Dans la vue langage, Π est enforçable s'il existe une propriété finitaire ψ telle que Π puisse s'exprimer $(A_f(\psi), A(\psi))$ ou $(E_f(\psi), E(\psi))$ (ou comme une union/intersection finie), ou bien $(R_f(\psi), R(\psi))$.
- Dans la vue logique, Π est enforçable s'il existe une propriété LTL Υ telle que Υ s'écrive $\Box\, p$, ou $\Diamond\, p$, ou en une disjonction/conjonction finie des deux, ou bien $\Box \Diamond\, p$, avec p une formule du passé, et $\mathit{Sat}(\Upsilon) = \Pi$.
- Dans la vue automate, Π est enforçable si elle est reconnue par un automate de Streett de safety, guarantee, k-obligation (pour un certain $k \in \mathbb{N}$), ou de response. ∗

La Fig. 4.7 complète la Fig. 4.3 avec l'ensemble des r-propriétés enforçables qui est exactement l'ensemble des r-propriétés de response.

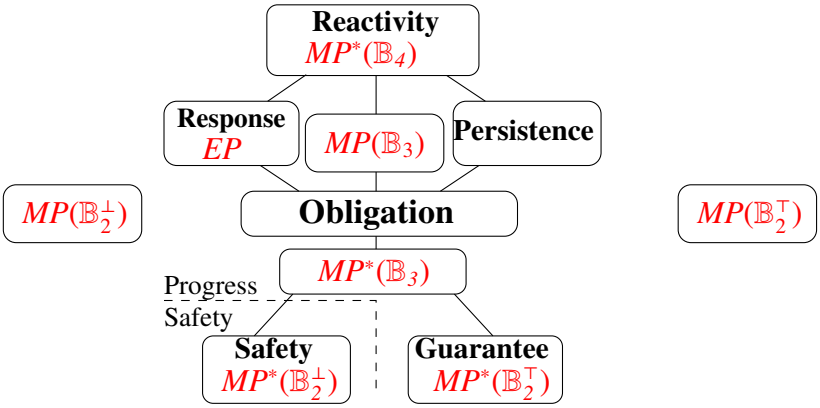


FIGURE 4.7 – r-propriétés monitorables et enforçables dans la classification *Safety-Progress*

4.4 Testabilité dans la classification *Safety-Progress*

Précédemment (Chapitre 2, Section 2.3), nous avons vu que la caractérisation antérieure des propriétés testables [NGH93] déclarait les classes de propriétés de safety et de propriétés de guarantee comme testables pour certaines relations entre les séquences produites par le système testé et la propriété examinée. Toutes les autres classes étaient annoncées comme non testables. En effet, pour chaque relation entre une propriété et les exécutions du système testé, il était uniquement possible de produire des verdicts *inconc* pour ces relations. Dans cette section, nous revenons sur cette caractérisation des propriétés testables. Nous montrons qu'il existe des propriétés de safety et de guarantee non testables, contrairement à ce qui a été annoncé dans [NGH93]. De plus, nous montrons qu'il existe des propriétés testables pour les classes supérieures de la hiérarchie *Safety-Progress*. Ensuite, nous exposons certaines situations où il est possible sous certaines hypothèses de produire des verdicts plus faibles. Enfin, nous montrons comment la notion de quiescence permet de tester certaines propriétés non testables dans les caractérisations précédentes.

Lorsque l'on teste un système, le but recherché est de déterminer une relation entre l'implantation sous test et une certaine propriété. La propriété exprime certains comportements observables ; qui peuvent être des comportements désirés ou indésirés. Généralement, les séquences d'interaction du testeur avec le système sont exprimées dans un vocabulaire différent du vocabulaire de la propriété. Un testeur par son interaction avec le programme sous test doit donc permettre d'interpréter le comportement observable du système sur le même vocabulaire que celui de la propriété. Dans la suite, nous appellerons testeur un programme interagissant avec le programme sous test. L'oracle, faisant partie du testeur, est la procédure de décision qui permet de déterminer le verdict du test.

Dans cette section nous commençons par introduire la notion de testabilité d'une r-propriété, *i.e.*, nous donnons les conditions pour lesquelles une r-propriété est testable sur un programme. Ensuite, nous caractérisons les r-propriétés testables par rapport à la classification *Safety-Progress*.

4.4.1 Notion de testabilité

À partir de son interaction avec le programme sous test, le testeur produit une interprétation. Cette interprétation est une séquence de Σ^*. À l'aide de son oracle, le testeur doit déterminer si une certaine relation existe entre :

- l'ensemble de toutes les séquences d'exécution que peut produire le programme : $Exec(\mathcal{P}_\Sigma)$,
- et les séquences décrites par la r-propriété Π.

Rappelons ici que la r-propriété est un couple formé de deux ensembles : un ensemble de séquences finies et un ensemble de séquences infinies. Nous souhaitons comparer ce couple à l'ensemble des séquences d'exécution du programme qui est un ensemble constitué de séquences finies et infinies. Dans la suite, pour alléger et simplifier les notations, lorsque nous comparons un tel couple avec l'ensemble des séquences d'exécution du programme, nous faisons bien entendu deux comparaisons :

- d'une part, nous comparons la partie finitaire de la r-propriété avec la restriction aux séquences finies de l'ensemble des séquences d'exécution du programme ;

– d'autre part, nous comparons la partie infinitaire de la r-propriété avec la restriction aux séquences infinies de l'ensemble des séquences d'exécution du programme.

Nous explicitons ces deux comparaisons dans la Définition 28, puis utilisons la notation simplifiée dans la suite de cette section.

Comme remarqué dans [NGH93], on peut considérer plusieurs relations possibles entre les séquences d'exécution produites par le programme et la propriété. Elles sont rappelées dans la définition suivante dans le contexte des r-propriétés.

Définition 28 (Relations entre les séquences du programme et une propriété [NGH93]). Il est possible de s'intéresser aux relations suivantes entre $Exec(\mathcal{P}_\Sigma)$ et Π :

– $Exec(\mathcal{P}_\Sigma) \cap \Sigma^* = \Pi \cap \Sigma^*$ et $Exec(\mathcal{P}_\Sigma) \cap \Sigma^\omega = \Pi \cap \Sigma^\omega$ (noté $Exec(\mathcal{P}_\Sigma) = \Pi$ dans la suite) : les comportements observables du programme sont exactement ceux décrits par la r-propriété.
– $Exec(\mathcal{P}_\Sigma) \cap \Sigma^* \subseteq \Pi \cap \Sigma^*$ et $Exec(\mathcal{P}_\Sigma) \cap \Sigma^\omega \subseteq \Pi \cap \Sigma^\omega$ (noté $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ dans la suite) : le programme $\mathcal{P}_\Sigma$ *respecte* la propriété ; tous les comportements du programme sont prévus par la r-propriété.
– $\Pi \cap \Sigma^* \subseteq Exec(\mathcal{P}_\Sigma \cap \Sigma^*)$ et $\Pi \cap \Sigma^\omega \subseteq Exec(\mathcal{P}_\Sigma \cap \Sigma^\omega)$ (noté $\Pi \subseteq Exec(\mathcal{P}_\Sigma)$ dans la suite) : le programme *implémente* la r-propriété ; tous les comportements décrits par la propriété *sont réalisables* par le programme.
– $(\Pi \cap \Sigma^*) \cap (Exec(\mathcal{P}_\Sigma) \cap \Sigma^*) \neq \emptyset$ et $(\Pi \cap \Sigma^\omega) \cap (Exec(\mathcal{P}_\Sigma) \cap \Sigma^\omega) \neq \emptyset$ (noté $\Pi \cap Exec(\mathcal{P}_\Sigma) \neq \emptyset$ dans la suite) : les comportements prévus par la r-propriété et ceux du programme *ne sont pas disjoints*.

Ensuite, le verdict du test est déterminé en fonction des conclusions qu'il est possible d'obtenir pour la relation considérée.

Définition 29 (Verdicts [NGH93]). Étant donnée une relation $\mathcal{R}$ entre $Exec(\mathcal{P}_\Sigma)$ et Π, définie au sens de la Définition 28, le testeur produit les verdicts comme suit :

– *pass* si l'exécution du test permet de déterminer que $\mathcal{R}$ est vérifiée ;
– *fail* si l'exécution du test permet de déterminer que $\mathcal{R}$ n'est pas vérifiée ;
– *inconc* si l'exécution du test ne permet pas de conclure sur la relation $\mathcal{R}$ entre $Exec(\mathcal{P}_\Sigma)$ et Π.

À partir d'une séquence de test $\sigma \in \Sigma^* \cap Exec(\mathcal{P}_\Sigma)$ produite par le testeur et exécutée sur le programme, nous notons $verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi))$ le verdict qu'il est possible de produire à partir de l'observation de σ.

Remarquons qu'en pratique les deux problèmes suivants peuvent se poser :

– Dans la plupart des cas, le système sous test peut être un programme exhibant des séquences d'exécution de longueur infinie. L'évaluation de ces séquences par rapport à la propriété Π n'est pas réalisable par un testeur. En effet, c'est uniquement à partir d'*interactions finies* qu'un testeur peut et doit délivrer un verdict.
– De plus, les séquences d'exécution finies contenues dans la propriété ne sont pas exploitables directement. Par exemple si l'exécution du test a permis d'exhiber une séquence $\sigma \notin \Pi$, il est délicat d'arrêter le test pour produire un verdict définitif. En effet, rien ne permet d'affirmer que laisser continuer le test ne permettra pas d'exhiber une nouvelle séquence $\sigma \cdot \sigma' \in \Pi$.

Le test doit donc s'arrêter lorsqu'il n'y a plus de doute quant au verdict à établir. C'est-à-dire au moment où il est inutile de laisser continuer le test car aucune continuation ne peut remettre en cause le verdict produit. Nous proposons maintenant une définition de la testabilité d'une r-propriété Π sur un programme $\mathcal{P}_\Sigma$ prenant en compte les limitations pratiques formulées précédemment.

Définition 30 (Testabilité). Une r-propriété Π définie sur Π est dite testable sur $\mathcal{P}_\Sigma$ par rapport à la relation $\mathcal{R}$ si les deux conditions suivantes sont vérifiées :

– Il existe un testeur qui peut interpreter les séquences d'exécution de $\mathcal{P}_\Sigma$, et en permettre l'observation. Cette action du testeur est modélisée par la fonction $Exec(\mathcal{P}_\Sigma)$ qui contient les séquences finies observables et infinies du programme $\mathcal{P}_\Sigma$.
– Il existe une séquence d'exécution $\sigma \in \Sigma^*$ telle que :

$$\sigma \in Exec(\mathcal{P}_\Sigma) \Rightarrow verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma, \Pi))) \in \{pass, fail\}$$

Intuitivement la deuxième condition impose l'existence d'une séquence qui, si on arrive à la jouer sur le système sous test, permet d'établir de façon sûre si la relation est vérifiée ou non.

Dans la suite nous supposerons l'existence d'un testeur qui nous permet d'abstraire les séquences d'interactions nécessaires avec le programme. Ce testeur interprète son interaction avec le programme sur le vocabulaire de la propriété. Pour caractériser la testabilité, nous nous intéresserons donc uniquement au deuxième critère. Notons que cette définition suppose l'existence d'un oracle de test qui permet de déterminer $\mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)$ à partir de l'observation d'une séquence $\sigma \in Exec(\mathcal{P}_\Sigma) \cap \Sigma^*$. Notons que cet oracle de test doit être une fonction calculable.

4.4.2 Caractérisation des propriétés testables

Le cadre des r-propriétés permet de déterminer selon chaque relation la testabilité des différents types de propriétés. De plus, ce cadre permet de fournir un oracle calculable, qui est une condition suffisante pour le test. Aussi, nous serons capables de caractériser quelles séquences de test permettent d'établir les verdicts recherchés. Ensuite, nous établirons le verdict à donner pour chaque séquence d'exécution qu'un programme peut produire.

Nous caractérisons maintenant l'espace des propriétés testables selon la relation recherchée entre $Exec(\mathcal{P}_\Sigma)$ et Π.

Testabilité vis-à-vis de la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$

Nous étudions la testabilité des r-propriétés vis-à-vis de la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$.

Verdicts obtenables et condition suffisante. Pour cette relation, les seuls verdicts qu'il est possible d'obtenir sont les verdicts *fail* et *inconc*. Ce que nous expliquons ci-dessous.

Un verdict *pass* signifie que toutes les séquences d'exécution de $\mathcal{P}_\Sigma$ appartiennent à Π. L'unique cas où il est possible d'établir un verdict *pass* est dans le cas trivial où $\Pi = (\Sigma^*, \Sigma^\omega)$, *i.e.*, la propriété Π est toujours vérifiée. Évidemment toutes les implantations satisfont cette relation. Dans les autres cas, il est impossible d'obtenir un tel verdict quelle que soit la classe de propriété considérée dès lors que le programme exhibe des comportements infinis.

Un verdict *fail* signifie que les séquences produites par le programme ne sont pas toutes dans Π, il y a donc une séquence $\sigma \in Exec(\mathcal{P}_\Sigma)$ qui n'appartient pas à Π. Il nous faut donc observer une telle séquence σ, forcément finie, qui est le préfixe d'une séquence d'exécution du programme. De plus, pour pouvoir arrêter le test, il faut être sûr qu'il n'existe pas de continuation de σ qui appartienne à Π. Autrement dit, il faut exhiber une séquence d'exécution de $\mathcal{P}_\Sigma$ telle que Π soit *négativement déterminée* par cette séquence. Ce qui peut s'exprimer assez simplement par le théorème suivant :

Théorème ϑ (Condition suffisante pour établir un verdict pour $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$) : *Il est possible d'établir que la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ n'est pas vérifiée, s'il existe une exécution du programme telle que la r-propriété Π soit négativement déterminée. Ce que nous écrivons comme suit :*

$$\left(\exists \sigma \in \Sigma^* \cap Exec(\mathcal{P}_\Sigma), \forall \sigma' \in \Sigma^\infty : \neg\Pi(\sigma \cdot \sigma')\right) \Rightarrow verdict(\sigma, Exec(\mathcal{P}_\Sigma) \subseteq \Pi) = fail \tag{4.3}$$

Preuve : La preuve de ce théorème est évidente. En effet, si le programme produit une séquence d'exécution σ qui ne peut être continuée en une séquence d'exécution finie ou infinie appartenant à la r-propriété Π, alors la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ est forcément non vérifiée. ■

Testabilité de cette relation dans la classification *Safety-Progress*. Pour chacune des classes suivantes, nous montrons à quelles conditions les propriétés de cette classe sont testables. De plus, nous exhibons les séquences de tests qu'il est possible d'essayer de jouer sur l'implantation pour déterminer un verdict *fail* pour la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$.

- Pour les r-propriétés de safety. Soit Π une r-propriété de safety, alors il existe $\psi \subseteq \Sigma^*$ telle que Π puisse s'exprimer $(A_f(\psi), A(\psi))$. Cette r-propriété est testable si l'ensemble $\overline{\psi}$ est non vide. En effet, prenons $\sigma \in \Sigma^* \cap \overline{\psi}$. Alors nous savons d'après la Propriété 1 (p. 40) sur la fermeture des r-propriétés de safety que :

  - toutes les continuations finies de σ n'appartiennent pas à $A_f(\psi)$; et
  - toutes les continuations infinies de σ n'appartiennent pas à $A(\psi)$.

  Ainsi toutes les continuations finies et infinies de σ n'appartiennent pas à $(A_f(\psi), A(\psi))$. La seule r-propriété de safety qui n'est donc pas testable, sous ces conditions, est la r-propriété toujours vraie : $(\Sigma^*, \Sigma^\omega)$. Comme dit précédemment, il est bien évident que tout programme dont le vocabulaire observable est Σ satisfait cette spécification.
- Pour les r-propriétés de guarantee. Soit Π une r-propriété de guarantee, alors il existe $\psi \subseteq \Sigma^*$ telle que Π puisse s'exprimer $(E_f(\psi), E(\psi))$. Cette r-propriété est testable si l'ensemble $\{\sigma \in \overline{\psi} \mid pref(\sigma) \subseteq \overline{\psi} \wedge cont^*(\sigma) \subseteq \overline{\psi}\}$ est non vide. En effet, soit σ une séquence appartenant à cet ensemble. Alors tous les préfixes de σ et toutes ses continuations n'appartiennent pas à ψ. Par conséquent, ces continuations ne peuvent également pas appartenir à $E_f(\psi)$ ni à $E(\psi)$ ni à Π.
- Pour les r-propriétés d'obligation. Soit Π une r-propriété d'obligation, alors Π peut s'exprimer en forme normale conjonctive et en forme normale disjonctive (cf. Lemme α, p. 44).
  - Lorsque Π est exprimée en forme normale conjonctive, alors il existe $k \in \mathbb{N}$, *t.q.* Π s'exprime $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi'_i))$ où $S_i(\psi_i)$ (resp. $G_i(\psi'_i)$) est une r-propriété de safety (resp. de guarantee) construite à partir de ψ_i (resp. de ψ'_i). Cette r-propriété est testable si l'ensemble $\bigcup_{i=1}^{k}(\overline{\psi_i} \cap \{\sigma \in \overline{\psi'_i} \mid cont^*(\sigma) \subseteq \overline{\psi'_i}\})$ est non vide.
  - Lorsque Π est exprimée en forme normale disjonctive, alors il existe $k \in \mathbb{N}$, *t.q.* Π s'exprime $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi'_i))$ où S_i (resp. G_i) est une r-propriété de safety (resp. de guarantee) construite à partir de ψ_i (resp. de ψ'_i). Cette r-propriété est testable si l'ensemble $\bigcap_{i=1}^{k}(\overline{\psi_i} \cup \{\sigma \in \overline{\psi'_i} \mid cont\sigma \subseteq \overline{\psi_i}$ est non vide.

  Pour montrer qu'une séquence appartenant à l'un des ensembles définis a toutes ses continuations finies et infinies qui ne satisfont pas la r-propriété, il suffit de réaliser une induction sur k et d'utiliser les raisonnements utilisés pour les r-propriétés de safety et guarantee.
- Pour les r-propriétés de response et persistence. Le raisonnement est similaire à celui pour les r-propriétés de guarantee. Soit Π une r-propriété de response (resp. persistence), alors il existe $\psi \subseteq \Sigma^*$ telle que Π puisse s'exprimer $(R_f(\psi), R(\psi))$ (resp. $(P_f(\psi), P(\psi))$). Cette r-propriété est testable si l'ensemble $\{\sigma \in \overline{\psi} \mid cont^*(\sigma) \subseteq \overline{\psi}\}$ est non vide. La différence est ici pour que la r-propriété soit négativement déterminée, elle peut avoir des séquences qui ont certains préfixes dans ψ.

Verdicts à délivrer. Chacune des conditions de testabilité exprimée ci-dessus est de la forme

$$f(\psi_1, \ldots, \psi_n) \neq \emptyset$$

où les ψ_i sont les propriétés finitaires utilisées pour construire la r-propriété et f une composition d'opérations ensemblistes sur les ψ_i. Par exemple, pour les r-propriétés exprimées en forme normale conjonctive, la condition de testabilité s'exprime $\bigcup_{i=1}^{k}(\overline{\psi_i} \cap \{\sigma \in \overline{\psi'_i} \mid cont^*(\sigma) \subseteq \overline{\psi'_i}\}) \neq \emptyset$

L'appartenance d'une séquence à $f(\psi_1, \ldots, \psi_n)$ est évidemment décidable (les ψ_i sont des langages réguliers). En conséquence à partir des propriétés finitaires ψ_i utilisées pour définir une r-propriété il est possible de définir un oracle de test permettant de délivrer un verdict pour les séquences d'exécution du programme :

Lorsqu'une séquence d'exécution $\sigma \in \Sigma^* \cap Exec(\mathcal{P}_\Sigma)$ appartient à $f(\psi_1, \ldots, \psi_n)$, l'oracle de test peut délivrer un verdict *fail* pour cette relation. En effet, pour chaque classe de r-propriétés, lorsqu'une séquence finie appartient à l'ensemble $f(\psi_1, \ldots, \psi_n)$, alors la r-propriété construite à partir des ψ_i est déterminée négativement.

Inversement, lorsqu'une séquence d'exécution $\sigma \in \Sigma^* \cap Exec(\mathcal{P}_\Sigma)$ n'appartient pas à $f(\psi_1, \ldots, \psi_n)$, l'oracle de test peut délivrer un verdict *inconc* pour cette relation.

Quand arrêter le test ? Le programme testé produisant une séquence d'exécution $\sigma \in \Sigma^*$, la question qui peut se poser est celle du moment de l'arrêt du test. Bien sur, une première réponse évidente est "lorsque l'on a trouvé un verdict *fail*". Mais dans les autres cas, lorsque l'interaction avec le programme produit une suite d'événements qui jusque là donnent une évaluation inconclusive pour la relation, la question est entière. Alors le test peut être arrêté dans les deux cas suivants. Le premier est lorsque la r-propriété est négativement déterminée par une séquence d'exécution produite par le programme ou qu'il n'existe pas de continuation de cette séquence telle que la r-propriété soit positivement déterminée. Dans les autres cas, cela reste à l'expertise du testeur de décider lorsque le test doit être arrêté.

$Exec(\mathcal{P}_\Sigma) \subseteq \Pi$	Verdicts possibles	Condition de testabilité
Safety $(A_f(\psi), A(\psi))$	*fail*, *inconc*	$\overline{\psi} \neq \emptyset$
Guarantee $(E_f(\psi), E(\psi))$	*fail*, *inconc*	$\{\sigma \in \overline{\psi} \mid pref(\sigma) \subseteq \overline{\psi} \wedge cont^*(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$
Obligation $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi'_i))$ $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi'_i))$	*fail*, *inconc*	$\bigcup_{i=1}^{k}(\overline{\psi_i} \cap \{\sigma \in \overline{\psi'_i} \mid cont^*(\sigma) \subseteq \overline{\psi'_i}\}) \neq \emptyset$ $\bigcap_{i=1}^{k}(\overline{\psi_i} \cup \{\sigma \in \overline{\psi'_i} \mid cont\sigma \subseteq \overline{\psi_i} \neq \emptyset$
Response $(R_f(\psi), R(\psi))$	*fail*, *inconc*	$\{\sigma \in \overline{\psi} \mid cont^*(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$
Persistence $(P_f(\psi), P(\psi))$	*fail*, *inconc*	$\{\sigma \in \overline{\psi} \mid cont^*(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$

TABLE 4.1 – Résumé des résultats de testabilité pour la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$

Exemple 14 (Testabilité de certaines r-propriétés vis-à-vis de $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$) Nous présentons la testabilité de trois r-propriétés.

Considérons un vocabulaire $\Sigma = \{a, b, c\}$. Le DFA représenté sur la Fig. 3.10 (p. 53) reconnaît la propriété finitaire $\psi_1 = a^* \cdot (b^* + c \cdot (c + a)^* \cdot b^+)$ (introduite dans l'Exemple 7). La r-propriété de safety construite à partir de ψ_1, reconnue par l'automate de Streett représenté sur la Fig. 3.11 (p. 53), est testable vis-à-vis de la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$. En effet, elle satisfait la condition de testabilité : l'automate reconnaissant ψ_1 possède un état non accepteur atteignable depuis l'état initial. Les séquences intéressantes à jouer pour obtenir un verdict *fail* sont celles menant à l'état *sink* sur l'automate de Streett ou aux états 3 et 4 sur le DFA.

Considérons le vocabulaire $\Sigma = \{a, b\}$, et la propriété finitaire $\psi = (a \cdot b)^+$ définie sur Σ. La propriété ψ est reconnue par le DFA représenté sur la Fig. 3.12 (p. 54).

- La r-propriété de guarantee construite à partir de ψ, et représentée par l'automate de Streett de la Fig. 3.13 (p. 54), est testable. En effet elle satisfait la condition de testabilité pour les propriétés de guarantee : il y a des séquences appartenant à $\overline{\psi}$ telles que tous les préfixes de ces séquences et toutes leur continuations sont dans $\overline{\psi}$ également. Les séquences intéressantes à jouer pour obtenir un verdict *fail* sont celles menant à l'état 5.
- La r-propriété de response construite à partir de ψ, et représentée par l'automate de Streett de la Fig. 3.14 (p. 54), est testable. En effet, elle satisfait la condition de testabilité pour les propriétés de response : il y a des séquences appartenant à $\overline{\psi}$ telles que tous les continuations de ces séquences appartiennent à $\overline{\psi}$ également. Les séquences intéressantes à jouer pour obtenir un verdict *fail* sont celles menant à l'état 5.

Les résultats de testabilité pour la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ sont résumés sur la Table 4.1. Ainsi, nous avons étendu et précisé certains résultats exposés dans [NGH93]. Notamment, nous avons montré qu'il existait une r-propriété de safety qui n'était pas testable. De plus, nous avons montré que certaines r-propriétés des autres classes pouvaient être testables également.

Testabilité vis-à-vis de la relation $Exec(\mathcal{P}_\Sigma) = \Pi$

Les raisonnements appliqués pour la testabilité vis-à-vis de la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ s'appliquent de la même manière. La caractérisation des r-propriétés testables est donc la même. En effet, lorsque l'on trouve une séquence $\sigma \in \Sigma^*$ telle qu'il soit possible de trouver un verdict *fail* pour $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$, alors ce même verdict s'applique pour $Exec(\mathcal{P}_\Sigma) = \Pi$, *i.e.*, $Exec(\mathcal{P}_\Sigma) \not\subseteq \Pi \Rightarrow Exec(\mathcal{P}_\Sigma) \neq \Pi$.

Testabilité vis-à-vis de la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$

Nous étudions la testabilité des r-propriétés vis-à-vis de la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$.

Verdicts obtenables et condition suffisante. Les seuls verdicts qu'il est possible d'obtenir pour cette relation sont les verdicts *pass* et *inconc*. Ce que nous expliquons ci-dessous.

Un verdict *fail* signifie que $\Pi \cap Exec(\mathcal{P}_\Sigma) = \emptyset$. Il est impossible d'obtenir un tel verdict dans le cas général [9]. Même si l'on trouve un ensemble de séquences d'exécution de $\mathcal{P}_\Sigma$ qui n'appartiennent pas à Π, on ne pourra jamais être sûr qu'il n'existe pas une autre exécution du programme sous test exhibant une séquence d'exécution qui appartiendra à Π.

Un verdict *pass* signifie que $\Pi \cap Exec(\mathcal{P}_\Sigma) \neq \emptyset$. Pour pouvoir produire un tel verdict, il faut pouvoir trouver une séquence d'exécution $\sigma \in \Sigma^*$ qui appartient à Π, et dont toutes les extensions appartiennent également à Π. Autrement dit, il faut exhiber une séquence d'exécution de $\mathcal{P}_\Sigma$ telle que Π soit déterminée positivement par cette séquence. Ce qui peut s'exprimer assez simplement par le théorème suivant :

Théorème ι (Condition suffisante pour la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ et un verdict *pass*) : *La relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ est vérifiée, s'il existe une exécution du programme telle que la r-propriété Π soit positivement déterminée. Ce que nous écrivons comme suit :*

$$\exists \sigma \in \Sigma^* \cap Exec(\mathcal{P}_\Sigma), \forall \sigma' \in \Sigma^\infty, \Pi(\sigma \cdot \sigma') \;\Rightarrow\; verdict(\sigma, Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset) = pass \tag{4.4}$$

Preuve : La preuve de ce théorème est aussi simple que celle du théorème ϑ. En effet, si nous pouvons trouver une séquence d'exécution du programme qui a toutes ses continuations qui satisfont la r-propriété Π, alors cette séquence montre que le programme et la propriété ont au moins une séquence en commun.■

Testabilité de cette relation dans la classification *Safety-Progress*. Nous établissons maintenant pour chaque classe de propriétés les conditions sous lesquelles la partie gauche de l'implication est vérifiée. Ce qui, grâce au théorème précédemment exprimé, permet d'établir qu'il existe une séquence d'exécution du programme nous donnant un verdict *pass* pour la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$.

- Pour les r-propriétés de safety. Soit Π une r-propriété de safety, alors il existe $\psi \subseteq \Sigma^*$ telle que Π puisse s'exprimer $(A_f(\psi), A(\psi))$. Cette r-propriété est testable si l'ensemble $\{\sigma \in \psi \mid cont^*(\sigma) \subseteq \psi\}$ est non vide. Les séquences appartenant à cet ensemble sont dans ψ et toutes leurs extensions le sont également. D'après la définition des r-propriétés de safety, ces séquences et toutes leur extensions appartiennent à Π.
- Pour les r-propriétés de guarantee. Soit Π une r-propriété de guarantee, alors il existe $\psi \subseteq \Sigma^*$ telle que Π puisse s'exprimer $(E_f(\psi), E(\psi))$. Cette r-propriété est testable si l'ensemble ψ est non vide. En effet, soit $\sigma \in \psi$, alors d'après la définition des r-propriétés de guarantee, une séquence appartenant à ψ a toutes ses extensions finies (resp. infinies) appartenant à $E_f(\psi)$ (resp. $E(\psi)$).
- Pour les r-propriétés d'obligation. Soit Π une r-propriété d'obligation, alors Π peut s'exprimer en forme normale conjonctive ou en forme normale disjonctive (cf. Lemme α p. 44).
  - Si Π est exprimée en forme normale conjonctive, alors il existe $k \in \mathbb{N}$, *t.q.* Π s'exprime $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi'_i))$ où $S_i(\psi_i)$ (resp. $G_i(\psi'_i)$) est une r-propriété de safety (resp. de guarantee) construite à partir de ψ_i (resp. de ψ'_i). Cette r-propriété est testable si l'ensemble $\bigcap_{i=1}^{k}(\{\sigma \in \psi_i \mid cont\sigma \subseteq \psi_i\} \cup \psi'_i)$ est non vide.
  - Si Π est exprimée en forme normale disjonctive, alors il existe $k \in \mathbb{N}$, *t.q.* Π s'exprime $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi'_i))$ où $S_i(\psi_i)$ (resp. $G_i(\psi'_i)$) est une r-propriété de safety (resp. de guarantee) construite à partir de ψ_i (resp. de ψ'_i). Cette r-propriété est testable si l'ensemble $\bigcup_{i=1}^{k}(\{\sigma \in \psi_i \mid cont^*(\sigma) \subseteq \psi_i\} \cap \psi'_i)$ est non vide.

  Pour montrer qu'une séquence appartenant à l'un des ensembles définis a toutes ses continuations finies et infinies qui satisfont une r-propriété de $k-$Obligation, il suffit de réaliser une induction sur k et d'utiliser les raisonnements utilisés pour les r-propriétés de safety et guarantee.
- Pour les r-propriétés de response et persistence. Le raisonnement est similaire à celui des r-propriétés de safety. Soit Π une r-propriété de response (resp. persistence), alors il existe $\psi \subseteq \Sigma^*$ telle que Π puisse s'exprimer $(R_f(\psi), R(\psi))$ (resp. $(P_f(\psi), P(\psi))$). Cette r-propriété est testable si l'ensemble $\{\sigma \in \psi \mid cont^*(\sigma) \subseteq \psi\}$ est non vide. La différence est la suivante : pour que la r-propriété soit positivement déterminée, elle peut avoir des séquences qui ont certains préfixes qui ne sont pas dans ψ.

Pour chaque classe de propriétés, sous les conditions exprimées, si l'on arrive à jouer une séquence de l'ensemble mis en évidence, il est possible d'établir le verdict *pass* pour la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$.

9. Rappelons que nous plaçons dans le cas où ne possédons pas le code source du système testé ni de spécification.

$Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$	Verdicts obtenables	Condition de testabilité
Safety $(A_f(\psi), A(\psi))$	*pass*, *inconc*	$\{\sigma \in \psi \mid pref(\sigma) \subseteq \psi \wedge cont^*(\sigma) \subseteq \psi\}$
Guarantee $(E_f(\psi), E(\psi))$	*pass*, *inconc*	$\psi \neq \emptyset$
Obligation $\bigcap_{i=1}^k (S_i(\psi_i) \cup G_i(\psi'_i))$ $\bigcup_{i=1}^k (S_i(\psi_i) \cap G_i(\psi'_i))$	*pass*, *inconc*	$\bigcap_{i=1}^k (\{\sigma \in \psi_i \mid cont\sigma \subseteq \psi_i\} \cup \psi'_i) \neq \emptyset$ $\bigcup_{i=1}^k (\{\sigma \in \psi_i \mid \forall \sigma' \in \Sigma^*, \psi_i(\sigma \cdot \sigma')\} \cap \psi'_i) \neq \emptyset$
Response $(R_f(\psi), R(\psi))$	*pass*, *inconc*	$\{\sigma \in \psi \mid cont^*(\sigma) \subseteq \psi\} \neq \emptyset$
Persistence $(P_f(\psi), P(\psi))$	*pass*, *inconc*	$\{\sigma \in \psi \mid cont^*(\sigma) \subseteq \psi\} \neq \emptyset$

TABLE 4.2 – Résumé des résultats de testabilité pour la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$

Verdicts à délivrer. De manière similaire à la relation précédente, les conditions de testabilité exprimées ci-dessus sont de la forme

$$f(\psi_1, \ldots, \psi_n) \neq \emptyset$$

où les ψ_i sont les propriétés finitaires utilisées pour construire la r-propriété. L'appartenance d'une séquence à $f(\psi_1, \ldots, \psi_n)$ est évidemment décidable. En conséquence, à partir des propriétés finitaires ψ_i utilisées pour définir une r-propriété, il est possible de définir un oracle de test permettant de délivrer un verdict pour les séquences d'exécution du programme :

Lorsqu'une séquence d'exécution $\sigma \in \Sigma^* \cap Exec(\mathcal{P}_\Sigma)$ appartient à $f(\psi_1, \ldots, \psi_n)$, l'oracle de test peut délivrer un verdict *pass* pour cette relation.

Inversement, lorsqu'une séquence d'exécution $\sigma \in \Sigma^* \cap Exec(\mathcal{P}_\Sigma)$ n'appartient pas à $f(\psi_1, \ldots, \psi_n)$, l'oracle de test peut délivrer un verdict *inconc* pour cette relation.

Quand arrêter le test ? De manière similaire à la relation précédente, il est possible d'arrêter le test dans les cas suivants. D'abord lorsqu'un verdict *pass* est établi. Et deuxièmement, lorsque la r-propriété est négativement déterminée par la séquence d'exécution produite par le programme ou lorsqu'il n'existe pas de continuation de cette séquence d'exécution telle que la r-propriété soit négativement déterminée.

Exemple 15 (Testabilité de certaines propriétés vis-à-vis de $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$) Considérons le vocabulaire $\Sigma = \{a, b\}$, et la propriété finitaire $\psi = (a \cdot b)^+$ définie sur Σ. La propriété ψ est reconnue par le DFA représenté sur la Fig. 3.12 (p. 54).

La r-propriété de guarantee construite à partir de ψ, et représentée par l'automate de Streett de la Fig. 3.13 (p. 54), est testable. En effet elle satisfait la condition de testabilité pour les propriétés de guarantee : $\psi \neq \emptyset$. Les séquences intéressantes à jouer pour obtenir un verdict *pass* sont celles menant à l'état 3.

Les résultats de testabilité pour la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ sont résumés sur la Table 4.2. Ainsi, nous avons étendu et précisé certains résultats exposés dans [NGH93]. Notamment, nous avons montré qu'il existait une r-propriété de guarantee qui n'était pas testable. De plus, nous avons montré que certaines r-propriétés des autres classes pouvaient être testables également.

Testabilité vis-à-vis de la relation $\Pi \subseteq Exec(\mathcal{P}_\Sigma)$

Il n'est pas possible d'établir de verdict pour cette relation, ce que nous explicitons ci-dessous.

Pour établir un verdict *pass* pour cette relation, il faudrait montrer que toutes les séquences d'exécution décrites par la propriété sont des séquences du programme. Ce qui est impossible dès que l'ensemble des séquences décrites par la r-propriété est infini.

Pour établir un verdict *fail*, il faudrait établir qu'au moins une séquence décrite par la propriété ne peut pas être jouée sur l'implantation. Même en trouvant une séquence du programme qui ne satisfait pas la r-propriété, cela ne permet pas d'affirmer que la relation n'est pas vérifiée. En effet, une autre exécution de l'implantation pourrait exhiber une telle séquence.

4.4.3 Raffinement de la notion de verdict

Similairement à l'introduction de la notion de valeurs de vérité plus faibles en monitoring, il est possible d'introduire des verdicts plus faibles en test. De façon sous-jacente, cela correspond à l'idée de ne plus voir le test comme une activité destructrice, mais comme une activité permettant de gagner de la confiance dans le fait que l'implantation respecte une propriété.

Nous avançons maintenant le fait que sous certaines conditions, il est possible de produire un verdict *faible* pour certaines classes de propriétés. Et ceci dans les deux sens suivants. Tout d'abord, arrêter le test et produire un verdict faible consiste à dire que la séquence vue appartient (ou pas) à la propriété *jusqu'à* présent. De plus, ce verdict s'exprime toujours *sur une séquence d'exécution*, et ne permet pas de conclure quoi que ce soit sur l'ensemble $Exec(\mathcal{P}_\Sigma)$ de toutes les séquences d'exécution du programme. Ainsi pour pouvoir amener une quelconque confiance dans le fait que les séquences d'exécution du programme sont bien toutes prévues par la r-propriété, cela nécessite d'exécuter le test plusieurs fois. De plus, une notion de couverture ici semble indispensable pour donner de la pertinence aux verdicts produits. L'idée est similaire à l'idée d'introduire des valeurs de vérité plus faibles en vérification à l'exécution. L'idée de parler de satisfaction "si le programme s'arrête là" en vérification à l'exécution correspond à l'idée "le test a montré assez de choses sur l'implantation" en test.

Revisite de la testabilité vis-à-vis de la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$. Pour cette relation, nous avons vu que les seuls verdicts qu'il était possible d'obtenir étaient *fail* et *inconc*. Nous proposons d'étudier les conditions sous lesquelles, il est possible de produire des verdicts *pass* faibles. Notons que le verdict *fail* qu'il était possible de produire précédemment est toujours réalisable sous les mêmes conditions. La production de verdict *pass* faibles est donc une extension, et une substitution du cas où un testeur produisait un verdict *inconc* précédemment.

La relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ parle de toutes les séquences du programme. Ainsi suivant les deux caractéristiques de verdicts faibles exposées précédemment, il y a deux types de verdicts *pass* faibles.

Cas 1 : Lorsque la séquence jouée sur le programme *détermine positivement* la r-propriété. Dans ce cas, le test peut être arrêté car quelque soit le futur comportement du programme testé, celui-ci exhibera une séquence d'exécution qui satisfera la r-propriété. Dans ce cas, de nouvelles exécutions de test peuvent être envisagées pour gagner en confiance.

Cas 2 : Lorsque la séquence jouée sur le programme satisfait la r-propriété *pour le moment*. Alors dans ce cas, le testeur peut prendre la décision d'arrêter le test. Cependant, le risque existe qu'une continuation, qui peut être produite en laissant continuer le test, peut ne plus satisfaire la r-propriété.

Nous revisitons pour chaque classe de r-propriétés en fonction de la séquence exhibée par le programme le type de verdict *pass* faible qu'il est possible de produire pour cette relation.

- Pour les r-propriétés de safety. Soit Π une r-propriété de safety, alors il existe $\psi \subseteq \Sigma^*$ telle que Π puisse s'exprimer $(A_f(\psi), A(\psi))$. Il est possible de produire un verdict *pass* faible si l'ensemble ψ n'est pas vide et est préfixe-clos.
  - Lorsque la séquence produite par le programme appartient à $\{\sigma \in \psi \mid pref(\sigma) \subseteq \psi \wedge cont^*(\sigma) \subseteq \psi\}$ le testeur peut produire un verdict *pass* faible du cas 1.
  - Lorsque $\sigma \in \psi$, le testeur peut produire un verdict *pass* faible du cas 2.
- Pour les r-propriétés de guarantee. Soit Π une r-propriété de guarantee, alors il existe $\psi \subseteq \Sigma^*$ telle que Π puisse s'exprimer $(E_f(\psi), E(\psi))$. Il est possible de produire un verdict *pass* faible si l'ensemble ψ n'est pas vide. Les verdicts *pass* faibles produits correspondent toujours au cas 1.
- Pour les r-propriétés d'obligation. Soit Π une r-propriété d'obligation, alors Π peut s'exprimer en forme normale conjonctive ou en forme normale disjonctive (cf. Lemme α, p. 44).
  - Si Π est exprimée en forme normale conjonctive, alors il existe $k \in \mathbb{N}$, *t.q.* Π s'exprime $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi'_i))$ où S_i (resp. G_i) est une r-propriété de safety (resp. de guarantee) construite à partir de ψ_i (resp. de ψ'_i). Il est possible de produire un verdict *pass* faible si l'ensemble $\bigcap_{i=1}^{k}(\psi_i \cup \psi'_i)$ est non vide.
    - Le testeur peut produire un verdict *pass* faible du cas 1 lorsque la séquence produite par le programme appartient à $\bigcap_{i=1}^{k} \psi'_i$.
    - Il produit un verdict *pass* faible du cas 2 pour les autres séquences d'exécution.
  - Si Π est exprimée en forme normale disjonctive, alors il existe $k \in \mathbb{N}$, *t.q.* Π s'exprime $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi'_i))$ où S_i (resp. G_i) est une r-propriété de safety (resp. de guarantee) construite à partir de ψ_i (resp. de ψ'_i). Il est possible de produire un verdict *pass* faible si l'ensemble $\bigcap_{i=1}^{k}(\psi_i \cup \psi'_i)$ est non

vide.
  - Le testeur peut produire un verdict *pass* faible du cas 1 lorsque la séquence produite par le programme appartient à $\bigcup_{i=1}^{k}(\{\sigma \in \psi_i \mid cont^*(\sigma) \subseteq \psi_i\} \cap \psi'_i)$.
  - Il produit un verdict *pass* faible du cas 2 pour les autres séquences d'exécution.
- Pour les r-propriétés de response et persistence. Le raisonnement est similaire à celui pour les r-propriétés de safety. Soit Π une r-propriété de response (resp. persistence), alors il existe $\psi \subseteq \Sigma^*$ telle que Π puisse s'exprimer $(R_f(\psi), R(\psi))$ (resp. $(P_f(\psi), P(\psi))$). Il est possible de produire un verdict *pass* faible si l'ensemble ψ n'est pas vide.
  - Lorsque la séquence produite par le programme appartient à $\{\sigma \in \psi \mid cont^*(\sigma) \subseteq \psi\}$ le testeur peut produire un verdict *pass* faible du cas 1.
  - Lorsque $\sigma \in \psi$, le testeur peut produire un verdict *pass* faible du cas 2.

Revisite de la testabilité vis-à-vis de la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$. Pour cette relation, nous avons vu que les seuls verdicts qu'il était possible d'obtenir étaient des verdicts *pass* et *inconc*. Le verdict *fail* faible qu'il est possible de produire pour cette relation est produisible sous les mêmes conditions que pour la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$. Dans le premier cas, la séquence produite par le programme détermine négativement la r-propriété. Alors, le test peut être arrêté car quelque soit la suite de l'exécution du test, la r-propriété ne pourra être satisfaite. Dans ce cas, nous avons trouvé une séquence d'exécution produite par le programme qui n'apparait pas dans la r-propriété. Cela ne permet pas de conclure un verdict définitif pour la relation. Cependant, un nombre important de *fail* faibles peut être intéressant et donner des informations sur l'implantation testée. Dans le deuxième cas, la r-propriété n'est *pas satisfaite pour le moment*. Dans ce cas, le testeur doit prendre la décision d'arrêter le test lui même. Ici le risque est que laisser continuer le test peut exhiber une séquence qui satisfera la r-propriété.

4.4.4 Introduction de la notion de quiescence

La notion de quiescence [Tre96, Tre97, JJ05] a été introduite dans le test de conformité pour représenter l'inactivité du système sous test. En pratique, différents types de quiescence peuvent survenir (voir par exemple [JJ05]). Ici nous utiliserons la quiescence de manière générale pour représenter le fait que le système sous test ne peut plus interagir avec le testeur. La quiescence est usuellement représentée en introduisant un nouvel événement observable δ sur le système sous test. L'ensemble des séquences d'exécution produisibles par le programme peut donc contenir désormais l'événement représentant la quiescence : $Exec(\mathcal{P}_\Sigma) \subseteq (\Sigma \cup \{\delta\})^\infty$.

Pour l'approche que nous proposons, l'utilité de la quiescence réside dans le fait qu'elle permet de déterminer que la séquence courante produite par le programme sous test ne possède pas de continuation. En conséquence, les conditions de testabilité, énoncées précédemment, peuvent être affaiblies. En effet, lorsque l'on a déterminé que l'interaction avec le système sous test était terminée, il n'est plus nécessaire d'exiger que, quelque soit la continuation de la séquence observée jusqu'à présent, la propriété sera évaluée pareillement. Cela revient à considérer, d'une certaine manière, que l'évaluation résultant du dernier événement avant l'observation de la quiescence du système sous test "termine" la séquence d'exécution. C'est-à-dire que si la propriété était satisfaite (resp. non satisfaite) par la dernière séquence observée, avant observation de la quiescence, alors la séquence observée détermine positivement (resp. négativement) la r-propriété.

Revisite de la testabilité vis-à-vis de la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$. Pour cette relation, le rôle du testeur consiste maintenant à "pousser" l'implantation dans un état qui ne satisfait pas la propriété et d'observer ensuite l'événement de quiescence. Informellement, la condition de testabilité repose maintenant sur l'existence d'une séquence rendant la r-propriété fausse. Le résumé des résultats de testabilité pour la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ est donc mis à jour sur la Table 4.3

Pour chaque classe de r-propriétés, les conditions de testabilité s'expriment toujours sous la même forme $(f(\psi_1, \ldots, \psi_n) \neq \emptyset)$. L'ensemble des séquences à jouer sur le système sous test sont donc les séquences constituées d'un préfixe appartenant à $f(\psi_1, \ldots, \psi_n)$ et se terminant par l'événement δ.

Exemple 16 (Testabilité d'une r-propriété vis-à-vis de $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ avec quiescence) Considérons le vocabulaire $\Sigma = \{a, b\}$, et la propriété finitaire $\psi = (a \cdot b)^+$ définie sur Σ. La propriété ψ est reconnue par le DFA représenté sur la Fig. 3.12 (p. 54).

$Exec(\mathcal{P}_\Sigma) \subseteq \Pi$	Verdicts possibles	Condition de testabilité
Safety $(A_f(\psi), A(\psi))$	*fail*, *inconc*	$\overline{\psi} \neq \emptyset$
Guarantee $(E_f(\psi), E(\psi))$	*fail*, *inconc*	$\{\sigma \in \overline{\psi} \mid pref(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$
Obligation $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi'_i))$ $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi'_i))$	*fail*, *inconc*	$\bigcup_{i=1}^{k}(\overline{\psi_i} \cap \{\sigma \in \overline{\psi'_i} \mid pref(\sigma) \subseteq \overline{\psi'_i}) \neq \emptyset$ $\bigcap_{i=1}^{k}(\overline{\psi_i} \cup \{\sigma \in \overline{\psi'_i} \mid pref(\sigma) \subseteq \overline{\psi'_i}\} \neq \emptyset$
Response $(R_f(\psi), R(\psi))$	*fail*, *inconc*	$\overline{\psi} \neq \emptyset$
Persistence $(P_f(\psi), P(\psi))$	*fail*, *inconc*	$\overline{\psi} \neq \emptyset$

TABLE 4.3 – Résumé des résultats de testabilité pour la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ avec la notion de quiescence

La r-propriété de response construite à partir de ψ, et représentée par l'automate de Streett de la Fig. 3.14 (p. 54), reste testable. Les séquences intéressantes à jouer pour obtenir un verdict *fail* sont maintenant :

- celles menant à l'état 5,
- et celles menant à l'état 4 et pour lesquelles on observe la quiescence du système sous test.

Revisite de la testabilité vis-à-vis de la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$. Les conditions de testabilité pour cette relation peuvent se mettre à jour de manière similaire. Il s'agit dans le cas de cette relation de trouver une séquence "satisfaisant" la r-propriété.

4.5 Conclusion du chapitre et perspectives

Conclusion. Nous nous sommes intéressés à l'applicabilité de différentes méthodes de validation de propriétés sur un système lors de son exécution. Pour cela, nous avons utilisé le cadre général de la classification *Safety-Progress* des propriétés. Ceci nous a permis de caractériser les ensembles de propriétés monitorables, enforçables, et testables de façon uniforme.

Nous sommes d'abord revenu sur la définition classique de la monitorabilité, et nous avons donné une caractérisation multivaluée des propriétés monitorables selon cette définition. En suivant la motivation d'utiliser un domaine de valeur de vérité plus riche, nous avons introduit une nouvelle définition de monitorabilité basée sur la distinguabilité des bons et mauvais préfixes. Cette définition est différente de la définition classique basée sur la détermination positive ou négative. Nous pensons que cette définition correspond mieux aux besoins pratiques et aux implantations des outils de vérification à l'exécution. De plus cette définition étend conservativement les précédentes. Pour cette nouvelle définition de monitorabilité, nous avons également donné une caractérisation multivaluée en fonction du domaine de vérité considéré.

Nous nous sommes ensuite intéressé à l'espace des propriétés enforçables. Nous avons donné une caractérisation de l'espace des propriétés enforçables indépendamment de tout mécanisme d'enforcement que l'on peut considérer. Ainsi, cette caractérisation représente une borne supérieure à tout mécanisme respectant les contraintes de correction et transparence.

Enfin nous avons étudié l'espace des propriétés testables. Nous sommes repartis d'une notion de testabilité dépendant d'une relation entre l'ensemble des séquences d'exécution productibles par le programme et une r-propriété. Selon les différentes relations considérées, nous avons donné l'ensemble des r-propriétés testables selon cette relation. De plus, nous avons vu que le cadre des r-propriétés permettait d'obtenir un oracle de test décidable pour produire un verdict en fonction de la séquence de test jouée sur le système sous test.

Perspectives. L'approche que nous avons proposé soulève de nouvelles questions ouvertes et perspectives.

Perspective générale Une première perspective[10] serait d'étudier l'impact d'une *observabilité partielle* des événements. Ceci pourrait arriver si la propriété désirée fait référence à des événements hors du cadre d'observation des moniteurs. Il paraît assez naturel d'envisager que le moniteur ne puisse pas avoir accès à un certain nombre d'événements du système. En effet, la visibilité des actions pour le moniteur repose sur la technique d'instrumentation du programme utilisée (voir Chapitre 2, Section 2.1). Si cette technique d'instrumentation est automatique, elle repose sur une technique d'analyse du programme. Avec une technique d'analyse donnée, il est possible que certains événements ne soient pas instrumentables. De plus, si l'ensemble du code du système à vérifier n'est pas accessible (approche boite grise ou boite noire), un certain nombre d'événements ne sont plus à la portée de l'observation du moniteur. Par exemple, les techniques d'instrumentation par aspects permettent l'observation d'événements bien particuliers sur le système. Il est possible d'observer les appels de procédures, les lectures/écritures de variables de classe ; à l'opposition des variables utilisées dans le corps des procédures. Les ensembles caractérisés précédemment pour les différentes techniques seraient sûrement réduits.

Une piste qui nous semble intéressante serait de tirer parti de la combinaison de cette approche avec celles présentées dans [DJM09, DJM07, JMRT04, MDJ09]. Ces approches sont assez similaires à celles présentées dans cette thèse. Les auteurs s'intéressent au problème du test, de la vérification à l'exécution, et du contrôle de propriétés sur les systèmes à événements discrets. Le type de propriétés auxquelles s'intéressent ces approches sont des propriétés orientées sécurité, *e.g.*, propriété de confidentialité, intégrité, contrôle d'accès... Une autre différence est que le système considéré dans ces approches est fourni avec son modèle dont le comportement est connu. Ceci permet de traiter une observabilité partielle du système.

Perspectives côté vérification à l'exécution. Une deuxième perspective serait de caractériser de façon similaire l'espace de propriétés que l'on peut considérer avec d'autres techniques basées sur la vérification à l'exécution (*e.g.*, la technique de *runtime reflection* [LS08, Bau08]).

Perspectives côté enforcement à l'exécution. Une autre perspective est motivée par l'ajout d'expressivité aux mécanismes d'enforcement. Ceci reposerait sur un relâchement des contraintes de correction et transparence. Ces dernières pourraient être définies par exemple selon une relation d'équivalence entre les séquences lues et produites par un moniteur d'enforcement. De tels mécanismes d'enforcement, augmentés du pouvoir de transformation suffisant, pourraient avoir plus de contrôle sur la séquence qui leur est fournie en entrée. Une application envisageable serait d'utiliser de tels mécanismes pour *corriger* les mauvaises séquences d'exécution produites par un programme non sûr. Le moniteur d'enforcement pourrait, lorsque le programme sous-jacent dévie du comportement attendu, faire appel à des routines produites par un autre programme prouvé correct.

L'approche enforcement de propriétés suppose l'utilisation d'une mémoire finie, mais non bornée. On peut remarquer que le fait de pouvoir enforcer les propriétés de *response* repose sur le fait que les séquences (infinies) correctes ont régulièrement des préfixes corrects. Ainsi, à partir d'un préfixe d'une séquence (infinie) correcte, il ne peut y avoir qu'un nombre fini de continuations incorrectes avant la prochaine continuation correcte. Ceci garantit que la mémoire nécessaire est toujours finie, mais non bornée. Introduire des contraintes sur le mécanisme mémoire impacte l'ensemble des propriétés enforçables. De nouvelles questions ouvertes sont alors soulevées. Notamment, nous pouvons, par exemple, nous demander quel est l'espace des propriétés enforçables dans le cas d'une mémoire de taille donnée.

Perspectives côté test. Les relations possibles entre l'ensemble des séquences du programme sous test et la r-propriété sont assez simples. Il semblerait intéressant d'étendre ces relations pour prendre en compte des propriétés plus fines. Par exemple, il est possible d'envisager de définir une relation entre le programme et la r-propriété exprimant une certaine *disponibilité* ou *réactivité* du système. Cette relation serait similaire à la relation $\Pi \subseteq \mathit{Exec}(\mathcal{P}_\Sigma)$ en exprimant (informellement) "depuis un état initialisé du système, il est possible de jouer chaque séquence dans Π" sur le système. Un cadre d'application serait de tester la disponibilité de service. Ce qui permettrait par exemple de fournir des tests contre les dénis de service. Ici, il semble intéressant de voir comment l'approche proposée dans [MDJ09] où des techniques de génération automatique de tests et de synthèse de contrôleurs sont utilisées pour tester des propriétés de sécurité.

10. Cette perspective est en partie le résultat d'une discussion avec Thierry Jéron et Hervé Marchand lors d'une visite à l'IRISA Rennes.

Un autre axe de travail est autour de l'affaiblissement de la condition de testabilité. D'une certaine manière, nous avons sur-approximé l'ensemble des comportements possibles du programme, après l'observation d'une séquence donnée. Prenons par exemple la relation $\mathit{Exec}(\mathcal{P}_\Sigma) \subseteq \Pi$. Pour obtenir un verdict *fail* pour cette relation, il faut trouver une séquence d'exécution du programme qui n'appartient pas à la r-propriété considérée. Pour cela, après une observation σ, il faut s'assurer qu'il n'existe pas de continuation $\sigma \cdot \sigma'$ (finie ou infinie) *dans l'ensemble des séquences d'exécution du programme*. Nous avons sur-approximé les continuations possibles de σ produites par le programme par l'ensemble des continuations possibles de σ. Ensuite, nous avons utilisé le fait que notre approximation était une condition suffisante pour établir le verdict recherché. Il semble intéressant d'étudier comment il serait possible d'affaiblir cette condition. Notamment, en utilisant une certaine forme de spécification du programme. S'il est possible de connaître à tout moment les comportements possibles du programme (ou un sur-ensemble de ceux-ci), alors certainement qu'il existerait plus de cas où des verdicts peuvent être déterminés.

Une autre perspective serait de combiner l'approche présentée pour les verdicts faibles à une notion de *couverture*. Des notions de couverture existent pour le test à partir d'exigences (cf. Chapitre 2 Section 2.3). Une piste à explorer serait d'étudier comment ces notions de couverture permettent de renforcer un ensemble de verdicts faibles.

Deuxième partie

Approches pour la validation à l'exécution

CHAPITRE 5

Préliminaires aux approches

Sommaire

5.1 À propos des états d'un automate de Streett 93
5.2 Retour sur les notions de monitorabilité et testabilité 95
5.3 Notion de moniteur 96

Ce chapitre présente quelques notions communes et préliminaires aux Chapitres 6 et 7. Dans la suite, nous considérons un automate de Streett $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \rightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$.

5.1 À propos des états d'un automate de Streett

Les moniteurs (de vérification et d'enforcement) seront obtenus à partir des automates de Streett. Pour cela, nous définissons un ensemble de sous-ensembles des états d'un automate de Streett : $\mathbb{P}^{\mathcal{A}} = \{Good^{\mathcal{A}}, Good_p^{\mathcal{A}}, Bad_p^{\mathcal{A}}, Bad^{\mathcal{A}}\}$ est un ensemble de sous-ensembles de $Q^{\mathcal{A}}$, *t.q.* $Good^{\mathcal{A}}$, $Good_p^{\mathcal{A}}$, $Bad_p^{\mathcal{A}}$, $Bad^{\mathcal{A}}$ désignent respectivement les états bons, présumablement bons, présumablement mauvais, et mauvais de $\mathcal{A}$. Les états bons sont les états marqués comme récurrent ou persistent pour chaque paire et à partir desquels on ne peut atteindre que de tels états. Les états présumablement bons sont les états marqués comme récurrents ou persistants pour chaque paire, mais à partir desquels on peut atteindre un état qui n'est pas marqué comme récurrent ou persistant pour toutes les paires. Similairement, les états mauvais sont les états qui ne sont pas marqués comme récurrent ni comme persistant pour au moins une paire, et à partir desquels on ne peut atteindre que de tels états. Les états présumablement mauvais sont les états qui ne sont pas marqués comme récurrents ni comme persistants pour au moins une paire, mais à partir desquels on peut atteindre un état marqué comme récurrent ou persistant pour toutes les paires.

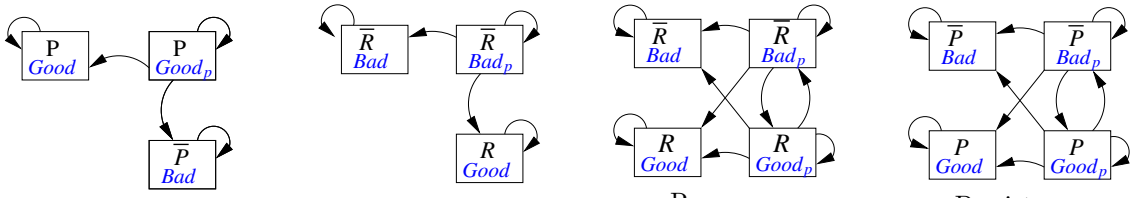
L'ensemble $\mathbb{P}^{\mathcal{A}}$ est défini comme suit :

Définition 31 (Caractérisation des états d'un automates de Streett). Étant donné un automate de Streett $\mathcal{A}$, nous définissons $\mathbb{P}^{\mathcal{A}} = \{Good^{\mathcal{A}}, Good_p^{\mathcal{A}}, Bad_p^{\mathcal{A}}, Bad^{\mathcal{A}}\}$ comme suit :

- $Good^{\mathcal{A}} = \{q \in Q^{\mathcal{A}} \cap \bigcap_{i=1}^m (R_i \cup P_i) \mid Reach_{\mathcal{A}}(q) \subseteq \bigcap_{i=1}^m (R_i \cup P_i)\}$
- $Good_p^{\mathcal{A}} = \{q \in Q^{\mathcal{A}} \cap \bigcap_{i=1}^m (R_i \cup P_i) \mid Reach_{\mathcal{A}}(q) \cap \bigcup_{i=1}^m (\overline{R_i} \cup \overline{P_i}) \neq \emptyset\}$
- $Bad_p^{\mathcal{A}} = \{q \in Q^{\mathcal{A}} \cap \bigcup_{i=1}^m (\overline{R_i} \cap \overline{P_i}) \mid Reach_{\mathcal{A}}(q) \not\subseteq \bigcup_{i=1}^m (\overline{R_i} \cap \overline{P_i})\}$
- $Bad^{\mathcal{A}} = \{q \in Q^{\mathcal{A}} \cap \bigcup_{i=1}^m (\overline{R_i} \cap \overline{P_i}) \mid Reach_{\mathcal{A}}(q) \subseteq \bigcup_{i=1}^m (\overline{R_i} \cap \overline{P_i})\}$

Notons que $\mathbb{P}^{\mathcal{A}}$ ou sous-ensemble [1] de ses élements forme toujours une partition des états de l'automate de Streett, *i.e.*, nous avons $Q^{\mathcal{A}} = Good^{\mathcal{A}} \cup Good_p^{\mathcal{A}} \cup Bad_p^{\mathcal{A}} \cup Bad^{\mathcal{A}}$, et ses éléments sont deux à deux d'intersection vide.

Exemple 17 (Caractérisation des états d'un automate de Streett) Nous illustrons la caractérisation de la Définition 31 sur des représentations schématiques des automates de Streett pour les classes de base.



Safety Guarantee Response Persistence

Exemple 18 (Caractérisation des états d'un automates de Streett) Nous présentons quelques exemples de caractérisation des états de certains automates de Streett introduits dans le Chapitre 3.

- Sur l'automate de la Fig. 6.3 (p. 112), nous avons : $Good_p^{\mathcal{A}_{\Pi_1}} = \{1,3\}$, et $Bad^{\mathcal{A}_{\Pi_1}} = \{2\}$.
- Sur l'automate de la Fig. 6.5 (p. 113), nous avons : $Good^{\mathcal{A}_{\Pi_2}} = \{3\}$, et $Bad_p^{\mathcal{A}_{\Pi_2}} = \{1,2\}$.

États d'un automate de Streett et évaluation de séquences. La caractérisation des états d'un automate de Streett permet de déterminer l'évaluation d'une séquence finie lorsque celle-ci est lue par l'automate de Streett. Lorsqu'une séquence est lue par un automate de Streett, le dernier état atteint lors du run de cette séquence nous donne l'évaluation par rapport à la r-propriété de cette séquence au sens de la Définition 24. Ce qui est précisé par la propriété suivante.

Propriété 7 (Correspondance entre $\mathbb{P}^{\mathcal{A}}$ et $\mathbb{B}_4$) : *Étant donnés un m-automate $\mathcal{A}_\Pi$, Π une r-propriété, et une séquence $\sigma \in \Sigma^*$ de longueur n t.q. $run(\sigma, \mathcal{A}_\Pi) = q_0 \cdots q_{n-1}$, nous avons que :*

$$q_{n-1} \in Good^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_4}(\sigma) = \top, \qquad q_{n-1} \in Bad_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_4}(\sigma) = \bot_p,$$
$$q_{n-1} \in Good_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_4}(\sigma) = \top_p, \qquad q_{n-1} \in Bad^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_4}(\sigma) = \bot.$$

◇

Preuve : Dans cette preuve, nous notons $[\![\Pi]\!]$ pour $[\![\Pi]\!]_{\mathbb{B}_4}$. Considérons une séquence d'exécution $\sigma \in \Sigma^*$ de longueur n.

Preuve de $q_{n-1} \in Good^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!](\sigma) = \top$

- Supposons que $q_{n-1} \in Good^{\mathcal{A}_\Pi}$. En utilisant le critère d'acceptation sur les séquences finies, nous avons que σ est acceptée par $\mathcal{A}_\Pi$. De plus, comme $\mathcal{A}_\Pi$ reconnaît Π, nous avons que $\Pi(\sigma)$. Maintenant, considérons $\mu \in \Sigma^+$ *t.q.* $|\sigma| + |\mu| = n' > n$ et $run(\sigma \cdot \mu, \mathcal{A}_\Pi) = q_0 \cdots q_{n'-1}$. Comme $q_{n-1} \in Good^{\mathcal{A}_\Pi}$, nous avons que $\forall k \in \mathbb{N} : n \leq k \leq n'-1 \Rightarrow q_k \in \bigcap_{i=1}^m R_i \cup P_i$ et par conséquent $\Pi(\sigma \cdot \mu)$. Considérons $\mu \in \Sigma^\omega$, nous pouvons remarquer que $\forall i \in [1,m] : vinf(\sigma \cdot \mu, \mathcal{A}_\Pi) \cap R_i \neq \emptyset \vee vinf(\sigma \cdot \mu, \mathcal{A}_\Pi) \subseteq P_i$, ce qui implique que $\Pi(\sigma \cdot \mu)$. Nous avons $\Pi(\sigma) \wedge \forall \mu \in \Sigma^\infty : \Pi(\sigma \cdot \mu)$, *i.e.*, $[\![\Pi]\!](\sigma) = \top$.
- Réciproquement, supposons que $[\![\Pi]\!](\sigma) = \top$. Par définition, cela signifie $\forall \mu \in \Sigma^\infty : \Pi(\sigma \cdot \mu)$. Selon le critère d'acceptation des automates de Streett, nous déduisons que $\forall k \geq n, \forall \mu \in \Sigma^* : run(\sigma \cdot \mu, \mathcal{A}_\Pi) = q_0 \cdots q_{n-1} \cdots q_k \Rightarrow q_k \in \bigcap_{i=0}^m R_i \cup P_i$. C'est-à-dire $Reach_{\mathcal{A}_\Pi}(q_{n-1}) \subseteq \bigcap_{i=1}^m (R_i \cup P_i)$, *i.e.*, $q_{n-1} \in Good^{\mathcal{A}_\Pi}$.

Preuve de $q_{n-1} \in Good_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!](\sigma) = \top_p$. La preuve de $q_{n-1} \in Good_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!](\sigma) = \top_p$ est directe en examinant le critère d'acceptation des séquences finies des automates de Streett.

- Supposons que $q_{n-1} \in Good_p^{\mathcal{A}_\Pi}$. En utilisant le critère d'acceptation des séquences finies, nous avons que σ est acceptée par $\mathcal{A}_\Pi$. De plus, comme $\mathcal{A}_\Pi$ spécifie Π, nous avons que $\Pi(\sigma)$. Maintenant, comme $Reach_{\mathcal{A}}(q) \nsubseteq \bigcap_{i=1}^m (R_i \cup P_i)$, il existe un état q' de $\mathcal{A}_\Pi$ atteignable depuis q et appartenant à $\bigcup_{i=1}^m (\overline{R_i} \cap \overline{P_i})$. En conséquence, il existe $\mu \in \Sigma^*$ *t.q.* $run(\sigma \cdot \mu) = q_0 \cdots q_{n-1} \cdots q'$. Suivant toujours le critère d'acceptation, nous déduisons que $\neg\Pi(\sigma \cdot \mu)$, *i.e.*, $[\![\Pi]\!](\sigma) = \top_p$.
- Réciproquement, le même raisonnement en utilisant le critère d'acceptation peut être mené pour prouver le résultat recherché. ■

1. Certains éléments de $\mathbb{P}^{\mathcal{A}}$ peuvent être vides.

Preuve de $q_{n-1} \in Bad_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!](\sigma) = \bot_p$. Similairement, prouver que $q_{n-1} \in Bad_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!](\sigma) = \bot_p$ est direct en examinant le critère d'acceptation des séquences finies des automates de Streett. La preuve complète peut être trouvée dans l'Annexe A.3.

Preuve de $q_{n-1} \in Bad^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!](\sigma) = \bot$. Prouver que $q_{n-1} \in Bad^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!](\sigma) = \bot$ peut être fait en suivant le même principe de preuve que celui que nous avons utilisé pour prouver $q_{n-1} \in Good^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!](\sigma) = \top$. La preuve complète peut être trouvée dans l'Annexe A.3.

Remarque 8 (Correspondance entre $\mathbb{P}$, $\mathbb{B}_3$, $\mathbb{B}_2^\bot$, et $\mathbb{B}_2^\top$) Une fois l'ensemble $\mathbb{P}$ défini pour un automate de Streett, il est facile d'obtenir les évaluations dans les domaines de vérité de cardinaux inférieurs :

- Pour $\mathbb{B}_3$:
  - $q_{n-1} \in Good^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_3}(\sigma) = \top$,
  - $q_{n-1} \in Good_p^{\mathcal{A}_\Pi} \cup Bad_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_3}(\sigma) =?$,
  - $q_{n-1} \in Bad^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_3}(\sigma) = \bot$.
- Pour $\mathbb{B}_2^\top$:
  - $q_{n-1} \in Good^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_2^\top}(\sigma) = \top$,
  - $q_{n-1} \in Good_p^{\mathcal{A}_\Pi} \cup Bad_p^{\mathcal{A}_\Pi} \cup Bad^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_2^\top}(\sigma) =?$,
- Pour $\mathbb{B}_2^\bot$:
  - $q_{n-1} \in Bad^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_2^\bot}(\sigma) = \bot$,
  - $q_{n-1} \in Bad^{\mathcal{A}_\Pi} \cup Good_p^{\mathcal{A}_\Pi} \cup Bad_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_2^\bot}(\sigma) =?$. *

5.2 Retour sur les notions de monitorabilité et testabilité

Nous avons vu dans le Chapitre 4, Section. 4.2 qu'il n'y a pas de caractérisation exacte (en terme de classe spécifique de la classification *Safety-Progress*) des propriétés vérifiables à l'exécution dans sa *définition classique* (Définition 23, p. 66). Les notions de monitorabilité et testabilité reposent sur la notion de détermination positive et détermination négative. Une r-propriété Π est déterminée négativement (resp. positivement) par une séquence d'exécution σ si lors de la lecture de la séquence par $\mathcal{A}_\Pi$ reconnaissant Π, l'automate $\mathcal{A}_\Pi$ est dans un état mauvais (resp. bon).

Retour sur la monitorabilité. Étant donnés une r-propriété Π et un automate de Streett $\mathcal{A}_\Pi$ reconnaissant Π, il est possible de déterminer si cette r-propriété est monitorable par une analyse d'atteignabilité des états de l'automate.

Propriété 8 (Propriétés monitorables (vue automate)) : *La r-propriété Π reconnue par le m-automate de Streett $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \rightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ est*

- *$\mathbb{B}_2^\bot$-monitorable ssi*
  $\forall q \in Q^{\mathcal{A}_\Pi}, q_{\text{init}} \rightarrow^*_{\mathcal{A}_\Pi} q \Rightarrow \exists q' \in Bad^{\mathcal{A}_\Pi}, q \rightarrow^*_{\mathcal{A}_\Pi} q'$;
- *$\mathbb{B}_2^\top$-monitorable ssi*
  $\forall q \in Q^{\mathcal{A}_\Pi}, q_{\text{init}} \rightarrow^*_{\mathcal{A}_\Pi} q \Rightarrow \exists q' \in Good^{\mathcal{A}_\Pi}, q \rightarrow^*_{\mathcal{A}_\Pi} q'$;
- *$\mathbb{B}_3$-monitorable ssi*
  $\forall q \in Q^{\mathcal{A}_\Pi}, q_{\text{init}} \rightarrow^*_{\mathcal{A}_\Pi} q \Rightarrow \exists q' \in Bad^{\mathcal{A}_\Pi} \cup Good^{\mathcal{A}_\Pi}, q \rightarrow^*_{\mathcal{A}_\Pi} q'$. ◇

Une r-propriété est donc $\mathbb{B}_2^\bot$-monitorable (resp. $\mathbb{B}_2^\top$-monitorable, $\mathbb{B}_3$-monitorable) si, dans son automate reconnaisseur, il est possible d'atteindre un état mauvais (resp. bon, mauvais ou bon) à partir de tout état de l'automate.

Preuve : Cette propriété est une conséquence de la Propriété 7. La preuve est évidente en remarquant que les automates de Streett que nous considérons sont déterministes et complets. Ainsi, les deux faits suivants sont équivalents :

- pouvoir atteindre, depuis tout état accessible depuis l'état initial, un état mauvais (resp. bon, mauvais ou bon) ;
- toute séquence finie possède une continuation qui détermine négativement (resp. positivement, négativement ou positivement) la propriété sous-jacente. ■

Retour sur la testabilité. De façon similaire, par un examen des états d'un automate de Streett, il est possible de savoir si la r-propriété reconnue par cette automate est testable vis-à-vis d'une relation.

DÉFINITION 32 (PROPRIÉTÉS TESTABLES (VUE AUTOMATE)). La r-propriété Π reconnue par le m-automate de Streett $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, {q_{\text{init}}}^{\mathcal{A}_\Pi}, \rightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ est

- testable par rapport à la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ si
  $\exists q \in Bad^{\mathcal{A}_\Pi} : {q_{\text{init}}}^{\mathcal{A}_\Pi} \rightarrow^*_{\mathcal{A}_\Pi} q$
- testable par rapport à la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ si
  $\exists q \in Good^{\mathcal{A}_\Pi} : {q_{\text{init}}}^{\mathcal{A}_\Pi} \rightarrow^*_{\mathcal{A}_\Pi} q$

Une r-propriété Π est donc testable par rapport à la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ (resp. par rapport à la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$) si, dans son automate reconnaisseur, il est possible d'atteindre un état mauvais (resp. bon) à partir de l'état initial de l'automate.

5.3 Notion de moniteur

Un moniteur est une procédure (cf. Chapitre 2, Section 2.1) consommant des événements produits par un programme sous-jacent et donnant une évaluation sur la séquence lue jusqu'à présent à propos d'une propriété. Les moniteurs que nous considérons sont des machines à états finis produisant une sortie dans un domaine approprié. Ce domaine sera caractérisé précisément pour les moniteurs dédiés à la vérification ou l'enforcement. Pour les moniteurs de vérification, cette fonction de sortie donne une valeur de vérité (un verdict) dans $\mathbb{B}_4$ concernant l'évaluation de la séquence d'exécution courante relativement à la propriété examinée (cf. Chapitre 6). Pour les moniteurs d'enforcement (EMs), cette fonction de sortie produit une opération d'enforcement induisant une modification de la séquence d'entrée de manière à enforcer la propriété désirée (cf. Chapitre 6). La définition des moniteurs que nous considérons est la suivante :

DÉFINITION 33 (MONITEUR). Un *moniteur* $\mathcal{A}$ est un 5-tuple $(Q^{\mathcal{A}}, {q_{\text{init}}}^{\mathcal{A}}, \longrightarrow_{\mathcal{A}}, X^{\mathcal{A}}, \Gamma^{\mathcal{A}})$ défini relativement à un ensemble d'événements Σ. L'ensemble fini d'états $Q^{\mathcal{A}}$ dénote les états de contrôle et ${q_{\text{init}}}^{\mathcal{A}} \in Q^{\mathcal{A}}$ est l'état initial. La fonction complète $\longrightarrow_{\mathcal{A}}: Q^{\mathcal{A}} \times \Sigma \rightarrow Q^{\mathcal{A}}$ est la fonction de transition. Dans la suite, nous notons $q \xrightarrow{a}_{\mathcal{A}} q'$ pour $\longrightarrow_{\mathcal{A}} (q, a) = q'$. L'ensemble des valeurs $X^{\mathcal{A}}$ dépend du but du moniteur (vérification ou enforcement). La fonction $\Gamma^{\mathcal{A}} : Q^{\mathcal{A}} \rightarrow X^{\mathcal{A}}$ est une fonction de sortie, produisant des valeurs dans $X^{\mathcal{A}}$ à partir des états.

À partir de cette définition générale de moniteur, il est possible de synthétiser des moniteurs dédiés pour la vérification et l'enforcement. Cette synthèse est basée sur la définition de $\mathbb{P}^{\mathcal{A}}$.

CHAPITRE 6

Approches Vérification et Enforcement de propriétés à l'exécution

Sommaire

6.1 Introduction **99**
6.2 Vérification par moniteur **99**
6.3 Enforcement par moniteur **100**
6.3.1 Notion de moniteur d'enforcement générique 100
6.3.2 Enforcement de propriétés par un moniteur. 102
6.3.3 Instantiation des moniteurs d'enforcement génériques 103
6.3.4 Propriétés des moniteurs d'enforcement instanciés 104
6.4 Operations de composition sur les moniteurs d'enforcement **105**
6.4.1 Négation 106
6.4.2 Union et intersection 108
6.5 Synthèse de moniteurs d'enforcement **110**
6.5.1 Transformations spécifiques aux classes de propriétés 111
6.5.2 Correction des transformations 115
6.5.3 Transformation générale 118
6.6 Comparaison avec d'autres approches d'enforcement **119**
6.6.1 Comparaison avec les mécanismes d'enforcement 119
6.6.2 Comparaison avec les moniteurs d'enforcement 119
6.7 Conclusion **120**

Chapter abstract

In this chapter, we present the verification and enforcement approaches via monitors operating at runtime on the system under scrutiny. These monitors take as input an execution sequence made of events produced by an underlying program or protocol. They produce as output a new execution sequence such that the soundness and transparency constraints, expressed in Chapter 4, hold. These enforcement monitors are based on a memory device and finite sets of control states and enforcement operations. Moreover, we specify their enforcement abilities wrt. the *general* safety-progress classification of properties. Furthermore, we propose a *systematic* technique to produce an enforcing monitor from the automaton recognizing a given safety, guarantee, obligation or response property. Finally, we show that this notion of enforcement monitors is more amenable to implementation and encompasses previous runtime enforcement mechanisms.

Résumé du chapitre

Dans ce chapitre nous présentons les approches de vérification et d'enforcement par moniteurs opérant à l'exécution des systèmes. Ces moniteurs prennent en entrée une séquence d'exécution consituée d'événements produits par exemple par un programme ou un protocole. Ils produisent en sortie une nouvelle séquence d'exécution telle que les contraintes de correction et transparence formulées au Chapitre 4 soient respectées. Ces moniteurs sont des mécanismes de type automate avec mémoire possédant un nombre fini d'états de contrôle et d'opérations d'enforcement. De plus, nous spécifions leur capacité d'enforcement par rapport à la classification *Safety-Progress* présentée dans le Chapitre 3. Nous verrons que ces moniteurs sont capables d'enforcer l'ensemble des propriétés enforçables délimitées dans le Chapitre 4. De plus, nous proposons une technique systématique pour produire de tels moniteurs d'enforcement depuis un automate de Streett reconnaissant une r-propriété donnée de safety, guarantee, obligation, response. Finalement, nous comparons nos moniteurs avec ceux existants, et montrons qu'ils sont plus facilement implantables.

6.1 Introduction

Motivations. Les modèles actuels de moniteurs d'enforcement souffrent de plusieurs limitations. Tout d'abord ceux-ci n'établissent pas de lien clair avec la propriété qu'ils enforcent. Plus précisément, il est difficile d'établir un lien entre l'état courant du moniteur d'enforcement et la satisfaction de la propriété par la séquence fournie en entrée. Ceci est du au fait qu'un edit automate n'est pas relié ou généré à partir d'un mécanisme reconnaisseur. En effet, les états d'un edit-automata sont uniquement définis par la modification opérée sur la séquence d'exécution (voir Chapitre 2 Section 2.2).

Nous souhaitons établir un cadre pour la vérification et l'enforcement de propriétés à l'exécution. Les approches que nous proposons utilisent le cadre de spécification défini précédemment dans le Chapitre 3. Nous définissons une notion de moniteurs génériques "assez puissants" pour vérifier et enforcer l'ensemble des propriétés vérifiables et enforçables trouvées au Chapitre 4. De plus, pour l'enforcement de propriétés, en tirant parti du cadre défini dans le Chapitre 3, nous verrons qu'il est possible de *synthétiser* facilement un moniteur d'enforcement pour une r-propriété (enforçable) donnée. Également, les moniteurs que nous proposons bénéficient d'un lien clair entre leur modèle (proposé dans ce chapitre) et leur implémentation (voir Chapitre 8). Aussi, nous nous intéresserons au problème de la composition de tels moniteurs, puis à leur synthèse.

Organisation du chapitre. La suite de ce chapitre est organisée comme suit. Dans la Section 6.2, nous présentons une notion générique de moniteur de vérification. La notion de moniteur d'enforcement générique est introduite dans la Section 6.3. Nous présentons également une instantiation de ces moniteurs d'enforcement. Pour les deux types de moniteurs, nous présentons la notion d'enforcement de propriétés. Nous montrons comment il est possible de composer ces moniteurs d'enforcement par des opérations booléennes en Section 6.4. Puis, dans la Section 6.5, nous montrons comment depuis un automate de Streett reconnaissant une r-propriété enforçable donnée, il est possible d'obtenir un moniteur d'enforcement respectant les contraintes de correction et transparence.

6.2 Vérification par moniteur

Dans cette section, nous considérons une r-propriété monitorable $\Pi \in MP(\mathbb{B}_4)$.

Définition 34 (Moniteur de vérification). Un *moniteur de vérification* (verification monitor, VM) $\mathcal{A}_?$ est un moniteur, *i.e.*, un 5-tuple $(Q^{\mathcal{A}_?}, q_{\text{init}}{}^{\mathcal{A}_?}, \longrightarrow_{\mathcal{A}_?}, \mathbb{B}_4, \Gamma)$. $\Gamma : Q^{\mathcal{A}_?} \to \mathbb{B}_4$ est une fonction de sortie, produisant des valeurs de vérité dans $\mathbb{B}_4$ depuis les états.

De tels moniteurs sont indépendants de tout formalisme de spécification, et peuvent être facilement adaptés au formalisme de spécification utilisé pour les générer. Classiquement, nous supposerons que les moniteurs sont déterministes et complets.

Nous définissons la notion de *séquence de vérification* produite par un moniteur et montrons ce que cela signifie pour un moniteur de vérifier une r-propriété sur un programme $\mathcal{P}_\Sigma$:

Définition 35 (Séquence de vérification). Nous définissons la vérification réalisée par un VM $\mathcal{A}_?$ en lisant un séquence d'entrée $\sigma \in \Sigma^*$ (produite par $\mathcal{P}_\Sigma$) et produisant une séquence dans $\mathbb{B}_4{}^+$. La fonction de vérification $[\![\mathcal{A}_?]\!](\cdot) : \Sigma^* \to \mathbb{B}_4{}^+$ dépend de la fonction auxiliaire $[\![\mathcal{A}_?]\!](\cdot,\cdot) : \Sigma^+ \times Q^{\mathcal{A}_?} \to \mathbb{B}_4{}^+$ définissant la vérification réalisée par $\mathcal{A}_?$ selon l'état courant : $[\![\mathcal{A}_?]\!](\sigma, q)$ est la séquence de vérification produite par $\mathcal{A}_?$ par la lecture de σ depuis l'état q. Les fonctions $[\![\mathcal{A}_?]\!](\cdot,\cdot)$ et $[\![\mathcal{A}_?]\!](\cdot)$ sont définies comme suit :

- $[\![\mathcal{A}_?]\!](a \cdot \sigma, q) = b \cdot [\![\mathcal{A}_?]\!](\sigma, q')$ avec $q \xrightarrow{a}_{\mathcal{A}_?} q' \wedge \Gamma(q') = b$.
- $[\![\mathcal{A}_?]\!](a, q) = \Gamma(q')$ avec $q \xrightarrow{a}_{\mathcal{A}_?} q'$
- $[\![\mathcal{A}_?]\!](\sigma) = \Gamma(q_{\text{init}}{}^{\mathcal{A}_?}) \cdot [\![\mathcal{A}_?]\!](\sigma, q_{\text{init}}{}^{\mathcal{A}_?})$.

Une séquence d'exécution $a \cdot \sigma$ est (incrémentalement) vérifiée selon la transition déclenchée par l'entrée a : l'entrée a est vérifiée avec la séquence lue jusqu'à présent et produit la valeur de vérité b. Ensuite, la séquence restante est vérifiée. La séquence σ est vérifiée par $\mathcal{A}_?$ si elle est vérifiée depuis l'état initial.

DÉFINITION 36 (VÉRIFICATION DE PROPRIÉTÉ). $\mathcal{A}_?$ vérifie $\Pi \in MP(\mathbb{B}_4)$ sur $\mathcal{P}_\Sigma$, ce qui est noté $Ver(\mathcal{A}_?, \Pi, \mathcal{P}_\Sigma)$, *ssi* pour toute séquence $\sigma \in Exec(\mathcal{P}_\Sigma) \cap \Sigma^*$, il existe $b \in \mathbb{B}_4{}^+$, *t.q.* les contraintes suivantes soient satisfaites :

$$[\![\mathcal{A}_?]\!](\sigma) = b \tag{6.1}$$

$$b_0 = [\![\Pi]\!](\epsilon) \tag{6.2}$$

$$\forall i \in \mathbb{N}, b_{i+1} = [\![\Pi]\!](\sigma_{\dots i}) \tag{6.3}$$

(6.1) exprime que la séquence de vérification b est produite par $\mathcal{A}_?$;
(6.2) et (6.3) expriment la correction de b.

En utilisant l'ensemble $\mathbb{P}^{\mathcal{A}_\Pi}$ d'un automate de Streett $\mathcal{A}_\Pi$, nous montrons comment il est possible d'obtenir un moniteur de vérification pour la r-propriété Π.

DÉFINITION 37 (TRANSFORMATION STREETT2VM). Soit $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \dots, (R_m, P_m)\})$ un automate de Streett reconnaissant $\Pi \in MP(\mathbb{B}_4)$. Nous définissons la transformation $\text{Streett2VM}(\mathcal{A}_\Pi) = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \rightarrow_{\mathcal{A}_\Pi}, \mathbb{B}_4, \Gamma)$ *t.q.* $\Gamma : Q^{\mathcal{A}_?} \rightarrow \mathbb{B}_4$ produit des valeurs de vérité depuis les états selon l'ensemble $\mathbb{P}^{\mathcal{A}_\Pi}$: $\forall q \in Q^{\mathcal{A}_\Pi}$:

$$q \in Good^{\mathcal{A}_\Pi} \Rightarrow \Gamma(q) = \top \qquad q \in Good_p^{\mathcal{A}_\Pi} \Rightarrow \Gamma(q) = \top_p$$
$$q \in Bad_p^{\mathcal{A}_\Pi} \Rightarrow \Gamma(q) = \bot_p \qquad q \in Bad^{\mathcal{A}_\Pi} \Rightarrow \Gamma(q) = \bot$$

Théorème κ (Correction de Streett2VM) : Π *est vérifiable sur* $\mathcal{P}_\Sigma$ *par un VM obtenu par l'application de Streett2VM. Plus formellement, étant donné* $\mathcal{A}_\Pi$ *reconnaissant* Π*, nous avons :* $(\Pi \in MP(\mathbb{B}_4) \wedge \mathcal{A}_{?\Pi} = \text{Streett2VM}(\mathcal{A}_\Pi)) \Rightarrow Ver(\mathcal{A}_{?\Pi}, \Pi, \mathcal{P}_\Sigma)$

PREUVE : La correction de cette transformation repose sur la correction de la caractérisation des états d'un automate de Streett donnée dans la Définition 31 et montrée dans la Propriété 7 (p. 94). ■

6.3 Enforcement par moniteur

Dans cette section, nous introduisons la notion de moniteur d'enforcement. Nous présentons une notion générale de moniteur paramétré par un ensemble générique d'opérations d'enforcement. Puis, nous définissons comment de tels moniteurs enforcent une propriété. Ensuite, nous instantions ces moniteurs avec un ensemble d'opérations d'enforcement "minimales" et étudions la notion d'enforcement pour ces moniteurs. Également, nous étudions les propriétés vérifiées par ces moniteurs particuliers.

6.3.1 Notion de moniteur d'enforcement générique

Nous définissons la notion centrale de moniteur d'enforcement. Un tel dispositif, prévu pour opérer à l'exécution des systèmes, surveille un programme cible en examinant ses événements intéressants. A chaque entrée produite par le programme son état évolue et une opération d'enforcement est produite. Les moniteurs d'enforcement sont paramétrés par un ensemble d'opérations d'enforcement *Ops*.

DÉFINITION 38 (OPÉRATIONS D'ENFORCEMENT *Ops*). Les opérations d'enforcement sont destinées à opérer une modification de la mémoire interne du moniteur d'enforcement et produisent potentiellement une sortie. Plus spécifiquement, ils prennent comme entrée un événement et un contenu mémoire (*i.e.*, une séquence d'événements) pour produire une séquence de sortie et un nouveau contenu mémoire : $Ops = (\Sigma \times \Sigma^*) \rightarrow (\Sigma^* \times \Sigma^*)$.

Définition 39 (Moniteur d'enforcement générique (EM(Ops))). Un *moniteur d'enforcement* (enforcement monitor : EM) $\mathcal{A}_\downarrow$ est un moniteur, *i.e.*, un 5-tuple $(Q^{\mathcal{A}_\downarrow}, {q_{\text{init}}}^{\mathcal{A}_\downarrow}, \longrightarrow_{\mathcal{A}_\downarrow}, Ops, \Gamma^{\mathcal{A}_\downarrow})$ où la fonction de sortie produit des opérations d'enforcement de Ops sur chaque état de l'automate.

Les notions de *run* et de *trace* pour les EMs se transposent naturellement depuis ces mêmes notions définies pour les automate de Streett (cf. Chapitre 3 Section 3.6). Dans la suite de cette section, $\sigma \in \Sigma^\infty$ désigne une séquence d'exécution d'un programme et $\mathcal{A}_\downarrow = (Q^{\mathcal{A}_\downarrow}, {q_{\text{init}}}^{\mathcal{A}_\downarrow}, \longrightarrow_{\mathcal{A}_\downarrow}, Ops, \Gamma^{\mathcal{A}_\downarrow})$ désigne un EM(Ops).

Définition 40 (Run et trace). Le *run* de σ sur $\mathcal{A}_\downarrow$ est la séquence des états impliqués par l'exécution de $\mathcal{A}_\downarrow$ lorsque σ lui est donné en entrée. Ceci est formellement défini comme $run(\sigma, \mathcal{A}_\downarrow) = q_0 \cdot q_1 \cdots$ où $q_0 = {q_{\text{init}}}^{\mathcal{A}_\downarrow} \wedge \forall i : (q_i \in Q^{\mathcal{A}_\downarrow} \wedge q_i \xrightarrow{\sigma_i}_{\mathcal{A}_\downarrow} q_{i+1})$. La *trace* résultant de l'exécution de σ sur $\mathcal{A}_\downarrow$ est la séquence (finie ou non) de triplets $(q_0, \sigma_0/\alpha_0, q_1) \cdot (q_1, \sigma_1/\alpha_1, q_2) \cdots (q_i, \sigma_i/\alpha_i, q_{i+1}) \cdots$ où $run(\sigma, \mathcal{A}_\downarrow) = q_0 \cdot q_1 \cdots$ et $\forall i : \alpha_i = \Gamma(q_{i+1})$.

Ainsi, un moniteur d'enforcement n'applique pas d'opération d'enforcement sur l'état initial lors de la soumission d'une séquence d'entrée. Celui-ci réalise la première opération sur le premier état visité et le premier événement de la séquence d'entrée.

Nous formalisons la manière dont un EM(Ops) réagit à une séquence d'exécution donnée en entrée, étant donné un programme cible. Ceci repose sur les notions standards de *configuration* et *dérivation*.

Définition 41 (Configurations et dérivations d'un EM(Ops)). Pour un EM(Ops) $(Q^{\mathcal{A}_\downarrow}, {q_{\text{init}}}^{\mathcal{A}_\downarrow}, \longrightarrow_{\mathcal{A}_\downarrow}, Ops, \Gamma^{\mathcal{A}_\downarrow})$, une *configuration* est un triplet $(q, \sigma, m) \in Q^{\mathcal{A}_\downarrow} \times \Sigma^* \times \Sigma^*$ où q dénote l'état de contrôle courant, σ la séquence courante d'entrée, et m le contenu mémoire courant.

Nous disons qu'une configuration (q', σ', m') est *dérivable en un pas* depuis la configuration (q, σ, m) et *produit la sortie* $o \in \Sigma^*$, et nous notons $(q, \sigma, m) \overset{o}{\hookrightarrow} (q', \sigma', m')$ *ssi* $\sigma = a \cdot \sigma' \wedge q \xrightarrow{a}_{\mathcal{A}_\downarrow} q' \wedge \Gamma(q') = \alpha \wedge \alpha(a, m) = (o, m')$.

Nous disons qu'une configuration C' est *dérivable en plusieurs pas* depuis la configuration C et *produit la sortie* $o \in \Sigma^*$, et nous notons $C \overset{o}{\Longrightarrow}_{\mathcal{A}_\downarrow} C'$, *ssi* il existe $k \geq 0$ et des configurations C_0, C_1, ..., C_k *t.q.* $C = C_0$, $C' = C_k$, $C_i \overset{o_i}{\hookrightarrow} C_{i+1}$ pour tout $0 \leq i < k$, et $o = o_0 \cdot o_1 \cdots o_{k-1}$.

La notion d'enforcement est basée sur la manière dont un moniteur transforme une séquence d'entrée en une séquence de sortie. Pour les définitions à venir, nous distinguerons les séquences finies et infinies.

Définition 42 (Transformation de séquences depuis un état et un contenu mémoire). La séquence $\sigma \in \Sigma^*$ est transformée par $\mathcal{A}_\downarrow$ depuis l'état $q \in Q^{\mathcal{A}_\downarrow}$ et le contenu mémoire $m \in \Sigma^*$ en la séquence $o \in \Sigma^*$, ce qui est noté $(q, \sigma, m) \downarrow_{\mathcal{A}_\downarrow} o$, si $\exists q' \in Q^{\mathcal{A}_\downarrow}, \exists m' \in \Sigma^*$ *t.q.* $(q, \sigma, m) \overset{o}{\Longrightarrow}_{\mathcal{A}_\downarrow} (q', \epsilon, m')$. C'est-à-dire, s'il existe une dérivation démarrant depuis une configuration dont l'état est q, la mémoire (initiale) m, et produisant o en sortie.

Définition 43 (Transformation de séquences par un moniteur). Nous définissons la transformation réalisée par un EM(Ops) lorsque celui-ci lit une séquence d'entrée $\sigma \in \Sigma^\infty$ et produisant une séquence de sortie $o \in \Sigma^\infty$. La relation $\Downarrow_{\mathcal{A}_\downarrow} \subseteq \Sigma^\infty \times \Sigma^\infty$ est définie comme suit :

- La séquence vide ϵ n'est pas transformée par $\mathcal{A}_\downarrow$, *i.e.*, $\epsilon \Downarrow_{\mathcal{A}_\downarrow} \epsilon$. Ceci se produit lorsque le programme sous-jacent ne produit pas d'événement.
- La séquence $\sigma \in \Sigma^+$ est transformée par $\mathcal{A}_\downarrow$ en la séquence $o \in \Sigma^*$, lorsque $({q_{\text{init}}}^{\mathcal{A}_\downarrow}, \sigma, \epsilon) \downarrow_{\mathcal{A}_\downarrow} o$, ce que nous notons $\sigma \Downarrow_{\mathcal{A}_\downarrow} o$. C'est-à-dire, la séquence est transformée depuis l'état initial de l'EM avec un contenu mémoire vide.

– La séquence $\sigma \in \Sigma^\omega$ est transformée par $\mathcal{A}_\downarrow$ en la *séquence finie* $o \in \Sigma^*$, ce qui est noté $\sigma \Downarrow_{\mathcal{A}_\downarrow} o$, si

$$\exists \sigma' \in \Sigma^* : \sigma' \prec \sigma \wedge \sigma' \Downarrow_{\mathcal{A}_\downarrow} o \wedge \forall \sigma'' \in \Sigma^* : \sigma' \prec \sigma'' \prec \sigma \Rightarrow \sigma'' \Downarrow_{\mathcal{A}_\downarrow} o.$$

C'est-à-dire, la séquence finie o est produite s'il existe un préfixe de σ qui produit o, et chaque continuation de ce préfixe produit o également.

– La séquence $\sigma \in \Sigma^\omega$ est transformée par $\mathcal{A}_\downarrow$ en la *séquence infinie* $o \in \Sigma^\omega$, ce qui est noté $\sigma \Downarrow_{\mathcal{A}_\downarrow} o$, si les deux contraintes suivantes sont vérifiées :

$$\forall o' \in \Sigma^* : o' \prec o \Rightarrow \exists \sigma'', o'' \in \Sigma^*, o' \prec o'' \wedge \sigma'' \Downarrow_{\mathcal{A}_\downarrow} o'' \tag{6.4}$$

$$\forall \sigma', o' \in \Sigma^* : \ (\sigma' \prec \sigma \wedge \sigma' \Downarrow_{\mathcal{A}_\downarrow} o') \Rightarrow o' \prec o \tag{6.5}$$

C'est-à-dire, la séquence infinie o est produite si pour tous les préfixes de o, il existe un préfixe plus long qui peut être produit.

Sur les séquences finies, la transformation de séquences peut être définie de manière inductive comme suit par une fonction de transformation.

Définition 44 (Transformation de séquence). La fonction de transformation $\Downarrow \mathcal{A}_!(\cdot) : \Sigma^* \to \Sigma^*$ repose sur la fonction $\Downarrow \mathcal{A}_!(\cdot,\cdot,\cdot) : \Sigma^* \times Q^{\mathcal{A}_!} \times \Sigma^* \to \Sigma^*$ définissant la transformation réalisée à partir de l'état courant et le contenu mémoire courant : $\Downarrow_{\mathcal{A}_!} (\sigma, q, m)$ est la séquence de sortie produite en lisant σ depuis l'état q et un contenu mémoire (initial) m.

– La séquence ϵ n'est pas transformée par $\mathcal{A}_!$, *i.e.*, $\forall q \in Q^{\mathcal{A}_!}, \forall m \in \Sigma^* : \Downarrow \mathcal{A}_!(\epsilon, q, m) = \epsilon$.

– Une séquence d'exécution $a \cdot \sigma$ est (incrémentalement) transformée selon la transition déclenchée par l'entrée a : on applique l'opération d'enforcement de l'état d'arrivé de la transition sur le contenu mémoire courant et l'entrée a de l'état ; ce qui induit un nouveau contenu mémoire et une sortie o. C'est-à-dire :

$$\Downarrow \mathcal{A}_!(a \cdot \sigma, q, m) = o \cdot \Downarrow \mathcal{A}_!(\sigma, q', m') \ \text{ avec } \ q \xrightarrow{a}_{\mathcal{A}_!} q' \wedge \Gamma(q') = \alpha \wedge \alpha(a, m) = (o, m')$$

La définition précédente montre que *le problème de l'enforcement, et la transformation de séquences peut se réaliser de manière incrémentale* (ce que fait exactement un moniteur).

6.3.2 Enforcement de propriétés par un moniteur.

Nous définissons maintenant la notion d'enforcement de propriétés par un EM. La notion d'enforcement relie la séquence d'entrée produite (et fournie à l'EM) par un programme (ou un générateur de séquences d'exécution) et la séquence de sortie autorisée par celui-ci (correcte vis-à-vis de la r-propriété considérée). Dans le cas général, la comparaison entre les séquences d'entrées et de sortie est réalisée selon une relation d'équivalence entre les séquences : $\approx \subseteq \Sigma^\infty \times \Sigma^\infty$. Notons que la relation d'équivalence utilisée dans ce cas doit préserver la r-propriété considérée.

Définition 45 (Enforcement d'une r-propriété). Pour une r-propriété $\Pi = (\phi, \varphi) \in EP$, nous disons que $\mathcal{A}_!$ enforce Π sur $\mathcal{P}_\Sigma$, ce qui est noté $Enf_\approx(\mathcal{A}_!, \Pi, \mathcal{P}_\Sigma)$, *ssi* pour toute séquence $\sigma \in Exec(\mathcal{P}_\Sigma)$, il existe $o \in \Sigma^\infty$, *t.q.* les contraintes suivantes sont vérifiées :

$$\sigma \Downarrow_{\mathcal{A}_!} o \tag{6.6}$$

$$\Pi(\sigma) \Rightarrow \sigma \approx o \tag{6.7}$$

$$\neg\Pi(\sigma) \wedge Pref_\prec(\phi, \sigma) = \emptyset \Rightarrow o \approx \epsilon \tag{6.8}$$

$$\neg\Pi(\sigma) \wedge Pref_\prec(\phi, \sigma) \neq \emptyset \Rightarrow o \approx Max(Pref_\prec(\phi, \sigma)) \tag{6.9}$$

(6.6), (6.7), (6.8), et (6.9) assurent la correction et la transparence de $\mathcal{A}_!$:

(6.6) stipule que la séquence σ est transformée par $\mathcal{A}_\downarrow$ en la séquence o ;

(6.7) assure que si σ satisfaisait déjà la propriété alors elle n'est pas transformée (ou est transformée en une séquence équivalente) ;

(6.8) assure que l'EM produit ne produit rien en sortie (la séquence vide ϵ) lorsqu'il n'y a pas de préfixe correct de σ satisfaisant la propriété ;

(6.9) assure que o est le plus long préfixe de σ satisfaisant la propriété, lorsque celui-ci existe.

La correction est due au fait que la séquence produite o, lorsqu'elle est différente de ϵ, satisfait toujours la propriété ϕ. La transparence est assurée par le fait que les séquences d'exécution correctes ne sont pas modifiées, et les séquences incorrectes sont tronquées en leur plus grand préfixe correct.

Remarque 9 (À propos de la définition de $Max(Pref_{\prec}(\phi,\sigma))$) Nous pouvons remarquer que nous aurions pu définir $Max(Pref_{\prec}(\phi,\sigma))$ comme étant égal à ϵ lorsque $Pref_{\prec}(\phi,\sigma) = \emptyset$ et fusionner les deux dernières contraintes. Cependant, nous choisissons de distinguer explicitement le cas dans lequel $Pref_{\prec}(\phi,\sigma) = \emptyset$ car cela met en évidence les différentes situations où un EM peut produire ϵ. Parfois, cela correspond à la seule séquence correcte de la propriété. Mais parfois cela peut être une séquence incorrecte vis-à-vis de la propriété. En pratique lors de l'implémentation d'un EM, il est facile de marquer cette séquence comme correcte ou incorrecte. *

6.3.3 Instantiation des moniteurs d'enforcement génériques

Dans la suite, nous nous concentrerons sur une forme particulière de moniteur d'enforcement. Cependant, nous verrons que ces moniteurs sont assez expressifs (du point de vue de l'enforcement). Nous allons, en effet, considérer un ensemble restreint d'opérations d'enforcement.

Les opérations d'enforcement permettent aux moniteurs :

- d'arrêter le programme cible (lorsque la séquence d'entrée viole irrémédiablement la propriété) : opération ***halt***,
- de mémoriser l'événement courant dans un *dispositif mémoire* (lorsqu'une décision définitive doit être prise plus tard) : opération ***store***,
- ou de vider le contenu de la mémoire (lorsque le programme est revenu à un comportement correct) : opération ***dump***,
- ou d'éteindre définitivement le moniteur (lorsque la propriété est satisfaite pour toujours) : opération ***off***.

Nous donnons une définition plus précise de ces opérations d'enforcement.

Définition 46 (Opérations d'enforcement $Ops = \{halt, store, dump, off\}$). Dans la suite, nous considérons un ensemble $Ops = \{halt, store, dump, off\}$ défini comme suit : $\forall a \in \Sigma \cup \{\epsilon\}, \forall m \in \Sigma^*$:

$$halt(a,m) = (\epsilon, m) \quad store(a,m) = (\epsilon, m.a) \quad dump(a,m) = (m.a, \epsilon) \quad off(a,m) = (m.a, \epsilon)$$

(a désigne l'événement fourni en entrée au moniteur et m le contenu du dispositif mémoire.)

Notons que la définition de l'opération *off* est la même que celle de l'opération *dump*. D'un point de vue théorique, cette opération n'est effectivement pas nécessaire. En revanche, elle revêt un intérêt pratique. En effet, pour limiter l'impact du moniteur sur le programme (du point de vue des performances), il peut être utile de savoir lorsque le moniteur n'est plus nécessaire. Ainsi la suite des raisonnements pourrait également se faire en considérant uniquement $halt, store, dump$.

De plus, pour que la séquence des opérations d'enforcement réalisée par ces moniteurs soit cohérente, nous supposerons que les moniteurs d'enforcement ne peuvent pas réaliser d'autres opérations que *halt* (resp. *off*) après avoir fait une opération *halt* (resp. *off*).

Dans la suite de cette thèse nous désignons par EM, un EM(Ops) instancié qui respecte ces contraintes. De plus pour un EM $\mathcal{A}_\downarrow$, nous noterons $Halt^{\mathcal{A}_\downarrow}$ l'ensemble des états où l'opération d'enforcement produite est *halt* ($Halt^{\mathcal{A}_\downarrow} = \{q \in Q^{\mathcal{A}_\downarrow} \mid \Gamma^{\mathcal{A}_\downarrow}(q) = halt\}$).

Exemple 19 (Moniteur d'enforcement) Nous illustrons l'enforcement de certaines r-propriétés introduites dans l'Exemple 1 (p. 1) avec des EMs.

- La partie droite de la Fig. 6.3 (p. 112) est un EM $\mathcal{A}_{\downarrow\Pi_1}$ pour la r-propriété de safety Π_1.
  Son état initial est l'état 1. Depuis l'état initial, il effectue simplement l'opération *dump* sur une première occurrence de *g_auth* et se déplace à l'état 3, où l'opération *op* est autorisée et donc produite en sortie (avec l'opération *dump*) et retourne à l'état 1. Autrement, si l'événement *op* précède *g_auth*, alors l'EM se déplace à l'état 2 et arrête définitivement le programme.
- Sur le bas de la Fig. 6.5 (p. 113) est esquissé un EM $\mathcal{A}_{\downarrow\Pi_2}$ pour la r-propriété de guarantee Π_2. L'état initial de $\mathcal{A}_{\downarrow\Pi_2}$ est l'état 1, et il n'a pas d'état où l'EM arrête le programme sous-jacent. Son comportement est le suivant : les occurrences d'événements différents d'un *req_auth* sont mémorisées (opération *store*) tant qu'un *req_auth* n'a pas lieu. Ensuite, le contenu mémoire global est produit en sortie. Enfin, cet EM n'a pas d'état d'arrêt.

Informellement nous pouvons remarquer que les EMs donnés dans cet exemple respectent bien les contraintes de correction et transparence formulées dans le Chapitre 4 Section 4.3. Par exemple le deuxième EM assure bien que la séquence de sortie satisfait toujours la r-propriété considérée et cette sortie est toujours le plus grand préfixe correct de la séquence d'entrée.

6.3.4 Propriétés des moniteurs d'enforcement instanciés

Nous étudions maintenant les propriétés des moniteurs d'enforcement avec l'ensemble d'opérations d'enforcement $\{halt, store, dump, off\}$. Nous donnons une propriété vérifiée par les EMs lorsque ceux-ci lisent des séquences infinies. Puis, pour les séquences finies, il est possible de relier à tout moment la séquence d'entrée fournie à l'EM, son contenu mémoire, et la sortie qu'il produit.

Propriété 9 (À propos de la transformation de séquence infinie) : *Étant donnés une séquence d'exécution $\sigma \in Exec(\mathcal{P}_\Sigma) \cap \Sigma^\omega$ et un EM $\mathcal{A}_\downarrow$, t.q. le run de σ sur $\mathcal{A}_\downarrow$ est exprimé par :*

$$(q_0, \sigma_0/\alpha_0, q_1) \cdot (q_1, \sigma_1/\alpha_1, q_2) \cdots (q_i, \sigma_i/\alpha_i, q_{i+1}) \cdots,$$

nous avons la propriété suivante :

$$\sigma \Downarrow_{\mathcal{A}_\downarrow} \sigma \Leftrightarrow \forall i \in \mathbb{N}, \exists j \in \mathbb{N} : \alpha_j \in \{dump, off\}$$

◇

La propriété stipule que, pour un EM, produire en sortie la même séquence qu'en entrée est équivalent à réaliser régulièrement l'opération d'enforcement *dump* ou *off*.

Pour les séquences finies, il est possible de relier la séquence d'entrée, la mémoire, et la séquence de sortie. Cette relation est exprimée par la relation suivante.

Propriété 10 (Relation entre l'entrée, la mémoire, et la sortie) : *La séquence d'entrée, le contenu mémoire, et la sortie produite sont reliés par la propriété suivante : $\forall \sigma \in \Sigma^+, \forall \sigma' \in \Sigma^*$,*

$$\exists q \in Q^{\mathcal{A}_\downarrow} : (q_{\text{init}}{}^{\mathcal{A}_\downarrow}, \sigma \cdot \sigma', \epsilon) \overset{o}{\Longrightarrow}_{\mathcal{A}_\downarrow} (q, \sigma', m)$$
$$\Longrightarrow$$
$$(\sigma = o \cdot m \wedge q \in Q^{\mathcal{A}_\downarrow} \setminus Halt^{\mathcal{A}_\downarrow}) \vee (o \prec \sigma \wedge q \in Halt^{\mathcal{A}_\downarrow})$$

◇

Informellement, la séquence $\sigma \cdot \sigma'$ (resp. σ, σ') est la séquence fournie en entrée à l'EM (resp. effectivement lue, restant à lire).

PREUVE : Cette preuve est faite par induction sur la longueur de la séquence d'entrée lue σ.

Cas de base. Pour le cas de base $|\sigma| = 1$; nous avons $\sigma = a$ avec $a \in \Sigma$. En utilisant la définition de l'évolution des configurations (Définition 41, p. 101), nous avons que $\exists q \in Q^{\mathcal{A}_\downarrow} : (q_{\text{init}}{}^{\mathcal{A}_\downarrow}, \sigma \cdot \sigma', \epsilon) \overset{o}{\Longrightarrow}_{\mathcal{A}_\downarrow} (q, \sigma', m)$ avec $\alpha(\sigma, \epsilon) = (o, m)$ et $q_{\text{init}}{}^{\mathcal{A}_\downarrow} \overset{\sigma}{\longrightarrow} q \wedge \Gamma(q) = \alpha$. Il y a quatre cas possibles pour α :

- Si $\alpha = halt$, alors $o = \epsilon, m = \epsilon$ et $q \in Halt^{\mathcal{A}_\downarrow}$. Nous avons $o \prec \sigma$.
- Sinon, si $\alpha = store$, alors $o = \epsilon, m = \sigma$. Nous avons $\sigma = o \cdot m$.
- Sinon ($\alpha = dump$ ou $\alpha = off$), $o = a, m = \epsilon$ et $\sigma = o \cdot m$. ■

Énoncé d'induction. Considérons que la propriété est vérifiée pour toute séquence d'exécution de longueur n et considérons une séquence d'exécution $\sigma \cdot a$ de longueur $n+1$, où $a \in \Sigma$. En lisant σ, $\mathcal{A}_{\downarrow}$ entre dans un état $q \in Q^{\mathcal{A}_{\downarrow}}$, produit une sortie o, et contient m dans sa mémoire. Plus formellement, nous avons que $\exists q \in Q^{\mathcal{A}_{\downarrow}} : (q_{\text{init}}{}^{\mathcal{A}_{\downarrow}}, \sigma \cdot a \cdot \sigma', \epsilon) \overset{o}{\Longrightarrow}_{\mathcal{A}_{\downarrow}} (q, a \cdot \sigma', m)$. De plus, l'hypothèse d'induction nous donne : $(q \in Halt^{\mathcal{A}_{\downarrow}} \wedge o \preceq \sigma) \vee (q \notin Halt^{\mathcal{A}_{\downarrow}} \wedge \sigma = o \cdot m)$. Comme $\mathcal{A}_{\downarrow}$ est complet par rapport à $Q^{\mathcal{A}_{\downarrow}} \times \Sigma$ (par définition des EMs), $\exists \alpha \in Ops, \exists q' \in Q^{\mathcal{A}_{\downarrow}} : q \overset{a}{\longrightarrow}_{\mathcal{A}_{\downarrow}} q' \wedge \Gamma(q') = \alpha$. Ainsi, $\exists o', m' \in \Sigma^* : (q, a \cdot \sigma', m) \overset{o'}{\hookrightarrow}_{\mathcal{A}_{\downarrow}} (q', \sigma', m')$ avec $\alpha(a, m) = (o', m')$. Ce qui donne $(q_{\text{init}}{}^{\mathcal{A}_{\downarrow}}, \sigma \cdot a \cdot \sigma', \epsilon) \overset{o \cdot o'}{\Longrightarrow}_{\mathcal{A}_{\downarrow}} (q', \sigma', m')$ et $(q \in Halt^{\mathcal{A}_{\downarrow}} \wedge o \preceq \sigma) \vee (q \notin Halt^{\mathcal{A}_{\downarrow}} \wedge \sigma = o \cdot m)$. Nous voulons montrer que $(q' \in Halt^{\mathcal{A}_{\downarrow}} \wedge o \cdot o' \preceq \sigma \cdot a) \vee (q' \notin Halt^{\mathcal{A}_{\downarrow}} \wedge \sigma \cdot a = o \cdot o' \cdot m')$. Traitons les trois cas pour l'opération d'enforcement α.

- Cas $\alpha = halt$. Nous avons $\alpha(a, m) = (\epsilon, m)$. Ainsi $o' = \epsilon$ et $m = m'$. Et nous avons également $q' \in Halt^{\mathcal{A}_{\downarrow}}$. Ainsi, nous appliquons notre hypothèse d'induction sur σ selon l'appartenance de q à $Halt^{\mathcal{A}_{\downarrow}}$. Si $q \in Halt^{\mathcal{A}_{\downarrow}}$, $o \preceq \sigma \Rightarrow o \cdot \epsilon \preceq \sigma \cdot a$. Sinon ($q \notin Halt^{\mathcal{A}_{\downarrow}}$), nous avons $o \cdot \epsilon \preceq \sigma \cdot a$.
- Cas $\alpha = store$. Nous avons $q \notin Halt^{\mathcal{A}_{\downarrow}}$, et $\alpha(a, m) = (\epsilon, m \cdot a)$, ainsi $o' = \epsilon$ et $m' = m \cdot a$. En appliquant l'hypothèse d'induction, $q' \notin Halt^{\mathcal{A}_{\downarrow}}$ (Définition 39) et $\sigma = o \cdot m$. Donc, nous avons $\sigma \cdot a = o \cdot m \cdot a = o \cdot o' \cdot m'$.
- Cas $\alpha \in \{dump, off\}$. Nous avons $q \notin Halt^{\mathcal{A}_{\downarrow}}$, et $\alpha(a, m) = (m.a, \epsilon)$. Alors $o' = m \cdot a$ et $m' = \epsilon$. En utilisant l'hypothèse d'induction, nous avons nécessairement $q' \notin Halt^{\mathcal{A}_{\downarrow}}$ (Définition 39), et $\sigma = o \cdot m$. Donc, nous obtenons $\sigma \cdot a = o \cdot m \cdot a = o \cdot o' \cdot m'$.

Il s'en suit que la relation d'équivalence $\approx$ considérée précédemment pour l'enforcement (Définition 45) devient la relation d'égalité. Ceci est due à la sémantique des opérations d'enforcement que nous avons considérée. Ainsi le prédicat d'enforcement $Enf_{\approx}(\mathcal{A}_{\downarrow}, (\phi, \varphi), \mathcal{P}_{\Sigma})$ devient $Enf_{=}(\mathcal{A}_{\downarrow}, (\phi, \varphi), \mathcal{P}_{\Sigma})$ (noté $Enf(\mathcal{A}_{\downarrow}, (\phi, \varphi), \mathcal{P}_{\Sigma})$ dans la suite) lorsque la r-propriété est enforcée par $\mathcal{A}_{\downarrow}$ sur $\mathcal{P}_{\Sigma}$. La propriété suivante est une conséquence directe de la Propriété 10 et de la définition des opérations d'enforcement.

Propriété 11 (Dernière opération d'enforcement) : *Étant donnés un moniteur d'enforcement $\mathcal{A}_{\downarrow}$, une r-propriété $\Pi = (\phi, \varphi)$ t.q. $Enf(\mathcal{A}_{\downarrow}, (\phi, \varphi), \mathcal{P}_{\Sigma})$ et une séquence d'exécution finie $\sigma \in Exec(\mathcal{P}_{\Sigma}) \cap \Sigma^+$ ($|\sigma| = n+1$) dont le run sur $\mathcal{A}_{\downarrow}$ est exprimé $(q_0, \sigma_0/\alpha_0, q_1) \cdots (q_n, \sigma_n/\alpha_n, q_{n+1})$, nous avons :*

- $\phi(\sigma) \Rightarrow \alpha_n \in \{dump, off\}$
- $\neg\phi(\sigma) \Rightarrow \alpha_n \in \{store, halt\}$ ◇

Ceci signifie que, considérant un EM qui enforce correctement une r-propriété, la dernière opération d'enforcement réalisée, lors de la lecture d'une séquence d'entrée est dump ou off (resp. halt ou store) lorsque la séquence fournie satisfait (resp. ne satisfait pas) la r-propriété.

Une autre conséquence de ces propriétés est que les sorties produites sont toujours des préfixes des séquences d'entrée, c'est-à-dire : $\forall \sigma, o \in \Sigma^{\infty} : \sigma \Downarrow_{\mathcal{A}_{\downarrow}} o \Rightarrow o \preceq \sigma$.

Ainsi, le modèle de moniteur d'enforcement considéré avec $\{halt, store, dump\}$, muni de l'ensemble des opérations d'enforcement, est un modèle convenant à nos contraintes de transparence et correction [1].

6.4 Operations de composition sur les moniteurs d'enforcement

Le développement actuel des systèmes d'information rend les spécifications de plus en plus complexes. Admettant la valeur des EMs comme des mécanismes potentiels pour augmenter la sûreté des systèmes, il semble souhaitable d'obtenir des techniques pour pouvoir les composer et pouvoir les mettre en correspondance avec leurs spécifications de tailles grandissantes. Dans cette section nous décrivons et nous intéressons au problème de la composition des EMs. Nous donnons la définition de la composition par rapport aux opérateurs booléens : négation, conjonction et disjonction. Nous prouvons également la correction des opération proposées. Notons que les développements suivants sont réalisés pour les moniteurs d'enforcement. Ceux-ci pourraient s'appliquer aux moniteurs de vérification par une adaptation immédiate.

Nous définissons le treillis complet $(Ops, \sqsubseteq)$ sur les opérations d'enforcement, où $halt \sqsubseteq store \sqsubseteq dump \sqsubseteq off$ ($\sqsubseteq$ est un ordre total). De plus, nous définissons une opération de complémentation sur ce treillis : pour

1. Plus précisément un modèle de moniteur d'enforcement avec $\{halt, store, dump\}$ permet de réaliser l'enforcement selon les contraintes de correction et transparence avec une mémoire finie mais non bornée. Un modèle de moniteur d'enforcement avec $\{store, dump\}$ permet de réaliser l'enforcement selon les contraintes de correction et transparence avec une mémoire infinie.

$\alpha \in Ops, \overline{\alpha}$ est l'opération d'enforcement complémentaire de α. Nous définissons $\overline{dump} = store$, $\overline{off} = halt$, et $\overline{\overline{\alpha}} = \alpha$ pour $\alpha \in Ops$.

6.4.1 Négation

Considérant une r-propriété de safety ou de guarantee (enforçable), nous montrons comment un EM peut être transformé de telle manière que l'EM obtenu enforce la négation de la r-propriété enforcée originalement. Remarquons que nous ne pouvons pas définir de négation pour un moniteur d'enforcement pour dédié à une r-propriété de response. En effet, la négation d'une r-propriété de response est une r-propriété de persistence (voir Chapitre 3). Cette transformation s'applique de manière indirecte pour les r-propriétés d'obligation, en utilisant leur décomposition comme des compositions booléennes de r-propriétés de safety et de guarantee.

Définition 47 (Négation d'un EM). Étant donné un EM $\mathcal{A}_{\downarrow\Pi} = (Q^{\mathcal{A}_{\downarrow\Pi}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi}}, \longrightarrow_{\mathcal{A}_{\downarrow\Pi}}, Ops, \Gamma^{\mathcal{A}_{\downarrow\Pi}})$ défini sur l'alphabet Σ et enforcant la r-propriété Π, nous définissons $Negation(\mathcal{A}_{\downarrow\Pi}) = (Q^{\mathcal{A}_{\downarrow\overline{\Pi}}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow\overline{\Pi}}}, \longrightarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}}, Ops, \Gamma^{\mathcal{A}_{\downarrow\overline{\Pi}}})$ comme :

- $Q^{\mathcal{A}_{\downarrow\overline{\Pi}}} = Q^{\mathcal{A}_{\downarrow\Pi}}$, ${q_{\text{init}}}^{\mathcal{A}_{\downarrow\overline{\Pi}}} = {q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi}}$,
- $\rightarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}}$ est définie comme $\rightarrow_{\mathcal{A}_{\downarrow\Pi}}$
- la fonction de sortie $\Gamma^{\mathcal{A}_{\downarrow\overline{\Pi}}}$, produisant les opérations d'enforcement est définie par

$$\forall \alpha \in Ops : \Gamma^{\mathcal{A}_{\downarrow\overline{\Pi}}}(\alpha) = \overline{\Gamma^{\mathcal{A}_{\downarrow\Pi}}(\alpha)}$$

Théorème λ (Correction de l'opération de négation d'un EM) : *Étant donné un* $(Q^{\mathcal{A}_{\downarrow\Pi}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi}}, \longrightarrow_{\mathcal{A}_{\downarrow\Pi}}, Ops, \Gamma^{\mathcal{A}_{\downarrow\Pi}})$ *défini relativement à un langage d'entrée* Σ *enforcant une* r-*propriété* Π *de safety ou de guarantee, l'EM obtenu en utilisant la transformation* Negation *enforce* $\overline{\Pi}$. *Plus formellement :* $\forall \Pi \subseteq \Sigma^* \times \Sigma^\omega$

$$Enf(\mathcal{A}_{\downarrow\Pi}, \Pi, \mathcal{P}_\Sigma) \Rightarrow Enf(\text{Negation}(\mathcal{A}_{\downarrow\Pi}), \overline{\Pi}, \mathcal{P}_\Sigma)$$

Preuve : Soit $\Pi = (\phi, \varphi)$ la r-propriété, avec $\phi \subseteq \Sigma^*$ et $\varphi \subseteq \Sigma^\omega$. Notons $\mathcal{A}_{\downarrow\overline{\Pi}} = \text{Negation}(\mathcal{A}_{\downarrow\Pi})$, et $\Rightarrow$ la relation de dérivation multipas définie sur les configurations de $\mathcal{A}_{\downarrow\overline{\Pi}}$ et $\longrightarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}}$. Aussi, comme $Q^{\mathcal{A}_{\downarrow\Pi}} = Q^{\mathcal{A}_{\downarrow\overline{\Pi}}}$, nous utiliserons Q pour noter les états des deux EMs. Similairement, q_{init} dénote les états de départ des deux EMs. Nous devons montrer que $Enf(\mathcal{A}_{\downarrow\overline{\Pi}}, \overline{\Pi}, \mathcal{P}_\Sigma)$, ce qui signifie que, pour toute séquence $\sigma \in Exec(\mathcal{P}_\Sigma)$, nous devons montrer que $\exists o \in \Sigma^\infty$ *t.q.* :

$$\sigma \Downarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}} o \tag{6.10}$$

$$\overline{\Pi}(\sigma) \Rightarrow \sigma = o \tag{6.11}$$

$$\neg\overline{\Pi}(\sigma) \wedge Pref_{\prec}(\overline{\phi}, \sigma) = \emptyset \Rightarrow o = \epsilon \tag{6.12}$$

$$\neg\overline{\Pi}(\sigma) \wedge Pref_{\prec}(\overline{\phi}, \sigma) \neq \emptyset \Rightarrow o = Max(Pref_{\prec}(\overline{\phi}, \sigma)) \tag{6.13}$$

La preuve se réalise en deux étapes. La première est pour les séquences finies, la seconde pour les séquences infinies.

Séquences finies. La preuve pour les séquences finies se fait par une induction sur $|\sigma|$.

Cas de base. Pour le cas de base $|\sigma| = 0$; nous avons $\sigma = \epsilon$, ensuite nous avons facilement (6.10) car $\epsilon \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$. De plus, $Pref_{\prec}(\phi, \epsilon) = \emptyset$, ce qui donne (6.12).

Énoncé d'induction. Soit $n \in \mathbb{N}$, supposons que pour toutes les séquences σ *t.q.* $|\sigma| = n$, nous avons l'existence d'une sortie $o \in \Sigma^*$ *t.q.* les contraintes (6.10), (6.11), (6.12) et (6.13) soit vérifiées. Maintenant considérons $a \in \Sigma$, et une séquence $\sigma \cdot a$ *t.q.* $|\sigma \cdot a| = n + 1$, nous étudions l'effet de la soumission en entrée du dernier événement a. Nous voulons prouver qu'il existe une nouvelle sortie *t.q.* les mêmes contraintes soient vérifiées.

Comme $\sigma \Downarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}} o$ (hypothèse d'induction), il existe une configuration $(q, \epsilon, m) \in Q \times \Sigma^* \times \Sigma^*$ *t.q.* $(q_{\text{init}}, \sigma, \epsilon) \overset{o}{\Rightarrow} (q, \epsilon, m)$. Ce qui implique que $(q_{\text{init}}, \sigma \cdot a, \epsilon) \overset{o}{\Rightarrow} (q, a, m)$. C'est-à-dire que, après avoir lu σ, $\mathcal{A}_{\downarrow\overline{\Pi}}$ est dans un état q avec a en entrée, et m comme contenu mémoire. Ensuite depuis la configuration (q, a, m), il évolue vers une configuration (q', ϵ, m'), *i.e.*, $(q, a, m) \overset{o'}{\hookrightarrow} (q', \epsilon, m')$ avec $\alpha(a, m) = (o', m'), \alpha \in Ops$. La lecture de $\sigma \cdot a$ sur $\mathcal{A}_{\downarrow\Pi}$, induit une évolution similaire des configurations :

$$(q_{\text{init}}, \sigma \cdot a, \epsilon) \overset{o}{\Rightarrow} (q, a, m) \overset{o'}{\hookrightarrow} (q', \epsilon, m')$$
$$(q_{\text{init}}, \sigma \cdot a, \epsilon) \overset{p}{\Rightarrow}_{\mathcal{A}_{\downarrow\Pi}} (q, a, n) \overset{p'}{\hookrightarrow}_{\mathcal{A}_{\downarrow\Pi}} (q', \epsilon, n'),$$

avec :
- $q \xrightarrow{a} q', \Gamma^{\mathcal{A}_{\downarrow\overline{\Pi}}}(q') = \alpha, \alpha(a, m) = (o', m'), \alpha \in Ops; q, q' \in Q; m, m', o, o' \in \Sigma^*$
- $q \xrightarrow{a}_{\mathcal{A}_{\downarrow\Pi}} q', \Gamma^{\mathcal{A}_{\downarrow\Pi}}(q') = \alpha', \alpha'(a, n) = (p', n'), \alpha' \in Ops; q, q' \in Q; n, n', p, p' \in \Sigma^*$

Il y a deux cas selon que $\phi(\sigma \cdot a)$:
- Le premier cas est $\phi(\sigma \cdot a)$. Comme $Enf(\Pi, \mathcal{A}_{\downarrow\Pi}, \mathcal{P}_\Sigma)$, $\mathcal{A}_{\downarrow\Pi}$ produit $\sigma \cdot a$, c'est-à-dire $\sigma \cdot a \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma \cdot a$. Nécessairement, $\alpha' = dump$. Cela correspond à une opération $\alpha = store$ sur $\mathcal{A}_{\downarrow\overline{\Pi}}$. Maintenant nous distinguons selon que $\phi(\sigma)$ ou non.
  - Si $\phi(\sigma)$, en utilisant l'hypothèse d'induction ($|\sigma| = n$), nous avons soit $o = \epsilon$ ($Pref_{\prec}(\overline{\phi}, \sigma) = \emptyset$) ou $o = Max(Pref_{\prec}(\overline{\phi}, \sigma))$ ($Pref_{\prec}(\overline{\phi}, \sigma) \neq \emptyset$).
    Si $Pref_{\prec}(\overline{\phi}, \sigma) = \emptyset$, alors nous avons également $Pref_{\prec}(\overline{\phi}, \sigma \cdot a) = \emptyset$. La sortie de $\mathcal{A}_{\downarrow\overline{\Pi}}$ est toujours ϵ, *i.e.*, $o \cdot o' = \epsilon$ (6.12).
    Si $Pref_{\prec}(\overline{\phi}, \sigma) \neq \emptyset$, en utilisant l'hypothèse d'induction, $o = Max(Pref_{\prec}(\overline{\phi}, \sigma))$. Aussi $\phi(\sigma \cdot a)$, ce qui implique que $o = Max(Pref_{\prec}(\overline{\phi}, \sigma \cdot a))$ (6.13).
  - Si $\neg\phi(\sigma)$, *i.e.*, $\overline{\phi}(\sigma)$, en utilisant l'hypothèse d'induction, nous avons que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}} o$ avec $\sigma = o$. Ensuite $\sigma = Max(Pref_{\prec}(\overline{\phi}, \sigma))$ car $\overline{\phi}(\sigma)$. Nous obtenons également (6.13).
- Le deuxième cas est $\overline{\phi}(\sigma \cdot a)$. Nous avons soit $Pref_{\prec}(\phi, \sigma \cdot a) \neq \emptyset$ soit $Pref_{\prec}(\phi, \sigma \cdot a) = \emptyset$.
  - Si $Pref_{\prec}(\phi, \sigma \cdot a) \neq \emptyset$, alors comme $Enf(\Pi, \mathcal{A}_{\downarrow\Pi}, \mathcal{P}_\Sigma)$, nous avons $p = Max(Pref_{\prec}(\phi, \sigma \cdot a))$. Nous savons que $p \prec \sigma \cdot a$ (car par hypothèse $\overline{\phi}(\sigma \cdot a)$). Il s'en suit que $\alpha' \in \{store, halt\}$. En conséquence $\alpha = dump$ et $\sigma \cdot a \Downarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}} \sigma \cdot a$, et par hypothèse, nous avons $\overline{\phi}(\sigma \cdot a)$. Nous avons (6.10) et (6.11).
  - Si $Pref_{\prec}(\phi, \sigma \cdot a) = \emptyset$. Nécessairement, $\alpha' \in \{store, halt\}$ et $\alpha \in \{dump, off\}$. Donc $\sigma \cdot a \Downarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}} \sigma \cdot a$.

Séquences infinies. Nous traitons maintenant la preuve pour les séquences infinies $\sigma \in \Sigma^\omega$. Dans la suite, nous distinguons selon la classe de la r-propriété Π. Considérons $\sigma \in \Sigma^\omega$.
- *Si* Π *est une r-propriété de safety.* Nous avons deux cas, selon que $\varphi(\sigma)$ ou non.
  - $\varphi(\sigma)$. Comme $Enf(\Pi, \mathcal{A}_{\downarrow\Pi}, \mathcal{P}_\Sigma)$, nous avons que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma$. De plus comme Π est une r-propriété de safety, tous les préfixes de σ satisfont ϕ, c'est-à-dire $\forall \sigma' \prec \sigma : \phi(\sigma')$, et en conséquence $\sigma' \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma'$. Il s'en suit (Propriété. 11) que la séquence des opération d'enforcement produite sur $\mathcal{A}_{\downarrow\Pi}$ est de la forme $dump^\omega + dump^* \cdot off^\omega$. Alors en utilisant la définitions de la transformation Negation, nous trouvons que la séquence des opérations d'enforcement sur $\mathcal{A}_{\downarrow\overline{\Pi}}$ est de la forme $store^\omega + store^* \cdot halt^\omega$. Il s'en suit que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}} \epsilon$. (6.10). Comme $Pref_{\prec}(\overline{\phi}, \sigma) = \emptyset$, nous obtenons (6.12).
  - $\overline{\varphi(\sigma)}$. Comme $Enf(\Pi, \mathcal{A}_{\downarrow\Pi}, \mathcal{P}_\Sigma)$, nous avons que soit $Pref_{\prec}(\phi, \sigma) = \emptyset \wedge o = \epsilon$ ou $\exists o \in \Sigma^* : o = Max(Pref_{\prec}(\phi, \sigma))$.
    - Traitons d'abord le cas où $Pref_{\prec}(\phi, \sigma) = \emptyset$. Dans ce cas, nous avons $\forall \sigma' \in \Sigma^* : \sigma' \prec \sigma \Rightarrow \neg\phi(\sigma')$. Il s'en suit que la séquences des opérations d'enforcement produite sur $\mathcal{A}_{\downarrow\Pi}$ est $store^\omega + store^* \cdot halt^\omega$. En utilisant la définition de Negation, la séquences des opérations d'enforcement sur $\mathcal{A}_{\downarrow\overline{\Pi}}$ est de la forme $dump^\omega + dump^* \cdot halt^\omega$. Il s'en suit que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}} \sigma$. Nous obtenons (6.10).
    - Maintenant $Pref_{\prec}(\phi, \sigma) \neq \emptyset$. Soit $n = |o|$. Comme Π est une r-propriété de safety, nous avons que $\forall i \leq n : \phi(\sigma_{\dots i-1}) \wedge \forall i > n : \neg\phi(\sigma_{\dots i})$. Ensuite, en utilisant la Propriété 11 (p. 105), nous pouvons trouver la séquence des opérations d'enforcement réalisée par $\mathcal{A}_{\downarrow\Pi}$: $dump^n \cdot \left(store^\omega + store^* \cdot halt^\omega\right)$. Sur $\mathcal{A}_{\downarrow\overline{\Pi}}$, en utilisant la définition de la transformation Negation, la séquence des opérations d'enforcement devient $store^n \cdot \left(dump^\omega + dump^* \cdot off^\omega\right)$. Il s'en suit que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}} \sigma$ (6.10). Ensuite, $\overline{\varphi(\sigma)}$ et $\sigma = \sigma$ assure (6.11).
- *Si* Π *est une r-propriété de guarantee.* Nous avons deux cas, selon que $\varphi(\sigma)$ ou pas.
  - $\varphi(\sigma)$. Comme $Enf(\Pi, \mathcal{A}_{\downarrow\Pi}, \mathcal{P}_\Sigma)$, nous avons que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma$. De plus Π est une r-propriété de guarantee, il existe un préfixe σ' de σ *t.q.* $\forall \sigma'' \in \Sigma^* : \sigma' \preceq \sigma'' \Rightarrow \phi(\sigma'') \wedge \forall \sigma'' \in \Sigma^* : \sigma'' \prec$

$\sigma' \Rightarrow \neg\phi(\sigma'')$. Notons $n = |\sigma'|$. En conséquence, comme Π est enforcée par $\mathcal{A}_{\downarrow\Pi}$, nous avons $\forall\sigma'' \in \Sigma^* : \sigma'' \geq \sigma' \Rightarrow \sigma'' \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma'' \wedge \forall\sigma'' \prec \sigma' : \sigma'' \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$. Il s'en suit que la séquence des opérations d'enforcement sur $\mathcal{A}_{\downarrow\Pi}$ est $store^n \cdot off^\omega$. Ensuite, en utilisant la définition de la transformation **Negation**, nous trouvons que la séquence des opérations d'enforcement sur $\mathcal{A}_{\downarrow\overline{\Pi}}$ est $dump^n \cdot halt^\omega$. Il s'en suit que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}} \sigma'$ (6.10). De plus, nous avons vu que $\phi(\sigma')$, nous avons (6.13).

– $\neg\varphi(\sigma)$. Sachant que Π est une r-propriété de guarantee, $\neg\varphi(\sigma)$ implique qu'il n'y a pas de préfixe de σ satisfaisant ϕ. Comme $Enf(\Pi, \mathcal{A}_{\downarrow\Pi}, \mathcal{P}_\Sigma)$, nous avons que $\forall\sigma' \prec \sigma \cdot \sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$. La séquence des opérations d'enforcement réalisée par $\mathcal{A}_{\downarrow\Pi}$ est de la forme $store^* \cdot halt^\omega + (halt)^\omega$. En utilisant la définition de la transformation **Negation**, la séquence des opérations d'enforcement sur $\mathcal{A}_{\downarrow\overline{\Pi}}$ est de la forme $dump^\omega + dump^* \cdot off^\omega$. Il s'en suit que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\overline{\Pi}}} \sigma$. Nous avons (6.10) et (6.11). ■

6.4.2 Union et intersection

Nous montrons comment la disjonction (resp. conjonction) de propriétés enforçables peut être enforcée en construisant l'union (resp. l'intersection) de leur moniteurs d'enforcement associés. Ces opérations entre les EMs sont basées sur la construction de produits entre les EMs et la combinaison des opérations d'enforcement par rapport au treillis complet $(Ops, \sqsubseteq)$.

Définition 48 (Union de EMs). Étant donnés deux EMs, $(Q^{\mathcal{A}_{\downarrow 1}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow 1}}, \longrightarrow_{\mathcal{A}_{\downarrow 1}}, Ops, \Gamma^{\mathcal{A}_{\downarrow 1}})$, $(Q^{\mathcal{A}_{\downarrow 2}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow 2}}, \longrightarrow_{\mathcal{A}_{\downarrow 2}}, Ops, \Gamma^{\mathcal{A}_{\downarrow 2}})$ définis relativement à un même langage d'entrée Σ, nous définissons $\mathcal{A}_{\downarrow\sqcup} = Union(\mathcal{A}_{\downarrow 1}, \mathcal{A}_{\downarrow 2})$ avec $Q^{\mathcal{A}_{\downarrow\sqcup}} = (Q^{\mathcal{A}_{\downarrow 1}} \times Q^{\mathcal{A}_{\downarrow 2}})$, ${q_{\text{init}}}^{\mathcal{A}_{\downarrow\sqcup}} = ({q_{\text{init}}}^{\mathcal{A}_{\downarrow 1}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow 2}})$ et $Halt^{\mathcal{A}_{\downarrow\sqcup}} = Halt^{\mathcal{A}_{\downarrow 1}} \times Halt^{\mathcal{A}_{\downarrow 2}}$. La relation de transition de ce moniteur d'enforcement est définie en prenant la borne supérieure ($\sqcup$) des opérations d'enforcement. Plus formellement $\rightarrow_{\mathcal{A}_{\downarrow\sqcup}}$: $Q^{\mathcal{A}_{\downarrow\sqcup}} \times \Sigma \times Ops \rightarrow Q^{\mathcal{A}_{\downarrow\sqcup}}$ est défini comme $\forall a \in \Sigma, \forall q = (q_1, q_2) \in Q^{\mathcal{A}_{\downarrow\sqcup}}$,

$$\frac{q_1 \xrightarrow{a}_{\mathcal{A}_{\downarrow 1}} q_1{}' \qquad q_2 \xrightarrow{a}_{\mathcal{A}_{\downarrow 2}} q_2{}'}{(q_1, q_2) \xrightarrow{a}_{\mathcal{A}_{\downarrow\sqcup}} (q_1{}', q_2{}')} \mathcal{A}_{\downarrow\sqcup}$$

$\Gamma^{\mathcal{A}_{\downarrow\sqcup}}((q_1, q_2)) = \sqcup\left\{\Gamma^{\mathcal{A}_{\downarrow 1}}(q_1), \Gamma^{\mathcal{A}_{\downarrow 2}}(q_2)\right\}$

Notons que cette construction n'introduit pas de non-déterminisme dans l'EM résultant. En effet, comme les deux EMs initiaux sont déterministes, il y a toujours une et une seule transition avec un élement de Σ dans l'EM résultant.

Notons que $\mathcal{A}_\sqcup$ peut être défini également de manière équivalente en utilisant des versions $\perp$-completée de $\mathcal{A}_1$ et $\mathcal{A}_2$ avec la construction prédemment définie de l'opération union lorsque $\mathcal{A}_1$ et $\mathcal{A}_2$ ne sont pas définis sur des vocabulaires identiques. Ceci est du au fait que $\perp$ est idem-potente pour $\sqcup$, et ainsi lorsque l'on compose une opération $op \in Ops$ avec $\perp$ en utilisant $\sqcup$, le résultat est op.

Exemple 20 (Union de EMs) Sur la droite de la Fig. 6.1 est représenté l'EM union $\mathcal{A}_{\downarrow\sqcup}$ construit depuis les EMs $\mathcal{A}_{\downarrow e_1}, \mathcal{A}_{\downarrow e_2}$. Suivant la définition de la construction, l'ensemble d'états est le produit cartésien $Q^{\mathcal{A}_{\downarrow 1}} \times Q^{\mathcal{A}_{\downarrow 2}}$ dans lequel l'état (1,2) n'est pas atteignable. L'état initial est (1,1). Notons qu'il n'y a pas d'état *Halt* dans l'EM résultant car $Halt^{\mathcal{A}_{\downarrow e_1}} \times Halt^{\mathcal{A}_{\downarrow e_2}} = \emptyset$.

L'opération d'intersection entre les moniteurs d'enforcement est définie de manière similaire en utilisant l'opération borne inférieure $\sqcap$ entre les opérations d'enforcement :

Définition 49 (Intersection d'EMs). Étant donnés deux EMs $(Q^{\mathcal{A}_{\downarrow 1}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow 1}}, \longrightarrow_{\mathcal{A}_{\downarrow 1}}, Ops, \Gamma^{\mathcal{A}_{\downarrow 1}})$, $(Q^{\mathcal{A}_{\downarrow 2}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow 2}}, \longrightarrow_{\mathcal{A}_{\downarrow 2}}, Ops, \Gamma^{\mathcal{A}_{\downarrow 2}})$ définis relativement à un même langage d'entrée Σ et un ensemble d'opérations d'enforcement Ops, nous définissons l'EM intersection $(Q^{\mathcal{A}_{\downarrow\sqcap}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow\sqcap}}, \longrightarrow_{\mathcal{A}_{\downarrow\sqcap}}, Ops, \Gamma^{\mathcal{A}_{\downarrow\sqcap}})$ avec $Q^{\mathcal{A}_{\downarrow\sqcap}} = (Q^{\mathcal{A}_{\downarrow 1}} \times Q^{\mathcal{A}_{\downarrow 2}})$, ${q_{\text{init}}}^{\mathcal{A}_{\downarrow\sqcap}} = ({q_{\text{init}}}^{\mathcal{A}_{\downarrow 1}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow 2}})$ et $Halt^{\mathcal{A}_{\downarrow\sqcap}} = Halt^{\mathcal{A}_{\downarrow 1}} \times Q^{\mathcal{A}_{\downarrow 2}} \cup Q^{\mathcal{A}_{\downarrow 1}} \times Halt^{\mathcal{A}_{\downarrow 2}}$. La relation de transition de ce moniteur d'enforcement est définie en prennant la borne inférieure ($\sqcap$) des opérations d'enforcement. Plus formellement $\rightarrow_{\mathcal{A}_{\downarrow\sqcap}}$: $Q^{\mathcal{A}_{\downarrow\sqcap}} \times \Sigma \times Ops \rightarrow Q^{\mathcal{A}_{\downarrow\sqcap}}$ est définie par $\forall a \in \Sigma, \forall q = (q_1, q_2) \in Q^{\mathcal{A}_{\downarrow\sqcap}}$,

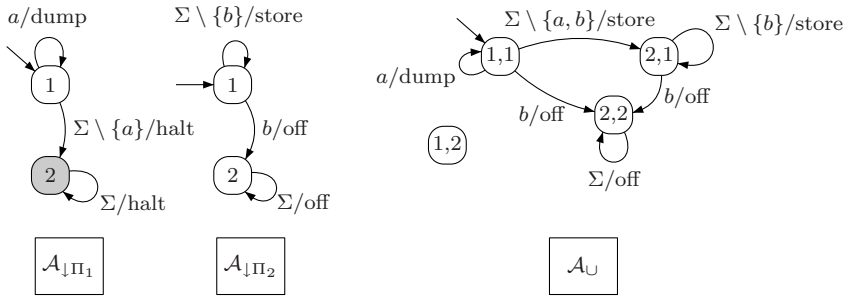


FIGURE 6.1 – Union de deux moniteurs d'enforcement : $\mathcal{A}_{\downarrow e_1}$ et $\mathcal{A}_{\downarrow e_2}$

$$\frac{q_1 \xrightarrow{a}_{\mathcal{A}_{\downarrow 1}} q_1{}' \qquad q_2 \xrightarrow{a}_{\mathcal{A}_{\downarrow 2}} q_2{}'}{(q_1, q_2) \xrightarrow{a}_{\mathcal{A}_{\downarrow\sqcap}} (q_1{}', q_2{}')} \mathcal{A}_{\downarrow\sqcap}$$

La fonction de sortie des opérations d'enforcement $\Gamma^{\mathcal{A}_{\downarrow\sqcap}}$ est définie par :

$$\forall (q_1, q_2) \in Q^{\mathcal{A}_{\downarrow\sqcap}} : \Gamma^{\mathcal{A}_{\downarrow\sqcap}}((q_1, q_2)) = \sqcap\left\{\Gamma^{\mathcal{A}_{\downarrow 1}}(q_1), \Gamma^{\mathcal{A}_{\downarrow 2}}(q_2)\right\}$$

Notons que entre la transformation réalisant l'union et celle réalisant l'intersection, seule la manière de composer la fonction de sortie change.

Théorème μ (Correction des opérations d'union et intersection d'EMs) : *Étant donnés* $(Q^{\mathcal{A}_{\downarrow\Pi_1}}, q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi_1}}, \longrightarrow_{\mathcal{A}_{\downarrow\Pi_1}}, Ops, \Gamma^{\mathcal{A}_{\downarrow\Pi_1}})$ *et* $(Q^{\mathcal{A}_{\downarrow\Pi_2}}, q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi_2}}, \longrightarrow_{\mathcal{A}_{\downarrow\Pi_2}}, Ops, \Gamma^{\mathcal{A}_{\downarrow\Pi_2}})$*, deux moniteurs d'enforcement enforcant deux propriétés* $\Pi_1, \Pi_2 \in EP$ *sur un programme* $\mathcal{P}_\Sigma$*, la r-propriété* $\Pi_1 \vee \Pi_2$ *(resp.* $\Pi_1 \wedge \Pi_2$*) est enforcée par le moniteur d'enforcement union (resp. intersection). Plus formellement :* $\forall \Pi_1, \Pi_2 \subseteq \Sigma^* \times \Sigma^\omega$,

$$Enf(\mathcal{A}_{\downarrow\Pi_1}, \Pi_1, \mathcal{P}_\Sigma) \wedge Enf(\mathcal{A}_{\downarrow\Pi_2}, \Pi_2, \mathcal{P}_\Sigma) \Rightarrow Enf(Union(\mathcal{A}_{\downarrow\Pi_1}, \mathcal{A}_{\downarrow\Pi_2}), \Pi_1 \vee \Pi_2, \mathcal{P}_\Sigma)$$
$$Enf(\mathcal{A}_{\downarrow\Pi_1}, \Pi_1, \mathcal{P}_\Sigma) \wedge Enf(\mathcal{A}_{\downarrow\Pi_2}, \Pi_2, \mathcal{P}_\Sigma) \Rightarrow Enf(Intersection(\mathcal{A}_{\downarrow\Pi_1}, \mathcal{A}_{\downarrow\Pi_2}), \Pi_1 \wedge \Pi_2, \mathcal{P}_\Sigma)$$

PREUVE : Nous traitons la preuve pour l'opérateur *Union*. Pour $i \in \{1, 2\}$, nous avons $Enf(\mathcal{A}_{\downarrow\Pi_i}, \Pi_i, \mathcal{P}_\Sigma)$, *i.e.*, pour tout $\sigma \in Exec(\mathcal{P}_\Sigma)$, il existe $o_i \in \Sigma^*$:

$$\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi_i}} o_i \tag{6.14}$$
$$\Pi_i(\sigma) \Rightarrow \sigma = o_i \tag{6.15}$$
$$\neg\Pi_i(\sigma) \wedge Pref_{\prec}(\phi_i, \sigma) = \emptyset \Rightarrow o_i = \epsilon \tag{6.16}$$
$$\neg\Pi_i(\sigma) \wedge Pref_{\prec}(\phi_i, \sigma) \neq \emptyset \Rightarrow o_i = Max(Pref_{\prec}(\phi_i, \sigma)) \tag{6.17}$$

Notons $\mathcal{A}_{\sqcup} = Union(\mathcal{A}_{\downarrow\varphi_1}, \mathcal{A}_{\downarrow\varphi_2})(Q, q_{\text{init}}, \longrightarrow, Ops, \Gamma)$, $\varphi = \varphi_1 \vee \varphi_2$, et $\Rightarrow$ la relation de dérivation multi-pas définie sur les configurations de $\mathcal{A}_{\sqcup}$ et $\longrightarrow$. Nous devons montrer $Enf(\mathcal{A}_{\sqcup}, \varphi, \mathcal{P}_\Sigma)$, ce qui signifie que, étant donné $\sigma \in Exec(\mathcal{P}_\Sigma)$, nous devons prouver l'existence de $o \in \Sigma^*$ *t.q.* :

$$\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} o \tag{6.18}$$
$$\Pi(\sigma) \Rightarrow \sigma = o \tag{6.19}$$
$$\neg\Pi(\sigma) \wedge Pref_{\prec}(\phi, \sigma) = \emptyset \Rightarrow o = \epsilon \tag{6.20}$$
$$\neg\Pi(\sigma) \wedge Pref_{\prec}(\phi, \sigma) \neq \emptyset \Rightarrow o = Max(Pref_{\prec}(\phi, \sigma)) \tag{6.21}$$

Nous considérons tout d'abord $\sigma \in \Sigma^*$, la preuve suivante est faite par récurrence sur $|\sigma|$.

Cas de base. Pour le cas de base $|\sigma| = 0$; nous avons $\sigma = \epsilon$. Ensuite nous avons facilement (6.18) car $\epsilon \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$. De plus, $Pref_{\prec}(\phi, \epsilon) = \emptyset$, ce qui donne (6.20).

Énoncé d'induction. Soit $n \in \mathbb{N}$ et supposons que pour toutes les séquences σ *t.q.* $|\sigma| = n$, nous avons l'existence d'une sortie o de $\mathcal{A}_{\sqcup}$ *t.q.* (6.19) et (6.20).

Comme $\sigma \Downarrow_{\mathcal{A}_{\sqcup}} o$ (hypothèse d'induction), il existe une configuration $(q, \epsilon, m) \in Q \times \Sigma^* \times \Sigma^*$ *t.q.* $(q_{\text{init}}, \sigma, \epsilon) \overset{o}{\Rightarrow} (q, \epsilon, m)$. Ce qui implique $(q_{\text{init}}, \sigma \cdot a, \epsilon) \overset{o}{\Rightarrow} (q, a, m)$. C'est-à-dire, après avoir lu σ, l'EM $\mathcal{A}_{\sqcup}$ est dans un état q avec a en entrée, et m comme contenu mémoire. Ensuite depuis la configuration (q, a, m), il évolue vers une configuration (q', ϵ, m'), c'est-à-dire $(q, a, m) \overset{o'}{\hookrightarrow} (q', \epsilon, m')$ avec $\alpha(a, m) = (o', m'), \alpha \in Ops$. En lisant $\sigma \cdot a$, $\mathcal{A}_{\sqcup}$ produit une sortie $o \cdot o'$. Aussi, la lecture de $\sigma \cdot a$ sur $\mathcal{A}_{\downarrow\Pi_i}$, $i \in \{1, 2\}$, induit l'évolution de configurations suivante :

$$(q_{\text{init}}, \sigma \cdot a, \epsilon) \overset{o_i}{\Rightarrow} (q_i, a, m_i) \overset{o'_i}{\hookrightarrow} (q'_i, \epsilon, m'_i),$$

avec $\alpha_i(a, m_i) = (o'_i, m'_i); q_i, q'_i \in Q^{\mathcal{A}_{\downarrow\Pi_i}}; m_i, m'_i, o_i, o'_i \in \Sigma^*$.

Il y a deux cas selon que $\phi(\sigma \cdot a)$ ou non.

- Le premier cas est $\phi(\sigma \cdot a)$. Dans ce cas, cela signifie que soit $\phi_1(\sigma \cdot a)$ ou $\phi_2(\sigma \cdot a)$. Traitons le cas $\phi_1(\sigma \cdot a)$, le cas $\phi_2(\sigma \cdot a)$ est identique. Comme $Enf(\Pi_1, \mathcal{A}_{\downarrow\Pi_1}, \mathcal{P}_\Sigma)$, nous avons que $\exists o_1 \in \Sigma^* : \sigma \cdot a \Downarrow_{\mathcal{A}_{\downarrow\Pi_1}} o_1$. De plus, $\phi_1(\sigma \cdot a)$ implique que $o_1 = \sigma \cdot a$. Inévitablement la dernière opération d'enforcement de $\mathcal{A}_{\downarrow\Pi_1}$ est *dump* ou *off*, *i.e.*, $\alpha_1 = \{dump, off\}$ (Propriété 11, p. 105). Il s'en suit que $\alpha = \sqcup(\{\alpha_1, \alpha_2\}) \in \{dump, off\}$. Selon la définition des opérations d'enforcement et la Propriété 10, nous obtenons que $\sigma \cdot a \Downarrow_{\mathcal{A}_{\sqcup}} \sigma \cdot a$, *i.e.*, (6.18) et (6.19).
- Le deuxième cas est $\neg\phi(\sigma \cdot a)$. Ceci implique que $\neg\phi_1(\sigma \cdot a) \wedge \neg\phi_2(\sigma \cdot a)$. En utilisant la définition des opérations d'enforcement, nous avons quatre cas selon que $Pref_{\prec}(\phi_i, \sigma \cdot a) = \emptyset, i \in \{1, 2\}$.
  - Le premier cas est $Pref_{\prec}(\phi_i, \sigma \cdot a) \neq \emptyset, i \in \{1, 2\}$. Pour $i \in \{1, 2\}$, comme $Enf(\Pi_i, \mathcal{A}_{\downarrow\Pi_i}, \mathcal{P}_\Sigma)$, $\neg\phi_i(\sigma \cdot a)$ nous donne $\exists o_i \in \Sigma^* : o_i = Max(Pref_{\prec}(\phi_i, \sigma \cdot a))$. Maintenant, nous avons soit $o_1 \prec o_2$, $o_2 \prec o_1$ soit $o_1 = o_2$.
    - Considérons le cas $o_1 \prec o_2$ ($o_2 \prec o_1$ est identique). Dans ce cas, nous avons $\forall o'_1 \in \Sigma^* : o_1 \prec o'_1 \preceq \sigma \cdot a \Rightarrow \neg\phi_1(o'_1)$, et $\forall o'_2 \in \Sigma^* : o_2 \prec o'_2 \preceq \sigma \cdot a \Rightarrow \neg\phi_2(o'_2)$. Ensuite $o_1 \prec o_2$ implique que $o_2 = Max(Pref_{\prec}(\phi, \sigma \cdot a))$. Nous devons montrer que $\sigma \cdot a \Downarrow_{\mathcal{A}_{\sqcup}} o_2$. Examinons la séquence des opérations d'enforcement réalisée par $\mathcal{A}_{\sqcup}$. Nous avons que $o_2 \Downarrow_{\mathcal{A}_{\sqcup}} o_2$, car la dernière opération d'enforcement réalisée en lisant $o_2 \preceq \sigma \cdot a$ est un *dump* ou un *off* ($\mathcal{A}_{\sqcup}$ est obtenu en prenant la borne supérieure des opérations d'enforcement).
    - Similairement au cas $o_1 = o_2$, nous avons que $o_1 = o_2 = Max(Pref_{\prec}(\phi, \sigma \cdot a))$. Le raisonnement précédent est vérifié.
  - Le deuxième cas est $Pref_{\prec}(\phi_i, \sigma \cdot a) = \emptyset, i \in \{1, 2\}$. Pour $i \in \{1, 2\}$, comme $Enf(\Pi_i, \mathcal{A}_{\downarrow\Pi_i}, \mathcal{P}_\Sigma)$, $\neg\phi_i(\sigma \cdot a)$ cela nous donne $\sigma \cdot a \Downarrow_{\mathcal{A}_{\downarrow\Pi_i}}, i \in \{1, 2\}$.
  - Le troisème cas est $Pref_{\prec}(\phi_1, \sigma \cdot a) = \emptyset \vee Pref_{\prec}(\phi_2, \sigma \cdot a) = \emptyset$. Supposons $Pref_{\prec}(\phi_1, \sigma \cdot a) = \emptyset$, le cas $Pref_{\prec}(\phi_2, \sigma \cdot a) = \emptyset$ est identique. Comme $Enf(\Pi_1, \mathcal{A}_{\downarrow\Pi_1}, \mathcal{P}_\Sigma)$, nous avons que $\sigma \cdot a \Downarrow_{\mathcal{A}_{\downarrow\Pi_1}} \epsilon$. Il s'en suit que la séquence des opérations d'enforcement sur $\mathcal{A}_{\downarrow\Pi_1}$ est $(store + halt)^*$. Sur $\mathcal{A}_{\downarrow\Pi_2}$, comme $Enf(\Pi_2, \mathcal{A}_{\downarrow\Pi_2}, \mathcal{P}_\Sigma)$, nous avons que $\exists o_1 \in \Sigma^* : o_1 = Max(Pref_{\prec}(\phi_1, \sigma \cdot a))$. Il s'en suit que la séquence des opérations d'enforcement réalisée par $\mathcal{A}_{\downarrow\Pi_2}$ est $(store + dump)^{k-1} \cdot dump \cdot (halt + store)^{n-k}$ où $k = |o_2|$. Ensuite la définition suivante de la construction Union, $\mathcal{A}_{\sqcup}$ réalise la même séquence d'opérations d'enforcement. Avec un raisonnement similaire nous obtenons le résultat attendu.■

Pour les séquences infinies, le raisonnement est similaire au raisonnement fait pour le cas inductif. Cela est fait sur la forme de la séquences des opérations d'enforcement réalisée et en distinguant deux cas selon que $\varphi(\sigma)$ ou $\neg\varphi(\sigma)$.

La preuve pour la construction intersection est conduite de manière similaire. Une conséquence de ce théorème est que la classe *EP* des propriétés enforçables est fermée par union et intersection.

6.5 Synthèse de moniteurs d'enforcement

Nous présentons maintenant la synthèse de mécanismes d'enforcement depuis des automates de Streett dans le cadre des r-propriétés.

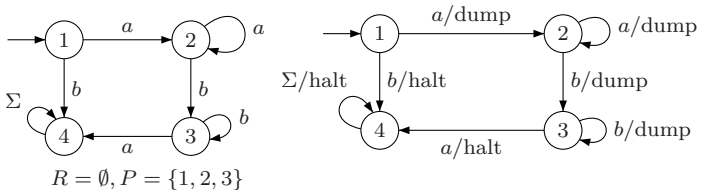


FIGURE 6.2 – Automate reconnaisseur et EM pour la r-propriété de safety $(a^* + a^+b^*, a^\omega + a^+b^\omega)$

6.5.1 Transformations spécifiques aux classes de propriétés

Nous définissons quatre transformations générales pour passer d'un automate reconnaissant une r-propriété de safety (resp. guarantee, obligation, response) vers un moniteur d'enforcement enforçant la même r-propriété. Les transformations sont spécifiques aux classes de propriétés enforçables de la classification.

Pour les r-propriétés de safety

Nous commençons par définir la transformation spécifique aux r-propriétés de safety et aux automates de Streett de safety.

DÉFINITION 50 (TRANSFORMATION SAFETY). Étant donné un automate de Streett de safety $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, (\emptyset, P))$ reconnaissant une r-propriété (enforçable) $\Pi \in \mathit{Safety}(\Sigma)$. Nous définissons la transformation $\mathsf{TransSafety}(\mathcal{A}_\Pi) = (Q^{\mathcal{A}_{\downarrow\Pi}}, q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}, \longrightarrow_{\mathcal{A}_{\downarrow\Pi}}, \mathit{Ops}, \Gamma^{\mathcal{A}_{\downarrow\Pi}})$ telle que :

- $Q^{\mathcal{A}_{\downarrow\Pi}} = Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}} = q_{\text{init}}{}^{\mathcal{A}_\Pi}$
- $\mathit{Halt}^{\mathcal{A}_{\downarrow\Pi}} = \overline{P}$
- $\rightarrow_{\mathcal{A}_{\downarrow\Pi}}$ est définie comme $\rightarrow_{\mathcal{A}_\Pi}$
- $\Gamma^{\mathcal{A}_{\downarrow\Pi}}$ est définie comme suit :
  - $\Gamma^{\mathcal{A}_{\downarrow\Pi}}(q) = \mathit{off}$ si $q \in P \wedge \mathit{Reach}_{\mathcal{A}_\Pi}(q) \subseteq P$ (TSAF1)
  - $\Gamma^{\mathcal{A}_{\downarrow\Pi}}(q) = \mathit{dump}$ si $q \in P \wedge \mathit{Reach}_{\mathcal{A}_\Pi}(q) \nsubseteq P$ (TSAF2)
  - $\Gamma^{\mathcal{A}_{\downarrow\Pi}}(q) = \mathit{halt}$ si $q \in \overline{P}$ (TSAF3)

Rappelons que dans un automate de safety, il n'y a pas de transition depuis $q \in \overline{P}$ vers $q' \in P$.

Nous notons $\mathcal{A}_{\downarrow\Pi} = \mathsf{TransSafety}(\mathcal{A}_\Pi)$. Informellement, le comportement d'un EM $\mathcal{A}_{\downarrow\Pi}$ obtenu depuis $\mathsf{TransSafety}(\mathcal{A}_\Pi)$ peut être compris comme suit. Tant que la séquence d'exécution courante satisfait la propriété considérée (*i.e.*, tant que l'exécution nous laisse dans les états P) de $\mathcal{A}_\Pi$, l'EM produit en sortie chaque événement (opération *dump*). Si la séquence courante satisfait la propriété et que toutes ses continuations possibles la satisfont également (*i.e.*, lorsque l'exécution atteint un état P tel que l'on ne puisse atteindre uniquement des états P), le moniteur s'éteint définitivement (opération *off*). Une fois que la séquence d'exécution dévie de la propriété (*i.e.*, lorsque l'exécution atteint un état de $\overline{P}$ de $\mathcal{A}_\Pi$), alors l'EM arrête immédiatement le programme sous-jacent avec une opération d'enforcement *halt*. Les exemples suivants illustrent ce principe.

Exemple 21 (Transformation safety) Considérant un ensemble d'événements $\Sigma = \{a, b\}$, sur la Fig. 6.2 (partie gauche) est représenté un automate de Streett reconnaissant la r-propriété de safety exprimée en utilisant des expressions ω-régulières : $(a^* + a^+b^*, a^\omega + a^+b^\omega)$. Son ensemble d'états est $\{1, 2, 3, 4\}$, l'état initial est 1, et nous avons $R = \emptyset$ et $P = \{1, 2, 3\}$. La partie droite montre l'EM correspondant obtenu en utilisant la transformation $\mathsf{TransSafety}$.

Exemple 22 (Transformation Safety) La partie droite de la Fig. 6.3 montre l'EM $\mathcal{A}_{\downarrow\Pi_1}$ obtenu en utilisant la transformation $\mathsf{TransSafety}$ appliquée sur $\mathcal{A}_{\Pi_1}$.

Pour les r-propriétés de guarantee

Nous définissons maintenant une transformation pour les r-propriétés de *guarantee*.

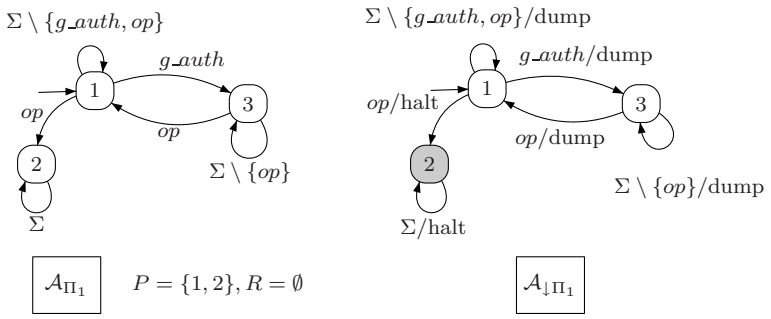


FIGURE 6.3 – Automate reconnaisseur et EM pour la r-propriété de safety Π_1

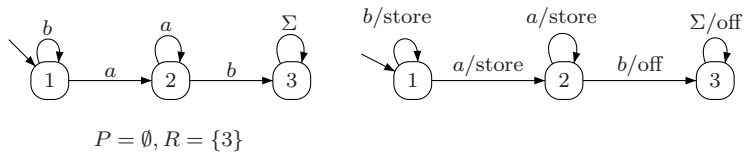


FIGURE 6.4 – Automate reconnaisseur et EM pour la r-propriété de guarantee $(b^* \cdot a^+ \cdot b \cdot \Sigma^*, b^* \cdot a^+ \cdot b \cdot \Sigma^\omega)$

DÉFINITION 51 (TRANSFORMATION GUARANTEE). Étant donné un automate de Streett de guarantee $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, (R, \emptyset))$ reconnaissant une r-propriété de guarantee (enforçable) $\Pi \in \mathit{Guarantee}(\Sigma)$. Nous définissons la transformation $\mathsf{TransGuarantee}(\mathcal{A}_\Pi) = (Q^{\mathcal{A}_{\downarrow\Pi}}, q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}, \longrightarrow_{\mathcal{A}_{\downarrow\Pi}}, \mathit{Ops}, \Gamma^{\mathcal{A}_{\downarrow\Pi}})$ *t.q.* :

- $Q^{\mathcal{A}_{\downarrow\Pi}} = Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}} = q_{\text{init}}{}^{\mathcal{A}_\Pi}$,
- $\rightarrow_{\mathcal{A}_{\downarrow\Pi}}$ est définie comme $\rightarrow_{\mathcal{A}_{\downarrow\Pi}}$
- $\Gamma^{\mathcal{A}_{\downarrow\Pi}}$ est définie comme suit :
  - $\Gamma^{\mathcal{A}_{\downarrow\Pi}}(q) = \mathit{off}$ si $q \in R$ (TGUAR1)
  - $\Gamma^{\mathcal{A}_{\downarrow\Pi}}(q) = \mathit{store}$ si $q \in \overline{R} \wedge \mathit{Reach}_{\mathcal{A}_\Pi}(q) \not\subseteq \overline{R}$ (TGUAR2)
  - $\Gamma^{\mathcal{A}_{\downarrow\Pi}}(q) = \mathit{halt}$ si $q \in \overline{R} \wedge \mathit{Reach}_{\mathcal{A}_\Pi}(q) \subseteq \overline{R}$ (TGUAR3)

Rappelons que dans un automate de guarantee, il n'y a pas de transition depuis les états $q \in R$ vers les états $q' \in \overline{R}$. Et, comme $P = \emptyset$, nous n'avons pas de transition depuis $q \in P$ vers $q' \in P$.

Nous notons $\mathcal{A}_{\downarrow\Pi} = \mathsf{TransGuarantee}(\mathcal{A}_\Pi)$. Un EM $\mathcal{A}_{\downarrow\Pi}$ obtenu à partir de $\mathsf{TransGuarantee}(\mathcal{A}_\Pi)$ se comporte comme suit. Tant que la séquence d'exécution courante ne satisfait pas la propriété désirée (*i.e.*, tant que le run sur $\mathcal{A}_\Pi$ reste dans les états $\overline{R}$), il mémorise chaque événement (opération *store*) donné en entrée dans sa mémoire. Dès que la séquence d'exécution satisfait la propriété (*i.e.*, lorsque $\mathcal{A}_\Pi$ atteint un état R), il réalise l'opération *off* pour vider le contenu mémoire, l'événement courant, et s'éteindre. Les exemples suivants illustrent ce principe.

Exemple 23 (Transformation guarantee) Considérant un vocabulaire $\Sigma = \{a, b\}$, sur la Fig. 6.4 (partie gauche) est montré un automate de Streett reconnaissant la r-propriété de guarantee exprimée par $(b^* \cdot a^+ \cdot b \cdot \Sigma^*, b^* \cdot a^+ \cdot b \cdot \Sigma^\omega)$. Son ensemble d'états est $\{1, 2, 3\}$, l'état initial est 1, et nous avons $R = \{3\}$ et $P = \emptyset$. La partie droite montre l'EM enforçant la même propriété, obtenue par la transformation $\mathsf{TransGuarantee}$. On peut remarquer qu'il n'y a pas d'état *Halt*.

Exemple 24 (Transformation guarantee) Sur la Fig. 6.5 (en bas) est montré l'EM enforçant la r-propriété Π_2, obtenue par $\mathsf{TransGuarantee}$ sur $\mathcal{A}_{\Pi_2}$. On peut remarquer que cet EM n'a pas d'état *Halt* car depuis tout état, un état R est atteignable.

Pour les r-propriétés d'obligation

Comme les r-propriétés d'obligation peuvent être définies comme des combinaisons booléennes positives de r-propriétés de safety et de guarantee (cf. Chapitre 3), le corollaire suivant est une conséquence directe du théorème μ de l'union des moniteurs d'enforcement (p. 109) et du théorème ν de la correction des

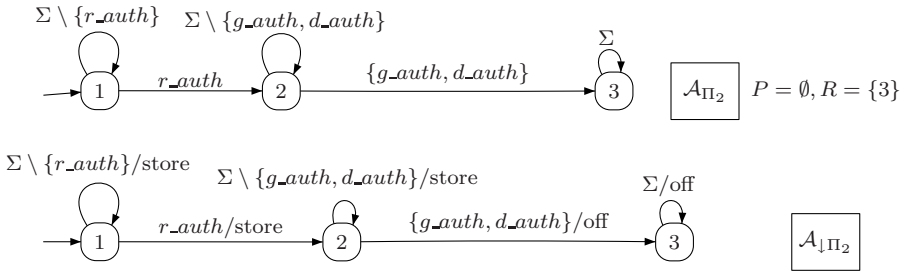


FIGURE 6.5 – Automate reconnaisseur et EM pour la r-propriété de guarantee pour Π_2

transformations spécifiques aux classes (p. 115). La preuve de ce corollaire fournit une construction systématique lorsque l'automate reconnaissant la r-propriété est décomposable en un produit d'automates de safety et guarantee.

Corollaire ε (Transformation indirecte pour les obligations): *Étant donné un programme $\mathcal{P}_\Sigma$, une r-propriété d'*obligation Π *est enforçable par un EM obtenu en utilisant les transformations safety et guarantee et des opérations Union et Intersection.*

PREUVE : Considérons un programme $\mathcal{P}_\Sigma$, une r-propriété d'obligation Π sur $\mathcal{P}_\Sigma$, nous voulons montrer qu'il existe un EM $\mathcal{A}_{\downarrow\Pi}$ *t.q.* $Enf(\mathcal{A}_{\downarrow\Pi}, \Pi, \mathcal{P}_\Sigma)$. Nous savons qu'il existe $n > 0$ *t.q.* Π peut être exprimée $\bigcap_{i=1}^{n}(\mathsf{Safety}_i \cup \mathsf{Guarantee}_i)$. La preuve se fait par une induction sur n. Depuis chaque r-propriété Safety_i (resp. $\mathsf{Guarantee}_i$) nous construisons un EM en utilisant la transformation TransSafety (resp. TransGuarantee). Ensuite, en utilisant les constructions union et intersection nous obtenons l'EM pour la r-propriété considérée. ■

Exemple 25 (Transformation indirecte pour les obligations) Pour un vocabulaire $\Sigma = \{a, b\}$, considérons la propriété $(a^* + \Sigma^* \cdot b \cdot \Sigma^*, a^\omega + \Sigma^* \cdot b \cdot \Sigma^\omega)$. Cette propriété est une r-propriété d'obligation (ce n'est ni une safety ni une guarantee) et est enforçable. En effet, il est possible d'obtenir deux EMs (cf. Fig. 6.1) pour la r-propriété de safety (a^*, a^ω) et la r-propriété de guarantee $(\Sigma^* \cdot b \cdot \Sigma^*, \Sigma^* \cdot b \cdot \Sigma^\omega)$. En utilisant la construction union, nous pouvons obtenir un EM $\mathcal{A}_\sqcup$ enforçant la r-propriété d'obligation.

Nous définissons également une transformation directe pour les r-propriétés d'obligation. Informellement, la transformation TransObligation combine les effets des deux transformations des classes de safety et guarantee sur chaque paire de l'automate de Streett.

DÉFINITION 52 (TRANSFORMATION OBLIGATION). Étant donné un automate de Streett de m-obligation $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, {q_{\text{init}}}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ reconnaissant une r-propriété (enforçable) d'obligation $\Pi \in \mathit{Obligation}(\Sigma)$. Nous définissons la transformation $\mathsf{TransObligation}(\mathcal{A}_\Pi) = (Q^{\mathcal{A}_{\downarrow\Pi}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi}}, \longrightarrow_{\mathcal{A}_{\downarrow\Pi}}, \mathit{Ops}, \Gamma^{\mathcal{A}_{\downarrow\Pi}})$ *t.q.* :

- $Q^{\mathcal{A}_{\downarrow\Pi}} = Q^{\mathcal{A}_\Pi}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi}} = {q_{\text{init}}}^{\mathcal{A}_\Pi}$,
- $\rightarrow_{\mathcal{A}_{\downarrow\Pi}}$ est définie comme $\rightarrow_{\mathcal{A}_\Pi}$
- $\forall q \in Q^{\mathcal{A}_{\downarrow\Pi}} : \Gamma(q) = \sqcap_{i=1}^{m}(\sqcup(\beta_i, \gamma_i))$ où les β_i et γ_i sont obtenus de la manière suivante :
  - $\beta_i = \mathit{off}$ si $q \in P_i \wedge \mathit{Reach}_{\mathcal{A}_\Pi}(q) \subseteq P_i$
  - $\beta_i = \mathit{dump}$ si $q \in P_i \wedge \mathit{Reach}_{\mathcal{A}_\Pi}(q) \nsubseteq P_i$
  - $\beta_i = \mathit{halt}$ si $q \notin P_i$
  - $\gamma_i = \mathit{off}$ si $q \in R_i$
  - $\gamma_i = \mathit{store}$ si $q \notin R_i \wedge \mathit{Reach}_{\mathcal{A}_\Pi}(q) \nsubseteq \overline{R_i}$
  - $\gamma_i = \mathit{halt}$ si $q \notin R_i \wedge \mathit{Reach}_{\mathcal{A}_\Pi}(q) \subseteq \overline{R_i}$

Notons qu'il n'y a pas de transition depuis un état $q \in R_i$ vers un état $q' \in \overline{R_i}$, et pas de transition depuis un état $q \in \overline{P_i}$ vers un état $q' \in P_i$.

Exemple 26 (Transformation spécifique aux obligations) Au bas de la Fig. 6.6 est représenté l'EM enforçant la r-propriété de 1-obligation Π_3 de l'Exemple 3, obtenu par la transformation TransObligation. On peut remarquer que cet EM n'a pas d'état *Halt*.

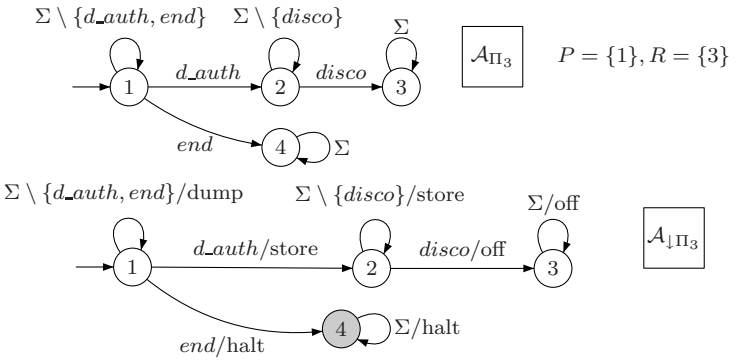


FIGURE 6.6 – Un automate de 1-obligation et l'EM correspondant pour la propriété Π_3

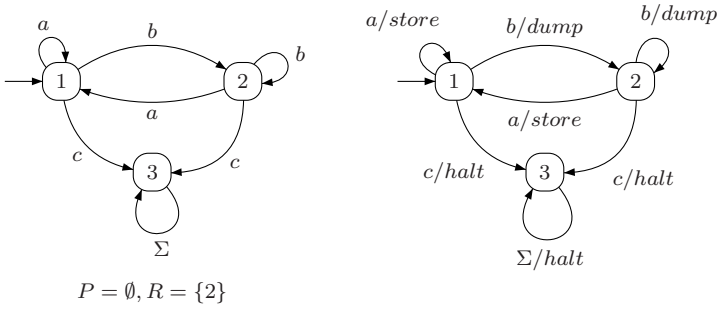


FIGURE 6.7 – Un automate de response et l'EM correspondant pour la r-propriété $((a^* \cdot b)^*, (a^* \cdot b)^\omega$

Pour les r-propriétés de response

Trouver une transformation pour une r-propriété de *response* Π nécessite d'étendre légèrement la définition de TransGuarantee pour prendre en compte les transitions d'un automate de Streett partant d'un état appartenant à R vers un état de $\overline{R}$ (de telles transitions sont absentes lorsque Π est une r-propriété de *guarantee*). Ainsi, nous introduisons une nouvelle transformation nommée TransResponse obtenue depuis la transformation TransGuarantee en ajoutant une règle pour prendre en compte la différence mentionnée.

DÉFINITION 53 (TRANSFORMATION RESPONSE). Étant donné un automate de Streett de response $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, (R, \emptyset))$ reconnaissant une r-propriété de response $\Pi \in \mathit{Response}(\Sigma)$. Nous définissons la transformation TransResponse($\mathcal{A}_\Pi$) = $(Q^{\mathcal{A}_{\downarrow\Pi}}, q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}, \longrightarrow_{\mathcal{A}_{\downarrow\Pi}}, \mathit{Ops}, \Gamma^{\mathcal{A}_{\downarrow\Pi}})$ en utilisant la transformations TransGuarantee et en remplaçant (TGUAR1) par :

- $\Gamma^{\mathcal{A}_{\downarrow\Pi}}(q) = \mathit{dump}$ si $q \in R \wedge \mathit{Reach}_{\mathcal{A}_\Pi}(q) \not\subseteq R$ (TRESP1)
- $\Gamma^{\mathcal{A}_{\downarrow\Pi}}(q) = \mathit{off}$ si $q \in R \wedge \mathit{Reach}_{\mathcal{A}_\Pi}(q) \subseteq R$ (TRESP2)

Un EM $\mathcal{A}_{\downarrow\Pi}$ obtenu par la transformation TransResponse($\mathcal{A}_\Pi$) traite la séquence d'exécution qui lui est donnée et enforce la propriété reconnue originellement par l'automate de Streett. Informellement le principe est similaire à celui de la transformation pour les r-propriétés de guarantee, excepté le fait qu'il peut y avoir une alternance dans le run des séquences d'exécution entre les états R et les états $\overline{R}$. Tant que la séquence d'exécution courante ne satisfait pas la propriété sous-jacente (l'état courant correspond à un état $\overline{R}$), il effectue l'opération *store* pour chaque événement de la séquence. Une fois que la séquence d'exécution satisfait la r-propriété (l'état courant est un état R), il effectue l'opération :

- *dump* pour vider le contenu mémoire des événements mémorisés jusque là et l'événement courant en sortie. Ceci dans le cas où un état $\overline{R}$ est accessible à partir de l'état courant.
- *off* pour vider le contenu mémoire des événements mémorisés jusque là et l'événement courant en sortie et éteindre le moniteur. Ceci dans le cas où l'on peut atteindre uniquement des états R depuis l'état courant.

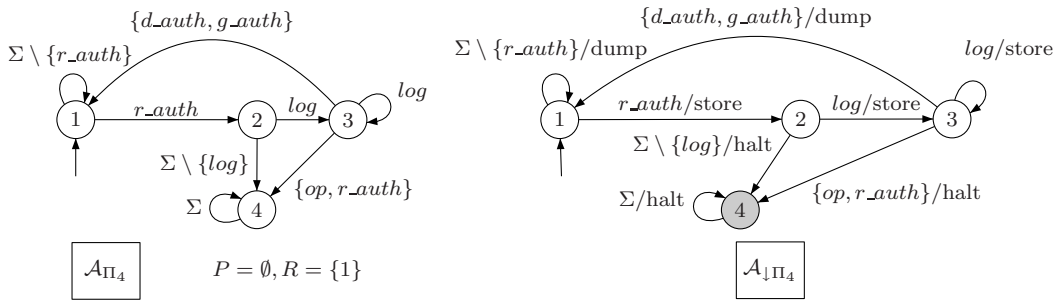


FIGURE 6.8 – Un automate de response et l'EM correspondant pour la r-propriété Π_4

Exemple 27 (Transformation response) Considérons un ensemble d'événements $\Sigma = \{a, b, c\}$, la Fig. 6.7 (partie gauche) montre un automate de Streett reconnaissant la r-propriété de response exprimée en utilisant les expressions régulières $((a^* \cdot b)^*, (a^* \cdot b)^\omega)$. Son ensemble d'états est $\{1, 2, 3\}$, l'état initial est 1, et nous avons $R = \{2\}$ et $P = \emptyset$. La partie droite montre l'EM enforçant la même propriété, obtenu par la transformation TransResponse. On peut remarquer qu'il y a un état *Halt* qui est l'état 3.

Exemple 28 (Transformation Response) La partie droite de la Fig. 6.8 montre l'EM enforçant la r-propriété de response Π_4 introduite dans l'Exemple 3, obtenu par la transformation TransResponse. Il y a un état *Halt* qui est l'état 4.

6.5.2 Correction des transformations

Correction des transformations spécifiques. En utilisant les transformations précédemment présentées, il est possible d'obtenir un EM pour une r-propriété depuis un automate reconnaisseur de cette r-propriété enforçable. Plus précisément, étant donné *n'importe quelle* r-propriété de safety (resp. guarantee, obligation, response) Π, et un automate de Streett reconnaissant Π, il est possible de *synthétiser* depuis cet automate un moniteur d'enforcement pour Π en utilisant les transformations systématiques précédemment présentées. Ceci est dû à la correction de ces transformations.

Théorème ν : *Étant donné un programme $\mathcal{P}_\Sigma$, une r-propriété de safety (resp. de guarantee, d'obligation, de response) $\Pi \in \text{Safety}(\Sigma)$ (resp. $\Pi \in \text{Guarantee}(\Sigma), \Pi \in \text{Obligation}(\Sigma), \Pi \in \text{Response}(\Sigma)$) est enforçable sur $\mathcal{P}_\Sigma$ par un EM obtenu par l'application de la transformation de safety (resp. guarantee, response) sur l'automate reconnaissant Π. Plus formellement, étant donné $\mathcal{A}_\Pi$ reconnaissant Π, nous avons :*

$$(\Pi \in \text{Safety}(\Sigma) \wedge \mathcal{A}_{\downarrow\Pi} = \text{TransSafety}(\mathcal{A}_\Pi)) \Rightarrow Enf(\mathcal{A}_{\downarrow\Pi}, \Pi, \mathcal{P}_\Sigma),$$
$$(\Pi \in \text{Guarantee}(\Sigma) \wedge \mathcal{A}_{\downarrow\Pi} = \text{TransGuarantee}(\mathcal{A}_\Pi)) \Rightarrow Enf(\mathcal{A}_{\downarrow\Pi}, \Pi, \mathcal{P}_\Sigma).$$
$$(\Pi \in \text{Obligation}(\Sigma) \wedge \mathcal{A}_{\downarrow\Pi} = \text{TransObligation}(\mathcal{A}_\Pi)) \Rightarrow Enf(\mathcal{A}_{\downarrow\Pi}, \Pi, \mathcal{P}_\Sigma).$$
$$(\Pi \in \text{Response}(\Sigma) \wedge \mathcal{A}_{\downarrow\Pi} = \text{TransResponse}(\mathcal{A}_\Pi)) \Rightarrow Enf(\mathcal{A}_{\downarrow\Pi}, \Pi, \mathcal{P}_\Sigma).$$

Pour prouver ce résultat, nous devons montrer que le moniteur d'enforcement $\mathcal{A}_{\downarrow\Pi}$, résultat de la transformation depuis un automate reconnaisseur $\mathcal{A}_\Pi$ de Π, vérifie : pour toute séquence d'exécution $\sigma \in Exec(\mathcal{P}_\Sigma)$, il existe $o \in \Sigma^*$ *t.q.* les contraintes suivantes sont vérifiées :

$$\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} o \tag{6.22}$$
$$\Pi(\sigma) \Rightarrow \sigma = o \tag{6.23}$$
$$\neg\Pi(\sigma) \wedge Pref_{\prec}(\phi, \sigma) = \emptyset \Rightarrow o = \epsilon \tag{6.24}$$
$$\neg\Pi(\sigma) \wedge Pref_{\prec}(\phi, \sigma) \neq \emptyset \Rightarrow o = Max(Pref_{\prec}(\phi, \sigma)) \tag{6.25}$$

Pour les preuves à venir, nous notons $(Q^{\mathcal{A}_{\downarrow\Pi}}, q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}, \longrightarrow_{\mathcal{A}_{\downarrow\Pi}}, Ops, \Gamma^{\mathcal{A}_{\downarrow\Pi}})$ le moniteur d'enforcement obtenu par la transformation appliquée.

Pour la classe des r-propriétés de safety. Nous notons $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, (\emptyset, P))$. Considérons une séquence d'exécution du programme $\sigma \in Exec(\mathcal{P}_\Sigma)$. Nous étudions l'effet de la soumission de σ à $\mathcal{A}_{\downarrow\Pi}$. Nous associerons l'exécution de σ sur $\mathcal{A}_\Pi$ à l'exécution de σ sur $\mathcal{A}_{\downarrow\Pi}$. L'exécution de σ sur $\mathcal{A}_\Pi$ produit une trace $(q_0, \sigma_0, q_1) \cdot (q_1, \sigma_1, q_2) \cdots (q_i, \sigma_i, q_{i+1}) \cdots$ qui correspond à une trace $(q_0, \sigma_0/\alpha_0, q_1) \cdots (q_i, \sigma_i/\alpha_i, q_{i+1}) \cdots$ sur $\mathcal{A}_{\downarrow\Pi}$ avec $q_0 = q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}$. Nous distinguons deux cas selon que la séquence σ satisfait la propriété Π ou non.

Le premier cas est $\Pi(\sigma)$. L'automate de Streett $\mathcal{A}_\Pi$ accepte σ, distinguons selon que σ est finie ou non.

- Si $\sigma \in \Sigma^*$, soit $n = |\sigma|$. Comme σ est acceptée par $\mathcal{A}_\Pi$ (et selon la Définition 17, p. 50), nous savons qu'en lisant σ, l'automate "reste dans les états P" : nous avons $R = \emptyset$ et pas de transition depuis les états $\overline{P}$ vers les états P puisque $\mathcal{A}_\Pi$ est un automate de safety.
  Si $\sigma = \epsilon$, nous avons (6.22) car $\epsilon \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$. De plus, $Pref_{\prec}(\phi, \epsilon) = \emptyset$, ce qui donne (6.24).
  Sinon ($\sigma \neq \epsilon$), selon les contraintes de la relation de transition d'un automate de safety et (TransSafety1), la trace de σ sur $\mathcal{A}_{\downarrow\Pi}$ est telle que $\forall i \leq n, \alpha_i = dump \vee \alpha_i = off$.
  En utilisant la trace d'exécution de $\mathcal{A}_{\downarrow\Pi}$ et la définition des opérations d'enforcement, nous déduisons la dérivation de configurations suivante :

$$(q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}, \sigma, \epsilon) \overset{\sigma_0/\alpha_0}{\hookrightarrow} (q_1, \sigma_{1\cdots}/\alpha_1, \epsilon) \cdots \overset{\sigma_{n-2}/\alpha_{n-2}}{\hookrightarrow} (q_{n-1}, \sigma_{n-1\cdots}/\alpha_{n-1}, \epsilon) \overset{\sigma_{n-1}}{\hookrightarrow} (q_n, \sigma_n, \epsilon)$$

  comme $\forall i < n : dump(\sigma_i, \epsilon) = off(\sigma_i, \epsilon) = (\sigma_i, \epsilon)$
  Par déduction, en utilisant les dérivations en plusieurs pas, nous avons $(q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}, \sigma, \epsilon) \overset{\sigma}{\Rightarrow}_{\mathcal{A}_{\downarrow\Pi}} (q_{n-1}, \epsilon, \epsilon)$. C'est-à-dire $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma$. Ce qui assure (6.22). Par ailleurs, selon le critère d'acceptation des r-propriétés, nous avons $\phi(\sigma)$, ce qui nous permet de déduire (6.23), car $\sigma = \sigma$.
- Si $\sigma \in \Sigma^\omega$, alors en utilisant la définition du critère d'acceptation des séquences finies (Définition 17, p. 50) pour un automate de Streett et la définition des automates de safety, nous avons $vinf(\sigma) \subseteq P$. Ce qui signifie que les seuls états visités infiniment souvent sont des états P. Comme il n'y a pas de transition depuis un état dans $\overline{P}$ vers un état dans P, aucun état de $\overline{P}$ n'est visité. Le run de σ sur $\mathcal{A}_\Pi$ est *t.q.* $\forall i \in \mathbb{N} : q_i \in P$. Ce qui par un raisonnement similaire nous conduit à trouver la trace de σ sur $\mathcal{A}_{\downarrow\Pi}$. La séquence des opérations d'enforcement vérifie ainsi l'expression régulière $dump^\omega + dump^* \cdot off^\omega$. Nous avons $\forall \sigma' \in \Sigma^* : \sigma' \prec \sigma \Rightarrow \sigma' \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma'$. Il s'en suit que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma$. Nous avons ainsi (6.22) et (6.23).

Le deuxième cas est $\neg\Pi(\sigma)$. La séquence σ n'est pas acceptée par $\mathcal{A}_\Pi$. Il y a deux cas selon que $Pref_{\prec}(\phi, \sigma) = \emptyset$ ou non.

- *Si* $Pref_{\prec}(\phi, \sigma) = \emptyset$. Selon les contraintes sur les automates de safety, $\mathcal{A}_\Pi$ démarre dans un état $\overline{P}$ et reste dedans : il n'y a pas de transition depuis les états $\overline{P}$ vers les états P. Nous déduisons que la trace d'exécution de σ sur $\mathcal{A}_\Pi$ est *t.q.* $\forall i > 0 : q_i \notin P$. En utilisant la définition de la transformation TransSafety nous pouvons trouver $trace(\sigma, \mathcal{A}_{\downarrow\Pi})$. Ainsi, l'opération d'enforcement réalisée par $\mathcal{A}_{\downarrow\Pi}$ est toujours *halt*. C'est-à-dire $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$. (6.22). Ensuite $Pref_{\prec}(\phi, \sigma) = \emptyset$ implique que $\forall \sigma' \prec \sigma : \neg\phi(\sigma)$. Nous avons (6.24).
- *Sinon* ($Pref_{\prec}(\phi, \sigma) \neq \emptyset$), il y a au moins un préfixe de σ satisfaisant ϕ. Comme Π est une r-propriété de safety, nous pouvons décomposer σ en $\sigma_{good} \cdot \sigma_{bad}$ où :
  - σ_{good} est le plus long préfixe de σ satisfaisant ϕ (et également Π), et tous les préfixes de σ_{good} satisfont ϕ ;
  - $\sigma_{bad} \neq \epsilon$.

  En utilisant un raisonnement similaire à celui utilisé dans le premier cas, nous pouvons trouver sur $\mathcal{A}_\Pi$ une trace $(q_0, \sigma_0, q_1) \cdots (q_{s-1}, \sigma_{s-1}, q_s)$, avec $s = |\sigma_{good}|$. À laquelle nous pouvons associer une trace sur $\mathcal{A}_{\downarrow\Pi}$: $(q_0, \sigma_0/\alpha_0, q_1) \cdots (q_{s-1}, \sigma_{s-1}/\alpha_{s-1}, q_s)$. De plus comme $\forall i \in [0, s-1] : Reach_{\mathcal{A}_\Pi}(q_i) \not\subseteq P$, nous avons $\forall i \in [0, s-1] : \alpha_i = dump$.
  L'exécution de σ_{bad} produit une trace $(q_s, \sigma_s, q_{s+1}) \cdots (q_{s+j}, \sigma_{s+j}, q_{s+j+1}) \cdots$, avec $j > 0$. Ensuite, $\mathcal{A}_\Pi$ passe de $q_{s-1} \in P$ à $q_s \notin P$. Après quoi, selon (TSafety2), nous obtenons pour $i > s$, $(q_0, \sigma_0/dump, q_1) \cdots (q_{s-1}, \sigma_{s-1}/dump, q_s) \cdot (q_s, \sigma_s/halt, q_{s+1}) \cdots (q_i, \sigma_i/halt, q_{i+1}) \cdots$ (avec $\forall i > s : q_i \notin P$).
  Comme dans le premier cas, comme $dump(\sigma_i, \epsilon) = (\sigma_i, \epsilon)$ nous trouvons que $(q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}, \sigma, \epsilon) \overset{\sigma_{good}}{\Rightarrow} (q_{s-1}, \sigma_{bad}, \epsilon)$. Ensuite, en utilisant $\forall i > |\sigma_{good}| : halt(\sigma_i, \epsilon) = (\epsilon, \epsilon)$, $\forall i > |\sigma_{good}| : (q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}, \sigma, \epsilon) \overset{\sigma_{good}}{\Rightarrow} (q_i, \sigma_{i\cdots}, \epsilon)$, *i.e.*, $\sigma_{\cdots i} \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma_{good}$. Il s'en suit que nous avons :
  - Pour tout $\sigma' \in \Sigma^*$ *t.q.* $\sigma' \preceq \sigma_{good} \prec \sigma$, nous avons $\sigma' \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma'$, ce qui donne (6.22). Ensuite comme

$\phi(\sigma')$, nous obtenons (6.23).
- Pour tout $\sigma' \in \Sigma^*$ *t.q.* $\sigma_{good} \prec \sigma' \prec \sigma$, nous avons $\sigma' \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma_{good}$ (6.22). De plus $\neg\phi(\sigma')$, et σ_{good} est le plus long préfixe de σ satisfaisant ϕ. C'est-à-dire $Pref_{\prec}(\phi, \sigma) \neq \emptyset$, et $Max(Pref_{\prec}(\phi, \sigma)) = \sigma_{good}$. Ce qui donne (6.24).

Pour la classe des r-propriétés de guarantee. La preuve pour les r-propriétés de guarantee suit le même principe que celle pour les r-propriétés de safety. Aussi, nous ne faisons que décrire la preuve. La preuve complète peut être trouvée dans l'Annexe A.4.1 (p. 197). Considérons une séquence d'exécution $\sigma \in Exec(\mathcal{P}_\Sigma)$. En examinant le run de σ sur $\mathcal{A}_\Pi$, nous déduisons, en utilisant la définition de la transformation TransGuarantee, la forme de la séquence des opérations d'enforcement. Nous distinguons deux cas : $\Pi(\sigma)$ et $\neg\Pi(\sigma)$.
- Le premier cas est $\Pi(\sigma)$. Cela signifie que le run de σ sur $\mathcal{A}_\Pi$ a atteint un état R et reste dedans. Si σ est une séquence finie, alors la séquence est de la forme $store^* \cdot off^+$. Sinon (σ est une séquence infinie) la séquence est de la forme $store^* \cdot off^\omega$.
- Le deuxième cas est $\neg\Pi(\sigma)$. Ce qui signifie que le run de σ sur $\mathcal{A}_\Pi$ n'atteint pas un état R et reste donc dans les états $\overline{R}$. Si σ est une séquence finie, la séquence des opérations d'enforcement est de la forme $store^* \cdot halt^*$. Sinon (σ est une séquence infinie), la séquence des opérations d'enforcement vérifie l'expression régulière $store^\omega + store^* \cdot halt^\omega$.

Pour la classe des r-propriétés d'obligation. Pour prouver la correction de cette transformation, nous allons montrer que : l'EM obtenu par l'application des transformations TransSafety, TransGuarantee, l'opération *Intersection* (cet EM est correct par construction) ; et l'EM obtenu par l'application directe de TransObligation, sont les mêmes.

Pour prouver le théorème, nous montrons que la propriété suivante est vérifiée : étant donné $\mathcal{A}_\Pi$ reconnaissant Π

$$\forall k \geq 1 : (\Pi \in \text{Obligation}_k(\Sigma) \wedge \mathcal{A}_{\downarrow\Pi} = \text{TransObligation}(\mathcal{A}_\Pi)) \Rightarrow Enf(\mathcal{A}_{\downarrow\Pi}, \Pi, \mathcal{P}_\Sigma)$$

Pour cela, nous réalisons une induction sur k où la r-propriété Π est une k-obligation.

Cas de base. Pour le cas de base, $k = 1$, Π est une simple obligation. Nous notons $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, {q_{\text{init}}}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R, P)\})$. Soit $\sigma \in \Sigma^\infty$. Étudions l'effet de la soumission de de σ sur $\mathcal{A}_{\downarrow\Pi}$. Il y a deux cas en fonction de $\Pi(\sigma)$.
- *Le premier cas est* $\Pi(\sigma)$. Dans ce cas, la séquence σ est acceptée par $\mathcal{A}_\Pi$. Selon les contraintes d'un automate d'obligation, l'exécution de σ sur $\mathcal{A}_\Pi$ produit une trace d'exécution $(q_0, \sigma_0, q_1) \cdot (q_1, \sigma_1, q_2) \cdots (q_i, \sigma_i, q_{i+1}) \cdots$ de σ telle que $q_0 = {q_{\text{init}}}^{\mathcal{A}_\Pi}$ et
  - soit chaque état q_i est dans P,
  - soit il existe $0 \leq k$ *t.q.* $\forall i < k : q_i \in \overline{R} \wedge \forall i \geq k : q_i \in R$

  (il n'y a pas de transition depuis $q \in \overline{P}$ vers $q' \in P$ et depuis $q \in R$ vers $q' \in \overline{R}$).

  Dans les deux cas, nous pouvons suivre le raisonnement précédemment utilisé pour les r-propriétés de safety et guarantee pour obtenir la trace sur $\mathcal{A}_{\downarrow\Pi}$. Notons que si les deux contraintes précédentes sont vraies simultanément, la forme de la séquence d'exécution est la même.
- *Le deuxième cas est* $\neg\Pi(\sigma)$. Similairement dans ce cas, cela revient aux cas où Π est soit une r-propriété de safety soit de guarantee.

Énoncé d'induction. Soit $n \in \mathbb{N}^*$, supposons que pour $k \leq n$, si Π est une k-obligation reconnue par un automate de k-obligation $\mathcal{A}_\Pi$, alors l'EM $\mathcal{A}_{\downarrow\Pi} = \text{TransObligation}(\mathcal{A}_\Pi)$ enforce Π, c'est-à-dire, nous avons $Enf(\mathcal{A}_{\downarrow\Pi}, \Pi, \mathcal{P}_\Sigma)$.

Maintenant considérons Π une r-propriété de $(k+1)$-obligation, $\mathcal{A}_\Pi$ reconnue par un automate de $(k+1)$-obligation, et $\mathcal{A}_{\downarrow\Pi} = \text{TransObligation}(\mathcal{A}_\Pi)$. Comme Π est une r-propriété de $(k+1)$-obligation, Π peut être exprimée comme $\bigcap_{i=1}^{k+1} \Pi_i$ où les Π_i sont des simples r-propriétés d'obligation (Lemme α, p. 44). L'expression de Π peut être récrite comme $\Pi = (\bigcap_{i=1}^{k} \Pi_i) \cap \Pi_{k+1}$. En utilisant le Lemme β (p. 51), on peut trouver deux automates reconnaisseurs $\mathcal{A}_{\Pi/\{1,\ldots,k\}}$ reconnaissant $\bigcap_{i=1}^{k} \Pi_i$ et $\mathcal{A}_{\Pi/\{k+1\}}$ reconnaissant Π_{k+1}. Maintenant en utilisant l'hypothèse d'induction, nous pouvons appliquer TransObligation() à ces deux automates pour obtenir deux EMs $\mathcal{A}_{\downarrow\Pi/\{1,\ldots,k\}}$ enforcant $\bigcap_{i=1}^{k} \Pi_i$ et $\mathcal{A}_{\downarrow\Pi/\{k+1\}}$ enforcant Π_{k+1}. En utilisant l'opération Intersection (Définition 49), on peut obtenir un EM $\mathcal{A}_{\downarrow\Pi}' = \text{Intersection}(\mathcal{A}_{\downarrow\Pi/\{1,\ldots,k\}}, \mathcal{A}_{\downarrow\Pi/\{k+1\}})$ enforcant (Théorème μ) $(\bigcap_{i=1}^{k} \Pi_i) \cap \Pi_{k+1} = \bigcap_{i=1}^{k+1} \Pi_i$, c'est-à-dire Π.

Maintenant, examinons l'EM $\mathcal{A}_{\downarrow\Pi}$ obtenu en appliquant directement la transformation TransObligation sur $\mathcal{A}_\Pi$. Nous comparons cet EM avec $\mathcal{A}_{\downarrow\Pi}'$ obtenu par l'hypothèse d'induction et la construction intersection (correct par construction).

- Pour $\mathcal{A}_{\downarrow\Pi}$, selon la Définition 52 de TransObligation :
  - $Q^{\mathcal{A}_{\downarrow\Pi}} = Q^{\mathcal{A}_\Pi}$,
  - ${q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi}} = {q_{\text{init}}}^{\mathcal{A}_\Pi}$,
  - et $\forall q \in Q^{\mathcal{A}_{\downarrow\Pi}'}, \Gamma(q)) = \sqcap_{i=1}^{k+1} \sqcup (\{\beta_i, \gamma_i\})$.
- Pour $\mathcal{A}_{\downarrow\Pi}'$, selon la Définition 49 de l'intersection entre deux EMs :
  - $Q^{\mathcal{A}_{\downarrow\Pi}'} = Q^{\mathcal{A}_{\downarrow\Pi/\{k+1\}}} \times Q^{\mathcal{A}_{\downarrow\Pi/\{1,\ldots,k\}}} = Q^{\mathcal{A}_\Pi} \times Q^{\mathcal{A}_\Pi}$,
  - ${q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi}'} = {q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi/\{k+1\}}} \times {q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi/\{1,\ldots,k\}}} = {q_{\text{init}}}^{\mathcal{A}_\Pi} \times {q_{\text{init}}}^{\mathcal{A}_\Pi}$,
  - et $\forall (q,q) \in Q^{\mathcal{A}_{\downarrow\Pi}'} : \Gamma((q,q)) = \sqcap_{i=1}^{k} \sqcup (\{\beta_i, \gamma_i\}) \sqcap (\sqcup(\{\beta_{k+1}, \gamma_{k+1})),\ i.e.,\ \alpha = \sqcap_{i=1}^{k+1} \sqcup (\{\beta_i, \gamma_i\})$.
- où, $\forall i \in [1,, k+1]$:
  - β_i est
    - *off* si $q \in P_i \wedge Reach_{\mathcal{A}_\Pi}(q) \subseteq P_i$
    - *dump* si $q \in P_i$
    - *halt* si $q \notin P_i$
  - γ_i est
    - *off* si $q \in R_i$
    - *halt* si $q \notin R_i \wedge \nexists q' \in R_i : q' \in Reach_{\mathcal{A}_\Pi}(q)$
    - *store* si $q \notin R_i \wedge \exists q' \in R_i : q' \in Reach_{\mathcal{A}_\Pi}(q)$

C'est-à-dire que nous pouvons exhiber une bijection entre $\mathcal{A}_{\downarrow\Pi}'$ et $\mathcal{A}_{\downarrow\Pi}$: $\forall q \in Q^{\mathcal{A}_\Pi}$, l'état q dans $\mathcal{A}_{\downarrow\Pi}$ est en relation avec l'état (q,q) dans $\mathcal{A}_{\downarrow\Pi}'$. Formellement, entre les deux EMs $\mathcal{A}_{\downarrow\Pi}$ et $\mathcal{A}_{\downarrow\Pi}'$, il y a une relation $\mathcal{R} \subseteq (Q^{\mathcal{A}_\Pi} \times (Q^{\mathcal{A}_\Pi} \times Q^{\mathcal{A}_\Pi}))$ définie par $\mathcal{R} = \{(q,(q,q)) \mid q \in Q^{\mathcal{A}_\Pi}\}$. Les deux EMs sont égaux (ils différent seulement par le nom de leur états). En conséquence, l'EM produit en appliquant directement TransObligation sur $\mathcal{A}_\Pi$ est correct. Ce qui conclut la preuve pour les r-propriétés de la classe *Obligation*.

Pour la classe des r-propriétés de response. De façon similaire aux r-propriétés de guarantee, nous examinons le run d'une séquence d'exécution $\sigma \in Exec(\mathcal{P}_\Sigma)$, et ensuite, suivant la définition de la transformation TransResponse, nous déduisons la forme de la séquence des opérations produites. La preuve complète peut être trouvée dans l'Annexe A.4.2.

- Le premier cas est $\Pi(\sigma)$. Nous distinguons deux cas selon le fait que σ soit une séquence finie ou non.
  - Si σ est une séquence finie, cela signifie que le run de σ sur $\mathcal{A}_\Pi$ termine dans un état R. Ainsi, la dernière opération d'enforcement produite par $\mathcal{A}_{\downarrow\Pi}$ est *dump*. La forme de la séquence des opérations d'enforcement est de la forme $(store + dump)^* \cdot dump$ ou $(store + dump)^* \cdot off^*$.
  - Si σ est une séquence infinie, alors cela signifie qu'un état R est visité infiniment souvent. D'où, $\mathcal{A}_{\downarrow\Pi}$ réalise régulièrement l'opération *dump* ou *off* (cf. Propriété 9, p. 104). Ainsi la forme de la séquence des opérations d'enforcement est $(store^* \cdot dump)^\omega$.
- Le deuxième cas est $\neg\Pi(\sigma)$. Nous distinguons selon que σ soit une séquence finie ou non.
  - Si σ est une séquence finie, alors cela signifie que le run de σ sur $\mathcal{A}_\Pi$ termine dans état $\overline{R}$. Ainsi, la dernière opération d'enforcement réalisée par $\mathcal{A}_{\downarrow\Pi}$ est *store* ou *halt*. La forme de la séquence des opérations d'enforcement est $\big((store + dump)^* \cdot (halt + store)\big) + store^* \cdot halt^*$.
  - Si σ est une séquence infinie, alors cela signifie que les états R ne sont pas visités infiniment souvent. Alors, $\mathcal{A}_{\downarrow\Pi}$ réalise l'opération *halt* ou *store* à partir d'un certain préfixe de σ. Ensuite, la forme de la séquence des opérations d'enforcement est $(store + dump)^* \cdot (store^\omega + store^* \cdot halt^\omega)$.

6.5.3 Transformation générale

Nous introduisons une transformation générale indépendante de la classe de la propriété reconnue par l'automate de Streett à laquelle elle s'applique.

Définition 54 (Transformation Streett2EM). Soit $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, {q_{\text{init}}}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ un automate de Streett reconnaissant une r-propriété enforçable $\Pi \in EP$. Nous définissons la transformation Streett2EM$(\mathcal{A}_\Pi) = (Q^{\mathcal{A}_\Pi}, {q_{\text{init}}}^{\mathcal{A}_\Pi}, \rightarrow_{\mathcal{A}_\Pi}, Ops, \Gamma)$ *t.q.* $\Gamma : Q^{\mathcal{A}_\Pi} \rightarrow Ops$ produit l'opération d'enforcement de la manière suivante : $\forall q \in Q^{\mathcal{A}_\Pi}$:

$$q \in Good^{\mathcal{A}_\Pi} \Rightarrow \Gamma(q) = off \qquad q \in Good_p^{\mathcal{A}_\Pi} \Rightarrow \Gamma(q) = dump$$
$$q \in Bad_p^{\mathcal{A}_\Pi} \Rightarrow \Gamma(q) = store \qquad q \in Bad^{\mathcal{A}_\Pi} \Rightarrow \Gamma(q) = halt$$

Correction de la transformation générale. La correction de la transformation générale *Streett2EM* est donnée par le théorème suivant.

Théorème ξ (correction de Streett2EM) : *La r-propriété $\Pi \in EP$ est enforçable sur $\mathcal{P}_\Sigma$ par un EM obtenu par l'application de la transformation Streett2EM. Plus formellement, étant donnée $\mathcal{A}_\Pi$ reconnaissant Π, nous avons :*

$$(\Pi \in EP \wedge \mathcal{A}_{!\Pi} = \text{Streett2EM}(\mathcal{A}_\Pi)) \Rightarrow Enf(\mathcal{A}_{!\Pi}, \Pi, \mathcal{P}_\Sigma)$$

Preuve : La preuve de la correction de *Streett2EM* repose sur le fait que cette transformation se réduit à la transformation spécifique à la classe de la propriété reconnue par l'automate de Streett. ■

6.6 Comparaison avec d'autres approches d'enforcement

Cette section fournit une comparaison avec les mécanismes proposés dans le domaine de l'enforcement.

6.6.1 Comparaison avec les mécanismes d'enforcement

En modifiant la séquence d'exécution, nos moniteurs d'enforcement peuvent être vus comme une forme restreinte de réécriture de programme (ce qui avait été remarqué aussi dans [HMS06]). Cependant, nous croyons que les mécanismes que nous proposons peuvent être adjoints à un programme en satisfaisant les contraintes d'un mécanisme à l'exécution. Cela nous semble être un bon compromis entre les moniteurs d'exécution purs et la réécriture de programme. Ceci dans le sens où nous donnons plus de capacité d'enforcement à nos mécanisme sans aucune modification du programme sous-jacent. Le seul contrôle dont nous avons besoin sur le programme sous-jacent est de pouvoir d'une certaine manière "encapsuler" les événements et les retarder avec un impact sémantique minimal. Dans ce travail, les moniteurs d'enforcement peuvent être vus comme des mécanismes appartenant à la catégorie des moniteurs d'exécution.

6.6.2 Comparaison avec les moniteurs d'enforcement

Les moniteurs définis jusqu'à maintenant ayant le plus de pouvoir d'enforcement étaient les edit-automates définis par Ligatti. Il est assez clair que les moniteurs d'enforcement que nous avons proposé sont similaires et inspirés des edit-automata. Le calcul opéré par nos moniteurs est réalisé grâce à leurs mémoire et un ensemble d'opérations. Dans les edit-automata, le calcul est réalisé en utilisant un ensemble d'états de contrôle. Toutefois, les moniteurs que nous proposons diffèrent en plusieurs points. À notre connaissance, ces fonctionnalités sont nouvelles concernant l'enforcement à l'exécution.

Tout d'abord, mettons en évidence la *généricité* des moniteurs introduits dans ce chapitre. Les automates de sécurité de Schneider peuvent facilement se voir comme une forme restreinte de nos moniteurs. En fait, il est possible de remarquer qu'en limitant l'ensemble des opérations *Ops* à l'ensemble $\{halt, dump\}$, il est possible de trouver un moniteur d'enforcement pour n'importe quel automate de sécurité. De plus comme les Shallow History Automata sont des restrictions des automates de sécurité, ils peuvent également être exprimés par nos moniteurs d'enforcement.

Les edit-automata tombent également dans le cadre de nos moniteurs d'enforcement. En effet, il est possible de remarquer que ces deux formes de moniteurs sont dotés de primitives équivalentes en terme de modification de séquence d'exécution. Il est assez évident qu'en considérant un ensemble d'opérations d'enforcement qui sont celles des edit-automates, nos moniteurs d'enforcement pourraient simuler n'importe quel edit-automate.

Nous proposons (Section 6.5) une traduction d'un automate reconnaisseur vers un automate enforceur. Cette transformation systématique facilite la définition de mécanismes d'enforcement. Trouver comment enforcer une propriété en utilisant les edit-automates n'est pas aisé. De plus, la taille du moniteur

d'enforcement est plus grande en utilisant les edit-automates qui encodent la mémorisation des événements dans les états de contrôle. Par exemple, dans le cas où la propriété à enforcer est connue, nous pouvons synthétiser un mécanisme d'enforcement plus concis en terme de nombre d'états. Par exemple, depuis une r-propriété de safety Π, nous synthétisons un moniteur d'enforcement en utilisant $\mathsf{TransSafety}(\mathcal{A}_\Pi)$ où $\mathcal{A}_\Pi$ est un automate reconnaisseur pour Π. Une remarque similaire s'applique aux edit-automates et pour l'ensemble des r-propriétés de response.

Nos moniteurs d'enforcement proposent une distinction claire entre états de contrôle (utilisés pour la reconnaissance de la propriété) et la mémorisation d'événements (lorsque la séquence d'exécution courante dévie de la propriété) dans le dispositif mémoire pour être rejoués éventuellement plus tard (si la séquence d'exécution satisfait la propriété à nouveau). Ainsi, un tel mécanisme est plus facile à implanter, car ils sont dotés d'un ensemble restreint d'états de contrôle. En conséquence, la liaison entre nos moniteurs d'enforcement et leurs implantations permet d'être établie formellement. Ce qui permet de donner plus de confiance dans l'implantation produite.

6.7 Conclusion

Dans ce chapitre, notre but était d'étendre les précédents travaux sur la vérification et l'enforcement de propriété à l'exécution. Nous avons proposés des notions génériques de moniteurs de vérification et d'enforcement. Les moniteurs d'enforcement sont basés sur un dispositif mémoire fini, des ensembles finis d'états de contrôle et d'opérations d'enforcement. La notion de moniteur d'enforcement généralise les moniteurs d'enforcement proposés précédemment : automates de sécurité (et en conséquence les Shallow History Automates) et edit-automates. De plus, nous avons montré que ces moniteurs pouvaient enforcer l'ensemble des propriétés déclarées comme enforçables au Chapitre 4 Section 4.3. Nous avons aussi montré comment il était possible de composer ces moniteurs d'enforcement (et les moniteurs de vérification, par un restriction immédiate). Nous avons également proposé deux techniques systématiques pour produire un moniteur de vérification ou d'enforcement depuis un automate de Streett reconnaissant une r-propriété donnée.

CHAPITRE 7

Approche test

Sommaire

7.1 Introduction **123**
7.2 Principe de l'approche **124**
7.3 Représentation algébrique des processus de test **125**
7.4 Phase de génération de tests **127**
7.5 Phase d'instanciation des tests **128**
7.6 Phase de sélection et d'exécution des tests **130**
7.7 Approche compositionnelle dirigée par la syntaxe **131**
7.7.1 Principe général 131
7.7.2 Génération des tests 132
7.8 Conclusion et perspectives **133**

Chapter abstract

This chapter presents an approach dedicated to property testing in which we do not use a complete behavioral specification of the implementation under test. From a Streett automaton recognizing a r-property, we synthesize a set of communicating test processes. Those processes are purposed to evaluate if a given relation holds between the sequences described by the r-property and the set of possible execution sequences of the implementation. Also, we present an overview of a compositional version of this approach which was previously proposed in [FFMR06, FFMR07a] (available in Appendix C).

Résumé du chapitre

Ce chapitre présente une approche de test de propriétés où nous n'utilisons pas de spécification comportementale du système testé. À partir d'un automate de Streett reconnaissant une r-propriété, nous générons un ensemble de processus de test communicant. Ces processus ont pour but d'évaluer si une certaine relation est vérifiée entre l'ensemble des séquences décrites par la r-propriété et l'ensemble des séquences que peut produire l'implantation sous test. Nous présentons également de manière succinte une version compositionnelle de cette approche de test qui avait été proposée précédemment dans [FFMR06, FFMR07a] (disponibles dans l'Annexe C).

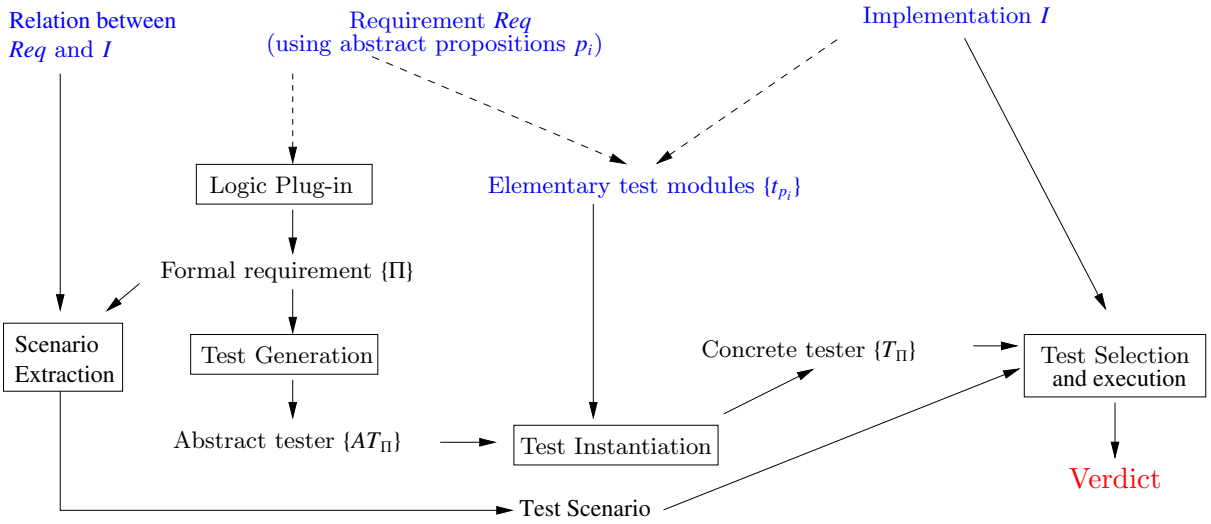


FIGURE 7.1 – Vue d'ensemble de l'approche de test

7.1 Introduction

Nous avons vu dans l'introduction que les méthodes de génération de tests de conformité boite noire reposaient souvent sur la disponibilité d'une spécification complète du système testé. Dans le Chapitre 2, nous avons identifié des situations où l'applicabilité de ces méthodes était difficile du fait de l'absence de telles spécifications. Nous proposons ici une approche alternative, pour produire une suite de tests, dédiée à la validation de propriétés fonctionnelles. Au lieu d'utiliser une spécification comportementale complète du système, nous utilisons une certaine forme de spécification partielle. Cette spécification partielle consiste en un ensemble d'exigences exprimées sur le comportement du système. L'exigence est construite à partir d'un ensemble de propositions abstraites décrivant des opérations (possiblement non atomiques) réalisables sur le système sous test. Un exemple typique de telle exigence peut être une politique de sécurité où les propositions abstraites pourraient dénoter des opérations de haut niveau comme "l'utilisateur A est authentifié", ou "le message M a été corrompu".

L'approche que nous proposons repose sur les considérations suivantes : une bonne connaissance des détails d'implantation est requise pour produire des modules de test capables de décider si de telles propositions sont satisfaites lors de l'exécution du logiciel. Ainsi, écrire les modules de test de ces propositions doit être laissé aux soins du programmeur (ou testeur) lorsqu'une spécification détaillée du système n'est pas disponible. Cependant l'orchestration correcte de l'exécution de ces "modules de test de base" et la combinaison de leur verdicts pour déduire une validité de la spécification dans son ensemble est plus facile à automatiser car elle dépend uniquement de la sémantique des opérateurs utilisés dans cette formule. Cette étape peut donc être réalisée par un générateur automatique de tests.

Cette génération peut même être faite, sous certaines conditions, de façon compositionelle (sur la syntaxe du formalisme utilisé) (voir Annexe C). Nous pensons que cette approche est assez générale pour être instanciée pour plusieurs formalismes logiques utilisés communément pour exprimer des exigences sur les traces d'exécution (*e.g.*, les expressions régulières ou la logique temporelle linéaire).

Il est important de noter ici la différence avec les approches de vérification et d'enforcement précédemment proposées. Les approches de vérification et d'enforcement supposaient le programme sous jacent instrumentable. Ainsi, nous pouvions supposer que l'ensemble des événements observables du programme et le vocabulaire de la propriété étaient identiques. Nous avons identifié des situations où cette hypothèses était difficilement réalisable en test. En conséquence, le "vocabulaire" de la propriété et l'ensemble des actions d'interaction avec l'implantation seront différents. Chaque proposition apparaissant dans la propriété testée s'évaluera par des séquences d'interactions exprimées dans un vocabulaire différent.

Vue d'ensemble de l'approche. L'approche que nous proposons peut être résumée comme suit (cf. Fig. 7.1) :

1. L'utilisateur doit fournir quatre entrées :

- une implantation logicielle sous test I (Implementation Under Test, IUT) ;
- une exigence informelle $\mathcal{R}$ définie à partir de propositions abstraites p_i ;
- la relation qu'il souhaite tester entre les séquences que l'implantation peut produire et les séquences d'exécution décrites par l'exigence informelle ;
- et un ensemble de modules de test élémentaires tp_i associés à chaque proposition p_i dédiée à l'implantation I.

2. L'utilisateur doit également exprimer l'exigence $\mathcal{R}$ par une expression formelle exprimée par une r-propriété Π (choisissant un formalisme d'expression adapté). Π est construite à partir des propositions abstraites p_i et un ensemble d'opérateurs dépendant du formalisme considéré.
3. A partir de la r-propriété Π, une fonction de génération de test produit automatiquement un testeur abstrait AT_Π. Ce testeur consiste en un driver de test et un ensemble de processus de test construits à partir des modules de test élémentaires Tp_i. Ainsi, AT_Π dépend uniquement de la r-propriété Π.
4. Un scénario de test est calculé à partir de l'exigence formelle Π et la relation entre Π et les séquence d'exécution de I qui doit être testée. Ce calcul du scénario se fait selon l'approche définie dans le Chapitre 4, Section 4.4.
5. Finalement, AT_Π est instancié en utilisant les modules de test élémentaires Tp_i pour obtenir un testeur concret T_Π pour la formule Π. Ce testeur concret est ensuite combiné avec un scénario de test (dédié à une relation entre les séquence de l'implantation et la propriété) pour être exécuté sur l'implantation sous test I et produire un verdict final pour la relation testée.

Organisation du chapitre. La suite de ce chapitre est organisée comme suit. Le principe de l'approche de test que nous proposons est présenté dans la Section 7.2. Dans la Section 7.3, nous proposons une algèbre de processus dédiée à l'expression des tests utilisés dans ce chapitre. Nous présentons la génération de test dans le cadre des r-propriétés de la classification *Safety-Progress* en Section 7.4. Ces tests sont destinés à évaluer une r-propriété sur une implantation sous test. Dans la Section 7.5, nous présentons l'instanciation d'un testeur abstrait en un testeur concret. Ensuite, dans la Section 7.6, nous présentons l'exécution de ces tests. À partir des tests obtenus lors de la phase de génération, nous savons établir un verdict pour les relations décrites au Chapitre 4 Section 4.4. Dans la Section 7.7, nous présentons une autre approche dirigée par la syntaxe pour la génération et l'exécution de cas de test. Enfin, dans la Section 7.8, nous concluons ce chapitre, et présentons les perspectives soulevées par l'approche proposée.

7.2 Principe de l'approche

Le but de l'approche de test est de produire un testeur concret T_Π (*i.e.*, un ensemble de processus de test communicants) associé à la r-propriété Π et qui sera utilisé pour tester une relation entre Π et les séquences d'exécution que l'implantation peut produire. Nous supposerons que la r-propriété Π sera construite à partir d'un ensemble de propositions $Prop$. Nous distinguons deux types de processus de test (dont la sémantique peut s'exprimer comme des LTSs) :

- *les modules de test* t_{p_i}, fournis par l'utilisateur, et associés aux propositions p_i de $Prop$ apparaissant dans Π. Leur but est de produire une évaluation indiquant si une exécution concrète, *i.e.*, une séquence de $(Act^t_{\text{ext}})^*$ satisfait p_i ou non. Chaque module de test t_{p_i} possède une "variable d'évaluation" v_{p_i} qui contient l'évaluation recherchée (true/false/unknown) par l'exécution de ce module de test.
- *le testeur abstrait* AT_Π, construit depuis une représentation de la r-propriété Π, et dépendant directement de la sémantique de Π. Son but est de contrôler l'exécution des processus de test associés aux propositions apparaissant dans Π à l'aide de signaux de base (start, stop, loop), et de collecter les évaluations produites par les processus de test t_{p_i} associés aux propositions p_i. Ce testeur est construit par la phase de génération de test (Section 7.4).
- *le testeur concret* T_Π, construit à partir d'un testeur abstrait et des modules de test. Leur but est de produire une évaluation de la r-propriété sur une exécution concrète, *i.e.*, une séquence de $(Act^t_{\text{ext}})^*$. Ce testeur concret est construit par la phase d'instanciation des tests (Section 7.5).

L'approche que nous proposons fait les hypothèses suivantes :

- les modules de test peuvent être exécutés à tout moment sur l'IUT ;
- chaque exécution d'un module de test produit une évaluation dans $\{true, false, unknown\}$ en temps fini.

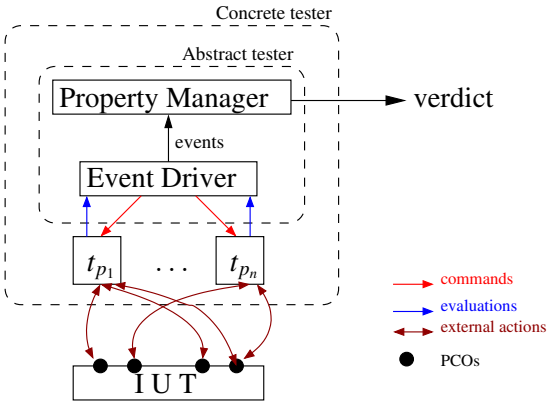


FIGURE 7.2 – Architecture de test

Architecture de test. L'architecture de test peut être présentée informellement comme suit (cf Fig. 7.2) : Le testeur abstrait est un processus constitué de la composition d'un processus "property manager" et "event driver". Le processus "property manager" a le rôle d'oracle pour la propriété testée. Il reçoit des événements du processus "event driver". Le testeur concret est constitué de la composition entre le testeur abstrait et l'ensemble des modules de test. Le processus "event driver" envoie des commandes d'exécution aux modules de test (démarrage, arrêt). Les modules de test évaluent les propositions en interagissant avec l'implantation sous test. Ces interactions sont des séquences d'actions externes (de Act_{ext}) aux points de contrôle et d'observation de l'implantation (PCOs).

7.3 Représentation algébrique des processus de test

Afin de pouvoir exprimer les tests de manière compositionnelle, nous exprimons les processus de test dans une notation algébrique. Nous utilisons ici une algèbre dédiée, introduite dans [FFMR06], mais d'autres algèbres de processus peuvent aussi bien être utilisées.

Syntaxe

Soit Act un ensemble d'*actions*, $\mathcal{T}$ un ensemble de *types* (avec $\tau \in \mathcal{T}$), Var un ensemble de *variables* (avec $x \in Var$), et Val un ensemble de *valeurs* (l'union des valeurs de types $\mathcal{T}$). Nous notons $expr_\tau$ (resp. x_τ) toute expression (resp. variable) de type τ. En particulier, nous supposons l'existence d'un type spécial $Eval$ dont les valeurs associées sont celles de l'ensemble $\{true, false, unknown\}$ et qui est utilisé pour noter les évaluations produites lors de l'exécution des tests associés aux propositions de $Prop$. La syntaxe d'un processus de test t est donnée par la grammaire suivante :

$$\begin{array}{lcl} t & ::= & [b]\,\gamma \circ t \mid t + t \mid nil \mid recX\ t \mid X \\ b & ::= & \text{true} \mid \text{false} \mid b \vee b \mid b \wedge b \mid \neg b \mid expr_\tau = expr_\tau \\ \gamma & ::= & x_\tau := expr_\tau \mid !c(expr_\tau) \mid ?c(x_\tau) \mid act \in Act^t_{ext} \end{array}$$

Dans cette grammaire, t dénote un test de base (nil étant le testeur vide ne réalisant rien), b une expression booléenne, c un nom de canal, γ une action, $\circ$ est l'opérateur de *préfixage*, $+$ l'opérateur de *choix*, X une variable de terme, $recX$ permettant de définir des processus *récursifs* (avec X une variable de terme) [1]. Lorsque la condition b est vraie, nous abrégeons $[true]\gamma$ par γ. Les actions atomiques réalisées par les testeurs de base sont soit des assignations internes ($x_\tau := expr_\tau$), l'émission de valeurs ($!c(expr_\tau)$), la réception de valeurs ($?c(x_\tau)$) entre processus de test sur un canal c [2], ou une action externe vers l'IUT $act \in Act^t_{ext}$.

1. Nous considérerons seulement les termes *clos* : chaque occurrence de X est liée à $recX$.
2. Pour simplifier le calcul, nous avons supposé que tous les canaux échangent une valeur. Dans les testeurs, nous utilisons aussi des "canaux de synchronisation" qui n'échangent pas de valeur, ce qui est une extension directe.

$$\frac{\gamma \in Act}{[b]\gamma \circ t \overset{[b]\gamma}{\rightharpoonup} t}\ (\circ) \qquad \frac{t[recX \circ t/X] \overset{[b]\gamma}{\rightharpoonup} t' \qquad \gamma \in Act}{recX \circ t \overset{[b]\gamma}{\rightharpoonup} t'}\ (rec)$$

$$\frac{\gamma \in Act \qquad t_1 \overset{[b]\gamma}{\rightharpoonup} t'_1}{t_1 + t_2 \overset{[b]\gamma}{\rightharpoonup} t'_2}\ (+) \qquad \frac{\gamma \in Act \qquad t_2 \overset{[b]\gamma}{\rightharpoonup} t'_2}{t_1 + t_2 \overset{[b]\gamma}{\rightharpoonup} t'_2}\ (+)$$

FIGURE 7.3 – Règles pour la réécriture de termes

$$\frac{\rho(expr_\tau) = v \qquad t \overset{[b]x_\tau := expr_\tau}{\rightharpoonup} t' \qquad \rho(b) = true}{(t,\rho) \overset{x_\tau := v}{\longrightarrow} (t', \rho[v/x_\tau])}\ (:=)$$

$$\frac{\rho(expr_\tau) = v \qquad t \overset{[b]!c(expr_\tau)}{\rightharpoonup} t' \qquad \rho(b) = true}{(t,\rho) \overset{!c(v)}{\longrightarrow} (t', \rho)}\ (!)$$

$$\frac{v \in Dom(\tau) \qquad t \overset{[b]?c(x_\tau)}{\rightharpoonup} t' \qquad \rho(b) = true}{(t,\rho) \overset{!c(v)}{\longrightarrow} (t, \rho[v/x_\tau])}\ (?)$$

FIGURE 7.4 – Règles de la modification de l'environnement

Sémantique

Nous donnons une sémantique aux testeurs de base (t) en utilisant des règles de réécriture entre les termes non interprétés dans un style ressemblant à l'algèbre CCS (cf. Fig. 7.3).

La sémantique d'un processus testeur de base t est ainsi donnée par un LTS $S_t = (Q^t, Act^t, T^t, q_0^t)$ de manière classique : les états Q^t sont les "configurations" de la forme (t, ρ), où t est un terme et $\rho : Var \to Val$ est un *environment*. Les états et les transitions de S_t (relation $\longrightarrow$) sont les plus petits ensembles définis par les règles données sur la Fig. 7.4 (en utilisant une relation auxiliaire $\rightharpoonup$ définie sur la Fig. 7.3). L'état initial q_0^t de S est la configuration (t_0, ρ_0), où ρ_0 lie toutes les variables vers une valeur indéfinie. Finalement, notons que les actions Act^t de S_t sont soit étiquetées par des assignations internes ($x_\tau := v$) soit des émissions externes ($!c(v)$). Dans la suite, nous utilisons les notations suivantes :

- $Act^t_{\text{ext}} \subseteq Act^t$ désigne les actions externes aux testeurs vers l'IUT. Classiquement, cet ensemble d'actions comporte des actions contrôlables par le testeur (les entrées de l'IUT) et les actions observables (les sorties de l'IUT).
- $Act^t_{\text{int}} \subseteq Act^t$ désigne les actions internes du testeur (*e.g.*, les assignations de variables internes).
- $Act^t_{\text{comI}} \subseteq Act^t$ désigne les actions de communications internes du testeur de type réception.
- $Act^t_{\text{comO}} \subseteq Act^t$ désigne les actions de communication internes de type émission.

L'ensemble Act^t des actions du processus de test t est donc partitionné, et nous avons $A^t = Act^t_{\text{ext}} \cup Act^t_{\text{int}} \cup Act^t_{\text{comI}} \cup Act^t_{\text{comO}}$.

Les testeurs plus complexes sont obtenus par la composition parallèle de processus de test sur un ensemble de canaux cs (opérateur $\|_{cs}$), ou en utilisant ce que nous appelons l'opérateur "d'exception jointe" ($\ltimes^{\mathcal{I}}$), autorisant l'arrêt d'un processus de test lors de la réception d'une communication sur l'un des canaux d'interruption de $\mathcal{I}$. Nous notons $\|$ pour $\|_\emptyset$ et $\mathsf{Act_chan}(s)$ l'ensemble des actions possibles en utilisant un canal de l'ensemble s.

Pour introduire la communication dans notre sémantique, nous donnons deux ensembles de règles spécifiant la manière dont les LTS sont composés relativement aux opérateurs de communication ($\|_{cs}, \ltimes^{\mathcal{I}}$). Ces règles sont destinées à maintenir une exécution asynchrone, et une communication par *rendez-vous*. Soient $S^t_i = (Q^t_i, A^t_i, T^t_i, q^t_{0i})$ deux LTSs modélisant le comportement de deux processus t_1 et t_2, nous définissons le LTS $S = (Q, A, T, q_0)$ modélisant les comportements de $S^t_1 \|_{cs} S^t_2$ (resp. $S^t_1 \ltimes^{\mathcal{I}} S^t_2$) comme le produit de S^t_1 et S^t_2 où $Q \subseteq (Q^t_1 \cup \{\bot\}) \times Q^t_2$ et les règles de transitions sont données sur la Fig. 7.5 (resp. sur la Fig. 7.6).

$$\frac{p_1 \xrightarrow{a} p_1' \qquad a \notin \text{Act_chan}(cs)}{(p_1, p_2) \xrightarrow{a} (p_1', p_2)} \; (\overline{\|^{l}_{cs}}) \qquad \frac{p_2 \xrightarrow{a} p_2' \qquad a \notin \text{Act_chan}(cs)}{(p_1, p_2) \xrightarrow{a} (p_1, p_2')} \; (\overline{\|^{r}_{cs}})$$

$$\frac{p_1 \xrightarrow{a} p_1' \qquad p_2 \xrightarrow{a} p_2' \qquad a \in \text{Act_chan}(cs)}{(p_1, p_2) \xrightarrow{a} (p_1', p_2')} \; \|_{cs}$$

FIGURE 7.5 – Composition de LTSs par rapport à $\|_{cs}$

$$\frac{p_1 \xrightarrow{a} p_1' \qquad a \notin \text{Act_chan}(\mathcal{I})}{(p_1, p_2) \xrightarrow{a} (p_1', p_2)} \; (\overline{\ltimes^{\mathcal{I}}}) \qquad \frac{p_2 \xrightarrow{a} p_2' \qquad a \in \text{Act_chan}(\mathcal{I}))}{(p_1, p_2) \xrightarrow{a} (\bot, p_2')} \; (\ltimes^{\mathcal{I}})$$

FIGURE 7.6 – Composition de LTSs par rapport à $\ltimes^{\mathcal{I}}$

7.4 Phase de génération de tests

Nous décrivons maintenant le principe de la génération de tests exprimés dans l'algèbre présentée dans la Section 7.3. Ces tests sont dédiés à l'évaluation d'une r-propriété sur une séquence d'exécution de l'implémentation sous test.

La phase de génération de tests peut être formalisée par une fonction appelée *GenTest*, produisant un testeur abstrait (un processus de test) à partir d'une r-propriété Π ($GenTest(\Pi) = AT_{\Pi}$). Pour simplifier sa définition, nous supposons que la r-propriété Π est représentée par son automate de Streett $\mathcal{A}_{\Pi}$. L'automate $\mathcal{A}_{\Pi}$ est défini sur un vocabulaire Σ dont les événements sont évalués grâce aux propositions $p_i \in Prop$. Nous nous plaçons dans le cas général où $\Sigma = 2^{Prop}$. À chaque événement de l'alphabet Σ correspond un ensemble de propositions de *Prop*. Ainsi, nous évaluerons un événement comme vrai si l'ensemble des propositions qui lui sont associées sont évaluées comme vrai. Les autres propositions devront être évaluées comme fausse. L'évaluation des propositions p_i de *Prop* est assurée d'une part par l'exécution des modules de test t_{p_i} associés aux propositions p_i et par la coordination par un processus "conducteur de test" qui dirige l'exécution des processus t_{p_i}. Cette association entre les événements de l'automate et les propositions de la r-propriété est illustrée sur l'Exemple 29.

Nous définissons donc la fonction *GenTest* qui génère, à partir d'un automate de Streett $\mathcal{A}_{\Pi}$ un processus qui peut conduire et coordonner l'exécution des différents modules de test correspondant aux événements qui apparaissent dans le vocabulaire de $\mathcal{A}_{\Pi}$.

DÉFINITION 55 (FONCTION $GenTest(\Pi)$). Soit Π une r-propriété définie à partir d'un ensemble de propositions *Prop* et reconnue par un automate de Streett $\mathcal{A}_{\Pi}$. La fonction $GenTest(\cdot)$ génère un processus "conducteur de tests" (test driver), un LTS AT_{Π}, à partir de Π, de l'ensemble *Prop* de ses propositions, et du vocabulaire Σ de $\mathcal{A}_{\Pi}$. $GenTest(\Pi)$ est le processus de test dont la sémantique peut être définie par la composition de LTSs :

$$PM \|_{cs_{\Sigma}} ED(Prop, \Sigma, cs_{Prop})$$

où *PM* (Property Manager) a le rôle d'oracle pour la r-propriété Π et *ED* (Event Driver) conduit l'exécution des tests associés aux événements. Les processus *PM* et *ED* sont définis comme suit :

- $cs_{\Sigma} = \{e \mid e \in \Sigma\}$ est l'ensemble des canaux (channel set) utilisés pour faire communiquer *PM* et *ED*. Ces canaux portent les noms des événements de Σ.
- $cs_{Prop} = \bigcup_{p_i \in Prop} \{c_start_{p_i}, c_stop_{p_i}, c_ver_{p_i}\}$.
- *PM* est le LTS $(Q^{PM}, {q_{\text{init}}}^{PM}, Act^{PM}, \rightarrow_{PM})$, associé à la r-propriété Π et défini à partir de l'automate de Streett $\mathcal{A}_{\Pi}$ comme suit :
  - $Q^{PM} = Q^{\mathcal{A}_{\Pi}}$
  - ${q_{\text{init}}}^{PM} = {q_{\text{init}}}^{\mathcal{A}_{\Pi}}$
  - $Act^{PM} = Act_I^{PM}$ avec $Act_I^{PM} \subseteq Act_{comI}$ tel que $Act_I^{PM} = \{?e() \mid e \in \Sigma\}$
  - $\rightarrow_{PM}$ est définie par $\cup\{(q, ?e(), q') \mid q, q' \in Q^{PM} \wedge (q, e, q') \in \rightarrow_{\mathcal{A}_{\Pi}}\}$

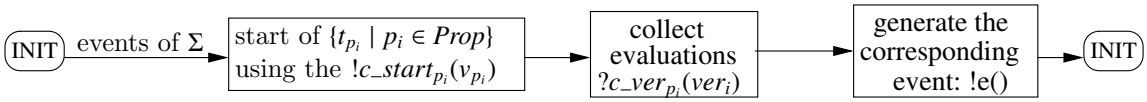


FIGURE 7.7 – Le processus $ED(Prop, \Sigma, cs_{Prop})$

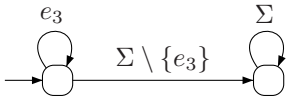


FIGURE 7.8 – L'automate $\mathcal{A}_{\Pi_1}$ reconnaissant la r-propriété correspondant à la formule $\Box\,(p_1 \wedge p_2)$

- $ED(Prop, \Sigma, cs_{Prop})$ est le LTS représenté de façon schématique sur la Fig. 7.7. À partir de la réception de l'ordre d'évaluation d'un événement de l'alphabet Σ, il démarre l'exécution de l'ensemble des modules de tests pour évaluer l'événement reçu. Il attend ensuite l'ensemble des évaluations qui doivent être produites par les modules de test t_{p_i} en utilisant les actions $?c_ver_{p_i}(ver_i)$ où le canal $c_ver_{p_i}$ est utilisé pour communiquer avec le processus t_{p_i} et la variable ver_i est utilisée pour mémoriser l'évaluation de l'exécution de t_{p_i}.

Le processus PM associé à une r-propriété Π joue le rôle d'oracle pour la r-propriété. Il suit la même structure que l'automate de Streett $\mathcal{A}_\Pi$. Dans chaque état, le processus PM attend l'événement produit par le processus ED. Le processus ED sera combiné avec un scénario de test lors de la phase de sélection et d'exécution des test (Section 7.6). La combinaison avec le scénario permettra de tester une séquence d'événements.

Exemple 29 (Testeur abstrait obtenu à partir de *GenTest*) Considérons un ensemble de propositions $Prop_1 = \{p_1, p_2\}$ et l'ensemble des événements associés $\Sigma_1 = 2^{Prop_1} = \{\{\emptyset\}, \{p_1\}, \{p_2\}, \{p_1, p_2\}\} = \{e_0, e_1, e_2, e_3\}$. Soit Π_1 la r-propriété exprimée en LTL par la formule $\Box\,(p_1 \wedge p_2)$. L'automate de Streett $\mathcal{A}_{\Pi 1}$ reconnaissant Π_1 est représenté sur la Fig. 7.8. Le processus $PM(\mathcal{A}_{\Pi 1})$ généré à partir de $\mathcal{A}_{\Pi 1}$ est représenté sur la Fig. 7.9. Le processus ED correspondant à l'ensemble de propositions $Prop_1$ et l'alphabet Σ_1 est représenté sur la Fig. 7.10. Par exemple, pour l'évaluation de l'événement e_1, il est nécessaire d'évaluer p_1 comme vrai et p_2 comme faux. Ainsi, le processus t_{p_1} (resp. t_{p_2}) est démarré avec pour directive d'évaluation vrai (resp. faux). Ensuite, l'évaluation des exécutions des deux modules de test sont collectées, de façon non-déterministe, grâce aux deux actions $c_ver_{p_1}(ver_1)$ et $c_ver_{p_2}(ver_2)$. Selon les évaluations reçues, l'événement correspondant est émis (pour le processus $PM(\mathcal{A}_{\Pi 1})$).

7.5 Phase d'instanciation des tests

La phase d'instantiation des tests a pour but de combiner le processus généré par la phase de génération (testeur abstrait) avec l'ensemble des modules de test correspondant aux propositions apparaissant dans la propriété utilisée pour générer le testeur abstrait.

DÉFINITION 56 (FONCTION D'INSTANTIATION DES TESTS *TestInstantiation*). Étant donnée une r-propriété Π spécifiée par un automate de Streett $\mathcal{A}_\Pi$, le test T_Π que nous générons pour Π est :

$$T_\Pi = TestInstantiation(\Pi) = \Big(Test(p_1, cs_1) \parallel \cdots \parallel Test(p_n, cs_n)\Big) \parallel_{cs} AT_\Pi$$

où

- $p_1, \ldots, p_n$ sont les propositions de $Prop$ apparaissant dans Π ;
- cs_i est un ensemble de noms de canaux frais utilisés par le processus $Test_Driver(\mathcal{A}_\Pi)$ pour communiquer avec le processus $Test(p_i, cs_i)$, avec $cs_i = \{c_start_{p_i}, c_stop_{p_i}, c_loop_{p_i}, c_ver_{p_i}\}$;
- cs est l'ensemble des noms de canaux utilisés par le processus $Test_Driver(\mathcal{A}_\Pi)$ pour communiquer avec les processus $Test(p_i, cs_i)$ pour $i = 1, \ldots, n$, *i.e.*, $cs = \bigcup_{i=1}^{n} cs_i$.

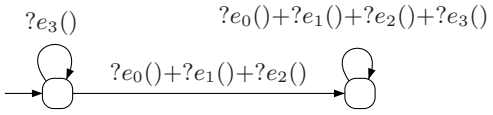


FIGURE 7.9 – Le processus Property Manager correspondant à l'automate $\mathcal{A}_{\Pi_1}$

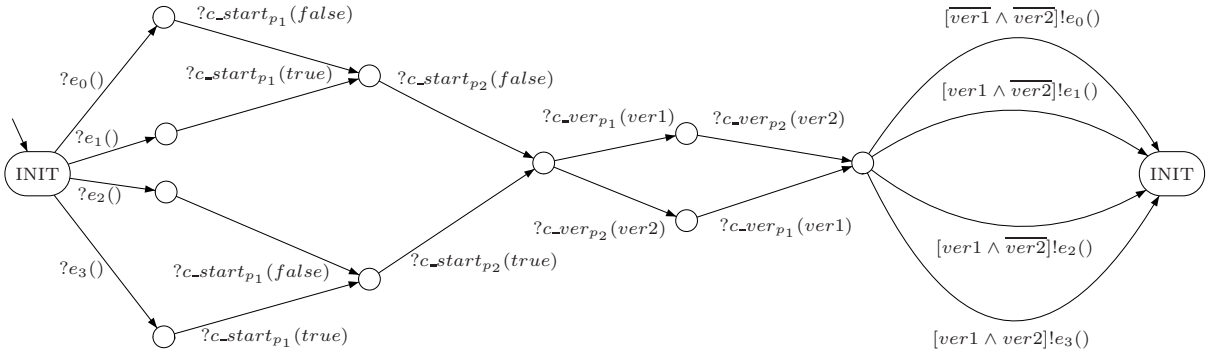


FIGURE 7.10 – Le processus ED correspondant à l'ensemble de propositions $Prop_1$ et l'alphabet Σ_1

– Le processus AT_Π, généré à partir d'un automate de Streett $\mathcal{A}_\Pi$, est le processus qui peut conduire et coordonner l'exécution des différents modules de test correspondant aux événements qui apparaissent dans le vocabulaire de $\mathcal{A}_\Pi$. Ce processus est obtenu à partir de la phase de génération de test.

La fonction *TestInstantiation* utilise la fonction intermédiaire *Test*. Cette fonction est utilisée pour générer un processus de test t_p à partir d'une proposition p et d'un ensemble de canaux cs. Les noms de canaux fournis dans cs sont les canaux à utiliser par le processus de test résultat t_p afin d'être contrôlable par le processus driver de test.

DÉFINITION 57 (FONCTION *Test* DÉDIÉE AUX MODULES DE TEST). La fonction $Test(\cdot,\cdot)$ est définie comme suit :

$$Test(t_p, \{c_start_p, c_stop_p, c_loop_p, c_ver_p\}) \stackrel{def}{=}$$
$$recX\Big(?c_start_p(v_p) \circ t_p \circ !c_ver_p(ver) \circ ?c_loop_p() \circ X\Big) \ltimes^{\{c_stop_p\}} (?c_stop_p() \circ nil)$$

où t_p est le module de test correspondant à la proposition p et $\{c_start_p, c_stop_p, c_loop_p, c_ver_p\}$ l'ensemble des canaux utilisés pour rendre le processus t_p contrôlable.

La fonction *Test* appliquée à :

– un module de test t_p dont la variable représentant l'évaluation est v_p
– et un ensemble de noms de canaux $cs = \{c_start_p, c_stop_p, c_loop_p, c_ver_p\}$

"instrumente" le module t_p de manière à le rendre contrôlable à travers les canaux de communication de l'ensemble cs. L'action $?c_start_p(v_p)$ permet au module de test d'être démarré lors de la réception

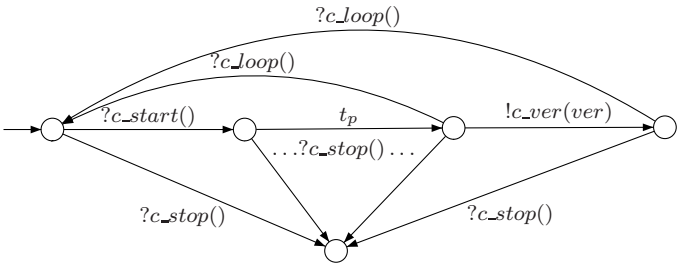


FIGURE 7.11 – Instantiation d'un module de test t_p par la fonction *Test*

d'un signal sur le canal c_start_p. De plus, une valeur dans true/false/unknown peut lui être transmise, initialisant sa variable v_p. L'action $!c_ver_p(ver)$ permet d'émettre le verdict *ver* sur le canal c_ver_p lors de la fin de l'exécution du module de test. L'action $?c_loop_p()$ permet de redémarrer le module de test à la fin de son exécution. Enfin, l'exécution du module de test peut être arrêtée à tout moment en utilisant l'action $?c_stop_p()$. Cette action est possible dans chaque état grâce à l'opérateur $\ltimes^{\{c_stop_p\}}$ suivi de l'action $?c_stop_p()$. Sur la Fig. 7.11 nous illustrons l'instanciation d'un module de test t_p.

7.6 Phase de sélection et d'exécution des tests

D'un point de vue formel, les séquences d'exécution du test sont les séquences d'exécution d'une composition parallèle entre le LTS S modélisant le comportement de l'IUT et le cas de test T_Π, avec un synchronisation par "rendez-vous" sur les actions visibles apparaissant dans T_Π, *i.e.*, les actions de $Act_{\text{ext}}^{T_\Pi}$.

Cependant, ce produit de LTSs peut contenir plusieurs exécutions possibles dues au non déterminisme du processus de test produit. Les sources de non-déterminisme sont les suivantes :

- Les modules de test sont non-déterministes. Ils peuvent en effet contenir plusieurs exécutions pour une même évaluation d'un événement.
- La composition parallèle engendre du non-déterminisme.
- Le test driver contient un nombre potentiellement infini d'exécutions menant à une évaluation donnée de la propriété à partir de laquelle il a été généré.

De plus les phases de génération et d'instanciation de test assurent simplement que l'évaluation produite par l'exécution du test est correcte par rapport à la r-propriété Π. La phase de génération n'aide pas à sélectionner les exécutions de test *intéressantes* qui peuvent exhiber les comportements amenant un verdict pour la relation entre les séquences de l'IUT et la r-propriété.

Scénario de test. Pour résoudre ce problème, nous proposons d'introduire la notion de scénario de test qui va aider le conducteur de test à guider l'exécution du cas de test. Intuitivement, un scénario de test est une séquence d'exécution dans l'automate de Streett amenant vers une satisfaction recherchée pour la propriété considérée. Le scénario de test construit à partir de $\sigma \in \Sigma^*$ est simplement la transformation de chaque événement de la séquence en une action de communication du testeur. Chacune de ces actions de communication est de type émission. Ainsi, l'exécution conjointe du scénario de test avec le processus *ED* entraîne la sélection d'un chemin dans le processus *PM* correspondant à l'automate de la r-propriété considérée.

Nous construisons maintenant l'ensemble des scénarios de test pour tester une relation entre une r-propriété et une IUT. Comme vu dans le Chapitre 4 Section 4.4, les séquences d'exécution intéressantes à tester sont celles qui, selon la relation considérée, amènent à déterminer la r-propriété de façon positive ou négative. Intuitivement, les séquences de scénarios de tests essayent d'amener l'automate de Streett sous-jacent de la r-propriété dans un état montrant la détermination négative ou positive. Ceux-ci peuvent être générés apriori en fonction de la relation testée.

Nous définissons une fonction intermédiaire *send* prenant une séquence d'événements du vocabulaire Σ de la r-propriété et produisant un processus de test exécutant une séquence d'actions de communication de type émission utilisant comme noms de canaux le nom des événements apparaissant dans la séquence :

Définition 58 (Fonction *send*). La séquence d'action de type émission destinée au processus "Event Driver" est produite par la fonction $send(\cdot)$. Elle est construite à partir d'une séquence d'exécution $\sigma \in \Sigma^*$, *t.q.* $\sigma = \sigma_0 \cdots \sigma_n$, et est définie comme suit :

$$send(\sigma) = !\sigma_0 \cdots !\sigma_n$$

Il est possible d'exprimer l'ensemble des jeux de test pour une r-propriété Π et une relation donnée à partir de la condition de testabilité. En effet, pour chaque relation, celle-ci peut se caractériser sur un automate de Streett (cf. Chapitre 5, Section 5.2).

Définition 59 (Scénario de tests). Pour une r-propriété Π, reconnue par un automate de Streett $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \rightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$, les suites de test que nous produisons sont les suivantes :

Pour la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$: $\{scenario(\sigma) \mid q_{\text{init}}{}^{\mathcal{A}_\Pi} \xrightarrow{\sigma} q \wedge q \in Bad^{\mathcal{A}_\Pi}\}$
Pour la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$: $\{scenario(\sigma) \mid q_{\text{init}}{}^{\mathcal{A}_\Pi} \xrightarrow{\sigma} q \wedge q \in Good^{\mathcal{A}_\Pi}\}$

Remarque 10 (Choix des scénarios de test en pratique) Chaque ensemble de scénarios de test est potentiellement infini. Il est facile en pratique, en utilisant l'automate de Streett de la propriété :
- d'imposer des contraintes sur les états qui doivent être visités ;
- de se restreindre à des jeux dépendant d'un paramètre de taille (*e.g.*, plus long ou plus court qu'une taille fixée, etc...).

Il est possible également de combiner les séquences de scénario, selon leurs préfixes, pour exprimer un scénario de test sous forme arborescente. *

Verdict produit. Pour produire le verdict, il nous suffit de "décorer les états" du processus Property Manager selon les états *Good* et *Bad* de l'automate de Streett utilisé pour le générer. Par exemple, pour la relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$, nous décorons l'ensemble des états *Bad* par le verdict *fail* et tous les autres états par le verdict *inconc*.

Le scénario couplé au driver de test guide donc l'exécution du test. Le verdict est obtenu si on arrive à atteindre la fin du scénario sans obtenir d'évaluation *unknown* lors de l'exécution des modules de test.

Le verdict de l'exécution du test guidée par un scénario est déterminé en pratique comme suit :
- si l'exécution de l'un des modules produit un résultat inconclusif, le verdict global du test est inconclusif ;
- sinon, on décide du verdict en fonction du dernier état atteint dans le processus Property Manager. L'état atteint peut se relier à un état de l'automate de Streett.

Remarque 11 (Verdicts faibles) Nous avions vu dans le Chapitre 4 Section 4.4.3 qu'il était possible de produire des verdicts faibles pour les relations étudiées. Il suffit de décorer les états des automates de Streett, selon les situations où un verdict faible est possible, en suivant le même raisonnement que celui utilisé pour les verdicts "forts". *

7.7 Approche compositionnelle dirigée par la syntaxe

Nous proposons une version *compositionnelle* de l'approche présentée dans les sections précédentes. Nous présentons de manière succincte cette approche et renvoyons à [FFMR06, FFMR07a, FMFR08b] (disponibles dans les sections suivantes) pour une présentation plus complète.

L'approche présentée dans les sections précédentes avait été développée pour une variante de la logique temporelle linéaire et les expressions régulières étendues. La synthèse de tests à partir d'automates de Streett peut être vue comme une généralisation de l'approche présentée ici dans le sens où les automates de Streett sont un formalisme général de description de propriétés.

Contrairement à l'approche précédente (dédiée au test d'une relation entre les séquences d'exécution qu'une implantation peut produire et les séquences d'une propriété), l'approche compositionnelle que nous avions proposée visait l'évaluation d'une propriété sur une implantation logicielle. En conséquence, les tests que nous produisions étaient associés à des verdicts différents : ces verdicts représentaient la satisfaction par l'implantation de la propriété considérée.

7.7.1 Principe général

Dans l'approche de test compositionnelle, nous considérons que les propriétés à tester sont exprimées dans un formalisme logique $\mathscr{L}$. Les formules de $\mathscr{L}$ sont construites à partir d'un ensemble fini d' *opérateurs n-aires* F^n et un ensemble de *prédicats finis* $\{p_1, p_2, \ldots, p_n\}$. La syntaxe abstraite de cette logique peut être définie comme suit :

$$\text{formula} ::= F^n(\text{formula}_1, \text{formula}_2, \ldots, \text{formula}_n) \mid p_i.$$

Les formules de $\mathscr{L}$ sont interprétées sur des séquences d'exécution finies. Cependant, cette sémantique possède également deux caractéristiques importantes :

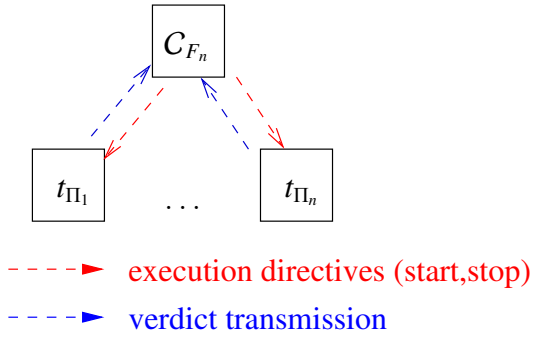


FIGURE 7.12 – Illustration du principe de la fonction *GenTest* compositionelle

- D'abord, cette sémantique est définie à deux niveaux. Les prédicats ne sont pas atomiques, *i.e.*, ils ne correspondent pas à l'occurrence de simple actions visibles, mais plutôt à des séquences d'actions visibles concrètes. Les opérateurs F^n sont ensuite interpretés sur les séquences d'exécution *abstraites*, *i.e.*, des séquences de prédicats.
- Ensuite, comme l'un des objectifs est de tester la validité ou la non-validité d'une r-propriété Π, la sémantique de Π définit trois sortes de séquences d'exécution, correspondant aux verdicts possibles qu'un testeur peut délivrer : celles qui satisfont Π (pass), celles qui ne satisfont pas Π (fail), et celles pour lesquelles nous ne pouvons pas conclure sur la satisfiabilité de Π (inconc).

Plus formellement, un triplet de langages finis $(L^P_{p_i}, L^F_{p_i}, L^I_{p_i})$ est associé à chaque prédicat p_i. Ces trois langages définissent respectivement les séquences d'exécution concrètes qui satisfont p_i, qui ne satisfont pas p_i, et celles pour lesquelles la satisfiabilité de p_i est inconnue. Les trois hypothèses suivantes sont nécessaires :

- $L^P_{p_i}$, $L^F_{p_i}$ et $L^I_{p_i}$ sont définis sur un alphabet $A_{p_i} \subseteq Act^t_{\text{ext}}$. Intuitivement, A_{p_i} est l'ensemble des actions visibles dont l'occurrence influence la valeur de vérité de p_i.
- Cet ensemble de trois langages définit une *partition* de $(A_{p_i})^*$.
- Pour deux prédicats distincts p_i, p_j, A_{p_i} et A_{p_j} sont disjoints.

La sémantique d'une formule non-atomique $\Pi(p_1, p_2, \ldots, p_n)$ est ainsi définie par trois ensembles $[\![\Pi]\!]^P$, $[\![\Pi]\!]^F$, $[\![\Pi]\!]^I$, calculés inductivement à partir de $L^P_{p_i}$, $L^F_{p_i}$ et $L^I_{p_i}$ pour chaque p_i apparaissant dans Π.

7.7.2 Génération des tests

Le but de la phase de génération [FFMR07a] est de produire un cas de test T_Π (*i.e.*, un ensemble de processus de test communicant) associé à la formule Π de $\mathscr{L}$ qui est testée.

Nous distinguons deux sortes de processus de test (qui sont tous les deux des LTSs) :

- Les *modules de test* t_{p_i}, fournis par l'utilisateur, et associés aux prédicats atomiques p_i de Π. Leur but est de produire un verdict pour le test indiquant si une séquence d'exécution concrète appartient à $L^P_{p_i}$, $L^F_{p_i}$ ou $L^I_{p_i}$. Des exemples de tels modules de test sont donnés dans l'Annexe C.
- Les *contrôleurs de test* t_{F^n}, associés à chaque opérateur n-aire F^n de la logique $\mathscr{L}$. Leur but est de contrôler l'exécution des processus de test associés à leurs opérandes en utilisant des signaux de base (start, stop, loop), et de collecter leurs verdicts dans le but de produire un verdict correspondant à cette instance de l'opérateur F^n. Nous donnons des exemples de contrôleurs pour plusieurs logiques dans le rapport de recherche [FFMR07b].

Cette technique de génération de test peut être formalisée par une fonction appelée $GenTest_{\mathscr{L}}$, telle que $GenTest_{\mathscr{L}}(\Pi) = T_\Pi$. Cette fonction est inductivement définie sur la syntaxe de $\mathscr{L}$ de la manière suivante (voir Fig. 7.12) :

- Si $\Pi = p_i$, alors $GenTest_{\mathscr{L}}(\Pi)$ retourne le module de test t_{p_i} (associé au prédicat p_i) étendu avec les opérations de communication nécessaires pour le rendre contrôlable par un autre processus de test (voir Fig. 7.11, p. 129).
- Si $\Pi = F^n(\Pi_1, \cdots, \Pi_n)$, alors $GenTest_{\mathscr{L}}(\Pi)$ retourne la composition parallèle entre les processus de test définis récursivement t_{Π_1}, ..., t_{Π_n} et une instance d'un contrôleur de test générique t_{F^n}.

Finalement, un processus de test spécial t_{main} est ajouté pour lancer l'exécution globale et collecter le verdict final. Selon cette technique de génération de test, l'architecture d'un cas de tes T_Π correspondant

exactement à l'arbre syntaxique abstrait de la formule Π : la racine est t_{main}, les feuilles sont des modules de test correspondant aux prédicats p_i de Π, et les noeuds intermédiaires sont les contrôleurs associés avec les opérateurs de Π (voir [FFMR07b] ou Annexe C pour un exemple).

Des contrôleurs de test et des algorithmes de génération de test ont été définis pour différents formalismes de spécification. Pour l'instant nous avons étudié cette approche pour deux formalismes généraux :

- Les logiques temporelles [Pnu77] comme LTL sont fréquemment utilisées par la communauté de la vérification pour exprimer les exigences sur les systèmes réactifs. Nous considérons ici un fragment de cette logique dont les modèles sont des ensembles de séquences d'exécution finis. Nos études de cas ont montré que plusieurs concepts des politiques de contrôles d'accès peuvent s'intégrer dans le cadre de cette logique.
- Les expressions régulières étendues [Kle51] sont un autre formalisme pour définir des motifs de comportement finis. Elles sont répandues et bien comprises des ingénieurs. Nous prenons en compte l'opérateur de négation de manière à pouvoir exprimer les comportements non désirés sur l'implantation.

Les sémantiques des variantes des formalismes utiisées sont définis dans [FFMR07a] et [FFMR07b].

7.8 Conclusion et perspectives

Conclusion. Nous avons proposé une approche générale pour le test de relations entre une r-propriété et l'ensemble des séquences d'exécution qu'une implantation sous test peut produire. Nous générons un ensemble de tests communicants exprimés dans une algèbre de processus dédiée. Ceci nous permet de séparer l'évaluation de la propriété sous-jacente, la production du verdict, de la mécanique d'exécution des tests. La déconnection entre "orchestration" de l'exécution et évaluation de la propriété a une motivation pratique. En effet, nous pensons que cette "orchestration" dépend uniquement de la sémantique de la r-propriété et peut donc être entièrement automatisée. Les modules de test, quant à eux, dépendent de l'implantation. Leur écriture nécessite l'utilisation d'une spécification.

Perspectives. Plusieurs perspectives sont soulevées par ce travail. Tout d'abord, il semble nécessaire de mieux prendre en compte la contrôlabilité et l'observabilité des actions de l'implantation pour la génération des scénarios de test. Ceci pourrait se faire en distinguant dans l'ensemble $Act_{\text{ext}}^{T_\Pi}$ les actions contrôlables et les actions observables. Alors les modules de test devraient satisfaire des contraintes de contrôlabilité, *e.g.*, le testeur ne doit jamais choisir entre deux actions contrôlables.

Pour pouvoir évaluer une campagne de test sur une implantation, il est nécessaire de définir une notion de couverture pour les tests qui ont été exécutés. La couverture de la propriété initiale peut être définie assez facilement par adaptation des notions de couverture données dans les approches de test où une spécification ainsi qu'une propriété sont utilisées pour générer des cas de test. Parmi les notions de couverture qui nous semblent pertinentes pour notre approche, nous pouvons citer [BDGJ06, TSL04]

Dans l'approche que nous avons proposée, nous nous sommes passé de modèle comportemental du système sous test. Il nous semble intéressant, sans aller jusqu'à un modèle complet de l'implantation, d'utiliser une certaine forme de modèle partiel. Ce modèle partiel pourrait aider à sélectionner les scénarios de test intéressants. Ces scénarios pourraient être dédiés au test de fonctionnalités précises sur le système, par exemple celles liées à la sécurité.

Troisième partie

Mise en œuvre pratique

CHAPITRE 8

j-VETO : a Java Verification and Enforcement TOolbox

Sommaire

8.1 Introduction **139**
8.2 Spécification de la propriété par l'utilisateur **139**
8.3 Vue d'ensemble **141**
8.4 Synthèse de moniteurs **142**
8.4.1 DFA2Streett : synthèse d'automates de Streett depuis des DFAs et des motifs . 142
8.4.2 Streett2Monitor : synthèse de moniteurs depuis les automates de Streett. . . . 144
8.5 Intégration du moniteur **147**
8.5.1 Mapping2Aspect : Génération d'aspects à partir de mappings 147
8.5.2 MonitorTranslator : traduction de moniteurs en Java 148
8.5.3 Bibliothèque pour le slicing 150
8.6 Modèle et utilitaires communs pour les automates **152**
8.6.1 GraphMaker : des graphes pour les DFAs, automates de Streett, et moniteurs . 152
8.6.2 Monitor Composer : Composition de moniteurs 153
8.6.3 Modèle pour les objets de type automate 153
8.7 Éxécution du programme avec son moniteur **153**
8.7.1 Expérimentations en vérification à l'exécution 154
8.7.2 Expérimentations en enforcement à l'exécution 155
8.8 Conclusion et Perspectives **155**

Chapter abstract

We present the toolbox j-VETO (Java Verification and Enforcement Toolbox). This toolbox is dedicated to the runtime verification and enforcement of properties on Java programs. The underlying theory of j-VETO is presented in Chapters 3, 4, 5, and 6. The j-VETO toolbox is able to synthesize and integrate a monitor in a target Java program. In this chapter, we describe the features and some implementation details of j-VETO. More details can be found in the technical report [FCMF09].

Résumé du chapitre

Nous présentons la chaîne d'outils j-VETO (Java Verification and Enforcement Toolbox). Cette boite à outils est dédiée à la vérification et l'enforcement de propriétés durant l'exécution de programmes Java. La théorie sous-jacente utilisée dans j-VETO est celle présentée dans les Chapitres 3, 4, 5, et 6. La boîte à outils j-VETO prend en charge la synthèse de moniteurs et leur intégration dans le programme cible. Dans ce chapitre, nous décrivons les fonctionnalités ainsi que certains détails d'implantation de j-VETO. Plus de détails peuvent être trouvés dans le rapport technique [FCMF09].

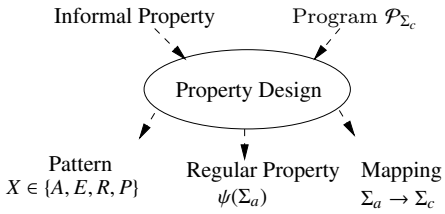


FIGURE 8.1 – Conception de la propriété

8.1 Introduction

La boîte à outils j-VETO (Java Verification and Enforcement TOolbox) est principalement une implantation de l'approche présentée dans les Chapitres 3, 4, 5, et 6. Cette approche a pour but la vérification et l'enforcement de propriétés fournies par un utilisateur sur les programmes Java. Les moniteurs de vérification et d'enforcement utilisés sont ceux définis au Chapitre 6 :

- les moniteurs de vérification utilisent le domaine de vérité $\mathbb{B}_4$, ainsi l'ensemble des r-propriétés de reactivity sont vérifiables.
- les moniteurs d'enforcement utilisent l'ensemble d'opérations $\{halt, store, dump, off\}$, ainsi l'ensemble des r-propriétés de response sont enforçables.

Les moniteurs (cf. Chapitre 2 Section 2.1.1) qui sont produits par j-VETO sont des moniteurs dont le placement peut être inline (les moniteurs sont dans le même espace d'adressage que le programme) ou outline (les moniteurs s'exécutent dans un thread différent). Également, il est possible d'obtenir des moniteurs qui analyseront l'exécution du programme online (pendant l'exécution du programme), ou bien offline (à posteriori).

Pour illustrer les différents concepts utilisés et le fonctionnement de la boite à outils j-VETO, nous utiliserons les deux exemples suivants :

Exemple 30 (Propriété informelle sur un programma Java) Considérons une propriété spécifiant que *"l'on ne doit pas modifier une `Collection` lorsque l'on parcourt ses éléments via l'interface `Iterator`"*.

Exemple 31 (Propriété informelle sur un programma Java) Considérons une propriété spécifiant que *"les opérations portant sur un fichier doivent être effectuées après que celui-ci ait été ouvert et avant de le fermer"*.

Ces propriétés sont des propriétés génériques de bonne utilisation des structures de données Java. Nous pouvons donc souhaiter les vérifier sur des programmes Java quelconques utilisant les éléments mentionnés.

Organisation du chapitre. La suite de ce chapitre est organisée comme suit. Dans la Section 8.2, nous présentons la phase de conception de la propriété. Cette étape préliminaire permet d'obtenir les entrées nécessaires à la boite à outils j-VETO. La Section 8.3 donne une vue d'ensemble de l'architecture et du principe de fonctionnement de j-VETO. La Section 8.4 décrit les outils dédiés à la synthèse de moniteurs. Ce module permet d'obtenir un moniteur à partir des entrées fournies lors de la phase de conception de la propriété. La Section 8.5 est dédiée aux outils permettant l'intégration du moniteur dans le programme Java. Ce module fait appel au compilateur ASPECTJ pour intégrer le moniteur synthétisé dans un programme Java. Ensuite, dans la Section 8.6, nous décrivons la partie "Modèles et Utilitaires" de j-VETO qui regroupe des modèles et utilitaires utilisés par les différents outils des modules décrits dans les sections précédentes. Dans la Section 8.7, nous présentons l'exécution du programme augmenté de son moniteur obtenu grâce à la boite à outils. Enfin la Section 8.8 est consacrée à la conclusion de ce chapitre.

8.2 Spécification de la propriété par l'utilisateur

La première étape nécessaire avant l'utilisation de la boite à outils est de produire les entrées nécessaires à son utilisation. C'est le rôle de la phase de conception de la propriété (property design) ; voir Fig. 8.1.

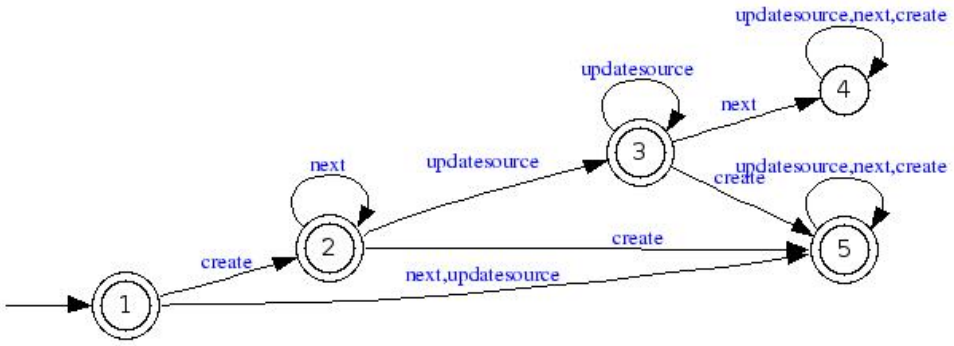


FIGURE 8.2 – DFA reconnaissant la propriété finitaire ψ_1

La phase de conception de la propriété est une opération manuelle que l'utilisateur doit réaliser.
Pour cela, l'utilisateur se sert des entrées suivantes :

- Un programme Java pour lequel un ensemble d'événements concrets est identifié, *e.g.*, appels de méthodes, modification de variables de classes.
  Ces événements doivent pouvoir être observables dans le sens où ils doivent être à portée de la technique d'instrumentation utilisée. Dans la version actuelle de la boite à outils j-VETO, nous utilisons le compilateur ASPECTJ [1] comme technique d'instrumentation support.
- Une propriété informelle portant sur le programme Java considéré.

La phase de spécification de la propriété produit les éléments suivants :

- Les éléments de base pour construire une r-propriété d'une classe de base que l'on souhaite vérifier ou enforcer sur un programme. L'utilisateur doit ainsi définir deux éléments (cf. Chapitre 3) :
  - une propriété finitaire ;
  - un "motif de répétition" pour cette propriété finitaire (cf. Définitions 9 et 10, p. 39).
- Une association (un mapping) entre les événements abstraits définis dans la propriété et les événements concrets du programme. Comme nous utilisons la technologie AspectJ pour l'intégration du moniteur, et notamment l'observation des événements, le type de ces événements est contraint par les possibilités d'AspectJ.

Notons que pour les r-propriétés des classes composées, il suffit d'effectuer cette étape pour les éléments de base de la r-propriété et de les composer ensuite grâce à l'outil *Monitor Composer* permettant de composer les moniteurs avec des opérations booléennes (voir Section 8.6).

Nous décrivons maintenant plus en détails la phase de conception de la propriété. Partant d'une propriété informelle et d'un programme Java à analyser, l'utilisateur doit réaliser une première étape consistant à spécifier à propriété qu'il souhaite voir vérifiée ou enforcée sur le programme. Pour cela, une formalisation de l'expression informelle doit être réalisée. L'utilisateur identifie une propriété régulière $\psi(\Sigma_a)$ et un "motif" apparaissant dans la propriété. La propriété est exprimée comme un DFA qui est défini sur un vocabulaire abstrait Σ_a. Le motif utilisé par l'utilisateur correspond à un opérateur de la classification *Safety-Progress*, déclarant que la propriété doit être vérifiée toujours (opérateur A), au moins une fois (opérateur E), régulièrement (opérateur R), ou de façon persistante (opérateur P). Aussi chaque élément de l'alphabet abstrait Σ_a doit être associé à un événement concret et observable du programme analysé dans l'alphabet Σ_c.

Exemple 32 (Conception de propriété) La propriété informelle introduite dans l'Exemple 30 peut être formalisée par une r-propriété Π_1 de safety exprimant que la propriété finitaire suivante doit *toujours* être vérifiée (*i.e.*, $\Pi_1 = (A_f(\psi_1), A(\psi_1))$). La propriété ψ_1 est représentée sur la Fig. 8.2, et définie formellement par l'expression régulière suivante :

$$\neg(\mathit{createIter} \cdot \mathit{next}^* \cdot \mathit{updatesource}^+ \cdot \mathit{next})$$

1. AspectJ, Aspect for Java, http://www.eclipse.org/aspectj/

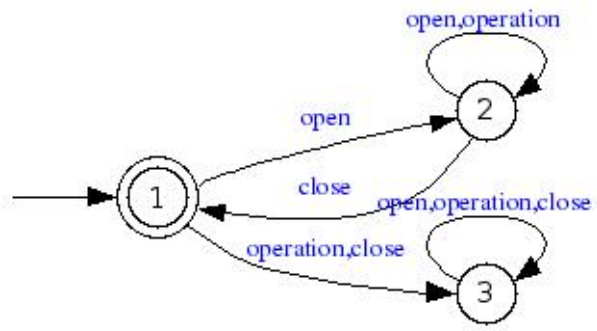


FIGURE 8.3 – DFA reconnaissant la propriété finitaire ψ_2

Elle définit un vocabulaire abstrait $\Sigma_a^1 = \{\mathit{createIter}, \mathit{next}, \mathit{updatesource}\}$. Les événements abstraits de la propriété sont associés aux événements concrets d'un programme Java de la manière suivante :

- *createIter* correspond à la création d'un objet de type `Iterator` par appel de la méthode `Collection.iterator()`.
- *next* correspond à l'appel de la méthode `Iterator.next()`.
- *updatesource* est la modification de la `Collection`, c'est-à-dire l'ajout ou la suppression d'éléments dans la `Collection`. Ce qui se fait respectivement par appel aux méthodes `Collection.add()` ou `Collection.remove()`.

Exemple 33 (Conception de propriété) La propriété informelle introduite dans l'Exemple 31 peut être formalisée par une r-propriété Π_2 de response exprimant que la propriété finitaire suivante doit *régulièrement* être vérifiée (*i.e.*, $\Pi_2 = (R_f(\psi_2), R(\psi_2))$. La propriété ψ_2 est représentée sur la Fig. 8.3, et définie formellement par l'expression régulière suivante :

$$\psi_2 = (\texttt{open} \cdot \texttt{operation}^* \cdot \texttt{close})^*$$

Elle définit un vocabulaire abstrait $\Sigma_a^2 = \{\mathit{open}, \mathit{operation}, \mathit{close}\}$. Les événements abstraits de la propriété sont associés aux événements concrets d'un programme Java de la manière suivante :

- *open* correspond à la création d'un objet de type `FileInputStream` par appel au constructeur de cette classe.
- *operation* correspond à l'appel d'une méthode portant sur un objet de type `FileInputStream`.
- *close* est la fermeture du flux associé à un objet de type `FileInputStream`.

8.3 Vue d'ensemble

Dans cette section nous décrivons la boite à outils prototype *j*-VETO. Une vue d'ensemble est représentée sur la Fig. 8.4 page suivante. La boite à outils est développée en Java et utilise XML [2], XSLT [3], XStream [4], et la programmation orientée aspects [KLM+97] pour Java (ASPECTJ) comme techniques sous-jacentes.

Utilisation des différents modules. Prenant en entrée une propriété régulière ψ définie sur un alphabet Σ_a ($\psi \subseteq \Sigma_a^*$) et représentée par un DFA, et un motif, l'utilisateur peut obtenir un moniteur (de vérification ou d'enforcement) et un squelette de mapping pour la r-propriété Π en utilisant les outils dédiés à la synthèse de moniteur. À partir de ce moniteur, le mapping complété ($\Sigma_a \leftrightarrow \Sigma_c$: une correspondance entre les événements abstraits et concrets), les directives d'intégration du moniteur, et un programme Java $\mathcal{P}_{\Sigma_c}$, les outils dédiés à l'intégration du moniteur sont capables de produire un nouveau programme $\mathcal{P}'_{\Sigma_c}$ de manière à ce que la propriété Π soit vérifiée ou enforcée en respectant les directives d'intégration. Ces dernières spécifient les différentes options lors de la vérification ou l'enforcement (*e.g.*, fichier de log, diagnostic en cas d'erreur, ...).

2. Extensible Markup Language - http://www.w3.org/XML/
3. The Extensible Stylesheet Language Family - http://www.w3.org/Style/XSL/
4. http://xstream.codehaus.org/

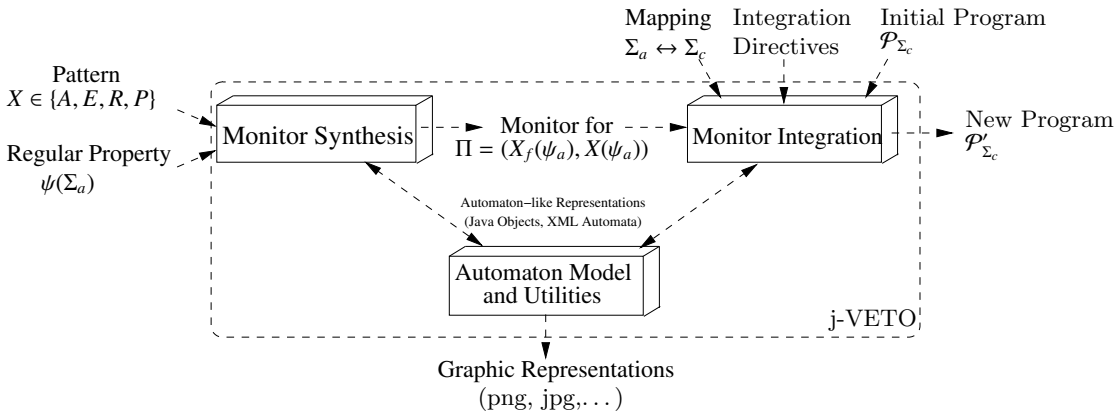


FIGURE 8.4 – Vue d'ensemble de la boite à outils j-VETO

Pour faciliter et standardiser l'interaction des composants de la boite à outils, nous avons défini un modèle de classes commun à tous les outils qui permet d'abstraire les concepts des différents types d'automates utilisés dans j-VETO (automates de Streett, DFAs, moniteurs de vérification et d'enforcement). La boite à outils utilise également la bibliothèque XStream pour faciliter la production et la consommation d'objets Java représentés dans le format XML.

Automatisation des différentes tâches par des scripts. La boîte à outils j-VETO est livrée également avec deux scripts :

- Le premier automatise la génération du moniteur à partir d'une propriété finitaire et du motif souhaité sur cette propriété. Ce script automatise les appels aux différents outils de la partie synthèse de moniteurs. Également, ce script fait appel à un utilitaire pour générer les graphes des différentes représentations basées sur un automate : propriété finitaire utilisée en entrée, automate de Streett représentant la r-propriété, et moniteur.
- Le second automatise l'intégration du moniteur dans le programme Java à vérifier ou enforcer. À partir du moniteur, du mapping, de directives d'intégration, et du programme initial, ce script automatise la transformation du moniteur précédemment généré en un moniteur représenté sous forme de programme Java. L'intégration, via le compilateur AspectJ, de ces éléments et de certaines bibliothèques (voir Section 8.5) dans le programme Java initial est automatisée via ce script basé sur ant [5].

8.4 Synthèse de moniteurs

La partie "synthèse de moniteurs" de la boite à outils j-VETO est constituée de deux outils (voir Fig. 8.5) :

DFA2Streett : cet outil synthétise un automate de Streett $\mathcal{A}_\Pi$ spécifiant une r-propriété $\Pi = (X_f(\psi), X(\psi))$ à partir d'un DFA $\mathcal{A}_\psi$ spécifiant une propriété régulière ψ et un motif donné X.

Streett2Monitor : cet outil transforme un automate de Streett reconnaissant une r-propriété Π donnée en entrée en un moniteur de vérification $\mathcal{A}_{?\Pi}$ ou un moniteur d'enforcement $\mathcal{A}_{!\Pi}$ pour cette r-propriété.

8.4.1 DFA2Streett : synthèse d'automates de Streett depuis des DFAs et des motifs

Cet outil transforme un DFA en un automate de Streett en utilisant Java et XSLT. L'outil prend en entrée un DFA encodé en XML représentant une propriété (régulière) ψ, et un motif (safety, guarantee, response, persistence). Les transformations définies dans le Chapitre 3 Section 3.6 sont implantées par des règles XSLT.

Pour synthétiser l'automate de Streett recherché, l'outil procède comme suit :

5. http://ant.apache.org/

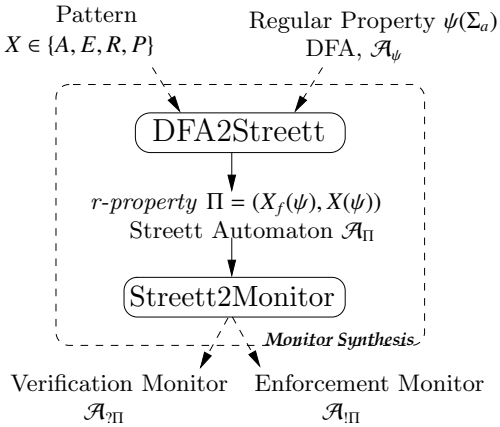


FIGURE 8.5 – La partie "synthèse de moniteurs" de j-VETO

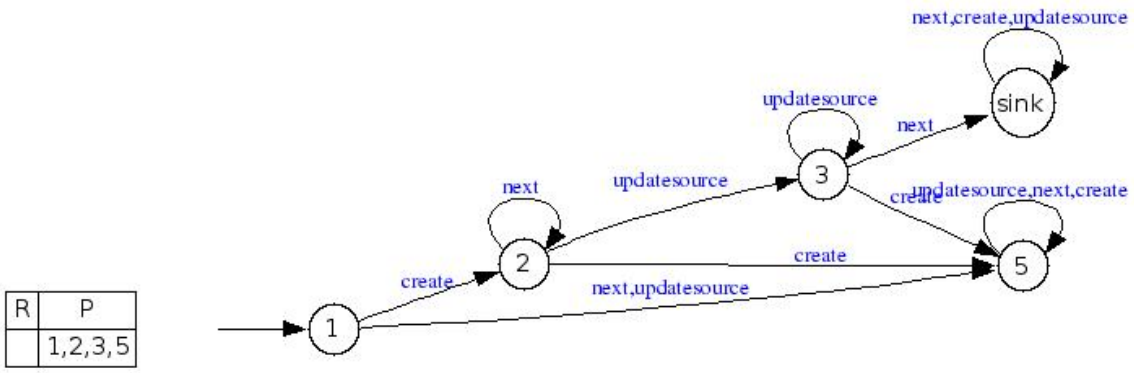


FIGURE 8.6 – Automate de Streett de Safety résultant de l'application de *DFA2Streett* sur le DFA reconnaissant ψ_1

1. Le DFA encodé en XML est désérialisé en des objets Java en utilisant la bibliothèque XStream. Dans ce processus, certaines vérification de base telles que la validation XML ou la vérification d'états dupliqués sont réalisées. L'outil rejette ainsi les documents XML mal formés ou les DFA encodés de manière invalide.
2. Une fois que le DFA a été chargé avec succès, l'outil vérifie sa correction. C'est-à-dire, l'outil vérifie le déterminisme et la complétude de l'automate encodé.
3. La transformation XSL est appliquée au DFA. De façon simplifiée, celle-ci modifie les états et transitions initialement présents. Cette modification est basée sur le motif (safety, guarantee, response ou persistence) fourni par l'utilisateur.

   Sur le Listing 8.1 est représenté un extrait de la transformation XSL utilisée pour déterminer les paires d'acceptation d'un automate de Streett. Par exemple, si l'on veut transformer un DFA $\mathcal{A}$ en un automate de Streett $\mathcal{A}_\Pi$ spécifiant une r-propriété de *response* Π, on peut voir que l'état accepteur $q \in Q^{\mathcal{A}}$ appartient aux états récurrents de l'automate de Streett correspondant $\mathcal{A}_\Pi$, si et seulement si il existe un autre état accepteur $q' \in Q^{\mathcal{A}}$ atteignable depuis q.
4. L'automate de Streett résultat est un nouveau fichier XML qui est traité de manière à supprimer les états non atteignables qui ont pu être laissés après la transformation.

Exemple 34 (Application de *DFA2Streett*) Nous appliquons l'outil *DFA2Streett* au DFA (Fig. 8.2) de la propriété ψ_1 de l'Exemple 32. L'automate de Streett résultant est représenté sur la Fig. 8.6. Conformément à la transformation DFA2Streett définie dans le Chapitre 3 Définition 18 (p. 52), nous pouvons observer que l'ensemble des états persistants est $\{1, 2, 3, 5\}$.

```
<xsl:template match="AcceptingCondition">
  <AcceptingCondition>
    <Pair>
      <xsl:choose>
        <!-- Response property: P=empty -->
        <xsl:when test="$property_type='response'">
          <xsl:attribute name="R">
            <xsl:for-each select="/DFAutomaton/AcceptingCondition/string">
              <xsl:variable name="reach_q">
                <xsl:call-template name="reach">
                  <xsl:with-param name="next_state">
                    <xsl:value-of select="text()"/>
                  </xsl:with-param>
                  <xsl:with-param name="stopCondition">
                    <xsl:value-of select="'true'"/>
                  </xsl:with-param>
                </xsl:call-template>
              </xsl:variable>
              <xsl:if test="contains($reach_q, 'true')">
                <xsl:value-of select="concat(text(), ',')"/>
              </xsl:if>
            </xsl:for-each>
          </xsl:attribute>
        </xsl:when>
      </xsl:choose>
    </Pair>
  </AcceptingCondition>
</xsl:template>
```

Listing 8.1 – Extrait de la transformation XSL utilisée dans DFA2Streett

Exemple 35 (Application de *DFA2Streett*) Nous appliquons l'outil *DFA2Streett* au DFA (Fig. 8.3) de la propriété ψ_2 de l'Exemple 33. L'automate de Streett résultant est représenté sur la Fig. 8.7. Conformément à la transformation DFA2Streett définie dans le Chapitre 3 Définition 20 (p. 54), nous pouvons observer que l'ensemble des états récurrents est $\{1\}$.

8.4.2 Streett2Monitor : synthèse de moniteurs depuis les automates de Streett.

Cet outil peut synthétiser des moniteurs de vérification (VM) $\mathcal{A}_{?\Pi}$ ou des moniteurs d'enforcement (EM) $\mathcal{A}_{!\Pi}$ pour une r-propriété Π. Il prend comme entrée un automate de Streett encodé en XML $\mathcal{A}_{\Pi}$ reconnaissant une r-propriété Π. Un fichier XSL implantant les règles de transformation générales pour les r-propriétés (introduites au Chapitre 6) est utilisé. Cet outil produit un nouveau fichier XML représentant le moniteur (soit de vérification ou d'enforcement) et un fichier de squelette de mapping dont le but est de lier les événements "abstraits" et "concrets" :

- Les événements abstraits sont ceux utilisés dans le moniteur. Ils sont une représentation abstraite d'événements du programme.
- Les événements concrets sont des événements observables du programme Java cible $\mathcal{P}_{\Sigma_c}$. Chaque événement concret correspond à la mise en œuvre sur le programme Java considéré d'un événement abstrait.

Synthèse de moniteurs

Étant donné un automate de Streett reconnaissant une r-propriété Π, le moniteur produit vérifie ou enforce Π.

Pour effectuer la transformation, l'outil procède selon les étapes suivantes :

1. L'automate de Streett encodé en XML est désérialisé en un ensemble d'objets Java en utilisant la bibliothèque XStream. Comme dans les outils précédemment introduits, certaines vérifications de base sont réalisées comme la validation XML ou la détection d'états dupliqués. Ceci est fait dans le but de rejeter les automates de Streett malformés. Pour cette étape, l'outil s'appuie sur la bibliothèque Automaton Library (voir Section 8.6).

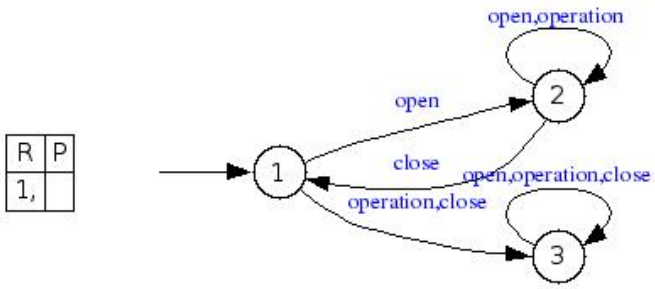


FIGURE 8.7 – Automate de Streett de Response résultant de l'application de *DFA2Streett* sur le DFA reconnaissant ψ_2

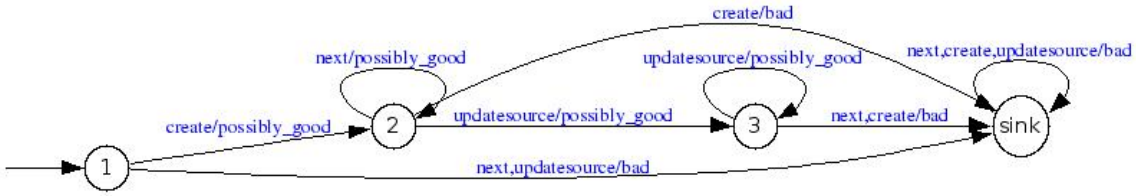


FIGURE 8.8 – Moniteur de vérification résultant de l'application de *Streett2Monitor* sur l'automate de Streett reconnaissant $(A_f(\psi_1), A(\psi_1))$

2. L'outil vérifie ensuite le déterminisme et la complétude de l'automate de Streett désérialisé. Nous ne faisons pas cette vérification durant le chargement car certains états référencés peuvent ne pas avoir été encore chargés. Ainsi, réaliser cette vérification au cours du chargement serait plus difficile à implanter et source d'erreurs.
3. La transformation XSL est ensuite appliquée au fichier XML. Cet outil peut générer des moniteurs en version automate de Moore ou de Mealy.

   Comme indiqué au Chapitre 5, les états d'un automate de Streett peuvent être caractérisés par un ensemble $\mathbb{P}^{\mathcal{A}} = \{Good^{\mathcal{A}}, Good_p^{\mathcal{A}}, Bad_p^{\mathcal{A}}, Bad^{\mathcal{A}}\}$ dont les éléments sont des sous-ensembles des états de l'automate de Streett $\mathcal{A}$. Ainsi, la transformation consiste à déterminer le sous-ensemble auquel appartient chaque état de l'automate.

Sur le Listing 8.2 est représenté un extrait de la transformation XSL s'appliquant aux états récurrents R_i. Cette transformation est utilisée pour traiter la partie récurrente d'un automate de Streett. Cette partie de la transformation est utilisée notamment pour les automates de guarantee, response et obligation. De façon simplifiée, pour une transition depuis `$currentState` vers `$nextState`, l'opération partielle *dump* est selectionnée si la variable `$nextState` appartient aux états récurrents R_i. Dans les autres cas, l'opération *store* ou *halt* est selectionnée selon qu'il existe au moins un état (partant de `$nextState`) qui appartient à l'ensemble des états récurrents R_i. On peut également remarquer que les automates d'obligation nécessitent un traitement spécial pour éviter les transitions depuis $\$currentState \in R_i$ vers $\$nextState \in \overline{R_i}$ (ces transitions sont par définition non autorisées).

Exemple 36 (Application de *Streett2Monitor*) Nous appliquons l'outil *Streett2Monitor* pour obtenir un moniteur de vérification pour la r-propriété reconnue par l'automate de Streett de l'Exemple 34. Le moniteur de vérification est représenté sur la Fig. 8.8. Le codage interne en XML est montré sur le Listing 8.3.

Exemple 37 (Application de *Streett2Monitor*) Nous appliquons l'outil *Streett2Monitor* pour obtenir un moniteur d'enforcement pour la r-propriété reconnue par l'automate de Streett de l'Exemple 35. Le moniteur d'enforcement est représenté sur la Fig. 8.9.

```
<xsl:template name="guarantee-automaton">
        <xsl:param name="m-pos" />
        <xsl:param name="currentState" />
        <xsl:param name="nextState" />
        <xsl:param name="Ri"/>
<xsl:choose>
  <!-- nextState belongs to Ri -->
  <xsl:when test="contains($Ri, $nextState)">
   <xsl:value-of select="dump"/>
  </xsl:when>
  <!-- nextState does not belong to Ri -->
  <xsl:otherwise>
    <!-- Determine if Reach(nextState) arrives to a state Ri. -->
    <xsl:variable name="reach_has_R">
      <xsl:call-template name="reach">
        <xsl:with-param name="next_state">
          <xsl:value-of select="$nextState"/>
        </xsl:with-param>
        <xsl:with-param name="R_states">
          <xsl:value-of select="$Ri"/>
        </xsl:with-param>
        <xsl:with-param name="visited_states">
          <xsl:text/>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:variable>

    <xsl:choose>
      <!-- For an obligation automaton, there is no transition from Ri to [not Ri] -->
      <xsl:when test="$m-count > 1 and contains($Ri, $currentState)"/>
      <xsl:otherwise>
        <xsl:choose>
          <!-- store if Reach(nextState) containes a state in R -->
          <xsl:when test="contains($reach_has_R, 'true')">
            <xsl:value-of select="'store'"/>
          </xsl:when>
          <!-- 'halt' otherwise -->
          <xsl:otherwise>
            <xsl:value-of select="'halt'"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:otherwise>
 </xsl:choose>
</xsl:template>
```

Listing 8.2 – Extrait de la transformation XSL utilisée dans Streett2Monitor

```
<?xml version="1.0" encoding="UTF-8"?>
<VerificationMonitor kind="MEALY">
  <Alphabet>
    <string>next</string>
    <string>create</string>
    <string>updatesource</string>
  </Alphabet>
  <States>
    <State id="1" initial="true">
      <Transition nextState="2" event="create" output="possibly_good" />
      <Transition nextState="sink" event="next" output="bad" />
      <Transition nextState="sink" event="updatesource" output="bad" />
    </State>
    <State id="2">
      <Transition nextState="2" event="next" output="possibly_good" />
      <Transition nextState="3" event="updatesource" output="possibly_good" />
      <Transition nextState="sink" event="create" output="bad" />
    </State>
    <State id="sink">
      <Transition nextState="sink" event="next" output="bad" />
      <Transition nextState="sink" event="create" output="bad" />
      <Transition nextState="sink" event="updatesource" output="bad" />
    </State>
    <State id="3">
      <Transition nextState="3" event="updatesource" output="possibly_good" />
      <Transition nextState="sink" event="next" output="bad" />
      <Transition nextState="sink" event="create" output="bad" />
    </State>
  </States>
</VerificationMonitor>
```

Listing 8.3 – Représentation interne en XML du moniteur de vérification pour la propriété SafeIterator

Génération d'un squelette de Mapping

Comme mentionné au début de la Section 8.4.2, un squelette de mapping est également généré par l'outil *Streett2Monitor*. Ce mapping permet d'utiliser des événements abstraits au niveau de la propriété. La correspondance est ainsi faite entre les événements de la spécification et les événements concrets du programma Java cible (*e.g.*, appel de méthode, modification d'attribut des classes, ...)

Le mapping est également généré en utilisant une transformation XSL. Cette transformation utilise l'automate de Streett encodé en XML et génère un nouveau fichier XML. Cette transformation récupère tous les éléments de l'alphabet de l'automate de Streett, et génère pour chacun une structure de données permettant à l'utilisateur de spécifier de manière abstraite et simplifiée les pointcuts et advices de l'aspect écrit en AspectJ (voir p. 18).

Ensuite, l'utilisateur indique les entrées correspondant à chaque événement abstrait, *i.e.*, associé à chaque événement abstrait l'événement concret correspondant. Ceci est réalisé en décrivant le pointcut de l'aspect final qui sera généré.

Une fois que le mapping a été rempli par l'utilisateur, l'outil *mapping2Aspect* l'utilise pour générer un aspect qui pourra être tissé avec le programme Java cible $\mathcal{P}_{\Sigma_c}$.

Des exemples de squelette de mappings et de mappings complétés sont donnés dans [FCMF09].

8.5 Intégration du moniteur

Cette section présente comment le moniteur, synthétisé par la phase précédente, est intégré dans un programme Java cible. Une vue d'ensemble des outils d'intégration est présentée sur la Fig. 8.10.

8.5.1 Mapping2Aspect : Génération d'aspects à partir de mappings

Nous avons vu que l'outil Streett2Monitor génère un fichier squelette de mapping qui doit être complété par l'utilisateur. Ce fichier fait le lien entre les événements abstraits de la propriété et les événements concrets du programme.

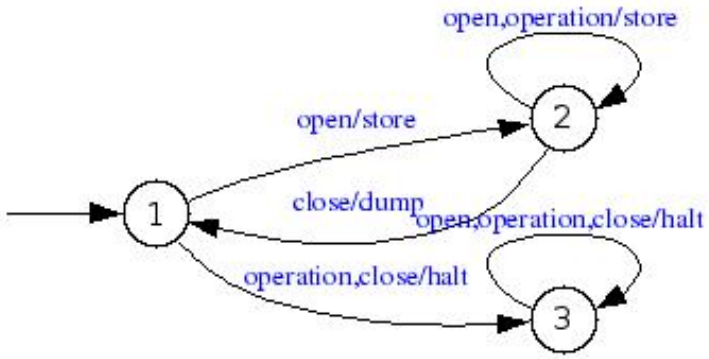


FIGURE 8.9 – Moniteur d'enforcement résultant de l'application de *Streett2Monitor* sur l'automate de Streett reconnaissant $(R_f(\psi_2), R(\psi_2))$

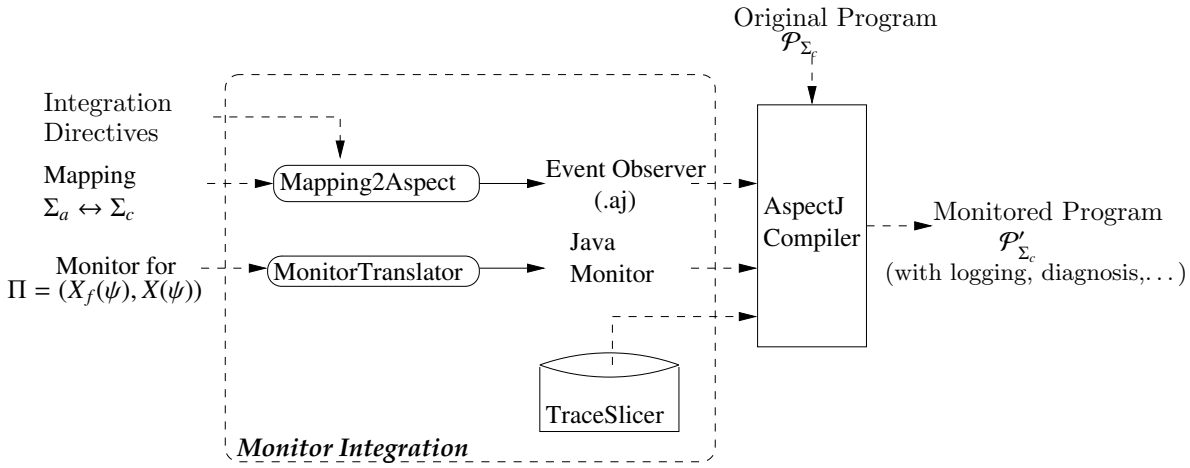


FIGURE 8.10 – La partie "Intégration du moniteur" de j-VETO

La génération du fichier d'aspect ASPECTJ est réalisée selon les étapes suivantes :

- Création des imports Java à effectuer.
- Création de chaque *pointcut* AspectJ. Ce sont ces *pointcut* qui vont permettre de récupérer chaque événement du programme cible associé à la propriété que l'on veut vérifier.
- Pour chaque *pointcut* son *advice* associé est généré. Un *advice* correspond au code exécuté lorsque son *pointcut* capte l'exécution d'une méthode correspondant à un événement de la propriété. Le rôle de chaque advice est de transmettre l'événement au moniteur.
- Ajout du *pointcut* captant la fin du programme (défini par l'appel à `System.exit()`).

Cette génération automatique produit le fichier nécessaire à l'intégration du moniteur dans le programme. C'est grâce à ce fichier que l'instrumentation du programme est faite. Pour la propriété *Safeiterator*, un extrait de mapping généré est représenté sur le Listing 8.4.

8.5.2 MonitorTranslator : traduction de moniteurs en Java

Cet outil transforme un moniteur codé en XML en une classe Java. Nous avons choisi, dans un souci d'efficacité à l'exécution, d'encoder les événements et les états du moniteur par des entiers. De plus, la fonction de transition du moniteur est traduite par un ensemble d'expressions conditionnelles imbriquées.

Exemple 38 (Génération d'un moniteur avec *MonitorTranslator*) La classe Java donnée dans le Listing 8.5 est obtenue par application de l'outil *MonitorTranslator* au moniteur de vérification pour la propriété *SafeIterator* (Fig. 8.8).

```
public aspect EventGenerator {
  // Some variable declarations

  public static void init(TraceSlicer slicer) {
    nbCreate = nbModify = nbAccess = overheadTime = 0;

    currentSlicer = slicer;
    collectionObjects=new WeakIdentityHashSet();
    iteratorObjects=new WeakIdentityHashSet();

    createColC = new ArrayList<Integer>(1); createColI = new ArrayList<Integer>(1);
    updatesourceC = new ArrayList<Integer>(1); nextI = new ArrayList<Integer>(1);

    createColC.add(null); createColI.add(null); updatesourceC.add(null);
    nextI.add(null);
  }

  pointcut analysis(): within(jveto..*) ;;

  pointcut dacapo(): within(dacapo..*) || within(dacapo.*) || within(dacapo.parser.*);

  pointcut createIterator(Collection c) :
        call(Iterator Collection+.iterator()) && target(c) && !analysis();
  after(Collection c) returning(Iterator i) : createIterator(c) {
    int collHash = System.identityHashCode(c),
    itHash = System.identityHashCode(i);
    collectionObjects.add(c); iteratorObjects.add(i);
    createColC.set(0, collHash); createColI.set(0, itHash);
    currentSlicer.monitorEvent(1, createColC, createColI);
  }

  pointcut updatesource(Collection c) :
         (call(* Collection+.remove*(..)) || call(* Collection+.add*(..))) && target(c)
        && !analysis();
        ...
    return proceed(c);
  }

  pointcut next(Iterator i) : call(* Iterator.next()) && target(i) && !analysis();
  Object around(Iterator i) : next(i) { ... }

  pointcut systemexit():
         call(* System.exit(..)) && (within(Harness) || within(TestSafeIterator));
  void around(): systemexit() {
    // Display number of events, slices
    // Computations and displays about memory usage, execution time
  }

  pointcut reinit() : call(* dacapo.Callback.start(..));
  before() : reinit() {
    if (currentSlicer != null)
      init(new TraceSlicer(new SafeIteratorMonitor()));
  }
}
```

Listing 8.4 – Extrait de l'aspect généré pour la propriété SafeIterator

```
public class SafeIteratorMonitor extends Monitor {
        private int getNewState(int event, Integer state) {
                // create=1; updatesource=2; next=3
                switch (state) {
                case 1:
                        if (event == 1) {
                                return 2;
                        } break;
                case 2:
                        if (event == 2) {
                                return 3;
                        } else if (event == 3) {
                                return 2;
                        } break;
                case 3:
                        return 3;break;
                }
                return 4;
        }

        @Override
        public void performOperation(int event, SliceState state) {
                state.setState(getNewState(event, state.getState()));
        }

        @Override
        public boolean propertyViolation(int state) { return state == 4; }
}
```

Listing 8.5 – Moniteur généré pour la propriété SafeIterator par MonitorTranslator (extrait)

8.5.3 Bibliothèque pour le slicing

Comme nous l'avons vu dans les sections précédentes, les événements abstraits d'une spécification sont reliés à des événements concrets du programme. Les événements reçus par le moniteur sont donc associés à des objets instanciés lors de l'exécution du programme cible. Si nous examinons les événements de la propriété *SafeIterator*, nous pouvons observer que les événements sont dépendants de paramètres. Par exemple, l'événement `next` est relié à une instance d'itérateur obtenue à l'exécution. En effet, la propriété *SafeIterator* s'applique en réalité à toute collection et tout itérateur produit à partir d'une collection. Cette propriété paramétrique peut se formaliser comme suit.

$$\forall c \in \texttt{Collection}, \forall i \in \texttt{Iterator}, \neg(\texttt{create(c,i)} \cdot \texttt{next(i)}^* \cdot \texttt{updatesource(c)}^+ \cdot \texttt{next(i)})$$

Les approches de vérification à l'exécution prenant en compte les propriétés paramétriques sont nombreuses ; ce qui montre l'intérêt pour ce type de propriétés. Récemment Feng et Rosu ont proposé dans [CR09] un cadre formel pour décrire le problème de la vérification de propriétés paramétrées. Ils ont également proposé un algorithme permettant de prendre en compte les propriétés paramétriques dans leur généralité.

Nous choisissons de nous baser sur ce cadre formel, mais d'utiliser un algorithme différent. Les performances de l'algorithme que nous proposons, en terme de temps d'exécution, se sont révélées être comparables. En revanche cet algorithme possède l'avantage d'être moins gourmand en terme d'espace mémoire consommé.

L'originalité de cet algorithme repose sur l'observation suivante. Dans les événements paramétrés produits par un programme Java, il est possible de partitionner les paramètres effectifs en paramètres *indépendants* et paramètres *dépendants*. Prenons l'exemple de la propriété *SafeIterator* qui dépend de deux paramètres `c` pour les `Collection` du programme et `i` pour les `Iterateur`. L'événement `create(c,i)` qui correspond à la création d'un itérateur `i` à partir d'une collection `c` indique que les paramètres effectifs dépendant du paramètre formel `i` sont toujours reliées à des paramètres effectifs dépendant du paramètre formel `c`. Ceci correspond à l'idée intuitive, que lors de l'exécution, les itérateurs peuvent être crées uniquement à partir de collections. Pour l'événement `create(c,i)`, nous partitionnons l'ensemble des paramètres $\{c, i\}$ comme $\{\{c\}, \{i\}\}$ où l'ensemble $\{c\}$ (resp. $\{i\}$) est l'ensemble des paramètres indépendants

(resp. dépendants).

Pour chaque événement paramétré d'une spécification, il est possible de réaliser cette partition par analyse de la spécification des événements concrets. En Java, cette spécification est simplement l'API de la classe où apparaît l'événement concret. Par exemple, suivant l'API de classe `Collection`, nous pouvons observer que la méthode `iterator()` retourne un itérateur à partir de l'objet de type `Collection` auquel elle est appliquée.

Notations. Nous considérons une r-propriété paramétrée $\Pi(X)$ où X est l'ensemble des paramètres de la r-propriété. Nous notons V_X l'ensemble des valeurs effectives que peuvent pendre les paramètres de l'ensemble X lors de l'exécution. Un paramètre effectif est donc une association entre un paramètre de X et une valeur obtenue à l'exécution dans V_X. L'ensemble des paramètres effectifs possibles est alors noté $X \times V_X$. Les listes ou séquences de paramètres effectifs sont des éléments de $(X \times V_X)^*$.

```
1  procedure monitorEvent (e, independents, dependents)
2  begin
3  |  UpdateTables(independents, dependents);
4  |  ∇ ← ∅;
5  |  foreach θ in independents do
6  |  |  ∇ ← ∇ ∪ getSubTree(θ);
7  |  end
8  |  ∇ ← ∇ ∪ dependents;
9  |  foreach θ in ∇ do
10 |  |  Θ(θ) ←→_Π (Θ(θ), e);
11 |  |  Ω(θ) ← Γ^Π(Θ(θ));
12 |  end
13 end
14 function UpdateTables (independents, dependents)
15 begin
16 |  foreach θ ∈ independents do
17 |  |  if θ ∉ Λ then
18 |  |  |  Λ ← Λ ∪ {(θ, dependents)};
19 |  |  else
20 |  |  |  Λ(θ) ← (Λ(θ) ∪ dependents)
21 |  |  end
22 |  end
23 |  foreach θ ∈ dependents do
24 |  |  if θ ∉ Θ then
25 |  |  |  Θ ← Θ ∪ {θ, q^Π_init};
26 |  |  end
27 |  end
28 end
29 function getSubTree (θ)
30 begin
31 |  ∇ ← {θ};
32 |  foreach θ' in Λ(θ) do
33 |  |  ∇ ← ∇ ∪ getSubTree(θ');
34 |  end
35 |  return ∇;
36 end
```

Algorithme 1 : Algorithme utilisé pour le slicing

Description de l'algorithme utilisé dans la bibliothèque de slicing. La fonction principale de l'algorithme de slicing que nous utilisons est la fonction `monitorEvent`. Celle-ci est appelée par l'advice de l'aspect à chaque production d'événement concret sur le programme. Dans l'advice, nous effectuons le partitionnement suivant les paramètres effectifs indépendants et dépendants. La fonction `monitorEvent(e,independents,dependents)` prend ainsi trois paramètres :

- `e` $\in \Sigma$: l'événement abstrait correspondant à l'événement concret produit par le programme ; *e.g.*, l'événement abstrait `create` lors de l'appel de la méthode `iterator()` sur un objet de type collection.
- `independents` $\in (X \times V_X)^*$: l'ensemble des paramètres effectifs *indépendants* de l'événement `e` ; *e.g.*, pour l'événement `create` c'est une instance de `Collection` ;
- `dependents` $\in (X \times V_X)^*$: l'ensemble des paramètres dépendants ; *e.g.*, pour l'événement `create` c'est une instance d'`Itérateur` obtenue à partir d'une `Collection` ;

De plus, l'algorithme que nous proposons utilise les variables globales suivantes :

- $\mathcal{M}$ est une moniteur pour la r-propriété non paramétrique Π : $\mathcal{M} = (Q^\Pi, {q_{\text{init}}}^\Pi, \rightarrow_\Pi, \Gamma^\Pi)$.
- Λ est une liste d'associations, pour les paramètres effectifs observés à l'exécution. Elle associe chaque paramètre effectif indépendant à sa liste de paramètres dépendants. C'est une séquence d'association d'éléments de $X \times V_X$ à une liste d'éléments de $X \times V_X$, *i.e.*, $\Lambda : ((X \times V_X) \times (X \times V_X)^*)^*$.
- Θ est une liste d'associations entre paramètres effectifs dépendants et l'état du moniteur qui leur est associée, *i.e.*, $\Theta : ((X \times V_X) \times Q^\Pi)^*$.
- Ω est une liste d'associations entre paramètres effectifs dépendants et la sortie du moniteur qui leur est associée, *i.e.*, $\Lambda : ((X \times V_X) \times \mathbb{B}_4)^*$.

La fonction `monitorEvent` utilise deux routines intermédiaires :

- La procédure `UpdateTables`$(\cdot,\cdot)$ prend deux paramètres `independents` et `dependents` qui sont des listes de paramètres effectifs. Cette fonction met à jour les tables Λ et Θ avec les nouvelles listes `independents` et `dependents` provenant de l'événement concret. L'ajout en fin de séquence est représentée par l'union ensembliste.
- La fonction récursive `getSubTree`$(\cdot)$, appliquée à un paramètre θ est utilisée pour obtenir l'ensemble des paramètres effectifs dépendant du paramètre θ. Cette fonction est récursirve car les dépendances entre paramètres effectifs peuvent former une structure d'arbre.

La procédure `monitorEvent` procède comme suit. D'abord, les tables Λ et Θ sont mises à jour par l'appel à la fonction `UpdateTable()` avec les listes des paramètres effectifs indépendants et dépendants associés à l'événement reçu. Ensuite (lignes 4 à 8), la procédure construit ∇ qui est la liste des paramètres effectifs pour lesquels l'état du moniteur doit être mis à jour. Elle est construite en deux étapes. La première (lignes 4 à 7) consiste à ajouter à ∇ tous les paramètres effectifs (dépendants) associés aux paramètres effectifs dans Λ reçus dans la liste `independents`. Ceci se fait par appel à la fonction `getSubTree` sur chaque paramètre effectif de la liste `independents`. La deuxième étape (ligne 8) consiste à ajouter dans ∇ la liste des paramètres effectifs dépendants reçus avec l'événement. Enfin, les instances de moniteurs, *i.e.*, les états et les sorties pour chaque paramètre dépendants sont mis à jour (lignes 9 à 12).

8.6 Modèle et utilitaires communs pour les automates

Le modèle et les utilitaires pour les entités basées sur un automate sont représentés sur la Fig. 8.11. Les outils *Automaton Utilities*, *Automaton Helper*, et *Automaton Model* sont utilisés par les autres outils. L'outil *GraphMaker* produit des représentations graphiques pour les différentes sortes d'automates. L'outil *Monitor Composer* permet la composition de moniteurs selon des opérations fournies par l'utilisateur.

8.6.1 GraphMaker : des graphes pour les DFAs, automates de Streett, et moniteurs

Nous avons implanté dans la boite à outils un composant auxiliaire permettant d'afficher une représentation graphique des objets basés sur les automates (DFAs, automates de Streett, moniteurs de vérification et d'enforcement). L'outil GraphMaker peut traiter n'importe quel objet encodé en XML de

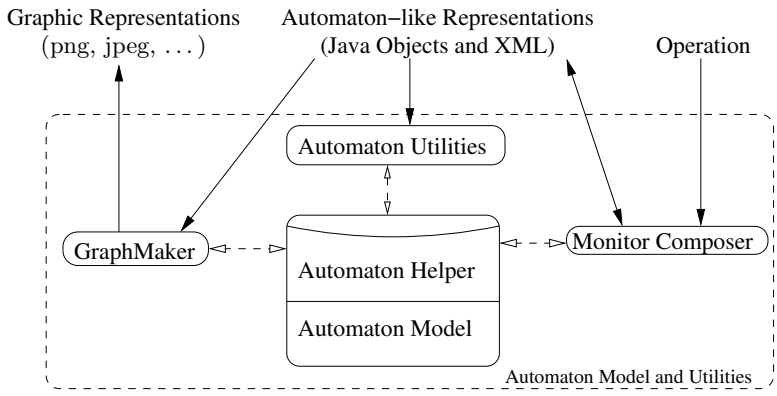


FIGURE 8.11 – Représentation des outils pour le modèle et les utilitaires des automates

type automate et utilise GraphViz [6] pour la génération du graphe. À titre d'exemple, tous les graphes des objets de type automate représentés dans ce chapitre sont obtenus avec l'outil GraphMaker.

8.6.2 Monitor Composer : Composition de moniteurs

L'outil MonitorComposer implémente la composition de moniteurs par des opérations générales (*e.g.*, union, intersection). Le principe formel est détaillé dans le Chapitre 6 Section 6.4. De façon abstraite, l'outil MonitorComposer construit un produit des deux automates sous-jacents des moniteurs passés en paramètres pour réaliser une opération binaire entre les deux. Ensuite, il combine les sorties des moniteurs (de vérification ou d'enforcement). Cette méthode générale, consistant à réaliser d'abord le produit des deux automates et d'ensuite combiner les sorties, est générique. En effet, il est possible de définir ses propres opérations de composition et d'utiliser l'outil MonitorComposer pour composer des moniteurs selon cette opération. Par exemple, ajouter un opérateur de composition pour les moniteurs d'enforcement consiste à définir une fonction de composition $Ops \times Ops \rightarrow Ops$ spécifiant comment combiner les sorties d'un moniteur de vérification ou les opérations d'enforcement d'un moniteur d'enforcement. Ce composant de la boite à outils permet de composer des moniteurs de manière à vérifier ou enforcer des propriétés plus complexes par composition de propriétés plus simples.

Cet outil est notamment utile pour obtenir des moniteurs pour les r-propriétés d'obligation. Ces propriétés sont obtenues par composition booléenne de r-propriétés de safety et de guarantee.

8.6.3 Modèle pour les objets de type automate

L'outil Automaton Model de j-VETO consiste en une hierarchie de classes modélisant toutes les sortes d'entités basées sur les automates qui sont utilisées par les différents outils. Nous décrivons celle-ci de manière simplifiée. Nous avons modélisé la notion d'automate par une classe *Automaton*. Cette classe (abstraite) contient l'alphabet, les états, et leur transitions. Elle est héritée à différents niveaux par différentes classes, spécialisant les entités de type automate. Par exemple, nous avons distingué les automates à entrée/sortie (les moniteurs) des automates classiques (DFA, automate de Streett). Également, nous avons modélisé par différentes classes les différentes conditions d'acceptation spécifiques aux automates utilisés dans j-VETO.

Nous avons implanté également, dans ce module, un composant *AutomatonHelper* qui fournit des moyens pour rendre les objets persistants en XML. Cet utilitaire consite à configurer et personnaliser la bibliothèque XStream pour réaliser la sérialisation et la désérialisation.

8.7 Éxécution du programme avec son moniteur

Nous présentons maintenant les moniteurs de j-VETO au travail, *i.e.*, comment les propriétés sont vérifiées ou enforcées sur les programmes Java.

6. http://www.graphviz.org

Programme	Nombre d'événements	Instances de Moniteurs	Tps d'éxéc. brut (ms)	Tps d'éxéc. monitoré (ms)	Temps suppl. (ms)	Ralentissement (%)
antlr	1990	0	6307	6401	94	1.4
bloat	90569284	38924	5443	202491	197048	3600
chart	568688	100	10121	10647	526	4.9
fop	49866	59	3669	3727	58	1.6
hsqldb	0	0	7871	7923	52	0.6
jython	137048	0	13931	14028	97	0.7
luindex	82162	908	11080	11102	22	0.1
lusearch	334495	0	13341	13742	401	3.0
pmd	25873875	34954	9554	39127	29573	309
xalan	869056	0	22462	23180	718	3.2

TABLE 8.1 – Résultats pour la propriété SafeIterator

Programme	Ralentissement Java-MOP (%)	Ralentissement j-VETO (%)
antlr	0.0	1.4
bloat	×	3600
chart	2.1	4.9
fop	2.5	1.6
hsqldb	0	0.6
jython	1.0	0.7
luindex	0.0	0.1
lusearch	0.0	3.0
pmd	216	309
xalan	3.1	3.2

TABLE 8.2 – Comparaison des ralentissements avec Java-MOP pour la propriété SafeIterator

8.7.1 Expérimentations en vérification à l'exécution

Pour tester les répercussions de l'intégration du moniteur sur les performances du programme cible, nous avons utilisé un benchmark Java nommé *DaCapo* [BGH+06]. Ce benchmark est constitué de plusieurs programmes Java open-sources ayant une utilisation intensive des structures de données Java.

Résultats expérimentaux L'évaluation de la boîte à outils à été effectuée sur un Intel Dual-Xeon 3.20GHz avec 2GB de mémoire vive, ayant pour système d'exploitation Debian GNU/Linux 4.0. nous avons utilisé le benchmark DaCapo version 2006-10 sur 10 applications : antlr, bloat, chart, fop, hsqldb, jython, luindex, lusearch, pmd et xalan. Chaque exécution a été effectuée avec l'option `-converge` qui exécute plusieurs fois le logiciel jusqu'à avoir un temps d'exécution ayant moins de 3% de variation avec les temps d'exécution précédents. De plus, comme le temps d'exécution de DaCapo avec l'option `-converge` présente tout de même des variations d'une exécution à l'autre, nous avons effectué plusieurs fois chaque expérimentation. Chaque benchmark a été ainsi exécuté 5 fois avec l'option converge, et nous avons pris les moyennes des résultats.

Le tableau 8.1 présente les résultats obtenus pour la propriété *SafeIterator*. La colonne *Nombre d'événements* indique le nombre total d'événements concernant la propriété que le fichier aspect a récupéré. Le champ *Moniteurs* indique le nombre d'instances de moniteurs créées durant l'exécution du programme cible, c'est-à-dire le nombre de traces qui ont été vérifiées séparément. La colonne *Tps d'éxec brut* est le temps d'exécution du programme initial sans moniteur intégré. La collone *Tps d'éxec monitoré* est le temps d'exécution du programme cible avec les moniteurs intégrés. *Temps suppl.* indique la différence de temps entre le programme initial et le programme monitoré. Enfin *Ralentissement* montre en pourcentage le ralentissement du programme monitoré par rapport au programme initial.

Nous avons comparé les performances en terme de temps d'exécution et d'occupation mémoire entre les moniteurs produits par j-VETO et ceux produits par JavaMOP. JavaMOP (voir Chapitre 2, Section 2.1) est à ce jour l'outil de vérification à l'exécution présentant les meilleures performances. Nous avons donc

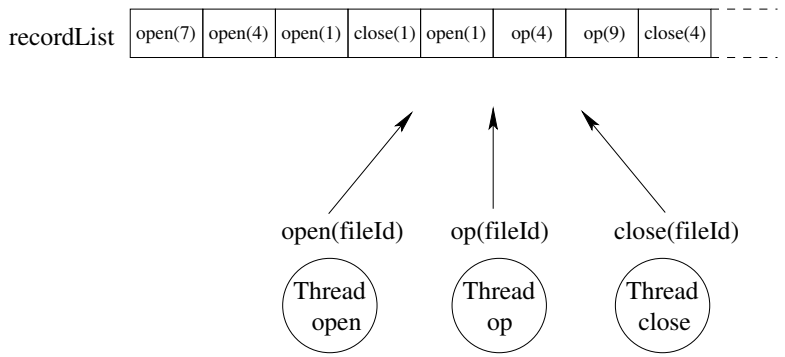


FIGURE 8.12 – Représentation schématique des opérations sur les fichiers

effectué les expérimentations dans les mêmes conditions pour les deux moniteurs. Par analyse de la Table 8.2, nous pouvons observer que les résultats en terme de ralentissement du programme cible sont sensiblement identiques [7]. En revanche, nous pouvons observer que l'exécution du programme `bloat` augmenté du moniteur produit par JavaMOP ne termine pas : une exception est levée lors de l'exécution indiquant que le tas Java a débordé (de taille 2 Go). Nous avons également observé que l'occupation mémoire maximale était à l'avantage de j-VETO. Sur les programmes générant peu d'événements, les performances sont sensiblement identiques. En revanche, pour les programmes générant un nombre important d'événements, la taille occupée par le programme augmenté d'un moniteur produit par j-VETO reste moindre.

8.7.2 Expérimentations en enforcement à l'exécution

Pour illustrer l'enforcement de propriétés, nous décrivons comment le moniteur d'enforcement généré pour la propriété Π_2 peut corriger le comportement d'un programme.

Nous proposons un programme modélisant la réalisation d'opérations sur les fichiers. Les opérations sur les fichiers sont réalisées par des threads Java dont le modèle est proposé dans le Listing 8.6 page suivante. L'ordre dans lequel les actions (ouverture, opération, fermeture) sont réalisées sur les fichiers est mémorisé dans la structure `recordList`. Pour un fichier donné, cet ordre est non prévisible car les actions sur les fichiers sont réalisées de manière non déterministe par les threads.

Nous avons exécuté ce programme pour différents nombres de fichiers (`MAX_FILES`) et en faisant varier "la probabilité" d'écriture de chaque type de threads (en faisant varier leur temps d'endormissement à l'aide de `SEED_SLEEPING`).

Les différentes exécutions ont montré que sur chaque fichier la propriété Π_2 est effectivement enforcée. L'ordre d'exécution est ainsi garanti par le moniteur d'enforcement.

8.8 Conclusion et Perspectives

Conclusion. Dans ce chapitre, notre but était de proposer une implémentation des différentes contributions apportées par les chapitres précédents aux domaines de la vérification et de l'enforcement à l'exécution. Nous nous sommes basés sur le cadre de la classification *Safety-Progress* pour la spécification de propriétés et la synthèse de moniteurs. Nous avons utilisé la technologie ASPECTJ pour l'intégration du moniteur dans le programme cible. Nous souhaitions proposer un cadre permettant d'intégrer des moniteurs dans des programmes faisant une utilisation intensive de ces derniers. A cette fin, nous avons évalué notre approche sur un benchmark couramment utilisé en vérification à l'exécution. Ce qui nous a permis de mesurer la performance de nos moniteurs lorsque la charge de ceux-ci était importante. Nos expérimentations se sont intéressées au temps d'exécution et à l'occupation mémoire des moniteurs. Ce qui nous a permis d'observer que ces derniers présentaient des performances comparables à l'approche de

7. De meilleures performances ont été annoncées dans les différentes publications associées à JavaMOP. Nous n'avons pas pu trouver les conditions expérimentales permettant d'obtenir de telles performances. Nous avons récupéré au cours du mois d'Août 2009, les moniteurs de vérification produits par JavaMOP et publiquement disponibles sur le site Web associé au projet : http://fsl.cs.uiuc.edu/index.php/MOP.

```
public class ThreadSpecial extends Thread{

        private boolean stopped = false;
        public String actionName;
        public List<String> recordList;

        private int MAX_FILES = 50;
        private static final int SEED_SLEEPING = 50;

        public void run() {
                try {
                        while(!stopped){
                                Integer fileNumber = (int)(Math.random() * MAX_FILES);
                                if (this instanceof ThreadAction) {
                                        Thread.sleep(SEED_SLEEPING*2);
                                }
                                else
                                        Thread.sleep(SEED_SLEEPING);
                                doAction(recordList, fileNumber);
                        }
                } catch (InterruptedException e) {e.printStackTrace();}
        }

        public void doAction(List<String> target, Integer i){
                target.add(actionName+"("+i+")");
        }

        public boolean isStopped() {return stopped;}

        public void stopMe() {this.stopped = true;}
}
```

Listing 8.6 – Modèle de Thread effectuant une suite d'opérations

vérification la plus efficace à ce jour. De plus, en terme d'occupation mémoire, ceux-ci se sont révélés être moins gourmands.

Perspectives. Une première direction de recherche serait d'approfondir l'évaluation de j-VETO. Il nous semble intéressant de poursuivre les investigations autour de la performance en terme de temps d'exécution. Également, pour obtenir une évaluation plus complète des performances en terme de consommation mémoire, nous souhaitons étudier la complexité de l'algorithme que nous proposons.

Une autre direction importante de recherche est de maintenant prendre en compte les perspectives soulevées en terme d'applicabilité de l'approche dans un contexte d'observation partielle. Il nous semble aussi intéressant d'étudier comment restreindre les limites pratiques en terme de performance (temps d'exécution et occupation mémoire). Les expérimentations suggèrent que l'occupation mémoire peut être réduite au prix d'un temps d'exécution plus important. De plus, d'autres techniques support pour l'intégration du moniteur semblent envisageables (non basées sur les aspects). Par exemple, la technologie BCEL [The09] d'insertion dynamique de code semble être une bonne candidate. Les bénéfices seraient de réaliser la vérification à l'exécution directement depuis les binaires du programme cible.

CHAPITRE 9

j-POST : a Java toolchain for Property-Oriented Software Testing

Sommaire

9.1 Introduction **159**
9.2 Conception des tests **160**
9.2.1 Exemple 161
9.3 Génération des tests **162**
9.3.1 Vue d'ensemble de la génération de test 162
9.3.2 Analyse syntaxique de l'exigence 162
9.3.3 Implantation de la fonction de génération de test 165
9.3.4 Exemple 167
9.4 Exécution des tests **167**
9.4.1 Vue d'ensemble 167
9.4.2 Principe de fonctionnement 169
9.4.3 Concrétisation des interactions avec l'IUT 170
9.4.4 Exemple 173
9.5 Conclusion et perspectives **174**

Chapter abstract

j-POST is an integrated tool chain for property-oriented software testing. This tool chain includes a test designer, a test generator, and a test execution engine. The test generation is based on an original approach which consists in deriving a set of *communicating test processes* obtained both from a requirement formula (expressed in a trace-based logic) and a behavioral specification of some specific parts of the implementation under test. The test execution engine is then able to coordinate the execution of these test processes against a distributed Java program. A typical application of j-POST is to check the correct deployment of security policies.

Résumé du chapitre

j-POST est une chaîne d'outils intégrés dédiée au test de logiciels orienté par des propriétés. Cette chaîne d'outils comprend un concepteur de test, un générateur de test, et un moteur d'exécution des tests. La génération de test est basée sur une approche originale qui consiste à dériver un ensemble de *processus de test communicants* obtenus à partir d'une exigence formelle exprimée dans un formalisme basé sur les traces et d'une spécification comportementale de certaines parties de l'implantation sous test. Le moteur d'exécution des tests est ensuite capable de coordonner l'exécution de ces processus de test avec un programme Java distribué. Une application typique de j-POST est de vérifier le déploiement correcte de politiques de sécurité.

9.1 Introduction

Dans le Chapitre 7 (et dans [FFMR06, FFMR07a]), nous avons présenté une technique boîte noire de génération de tests capable de construire des cas de test abstraits depuis une exigence (une propriété que le système est supposé satisfaire). Cette méthode, dont l'implantation est présentée dans ce chapitre, est basée sur un calcul de test permettant de définir la méthode formellement et de façon compositionnelle. Dans ce cadre, une exigence est exprimée par une formule logique construite à partir d'un ensemble de prédicats abstraits. Chaque prédicat correspond à une opération possiblement non atomique à réaliser sur l'implantation sous test. Cette opération est founie par l'utilisateur comme un *module de test* indiquant comment réaliser cette opération sur l'implantation réelle, et comment décider si cette exécution a réussi ou non. L'étape de génération de test consiste à construire, par *composition* de modules de test, un ensemble de *processus de test communicants*, depuis la propriété. Dans ce chapitre nous proposons l'implantation de cette technique de test. D'abord, nous présentons formellement comment les tests précédemment générés peuvent être exécutés. De plus, nous présentons j-POST, une chaîne d'outils intégrés dédiée au test orienté par les propriétés.

Une vue d'ensemble du principe de fonctionnement de j-POST est dépeinte sur la Fig. 9.1. L'utilisateur de j-POST veut tester une implantation pour laquelle une interface est disponible (test boite noire). Premièrement, en les fournissant au concepteur de test (étape 1), l'utilisateur élabore une bibliothèque de modules de test abstraits par combinaison des actions offertes par l'interface de l'IUT. Deuxièmement, l'utilisateur définit une (ou plusieurs) exigence(s). Pour cela, il utilise les prédicats élémentaires à disposition dans la bibliothèque. Ces deux données sont utilisées par le générateur de tests (étape 2). Ce qui produit un ensemble de composants de test communicants, qui forme un cas de test pour la propriété. La troisème étape est l'exécution. L'utilisateur peut fournir un objectif de test. Le moteur d'exécution des tests utilise ce dernier pour guider l'exécution du test et sélectionner les exécutions désirées parmis celles contenues dans le cas de test.

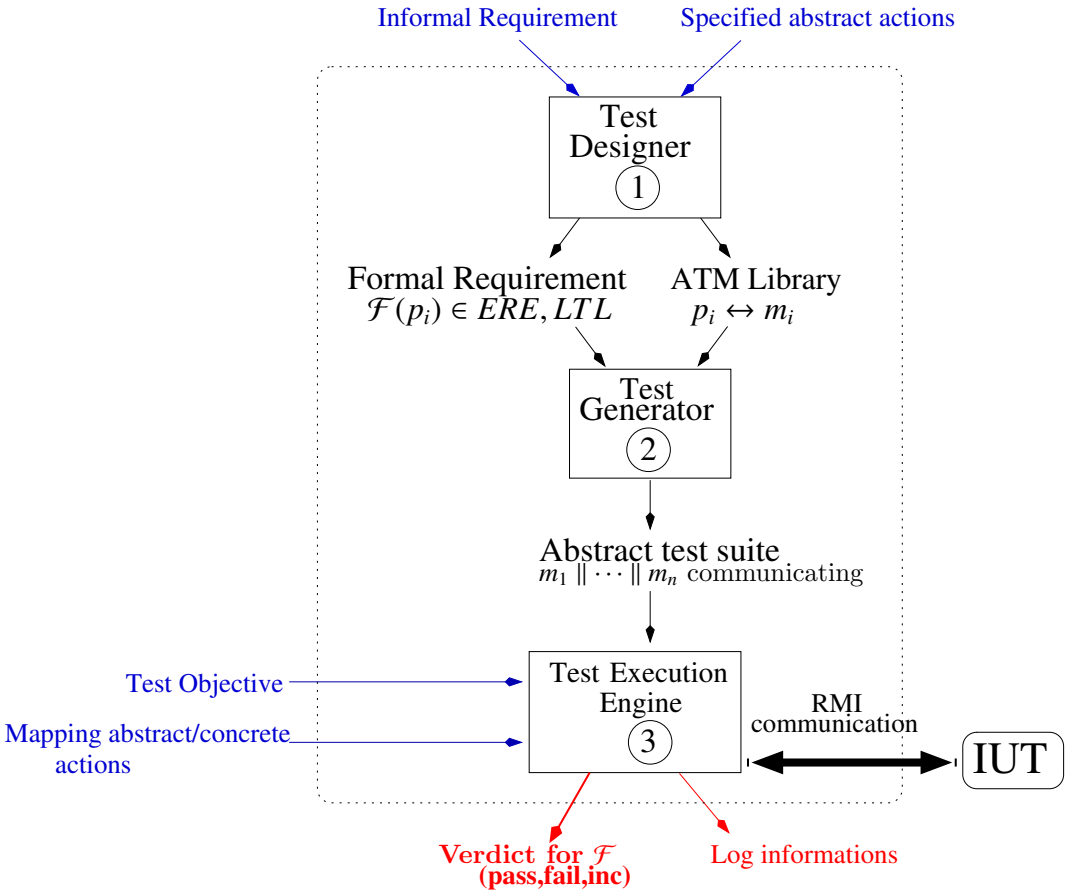


FIGURE 9.1 – Vue abstraite de la chaîne d'outils j-POST

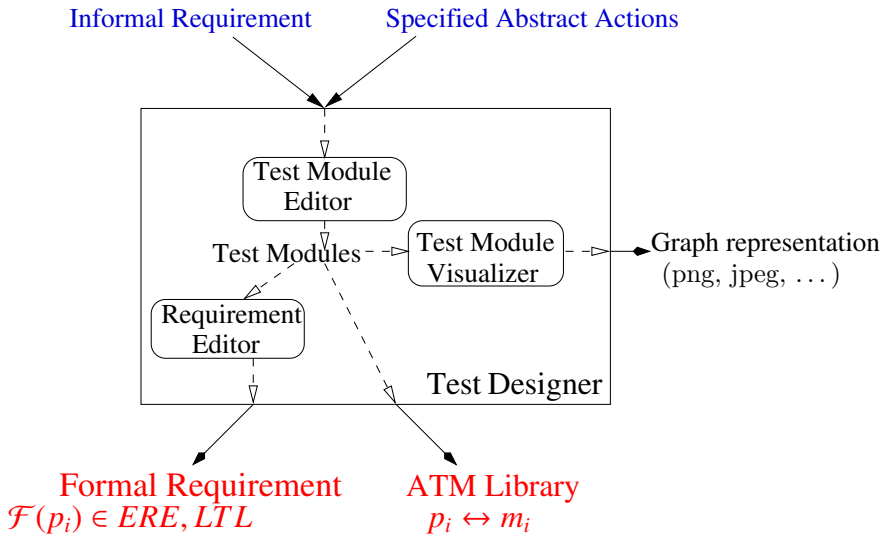


FIGURE 9.2 – Architecture du Test Designer de j-POST

Exemple. Pour illustrer l'utilisation de j-POST, nous utiliserons l'un des exemples qui nous a servi pour l'expérimentation de j-POST : l'application *Travel* et sa politique de sécurité.

L'exemple de l'application Travel vient du projet français RNRT Politess [1]. Les études dans ce projet ont mené à une spécification de cette application. L'application *Travel* permet la gestion de requêtes de missions concurrentes de ses utilisateurs. Dans le but d'enregistrer des requêtes de mission, un utilisateur doit détenir un compte sur l'application. Chaque utilisateur possède son propre valideur (également un utilisateur de Travel) qui est l'unique personne pouvant vérifier l'état de la mission et faire la validation finale. Dans le cas d'une absence prolongée, un utilisateur de *Travel* peut déléguer ses droits d'accès à un autre utilisateur. La personne déléguée peut alors agir au nom de l'utilisateur original.

En étudiant la spécification et le descriptif initial de cette application, nous avons développé une implantation prototype dédiée à l'expérimentation de j-POST. Nous renvoyons à [Fal07] pour une description complète de l'application. L'architecture de l'application est assez simple, elle suit un schéma de type client-serveur. Nous avons implémenté l'application en Java, et utilisé le mécanisme de RMI pour la communication depuis un client vers le serveur applicatif.

Dans ce chapitre, nous illustrerons la conception, la génération, et l'exécution des tests avec j-POST pour tester des propriétés sur *Travel*. L'exigence que nous choisissons pour cette illustration peut être informellement exprimée comme : "il est impossible de valider une mission qui n'a pas été d'abord crée".

Organisation du chapitre. La suite de ce chapitre est organisée comme suit. Dans la Section 9.2, nous présentons la conception de tests avec j-POST. Dans la Section 9.3, nous nous intéressons à la phase de génération de tests. Puis, en Section 9.4, l'exécution des test est présentée. Enfin, la Section 9.5 est consacrée à la conclusion et aux perspectives.

9.2 Conception des tests

Cette section décrit comment il est possible de concevoir les entités de test (une bibliothèque de tests abstraits et une exigence) avec le Test Designer de j-POST. Son architecture est schématisée sur la Fig. 9.2 et une capture d'écran est proposée sur la Fig. 9.3.

L'interface de l'implantation sous test contient un ensemble de variables et méthodes publiques. En les utilisant comme entrées, l'utilisateur écrit une bibliothèque de modules de test, correspondant à des opérations de haut niveau qui peuvent être réalisées sur l'implantation (*e.g.*, "créer une mission" ou la "connexion d'un utilisateur"). Ces composants peuvent être vus comme des termes de l'algèbre de processus décrite dans le Chapitre 7 Section 7.3. Les actions élémentaires peuvent être soit des actions

1. Projet ANR Politess - http://www.rnrt-politess.info/

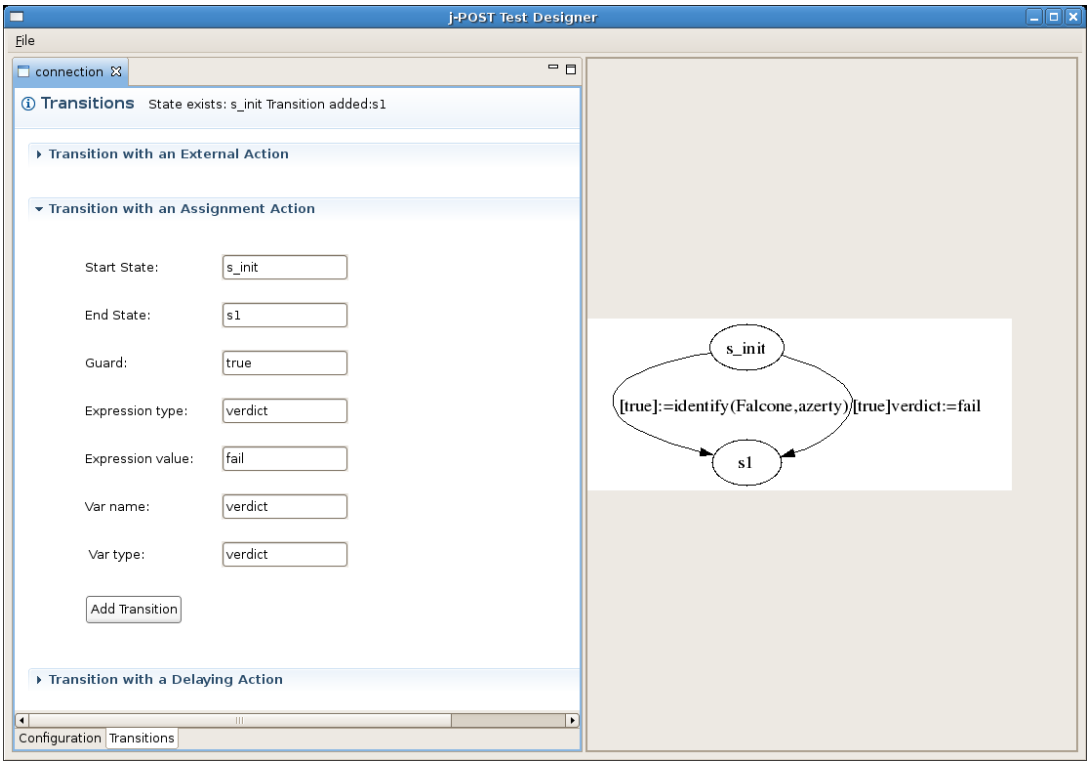


FIGURE 9.3 – Édition d'un module de test avec le Test Designer

internes des composants de test (comme des assignations de variables) ou des appels externes à l'interface de l'IUT. Notons que l'exécution d'une action externe est considérée comme atomique.

Le Test Designer de j-POST fournit une interface (Fig. 9.3) à l'utilisateur pour concevoir les modules de test et les visualiser de manière intelligible. Le format XML a été choisi pour l'encodage interne des modules de test. Nous utilisons l'Eclipse Modeling Framework [2] pour générer l'éditeur de modules de test (Test Module Editor) : l'utilisateur du Test Designer de j-POST peut produire les modules de test en composant les actions paramétrées de l'interface sans manipuler directement le format XML. Un composant du Test Designer permet la visualisation des modules de test comme des systèmes de transitions étiquetés. Cette dernière partie du Test Designer (Test Module Visualizer) utilise GraphViz [3] pour générer les images des modules de test.

9.2.1 Exemple

Nous illustrons la construction de modules de test avec le Test Designer.

Exemple 39 (Modules de test) Les deux exemples de modules de test donnés ci-dessous évaluent des prédicats sur l'application Travel :

- Le module de test *missionCreation* (voir Fig. 9.4) réalise essentiellement un appel à la méthode abstraite `missionRequest2(id, New York,01/12/2008,20/12/2008)`, où `id` est un paramètre indiquant l'identité de l'utilisateur. Cette valeur est interne à l'application *Travel*, et est générée lorsque l'utilisateur ouvre une connexion vers *Travel*. Notons que l'opération `missionRequest2` retourne un `missionNumber` dont la valeur peut être soit un entier positif, auquel cas le verdict retourné est `pass`, ou -1, auquel cas le verdict retourné est `fail`. Les variables `id` et `missionNumber` sont déclarées comme *partagées*, ce qui signifie qu'elle peuvent être lues et assignées par plusieurs modules de tests. Notons que les transitions étiquetées avec `c_start2`, `c_loop2`, `c_ver2` et `c_stop2` sont automatiquement ajoutées par le générateur de test pour gérer l'exécution du test (voir Section 7.4. Ce n'est pas à l'utilisateur d'écrire ces transitions.

2. http://www.eclipse.org/modeling/emf
3. Graph Visualization Software - http://www.graphviz.org

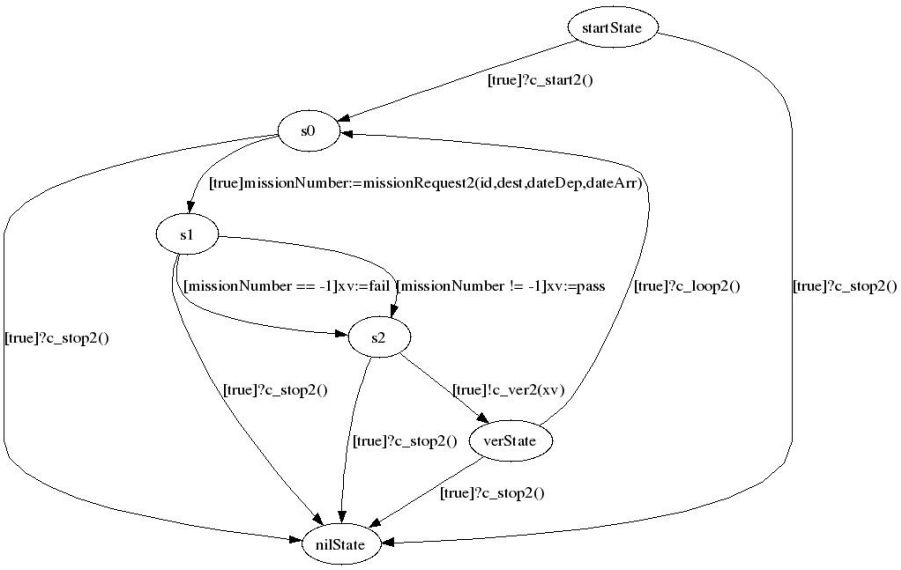


FIGURE 9.4 – Module de test pour le prédicat *missionCreation*

– Le module de test *missionValidation* (Fig. 9.5) réalise un appel à la méthode abstraite `validMission(id, missionNumber)`. Cet appel retourne soit une valeur correcte `ok` auquel cas le verdict résultant est `pass`, soit une valeur incorrecte, auquel cas le verdict est `fail`. Ici également, les transitions étiquetées par `c_start3`, `c_loop3`, `c_ver3` et `c_stop3` sont automatiquement ajoutées par le générateur de test.

9.3 Génération des tests

Cette section décrit le fonctionnement du générateur de test (Test Generator) et le processus de génération de test. L'architecture du Test Generator est représentée sur la Fig. 9.6 et une capture d'écran est proposée sur la Fig. 9.7.

9.3.1 Vue d'ensemble de la génération de test

Le générateur de tests de j-POST consiste principalement en l'implantation de la fonction de génération de test décrite dans le Chapitre 7 Section 7.7. Cette fonction prend pour entrée une exigence formelle, exprimée dans un formalisme logique basé sur les traces. La fonction produit un cas de test complet suivant une approche dirigée par la syntaxe du formalisme.

Des contrôleurs de test et des algorithmes de génération de test ont été définis pour différents formalismes de spécification. Le format interne pour l'encodage des contrôleurs est le XML. Par exemple, sur la Fig. 9.8 et le Listing 9.1 sont donnés respectivement une représentation graphique et l'encodage XML associé pour un contrôleur réalisant la conjonction logique. Pour l'instant j-POST prend en charge les deux formalismes généraux pour lesquels nous avons définis l'approche : une variante de la logique temporelle et les expressions régulières étendues. Toutefois, il est aisé de prendre en charge de nouveaux formalisme. Il suffit pour cela de fournir un plugin logique dédié au formalisme souhaité. Ce plugin consiste à fournir les contrôleurs pour l'ensemble des opérateurs du formalisme.

9.3.2 Analyse syntaxique de l'exigence

La première étape est la construction d'un "arbre de communication" obtenu grâce à l'arbre syntaxique de la formule. Cet arbre représente l'architecture de communication entre les processus de test qui sera produit par le générateur de test. Les feuilles de cet arbre sont les modules de test correspondant aux prédicats atomiques de la formule fournis par l'utilisateur. Les nœuds internes sont des copies de contrôleurs de test génériques correspondant aux opérateurs logiques apparaissant dans la formule. Ils

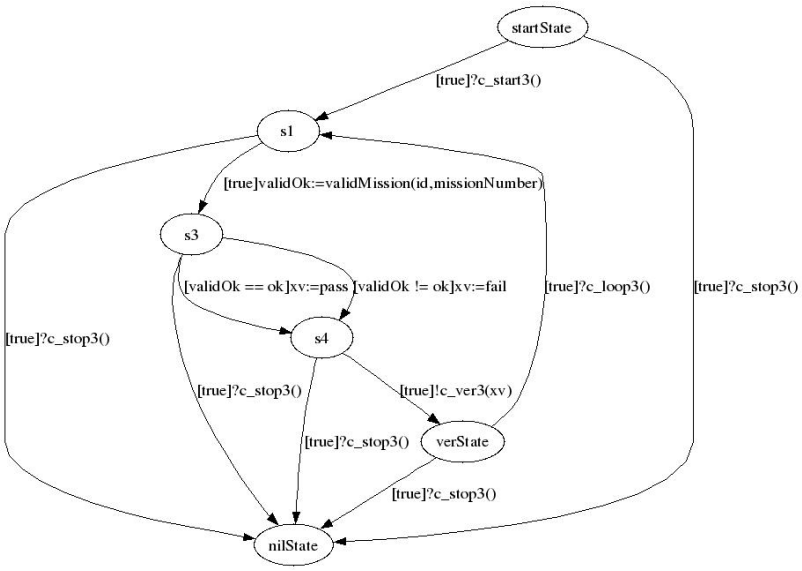


FIGURE 9.5 – Module de test pour le prédicat *missionValidation*

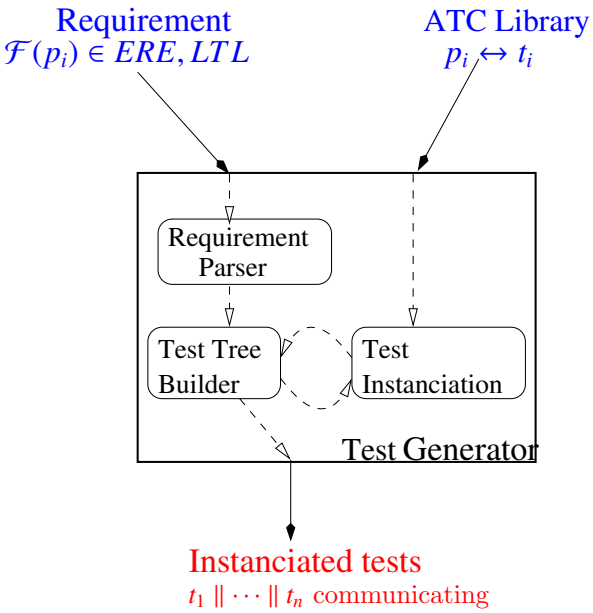


FIGURE 9.6 – Architecture du Test Generator de j-POST

```
<testController id="andController">
 <param id="cs" type="channelSet">
  <channelList startC="c_start" stopC="c_stop" verC="c_ver" loopC="c_loop" />
 </param>
 <param id="cs1" type="channelSet">
  <channelList startC="c_startR" stopC="c_stopR" verC="c_verR" loopC="c_loopR" />
 </param>
 <param id="cs2" type="channelSet">
   <channelList startC="c_startL" stopC="c_stopL" verC="c_verL" loopC="c_loopL" />
  </param>

 <declare id="xv" type="verdict" value="fail" />
 <declare id="xvL" type="verdict" value="fail" />
 <declare id="xvR" type="verdict" value="fail" />

 <state id="s1" initial="true">
        <transition nextState="s2">
                <interaction guard="true" type="reception" scope="communication">
                        <channel>c_start</channel>
                </interaction>
        </transition>
        <transition nextState="stop">
                <interaction guard="true" type="reception" scope="communication">
                        <channel>c_stop</channel>
                </interaction>
        </transition>
 </state>
...
 <state id="s13">
        <transition nextState="s8">
                <interaction guard="true" type="emission" scope="communication">
                        <channel>c_stopR</channel>
                </interaction>
        </transition>
        <transition nextState="stop">
                <interaction guard="true" type="reception" scope="communication">
                        <channel>c_stop</channel>
                </interaction>
        </transition>
  </state>
...
</testController>
```

Listing 9.1 – Extrait du XML du contrôleur pour l'opérateur logique de conjonction

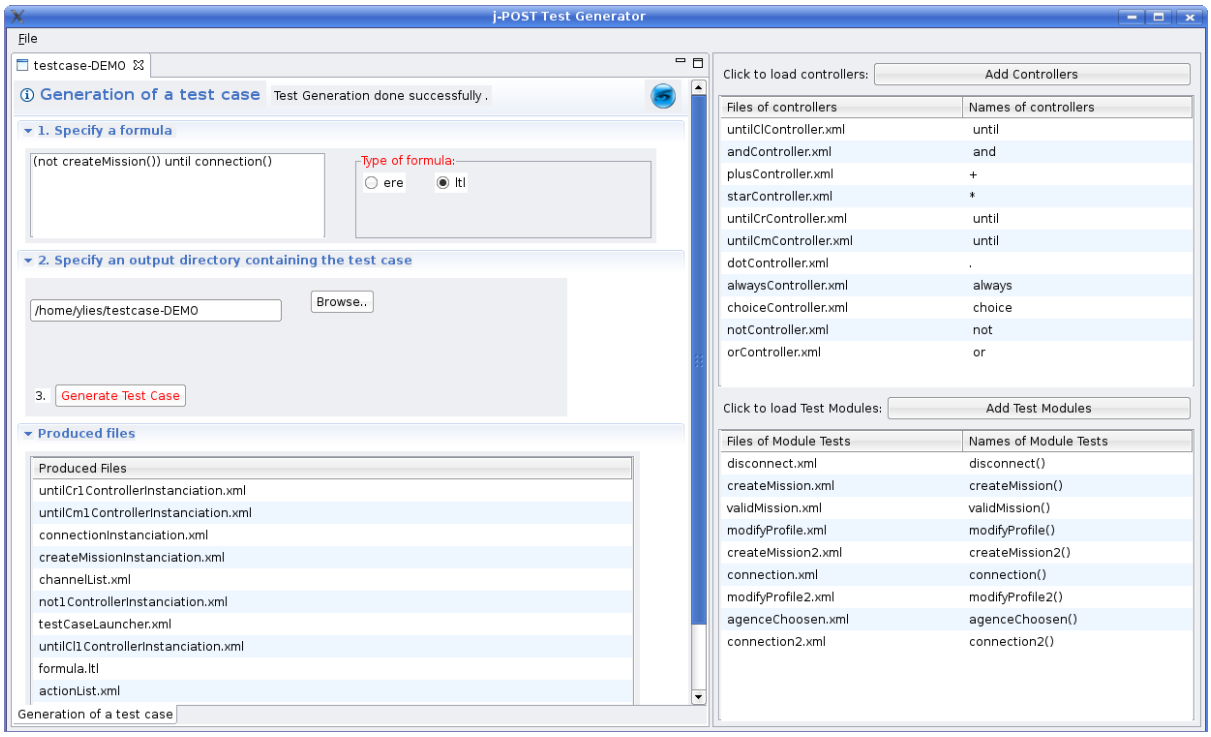


FIGURE 9.7 – Capture d'écran du Test Generator de j-POST

sont obtenus à partir d'un ensemble fini de contrôleurs de test génériques fournis par le plugin logique. Finalement la racine de cet arbre est un processus de test spécial, appelé `testCaseLauncher`, dont le but est d'initialiser l'exécution du test et de délivrer le verdict final (voir Fig. 9.9).

L'analyse de la formule (exprimant l'exigence) est réalisée grâce à un analyseur syntaxique. Pour réaliser l'analyse grammaticale, nous avons choisi Java-CC [4]. Cette bibliothèque génère un ensemble de classes Java par analyse d'un ensemble de règles de grammaire décrivant le langage (voir [FMFR08a] pour plus de détails).

9.3.3 Implantation de la fonction de génération de test

Pour mettre en œuvre le principe de la génération de test (Chapitre 7 Section 7.7), nous construisons un arbre de test syntaxique. Par exemple, si nous considérons la formule LTL : $(p_1() \vee p_2()) \, \mathcal{U} \, p_3(a,b,c)$, l'arbre abstrait est représenté sur la Fig. 9.10. Dans la suite, nous illustrons seulement l'implantation pour la version LTL de *GenTest*, le principe est identique pour les expression régulières. Pour représenter l'arbre en mémoire, nous avons défini un ensemble de classes Java représentant de manière générique la syntaxe d'une exigence (voir [FMFR08a] pour plus de détails).

Comme vu précédemment, les processus de test (prédicats et contrôleurs) sont représentés en XML. Chacun de ces processus est instancié par la fonction *GenTest*, qui modifie leur structure et paramètres. Deux procesus de test instanciés sont représentés sur les Fig. 9.4 et 9.5. Nous avons utilisé JDOM [5], un outil permettant la manipulation et la modification de fichiers XML de données.

Pour chaque formalisme d'expression d'exigences, le principe est le suivant :

- *GenTest* : Crée un contrôleur maître gérant les processus de test au plus haut niveau de l'arbre. Dans un second temps, elle appelle une méthode de génération associée à ce processus (avec un nom de canal de communication frais pour pouvoir l'instancier).
- $GT(F^n(\phi_1, \cdots, \phi_n), cs)$: Prend en paramètre un canal de communication cs et génère un nouveau canal de communication pour chaque opérande de F^n. Le contrôleur de test générique associé à l'opérateur F^n est ensuite instancié avec ces noms de canaux, et la méthode de génération est récursivement appelée sur chaque opérande de F^n avec le nom de canal correspondant en paramètre.

4. http://javacc.dev.java.net
5. http://www.jdom.org

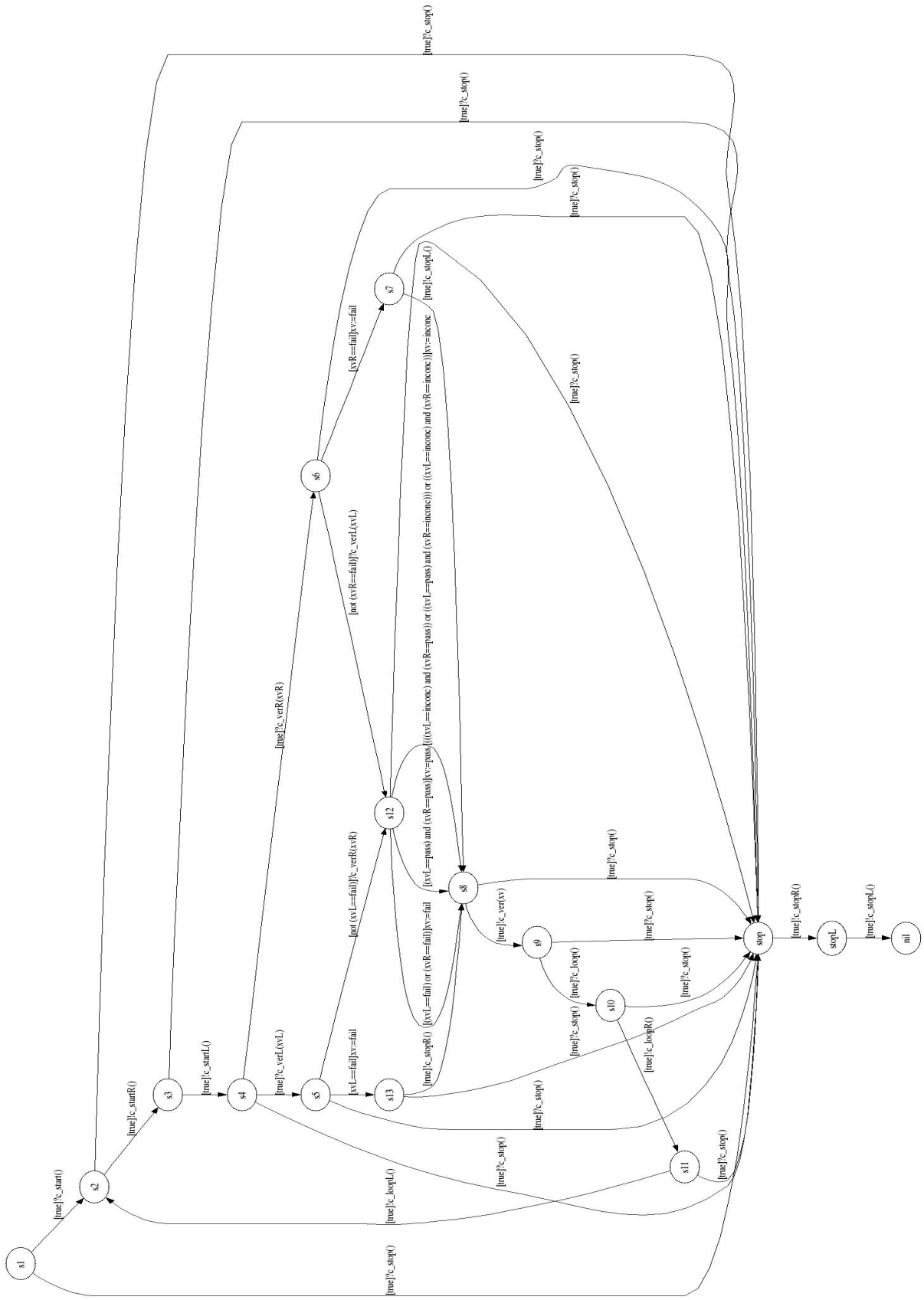


FIGURE 9.8 – Représentation graphique du contrôleur pour l'opérateur logique de conjonction

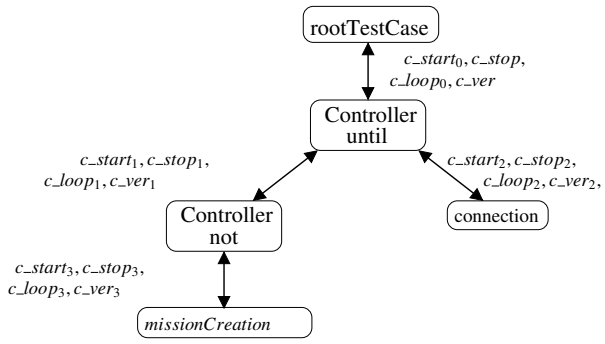


FIGURE 9.9 – Structure du cas de test produit pour $\neg(missionCreation)\,\mathcal{U}\,connection$

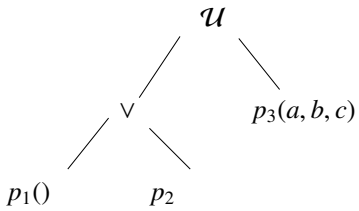


FIGURE 9.10 – Arbre syntaxique abstrait correspondant à $(p_1() \vee p_2())\mathcal{U}p_3(a,b,c)$

– La fonction $Test(t_p, \{c_start, c_stop, c_loop, c_ver\})$ prend un numéro de canal et génère les états de contrôle et les transitions pour ce prédicat. La structure du module de test est ainsi modifiée suivant le principe décrit dans le Chapitre 7 Section 7.7 et schématisé sur la Fig. 7.11, p. 129.

Une fois cette étape terminée, nous obtenons une liste de fichiers XML représentant les processus de test instanciés. Ceux-ci peuvent être utilisés comme entrée pour le moteur d'exécution de test.

9.3.4 Exemple

Pour l'exigence de l'Exemple 39 formalisée par la formule logique $\neg(missionCreation)\,\mathcal{U}\,connection$, et dont les deux modules de test ont été conçus, nous pouvons procéder à la génération du cas de test. Une représentation de "l'arbre de communication" du cas de test obtenu est représentée sur la Fig. 9.9. La structure de ce cas de test suit la syntaxe de la formule. Il contient un contrôleur de test pour chaque opérateur apparaissant dans la formule (*Until* et *Not*), et un module de test pour chaque prédicat (*missionCreation()* et *Connection()*). Le processus `testCaseLauncher` prend en charge la gestion de l'exécution du cas de test et l'émission du verdict final. Le canal *c_start* (resp. *c_stop*, *c_loop*, *c_ver*) est utilisé par le processus pour réaliser les opérations de démarrage (resp. d'arrêt, de redémarrage, et de transmission du verdict).

9.4 Exécution des tests

Cette section décrit comment les test sont exécutés avec le moteur d'exécution de test de j-POST (Test Execution Engine). Son architecture est représentée sur la Fig. 9.11.

9.4.1 Vue d'ensemble

Le but du moteur d'exécution des test est de produire un verdict pour l'exigence testée. Pour cela, le moteur utilise le cas de test produit par le générateur et un *objectif de test*.

Sélection des test en utilisant des objectifs de test comportementaux. Le cas de test produit par le générateur peut décrire plusieurs exécutions. Ceci provient du non-déterminisme possible à la fois dans les modules de test et introduit par la composition parallèle des modules de test. De plus, la fonction de génération de test assure simplement que le verdict produit par l'exécution du test est correcte

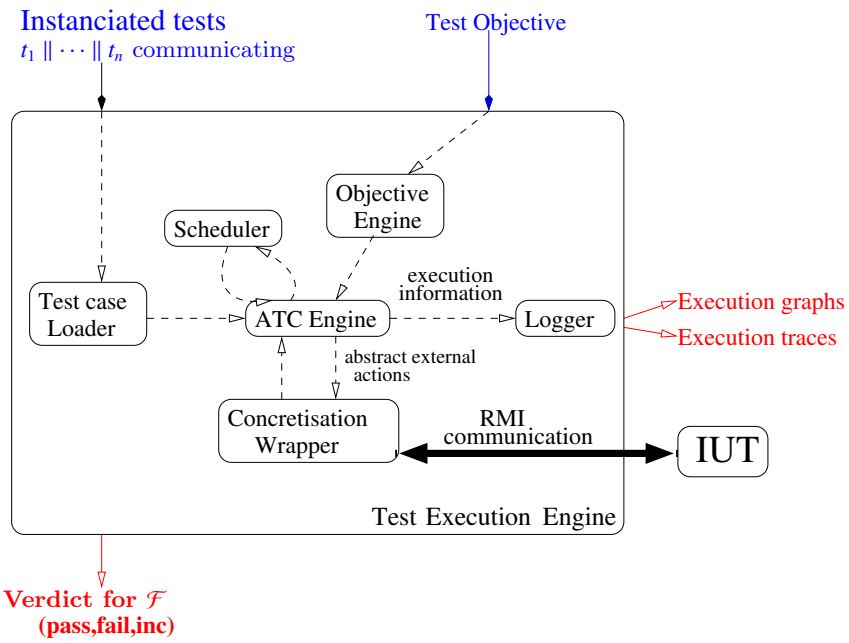


FIGURE 9.11 – Architecture du Test Execution Engine de j-POST

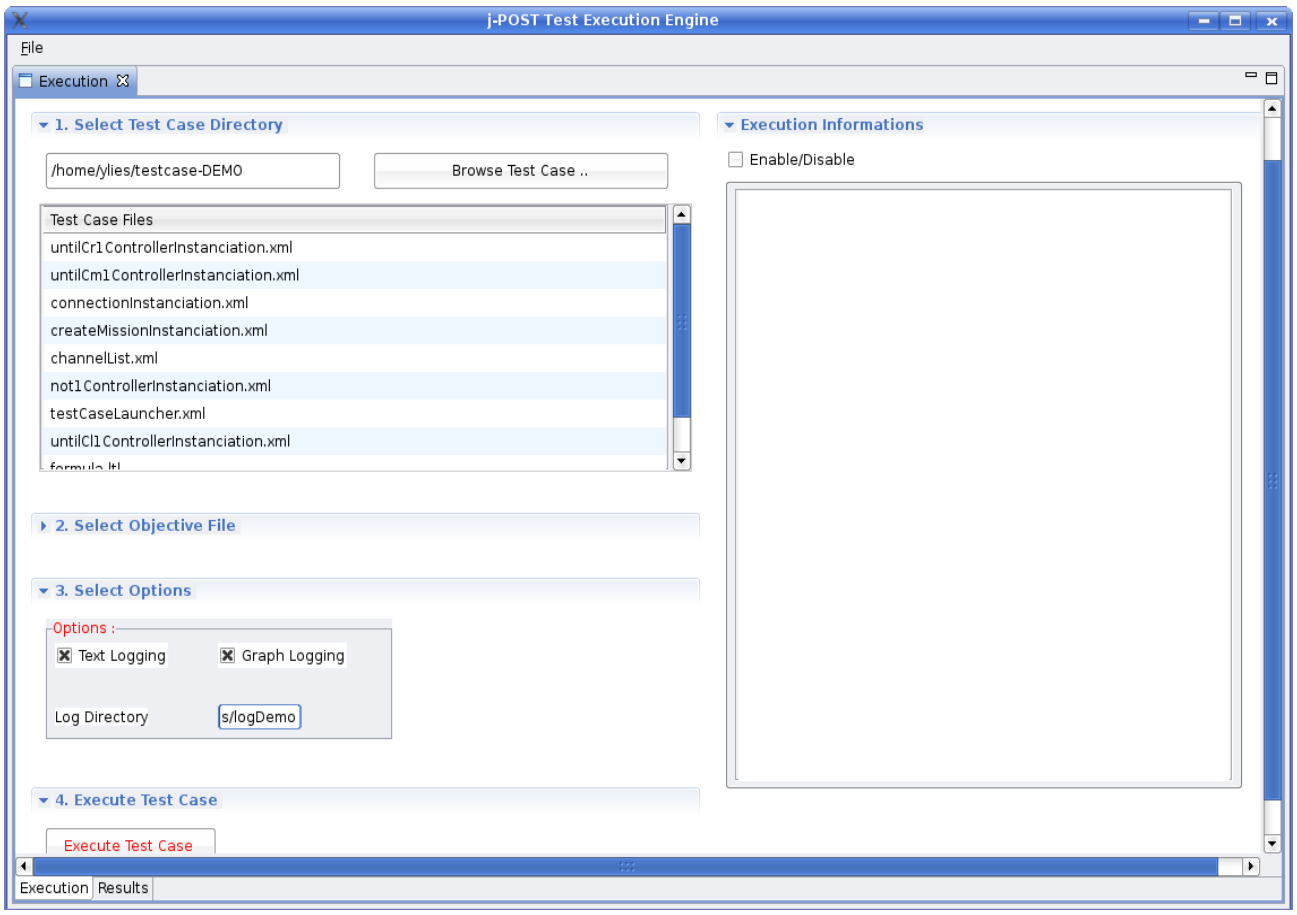


FIGURE 9.12 – Capture d'écran du moteur d'exécution des tests

vis-à-vis de la formule initiale : elle n'aide pas à sélectionner les séquences d'exécution intéressantes qui sont susceptibles de montrer les comportements incorrectes de l'implantation.

Pour résoudre ce problème, nous proposons d'utiliser la notion d'objectif de test comportemental, déjà introduite dans plusieurs outils de génération de test basés sur les modèles (*e.g.*, dans [JJ05, KGHS98]). Leur but est d'injecter des scénarios d'exécution dans le cas de test, soit en forçant un ordre d'exécution pour les actions visibles, soit en introduisant de nouvelles actions visibles pour emmener l'implantation dans un état particulier. Dans la plupart des cas, dans le test basé sur les spécifications, cette sélection des tests est réalisée durant la phase de génération de test, en élaguant les exécutions non désirées de la spécification globale de l'implantation. Dans notre approche, la sélection n'est pas réalisée durant la génération, mais durant l'exécution (similairement à la "walk guidance" de TorX [TB03]). Ceci est dû au fait que nous ne nous basons pas sur une telle spécification. La sélection des test est ainsi combinée à l'exécution du test : l'objectif de test est exprimé par un LTS avec des états d'acceptation, et les séquences de tests menant à ces états sont privilégiées durant l'exécution du test. La politique d'ordonnancement utilisée dans le moteur d'exécution peut être ainsi modifiée au moyen d'un objectif de test. Formellement, un objectif de test est un automate déterministe avec deux états spéciaux nommé "Accept" et "Reject". Leur transitions sont étiquetées soit avec des actions de communication internes soit des actions externes (avec l'IUT). De plus, des priorités entre les actions externes peuvent être spécifiées dans chaque état. Intuitivement, les séquences d'exécution menant vers l'état marqué "Accept" représentent les scénarios de test qui sont attendus, alors que celles menant à l'état marqué "Reject" sont celles non désirées. Notons que les séquences d'exécution contenues dans l'objectif de test peuvent contenir des actions supplémentaires, *i.e.*, des interactions avec l'IUT qui n'étaient pas présentes dans le cas de test. La possibilité d'introduire de telles actions dans l'objectif est conditionnée par la présence de ces actions dans l'interface de l'IUT. Nous renvoyons à [FMFR08b] pour plus de détails concernant les objectifs de test utilisés dans j-POST.

Exécution du test multi-thread avec concrétisation "à la volée". Dès qu'un processus de test doit réaliser une action externe, cette action est concrétisée "à la volée". Pour cela, l'utilisateur fournit un mapping entre les actions externes utilisées dans les modules de test et leur correspondant sur l'interface de l'IUT. En utilisant le mapping, le moteur d'exécution des test concrétise l'action et produit l'appel à l'interace correspondant. Chaque processus de test produit par le générateur de test est exécuté dans un thread Java correspondant. Ce qui permet à l'utilisateur de définir des politiques d'ordonnancement entre chaque type d'interaction.

9.4.2 Principe de fonctionnement

Le but de cet outil est d'exécuter un ensemble de processus de test pour pouvoir décider un verdict validant ou non l'exigence utilisée pour la génération de test. Un ensemble de processus de test communicants est utilisé comme entrée de cet outil. Dans un premier temps, les noms de canaux, et les listes d'actions sont récupérés des processus de test. Dans un second temps, le fichier de chaque processus de test est analysé pour construire le système de transition correspondant en mémoire. Chaque processus de test est modélisé en utilisant un thread Java en mémoire. Ainsi, pour lancer le processus de test, il suffit de lancer le thread correspondant.

Construction et évaluation du système de transition correspondant à un processus de test

Chaque processus de test possède sa propre mémoire contenant un ensemble des variables et paramètres. Un processus est constitué d'un ensemble d'états et de transitions dont les actions sont de plusieurs types. De plus, les processus de test partagent une mémoire commune pour les variables partagées. Pour l'exécution d'une action, on commence par évaluer les gardes des transitions possibles dans l'état courant. Les gardes sont des expressions booléennes dont l'évaluation est réalisée par un analyseur généré par Java-CC. L'exécution de chaque processus de test est régie par la politique d'ordonnancement globale définie dans le moteur d'exécution des tests (voir [FMFR08a] pour plus de détails).

Sélection des tests en utilisant un objectif

Nous décrivons maintenant comment le moteur d'exécution des tests procède pour sélectionner un séquence d'exécution particulière parmis celles contenues dans un cas de test. Plus précisement, nous décrivons comment l'utilisation d'un objectif de test aide à, d'une part prendre en compte des directives

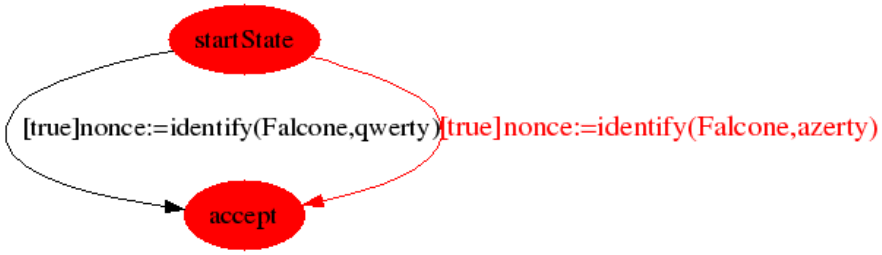


FIGURE 9.13 – Exemple d'objectif de test

d'ordonnancement (internes à l'exécution du cas de test), et d'autre part à enrichir l'exécution du test avec des interactions supplémentaires entre le testeur et l'IUT.

Tout d'abord, rappelons les trois types d'actions qu'un processus de test peut réaliser :

Actions internes Ces actions sont locales aux processus de test. Ce sont typiquement des actions modifiant les données du processus ou des actions de temporisation.

Communications internes Ces actions sont des opérations de synchronisation entre processus de test. Ces actions sont utilisées pour permettre l'échange de signaux (possiblement paramétrés) à travers des canaux de communication. Tous les processus qui souhaitent se synchroniser en utilisant un canal attendent que les autres participants soient au rendez-vous. Plusieurs politiques peuvent être spécifiées (*e.g.*, nombre de processus récepteurs nécessaires). Dans tous les cas, au moins un processus émetteur et un processus récepteur sont nécessaires pour que la communication ait lieu.

Actions externes (au processus de test) Ces actions sont utilisées pour communiquer avec l'IUT, *e.g.*, appel de méthode distant. Une phase de concrétisation est nécessaire pour d'abord traduire l'action abstraite (apparaissant dans un processus de test) vers une action concrète (comme attendue par l'IUT). Cette étape est décrite dans la Section 9.4.3.

Plusieurs politiques d'ordonnancement peuvent être considérées pour favoriser un type d'action. Pour rester modulaire, une classe spécifique encode la politique utilisée. Celle-ci peut donc être modifiée par l'utilisateur. Dans la politique utilisée par défaut, l'ordonnanceur privilégie l'exécution des actions internes parmis les autres, puis c'est au tour des actions externes, et finalement les communication internes au testeur. Les conflits pouvant survenir sont résolus par l'ordonnanceur Java (car chaque module est implanté par un thread). En pratique, chaque processus de test envoie la liste de ses actions possibles à l'ordonnanceur de j-POST, et attend la notification de celui-ci pour réaliser une action.

Les objectifs de test sont utilisés par l'ordonnanceur pour opérer la sélection de test suivant les règles suivantes :

Cas 1 : Certaines actions proposées par les processus de tests correspondent aux étiquettes des transitions sortantes de l'état courant dans l'objectif de test. Dans ce cas, une des actions les plus prioritaires est choisie de manière aléatoire, le processus de test correspondant est notifié de ce choix.

Cas 2 : L'intersection entre les actions proposées par le processus de test et les actions des transitions sortantes de l'état courant de l'objectif est vide. Dans ce cas, il y a deux possibilités :

- s'il existe une autre interaction possible offerte par l'objectif de test dans l'état courant : une des actions les plus prioritaires est choisie de manière aléatoire ;
- sinon : le conflit est résolu par l'ordonnanceur Java par défaut.

À titre d'illustration, les Fig. 9.13 et Fig. 9.14 illustrent la sélection dans un module de test grâce à un objectif. L'objectif de la Fig. 9.13 a permis de sélectionner le chemin exécutant l'action `identify(Falcone,azerty)` en fixant une priorité supérieure à cette action dans l'objectif. Ces figures sont produites par le moteur d'exécution des test qui indique (en rouge) les états visités et les transitions effectuées lors de l'exécution du test.

9.4.3 Concrétisation des interactions avec l'IUT

Les interactions externes (*i.e.*, avec l'IUT) sont représentées sous forme abstraite dans les processus de test. Cette abstraction permet de rendre les modules de test indépendants de l'architecture testée. Une

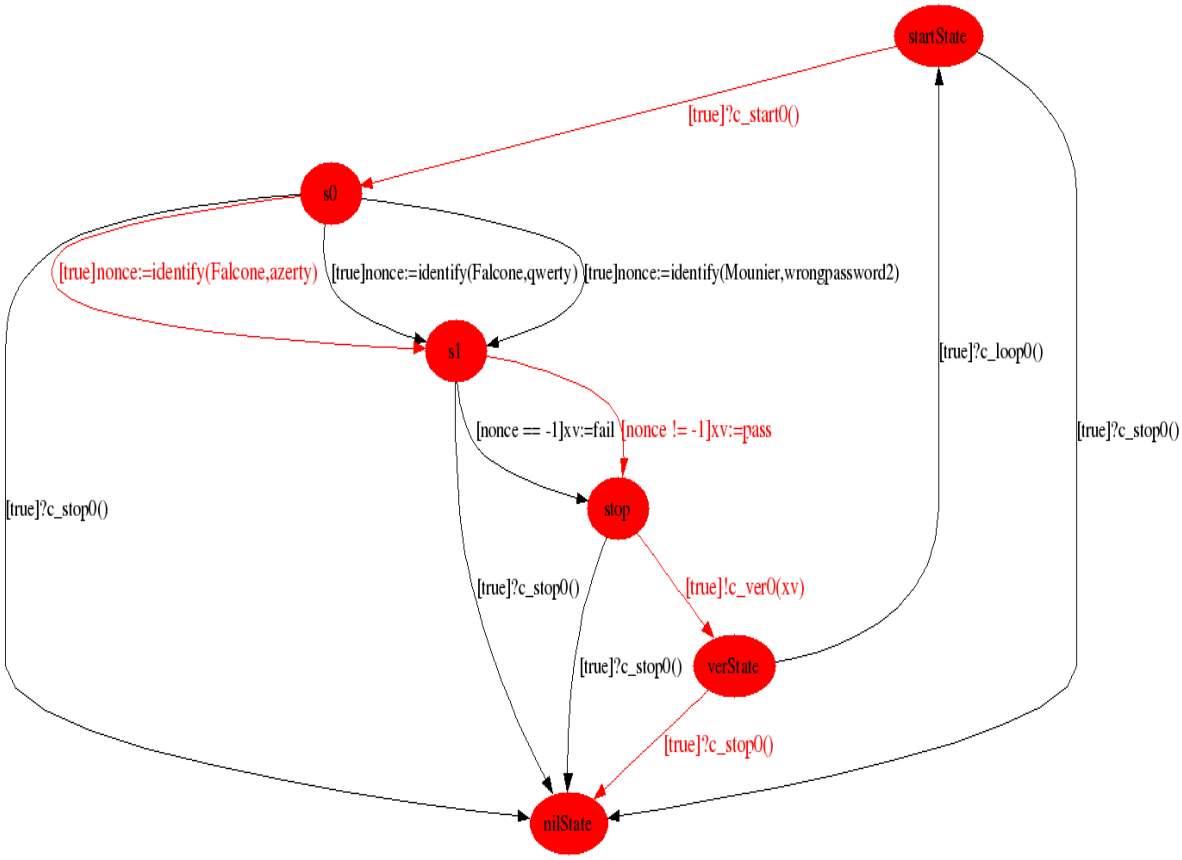


FIGURE 9.14 – Le module de test connection conduit par l'objectif de la Fig. 9.13

```
<interaction guard="true" type="emission_reception" scope="external">
        <call name="identify">
          <param type="string" value="A_Possible_Login" />
          <param type="string" value="A_Password" />
        </call>
        <var name="id" type="integer" />
</interaction>
```

Listing 9.2 – Exemple de codage en XML d'une action externe

```
<mapping>
   <interface name="diag" url="rmi://localhost/" packageName="mappFile" className="Diag">
     <method abstractName="choseAgency" realName="choseAgency" arity="3" >
        <msgReturn concreteValue="Wrong␣number,␣try␣again." abstractValue="notOk" />
        <msgReturn concreteValue="Incorrect␣status␣of␣travel." abstractValue="notOk" />
        <msgReturn concreteValue="Wrong␣number." abstractValue="notOk" />
        <msgReturn concreteValue="Agency␣correctly␣chosen" abstractValue="Ok" />
     </method>
   </interface>
</mapping>
```

Listing 9.3 – Extrait de mapping

phase de concrétisation est donc nécessaire pour rendre ces actions exécutables sur l'IUT. Afin d'être aussi générale que possible, la phase de concrétisation est indépendante du moteur d'exécution. Elle utilise un mapping permettant la traduction des interactions externes. Pour l'instant, la concrétisation des actions en utilisant la technologie Java-RMI [6] est supportée par l'outil. Il serait intéressant d'étudier comment d'autres technologies peuvent être utilisées avec notre moteur d'exécution. Par exemple, la technologie JMS [7] permet l'envoi de messages asynchrones.

Modélisation des actions externes

Les sortes d'actions externes possibles sont :

- actions d'émission : elles correspondent à l'envoie de messages vers l'IUT. Aucune réponse de celui-ci n'est attendue ;
- actions de réception : elles correspondent à la réception de messages provenant de l'IUT ;
- emission-reception : elles correspondent à l'envoie de messages vers l'IUT et ce dernier répond par l'émission d'un message en retour ;

Sur le Listing 9.2 est représenté un exemple de code XML qui décrit une action externe. Pour qu'une action soit déclarée comme externe, l'attribut `scope` doit être déclaré comme `external`. Pour cet exemple, le processus de test effectuera l'action abstraite "*identify*", qui prend deux paramètre de type `string` et attend un retour qui est une valeur de type `integer` qui sera mémorisée dans la variable *id*.

Éléments nécessaires au mapping

Le but du mapping est de traduire les actions abstraites en actions concrètes de manière pratique. Pour cela, la connaissance de l'architecture cible est nécessaire. Pour le moment, j-POST prend en charge le test boite noire en utilisant la technologie RMI. Les éléments nécessaires à la concrétisation sont les noms des méthodes concrètes qui peuvent être appelées. Ce sont les méthodes qui sont disponibles sur l'interface de l'IUT. Le mapping doit donc définir l'interface distante et les méthodes appelables.

Exemple 40 (Extrait de mapping) Sur la Fig. 9.3 est représenté un extrait de fichier de mapping. Dans ce fichier, une interface de nom `diag` est déclarée. L'URL correspond à la localisation de cette interface. Dans la technologie RMI, pour obtenir un objet distant, il est nécessaire d'utiliser un `rmiregistry` : un répertoire listant toutes les interfaces distantes disponibles. À partir du nom de l'interface et de son URL, le rmiregistry retourne une référence sur l'objet distant.

6. http://java.sun.com/docs/books/tutorial/rmi/index.html
7. http://java.sun.com/products/jms/

```
Tester execution started at Wed Jul 02 15:55:29 GMT+01:00 2008
Tester is executing...
Test objective: ACCEPT state
The verdict for the test execution is: PASS
Tester execution finished at Wed Jul 02 15:55:32 GMT+01:00 2008
```

FIGURE 9.15 – Exécution d'un cas de test sur la version ligne de commande du moteur d'exécution

```
//////////////////////////////////////////////
createMission
Path=formulaltl/createMissionInstanciation.xml
//////////////////////////////////////////////
* Starting time: Wed Sep 17 17:22:16 MEST 2008
TRANSITION START
startState;[true]?c_start2()
s0;[true]missionNumber:=
      missionRequest2(id,dest,dateDep,dateArr)
s1;[missionNumber != -1]xv:=pass
s2;[true]!c_ver2(xv)
verState;[true]?c_loop2()
s0;[true]missionNumber:=
      missionRequest2(id,dest,dateDep,dateArr)
s1;[missionNumber != -1]xv:=pass
s2;[true]?c_stop2()
nilState;TRANSITION END
createMission> no possible transition, finishing
* Ending time: Wed Sep 17 17:22:17 MEST 2008

The verdict: PASS
```

FIGURE 9.16 – Fichier de log produit pour le module createMission

Les éléments de type `method` dont déclarés pour l'interface `diag`. Ces éléments correspondent aux méthodes qu'il est possible d'appeler. Dans ces éléments, il est possible d'indiquer le nom abstrait (le nom utilisé dans les processus de test), le nom concret (le nom réel sur l'interface de l'IUT) et l'arité de la méthode. De plus, les valeurs des paramètres peuvent être associées à un mapping. Des valeurs abstraites (utilisée dans les processus de test) peuvent correspondre aux valeurs effectivement retournées par l'IUT.

9.4.4 Exemple

Trois versions de l'application *Travel* ont été testées. Lors de chaque exécution, le moteur d'exécution fournit des messages à l'utilisateur pour l'informer du résultat du test : temps d'exécution, état atteint dans l'objecif, et verdict du test (cf. Fig. 9.15).

- Expérimentation 1. Dans la première version (erronée) de *Travel* une requête de création de mission est toujours acceptée, ainsi notre exigence est fausse : une mission peut être crée par un utilisateur non connecté. Le moteur d'exécution des tests détecte cette erreur : il délivre un verdict *fail* et produit les traces d'exécution pour le cas de test et chaque module. Par exemple, sur la Fig. 9.16 est représenté un fichier de log produit par le moteur d'exécution pour le module de test *createMission*.
- Expérimentation 2. Dans la deuxième version (erronée) de *Travel*, une requête de création de mission est acceptée soit si le numéro d'identification fourni est correct (il correspond à un numéro de session valide), soit si c'est la troisième requête pour créer cette mission. Ainsi, notre exigence est toujours fausse : si un utilisateur non connecté essaye de façon répétée de créer une mission, il finira par réussir. Cette erreur est détectée par le moteur de test qui délivre un verdict *fail*.
- *Expérimentation 3.* Finalement, la troisième version de *Travel* refuse toujours une requête de mission dès que le numéro de mission fourni est incorrect. Ainsi, la seule façon pour un utilisateur non connecté de créer une mission est de "deviner" un numéro d'identification correct. Ceci ne peut être réalisé par notre moteur d'exécution, qui délivre ici un verdict *pass*, déclarant que l'exigence est satisfaite sur cette expérimentation.

9.5 Conclusion et perspectives

Ce chapitre présente l'implantation d'une approche originale pour le test orienté par les propriétés (Property-Oriented Software Testing, POST). Partant d'une exigence formalisée dans une logique basée sur les traces, l'utilisateur produit d'abord un ensemble de modules de test (en utilisant le Test Designer) dédiés à chaque prédicat apparaissant dans la formule. La phase de génération de test consiste ensuite à produire un cas de test comme un ensemble de processus de tests communicants par combinaison des modules de tests où des contrôleurs de tests sont associés à chaque opérateur logique. Ce cas de test peut être ensuite exécuté par un moteur de test, capable de prendre en compte un objectif de test pour contraindre l'ensemble des séquences de test à exécuter. La chaîne d'outils j-POST a été appliquée à une étude de cas non triviale. Son architecture le rend *ouvert*, et permet à la chaîne d'outils de prendre en compte aisément de nouveaux formalismes logiques par ajout de plugins logiques.

L'avantage principal de j-POST réside dans le fait qu'il ne nécessite pas de spécification comportementale "globale". En fait, l'utilisateur doit simplement expliciter comment évaluer des prédicats grâce aux modules de test. La génération de test opérée dans le Test Generator est donc directe et ne souffre pas des limitations dues à l'explosion d'états. Évidemment, les cas de test produits peuvent contenir plusieurs exécutions, mais l'utilisation d'objectifs de test dans le moteur d'exécution permet à l'utilisateur de sélectionner les scénarios les plus intéressants. La chaîne d'outils j-POST nous semble particulièrement adaptée pour le test de sécurité ou de robustesse. Dans ces approches, le modèle fonctionnel de l'IUT peut être assez conséquent (et ainsi pas facilement disponible comme une unique spécification formelle). De plus, les exigences à vérifier peuvent concerner des parties spécifiques de ce modèle. En fait, une des motivation de ce travail était la validation du déploiement correct de politiques de sécurité dans le cadre du projet Français Politess.

L'étude de cas *Travel* a permis plusieurs améliorations pour j-POST et ouvre plusieurs perspectives de recherche. En particulier, il apparaît que la conception des modules de test pourrait être facilitée par l'utilisation de domaines abstraits. Par exemple, au niveau des modules de test, il est nécessaire uniquement de distinguer les mots de passe corrects et incorrects, sans faire référence à une valeur concrète. Cette concrétisation pourrait s'effectuerlors de l'exécution des tests en sélectionnant les valeurs pertinentes dans un domaine concret (qui peuvent dépendre de l'état courant de l'IUT). Cette concrétisation pourrait être réalisée, par exemple, selon des critères de couverture de tests définis par rapport à l'exigence testée. Il nous semble particulièrement intéressant de relier ce travail à [LG02].

Quatrième partie

Conclusions et Perspectives

Conclusion et Perspectives

Ce chapitre clôt le manuscript en dressant un bilan du travail proposé en revenant sur les résultats principaux. Ensuite, nous exposons les perspectives ouvertes par cette thèse.

Bilan

Les méthodes d'analyse dynamiques de programmes, *i.e.*, opérant spécifiquement à l'exécution, sont complémentaires des techniques d'analyse dites statiques. Elles offrent la possibilité de faire face à des problématiques de validation difficilement surmontables par ces dernières : problème de passage à l'échelle, de faux positif, non disponibilité d'une spécification formelle du système à vérifier.

Pour cela, l'étude de cette thèse s'est portée sur trois techniques dynamiques de validation de programmes. Nous avons procédé en trois étapes. Nous avons défini d'abord le cadre général de notre étude. Ensuite dans ce cadre, nous avons proposé une approche pour chaque technique de validation. Enfin, nous avons mis en œuvre ces approches sur les programmes Java.

Cadre général (Partie I). Pour définir le cadre de notre étude, nous nous sommes placés dans la classification *Safety-Progress* des propriétés. Nous avons adapté les résultats (Chapitre 3) de cette classification pour prendre en compte également les séquences finies. En effet, nous avons vu que les méthodes de validation dynamiques de programme reposaient sur l'interprétation des propriétés à valider *sur les séquences finies*. Ces séquences finies sont des représentations abstraites de l'exécution d'un programme vues par un mécanisme de validation (moniteur de vérification, d'enforcement, ou testeur).

Tirant parti de la classification *Safety-Progress*, nous avons étudié l'espace des propriétés qu'il était possible de considérer avec ces trois techniques de validation dynamique de programme (Chapitre 4). Nous avons revisité certains résultats existants et donné de nouvelles caractérisations :

- Pour la technique de vérification à l'exécution (Chapitre 4, Section 4.2), nous avons caractérisé l'espace des propriétés définies comme monitorables. Nous avons ensuite considéré une version multivaluée de cette définition, reposant sur l'idée qu'un moniteur de vérification pouvait être dédié à la détection de validation ou de violation de propriétés. Ensuite, nous avons proposé une définition alternative de propriété monitorable. Nous pensons que cette définition correspond mieux aux besoins pratiques et aux implémentations d'outils de vérification. Les moniteurs de vérification que nous avons proposés produisent une évaluation dans un domaine de vérité allant jusqu'à quatre valeurs. L'introduction de verdicts faibles, à notre sens, permet de mieux représenter les situations qui peuvent se produire lors de l'exécution d'un programme, et qui n'étaient pas prises en compte par les approches de vérification à l'exécution existantes.
- Pour la technique d'enforcement à l'exécution (Chapitre 4, Section 4.3), nous avons donné une caractérisation de l'espace des propriétés enforçables. Cette caractérisation a l'avantage de ne pas utiliser la définition d'un mécanisme d'enforcement particulier. En effet, elle se base sur la définition même de ce qu'est l'enforcement en terme de relation entre une séquence d'entrée (fournie au moniteur) et une séquence de sortie (produite par le moniteur). Ceci donne une borne supérieure à tout mécanisme d'enforcement respectant les contraintes spécifiques de correction et transparence.

– Pour la technique de test (Chapitre 4, Section 4.4), nous nous sommes basés sur une définition existante de testabilité des propriétés. Nous avons corrigé et étendu les résultats existants, dans le sens où nous avons montré que certaines propriétés étaient testables, bien que déclarées non testables précédemment. Similairement à l'introduction de verdicts faibles en vérification à l'exécution, nous avons introduit des verdicts faibles en test. Ces verdicts faibles permettent de raffiner les verdicts *inconclusif*. Lors de la production de verdicts faibles, nous avons identifié les différentes situations pour lesquelles il était possible de produire de tels verdicts ainsi que les hypothèses nécessaires.

Approches pour la validation de propriétés à l'exécution (Partie II). Nous avons ensuite proposé trois applications des techniques de vérification, d'enforcement, et de test de propriétés sur un système lors de son exécution. Nous avons défini des moniteurs de vérification et d'enforcement génériques, étudié la question de leur composition, et avons montré comment obtenir de tels moniteurs depuis des automates de Streett. L'approche de test proposée permet de s'intéresser aux relations entre une r-propriété et les séquences d'exécution d'un programme en l'absence de spécification comportementale complète. Nous avons également décrit comment mettre en œuvre cette approche de manière compositionnelle à partir d'une variante de LTL ou des expressions régulières (Chapitre 7, Section 7.7, et Annexes C.1 et C.2).

Mise en œuvre sur les programmes Java (Partie III). Nous avons proposé deux implantations des approches définies dans cette thèse : j-VETO et j-POST.

La boite à outils j-VETO (Chapitre 8) implémente les approches de vérification et d'enforcement. À notre connaissance, les moniteurs de vérification proposés dans la boite à outils j-VETO sont les premiers moniteurs produisant des valeurs dans un domaine de vérité à quatre valeurs. Nous avons vu que l'utilisation d'un tel domaine plus riche offre l'avantage de mieux comprendre ce qui est vérifié lors de l'exécution du programme.

Nous avons également proposé j-POST (Chapitre 9), une chaîne d'outils dédiée au test de propriétés sur les programmes Java. Cette chaîne d'outils implante l'approche compositionnelle proposée dans le Chapitre 7 Section 7.7.

Perspectives

Retour rapide sur les perspectives proposées dans les chapitres précédents

Nous avons déjà proposés des perspectives à nos travaux dans chaque chapitre. Nous revenons ici sur ces perspectives de façon synthétique.

Prendre en compte une observabilité ou contrôlabilité partielle. Un premier axe de travail serait d'étudier l'impact d'une *observabilité partielle* des événements. Ceci pourrait arriver si la propriété désirée fait référence à des événements hors du cadre d'observation des moniteurs. Il parait assez naturel d'envisager que le moniteur ne puisse pas avoir accès à un certain nombre d'événements du système, *e.g.*, dans le cas d'une approche boite grise ou boite noire. Les ensembles des propriétés vérifiables et enforçables, caractérisés précédemment, seraient sûrement réduits ou augmentés.

Pour faire face à l'observabilité partielle du système, une certaine forme de modèle du système pourrait être utile. Notamment, il nous semble intéressant de combiner l'approche que nous avons proposée avec celles présentées dans [DJM09, JMRT04, MDJ09]. Les auteurs s'intéressent au problème du test, de la vérification à l'exécution, et du contrôle de propriétés sur les systèmes à événements discrets. Les propriétés considérées sont des propriété de sécurité, *e.g.*, propriété de confidentialité, intégrité, contrôle d'accès... Une autre différence est que le système considéré dans ces approches est fourni avec son modèle (complet) dont le comportement est connu. Ce qui permet de traiter une observabilité partielle du système.

Également, sans aller jusqu'à avoir un modèle complet du système, nous pourrions nous appuyer sur une connaissance partielle du système (*e.g.*, une sur-approximation du comportement du système) pour traiter du cas de l'observation partielle. Un autre point important concerne la contrôlabilité des événements. Par exemple, il semble impossible d'empêcher l'occurrence d'événements internes dans le but d'enforcer une propriété. L'idée serait alors de coupler les techniques d'enforcement à celle de la synthèse automatique de contrôleurs développée par exemple dans [DDM08, MDJ09].

Utilisation d'autres domaines de vérité. Pour les propriétés des classes de response et de persistence, la sémantique finitaire des r-propriétés nous semble mériter d'être raffinée. En effet, comme ces propriétés décrivent intrinsèquement des comportements infinis, il est peut être souhaitable que la sémantique permette d'exprimer une certaine "densité" de la validité de la formule p, *e.g.*, représenter la "fréquence" à laquelle p est satisfaite. Ceci reposerait sur un raffinement des opérateurs récurrents (R_f et R) et persistants (P_f et P).

Aussi, il nous semble intéressant d'évaluer de telles formules sur des modèles probabilistes. Plusieurs applications peuvent être envisagées, comme par exemple en ordonnancement pour surveiller l'exécution de processus, et vérifier des propriétés comme l'absence de famine ou l'équité entre les temps d'exécution.

Perspectives générales

Les travaux proposés dans cette thèse nous permettent d'entrevoir plusieurs autres perspectives plus générales.

Autour de la programmation orientée aspect. La technique de la programmation par aspect est devenue, ces dernières années, une technique "support" pour la technique de vérification à l'exécution. Cette technique a grandement facilité l'instrumentation des programmes à vérifier. Également, les besoins de la communauté de la vérification à l'exécution ont contribué à l'enrichissement des langages tels que AspectJ pour Java. Nous envisageons ci-dessous plusieurs perspectives liées à la programmation par aspect.

Des constructions pour l'enforcement. Pour l'observation des événements du programme, le mécanisme de pointcut est utilisé. Ce mécanisme permet d'extraire, pour le moniteur, un événement du flot d'exécution du programme. Il nous semble intéressant d'enrichir les mécanismes de la programmation par aspect avec des constructions qui permettraient à la fois de pouvoir observer les événements du programme, mais également de fournir les fonctionnalités nécessaires pour un moniteur d'enforcement, *e.g.*, mémorisation des informations nécessaires pour "geler" l'événement, et le "rejouer" plus tard.

Aspects et préservation de propriétés. Récemment, Katz [Kat06] s'est intéressé à l'impact des aspects sur la préservation de propriétés dans un programme. Une propriété est préservée par un aspect, si étant vérifiée par le programme original, elle est encore vérifiée dans le programme où l'aspect a été tissé. Ainsi, des classes de propriétés préservées par les aspects sont identifiées. Nous pensons que faire le lien entre l'ensemble des classes de propriétés vérifiables à l'exécution et les catégories d'aspects mis en œuvre pour les vérifier, est une étape pour amener plus de confiance dans le programme augmenté de son moniteur.

Extension à des propriétés non-fonctionnelles. Les approches proposées par cette thèse permettent de vérifier, enforcer, ou tester des propriétés assez générales. Plusieurs adaptations de ces approches nous semblent intéressantes. Opérer à l'exécution du programme permettrait de vérifier la qualité de service d'une application, *e.g.*, temps d'exécution, utilisation des ressources. Utiliser des formalismes de spécification prenant en compte l'aspect temporisé de ce type de spécifications serait surement nécessaire. Notons que plusieurs approches temporisées existent déjà en vérification à l'exécution (*e.g.*, [BLS07b]) et en test (*e.g.*, [KT09]).

Cibler des classes de programmes plus spécifiques. Les méthodes de validation proposées se prêtent bien aux systèmes où le comportement revêt un côté dynamique. Les systèmes évoluants ou reconfigurables (*e.g.*, services Web) lors de l'exécution nous semblent être de bons candidats. Ces classes de programmes pourraient donner des domaines spécifiques aux applications permettant de mener des expérimentations plus soutenues. Ceci participerait à une meilleure validation expérimentale de l'approche proposée.

À noter toutefois que l'approche de vérification et d'enforcement à l'exécution semble être mal adaptées à des système où des contraintes temps réels dur s'appliquent. En effet, il semble difficile de borner le surcoût en terme de temps d'exécution imposé par le moniteur sur l'application.

Bibliographie

[ABB+03] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, June 2003.

[ABH+06] Pavel Avgustinov, Eric Bodden, Elnar Hajiyev, Laurie J. Hendren, Ondrej Lhoták, Oege de Moor, Neil Ongkingco, Damien Sereni, Ganesh Sittampalam, Julian Tibble, and Mathieu Verbaere. Aspects for trace monitoring. In *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*, pages 20–39, 2006.

[AH07] Gail-Joon Ahn and Hongxin Hu. Towards Realizing a Formal RBAC Model in Real Systems. In *SACMAT '07 : Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 215–224, New York, NY, USA, 2007. ACM.

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Softw. Test., Verif. Reliab.*, 13(4) :207–227, 2003.

[AS85] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4) :181–185, October 1985.

[BALT90] Ed Brinksma, Rudie Alderden, Jeroen Langerak, Rom Van de Lagemaat, and Jan Tretmans. A formal approach to conformance testing. In J. De Meer, L. Mackert, and W. Effelsberg, editors, *Second International Workshop on Protocol Test Systems*, pages 349–363. North Holland, 1990.

[Bau08] Andreas Bauer. *The Theory and Practice of Runtime Reflection—A Model-based Framework for Dynamic Analysis of Distributed Reactive Systems.* Number 1. VDM Verlag Dr. Müller, Saarbrücken, 2008.

[BBNS09] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-Finder : A tool for compositional deadlock detection and verification. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 614–619. Springer, 2009.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BDGJ06] Fabrice Bouquet, Frédéric Dadeau, Julien Groslambert, and Jacques Julliand. Safety property driven test generation from JML specifications. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2006.

[Bei90] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold Company, 1990.

[BFHM06] Saddek Bensalem, Jean-Claude Fernandez, Klaus Havelund, and Laurent Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *Proceedings of the 4th*

Workshop on Parallel and Distributed Systems : Testing, Analysis, and Debugging, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), PADTAD 2006, Portland, Maine, USA, July 17, 2006, pages 41–50, 2006.

[BFS04] Axel Belinfante, Lars Frantzen, and Christian Schallhart. Tools for Test Case Generation. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, LNCS 3472, pages 391–438, 2004.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks : Java benchmarking development and analysis. In *OOPSLA '06 : Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[BH05] Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, pages 208–223, 2005.

[BH08] Eric Bodden and Klaus Havelund. Racer : Effective race detection using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA, July 20-24 2008*, pages 155–165, New York, NY, USA, 07 2008. ACM.

[BH09] Eric Bodden and Klaus Havelund. Racer : Effective race detection using AspectJ. *IEEE Transactions on Software Engineering*, December 2009. To appear.

[BHRG07] Howard Barringer, Klaus Havelund, David Rydeheard, and Alex Groce. Rule systems for runtime verification : A short tutorial. In *In Runtime Verification'09*. Springer, 2007.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BLS06] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and N. Garg, editors, *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 4337 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, December 2006. Springer-Verlag.

[BLS07a] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly ? In O. Sokolsky and S. Tasiran, editors, *Proceedings of the 7th International Workshop on Runtime Verification (RV)*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138, Berlin, Heidelberg, November 2007. Springer-Verlag.

[BLS07b] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. Technical Report TUM-I0724, Institut für Informatik, Technische Universität München, December 2007.

[BLS07c] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. Technical Report TUM-I0724, Institut für Informatik, Technische Universität München, December 2007.

[BLS09] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 2009.

[BLW05] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *PLDI '05 : Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM.

[BRH07] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring : From eagle to ruler. *Lecture Notes in Computer Science*, 2839 :111–125, December 2007. ISBN 9783540773.

[Bri88] Ed Brinksma. A theory for the derivation of tests. In *Protocol Specification, testing and verification VIII*, pages 63–74. University of Twente, The Netherlands, 1988.

[BT01] Ed Brinksma and Jan Tretmans. Testing transition systems : An annotated bibliography. In *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195, Berlin, 2001. Springer-Verlag.

[CC92] Patrick Cousot and Rahida Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2-3) :103–179, 1992.

[CCBR06] Frederic Cuppens, Nora Cuppens-Boulahia, and Tony Ramard. Availability enforcement by obligations and aspects identification. In *ARES '06 : Proceedings of the First International Conference on Availability, Reliability and Security (ARES'06)*, pages 229–239, Washington, DC, USA, 2006. IEEE Computer Society.

[CCBS05] Frederic Cuppens, Nora Cuppens-Boulahia, and Thierry Sans. Nomad : A security model with non atomic actions and deadlines. In *CSFW '05 : Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 186–196, Washington, DC, USA, 2005. IEEE Computer Society.

[CDR05] Feng Chen, Marcelo D'Amorim, and Grigore Roşu. Checking and correcting behaviors of Java programs at runtime with java-mop. In *Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 3–20, 2005.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CM05] Severine Colin and Leonardo Mariani. Run-time verification. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-based Testing of Reactive Systems*, volume 3472 of *LNCS*, pages 525–556. Springer Verlag, 2005.

[CMP92a] Edward Chang, Zohar Manna, and Amir Pnueli. The Safety-Progress Classification. Technical report, Stanford University, Dept. of Computer Science, 1992.

[CMP92b] Edward Y. Chang, Zohar Manna, and Amir Pnueli. Characterization of Temporal Property Classes. In *Automata, Languages and Programming*, pages 474–486, 1992.

[CP03] Ivana Cerna and Radek Pelanek. Relating the hierarchy of temporal properties to model checking. In *Proceedings of Mathematical Foundations of Computer Science (MFCS 2003)*, pages 318–327. Springer-Verlag, 2003.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and J. A. Plaice. Lustre : a declarative language for real-time programming. In *POPL '87 : Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[CR94] Judy Crow and John Rushby. Model-based reconfiguration : Diagnosis and recovery. NASA Contractor Report 4596, NASA Langley Research Center, Hampton, VA, may 1994. (Work performed by SRI International).

[CR05] Feng Chen and Grigore Roşu. Java-MOP : A monitoring oriented programming environment for java. In *Proceedings of the Eleventh International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS'05)*, volume 3440 of *LNCS*, pages 546–550. Springer-Verlag, 2005.

[CR07] Feng Chen and Grigore Roşu. MOP : An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, pages 569–588. ACM press, 2007.

[CR09] Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009.

[CŞR08] Feng Chen, Traian Florin Şerbănuţă, and Grigore Roşu. jPredictor : a predictive runtime analysis tool for Java. In *ICSE '08 : Proceedings of the 30th international conference on Software engineering*, pages 221–230, New York, NY, USA, 2008. ACM.

[CTS08] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy : dynamic symbolic execution for invariant inference. In *ICSE '08 : Proceedings of the 30th international conference on Software engineering*, pages 281–290, New York, NY, USA, 2008. ACM.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods : State of the art and future directions. *ACM Computing Surveys*, 28 :626–643, 1996.

[DDM08] Jérémy Dubreil, Philippe Darondeau, and Hervé Marchand. Opacity Enforcing Control Synthesis. In B. Lennartson, M. Fabian, K. Akesson, A. Giua, and R. Kumar, editors, *Proceedings of the 9th International Workshop on Discrete Event Systems (WODES'08)*, pages 28–35, Goteborg, Sweden, May 2008. IEEE.

[DFG+05] Vianney Darmaillacq, Jean-Claude Fernandez, Roland Groz, Laurent Mounier, and Jean-Luc Richier. Eléments de modélisation pour le test de politique de sécurité. In *CRiSIS*, Bourges, 2005.

[DFG+06] Vianney Darmaillacq, Jean-Claude Fernandez, Roland Groz, Laurent Mounier, and Jean-Luc Richier. Test Generation for Network Security Rules. In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference, TestCom 2006, New York, NY, USA, May 16-18, 2006*, volume 3964 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2006.

[DGR04] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12) :859–872, 2004.

[DJM07] Jeremy Dubreil, Thierry Jéron, and Hervé Marchand. Construction de moniteurs pour la surveillance de propriétés de sécurité. In *6ème Colloque Francophone sur la Modélisation des Systèmes Réactifs*, Lyon, France, October 2007.

[DJM09] Jeremy Dubreil, Thierry Jéron, and Hervé Marchand. Monitoring confidentiality by diagnosis techniques. In *European Control Conference*, Budapest, Hungary, August 2009.

[dOWKK07] Anderson Santana de Oliveira, Eric Ke Wang, Claude Kirchner, and Hélène Kirchner. Weaving rewrite-based access control policies. In *ACM Conference on Computer and Communication Security*, November 2007.

[dR05] Marcelo d'Amorim and Grigore Roşu. Efficient monitoring of ω-languages. In *Proceedings of 17th International Conference on Computer-aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 364 – 378. Springer, 2005.

[DRG08] Vianney Darmaillacq, Jean-Luc Richier, and Roland Groz. Test generation and execution for security rules in temporal logic. In *ICSTW '08 : Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 252–259, Washington, DC, USA, 2008. IEEE Computer Society.

[DRP99] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing : introduction, management, and performance.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[dV01] R. G. de Vries. Towards formal test purposes. In Jan Tretmans and Ed Brinksma, editors, *Formal Approaches to Testing of Software 2001 (FATES'01), Aarhus, Denmark*, volume NS-01-4 of *BRICS Notes Series*, pages 61–76, Aarhus, Denkmark, August 2001.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag.

[EFH+03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *CAV*, pages 27–39, 2003.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3) :35–45, December 2007.

[ES00a] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *In IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[ES00b] Ulfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies : A retrospective. In *WNSP : New Security Paradigms Workshop*. ACM Press, 2000.

[Fal07] Yliès Falcone. A Travel Agency Application. Technical report, Vérimag, 2007.

[FCMF09] Ylies Falcone, Alexander Concha, Laurent Mounier, and Jean-Claude Fernandez. j-VETO : a Java Verification and Enforcement Toolbox. Technical Report TR-2009-13, Verimag, Centre Équation, 38610 Gières, Sep 2009.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer : a dynamic atomicity checker for multithreaded programs. In *POPL '04 : Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, New York, NY, USA, 2004. ACM.

[FFM08] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In *ICISS '08 : Proceedings of the 4th International Conference on Information Systems Security*, pages 41–55, Berlin, Heidelberg, 2008. Springer-Verlag.

[FFM09a] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Enforcement monitoring wrt. the safety-progress classification of properties. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 593–600. ACM, 2009.

[FFM09b] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In *RV'09 : Proceedings of the 9th International Workshop on Runtime Verification*. Springer-Verlag, 2009.

[FFM09c] Ylies Falcone, Jean-Claude Fernandez, and Laurent Mounier. Specifying properties in the safety-progress classification for runtime verification. Technical Report TR-2009-5, Verimag Research Report, 2009.

[FFMR06] Yliès Falcone, Jean-Claude Fernandez, Laurent Mounier, and Jean-Luc Richier. A test calculus framework applied to network security policies. In *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*, pages 55–69, 2006.

[FFMR07a] Yliès Falcone, Jean-Claude Fernandez, Laurent Mounier, and Jean-Luc Richier. A compositional testing framework driven by partial specifications. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2007.

[FFMR07b] Ylies Falcone, Jean-Claude Fernandez, Laurent Mounier, and Jean-Luc Richier. A partial-specification driven compositional testing method. Technical Report TR-2007-4, Verimag, Centre Équation, 38610 Gières, Oct 2007.

[FFMR08] Yliès Falcone, Jean-Claude Fernandez, Laurent Mounier, and Jean-Luc Richier. Runtime enforcement monitors : composition, synthesis, and enforcement abilities, 2008. Submitted to FMSD.

[FJJR94] Eddy Fromentin, Claude Jard, Guy-Vincent Jourdan, and Michel Raynal. On-the-fly analysis of distributed computations. In *Research Report 859, Irisa, Campus de Beaulieu 35042 Rennes Cedex*, 1994.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics / Mathematical Aspects of Computer Science*, pages 19–32. AMS, 1967.

[FMFR08a] Ylies Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. j-POST : a Java tool chain for Property-Oriented Software Testing. Technical Report TR-2008-15, Verimag, Centre Équation, 38610 Gières, September 2008.

[FMFR08b] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. j-POST : a Java toolchain for property-oriented software testing. In *Model-Based Testing, MBT*, volume 220, pages 29–41. Electr. Notes Theor. Comput. Sci., 2008.

[FMP03] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. Property oriented test case generation. In *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003*, pages 147–163, 2003.

[Fon04] Philip W. L. Fong. Access control by tracking shallow execution history. In *In Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 43–55. IEEE Computer Society Press, 2004.

[FQ03] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI '03 : Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2003. ACM.

[GH01] Dimitra Giannakopoulou and Klaus Havelund. Runtime analysis of linear temporal logic specifications. Technical report, RIACS, 2001.

[Gra94] Jens Grabowski. SDL and MSC based test case generation– an overall view of the SAMSTAG method. Technical report, University of Berne IAM-94-0005, 1994.

[GVS94] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. An annotated bibliography of interactive program steering. *SIGPLAN Not.*, 29(9) :140–148, 1994.

[Ham06] Kevin W. Hamlen. *Security Policy Enforcement By Automated Program-Rewriting.* PhD thesis, Cornell University, 2006.

[Har00] Jerry J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 331–342, London, UK, 2000. Springer-Verlag.

[Har02] Alan Hartman. Model based test generation tools survey. Technical report, AGEDIS Consortium, 11 2002.

[Hav00] Klaus Havelund. Using runtime analysis to guide model checking of Java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264, London, UK, 2000. Springer-Verlag.

[HBUP03] Hesham Hallal, Sergiy Boroday, Andreas Ulrich, and Alexandre Petrenko. An automata-based approach to property testing in event traces. In *Proceedings of the IFIP TC6/WG6.1 XV International Conference on Testing of Communicating Systems (TestCom 2003), volume 2644 of LNCS*, page 180, 2003.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[Hei98] Constance Heitmeyer. On the need for practical formal methods. In *In Formal Techniques in RealTime and Real-Time Fault-Tolerant Systems, Proc., 5th Intern. Symposium (FTRTFT'98*, pages 18–26. Springer Verlag, 1998.

[HFB93] N. Halbwachs, J.-C. Fernandez, and A. Bouajjanni. An executable temporal logic to express safety properties and its connection with the language lustre. In *Sixth International Symp. on Lucid and Intensional Programming, ISLIP'93, Quebec*, April 1993.

[HG08] Klaus Havelund and Allen Goldberg. Verify your runs. In *Verified Software : Theories, Tools, Experiments : First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 374–383, Berlin, Heidelberg, 2008. Springer-Verlag.

[HLR94] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *AMAST '93 : Proceedings of the Third International Conference on Methodology and Software Technology*, pages 83–96, London, UK, 1994. Springer-Verlag.

[HMS06] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1) :175–205, 2006.

[HMU79] John Hopcroft, , Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4) :366–381, 2000.

[HR01] Klaus Havelund and Grigore Rosu. Monitoring java programs with Java PathExplorer. Technical report, RIACS, 2001.

[HR02] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *TACAS '02 : Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356, London, UK, 2002. Springer-Verlag.

[IEE05] IEEE. Ieee standard 1012-2004 for software verification and validation, 2005.

[ISO96] ISO. ISO/IEC JTC1/SC21 WG7 I.-T. S. Q., Information retrieval, transfer and management for osi ; framework : Formal methods in conformance testing, Technical Report Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500, ISO - ITU-T, Geneve, 1996.

[JJ89] Claude Jard and Thierry Jéron. On-line model checking for finite linear temporal logic specifications. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 189–196. Springer, 1989.

[JJ05] Claude Jard and Thierry Jéron. Tgv : theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(4) :297–315, 2005.

[JJJR94] Thierry Jéron, Claude Jard, Guy-Vincent Jourdan, and Jean-Xavier Rampon. A general approach to trace-checking in distributed computing systems. In *Proc. 14th IEEE Int. Conf. on DCS*, pages 396–403. IEEE, 1994.

[JLD07] Jay Ligatti, Lujo Bauer, and David Walker. Composing expressive run-time security policies. *ACM Transactions on Software Engineering and Methodology*, November 2007.

[JMRT04] Thierry Jéron, Hervé Marchand, Vlad Rusu, and Valéry Tschaen. Ensuring the conformance of reactive discrete-event systems by means of supervisory control. *International Journal of Production Research*, 42(14) :2809–2826, 2004.

[Kam68] Hans Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, 1968.

[Kat06] Shmuel Katz. Aspect categories and classes of temporal properties. *Theoritical Aspect-Oriented Software Development I*, 3880 :106–134, 2006.

[KGHS98] Beat Koch, Jens Grabowski, Dieter Hogrefe, and Michael Schmitt. Autolink : A tool for automatic test generation from sdl specifications. *wift*, 00 :114, 1998.

[KHKT03] Ab Teknillinen Korkeakoulu, Tekniska Hgskolan, Teknillinen Korkeakoulu, and Heikki Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Technical report, Helsinki University of Technology Laboratory for Theoritical Computer Science, 2003.

[Kle51] Stephen Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, Princeton, N.J., 1951.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242. Springer-Verlag, 1997.

[KLT+07] Bohuslav Krena, Zdenek Letko, Rachel Tzoref, Shmuel Ur, and Tomáš Vojnar. Healing data races on-the-fly. In *PADTAD '07 : Proceedings of the 2007 ACM workshop on Parallel and distributed systems : testing and debugging*, pages 54–64, New York, NY, USA, 2007. ACM.

[KLT+09] Bohuslav Krena, Zdenek Letko, Rachel Tzoref, Shmuel Ur, and Tomáš Vojnar. A concurrency testing tool and its plug-ins for dynamic analysis and runtime healing. Technical report, Faculty of Information Technology, Brno University of Technology, 2009.

[KPS08] Marcel Kyas, Cristian Prisacariu, and Gerardo Schneider. Runtime monitoring of electronic contracts. In *In ATVA'08*. Springer-Verlag, 2008.

[KT09] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3) :238–304, 2009.

[KV01] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3) :291–314, 2001.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2) :125–143, 1977.

[Lam83] Leslie Lamport. What good is temporal logic ? In *IFIP Congress*, pages 657–668, 1983.

[LBW05] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing Non-safety Security Policies with Program Monitors. In *ESORICS*, pages 355–373, 2005.

[LBW09] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3) :1–41, January 2009.

[LG02] Grégory Lestiennes and Marie-Claude Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *ISSRE*, pages 3–14. IEEE Computer Society, 2002.

[Lig06] Jarred Adam Ligatti. *Policy Enforcement via Program Monitoring.* PhD thesis, Princeton University, June 2006.

[LKK+99] Insup Lee, Sampath Kannan, Monjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime assurance based on formal specifications. In *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[LS08] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5) :293–303, may/june 2008.

[LY96] David Lee and Mihalis Yannakakis. Principles and Methods of Testing Finite State Machines : a survey. *Proceedings of the IEEE*, 84(8) :1090–1123, Aug 1996.

[Mar03] Nicolas Markey. *Logiques temporelles pour la vérification : expressivité, complexité, algorithmes.* Thèse de doctorat, Laboratoire d'Informatique Fondamentale d'Orléans, France, April 2003.

[Mat07] Ilaria Matteucci. Automated synthesis of enforcing mechanisms for security properties in a timed setting. *Electron. Notes Theor. Comput. Sci.*, 186 :101–120, 2007.

[MDJ09] Hervé Marchand, Jeremy Dubreil, and Thierry Jéron. Génération automatique de tests pour des propriétés de sécurité. In *4ème Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information*, pages 157–174, June 2009.

[MM07] Fabio Martinell and Ilaria Matteucci. Through modeling to synthesis of security automata. *Electron. Notes Theor. Comput. Sci.*, 179 :31–46, 2007.

[MMC08a] Wissam Mallouli, Amel Mammar, and Ana R. Cavalli. Modeling system security rules with time constraints using timed extended finite state machines. In David Roberts, Abdulmotaleb El-Saddik, and Alois Ferscha, editors, *DS-RT*, pages 173–180. IEEE Computer Society, 2008.

[MMC08b] Wissam Mallouli, Gerardo Morales, and Ana Cavalli. Testing security policies for web applications. In *ICSTW '08 : Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 269–270, Washington, DC, USA, 2008. IEEE Computer Society.

[MOC+07] Wissam Mallouli, Jean-Marie Orset, Ana Cavalli, Nora Cuppens, and Frederic Cuppens. A formal approach for testing security rules. In *SACMAT '07 : Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 127–132, New York, NY, USA, 2007. ACM.

[MP71] Robert McNaughton and Seymour A. Papert. *Counter-Free Automata (M.I.T. research monograph no. 65).* The MIT Press, 1971.

[MP84] Zohar Manna and Amir Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Sci. Comput. Program.*, 4(3) :257–289, 1984.

[MP90a] Oded Maler and Amir Pnueli. Tight bounds on the complexity of cascaded decomposition of automata. In *SFCS '90 : Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 672–682 vol.2, Washington, DC, USA, 1990. IEEE Computer Society.

[MP90b] Zohar Manna and Amir Pnueli. A Hierarchy of Temporal Properties (invited paper, 1989). In *PODC '90 : Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 377–410, New York, NY, USA, 1990. ACM.

[MS97] Jean-François Monin and Joseph Sifakis. Eléments de classification des méthodes formelles, Juin 1997.

[MSM07] Patricia D. L. Machado, Daniel A. Silva, and Alexandre C. Mota. Towards property oriented testing. *Electron. Notes Theor. Comput. Sci.*, 184 :3–19, 2007.

[MWC08] Wissam Mallouli, Bachar Wehbi, and Ana R. Cavalli. Distributed monitoring in ad hoc networks : Conformance and security checking. In David Coudert, David Simplot-Ryl, and Ivan Stojmenovic, editors, *ADHOC-NOW*, volume 5198 of *Lecture Notes in Computer Science*, pages 345–356. Springer, 2008.

[Mye04] Glenford J. Myers. *The Art of Software Testing, Second Edition.* Wiley, 2 edition, 2004.

[NGH93] Robert Nahm, Jens Grabowski, and Dieter Hogrefe. Test case generation for temporal properties. Technical report, Bern University, 1993.

[NH83] Rocco De Nicola and Matthew Hennessy. Testing equivalence for processes. In *Proceedings of the 10th Colloquium on Automata, Languages and Programming*, pages 548–560, London, UK, 1983. Springer-Verlag.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6) :89–100, June 2007.

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3) :455–495, 1982.

[PM91] Amir Pnueli and Zohar Manna. *The Temporal Logic of Reactive and Concurrent Systems : Specification : 001.* Springer-Verlag Gmbh, 1 edition, December 1991.

[PMCR08] Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Proceedings of the 29th IEEE Real-Time System Symposium (RTSS'08)*, pages 481–491, 2008.

[PMMD05] Jaime A. Pavlich-Mariscal, Laurent Michel, and Steven A. Demurjian. A formal enforcement framework for role-based access control using aspect-oriented programming. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 537–552. Springer, 2005.

[PMT08] Alexander Pretschner, Tejeddine Mouelhi, and Yves Le Traon. Model-based tests for access control policies. In *ICST '08 : Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 338–347, Washington, DC, USA, 2008. IEEE Computer Society.

[PN81] B. Plattner and J. Nievergelt. Special feature : Monitoring program execution : A survey. *Computer*, 14(11) :76–93, 1981.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *SFCS '77 : Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[PRB09] Charles Pecheur, Franco Raimondi, and Guillaume Brat. A formal analysis of requirements-based testing. In *ISSTA '09 : Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 47–56, New York, NY, USA, 2009. ACM.

[PS07] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *In FMOODS'07, volume 4468 of LNCS*, pages 174–189. Springer, 2007.

[PZ06] Amir Pnueli and Aleksandr Zaks. PSL Model Checking and Run-Time Verification Via Testers. In *FM*, pages 573–586, 2006.

[RKF+08] Frank Rogin, Thomas Klotz, Görschwin Fey, Rolf Drechsler, and Steffen Rülke. Automatic generation of complex properties for hardware designs. In *DATE '08 : Proceedings of the conference on Design, automation and test in Europe*, pages 545–548, New York, NY, USA, 2008. ACM.

[RMJ04] Vlad Rusu, Hervé Marchand, and Thierry Jéron. Verification and Symbolic Test Generation for Safety Properties. Research Report RR-5285, INRIA, 2004.

[Run09] Runtime Verification. http ://www.runtime-verification.org, 2001-2009.

[RWH07] A. Rajan, M.W. Whalen, and M.R. Heimdahl. Model validation using automatically generated requirements-based tests. In *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE*, pages 95–104, Nov. 2007.

[Sai98] Hossein Saiedian. Research directions in formal methods technology transfer. *J. Syst. Softw.*, 40(3) :187–189, 1998.

[Sch95] Beth A. Schroeder. On-line monitoring : A tutorial. *Computer*, 28(6) :72–78, 1995.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1) :30–50, 2000.

[Sif82] Joseph Sifakis. A unified approach for studying the properties of transition systems. *Theor. Comput. Sci.*, 18 :227–258, 1982.

[Sif96] Joseph Sifakis. Research directions for formal methods. In *ACM workshop on Strategic Directions in Computing Research*, June 1996.

[Sis85] A. P. Sistla. On characterization of safety and liveness properties in temporal logic. In *PODC '85 : Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 39–48, New York, NY, USA, 1985. ACM.

[Smi91] Scott Smith. A computational induction principle, 1991.

[Sta97] Ludwig Staiger. ω-languages. Technical report, Martin-luther-universitat Halle-wittenberg, 1997.

[Str81] Robert S. Streett. Propositional dynamic logic of looping and converse. In *STOC '81 : Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 375–383, New York, NY, USA, 1981. ACM.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972.

[TB03] Jan Tretmans and Ed Brinksma. TorX : Automated Model Based Testing - Côte de Resyste. In *Proceedings of the First European Conference on Model-Driven Software Engineering*, pages 13–25, 2003.

[The09] The Apache Jakarta Project. Byte Code Engineering Library. http ://jakarta.apache.org/bcel/, 2009.

[Tho97] Wolfgang Thomas. *Languages, automata, and logic*, pages 389–455. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[TMB07] Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. Testing security policies : Going beyond functional testing. *Software Reliability Engineering, International Symposium on*, 0 :93–102, 2007.

[Tre90] Jan Tretmans. Test case derivation from lotos specifications. In *FORTE '89 : Proceedings of the IFIP TC/WG6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 345–359, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co.

[Tre96] J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.

[Tre97] Jan Tretmans. Repetitive quiescence in implementation and testing. In Adam Wolisz, Ina Schieferdecker, and Axel Rennoch, editors, *FBT*, volume 315 of *GMD-Studien*, pages 23–37. GMD-Forschungszentrum Informationstechnik GmbH, 1997.

[TS07] Nora Cuppens-Boulahia Thierry Sans, Frederic Cuppens. A framework to enforce access control, usage control and obligations. In *Annals of Telecommunications*, 2007.

[TSL04] Li Tan, Oleg Sokolsky, and Insup Lee. Specification-based testing with linear temporal logic. In Du Zhang, Éric Grégoire, and Doug DeGroot, editors, *IRI*, pages 493–498. IEEE Systems, Man, and Cybernetics Society, 2004.

[Tur36] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42) :230–265, 1936.

[vdBRT05] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Action refinement in conformance testing. In F. Khendek and R. Dssouli, editors, *Testing of Communicating Systems (TESTCOM)*, volume 3205 of *Lecture Notes in Computer Science*, pages 81–96. Springer-Verlag, 2005.

[Vis00] Mahesh Viswanathan. *Foundations for the run-time analysis of software systems.* PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 2000.

[VK04] Mahesh Viswanathan and Moonzoo Kim. Foundations for the run-time monitoring of reactive systems - fundamentals of the mac language. In Zhiming Liu and Keijiro Araki, editors, *ICTAC*, volume 3407 of *Lecture Notes in Computer Science*, pages 543–556. Springer, 2004.

[Wol81] Pierre Wolper. Temporal logic can be more expressive. In *SFCS '81 : Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, pages 340–348, Washington, DC, USA, 1981. IEEE Computer Society.

[WRHM06] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In *ISSTA '06 : Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36, New York, NY, USA, 2006. ACM.

[Zuc86] Lenore Zuck. *Past Temporal Logic.* PhD thesis, Weizmann Institute, 1986.

Cinquième partie

Annexes

ANNEXE A

Preuves

Cette annexe regroupe certaines preuves du manuscrit qui ont été différées par soucis de lisibilité.

A.1 Preuves du Chapitre 3

A.1.1 Propriétés des opérateurs E_f et E

Propriété 12 : *Soient ψ_1 et ψ_2 deux propriétés finitaires, les opérateurs E_f et E vérifient les propriétés suivantes lorsqu'ils sont utilisés avec les opérations d'union et d'intersection :*

$$E(\psi_1) \cup E(\psi_2) = E(\psi_1 \cup \psi_2) \qquad E_f(\psi_1) \cup E_f(\psi_2) = E_f(\psi_1 \cup \psi_2)$$
$$E(\psi_1) \cap E(\psi_2) = E(E_f(\psi_1) \cap E_f(\psi_2)) \qquad E_f(\psi_1) \cap E_f(\psi_2) = E_f(E_f(\psi_1) \cap E_f(\psi_2))$$

◇

Les preuves se réalisent facilement en montrant à chaque fois l'inclusion des deux ensembles l'un dans l'autre.

Commençons par montrer $E_f(\psi_1) \cup E_f(\psi_2) = E_f(\psi_1 \cup \psi_2)$.

- Preuve de $E_f(\psi_1) \cup E_f(\psi_2) \subseteq E_f(\psi_1 \cup \psi_2)$. Soit $\sigma \in E_f(\psi_1) \cup E_f(\psi_2)$, alors σ appartient à $E_f(\psi_1)$ ou $E_f(\psi_2)$. Par définition de l'opérateur E_f (cf. Définition 10), cela signifie que σ possède un préfixe qui appartient à ψ_1 ou un préfixe qui appartient à ψ_2. Dans les deux cas, le préfixe appartient à $\psi_1 \cup \psi_2$, ce qui donne le résultat attendu.
- Preuve de $E_f(\psi_1 \cup \psi_2) \subseteq E_f(\psi_1) \cup E_f(\psi_2)$. Soit $\sigma \in E_f(\psi_1 \cup \psi_2)$. C'est-à-dire que σ a un préfixe $\sigma' \in \psi_1 \cup \psi_2$, *i.e.*, $\sigma' \in \psi_1$ ou $\sigma' \in \psi_2$. Il y a donc deux cas possibles. Si $\sigma' \in \psi_1$, alors $\sigma \in E_f(\psi_1)$, puis $\sigma \in E_f(\psi_1) \cup E_f(\psi_2)$. De la même manière, si $\sigma' \in \psi_2$, alors $\sigma \in E_f(\psi_1) \cup E_f(\psi_2)$. Dans les deux cas, nous avons le résultat recherché.

Nous montrons maintenant $E_f(\psi_1) \cap E_f(\psi_2) = E_f(E_f(\psi_1) \cap E_f(\psi_2))$.

- Preuve de $E_f(\psi_1) \cap E_f(\psi_2) \subseteq E_f(E_f(\psi_1) \cap E_f(\psi_2))$. Soit $\sigma \in \Sigma^*$, telle que $\sigma \in E_f(\psi_1) \cap E_f(\psi_2)$. Cela signifie que σ possède deux préfixes, l'un appartenant à ψ_1, l'autre à ψ_2, *i.e.*, $\exists \sigma_1 \in \Sigma^*, \sigma_1 \preceq \sigma \wedge \psi_1(\sigma_1)$ et $\exists \sigma_2 \in \Sigma^*, \sigma_2 \preceq \sigma \wedge \psi_2(\sigma_2)$. Il nous faut montrer que σ a un préfixe σ' qui appartient à $E_f(\psi_1) \cap E_f(\psi_2)$, *i.e.*, que σ' a un préfixe σ'_1 (resp. σ'_2) qui appartient à ψ_1 (resp. ψ_2). Il suffit de prendre $\sigma' = \sigma$ pour obtenir le résultat recherché.
- Preuve de $E_f(E_f(\psi_1) \cap E_f(\psi_2)) \subseteq E_f(\psi_1) \cap E_f(\psi_2)$. Soit $\sigma \in E_f(E_f(\psi_1) \cap E_f(\psi_2))$, cela signifie que σ possède un préfixe σ' qui appartient à $E_f(\psi_1) \cap E_f(\psi_2)$. C'est-à-dire que σ' a un préfixe σ'_1 (resp. σ'_2) qui appartient à ψ_1 (resp. ψ_2). Comme σ'_1 et σ'_2 sont des préfixes de σ', ils sont également des préfixes de σ. Ce qui donne le résultat attendu.

Les preuves pour l'opérateur E de construction de propriétés infinitaires se traitent de manière analogue. La différence est que l'on considère des préfixes stricts.

A.2 Preuves du Chapitre 4

Preuve du Lemme δ. Nous montrons cette propriété par l'absurde. Supposons l'existence d'une r-propriété $\Pi = (\phi, \varphi)$ de reactivity définie sur Σ qui n'est ni une safety ni une guarantee : $\Pi \in \text{Reactivity}(\Sigma) \setminus (\text{Safety}(\Sigma) \cup \text{Guarantee}(\Sigma))$ et qui soit monitorable au sens de la Définition 25 avec $\mathbb{B}_3$.

Comme $\Pi \in MP^*(\mathbb{B}_3)$, nous avons par définition que :

$$\forall \sigma_{good} \in \phi, \forall \sigma_{bad} \in \overline{\phi}, [\![\Pi]\!]_{\mathbb{B}_3}(\sigma_{good}) \neq [\![\Pi]\!]_{\mathbb{B}_3}(\sigma_{bad})$$

Remarquons que $\phi \neq \emptyset$ et $\overline{\phi} \neq \emptyset$ car Π n'est ni une safety ni une guarantee. En effet, si $\phi = \emptyset$, alors Π serait forcément la r-propriété toujours fausse, qui est une safety. De même, si $\overline{\phi} = \emptyset$, *i.e.*, $\phi = \Sigma^*$, Π serait la r-propriété toujours vraie, qui est une safety également.

Nous considérons alors deux séquences σ_{good} et σ_{bad} dans Σ^∞, comme suit :

- Soit $\sigma_{good} \in \phi$ telle qu'il existe $\sigma'_g \in \Sigma^\infty$ avec $\neg\Pi(\sigma_{good} \cdot \sigma'_g)$. Nous savons qu'une telle séquence existe car $\Pi \notin \text{Guarantee}(\Sigma)$. Ceci est une conséquence de la Propriété 1 (p. 40).
- Soit $\sigma_{bad} \in \overline{\phi}$ telle qu'il existe $\sigma'_b \in \Sigma^\infty$ avec $\Pi(\sigma_{bad} \cdot \sigma'_b)$. Nous savons qu'une telle séquence existe car $\Pi \notin \text{Safety}(\Sigma)$. Ceci est une conséquence de la Propriété 1 (p. 40).

D'après la définition de la fonction d'évaluation d'une r-propriété dans un domaine de vérité (Définition 24, p. 70), nous avons que :

$$[\![\Pi]\!]_{\mathbb{B}_3}(\sigma_{good}) = [\![\Pi]\!]_{\mathbb{B}_3}(\sigma_{bad}) =?$$

Ce qui est une contradiction avec $\Pi \in MP^*(\mathbb{B}_3)$.

A.3 Preuves du Chapitre 5

Preuve de la Propriété 7. Les équivalences suivantes ont été prouvées dans le Chapitre 5.

$$q_{n-1} \in Good^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_4}(\sigma) = \top,$$
$$q_{n-1} \in Good_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_4}(\sigma) = \top_p,$$

Il nous reste donc à prouver les deux dernières :

$$q_{n-1} \in Bad_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_4}(\sigma) = \bot_p,$$
$$q_{n-1} \in Bad^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!]_{\mathbb{B}_4}(\sigma) = \bot.$$

Preuve de $q_{n-1} \in Bad^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!](\sigma) = \bot$.

- Supposons que $q_{n-1} \in Bad^{\mathcal{A}_\Pi}$. En utilisant le critère d'acceptation sur les séquences finies, nous avons que σ n'est pas acceptée par $\mathcal{A}_\Pi$. De plus, comme $\mathcal{A}_\Pi$ reconnaît Π, nous avons que $\neg\Pi(\sigma)$. Maintenant, considérons $\mu \in \Sigma^+$ *t.q.* $|\sigma| + |\mu| = n' > n$ et $run(\sigma \cdot \mu, \mathcal{A}_\Pi) = q_0 \cdots q_{n'-1}$. Comme $q_{n-1} \in Bad^{\mathcal{A}_\Pi}$, nous avons que $\forall k \in \mathbb{N}, n \leq k \leq n'-1 \Rightarrow q_k \in \bigcup_{i=1}^m \overline{R_i} \cap \overline{P_i}$ et par conséquent $\neg\Pi(\sigma \cdot \mu)$. Considérons $\mu \in \Sigma^\omega$, nous pouvons remarquer que $\forall i \in [1, m], vinf(\sigma \cdot \mu, \mathcal{A}_\Pi) \cap R_i = \emptyset \wedge vinf(\sigma \cdot \mu, \mathcal{A}_\Pi) \not\subseteq P_i$, ce qui implique que $\neg\Pi(\sigma \cdot \mu)$. Nous avons $\neg\Pi(\sigma) \wedge \forall \mu \in \Sigma^\infty, \neg\Pi(\sigma \cdot \mu)$, *i.e.*, $[\![\Pi]\!](\sigma) = \bot$.
- Réciproquement, supposons que $[\![\Pi]\!](\sigma) = \bot$. Par définition, cela signifie $\forall \mu \in \Sigma^\infty, \neg\Pi(\sigma \cdot \mu)$. Selon le critère d'acceptation des automates de Streett, nous déduisons que $\forall k \geq n, \forall \mu \in \Sigma^*, run(\sigma \cdot \mu, \mathcal{A}_\Pi) = q_0 \cdots q_{n-1} \cdots q_k \Rightarrow q_k \in \bigcup_{i=0}^m \overline{R_i} \cap \overline{P_i}$. C'est-à-dire $Reach_{\mathcal{A}_\Pi}(q_{n-1}) \subseteq \bigcup_{i=1}^m (\overline{R_i} \cap \overline{P_i})$, *i.e.*, $q_{n-1} \in Bad^{\mathcal{A}_\Pi}$.

Preuve de $q_{n-1} \in Bad_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!](\sigma) = \bot_p$. La preuve de $q_{n-1} \in Bad_p^{\mathcal{A}_\Pi} \Leftrightarrow [\![\Pi]\!](\sigma) = \bot_p$ est directe en examinant le critère d'acceptation des séquences finies des automates de Streett.

- Supposons que $q_{n-1} \in Bad_p^{\mathcal{A}_\Pi}$. En utilisant le critère d'acceptation des séquences finies, nous avons que σ n'est pas acceptée par $\mathcal{A}_\Pi$. De plus, comme $\mathcal{A}_\Pi$ spécifie Π, nous avons que $\neg\Pi(\sigma)$. Maintenant, comme $Reach_{\mathcal{A}}(q) \not\subseteq \bigcup_{i=1}^m (\overline{R_i} \cup \overline{P_i})$, il existe un état q' de $\mathcal{A}_\Pi$ atteignable depuis q et appartenant à $\bigcap_{i=1}^m (R_i \cup P_i)$. En conséquence, il existe $\mu \in \Sigma^*$ *t.q.* $run(\sigma \cdot \mu) = q_0 \cdots q_{n-1} \cdots q'$. Suivant toujours le critère d'acceptation, nous déduisons que $\Pi(\sigma \cdot \mu)$, *i.e.*, $[\![\Pi]\!](\sigma) = \bot_p$.

– Réciproquement, le même raisonnement peut être mené en utilisant le critère d'acceptation pour prouver le résultat recherché.

A.4 Preuves du Chapitre 6

A.4.1 Preuve de la transformation spécifique aux r-propriétés de guarantee

Nous notons $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, {q_{\text{init}}}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, (R, \emptyset))$. Considérons une séquence d'exécution du programme $\sigma \in Exec(\mathcal{P}_\Sigma)$. Nous étudions l'effet de la soumission de σ à $\mathcal{A}_{\downarrow\Pi}$. Nous allons, comme précédemment, associer l'exécusion de σ sur $\mathcal{A}_\Pi$ à l'exécution de σ sur $\mathcal{A}_{\downarrow\Pi}$.

L'exécution de σ sur $\mathcal{A}_\Pi$ produit une trace $(q_0, \sigma_0, q_1) \cdot (q_1, \sigma_1, q_2) \cdots (q_i, \sigma_i, q_{i+1}) \cdots$ ce qui correspond à la trace $(q_0, \sigma_0/\alpha_0, q_1) \cdots (q_i, \sigma_i/\alpha_i, q_{i+1}) \cdots$ sur $\mathcal{A}_{\downarrow\Pi}$ avec $q_0 = {q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi}}$. Nous distinguons encore selon que la séquence σ satisfait la propriété Π ou non.

Le premier cas est $\Pi(\sigma)$. Nous savons que l'automate $\mathcal{A}_\Pi$ accepte σ, distinguons selon que σ soit finie ou non.

– Si $\sigma \in \Sigma^*$, alors $\phi(\sigma)$. Soit $n = |\sigma|$. Comme σ est acceptée par $\mathcal{A}_\Pi$, et selon le critère d'acceptation pour les séquences finies (Définition 17, p. 50), nous avons que en lisant σ, il existe un état $q \in R$ atteignable depuis ${q_{\text{init}}}^{\mathcal{A}_\Pi}$, depuis lequel on reste dans les états R (nous avons $P = \emptyset$ et pas de transition depuis les états R vers les états R comme $\mathcal{A}_\Pi$ est un automate de guarantee).
  Si $\sigma = \epsilon$, alors nous avons facilement (6.22) comme $\epsilon \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$. De plus, $Pref_{\prec}(\phi, \epsilon) = \emptyset$, ce qui donne (6.24).
  Sinon ($\sigma \neq \epsilon$), selon les contraintes de la relation de transition d'un automate de guarantee, le run et la trace de σ sur $\mathcal{A}_\Pi$ sont tels qu'il existe $0 \leq k \leq n$ *t.q.* $\forall i \leq k \leq n \cdot q_i \in \overline{R} \wedge \forall n \geq i > k \cdot q_i \in R$.
  Selon (TGuar1) et (TGuar2), la trace de σ sur $\mathcal{A}_{\downarrow\Pi}$ est *t.q.* $\forall i < k \cdot \alpha_i = store \wedge \forall i \geq k \cdot \alpha_i = dump$.
  Depuis la trace d'exécution sur $\mathcal{A}_{\downarrow\Pi}$ et la définition des opérations d'enforcement, nous déduisons les dérivations de configurations suivantes :
  – $({q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi}}, \sigma, \epsilon) \overset{\epsilon}{\hookrightarrow} (q_1, \sigma_{1\cdots}, \sigma_0) \cdots \overset{\epsilon}{\hookrightarrow} (q_{k-1}, \sigma_{k-1\cdots}, \sigma_{\cdots k-2}) \overset{\epsilon}{\hookrightarrow} (q_k, \sigma_{k\cdots}, \sigma_{\cdots k-1})$ comme $\forall i < k, store(\sigma_i, \sigma_{\cdots i-1}) = (\epsilon, \sigma_{\cdots i})$;
  – $(q_k, \sigma_{k\cdots}, \sigma_{\cdots k-1}) \overset{\sigma_{\cdots k}}{\hookrightarrow} (q_{k+1}, \sigma_{k+1\cdots}, \epsilon) \cdots \overset{\sigma_{n-2}}{\hookrightarrow} (q_{n-1}, \sigma_{n-1}, \epsilon) \overset{\sigma_{n-1}}{\hookrightarrow} (q_n, \epsilon, \epsilon)$ comme $dump(\sigma_k, \sigma_{\cdots k-1}) = (\sigma_{\cdots k}, \epsilon)$ et $\forall i > k \cdot dump(\sigma_i, \epsilon) = (\sigma_i, \epsilon)$.
  Par déduction, en utilisant la dérivation multi-pas, nous avons $({q_{\text{init}}}^{\mathcal{A}_{\downarrow\Pi}}, \sigma, \epsilon) \overset{\epsilon}{\Rightarrow} (q_k, \sigma_{k\cdots}, \sigma_{\cdots k-1})$ et $(q_k, \sigma_{k\cdots}, \sigma_{\cdots k-1}) \overset{\sigma}{\Rightarrow} (q_n, \epsilon, \epsilon)$. C'est-à-dire $\sigma_{\cdots k-1} \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$, et ensuite $\sigma_{\cdots k} \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma_k$, et enfin $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma$. Ce qui assure (6.22). Par ailleurs, selon le critère d'acceptation des r-propriétés, nous avons $\phi(\sigma)$, ce qui nous permet de déduire (6.23), car $\sigma = \sigma$.
– Si $\sigma \in \Sigma^\omega$, et alors $\varphi(\sigma)$. En utilisant la Définition 17 (p. 50) et la définition d'un automate de guarantee, nous avons $vinf(\sigma) \cap R \neq \emptyset$. Ce qui signifie qu'il existe un état de R qui est visité infiniment souvent. Comme il n'existe pas de transition depuis les états de R vers les états de $\overline{R}$, les états de $\overline{R}$ sont visités au plus un nombre fini de fois. C'est-à-dire, nous pouvons trouver un indice k et deux états q, q' *t.q.* $q \xrightarrow{\sigma_k}_{\mathcal{A}_{\downarrow\Pi}} q'$ avec $q \notin R \wedge q' \in R$. Le run de σ sur $\mathcal{A}_\Pi$ est *t.q.* $\forall i < k \cdot q_i \notin R \wedge \forall i \geq k \cdot q_i \in R$. Ce qui par un raisonnement similaire nous mène à trouver la trace de σ sur $\mathcal{A}_{\downarrow\Pi}$. La séquences des opérations d'enforcement est alors $(store)^{k-1} \cdot (off)^\omega$. Il s'en suit que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma$. Nous avons alors (6.22) et (6.23).

Le deuxième cas est $\neg\Pi(\sigma)$. La séquence σ n'est pas acceptée par $\mathcal{A}_\Pi$. Comme $\neg\varphi(\sigma)$ et Π est une r-propriété de guarantee, nous avons $Pref_{\prec}(\phi, \sigma) = \emptyset$. En effet, comme Π est une r-propriété de guarantee, l'existence d'un préfixe de σ satisfaisant φ aurait impliqué que $\varphi(\sigma)$.

Selon les contraintes d'un automate de guarantee, $\mathcal{A}_\Pi$ démarre et termine dans un état de $\overline{R}$ et reste dans $\overline{R}$ (il n'existe pas de transition depuis $q \in R$ vers $q' \notin R$). Nous déduisons que la trace d'exécution de σ sur $\mathcal{A}_\Pi$ est *t.q.* $\forall i > 0 \cdot q_i \notin R$. En utilisant la définition de la transformation TransGuarantee nous pouvons trouver $trace(\sigma, \mathcal{A}_{\downarrow\Pi})$. Ensuite, l'opération d'enforcement réalisée par $\mathcal{A}_{\downarrow\Pi}$ est toujours *halt* ou *store*. Il existe deux formes possibles pour la séquences des α_i : soit $\forall i > 0 \cdot \alpha_i = store$, ou $\exists k > 0 \cdot (\forall i \leq k \cdot \alpha_i = store \wedge \forall k > i \cdot \alpha_i = halt)$. Dans les deux cas, en utilisant les définitions de *halt* et *store*, nous pouvons trouver facilement que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$ (6.22).

Donc, comme $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$, nous avons (6.22) et (6.24).

A.4.2 Preuve de la transformation spécifique aux r-propriétés de response

La preuve suit le même principe que la preuve pour les r-propriétés de guarantee. Intuitivement, la preuve peut être comprise comme suit. Lorsqu'une séquence satisfait une r-propriété de response, il existe une alternance de satisfaction entre les préfixes de cette séquence. Lorsqu'une séquence ne satisfait pas une r-propriété de response, il existe un rang à partir duquel le run de cette séquence est composé de "mauvais états" pour toujours.

Nous notons $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}{}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, (R, \emptyset))$. Considérons une séquence d'exécution du programme $\sigma \in Exec(\mathcal{P}_\Sigma)$. Nous étudions l'effet de la soumission de σ à $\mathcal{A}_{\downarrow\Pi}$. Nous allons, comme précédemment, associer l'exécution de σ sur $\mathcal{A}_\Pi$ à l'exécution de σ sur $\mathcal{A}_{\downarrow\Pi}$. L'exécution de σ sur $\mathcal{A}_\Pi$ produit une trace $(q_0, \sigma_0, q_1) \cdot (q_1, \sigma_1, q_2) \cdots (q_i, \sigma_i, q_{i+1}) \cdots$ qui correspond à une trace $(q_0, \sigma_0/\alpha_0, q_1) \cdots (q_i, \sigma_i/\alpha_i, q_{i+1}) \cdots$ sur $\mathcal{A}_{\downarrow\Pi}$ avec $q_0 = q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}$. Nous distinguons encore selon que la séquence σ satisfait la r-propriété Π ou non.

Le premier cas est $\Pi(\sigma)$. Nous savons que l'automate $\mathcal{A}_\Pi$ accepte σ, distinguons selon que σ soit finie ou non.

- Si $\sigma \in \Sigma^*$, et alors $\phi(\sigma)$. Soit $n = |\sigma|$. Selon le critère d'acceptation des séquence finies (Définition 17, p. 50), le run de σ sur $\mathcal{A}_\Pi$ termine dans un état R.
  Si $\sigma = \epsilon$, alors nous avons facilement (6.22) car $\epsilon \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$. De plus, $Pref_{\prec}(\phi, \epsilon) = \emptyset$, ce qui nous donne (6.24).
  Sinon ($\sigma \neq \epsilon$), selon les contraintes de la fonction de transition d'un automate de response, le run et la trace de σ sur $\mathcal{A}_\Pi$ sont tels que $q_n \in R$. Selon (TGUAR1), la trace de σ sur $\mathcal{A}_{\downarrow\Pi}$ est telle que $\alpha_n = dump$. Depuis la trace d'exécution de l'exécution sur $\mathcal{A}_{\downarrow\Pi}$ et la définition des opérations d'enforcement, nous déduisons les dérivations de configurations suivantes :

$$(q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}, \sigma, \epsilon) \overset{o_0}{\hookrightarrow} (q_1, \sigma_{1\cdots}, m_1) \cdots \overset{o_{n-2}}{\hookrightarrow} (q_{n-2}, \sigma_{k-1\cdots}, m_{n-1}) \overset{o_{n-1}}{\hookrightarrow} (q_n, \epsilon, \epsilon)$$

  avec $o_0 \cdot o_1 \cdots o_{n-1} = \sigma$ du fait que la dernière opération d'enforcement (α_{n-1}) est *dump* ou *off*.
  Par déduction, en utilisant la relation de dérivation à plusieurs pas, nous avons $(q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}, \sigma, \epsilon) \overset{\sigma}{\Rightarrow} (q_n, \epsilon, \epsilon)$. C'est-à-dire $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma$. Ce qui assure (6.22). Par ailleurs, selon le critère d'acceptation des r-propriétés, nous avons $\phi(\sigma)$, ce qui permet de déduire (6.23), car $\sigma = \sigma$.
- Si $\sigma \in \Sigma^\omega$, alors selon la définition d'un automate de response, nous trouvons en utilisant le critère d'acceptation que $vinf(\sigma, \mathcal{A}_\Pi) \cap R \neq \emptyset$. En d'autres termes, il existe un état $q \in Q^{\mathcal{A}_\Pi} \cap R$ visité infiniment souvent. Comme il n'y a pas de restriction sur la fonction de transition de $\mathcal{A}_\Pi$, le run de σ sur $\mathcal{A}_\Pi$ contient quelques états dans $\overline{R}$, quelques états dans R, mais visite q infiniement souvent. Formellement, $\forall i \in \mathbb{N}, \exists j \in \mathbb{N}, j \geq i \wedge q_j \in R$. Il s'en suit que la trace de σ sur $\mathcal{A}_\Pi$ vérifie $\forall i \in \mathbb{N}, \exists j \in \mathbb{N}, j \geq i \wedge (q_{j-1}, \sigma_{j-1}, q_j) \in trace(\sigma, \mathcal{A}_\Pi) \wedge q_j \in R$. Alors nous pouvons en déduire que la trace sur le moniteur d'enforcement $\mathcal{A}_{\downarrow\Pi}$ (en utilisant la définition de TransResponse, Définition. 53) vérifie la propriété : $\forall i \in \mathbb{N}, \exists j \in \mathbb{N}, j \geq i \wedge (q_{j-1}, \sigma_{j-1}/dump, q_j) \in trace(\sigma, \mathcal{A}_{\downarrow\Pi})$. C'est-à-dire : $\forall i \in \mathbb{N}, \exists j \in \mathbb{N}, j \geq i, \alpha_j = dump$. Ainsi, cela nous permet de déduire (en utilisant la Propriété 9, p. 104) que $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma$, *i.e.*, (6.22). De plus, nous avons (6.23) car $\varphi(\sigma) \wedge \sigma = \sigma$.

Le deuxième cas est $\neg\Pi(\sigma)$. La séquence σ n'est pas acceptée par $\mathcal{A}_\Pi$, distinguons selon que σ soit finie ou non.

- Si $\sigma \in \Sigma^*$ et alors $\neg\phi(\sigma)$. Soit $n = |\sigma|$. Il y a deux cas dépendant du fait que $Pref_{\prec}(\phi, \sigma) = \emptyset$ ou non.
  - *Si* $Pref_{\prec}(\phi, \sigma) = \emptyset$. Selon le critère d'acceptation des automates de Streett de response, $\mathcal{A}_\Pi$ démarre dans un état dans $\overline{R}$ et reste dedans. Nous en déduisons que la trace d'exécution de σ sur $\mathcal{A}_\Pi$ est *t.q.* $\forall i > 0 \cdot q_i \notin R$. En utilisant la définition de la transformation TransResponse nous pouvons trouver $trace(\sigma, \mathcal{A}_{\downarrow\Pi})$. Ensuite, l'opération d'enforcement réalisée par $\mathcal{A}_{\downarrow\Pi}$ est toujours *halt* ou *store*. C'est-à-dire $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \epsilon$ (6.22). Par suite $Pref_{\prec}(\phi, \sigma) = \emptyset$ implique que $\forall \sigma' \prec \sigma \cdot \neg\phi(\sigma)$. Nous avons (6.24).
  - *Sinon* ($Pref_{\prec}(\phi, \sigma) \neq \emptyset$). Il y a au moins un préfixe de σ satisfaisant ϕ. Notons σ_{good} le plus long préfixe de σ satisfaisant ϕ, *i.e.*, $\sigma_{good} = Max(Pref_{\prec}(\sigma, \phi))$. Soit $k = |\sigma_{good}|$. Alors le run et la trace de $\mathcal{A}_\Pi$ sur σ est *t.q.* $q_{k-1} \in R \wedge \forall i \in [k, n-1], q_i \in \overline{R}$. Selon la définition de la transformation TransResponse, la trace de σ sur $\mathcal{A}_{\downarrow\Pi}$ est *t.q.* $\alpha_{k-1} = dump \wedge \forall i \in [k, n-1], \alpha_i \in \{store, halt\}$. A partir de la trace d'exécution sur $\mathcal{A}_{\downarrow\Pi}$ et la définition des opérations d'enforcement, nous déduisons les dérivations de configurations suivantes :

$$(q_{\text{init}}{}^{\mathcal{A}_{\downarrow\Pi}}, \sigma, \epsilon) \overset{o_0}{\hookrightarrow} \cdots \overset{o_{k-2}}{\hookrightarrow} (q_{k-1}, \sigma_{(k-1)\cdots}, m_{k-1}) \overset{o_{k-1}}{\hookrightarrow} (q_k, \sigma_{k\cdots}, \epsilon)$$

$$(q_k, \sigma_{k\cdots}, \epsilon) \overset{\epsilon}{\hookrightarrow} (q_{k+1}, \sigma_{k+1\cdots}, m_{k+1}) \overset{\epsilon}{\hookrightarrow} \cdots \overset{\epsilon}{\hookrightarrow} (q_n, \epsilon, m_n)$$

avec $\sigma_{good} = \sigma_{\cdots(k-1)} = o_0 \cdot o_1 \cdots o_{k-1}$. En effet, nous avons $dump(\sigma_{k-1}, m_{k-1}) = (m_{k-1} \cdot \sigma_{k-1}, \epsilon)$ et $\forall i \geq k \cdot \alpha_i \in \{store, halt\}$, α_i fait que $\mathcal{A}_{\downarrow\Pi}$ produit ϵ en sortie (pour $k \leq i \leq n-1$). C'est-à-dire $\sigma_{\cdots k-1} \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma_{\cdots k-1}$ et $\sigma \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma_{\cdots k-1}$. Ce qui assure (6.22). Par ailleurs, selon le critère d'acceptation des r-propriétés, nous avons $\neg\phi(\sigma)$, ce qui permet de déduire (6.25), car $\sigma_{\cdots k-1} = Max(Pref_{\prec}(\phi, \sigma))$.

– Si $\sigma \in \Sigma^{\omega}$ et alors $\neg\varphi(\sigma)$. Ce cas est similaire au cas $\neg\varphi(\sigma)$ pour les r-propriétés de guarantee. Le critère d'acceptation pour les automates de response implique que $vinf(\sigma, \mathcal{A}_{\Pi}) \cap R = \emptyset$. Nous en déduisons qu'il existe un entier n *t.q.* le run de σ sur $\mathcal{A}_{\Pi}$ s'exprime comme $run(\sigma, \mathcal{A}_{\Pi}) = q_0 \cdots q_n \cdots$ avec $q_0 = q_{\text{init}}{}^{\mathcal{A}_{\Pi}} \wedge (\forall i \geq n \cdot q_i \in \overline{R})$. Considérons n_{min} le plus petit entier n vérifiant cette propriété. Pour $k \leq n_{min}$, il est possible d'appliquer le raisonnement précédent (celui du cas $\phi(\sigma)$) pour $\sigma_{\cdots k}$. Donc nous trouvons une alternance dans le run de la séquence d'exécution $\sigma_{\cdots n_{min}}$ entre les états appartenant à R et $\overline{R}$. Nous trouvons de manière similaire que pour $k > n_{min}$, $\sigma_{\cdots k} \Downarrow_{\mathcal{A}_{\downarrow\Pi}} \sigma_{\cdots n_{min}}$ et $\phi(\sigma_{\cdots n_{min}})$. Il est facile de voir que $\sigma_{\cdots n_{min}}$ est le plus long préfixe (par définition de n_{min}) satisfaisant ϕ, *i.e.*, $\sigma_{\cdots n_{min}} = Max(Pref_{\prec}(\phi, \sigma_{\cdots k}))$.

ANNEXE B

Usage of j-VETO and j-POST

B.1 Usage of j-VETO

This section describes tool usages. All our tools accept the following options.

-h, - ? : shows the specific tool usage and lists all the offered options.
-i, –input : specifies the input file or directory required by a tool.
-o, –output : specifies the output file or directory.
-r, –recursive : forces a tool to traverse directories recursively. That is, it processes all the relevant files of the input directory.
-v, –verbose : shows additional information messages.

B.1.1 DFA2Streett

Usage of the DFA2Streett tool can be obtained by invoking the help directive to the tool (option `-?` or `-h`). A screenshot is shown below.

```
> java -jar DFA2Streett.jar -h

Usage: java -jar DFA2Streett.jar options
Option                                   Description
------                                   -----------
-?, -h                                   show help
-i, --input <File: file or directory
  containing DFA automata>
-o, --output [File: output file/dir]
-p, --property-type <safety,
  guarantee, response, persistence>
-r, --recursive                          traverse directories recursively
-v, --verbose                            print additional messages

Required options:
-----------------
-i [DFA path] and -p [property]

Example:
--------
java -jar DFA2Streett.jar -i ./DFA-Automaton.xml -p safety
```

The parameter *-p* or *–property-type* specifies which type of pattern (safety, guarantee, response, persistence) should be used to generate the Streett Automaton.

B.1.2 streett2Monitor

Usage of the streett2Monitor tool can be obtained by invoking the help directive to the tool (option `-?` or `-h`). A screenshot is shown below.

```
> java -jar streett2Monitor.jar -h

Usage: java -jar streett2EM.jar options
Option                                  Description
------                                  -----------
-?, -h                                  show help
-i, --input <File: path to Streett
  automata>
-o, --output [File: output file/dir]
-r, --recursive                         traverse directories recursively
-t, --monitor-type <The type of
  monitor (VM for Verification Monitor
  and EM for enforcement monitor) that
  should be generated.>
-v, --verbose                           print additional messages

Required options:
-----------------
-i [path to Streett automata]

Example:
--------
java -jar streett2EM.jar -i ./Streett-Automaton.xml
```

The parameter *-t* or *–monitor-type* specifies the type of monitor that should be generated from the Streett automaton.

B.1.3 compositionEM

Usage of the compositionEM tool can be obtained by invoking the help directive to the tool (option `-?` or `-h`). A screenshot is shown below.

```
Usage: java -jar composition.jar options

EM     ->  Enforcement Monitor
<File> ->  List of files
[File] ->  Single File

Option                                  Description
------                                  -----------
-?, -h                                  shows help
-i, --input <File>                      the two files which contain EM's
-o, --output [File]                     output file
--op, --operation <File>                file which contains operation
-r, --reduction                         removes unreachable states from final
                                          EM
-s, --separator                         separator for separating the composed
                                          state Id
-v, --verbose                           prints additional messages

Required options:
-----------------
-i [EM's path], -op [Operation File]

Examples:
--------
java -jar composition.jar -i ./em1.xml ./em2.xml -op union.txt
java -jar composition.jar -i ./em1.xml -i ./em2.xml -o ./em.xml -op intersection.txt
```

The parameter *–op* or *–operation-type* specifies the file that contains the operator definition and how the monitors should be combined.

The parameter *-r* or *–reduction* forces the tool remove possible unreachable states left once the composition has been done.

The parameter *-s* or *–separator* specifies the string that is used to concatenate the states in the combined monitor.

B.2 Usage of j-POST

This section descibres usages of j-POST tools.

B.2.1 Test generator

The option list can be obtained by launching the executable JAR file and providing the `--help` option.

```
ruitor:~/workspace/jpost2/dist/testGenerator> java -jar testGenerator.jar --help
Usage : java -jar testGenerator.jar options
         -in,-input The input path of the requirement
         -out,-output The output path
                         if omited, the output path is determined automatically
         -log [path] log in path (the path to the directory)
         -v use verbose mode
         --help display this help message
         where testGenerator.jar is the name of this JAR file
```

The options that can be provided to the Test Generator are rather simple :

- Option `-in` or `-input` is used to indicate the path to the requirement file.
- Option `-out` or `-output` is used to indicate the path to the output directory for the test case.
- Option `-log` is used to indicate to log information of the process of generation to a log file indicated in argument.
- Option `-v` indicates to the tester to use verbose mode. In this case more information about the generation is displayed on the console.
- Option `--help` is used to display the available options.

B.2.2 Test engine

The option list can be obtained by launching the executable JAR file and providing the `--help` option.

```
ruitor:~/workspace/jpost2/dist/testEngine> java -jar testEngine.jar --help
Usage : java -Djava.security.policy=java.policy -jar Test-Engine.jar options
         -tc [path] : indicates the path of the test case
         -map|-mapping [path] : indicates the path of the mapping file
         -obj [path] -> Use the test objective to drive the execution
         -log [path] : the path to the directory used to log
         -graph : produce a graph illustrating the execution
         -debug|-d [path] : the path to the directory use to debug
         -verbose|-v use verbose mode
         --help : displays this help message
```

ANNEXE C

Autres travaux

C.1 A Test Calculus Framework Applied to Network Security Policies

K. Havelund et al. (Eds.) : FATES/RV 2006, LNCS 4262, pp. 55–69, 2006.
© Springer Verlag Berlin Heidelberg 2006

A Test Calculus Framework Applied to Network Security Policies

Yliès Falcone[1], Jean-Claude Fernandez[1], Laurent Mounier[1], and Jean-Luc Richier[2]

[1] Vérimag Laboratory, Gières, France
[2] LSR-IMAG Laboratory, St Martin d'Hères, France
{Ylies.Falcone, Jean-Claude.Fernandez, Laurent.Mounier, Jean-Luc.Richier}@imag.fr

Abstract. We propose a syntax-driven test generation technique to automaticaly derive abstract test cases from a set of requirements expressed in a linear temporal logic. Assuming that an elementary test case (called a "tile") is associated to each basic predicate of the formula, we show how to generate a set of test controlers associated to each logical operator, and able to coordinate the whole test execution. The test cases produced are expressed in a process algebraic style, allowing to take into account the test environment constraints. We illustrate this approach in the context of network security testing, for which more classical model-based techniques are not always suitable.

1 Introduction

Testing is a very popular validation technique, used in various application domains, and for which several formalizations have been proposed. In particular, a well-defined theory is the one commonly used in the telecommunication area for *conformance testing* of communication protocols [1]. This approach, sometimes called "model-based" approach, consists in defining a conformance relation [2,3] between a specification of the system under test and a formal model of its actual implementation. The purpose of the test is then to decide if this relation holds or not. A practical interest is that test cases can be automatically produced from this specification. Several tools implement this automatic generation technique, e.g. [4,5,6,7].

However, this model-based approach requires a rather complete *operational specification* of the system under test, defined on a precise interface level. If this constraint can be usually fulfilled for specific pieces of software (e.g., a communication protocol), it may be difficult to achieve for large systems. A typical example of such situation is testing the compliance of a network to a given security policy. Indeed, security rules are usually enforced by combining several mechanisms, operating at different architectural levels (fire-walls, anti-virus softwares, cryptographic protocols, etc.). Clearly, all these levels can be hardly encompassed in a single operational model of the network behaviour.

In a previous work [8], we have proposed an alternative approach for testing system requirements expressed as a set of (temporal) logic formulae. For each

K. Havelund et al. (Eds.): FATES/RV 2006, LNCS 4262, pp. 55–69, 2006.
© Springer-Verlag Berlin Heidelberg 2006

formula ϕ, an abstract test case t_ϕ is produced following a syntax-driven technique: assuming that an elementary test case t_i (called hereafter a "tile") has been associated to each literal p_i of formula ϕ, the whole test case t_i is obtained by combining the tiles using test operators corresponding to the logical operators appearing in formula ϕ. This provides a structural correspondence between formulae and tests and it is easy to prove that the test obtained are sound with respect to the semantics of the formulae (in other words we give a "test based" semantics of the logic which is compatible with the initial one). The originality of this approach is then that a part of the system specification is encoded into the tiles, that can be provided by the system designer or a by a test expert. We claim that it is easier to obtain that a global operational specification.

This paper extends this previous work from the test execution point of view. In [8], abstract test cases were directly expressed by labelled transition systems, independently of the test architecture. We propose here to better take into account the test execution and to express the test cases in a higher level formalism. In particular we show how to produce well structured test cases consisting of a set of test drivers (one test driver for each elementary tile), coordinated by a set of test controllers (corresponding to the logical operators appearing in the formula). Thus, independent parts of the formula can be tested in parallel (either to speed up the test execution, or due to test environment constraints), each local verdicts being combined in a consistent way by the test controllers. Formally, test cases are expressed in a classical process algebra (called a "test calculus"), using basic control operators (parallel composition and interruption) and data types to handle test parameters and verdicts.

This paper is organized as follows: section 2 introduces our "test calculus" process algebra, and section 3 defines the notions of test execution and test verdicts. We propose in section 4 a simple temporal logic allowing to express network security requirements, and we show how to produce test cases from this logic in section 5. Finally, section 6 provides some examples in the context of network security policies.

2 Test Process Algebra

To model processes, we define a rather classic term algebra with typed variables, inspired from CCS [9], CSP [10] and Lotos. We suppose a set of predefined actions *Act*, a set of types $\mathcal{T}$, and a set of variables *Var*. Actions are either modifications of variables or communications through channels which are also typed. In the following, we do not address the problem of verifying that communications and assignments are well-typed. We denote by $expr_\tau$ (resp. x_τ) any expression (resp. variable) of type τ. Thus, when we write $x_\tau := expr_\tau$, we consider that this assignment is well typed.

A test is described as a term of our process algebra. We distinguish between elementary test cases, which are elements of a basic process algebra and compound test cases. We give the syntax and an operational semantics of this test process algebra.

2.1 Basic Processes

Our basic process algebra allows to describe sequences of atomic actions, communication and iteration. A term of this algebra is called a tile, which are the elementary test components and we note *TILE* the set of all tiles.

The syntax of tiles and actions is given by the following grammar:

$$\begin{array}{l} e ::= \alpha \circ e \mid e + e \mid nil \mid recX\ e \mid X \\ \alpha ::= [b]\gamma \\ \gamma ::= x_\tau := expr_\tau \mid !c(expr_\tau) \mid ?c(x_\tau) \\ b ::= \text{true} \mid \text{false} \mid b \vee b \mid b \wedge b \mid \neg b \mid expr_\tau = expr_\tau \end{array}$$

where $e \in TILE$ is a tile, b a boolean expression, c a channel name, γ an action, $\circ$ is the *prefixing* operator ($\circ : Act \times TILE \rightarrow TILE$), $+$ the *choice* operator, X a term variable, and $recX : TILE \rightarrow TILE$ allows *recursive* tile definition (with X a term variable) [1]. When the condition b is true, we abbreviate $[true]\gamma$ by γ. The special tile *nil* does nothing.

There are two kinds of actions ($\gamma \in Act$). The first ones are the internal actions (modification of variables). The second ones are the communication actions. Two kinds of communications exist: $?c(x_\tau)$ denotes value reception on a channel c which is stored in variable x_τ; $!c(expr_\tau)$ denotes the emission of a value $expr_\tau$ on a channel c. Communication is done by "rendez-vous".

2.2 Composing Processes

Processes are compositions of tiles. Choices we made about composition operators came from needs appearing in our case studies in network security policies [8]. Composing tests in sequence is quite natural; however, for independent actions, and in order to speed-up test executions, one might want to parallelize some tests executions, for example, if one wants to scan several computers on a network. The parallel composition is also used to model the execution and communication between the test processes and the rest of the system. We assume a set $\mathcal{C}$ of channels used by tiles to communicate. We distinguish internal channels (set $\mathcal{C}_{\text{in}}$) and external channels (set $\mathcal{C}_{\text{out}}$), and we have $\mathcal{C} = \mathcal{C}_{\text{in}} \cup \mathcal{C}_{\text{out}}$.

In case of several processes executing in parallel, one might want to interrupt them. We choose to add an operator providing an exception mechanism: it permits to replace a process by an other one on the reception of a communication signal.

So, we define a set of operators, $\{\|_{\mathcal{L}}, \ltimes^{\mathcal{I}}\}$, respectively the parallel (with communication through a channel list $\mathcal{L} \subseteq \mathcal{C}$), and exception (with an action list $\mathcal{I}$) compositions.

The grammar for term processes (*TERM*) is:

$$t ::= e \mid t \|_{\mathcal{L}} t \mid t \ltimes^{\mathcal{I}} t$$

[1] we will only consider *ground* terms: each occurence of X is binded to $recX$.

The parallel operator $\|_{\mathcal{L}}$ is associative and commutative. It expresses either the interleaving of independant action or the emission $!c(expr_\tau)$ of the value of an expression $expr_\tau$ on a channel c. When the value is received by a process $?c(x_\tau)$, the communication is denoted at the syntactic level by $c(expr_\tau/x_\tau)$. The independent and parallel execution $\|_\emptyset$ is noted $\|$.

The Join-Exception operator $\ltimes^{\mathcal{I}}$ is used to interrupt a process and replace it with an other when a synchronization/global/communication action belonging to its synchronization list $\mathcal{I}$ occurs. Intuitively, considering two processes t, t' and a communication action α, $t \ltimes^{\{\alpha\}} t'$ means that if α is possible, t is replaced by t', else t continues normally.

2.3 Semantics

$$\frac{\alpha \in Act}{\alpha \circ t \xrightarrow{\alpha} t}\ (\circ) \qquad \frac{t[recX \circ t/X] \xrightarrow{\alpha} t' \qquad \alpha \in Act}{recX \circ t \xrightarrow{\alpha} t'}\ (rec)$$

$$\frac{\alpha \in Act \qquad t_2 \xrightarrow{\alpha} t_2'}{t_1 + t_2 \xrightarrow{\alpha} t_2'}\ (+)$$

$$\frac{\alpha \notin \{[b]!c(expr_\tau), [b]?c(x_\tau) | c \in \mathcal{L}, b \in \mathbb{B}_{exp}\} \qquad t_1 \xrightarrow{\alpha} t_1'}{t_1 \|_{\mathcal{L}} t_2 \xrightarrow{\alpha} t_1' \|_{\mathcal{L}} t_2}\ (\|_{\neg\mathcal{L}})$$

$$\frac{c \in \mathcal{C}_{in} \wedge c \in \mathcal{L} \qquad t_1 \xrightarrow{[b]!c(expr_\tau)} t_1' \qquad t_2 \xrightarrow{[b]?c(x_\tau)} t_2'}{t_1 \|_{\mathcal{L}} t_2 \xrightarrow{[b]c(expr_\tau/x_\tau)} t_1' \|_{\mathcal{L}} t_2'}\ (\|_{\mathcal{C}_{in}})$$

$$\frac{c \in \mathcal{C}_{out} \wedge c \in \mathcal{L} \qquad t_1 \xrightarrow{[b]!c(expr_\tau)} t_1'}{t_1 \|_{\mathcal{L}} t_2 \xrightarrow{[b]!c(expr_\tau)} t_1' \|_{\mathcal{L}} t_2}\ (! \|_{\mathcal{C}_{out}})$$

$$\frac{c \in \mathcal{C}_{out} \wedge c \in \mathcal{L} \qquad t_1 \xrightarrow{[b]?c(x_\tau)} t_1'}{t_1 \|_{\mathcal{L}} t_2 \xrightarrow{[b]?c(x_\tau)} t_1' \|_{\mathcal{L}} t_2}\ (? \|_{\mathcal{C}_{out}})$$

$$\frac{\alpha \in \mathcal{I} \qquad t_2 \xrightarrow{\alpha} t_2'}{t_1 \ltimes^{\mathcal{I}} t_2 \xrightarrow{\alpha} t_2'}\ (\ltimes^{\mathcal{I}}_{\alpha}) \qquad \frac{\alpha \notin \mathcal{I} \qquad t_1 \xrightarrow{\alpha} t_1'}{t_1 \ltimes^{\mathcal{I}} t_2 \xrightarrow{\alpha} t_1' \ltimes^{\mathcal{I}} t_2}\ (\ltimes^{\mathcal{I}}_{\neg\alpha})$$

Fig. 1. Rules for term rewriting

Let $Dom(\tau)$ be the domain of the value s of type τ. A runtime environment ρ maps the set of variables to the set of values. We note $\mathcal{E}$ the set of all environments. Actions modify environments in a classical way; we note $\rho \xrightarrow{\gamma} \rho'$ the modification of environment ρ into ρ' by action γ. For example, $\rho \xrightarrow{x_\tau := exp_\tau} \rho[\rho(exp_\tau)/x_\tau]$, where $\rho[\rho(exp_\tau)/x_\tau]$ is the environment ρ in which variable x_τ is associated the value $\rho(exp_\tau)$. In the following, environments are extended to any typed expression.

A *labelled transition system* (LTS, for short) is a quadruplet (Q, A, T, q^0) where Q is a set of states, A a set of labels, T the transition relation ($T \subseteq Q \times A \times Q$)

and q^0 the initial state ($q^0 \in Q$). We will use the following definitions and notations: $(p, a, q) \in T$ is noted $p \xrightarrow{a}_T q$ (or simply $p \xrightarrow{a} q$). An *execution sequence* λ is a composition of transitions: $q^0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots \xrightarrow{a_n} q_n$. We denote by σ^λ (resp. α^λ) the sequence of states (resp. observable actions) associated with λ. The sequence of actions α^λ is called a *trace*. We note by Σ_S, the set of finite execution sequences starting from the initial state q^0 of S. For any sequence λ of length n, λ_i or $\lambda(i)$ denotes the i-th element and $\lambda_{[i \cdots n]}$ denotes the suffix $\lambda_i \cdots \lambda_n$.

The semantics of a process is based on a LTS where states are "configurations", pairs (t, ρ), t being a term of the process algebra, ρ an environment, and transitions are given by definition 2. Configurations are used to represent process evolutions. We note $\mathcal{C}_{term} \stackrel{\text{def}}{=} TERM \times \mathcal{E}$ the set of configurations.

Definition 1 (Term-transition). *A term rewriting transition $\rightharpoonup$ is an element of $TERM \times Act \times TERM$. We say that the term t is rewritten in t' by action α. We note: $t \stackrel{\alpha}{\rightharpoonup} t'$. This semantics is similar with the CCS one [9].*

Term-transitions are defined in Figure 1 (using the fact that $\|$ and $+$ are commutative and associative).

Definition 2 (Transitions). *A transition is an element of $\mathcal{C}_{term} \times Act \times \mathcal{C}_{term}$. We say that the term t in the environment ρ is rewritten in t' modifying the environment ρ in ρ'.*

We have four transition rules, one for an assignment, and three for communication exchange. They are defined in Figure 2.

$$\frac{\rho(b) = true \qquad \rho(expr_\tau) = v \qquad t \stackrel{[b]x_\tau := expr_\tau}{\rightharpoonup} t'}{(t, \rho) \xrightarrow{x_\tau := v} (t', \rho[v/x_\tau])} \ (:=)$$

$$\frac{\rho(expr_\tau) = v \qquad t \stackrel{[b]!c(expr_\tau)}{\rightharpoonup} t' \qquad \rho(b) = true}{(t, \rho) \xrightarrow{!c(v)} (t', \rho)} \ (!)$$

$$\frac{v \in Dom(\tau) \qquad t \stackrel{[b]?c(x_\tau)}{\rightharpoonup} t' \qquad \rho(b) = true}{(t, \rho) \xrightarrow{?c(v)} (t, \rho[v/x_\tau])} \ (?)$$

$$\frac{\rho(expr_\tau) = v \qquad t \stackrel{[b]c(expr_\tau/x_\tau)}{\rightharpoonup} t' \qquad \rho(b) = true}{(t, \rho) \xrightarrow{c(v)} (t, \rho[v/x_\tau])} \ (c(expr_\tau/x_\tau))$$

Fig. 2. Rules for environment modification

3 Test Execution and Test Verdicts

As seen in the previous section, the semantics of a test case represented by a $TERM$ process t is expressed by a LTS $S_t = (Q^t, A^t, T^t, q_0^t)$. From a practical

point of view the System Under Test (SUT) is not a formal model (it is a black-box implementation interacting with a tester). However, and similarly to the classical conformance testing theory, we consider here that its underlying execution model can be expressed by a LTS $I = (Q^I, A^I, T^I, q_0^I)$. A *test execution* is then a sequence of interactions between t and the SUT to deliver a *verdict* indicating whether the test succeeded or not. We first explain how verdicts are computed in our context, and then we give a formal definition of a test execution.

3.1 Tiles Verdicts

We assume in the following that any elementary tile t_i owns at least one variable used to store its *local verdict*, namely a value of enumerated type $Verdict = \{pass, fail, inc\}$. This variable is supposed to be set to one of these values when tile execution terminates. The intuitive meaning we associate to each of these values is similar to the one used in conformance testing:

- $pass$ means that the test execution of t_i did not reveal any violation of the requirement expressed by t_i;
- $fail$ means that the test execution of t_i did reveal a violation of the requirement expressed by t_i;
- inc means that the test execution of t_i did not allow to conclude about the validity of the requirement expressed by t_i.

We now have to address the issue of combing the different verdicts obtained by each tile execution of a whole test case.

3.2 Verdict Management

The solution we adopt is to include in the test special processes (called *test controlers*) for managing tile verdicts. When tiles end their execution, i.e. have computed a verdict, they emit it toward a designated test controler which captures it. Depending on verdicts received, the controller emits a final verdict – and may halt the executions of some tests if they are not needed anymore. The "main" controler then owns a variable v_g to store the final verdict.

Test controllers can easily be written in our process algebra with communication operations as shown on the following example. The whole test case is then expressed as a term of our process algebra (with parallel composition and interuptions between processes).

An example of test controller. Let us consider a test controller waiting to receive two pass verdicts in order to decide a global $pass$ verdict (in other cases, it emits the last verdict received). Let c_v, be the channel on which verdicts are waited. The environment of this controller contains three variables, v for the verdicts received, v_g for the global verdict, and N to count numbers of verdicts remaining. An LTS representation is shown in Figure 3 and a corresponding algebraic expression is:

$$
\begin{aligned}
\mathsf{C} \stackrel{\text{def}}{=} \; & (recX\ ?c_v_i(v_i) \circ [v_i = pass]\ N\text{--} \circ X)\ + \\
& ([v_i \in \{inc, fail\}]\ v_g := v_i \circ !c_v_g(v_g) \circ !Stop\ \circ nil)\ + \\
& ([N = 0]\ v_g := pass \circ !c_v_g(v_g) \circ !Stop\ \circ nil)
\end{aligned}
$$

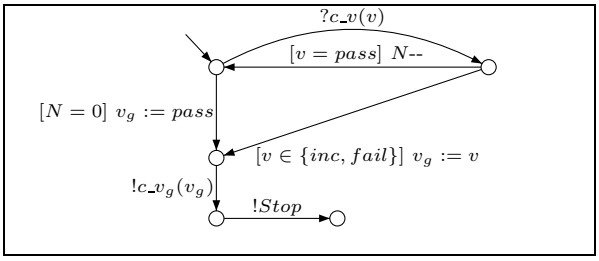


Fig. 3. Verdict controller combining pass verdicts

3.3 Test Execution

An execution of a test t (modelled by an LTS S_t) on a SUT (modelled by a LTS I), noted $\mathrm{Exec}(t, I)$, is simply expressed as a set of common execution sequences of S_t and I, defined by a composition operator $\otimes$.

Let $\lambda_I = q_0^I \xrightarrow{a_1} q_1^I \xrightarrow{a_2} q_2^I \cdots \xrightarrow{a_n} q_n^I \cdots \in \Sigma_I$ and $\lambda_{S_t} = q^{0,t} \xrightarrow{a_1} q_1^t \xrightarrow{a_2} q_2^t \cdots \xrightarrow{a_n} q_n^t \in \Sigma_{S_t}$, then $\lambda_{S_t} \otimes \lambda_I = (q^{0,t}, q_0^I) \xrightarrow{a_1} (q_1^t, q_1^I) \cdots \xrightarrow{a_n} (q_n^t, q_n^I) \in \mathrm{Exec}(t, I)$.

Let $\Sigma_{S_t}^{\mathrm{pass}}$ (resp. $\Sigma_{S_t}^{\mathrm{fail}}, \Sigma_{S_t}^{\mathrm{inconc}}$) be the sets of states of S_t where variable v_g is set to *pass* (resp. *fail*, *inc*):

$$\Sigma_{S_t}^{\mathrm{pass}} = \{(r, \rho) \mid \rho(v_g) = pass\}$$
$$\Sigma_{S_t}^{\mathrm{fail}} = \{(r, \rho) \mid \rho(v_g) = fail\}$$
$$\Sigma_{S_t}^{\mathrm{inc}} = \{(r, \rho) \mid \rho(v_g) = inc\}$$

For $\lambda \in \mathrm{Exec}(t, I)$, we define the verdict function: $\mathrm{VExec}(\lambda) = pass$ (resp. $fail, inconc$) iff there is $\lambda_{S_t} \in \Sigma_{S_t}^{\mathrm{pass}}$ (resp. $\Sigma_{S_t}^{\mathrm{fail}}, \Sigma_{S_t}^{\mathrm{inconc}}$) and $\lambda_I \in \Sigma_I$ such that $\lambda_{S_t} \otimes \lambda_I = \lambda$.

4 Security Rules Formalization

This work was initiated by a case study those objectives was to test the compliance of the IMAG network (which connects the IMAG's laboratories) to a security policy. This security policy is expressed as a set of informal rules describing (conditional) *obligations* and *interdictions* that have to be fulfilled by the network administrators. We focussed our attention to a subset of rules dedicated to electronic mail and users account management. As a mater of fact, it happened that most of these rules could be formalized using a simple logic where interdiction, obligation and permission are expressed by means of temporal modalities. We give here the syntax and semantics of this logic.

4.1 Syntax

A security policy rule is expressed by a logical *formula* (φ), built upon *literals*. Each literal can be either a *condition literal* ($p_c \in P_c$), or an *event literal*

($p_e \in P_e$). A condition literal is a (static) predicate on the network configuration (e.g., $extRelay(h)$ holds iff machine h is configured as an external mail relay), and an event literal corresponds to the occurrence of a transition in the network behavior (e.g., $enterNetwork(m)$ holds if message m is received by the network). A conjunction of condition literals is simply called a *condition* (C), whereas a conjunction of a single event literal and a condition is called a *(guarded) event* (E). The abstract syntax of a formula is given in Table 1. The intuitive meaning of these formulae is the following:

- An $\mathcal{O}$-Rule expresses a *conditional obligation*: when a particular condition holds, then another condition should also hold (logical implication).
- An $\mathcal{O}_T$-Rule expresses a *triggered obligation*: when a given event happens, then another condition should hold (or some event should occurs) before expiration of a given amount of time.
- An $\mathcal{F}$-Rule expresses an *interdiction*: when a given condition holds, or when a given event happens, then a given event is always prohibited.

Table 1. Syntax of logic formulae

φ	$::= C \Rightarrow \mathcal{O}\, C$	($\mathcal{O}$-Rule)
	$\mid\ E \Rightarrow \mathcal{O}_T\, C \mid E \Rightarrow \mathcal{O}_T\, E$	($\mathcal{O}_T$-Rule)
	$\mid\ C \Rightarrow \mathcal{F}\, C \mid C \Rightarrow \mathcal{F}\, E$	($\mathcal{F}$-Rule)
E	$::= p_e[C] \mid p_e$	(Event)
C	$::= \bigwedge_{i=1}^{n} p_{c_i}$	(Condition)

4.2 Semantics

Formulas are interpreted over LTS. Intuitively, a LTS S satisfies a formula φ iff *all* its execution sequences λ do, where condition literals are interpreted over *states*, event literals are interpreted over *labels*. We first introduce two interpretation functions for condition and event literals:
$f_c : P_c \to 2^Q$, associates to p_c the set of states on which p_c holds;
$f_e : P_e \to 2^A$, associates to p_e the set of labels on which p_e holds.

The satisfaction relation of a formula φ on an execution sequence λ ($\lambda \models \varphi$) is then (inductively) defined as follows:

- $\lambda \models C$ for $C = p_c^1 \wedge \cdots \wedge p_c^n$ iff $\forall i.\ \sigma^\lambda(1) \in f_c(p_c^i)$
- $\lambda \models p_e$ iff $\alpha^\lambda(1) \in f_e(p_e)$
- $\lambda \models p_e[C]$ iff $(\alpha^\lambda(1) \in f_e(p_e) \wedge \lambda(2) \models C)$
- $\lambda \models \varphi_1 \Rightarrow \mathcal{O}\,\varphi_2$ iff $((\lambda \models \varphi_1) \Rightarrow (\lambda \models \varphi_2))$
- $\lambda \models \varphi_1 \Rightarrow \mathcal{O}_T\varphi_2$ iff $((\lambda \models \varphi_1) \Rightarrow (\exists j \in [1, |\lambda|].\ \lambda(j) \models \varphi_2))$
- $\lambda \models \varphi_1 \Rightarrow \mathcal{F}\varphi_2$ iff $((\lambda \models \varphi_1) \Rightarrow (\forall j \in [1, |\lambda|].\ \lambda(j) \not\models \varphi_2))$

Finally, $S \models \varphi$ iff $\forall \lambda \in \Sigma_S.\ \lambda \models \varphi$.

5 Test Generation

We define a structural generation function $GenTest$ to convert a rule into the desired combination of elementary tiles with controllers. It associates controllers in such a way that the final verdict is *pass* iff the rule is satisfied by the SUT. Each controller emits its verdict on a channel, and may uses variables. In the following, new variables and channels will be silently created whenever necessary.

$GenTest$ generates parallel and architecturally independent sub-tests. Formula semantics is ensured by the controller verdict combinations. Suitable scheduling of sub-tests is supplied by the controllers through channels used to start and stop sub-tests (given below by $Test$ function).

5.1 Test Generation Function *GenTest*

Transformation of tiles. Given a tile t_p (computing its verdict in the variable ver) associated to an elementary predicate p, the $Test$ function transforms it. Intuitively, $Test(t_p, \mathcal{L})$, where $\mathcal{L}$ is a channel list, is t_p modified in order to be controlled through the channel list $\mathcal{L}$. More formally:

$$Test(t_p, \{c_start, c_stop, c_loop, c_ver\}) \stackrel{\text{def}}{=}$$
$$recX\ (?c_start() \circ t_p \circ (?c_loop() \circ X + !c_ver(ver) \circ nil)) \ltimes^{\{?c_stop()\}} ?c_stop() \circ nil$$

A representation on a LTS is shown in Figure 4.

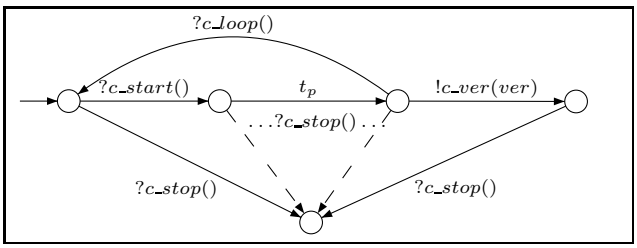


Fig. 4. Extension of tile t_p in a testing form

***GenTest* definition.** The rule general form is: $P_l \Rightarrow \mathcal{M}\, P_r$ where $P_l, P_r \in \{E, C\}$ are predicates and $\mathcal{M} \in \{\mathcal{O}, \mathcal{O}_T, \mathcal{F}\}$ a modality.

The $GenTest$ function is defined on the rule structure, giving an expression to be instantiated according to the different modalities. We suppose that the final verdict is emitted on the *main* channel, and t_{c_i}, t_{p_e} are the tiles respectively associated to elementary predicates c_i, p_e.

$$GenTest(P_l \Rightarrow \mathcal{M}\, P_r) \stackrel{def}{=} (GenTest_P(P_l, \mathcal{L}_l) \parallel GenTest_P(P_r, \mathcal{L}_r)) \parallel_{\mathcal{L}} \mathtt{C}_{\mathcal{M}}(\mathcal{L}_l, \mathcal{L}_r)$$
$$\text{with } \mathcal{L} = \mathcal{L}_l \cup \mathcal{L}_r,$$
$$\mathcal{L}_l = \{c_start_l, c_stop_l, c_loop_l, c_ver_l\},\ \mathcal{L}_r = \{c_start_r, c_stop_r, c_loop_r, c_ver_r\}$$

$$GenTest_P(p_e[C], \{c_start, c_stop, c_loop, c_ver\}) \stackrel{def}{=}$$
$$\big(Test(t_{p_e}, \mathcal{L}_e) \parallel GenTest_C(C, \mathcal{L}_c)\big) \parallel_{\mathcal{L}} \mathsf{C}_E(\{c_start, c_stop, c_loop, c_ver\}, \mathcal{L}_e, \mathcal{L}_c)$$
$$\text{with } \mathcal{L} \stackrel{\text{def}}{=} \mathcal{L}_e \cup \mathcal{L}_c$$
$$\mathcal{L}_e = \{c_start_e, c_stop_e, c_loop_e, c_ver_e\},\ \mathcal{L}_c = \{c_start_c, c_stop_c, c_loop_c, c_ver_c\}$$
$$GenTest_P(p_e, \{c_start, c_stop, c_loop, c_ver\}) \stackrel{\text{def}}{=} Test(t_{p_e}, \{c_start, c_stop, c_loop, c_ver\})$$

$$GenTest_P(C, \mathcal{L}) \stackrel{\text{def}}{=} GenTest_C(C, \mathcal{L})$$

$$GenTest_C(\wedge_{i=1}^{n} c_i, \{c_start, c_stop, c_loop, c_ver\}) \stackrel{def}{=}$$
$$\text{if } n = 1,\quad Test(t_{c_1}, \{c_start, c_stop, c_loop, c_ver\})$$
$$\text{else } /^*\ n > 1\ ^*/$$
$$\|_{i=1}^{n}\ Test(t_{c_i}, \mathcal{L}_i) \parallel_{\mathcal{L}} \mathsf{C}_{\wedge}(\{c_start, c_stop, c_loop, c_ver\}, (\mathcal{L}_i)_{i=1\ldots n}, n)$$
$$\text{with } \mathcal{L} = \cup_{i=1}^{n} \mathcal{L}_i; \forall i \in \{1 \ldots n\}, \mathcal{L}_i = \{c_start, c_stop, c_loop, c_ver_i\}$$

5.2 Verdict Controllers

Several verdict controllers are used in the *GenTest* definition. Controllers have different purposes. They are first used to manage the execution of sub-tests corresponding to the components of the rule. For example, for a $\mathcal{O}_T$ formula, we have to wait for the left-side subtest before starting the right-side subtest. Controllers are also used to "implement" the formula semantics by combining verdicts from sub-tests.

Controllers definitions are parameterized with channel parameters. We give here an informal description of the controllers, with a graphical representation for the more important ones. Other controllers are similar and are easy to formalize in our test calculus (see [11] for a complete description).

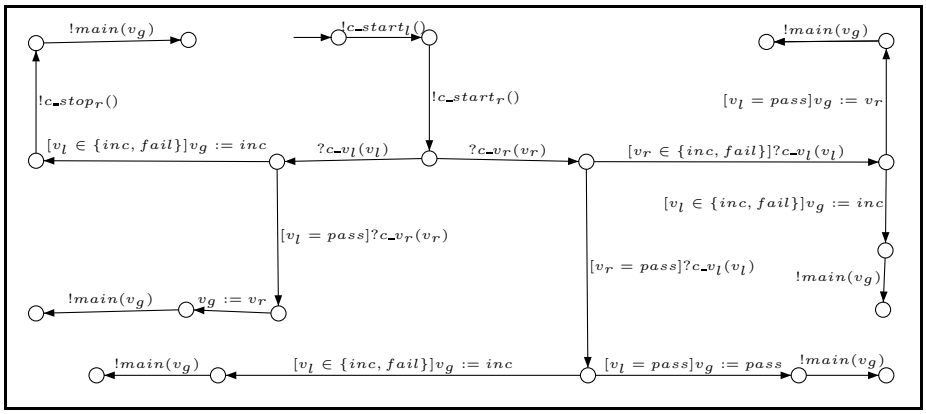


Fig. 5. An instantiated LTS representation of the $\mathsf{C}_{\mathcal{O}}$ controller

Formula level controllers. They emit their verdict on the channel *main*.

1. $\mathsf{C}_{\mathcal{O}}(channel_list, channel_list)$. This controller is used to manage the execution of tiles corresponding to the left and right part of a static implication. The controller starts the two tests corresponding to the two sides of the implication. Then it waits for the reception of a verdict (verdicts can arrive in any order). According to the semantics of implication and the first verdict received, it decides either to wait for the second verdict or to emit a verdict immediately. The controller takes two channel lists as parameters for managing the execution and verdict of each side of the implication. The associated environment contains three variables. A LTS representation of $\mathsf{C}_{\mathcal{O}}(\{c_start_l, c_stop_l, c_v_l, c_loop_l\}, \{c_start_r, c_stop_r, c_v_r, c_loop_r\})$ is shown in Figure 5.
2. $\mathsf{C}_{\mathcal{O}_T}(channel_list, channel_list)$. This controller is used to manage the execution of tiles corresponding to the sides of an implication with a triggered obligation. The controller starts the test corresponding to the left side of the implication. If this test is inconclusive or fails, a *inc* verdict is decided. Otherwise, the timer and the second test are started. If the test emits *pass*, the final verdict is *pass*. As long as the timer is not expired, (that is, the boolean variable *t_out* is false), if the second test ends with $fail$ or *inc*, the test is started again. When the the timer expires, a stop signal ($!c_stop_r$) is sent to the right side test. In that case, the final verdict is *inc* if an *inc* verdict occurred, $fail$ otherwise. A LTS representation of $\mathsf{C}_{\mathcal{O}_T}(\{c_start_l, c_stop_l, c_loop_l, c_ver_l\}, \{c_start_r, c_stop_r, c_loop_r, c_ver_r\})$ is shown in Figure 6.
3. $\mathsf{C}_{\mathcal{F}}(channel_list, channel_list)$. This controller is similar to the $\mathsf{C}_{\mathcal{O}}$ controller. It waits for a fail verdict for the right-side subtest in order to conclude on a *pass* verdict.

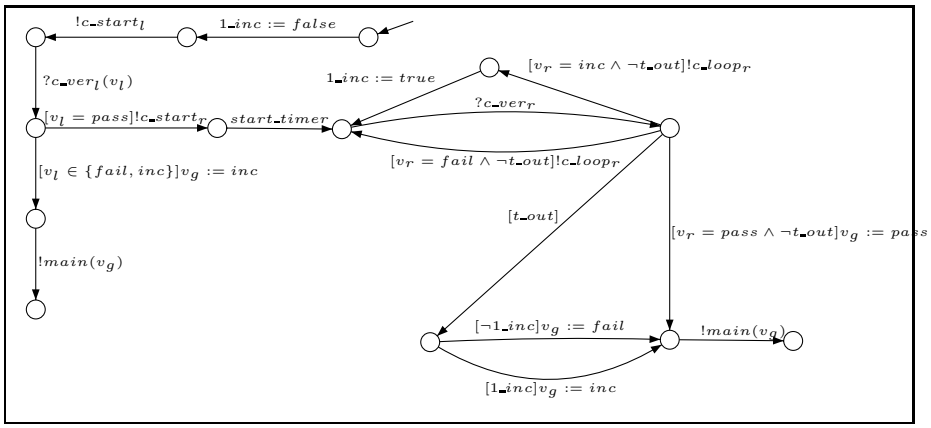


Fig. 6. An instantiated LTS representation of the $\mathsf{C}_{\mathcal{O}_T}$ controller

Predicate Level Controllers

1. $\mathsf{C}_E(channel_list, channel_list, channel_list)$. This controller is used to manage executions and verdicts around an event. The controller starts the execution of the event, and then, depending on the verdict received, it starts the sub-tests associated to the condition predicates in E. These conditions have to be tested after the event.
2. $\mathsf{C}_\wedge(channel_lists, integer)$. Informally this controller starts different tests and waits for verdicts. Like the other controllers it controls sub-tests with a channel. If all tests succeed, the controller emits a *pass* verdict. If some tests do not respond *pass* the controller emits the last verdict received and stops the other potentially executing sub-tests. This controller is a generalization of the one presented in 3.1.

5.3 Soundness Proposition

We now express that an abstract test case produced by function $GenTest$ is always *sound*, i.e. it delivers a *fail* verdict when executed on a network behavior I only if formula ϕ does not hold on I. To do this, we follow a very similar approach than in [8]. Two hypotheses are required in order to prove this soundness property:

H1. First, for any formula φ, we assume that each elementary test case t_i provided for the (event or condition) literals p_i appearing in φ is *strongly sound* in the following sense:
Execution of t_i on SUT I always terminate, and
$\forall \lambda \in \mathrm{Exec}(t_i, I) \cdot \mathrm{VExec}(\lambda) = Pass \Rightarrow \lambda \models p_i \wedge (\mathrm{VExec}(\lambda) = Fail \Rightarrow \lambda \not\models p_i)$.

H2. Second, we assume that the whole execution of a (provided or generated) test case t associated to a condition C is *stable* with respect to condition literals: the valuation of these literal does not change during the test execution. This simply means that the network configuration is supposed to remain stable when a condition is tested. Formally:
$\forall p_i \in P_c.\ \forall \lambda \in \Sigma_I \cdot \lambda_{S_t} \otimes \lambda \in \mathrm{Exec}(t, I) \Rightarrow (\sigma^\lambda \subseteq f_c(p_i) \vee \sigma^\lambda \cap f_c(p_i) = \emptyset)$
where σ^λ denotes here tacitly a set of states instead of a sequence.

We now formulate the soundness property:
Proposition: Let φ a formula, I an LTS and $t = \mathrm{GenTest}(\varphi)$. Then:

$$\lambda \in \mathrm{Exec}(t, I) \wedge \mathrm{VExec}(\lambda) = fail \Longrightarrow I \not\models \varphi$$

The proof of this proposition relies on a structural induction of the formula (using auxiliary lemmas to show the correctness of each intermediate $GenTest$ function).

6 Application

In this section we apply the $GenTest$ function to two rule patterns taken from the case study presented in [8].

6.1 $\mathcal{O}$-Rule

Consider the requirement *"External relays shall be in the DMZ* [2]*"*, this could be reasonably understood as *"If a host is an external relay, it has to be in the DMZ"*. A possible modelisation is:

$$extRelay(h) \Rightarrow \mathcal{O}\,(inDMZ(h))$$

The goal of this test is to verify that each external relay h is in the DMZ. The $GenTest$ function can be applied on this formula, leading to the following test:

$$\begin{aligned}
&GenTest(extRelay(h) \Rightarrow \mathcal{O}\,(inDMZ(h)))\\
&\quad = \Big(GenTest_P(extRelay(h), \mathcal{L}_l) \parallel GenTest_P(inDMZ(h), \mathcal{L}_r)\Big) \parallel_{\mathcal{L}} \complement_{\mathcal{O}}(\mathcal{L}_l, \mathcal{L}_r)\\
&\quad = \Big(Test(t_{extRelay(h)}, \mathcal{L}_l) \parallel Test(t_{inDMZ(h)}, \mathcal{L}_r)\Big) \parallel_{\mathcal{L}} \complement_{\mathcal{O}}(\mathcal{L}_l, \mathcal{L}_r)\\
&\text{where } \mathcal{L} = \mathcal{L}_l \cup \mathcal{L}_r\\
&\qquad \mathcal{L}_l = \{c_start_l, c_stop_l, c_v_l, c_loop_l\} \text{ and } \mathcal{L}_r = \{c_start_r, c_stop_r, c_v_r, c_loop_r\}
\end{aligned}$$

For a given machine h, predicates $extRelay(h)$ and $inDMZ(h)$ can be checked either by analyzing the configuration of devices in the network and/or administrators' databases (if this information can be trusted), or rather by testing the actual behaviour of this machine (does it act as an external relay?). In this last case, we need some tiles for these two predicates.

A possible tile for $t_{extRelay(h)}$ consists in attempting to send a mail m from an external machine he to an internal machine hi by first opening a connection from he to h ($!connect(he, h)$), and then asking for mail transfers from he to h ($!transfer(he, h, m)$) and from h to hi ($!transfer(h, hi, m)$). If these operation succeed the verdict is $pass$, otherwise it is $fail$ (h does not act as an external relay). Not that if the connection from he to h fails for external reasons (e.g, network overloading) then the verdict is inc (inconclusive). This tile can be formalized as follows:

$$\begin{aligned}
&t_{extRelay(h)} \stackrel{\text{def}}{=}\\
&\quad !connect(he, h)\circ\\
&\qquad \big(?ok\circ!transfer(he, h, m)\circ\\
&\qquad\qquad [?ok\circ!tranfer(h, hi, m)\circ(ver := pass)\circ nil + ?ko\circ(ver := fail)\circ nil]\\
&\qquad + (?ko\circ(ver := inc)\circ nil)\big)
\end{aligned}$$

6.2 $\mathcal{O}_T$-Rule

Security policies may also express availability requirements. Consider *"When there is a request to open an account, user privileges and resources must be activated within one hour"*. We formalize this requirement as:

$$request_open_account(c)[\neg ex_account(c)] \Rightarrow \mathcal{O}_{1\mathrm{H}}(open_account(c)[allocate_disk(c)])$$

[2] for *demilitarized zone*, a strongly controlled buffer zone between the inside and outside of the network.

Supposing that there exists a tile for each predicate and that all tiles are independent. One could generate a test from appropriate derivation:

$$
\begin{aligned}
&GenTest(req_acc(c)[\neg ex_acc(c)] \Rightarrow \mathcal{O}_{1H}(op_acc(c)[alloc(c)])) \\
&\quad = \Big(\big(Test(t_{req_acc(c)}, \mathcal{L}_{le}) \parallel Test(ex_acc(c), \mathcal{L}_{lc})\big) \parallel_{\mathcal{L}_{le} \cup \mathcal{L}_{lc}} \mathsf{C}_E(\mathcal{L}_l, \mathcal{L}_{le}, \mathcal{L}_{lc})\Big) \\
&\qquad \parallel \Big(\big(Test(t_{op_acc(c)}, \mathcal{L}_{re}) \parallel Test(t_{alloc_disk(c)}, \mathcal{L}_{rc})\big) \parallel_{\mathcal{L}_{re} \cup \mathcal{L}_{rc}} \mathsf{C}_E(\mathcal{L}_r, \mathcal{L}_{re}, \mathcal{L}_{rc})\Big) \\
&\qquad \parallel_{\mathcal{L}} \mathsf{C}_{\mathcal{O}_{1H}}(\mathcal{L}_l, \mathcal{L}_r) \\
&\text{with: } \mathcal{L} = \mathcal{L}_l \cup \mathcal{L}_r,\ \mathcal{L}_x = \{c_start_x, c_stop_x, c_loop_x, c_ver_x\}_x, x \in \{l, r, le, lc, re, rc\}
\end{aligned}
$$

7 Conclusion

We have proposed a test generation technique for testing the validity of a temporal logical formula on a system under test. The originality of this approach is to produce the tests by combinations of elementary test cases (called tiles), associated to each atomic predicates of the formula. These tiles are supposed to be provided by the system designer or a test expert, and, assuming they are correct, it can be proved that the whole test case obtained is sound. The practical interest of this approach is that it can be applied even if a formal specification of the system under test is not available, or if the test execution needs to mix several interface levels. A concrete example of such a situation is network security testing, where the security policy is usually expressed as a set of logical requirements, encompassing many network elements (communication protocols, firewalls, antivirus softwares, etc.) and those behavior would be hard to describe on a single formal specification. The abstract test cases we obtain are expressed in a process algebraic style, and they are structured into test drivers (the tiles), and test controllers (encoding the logical operators). This approach makes them close to executable test cases, and easy to map on a concrete (and distributed) test architecture. Independent parts of the tests can then be executed concurrently.

This work could be continued in several directions. First, the logic we proposed here could be extended. So far, the kind of formulae we considered was guided by a concrete application, but, staying in the context of network security, other deontic/temporal modalities could be foreseen, like“interdiction within a delay”, or “permission”. We also believe that this approach would be flexible enough to be used in other application domains, with other kinds of logical formulae (for instance with nested temporal modalities, which were not considered here). A second improvement would be to produce a clear diagnostic when a test execution fails. So far, test controllers only propagate “fail” verdicts, but it could be useful to better indicate to the user why a test execution failed (which sub-formula was unsuccessfully tested, and what is the incorrect execution sequence we obtained). Finally, we are currently implementing this test generation technique, and we expect that practical experimentations will help us to extend it towards the generation of *concrete* test cases, that could be directly executable.

References

1. ISO/IEC 9946-1: OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework. International Standard ISO/IEC 9646-1/2/3 (1992)
2. Brinksma, E., Alderden, R., Langerak, R. Van de Lagemaat, J., Tretmans, J.: A Formal Approach to Conformance Testing. In De Meer, J., Mackert, L., Effelsberg, W., eds.: Second International Workshop on Protocol Test Systems, North Holland (1990) 349–363
3. Tretmans, J.: Test Generation with Inputs, Outputs, and Quiescence. In Margaria, T., Steffen, B., eds.: Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96). Volume 1055 of Lecture Notes in Computer Science., Springer-Verlag (1996) 127–146
4. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. In: The Sixth World Conference on Integrated Design & Process Technology (IDPT'02), Pasadena, California, USA (2002)
5. Belinfante, A., Feenstra, J., de Vries, R., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal Test Automation : a Simple Experiment. In: 12th International Workshop on Testing of Communicating Systems, G. Csopaki et S. Dibuz et K. Tarnay, Kluwer Academic Publishers (1999)
6. Schmitt, M., Koch, B., Grabowski, J., Hogrefe, D.: Autolink - A Tool for Automatic and Semi-Automatic Test Generation from SDL Specifications. Technical Report A-98-05, Medical University of Lübeck (1998)
7. Groz, R., Jéron, T., Kerbrat, A.: Automated test generation from SDL specifications. In Dssouli, R., von Bochmann, G., Lahav, Y., eds.: SDL'99 The Next Millenium, 9th SDL Forum, Montreal, Quebec, Elsevier (1999) 135–152
8. Darmaillacq, V., Fernandez, J.C., Groz, R., Mounier, L., Richier, J.L.: Test Generation for Network Security Rules. In: 18th IFIP International Conference, TestCom 2006, New York, LNCS 3964, Springer (2006)
9. Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (1980)
10. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
11. Falcone, Y.: Un cadre formel pour le test de politiques de sécurité. Master's thesis, Université Joseph Fourier, Grenoble, France (2006)

C.2 A Compositional Testing Framework Driven by Partial Specifications

A. Petrenko et al. (Eds.) : TestCom/FATES 2007, LNCS 4581, pp. 107–122, 2007.
© IFIP- International Federation for Information Processing 2007

A Compositional Testing Framework Driven by Partial Specifications

Yliès Falcone[1], Jean-Claude Fernandez[1], Laurent Mounier[1], and Jean-Luc Richier[2]

[1] Vérimag Laboratory, 2 avenue de Vignate 38610 Gières, France
[2] LIG Laboratory, 681, rue de la Passerelle, BP 72, 38402 Saint Martin d'Hères Cedex, France
{Ylies.Falcone,Jean-Claude.Fernandez,Laurent.Mounier, Jean-Luc.Richier}@imag.fr

Abstract. We present a testing framework using a compositional approach to generate and execute test cases. Test cases are generated and combined with respect to a partial specification expressed as a set of requirements and elementary test cases. These approach and framework are supported by a prototype tool presented here. The framework is presented here in its LTL-like application, besides other specification formalisms can be added.

1 Introduction

Testing is a popular validation technique which purpose is essentially to find defects on a system implementation, either during its development, or once a final version has been completed. Therefore, and even if lots of work have already been carried out on this topic, improving the effectiveness of a testing phase while reducing its cost and time consumption remains a very important challenge, sustained by a strong industrial demand.

From a practical point of view, a test campaign consists in producing a test suite (test generation), and executing it on the target system (test execution). Automating test generation means deriving the test suite from some initial description of the system under test. The test suite consists in a set of test cases, where each test case is a set of interaction sequences to be executed by an external tester. Any execution of a test case should lead to a test verdict, indicating if the system succeeded or not on this particular test (or if the test was not conclusive).

The initial system description used to produce the test cases may be for instance the source code of the software, some hypothesis on the sets of inputs it may receive (user profiles), or some requirements on its expected properties at run-time (i.e., a characterization of its (in)-correct execution sequences). In this latter case, when the purpose of the test campaign is to check the correctness of some behavioral requirements, an interesting approach for automatic test generation is the so-called model-based testing technique. Model-based testing

A. Petrenko et al. (Eds.): TestCom/FATES 2007, LNCS 4581, pp. 107–122, 2007.
© IFIP- International Federation for Information Processing 2007

is rather successful in the communication protocol area, especially because it is able to cope with some non-determinism of the system under test. It has been implemented in several tools, see for example [1] for a survey. However, it suffers from some drawbacks that may prevent its use in other application areas. First of all, it strongly relies on the availability of a system specification, which is not always the case in practice. Moreover, when it exists, this specification should be complete enough to ensure some relevance of the test suite produced. Finally, it is likely the case that this specification cannot encompass all the implementation details, and is restricted to a given abstraction level. Therefore, to become executable, the test cases produced have to be *refined* into more concrete interaction sequences. Automating this process in the general case is still a challenging problem [2], and most of the time, when performed by hand, the soundness of the result cannot be fully guaranteed.

We propose here an alternative approach to produce a test suite dedicated to the validation of behavioral requirements of a software (see Fig. 1). In this framework the requirements $\mathcal{R}$ are expressed by logical formulas φ built upon a set of (abstract) predicates P_i describing (possibly non-atomic) operations performed on the system under test. A typical example of such requirements could be for instance a security policy, where the abstract predicates would denote some high-level operations like "user A is authenticated", or "message M has been corrupted". The approach we propose relies on the following consideration: a perfect knowledge of the implementation details is required to produce elementary test cases Tc_i able to decide whether such predicates hold or not at some state of the software execution. Therefore, writing the test cases dedicated to these predicates should be left to the programmer (or tester) expertise when a detailed system specification is not available. However, correctly orchestrating the execution of these "basic test cases" and combining their results to deduce the validity of the overall logical formula is much easier to automate since it depends only of the semantics of the operators used in this formula. This step can therefore be produced by an automatic test generator, and this test generation can even be performed in a compositional way (on the structure of the logical formula). More precisely, from the formula φ, a test generation function automatically produces an (abstract) tester AT_φ. This tester consists of a set of communicating *test controllers*, one for each operator appearing in φ. Thus, AT_φ depends only on the structure of formula φ. AT_φ is then instantiated using the elementary test cases Tc_i to obtain a concrete tester T_φ for the formula φ. Execution of this tester on the implementation I produces the final verdict.

We believe that this approach is general enough to be instantiated with several logic formalisms commonly used to express requirements on execution traces (e.g., extended regular expressions or linear temporal logics).

This works extends some preliminary descriptions on this technique [3,4] in several directions: first we try to demonstrate that it is general enough to support several logical formalisms, then we apply it for the well-known LTL temporal logic, and finally we evaluate it on a small case study using a prototype tool under development.

In addition to the numerous works proposed in the context of model-based test generation for conformance testing, this work also takes credits from the community of run-time verification. In fact, one of the techniques commonly used in this area consists in generating a monitor able to check the correctness of an execution trace with respect to a given logical requirement (see for instance [5,6] or [7] for a short survey). In practice, this technique needs to *instrument* the software under verification with a set of observation points to produce the traces to be verified by the monitor. This instrumentation should of course be correlated with the requirement to verify (i.e., the trace produced should contain enough information). In the approach proposed here, these instrumentation directives are replaced by the elementary test cases associated to each elementary predicates. The main difference is that these test cases are not restricted to pure observation actions, but they may also contain some active testing operations, like calling some methods, or communicating with some remote process to check the correctness of an abstract predicate.

The rest of the paper is organized as follows: Sect. 2 introduces the general approach, while Sect. 3 details its sound-proved application for a particular variant of the linear temporal logic LTL. Section 4 and 5 respectively describe the architecture of a prototype tool based on this framework, and its application on a small case study. The conclusion and perspectives of this work are given is Sect. 6.

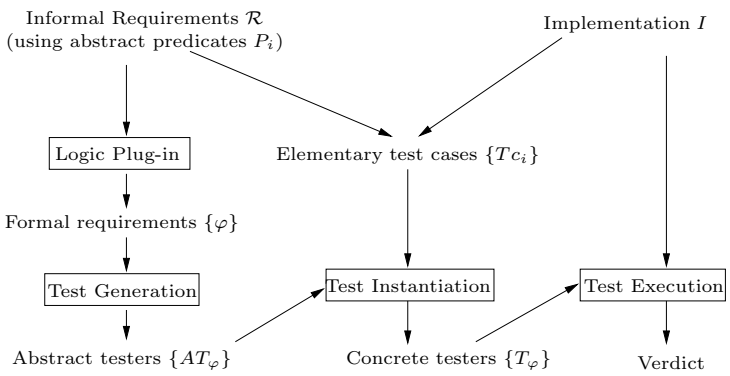


Fig. 1. Test generation overview

2 The General Approach

We describe here more formally the test generation approach sketched in the introduction. As it has been explained, this approach relies on the following steps:

- generation of an *abstract tester* AT_φ from a formal requirement φ;
- instantiation of AT_φ into a concrete tester A_φ using the set of elementary testers associated to each atomic predicate of φ;
- execution of T_φ against the System Under Test (SUT) to obtain a test verdict.

2.1 Notations

A *labelled transition system* (LTS, for short) is a quadruplet $S = (Q, A, T, q_0)$ where Q is a set of states, A a set of labels, $T \subseteq Q \times A \times Q$ the transition relation and $q_0 \in Q$ the initial state. We will denote by $p \xrightarrow{a}_T q$ (or simply $p \xrightarrow{a} q$) when $(p, a, q) \in T$. A *finite execution sequence* of S is a sequence $(p_i, a_i, q_i)_{\{0 \leq i \leq m\}}$ where $p_0 = q_0$ and $p_{i+1} = q_i$. For each finite execution sequence λ, the sequence of actions $(a_0, a_1, \ldots, a_m)$ is called a *finite execution trace* of S. We denote by $Exec(S)$ the set of all finite execution traces of S. For an execution trace $\sigma = (a_0, a_1, \ldots, a_m)$, we denote by $\mid \sigma \mid$ the length $m+1$ of σ, by $\sigma^{k \ldots l}$ the sub-sequence $(a_k, \ldots, a_l)$ when $0 \leq k \leq l \leq m$, and by $\sigma^{k \cdots}$ the sub-sequence $(a_k, \ldots, a_m)$ when $0 \leq k \leq m$. Finally, $\sigma_{\downarrow X}$ denotes the *projection* of σ on action set X. Namely, $\sigma_{\downarrow X} = \{a_0 \cdot \cdots \cdot a_m \mid \forall i \cdot a_i \in X \wedge \sigma = w_0 \cdot a_0 \cdots w_m \cdot a_m \cdot w_{m+1} \wedge w_i \in (A \setminus X)^*\}$.

2.2 Formal Requirements

We assume in the following that the formal requirements φ we consider are expressed using a logic $\mathscr{L}$. Formulas of $\mathscr{L}$ are built upon a finite set of *n-ary operators* F^n and a finite set of *abstract predicates* $\{p_1, p_2, \ldots, p_n\}$ as follows:

$$\text{formula} ::= F^n(\text{formula}_1, \text{formula}_2, \ldots, \text{formula}_n) \mid p_i$$

We suppose that each formula of $\mathscr{L}$ is interpreted over a finite execution trace of a LTS S, and we say that S satisfies φ (we note $S \models \varphi$) iff *all* sequences of $Exec(S)$ satisfy φ. Relation $\models$ is supposed to be defined inductively on the syntax of $\mathscr{L}$ in the usual way: abstract predicates are interpreted over $Exec(S)$, and the semantics of each operator $F^n(\varphi_1, \ldots, \varphi_n)$ is defined in terms of sets of execution traces satisfying respectively $\varphi_1, \ldots, \varphi_n$.

2.3 Test Process Algebra

In order to outline the compositionality of our test generation technique, we express a tester using an algebraic notation. We recall here the dedicated "test process algebra" introduced in [4], but other existing process algebras could also be used.

Syntax. Let *Act* be a set of *actions*, $\mathcal{T}$ be a set of *types* (with $\tau \in \mathcal{T}$), *Var* a set of *variables* (with $x \in Var$), and *Val* a set of *values* (union of values of types $\mathcal{T}$). We denote by $expr_\tau$ (resp. x_τ) any expression (resp. variable) of type τ. In particular, we assume the existence of a special type called *Verdict* which associated values are $\{pass, fail, inconc\}$ and which is used to denote the *verdicts* produced during the test execution.The syntax of a test process t is given by the following grammar:

$$\begin{array}{l} t ::= [b]\,\gamma \circ t \mid t + t \mid nil \mid recX\ t \mid X \\ b ::= \text{true} \mid \text{false} \mid b \vee b \mid b \wedge b \mid \neg b \mid expr_\tau = expr_\tau \\ \gamma ::= x_\tau := expr_\tau \mid !c(expr_\tau) \mid ?c(x_\tau) \end{array}$$

In this grammar t denotes a basic tester (*nil* being the empty tester doing nothing), b a boolean expression, c a channel name, γ an action, $\circ$ is the *prefixing* operator, $+$ the *choice* operator, X a term variable, $recX$ allows *recursive* process definition (with X a term variable)[1]. When the condition b is true, we abbreviate $[true]\gamma$ by γ. Atomic actions performed by a basic tester are either internal assignments ($x_\tau := expr_\tau$), value emissions ($!c(expr_\tau)$) or value receptions ($?c(x_\tau)$) over a channel c[2].

Semantics. We first give a semantics of basic testers (t) using rewriting rule between uninterpreted terms in a CCS-like style (see Fig. 2).

$$\frac{\gamma \in Act}{[b]\gamma \circ t \overset{[b]\gamma}{\rightharpoonup} t}\ (\circ) \qquad \frac{t[recX \circ t/X] \overset{[b]\gamma}{\rightharpoonup} t' \qquad \gamma \in Act}{recX \circ t \overset{[b]\gamma}{\rightharpoonup} t'}\ (rec)$$

$$\frac{\gamma \in Act \qquad t_1 \overset{[b]\gamma}{\rightharpoonup} t_1'}{t_1 + t_2 \overset{[b]\gamma}{\rightharpoonup} t_1'}\ (+)_l \qquad \frac{\gamma \in Act \qquad t_2 \overset{[b]\gamma}{\rightharpoonup} t_2'}{t_1 + t_2 \overset{[b]\gamma}{\rightharpoonup} t_2'}\ (+)_r$$

Fig. 2. Rules for term rewriting

The semantics of a basic test process t is then given by means of a LTS $S_t = (Q^t, A^t, T^t, q_0^t)$ in the usual way: states Q^t are "configurations" of the form (t, ρ), where t is a term and $\rho : Var \to Val$ is an *environment*. States and transition of S_t (relation $\longrightarrow$) are the smallest sets defined by the rules given in Fig. 3 (using the auxiliary relation $\rightharpoonup$ defined in Fig. 2). The initial state q_0^t of S is the configuration (t_0, ρ_0), where ρ_0 maps all the variables to an undefined value. Finally, note that actions A^t of S_t are labelled either by internal assignments ($x_\tau := v$) or external emission ($!c(v)$). In the following we denote by $A^t_{\text{ext}} \subseteq A^t$ the external emissions and receptions performed by the LTS associated to a test process t.

Complex testers are obtained by parallel composition of test processes with synchronisation on a channel set cs (operator $\|_{cs}$), or using a so-called "join-exception" operator ($\ltimes^{\mathcal{I}}$), allowing to interrupt a process on reception of a communication using the interruption channel set $\mathcal{I}$. We note $\|$ for $\|_\emptyset$ and Act_chan(s) all possible actions using a channel in the set s. To tackle with communication in our semantics, we give two sets of rules specifying how LTSs are composed relatively to the communication operators ($\|_{cs}, \ltimes^{\mathcal{I}}$). These rules aim to maintain asynchronous execution, communication by *rendez-vous*. Let $S_i^t = (Q_i^t, A_i^t, T_i^t, q_{0i}^t)$ be two LTSs modelling the behaviours of two processes t_1 and t_2, we define the LTS $S = (Q, A, T, q_0)$ modelling the behaviours of $S_1^t \|_{cs} S_2^t$

[1] We will only consider *ground* terms: each occurrence of X is bound to $recX$.

[2] To simplify the calculus, we supposed that all channels exchange one value. In the testers, we also use "synchronisation channels", without exchanged argument, as a straightforward extension.

$$\frac{\rho(expr_\tau) = v \qquad t \overset{[b]x_\tau := expr_\tau}{\rightharpoonup} t' \qquad \rho(b) = true}{(t,\rho) \xrightarrow{x_\tau := v} (t', \rho[v/x_\tau])} \ (:=)$$

$$\frac{\rho(expr_\tau) = v \qquad t \overset{[b]!c(expr_\tau)}{\rightharpoonup} t' \qquad \rho(b) = true}{(t,\rho) \xrightarrow{!c(v)} (t', \rho)} \ (!)$$

$$\frac{v \in Dom(\tau) \qquad t \overset{[b]?c(x_\tau)}{\rightharpoonup} t' \qquad \rho(b) = true}{(t,\rho) \xrightarrow{!c(v)} (t, \rho[v/x_\tau])} \ (?)$$

Fig. 3. Rules for environment modification

$$\frac{p_1 \xrightarrow{a} p_1' \qquad a \notin \text{Act_chan}(cs)}{(p_1,p_2) \xrightarrow{a} (p_1', p_2)} \ (\overline{\|^{\mathrm{l}}_{cs}}) \qquad \frac{p_2 \xrightarrow{a} p_2' \qquad a \notin \text{Act_chan}(cs)}{(p_1,p_2) \xrightarrow{a} (p_1, p_2')} \ (\overline{\|^{\mathrm{r}}_{cs}})$$

$$\frac{p_1 \xrightarrow{a} p_1' \qquad p_2 \xrightarrow{a} p_2' \qquad a \in \text{Act_chan}(cs)}{(p_1,p_2) \xrightarrow{a} (p_1', p_2')} \ (\|_{cs})$$

$$\frac{p_1 \xrightarrow{a} p_1' \qquad a \notin \text{Act_chan}(\mathcal{I})}{(p_1,p_2) \xrightarrow{a} (p_1', p_2)} \ (\overline{\ltimes^{\mathcal{I}}}) \qquad \frac{p_2 \xrightarrow{a} p_2' \qquad a \in \text{Act_chan}(\mathcal{I})}{(p_1,p_2) \xrightarrow{a} (\bot, p_2')} \ (\ltimes^{\mathcal{I}})$$

Fig. 4. LTS composition related to $\|_{cs}$ and $\ltimes^{\mathcal{I}}$

and $S_1^t \ltimes^{\mathcal{I}} S_2^t$ as the product of S_1^t and S_2^t where $Q \subseteq (Q_1^t \cup \{\bot\}) \times Q_2^t$ and the transition rules are given in Fig. 4.

2.4 Test Generation

Principle. The test generation technique we propose aims to produce a tester process t_φ associated to a formal requirement φ and it can be formalized by a function called *GenTest* in the rest of the paper ($GenTest(\varphi) = t_\varphi$). This generation step depends of course of the logical formalism under consideration, but it is compositionally defined in the following way:

- a basic tester t_{p_i} is associated with each abstract predicate p_i of φ;
- for each sub-formula $\phi = F^n(\phi_1, \cdots, \phi_n)$ of φ, a test process t_ϕ is produced, where t_ϕ is a parallel composition between test processes $t_{\phi_1}, \ldots, t_{\phi_n}$ and a test process C_{F^n} called a *test controller* for operator F^n.

The purpose of test controllers C_{F^n} is both to schedule the test execution of the t_{ϕ_k} (starting, stopping or restarting their execution), and to combine their verdicts to produce the overall verdict associated to ϕ. As a result, the architecture of a tester t_φ matches the abstract syntax tree corresponding to formula φ: leaves are basic tester processes corresponding to abstract predicates p_i of φ, intermediate nodes are controllers associated with operators of φ.

Hypothesis. To allow interactions between the internal sub-processes of a tester t_φ, we assume the following hypotheses:

Each tester sub-process t_{ϕ_k} (basic tester or controller) owns a special variable used to store its *local verdict*. This variable is supposed to be set to one of these values when the test execution terminates – its intuitive meaning is similar to the conformance testing case:

- *pass* means that the test execution of t_{ϕ_k} did not reveal any violation of the sub-formula associated to t_{ϕ_k};
- *fail* means that the test execution of t_{ϕ_k} did reveal a violation of the sub-formula associated to t_{ϕ_k};
- *inconc* (inconclusive) means that the test execution of t_{ϕ_k} did not allow to conclude about the validity of the sub-formula associated to t_{ϕ_k}.

Each tester process t_{ϕ_k} (basic tester or controller) owns a set of four dedicated communication channels $cs_k = \{c_start_k, c_stop_k, c_loop_k, c_ver_k\}$ used respectively to start its execution, to stop it, to resume it from its initial state and to deliver a verdict. In the following, we denote by $\mathsf{C}(cs, cs_1, \cdots, cs_n)$ each controller C where cs is the channel set dedicated to the communication with the embracing controller whereas the (cs_i) are the channel sets dedicated to the communication with the sub-test processes. Finally, a "starter" process is also required to start the topmost controller associated to t and to read the verdict it delivered.

Each basic tester process t_{p_i} associated to an LTS $S_{t_{p_i}}$ is supposed to have a subset of actions $A^{t_{p_i}}_{\text{ext}} \subseteq A^{t_{p_i}}$ used to communicate with the SUT. Considering $t = GenTest(\varphi)$, the set A^t_{ext} is defined as the union of the $A^{t_{p_i}}_{\text{ext}}$ where p_i is a basic predicate of φ.

Test generation function definition (GenTest). *GenTest* can then be defined as follows using *GT* as an intermediate function:

$$GenTest(\varphi) \stackrel{\text{def}}{=} GT(\varphi, cs) \parallel_{\{c_start, c_ver\}} (!c_start() \circ ?c_ver(x) \circ nil)$$

where cs is the set $\{c_start, c_stop, c_loop, c_ver\}$ of channel names associated to t_φ.

$$GT(p_i, cs) \stackrel{\text{def}}{=} Test(t_{p_i}, cs)$$

$$GT(F^n(\phi_1, \ldots, \phi_n), cs) \stackrel{\text{def}}{=} (GT(\phi_1, cs_1) \parallel \cdots \parallel GT(\phi_n, cs_n)) \parallel_{cs'} \mathsf{C}_{F^n}(cs, cs_1, \ldots, cs_n)$$

where $cs_1, \ldots, cs_n$ are sets of fresh channel names and $cs' = cs_1 \cup \cdots \cup cs_n$.

$$Test(t_p, \{c_start, c_stop, c_loop, c_ver\}) \stackrel{\text{def}}{=}$$

$$recX\,(?c_start() \circ t_p \circ !c_ver(ver) \circ ?c_loop() \circ X) \ltimes^{\{c_stop\}} (?c_stop() \circ nil)$$

2.5 Test Execution and Test Verdicts

As seen in the previous subsections, the semantics of a tester represented by a test process t is expressed by a LTS $S_t = (Q^t, A^t, T^t, q^t_0)$ where $A^t_{\text{ext}} \subseteq A^t$ denotes the external actions it may perform. Although the system under test I is not described by a formal model, its behaviour can also be expressed by a LTS $S_I = (Q^I, A^I, T^I, q^I_0)$. A *test execution* is a sequence of interactions (on

A^t_{ext}) between t and I in order to deliver a *verdict* indicating whether the test succeeded or not. We define here more precisely these notions of test execution and test verdict.

Formally speaking, a test execution of a test process t on a SUT I can be viewed as an execution trace of the parallel product $\otimes_{A^t_{\text{ext}}}$ between LTSs S_t and S_I with synchronizations on actions of A^t_{ext}. This product is defined as follows:

$S_t \otimes_{A^t_{\text{ext}}} S_I$ is the LTS (Q, A, T, q_0) where $Q \subseteq Q^t \times Q^I$, $A \subseteq A^t \cup A^I$, $q_0 = (q^t_0, q^I_0)$, and
$T = \{(p^t, p^I) \xrightarrow{a} (q^t, q^I) \mid (p^t, a, q^t) \in T^t \wedge (p^I, a, q^I) \in T^I \wedge a \in A^t_{\text{ext}}\} \cup \{(p^t, p^I) \xrightarrow{a} (q^t, p^I) \mid (p^t, a, q^t) \in T^t \wedge a \in A^t \setminus A^t_{\text{ext}}\} \cup \{(p^t, p^I) \xrightarrow{a} (p^t, q^I) \mid (p^I, a, q^I) \in T^I \wedge a \in A^I \setminus A^t_{\text{ext}}\}$.

For any test execution $\sigma \in \text{Exec}(S_t \otimes_{A^t_{\text{ext}}} S_I)$, we define the verdict function: $\text{VExec}(\sigma) = \mathit{pass}$ (resp. $\mathit{fail}, \mathit{inconc}$) iff $\sigma = \mathit{c_start}() \cdot \sigma' \cdot \mathit{c_ver}(\mathit{pass})$ (resp. $\sigma = \mathit{c_start}() \cdot \sigma' \cdot \mathit{c_ver}(fail)$, $\sigma = \mathit{c_start}() \cdot \sigma' \cdot \mathit{c_ver}(\mathit{inconc})$) and $\mathit{c_start}$ (resp. $\mathit{c_ver}$) is the starting (resp. the verdict) channel associated to the topmost controller of t.

3 Application to Variant of LTL

This section presents an instantiation of the previous framework for a (non atomic) action-based version of LTL-X, the next-free variant of LTL [8].

3.1 The Logic

Syntax. The syntax of a formula φ is given by the following grammar, where the atoms $\{p_1, \ldots, p_n\}$ are action predicates.

$$\varphi \ ::= \ \neg\,\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \varphi \wedge \varphi \mid p_i$$

Semantics. Formulas φ are interpreted over the finite execution traces $\sigma \in A^*$ of a LTS. We introduce the following notations.

To each atomic predicate p_i of φ we associate a subset of actions A_{p_i} and two subsets L_{p_i} and $L_{\overline{p_i}}$ of $A^*_{p_i}$. Intuitively, A_{p_i} denotes the actions that influence the truth value of p_i, and L_{p_i} (resp. $L_{\overline{p_i}}$) the set of finite execution traces satisfying (resp. non satisfying) p_i. We suppose that the action sets A_{p_i} are such that $\{(A_{p_i})_i\}$ forms a partition of A, that for all i, j, $L_{p_i} \cap L_{\overline{p_i}} = \emptyset$ and $(L_{p_i} \cup L_{\overline{p_i}}) \cap (L_{p_j} \cup L_{\overline{p_j}}) = \emptyset$. The sets of actions for a predicate are easily extended to sets of actions for a formula: $A_{\neg\varphi} = A_\varphi$, $A_{\varphi_1 \wedge \varphi_2} = A_{\varphi_1 \mathcal{U} \varphi_2} = A_{\varphi_1} \cup A_{\varphi_2}$.

The truth value of a formula is given in a three-valued logic matching our notion of test verdicts: a formula φ can be evaluated to *true* on a trace σ ($\sigma \models_T \varphi$), or it can be evaluated to *false* ($\sigma \models_F \varphi$), or its evaluation may remain inconclusive ($\sigma \models_I \varphi$).

The semantics for a formula φ is defined by three sets. The set of sequences that satisfy (resp. violate) the formula φ is noted $[\![\varphi]\!]^T$ (resp. $[\![\varphi]\!]^F$). We also note $[\![\varphi]\!]^I$ the set of sequences for which the satisfaction remains inconclusive.

- $[\![p_i]\!]^T = \{\omega \mid \exists \omega', \omega'' \cdot \omega = \omega' \cdot \omega'' \wedge \omega'_{\downarrow A_{p_i}} \in L_{p_i}\}$
  $[\![p_i]\!]^F = \{\omega \mid \exists \omega', \omega'' \cdot \omega = \omega' \cdot \omega'' \wedge \omega'_{\downarrow A_{p_i}} \in L_{\overline{p_i}}\}$
- $[\![\neg\varphi]\!]^T = [\![\varphi]\!]^F$
  $[\![\neg\varphi]\!]^F = [\![\varphi]\!]^T$
- $[\![\varphi_1 \wedge \varphi_2]\!]^T = \{\omega \mid \exists \omega', \omega'' \cdot \omega = \omega' \cdot \omega'' \wedge \omega'_{\downarrow A_{\varphi_1}} \in [\![\varphi_1]\!]^T \wedge \omega'_{\downarrow A_{\varphi_2}} \in [\![\varphi_2]\!]^T\}$
  $[\![\varphi_1 \wedge \varphi_2]\!]^F = \{\omega \mid \exists \omega', \omega'' \cdot \omega = \omega' \cdot \omega'' \wedge \omega'_{\downarrow A_{\varphi_1}} \in [\![\varphi_1]\!]^F \vee \omega'_{\downarrow A_{\varphi_2}} \in [\![\varphi_2]\!]^F\}$
- $[\![\varphi_1 \mathcal{U} \varphi_2]\!]^T = \{\omega \mid \exists \omega_1, \ldots, \omega_n, \omega' \cdot \omega = \omega_1 \cdots \omega_n \cdot \omega'$
  $\wedge \forall i < n \cdot \omega_{i\downarrow A_{\varphi_1}} \in [\![\varphi_1]\!]^T \wedge \omega_{n\downarrow A_{\varphi_2}} \in [\![\varphi_2]\!]^T\}$
  $[\![\varphi_1 \mathcal{U} \varphi_2]\!]^F = \{\omega \mid \exists \omega_1, \ldots, \omega_n, \omega' \cdot \omega = \omega_1 \cdots \omega_n \cdot \omega'$
  $\wedge \Big(\forall i \leq n \cdot \omega_{i\downarrow A_{\varphi_2}} \in [\![\varphi_2]\!]^F \vee (\exists l \leq n \cdot \omega_{l\downarrow A_{\varphi_2}} \in [\![\varphi_2]\!]^T \wedge \exists k < l \cdot \omega_{k\downarrow A_{\varphi_1}} \in [\![\varphi_1]\!]^F)\Big)\}$
- $[\![\varphi]\!]^I = A^* \setminus ([\![\varphi]\!]^P \cup [\![\varphi]\!]^F)$

Finally we note $\sigma \models_T \varphi$ (resp. $\sigma \models_F \varphi$, $\sigma \models_I \varphi$) for $\sigma \in [\![\varphi]\!]^T$ (resp. $\sigma \in [\![\varphi]\!]^F$,$\sigma \in [\![\varphi]\!]^I$).

3.2 Test Generation

Following the structural test generation principle given in Sect. 2.4, it is possible to obtain a *GenTest* function for our LTL-like logic. The *GenTest* definition can be made explicit simply by giving controller definitions. So, we give a graphical description of each controller used by *GenTest*. To simplify the presentation, the *stop* transitions are not represented: the receptions all lead from each state of the controller to some "sink" state corresponding to the nil process, and emissions are sent by controllers to stop sub-tests when their execution is not needed anymore for the verdict computation.

The $\mathsf{C}_\neg(\{c_start, c_loop, c_ver\}, \{c_start', c_loop', c_ver'\})$ controller is shown on Fig. 5. It inverts the verdict received by transforming *pass* verdict into *fail* verdict (and conversely) and keeping *inconc* verdict unchanged.

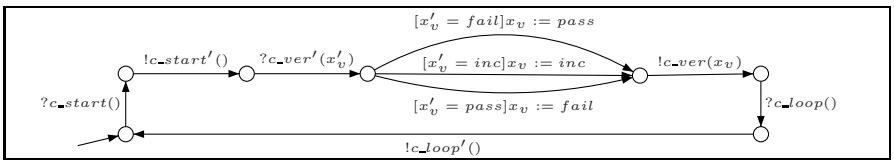


Fig. 5. The $\mathsf{C}_\neg$ controller

The $\mathsf{C}_\wedge(\{c_start, c_loop, c_ver\}, \{c_start_l, c_loop_l, c_ver_l\}, \{c_start_r, c_loop_r, c_ver_r\})$ controller is shown on Fig. 6. It starts both controlled sub-tests and waits for their verdict returns, and sets the global verdict depending on received values.

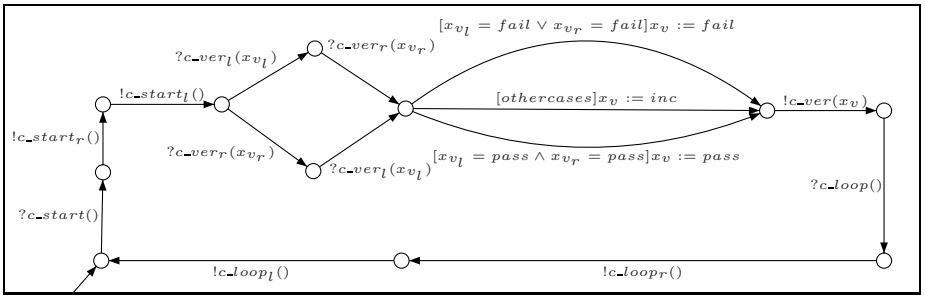


Fig. 6. The $\mathsf{C}_\wedge$ controller

The $\mathsf{C}_{\mathcal{U}}(\{c_start, c_loop, c_ver\}, \{c_start_l, c_loop_l, c_ver_l\}, \{c_start_r, c_loop_r, c_ver_r\})$ controller is shown on Fig. 7 and Fig. 8. It is composed of three sub-processes executing in parallel and starting on the same action $?c_start()$. The first sub-process C_m is represented on Fig. 7. The second and third ones corresponds to two instantiations

$$\mathsf{C}_l(\{c_start, c_loop, c_ver\}, \{c_start_l, c_loop_l, c_ver_l\}),$$
$$\mathsf{C}_r(\{c_start, c_loop, c_ver\}, \{c_start_r, c_loop_r, c_ver_r\})$$

of $\mathsf{C}_x(\{c_start, c_loop, c_ver\}, \{c_start_x, c_loop_x, c_ver_x\})$ for the two controlled sub-test for the two sub-formulas. An algebraic expression of this controller could be

$$\mathsf{C}_{\mathcal{U}}(\cdots) = (\mathsf{C}_l(\cdots) \parallel \mathsf{C}_r(\cdots)) \parallel_{\{r_fail, l_fail, r_pass, l_pass\}} \mathsf{C}_m(\cdots)$$

One could understand C_l and C_r as two sub-controllers in charge of communicating with the controlled tests that send relevant information to the "main" sub-controller C_m deciding the verdict. The reception of an inconclusive verdict from a sub-test process interrupts the controller which emits an inconclusive verdict (not represented on the figure). If no answer is received from the sub-processes after some finite amount of time, then the tester delivers its verdict (*timeout* transitions). For the sake of clarity we simplified the controller representation. First, we represent the emission of the controller verdict and the return to the initial state under a reception of a loop signal ($?c_loop()$) by a state which name represents the value of the emitted verdict. Second, we do not represent *inconc* verdict, the controller propagates it.

3.3 Soundness Proposition

We express that an abstract test case produced by the *GenTest* function is always *sound*, *i.e.* it delivers a *pass* (resp. *fail*) verdict when it is executed on a SUT behavior I only if the formula used to generate it is satisfied (resp. violated) on I. This proposition relies on one hypothesis, and two intermediate lemmas.

Hypothesis 1. *Each test case t_{p_i} associated to a predicate p_i is strongly sound in the following sense:*

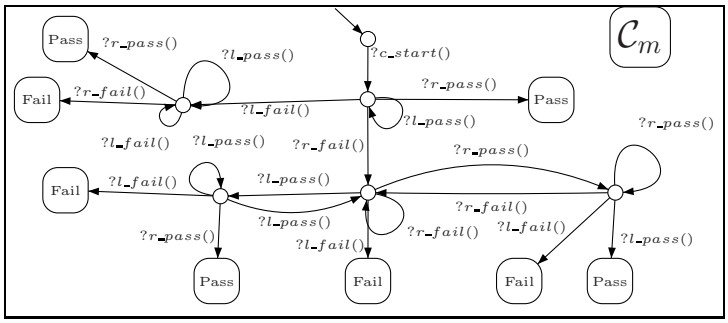


Fig. 7. The $\mathsf{C}_{\mathcal{U}}$ controller, the C_m part

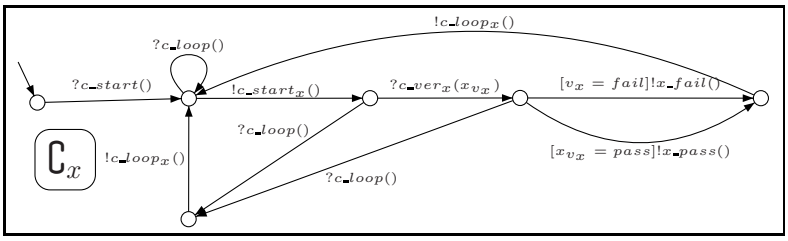


Fig. 8. The $\mathsf{C}_{\mathcal{U}}$ controller, the C_x part

$$\forall \sigma \in \mathrm{Exec}(t_{p_i} \otimes_{A_{p_i}} I), \mathrm{VExec}(\sigma) = pass \Rightarrow \sigma \models_T p_i$$
$$\forall \sigma \in \mathrm{Exec}(t_{p_i} \otimes_{A_{p_i}} I), \mathrm{VExec}(\sigma) = fail \Rightarrow \sigma \models_F p_i$$

The lemmas state that the verdict computed by t_φ on a sequence σ only depends on actions of σ belonging to A_φ.

Lemma 1. *All execution sequences with the same projection on a formula φ actions have the same satisfaction relation towards φ. That is:*

$$\forall \sigma, \sigma' \cdot \sigma_{\downarrow A_\varphi} = \sigma'_{\downarrow A_\varphi} \Rightarrow (\sigma \models_T \varphi \Leftrightarrow \sigma' \models_T \varphi) \wedge (\sigma \models_F \varphi \Leftrightarrow \sigma' \models_F \varphi)$$

Lemma 2. *For each formula φ, each sequence σ, the verdicts pass and fail of a sequence do not change if we project it on φ's actions. That is:*

$$\forall \varphi, \forall \sigma \cdot \sigma \models_T \varphi \Rightarrow \sigma_{\downarrow A_\varphi} \models_T \varphi$$
$$\forall \varphi, \forall \sigma \cdot \sigma \models_F \varphi \Rightarrow \sigma_{\downarrow A_\varphi} \models_F \varphi$$

These lemmas come directly from the definition of our logic and the controllers used in *GenTest*. Now we can formulate the proposition.

Theorem 1. *Let φ be a formula, and t=GenTest(φ), S a LTS, $\sigma \in Exec(t \otimes_{A_\varphi} S)$ a test execution sequence, the proposition is:*

$$VExec(\sigma) = pass \Longrightarrow \sigma \models_T \varphi$$
$$VExec(\sigma) = fail \Longrightarrow \sigma \models_F \varphi$$

Sketch of the soundness proof. The proof is done by structural induction on φ. We give the proof for two cases.

For the predicates. The proof relies directly on predicate strong soundness (Hypothesis 1).

For the negation operator. Let suppose $\varphi = \neg\varphi'$. We have to prove that:

$$\forall\sigma \in \text{Exec}(GT(\neg\varphi', \mathcal{L}) \otimes_{A_\varphi} I), \text{VExec}(\sigma) = pass \Rightarrow \sigma \models_T \neg\varphi'$$
$$\forall\sigma \in \text{Exec}(GT(\neg\varphi', \mathcal{L}) \otimes_{A_\varphi} I), \text{VExec}(\sigma) = fail \Rightarrow \sigma \models_F \neg\varphi'$$

Let $\sigma \in \text{Exec}(GT(\neg\varphi', \mathcal{L}) \otimes_{A_\varphi} I)$ suppose that $\text{VExec}(\sigma) = pass$.
By definition of GT,

$$GT(\neg\varphi', \mathcal{L}) = GT(\varphi', \mathcal{L}') \parallel_{\mathcal{L}'} \mathcal{C}_\neg(\mathcal{L}, \mathcal{L}')$$

Since controller $\mathcal{C}_\neg$ does not trigger the *c_loop* transition of its subtest when it is used as a main tester process, execution sequence σ is necessarily in the form:

$$c_start() \cdot \sigma_I \cdot \sigma' \cdot \sigma_I \cdot$$
$$([x_v = pass]x_{v_g} := fail \,|\, [x_v = fail]x_{v_g} := pass \,|\, [x_v = inconc]x_{v_g} :=$$
$$inconc) \cdot \sigma_I \cdot c_ver(x_{v_g})$$

with $\sigma' \in \text{Exec}(GT(\varphi', \mathcal{L}') \otimes_{A_{\varphi'}} I)$, σ_I denoting SUT's actions, and $\omega \cdot (a \mid b) \cdot \omega'$ denoting the sequences $\omega \cdot a \cdot \omega'$ and $\omega \cdot b \cdot \omega'$.

As the controller emits a *pass* verdict ($!c_ver(x_{v_g})$ with x_{v_g} evaluated to *pass* in the $\mathcal{C}_\neg$'s environment) it means that it necessarily received a $fail$ verdict ($[x_v = fail]x_{v_g} := pass$) on *c_ver'* from the sub-test corresponding to $GT(\varphi', \mathcal{L}')$. So we have $\sigma' \in \text{Exec}(GT(\varphi', \mathcal{L}') \otimes_{A'_\varphi} I)$ and $\text{VExec}(\sigma') = fail$.

The induction hypothesis implies that $\sigma' \models_F \varphi'$. The Lemma 2 gives that ${\sigma'}_{\downarrow A_{\varphi'}} \models_F \varphi'$. And we have:

$$\begin{aligned} \sigma'_{\downarrow A_{\varphi'}} &= \sigma'_{\downarrow A_\varphi} \ (\forall\varphi, A_\varphi = A_{\neg\varphi}) \\ &= \sigma_{\downarrow A_\varphi} \ (c_start, \sigma_I \notin {A_\varphi}^*) \end{aligned}$$

So $\sigma_{\downarrow A_\varphi} \models_F \varphi'$. We conclude using the Lemma 1 that $\sigma \models_F \varphi'$ that is $\sigma \models_T \neg\varphi'$. The proof for $\forall\sigma \in \text{Exec}(GT(\neg\varphi', \mathcal{L}) \otimes_{A_\varphi} I), \text{VExec}(\sigma) = fail \Rightarrow \sigma \models_F \neg\varphi'$ is similar.

Others operators. Proofs for the other operators follow the same principle and can be found in [9].

4 Java-CTPS

We now present Java-CTPS, a prototype of a testing framework tool for the Java environment which follows our approach. We just describe an abstract view of the tool. Interested readers can refer to [9] which contains a more detailed description.

Java-CTPS contains a test generator using the compositional approach for Java, *i.e.* the tester is generated in Java, for a SUT written in the same language. An interface is provided for the user to write a library of elementary test cases from the SUT interface. Indeed, the interface defines a set of methods that can be called. Elementary test cases are terms of our test calculus which external actions correspond to these methods: execution of an external action on the tester leads to a complete execution of the method on the SUT (from the call to the return). An elementary test case execution on the tester leads to the execution of some methods in the SUT interface.

Afterwards, using our method, the tool transforms a specification in a given formalism in a abstract test case. Then it is combined with the library to provide an executable test case.

Synthesis algorithms of controlled tests for different formalisms have been defined and implemented. Two interfaces are provided to the user: a command-line mode and a graphic interface. A simplified version of the test generation and execution is depicted on Fig. 9.

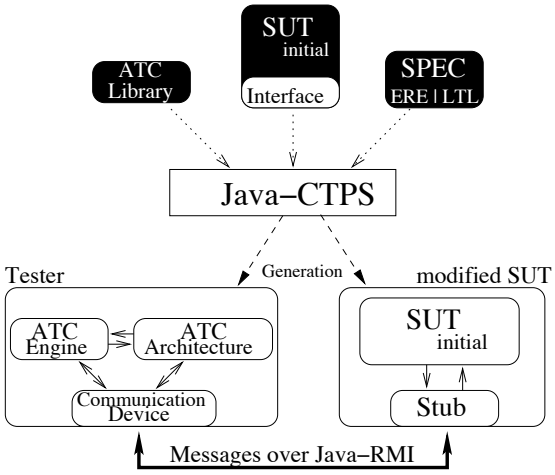


Fig. 9. Simplified working principle

Elementary test cases library establishment. The SUT's architecture description provides the set of controllable and observable actions on the system. The user can compose them, with respect to the test calculus, and write new elementary test cases. Programming is eased by the abstraction furnished by the SUT interface.

Specification as a set of requirements. Java-CTPS offers several formalisms to express requirements on the system.

- *Temporal logics.* Temporal logics [8] are frequently used to express specification for reactive systems. Their use has proved to be useful and appreciated

for system verification. Our case studies have shown that many concepts in security policies are in the scope of these logics.
- *Regular Expression.* Regular expressions [10] allows to define behaviour schemes expressed on a system traces. They are commonly used and well-understood by engineers for their practical aspect.

Test of a system. Java-CTPS translates the specification into abstract test cases following the specification formula structure. Depending on the used specification formalism and the expressed requirement, the tool generates a test architecture whose test cases are coming from the controller library in accordance with *Gen-Test*. An execution engine is also generated. So, the generated tester can execute different test cases translated into a unique representation formalism on the test calculus engine. The initial SUT is also modified by adding a stub to communicate with the tester. This component provides means to launch method calls on the reception of specific signals. Thus, abstract test cases executing on the tester guide concrete test cases execution on the modified SUT. Communication between tester and SUT is done using the Java-RMI mechanism as we plan to support distributed SUT in a future evolution of our tool.

5 Case Study

We present a case study illustrating the approach presented above. From some credit card security documents [11], we established a security policy and a credit card model. We applied our method with the security policy as a partial specification and the executable credit card model as a SUT. The credit card model and part of its security policy are overviewed here.

The card. The architecture of the credit card is presented on Fig. 10. The interface is modeled by the *Device* component, corresponding to the possible action set on the card. Several banking operations are proposed, *e.g. provide_pin*, *change_pin*, *init_session*, *transaction*, *close_session*. Choice was made to use a Java interface to model the banking application one. The *Device* component interacts with a *Memory Abstraction* component providing, as its name indicates, some basic operations on the memory's areas. The *Memory* is just the credit card memory represented as a fixed size integer array.

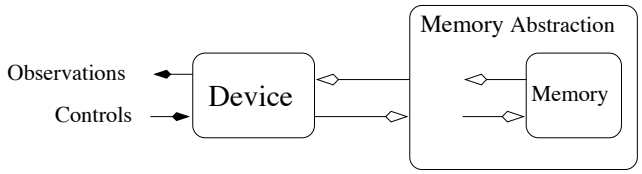


Fig. 10. The credit card architecture

The security policy. Our study allowed us to extract several security requirement specific to the credit card security domain. These requirements concerned several specification formalisms: regular expressions, and temporal logics. Some examples of properties that we were able to test can be expressed at an informal level:

1. After three failed authentications, the card is blocked, i.e. no action is permitted anymore.
2. If the card is suddenly removed the number of remaining authentications tries is set to 0.
3. No action is permitted without an identification.

For example one could see the first property formalised in our logic as several rules, one for each possible action:

$$try_3_authentications(all_failed) \Longrightarrow action(blocked)$$

The third one could be reasonably understood as:

$$action(blocked)\, \mathcal{U}\, authentication(success)$$

With this formalisation, these properties were tested with test cases that use elementary combinations of card interface actions. For example we wrote an abstract test case leading to three failed authentications using actions *provide_pin* and *init_session*.

6 Conclusion

In this work we have proposed a testing framework allowing to produce and execute test cases from a partial specification of the system under test. The approach we follow consists in generating the test cases from some high-level requirements on the expected system behaviour (expressed in a trace-based temporal logic), assuming that a concrete elementary tester is provided for each abstract predicate used in these requirements. This "partial specification" plays a similar role to the instrumentation directives currently used in run-time verification techniques, and we believe that they are easier to obtain in a realistic context than a complete operational specification. Furthermore, we have illustrated how this approach could be instantiated on a particular logic (an action-based variant of LTL-X), while showing that it is general enough to be applied to other similar trace-based logics. Finally, a prototype tool implementing this framework is available and preliminary experiments have been performed on a small case study.

Our main objective is now to extend this prototype in order to deal with larger examples. A promising direction is to investigate how the so-called MOP technology [6] could be used as an implementation platform. In particular, it already offers useful facilities to translate high-level requirements (expressed in various logics) into (passive) observers, and to monitor the behaviour of a program under

test using these monitors. A possible extension would then be to replace these observers by our active basic testers (using the aspect programming techniques supported by MOP).

Acknowledgement. The authors thank the referees for their helpful remarks.

References

1. Hartman, A.: Model based test generation tools survey. Technical report, AGEDIS Consortium (2002)
2. van der Bijl, M., Rensink, A., Tretmans, J.: Action refinement in conformance testing. In: Khendek, F., Dssouli, R. (eds.) Testing of Communicating Systems (TESTCOM) LNCS, vol. 3205, pp. 81–96. Springer, Heidelberg (2005)
3. Darmaillacq, V., Fernandez, J.C., Groz, R., Mounier, L., Richier, J.L.: Test generation for network security rules. In: TestCom, pp. 341–356 (2006)
4. Falcone, Y., Fernandez, J.C., Mounier, L., Richier, J.L.: A test calculus framework applied to network security policies. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) Formal Approaches to Software Testing and Runtime Verification. LNCS, vol. 4262, pp. 55–69. Springer, Heidelberg (2006)
5. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002)
6. Chen, F., D'Amorim, M., Roşu, G.: Checking and correcting behaviors of java programs at runtime with java-mop. In: Workshop on Runtime Verification (RV'05), ENTCS, vol. 144(4), pp. 3–20 (2005)
7. Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W., Washington, R.: Combining test case generation and runtime verification. Theor. Comput. Sci. 336(2-3), 209–234 (2005)
8. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. New York, Inc. Springer, Heidelberg (1995)
9. Falcone, Y., Fernandez, J.C., Mounier, L., Richier, J.L.: A partial specification driven compositional testing method and tool. Technical Report TR-2007-04, Vérimag Research Report (2007)
10. Kleene, S.C.: Representation of events in nerve nets and finite automata. In: Shannon, C.E., McCarthy, J. (eds.) Automata Studies, pp. 3–41. Princeton University Press, Princeton, New Jersey (1956)
11. Mantel, H., Stephan, W., Ullmann, M., Vogt, R.: Guideline for the development and evaluation of formal security policy models in the scope of itsec and common criteria. Technical report, BSI,DFKI (2004)

C.3 j-POST : A Java Toolchain for Property-Oriented Software Testing

1571-0661$ - see front matter © 2008 Elsevier B.V. All rights reserved.
www.elsevier.com/locate/entcs
doi :10.1016/j.entcs.2008.11.004

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 220 (2008) 29–41
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

j-POST: a Java Toolchain for Property-Oriented Software Testing

Yliès Falcone[1] Laurent Mounier[1] Jean-Claude Fernandez[1]

Verimag
Université Grenoble I, INPG, CNRS
Grenoble, France

Jean-Luc Richier[1]

LIG Laboratory
Université Grenoble I, INPG, CNRS
Grenoble, France

Abstract

j-POST is an integrated toolchain for property-oriented software testing. This toolchain includes a test designer, a test generator, and a test execution engine. The test generation is based on an original approach which consists of deriving a set of *communicating test processes* obtained both from a requirement formula (expressed in a trace-based logic) and a behavioral specification of some specific parts of the software under test. The test execution engine is then able to coordinate the execution of these test processes against a distributed Java program. j-POST was applied to check the correct deployment of a security policy for a travel management application.

Keywords: property testing, tool, composition, Java

1 Introduction

Testing is a validation technique aimed to find defective behaviours on a system either during its development, or once a final version is issued. It remains one of the most feasible methodologies to ensure the expected behaviour of a software. This is notably due to its ability to cope with continual growth of system complexity. However, reducing its cost and time consumption remains a very important challenge sustained by a strong industrial demand.

In previous work [5,6] we have presented a black-box test generation method able to construct abstract test cases from a formal requirement (a property that

[1] Email: FirstName.LastName@imag.fr

1571-0661/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2008.11.004

the system is expected to fulfill). This method (implemented in a prototype tool) is based on a test calculus allowing the method to be compositionally and formally defined. In this framework, a requirement is expressed by a logical formula built upon a set of (abstract) predicates. Each predicate corresponds to a (possibly non-atomic) operation to be performed on the system under test, and is user-provided as a *test module* indicating how to perform this operation on the actual implementation, and how to decide whether its execution succeeds or not. The test generation step consists in building, by *composition* of test modules, a set of *communicating test processes* from this property. In this paper we present a significant step from this previous work. First off, we present formally how the previously generated test can be executed. Besides we present j-POST, an integrated toolchain for property-oriented software testing. In addition to a full implementation of the test generation tool, we present the associated test designer and test execution engine resulting in a fully integrated toolchain. The test designer helps the user to provide inputs to the test generator. The test execution engine is able to coordinate the execution of the generated processes against a possibly distributed program, leading to a satisfiability verdict with respect to the given requirement.

Comparison with classical model-based testing.

This approach offers several advantages over more classical model-based test generation techniques [15] implemented in several existing tools (*e.g.* TGV [9], TorX [16], see [2,7] for more exhaustive surveys). First, j-POST is able to deal with *piecewise specifications* restricted to specific functionalities. We strongly believe that this feature is really important in practice, especially in application domains where formal modeling of software is not a common practice. Specifying only some global requirement and some specific implementation features in an operational way (*i.e.* the test modules) seems much easier for test engineers than building a complete model of a software. As a consequence, the test generation step will not require the exploration of such a complete model, avoiding the well-known state explosion problem. Furthermore, this toolchain remains *open* in the sense that various logics can be considered to express the requirements, and new logic plugins can be easily added. Finally, this toolchain integrates a large spectrum of the whole test process, from the test design to the test execution.

The remainder of this paper is organized as follows. Sect. 2 describes the underlying theory of j-POST and Sect. 3 describes the toolchain itself. In Sect. 4, we depict one of the experiments conducted with j-POST on a travel agency application. Sect. 5 exposes some conclusions and perspectives opened by this work.

2 Underlying testing theory

This section briefly presents the background of j-POST, namely how to produce and execute test cases from a formal requirement following a syntax-driven approach.

More details can be found in the research reports available in [8].

We consider in the following that the behaviour of the software under test (SUT) can be modelled using a *labelled transition system* (LTS), noted Sut, namely a quadruplet $(Q^{Sut}, Act^{Sut}, \rightarrow, q^0)$ where Q is a set of states, Act^{Sut} a set of actions (labels), $\rightarrow \subseteq Q^{Sut} \times Act^{Sut} \times Q^{Sut}$ the transition relation and $q^0 \in Q^{Sut}$ the initial state. In *black-box* testing this behaviour can be accessed only through a SUT interface, namely a set of *visible* actions $Act^{vis} \subseteq Act^{Sut}$. Non visible actions are supposed to be labelled by τ. We will denote by $p \xrightarrow{a} q$ when $(p, a, q) \in \rightarrow$, and by $p_1 \overset{\tau^* a}{\Longrightarrow} q$ when there exist p_2, p_3, $\ldots$, p_n s.t. $p_i \xrightarrow{\tau} p_{i+1}$ and $p_n \xrightarrow{a} q$. Finally, we define the *execution sequences* of Sut as the set of *finite* sequences of visible actions that can be performed from its initial state: $Exec(Sut) = \{a_1.a_2.\cdots.a_n \mid \exists q_1, \ldots, q_{n+1}\ s.t.\ q_i \overset{\tau^* a_i}{\Longrightarrow} q_{i+1} \wedge q_1 = q^0\}$.

2.1 The properties to test

We assume in the following that the properties to test are expressed using a logic $\mathscr{L}$. Formulas of $\mathscr{L}$ are built upon a finite set of *n-ary operators* F^n and a finite set of *predicates* $\{p_1, p_2, \ldots, p_n\}$. The abstract syntax of such a logic could be defined as follows: $\text{formula} ::= F^n(\text{formula}_1, \text{formula}_2, \ldots, \text{formula}_n) \mid p_i$.

Formulas of $\mathscr{L}$ are interpreted over *finite* execution sequences. However, this semantics also takes into account two other important features:

- First, this semantics is defined on two levels. Predicates are not *atomic*, *i.e.* they do not necessarily correspond to occurrences of *single* visible actions, but rather of (concrete) *sequences* of visible actions. Operators F^n are then interpreted over *abstract* execution sequences, *i.e.*, sequences of predicates.
- Second, since our objective is to *test* either the validity or the non-validity of a formula φ, the semantics of φ defines three kinds of execution sequences, corresponding to the possible verdicts delivered by a tester: the ones that *satisfy* φ (pass), the ones that *do not satisfy* φ (fail), and the ones for which *we cannot conclude* about the satisfiability of φ (inconc).

More formally, a triplet of finite languages $(L^P_{p_i}, L^F_{p_i}, L^I_{p_i})$ is associated with each predicate p_i. These three languages define respectively concrete execution sequences that satisfy p_i, that do not satisfy p_i, and for which the satisfiability of p_i is unknown. The following assumptions are required:

- $L^P_{p_i}$, $L^F_{p_i}$ and $L^I_{p_i}$ are defined over an alphabet $A_{p_i} \subseteq Act^{vis}$. Intuitively, A_{p_i} is the set of visible actions whose occurrences influence the truth value of p_i.
- This set of three languages defines a *partition* of $(A_{p_i})^*$.
- For two distinct predicates p_i, p_j, A_{p_i} and A_{p_j} are disjoint.

The semantics of a non-atomic formula $\varphi(p_1, p_2, \ldots, p_n)$ is then defined by three sets $[\![\varphi]\!]^P$, $[\![\varphi]\!]^F$, $[\![\varphi]\!]^I$, inductively computed from $L^P_{p_i}$, $L^F_{p_i}$ and $L^I_{p_i}$ for each p_i appearing in φ.

Finally, we say that an LTS S satisfies φ (we note $S \models \varphi$) iff *all* sequences of $Exec(S)$ belong to $[\![\varphi]\!]^P$, and we say that it does not satisfy φ iff there exists a sequence of $Exec(S)$ that belongs to $[\![\varphi]\!]^F$.

2.2 A set of communicating test processes

The test cases we aim to produce consist of a set of *sequential communicating test processes*. Roughly speaking, each test process is built from classical programming primitives such as variable assignment, sequential composition, (non-deterministic) choice, and iteration. It can also perform communications with the other test processes, and interact with the SUT. This sequential behaviour can be modelled by an LTS extended with variables.

Test processes run asynchronously and communicate with each other either by "rendez-vous" on dedicated communication channels or through shared variables. The semantics of a whole test process T_φ can be expressed by an LTS S_{T_φ}. A complete syntax and semantics of such a "test calculus" can be found in [5], but other classical process algebra could be used as well.

2.3 Test generation

The purpose of the test generation phase [6] is to produce a test case T_φ (*i.e.* a set of communicating test processes) associated to the $\mathscr{L}$-formula φ under test. We distinguish two kinds of test processes (which are both LTSs):

- *test modules* t_{p_i}, provided by the user, and associated with the predicates p_i of φ. Their purpose is to produce a test verdict indicating whether a given concrete execution sequence belongs either to $L^P_{p_i}$, $L^F_{p_i}$ or $L^I_{p_i}$. Examples of such test modules are given on Fig. 5 in Sect. 4.
- *test controllers* t_{F^n}, associated with each n-ary operator F^n of the logic $\mathscr{L}$. Their purpose is to control the execution of the test process associated to each of their operands by means of basic signals (start, stop, loop), and to collect their verdicts in order to produce a resulting verdict corresponding to this instance of operator F^n. One can find controllers for several logics in the research reports provided in [8].

This test generation technique can be formalized by a function called $GenTest_{\mathscr{L}}$, such that $GenTest_{\mathscr{L}}(\varphi) = T_\varphi$. This function is inductively defined on the syntax of $\mathscr{L}$ in the following way:

- If $\varphi = p_i$, then $GenTest_{\mathscr{L}}(\varphi)$ returns the test module t_{p_i} (associated with the predicate p_i) extended with the communication operations required to make it controlable by another test process (see Fig. 3 in Sect. 3).
- If $\varphi = F^n(\varphi_1, \cdots, \varphi_n)$, then $GenTest_{\mathscr{L}}(\varphi)$ returns a parallel composition between (recursively defined) test processes $t_{\varphi_1}, \ldots, t_{\varphi_n}$ and an instance of the (generic) test controller t_{F^n}.

Finally, a special test process t_{main} is added to launch the whole test execution and collect the final verdict. According to this generation technique, the architecture of a test case T_φ exactly matches the abstract syntax tree corresponding to formula φ: the root is t_{main}, leaves are test modules corresponding to predicates p_i of φ, and intermediate nodes are controllers associated with operators of φ (see Fig. 6 in Sect. 4.2 for an example).

2.4 Test selection and execution

From a formal point of view, the test execution sequences are the execution sequences of a parallel composition between the LTS S modelling the SUT behaviour and the test case S_{T_φ}, with a "rendez-vous" synchronization on the visible actions appearing in S_{T_φ}.

However, this LTS product may still contain a bunch of possible test executions (due to possible non-determinism both inside the test modules and introduced by the parallel composition). Moreover, the test generation function only ensures that the verdicts produced by the test execution are *sound* with respect to the initial formula φ: it does not help to select the *interesting* test executions that are likely to exhibit an incorrect behaviour of the SUT. To solve this problem we propose to use *behavioural test objectives*, already introduced in several model-based testing tools (*e.g.* in [9,14]). Their purpose is to inject some execution scenario in the test cases produced by the test generation phase, either by enforcing the execution order of some visible actions, or by introducing other additional visible actions to lead the SUT into some particular state. Most of the time, in specification based testing, this test selection is performed during the test generation phase, by pruning the undesired test executions from the whole SUT specification. In our approach, the selection is not performed during test generation, but during the test execution (similarly to walk guidance in TorX). This is due to the fact that we do not rely on such a specification. So, the test selection phase is combined with the test execution: the test objective is expressed by an LTS with *accepting* states, and the test sequences leading to such states are privileged during the text execution.

This approach is formalized below. In a LTS S, for two states q, q' we note $q' \in Reach_S(q)$ the fact that q' is accessible from q in S. Also, when $q' \in Reach_S(q)$, we note $d_S(q, q')$ the distance between q and q', *i.e.* the minimal length of the existing paths between q and q'.

Definition 2.1 [Behavioural test objective] A test objective O relatively to a test case t which semantics can be expressed by a LTS $(Q^{S_t}, Act^{S_t}, \rightarrow_{S_t}, q_0^{S_t})$ is a deterministic LTS $(Q^O, Act^O, \rightarrow_O, q_0^O)$ complete wrt. Act^{S_t} (*i.e.* $\forall q \in Q^{S_t}, \forall a \in Act^{S_t}, \exists q' \in Q^O \cdot q \xrightarrow{a}_O q'$). Q^O contains two sink states $Accept^O$ and $Reject^O$, and $Act^O \subseteq Act^{vis}$.

Using a test objective, it is possible to operate "on the fly" a test selection on the test case during the execution.

Definition 2.2 [Selection using a behavioural test objective] Let t be a test case

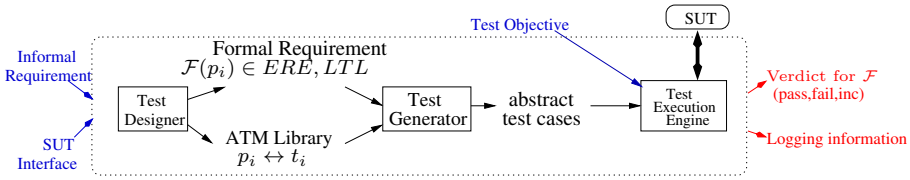


Fig. 1. Abstract view of the j-POST testing toolchain

which semantics can be expressed by $S_t = (Q^{S_t}, Act^{S_t}, \rightarrow_{S_t}, q_0^{S_t})$, and a behavioural test objective $O = (Q^O, Act^O, \rightarrow_O, q_0^O)$. The execution of t guided by O can be defined as a synchronous product $S_t^O = S_t \times O$ such as $Act^{S_t^O} = Act^O$, $Q^{S_t^O} \subseteq Q^{S_t} \times Q^O \cup Inc$, and $\rightarrow_{S_t^O}$ is defined by the following rules. Note that control and observation actions are not distinguished.

$$\frac{t \xrightarrow{a}_{S_t} t' \qquad Accept^O \in Reach_O(o), o \xrightarrow{a}_O o' \qquad d_S(o', Accept^O) < d_S(o, Accept^O), a \in Act^{S_t}}{(t,o) \xrightarrow{a}_{S_t^O} (t', o')} \quad (1)$$

$$\frac{Accept^O \in Reach_O(o), o \xrightarrow{a}_{S_O} o' \qquad d_S(o', Accept^O) < d_S(o, Accept^O), a \notin Act^{S_t}}{(t,o) \xrightarrow{a}_{S_t^O} (t, o')} \quad (1')$$

$$\frac{t \xrightarrow{a}_{S_t} t' \qquad Accept^O \in Reach_O(o), o \xrightarrow{a}_O o' \qquad d_S(o', Accept^O) \geq d_S(o, Accept^O)}{(t,o) \xrightarrow{a}_{S_t^O} (t', o')} \quad (2)$$

$$\frac{o \xrightarrow{a}_O Reject^O}{(t,o) \xrightarrow{a}_{S_t^O} Inc} \quad (3)$$

Some priorities are associated with these rules to favour the execution of transitions bringing closer to an *Accept* state. The rules (1) and (2) are of the highest prioriy, then is rule (2), and at last the rule (3) is of the lowest priority.

Finally, the set of test execution sequences obtained from an SUT S and a test case S_{T_φ} when taking into account a test objective O is defined as the execution sequences of the parallel composition between the SUT S and the LTS $S_{T_\varphi} \times O$. Note that when the Inc state is reached in this composition, the whole test execution is stopped and an inconclusive verdict is issued. This general framework has been instantiated for two particular logics, namely LTL-X, and extended regular expressions (see Sect. 3.2).

3 Architecture and functionalities of j-POST

The architecture of the toolchain is depicted in Fig. 1. It is built upon three main components, a test designer, a test generator and a test execution engine. Two interfaces are provided: a command-line mode and a graphic interface.

The purpose of j-POST is to check through black-box testing whether a Java application fulfills a given requirement. To do so, the *test designer* (step 1) helps the user both to formalize this requirement in a trace-based logic and to elaborate a test module library. Each test module (corresponding to a predicate used in the requirement) is obtained by combining some of the actions offered by the SUT interface. The test modules are used by the *test generator* (step 2), according to a

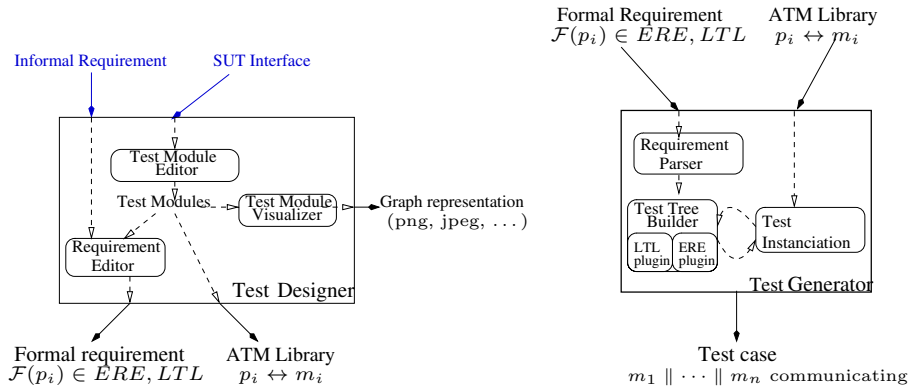


Fig. 2. Abstract view of the j-POST test designer and test generator architecture

logic plugin, to produce a test case as a set of communicating test processes. Finally, this test case can be launched by the *test engine* (step 3), taking into account a test objective to select the more promising test sequences.

3.1 Test designer

The test designer of j-POST is a user assistant that helps to elaborate the formal requirements and the corresponding test modules through dedicated editors available within the Eclipse Modeling Framework. Each test module is stored into an XML file (their j-POST internal representation). Moreover, the test designer provides a tool (based on GraphViz [1]) to vizualise them in a more intelligible way. This avoids any error-prone manipulations of XML files from the user.

3.2 Test generator

The j-POST test generator consists mainly in implementing the $GenTest_{\mathscr{L}}$ function. It produces a test case following the syntax-driven approach recalled in Sect. 2.3 in two stages:

The first stage is the construction of a communication tree obtained from the abstract syntax tree of the formula. This tree expresses the communication architecture between the test processes that will be produced by the test generator. Its leaves are abstract test modules (ATM) corresponding to the atomic predicates of the formula, taken from the library. Its internal nodes are (copies of) generic test controllers, corresponding to the logical operators appearing in the formula (they are obtained from a finite set of generic controllers provided by the logic plugin). Finally, the root of this tree is a special test process, called `testCaseLauncher`, whose purpose is to initiate the test execution and deliver the resulting verdict.

The second stage consists of *instantiating* the communication tree by associating fresh channel names to each local communication between test processes. It relies on a traversal of this communication tree in order to modify the test modules. In particular the test modules provided by the user are automatically extended with additional communication actions to be managed by the test controllers, *e.g.* starting signal, verdict emission (see Fig. 3). The resulting test case is a set of XML

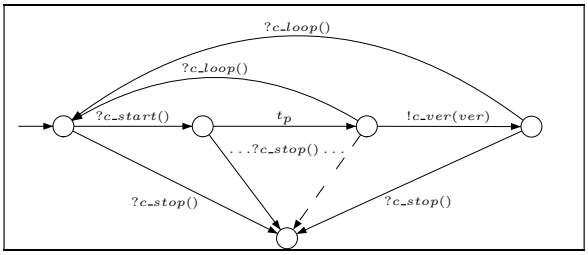


Fig. 3. Instantiation of an abstract test component t_p

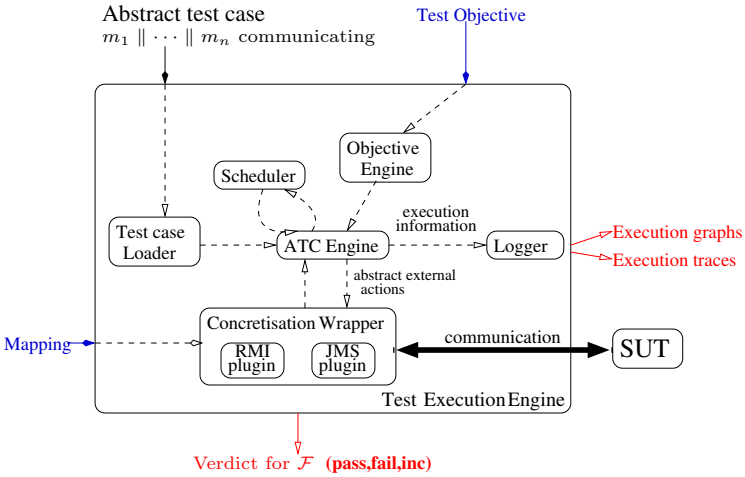


Fig. 4. Abstract view of the j-POST test execution engine

files, one per test process.

Generic test controllers and test generation algorithms have been defined for different specification formalisms. So far, j-POST TestGenerator supports two common-use formalisms, by means of *logic plugins*:

- Temporal logics [12] like LTL are frequently used in the verification community to express requirements on reactive systems. We consider here fragments of such logics whose models are set of *finite* execution traces. We did not include the next operator in order to be insensitive to stuttering [3]. The complete definition of the variant of LTL-X we use is given in [6].
- Extended Regular Expressions [10] are another formalism to define behavior patterns expressed by finite execution traces. They are commonly used and well-understood by engineers.

3.3 Test execution engine

The purpose of the test execution engine is to produce a verdict for the initial requirement. It takes as inputs the test case produced by the test generator, a test objective, and a mapping describing how to execute SUT interactions used in the test modules.

The architecture of the engine is depicted in Fig. 4. First the test case (a set of

XML files) is loaded using the test case loader. Each test process is executed in a separate Java thread. A centralized scheduler implements both the internal communications between the test processes (based on "rendez-vous" and shared variables), and solves the priority conflicts between their actions (according to a predefined policy). Moreover, interactions to be performed on the SUT transit through a Concretisation Wrapper. This component is in charge of transforming these interactions into executable operations on the SUT (depending on the communication medium used, *e.g.* Java RMI). This transformation may also add some parameters omitted at the test module level (for the sake of simplicity), but mandatory for the test execution. Finally, the test selection operation described in Sect. 2.4 is performed by the Objective Engine. When the test execution terminates a verdict is issued and the Logger produces some execution traces that help the diagnostic phase.

4 j-POST at work

We describe in this section the use of j-POST on an example. Tests are designed, generated, executed using the j-POST toolchain to check some properties on a travel agency application [4], called *Travel*. We take as inputs an informal requirement extracted from the functional specification of *Travel* and the application interface. The requirement we choose for the demonstration purpose is informally expressed as "it is impossible to create a mission in *Travel* before being connected".

4.1 Test design

We start by presenting the test design stage, that is the requirement formalization and the edition of test modules.

Requirement formalization.

A possible understanding of our requirement could be that a behaviour in which it is possible to create a mission before performing the identification action is not desired. In other words, we can say that we require no mission creation *until* a connection is open. This informal statement refers to two abstract operations: "create a mission", and "open a connection". In the following we respectively designate these two operations by the predicates $missionCreation()$ and $connection()$. The requirement can be expressed formally by an LTL formula: $(\neg missionCreation()) \ \mathcal{U} \ connection()$.

Test module edition.

Test modules have to be created by the user for the predicates $missionCreation()$ and $connection()$. Each of this module should describe:

- how to perform the abstract operation using the *Travel* interface;
- what is the *test verdict* obtained (depending on how *Travel* reacts).

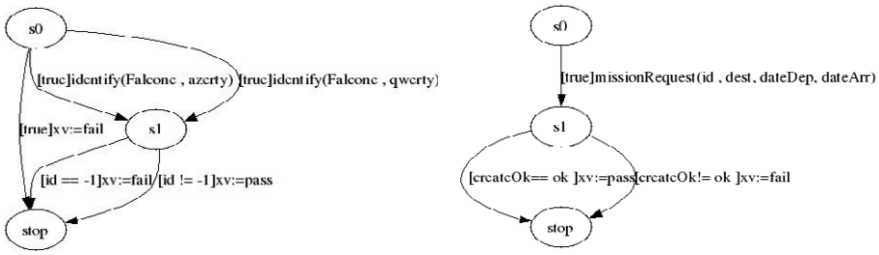


Fig. 5. Test modules for predicates *connection* and *createMission*

Possible test modules are proposed in Fig. 5, produced with the j-POST test designer. The *connection* test module (left-hand side) contains three possible execution sequences: a correct call to the connection method `identify` (the user is "Falcone", the correct password is "azerty", which corresponds to a registered user of *Travel*), an incorrect one (the password is "qwerty", it is not valid), and an execution where the connection procedure is never called. Note that the call to the `identify()` method returns an identification number which is stored in a *shared* variable (between test components) called `id`. The *createMission* test module (right-hand side) consists of calling the `missionRequest()` method, supplying the shared variable `id` as an identification number. Depending on the return value (`createOk`), it delivers the corresponding verdict.

Inside the toolchain these modules are represented using an XML format, but, from a practical point of view, they can be written and viewed using the j-POST test designer.

4.2 Test generation

The requirement stated, and the test modules designed (Fig. 5), we are now able to perform the test generation. In order to illustrate such a process, we give an insight of the generated test case on Fig. 6. The structure of this test case follows the structure of the formula. It contains a test controller for each operator appearing in the formula (*Until* and *Not*), and a test module for each predicate (*missionCreation()* and *Connection()*). The `testCaseLauncher` is in charge of managing the execution of the testcase and emitting the final verdict. The *c_start* (resp. *c_stop*, *c_loop*, *c_ver*) channels are used by the processes to perform starting (resp. stopping, rebooting, verdict transmission) operations.

4.3 Test execution

The next operation to perform is to choose a *test objective* in order to restrict the set of potential test executions. Regarding the requirement we consider ("no mission creation until a connection is open"), an interesting objective is to try to *falsify* this requirement in order to exhibit an incorrect behaviour of the software under test. Falsifying such a requirement means for instance producing an execution sequence where :

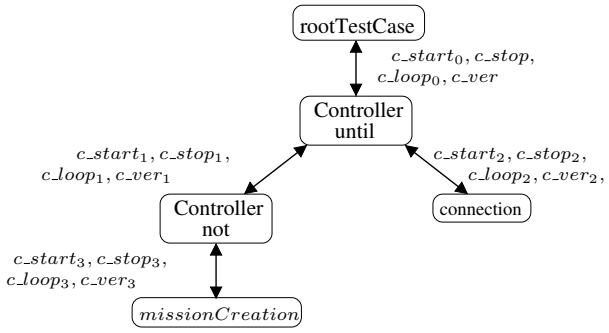


Fig. 6. Test case produced from $\neg(missionCreation)\,\mathcal{U}\,connection$

- the verdict delivered by $missionCreation()$ is *pass* (possibly after several previous *fail* results) ;
- in the meantime, the verdict delivered by $connection()$ remains always *fail*.

Such a test objective can be obtained from the test modules given on Fig. 5. However, obtaining a *fail* verdict for a connection operation can be fully controlled by the test execution engine (*e.g.*, by supplying an incorrect password), whereas the verdict returned by a mission creation cannot be controlled (it only depends on the SUT behaviour).

Three versions of the *Travel* application have been tested:

- Experiment 1. In the first (erroneous) version of *Travel* a mission creation request is always accepted, therefore our requirement is false (a mission can be created by a non connected user). The test execution engine detects this error (it delivers a *fail* verdict) and produces the test execution traces and graphs for the test case and each module.
- Experiment 2. In the second (erroneous) version of *Travel* a mission creation request is accepted either if the identification number supplied is correct (it corresponds to a return value of a connection request), or if it is the third attempt to create this mission. Therefore our requirement is still false: if a non connected user tries repeatedly to create a mission, it eventually succeeds. This error is detected by the test engine, which delivers a *fail* verdict.
- *Experiment 3.* Finally, the third version of *Travel* always refuses a mission request as long as the identification number supplied is invalid. Thus, the only way for a non connected user to create a mission is to "guess" a correct identification number. This cannot be achieved by our test execution engine, which delivers here a *pass* verdict.

5 Conclusion and perspectives

This paper presents an original approach for property-oriented software testing (POST). Starting from a formula expressed in a trace-based logic, the user

first provides a test module (using the test designer) dedicated to each predicate appearing in this formula. The test generation phase then consists of producing a test case as a set of communicating test processes by combining the test modules with some test controllers associated to each logical operator. This test case can be executed by a test engine, able to take into account a test objective to constrain the set of test sequences to execute. This whole testing approach has been implemented in a working tool and applied to some non-trivial case studies. The architecture makes it *open*, and easily allows the toolchain to support new logical formalisms by adding logic plugins.

The main advantage of this approach is that it does not require a "global" behavioural specification of the software under test, as is the case in many model-based testing approaches. In fact the user only needs to make explicit the evaluation of a predicate in the test modules. The test generation phase is therefore rather straightforward and does not suffer from state explosion limitations. Of course, the test case produced may encompass many possible test executions, but the use of test objective allows the user to select the most interesting scenarios. This approach seems particularly relevant to dealing with security or robustness testing, where the functional model of the SUT can be very large (and hence not easily available as a single formal specification), and where the requirements to be checked only concern specific parts of this model. In fact, one of the motivations for this work was the validation of the correct deployment of security policies within the French Politess [13] project.

The *Travel* case study allowed many enhancements for j-POST and opens several research perspectives. In particular, it appears that the design of test modules could be facilitated by the use of abstract domains (*e.g.*, at the test module level one only needs to distinguish between *correct* passwords and *incorrect* ones, without referring to a concrete value). These abstract domains could then be concretized only at the test execution level by selecting relevant values within a concrete domain (which may depend on the SUT's current state). This concretisation could be performed, for instance, according to coverage criteria that could be defined with respect to the requirement under test. It seems particularly worthwile to relate this work with [11].

References

[1] AT&T Research, *Graph Visualization Software*, http://www.graphviz.org (2007).

[2] Belinfante, A., L. Frantzen and C. Schallhart, *Tools for Test Case Generation.*, in: M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, LNCS 3472, 2004, pp. 391–438.

[3] Clarke, E., O. Grumberg and S. Peled, "Model Checking," The MIT Press, 1997.

[4] Falcone, Y., *A Travel Agency Application*, Technical report, Vérimag (2007).

[5] Falcone, Y., J.-C. Fernandez, L. Mounier and J.-L. Richier, *A Test Calculus Framework Applied to Network Security Policies.*, in: *FATES/RV*, LNCS 4262, 2006, pp. 55–69.

[6] Falcone, Y., J.-C. Fernandez, L. Mounier and J.-L. Richier, *A Compositional Testing Framework Driven by Partial Specifications*, in: *TestCom/FATES*, LNCS 4581, 2007, pp. 107–122.

[7] Hartman, A., *Model Based Test Generation Tools Survey*, Technical report, AGEDIS Consortium (2002).

[8] j-POST Reference Page (2007). URL http://www-verimag.imag.fr/~async/jpost.html

[9] Jard, C. and T. Jéron, *TGV: theory, principles and algorithms, A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems*, Software Tools for Technology Transfer (STTT) **6** (2004).

[10] Kleene, S. C., *Representation of events in nerve nets and finite automata*, in: C. E. Shannon and J. McCarthy, editors, *Automata Studies*, Princeton University Press, Princeton, New Jersey, 1956 pp. 3–41.

[11] Lestiennes, G. and M.-C. Gaudel, *Testing processes from formal specifications with inputs, outputs and data types*, in: *ISSRE* (2002), pp. 3–14.

[12] Manna, Z. and A. Pnueli, "Temporal verification of reactive systems: safety," Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[13] Project Politess, *ANR-05-RNRT-01301* (2007). URL http://www.rnrt-politess.info

[14] Schmitt, M., M. Ebner and J. Grabowski, *Test Generation with Autolink and Testcomposer*, in: *2nd Workshop of the SDL Forum Society on SDL and MSC - SAM'*, 2002.

[15] Tretmans, J., *Test Generation with Inputs, Outputs and Repetitive Quiescence*, Software - Concepts and Tools **17** (1996), pp. 103–120.

[16] Tretmans, J. and E. Brinksma, *TorX: Automated Model Based Testing - Côte de Resyste*, in: *Proceedings of the First European Conference on Model-Driven Software Engineering*, 2003, pp. 13–25.

ANNEXE D

Liste des définitions, théorèmes, lemmes, propositions, et exemples

D.1 Liste des définitions

1 Vérification [IEE05] . . . 2
2 Propriétés et ensembles de séquences d'exécution . . . 13
3 Runtime Verification - Vérification à l'exécution ([LS08]) . . . 17
4 Moniteur . . . 18
5 Limite de propriétés . . . 34
6 Propriété infinitaire limite-close . . . 34
7 Safety/Liveness . . . 34
8 r-propriétés . . . 38
9 Opérateurs A, E, R, P . . . 39
10 Opérateurs A_f, E_f, R_f, P_f . . . 39
11 r-propriétés des classes de base . . . 40
12 Extension minimale d'un langage par un autre . . . 42
13 Sémantique des r-propriétés en vision logique . . . 45
14 Fonction Sat d'interprétation des formules . . . 46
15 m-automate de Streett . . . 49
16 Condition d'acceptation (séquences infinies) [CMP92a] . . . 50
17 Condition d'acceptation (séquences finies) . . . 50
18 DFA vers automate de Streett de safety . . . 52
19 DFA vers automate de Streett de guarantee . . . 53
20 DFA vers automate de Streett de response . . . 54
21 DFA vers automate de Streett de persistence . . . 55
22 Détermination Positive/Négative [PZ06] . . . 66
23 Monitorabilité [PZ06] r-propriétés . . . 66
24 Évaluation de propriété dans un domaine de vérité . . . 70
25 Monitorabilité . . . 70
26 Critère d'enforcement sur une r-propriété . . . 73
27 Critère suffisant d'enforcement dans la vue automate . . . 74
28 Relations entre les séquences du programme et une propriété [NGH93] . . . 79
29 Verdicts [NGH93] . . . 79
30 Testabilité . . . 79
31 Caractérisation des états d'un automates de Streett . . . 93
32 Propriétés testables (vue automate) . . . 96
33 Moniteur . . . 96
34 Moniteur de vérification . . . 99

35 Séquence de vérification . . . 99
36 Vérification de propriété . . . 100
37 Transformation Streett2VM . . . 100
38 Opérations d'enforcement *Ops* . . . 100
39 Moniteur d'enforcement générique (EM(*Ops*)) . . . 101
40 Run et trace . . . 101
41 Configurations et dérivations d'un EM(*Ops*) . . . 101
42 Transformation de séquences depuis un état et un contenu mémoire . . . 101
43 Transformation de séquences par un moniteur . . . 101
44 Transformation de séquence . . . 102
45 Enforcement d'une r-propriété . . . 102
46 Opérations d'enforcement $Ops = \{halt, store, dump, off\}$ . . . 103
47 Négation d'un EM . . . 106
48 Union de EMs . . . 108
49 Intersection d'EMs . . . 108
50 Transformation Safety . . . 111
51 Transformation Guarantee . . . 112
52 Transformation Obligation . . . 113
53 Transformation Response . . . 114
54 Transformation Streett2EM . . . 118
55 Fonction *GenTest*(Π) . . . 127
56 Fonction d'instantiation des tests *TestInstantiation* . . . 128
57 Fonction *Test* dédiée aux modules de test . . . 129
58 Fonction *send* . . . 130
59 Scénario de tests . . . 130

D.2 Liste des Lemmes, Théorèmes, et Propositions

Lemme α Formes normales des r-propriétés d'obligation . . . 43
Théorème α La classe des formules de reactivity est la plus générale . . . 48
Théorème β Correspondance vue logique et vue langage . . . 49
Lemme β Oubli de paires d'acceptation pour les propriétés d'obligation . . . 51
Théorème γ Correction des transformations *DFA2StreettX* . . . 55
Lemme γ Combinaison booléennes de propriétés monitorables . . . 67
Théorème δ Obligation(Σ) ⊂ $MP(\mathbb{B}_3)$ . . . 67
Lemme δ $MP^*(\mathbb{B}_3)$ et propriétés de reactivity qui ne sont pas des safety ou guarantee . . . 70
Théorème ε Caractérisation multivaluée de la monitorabilité . . . 71
Théorème ζ Les r-propriétés de *response* sont enforçables . . . 75
Théorème η Les r-propriétés de persistence enforçables sont des *response* . . . 76
Théorème ϑ Condition suffisante pour établir un verdict pour $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ . . . 80
Théorème ι Condition suffisante pour la relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ et un verdict *pass* . . . 83
Théorème κ Correction de Streett2VM . . . 100
Théorème λ Correction de l'opération de négation d'un EM . . . 106
Théorème μ Correction des opérations d'union et intersection d'EMs . . . 109
Théorème ν . . . 115
Théorème ξ correction de Streett2EM . . . 119

D.3 Liste des Exemples

1 Propriétés informelles pour les classes de base . . . 38
2 Construction de propriétés finitaires et infinitaires . . . 41
3 r-propriétés . . . 41
4 Propriété finitaire, formule du passé, et *esat*() . . . 46
5 Spécification de r-propriétés par des automates de Streett . . . 50
6 Classes d'automates et restrictions syntaxiques . . . 51

7 DFA vers automate de Streett de safety . . . 53
8 DFA vers automate de Streett de guarantee . . . 53
9 DFA vers automate de Streett de response . . . 54
10 Propriété de *response* non monitorable [BLS07c] . . . 68
11 Propriété de *response* monitorable . . . 68
12 Monitoring d'une propriété d'obligation . . . 72
13 Une r-propriété de (pure) persistence non enforçable . . . 76
14 Testabilité de certaines r-propriétés vis-à-vis de $\mathit{Exec}(\mathcal{P}_\Sigma) \subseteq \Pi$ . . . 82
15 Testabilité de certaines propriétés vis-à-vis de $\mathit{Exec}(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ . . . 84
16 Testabilité d'une r-propriété vis-à-vis de $\mathit{Exec}(\mathcal{P}_\Sigma) \subseteq \Pi$ avec quiescence . . . 86
17 Caractérisation des états d'un automate de Streett . . . 94
18 Caractérisation des états d'un automates de Streett . . . 94
19 Moniteur d'enforcement . . . 103
20 Union de EMs . . . 108
21 Transformation safety . . . 111
22 Transformation Safety . . . 111
23 Transformation guarantee . . . 112
24 Transformation guarantee . . . 112
25 Transformation indirecte pour les obligations . . . 113
26 Transformation spécifique aux obligations . . . 113
27 Transformation response . . . 114
28 Transformation Response . . . 115
29 Testeur abstrait obtenu à partir de *GenTest* . . . 128
30 Propriété informelle sur un programma Java . . . 139
31 Propriété informelle sur un programma Java . . . 139
32 Conception de propriété . . . 140
33 Conception de propriété . . . 141
34 Application de *DFA2Streett* . . . 143
35 Application de *DFA2Streett* . . . 144
36 Application de *Streett2Monitor* . . . 145
37 Application de *Streett2Monitor* . . . 145
38 Génération d'un moniteur avec *MonitorTranslator* . . . 148
39 Modules de test . . . 161
40 Extrait de mapping . . . 172

Table des figures

1 Moniteurs de vérification (gauche) et d'enforcement (droite) 4

3.1 Les classes de la classification *Safety-Progress* en représentation ensembliste 36
3.3 Illustration informelle des différences entre les classes (de base) de propriétés 37
3.2 Hiérarchie des classes 37
3.4 Automate reconnaisseur pour la r-propriété de safety Π_1 $(P = \{1, 3\})$ 50
3.5 Automate reconnaisseur pour la r-propriété de guarantee Π_2 $(R = \{3\})$ 50
3.6 Automate reconnaisseur pour la r-propriété de 1-obligation Π_3 $(P = \{1\}$ et $R = \{3\})$ 51
3.7 Automate reconnaisseur pour la r-propriété de response Π_4 $(R = \{1\})$ 51
3.8 Automate reconnaisseur pour la r-propriété de persistence Π_5 $(P = \{3\})$ 52
3.9 Illustrations schématiques de la forme des automates de Streett pour les classes de base . 52
3.10 DFA reconnaissant la propriété finitaire $a^* \cdot (b^* + c \cdot (c + a)^* \cdot b^+)$ 53
3.11 DFA2StreettSafety appliquée sur le DFA de la Fig. 3.10 $(P = \{1, 2\})$ 53
3.12 DFA $\mathcal{A}_\psi$ reconnaissant la propriété finitaire $\psi = (a \cdot b)^+$ 54
3.13 DFA2StreettGuarantee appliquée sur le DFA de la Fig. 3.12 $(R = \{3\})$ 54
3.14 DFA2StreettResponse appliquée sur le DFA de la Fig. 3.12 $((R = \{3\})$ 54
3.15 Principe du marquage des états persistants pour DFA2StreettPersistence 55
3.16 Représentation hiérarchique de la classification *Safety-Progress* des r-propriétés 60

4.1 r-propriétés de *response* non monitorable (gauche) et monitorable (droite) 68
4.2 Évaluation d'une r-propriété dans les différents domaines de vérité 70
4.3 r-propriétés monitorables dans la classification *Safety-Progress* 72
4.4 Schéma de principe de l'enforcement 73
4.5 Automate de 2-reactivity pour lequel (4.1) $\not\Rightarrow$ (4.2) 75
4.6 Une r-propriété de (pure) persistence non enforçable 76
4.7 r-propriétés monitorables et enforçables dans la classification *Safety-Progress* 78

6.1 Union de deux moniteurs d'enforcement : $\mathcal{A}_{\downarrow e_1}$ et $\mathcal{A}_{\downarrow e_2}$ 109
6.2 Automate reconnaisseur et EM pour la r-propriété de safety $(a^* + a^+b^*, a^\omega + a^+b^\omega)$ 111
6.3 Automate reconnaisseur et EM pour la r-propriété de safety Π_1 112
6.4 Automate reconnaisseur et EM pour la r-propriété de guarantee $(b^* \cdot a^+ \cdot b \cdot \Sigma^*, b^* \cdot a^+ \cdot b \cdot \Sigma^\omega)$ 112
6.5 Automate reconnaisseur et EM pour la r-propriété de guarantee pour Π_2 113
6.6 Un automate de 1-obligation et l'EM correspondant pour la propriété Π_3 114
6.7 Un automate de response et l'EM correspondant pour la r-propriété $((a^* \cdot b)^*, (a^* \cdot b)^\omega$ . . . 114
6.8 Un automate de response et l'EM correspondant pour la r-propriété Π_4 115

7.1 Vue d'ensemble de l'approche de test 123
7.2 Architecture de test 125
7.3 Règles pour la réécriture de termes 126
7.4 Règles de la modification de l'environnement 126

7.5 Composition de LTSs par rapport à $\|_{cs}$ . . . 127
7.6 Composition de LTSs par rapport à $\ltimes^{\mathcal{I}}$ . . . 127
7.7 Le processus $ED(Prop, \Sigma, cs_{Prop})$ . . . 128
7.8 L'automate $\mathcal{A}_{\Pi_1}$ reconnaissant la r-propriété correspondant à la formule $\Box\,(p_1 \wedge p_2)$ . . . 128
7.9 Le processus Property Manager correspondant à l'automate $\mathcal{A}_{\Pi_1}$ . . . 129
7.10 Le processus ED correspondant à l'ensemble de propositions $Prop_1$ et l'alphabet Σ_1 . . . 129
7.11 Instantiation d'un module de test t_p par la fonction *Test* . . . 129
7.12 Illustration du principe de la fonction *GenTest* compositionelle . . . 132

8.1 Conception de la propriété . . . 139
8.2 DFA reconnaissant la propriété finitaire ψ_1 . . . 140
8.3 DFA reconnaissant la propriété finitaire ψ_2 . . . 141
8.4 Vue d'ensemble de la boite à outils j-VETO . . . 142
8.5 La partie "synthèse de moniteurs" de j-VETO . . . 143
8.6 Automate de Streett de Safety résultant de l'application de *DFA2Streett* sur le DFA reconnaissant ψ_1 . . . 143
8.7 Automate de Streett de Response résultant de l'application de *DFA2Streett* sur le DFA reconnaissant ψ_2 . . . 145
8.8 Moniteur de vérification résultant de l'application de *Streett2Monitor* sur l'automate de Streett reconnaissant $(A_f(\psi_1), A(\psi_1))$ . . . 145
8.9 Moniteur d'enforcement résultant de l'application de *Streett2Monitor* sur l'automate de Streett reconnaissant $(R_f(\psi_2), R(\psi_2))$ . . . 148
8.10 La partie "Intégration du moniteur" de j-VETO . . . 148
8.11 Représentation des outils pour le modèle et les utilitaires des automates . . . 153
8.12 Représentation schématique des opérations sur les fichiers . . . 155

9.1 Vue abstraite de la chaîne d'outils j-POST . . . 159
9.2 Architecture du Test Designer de j-POST . . . 160
9.3 Édition d'un module de test avec le Test Designer . . . 161
9.4 Module de test pour le prédicat *missionCreation* . . . 162
9.5 Module de test pour le prédicat *missionValidation* . . . 163
9.6 Architecture du Test Generator de j-POST . . . 163
9.7 Capture d'écran du Test Generator de j-POST . . . 165
9.8 Représentation graphique du contrôleur pour l'opérateur logique de conjonction . . . 166
9.9 Structure du cas de test produit pour $\neg(missionCreation)\,\mathcal{U}\,connection$ . . . 167
9.10 Arbre syntaxique abstrait correspondant à $(p_1() \vee p_2())\mathcal{U}p_3(a,b,c)$ . . . 167
9.11 Architecture du Test Execution Engine de j-POST . . . 168
9.12 Capture d'écran du moteur d'exécution des tests . . . 168
9.13 Exemple d'objectif de test . . . 170
9.14 Le module de test connection conduit par l'objectif de la Fig. 9.13 . . . 171
9.15 Exécution d'un cas de test sur la version ligne de commande du moteur d'exécution . . . 173
9.16 Fichier de log produit pour le module createMission . . . 173