Mémoire de Stage de Master Recherche

Conception de stratégies pour tester les propriétés de sûreté

Septembre 2008

PAR

Sébastien ROY

Encadrant :
DADEAU Frédéric — Maître de Conférences à l'Université de Franche-Comté

Co-encadrant :
JULLIAND Jacques — Professeur à l'Université de Franche-Comté
MASSON Pierre-Alain — Maître de Conférences à l'Université de Franche-Comté

Composition du Jury :
BOUQUET Fabrice — Professeur à l'Université de Franche-Comté
NICOD Jean-Marc — Maître de Conférences à l'Université de Franche-Comté

Laboratoire Informatique de Franche-Comté, U.F.R. DES SCIENCES ET TECHNIQUES DE L'UNIVERSITE DE FRANCHE-COMTE

Table des matières

Remerciements

Je souhaite tout d'abord à remercier mon encadrants de stage, Frédéric Dadeau, et mes deux co-encadrant Jacques Julliand et Pierre-Alain Masson pour m'avoir énormément guidé dans les lectures, les travaux et l'écriture de ce présent mémoire. Je les remercie tout particulièrement pour leur disponibilité, mais aussi pour m'avoir laissé une grande liberté sur la manière de travailler et sur le rythme de travail. J'espère que les travaux réalisés leur seront utiles, ca serait la récompense minimale pour toutes les lectures et relectures de ces dizaines de pages.

En cette fin de formation de master, il me semble naturel de remercier aussi toute l'équipe enseignante pour m'avoir permis pendant ces trois années de compléter la formation que j'ai eue à l'IUT de Belfort. Trois années pendant lesquelles j'ai appris beaucoup de choses, que ce soit pendant les cours ou en dehors. Ces trois années sont passées très vite, et malgré quelques projets mémorables, provoquant une étrange excitation dans toute la promotion quand l'heure de les rendre approchait, l'ambiance est toujours restée superbe, permettant même, quand on dormait plus beaucoup, de tenir le choc.

Evidemment, je ne peux finir ces remerciements sans parler de mes collègues chercheurs, avec qui je me suis bien amusé, entre les périodes de travail intensives bien sûr. Et une pensée pour les collègues "professionnels" qui nous ont lâché il y a quelques mois, mais qui sont d'une manière ou d'une autre restés présents pendant le stage.

Introduction

Dans le processus de développement d'un logiciel, le test est un point vital : il permet de s'assurer que le travail effectué correspond aux attentes, et de détecter des défaillances du logiciel le plus tôt possible. Cela devient une préoccupation tellement importante qu'on estime que le travail de test de logiciel représente entre 30 et 60% du travail général réalisé durant le développement d'un logiciel.

Et plus les exigences en terme de sûreté et de sécurité sont importantes, plus le logiciel devra être testé. C'est le cas notamment des systèmes assurant des responsabilités importantes, pouvant causer, en cas de dysfonctionnement, des dommages importants, que ce soit en vies humaines ou en argent. Ces systèmes sont appelés *applications critiques*. Pour ces applications, il est nécessaire de s'assurer qu'ils ne peuvent pas être défaillants dans toutes les situations, et demandent donc une quantité et une variété de tests très importantes. C'est dans ce cadre que de nombreuses recherches universitaires et industrielles sont faites pour faciliter le développement de telles applications, en développant des outils et des méthodes pour les tester plus efficacement.

Pour aider les développeurs dans la conception de logiciel, il est devenu courant de rédiger un modèle formel comportemental du système, c'est-à-dire une description schématique traduisant de manière formelle une ou plusieurs spécifications du cahier des charges que doit assurer le logiciel. Ce modèle est une représentation du système, ou plus précisément une abstraction du système : on ne représente pas tous les détails, tout ce qui ne concerne que l'implémentation en elle-même. Ces modèles permettent d'avoir une base fiable, ou au moins que l'on peut fiabiliser, permettant d'augmenter la confiance dans le bon fonctionnement, théorique à ce stade de développement, d'un système.

Le modèle étant une représentation simplifiée ou partielle du système, il est d'une taille moins importante que le système, et son caractère formel lui permet d'être vérifié par des outils pour une partie ou l'ensemble des propriétés exprimées dans le cahier des charges. Mais ces vérifications de propriétés sur le modèle n'assurent pas que l'implémentation vérifie elle aussi ces propriétés, et on ne peut pas appliquer la même démarche de vérification sur l'implémentation, à cause de sa taille et de sa complexité. On est donc contraint de tester l'implémentation pour la valider.

Pour tester l'implémentation, on peut se servir du modèle formel déjà écrit et vérifié pour calculer les tests à exécuter : c'est le principe du *Model-Based Testing*. Il s'agit en fait de valider l'implémentation en utilisant le modèle pour générer des tests. Il existe déjà des outils permettant de générer des suites de tests selon divers critères de sélection, notamment en simulant un scénario, c'est-à-dire une suite d'opérations, sur le modèle (principe de l'animation) pour générer les tests qui permettent de tester l'implémentation dans les conditions désirées. En effet, générer une suite de tests exhaustive est irréaliste, voire impossible dans certains cas, et l'objectif de la génération de tests est de trouver le

compromis entre quantité et qualité des tests de la suite. Et de ce point de vue, les outils utilisent des critères pour sélectionner les tests, comme la couverture des opérations, mais se privent par définition de nombreux tests qui semblent *a priori* inutiles.

Mais dans le cas des applications critiques, il y a la volonté de tester au maximum les applications, y compris dans des situations considérées *a priori* comme improbables dans une utilisation normale. Donc quand on veut tester de manière plus approfondie une propriété, on peut parfois, d'après l'expérience de l'ingénieur sécurité, vouloir tester la propriété dans des situations non testées par des suites de tests "classiques". C'est pour cette raison que sont utilisés les scénarios comme critère de sélection par certains outils. Les tests ainsi générés permettent de tester plus finement la propriété, et donc de détecter éventuellement plus d'erreurs. Mais les scénarios doivent être écrits par les ingénieurs sécurité, et ce travail peut devenir rapidement laborieux. De plus, ces scénarios ont pour objectif de se placer dans des situations spécifiques de l'implémentation, et sont donc très dépendants de la propriété et surtout du système, et évidemment peuvent être très complexes à écrire, et à interpréter aussi.

Cette tâche fastidieuse et difficile d'écriture des scénarios réalisée à la main gagnerait beaucoup à être facilitée, voire automatisée le plus possible, permettant à partir d'une propriété et d'un besoin de test particulier de l'ingénieur, de générer le ou les scénarios correspondants. C'est à ce problème que ces travaux se proposent de contribuer.

Pour cela, le mémoire sera décomposé ainsi :

- La première partie exposera d'abord le contexte scientifique, c'est-à-dire les raisons qui nous ont motivé à réaliser le travail présenté dans ce mémoire, en expliquant le cadre dans lequel le travail se place, à savoir celui du Model-Based Testing, ainsi que les notions de propriétés, de besoins de test et de scénarios qui ont conduit à la réalisation des travaux de ce mémoire. Nous présenterons ensuite les travaux scientifiques ayant servi pour ce projet, notamment ceux réalisés à l'intérieur du laboratoire LIFC permettant de définir des bases pour les travaux, puis ceux dont nous nous sommes inspirés pour progresser durant le stage.
- La seconde partie présentera les contributions réalisées durant le stage. La contribution principale est l'élaboration d'une démarche permettant de générer des scénarios utilisables par d'autres outils à partir d'une propriété et d'un besoin de test. Afin de s'assurer de la faisabilité d'un outil concrétisant ce processus, nous avons developé une application, qui bien qu'incomplète a permis de tester ce processus. Et pour s'assurer de son utilité, nous l'avons expérimenté sur un cas d'étude et nous présenterons les résultats.
- Nous finiront par la conclusion et donnerons quelques perspectives de continuation

Première partie

Contexte

Chapitre 1

Contexte Technologique et Problématique

Les travaux présentés ici ont pour objectif d'aider à la génération de tests d'une propriété sur une implémentation. La conception de modèle formel d'un système devenant progressivement une pratique courante dans le monde du génie logiciel, et les outils permettant de travailler avec se développant de plus en plus, les travaux de ce stage s'intégrent tout naturellement dans la démarche du Model-Based Testing, dont nous présenterons une vue d'ensemble d'abord, puis plus précisément le calcul de tests à partir du modèle.

Après avoir vérifié ou validé une propriété sur le modèle formel, il est nécessaire de vérifier cette propriété sur l'implémentation, pour s'assurer qu'il n'y a pas d'erreur de conformité entre le modèle et l'implémentation. Cette partie présentera aussi la génération de tests à partir d'une propriété, mais aussi la notion de besoin de tests et de scénarios.

Toutes ces explications permettront de mettre en évidence la problématique de ce mémoire, que nous présenterons pour conclure cette partie.

1.1 Le test à partir de modèles

Il ne suffit pas de vérifier un modèle pour une propriété pour s'assurer que le système modèlisé vérifie aussi cette propriété, il est encore nécessaire de vérifier la cohérence entre le modèle et l'implémentation. Une méthode de plus en plus courante est la génération de tests à partir d'un modèle, c'est-à-dire *Model-Based Testing* ou MBT [BJK+05, UL06], dont nous vous présentons ici une vue générale de la démarche, puis de manière plus précise le calcul de tests à partir d'un modèle.

1.1.1 Vue d'ensemble de la démarche MBT

Un des principaux problèmes du principe de tests est l'impossibilité, théorique parfois mais surtout pratique, de tester toutes les situations. L'ensemble des tests possibles est la plupart du temps infini, et dans tous les cas, trop important pour être exécutable sur l'implémentation dans un temps raisonnable. Le défi lors de la génération de tests est donc de sélectionner les tests qui ont le plus de chance de mettre en évidence des défauts dans l'implémentation.

Le modèle formel de comportements Au lieu d'écrire à la main des centaines de tests dit *boîte noire* [Bei95], c'est-à-dire en ignorant toute information relative à la structure du logiciel, mais en se basant uniquement sur les spécifications fournies par le cahier des charges, les testeurs doivent concevoir un modèle des comportements attendus de l'implémentation sous test, qui comprend tout ou partie des spécifications du cahier des charges. Un modèle est une réprésentation simplifiée (les détails de l'implémentation sont omis) et partielle (seules les parties intéressantes peuvent être modèlisées). Le modèle est donc d'une taille moins importante que le système lui même, et ne demande pas autant de temps à concevoir, d'autant plus qu'il existe en général plus ou moins déjà sous diverses formes, et qu'il suffit de le rédiger concrètement. Le modèle est formel, c'est-à-dire décrit par des notations mathématiques et logiques, facilement gérable par des outils. L'utilisation d'un modèle formel de comportement permet aussi d'utiliser des outils vérifiant le modèle, sa cohérence et son respect des fonctionnalités prévues par le cahier des charges.

Une fois le modèle formel conçu, nous pouvons utiliser un des nombreux générateurs de tests existants pour générer automatiquement une suite de tests à partir de ce modèle. Extraire un test à partir du modèle, ou jouer un test sur le modèle permet d'avoir le résultat prévu par les spécifications du cahier des charges, sous condition que le modèle ait été vérifié avant. Il existe de nombreux outils proposant la génération de tests dans le cadre du Model-Based Testing, chacun possèdant ses caractéristiques et ses objectifs. Citons pour exemple TGV [FJJV96, JJ05], LTG [JL07], développé à partir de BZ-TT [LPU02] dont le principe est expliqué un peu plus en détail en 1.1.2, TOBIAS [LDdB+07], Spec Explorer [CGN+05], ASML Test Tool [BGS+03], STG [CJRZ02] ou JPOST [YFR08].

Un oracle pour tous Dans le processus général de test, après toute exécution de tests, il faut décider si le comportement observé est une faille ou non, s'il met en évidence une incohérence entre l'implémentation et le cahier des charges. C'est ce qu'on appelle le problème de l'oracle. Il est souvent résolu par une observation manuelle du résultat, mais cette tâche longue et répétitive doit être automatisée pour améliorer l'efficacité du processus général de test. Utiliser le modèle pour générer les tests permet d'avoir un oracle pour chaque test, car le modèle possède les informations nécessaires pour prévoir le résultat spécifié pour un test.

Démarche du Model-Based Testing La figure 1.1 présente la démarche générale du Model-Based Testing. La première étape est de concevoir le modèle formel de comportements, ou en adapter un si l'ingénieur en avait déjà utilisé un pour ce système. Le modèle doit être vérifié. On le donne en entrée à un générateur de tests, qui en fonction du critère de sélection génère une suite de tests.

Les tests générés sont des tests abstraits, c'est-à-dire ne contenant que des opérations abstraites et valeurs utilisées par le modèle, ce qui est normal car le modèle ne dispose pas des détails de l'implémentation. Il faut donc les concrétiser pour pouvoir les passer sur l'implémentation sous test (IST). Cette transformation se fait par une couche d'adaptation.

Il ne reste plus qu'à exécuter la suite de tests concrétisés sur l'IST et à comparer les résultats obtenus et prévus. En cas de non conformité, il y a deux possibilités, si on met de côté des erreurs lors de la concrétisation (très rapidement décelées et corrigées) : soit

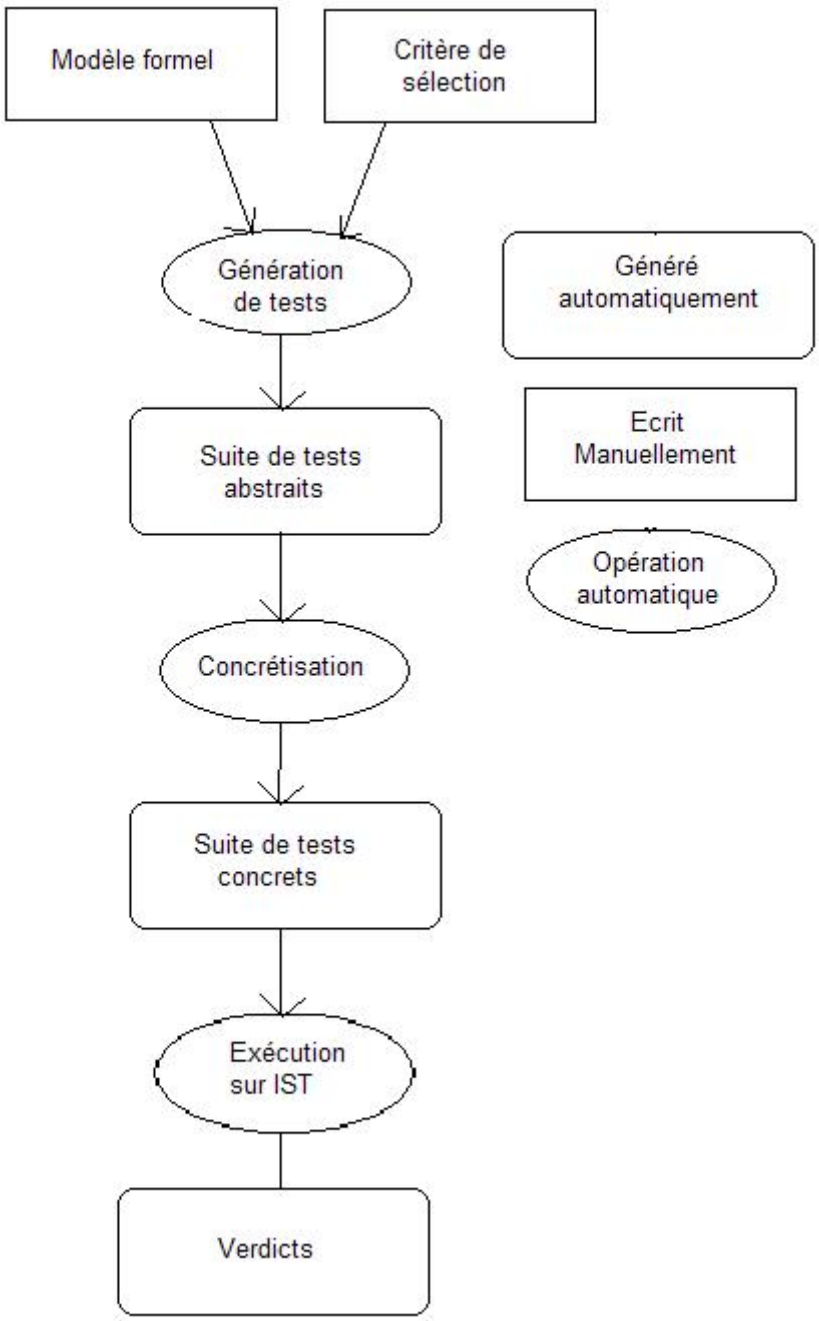

FIG. 1.1 – Démarche générale du Model-Based Testing

l'implémentation est incorrecte, soit l'implémentation est bonne mais le modèle est imparfait, bien que vérifiant le cahier des charges. Une analyse de tous les verdicts de la suite permet souvent de déterminer un peu plus la source d'erreur, et il est toujours possible de refaire une suite de tests, plus précise pour avoir des compléments d'information.

1.1.2 Calculer des tests à partir d'un modèle

Critères de sélection Les générateurs de tests doivent sélectionner des tests parmi un ensemble de tests possibles, et nécessitent donc un guide pour le faire : ce sont les critères de sélection. Les critères de sélection sont le principal moyen à l'ingénieur d'exprimer son expérience du système, et il peut influer sur l'efficacité de la suite de tests en choisissant le bon critère de sélection. Les critères de sélection ne portent que sur le modèle et les spécifications du cahier des charges, et les critères de couverture, souvent utilisés pour évaluer comment une suite de tests couvre le modèle, ne s'appuie en aucun cas sur la couverture du code de l'IST ou sur le code source lui-même.

L'évaluation de la couverture du modèle permet de mesurer l'adéquation de la suite de tests par rapport au modèle. On peut mesurer la couverture des états ou des transitions, et générer des tests correspondant plus spécifiquement aux parties non testées.

Il existe plusieurs familles de critères de sélection, que nous allons présenter brièvement et de manière non exhaustive.

Critère de couverture de modèle structurel Il est le plus répandu des critères de sélection, car c'est une adaptation du critère de couverture basé sur le code source, et donc bien étudié et approfondi. Il inclut des familles de critères tels que critère de couverture orienté flot de contrôle, orienté flot de données ou basé sur les transitions. Nous ne présenterons pas ici tous les critères, décrits et hiérarchisés dans [Mye79]. Nous citerons par exemple la couverture d'états, des décisions, des chemins, des conditions (et de ses dérivées), des lectures/écritures des variables, des configurations, des chemins linéaires, des chemins ayant au moins une boucle etc.

Critère de couverture des données Il concerne la couverture de l'espace des données d'entrée d'une opération ou d'une opération dans le modèle. Il s'agit de guider la sélection en fonction des valeurs affectées à une variable. Cela va de "une valeur" à "toutes les valeurs" qui génèrera une suite dans laquelle la variable aura au moins une fois toutes les valeurs possibles de son domaine. Entre ces deux extrêmes, il y a "tester les valeurs limites du domaine". Pour les prédicats ayant plusieurs variables, il existe encore de nombreux critères permettant de définir quelles limites de quelle variable on veut. Il existe des critères de *couverture statistique des données*, assurant que les valeurs affectées à une variable représente une distribution statistique (uniforme, normale ou de Gauss, de Poisson). Nous citerons un dernier critère de cette catégorie : test par paire. Il consiste à ne pas tester la combinaison de toutes les valeurs de toutes les variables, mais seulement de s'assurer que les combinaisons des couples de deux variables sont testées. Il est possible de tester les combinaisons de N variables, jusqu'au critère toutes les combinaisons, qui revient à choisir le critère toutes les valeurs sur tous les paramètres.

Critère modèle de faute Les tests générés par ce critère sont efficaces dans la détection de certains types de fautes dans le modèle. Si on suppose que l'IST a les mêmes types de défauts, la suite de tests peut détecter ces défauts. En Model-Based Testing, il s'agit de faire coexister ce principe de modèle de faute avec celui de mutation. Concrètement, on réalise des mutations du modèle de comportement (par des opérateurs de mutations bien définis) et les tests de séquences ou de données en entrée distinguant tous les mutants du modèle original sont conservés.

Critère basé sur les spécifications L'objectif est de tester toutes les spécifications informelles. Pour y parvenir, il y a deux solutions : les spécifications sont enregistrées à l'intérieur même du modèle, dans diverses parties du modèle par des annotations ; ou en formalisant les spécifications et en les utilisant en critère de sélection pour le générateur de tests (ce qui s'apparente à la spécification de cas de tests explicite). Ce système permet d'assurer faciliment une traçabilité en établissant une relation entre modèle et spécification.

Spécifications de cas de test explicite Cela permet à l'ingénieur de définir explicitement qu'un test, ou une suite de tests doit être généré à partir du modèle. Cela consiste à écrire un cas de test de manière formelle (système de transitions, diagramme de séquence, expressions régulières, etc.) et de générer les tests à partir de cette spécification formelle. Cela permet de contrôler de manière plus précise la génération de test, de la guider vers une situation estimée critique, mais demande un travail non négligeable : écrire une telle spécification peut être difficile, et de toute manière, prend plus de temps que de choisir un critère de couverture structurel.

Méthode de génération de tests statistique La génération aléatoire permet d'explorer une grande partie des comportements d'un système. Cette méthode est plutôt utilisée sur des modèles basés sur l'environnement du système, car il recquiert de pouvoir déterminer les probabilités de telles utilisations de l'IST. On choisit aléatoirement la prochaine transition du test, chacune des transitions étant pondérées par leur probabilité d'être utilisée. Ainsi, le cas de tests ayant le plus de chances d'arriver est celui qui a le plus de chances d'être généré le premier. Le modèle utilisé est un modèle d'utilisation et plus un modèle de comportement, et ce modèle ne peut prévoir la réponse attendue, donc ne peut plus servir à générer d'oracles.

Calcul de tests par animation L'animation consiste à simuler l'excécution sur le modèle, en choisissant manuellement les opérations et leurs paramètres à activer. L'animation symbolique permet de ne pas devoir choisir des valeurs pour tous les paramètres. L'animation sert à valider un modèle [TM01], puis a été utilisée pour générer les tests animés, pour permettre de valider aussi le programmne.

Nous expliquerons l'animation symbolique par la présentation de BZ-Testing-Tools [LPU02]. L'outil prend en entrée un modèle formel B dont on peut vérifier la conformité avec les spécifications en animant le modèle. Mais on peut aussi vérifier la conformité du système en l'utilisant pour générer des tests, toujours par animation.

L'animation d'un modèle est basée sur la décomposition des opérations en comportements, représentés par des prédicats avant/après. Un comportement peut être vu comme

un graphe de flot de contrôle, extrait de la substitution généralisée d'une opération B. On parle d'animation symbolique, car on peut laisser des paramètres d'opération non valués, et donc ils sont remplacés par des valeurs symboliques.

Chaque comportement contient un prédicat devant être vrai pour que le comportement puisse être activé. C'est ce qu'on appelle *cible de test*. BZ-TT va alors calculer automatiquement une séquence d'appels d'opération permettant d'atteindre un état vérifiant la cible de test. Cette séquence est appelée *préambule*. BZ-TT active le comportement en activant l'opération avec les paramètres appropriés. La construction du test se poursuit par l'*identification*, c'est-à-dire des appels d'opérations spécifiques permettant d'observer l'état du système (et donc observer le résultat du test). Le *postambule* permet si on le souhaite, de remettre le système à l'état initial.

Le préambule et le postambule sont calculé par une exploration de l'espace des états symboliques, en utilisant un algorithme best-first [CLP04] trouvant le chemin le plus court pour atteindre l'état désiré. Cet algorithme utilise lui aussi l'animation symbolique. BZ-TT utilise un solveur de contrainte pour valuer les paramètres symboliques pour pouvoir produire des test exécutables, c'est-à-dire complètement paramétrés.

Le Model-Based Testing permet donc de générer une grande variété de suites de tests, grâce à la diversité des outils et des critères de sélection. La démarche du Model-Based Testing permet d'automatiser plusieurs tâches laborieuses : écriture des nombreux tests, écriture des oracles et comparaison des résultats. Mais les critères de sélection ne permettent pas de pouvoir caractériser suffisament un jeu de test, car les générateurs de tests sont soumis au choix entre la précision d'une sélection et la redondance de tests qui provoquerait une explosion du nombre de tests sans augmenter l'efficacité de la suite. Les critères de sélection ne peuvent que donner les grandes lignes générales de décision. Nous verrons dans la prochaine section comment il est possible d'améliorer l'efficacité de cette démarche, en focalisant la sélection des tests sur certains points particuliers, notamment par un choix plus précis du critère de sélection.

1.2 Propriété, besoin de tests et scénarios

Notion de propriété Lors de la rédaction du modèle formel à partir des spécifications listées dans le cahier des charges, il est possible d'extraire de ce cahier un ensemble de propriétés que doit respecter le système. Les propriétés portant sur le comportement sont de différentes natures : vivacité, sûreté, atteignabilité, invariant ...

Ces propriétés peuvent être vérifiées formellement sur un modèle, par un prouveur, interactif comme COQ [OG02] ou automatique, ou par model-checking explicite (citons pour exemple SPIN [Hol91] ou ProB [LB03]) ou symbolique (citons les travaux de K. MacMillan [McM92] et B. Parreaux [Par00]). On peut donc s'assurer que le modèle de comportement vérifie une propriété, mais une implémentation est souvent d'une taille trop importante, et n'est naturellement pas écrit en notation formelle. Il est donc impossible de vérifier ces propriétés de la même manière sur l'implémentation, et la conformité entre le modèle et l'implémentation vis-à-vis de la propriété est encore à valider.

Génération de tests à partir de propriété En utilisant les critères de sélection d'un générateur de tests usuels, à savoir les critères de couverture, les suites de tests ne peuvent tester correctement une propriété en particulier, car la sélection ne peut se focaliser sur telle partie du modèle, ou sur telles situations correspondant à la propriété. En utilisant le critère de couverture des spécifications, en faisant le choix d'annoter le modèle, on s'assure seulement de n'avoir pas oublié de spécifications, et donc qu'aucune n'est pas respectée, mais cela ne permet pas de tester de manière approfondie une spécification. De plus, une propriété peut ne pas être explicitement écrite dans le cahier des charges, et vouloir tester une nouvelle propriété signifie devoir réécrire, ou du moins réannoter le modèle. En faisant le choix de formaliser la spécification, puis de l'utiliser en critère de sélection, on se rapproche du critère spécification du cas de test explicite. Si au lieu de formaliser une spécification, on formalise une propriété, on pourra l'utiliser comme critère de sélection pour un générateur de tests, et donc obtenir une suite de tests destinée à tester la propriété.

Notion de besoin de test Les suites de tests générées à partir d'une propriété et du modèle ne peuvent tester de manière exhaustive toutes les situations où la propriété peut être mise en difficulté. Le générateur de tests est obligé de faire des choix dans la sélection, et se prive d'un grand nombre de situations *a priori* testées par une situation similaire qui, elle, est testée. En d'autres termes, le générateur considère des situations comme trop proches pour être toutes testées, et considère qu'en en testant une seule, les autres sont testées. Or un ingénieur peut estimer que ces situations ne sont pas identiques, et qu'un générateur ne sélectionne pas des situations que son expérience du système lui recommande de tester tout de même. C'est ce que nous appelons un besoin de test, c'est-à-dire un test nécessaire pour tester une propriété dans une situation précise, potentiellement dangereuse.

Notion de scénario Cette notion de besoin de test se rapproche de ce qui est appelé spécification de cas de test explicite dans les critères de sélection. Pour tester une propriété dans une situation particulière, il faut nécessairement se placer dans cette situation, c'est-à-dire parvenir à générer des enchaînements d'appels d'opérations nous permettant de créer une situation où on veut tester la propriété. Il s'agit principalement de restreindre l'ensemble des tests possibles en imposant à certaines positions certaines opérations, c'est-à-dire d'indiquer quand il le faut le chemin à prendre pour le générateur de tests. Ces indications sont fournies de manière formelle, par exemple sous forme d'un système de transitions, diagramme de séquence, expressions régulières etc. Nous appelons scénario cette séquence d'appels d'opérations permettant de tester une partie précise d'une propriété répondant à un besoin de test.

Ce scénario peut servir de critère de sélection dans certains outils comme TOBIAS [LDdB+07]. L'outil cherche des séquences respectant le scénario, en utilisant un critère de couverture des opérations par exemple.

Le Model-Based Testing et les outils développés jusqu'à aujourd'hui permettent d'automatiser une grande partie du processus de validation de système, en particulier en générant avec peu d'effort beaucoup de tests. Cela permet aussi de guider la génération de tests, en choisissant un critère de sélection adéquat. Il est possible aussi de valider

une propriété sur un système en utilisant cette propriété comme critère de sélection pour générer une suite de tests focalisée sur cette propriété. Et si l'ingénieur remarque que certaines situations potentiellement dangereuses ne sont pas testées pour une propriété, ou s'il souhaite tester d'avantage ces situations, il peut utiliser la notion de besoin de tests et écrire un scénario répondant à ce besoin de test. En fournissant le scénario comme critère de sélection au générateur de tests, il a à disposition une suite de tests pouvant éprouver la propriété dans des conditions du besoin de test. Mais si l'ingénieur n'a plus à produire manuellement des tests, il reste à l'ingénieur l'écriture des scénarios, qui peuvent être complexes à maîtriser.

L'objectif de ce mémoire est donc de présenter comment il est possible d'aider l'ingénieur à passer cette étape.

1.3 Problématique

Comme nous venons de le voir, il est intéressant de pouvoir utiliser un scénario comme critère de sélection pour la génération de tests, et il existe déjà des outils basés sur la technique du Model-Based Testing permettant d'avoir en entrée des scénarios, qui décrivent de manière abstraite une partie plus ou moins importante des tests à générer, et permettent donc de construire un jeu de test plaçant le système sous test dans une situation précise lors de son exécution.

D'un autre côté, les ingénieurs peuvent avoir des besoins de test particuliers pour une propriété, qui sont en plus dépendants non seulement de la propriété mais aussi du système à tester. Comme ce sont des besoins qui ne sont généralement pas couverts par les méthodes traditionnelles, ils peuvent être complexes, et leur écriture peut être rapidement fastidieuse. Il existe déjà bien sûr la possibilité de générer des tests à partir de scénarios, mais il reste toujours à l'ingénieur de créer les scénarios manuellement, dans les limites des possibilités imposées par le format utilisé entrée par l'outil utilisé.

Ces besoins de test se font surtout sentir dans le domaine de la sécurité. Les failles de sécurité des systèmes proviennent le plus souvent soit de vulnérabilités introduites par la complexité des systèmes qui rend leur développement justement trop complexe pour être entièrement certifié, soit de non-respect des politiques de sécurité dans la configuration des systèmes. Donc parallèlement aux campagnes de test de validation fonctionnelle d'un système, c'est une véritable campagne de validation du respect des politiques de sécurité qui devrait être réalisée en s'appuyant sur des techniques de modélisation formelle et de génération automatique de tests, accompagnée d'une traçabilité des exigences de sécurité de haut-niveau vers les tests mis en oeuvre. C'est dans ce cadre qu'est réalisé le stage, puisque ceci était l'objectif du projet POSE, présenté en 2.1.1 auquel a participé le LIFC et dans lequel pourrait s'inscrire les travaux de ce stage.

Partant de ce constat, il nous a semblé intéressant d'étudier la possibilité de créer une démarche, puis un outil, permettant de générer un scénario, ou schéma de test, à partir d'une propriété de sûreté et d'un besoin de test. L'ingénieur n'aurait alors plus qu'à formuler le besoin de test et la propriété à laquelle il est lié, et il aurait le ou les scénarios qui correspondent.

La motivation de réaliser une telle démarche est aussi de générer automatiquement le schéma de test, de manière la plus générique possible pour être le plus exhaustif pos-

sible, c'est-à-dire avoir une suite de tests la plus complète possible, tout en permettant à l'ingénieur de sélectionner par un mécanisme, par exemple un filtre, les tests les plus intéressants. La plupart des générateurs de tests à partir de scénario construisent les tests avec pour seul objectif de se placer dans la situation concernée par la propriété, mais la finalité de cet outil dans le cadre de ce stage est surtout de pouvoir tester non seulement cette situation, mais aussi ce qu'on appelle le préambule, c'est-à-dire la séquence d'opérations permettant de se mettre dans cette situation. Or par soucis d'efficacité, les générateurs de tests à partir d'un scénario n'assure aucune couverture lors de la recherche des états cibles et empêchant par ce fait de tester le préambule en même temps que le scénario. Or c'est un des objectifs de la démarche. Par exemple, cela permet de s'assurer que appeler plusieurs fois la même opération, successivement, ou alternativement avec une autre, n'influe pas sur le verdict du test.

La démarche permet donc de générer tous les tests correspondant à la propriété, mais pour diminuer le nombre de tests s'il est trop important, un mécanisme tel que le filtrage permettrait de sélectionner plus finement les tests, sans avoir à réécrire la propriété qui deviendrait plus complexe, et de moins en moins compréhensible telle quelle. Il serait désagréable et peu productif de devoir recommencer la dernière suite de test réalisée car la suite est trop grande, alors que l'ingénieur validation sait comment la réduire, par exemple en excluant les tests ayant telle opération apparaissant plus d'une fois successivement. Plutôt que de le contraindre à réécrire le schéma de test pour le faire, le mécanisme de filtre appliqué sur la suite permettrait d'éliminer tous les tests indésirables. L'ingénieur peut, par expérience, savoir que telle opération ne peut avoir d'influence, et donc il peut filtrer tous les cas ou elle apparaît souvent, ou à l'inverse, il a un doute sur l'influence de quelques opérations, et donc il filtre les tests de manière à se concentrer sur ceux qui ont une occurence d'une opération, ou d'au moins deux des opérations de la liste, etc. Et en cumulant les filtres, il peut contrôler la taille de la suite, tout en contrôlant son efficacité : c'est son expérience qui lui permet d'avoir ce qu'il veut. L'objectif de la démarche n'est donc pas dans un premier temps de prendre en compte le nombre de tests, la faisabilité de l'exécution de la suite ou autre. Elle doit fournir un schéma de test qui puisse générer toutes les séquences d'opération concernées par la propriété et le besoin de test, et ensuite fournir aussi un moyen de sélectionner tous les tests vérifiant une caractéristique selon l'expérience de l'ingénieur, ses connaissances du modèle et du système.

Pour cela il sera nécessaire dans un premier temps de formaliser les données et les résultats, car l'objectif étant que les besoins de test soient le plus facilement compréhensible, et idéalement décrits par des phrases pour que l'ingénieur puisse aisément les écrire. Il s'agit aussi de définir un formalisme pour les propriétés, le plus générique possible, et un format de sortie pour les schémas de test, bien que pour celui-ci, on soit justement limité par la volonté d'être compatible avec les outils utilisant les résultats.

Dans le cadre du stage, il s'agit seulement de trouver une démarche permettant de générer les schémas de tests pour une propriété de sûreté, et un besoin de test fixé. Mais surtout, l'outil developpé à partir de cette démarche serait utilisé dans le cadre de l'approche dévelopée lors du projet POSE, auquel le LIFC de l'université de Franche-Comté a participé. Les choix faits pendant le stage doivent donc prendre en compte de préférence les outils utilisés dans cette approche.

Comme à long terme, l'outil qui serait issu des travaux réalisés durant le stage devra pouvoir être utilisé dans des domaines d'application variés, il serait utile qu'il soit facile-

ment adaptable, notamment par l'utilisateur. Idéalement, il devrait être possible de gérer des propriétés autres que celles pour lesquelles il a été réalisé dans le cadre du stage. Et de la même manière, il pourrait permettre à l'utilisateur de créer ses propres besoins de test, et de définir lui-même les schémas de test correspondant.

La motivation de ces travaux est apparue lors de la réalisation du projet POSE, et tous les travaux ont été réalisés dans l'optique de pouvoir l'intégrer dans cette méthodologie ainsi qu'assurer la compatibilité des outils utilisés dans celle-ci. Le prochain chapitre exposera donc le contexte scientifique en commençant par la présentation du projet POSE et des travaux menés au sein du LIFC ayant servi pour la réalisation de ce stage, puis les autres travaux ayant eux aussi été utiles pour ce travail.

Chapitre 2

Contexte Scientifique

Ce chapitre présente les travaux qui ont été utilisés lors des travaux de ce mémoire. Nous verrons dans un premier temps ceux réalisés au sein du laboratoire LIFC dans lequel le stage s'est déroulé, puis les autres, à savoir ceux de Dwyer sur le formalisme des propriétés. Nous conclurons ce chapitre par une synthèse des différents travaux et les directions de travail qui en découlent.

2.1 Travaux menés au sein du LIFC

2.1.1 Le projet POSE

Le projet POSE est un projet réalisé au sein du LIFC de l'université de Franche-Comté en collaboration avec des partenaires industriels et académiques.

Objectif du projet POSE L'objectif du projet POSE était l'automatisation de la génération et l'exécution de tests permettant la validation de conformité d'un système à la politique de sécurité qui lui est assignée.

Il s'agit de s'assurer que les différents niveaux de propriétés de sécurité (par exemple confidentialité, atomicité, authentification, intégrité, standardisation des exceptions, cycle de vie, ...) font l'objet chacune de tests spécifiques, correspondant à la simulation d'attaques possibles et au test des réponses du système, alors que les techniques de générations de tests fonctionnels actuelles restent plus générales. Cette démarche ne remplace pas la génération de tests fonctionnels, mais complète celle-ci par des tests de sécurité [JMT08].

Il s'agit d'adapter ces techniques et de prendre en compte les besoins effectifs issus du terrain (c'est-à-dire les défis de la validation des politiques de sécurité sur les applications carte à micro-processeur ou sur terminaux développées et livrées aujourd'hui) pour mettre au point un démonstrateur d'environnement dédié au test de conformité des politiques de sécurité d'un système.

Partenaires académiques et industriels Le projet POSÉ est un projet qui s'appuie sur un ensemble de savoir-faire et de connaissances scientifiques des partenaires : d'une part en génération automatique de tests fonctionnels à partir d'un modèle formel et d'autre part en formalisation et vérification de propriétés de sécurité. Les partenaires industriels ont notamment permis d'avoir des exemples concrets des besoins de tests, sur

des systèmes de taille industriels et d'avoir des résultats permettant d'évaluer l'efficacité de la méthodologie. Voici une brève présentation des partenaires du LIFC.

Voici les trois partenaires industriels :

Leirios Technologies, éditeur de solutions de génération automatique de tests, en charge du pilotage du projet validation et des ingénieurs sécurité)

Gemalto, leader mondial dans le domaine des cartes à micro-processeur, qui amène ses besoins, ses compétences et un contexte d'expérimentation en situation réelle. Gemalto a apporté une expertise importante dans le processus actuel du test des propriétés de sécurité.

Silicomp-AQL, société de conseil, possédant une forte expertise en sécurité, intégrant un laboratoire d'évaluation de la sécurité des systèmes d'information (CESTI) et possédant des compétences en modélisation, méthodes formelles et génération automatique de tests.

Et pour finir, les deux partenaires académiques :

LIG (Grenoble) qui développe une forte activité de recherche dans le domaine des méthodes formelles, du test et de la sécurité des systèmes informatiques

INRIA/Projet CASSIS qui développe une activité de recherche dans le domaine des techniques symboliques pour la vérification et le test de systèmes infinis, paramétrés et de grande taille

Démarche de l'approche POSE La démarche expérimentée, présentée par la figure 2.1 consiste à utiliser une propriété comme un critère de sélection des tests à extraire du modèle. Plus précisément, les objectifs de test sont décrits au moyen de séquences d'appels d'opération qui mettent à l'épreuve une propriété de sécurité de manière à répondre à un certain besoin de test, associé à la propriété. Ces besoins de test ont été au départ exprimés par des ingénieurs sécurité : leur savoir-faire leur permet d'imaginer des scénarios de tests particuliers pour tester de manière approfondie un ensemble de propriétés de sécurité. Un besoin de test est ajouté à une propriété de sécurité en la transformant syntaxiquement, de manière à décrire un scénario exerçant la propriété dans les conditions décrites par le besoin de test. Ceci correspond à une spécialisation de la propriété au contexte décrit par le besoin de test. Le besoin de test est combiné formellement et par l'ingénieur à une propriété de sécurité et d'un besoin de test pour former un schéma de test. Ces schémas de tests sont écrits dans un langage défini dans le cadre du projet POSE, décrit en 2.1.2 et permettant d'exprimer de tels schémas de test.

Les propriétés de sécurité à tester sont extraites de la cible de sécurité, qui est une représentation semi-formelle des exigences de sécurité devant être mise en oeuvre pour le système. Les schémas de test sont utilisés comme des guides par l'outil de génération de tests LTG de LEIRIOS, qui calcule par animation symbolique à partir du modèle de sécurité des tests abstraits correspondant aux scénarios décrits par le schéma. Dans ce sens, l'objectif de test formalisé par le schéma constitue le critère de sélection des tests calculés. On continue ensuite la méthodologie classique du Model-Based Testing : les tests abstraits sont finalement concrétisés, puis on exécute les tests sur le système sous test.

Résultats du projet POSE Le projet POSÉ a permis la mis en place d'une approche méthodologique pour la production de tests de sécurité afin de valider la conformité d'un

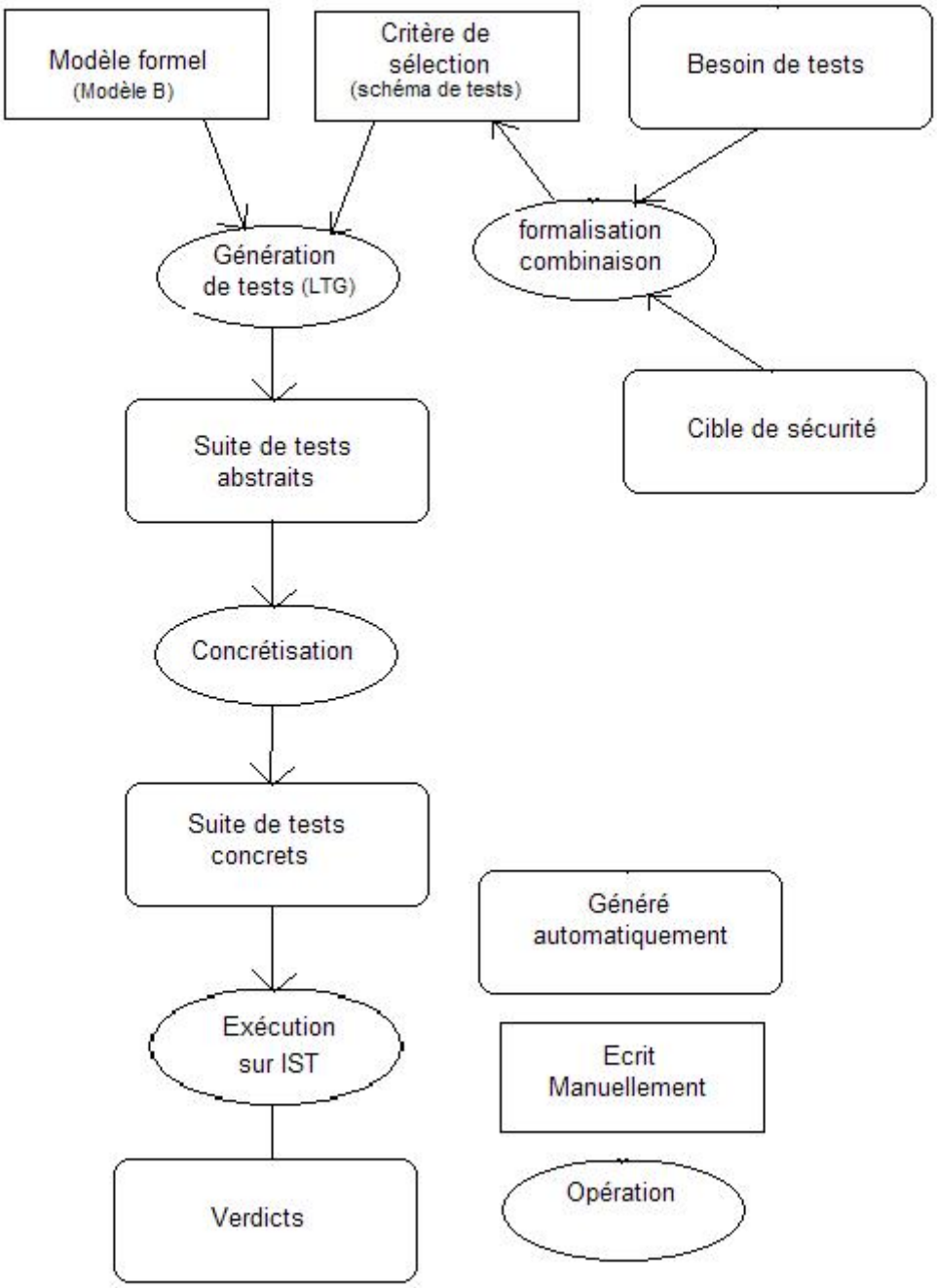

FIG. 2.1 – Démarche générale de l'approche POSE

système aux politiques de sécurité qui lui sont assignées. Le projet a mis en évidence par l'expériementation que l'approche est faisable au niveau industriel.

Dans le cadre de ce projet, il a été montré qu'il est possible de représenter les besoins de test par des règles de transformation syntaxique : un besoin de test est « injecté » dans une propriété de sécurité en la transformant syntaxiquement de manière à décrire un scénario exerçant la propriété dans les conditions décrites par le besoin de test.

Expérimentation de l'approche L'approche a été expérimentée sur une application réelle de taille industrielle : la plate-forme IAS (Identification, Authentification et Signature électronique). IAS est une norme, spécifiée par GIXEL [GIX04], pour les cartes à puces développées comme plate-forme pour l'e-administration en France, et fournit les services d'identification, authentification et signature aux autres applications de la carte à puces. Des tests fonctionnels ont d'abord été générés à partir du modèle fonctionnel avec LTG, puis trois schémas de test ont été écrit pour trois besoins de tests. Les résultats de l'expérience sont encourageants [JMT08]. Les tests générés à partir des trois schémas ont permis de révèler certaines différences connues d'interprétation des spécifications entre les developpeurs et les concepteurs du modèle. De plus, les tests ainsi générés ne sont pas redondants avec les tests fonctionnels.

2.1.2 Le langage de description de schémas de tests

Dans le cadre du projet POSE, il a été nécessaire d'écrire un langage, présenté dans cette section, permettant de réprésenter de manière formelle les scénarios, qui sont des séquences d'appels d'opération.

Il a donc été conçu dans le but de pouvoir exprimer un certains nombre de caractéristiques que peuvent nécessiter les scénarios imaginés à partir des besoins de tests à disposition. Cette analyse des scénarios a conduit à la conclusion suivante : le langage doit pouvoir spécifier les appels d'opération, mais aussi des états à atteindre. En effet, il n'est pas nécessaire parfois de savoir quelle opération on doit activer, mais seulement s'assurer qu'on arrive dans un état vérifiant une propriété particulière, donc ne spécifier uniquement une caractéristique de l'état à atteindre, et non le moyen d'y parvenir.

Il est construit sur la base d'expressions régulières, avec les notions de choix ou d'itérations, ce qui permet à un schéma de représenter de manière symbolique tout un ensemble de scénarios. Ce langage a été voulu comme le plus générique possible. Il est structuré en trois couches distinctes : la *couche modèle*, la *couche séquence* et la *couche directive* de génération de tests.

Syntaxe de la couche modèle Elle permet de décrire les éléments de l'interface du modèle formel, c'est-à-dire, d'une part les appels d'opérations (règle OP), et d'autre part les conditions d'accès (règle SP pour State Propositions) comme des propriétés sur les variables d'état du modèle. Les opérations, comme on peut le voir sur le figure 2.2 sont de trois types :

- un nom d'une opération qui représente un appel d'une opération pour des valeurs quelconques de ses paramètres,
- un appel d'une opération quelconque notée $OP,

$$
\begin{array}{lcl}
OP & ::= & \underline{operation\_name} \\
 & | & \underline{\$OP} \\
 & | & \underline{\$OP\text{"}\backslash\{\text{"}OPLIST\text{"}\}\text{"}}
\end{array}
$$

$$
\begin{array}{lcl}
OPLIST & ::= & \underline{operation\_name} \\
 & | & \underline{operation\_name}\text{","}OPLIST
\end{array}
$$

$$
\begin{array}{lcl}
SP & ::= & \underline{state\_predicate}
\end{array}
$$

FIG. 2.2 – Règles de la couche modèle

$$
\begin{array}{lcl}
CHOICE & ::= & \text{"}|\text{"} \\
 & | & \text{"}\otimes\text{"} \\
 & & \\
OP1 & ::= & OP \\
 & | & \text{"}[\text{"}OP\text{"}]\text{"} \\
 & | & \text{"}[\text{"}\underline{\$OP}\text{"}/w\text{"}CPTLIST\text{"}]\text{"} \\
 & | & \text{"}[\text{"}OP\text{"}/w\,\{\text{"}CPTLIST\text{"}\}]\text{"} \\
 & | & \text{"}[\text{"}OP\text{"}/e\,\{\text{"}CPTLIST\text{"}\}]\text{"} \\
CPTLIST & ::= & \underline{cpt\_label} \\
 & | & \underline{CPTLIST}\text{","}\underline{cpt\_label}
\end{array}
$$

FIG. 2.3 – Règles de la couche directive

– ou un appel d'opération quelconque à l'exception de quelques opérations énumérées dans une liste appelée OP_LIST.

Syntaxe de la couche directive de la génération de tests La couche directive de la génération de tests décrite par la figure 2.3 permet de spécifier les lignes directrices des étapes de générations de tests. Cette couche permet en effet d'introduire des instructions de pilotage pour la génération de tests. Ces instructions permettent de modifier le type de couverture appliquée pour certaines sous-parties d'un schéma de test (couverture de chemin pour le niveau séquence et couverture de comportement pour les opérations du niveau modèle). Elle met à disposition deux sortes de directives permettant de réduire la recherche pour l'instanciation des schémas de test. La règle CHOICE introduit deux opérateurs, noté | et $\otimes$ pour couvrir les branches d'un choix. Soient deux schémas de tests S_1 et S_2. $S_1 \mid S_2$ spécifie que le générateur de tests doit générer des tests pour chacun des deux schémas. $S_1 \otimes S_2$ spécifie que le générateur de tests doit générer des tests pour l'un ou l'autre.

La règle OP1 dicte au générateur de tests de couvrir un des comportements de l'opération OP (option par défaut). Il est possible de couvrir l'ensemble des comportements de l'opération en l'entourant par des crochets. La règle permet aussi de choisir la couverture des opérations selon le type de comportements : "OK", "KO" par exemple. Dans ce mémoire, l'écriture OP /w $\{ok\}$ est remplacée par OP_{OK} pour faciliter la lecture.

$$
\begin{array}{lll}
SEQ & ::= & OP1 \mid \text{"("}SEQ\text{")"} \mid SEQ\text{"} \rightsquigarrow \text{("}SP\text{")"} \\
 & \mid & SEQ \text{"."} SEQ \\
 & \mid & SEQ\ REPEAT \\
 & \mid & SEQ\ CHOICE\ SEQ \\
 & & \\
REPEAT & ::= & \text{"}*\text{"} \mid \text{"}+\text{"} \mid \text{"?"} \\
 & \mid & \text{"\{"}\underline{n}\text{"\}"} \mid \text{"\{"}\underline{n}\text{",\}"} \\
 & \mid & \text{"\{,"}\underline{n}\text{"\}"} \mid \text{"\{"}\underline{n}\text{","}\underline{m}\text{"\}"}
\end{array}
$$

FIG. 2.4 – Règles de la couche séquence

Syntaxe de la couche séquence La figure 2.4 présente les règles de la couche séquence. Le couche séquence permet de modéliser les enchainement d'opérations et d'états symboliques sous la forme d'expressions régulières. La règle SEQ décrit une séquence d'appels d'opérations comme une expression régulière. Un pas de séquence est soit un appel d'opération, noté OP1, soit un appel d'opération menant à un état satisfaisant une propriété d'état, noté SEQ $\rightsquigarrow$ (SP). Cette dernière règle représente l'amélioration majeure par rapport aux langages usuels de description de scénarios, car il permet de définir la cible d'une séquence d'opération, sans nécessairement énumérer les opérations composant cette séquence.

Les séquences peuvent être composées par la concaténation de deux séquences, la répétition d'une séquence ou le choix entre deux séquences. Nous utilisons les opérateurs usuels des expressions régulières, auxquels on ajoute les opérateurs de répétitions bornés (exactement n fois, au moins n fois, entre n et m fois).

Utilisation des schémas Les schémas de tests pourront être dépliés et animés par BZ-TT **??** pour instancier les tests, et on aura alors une suite de tests concrétisés que l'on pourra exécuter sur l'Implémentation Sous Test.

2.1.3 Les stratégies associées à une propriété JML

Les langages annotés sont une approche intéressante pour la vérification et la validation d'un système. Il permet de décrire par des annontations le comportement normal d'une classe. De plus, il a le même niveau d'abstraction que celui du langage qu'il annote. Le Java Modelling Language (JML) permet d'annoter un programme en Java. C'est de plus un langage très bien outillé et qui a fait ses preuves au niveau industriel.

Et dans [BDJG06] est présentée une approche utilisant un Model Based Testing automatique pour la validation de propriété de sûreté définie par l'utilisateur sur une application Java Card. Les propriétés sont écrites en JTPL [TH02]. Il est proposé d'utiliser deux outils JAG-Tool [GG06] et JML-TT [FBL06] dont la combinaison permet de générer des cas de tests complémentaires des tests fonctionnels et pertinents par rapport à la propriété JTPL. Le principe général est celui-ci :

- analyse de la propriété JTPL et génération de la stratégie de génération de séquence de tests ;
- à partir de l'interface JML et de la stratégie, JML-TT génère une suite de tests

pertinents par rapport à la propriété et couvrant les comportements fonctionnels de l'application ;
- la suite de tests est exécutée sur l'implémentation annotée enrichie par les annotations spécifiques à la propriété ;
- l'exécution du programme compilé par JML Runtime Assertion Checking (RAC) permet de vérifier à la volée toutes les assertions JML.

RAC est un compilateur qui enrichit le code du programme Java pour la vérification des différentes clauses JML. Lors de l'exécution du programme, si une assertion JML est violée, une exception spécifique est levée, et grâce à la traçabilité, il est possible de trouver quelle propriété temporelle a été violée et quelle politique de sécurité a été mal implémentée.

Les stratégies JML-TT Comme nous venons de le voir, la génération de la suite de tests est faite grâce à des stratégies. Les stratégies découlent d'une analyse syntaxique de la propriété JTPL. Une fonction convertit syntaxiquement en stratégies JML-TT qui sont des séquences de primitives, et à chaque primitive, JML-TT associe un algorithme spécifique permettant donc de tester spécifiquement une propriété.

2.2 Les schémas de propriété de Dwyer

Dans l'article [DAC99], les auteurs Dwyer, Avrunin et Corbett ont voulu faciliter l'écriture des propriétés en définissant des modèles types auxquels ils associent une formule LTL , CTL [CES86] ou une expression Query Regular Expression (QRE) [OO90]. Avec cette approche, il suffit de réussir à traduire la propriété textuelle dans un des modèles pour avoir la traduction en formule. Ce système de spécification de propriété a pour objectif de servir pour les outils de vérification de modèles formels à états finis, ce qui est aussi notre cadre de travail.

Le système proposé est un ensemble de patrons organisé en une ou plusieurs hiérarchies, avec des connexions pour faciliter l'exploration. Ils ont donc défini un système de patrons, paramétrables, de haut niveau, indépendant de tout formalisme de spécification des propriétés (basé sur les états ou sur les événements). Ils ont prévu aussi de nombreuses astuces permettant de nuancer les patrons de base permettant ainsi de couvrir davantage de propriétés. Similairement, ils ont adopté une méthode particulière : ils ont défini leur système, puis récolté plus de 500 propriétés utilisées dans de nombreux projets très variés. 92% de ces propriétés ont pu être instanciées grâce à ce système, et pour s'occupper des 8% restants, ils ont complèté leur système.

Les patrons de spécification de propriété Un patron de spécification de propriété est une description généralisée des exigences les plus communément utilisées sur les séquences état/événement acceptées dans tel modèle d'état fini d'un système. Il décrit la structure des aspects des comportements d'un système et fournit les expressions de ce comportement dans les formalismes communs.

Il y a huit types de patrons :
- **Absence** : un état/événement donné n'arrive pas dans la portée ;
- **Existence** : un état/événement donné doit arriver au moins une fois dans la portée ;

- **Bounded Existence** : un état/événement donné doit arriver un nombre de fois limite dans la portée ;
- **Universality** : un état/événement donné arrive dans toute la portée ;
- **Precedence** : un état/événement donné doit toujours être précédé par un autre état/événement donné dans la portée ;
- **Response** : un état/événement donné doit être suivi par un autre état/événement donné dans la portée ;
- **Chain Precedence** : une séquence d'états/événements donnée doit toujours être précédée par une séquence d'états/évenements donnée ;
- **Chain Response** : une séquence d'états/événements donnée doit toujours être suivie par une séquence d'états/événements donnée.

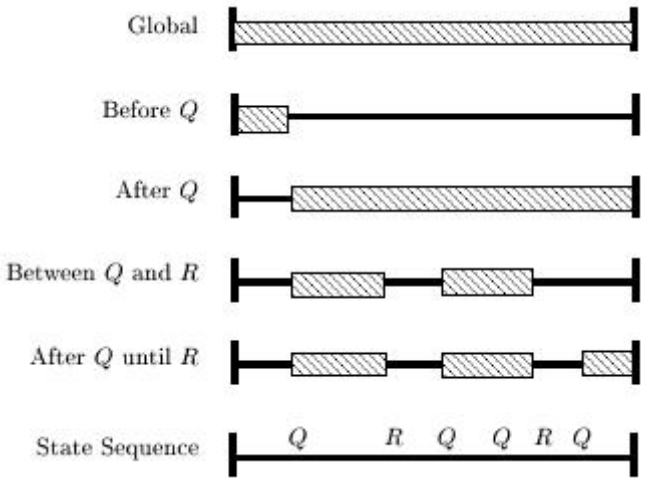

FIG. 2.5 – Portées des patrons

Chaque patron contient une portée (*scope*) définissant la partie de séquence d'éxécution sur laquelle le modèle doit être maintenu (cf 2.5).

Il y a cinq types de portée décrite par la figure 2.5 :

- **global** : la séquence d'exécution entière ;
- **before** : la séquence d'exécution avant un état/événement donné ;
- **after** : la séquence d'exécution à partir d'un état/événement donné ;
- **between** : toute séquence d'exécution située entre un état/événement donné et un autre état/événement donné ;
- **after-until** : comme after si le second état/événement n'apparaît pas dans la séquence, et comme between dans le cas contraire.

Une portée est définie par un état/événement de départ et un état/événement de fin. Pour des portées délimitées par des états, l'intervalle évalué est fermé à gauche et ouvert à droite. Ce qui signifie que la portée concerne tous les états depuis l'état de départ, inclus, jusqu'à l'état de fin, non inclus. Ce choix a été fait d'une part parce qu'il est facilement exprimable en spécification et il représente le cas le plus courant rencontré lors des études de cas réels de propriétés. Pour les formalismes basés sur les événements, le modèle ne permet pas à deux événements de coïncider, ce qui implique que les intervalles sont ouverts des deux côtés.

CTL S precedes P:

Globally	$\neg E[\neg S\ \mathcal{U}(P \wedge \neg S))]$
Before R	$\neg E[(\neg S \wedge \neg R)\ \mathcal{U}(P \wedge \neg S \wedge \neg R \wedge EF(R))]$
After Q	$\neg E[\neg Q\ \mathcal{U}(Q \wedge \neg E[\neg S\ \mathcal{U}(P \wedge \neg S)])]$
Between Q and R	$AG(Q \rightarrow \neg E[(\neg S \wedge \neg R)\ \mathcal{U}(P \wedge \neg S \wedge \neg R \wedge EF(R))])$
After Q until R	$AG(Q \rightarrow \neg E[(\neg S \wedge \neg R)\ \mathcal{U}(P \wedge \neg S \wedge \neg R)])$

LTL S precedes P:

Globally	$\Diamond P \rightarrow (\neg P\ \mathcal{U}(S \wedge \neg P))$
Before R	$\Diamond R \rightarrow (\neg P\ \mathcal{U}(S \vee R))$
After Q	$\Box\neg Q \vee \Diamond(Q \wedge (\neg P\ \mathcal{U}(S \vee \Box\neg P)))$
Between Q and R	$\Box((Q \wedge \Diamond R) \rightarrow (\neg P\ \mathcal{U}(S \vee R)))$
After Q until R	$\Box(Q \rightarrow ((\neg P\ \mathcal{U}(S \vee R)) \vee \Box\neg P))$

Quantified Regular Expressions

Event S precedes P:

Globally	$[-P]^* \mid ([-S, P]^*; S; \Sigma^*)$
Before R	$[-R]^* \mid ([-P, R]^*; R; \Sigma^*) \mid ([-S, P, R]^*; S; \Sigma^*)$
After Q	$[-Q]^*; (Q; ([-P]^* \mid ([-S, P]^*; S; \Sigma^*)))^?$
Between Q and R	$[-Q]^*; (Q; ([-P, R]^* \mid ([-S, P, R]^*; S; [-R]^*)); R; [-Q]^*)^*; (Q; [-R]^*)^?$
After Q until R	$[-Q]^*; (Q; ([-P, R]^* \mid ([-S, P, R]^*; S; [-R]^*)); R; [-Q]^*)^*;$ $(Q; ([-P, R]^* \mid ([-S, P, R]^*; S; [-R]^*)))^?$

FIG. 2.6 – Exemple de spécification PRECEDENCE

Exemple de spécification La figure 2.6 présente le *mapping* du patron PRECEDENCE par les cinq portées et avec les sémantiques pour trois formalismes. Cette figure permet de décrire une relation entre deux événements/états (ici S et P) où l'occurence du premier (S) est une précondition pour l'occurence du second (P). Selon la portée que l'on souhaite, on obtient la formule immédiatement. Ainsi, il suffit de pouvoir exprimer une propriété avec un couple patron/portée pour avoir la formule équivalente.

La description des patrons en détail, avec des explications supplémentaires, des exemples et les traductions dans les formalismes les plus communs est disponible sur le site internet [DAC97]. Nous ne présentons dans ce rapport qu'une brève description des patrons. Par simplification d'écriture, nous dirons "un état/événement donné arrive" pour signifier qu'un état pour lequel la formule d'état donnée est vraie, ou un événement de la disjonction d'événements donnée arrive.

En plus de ces patrons, le système fournit *des notes de patrons* permettant de décrire comment construire les variantes connues des patrons basiques. Par exemple, pour changer la fermeture de l'intervalle de la portée, par défaut fermé-ouvert. Elles décrivent aussi comment certaines substitutions peuvent être réalisées à l'intérieur d'une portée. La sémantique de ce système est décrit dans cinq formalismes différents : les trois premiers, décrits dans [DAC99] et sur le site internet : LTL, CTL et QRE ; et deux autres, ajoutés par la suite : Graphic Interval Logic (GIL) et INCA query language [Sie02].

Les spécifications de propriété testées Pour expérimenter leur système, les auteurs ont collecté 555 spécifications d'au moins 35 sources différentes et de domaines d'application variés (protocole hardware, protocole de communication, GUI, système de contrôle, types de données abstraites, systèmes d'exploitation, systèmes distribués et bases de données). Ces spécifications ont été trouvées dans :

- des articles sur la vérification dans la littérature ;
- les travaux des personnes ayant écrit ou collecté des spécifications par exemple les chercheurs ayant construit ou utilisé les outils de vérification ;
- des projets d'étudiants.

Les descriptions de toutes les spécifications sont disponibles sur le site internet [DAC97].

Résultats Nous ne présenterons pas ici en détail les résultats, mais juste ceux qui sont intéressants dans le cadre de ce rapport. On remarque dans la figure 2.7 qu'une très grande partie des propriétés sont exprimables à partir de quelques patrons et 78% des propriétés sont exprimables par une seule portée : *Global*. Si on ajoute que certains patrons sont liés, que ce soit par dualité (*Absence* et *Universality*) ou par généralisation (*Existence* et *Bound Existence*, *Response* et *Response Chain*, et *Precedence* et *Precedence Chain*), il apparaît qu'en ne traitant dans un premier temps que certaines combinaisons, nous pouvons très facilement atteindre une couverture importante. On constate surtout que ce système permet d'exprimer une très grande partie des propriétés collectées.

2.3 Synthèse et directions de travail

Les spécifications de propriété de DWYER Ce système de modèles de spécification de propriété peut servir de base pour la conception d'outils de vérification, et on peut

Pattern	Scope					
	Glbl	Bfr	Aftr	Btwn	Untl	Tot
Absence	41	5	12	18	9	85
Universality	110	1	5	2	1	119
Existence	12	1	4	8	1	26
Bound Exist	0	0	0	1	0	1
Response	241	1	3	0	0	245
Precedence	25	0	1	0	0	26
Resp. Chain	8	0	0	0	0	8
Prec. Chain	1	0	0	0	0	1
UNKNOWN						44
Total	438	8	25	29	11	555

FIG. 2.7 – Répartition des propriétés de l'étude

s'en inspirer pour exprimer les propriétés de sûreté. D'après cette étude, en ne traitant seulement que quelques cas, on couvre une très grande partie des propriétés étudiées. Il faut nuancer ce résultat par le fait que dans le cadre de ce stage, nous nous interessons à des propriétés particulières (sûreté) et surtout ce sont des propriétés qui peuvent être plus complexes que la moyenne, car pour exprimer un besoin de test précis il faudra parfois que la propriété soit elle aussi précise. De plus, le manque de retour sur ce système ainsi que la difficulté et le temps nécessaire pour expérimenter sérieusement et exhaustivement ce système n'ont pas permis aux auteurs de conclure sur la pertinence de leur système.

Les stratégies de JML-TT Cette approche, suivie par un outil testé et expérimenté, peut servir de point de départ, puisqu'il s'agit d'analyser syntaxiquement une propriété de sûreté et de générer une suite de tests spécifique grâce à l'utilisation de stratégie. La décomposition des stratégies en séquence de primitives donne aussi des idées sur les mécanismes permettant de construire un test en se plaçant dans la situation de la propriété.

Mais les différences sur les objectifs et sur les langages limitent rapidement les similitudes : seules les opérations sont prises en compte lors de la construction des stratégies car les propriétés JML sont vérifiées lors de l'exécution du programme. Elles n'apparaissent donc pas dans les stratégies. Alors que nous avons considéré qu'il serait utile dans le but de pouvoir exprimer le plus de besoins de tests différents, de pouvoir prendre en compte les prédicats sur les variables lors de la construction des schémas de tests, puisque le langage des schémas de test le permet justement.

L'approche présentée dans [BDJG06] n'a pour objectif de générer une liste exhaustive de tests pour la propriété. Elle couvre par les primitives et les algorithmes qui sont associés le plus de comportements possibles, et privilégie l'efficacité : un comportement déjà activé sera évité le plus possible dans la suite du test, et l'ordre dans lequel sont appelées les opérations n'est jamais remis en cause. Or cela correspond précisément aux besoins de test exprimés dans le projet POSE : s'assurer qu'aucune séquence n'a d'effets de bord indésirables, et avoir toutes les combinaisons de séquences est donc primordial. Pour ces raisons, la démarche choisie s'inspirera de celle-ci, même si de nombreux changements seront réalisés pour correspondre aux objectifs du stage.

Le projet POSE et langage des schémas de test Le projet POSE a permis la mise en place d'un langage de schémas de test et de l'utiliser dans un projet de taille industrielle. Le langage a donc permis d'exprimer ce qui était nécessaire pour le projet. Le langage de schéma sera repris comme format de sortie, en s'assurant que d'éventuelles modifications soient réalisables en pratique. Cela permet aussi de se limiter dans un premier temps aux besoins exprimés dans ce projet, par exemple de ne pas considérer toutes les propriétés possibles mais seulement les propriétés de sûreté, même s'il serait un plus évident de pouvoir *in fine* traiter plus que ce qui est demandé dans le cadre de ce projet.

Deuxième partie

Contributions

Chapitre 3

Méthode de génération de scénarios

Cette partie présente la génération des scénarios à partir d'un besoin de test et d'une propriété de sûreté. Afin de comprendre comment nous sommes arrivés à ce processus, nous allons d'abord présenter la méthode de travail, qui expliquera comment nous avons exploité les travaux présentés en 2 et les choix faits durant le stage.

3.1 Description de la méthode de travail

Pour réaliser ce travail, nous avons du d'abord définir les formats d'entrée et sortie, à savoir les propriétés et les scénarios. Puis, il a fallu définir une méthode pour fournir des besoins de tests concernant toutes les propriétés, en prenant en compte que les besoins de tests sont a priori indéfinissables puisque ce sont les utilisateurs qui ressentent d'après leur expérience du modèle qu'ils ont la nécessité de pouvoir tester la propriété de telle manière. Pour permettre de sélectionner parmi un nombre potentiellement très important de suite, et pour correspondre un peu plus aux besoins de test complexes, nous avons prévu un système de filtres permettant d'affiner et de préciser la suite de tests. Cette partie présente donc les choix faits pour chacun de ces points, mais ne présentera pas les résultats.

Les propriétés Pour les propriétés, nous avons utilisé le modèle proposé par Dwyer présenté dans 2.2. Nous avons repris le même système de portée/patron ainsi que leurs sémantiques.

Les paramètres des propriétés peuvent être indifférement des prédicats ou des opérations.

Pour exploiter ce système, nous avons rapidement remarqué qu'il était impossible de traiter une portée indépendament du patron, et réciproquement. Ce qui nous impose de travailler à partir d'un couple portée/patron pour définir une stratégie, et donc de traiter toutes les combinaisons portée/patron. Mais en pratique, il s'est avéré que beaucoup de couples étaient traitables en fonction d'autres couples, voire même de réécrire une propriété de manière à ce qu'on puisse la traiter comme un couple patron/portée déjà traité. Par exemple une propriété de type *AFTER A UNTIL B EXISTENCE C* peut être traitée en utilisant les deux propriétés *AFTER A EXISTENCE C* et *BETWEEN A AND B EXISTENCE C*. Cela évite lors de l'implémentation d'un outil reprenant la démarche une grande partie d'un travail laborieux.

Les scénarios Les scénarios sont dans le format défini par le projet POSE et expliqué dans la section 2.1.2. Le langage de schéma permet de construire des séquences en fonction des prédicats et/ou des opérations, voire des comportements. Pour simplifier les expressions, nous avons ajouté un raccourci d'écriture pour signifier "'les comportements de l'opération A terminant normalement"', noté dans ce rapport $[A_{OK}]$, et pour "'les comportements de l'opération A terminant par une erreur"' le raccourci $[A_{KO}]$.

Les besoins de test Les besoins de test sont des demandes textuelles faites par un ingénieur pour tester une propriété sur un modèle. Comme il est impossible de les lister, car ils dépendent de l'expérience de l'ingénieur, du modèle, du domaine de l'application, de la propriété, nous avons préféré fournir une méthode permettant de traiter des besoins de tests simples, mais de manière la plus exhaustive : à partir d'un couple portée/patron, on définit une liste de besoins de test de manière systématique. Cette analyse systématique des propriétés, basée sur la manière de construire la séquence, a permis de définir un système permettant à partir d'une propriété une liste de besoins de test disponible. et parmi ceux ci, il y a un besoin de test qui permet en fait de tester si le modèle vérifie la propriété, puisqu'il essaiera de construire toutes les séquences que le modèle devrait empêcher, car violant la propriété. L'explication des besoins de test et les différences entre chacun sera faite à partir d'un exemple dans la partie 3.3

3.2 Description des primitives

Les primitives sont des opérations plus élémentaires que les stratégies : ce sont en effet les briques, constituant les stratégies, qui permettent de construire une séquence vérifiant une propriété en particulier : absence d'une opération et/ou d'un prédicat, finissant par telle opération, dans tel état etc. D'une manière générale, elles consistent à construire des séquences et à ne conserver que celles qui vérifient le critère de sélection spécifique à chaque primitive. C'est pour cela que parfois on utilise le terme de construction uniquement, pour résumer le processus construction/sélection. Les primitives explorent tous les comportements, et donc construisent des séquences, mais elles ne retournent que celles qui doivent être sélectionnées, dès que le critère (ou un des critères) est vérifié.

Cette partie décrit les primitives utilisées pour les travaux réalisés du stage, notament la manière dont elles ont été définies. Pour chacune des primitives, nous expliquerons sa signification, son utilité et la manière de la traduire en schéma de test.

Inspirées de l'article [BDJG06], nous avons du les adapter afin de palier les manques exprimés dans la partie 2.1.3 à savoir :

- les prédicats ne sont pas gèrés dans les stratégies ;
- les primitives n'ont pas pour objectif de construire toutes les séquences.

Pour le second point, il a été comblé naturellement en définissant pour chaque primitive un schéma de test qui représente toutes les séquences.

Pour le premier point, il a fallu doté les primitives de deux paramètres permettant de gérer les prédicats et les opérations. Plus précisément, les paramètres sont des listes de prédicats et d'opérations car selon les propriétés, il est possible que des stratégies aient besoin de primitives avec plusieurs opérations, ou plusieurs prédicats. La signification des listes n'est pas la même selon que ce sont des prédicats ou des opérations. Une liste

d'opérations s'interprète comme une disjonction des opérations. Une liste de prédicats s'interprète comme une conjonction des opérations. Cette distinction vient en fait de la manière dont sont utilisées les primitives dans les stratégies : une primitive permet de construire une séquence satisfaisant une caractéristique, dépendant des paramètres. Quand ce sont des opérations, la primitive doit constuire une séquence pour chaque opération, alors que si ce sont des prédicats, il faut construire une séquence dont la caractéristique est maintenant relative à la conjonction des prédicats. Par exemple, une primitive qui consisterait à construire une séquence finissant ce qui est en paramètre : si c'est une liste d'opération, cela signifie construire toutes les séquences finissant par une des opérations, alors que si c'est une liste de prédicats, cela signifie construire toutes les séquences finissant par un état vérifiant la conjonction de tous les prédicats. Si les deux listes données en paramètre sont non vides, alors le critère de sélection est donc de trouver un comportement d'une opération de la liste vérifiant tous les prédicats de la liste des prédicats. Il s'agit donc de réaliser une conjonction des deux critères précédents.

Paramètres Comme toutes les primitives sont traduites en schéma de test, et que toutes ont le même typage de paramètres, nous avons utilisé par soucis de lisibilité dans ce rapport les mêmes noms pour les paramètres de même type :

- lo : pour la liste d'opération, définie ainsi : $lo = o_1, o_2, ..., o_m$
- lp : pour la liste des prédicats, définie ainsi : $lp = p_1, p_2, ..., p_q$

Profondeur d'exploration L'objectif des primitives est de construire toutes les séquences vérifiant une propriété spécifique, mais il faut limiter la longueur des séquences qu'elles peuvent construire, d'une part pour éviter une explosion combinatoire trop importante et inutile, et d'autre part car il n'y a aucune raison *a priori* que le nombre de séquences soit fini. La profondeur est donc paramétrée par un entier n. La valeur de ce paramètre peut avoir une importance particulière dans certaines séquences de primitives, et une certaine expérience sur l'influence de celle-ci permettra surement d'améliorer l'efficacité des tests produits.

3.2.1 CoverStop (lp,lo)

Définition *CoverStop* réalise une couverture de tous les comportements de toutes les opérations en profondeur. Il s'arrête quand il a atteint la longueur maximale autorisée ou quand il ne peut plus activer de comportements. Le critère de sélection de séquence est donc de conserver toutes séquences finissant par :

- soit un comportement d'une opération de lo si lp est la seule liste vide.
- soit un état vérifiant toutes les préconditions de lp si lo est la seule liste vide.
- soit un comportement d'une opération de lo menant dans un état vérifiant toutes les préconditions de lp.

Donc si lp est vide, alors la séquence finira par une des opérations de lo. La primitive construira alors toutes les séquences finissant par un des comportements d'une des opérations de la liste : $[\$op_{OK}]^{0..n-2}$. $[\$op]?$. [o_1 | o_2 | ... | o_m]

Et si lo est vide, alors la séquence finira par un état vérifiant la conjonction des prédicats. La primitive construira alors toutes les séquences finissant par un état vérifiant la conjonction de tous les prédicats de la liste : $[\$op_{OK}]^{0..n-1}$. $[\$op] \rightsquigarrow (p_1 \wedge p_2 \wedge ... \wedge p_q)$.

La primitive parcourt donc tous les comportements de toutes les opérations, et renvoie la séquence parcourue dès que un des critères de sélection est trouvé (comportement d'une opération de la liste ou état vérifiant tous les prédicats) car cela signifie que la séquence finit comme on le souhaite, et continue l'exploration, y compris celle de la séquence. Il est donc possible que les séquences sélectionnées possèdent plusieurs états vérifiant la condition.

CoverStop({},{}) avec des listes vides en paramètre réalise une couverture de tous les comportements de toutes les opérations sans restriction. Autorisée, cette utilisation est toutefois à éviter car n'a pas vraiment de sens. Pour une couverture générale, il vaut mieux utiliser CoverExcept({},{}) (cf 3.2.3).

Exemple d'utilisation : Cette primitive permet donc de récupérer les plus longues séquences finissant par un état particulier et ce dans toutes les branches. Ce qui est particulièrement intéressant quand on veut tester la présence de plusieurs états particuliers et avoir la plus grande variété et quantité d'opérations avant l'état dont on a besoin.

Schémas de test : CoverStop(lp,lo) :

si $lp = \{\}$ alors

$[\$op_{OK}]^{0..n-2}$. $[\$op]?$. [o_1 | o_2 | ... | o_m]

si $lo = \{\}$ alors

$[\$op_{OK}]^{0..n-1}$. $[\$op]$ $\rightsquigarrow$ $(p_1 \wedge p_2 \wedge ... \wedge p_q)$

si $lo \neq \{\}$ et $lp \neq \{\}$ alors

$[\$op_{OK}]^{0..n-2}$. $[\$op]?$. [o_1 | o_2 |...| o_m] $\rightsquigarrow$ $(p_1 \wedge p_2 \wedge ... \wedge p_q)$

si $lp = \{\}$ et $lo = \{\}$ alors

$[\$op]^{0..n}$

3.2.2 CoverStopExact(lp,lo)

Définition : *CoverStopExact* est une variante de *CoverStop*. Elle réalise elle aussi une couverture des comportements des opérations en profondeur, et s'arrête dans les mêmes conditions. Toutefois, cette opération ne tolère qu'une présence d'un état vérifiant les conditions d'arrêt, et donc arrête l'exploration de cette branche dès qu'on trouve un critère de sélection. Les critères de sélection sont donc, selon les paramètres :

Soit un comportement d'une opération de lo si lp est la seule liste vide :

$[\$op_{OK} \backslash lo]^{0..n-2}$. $[\$op \backslash lo]?$. [o_1 | o_2 |...| o_m]

Soit un état vérifiant toutes les préconditions de lp si lo est la seule liste vide :

$[\$op_{OK} \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]^{0..n-1}$. $[\$op]$ $\rightsquigarrow$ $(p_1 \wedge p_2 \wedge ... \wedge p_q)$

Soit un comportement d'une opération de lo dans un état vérifiant toutes les préconditions de lp :

$[\$op_{OK} \setminus lo \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]^{0..n-2}$. $[\$op \setminus lo \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]?$. $[o_1 \mid o_2 \mid ... \mid o_m]$ $\rightsquigarrow$ $(p_1 \wedge p_2 \wedge ... \wedge p_q)$

Cette primitive permet donc de parcourir tous les comportements de toutes les opérations, mais arrête les tentatives de prolonger une séquence dès qu'un des états vérifient le critère de sélection. La séquence est alors sélectionnée mais pas prolongée.

CoverStopExact({},{}) avec des listes vides en paramètre réalise une couverture de tous les comportements de toutes les opérations sans restriction. Autorisée, cette utilisation est toutefois à éviter car n'a pas vraiment de sens. Pour une couverture générale, il vaut mieux utiliser CoverExcept({},{}) (3.2.3).

Exemple d'utilisation : Cela permet de trouver la séquence la plus courte finissant par un état vérifiant un critère de sélection. C'est interessant quand on veut seulement tester la présence d'un et un seul état vérifiant un critère de sélection, et que la présence de plusieurs occurences ne peut rien changer, car en dehors du cadre de test que l'on veut définir.

Schéma de test : CoverStopExact(lp,lo)
si $lp = \{\}$ alors
$[\$op_{OK} \backslash lo]^{0..n-2}$. $[\$op \backslash lo]?$. [o_1 | o_2 |...| o_m]
si $lo = \{\}$ alors
$[\$op_{OK} \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]^{0..n-1}$. $[\$op] \rightsquigarrow (p_1 \wedge p_2 \wedge ... \wedge p_q)$
si $lo \neq \{\}$ et $lp \neq \{\}$ alors
$[\$op_{OK} \setminus lo \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]^{0..n-2}$. $[\$op \setminus lo \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]?$. $[o_1 \mid o_2 \mid ... \mid o_m] \rightsquigarrow (p_1 \wedge p_2 \wedge ... \wedge p_q)$
si $lp = \{\}$ et $lo = \{\}$ alors
$[\$op]^{0..n}$

3.2.3 CoverExcept(lp,lo)

Définition : *CoverExcept(lp,lo)* permet de construire toutes les séquences vérifiant une caractéristique précise. L'exploration s'arrête quand on a atteint la longueur maximale, quand on ne peut plus activer de comportements ou quand on ne peut plus activer de comportement sans violer la caractéristique. La sémantique est en fait :

- soit l'absence de tous les comportements de toutes les opérations de lo si lp est la seule liste vide.
- soit l'absence d'état vérifiant un des prédicats de lp, c'est-à-dire la présence exclusivement d'états vérifiant la conjonction de la négation de tous les prédicats de lp si lo est la seule liste vide.
- soit la présence de seulement des comportements n'appartenant pas à des opérations de lo et vérifiant la conjonction de la négation de tous les prédicats de lp

Elle réalise une couverture de tous les comportements de toutes les opérations, hormis naturellement celles de lo.

Exemple d'utilisation : *CoverExcept* permet donc, en pratique, de s'assurer de l'absence d'une opération ou d'un prédicat dans une séquence. C'est pour cette raison que l'on n'utilise pas la négation de la conjonction des prédicats, ce qui semblait plus intuitif, car dans les stratégies, c'est bien l'absence de tous les prédicats que l'on veut observer, et pas seulement l'absence de tous en même temps. *CoverExcept({},{})* avec des listes vides en paramètre réalise une couverture de tous les comportements de toutes les opérations sans restriction.

Schéma de test : CoverExcept(lp,lo)
si $lp = \{\}$ alors
$[\$op_{OK} \setminus lo]^{0..n-1}$. $[\$op \setminus lo]$
si $lo = \{\}$ alors
$[\$op_{OK} \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]^{0..n-1}$. $[\$op \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]$
si $lo \neq \{\}$ et $lp \neq \{\}$ alors
$[\$op_{OK} \setminus lo \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]^{0..n-1}$. $[\$op \setminus lo \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]$
si $lp = \{\}$ et $lo = \{\}$ alors
$[\$op]^n$

3.2.4 Active(lo)

Définition : *Active(lo)* est une primitive permettant d'activer tous les comportements de toutes les opérations de lo.

Exemple d'utilisation : Cela permet de forcer la présence d'un comportement directement, en évitant un parcours en profondeur d'un *CoverStopExact* ou d'un *Research* qui ne nous permet pas de connaître quoique ce soit sur les opérations et états arrivant avant le comportement recherché.

Schéma de test : Active(lo)
$[o_1|o_2|...|o_m]$

3.2.5 Find(lp)

Définition : *Find(lp)* est l'équivalent de *Active* pour les propriétés d'états. Il permet de s'assurer de la présence d'un état vérifiant la conjonction des prédicats . Elle active un comportement permettant d'être dans un état vérifiant le critère de sélection.

Schéma de test : Find(lp)
$[\$op] \rightsquigarrow (p_1 \wedge p_2 \wedge ... \wedge p_q)$

3.2.6 Research(lp,lo)

Définition : *Research(lp,lo)* permet une recherche efficace d'un des comportements de l'une des opétations de lp sans se soucier de la manière (tant des opérations que de des états et de leurs propriétés) ou de la couverture. Elle ne retourne qu'une séquence, la plus courte possible, mais rien ne permet d'assurer qu'il n'en existe pas de plus courte, car il s'agit d'un parcours en profondeur. Il n'y a donc pas de couverture.

Schéma de test : Research(lp,lo)
si $lp = \{\}$ alors
$[\$op_{OK} \setminus lo]^{0..n-2}$. $[\$op \setminus lo]$. $[\ o_1 \mid o_2 \mid ... \mid o_m\ ]$
si $lo = \{\}$ alors
$[\$op_{OK} \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]^{0..n-1}$. $[\$op \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]$. $[\$op] \rightsquigarrow (p_1 \wedge p_2 \wedge ... \wedge p_q)$

si $lo \neq \{\}$ et $lp \neq \{\}$ alors
$[\$op_{OK} \setminus lo \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)]^{0..n-2} . [\$op \setminus lo \rightsquigarrow (\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_q)] . [o_1|o_2|...|o_m] \rightsquigarrow (p_1 \wedge p_2 \wedge ... \wedge p_q)$

Les primitives ne permettent que de construire des bouts de schémas de test, et pour avoir une relation avec une propriété, il faut définir un lien entre propriété et primitive. Ce lien est la stratégie, et cette notion est décrite dans la partie suivante.

3.3 Description des stratégies

Les stratégies sont des séquences de primitives permettant de générer un schéma de tests en imposant certaines contraintes sur la séquence d'état/événement. Elles permettent donc de générer un schéma de tests spécifique à une propriété. Dans cette partie, nous présentons pour quelques propriétés les besoins de tests pouvant y être associés et les stratégies qui correspondent au couple propriété/besoins de tests. L'explication de cette partie se fera sur un exemple.

Pour illustrer cette partie, nous nous servirons de l'exemple d'une propriété : *Globally S Precedes P*, où S et P sont des paramètres de la propriété, connus lorsque l'ingénieur arrive à cette étape qui consiste à générer les schémas de tests. Toutefois, S et P peuvent être indifféremment des prédicats ou des opérations. Nous commenterons la présentation de cette stratégie, sachant que toutes les stratégies ont une présentation similaire.

3.3.1 GLOBALLY S PRECEDES P

Définition informelle : Si un P arrive, alors on a un S avant (on ne peut avoir P avant le premier S)

Définition formelle :

Cette partie donne la définition formelle de la propriété donnée par Dwyer dans les trois formalismes LTL, CTL et QRE.

Dans le formalisme QRE, . est l'ensemble des événements, $[-P, Q, R]$ signifie un événement sauf P, Q et R, $A?$ signifie 0 ou 1 fois A.

Voici les formules correspondantes : $LTL : \neg P\ W\ S$ ou $\Diamond(P) \rightarrow (\neg P\ U\ S)$
$CTL : A [\neg P\ W\ S]$
$QRE : [-P] * ; P ; . *$

Besoins de Tests :

(8) on teste une séquence sans S finissant par P

(9) on teste une séquence sans S

(10) on teste une séquence finissant par P

En plus des besoins de tests liés à la propriété, nous avons ajouté des exemples de besoins de tests "limites" : ce sont des besoins de tests non définis par la propriété mais qui peuvent être intéressants : intervalle vide, simultanéité de plusieurs particularités sur un même état par exemple. Il est cependant évident que deux événements ne peuvent arriver simultanément, et donc il est possible que selon la nature des paramètres de la propriété S

et P, certains tests limites ne pourront exister. Pour cet exemple, une séquence où S et P arrivent en même temps n'est pas concerné par la propriété. Mais il peut être intéressant pour l'utilisateur de le tester quand même, mais ce sera à lui de décider quel sera le verdict du test. Si S et P sont des comportements, ils ne peuvent arriver simultanément, et donc, ce besoin de test devra être négligé.

Besoins de Tests limites : Tester une séquence ou S et P arrivent en même temps.

Stratégie :

Cas nominal : Nous présenterons aussi la stratégie pour construire le cas nominal. Toutefois, cette stratégie ne sera pas réellement détaillée selon toutes les combinaisons des types de paramètres, mais seulement par des primitives généralisées . On ne détaillera pas en effet les paramètres selon le type opération/événement.

CoverStop(S).Research(P)

Construction d'une séquence avec un S suivi d'un P.

Nous avons procédé par une analyse combinatoire des propriétés en fonction des types des paramètres. Il en résulte qu'un très grand nombre de stratégies ont été élaborées, et pour des raisons pratiques, elles sont présentées dans ce rapport dans des tableaux. Chaque propriété ayant un nombre de paramètres différents, allant de un à quatre, les tableaux ont une morphologie différente, mais le principe de lecture est le même.

Cette méthode, appliquée à tous les couples portée/patron, génèrent des stratégies qui peuvent parfois ne plus être toujours pertinentes, mais c'est généralement parce que ce ne sont plus alors des propriétés de sûreté. Toutefois, comme le tableau est défini facilement, et les cases voisines se remplissant très rapidement (il s'agit souvent de déplacer un paramètre dans une primitive) avec un peu de pratique, on peut tout de même proposer la stratégie, car si elle ne permet pas toujours de conclure, elle peut tout de même trouver des erreurs.

Lecture des tableaux Chaque tableau a en entrée les paramètres de la propriété (S et P dans cet exemple). Il y a donc une pour chaque paramètre une entrée par type possible : propriété et événement. Toutefois, lors de l'analyse, il est apparu qu'il est parfois intéressant de ne pas utiliser un paramètre pour la construction des schémas : quand une propriété utilise P et S, on peut vouloir construire une séquence en fonction seulement de P, et vérifier que l'exécution du test vérifie l'apparition ou l'absence de S selon la propriété (dans l'exemple, on vérifiera à l'exécution que S apparaisse dans la séquence strictement avant P). C'est pour cela que pour chaque paramètre il y a une troisième colonne, "'non utilisé"' qui correspond aux stratégies qui construisent les séquences sans tenir compte de ce paramètre. Cela constitue d'ailleurs un besoin de test à part entière : omettre un paramètre dans la construction d'une séquence permet de tester des séquences intéressantes.

En effet, si le modèle vérifie la propriété de notre exemple, si S et P sont des opérations, il sera impossible de construire une séquence correspondant à la stratégie, car il sera impossible de générer une séquence avec un P sans S avant. Mais un besoin de test peut être de construire une séquence finissant par un P, ce qui est toujours possible (sinon cela signifie que P ne peut jamais être activé), et à l'exécution, on vérifie que S est apparu avant. Cela est aussi valable si les paramètres sont des prédicats. Pour trouver la statégie

S ⋱ P	evénement	propriété	non utilisé
evénement	CExc({},S).active(P) (BT8)	CExc({},S).Find(P) (BT8)	CExc({},S) (BT9)
propriété	CExc(S,{}).active(P) (BT8)	CExc(S,{}).Find(P) (BT8)	CExc(S,{}) (BT9)
non utilisé	CSExact({},P) (BT10)	CSExact(P,{}) (BT10)	CExc({},{})

FIG. 3.1 – Stratégies pour la propriété Globally S Precedes P

correspondant à la propriété, des paramètres définis et un besoin de tests fixé, il suffit de trouver l'unique case du tableau correspondant à ce cas.

Par exemple, pour la propriété *GLOBALLY S PRECEDES P*, ou S est une propriété et P un événement : dans le tableau 3.1 de la page 43, on trouve les stratégies correspondant à la propriété. Plus précisément, dans la case S propriété et P événement, on trouve la stratégie *CoverExcept(S,{}). Active(P)* correspondant au besoin de test 8 *on teste une séquence sans S finissant par P.*

Toutefois, il y a deux autres besoins de tests disponibles :

– BT9 : on teste une séquence sans S

– BT10 on teste une séquence finissant par P

On remarque que pour le besoin de test 9, P n'apparaît pas. Cela signifie que les tests sont construits sans tenir compte de P, et que donc l'existence ou l'absence de P aux endroits voulus seront vérifiés lors de l'exécution sur l'implémentation. Comme P n'est pas utilisé, il faut regarder la case S propriété et P non utilisé, et on trouve la stratégie *CoverExcept(S,{})*, qui construit bien une séquence sans S. Lors de l'exécution, il faudra vérifier que P n'apparait pas.

De la même manière, pour le besoin de test 10, c'est S qui n'est pas utilisé. Il faut donc chercher la case P événement et S non utilisé, et on obtient *CoverStopExact({},P)* qui construit bien une séquence avec un seul P finissant par ce P. Lors de l'exécution, on vérifiera la présence d'un S avant (strictement) le P.

Dans tous les tableaux, nous utilisons des abréviations :

CExc pour CoverExcept

CSExact pour CoverStopExact

CS pour CoverStop

Voir la figure 3.1

3.3.2 BEFORE R EXISTENCE P

Définition formelle : Si R arrive, alors P doit arriver au moins une fois avant.

Définition informelle :

$LTL : \neg R\, W\, (P \wedge \neg R)$

$CTL : A\, [\neg R\, W\, (P \wedge \neg R)]$

$QRE : [-R]\ *\ |\ [-P, R]\ *\ ;\ P;\ .\ *$

Besoins de Tests (1) Tester une séquence terminant par R sans P

P`··. R	evénement	propriété	non utilisé
événement	CExc({},P).active(R) (BT1)	CExc({},P).Find(R) (BT1)	CExc({},P) (BT2)
propriété	CExc(P,{}).active(R) (BT1)	CExc(P,{}).Find(R) (BT1)	CExc(P,{}) (BT2)
non utilisé	CSExact({},R) (BT3)	CSExact(R,{}) (BT3)	

FIG. 3.2 – Stratégies pour la propriété Before R existence P.

(2) Tester une séquence sans P
(3) Tester une séquence finissant par R
Besoins de Tests limites : Tester une séquence avec P et R en même temps.

Stratégie : cas nominal : Construction d'une séquence ayant un P suivi d'un R.
CoverStop(P).CoverStopExact(R)
Voir la figure 3.2

3.3.3 BETWEEN Q AND R EXISTENCE P

Définition informelle : Dans tout intervalle minimum]Q,R[, on doit avoir au moins un P. Un intervalle minimum est la plus petite séquence commençant par Q finissant par R dans laquelle il n'existe aucun autre intervalle.

Définition formelle : $LTL : \Box\,(Q \wedge \neg R \rightarrow (\neg R\, U\, (P \wedge \neg R))\,)$
$CTL : AG\,(Q \wedge \neg R \Rightarrow A\,[\neg R\, W\, (P \wedge \neg R)\,]$
$QRE : ([-Q]*\,;\, Q\,;\, [-P,R]*\,;\, P\,;\, [-R]*\,;\, R)*\,;\, [-Q]*;\, (Q\,;\, [-R]*)\,?$

Besoins de Tests : (4) on teste une séquence avec un Q suivi d'une suite sans P finissant par R
(5) on teste une séquence avec un Q suivi d'une suite sans P
(6) on teste une séquence avec un Q
(7) on teste une séquence avec un Q suivi d'une séquence sans R suivie d'un R
Besoins de Tests limites :
Tester une séquence avec un intervalle vide (Q et R apparaissent simultanément).
Tester une séquence avec Q et P apparaissent simultanément.
Tester une séquence avec R et P apparaissent simultanément.

Stratégie : **Cas nominal :** CoverStop(Q).CoverStop(P).CoverStopExact(R)
Construction d'une séquance avec un Q puis au moins un P puis un R.
Voir la figure 3.3
Dans le tableau ne sont pas définies les stratégies quand R est une propriété. Pour les avoir, il suffit, à partir des stratégies avec R événement (avec Q et P de même nature), de remplacer les primitives *Active(R)* par *Find(R)* et dans les paramètres des primitives avec deux listes distinctes, de déplacer R dans la première liste. En effet la première liste étant

Q ⋯ R		evénement	non utilisé
	P		
evenement	evt	CS({},Q). CExc({,},P). Active(R) (BT4)	CS({},Q). CExc({},P) (BT5)
	prop	CS({},Q). CExc(P,{}). Active(R) (BT4)	CS({},Q). CExc(P,{}) (BT5)
	ind	CS({},Q). CSExact({},R) (BT7)	CS({},Q). CExc({},{}) (BT6)
propriété	evt	CS(Q,{}). CExc({},P). Active(R) (BT4)	CS(Q,{}). CExc({},P) (BT5)
	prop	CS(Q,{}). CExc(P,{}). Active(R) (BT4)	CS(Q,{}). CExc(P,{}) (BT5)
	ind	CS(Q,{}). CSExact({},R) (BT7)	CS(Q,{}). CExc({},{}) (BT6)

FIG. 3.3 – Stratégies pour la propriété Between Q and R existence P.

celle des prédicats, si R est une propriété, il faut alors l'ôter de la liste des opérations lo pour le mettre dans la liste des prédicats lp.

3.3.4 AFTER Q UNTIL R EXISTENCE P

Définition informelle : Si Q arrive, alors s'il y a un R après, alors il doit y avoir un P entre le Q et le R

Définition formelle : $LTL : \Box(Q \wedge \neg R \rightarrow (\neg R\, W\, (P \wedge \neg R)))$
$CTL : AG\,(Q \wedge \neg R \rightarrow A\,[\,(\neg R\, U\, (P \wedge \neg R)\,])$
$QRE : ([-Q]*;\, Q;\, [-P,R]*;\, P;\, [-R]*;\, R)*;\, [-Q]*;\, (\,Q;$
$[-P,R]*;\, P;\, [-R]*)?\,(\,[-Q]*;\, Q;\, [-P,R]*;\, P;\, [-R]*;\, R)*;$
$[-Q]*;\, (Q;\, [-P,R]*;\, P;\, [-R]*)?$

Besoins de Tests et Stratégies : Comme ce cas est une version moins forte de BETWEEN Q AND R EXISTENCE P, il n'y a pas de stratégies nouvelles : si R arrive, nous sommes dans le cas du scope BETWEEN, sinon celui du cas AFTER (vivacité). Il s'agit donc en fait de réaliser un des besoins de tests de l'une de ces deux propriétés afin de déterminer la stratégie à suivre.

3.3.5 BEFORE R S PRECEDES P

Définition informelle : Si R arrive, alors si P apparait avant R, alors S apparait avant P

Définition formelle : $LTL : \Diamond R \rightarrow (\neg P\, U\, (\,S \mid R\,))$
$CTL : A\,[\,(\,\neg P \mid AG\,(\,\neg R)\,)\, W\, (S \mid R\,)\,]$
$QRE : [-R]\,* \mid (\,[-P,R]*;\, R;\, .*\,) \mid (\,[-S,P,R]*;\, S;\, .*)$

S ⋱ P		evénement	non utilisé
	R		
evenement	evt	CExc({},[R,S,P]). CExc({},[R,P]). Active(P). Research({},R) (BT11)	CExc({},S). Active(R) (BT12)
	prop	CExc(R,[S,P]). CExc(R,P). Active(P). Research(R,{}) (BT11)	CExc({},S). Find(R) (BT12)
	non util	CExc({},[S,P]). Active(P). CExc({},{}) (BT14)	CExc({},S) (BT13)
propriété	evt	CExc(S,[R,P]). CExc({},[R,P]). Active(P). Research({},R) (BT11)	CExc(S,{}). Active(R) (BT12)
	prop	CExc([R,S],P). CExc(R,P). Active(P). Research(R,{}) (BT11)	CExc(S,{}). Find(R) (BT12)
	non util	CExc(S,P). Active(P). CExc({},{}) (BT14)	CExc(S,{}) (BT13)
non utilisé	evt	CExc({},[R,P]). Active(P). Research({},R) (BT15)	CSExact({},R) (BT17)
	prop	CExc(R,P). Active(P). Research(R,{}) (BT15)	CSExact(R,{}) (BT17)
	non util	CS({},P).CExc({},{}) (BT16)	CExc({},{})

FIG. 3.4 – Stratégies pour la propriété Before R S precedes P.

Besoins de Tests :

(11) on teste une séquence où tout P suivi d'un R est précédé d'un S

(12) on teste une séquence sans S finissant par R

(13) on teste une séquence sans S

(14) on teste une séquence sans S suivi d'un P

(15) on teste une séquence avec un P suivi d'un R

(16) on teste une séquence avec un P

(17) on teste une séquence finissant par R

Besoins de Tests limites :

Tester une séquence où R et P arrivent en même temps.

Tester une séquence où R et S arrivent en même temps.

Stratégie Cas nominal : CoverStop(S).CoverStop(P).Research(R)

Construction d'une séquence avec un S suivi d'un P puis d'un R.

Voir la figure 3.4

Dans le tableau ne sont pas définies les stratégies quand R est une propriété. Pour les avoir, il suffit, à partir des stratégies avec R événement (avec Q et P de même nature), de remplacer les primitives *Active(R)* par *Find(R)* et dans les paramètres des primitives avec deux listes distinctes, de déplacer R dans la première liste. En effet la première liste étant celle des prédicats, si R est une propriété, il faut alors l'ôter de la liste des opérations *lo* pour le mettre dans la liste des prédicats *lp*.

3.3.6 AFTER Q S PRECEDES P

Définition informelle : Si Q arrive, alors si P arrive, S arrive avant.

Définition formelle : LTL : $\Box\ \neg Q \mid \Diamond\ (Q\ \&\ (\ \neg P\ W\ S\ )\ )$
CTL : $A\ [\ \neg Q\ W\ (\ Q\ \&\ A\ [\ \neg P\ W\ S\ ]\ )\ ]$
QRE : $[-Q]*\ ;\ (\ Q\ ;\ (\ [-P]\ *\ \mid (\ [-S,P]*\ ;\ S\ ;\ .*\ )\ )\ )?$

Besoins de Tests :

(18) on teste une séquence avec un Q et un P sans S entre les 2
(19) on teste une séquence avec un Q suivi d'une séquence sans S
(20) on teste une séquence sans S finissant par P
(21) on teste une séquence sans S
(22) on teste une séquence avec un Q suivi d'une séquence sans P puis d'un P
Besoins de Tests limites :
Tester une séquence où Q et P arrivent en même temps.
Tester une séquence où Q et S arrivent en même temps.
Tester une séquence où S et P arrivent en même temps.

Stratégie Cas nominal : CoverStop(Q).CoverStop(S).Research(R)

Construction d'une séquence avec un P suivi d'un S puis d'un R.
Voir la figure 3.5

Dans le tableau ne sont pas définies les stratégies quand R est une propriété. Pour les avoir, il suffit, à partir des stratégies avec R événement (avec Q et P de même nature), de remplacer les primitives *Active(R)* par *Find(R)* et dans les paramètres des primitives avec deux listes distinctes, de déplacer R dans la première liste. En effet la première liste étant celle des prédicats, si R est une propriété, il faut alors l'ôter de la liste des opérations lo pour le mettre dans la liste des prédicats lp.

3.3.7 BETWEEN Q AND R S PRECEDES P

Définition informelle : Dans tout intervalle minimum, Si P arrive, alors S arrive avant.

Définition formelle : LTL : $\Box\ (\ (\ Q\ \wedge\ \neg R\ \wedge \Diamond\ R)\ \rightarrow\ (\ \neg P\ U\ (\ S \mid R\ )\ )\ )$
CTL : $AG\ (Q\ \wedge\ \neg R\ \rightarrow\ A\ [\ (\neg P \mid AG\ (\ \neg R\ )\ )\ W\ (\ S \mid R\ )\ ])$
QRE : $[-Q]*\ ;\ (\ Q\ ;\ [-P,R]\ *\ \mid (\ [-S,P,R]*\ ;\ S\ ;\ [-R]*)\ R\ ;\ [-Q]*\ )*;$
$(Q;\ [-R]*)?$

Besoins de Tests :

(23) on teste une séquence ayant un intervalle]Q,R[avec un P mais sans S entre Q et R
(24) on teste une séquence ayant un intervalle]Q,R[sans S entre Q et R
(25) on teste une séquence ayant un Q suivi d'un P sans S entre les deux
(26) on teste une séquence ayant un Q suivi d'une séquence sans S

S ⋱ P		evénement	non utilisé
	Q		
evenement	evt	CS({},Q). CExc({},S). Active(P) (BT18)	CS({},Q). CExc({},S)(BT19)
	prop	CS(Q,{}). CExc({},S). Active(P) (BT18)	CS(Q,{}). CExc({,},S)(BT19)
	non util	CExc({},S). Active(P) (BT20)	CExc({},S) (BT21)
propriété	evt	CS({},Q). CExc(S,{}). Active(P) (BT18)	CS({},Q). CExc(S,{})(BT19)
	prop	CS(Q,{}). CExc(S,{}). Active(P) (BT18)	CS(Q,{}). CExc(S,{})(BT19)
	non util	CExc(S,{}). Active(P) (BT20)	CExc(S,{}) (BT21)
non utilisé	evt	CS({},Q). CSExact({},P) (BT22)	
	prop	CS(Q,{}). CSExact(P,{}) (BT22)	

FIG. 3.5 – Stratégies pour la propriété After Q S precedes P.

(27) on teste une séquence ayant un intervalle]Q,R[avec un P entre Q et R
(28) on teste une séquence avec un Q suivi d'une séquence sans P finissant par P
(29) on teste une séquence avec un intervalle]Q,R[
Besoins de Tests limites :
Tester une séquence avec un intervalle vide (Q et R arrivent en même temps).
Tester une séquence avec S et P arrivant en même temps.
Tester une séquence avec S et Q arrivant en même temps.
Tester une séquence avec R et P arrivant en même temps.

Stratégie : **Cas nominal :** CoverStop(Q).CoverStop(S).CoverStop(P).Research(R)
Construire une séquence avec un Q suivi d'un S, puis d'un P et finalement d'un R.
Voir la figure 3.6

3.3.8 BETWEEN Q AND R ABSENCE P

Définition informelle : Dans tout intervalle minimum]Q,R[, on ne doit pas avoir de P. Un intervalle minimum est la plus petite séquence commençant par Q finissant par R dans laquelle il n'existe aucun autre intervalle.

Définition formelle : $LTL\ :\ \Box\ ((Q\ \wedge\ \neg R\ \wedge \diamond\ R)\ \rightarrow\ (\neg\ P\ U\ R))$
$CTL\ :\ AG\ (Q\ \wedge \neg R \Rightarrow\ A\ [(\neg P\ |\ AG(\neg R))\ W\ R])$
$QRE\ :\ ([-Q] *\ ;\ Q\ ;\ [-P, R] *;\ R) *\ ;\ [-Q] *\ ;\ (Q;\ [-R] *\ )?$

Besoins de Tests :

(30) on teste une séquence avec un intervalle [Q,R] avec un P

S⋯ P			evénement	non utilisé
	P	R		
evenement	evt	evt	CS({},Q).CExc({},[R,S,P]). Active(P). Research({},R) (BT23)	CS({},Q).CExc({},[R,S]). Active(R) (BT24)
		prop	CS({},Q).CExc(R,[S,P]). Active(P). Research(R,{}) (BT23)	CS({},Q).CExc(R,S). Find(R) (BT24)
		non util.	CS({},Q).CExc({},[S,P]). Active(P). CExc({},{}) (BT25)	CS({},Q). CExc({},S) (BT26)
	prop	evt	CS(Q,{}).CExc({},[R,S,P]). Active(P). Research({},R) (BT23)	CS(Q,{}).CExc({},[R,S]). Active(R) (BT24)
		prop	CS(Q,{}).CExc(R,[S,P]). Active(P). Research(R,{}) (BT23)	CS(Q,{}).CExc(R,S). Find(R) (BT24)
		non util.	CS(Q,{}).CExc({},[S,P]). Active(P). CExc({},{}) (BT25)	CS(Q,{}). CExc({},S) (BT26)
propriété	evt	evt	CS({},Q).CExc(S,[R,P]). Active(P). Research({},R) (BT23)	CS({},Q).CExc(S,R). Active(R) (BT24)
		prop	CS({},Q).CExc([S,R],P). Active(P). Research(R,{}) (BT23)	CS({},Q).CExc([S,R],{}). Find(R) (BT24)
		non util.	CS({},Q).CExc(S,P). Active(P). CExc({},{}) (BT25)	CS({},Q). CExc(S,{}) (BT26)
	prop	evt	CS(Q,{}).CExc(S,[R,P]). Active(P). Research({},R) (BT23)	CS(Q,{}).CExc(S,R). Active(R) (BT24)
		prop	CS(Q,{}).CExc([R,S],P). Active(P). Research(R,{}) (BT23)	CS(Q,{}).CExc([S,R],{}). Find(R) (BT24)
		non util.	CS(Q,{}).CExc(S,P). Active(P). CExc({},{}) (BT25)	CS(Q,{}). CExc(S,{}) (BT26)
non utilisé	evt	evt	CS({},Q).CExc({},[R,P]). Active(P). Research({},R) (BT27)	CS({},Q).CExc({},R). Active(R) (BT29)
		prop	CS({},Q).CExc(R,P). Active(P). Research(R,{}) (BT27)	CS({},Q).CExc(R,{}). Find(R) (BT29)
		non util.	CS({},Q).CExc({},P). Active(P). CExc({},{}) (BT28)	CS({},Q). CExc({},{}) (BT30)
	prop	evt	CS(Q,{}).CExc({},[R,P]). Active(P). Research({},R) (BT27)	CS(Q,{}).CExc({},R). Active(R) (BT29)
		prop	CS(Q,{}).CExc(R,P). Active(P). Research(R,{}) (BT27)	CS(Q,{}).CExc(R,{}). Find(R) (BT29)
		non util.	CS(Q,{}).CExc({},P). Active(P). CExc({},{}) (BT28)	CS(Q,{}). CExc({},{}) (BT30)

FIG. 3.6 – Stratégies pour la propriété Between Q and R S precedes P.

R`··. Q		evénement	non utilisé
	P		
evenement	evt	CS({},Q). CS({,},P). CS({,},R) (BT30)	CS({},P). CS({},R) (BT32)
	prop	CS({},Q). CS(P,{}). CS(({,},R) (BT30)	CS(P,{}). CS({},R) (BT32)
	non util.	CS({},Q). CS({},R) (BT31)	CS({},R) (BT33)
propriété	evt	CS({},Q). CS({,},P). CS(R,{,}) (BT30)	CS({},P). CS(R,{}) (BT32)
	prop	CS({},Q). CS(P,{}). CS(R,{,}) (BT30)	CS(P,{}). CS(R,{}) (BT32)
	non util.	CS({},Q). CS(R,{}) (BT31)	CS(R,{}) (BT33)
non utilisé	evt	CSExact({},Q). CS({},P). CExc({},{}) (BT34)	

FIG. 3.7 – Stratégies pour la propriété Betwenn Q and R absence P.

(31) on teste une séquence avec un intervalle [Q,R]
(32) on teste une séquence finissant par un R avec un P
(33) on teste une séquence finissant par un R
(34) on teste une séquence avec un Q puis un P
Besoins de Tests limites :
Tester une séquence avec un intervalle vide (Q et R apparaissent simultanément).

Stratégie : **Cas nominal :** CoverStop(Q).CoverExcept(P,R).Active(R)
Construction d'une séquance avec un Q sans P ni R puis un R.
Voir la figure 3.3
Dans le tableau ne sont pas définies les stratégies quand R est une propriété. Pour les avoir, il suffit, à partir des stratégies avec R événement (avec Q et P de même nature), de remplacer les primitives *Active(R)* par *Find(R)* et dans les paramètres des primitives avec deux listes distinctes, de déplacer R dans la première liste. En effet la première liste étant celle des prédicats, si R est une propriété, il faut alors l'ôter de la liste des opérations *lo* pour le mettre dans la liste des prédicats *lp*.

Chapitre 4

Expérimentation

Afin de s'assurer que la démarche proposée dans ce rapport soit réalisable, nous avons développé un outil implémentant cette démarche, même si toutes les combinaisons portée/patrons n'ont pas été traitées dans ce rapport, et donc dans cet outil.

4.1 Implémentation d'un outil de production de scénarios

L'outil développé ne présente aucune interface graphique. Développé en Java sous Eclipse, il se présente sous le mode console. Toutefois, il a été conçu avec les mêmes restrictions que si une interface graphique était présente, c'est-à-dire que pour implémenter une interface graphique, il suffira de reprendre le code de la classe principale qui pilote le déroulement du programme, et les accès aux données seront les mêmes.

Alors que dans les objectifs du stage les besoins de tests étaient choisis en même temps que la propriété, la démarche que l'on a choisi impose de d'abord sélectionner le type de la propriété, c'est-à-dire la portée et le patron, et ensuite en fonction du couple, on calcule le nombre de paramètres nécessaires, et on les définit. On choisit d'abord leur type (prédicat ou événement) puis le nom. Pour les événements, on doit aussi définir si on souhaite que l'événement termine normalement (ok) ou non (ko), ou s'il suffit qu'il soit appelé (called). Il est important de saisir les noms exacts des opérations et des prédicats valides, car aucun contrôle n'est réalisé à ce niveau. Ce qui est normal, car il a été réalisé indépendamment de tout modèle pour le moment.

Bien que cet outil ne soit pas du tout optimisé, il est quand même conçu pour qu'on puisse facilement l'enrichir : alors que d'un point de vue pratique, pour un besoin de test, il n'existe que quelques schémas de test et qu'il aurait suffit de tester la nature des paramètres pour générer directement le schéma de test désiré, nous avons tout de même gardé dans l'implémentation la notion de stratégie et de primitive. Cela permet de rapidement construire de nouveaux besoins de test, car il suffira de lui lier une stratégie et de la définir par une suite de primitives.

4.2 Evaluation sur une étude de cas

Après avoir vérifié que la démarche était implémentable en développant un outil, nous avons voulu nous assurer de son efficacité, et donc de son utilité, pour la détection des erreurs, puis par rapport à d'autres outils existants. Nous présenterons d'abord le modèle que nous allons utilisé, puis la procédure utilisée et les résultats.

4.2.1 Le modèle Demoney

Le système considéré est inspiré de la spécification de Demoney (Demonstrative Electronic Purse), un porte-monnaie électronique développé par Trusted Logics à des fins de recherche. Aucune implantation de Demoney n'est embarquée sur une carte à puce utilisée au quotidien. Néanmoins, cette spécification comprend des propriété et un mode de fonctionnement très réaliste qui lui donne tout son intérêt. Demoney, comme tous les porte-monnaies, gère un solde qui évolue au gré des crédits/débits effectués sur la carte. Il est régi par deux codes PIN, l'un identifiant le porteur, l'autre identifiant la banque. Le solde du porte-monnaie est plafonné, tout comme le montant maximal d'un débit. Ce porte-monnaie fonctionne comme un carte à puce standard, notamment, il suit un cycle de vie. La vie de l'application commence par une phase de personnalisation permettant de fixer les codes PIN, et les plafonds autorisés de débit et du solde. Une fois la personnalisation correctement effectuée, la carte passe en phase d'utilisation. Durant cette phase, l'utilisateur peut utiliser le porte-monnaie en lui-même, c'est-à-dire payer une certaine somme d'argent ou créditer son solde. L'opération de crédit nécessite au préalable une authentification de l'utilisateur par l'intermédiaire de son code PIN. Lorsque l'utilisateur échoue toutes ses tentatives d'authentification, la carte est bloquée. Seule la banque peut la débloquer après s'être authentifiée avec son code PIN banque. Si la banque échoue toutes ses tentatives d'authentification, la carte est définitivement perdue. Si la carte est bloquée, la banque peut débloquer le code PIN utilisateur, en réinitialisant le code PIN utilisateur (compteur d'essais et valeur du PIN). La particularité de Demoney est son mode de fonctionnement "en deux temps", qui demande, dans un premier temps, d'initialiser une procédure (crédit, débit, paramétrage, etc.), puis, dans un second temps, d'appliquer cette procédure. Similairement à toutes les applications carte à puce, toutes les commandes peuvent toujours être invoquées.

Les opérations ne seront pas listées ni expliquées dans ce mémoire. Nous n'expliquerons que la signification informelle des propriétés que nous voulons tester.

4.2.2 Procédure de l'expérimentation

A partir d'une propriété, on applique la démarche proposée, ce qui donne un certain nombre de besoins de tests possibles, qu'on convertit en stratégies puis en schémas de tests. Les schémas de tests sont traduits en Sicstus - Prolog[1] pour que l'on puisse les exploiter par l'animation symbolique de BZ-TT [LPU02]. Lors de l'exécution, ils sont automatiquement dépliés par Prolog. Ils sont ensuite joués et instanciés et enregistrés dans un fichier au format BZTT/Prolog.

[1] http ://www.sics.se/sicstus

Les cas de tests sont ensuite lus et convertis en format XML, de manière à pouvoir réutiliser les traducteurs XML LTG vers JUnit déjà existants. JUnit est une API Java permettant de décrire des tests unitaires d'une application. Les cas de tests JUnit standards ainsi créés sont ensuite exécutés par la Machine Virtuelle Java sur l'implémentation de Demoney.

Les résultats sont comparés à ceux d'une suite de tests générée automatiquement par LTG

Des mutants de l'implémentation ont été créés manuellement. Pour chacune des propriétés, il y a six mutants spécifiques à la propriété, et l'objectif est de pouvoir les détecter les six.

4.2.3 Propriétés testées

Nous avons testé trois propriétés. Nous ne décrirons les besoins de tests, stratégies et schémas de tests uniquement pour la première propriété.

Première propriété *BETWEEN store_data ok AND commit_transaction ok EXISTENCE initialize_transaction ok*

Cette propriété signifie qu'une fois les données enregistrées, il existe au moins une initialisation de transaction avant sa validation, c'est-à-dire qu'entre l'appel réussi à *store_data* et celui réussi de *commit_transaction*, il existe au moins un appel réussi de *initialize transaction*.

Il a quatre besoins de tests possibles, mais seulement deux seront utilisés :

- (BT1) on teste une séquence avec un *store_data ok* suivi d'une suite sans *initilize_transition ok* finissant par *commit_transaction ok*
- (BT2) on teste une séquence avec un *store_data ok* suivi d'une séquence sans *commit_transaction ok* suivie d'un *commit_transaction ok*

La traduction en stratégie donne :

- BT1 : CoverStop({},*store_data ok*). CoverExcept({,},*initialize_transition ok*). Active(*commit_transition ok*)
- BT2 : CoverStop({},*store_data ok*). CoverStopExact({},*commit_transition ok*)

La traduction en schémas de tests donne :

- BT1 : $[\$OP\ /w\ \{ok\}]^{0,4}$. $[\$OP]$. $[STORE\_DATA\ /w\ \{ok\}]$. $[\$OP\ /w\ \{ok\}\ \backslash\{INITIALIZE\_TRANSACTION\}]^{0,4}$. $[\$OP\backslash\{INTIALIZE\_TRANSACTION\}]$. $[COMMIT\_TRANSACTION]$
- BT2 : $[\$OP\ /w\ \{ok\}]^{0,4}$. $[\$OP]$. $[STORE\_DATA\ /w\ \{ok\}]$. $[\$OP\ /w\ \{ok\}\ \backslash\{COMMIT\_TRANSACTION]^{0,4}\}]$. $[\$OP\backslash\{COMMIT\_TRANSACTION\}]$. $[COMMIT\_TRANSACTION\ /w\ \{ok\}]$

Le cas nominal est défini par la stratégie
CoverStop({},*store_data ok*).
CoverStop({,},*init_transition ok*).
CoverStopExact(*commit_transition ok*)

et par le schéma : $[\$OP\ /w\ \{ok\}]^{0,4}$. $[\$OP]?$. $[STORE\_DATA\ /w\ \{ok\}]$. $[\$OP\ /w\ \{ok\}]^{0,4}$. $[\$OP]?$. $[INITIALIZE\_TRANSACTION\ /w\ \{ok\}\ ]$.

$[\$OP\ /w\ \{ok\}\ ]^{0,4}\ .\ [\$OP]?\ .\ [COMMIT\_TRANSACTION\ /w\ \{ok\}]$

Deuxième propriété *BEFORE store_data ok*
ABSENCE (initialize_transaction/verify_pin) ok

Cette propriété signifie qu'il est impossible d'avoir une initialisation de transaction ou une vérification de PIN réussissant avant d'avoir enregistré les données, c'est-à-dire qu'avant l'appel réussi à *store_data* il est impossible d'appeler avec succès *initialize_transaction* et *verify PIN*. Toutes les combinaisons patrons/portée du système de Dwyer ne sont pas détaillée dans ce rapport, mais toutes sont traitables par cette démarche, comme cette propriété dont les stratégies ne sont pas présentées dans ce rapport.

Il a trois besoins de tests possibles :

- (BT1) on teste une séquence avec un *initialize_transaction ok* ou un *verify PIN* suivi d'un *store_data ok*
- (BT2) on teste une séquence avec un *initialize_transaction ok* ou un *verify PIN*
- (BT3) on teste une séquence finissant par un *store_data ok*

Troisième propriété *BETWEEN store_data ok AND mode = invalid*
ABSENCE (verify_pin bank) ok

Cette propriété signifie qu'il est impossible de véifier le code PIN vers la banque une fois les données enregistrées et en étant en mode valide, c'est-à-dire qu'après un appel réussi à *store_data* et être dans un état de mode=invalid, il est impossible d'appeler avec succès *verify_pin_bank*.

Il a quatre besoins de tests possibles, mais un seul est utilisé :

- (BT1) on teste une séquence ayant un intervalle [*store_data*,*mode = invalid*]

4.2.4 Résultats

Les résultats sont présentés sous forme de deux tableaux par propriété :

- le premier présentant le nombre de tests, la taille moyenne d'un jeu (en opération) et le temps pour générer la suite
- le second présentant pour les six mutants le verdict pour chaque suite

OK signifie que le mutant n'a pas été repéré, donc que la suite de tests n'a pas permis de détecter l'erreur. KO signifie que le mutant a été tué, c'est-à-dire repéré et donc que la suite a révélé l'erreur.

Résultat pour la première propriété On remarque dans le tableau 4.1 que la longueur moyenne des tests créés par notre démarche est plus longue que celle des tests créés par LTG, ce qui indiquerait que les tests sont plus complexes, tout en étant dans le même ordre de grandeur de temps pour la génération. Dans le tableau 4.2, on remarque que la suite correspondant aux cas nominaux avec celle correspondant au besoin de test BT1 permettent de révéler tous les mutants, contrairement à la suite générée par LTG. Les besoins de tests n'ont pas pour objectif d'être tous traités successivement, mais ils sont tout de même complémentaires. Cela permet de confirmer l'intérêt de besoins de tests *a priori* déjà couvert, mais qui, non seulement permettent d'ores et déjà de trouver des erreurs spécifiques, mais en plus laissent une plus grande liberté à l'ingénieur pour tester

des situations plus complexes (en utilisant des filtres adéquats). Si on se contentait de ne proposer que les besoins de tests les plus précis, on priverait l'ingénieur de la possibilité de créeer des schémas plus généraux filtrés ensuite pour obtenir une caractéristique impossible à représenter dans la propriété.

Type de test	Nombre de tests	Taille moyenne	Temps de génération
cas nominaux	544 tests	12.8	171s
cas BT1	419 tests	9.2	64s
cas BT2	11 tests	9.3	9s
LTG	59 tests	7.6	$<$ 3min

FIG. 4.1 – Statistique pour la première propriété

Mutants	M1	M2	M3	M4	M5	M6	Nb Tués
Cas nom.	KO	KO	OK	OK	OK	KO	3/6
Cas BT1	OK	OK	KO	KO	KO	OK	3/6
Cas BT2	OK	KO	OK	OK	OK	KO	2/6
Cas BT3	KO	KO	OK	OK	OK	KO	3/6
LTG	OK	KO	OK	OK	KO	KO	3/6
Nb de fois tué	2	4	1	1	2	4	

FIG. 4.2 – Résultats pour la première propriété

Résultat pour la deuxième propriété On remarquera encore dans le tableau 4.3 que les tests de LTG sont moins longs en moyenne. Le cas BT1 n'est pas instanciable : il correspond au besoin de tests visant à générer une séquence qui ne vérifie pas la propriété, et donc on ne peut pas générer de tests à partir du modèle, car le modèle vérifiant la propriété, on ne peut pas animer ce schéma. Si un test avait été instanciable, cela signifiait que le modèle lui-même violait la propriété. Les besoins de tests et cas nominaux ont présenté une explosion combinatoire des états, provoquant un temps de génération plus long, voire trop long : les calculs de tests pour le cas BT3 ont été arrêtés àprès une heure de calcul. Dans le tableau 4.4, on voit que les cas de tests, même celui arrêté avant la fin du calcul, détectent plus d'erreurs que les tests de LTG.

Type de test	Nombre de tests	Taille moyenne	Temps de génération
cas nominaux	576 tests	5.9	313s
cas BT1	0 tests	0	179s
cas BT2	1931 tests	9.2	275s
cas BT3	112 tests	6.9	time out
Test LTG	44 tests	1.9	$<$ 3min

FIG. 4.3 – Statistique pour la deuxième propriété

Mutants	M1	M2	M3	M4	M5	M6	Nb Tues
Cas nom.	KO	KO	OK	OK	OK	KO	3/6
Cas BT1							0/6
Cas BT2	KO	KO	KO	KO	OK	KO	5/6
Cas BT3	KO	OK	KO	KO	KO	KO	5/6
LTG	OK	OK	OK	OK	OK	KO	1/6
Nb de fois tué	3	2	2	2	1	4	

FIG. 4.4 – Résultats pour la deuxième propriété

Résultats pour la troisième propriété La lecture des tableaux 4.5 et 4.6 démontrent les limites de la démarche : pour des propriétés simples, ou des erreurs "faciles", elle produit plus de tests que LTG, et les besoins de tests sont redondants avec les cas nominaux. Cela confirme l'intuition que cette démarche s'applique avec parcimonie, pour des cas précis et non systématiquement pour tous les cas avec tous les besoins de tests. Les bons résultats de LTG s'expliquent par le fait que la propriété concernce les états où la carte à puce est en phase d'utilisation, et que LTG réalise une bonne couverture de ces configurations : il détecte donc tous les mutants, avec moins de tests.

Type de test	Nombre de tests	Taille moyenne	Temps de génération
cas nominaux	96 tests	9.4	109s
cas BT1	302 tests	11.3	179s
Test LTG	59 tests	7.6	$<$ 3min

FIG. 4.5 – Statistique pour la troisième propriété

Mutants	M1	M2	M3	M4	M5	M6	Nb Tues
Cas nom.	KO	KO	KO	KO	KO	KO	6/6
Cas BT1	KO	KO	KO	KO	KO	KO	6/6
LTG	KO	KO	KO	KO	KO	KO	6/6
Nb de fois tué	3	3	3	3	3	3	

FIG. 4.6 – Résultats pour la troisième propriété

Synthèse des résultats Les résultats sont encourageants : les tests générés par cette démarche ont permis de détecter des erreurs que LTG n'avait pas trouvées. Les tests sont en moyenne plus longs et surtout plus nombreux que pour LTG : ce sont deux conséquences normales et espèrées de la démarche, car on souhaite tester plus de configurations plus complexes.

Il apparaît aussi une conséquence attendue de la démarche : il est inutile d'appliquer tous les besoins de tests de toutes les propriétés de manière systématique. Cette démarche se fait pour des cas que l'on a ciblé, des cas que l'ingénieur sait ne pas avoir été testé par un logiciel comme LTG.

Conclusion

L'objectif de ce stage était d'étudier la possibilité de générer automatiquement des scénarios à partir d'une propriété de sûreté et d'un besoin de test. Bien sûr, même si nous étions finalement libre de la manière de le faire, libre de choisir les formats qu'il nous semblait utile de prendre, il fallait quand même que ce soit intégrable au projet POSE qui propose une approche spécifique pour la validation des politiques de sécurité. Cette approche utilise différents outils, et les travaux réalisés et présentés dans ce rapport devaient pouvoir s'intégrer dans cet ensemble, même si certains éléments étaient modifiables si nécessaire (et si possible), comme des formats internes (langage des schémas). Cela dit, l'objectif étant de remplacer une écriture manuelle des scénarios qui sont à la base du processus, cela signifie que nous étions au début de la chaîne, et donc nous n'avions aucune contrainte sur le format d'entrée, c'est-à-dire celui de l'expression des propriétés. Durant tout le stage, nous nous sommes efforcés de garder à l'esprit que cela serait utilisé par un ingénieur sécurité, et à chaque étape, nous nous mettions à la place de l'utilisateur pour savoir ce dont il aurait besoin, et comment il voudrait le faire.

Nous nous sommes principalement inspirés de deux travaux, ceux de Dwyer sur l'expression des propriétés [DAC97] et ceux réalisés à l'intéreur du LIFC sur la génération de tests guidée par une propriété de sûreté à partir de spécification JML [BDJG06] qui nous ont servi de points de départ : point de départ du processus pour l'expression des propriétés, et point de départ de réflexion pour les stratégies relatives aux propriétés. A partir de l'analyse de ces travaux, nous avons pu définir ce qui nous manquait pour pouvoir faire des choses similaires dans notre cadre de travail, puis définir précisément la démarche en entier, depuis la saisie de la propriété jusqu'à la création du schéma de test.

Après avoir défini la démarche et détaillé les stratégies de certaines propriétés, nous avons développé un outil permettant de s'assurer de la faisabilité en pratique des objectifs du stage. Même s'il n'a pas pour vocation d'être complet, il a permis de gérer différentes propriétés, avec des paramètres de nature différente, et utilisant toutes les primitives. Comme il n'y a eu aucun problème pour le faire, il y a fort a parier qu'il sera possible de gérer toutes les combinaisons patron/portée et les besoins de test associés.

Parmi les perspectives envisagées, le mécanisme des filtres est le plus important : il fait partie de la démarche, mais n'a pas été developpé ici et devra être implémenté dans l'outil générant les tests, que ce soit pendant la génération (plus efficace) ou après la génération (plus complet). Les filtres représentent en effet le mécanisme permettant d'exprimer des parties d'un besoin de test sans devoir réécrire la propriété. Sur une propriété dans laquelle une opération n'apparaît pas syntaxiquement, il est impossible de prévoir un besoin de test

prenant en compte cette opération, mais par un filtre, indépendant de la propriété, cela permettra de prendre en compte cette opération, que ce soit pour imposer sa présence, son absence, un nombre précis de fois ou minimal d'apparition. Et la possibilité de cumuler plusieurs filtres permettraient à partir d'une suite de test qui se veut exhaustive, mais peut être trop générale si le besoin de test est très particulier, d'éliminer des tests que l'ingénieur estime de par son expérience n'avoir aucune utilité. Les filtres seraient des algorithmes prenant en entrée une liste de tests et rendant une liste de tests. La plupart des filtres sont algorithmiquement très simples, mais l'élaboration de filtres plus complexes peut aussi être très intéressant dans l'optique de pouvoir encore couvrir potentiellement plus de besoins de test.

Il apparaît nécessaire aussi d'expérimenter cette démarche sur d'autres études de cas. L'expérimentation effectuée pendant le stage portait sur un modèle qui a souvent servi d'exemple, que ce soit pour les propriétés ou les besoins, et donc sur la conception même des stratégies, et donc il est normal que la démarche s'applique parfaitement à ce modèle, voire à tout modèle similaire. Il serait donc intéressant de pouvoir expérimenter la démarche sur des projets plus variés.

Une perspective possible est d'utiliser la démarche pour tester le modèle. Lors de l'élaboration des besoins de tests, nous avons remarqué que certains n'étaient pas utilisable pour tester l'implémentation, malgré l'existence de stratégies et de scénario pour le faire. Mais le modèle vérifiant la propriété ne peut par définition être animé de manière à violer une propriété, et donc générer les tests. Mais il est alors possible de tester le modèle lui-même en utilisant des besoins de tests générant des scénations ne pouvant pas être instanciable si le modèle est correct vis-à-vis de la propritété. La génération de tests à partir d'un tel besoin de tests démontrerait que le modèle est incorrect. Cela nécessiterait une étude plus approfondie pour fournir plus de tels besoins de tests.

La dernière perspective est évidement le developpement d'un outil plus évolué et complet permettant de gèrer tous les cas prévus par le système de Dwyer, avec une interface graphique suivant l'utilisation normale d'un ingénieur. La démarche a été conçue dans l'optique de developpement d'un outil, et donc est facilement adaptable.

Les objectifs de ces travaux ont donc été atteints, mais il reste encore quelques expérimentations à faire et compléments à ajouter pour pouvoir perfectionner encore cette démarche de génération de scénarios.

Bibliographie

[BDJG06] F. Bouquet, F. Dadeau, J. Julliand, and J. Groslambert. . In *Proceedings of the International Workshop on Formal Approaches in the Testing of Software/Runtime Verification (FATES/RV'06)*, LNCS, Seattle, USA, August 2006. Springer-Verlag.

[Bei95] Boris Beizer. *Black-box testing : techniques for functional testing of software and systems.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[BGS+03] Mike Barnett, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Validating use-cases with the asml test tool. In *QSIC '03 : Proceedings of the Third International Conference on Quality Software*, page 238, Washington, DC, USA, 2003. IEEE Computer Society.

[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.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.

[CGN+05] Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Testing concurrent object-oriented systems with spec explorer. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 542–547. Springer, 2005.

[CJRZ02] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Stg : a symbolic test generation tool. In *(Tool paper) Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02). Volume 2280 of LNCS*. Springer-Verlag, 2002.

[CLP04] S. Colin, B. Legeard, and F. Peureux. Preamble computation in automated test case generation using Constraint Logic Programming. *The Journal of Software Testing, Verification and Reliability*, 14(3) :213–235, 2004. Selected papers from the 2003 UK-Test Workshop.

[DAC97] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. A system of specification patterns. In *http ://patterns.projects.cis.ksu.edu/*, 1997.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[FBL06] F. Dadeau F. Bouquet and B. Legeard. Automated boundary test generation from jml specifications. *LNCS*, 4085 :428–443, 2006.

[FJJV96] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV '96 : Proceedings of the 8th International Conference on Computer Aided Verification*, pages 348–359, London, UK, 1996. Springer-Verlag.

[GG06] A. Georgetti and J. Groslambert. Jag : Jml annotation generation for verifying temporal properties. *LNCS*, 3922 :262–284, 2006.

[GIX04] GIXEL. Common ias platform for eadministration, technical specifications, 1.01 premium edition. In *http ://www.gixel.fr*, 2004.

[Hol91] G. Holzmann. *Design and validation of protocols*. Prentice-Hall Software Series, 1991.

[JJ05] Claude Jard and Thierry Jéron. Tgv : theory, principles and algorithms : A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4) :297–315, 2005.

[JL07] E. Jaffuel and B. Legeard. LEIRIOS Test Generator : Automated test generation from B models. In *B'2007, the 7th Int. B Conference - Industrial Tool Session*, volume 4355 of *LNCS*, pages 277–280, Besancon, France, January 2007. Springer.

[JMT08] Jacques Julliand, Pierre-Alain Masson, and Regis Tissot. Generating security tests in addition to functional tests. In *AST '08 : Proceedings of the 3rd international workshop on Automation of software test*, pages 41–44, New York, NY, USA, 2008. ACM.

[LB03] M. Leuschel and M. Butler. . In *Prob : A model-checker for B*, LNCS. Springer-Verlag, August 2003.

[LDdB+07] Yves Ledru, Frédéric Dadeau, Lydie du Bousquet, Sébastien Ville, and Elodie Rose. Mastering combinatorial explosion with the tobias-2 test generator. In *ASE '07 : Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 535–536, New York, NY, USA, 2007. ACM.

[LPU02] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proc. of the Int. Conf. on Formal Methods Europe, FME'02*, volume 2391 of *LNCS*, pages 21–40, Copenhaguen, Denmark, July 2002. Springer.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking : an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, Pittesburg, PA, USA, 1992.

[Mye79] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[OG02] Martijn Oostdijk and Herman Geuvers. Proof by computation in the coq system. *Theor. Comput. Sci.*, 272(1-2) :293–314, 2002.

[OO90] K.M. Olender and L.J. Osterweil. Cecil : A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3) :268–280, 1990.

[Par00] B. Parreaux. *Vérification de systèmes d'événements B par model-checking PLTL*. PhD thesis, Université de Franche-comté, 2000.

[Sie02] S.F. Siegel. The inca query language. *Technical Report CMPSCI TR 02-18*, 2002.

[TH02] K. Trentelman and M. Huisman. Extending jml specifications with temporal logic. *LNCS*, 2422 :334–348, 2002.

[TM01] Paul Strooer Tim Miller. Animation can show only the presence of errors never their absence. In *ASWEC '01 : Proceedings of the 13th Australian Conference on Software Engineering*, page 76, 2001.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier Science, 2006. 550 pages, ISBN 0-12-372501-1.

[YFR08] Jean-Claude Fernandez Ylies Falcone, Laurent Mounier and Jean-Luc Richier. j-post : a java toolchain for property-oriented software testing. Technical report, Verimag, Centre Équation, 38610 Gières, April 2008.

Table des figures

Résumé

Les travaux présentés dans ce mémoire se placent dans le cadre de la génération de tests à partir de modèles formels *Model-Based Testing*, plus précisément celui de la génération de tests de propriété à partir de scénario. Pour tester de manière plus précise une propriété, il peut être intéressant d'utiliser l'expérience de l'ingénieur à propos du système pour tester la propriété dans des situations précises. Ces situations sont appelées *besoins de test*. L'objectif des travaux est de présenter une démarche permettant de générer automatiquement des scénarios à partir d'une propriété et d'un besoin de test.

La démarche proposée dans ce mémoire est la suivante : à partir d'une propriété exprimée selon la spécification de Dwyer, et d'un besoin de tests, la stratégie est définie, c'est-à-dire une séquence de primitives, chaque primitive est traduite en partie de schéma. Nous avons donc généré automatiquement un scénario permettant de tester le besoin de test.

Une implémentation d'un outil générateur de scénarios a permis de s'assure de la faisabilité de la démarche, et l'expérimentation a donné des résultats très encourageants, en trouvant sur un cas d'étude des erreurs non détectées par un outil comme LTG. La suite de tests générée par la démarche est plus importante celle de LTG, et les tests sont en moyenne plus longs que ceux de LTG.