N° d'ordre : 3058

Thèse

présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : **Docteur de l'Université de Rennes 1**

Mention *INFORMATIQUE*

par

Clémentine Nebut

Équipe d'accueil :	*Triskell*
École doctorale :	*Mathématiques, Informatique, Signal, Électronique, Télécommunications (MATISSE)*
Composante universitaire :	*Institut de Formation Supérieure en Informatique et Communication (IFSIC)*

Génération automatique de tests à partir des exigences et application aux lignes de produits logicielles

soutenue le ? ? ? 2004 devant la commission d'examen :

Composition du Jury

Président :	? ? ?
Rapporteurs :	Mme Antonia Bertolino M. Lionel Briand M. Jean-Marc Geib
Examinateurs :	M. Daniel Herman M. Jean-Marc Jézéquel M. Yves Le Traon

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Première partie

Introduction

Depuis la création des premiers langages de programmation, la taille des logiciels développés croît de manière exponentielle. Si les « grosses applications » comme les compilateurs étaient jadis constituées d'environ 10000 lignes de code, actuellement il n'est pas rare de trouver des applications d'une taille de l'ordre de 10^8 lignes de code. De plus, on ne parle plus seulement de logiciels, mais aussi de lignes de produits logiciels : de plus en plus souvent, les fournisseurs de logiciels proposent une gamme de produits plutôt qu'un produit unique – les différents produits de la gamme varient par les fonctionnalités offertes (versions limitées ou complètes), par la qualité de service assurée, par la plate-forme supportée, etc. Bien évidemment, ce changement d'échelle de la taille des logiciels a provoqué de gros changements dans les techniques et les processus utilisés pour les réaliser. Ainsi, les langages de programmation ont par exemple évolué vers la programmation par objets, et des pratiques de modélisation des applications sont apparues, allant d'un niveau de formalisme élevé comme la méthode B à un niveau de formalisme faible comme OMT, en passant par des intermédiaires comme Merise ou UML.

La croissance de la complexité des programmes ainsi que la modification du mode de construction des logiciels impactent évidemment leur validation. Ainsi, l'apparition du paradigme objet a donné lieu à de nouvelles techniques de test [Binder, 2000a], et le développement pour la conception de logiciels orientés objets de standards méthodologiques (comme le Rationnal Unified Process [Kruchten, 2000], la conception par contrats [Meyer, 1992], ou Catalysis [D'Souza and Wills, 1999]) ou de notations (comme UML [OMG, 2003b, OMG - U2 partners, 2001]) est prise en compte par les techniques de test [Briand and Labiche, 2002, Lugato et al., 2002, Offutt and Abdurazik, 1999, Abdurazik and Offutt, 1999, Kim et al., 1999]. En effet, le test s'est affirmé comme le principal moyen pour valider le fonctionnement d'un programme ; le test consiste à exécuter le programme (ou à l'observer dans le cas du test statique) afin d'y détecter des erreurs. L'espace des comportements d'un programme étant en général infini, il n'est pas envisageable de générer des tests permettant d'exhiber l'ensemble des comportements du programme sous test (en d'autres termes, le test « exhaustif » n'a en général pas de sens). De ce fait, une problématique majeure du test est de générer un minimum de tests pertinents, permettant de détecter un maximum de fautes : il ne s'agit pas de générer un grand nombre de tests pour couvrir un maximum de comportements, mais plutôt de générer des tests efficaces, c'est-à-dire les plus susceptibles de détecter des erreurs.

La tâche de test est en général critique dans le cycle de vie d'un logiciel, et la mener à bien nécessite un investissement en temps très important. Ce fort coût de la phase de test provient principalement de son aspect manuel : dans un contexte industriel, les tests sont en général générés à la main par les testeurs. La taille des logiciels à tester devenant extrêmement grande, l'automatisation de la tâche de test est donc devenue une problématique majeure pour le test de logiciels. L'automatisation est également primordiale dans le cadre des lignes de produits. En effet, dans la mesure où l'objectif des lignes de produits est un gain en temps, en coût et en qualité, l'approche des lignes de produits conduit naturellement à rechercher une factorisation de la tâche de test, et si possible son automatisation. L'automatisation de la tâche de test peut se faire en prenant comme entrée le programme lui même (dans le cas du test structurel)

ou bien une modélisation du programme. Ainsi, pour la validation fonctionnelle – qui cherche à vérifier que le programme fournit correctement les fonctionnalités requises – l'automatisation la plus répandue utilise une modélisation des besoins pour générer des tests fonctionnels.

L'utilisation de modèles très formels des exigences d'un logiciel comme point d'entrée pour la génération de test et plus généralement la validation du logiciel a fait ses preuves depuis les années 80. Ainsi, les techniques de *model-checking* (l'étude de la modélisation d'un programme en vue de l'analyse de l'accessibilité de tous ses comportements possibles) se sont avérées très efficaces pour l'établissement de propriétés des logiciels. De plus, une bonne formalisation des exigences permet la construction d'un graphe ou d'un automate représentant le comportement souhaité du logiciel. Il est alors possible d'extraire, à l'aide de critères de couverture, de cette modélisation du comportement souhaité, des chemins représentant les comportements que l'on souhaite observer sur le logiciel. Ainsi, les méthodes formelles et les outillages du type model-checking et couverture de graphe comportemental semblent résoudre le problème de la génération automatique de test fonctionnel.

Cependant, le prix à payer pour l'application de ces méthodes est fort [Chechik and Wong, 1999, Stidolph and Whitehead, 2003, Ryser et al., 1998] : la modélisation formelle d'un gros logiciel est extrêmement coûteuse. Si le coût de modélisation avec des langages formels est très élevé pour un produit unique, il devient rédhibitoire dans le cadre de lignes de produits logicielles. Une approche ligne de produits logicielles implique le développement d'une architecture visant à pouvoir dériver rapidement de nouveaux produits fiables. Le coût de la génération de test est alors particulièrement important. En effet, soit une ligne de n produits de même taille p, partageant 90% de leur code et ayant chacun 10% de code spécifique. Si on adopte pour chacun des produits une méthode de génération de test classique, on a alors un coût de test linéaire en $n \times p$, en considérant le coût du test linéaire en la taille du logiciel sous test. Dans l'idéal, on souhaite tester de la même façon les parties communes de la ligne, et ne générer des tests spécifiques que pour les parties variantes ; le coût de test souhaité pour cette même ligne de produits est donc bien moindre : $p \times \frac{90}{100} + n \times p \times \frac{10}{100}$. Dans un contexte où le nombre de produits est grand (Nokia estime par exemple que certains de ses logiciels présents sur ses téléphones portables se déclinent en 10 000 versions différentes), le coût de test « classique » devient prohibitif. Tous les produits d'une même ligne ont des spécifications différentes, et l'utilisation de méthodes formelles dans ce cadre nécessite de reformaliser les spécifications de chaque produit, ce qui est économiquement irréaliste car long et coûteux. Actuellement, il n'existe en effet pas d'approche permettant de capitaliser un ensemble d'exigences communes à l'ensemble de la ligne de produits, et de spécifier quelles parties sont variables. Les méthodes formelles se prêtent cependant difficilement à ce genre de pratique : contrairement aux méthodes objets, elles n'ont pas été conçues pour gérer facilement les extensions et modifications.

Si les méthodes formelles peuvent donc être utilisées dans certains cas pour réduire le coût de la génération de test [Helke et al., 1997, Legeard et al., 2002, Tahat et al., 2001, Gargantini and Heitmeyer, 1999], la formalisation complète de grands systèmes a un coût que beaucoup d'entreprises ne sont pas prêtes à payer. Les problèmes sont multiples et récurrents : difficulté de trouver des personnes compétentes

pour la rédaction d'exigences formelles, interactions délicates avec le client à cause du peu de lisibilité de telles spécifications et manque d'intégration des méthodes formelles avec les cycles de vie classiques du développement [Le Guennec, 2000]. Du fait des changements constants dans les exigences, émergent des problèmes de maintenabilité des spécifications formelles et de traçabilité. La démarche que nous avons suivie dans nos recherches a donc été de partir des pratiques de conception et de développement bien établies de l'industrie, et de les amener progressivement vers des modèles formellement exploitables, afin de les faire bénéficier de méthodes de génération automatique de test et ainsi de réduire le coût de test.

Ainsi, la thèse que nous défendons dans ce manuscrit est qu'il est possible de faire baisser le coût de la génération de test en utilisant une approche dont les points d'entrée sont proches de ceux actuellement pratiqués dans l'industrie, mais se basant sur un formalisme permettant la génération automatique de tests efficaces.

Pour pouvoir être applicable par l'industrie, une méthode de génération de test ne doit pas aller à l'encontre des pratiques de spécification et de modélisation utilisées actuellement. Les spécifications d'un logiciel sont le plus souvent exprimées sous forme de texte. Ces spécifications ont l'avantage de pouvoir être facilement rédigées et être comprises par le plus grand nombre. En revanche, elles sont ambiguës, et peuvent donc prêter à de multiples interprétations. De plus, elles ne se prêtent pas à un traitement automatique comme la génération de test. Un certain nombre de recherches [Ambriola and Gervasi, 1997, Fantechi et al., 1994, Fuchs et al., 1999a] ont porté sur la traduction automatique du langage naturel des spécifications vers un modèle formel. Ces approches sont intéressantes puisque dans l'idéal, elles permettent de bénéficier des avantages du langage naturel ainsi que de ceux des langages formels. Nous avons donc cherché à nous en inspirer, en générant un modèle formalisé à partir de spécifications en langage contrôlé. Le modèle sur lequel nous nous appuyons est une extension des cas d'utilisation UML (UML étant devenu un standard de notation très utilisé dans l'industrie). Notre modèle, sans doute plus simple que la plupart des modèles formels, permet une maîtrise de la complexité des logiciels traités, et une plus grande facilité dans la maintenance. Plus souple, il est adaptable au contexte des lignes de produits, et la dérivation des spécifications d'un nouveau produit est rapide et peu coûteuse. Nous montrons dans ce manuscrit que de tels mécanismes de spécifications, bien que très simples, sont suffisants pour générer des tests système efficaces, et facilement gérables.

La contribution présentée dans ce manuscrit est une approche de génération automatique de tests fonctionnels, à partir des exigences d'un système, prenant en compte l'ensemble des contraintes citées précédemment : la maîtrise du coût de test, l'adaptabilité au contexte des lignes de produits, la compatibilité avec les pratiques industrielles courantes et la complexité des logiciels réels.

Notre approche est fondée sur un modèle de cas d'utilisation étendus, autour duquel s'articulent un analyseur de langage naturel contrôlé en amont et un générateur de test en aval.

En UML, les cas d'utilisation représentent les grandes fonctionnalités du système, et sont reliés aux acteurs qui interagissent avec eux. Pour décrire un cas d'utilisation, UML propose d'y associer une vue comportementale comme un diagramme de séquence, une

machine à états ou un diagramme d'activités. Nous proposons d'enrichir chaque cas d'utilisation avec des paramètres, une pré-condition et une post-condition exprimant respectivement quels sont les pré-requis pour l'application du cas d'utilisation, et quelles en sont les conséquences. Si la norme UML ne prévoit pas d'attacher ces contraintes aux cas d'utilisation (elles sont réservées aux méthodes en UML 1.4 ainsi qu'aux transitions de machine à état en UML 2.0), la méthode Catalysis [D'Souza and Wills, 1999] recommande quant à elle de voir les cas d'utilisation comme des types particuliers d'action, possédant donc des pré et des postconditions. Nous reprenons cette idée en proposant un langage d'expression de ces contrats, en vue de pouvoir les exploiter automatiquement pour la génération de tests.

Un tel modèle de cas d'utilisation peut être obtenu par analyse d'un langage naturel contrôlé. Nous proposons dans ce manuscrit un processus incrémental de rédaction des exigences basé sur un tel langage : au bout de ce processus mélioratif, les exigences peuvent être analysées, interprétées et transformées en un modèle de cas d'utilisation tel que nous l'avons défini.

Le modèle de cas d'utilisation est la base d'un modèle de simulation qui permet de simuler les cas d'utilisation, et donc les exigences. Nous utilisons le principe de simulation pour la génération de test : une simulation exhaustive permet de produire un graphe comportemental abstrait du système, duquel nous extrayons automatiquement un ensemble de chemins sélectionnés par des critères de test que nous avons défini. De tels chemins dans le graphe comportemental forment des *objectifs de test*.

Ces objectifs de test sont de haut niveau : pour pouvoir les transformer en cas de test, nous utilisons les scénarios attachés aux cas d'utilisation, qui permettent de passer d'un niveau d'abstraction élevé – celui des spécifications – vers un niveau de modélisation plus bas prenant en compte les interfaces réelles du système. Dans le cas des systèmes déterministes, la simple utilisation des scénarios dans les objectifs de test est suffisante pour obtenir des cas de test ; dans des cas plus complexes, par exemple des systèmes exhibant du non-déterminisme ou de la concurrence interne, nous proposons le recours à des outils de synthèse de test [Nebut et al., 2003c, Nebut et al., 2002].

Globalement, l'approche que nous proposons permet de générer des cas de test à partir des spécifications d'un système. Elle se décompose en trois grandes phases : la première est consacrée à la rédaction des exigences, à leur analyse, et leur interprétation en termes de cas d'utilisation ; la deuxième vise à générer des objectifs de test pertinents à partir des cas d'utilisation ; et la troisième vise à dériver chaque objectif de test en cas de test directement exécutables par un pilote de test. La validation de l'approche est double : académique et industrielle. La validation académique repose sur 3 études de cas concernant l'efficacité de l'approche, des critères de test proposés, et des tests générés. La validation industrielle s'est effectuée dans le cadre d'un projet coopératif avec Thalès Airborne System (TAS). Nous avons pu disposer de composants d'avionique embarquée développés par TAS afin de tester nos prototypes sur un de ces composants, et TAS applique notre approche sur un autre composant. Notre méthode ayant été développée dans le contexte des lignes de produits, elle y a ensuite été appliquée [Nebut et al., 2003a, Nebut et al., 2002].

Ce manuscrit est structuré comme suit.

La partie II présente un état de l'art en trois chapitres : le premier donne un aperçu des principaux concepts du test de logiciel, le deuxième détaille la génération automatique de test, et le troisième expose les problématiques spécifiques aux lignes de produits et aux méthodes de test associées.

La partie III détaille les contributions de ce manuscrit, et se compose de 6 chapitres. Le chapitre 4 donne une vue globale de l'approche proposée, et de son contexte dans lequel cette approche a été proposée. Il permet de mieux distinguer les 3 parties de l'approche, afin que chacune d'entre elles puisse ensuite faire l'objet d'un chapitre. Le chapitre 5 détaille la première phase de notre approche, c'est à dire la rédaction, l'analyse et la transformation des exigences. Le chapitre 6 présente la modélisation des cas d'utilisation que nous proposons, explicite notre modèle de simulation, et présente le mécanisme de génération d'objectifs de test. Le chapitre 7 décrit notre méthode pour dériver de tels objectifs de test en cas de test. Le chapitre 8 présente la validation expérimentale de notre approche, et enfin le chapitre 9 présente son application aux lignes de produits logicielles.

La partie IV conclut et présente nos perspectives quant à la poursuite des travaux présentés dans ce manuscrit.

Deuxième partie

État de l'art

Chapitre 1

Le test de logiciels

Dans ce premier chapitre, nous présentons brièvement les principales problématiques du test de logiciel : quels sont ses objectifs, quelles sont les grandes classes de méthodes de test, quelles sont les différentes phases de test. Nous insistons plus particulièrement sur les travaux relatifs au test de logiciels modélisés avec UML. Les deux chapitres suivants approfondissent les domaines relatifs aux contributions de ce manuscrit, c'est-à-dire la génération automatique de test, et le test de lignes de produits.

L'objectif du test de logiciel est de détecter des fautes dans un programme. On parle aussi de détection d'erreurs ou de défaillances ; ces termes ont été précisément définis dans [Xanthakis et al., 2000, Laprie, 1995]. Dans la suite de ce manuscrit, nous utiliserons indifféremment les termes de faute et d'erreur. Il existe deux grandes classes de tests : le test statique et le test dynamique. Le test dynamique cherche à détecter des fautes dans un programme en l'exécutant, tandis que le test statique cherche à y détecter des fautes sans l'exécuter. Dans la suite de ce manuscrit, nous ne nous intéressons qu'au test dynamique.

Le test dynamique de logiciel consiste à sélectionner un jeu de données à fournir en entrée du programme, et à déterminer quelles sont les sorties attendues du programme. Ensuite, on exécute le programme avec le jeu de données et on compare les sorties effectives avec les sorties attendues. Si ces sorties sont identiques, alors aucune faute n'a été détectée dans le programme. Si ces sorties diffèrent, une faute a été trouvée par le test, et il faut alors la localiser et la corriger. Bien sûr, ce n'est pas parce qu'aucune faute n'a été détectée qu'aucune faute n'est présente : en effet, les comportements à tester sont potentiellement infinis. Ainsi, il n'est pas possible de tester exhaustivement tous les comportements du programme ; il faut donc trouver un moyen de décider que le programme a été testé avec un nombre suffisants de tests pertinents. En d'autre termes, il faut définir des critères objectifs permettant de décider de l'arrêt de la phase de test.

Le test de logiciel possède une terminologie précise : on résumera ici les principaux termes utilisés dans ce manuscrit. Ces définitions sont inspirées de [Binder, 2000b, Beizer, 1990, IEEE90, 1990].

On appelle *programme sous test* le programme testé et *données de test* les données à fournir en entrée du programme. Un *cas de test* désigne un ensemble de données, de

conditions d'exécution, et de résultats attendus, développé dans un objectif particulier. Un cas de test est souvent couplé avec un pilote de test (*test driver*) qui est le programme chargé d'exécuter le programme sous test avec les données de test choisies. Par abus de langage, on assimile parfois ces deux notions. Le cas de test doit également quand cela est nécessaire, initialiser le programme sous test de manière à le placer dans un état dans lequel les données de test peuvent être appliquées. Cette notion de préambule sera plus amplement détaillée dans la section 7.2.2.1. Les cas de test sont souvent structurés en *suites de tests*, qui les regroupent de manière logique.

Après exécution d'un cas de test, il y a émission d'un *verdict*, issu de la comparaison entre le résultat obtenu et le résultat attendu. Cette comparaison est appelée *oracle* ou *fonction d'oracle*. L'oracle peut prendre la forme d'instructions exécutables ; il peut être obtenu avec des techniques de type vérification de modèle (*model-checking*)[Clarke et al., 2000], ou, dans le pire cas, être laissé à l'appréciation humaine. Le verdict est « succès » si le résultat obtenu est conforme à l'oracle, « échec » dans le cas contraire. Dans la suite du manuscrit, on utilisera également le verdict « inconclusif » si le cas de test n'a pas pu être exécuté normalement.

À l'échec d'un cas de test commence une phase dite de *diagnostic*, qui consiste à localiser la faute détectée pour pouvoir la corriger. En revanche, si un cas de test réussit, la question qui se pose au testeur est de savoir s'il estime que l'ensemble de cas de test qu'il a exécuté jusque-là est suffisant pour considérer le programme testé, ou s'il lui faut continuer de générer et d'exécuter de nouveaux cas de test. Cette décision est liée à un *critère d'arrêt* ou critère de test, qui spécifie et vérifie si un ensemble de cas de test est suffisant ou pas pour tester un programme. Nous proposons de tels critères d'arrêt dans les chapitres 6 et 7. En pratique, les techniques de génération de cas de test cherchent plus à atteindre un critère de test qu'à tester le logiciel à proprement parler.

Deux grands modes peuvent s'appliquer pour la génération de test : le test structurel ou le test fonctionnel. On parle également, de manière plus imagée, de test en boîte blanche et de test en boîte noire. Le test en boîte blanche consiste à se baser sur la connaissance de la structure interne du programme sous test pour générer des tests et définir des critères d'arrêt. Ce type de test présuppose bien évidemment que l'on dispose des sources du programme et non pas seulement de son interface, ou de ses spécifications. La plupart des techniques de génération de tests structurels se basent sur une abstraction du programme qui est ensuite couverte selon différents critères. Les modèles utilisés sont par exemple les graphes de flot de contrôle [Beizer, 1990] ou les graphes de flot de données [Rapps and Weyuker, 1985], et sont couverts par des critères de couverture des arcs, des nœuds, ou des chemins du graphe.

Le test fonctionnel consiste à considérer le programme comme une boîte noire dont on ne connaît que les spécifications. Pour la génération de test, ces spécifications sont la plupart du temps exprimées de manière la plus formelle possible, par exemple à l'aide de logique temporelle, ou de langages de spécifications comme Z ou B [Abrial, 1996, Imperato, 1991]. Le test fonctionnel ne prend pas en compte la structure du programme, et les tests générés sont ainsi indépendants de l'implémentation du programme sous test, ce qui n'est pas le cas pour le test structurel.

Pour qualifier la qualité globale d'une suite de tests, on peut utiliser une analyse par mutation qui est une technique de test proposée par DeMillo [Millo et al., 1978] et fondée sur l'insertion de fautes. Le principe de l'analyse par mutation est de générer un certain nombre de versions erronées du programme sous test appelées mutants, et de lancer un cas de test sur chacun de ces mutants. Un mutant est une copie exacte du programme sous test dans laquelle une erreur simple a été injectée. Les mutants sont obtenus de manière automatique, par application d'opérateurs de mutation au programme sous test. Chaque opérateur de mutation insère un certain type d'erreurs comme par exemple la suppression d'une ligne de code, ou bien la modification d'une valeur numérique. Le nombre de mutants généré est limité par des facteurs purement techniques (capacité mémoire, temps d'exécution). L'exécution d'un jeu de test sur les mutants permet de déterminer la qualité du cas de test : meilleur est le cas de test, plus il détectera de mutants (quand un mutant est détecté, on dit qu'il est tué par le jeu de test). La proportion de mutants détectés est appelée score de mutation. Par ailleurs, l'analyse des erreurs permet de guider la génération de tests vers de nouveaux cas de test plus efficaces : il faut alors chercher à couvrir les zones d'erreur avec des données de test spécifiques aux cas particuliers.

On peut distinguer trois grandes phases de test dans le cycle de vie d'un logiciel :

- le test unitaire ; cette phase consiste à tester indépendamment les unes des autres toutes les unités élémentaires qui composent le système. Dans le cadre de la programmation par objet, ces unités peuvent être la classe ou la méthode ;
- le test d'intégration ; cette phase intervient lors de l'assemblage de différentes unités, elle consiste à tester un assemblage ainsi constitué de manière à détecter des erreurs provenant des interactions entre les unités assemblées ; cette phase peut également permettre de détecter des erreurs non liées aux interactions entre unités et n'ayant pas été détectées au niveau du test unitaire ;
- le test système ; cette phase consiste à tester le système dans sa totalité. Elle comprend souvent des tests de montée en charge et des tests de performances.

Le test de logiciels modélisés en UML

UML (Unified Modeling Language) est un standard dont la première version est apparue en 1995. UML est né de la fusion des trois principales méthodes de modélisation objets existant dans les années 90 : OMT, Booch et OOSE. UML est un langage qui permet de raisonner avec les concepts objets, et qui sert aussi à supporter une démarche d'analyse et de conception pour les systèmes orientés objets.

UML n'est pas un langage très formalisé, c'est peut être ce qui a fait son succès. C'est par contre un handicap en ce qui concerne le test. En effet, les méthodes formelles ne peuvent pas s'y appliquer directement, puisqu'un modèle UML n'a pas le même niveau de formalisme que les langages comme B, ou Z. Toute la difficulté du test à partir de modèle UML consiste donc à faire un compromis entre l'information dont on souhaiterait disposer, et la difficulté pour le concepteur UML de la fournir. En d'autres termes, il est illusoire de bâtir des méthodes de test basées sur des modèles UML formalisés à outrance : il faut savoir s'adapter à ce qui est fourni, en ne modifiant que

peu les pratiques couramment adoptées. C'est ce qui nous a guidé dans nos travaux de génération de test.

Si en 1999 un manque d'intérêt de la communauté de test pour UML était noté dans [Williams, 1999], les méthodes de test basées sur UML sont désormais assez nombreuses. Un modèle UML est composé de plusieurs vues qui représentent les facettes d'un même logiciel. La plupart des méthodes de test de logiciels modélisés avec UML n'utilisent pas pour autant chacune de ces vues, mais plutôt la vue qui semble la plus adaptée au type de test ciblé. Pour le test d'intégration, la vue la plus souvent utilisée est le diagramme de classes [Briand et al., 2003], qui représente toutes les classes d'un système et leurs associations. Ainsi, toutes les dépendances entre les classes peuvent être déduites de ce diagramme et un plan de test peut être établi. Pour le test unitaire, les vues dynamiques comme les machines à états sont préférentiellement utilisées [Offutt and Abdurazik, 1999, Kim et al., 1999, Antoniol et al., 2002]. Dans [Abdurazik and Offutt, 2000] est introduite une technique de vérification statique et de génération de test pour s'assurer de la bonne collaboration des différents objets participant au système. Cette méthode se base sur les diagrammes de collaboration. Concernant le test système, plusieurs méthodes utilisent des techniques issues du model-checking. Le model-checking de modèles UML se base sur une vue statique (diagramme de classe, plus diagramme d'objets pour la spécification d'un état initial), et d'une vue dynamique par classe (classiquement une machine à état), de manière à pouvoir simuler le modèle UML et y appliquer des techniques de model-checking [Pickin et al., 2002, Lugato et al., 2002, Ghosh et al., 2003]. Les tests ainsi générés sont des tests systèmes. Nous nous appuyons sur certains des outils associés à ces techniques dans notre approche de génération de cas de test, comme présenté dans la section 7.2. D'autres méthodes permettant la génération de test système se basent sur le diagramme de cas d'utilisation. Les cas d'utilisation représentent en effet les grandes classes de fonctionnalités offertes par le logiciel, et s'ils sont suffisamment documentés, notamment au moyen d'autres vues UML comme le diagramme de séquence ou d'activités, ils peuvent guider la tâche de test fonctionnel[Briand and Labiche, 2002, Fröhlich and Link, 2000, Nebut et al., 2003b, Riebisch et al., 2002b]. Ces méthodes sont plus ou moins outillées et outillables, selon que l'on utilise ou pas le langage naturel comme langage de description des principales caractéristiques d'un cas d'utilisation. L'approche que nous proposons dans ce manuscrit se base sur les cas d'utilisation et leur associe un langage de contraintes qui permet de les rendre simulables et d'utiliser un outil de génération automatique de tests que nous avons développé, comme expliqué dans le chapitre 6.

Chapitre 2

La génération automatique de cas de test

L'automatisation de la tâche de test est un de ses aspects les plus fondamentaux. Les auteurs de [Xanthakis et al., 2000] soulignent que le test de logiciels nécessite presque toujours un outil automatique. L'automatisation peut intervenir à deux principaux niveaux :

- des scripts de test qui n'automatisent que l'exécution de cas de test, la conception de ces cas de test, et l'évaluation des résultats restant à la charge du testeur ;
- des outils de génération de cas de test, les cas de test générés pouvant inclure un oracle rendant automatique l'évaluation des résultats.

Si la réalisation de scripts de test ou de pilotes de test est une tâche assez facile, la génération de cas de test est plus ardue. Ce chapitre détaille les principaux travaux existant sur la génération automatique de test, domaine sur lequel porte la contribution principale de ce manuscrit. Nous insisterons moins sur les techniques structurelles, dans la mesure où nos travaux n'ont trait qu'aux approches fonctionnelles.

2.1 Les approches structurelles

Le test dynamique structurel se base sur le code source pour produire des données de test, qui sont ensuite utilisées comme entrée à l'exécution du programme dont les résultats sont comparés à ceux attendus. Nous aborderons dans cette section principalement les techniques fondées sur la couverture d'abstraction du programme sous test, puis nous aborderons brièvement des techniques liées à l'analyse par mutation et aux algorithmes génétiques.

Couverture de graphes de contrôle

Beaucoup de techniques se basent sur des techniques de couverture d'abstractions de la structure du programme. Les abstractions les plus couramment utilisées sont le graphe de flot de contrôle et le graphe Def/Use.

Les graphes de flot de contrôle

N'importe quel programme peut être représenté par un graphe de flot de contrôle. Les nœuds du graphe représentent des blocs d'instructions (i.e. des séquences d'instructions), et les arcs sont orientés et représentent la possibilité de transfert de l'exécution d'un nœud à un autre. Un graphe de flot de contrôle possède un unique nœud racine à partir duquel on peut visiter tous les autres nœuds (le graphe de flot de contrôle est donc connexe). Il ne possède également qu'un seul nœud de sortie.

Critères de couverture

Beaucoup de critères ont été proposés pour couvrir les graphes de flot de contrôle. Chaque critère cherche à extraire du graphe des chemins pertinents, un chemin étant défini comme une traversée du graphe de contrôle depuis le nœud initial jusqu'au nœud final.

- Le critère *tous les sommets* vise à extraire du graphe un ensemble de chemins suffisants pour passer au moins une fois par chaque sommet. L'idée sous jacente est d'assurer la couverture des instructions.
- Le critère *tous les arcs* sélectionne un ensemble de chemins passant au moins une fois par chacun des arcs du graphe. La motivation est alors de couvrir chaque prise de décision.
- Le critère *tous les chemins simples* vise à couvrir tous les chemins d'exécution sans itérer plus d'une fois dans les boucles.
- Le critère *tous les chemins* vise à couvrir tous les chemins d'exécution du programme, ce qui est impossible dans le cas général puisque le nombre de chemins d'exécution est potentiellement infini.

Couvrir tous les chemins implique de couvrir aussi tous les arcs, et couvrir tous les arcs implique de couvrir également tous les sommets. Il est toutefois à noter que le critère *tous les chemins* n'est pas fiable (un critère est *fiable* s'il produit uniquement des jeux de test dont toutes les données produisent un résultat correct ou uniquement des jeux de test dont au moins une donnée produit un résultat incorrect). Une fois qu'un ensemble de chemins a été extrait du graphe de flot de contrôle en utilisant un des critères cités précédemment, le problème de savoir si le chemin sélectionné est faisable (c'est-à-dire s'il existe une donnée de test capable de produire ce chemin) est dans le cas général indécidable. Plus généralement, déterminer si un sommet, un arc, ou un chemin du graphe de flot de contrôle est exécutable est indécidable dans le cas général [Weyuker, 1979].

Graphes de flot de données

Tous les critères précédemment cités se basent sur le recouvrement des chemins d'exécution tels qu'ils sont dictés par le séquencement des prédicats. Il existe d'autres critères, basés eux sur le flot de données, qui sélectionnent des données de test dans le but de couvrir les différentes manières de définir et d'utiliser les variables qui apparaissent dans le programme. Ces critères se basent en général sur un graphe ap-

pelé *Def/use* ou de flot de données, qui a été introduit en 1985 par *Rapps et al* [Rapps and Weyuker, 1985], et qui peut être vu comme un graphe de flot de contrôle décoré par des informations sur la définition et l'utilisation des variables. À chaque sommet j sont associés les ensembles *def(j)* (ensemble des variables définies en j) et *c-use(j)* (ensemble des variables utilisées en j dans un calcul et ayant été définies dans un autre sommet). À chaque arc *j-k* issu d'une décision est associé l'ensemble *p-use(j,k)* qui est l'ensemble des variables utilisées dans le prédicat et conditionnant l'exécution de k. Les hypothèses que doit vérifier un programme sont que toute variable possède une définition et qu'à chaque définition corresponde au moins une utilisation. Un critère de couverture basé sur les flots de données est par exemple *toutes les définitions* qui nécessite que l'on couvre au moins un utilisateur pour chaque définition du graphe. On trouve aussi par exemple les critères *toutes les utilisations* et *tous les chemins définition utilisation* (un chemin *du* ou *définition/utilisation* par rapport à la variable x est un chemin qui part d'une définition de x et dont la première utilisation de x se trouve sur le dernier nœud ou le dernier arc du chemin). Les liens entre les principaux critères liés au graphe de flot de contrôle et ceux liés au graphe Def/Use sont donnés dans [Rapps and Weyuker, 1985].

Application aux programmes orientés à objets et automatisation

Si l'idée de couvrir de tels graphes abstrayant le programme est assez ancienne, des recherches plus récentes investiguent d'une part l'application de telles techniques aux langages à objets, et d'autre part l'obtention des graphes d'abstraction et la satisfaction de tels critères par des procédures automatiques telles que la mutation [Pargas et al., 1999]. Dans le cas de l'application aux langages à objets, on citera notamment les travaux de *Mary Jean Harrold et al* et de *Amie L. Souter et al.* Les travaux de *M. J. Harrold et al* [Harrold and Rothermel, 1994] portent sur le test intra-méthode, inter-méthode et intra-classe. Pour le test intra-classe, les auteurs proposent un pilote de test qui génère aléatoirement des séquences d'appels de méthodes publiques, dans un ordre arbitraire. Les techniques utilisées se basent sur la construction d'un graphe de flot de contrôle puis sur la déduction de toutes les paires définition/utilisation. Les auteurs soulignent qu'ils ne traitent pas un certain nombre de problèmes tels que l'aliasing, l'héritage ou le polymorphisme, et le test inter-classes. Les travaux de *Souter et Al* abordent le test inter-classes [Souter et al., 1999] ainsi que l'obtention de paires Définition/Utilisation dans le cas d'agrégations d'objets [Souter and Pollock, 2001].

Analyse par mutation et algorithmes génétiques

Les auteurs de [Millo and Offutt, 1991] proposent une technique de génération automatique de tests utilisant l'analyse par mutation. Cette technique est fondée sur l'idée de prendre comme objectif de test l'endroit du programme où se trouve l'erreur, et de remonter toutes les contraintes sur les données entre cet endroit et l'entrée du programme. De cette façon, il est possible de générer une donnée de test qui atteint l'erreur et détecte le mutant. Un outil basé sur cette idée a été proposé dans [Millo and Offutt, 1993].

Une autre technique consiste à utiliser des algorithmes génétiques pour muter les cas de test, de manière à apporter de nouveaux cas de test capables de détecter plus de mutants [Baudry et al., 2002a, Baudry et al., 2002b]. On ne conservera un cas de test dans le jeu de tests que s'il augmente l'efficacité du jeu de test. L'efficacité des tests est déterminée à l'aide d'une fonction d'évaluation qui peut être par exemple la couverture de code, le score de mutation ou la couverture de graphes de flot de contrôle ou de graphes Def/Use. Si l'idée de base est bien l'algorithmique génétique, les auteurs ont détecté certaines limites des algorithmes génétiques pour la génération automatique de cas de test, et ont donc développé le modèle bactériologique, proche du modèle génétique, mais qui permet d'obtenir un jeu de test efficace et de taille raisonnable.

2.2 Les approches fonctionnelles

Les approches fonctionnelles pour la génération de test se basent non pas sur le programme à tester mais sur la spécification de ce que le programme est censé réaliser comme fonctionnalité. Ainsi, le point d'entrée des générateurs de test fonctionnels est une représentation ou un modèle du programme. Cette représentation peut être plus ou moins formelle, de spécifications textuelles à des spécifications dans des langages comme B ou Z. Bien sûr, le niveau de formalisation a une grande influence sur la génération de test. Une spécification très formelle sera fastidieuse à écrire, mais en contrepartie permettra une bonne automatisation de la génération de test. Une spécification informelle rendra impossible l'outillage de la génération de test. Un compromis entre les deux approches consiste à utiliser un modèle semi-formel, qui peut être utilisé pour la génération automatique de cas de tests qui doivent être contrôlés et éventuellement enrichis par le testeur avant leur exécution. L'approche que nous proposons se base sur une représentation semi-formelle des exigences fonctionnelles sous forme de cas d'utilisation et de scénarios UML, et permet de générer des scénarios ou des cas de test.

Nous présentons tout d'abord les approches fonctionnelles qui proposent de traduire les spécifications fonctionnelles en un modèle d'automate, puis couvrent ce modèle avec différents critères pour extraire des tests. Ces approches nous concernent directement puisque la méthode que nous proposons dans ce manuscrit se base sur le même principe : définition d'un langage de spécification, traduction vers un modèle de simulation permettant la construction d'un système de transition modélisant les comportements du système, puis définition de critères permettant de couvrir judicieusement cette structure. Nous abordons ensuite les approches basées sur la partition des domaines d'entrée, puis les langages formels B et Z, et enfin les méthodes basées sur les cas d'utilisation, qui sont les plus proches de nos travaux.

2.2.1 Des langages de spécification aux modèles d'automates

Beaucoup de méthodes de génération de tests fonctionnels peuvent se ramener à la couverture d'un graphe ou automate comportemental qui représente le système sous test. Ces graphes ou automates peuvent prendre différentes formes, comme par exemple

les automates d'états finis, les automates étendus à états finis, ou les différents systèmes de transitions [Arnold, 1992] (Labelled Transition System – LTS, Input Output Transition System – IOLTS, ...). Construire « à la main » une telle représentation du système est une tâche ardue voir quasi irréalisable de par la taille des structures construites, c'est pourquoi ces représentations sont en général construites automatiquement à partir des spécifications du système sous test.

Ainsi, pour générer des tests fonctionnels, un grand nombre de méthodes supposent qu'on dispose de spécifications formelles d'un système (écrites en SDL [Belina and Hogrefe, 1989], ESTELLE [Budkowski and Dembinski, 1991], UML formalisé [Le Guennec, 2001], LOTOS [Bolognesi and Brinksma, 1986], ...), et proposent une traduction du langage formel vers un modèle d'automate, automate qui est alors couvert par différents critères, comme nous le verrons par la suite.

La méthode la plus classique – et la plus référencée – concernant la génération de tests à partir d'un modèle formel est celle de J. Dick et A. Faivre [Dick and Faivre, 1993]. Cette méthode se compose de deux étapes :

1. la génération d'un automate d'états finis abstrait à partir du modèle formel, par analyse et partition de l'espace d'états et des opérations ;
2. la sélection de cas de test, les cas de test étant des chemins dans l'automate, en utilisant différents critères de sélection comme *all-transition* (couverture de toutes les transitions) ou *all-transition-pairs* (couverture de toutes les paires de transitions adjacentes).

Cette approche a été la base de beaucoup de recherches pour l'automatiser et l'appliquer à différents types de spécifications.

Concernant les spécifications écrites en SDL, un processus de génération automatique de test a été proposé dans [Tahat et al., 2001]. L'idée est de regrouper en une spécification SDL système tous les fragments SDL constituant la spécification, et d'en déduire automatiquement un automate étendu à états finis. Les cas de test sont ensuite générés en utilisant les critères classiques de la couverture d'automates, comme *tous les états*, *tous les chemins*, *toutes les transitions*.

Cette même approche a été reprise dans [Murray et al., 1998], cette fois pour le test unitaire de classes. Chaque classe est spécifiée avec *Object-Z* ; un automate à états finis est généré à partir de cette spécification, et est utilisé pour la génération de cas de tests.

Après une formalisation d'un modèle d'exécution pour UML, les auteurs de [Lugato et al., 2002] proposent une approche similaire pour le test système. À partir des spécifications UML, un IOLTS étendu est construit. Une exécution symbolique est utilisée pour générer un graphe d'exécution symbolique, dont chaque chemin représente un cas de test symbolique, qu'il faut ensuite valoriser. D'autres approches se basent sur les machines à états UML. Ainsi, les auteurs de [Kim et al., 1999] proposent un mécanisme de transformation des machines à états UML vers des EFSMs (Extended Finite State Machines). La transformation traite essentiellement les constructions particulières des machines à états UML comme les super états, ou les compositions parallèles.

Ainsi, beaucoup de méthodes de test fonctionnels consistent en la construction préliminaire d'un graphe comportemental du système sous test, graphe dont on extrait des cas de test en utilisant différents critères, ainsi que nous le verrons dans la section suivante.

Globalement, toutes ces méthodes font l'hypothèse forte qu'il est possible de construire exhaustivement le graphe comportemental d'un système à partir de ses spécifications. Pour de gros systèmes, l'explosion combinatoire du nombre d'états rend cette hypothèse caduque : le graphe comportemental est potentiellement infini, et dans le cas où il est fini, sa taille le rend très difficile à construire, tant pour des problèmes de mémoire que pour des problèmes de temps. L'outil TGV [Jard and Jéron, 2002] permet de pallier ce problème en construisant à la volée le graphe comportemental. Cette construction est guidée par un objectif de test à atteindre, donné sous forme d'un IOLTS. L'outil TGV doit donc être couplé à un simulateur de spécifications lui permettant de construire le graphe à la volée. Les travaux présentés dans [Pickin et al., 2002] illustrent le couplage entre TGV et UMLAUT [UMLAUT, 2002], un outil capable (entre autres) de rendre un modèle UML simulable.

2.2.2 Couvertures de machines à état

Beaucoup de travaux se sont intéressés à la génération de tests à partir de machines à états. Plusieurs problèmes peuvent être identifiés pour la génération de test : la génération de séquences de test à proprement parler (extraction de chemins pertinents dans les automates), le problème de l'oracle, et la faisabilité des chemins sélectionnés. Ces travaux nous intéressent particulièrement dans la mesure où la méthode de test décrite dans ce manuscrit nécessite de couvrir un système de transitions étiquetées.

L'extraction de chemins

Concernant l'extraction de chemins dans le graphe, le problème réside dans la définition de la couverture de test que l'on souhaite obtenir. Bien entendu, l'idéal est de tester exhaustivement la machine à états, mais cela peut s'avérer infaisable du fait que le nombre de chemins dans un automate est souvent infini.

Les critères de couverture classiques sont la couverture de tous les états, ou de toutes les transitions de la machine à états [Binder, 2000a, Offutt et al., 1999]. Pour les machines à états UML, [Binder, 2000a] propose également la couverture de tous les événements et de toutes les actions. Un autre critère classique consiste à vouloir couvrir toutes les transitions, ou bien encore toutes les séquences de n transitions [Binder, 2000a].

Des critères plus évolués sont évoqués dans [Binder, 2000a], comme le critère *tous les chemins aller-retour* (traduction française de *all round-trip paths*). Ce critère assure la couverture de tous les chemins depuis l'état initial jusqu'à l'état final, ainsi que la couverture de tous les chemins élémentaires dont l'état origine est identique à l'état destination (i.e. toutes les boucles élémentaires). Ce critère est issu des travaux publiés dans [Chow, 1978], dans lesquels l'auteur propose de construire un ensemble de che-

mins nommé P, qui est défini comme un ensemble de couverture des transitions de la machine à état. Plus exactement, pour chaque transition de l'état S_i vers l'état S_j de la machine à états, ayant comme entrée l'état x et comme sortie l'état y, il doit exister dans P deux séquences p et $p.x$ telles que la séquence p mène de l'état initial à l'état S_i et la séquence $p.x$ mène de l'état initial à l'état S_j. [Chow, 1978] propose de prendre comme ensemble P l'ensemble des chemins partiels de l'arbre de test de la machine à états. Un algorithme de construction d'un tel arbre est donné dans [Chow, 1978]. Dans [Binder, 2000a], un tel algorithme est également donné pour les machines à états UML. Le principe de construction est un parcours en largeur d'abord (même si un parcours en profondeur d'abord est également envisageable). Le parcours s'arrête quand toutes les feuilles sont soit l'état final, soit un état déjà rencontré. Une stratégie de manipulation des gardes des transitions est également adoptée : si la transition est gardée par un prédicat simple ou une conjonction de prédicats simples, une seule branche est construite dans le graphe de test, si la garde contient des « ou » logiques, alors une branche par combinaison possible rendant la garde vraie est construite dans le graphe de test. Cette stratégie se rapproche ainsi du critère sémantique que nous définissons Section 6.3. Dans [Antoniol et al., 2002], l'efficacité d'une telle approche est étudiée, l'efficacité étant évaluée en utilisant une technique d'analyse par mutation. Cette étude n'est menée que sur un seul exemple, et ne peut donc pas être considérée comme vraiment représentative, mais montre cependant sur l'exemple que cette approche ne détecte pas 10% des fautes introduites dans la machine à états de l'implémentation. [Antoniol et al., 2002] suggère donc de combiner le critère *tous les chemins aller-retour* avec une approche par catégorie partition. Il est à noter que la méthode de [Chow, 1978] a été revisitée dans [Fujiwara et al., 1991] de manière à générer des suites de test plus courtes.

Dans [Offutt et al., 1999], quatre critères sont proposés pour couvrir un graphe représentant les spécifications du système sous test. Dans ce graphe, les nœuds sont les états du système, et les transitions sont des gardes mettant en jeu les variables du système et des opérateurs arithmétiques, relationnels ou binaires.

Le premier critère est le critère de couverture de toutes les transitions du graphe, déjà mentionné plus haut.

Le second critère consiste à couvrir tous les prédicats. Si on considère qu'une garde est donnée sous forme d'un prédicat défini comme une combinaison booléenne de clauses, alors ce critère cherche à ce que chaque clause intervienne indépendamment des autres clauses dans le tirage d'une transition. Par exemple, les 6 données de test résumées dans la table 2.1 permettent de satisfaire le critère *tous les prédicats* pour une transition gardée par l'expression $(A \vee B) \wedge C$.

Ce critère est proche du critère *tous les termes des pré-conditions* que nous proposons à la section 6.3.2. Cependant, les chemins générés avec le second critère proposé dans [Offutt et al., 1999] ne seront pas forcément satisfiables dans la mesure où les relations arithmétiques n'auront pas été prises en compte lors de l'application de ce critère. De plus, [Offutt et al., 1999] ne traite que les opérateurs booléens pour combiner les clauses, et n'utilise pas de quantificateurs. Par ailleurs, dans le cas où la couverture des prédicats de [Offutt et al., 1999] est assurée, alors la couverture des transitions est également assurée, ce qui implique que ce critère sélectionne un grand nombre de che-

(A $\vee$ B) $\wedge$ C	$(A \vee B) \wedge C$
T F T	**T**
F F T	**F**
F **T** T	**T**
F **F** T	**F**
T T **T**	**T**
T T **F**	**F**

TAB. 2.1 – Données de test satisfaisant le critère *tous les prédicats* sur l'expression $(A \vee B) \wedge C$

mins dans le graphe. Comme nous le verrons dans le chapitre 8, le critère que nous appliquons génère beaucoup moins de chemins dans la mesure où il s'applique non pas sur chaque transition comme c'est le cas pour Offutt, mais sur chaque pré-condition.

Le troisième critère identifié dans [Offutt et al., 1999] est la couverture de toutes les paires de transitions. Dans la mesure où beaucoup d'erreurs dans un logiciel proviennent d'interactions complexes dans les séquences d'états, *Offutt et al.* proposent de générer des chemins qui contiennent toutes les séquences possibles de transitions adjacentes du graphe. Deux transitions $t_1$et t_2 sont dites adjacentes s'il existe un état e tel que $t_1$est une transition entrant dans e et t_2 est une transition sortante de e.

Le dernier critère proposé dans [Offutt et al., 1999] revient à tester toutes les séquences de transition possibles, bien que les auteurs précisent que dans le cas général, il y en a un nombre infini.

L'application de ces critères à des machines à état UML est proposée dans [Offutt and Abdurazik, 1999]. Moyennant certaines suppositions sur la structure des machines à états (entre autres : absence de transitions spontanées, événements et gardes exprimées à l'aide d'attributs de classe booléens, et déterminisme de la machine à états), les algorithmes proposés permettent de générer les cas de test satisfaisant chacun des critères. Les résultats sont donnés sur un programme C de 400 lignes, il est donc difficile de juger de l'efficacité de ces critères dans un contexte orienté objets. On peut supposer toutefois que si cette technique était appliquée à un modèle UML décrivant un programme orienté objets (et non pas C), les cas de test obtenus seraient des tests unitaires et non pas des tests systèmes, à moins de supposer qu'on dispose d'une machine à états UML représentant l'ensemble des comportements de l'application, ce qui est irréaliste. De ce fait, cette méthode n'est applicable que dans le contexte de la génération de tests unitaires, et pas pour la génération de tests système qui nous intéresse dans ce manuscrit.

Des techniques de flots de données peuvent aussi être appliquées. Ainsi, les auteurs de [Kim et al., 1999] proposent pour générer des tests à partir de machines à états UML de transformer les machines à états UML en machines à états finis étendues, puis de les couvrir par des critères comme *toutes les définitions*, *toutes les utilisations*, et *tous les chemins définition-utilisation* de manière analogue aux critères de test structurel présentés dans [Rapps and Weyuker, 1985].

Selon les auteurs de [Kantamneni et al., 2002], la plupart des critères de sélection de tests échouent pour la couverture de certaines branches difficiles à couvrir. Ces branches sont identifiées dans [Kantamneni et al., 2002] comme étant des branches profondément « enfouies » (« deeply nested ») ou les branches gardées par les opérateurs : *égal* (=), *différent* (!=), et *et logique* (&&). Les auteurs de [Kantamneni et al., 2002] proposent une méthode pour tester de manière adéquate ces branches. Cette méthode se base sur la définition d'une métrique nommée le *potentiel* d'une branche. Le potentiel d'une branche est calculé dynamiquement à l'exécution d'un cas de test :

- Il vaut 0 si une branche n'a pas été couverte.
- Si une branche a été couverte, alors son potentiel est la somme du nombre de branches couvertes nichées juste après cette branche, avec les potentiels des branches non couvertes nichées juste après cette branche (si toutes les branches nichées sont couvertes, le potentiel est nul).

La méthode pour couvrir les branches difficiles est la suivante :

- On génère tout d'abord un cas de test avec une méthode quelconque, puis on l'exécute et on calcule le potentiel de chaque branche.
- La branche avec le plus fort potentiel est sélectionnée, et on compare son potentiel avec le potentiel de la branche précédemment sélectionnée. S'il lui est supérieur, alors on réduit l'espace des entrées des nouveaux cas de test pour se focaliser sur cette branche. Puis on regénère un cas de test et on itère le processus.
- Quand on ne couvre plus de nouvelles branches ou qu'on ne découvre plus de branches de potentiels supérieurs à celles déjà examinées au bout d'un nombre d'itérations fixé, on étend l'espace d'entrée pour les nouveaux cas de test.

Pour augmenter l'efficacité de la méthode sus-décrite, [Kantamneni et al., 2002] propose également d'instrumenter le code (sortant ainsi d'une approche boîte noire) pour détecter les branches avec des prédicats difficiles à couvrir.

Le problème de l'oracle

Si l'extraction de cas de test à partir d'une spécification basée sur des états est largement abordée dans la littérature, le problème de l'oracle l'est beaucoup moins. Ainsi, dans tous les travaux précédemment cités, seul celui de Chow [Chow, 1978] explicite la construction de l'oracle. L'oracle est défini d'une part par les erreurs de sorties, et d'autre part par les erreurs de transitions. Ainsi, Chow propose de construire pour chaque couple d'états des séquences discriminantes permettant de différencier les états les uns des autres. Des telles séquences permettent de vérifier que la machine à états de l'implémentation a changé d'état en accord avec la spécification. Cette technique est reprise dans [Binder, 2000a] sous le nom de *signature de longueur M* (*M-lenght signature*).

La faisabilité des chemins

Les procédés d'extraction décrits ci-avant ne garantissent en général pas que les chemins extraits sont faisables : un chemin extrait peut parfaitement avoir des gardes

incompatibles. Dans [Helke et al., 1997], des mécanismes de preuve sont utilisés pour détecter et supprimer les cas de test (générés à partir de spécifications Z) non satisfiables. Ce mécanisme de preuve est automatique et mis en œuvre en utilisant l'outil Estelle.

2.2.3 Les méthodes basées sur la partition du domaine d'entrée

Un grand nombre de méthodes de test fonctionnel sont basées sur l'idée de partitionner le domaine d'entrée : catégorie-partition, graphes cause-effet, et tables de conditions. Plusieurs de ces méthodes s'appliquent à partir des exigences textuelles, bien qu'aucune n'automatise la transcription vers un modèle formel.

La méthode catégorie-partition

La méthode catégorie-partition est peut être la plus connue des méthodes basées sur la partition du domaine d'entrée, et a été adaptée dans de nombreux travaux. L'idée générale est de partitionner les domaines d'entrée pour identifier certains intervalles-clefs, de manière à générer ensuite des données de test dans chaque partition. La méthode catégorie-partition a été proposée initialement par Ostrand et Balcer dans [Ostrand and Balcer, 1988]. Cette méthode part du postulat que les spécifications sont écrites en langage naturel. La méthode se compose de 6 grandes étapes :

1. Analyse des spécifications. Les unités fonctionnelles sont identifiées et caractérisées par des paramètres, et les objets de l'environnement pouvant interagir avec ces unités. De telles caractéristiques d'une unité fonctionnelle sont les *catégories* qui influencent l'unité.
2. Partition des catégories en choix. Le testeur détermine les différents cas signifiants qui peuvent se produire pour chaque catégorie.
3. Détermination des contraintes sur les choix. Le testeur détermine comment les choix interagissent les uns avec les autres.
4. Écriture et traitement des spécifications de test. Les informations sur les catégories, les choix et les contraintes sont écrites en TSL (Test Specification Language), un langage formel de spécification de test proposé par les auteurs. Ces spécifications sont traitées par le générateur proposé par les auteurs, qui produit des canevas de test pour l'unité fonctionnelle choisie.
5. Évaluation des sorties du générateur. Après évaluation des canevas de test produits, le testeur peut décider de modifier la spécification des tests.
6. Transformation en scripts de test. L'outil proposé par les auteurs permet alors de transformer les canevas de test en scripts de test.

Cette méthode est bien sûr très orientée données. L'utilisation de l'outil est finalement anecdotique, l'important est l'utilisation de catégories et la partition de ces catégories. L'inconvénient de cette technique reste le fort besoin de l'intervention du testeur et de son expérience pour déterminer les choix et transcrire catégories, choix et contraintes en un langage formel.

La technique de catégorie-partition a été reprise au sein de beaucoup de méthodes, pour générer des données de test. Dans la plupart de ces méthodes, le mécanisme de partitionnement est automatique, grâce à l'utilisation de spécifications formelles du système. On peut alors automatiquement déterminer des ensemble de valeurs à tester et les valeurs limites du domaine pour faire du test aux limites. L'analyse de partition automatique est présentée dans des travaux comme [Bernot et al., 1991]. Beaucoup de travaux [Dick and Faivre, 1993, Aertryck et al., 1997] se basent sur une analyse de partition pour construire une abstraction du graphe d'accessibilité sous forme d'un automate à états finis. Dans [Legeard et al., 2002], en partant de spécifications B ou Z, une analyse est faite de manière à déterminer les valeurs limites des domaines et générer ainsi des tests aux limites. Les partitions sont générées par une technique de programmation logique par contraintes, et les jeux de test trouvés sont intégrés à des suites de test.

Les graphes cause-effet

Les graphes cause-effet ont été introduits par Elmendorf [Elmendorf, 1974] mais ont été popularisés par Myers [Glenford J. Myers, 1976]. La méthode des graphes cause-effet consiste tout d'abord à identifier les principales fonctions du système à tester. Puis, pour chaque fonction, le testeur doit déterminer les *causes* explicites et implicites influençant le comportement de cette fonction, ainsi que tous les *effets* de cette fonction. Cette phase peut s'apparenter à la définition des catégories dans l'approche catégorie-partition. L'étape suivante consiste à relier dans un graphe logique les causes et les effets à l'aide d'un réseau d'opérateurs booléens. Les cas de test sont définis pour chaque effet en considérant toutes les combinaisons de causes produisant cet effet. La principale limitation de cette approche est sa complexité : le graphe cause-effet peut être très complexe et donc extrêmement difficile à construire. Comme on le verra, cette idée de couvrir les combinaisons de données validant ou invalidant un effet se retrouve en filigrane dans le critère de couverture des pré-conditions décrit dans la section 6.3.2.

Les tables de conditions

La méthode des tables de condition a été proposée en 1975 par Goodenough et Gerhart [Goodenough and Gerhart, 1975]. Le principe est de construire une table dans laquelle chaque colonne représente une combinaison de conditions qui peuvent se produire au cours de l'exécution du programme. On le fait en examinant les spécifications du programme et en extrayant les principales conditions pouvant intervenir dans le comportement du programme. Ceci peut s'apparenter à la recherche de catégories. Puis chaque condition se voit attribuer l'ensemble de ses valeurs possibles, ce qui peut revenir à la phase de partitionnement de la méthode catégorie-partition. Des conditions sont également définies, liant les causes entre elles.

Les méthodes basées sur la partition des domaines d'entrée sont très orientées données, puisqu'il s'agit de partitionner les domaines de données pour en trouver les valeurs « intéressantes » pour le test. Les méthodes basées sur la construction de graphes comportementaux sont, elles, plus orientées contrôle, puisqu'il s'agit d'extraire des chemins

d'exécution intéressants pour le test (les chemins d'exécution étant évidemment choisis dans l'abstraction qu'est le graphe comportemental). Si nous avons découplé la présentation de ces deux méthodes dans un souci de clarté, en pratique elles peuvent être utilisées ensemble. Par exemple, la méthode proposée par Dick et Faivre dans [Dick and Faivre, 1993] repose sur le partitionnement des domaines d'entrée des opérations de la spécification en utilisant une technique de réduction en forme normale disjonctive, et ce partitionnement sert ensuite à la construction d'un automate à états finis qui modélise l'ensemble des séquences autorisées. Cet automate est ensuite utilisé pour sélectionner des cas de test.

2.2.4 Les méthodes basées sur des spécifications B ou Z

Les langages B et Z sont des langages permettant l'écriture de spécifications formelles, et plusieurs travaux se sont intéressés à la génération de test pour de telles spécifications.

Les travaux de [Aertryck et al., 1997] proposent une approche de génération automatique de tests pour des spécifications B, supporté par l'outil B-Casting (ces travaux couvrent en fait un spectre plus large de langages de spécification, mais les exemples sont généralement donnés en B). L'outil se base sur un formalisme simple de schéma avec pré-condition et post-condition comme format d'entrée. Le testeur définit pour chaque unité pour laquelle il souhaite générer automatiquement des tests ce que les auteurs appellent « une stratégie de test », qui peut être vue comme un critère de test. Une stratégie de test peut par exemple consister à couvrir toutes les disjonctions de la mise sous forme normale disjonctive des pré-conditions des schémas impliqués. À partir des spécifications B et d'une telle stratégie, l'outil B-Casting génère des spécifications de cas de test, qui peuvent être vues comme des objectifs de test. Les spécifications de cas de test sont ensuite transformées en scénarios de test, c'est-à-dire en une suite d'invocations permettant d'exhiber la spécification de cas de test. L'outil procède pour cela à la construction d'un graphe symbolique de la spécification pour en extraire les scénarios de test à l'aide d'un solveur de contrainte. La construction de l'oracle se base sur l'hypothèse forte que toutes les variables de la spécification sont observables.

Dans [Hierons, 1997], l'auteur propose la construction exhaustive à partir de la partition des domaines d'entrée des opérations. Une approche comme celle de [Legeard et al., 2002] permet de générer un tel graphe à la volée. Les travaux de *Legeard et al* se concentrent sur la génération de tests aux limites à partir de spécifications B ou Z. Les auteurs proposent de déterminer les états limites du système avec une analyse de partition de domaines, puis de générer la séquence d'invocations constituant le test (c'est-à-dire permettant d'atteindre l'état limite) grâce à la construction à la volée du graphe d'atteignabilité des spécifications.

Une approche différente est proposée dans [Helke et al., 1997], qui se base sur l'outil de preuve Isabelle. Cette approche propose la mise sous forme normale disjonctive de chaque schéma de prédicat, puis la sélection d'un cas de test par disjonction. Les cas de test insatisfiables sont détectés et supprimés. Tout ce processus est automatique grâce à l'utilisation de l'outil Isabelle.

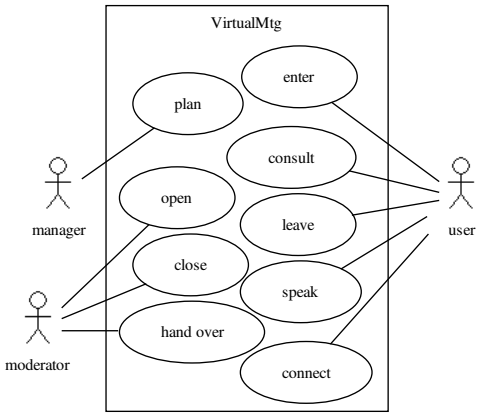


FIG. 2.1 – Exemple de diagramme de cas d'utilisation

2.2.5 Les méthodes basées sur les cas d'utilisation

Les cas d'utilisation sont un moyen de spécifier les utilisations attendues (et exigées) d'un système. Classiquement, ils sont utilisés pour capturer les exigences d'un système, c'est-à-dire ce qu'un système est censé faire. Les cas d'utilisation appartiennent à un système ; tous les utilisateurs ou les autres systèmes qui peuvent interagir avec ce système sont des acteurs. Les comportements requis d'un système sont spécifiés par un ou plusieurs cas d'utilisation, qui sont définis en fonction des besoins des acteurs. Le système, les cas d'utilisation et les acteurs peuvent être représentés en UML par un diagramme de cas d'utilisation tel que celui donné figure 2.1. Chaque cas d'utilisation peut avantageusement être détaillé en suivant par exemple le schéma proposé par Cockburn [Cockburn, 1997], comme cela est illustré dans la figure 2.2(a).

Assez peu de méthodes existent jusqu'à présent pour générer des tests à partir d'un modèle de cas d'utilisation. Les cas d'utilisation représentent pourtant une modélisation des principales fonctions d'un système. Les cas d'utilisation permettent de modéliser les besoins des clients d'un système. Ils ne modélisent pas exhaustivement les fonctions du système, mais permettent de clarifier, filtrer et organiser les besoins. Une fois identifiés et structurés, ces besoins définissent le contour du système à modéliser (ils précisent le but à atteindre), et permettent d'identifier les fonctionnalités principales (critiques) du système. Les cas d'utilisation ne décrivent pas de solutions d'implémentation, leur but est justement d'éviter de tomber dans la dérive d'une approche fonctionnelle, où l'on liste une litanie de fonctions que le système doit réaliser. En ce sens, les cas d'utilisation sont une bonne base pour le test fonctionnel. Ce qui freine sans doute l'élaboration de méthodes de test basées sur les cas d'utilisation est le peu de formalisation des cas d'utilisation. En effet, la description des cas d'utilisation est essentiellement textuelle. On peut même s'interroger sur le bien-fondé d'avoir des cas d'utilisation très formalisés. Les cas d'utilisation sont par essence simples, et peuvent servir de base à la communication entre le client d'un logiciel et ses concepteurs. Une forte formalisation des cas d'utilisation leur ferait perdre cette vocation de clarification et de synthèse qu'ils ont

actuellement.

Ainsi, les cas d'utilisation offrent une base intéressante bien qu'assez floue à la génération de tests fonctionnels. Les approches s'intéressant à la génération de tests à partir de cas d'utilisation sont à notre connaissance très peu nombreuses. Elles représentent pourtant une alternative à la génération de test à partir de spécifications formelles, et s'insèrent plus facilement dans un contexte classique de développement logiciel. Nous présenterons dans cet état de l'art quatre méthodes [Briand and Labiche, 2002, Fröhlich and Link, 2000, Ryser and Glinz, 1999, Riebisch et al., 2002b]. La méthode proposée par *Frolich et al* [Fröhlich and Link, 2000] repose sur une phase non outillée d'analyse et de raffinement des cas d'utilisation vers l'obtention d'un modèle plus formel qui sert de base à la génération de test avec des outils de planification. Ce même modèle est utilisé également par *Riebisch et al* [Riebisch et al., 2002b] pour faire de la génération de test statistique. La méthode de *Briand et al* [Briand and Labiche, 2002] se base sur l'expression des dépendances entre cas d'utilisation et l'extraction de chemins du graphe sous-jacent, puis sur la substitution des cas d'utilisation par des diagrammes de séquence transformés sous forme d'expressions régulières. Cette méthode n'est pas outillée mais les auteurs donnent des pistes pour la réalisation d'un outil automatisant la méthode. La méthode de *Ryser et al* [Ryser and Glinz, 1999] est une procédure de test systématique qui va de l'identification des scénarios primordiaux à la construction de scénarios de test.

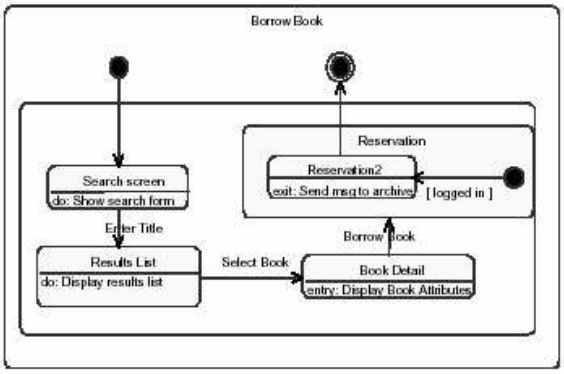
La méthode proposée par *Frolich et al* part du principe que les cas d'utilisation sont documentés en utilisant une structure proche de celle proposée par [Cockburn, 1997]. Cette documentation se compose d'un objectif, de pré- et post- conditions, d'un scénario principal de succès, d'extensions, et de variations, ainsi que d'une liste de cas d'utilisation inclus, comme cela est illustré dans la partie gauche de la figure 2.2. Tous ces éléments de description sont supposés être sous forme de langage naturel.

À partir d'une telle description textuelle, *Frolich et al* proposent une manière systématique de dériver une machine à états décrivant le comportement de chaque cas d'utilisation. La construction d'une telle machine à états se base tout d'abord sur le scénario principal de succès. Chaque étape du scénario correspond à un événement, et l'intervalle entre deux messages est un état. Le début du cas d'utilisation est modélisé par un état initial, et la fin par un état final. La machine à états est encapsulée dans un super état portant le nom du cas d'utilisation, afin de pouvoir être réutilisée par la suite. La figure 2.2 illustre cette construction par l'exemple donné dans [Fröhlich and Link, 2000], pour un cas d'utilisation représentant l'emprunt d'un livre dans une bibliothèque.

La machine à états ainsi obtenue est complétée en prenant en compte les variations qui sont des alternatives locales à l'exécution d'une étape. Les extensions sont ensuite elles aussi incorporées à la machine à états : les extensions décrivent en général une solution alternative pour atteindre un but secondaire du cas d'utilisation, comme par exemple l'échec. Ces extensions sont représentées en utilisant des sous-états de l'état étendu. Les pré et les post conditions du cas d'utilisation sont ensuite traitées. Les pré-conditions sont utilisées pour ajouter des gardes à la transition issue de l'état initial de la machine à états. Les post-conditions sont utilisées pour rajouter une action finale avant l'atteinte de l'état final. La figure 2.3 illustre la traduction des pré-conditions et

Name	Borrow Book
Goal	This use case describes how a library user selects and then borrows a book from the library.
Preconditions	None
Postconditions	The user is registered as the borrower of the book in the library system.
Main Success Scenario	1. The user selects the search function from the main menu. 2. The system displays the search form. 3. The user enters the title of a book (possibly using wild-cards). 4. The library system presents a list of all matching books 5. The user selects a book. 6. The system displays the detail view for this book. 7. The user selects borrow from the menu for this book. 8. The user is already logged in. The system issues a message to the archive that the book is reserved for the user.
Extensions	4a) There are no matches to the query. 4a1) The system returns to the main screen. 8a)The user is not logged in. 8a1)The user logs in as described in Log in.
Variations	3a) The user enters the name of the author. 3b)The user selects the author from an author list. 3b1)The user clicks on "select author". 3b2)The system displays a selection list of all authors. 3b3)The user selects an author from the list.
Included Use Cases	Log in

(a) Description du cas d'utilisation Borrow



(b) Machine à état représentatnt le scénario principal de succès

FIG. 2.2 – Illustration de la construction de la machine à états représentant le scénario principal de succès du cas d'utilisation *Borrow* – Figures extraites de [Fröhlich and Link, 2000].

post-conditions. Quand un cas d'utilisation en inclut un autre, les machines correspondantes sont reliées en utilisant soit un état référence à une sous machine de manière à formaliser le cas d'utilisation subordonné dans le cas d'utilisation englobant, soit des états bouchons pour connecter les états de la machine subordonnée aux bons états de la machine englobante.

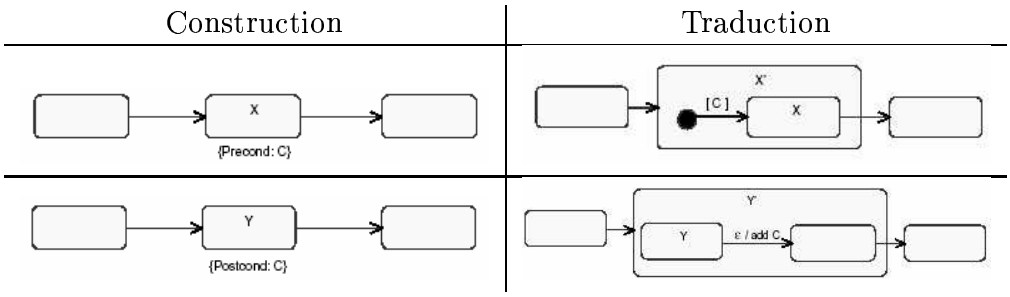


FIG. 2.3 – Traduction des préconditions et des postconditions – Figures extraites de [Fröhlich and Link, 2000].

Ainsi, cette méthode systématique permet d'obtenir une machine à états modélisant chaque cas d'utilisation. L'extraction de cas de tests à partir de telles machines à états se fait en modélisant le problème sous forme de planification. L'outil de planification utilisé est STRIPS [Fikes and Nilsson, 1971]. Les auteurs proposent de transcrire le problème de génération de cas de tests en un problème de planification dont le but est d'atteindre

une certaine transition, en aboutissant dans l'état final. Les auteurs définissent par ailleurs des procédures pour atteindre des critères classiques de couverture de machines à états. Les scénarios de test générés sont très nombreux. Dans la mesure où chaque scénario devra faire l'objet d'une étude par le testeur (pour en déterminer l'oracle notamment, ainsi que la séquence exacte d'actions mises en jeu), la génération de trop nombreux scénarios probablement redondants nous paraît un handicap à cette méthode. Nous proposons dans notre approche des critères générant peu de scénarios de test, ces critères sont décrits dans la section 6.3.

L'avantage de la méthode de *Frolich et al* réside principalement dans l'automatisation et la rapidité de génération des cas de test une fois que la phase de construction des machines à états des cas d'utilisation est terminée. Cependant, le terme de cas de test semble quelque peu abusif. En effet, la spécification des cas d'utilisation a lieu tôt dans l'analyse du logiciel, et donc les protocoles exacts de communication ne sont pas encore connus. C'est pour cela pour nous préférons utiliser le terme de *scénario de test* plutôt que de cas de test dans la partie de ce manuscrit consacrée à la génération de cas de test à partir des cas d'utilisation et des scénarios, dans le chapitre 7. Chaque scénario de test généré vise à tester un cas d'utilisation particulier, et donc les interactions entre cas d'utilisation ne sont pas testées indépendamment des inclusions de cas d'utilisation. Nous verrons que d'autres méthodes visent plutôt à générer des tests couvrant ces interactions. En particulier, la méthode que nous proposons a été créée dans ce but. Concernant la description initiale nécessaire pour chaque cas d'utilisation, l'utilisation de pré et post conditions nous semble particulièrement intéressante. Comme nous le verrons dans le chapitre 6, nous exploiterons beaucoup ce type d'informations. De même, l'utilisation de scénarios et de variantes nous semble primordial. Dans notre méthode, nous utilisons un vocabulaire et une modélisation différente, mais nous utilisons ces mêmes concepts avec la définition de scénarios nominaux et exceptionnels, comme nous le verrons dans la section 7.1.1.

La méthode proposée par *Riebisch et al* se base sur une transformation de la description des cas d'utilisations vers une machine à états très similaire à celle proposée par *Frolich et al.* L'utilisation de la machine à états ainsi construite est par contre très différente, puisqu'elle sert de base à une technique de test statistique. Le test statistique est une technique de test système qui vise à s'assurer que les produits logiciels atteignent un certain niveau de confiance : l'idée est d'étudier la distribution que les différentes entrées obtiennent lors de l'exécution du logiciel, et d'ensuite générer aléatoirement plus de données dans les plages de valeurs apparaissant le plus souvent. Ainsi, l'utilisateur sait que le logiciel a une bonne garantie de fonctionnement dans des configurations « normales » pour lesquelles il a été testé de manière plus approfondie. L'idée sous-jacente est que les différentes parties d'un programme ne nécessitent pas d'être testées avec la même attention, dans la mesure où différentes parties d'un logiciel seront exécutées avec une fréquence plus élevée que d'autres. Le test statistique vise à détecter les zones à haute fréquence d'exécution pour pouvoir les tester plus intensément.

Pour appliquer des techniques de test statistique aux machines à états des cas d'utilisation, *Riebisch et al* appliquent aux machines à états différents traitement successifs. Tout d'abord les machines à états sont transformées en graphes d'usage. Pour cela, il y

a, en particulier, mise à plat des machines à états, unification des stimuli et remplacement des transitions spontanées par des transitions ayant pour activateur l'événement particulier ε. Les graphes d'usage obtenus sont ensuite transformés en modèles d'usage, c'est-à-dire que chaque transition se voit attribuer une probabilité. La distribution de probabilité est manuelle, et son obtention n'est pas décrite dans [Riebisch et al., 2002b]. Les tests sont ensuite générés à partir de ces modèles d'usage. Chaque cas de test est une traversée du graphe d'usage, le choix de la transition de sortie est déterminée à chaque passage dans un état par la probabilité attachée aux transitions. De même que dans la méthode de *Frolich et al*, le terme de cas de test nous semble abusif, pour la même raison que celle précédemment citée. L'application de techniques de test statistique au niveau des cas d'utilisation nous semble extrêmement intéressante.

La méthode proposée par *Ryser et al* [Ryser and Glinz, 1999, Ryser and Glinz, 2000] est avant tout une démarche de génération manuelle et systématique des cas de test à partir des cas d'utilisation, sans vocation à rendre la génération automatique. Il est à noter que les auteurs utilisent un vocabulaire différent du nôtre et de celui des travaux précédemment cités. En effet, dans cette méthode, les notions de cas d'utilisation et de scénarios sont amalgamées. Nous conservons le vocabulaire choisi par les auteurs dans la description de la méthode qu'ils proposent. La première étape de la méthode est l'identification des scénarios. Cette identification est guidée par une procédure en 15 étapes, de l'identification des acteurs à la vérification et la validation des scénarios par des revues. Les scénarios, jusqu'alors exprimés en langage naturel, sont ensuite formalisés et annotés. Le formalisme utilisé est le *statechart*. Le processus de création des statecharts n'est pas systématique, mais est guidé par huit « heuristiques ». Cette transformation est bien sûr coûteuse, mais les bénéfices attendus justifient ce coût. Les statecharts sont annotés en utilisant le langage le plus adapté. Ces annotations peuvent inclure des pré-conditions, des aspects concernant les données, ainsi que des besoins non-fonctionnels. Ces annotations ont pour but de faciliter la recherche de données de test. Les cas de test peuvent ensuite être générés. Pour cela, les auteurs proposent une couverture de toutes les transitions du statechart, même si n'importe quel critère de couverture pourrait être adopté. Les annotations aident le testeur à sélectionner des chemins faisables, et à les valoriser, ainsi qu'à trouver le préambule. L'oracle est déduit de la machine à états qui est censée exprimer le comportement attendu du système.

Une seconde manière de générer les cas de test est proposée. Cette fois le but n'est plus de trouver des tests pour un scénario, mais de tester les dépendances entre scénarios. Pour cela, les dépendances entre scénarios doivent être explicitées. Les auteurs proposent un langage graphique pour ce faire, appelé diagramme de dépendance (*dependency chart*). Ce langage permet de capturer les dépendances logiques et temporelles entre scénarios. La génération de test à partir de tels diagrammes n'est pas très détaillée ; il s'agit juste selon les auteurs de générer des cas de test pour chaque dépendance. Le fait de tester les dépendances entre les scénarios (i.e. dans notre vocabulaire entre les cas d'utilisation) nous paraît pourtant primordiale et nos travaux exploitent cette idée. Par ailleurs aucun oracle précis n'est défini dans le cas de la génération à partir de diagrammes de dépendances.

La méthode proposée par *Briand et al* vise à générer des test systèmes enchaînant les

cas d'utilisation du système sous test. De même que *Ryser et al* proposaient d'exprimer les dépendances entre cas d'utilisation avec des diagrammes de dépendance, *Briand et al* proposent, eux, des diagrammes d'activité UML étendus par deux stéréotypes $<<*>>$ et $<<+>>$ qui, appliqués à un cas d'utilisation, précisent que celui-ci peut être appliqué plusieurs fois (respectivement un nombre quelconque de fois ou un nombre strictement positif de fois). Comme on le verra dans le chapitre 8, un tel diagramme d'activité peut s'avérer très difficile à construire, et certaines dépendances ne sont pas exprimables avec un tel formalisme. D'un tel diagramme d'activité, des chemins sont extraits sous forme d'expressions régulières. Les détails des algorithmes proposés sont donnés dans [Briand and Labiche, 2001]. L'extraction de chemins n'est pas guidée par l'application d'un critère de couverture.

Chaque cas d'utilisation est supposé documenté par un diagramme de séquence très détaillé. Chaque diagramme de séquence est transformé en expression régulière. Puis, pour chaque expression régulière portant sur les cas d'utilisation, les cas d'utilisation sont substitués par chacune des expressions régulières de leurs scénarios. On obtient ainsi une expression régulière décrivant une séquence de test. Il faut ensuite s'assurer de la faisabilité des chemins en utilisant les gardes OCL présentes dans les scénarios des cas d'utilisation, et trouver les séquences d'appels de méthodes exactes des cas de test. En particulier, il faut déterminer dans le cas de la présence d'éléments répétables, le nombre de leur répétition. Les cas de test ainsi obtenus sont ensuite complétés par des oracles. Les oracles sont obtenus en utilisant les post-conditions OCL des méthodes invoquées.

Cette méthode n'est pas automatisée, des pistes sont néanmoins données pour la création d'un outil prototype. Son application nécessite de disposer d'une modélisation extrêmement détaillée du système, puisque les diagrammes de séquence utilisés sont des diagrammes très détaillés, avec notamment l'expression précise de gardes en OCL ; les méthodes utilisées doivent de plus être documentées par des pré et post conditions en OCL. Par ailleurs, cette méthode peut être rendue inapplicable quand les diagrammes d'activité ne suffisent pas à modéliser les dépendances entre cas d'utilisation. La méthode que nous esquissons dans le chapitre 4 prolonge celle de *Briand et al.* et se compose aussi de deux étapes principales, une traitant les cas d'utilisation, et la seconde utilisant les cas d'utilisation. Cependant, nous utilisons une autre manière de modéliser les dépendances entre cas d'utilisation qui, étant plus expressive, nous permet de traiter des cas d'étude plus complexes. De plus, notre modèle peut être dérivé directement des exigences textuelles, ce qui semble difficile à réaliser avec l'approche de *Briand et al.* Nous proposons également des critères de couverture pour la génération de séquences de cas d'utilisation.

2.2.6 Bilan et conclusions

Nous avons dans ce chapitre . Nous avons brièvement donné un aperçu des techniques structurelles avec notamment l'utilisation de graphes de flots de contrôle construits à partir du programme sous test. Nous nous sommes ensuite concentrés sur les approches fonctionnelles qui nous intéressent plus particulièrement. Comme nous l'avons vu, la plupart des approches se basent sur la couverture de machines à états

construites à partir des spécifications du programme sous test. Les autres approches sont plutôt basées sur les données comme les approches catégories-partitions. Nous avons également abordé la génération de test à partir de spécifications formelles en B ou Z, ainsi que la génération de test à partir des cas d'utilisation.

Concernant les techniques fonctionnelles qui nous intéressent plus particulièrement dans ce document, la plupart des approches que nous avons citées ici ne peuvent être appliquées que dans le cadre de test unitaire, ou de petits systèmes. La construction de graphes comportementaux est en effet à la base de beaucoup d'approches, ce qui pose le problème du passage à l'échelle. De plus, on peut noter que peu des approches citées peuvent être appliquées facilement à échelle industrielle, de par leur peu d'intégration aux phases classiques de développement.

Dans ce manuscrit, nous proposons une méthode orientée vers le test système, et qui se base sur les cas d'utilisation couramment utilisés dans les cycles classiques de développement industriel. Les cas d'utilisation sont de haut niveau d'abstraction mais permettent néanmoins de raisonner sans se heurter au problème de passage à l'échelle. Nous nous proposons de contribuer sur quatre points principaux par rapport aux autres approches existantes que nous avons citées ici :

- la génération de cas de test prenant en compte des séquences de cas d'utilisation et non pas seulement des cas de test correspondant à un unique cas d'utilisation comme cela est proposé dans [Fröhlich and Link, 2000] et [Riebisch et al., 2002b].
- la définition d'un format d'entrée simple et non ambigu permettant la génération de test, le diagramme d'activité utilisé par *Briand et al* ne nous semblant pas adapté,
- la génération de cas de test concrets, en utilisant des informations de traçabilité entre les modèles d'exigences et de conception, et via l'utilisation d'outils de synthèse de test,
- l'utilisation de critères de test permettant de générer un nombre raisonnable de tests pertinents.

Chapitre 3

Le test de lignes de produits

L'idée de construire des lignes de produits logicielles est née en 1976 avec la proposition de *Parnas* de construire des familles de programmes [Parnas, 1976]. Cependant, ce n'est que depuis cette dernière décennie que les lignes de produits logicielles ont bénéficié d'un réel intérêt, tant de la part des industriels que de la communauté académique, avec notamment la création de conférences et de projets de recherches européens (tels que ARES, PRAISE, CAFE et FAMILIES) y étant totalement consacrés. En ce sens, l'ingénierie des lignes de produits peut être vue comme une approche récente du génie logiciel. Elle regroupe les activités de développement d'un ensemble de logiciels appartenant à un domaine particulier. Les *lignes de produits logicielles* sont une transposition des chaînes de production au monde du logiciel. Le principe est de minimiser les coûts de construction de logiciels dans un domaine d'application particulier en ne développant plus chaque logiciel séparément, mais plutôt en le concevant à partir d'éléments réutilisables, appelés *assets* (ou *avoir conceptuel*). Un asset peut ainsi être une exigence, un modèle, un composant, un plan de test ou tout simplement un document. La première difficulté liée à cette approche réside dans la conception d'une architecture permettant de définir plusieurs produits. Les membres d'une ligne de produits sont caractérisés par leurs points communs, mais aussi par leurs différences (aussi appelées « points de variation »). La gestion de cette variabilité est un des concepts clef des lignes de produits. Dans une chaîne de production de véhicules, des voitures sont fabriquées à partir d'un ensemble d'éléments communs (roues, volant, vitres, etc) mais peuvent comporter certaines caractéristiques qui les différencient (nombre de chevaux du moteur, présence ou non de la climatisation, ...). Dans le monde logiciel, les différences peuvent apparaître de manière analogue, en fonction de choix techniques (utilisation d'un type particulier de matériel), commerciaux (création d'une version limitée), ... Une autre difficulté réside dans l'utilisation d'une ligne de produits. La construction d'un produit logiciel (on parle aussi de *dérivation de produit*) consiste en partie à figer certains choix (cristallisation) vis-à-vis des points de variation définis dans la ligne de produits. De toute évidence, certains choix sont incompatibles entre eux. Si l'on reprend l'analogie automobile, une voiture comporte généralement un seul moteur, et il faut alors choisir entre une motorisation essence ou diesel. De la même manière, un choix particulier lors de la dérivation d'un logiciel peut exclure certaines variantes. Par exemple le choix d'un cabriolet à toit amovible exclura la possibilité de choisir un toit

ouvrant. Une ligne de produits doit donc aussi intégrer des contraintes de cohérence permettant de faciliter les choix lors de la dérivation.

La problématique de test sous-jacente aux lignes de produits logiciels concerne essentiellement la factorisation et la réutilisabilité des tests. Il semble en effet naturel de vouloir tester de la même façon les parties communes à tous les produits, et de ne pas écrire autant de tests que de produits pour un même objectif de test. Cette réutilisation de tests ne peut bien sûr pas avoir lieu au niveau du code de test, puisque les variabilités entre les produits impliquent des différences assez grandes ne serait-ce qu'au niveau du type des objets manipulés. Il est donc nécessaire d'écrire des tests génériques, et ce dans un format dédié. Ces tests sont ainsi définis au niveau de la ligne de produits et non pas au niveau des produits. Les tests génériques sont ensuite instanciés pour chaque produit au moment de la phase de dérivation. Les tests génériques peuvent ainsi être considérés comme des assets de la ligne de produits. Construire de tels assets n'a d'intérêt que si d'une part l'écriture de tests génériques est une tâche de complexité similaire à l'écriture de tests concrets, et d'autre part si l'instanciation de tests concrets est automatisée.

Les lignes de produits ont donné lieu depuis quelques années à une abondante production de publications, en général à forte tendance industrielle. De toutes ces publications ne se dégage pas encore vraiment de consensus, ni dans la façon de procéder pour construire une ligne de produits, ni dans les techniques de modélisation. De plus, peu d'exemples de lignes de produits sont diffusés. Les différentes équipes universitaires ont donc chacune fabriqué une ligne de produits, en y incluant leurs propres problématiques et en utilisant leurs propres solutions. Il est de ce fait difficile d'homogénéiser et d'analyser la vaste somme de travail réalisée dans ce domaine, et notamment très peu d'études comparatives ont été menées.

Dans cette section, nous décrirons brièvement les différentes techniques et méthodes proposées pour la construction de lignes de produits logiciels, puis nous insisterons sur les aspects test.

3.1 Les lignes de produits logicielles

Une ligne de produits est un ensemble de produits qui partagent un ensemble commun d'exigences, mais qui possèdent aussi d'importantes variations dans les exigences [Griss, 2000]. On parle aussi de familles de produits (on utilisera les deux termes indifféremment dans ce manuscrit). Le concept-clef d'une ligne de produits est la variabilité, et c'est sa gestion à toutes les étapes du cycle de vie du logiciel qui pose difficulté. Dans cette section, les concepts liés à la variabilité sont d'abord étudiés. Nous insistons ensuite sur les aspects exigences dans la mesure où ce manuscrit aborde le test de lignes de produits à partir de ses exigences, puis nous décrivons brièvement différentes techniques de réalisation d'architecture de lignes de produits.

3.1.1 La variabilité

Les parties communes d'une famille de produits consistent en une liste structurée d'hypothèses qui sont vraies pour chaque membre de la famille, ou peuvent être vues comme les parties partagées par tous les produits. *Weiss et Al* définissent la variabilité comme une hypothèse sur la façon dont les membres de la famille peuvent différer les uns des autres [Weiss and Lai, 1999]. Les variabilités spécifient les particularités d'un système correspondant aux attentes particulières d'un client.

La variabilité est souvent décrite avec des points de variation et des variants. Un point de variation localise une variabilité et ses possibles valeurs sont décrites par plusieurs variants. Chaque variant correspond à une alternative conçue pour réaliser une variabilité particulière. Selon *Bachmann et al* [Bachmann and Bass, 2001], les variabilités peuvent être classées selon plusieurs critères : variabilité de fonction, de données, de flot de contrôle, de technologie, d'objectifs de qualité, ou d'environnement du produit. Une autre classification est proposée par *Halmans et al* [Halmans and Pohl, 2003], qui se base plutôt sur une distinction entre variabilités essentielles (e.g. fonctionnelles, de l'environnement système, ...) et techniques (d'implémentation par exemple).

On définit par ailleurs souvent les catégories suivantes de variations :

- exigences obligatoires (supportées par tous les systèmes dans un domaine)
- exigences optionnelles (nécessaires uniquement dans certains systèmes)
- exigences alternatives (choix entre plusieurs exigences)
- exigences pré-requises (requises par une autre exigence).

La variation peut intervenir à différents niveaux :

- niveau ligne de produits
- niveau produit
- niveau composant
- niveau sous-composant
- niveau code.

3.1.2 Les exigences

L'ingénierie des exigences contient classiquement cinq phases qui doivent être revisitées dans le contexte des lignes de produits :

1. le choix des exigences
2. l'analyse des exigences
3. la spécification des exigences
4. la vérification des exigences
5. la gestion des exigences.

On décrira ici plus précisément la spécification des exigences, et donc en particulier la gestion de la variabilité à ce niveau.

On distingue plusieurs types d'approches dans la modélisation des exigences d'une ligne de produits, entre autres les approches orientées caractéristiques et orientées cas

d'utilisation.

Dans [Kuusela and Savolainen, 2000], l'approche proposée ne repose sur aucun mécanisme de modélisation particulier. Les auteurs proposent une hiérarchisation des exigences, chaque exigence contenant la priorité qui lui est associée pour chaque produit de la ligne. Chaque priorité reflète l'importance de chaque nœud exigence dans la réalisation de l'intention de ses parents dans la hiérarchie.

Les approches basées caractéristiques

Selon [van Gurp et al., 2001], la première étape pour interpréter et ordonner les besoins est de faire une analyse des caractéristiques (*features*). Ces caractéristiques peuvent varier selon les produits. Dans [Griss et al., 1998], trois types de variations sont identifiés :

- Les caractéristiques obligatoires, qui sont les caractéristiques qui identifient un produit ;
- Les caractéristiques optionnelles, qui, quand elles sont présentes, donnent une valeur ajoutée aux caractéristiques principales du produit ;
- Les caractéristiques variantes, qui correspondent à une manière alternative de configurer une caractéristique.

Une quatrième catégorie est ajoutée dans [van Gurp et al., 2001] :

- Les caractéristiques externes, qui sont les caractéristiques offertes par la plateforme cible.

Par ailleurs, on définit deux types de points de variations : le choix multiple (choix de plusieurs caractéristiques parmi plusieurs) ou le choix exclusif (choix d'une caractéristique parmi plusieurs).

On peut alors définir un arbre de caractéristiques pour la ligne de produits qui modélise les exigences de la ligne de produits.

Différentes façons de modéliser la variation avec des approches basées caractéristiques ont été proposées dans [Griss et al., 1998] pour l'intégration avec l'approche RSEB, dans [Kang et al., 1990] pour la notation FODA, dans [Kang et al., 2002] pour la notation FORM, ainsi que dans [van Gurp et al., 2001, Riebisch et al., 2002a].

Les approches basées cas d'utilisation

Les approches basées cas d'utilisation proposent soit des extensions à UML, soit de nouveaux stéréotypes UML pour représenter la variabilité.

Dans [Halmans and Pohl, 2003], les points de variations sont représentés dans les diagrammes de cas d'utilisation par des triangles (notation et symbole ajoutés à la notation UML par les auteurs). Le triangle est noir dans le cas d'un point de variation obligatoire (i.e. au moins un de ces variants doit être sélectionné) et gris clair si le point de variation est optionnel. Chaque point de variation est lié à un ensemble de variants. Les variants sont représentés par des cas d'utilisation stéréotypés $<<$variant$>>$ (stéréotype introduit par les auteurs). Les liens entre un point de variation et ses variants

possèdent une cardinalité, qui est 1..1 pour les variants obligatoires et qui peut par exemple être 0..2 pour un variant optionnel (choix de 0, 1 ou 2 des variants). Le type de variant vis-à-vis du point de variation est également stigmatisé par un petit rond gris clair ou noir sur le lien entre point de variation et variant. De plus, les mêmes auteurs définissent dans [Bühne et al., 2003] de nouvelles notations introduites pour modéliser les dépendances entre variants.

Les auteurs de [Gomaa and Shin, 2002] suggèrent d'utiliser principalement les notations UML existantes pour modéliser les lignes de produits, en introduisant juste les stéréotypes $<<$kernel$>>$, $<<$optional$>>$ et $<<$variant$>>$. De plus, la relation d'extension (*extend*) est utilisée pour représenter une variation dans les exigences au travers de chemins alternatifs qu'un cas d'utilisation de base peut emprunter si une condition particulière est vérifiée. Similairement, la relation d'inclusion (*includes*) modélise la variation par réutilisation d'un cas d'utilisation commun et utilisé par plusieurs autres cas d'utilisation. La seule notion introduite est celle de condition pour la sélection d'une caractéristique (*feature condition*) qui est utilisée pour représenter une caractéristique optionnelle (et qui sera sélectionnée si la condition est vérifiée).

Dans [Bertolino et al., 2002], deux approches sont proposées pour décrire les cas d'utilisation d'une ligne de produits. La première se base sur une modélisation à deux niveaux : le niveau ligne de produits qui s'instancie vers le niveau produits. Les scénarios associés au cas d'utilisation de niveau ligne de produits sont alors augmentés d'étiquettes (*tags*) pour y représenter la variation et en permettre l'instanciation pour un produit particulier. La deuxième représentation est à un seul niveau et propose de voir un cas d'utilisation comme incluant toutes les alternatives pouvant se produire. Un cas d'utilisation possède alors un scénario principal et une liste de scénarios alternatifs, une pré-condition principale et une liste de pré-conditions alternatives, etc.

John et al introduisent dans [John and Muthig, 2002] des cas d'utilisation dits génériques. Les parties variantes du diagramme de cas d'utilisation (dans l'exemple donné dans [John and Muthig, 2002], quatre cas d'utilisation ayant trait à un même composant optionnel), sont regroupées dans une partie grisée du système. Les cas d'utilisation variants sont stéréotypés comme tels, ainsi que les acteurs variants s'y rapportant. Cette notation ne nous paraît pas très intéressante ni applicable en pratique. Des labels proches de ceux introduits dans [Bertolino et al., 2002] sont insérés dans la description textuelle des cas d'utilisation.

3.1.3 Les architectures de ligne de produits

L'architecture d'un système logiciel définit ce système en termes de composants et de connexions entre ces composants [Show and Garlan, 1996]. Une ligne de produits logicielle est un ensemble de systèmes qui partagent une architecture logicielle commune et un ensemble de composants réutilisables [Bosch, 2000]. L'architecture d'une ligne de produits définit donc les concepts, la structure et la texture nécessaire pour réaliser la variation de caractéristiques des produits, tout en ayant un maximum de parties partagées au niveau de l'implémentation [Jazayeri et al., 2000].

Différents mécanismes existent pour permettre la variation [Bosch, 2000,

Svahnberg and Bosch, 2001] : l'héritage, les extensions, la configuration, la paramétrisation, la génération, ainsi que les directives de compilation. Plusieurs autres approches ont été proposées comme l'utilisation de patrons de conception [Barry Keepence, 1999], la transformation de modèle [Monestel et al., 2002], ou le tissage d'aspects [Griss, 2000].

3.2 Le test de lignes de produits

Dans la mesure où l'essor des lignes de produits est assez récent, et où les principaux concepts de modélisation commencent à peine à s'éclaircir, peu de méthodes de test de lignes de produits ont été proposées. Le test de lignes de produits pose pourtant de nouveaux problèmes : comment factoriser et capitaliser les tests ? quel plan de test pour une ligne de produits ? Plusieurs articles traitent de ces nouveaux problèmes, sans pour autant avoir vocation à en proposer une solution.

John McGregor propose dans [McGregor, 2001] une description du test de lignes de produits. Cette description met plus en avant les problèmes que pose le test de lignes de produits qu'il ne propose réellement de solutions, et les travaux décrits concernant le test de lignes de produits à proprement parler sont assez peu nombreux. De nombreux rappels sont faits sur le test de logiciel « classique », puis sont mis en parallèle avec le test de lignes de produits. Sont notamment décrits tous les produits issus de l'activité test et la spécificité qu'ils ont dans le cadre des lignes de produits. Les produits décrits sont les plans de test, les cas et données de test, les logiciels et scripts de test, et les rapports de test. McGregor part du principe que les produits d'une même ligne partagent un certain nombre d'avoirs conceptuels (ou assets), et propose donc de tester ces assets séparément d'une part, de tester ensuite chaque produit d'autre part. McGregor propose notamment une méthode de dérivation des cas de test à partir des exigences : les exigences sont dérivées en cas d'utilisation qui sont utilisés pour écrire des scénarios de test statiques, eux-mêmes ensuite transformés en cas de test dynamiques. L'idée que nous proposons est assez similaire pour la génération automatique de cas de test, à ceci près que nous nous basons sur une formalisation des cas d'utilisation qui nous permet un processus automatique plutôt qu'à forte intervention humaine. Concernant l'automatisation des tests, McGregor souligne leur importance cruciale dans le cadre des lignes de produits, mais insiste également sur sa difficulté, et détaille les scripts et harnais de test plus que la génération automatique de tests.

Henry Muccini et al [Muccini and van der Hoek, 2003] soulignent et discutent les nouveaux défis posés par les lignes de produits en matière de test. Concernant le test unitaire, chaque composant d'une ligne de produits doit être testé unitairement, et tous les variants compris dans le composant doivent l'être également. Une priorité dans l'effort de test peut toutefois être donnée. Pour le test d'intégration, une solution est d'intégrer dans un premier temps les classes centrales (communes à la ligne de produits), puis d'assembler les autres éléments. Les auteurs proposent une intégration big-bang : il nous semblerait plus judicieux de calculer automatiquement un plan de test adéquat par produit.

La méthode de *Bertolino et al* [Bertolino and Gnesi, 2003] se base sur une modéli-

sation des cas d'utilisation telle que décrite section 3.1.2, et applique une méthode de catégorie-partition. Les catégories sont identifiées manuellement à partir de la description de chaque cas d'utilisation, et sont partitionnées en choix sensés. Une spécification de test est ainsi obtenue. Un générateur est ensuite appliqué pour générer un ensemble de cas de test correspondant à toutes les combinaisons possibles de choix, pour toute la ligne de produits. Il est également possible, ce qui est plus intéressant, de ne dériver que les cas de test relatifs à un produit particulier.

Kamsties et al proposent dans [Kamsties et al., 2003] une méthode basée sur différentes stratégies selon la façon dont la variabilité apparaît dans le cas d'utilisation. Quatre stratégies sont identifiées :

- L'abstraction qui consiste à écrire des tests abstraits pour la ligne de produits, et ensuite à spécialiser ces tests pour chaque produit, en prenant en compte le choix des variants.
- La paramétrisation qui consiste à écrire des test paramétrés pour la ligne de produits, et ensuite d'affecter des valeurs concrètes aux paramètres pour chaque produit.
- La segmentation qui consiste à écrire une collection de segments de test pour la ligne de produits, un segment est ensuite sélectionné en prenant en compte le choix des variants pour former un test spécifique à chaque produit.
- La fragmentation qui consiste à écrire des fragments de test pour les parties communes et les parties variantes du cas d'utilisation, puis à les concaténer pour former un cas de test spécifique à chaque produit.

Selon le type de variabilité dans le cas d'utilisation (dans le flot d'événements, la pré-condition, la post-condition, les acteurs, les relations), les stratégies les plus adéquates sont discutées.

Contrairement à la méthode de *Bertolino et Al* et à la nôtre, la méthode proposée par *Kamsties et al* est systématique mais ne repose pas sur une automatisation de la génération de test.

Conclusion

Le test entraine un coût très important dans le développement d'un logiciel, et son automatisation reste une problématique cruciale. Dans cette partie, nous avons tout d'abord rappelé les problématiques majeures du test de logiciel, avant de nous concentrer sur la génération automatique de test. Nous avons ensuite évoqué les problématiques inhérentes aux lignes de produits logicielles, et les techniques de test existant pour les tester.

La génération automatique de tests est un enjeu majeur du test, en particulier dans le cadre des lignes de produits. Si de nombreux travaux s'intéressent à cette automatisation dans le cadre du test de logiciels « classiques », l'automatisation du test de lignes de produits ne fait pas encore l'objet de beaucoup d'études.

Parmi les nombreuses approches proposées pour la génération automatique de test à partir des exigences dont nous avons donné un aperçu dans le chapitre 2, peu sont applicables au niveau système. En effet, la plupart des approches proposées se basent sur un modèle très détaillé d'exigences dont la complexité rend impossible l'application au niveau système ou même composant. Si la plupart des techniques que nous avons présentées permettent une génération efficace de cas de test, peu d'entre elles peuvent être transférées facilement vers l'industrie. En effet, pour pouvoir être appliquée, une nouvelle approche de génération de test à partir des exigences doit prendre en compte la pratique industrielle concernant la rédaction et la gestion des exigences ainsi que l'utilisation de standards méthodologiques. C'est pourquoi un des problèmes que nous nous proposons de résoudre est la facilité d'accès aux méthodes de génération automatique de test. Pour cela, nous avons pris en compte la pratique industrielle (rédaction d'exigences en langage naturel, utilisation de cycle classiques de développement, ...) pour élaborer notre approche.

Par ailleurs, la pratique veut que les exigences d'un système varient très rapidement au cours de la conception du système, et les approches de génération de test existantes sont mal adaptées à de telles modifications, ou même à l'élaboration de nouvelles versions du même logiciel. De même, très peu d'approches permettent la génération de test pour les lignes de produits. Ainsi, les techniques de génération de test actuelles ont un coût qui reste grosso modo proportionnel à la taille du logiciel à tester, et ne prend en compte ni les différentes versions du logiciel qui restent pourtant proches, ni les lignes de produits. C'est un des points majeurs auquel nous nous proposons de contribuer.

Troisième partie

Contributions

Chapitre 4

Contexte et structuration des contributions

Ce chapitre a tout d'abord pour but de situer les recherches décrites dans ce manuscrit dans les différents contextes dans lesquels elles ont été menées. Nous introduirons également de manière très schématique les solutions que nous avons apportées au problème de la génération automatique de test système dans les contextes particuliers que nous avons traités. Cette brève introduction nous permettra de donner l'idée directrice de ce manuscrit, et d'en faciliter la lecture. L'approche que nous proposons est composée de trois composantes méthodologiques et outillées qui s'assemblent pour former une méthode cohérente permettant de dériver automatiquement des tests fonctionnels systèmes à partir des exigences. Nous présentons dans ce chapitre chacune des composantes et la manière dont elles s'imbriquent, de façon à ce qu'ensuite chaque composante puisse faire l'objet d'un chapitre indépendant.

4.1 Contexte

Les travaux de recherche qui ont donné lieu à ce manuscrit ont été menés dans le cadre de trois projets : deux projets européens sur les lignes de produits logicielles, et le troisième, français, collaboration entre Thalès, le Cea et l'Inria. Ces projets ont fortement orienté nos recherches, puisqu'ils les ont cadrées dans deux contextes industriels différents : le contexte des lignes de produits, et le contexte de développement à Thalès Airborne System (Tas).

4.1.1 Le projet Mutation

Le projet Mutation (Modélisation Uml pour l'automatisation de la production de test) fait partie d'un programme de recherche plus large, Carroll [Thalès et al., 2003], lancé par le groupe Thalès et traitant des technologies au cœur du développement logiciel pour les systèmes embarqués et de grande échelle. Ce programme de recherche regroupe plusieurs organisations : le Commissariat à l'Énergie

Atomique (CEA), l'INRIA, ainsi que deux composantes du groupe THALÈS : THALÈS Airborne System (TAS) et THALÈS Research and Technology (TRT).

L'objectif du projet MUTATION est d'étudier la faisabilité du transfert des méthodes académiques de génération de test vers TAS, de manière à y automatiser la tâche de test, largement manuelle jusqu'alors. L'étude de faisabilité de l'automatisation des procédures de test portait sur trois points essentiels : l'assistance à la définition de scénarios de tests à différents niveaux d'abstraction, la génération de cas de test, et l'aide à leur interprétation à travers de critères de couverture. Les objectifs techniques du projet étaient de :

1. définir les règles de formalisation des exigences,
2. définir des règles de formalisation de la conception détaillée,
3. permettre la génération automatique de tests à différents niveaux d'abstraction,
4. assurer un coût réduit de formation et d'exploitation pour une équipe industrielle.

Notre contribution a essentiellement porté sur les points 1 et 3, tout en prenant en compte les contraintes associées au point 4.

TAS développe les systèmes de navigation des armements de la dernière génération de l'aviation de combat (Mirage 2000-9 et Rafale). Pour permettre aux équipages de mener à bien leurs missions dans un contexte toujours plus complexe, ces systèmes assurent de plus en plus de fonctions, et utilisent à leur maximum les processeurs et les capacités des logiciels embarqués. Le projet MUTATION s'est focalisé sur les systèmes de navigation des armes, dont TAS développe le processeur principal et les logiciels d'application. Ces logiciels assurent la gestion des missions (gestion de vol, navigation, interaction homme-machine, ...) et la gestion des systèmes d'armement (radar, systèmes d'auto-protection, ...). TAS développe cette famille logicielle depuis les années 70, et utilise un processus objet basé sur la notation UML.

Dans ce contexte, d'une part les exigences des clients deviennent de plus en plus pressantes, ce qui force le processus de développement à être réactif et en constante évolution. D'autre part, les systèmes devenant de plus en plus complexes, le logiciel joue une part de plus en plus importante, en termes de code et de délai. Cela implique une amélioration de la productivité, en particulier concernant le développement logiciel. Pour s'assurer du niveau de qualité désiré par les clients, plus de 50% de ce développement est dédié à la phase de test. L'augmentation de la productivité du test est donc un axe majeur d'amélioration pour TAS. Dans la mesure où les spécifications des systèmes sont écrites en langage naturel, la difficulté majeure durant la phase de validation est leur interprétation correcte pour construire les cas de test.

Dans l'état actuel des choses, les tests à TAS sont écrits manuellement, à partir des spécifications textuelles du système et des éléments de traçabilité avec l'implémentation. L'objectif du projet MUTATION était de permettre une automatisation, même partielle, de cette tâche de test.

4.1.2 Les projets CAFÉ et FAMILIES

Les 2 projets CAFÉ [Café, 2003] et FAMILIES [Families, 2004] ont pour thème central les lignes de produits logicielles. De nombreuses composantes y participent, tant académiques (entre autres : Université de Madrid, Université d'Essen, Université d'Helsinki) qu'industrielles (comme Nokia, Thalès, Siemens). Le projet FAMILIES fait suite au projet CAFÉ, qui lui même prolongeait le projet ESAPS. Le projet CAFÉ avait pour but de mettre en œuvre les concepts identifiés par le projet ESAPS (formant ce qui est appelé le *6-pack*), à travers l'élaboration de méthodes et d'outils. Notre travail s'inscrivait dans le groupe de travail consacré à la validation et au test de lignes de produits, dont le but était d'aborder les problèmes de conception en vue de la testabilité, d'automatisation des efforts de test, et de la réutilisation des résultats de test dans un contexte de lignes de produits. Le projet FAMILIES se focalise sur l'institutionalisation, la pertinence métier, la standardisation et la dissémination. Nous travaillons dans le groupe de travail sur l'ingéniérie des familles de produits basée sur les modèles dont le but est de proposer une méthodologie supportant la séparation des aspects propres au domaine, des aspects techniques (qualité de service) et des aspects technologiques (plate-formes) dans une approche MDA (Model-Driven Architecture) [Soley and OMG, 2000]. Nous nous intéressons plus particulièrement au test basé sur les modèles.

4.2 Approche proposée

Les problématiques de TAS et celles inhérentes aux lignes de produits logicielles nous ont guidé dans notre approche de génération de test. Notre objectif était de générer des cas de test système à partir des exigences d'un système. Dans notre approche, nous avons cherché à atteindre quatre objectifs.

1. Nous avons cherché à élaborer une méthode flexible. Nous voulions pouvoir regénérer facilement des tests après modifications des exigences. En effet, les exigences d'un système, et en particulier celles d'une ligne de produits, changent très vite. Pour certains de ses produits, l'entreprise Nokia a constaté que 69 % des exigences du cahier des charges avaient été modifiées, dont 22 % avaient été modifiées deux fois, même après fixation du cahier des charges définitif. De manière à réagir à de telles modifications du cahier des charges dans les meilleurs délais, il faut pouvoir rapidement valider le nouveau produit. C'est pour cela que dans notre démarche, nous avons cherché à ce que la génération des tests pour les nouvelles exigences soit elle aussi la plus rapide possible.
2. Le coût de modélisation a aussi été pris en compte. L'utilisation de spécifications formelles permet en effet de générer des tests ou d'effectuer des validations des exigences. Cependant, la rédaction ainsi que la modification d'exigences dans un langage de spécifications comme B ou Z est très coûteuse. Dans le cadre des lignes de produit, ce coût est encore démultiplié, et toutes les entreprises ne sont pas prêtes à le payer. Nous avons donc cherché à ce que notre approche n'ait pas un point d'entrée trop coûteux.
3. Pour ce faire, nous avons considéré que les modélisations nécessaires à notre tech-

nique de génération de test devaient être d'un accès assez intuitif, de manière à ce qu'il n'y ait pas besoin de formation particulière. Nous avons également cherché à nous adapter aux pratiques existantes, de manière à ce que l'adoption de notre approche dans un contexte industriel soit le plus facile possible, et ne bouleverse pas les pratiques de l'entreprise.

4. Nous avons pris en compte les contraintes liées aux lignes de produits. Pour pouvoir être intéressante, une approche par ligne de produits doit pouvoir permettre de mettre sur le marché un nouveau produit fiable en un temps minimum. Pour cela, une phase importante est la réduction du coût de test du nouveau produit, notamment le test à partir des exigences du produit. Quand on sait que Nokia estime qu'à un instant donné, coexistent 10 000 versions différentes d'un même logiciel, on voit bien qu'il est impensable de développer pour chaque nouveau produit de nouvelles spécifications formelles, car le coût serait alors trop important, et le délai de mise sur le marché très long. Nous avons donc cherché une approche qui permette de s'adapter facilement à des exigences variables, et de générer rapidement des tests spécifiques à chaque produit en fonction des spécificités du produit en question.

Pour nous adapter au contexte industriel traité, nous sommes parti des pratiques existantes, et avons proposé des méthodes et des outils permettant de pousser ces pratiques industrielles vers des modèles plus rigoureux, qui sont ensuite la base de notre mécanisme de génération de tests. Pour rester en adéquation avec les pratiques industrielles, nous nous sommes basé sur la modélisation UML comme support à un processus de développement orienté à objets. Les modèles semi-formels que nous proposons d'utiliser sont ainsi familiers aux industriels, et, en leur donnant une sémantique non ambiguë, nous avons permis de les utiliser comme des modèles formels. Nos modèles sont sans aucun doute moins élaborés que bien des modèles formels, mais ils ont l'avantage de pouvoir être maîtrisés tant par des outils de génération de tests, que par les industriels qui contrôlent ainsi de bout en bout l'approche de génération de tests.

Plus concrètement, le processus de génération de tests système à partir des exigences se décompose en trois étapes plus simples, chacune étant assistée par un outil. La figure 4.1 illustre cette approche en trois étapes.

Première étape : spécification des exigences et dérivation d'un modèle de cas d'utilisation

La première étape est une semi-formalisation des exigences. La pratique la plus largement répandue pour les exigences d'un système est la spécification en langage naturel. L'avantage du langage naturel est bien sûr de ne pas nécessiter d'autre compétences que celles du domaine métier traité, en particulier, il ne nécessite aucune faculté de formalisation. Des cahiers des charges en langage naturel peuvent ainsi être rédigés par les clients du système, et sont compris par le plus grand nombre. Les inconvénients de cette pratique sont par ailleurs légion, le plus important étant sans doute l'ambiguïté. Une exigence écrite en langage naturel donne souvent lieu à de nombreuses interprétations. Or ces exigences sont la base de plusieurs phases de développement :

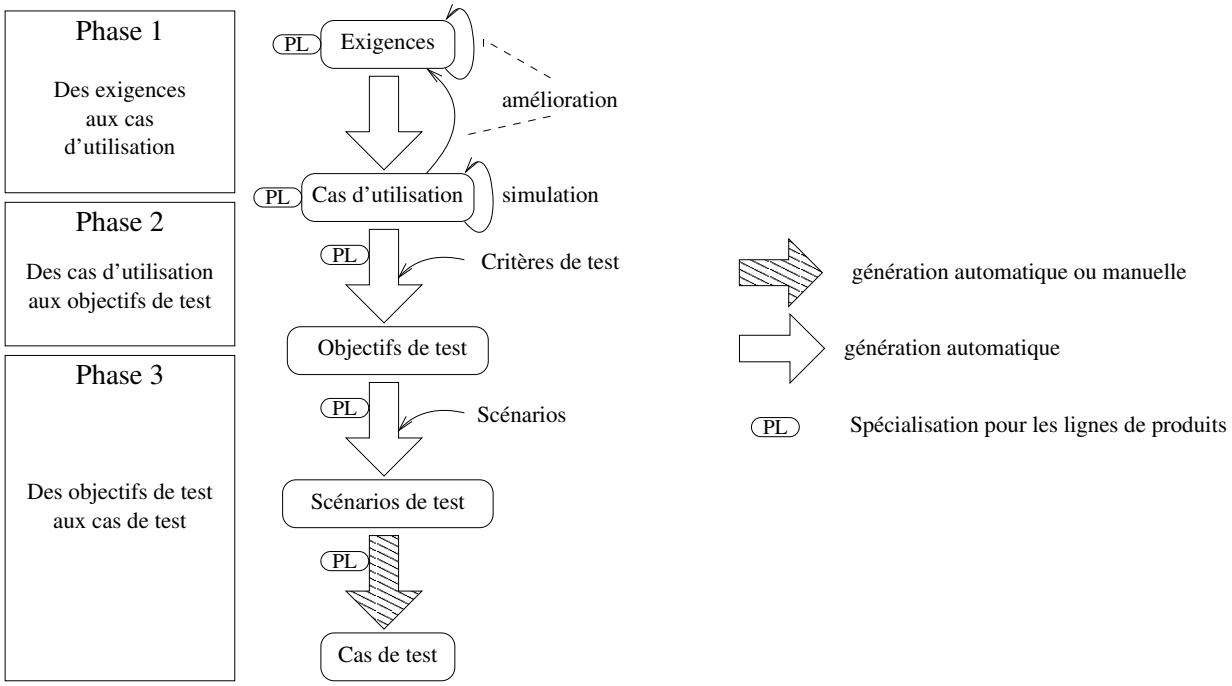


FIG. 4.1 – Approche de génération de test à partir des exigences en 3 étapes

l'élaboration des premiers modèles d'analyse et l'écriture de tests systèmes. Le danger est bien sûr qu'au cours de ces deux phases, une même exigence soit interprétée différemment par différents acteurs, et même différemment de l'interprétation initiale qu'en avait le rédacteur de l'exigence. De manière à pouvoir à la fois conserver l'utilisation du langage naturel et supprimer les dangers liés aux ambiguïtés, nous proposons un processus d'amélioration progressive des exigences, basé sur la définition d'un langage naturel contrôlé. La sémantique du langage est donnée en termes de cas d'utilisation augmentés. Ces cas d'utilisation ont été définis pour pouvoir être simulés, ce qui permet de simuler les exigences. Ainsi, les exigences sont écrites dans un langage contrôlé dédié, et peuvent être automatiquement interprétées et simulées. L'interprétation et la simulation permettent d'améliorer les exigences en en supprimant les ambiguïtés, et en s'assurant de leur complétude et de leur cohérence.

Deuxième étape : simulation des cas d'utilisation et génération d'objectifs de test

Les exigences obtenues à l'issue de ce processus sont ensuite transformées en un modèle de cas d'utilisation, et alors débute la seconde étape de notre approche. L'objectif de cette étape est la génération automatique d'objectifs de test pertinents. Le modèle de cas d'utilisation que nous utilisons rend explicite les différentes entités mises en jeu par le cas d'utilisation sous forme de paramètres. Il explicite également les conditions d'application du cas d'utilisation à l'aide d'une pré-condition, et les conséquences du cas d'utilisation sur le système et son environnement, à l'aide d'une post-condition.

Ces contrats sont exprimés à l'aide d'un langage de contrats dédié ; ce langage est néanmoins simple d'utilisation et ne nécessite pas de compétences particulières hormis la connaissance de la logique du premier ordre. Ce modèle de cas d'utilisation est la base d'un modèle de simulation des exigences. La simulation permet de s'assurer de la validité et de la complétude des exigences. C'est également le principe sous-jacent de la génération d'objectifs de test. En effet, une simulation exhaustive permet de construire un modèle de tous les ordres possibles de cas d'utilisation. Des critères sont alors définis pour sélectionner dans un tel modèle des chemins pertinents qui nous servent d'objectifs de test. Ces objectifs de test sont d'un niveau de modélisation similaire à celui des exigences. Pour nous rapprocher des cas de test, nous utilisons une troisième étape de raffinement des objectifs de test vers les cas de test.

Troisième étape : des objectifs de test aux cas de test

La troisième étape consiste à transformer les objectifs de test vers les cas de test, et nécessite donc d'utiliser des informations sur la façon dont les cas d'utilisation sont implémentés dans le système final. Plus exactement, il est nécessaire de savoir quels messages doivent être échangés entre le système et l'environnement de manière à réaliser le service rendu par le cas d'utilisation. Nous proposons d'utiliser des scénarios sous forme de diagramme de séquence. De tels scénarios, attachés aux cas d'utilisation, permettent d'obtenir l'information nécessaire pour dériver les objectifs de test en scénarios de test. Les objectifs de test sont sous forme de séquences de cas d'utilisation. En remplaçant chaque cas d'utilisation par un scénario représentant les échanges de messages nécessaires pour réaliser le cas d'utilisation, on obtient alors un scénario de test relativement proche d'un cas de test. Pour obtenir un cas de test directement exécutable par un pilote de test, on peut alors utiliser un outil de synthèse de test, ou dériver les cas de test manuellement.

Structuration du manuscrit et articulation des contributions

Chacune de ces trois phases fait l'objet d'un chapitre. L'approche proposée est indépendante des lignes de produits, même si leurs contraintes intrinsèques ont été prises en compte. Une spécialisation explicite de chacune des phases pour les lignes de produits est donc proposée dans un chapitre indépendant.

Le chapitre 5 détaille le processus d'amélioration des exigences textuelles : il présente le langage contrôlé que nous proposons, et explique comment une sémantique lui est attachée. La sémantique du langage étant donnée en termes d'un modèle de cas d'utilisation simulables, ce chapitre explique également à quelles fins la simulation peut améliorer les exigences, sans détailler le mécanisme de simulation ni le modèle de cas d'utilisation sous-jacent – décrits dans le chapitre suivant.

Le chapitre 6 détaille le modèle de cas d'utilisation utilisé, le modèle de simulation et la génération d'objectifs de test. Différents critères de test sont définis, afin de permettre de sélectionner des objectifs de test sous forme de séquences de cas d'utilisation avec leurs paramètres effectifs.

Le chapitre 7 expose l'utilisation de scénarios pour générer des séquences et des cas de test selon deux approches : une approche entièrement automatique basée sur un outil de synthèse de tests, et une approche comprenant une phase de dérivation manuelle.

Les résultats obtenus par application de cette approche sur des cas d'étude seront présentés dans le chapitre 8.

Le chapitre 9 détaille comment nous avons spécialisé notre approche aux lignes de produits.

Nous illustrerons ces quatre chapitres avec le même exemple, que nous présentons dans la section suivante.

4.3 Présentation d'un exemple illustratif

L'exemple que nous utiliserons tout au long de ce manuscrit est un système de réunions virtuelles. Les réunions tendent à imiter le plus possible des réunions de travail : elles doivent être ouvertes, fermées, et on peut y entrer et en sortir. Les réunions sont planifiées par leur organisateur, qui en fixe les principales caractéristiques (son nom, sa date, son ordre du jour). Trois types de réunions peuvent être organisées. Pour intervenir dans une réunion, un participant se connecte au serveur et y rentre.

- Une réunion standard est animée par un animateur qui désigne l'orateur courant parmi tous les participants qui ont demandé la parole. L'animateur est nommé par l'organisateur, et a aussi la charge d'ouvrir et de fermer la réunion.
- Une réunion démocratique est une réunion standard ne possédant pas d'animateur : l'ordre d'obtention de la parole est l'ordre dans lequel les participants ont demandé à y parler. Ces réunions s'ouvrent et se ferment spontanément.
- Une réunion privée est une réunion standard dont l'accès est limité à une liste de participants nommés par l'organisateur.

La démarche classique de développement d'un tel système est illustrée Figure 4.2. Un diagramme de cas d'utilisation tel que celui présenté Figure 2.1 est d'abord construit à partir des exigences et il est documenté par des scénarios. Puis des modèles de classes de différents niveaux de précision sont conçus, et guident l'implémentation. Alors intervient la phase de test système, pour laquelle les tests systèmes sont écrits à la main en prenant en compte les exigences, les cas d'utilisation, et les scénarios précédemment conçus. L'approche que nous proposons dans ce manuscrit permet deux automatisations dans cette démarche de développement ; elles sont représentées dans la figure 4.2 par des flèches en pointillés. La première consiste à générer des cas d'utilisation étendus à partir des exigences si celles-ci sont écrites dans un langage contrôlé que nous proposons. La seconde consiste à générer automatiquement les tests système à partir des cas d'utilisation et des scénarios.

Dans le chapitre 9, nous introduirons de nouvelles fonctionnalités au système de réunions virtuelles, et nous définirons un certain nombre de points de variation menant à des milliers de produits potentiels. On voit bien alors que la démarche classique de la Figure 4.2 avec l'écriture manuelle des tests système ne pourra être appliquée pour

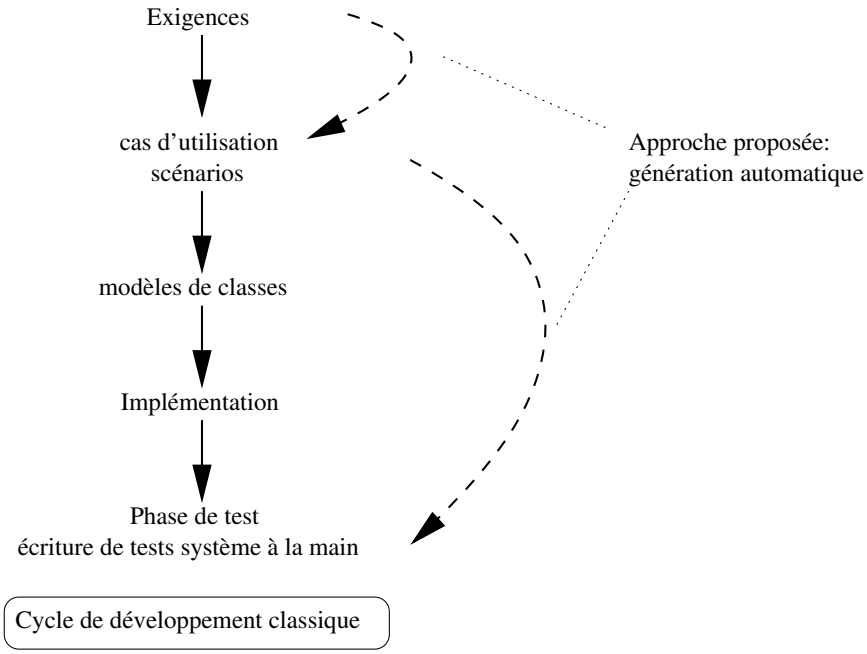


FIG. 4.2 – Démarche classique pour construire le système de réunions virtelles

chaque produit qu'au prix d'un coût énorme. Nous montrerons alors comment notre approche permet la génération automatique de tests système pour l'ensemble de la ligne de produits et à faible coût.

Chapitre 5

Des exigences aux cas d'utilisation

La continuité dans le processus de raffinement des spécifications d'un système vers sa modélisation, son implémentation et sa validation, est un problème crucial, en particulier pour les systèmes de grande taille. En effet, les premières étapes de spécification des exigences, écrites en langage naturel, doivent être intensivement utilisées pour la conception et le test du produit final. Ce problème de continuité concerne tant les aspects d'analyse et traitement du langage naturel, que les aspects de complétude et de cohérence des exigences. Ces problèmes ne sont pas originaux, et sont difficiles à résoudre sans utilisation de méthodes formelles pour exprimer les exigences.

La pratique industrielle courante est de rédiger les exigences d'un logiciel en langage naturel ou en langage contrôlé, c'est-à-dire un langage naturel restreignant l'usage de certaines constructions du langage trop susceptibles d'apporter des ambiguïtés, ou au contraire imposant l'usage d'un certain nombre de constructions. Les problèmes suscités par ce type de pratique concernent l'ambiguïté du langage naturel – même restreint – la complétude des exigences, et leur cohérence. En effet, quand on a un grand ensemble d'exigences, il est difficile de s'assurer que celles-ci ne peuvent pas conduire à de multiples interprétations, qu'elles n'omettent pas de spécifier certains cas, ou qu'elles sont bien toutes cohérentes les unes avec les autres.

Des solutions basées sur l'utilisation de méthodes formelles ont été proposées ; soit elles préconisent de rédiger les exigences directement sous forme de langage formel, soit elles proposent des mécanismes de traduction du langage naturel vers des langages formels :

- Concernant la spécification directe en langage formel, le problème vient du fait qu'il n'est pas envisageable dans tous les contextes industriels de disposer de personnel compétent dans les langages formels, et les spécialistes du domaine ne sont pas toujours des informaticiens. De plus, résoudre le problème par l'utilisation de langages comme Z pour pallier les ambiguïtés du langage naturel reste un peu utopique, dans la mesure où comme le soulignent *Finkelstein et Emmerich* [Finkelstein and Emmerich, 2000], la plupart des canevas formels comme Z sont accompagnés de larges parts de langage naturel, et sont inutilisables sans elles. Une contrainte supplémentaire dans la résolution du problème est l'indépendance vis-à-vis du domaine industriel traité : proposer un langage contrôlé orienté vers

le domaine traité est tentant, mais bien sûr la méthode ne peut alors pas être transférée rapidement à un autre domaine industriel.

- Plusieurs travaux abordent la traduction des exigences en langage naturel vers un langage formel.
  Fantechi et al proposent ainsi dans [Fantechi et al., 1994] une traduction automatique du langage naturel vers le langage de logique temporelle ACTL. La traduction s'appuie sur un dictionnaire rempli par l'utilisateur. À partir de la formalisation en ACTL, des ambiguïtés peuvent être détectées : un manque d'information, ou la portée des conjonctions qui ne peuvent pas être résolues par le parenthésage du langage naturel.
  Les auteurs de [Ambriola and Gervasi, 1997] proposent un analyseur du langage naturel qui permet de produire différents types de modèles comme des modèles de flot de données, des diagrammes OMT, etc. L'analyse des différents modèles produits permet alors de détecter un certain nombre d'anomalies dans les exigences : l'inconsistance des diagrammes de flots de données, l'ambiguïté, et la redondance.
  Fuchs et al se basent sur le fait qu'un langage contrôlé peut remplacer la logique du premier ordre [Fuchs et al., 1999b], et proposent un tel langage, et le moyen de le transformer en logique du premier ordre [Fuchs et al., 1999a].

L'approche que nous proposons vise à découpler la syntaxe du langage de description des exigences proposé de sa sémantique. Les exigences sont d'abord analysées syntaxiquement, la sémantique étant associée dans un second temps, en utilisant des *patrons d'interprétation* qui sont des structures définies par les experts du domaine pour associer des éléments de syntaxe avec une sémantique non ambiguë. Dans la mesure où cette approche a été proposée dans l'optique de faire de la génération de tests à partir des exigences, nous proposons une sémantique en termes de cas d'utilisation contractualisés, ce qui nous permet de pouvoir accéder aux principes de simulation et de génération de test exposés au chapitre 4 et qui seront détaillés dans le chapitre 6. Être capable de simuler les exigences permet de les valider et de s'assurer de leur cohérence, et nous paraît être une caractéristique importante de cette approche : si la littérature propose pléthore de méthodes de validation des exigences exprimées avec un langage formel, aucune ne garantit l'équivalence entre ce qui est écrit dans les exigences formalisées et les exigences fournies par le client, ou plus largement les souhaits du client.

Dans cette section, nous décrirons les grandes lignes de notre approche, puis nous détaillerons le langage de description des exigences que nous proposons et les patrons d'interprétation lui donnant une sémantique non ambiguë.

5.1 Vue globale et méthodologique de l'approche

L'approche que nous proposons vise à traiter des exigences fonctionnelles informelles et à en détecter les inconsistances de manière à ce que les exigences puissent être améliorées. L'approche sépare l'analyse syntaxique de la connaissance des experts du domaine, elle est donc en ce sens indépendante du domaine. Les résultats escomptés sont doubles : d'une part les exigences sont complétées et rendues non-ambiguës pour qu'elles

puissent servir à des fins de simulation, d'assistance à la conception, etc. et, d'autre part, un ensemble de règles d'interprétation sémantique est obtenu incrémentalement et peut servir à la conception de règles non ambiguës d'écriture des exigences. Ces règles seront par la suite appelées *patrons d'interprétation* et seront définies plus précisément Section 5.3 : ce sont des règles de traduction du modèle syntaxique au modèle sémantique.

Ainsi, le point d'entrée de notre approche est un ensemble d'exigences textuelles écrites dans notre Langage de Description des Exigences (LDE), qui, même s'il restreint l'usage du langage naturel, permet malgré tout d'écrire des phrases ambiguës, incomplètes, voir incorrectes. Ce langage est très proche de la manière de décrire les exigences chez TAS. La contrainte la plus forte à respecter est de ne pas modifier radicalement la pratique de rédaction des exigences. Les objectifs sont alors :

- d'unifier un large ensemble d'exigences vers un plus petit ensemble de cas d'utilisation représentant la synthèse des services proposés par le système,
- de rejeter les exigences ambiguës ou incomplètes,
- et d'identifier les acteurs du système et les exigences qui ne sont pas du bon niveau d'abstraction.

Pour atteindre ces objectifs, nous proposons que les exigences soient rédigées en suivant un processus d'amélioration incrémental, de manière à ce qu'elles atteignent un niveau de précision équivalent à celui d'une approche formelle. Nous pensons qu'un processus incrémental de raffinement des exigences est plus adapté qu'une approche où les exigences sont considérées écrites directement correctes et cohérentes. En effet, s'il est facile pour un rédacteur d'exigences d'écrire « d'un seul trait » un ensemble d'exigences cohérentes, cette tâche est beaucoup plus ardue dans le cas où la rédaction des exigences est le fruit de la collaboration d'une équipe de 50 rédacteurs, et où il faut décrire les comportements attendus de systèmes complexes. Ainsi, après une première étape d'écriture d'exigences, celles-ci sont progressivement modifiées de manière à les rendre complètes, non ambiguës et cohérentes, ce qui est difficile à atteindre en une seule étape.

Cette approche incrémentale est présentée Figure 5.1, et se compose de trois étapes principales : l'analyse syntaxique, l'interprétation, et la simulation des exigences.

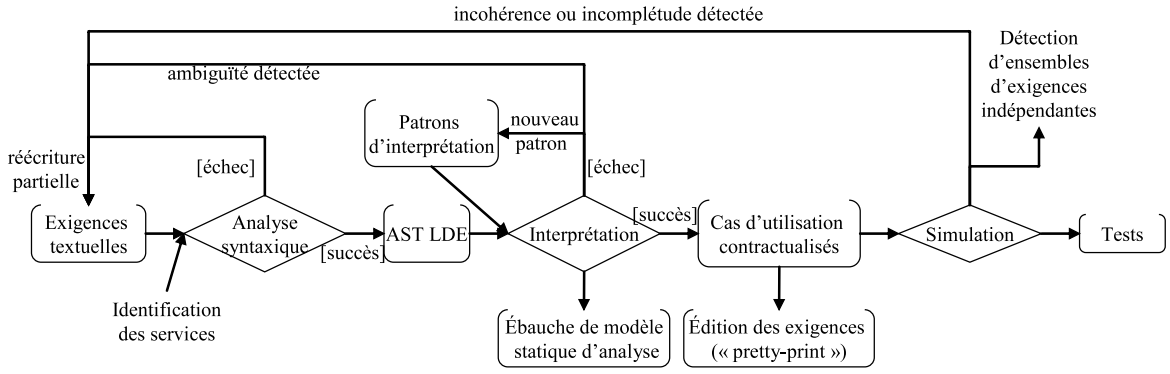


FIG. 5.1 – Approche incrémentale d'amélioration des exigences

La phase d'analyse syntaxique

L'analyse syntaxique produit un arbre syntaxique abstrait (AST, Abstract Syntax Tree) représentant les exigences. Si en pratique on produit un unique arbre pour toutes les exigences, il est plus intuitif de considérer un AST LDE par exigence. Si une exigence ne peut pas être analysée syntaxiquement, elle doit être modifiée jusqu'à ce qu'elle soit correcte vis-à-vis de la syntaxe LDE.

La phase d'interprétation

La phase d'interprétation a pour but de déterminer l'interprétation de l'AST LDE issue de l'analyse syntaxique. Le principe est pour cela d'appliquer des patrons d'interprétation qui fixent la sémantique associée au LDE : certains sont définis pour tous les types d'exigences, d'autres sont définis par des experts du métier et sont spécifiques à un domaine d'application particulier. Les patrons d'interprétation lient les structures de l'AST LDE à une sémantique formelle. Ces patrons d'interprétations sont similaires aux règles de déduction utilisées dans la plupart des analyseurs de langage naturel : un patron d'interprétation définit la sémantique de chaque structure du langage LDE en terme de cas d'utilisation enrichis.

Si une exigence ne peut être interprétée à l'aide des patrons d'interprétation, alors :

- soit cette exigence n'a pas de signification telle qu'elle est écrite, et doit donc être réécrite
- soit un nouveau patron d'interprétation doit être défini et ajouté à l'ensemble des patrons d'interprétation existants.
- soit l'exigence n'est pas d'un niveau correct d'abstraction (par exemple si l'exigence décrit le *comment* plutôt que le *quoi* : si elle décrit comment le système fonctionne et non pas ce que fait le système).

Il se peut également que l'exigence ait de multiples interprétations. Le rédacteur des exigences doit alors choisir l'interprétation correcte, et si possible modifier l'exigence textuelle pour supprimer l'ambiguïté de la phrase. Cette ambiguïté est classiquement due à la combinaison d'opérateurs logiques et temporels, sans parenthésage explicite. L'interprétation peut déboucher sur deux types de modèles : un modèle fonctionnel, et une ébauche de modèle statique.

- Le modèle fonctionnel se compose de cas d'utilisation contractualisés, c'est-à-dire augmentés de pré et post conditions (le modèle de cas d'utilisation utilisé est décrit dans le chapitre suivant). Ce modèle reflète en fait la sémantique associée aux exigences. Ces cas d'utilisation sont issus de l'agrégation et de l'unification automatiques des exigences. Il est à noter qu'à ce stade, les cas d'utilisation déduits peuvent être reformatés sous forme textuelle, par rétro-ingénierie, de manière à vérifier la compatibilité entre l'interprétation formelle des exigences, et ce que le rédacteur de l'exigence avait en tête (sortie « pretty printing »).
- L'ébauche de modèle statique est une synthèse des différentes composantes du système telles qu'elles peuvent être déduites des exigences. Pour rester dans le formalisme UML, cette ébauche est présentée sous forme de diagramme de classes d'analyse stéréotypé. L'ébauche n'est pas à proprement parler un diagramme de

classes d'analyse, mais c'est un support à la conception d'un tel diagramme. Cet aspect est illustré Section 5.4.

La phase de simulation

Le modèle de cas d'utilisation qui est produit par la phase d'interprétation est simulable. Les principes de la simulation seront exposés dans le chapitre suivant.

Pratiquée exhaustivement, la simulation permet tout d'abord de détecter les exigences totalement déconnectées des autres exigences. De telles exigences doivent être étudiées pour s'assurer qu'elles ont bien lieu d'être dans le système décrit, s'il n'y a pas lieu de scinder le système en sous-systèmes plus ou moins indépendants, ou si des liens existant entre les exigences n'ont pas été omis ou sous-spécifiés.

La simulation est également et surtout un moyen de valider les exigences, ou d'y détecter des erreurs ou des manques. Ainsi, la simulation permet l'amélioration et la complétion des exigences.

La simulation permet aussi de générer des tests à partir des exigences, comme nous le verrons dans le chapitre suivant.

La traçabilité au centre du processus

La traçabilité est une problématique centrale de notre processus : tout au long des différentes transformations à partir des exigences textuelles, des informations de traçabilité sont conservées ; ainsi, à tout moment, on peut déterminer pour chaque élément construit par nos transformations quelle est l'exigence ou quel est l'ensemble d'exigences qui lui a donné naissance. Nous assurons bien sûr la traçabilité entre cas d'utilisation et exigences, ce qui est un minimum. Nous permettons également la traçabilité entre le modèle statique et les exigences. En effet, l'ébauche de modèle statique que nous générons contient toutes les informations de traçabilité nécessaires : elles peuvent être réutilisées lors de la conception des modèles statiques. Enfin, puisque nous générons des tests à partir des cas d'utilisation comme cela sera détaillé dans le chapitre suivant, chaque test généré se réfère à la liste d'exigences concernées.

5.2 Le LDE, langage de description des exigences

Cette section présente les principaux éléments de syntaxe du LDE. Le LDE a été originellement défini pour les systèmes d'aviation THALÈS ; cependant il n'est pas spécifique à ce domaine d'application.

Il est important de noter que dans notre présentation du LDE, nous considérons implicitement que sa sémantique existe, mais elle est ambiguë avant la définition des patrons d'interprétation. La sémantique implicite est celle du langage naturel. En ce sens, le LDE ne définit qu'une formalisation structurelle des exigences, c'est-à-dire une syntaxe : il n'y a pas de sémantique particulière associée à un AST LDE. La sémantique est définie par les patrons d'interprétation qui sont appliqués à l'AST LDE. De cette

façon, d'une part on sépare la syntaxe – donnée par les experts du domaine – de la sémantique, et d'autre part, différentes applications peuvent utiliser l'AST de manière à construire différents types de productions, sans que la grammaire du LDE ne soit « polluée » par des besoins spécifiques de l'une ou l'autre des applications.

Dans cette section, le contexte et les principes ayant mené à la conception du LDE sont tout d'abord rappelés, le LDE est ensuite décrit.

5.2.1 Contexte et principes

Dans un contexte industriel, les ingénieurs en charge de la rédaction des exigences ont souvent une forte connaissance du domaine, mais peu ou pas de connaissance des langages formels. Le problème est alors que les exigences doivent être écrites de manière informelle et ne peuvent pas être utilisées comme base pour un processus automatique comme la simulation des exigences ou la génération de test.

Par ailleurs, rédiger des exigences n'implique pas nécessairement d'utiliser des outils sophistiqués conçus par des informaticiens pour des informaticiens, comme les ateliers B [atelierB, 2004].

L'idée est alors de proposer un langage facile à utiliser par les rédacteurs d'exigences, c'est-à-dire ne nécessitant qu'un apprentissage rapide, tout en fournissant une formalisation des exigences. Le langage des exigences que nous proposons et décrivons dans cette section a été conçu dans cette optique, et a les caractéristiques suivantes :

- c'est un langage pseudo-naturel (un langage naturel contrôlé), dans le sens où c'est un sous-ensemble de constructions provenant du langage naturel (l'anglais). Son apprentissage est de ce fait facile pour des personnes non habituées aux langages formels ;
- c'est un langage indépendant du domaine d'application : même si ce langage a été construit dans un contexte industriel spécifique, il n'y est absolument pas restreint ;
- c'est un langage conçu pour décrire les exigences de haut-niveau d'un système. Les exigences sont écrites en utilisant des entités comme des acteurs, des actions et des propriétés du système décrit ;
- le LDE fournit des moyens d'organiser les exigences. En particulier, le système complet peut être décrit comme un composant contenant des sous-composants.

5.2.2 Description du LDE

Le fait que le LDE est indépendant d'un quelconque domaine d'application impacte directement sa grammaire, qui ne doit contenir aucun mot clef spécifique à un domaine particulier. Les terminaux du langage sont de deux types : les terminaux de la grammaire, et les termes du domaine. Nous n'utilisons pas de dictionnaire contenant les termes du domaine, et ceux-ci doivent donc être identifiés par un autre biais. Ce problème est résolu dans le LDE en marquant (ici en mettant entre guillemets) tous les mots du domaine. Ces mots spécifiques au domaine ne font pas partie de la grammaire du langage, c'est le rédacteur des exigences qui les définit comme tels en les mettant

entre guillemets, comme cela est illustré dans l'exemple suivant.

Exemple 1. `Before a “manager” does “plan” a “meeting”, this “manager” must be “connected”.`

Dans cet exemple, chaque mot spécifique à l'exemple de réunion virtuelle est placé entre guillemets. Les autres mots de cette phrase exemple font partie du LDE, et n'ont qu'une signification grammaticale. Les mots placés entre guillemets sont considérés comme des terminaux du langage et ne sont donc pas analysés. Cette approche implique que les structures fournies par le LDE sont des squelettes de structures grammaticales extraites du langage naturel (ici de l'anglais). De ce point de vue, analyser une phrase LDE va générer une formalisation des exigences reflétant leur structure grammaticale. Dans notre cas, cette formalisation est un AST. Un aperçu du modèle de l'AST est donné Figure 5.2 sous forme d'un diagramme de classe UML. Des vues plus détaillées sont données en annexe page 183.

Le LDE a été construit dans le but de rester aussi proche que possible du langage naturel, et le langage s'appuie donc sur des constructions de phrase de type sujet/verbe. Dans la mesure où un autre but du LDE est de décrire un système à un haut niveau, le couple sujet/verbe doit être naturellement compris comme un couple acteur/action. Le rédacteur des exigences peut définir une propriété sur un objet du domaine, et le langage lui permet d'utiliser des liens logiques entre ces propriétés telles que la négation, la conjonction, et la disjonction, comme cela est illustré dans l'exemple suivant.

Exemple 2. Propriétés et opérateurs

```
- The “participant” is “connected”.
- The “status” of a “meeting” is “open”.
- The “participant” is “connected” and the “meeting” is not “closed”.
```

Le LDE permet de plus de décrire un système avec des structures causales et causales temporelles. Les structures temporelles permettent de définir les conditions qui doivent être vérifiées avant ou après l'exécution d'une action. Ces structures sont illustrées dans l'exemple suivant.

Exemple 3. Structures causales et causales temporelles en LDE

```
- If a “participant” does “close” a “meeting” then each “participant”
does “leave” this “meeting”.
- Before a “participant” does “enter” a “meeting”, this “meeting” must
be “opened”.
- After a “participant” did “open” a meeting, each “participant” being
“connected” can “enter” this “meeting”.
```

Pour décrire le système, il est possible de déclarer les objets du domaine présents dans le système, et leur statut à l'état initial du système, comme dans l'exemple suivant.

Exemple 4. État initial en LDE

```
There is a “participant” named “Emma”. The “status” of ”Emma” is
“connected”.
```

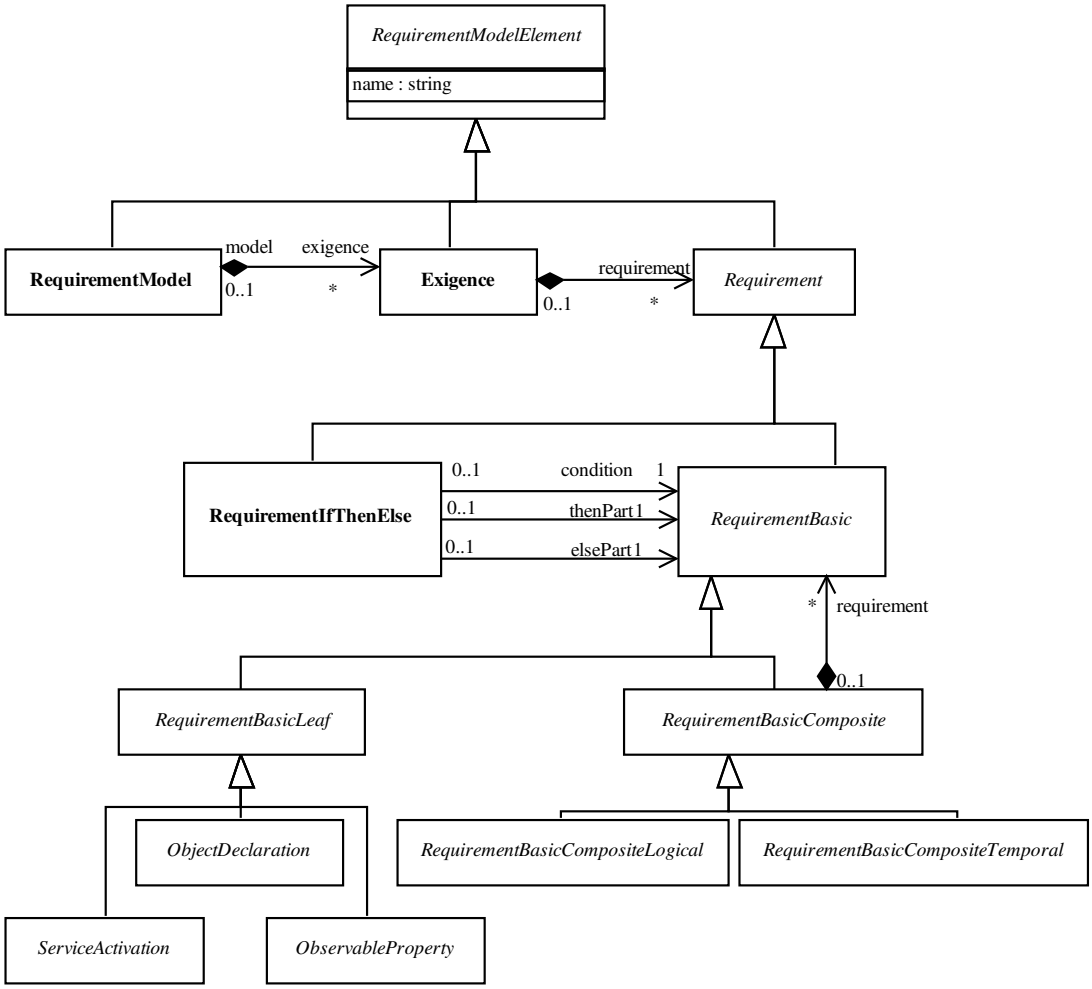


FIG. 5.2 – Modèle de l'AST LDE

Une part importante du langage est l'usage de quantificateurs. Pour le moment, cinq quantificateurs peuvent être utilisés : a/an, this, the, each, one.

- Le quantificateur "a" ou "an" suivi d'un identificateur représente n'importe quel objet de la catégorie référencée par l'identificateur. Par exemple, a "meeting" représente n'importe quelle réunion définie dans le système, comme dans les phrases LDE de l'exemple 3.
- Le quantificateur "this" référence un objet non défini précisément (c'est-à-dire quantifié par un "a /an") qui apparaît dans la même phrase.
- Le quantificateur "the" souligne que l'objet référencé doit être un singleton, c'est-à-dire qu'il ne doit y avoir qu'un seul objet de cette catégorie dans le système.
- Le quantificateur "each" décrit l'universalité (comme dans le troisième phrase LDE de l'exemple 3) et le quantificateur "one" l'existence.

La grammaire complète du LDE est donnée en annexe B.1 page 179.

Dans cette section, nous avons décrit les principes du LDE, et ses principaux éléments syntaxiques. Comme nous l'avons souligné en début de section, la sémantique du langage est donnée par la définition de patrons d'interprétation. Dans la section suivante, nous décrivons des patrons d'interprétation donnant au LDE une sémantique en termes de cas d'utilisation contractualisés.

5.3 Sémantique du LDE et patrons d'interprétations

Les principales structures du LDE ont été décrites dans la section précédente à un niveau syntaxique. Nous introduisons ici un formalisme pour définir une sémantique non ambiguë à chaque structure du LDE.

5.3.1 Définition de la sémantique du LDE

Nous proposons ici une sémantique en termes de cas d'utilisations contractualisés.

Conceptuellement, la traduction des exigences textuelles en un modèle de cas d'utilisation est une transformation de modèles, comme suggéré dans l'approche Model-driven Engineering [Bézivin et al., 2003]. Les méta-modèles des modèles d'entrée et de sortie sont exprimés de manière à ce qu'ils soient compatibles avec le méta-méta modèle MOF (Meta Object Facility, [OMG, 2004]). La transformation de modèles que nous avons ainsi définie peut être vue comme une sémantique du LDE. Techniquement, cette transformation fonctionne par application d'une technique de recherche de motifs (*pattern matching*).

Un patron d'interprétation est un couple [motif, production], où le motif est une structure du modèle d'entrée, et où la production représente ce qui est produit dans le modèle de sortie à la rencontre du motif. Nous avons défini un ensemble de tels patrons d'interprétation, dans lesquels les motifs sont des structures dans l'AST LDE des exigences LDE analysées, et une production est un cas d'utilisation contractualisé (ou au moins une partie d'un cas d'utilisation).

Les patrons d'interprétation permettent de transformer un AST LDE en cas d'utilisation contractualisés. Un ensemble de patrons d'interprétation forment pour une entreprise un acquis réutilisable (ou « asset » en anglais) dans le domaine des exigences, puisqu'ils décrivent le sens des exigences.

La syntaxe du LDE a volontairement été conçue de manière à ce qu'elle soit suffisamment générale pour pouvoir exprimer un grand nombre de phrases. La transformation d'un modèle LDE vers un modèle de cas d'utilisation agit comme une phase de vérification sémantique et détermine quelles phrases LDE syntaxiquement correctes ont vraiment un sens vis-à-vis des patrons d'interprétation définis.

5.3.2 Structure d'un patron d'interprétation

Un patron d'interprétation est défini en termes d'objets et d'attributs du modèle LDE, et la production qui y est liée est définie à la fois en termes d'objets et d'attributs du modèle LDE et du modèle de cas d'utilisation.

L'ensemble des patrons d'interprétation [motif, interprétation] inclut des moyens d'exprimer :

- des pré et post conditions sur des actions agissant sur le système,
- des changements sur les attributs des objets du système après l'exécution d'une action,
- l'activation d'une action déclenchée par l'activation d'une autre action,
- et la réaction à une modification d'attribut, c'est-à-dire l'activation d'une action déclenchée par un changement de valeur d'attribut.

Pour illustrer la notion de patron d'interprétation, considérons un patron d'interprétation PI_1 décrivant le fait qu'une propriété d'un objet change à l'activation d'un cas d'utilisation.

Le motif M_1 de ce patron d'interprétation PI_1 est donné Table 5.1. Un patron d'interprétation s'appuie fortement sur la structure de l'AST LDE donnée Figure 5.2 ainsi qu'en annexe page B.2. La première ligne de ce tableau décrit la forme générale du motif, et la deuxième ligne du tableau décrit des contraintes additionnelles nécessaires à l'application du patron d'interprétation correspondant (i.e. des contraintes sur les éléments lexicaux provenant de l'analyse lexicale des exigences LDE). Ainsi, ce motif détecte les structures *IF ... THEN* dont la conséquence est une propriété observable (*observable property*) de type *BECOMES* (c'est-à-dire la modification d'un attribut) et s'applique à un objet non explicitement identifié (l'objet référencé est quantifié par *A*), et dont la cause est une action de type *DOES* (c'est-à-dire une activation simple de service).

Pour définir le patron d'interprétation PI_1, il faut associer le motif M_1 avec une production P_1. Dans notre exemple, P_1 doit produire un cas d'utilisation indiquant que la propriété observable O1 doit changer de valeur à l'activation de l'action S1. La production P_1 est donnée Table 5.2. Cette production définit une transformation (et donc la sémantique) d'une phrase LDE donnée en un cas d'utilisation décrit en termes d'objets et d'attributs LDE.

IF S1=Action THEN O1=ObservableProperty	
O1.type	BECOMES
O1.reference	A
S1.type	DOES

TAB. 5.1 – Un exemple de motif de patron d'interprétation

Type	USE CASE
Titre	S1.title
Paramètres	x1 : S1.activator.type x2 : O1.reference.owner.type
Précondition	not O1.reference.observable = O1.reference.value
Postcondition	O1.reference.observable = O1.reference.value

où S1.title (resp. S1.activator.type) est le nom (resp. le type de l'activateur) du service S1, et où O1.reference.owner.type est le type de l'observable O1, O1.reference.observable est 1'observable O1, et O1.reference.value sa valeur.

TAB. 5.2 – Exemple de production de patron d'interprétation

Par exemple, l'application du patron d'interprétation PI_1 à la phrase LDE suivante :

```
if a participant does plan a meeting then this meeting becomes planned.
```

produit le cas d'utilisation suivant :

```
UC Plan(p :Participant,m :Meeting)
pre not planed(m)
post planed(p,m)
```

Il s'agit d'un cas d'utilisation contractualisé tel qu'il sera défini dans le chapitre suivant.

En d'autres termes, le patron d'interprétation PI_1 exprime le fait que l'utilisation du mot-clef *becomes* permet d'exprimer le fait qu'une propriété booléenne d'un objet change de valeur au cours d'une action spécifique.

5.3.3 Agrégation de cas d'utilisation

Pendant l'interprétation d'une phrase LDE, il peut arriver (et c'est même le cas général) que plus d'une phrase décrive le même service du système. Par exemple, une phrase peut décrire une façon d'utiliser un service, et une autre phrase peut en décrire une autre utilisation. La transformation du modèle LDE vers le modèle de cas d'utilisation va alors générer deux cas d'utilisation pour le même service. Il faut donc agréger ces deux cas d'utilisation en un seul.

Ainsi, considérons deux cas d'utilisation UC_1 et UC_2 décrivant le même service. Ces deux cas d'utilisation sont alors agrégés en un cas d'utilisation UC_a décrit Table 5.3.

Le processus d'agrégation permet au rédacteur des exigences de décrire un même service complexe en plusieurs phrases simples. Un autre bénéfice de l'agrégation est

Cas d'utilisation	UC_1	UC_2	UC_a
Précondition	pre_1	pre_2	$pre_1 \, or \, pre_2$
Postcondition	$post_1$	$post_2$	$(pre_1 \, implies \, post_1) \, and \, (pre_2 \, implies \, post_2)$

TAB. 5.3 – Agrégation de cas d'utilisation

qu'il est possible de décrire un même service à différents endroits des exigences, ce qui peut être utile pour l'organisation des exigences. L'agrégation est enfin un moyen de détecter des incohérences entre plusieurs exigences : par exemple si on génère un cas d'utilisation nécessitant une chose ou son contraire, on doit étudier si les exigences sont bien cohérentes.

5.3.4 Génération d'une ébauche de modèle statique

Nous avons défini une autre transformation prenant en entrée un modèle LDE, mais fournissant cette fois en sortie un modèle statique. Ce modèle est en fait une synthèse de toutes les entités décrites dans les exigences : leurs propriétés et leur liens avec les autres entités notamment. Ce modèle n'est en aucun cas un premier diagramme de classe d'analyse : il serait abusif et irréaliste de générer automatiquement un tel modèle à partir des exigences. En revanche, c'est un bon support pour sa conception. Pour rester dans la notation UML, nous présentons le modèle statique généré sous forme de diagramme de classes, mais les classes et les attributs sont stéréotypés de manière à les distinguer de vraies classes.

Le principe de génération de cette ébauche est très simple. Les propriétés observables de chaque entité du système sont collectées. Les entités sont représentées par des classes stéréotypées *Requirement* et les propriétés par des attributs de classe stéréotypés *Requirement*. Les propriétés impliquant deux entités sont transformées en associations entre ces deux entités. Pour chaque propriété, on détermine si elle est de type booléen ou énuméré. Si elle est de type énuméré, on créé le type énuméré correspondant. À titre d'exemple, l'ébauche de modèle statique ainsi construit pour la réunion virtuelle est donné Figure 5.4.

5.4 Exemple

Dans cette section, nous illustrons le contenu de ce chapitre avec l'exemple de réunion virtuelle. Les exigences de la réunion virtuelle exprimées en LDE sont données Figure 5.3. À partir de ce jeu d'exigences, nous avons généré d'une part le modèle d'analyse donné Figure 5.4, et d'autre part un modèle de cas d'utilisation qui n'est pas donné ici mais qui est proche de celui fourni en annexe page A.1.

```
requirement 1.0 "Initialization"
[]
there is a "participant" named "Smith".


requirement 1.1 "Connection"
[]
a "participant" is "connected" after this "participant" does "connect to" the "server".


requirement 1.2 "Plan"
[]
if a "participant" is "connected" then this "participant" can "plan" a "meeting".

a "participant" for a "meeting" is "the manager" after this "participant" did "plan" this "meeting".


requirement 1.1 "Open"
[]
if a "participant" is "connected" and this "participant" for this "meeting" is "the moderator"
then this "participant" can "open" a "meeting" .

a "meeting" is "opened" after a "participant" did "open" this "meeting".


requirement 1.1 "Enter"
[]
if a "meeting" is "opened" and a "participant" is "connected" then this "participant" can "enter" this "meeting".

a "participant" for a "meeting" is "entered" after this "participant" did "enter" this "meeting".


requirement 1.1 "Close"
[]
if a "participant" is "connected" and this "participant" for a "meeting" is "the moderator" and this "meeting" is "opene
then this "participant" can "close" this "meeting".

after a "participant" did "close" a "meeting",
this "meeting" is not "opened" and
each "participant" for this "meeting" is not "entered" and
each "participant" for this "meeting" is not "registered" and
each "participant" for this "meeting" is not "speaker".


requirement 1.1 "Ask"
[]
if a "participant" for a "meeting" is "entered" then  this "participant" can "ask to speak in" this "meeting" .

after a "participant" did "ask to speak in" a "meeting", this "participant" for this "meeting" is "registered".


requirement 1.1 "Speak"
[]
if "participant" for a "meeting" is "the speaker" then this "participant" can "speak".


requirement 1.1 "Over"
[]
if a "participant" for a "meeting" is "the speaker" then this "participant" can "stop speaking".

after a "participant" did "stop speaking", this "participant" is not "the speaker".
```

FIG. 5.3 – Exigences du système de réunion virtuelle en LDE

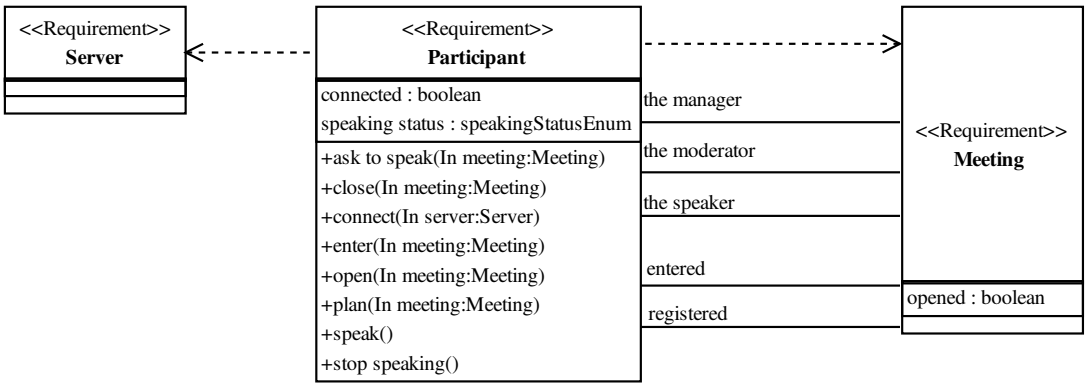


FIG. 5.4 – Modèle statique d'analyse généré pour le système de réunion virtuelle

5.5 Limitations et extensions possibles

Les principales faiblesses de cette approche proviennent du formalisme de cas d'utilisation sur lequel nous nous appuyons, et qui ne prend pas en compte les aspects numériques, ni la qualité de service. Nous aborderons ces problèmes dans le chapitre suivant.

Le LDE est un langage fortement contrôlé : assez peu de structures y sont autorisées. Le but de notre approche n'était pas de proposer une analyse complète du langage naturel, mais plutôt de proposer une méthode d'écriture des exigences, basée sur un langage relativement facile d'utilisation. Pour traiter une plus large part du langage naturel, il serait possible de nous appuyer sur des bibliothèques existantes d'analyse du langage. Une amélioration possible serait également la gestion des synonymes. Une solution envisageable est la génération d'un dictionnaire des mots du domaine après analyse syntaxique des exigences. Un tel dictionnaire pourrait être soumis au rédacteur des exigences de manière à ce qu'il puisse identifier les synonymes, et les termes équivalents. Ce dictionnaire pourrait ensuite être utilisé dans la phase d'interprétation.

La construction d'un LDE en français a été envisagée mais il est peu probable qu'elle puisse aboutir en utilisant un analyseur et une grammaire aussi simples que ceux que nous avons utilisés : le LDE s'appuie en effet sur un certain nombre de facilités offertes par la langue anglaise, et qui n'ont pas d'équivalent en français, entre autres les modaux.

Par ailleurs, pour être plus performante, notre approche nécessiterait que les patrons d'interprétation puissent être définis facilement, sans avoir une connaissance aiguë de l'AST LDE. Une interface pour faciliter la définition de patrons d'interprétation est donc nécessaire.

5.6 Conclusion

Dans ce chapitre, nous avons présenté un processus itératif d'amélioration des exigences, basé d'une part sur un langage naturel contrôlé indépendant de tout domaine (le LDE), et d'autre part sur la définition de règles d'interprétation sémantique. Les contributions de ce chapitre sont plus d'ordre méthodologique que technique : nous ne prétendons pas proposer une technique performante d'analyse du langage naturel. En revanche, le processus et les techniques présentées permettent une écriture non laborieuse des exigences par des équipes coopérantes, et la détection des incohérences ou des imprécisions via une analyse sémantique ou via la simulation. De plus, notre méthode permet de capitaliser des règles d'écriture des exigences et leur sémantique exacte, sous forme de patrons d'interprétation. Enfin, nous générons une ébauche de modèle statique synthétisant les connaissances extraites de l'ensemble des exigences pour chaque entité du système. Ce processus est en cours d'application sur des exemples TAS, différents résultats préliminaires sont présentés dans le chapitre 8.

Les contributions principales présentées dans ce chapitre sont donc :

- la définition d'un processus itératif d'amélioration et de précision des exigences,
- la définition d'un langage contrôlé et indépendant du domaine, permettant l'écriture d'exigences dans un langage plus facilement manipulable qu'un langage formel,
- la définition de structures d'interprétation sémantique des exigences, permettant de transformer les exigences en différents modèles,
- la transformation des exigences vers un modèle simulable, et un modèle statique des exigences.

Le chapitre suivant présente en détail le modèle de cas d'utilisation qui a été utilisé dans ce chapitre, et détaille comment un tel modèle, simulable par construction, permet de générer des objectifs de test.

Chapitre 6

Des cas d'utilisation à la génération d'objectifs de test

Ce chapitre détaille la génération d'objectifs de test à partir d'un modèle de cas d'utilisation augmentés de contrats.

Il est reconnu dans la littérature sur le génie logiciel [Binder, 2000a, Briand and Labiche, 2002, Jacobson et al., 1992, Ryser et al., 1998] que les cas d'utilisation et les scénarios sont une bonne base pour générer des tests et des oracles : un système spécifié avec des cas d'utilisation fournit un certain nombre d'informations nécessaires au test système. Jacobson [Jacobson et al., 1992] suggère quatre types de tests qui peuvent être développés à partir des cas d'utilisation :

- le test des flots d'événements nominaux,
- le test des flots d'exécution exceptionnels,
- le test de toutes les exigences traçables vers un cas d'utilisation,
- le test des caractéristiques décrites dans la documentation utilisateur et traçable vers un cas d'utilisation.

Cependant, ainsi que le souligne Binder [Binder, 2000b], pour utiliser les cas d'utilisation dans l'optique de guider la génération de test, il faut au préalable répondre à un certain nombre de questions :

1. comment choisir les cas de test ?
2. dans quel ordre appliquer les tests ?
3. sur quel critère s'arrêter ?

La première question rappelle le fait qu'aucune méthode n'existe pour la dérivation de cas de test à partir des cas d'utilisation. Par exemple, des techniques de partitionnement ne peuvent pas s'appliquer dans la mesure où les cas d'utilisation ne décrivent pas de variables d'entrées ou de sorties, ni de domaines auxquels appartiendraient de telles variables.

La seconde question met en avant le problème des contraintes existant entre les cas d'utilisation. Binder explique que des systèmes classiques possèdent des centaines de cas d'utilisation, ces cas d'utilisation étant liés entre eux par des contraintes et des

dépendances. Les tests construits doivent donc prendre en compte ces dépendances, afin de les révéler et les observer. Il faut pour cela un moyen d'expliciter de telles dépendances.

La troisième question souligne la nécessité d'avoir une stratégie de test et notamment un critère objectif permettant de déterminer la fin de la campagne de test système.

Pour répondre à ces questions de manière à être en mesure d'élaborer une stratégie de génération de tests système à partir des cas d'utilisation, nous proposons :

1. d'enrichir les cas d'utilisation avec des paramètres ;
2. d'expliciter les dépendances entre cas d'utilisation à l'aide de contrats (i.e. de pré- et post-conditions) ;
3. de construire un graphe comportemental basé sur les cas d'utilisation, et de le couvrir à l'aide de critères de test permettant d'en extraire des chemins.

Dans ce chapitre, nous présenterons dans un premier temps le modèle des cas d'utilisation utilisé (utilisation de paramètres et de pré- et post-conditions), ainsi que le langage proposé pour l'expression des contrats. Dans un second temps, nous décrirons le modèle de simulation des cas d'utilisation. Enfin, nous présenterons dans un troisième temps la génération d'objectifs de test proprement dits (formalisation de la notion d'objectif de test et définition des critères de test).

6.1 Modèle de cas d'utilisation

Le modèle de cas d'utilisation que nous proposons fait intervenir pour chaque cas d'utilisation des paramètres et des contrats. Le but d'un tel modèle est de formaliser suffisamment la notion de cas d'utilisation de manière à rendre leur évaluation possible sans ambiguïté. Le moyen que nous proposons pour rendre un cas d'utilisation simulable est l'introduction de contrats écrits dans un langage non ambigu. Ce qui nous a guidé dans la conception d'un tel modèle n'est pas son expressivité sous-jacente mais plutôt sa non-ambiguïté, et sa simplicité.

Ainsi, la formalisation que nous proposons n'est pas aussi expressive que le langage naturel : nous avons pour le moment réduit volontairement l'expressivité du modèle de cas d'utilisation de manière à éviter toute construction non évaluable. Il est à noter qu'un tel modèle reste cependant facilement utilisable comme le montrent nos expériences avec THALÈS (voir Chapitre 8).

6.1.1 Le modèle

Nous proposons de manipuler des cas d'utilisation paramétrés, et dotés de pré- et post-conditions. L'utilisation de paramètres et de contrats pour les cas d'utilisation est également préconisée par la méthode Catalysis [D'Souza and Wills, 1999], qui voit les cas d'utilisation comme des actions jointes. Plusieurs guides à l'écriture de cas d'utilisation ont déjà été proposés dont le plus connu est sans doute le canevas de Cockburn

[Cockburn, 1997, Cockburn, 2001] qui préconise un certain nombre d'éléments essentiels à la description d'un cas d'utilisation (un exemple d'une telle description est donné Figure 2.2 (a) page 29). Le modèle que nous proposons ne va pas à l'encontre du modèle de Cockburn, mais il propose d'y ajouter les paramètres de cas d'utilisation, et d'en formaliser les pré- et post-conditions.

Les paramètres

Les paramètres d'un cas d'utilisation représentent les concepts qui y sont impliqués. Ainsi, les acteurs d'un cas d'utilisation sont un type particulier de paramètres. Les paramètres sont souvent des concepts métiers manipulés ou utilisés par le cas d'utilisation. Ces concepts métiers pourront être réifiés dans la conception du système ; dans la phase d'analyse des exigences, ils sont juste identifiés comme des concepts métiers auxquels les cas d'utilisation sont intrinsèquement liés. À titre d'exemple, si on s'intéresse au cas d'utilisation *planifier* de la réunion virtuelle, on lui associe deux paramètres : la réunion qui est planifiée, et le participant qui planifie. Le participant impliqué est l'acteur du cas d'utilisation *planifier*. La réunion est un concept métier de la réunion virtuelle, qui sera d'ailleurs réifié dans la conception de serveur de réunion virtuelle.

Les paramètres sont ainsi typés. Nous insistons sur le fait que ces types ne correspondront pas nécessairement à des classes dans le modèle d'analyse : ce sont les types des objets métiers manipulés dans les exigences. Dans la suite, nous utiliserons le terme *entité métier* pour décrire ces types d'objets. Les types manipulés sont énumérés, la liste des valeurs effectives possibles pour chaque type (i.e. la liste des instances possibles des entités métiers) devra être décrite pour l'ensemble du système lors de la construction du modèle de simulation, comme cela sera expliqué par la suite.

Une fois les paramètres déterminés pour chaque cas d'utilisation, ils pourront être utilisés dans l'expression des contrats.

Les contrats

Chaque cas d'utilisation peut posséder une pré-condition et une post-condition. La pré-condition d'un cas d'utilisation exprime les conditions nécessaires pour l'exécution du cas d'utilisation. La pré-condition peut refléter des contraintes sur les paramètres, ou sur l'état du système du point de vue métier. La post-condition exprime la manière dont le système est modifié par l'exécution du cas d'utilisation. Dans les différentes approches préconisant d'ajouter de tels contrats aux cas d'utilisation, les contrats sont écrits sous forme textuelle. En effet, ni Cockburn ni Catalysis ne proposent de langage d'action ou de contraintes pour les cas d'utilisation. Nous proposons de les formaliser de manière à les rendre évaluables sans ambiguïté. L'approche que nous suivons est ainsi fortement inspirée de l'approche de conception par contrats (*Design by Contract*) de Bertrand Meyer [Meyer, 1992] qui préconise de contractualiser chaque méthode à l'aide de pré- et post-conditions exécutables.

Nous décrirons précisément le langage que nous proposons pour les contrats dans la section 6.1.2.

Syntaxe textuelle et représentation en UML

Nous proposons pour la description des cas d'utilisation une syntaxe textuelle (qui est notamment utilisée par nos outils), ainsi qu'une représentation en UML.

La grammaire de la description de l'ensemble des cas d'utilisation d'un système est donnée figure 6.1.

UCSYSTEM : := UC*
UC : := UC ident (LIST_PARAM) Pre constraint Post constraint
LIST_PARAM : := PARAM | PARAM , LIST_PARAM | ∅
PARAM : := ident : ident
Les terminaux sont soulignés. Les non terminaux ident *et* constraint *représentent respectivement un identificateur et une contrainte.*

FIG. 6.1 – Grammaire d'un système de cas d'utilisation

En ce qui concerne la représentation UML, comme le montre l'extrait du méta-modèle UML 2.0 de la figure 6.2, un cas d'utilisation UML n'est pas considéré comme étant un comportement (*behavior*) mais comme un classifieur de comportement (*classifierBehavior*). Ainsi, si un cas d'utilisation UML est décrit par des comportements, il n'en est pas un et à ce titre ne possède pas les contraintes pré- et post-conditions. De manière à ce que nos cas d'utilisation puissent être représentés en UML, nous avons donc recours au mécanisme de notes UML. Nous attachons à chaque cas d'utilisation une note décrivant ses paramètres, ainsi qu'une note pré-condition et une note post-condition.

Nous avons réalisé sous l'atelier de modélisation Objecteering un profil UML définissant ces notes et permettant une exportation d'un modèle UML décrivant un ensemble de cas d'utilisation vers la syntaxe textuelle précédemment décrite. Un exemple de cas d'utilisation décrit sous forme textuelle et modélisé en UML est donné figure 6.3.

UML propose 3 types de relations entre les cas d'utilisation : l'inclusion, l'extension, et la généralisation. Notre modèle prend en compte ces relations, comme nous le verrons section 6.2.3.

6.1.2 Un langage de contrats pour les cas d'utilisation

De manière à rendre les contrats évaluables de manière non ambiguë, nous avons défini un langage de contrats pour les cas d'utilisation nommé UCL (*Use case Contracts Language*). Le langage UCL est très proche des expressions logiques du premier ordre : UCL inclut les expressions logiques du premier ordre et y ajoute un certain nombre de constructions. Les expressions logiques que nous proposons manipulent à l'aide d'opérateurs booléens et de quantifieurs des propriétés booléennes sur le système. Nous insistons sur le fait que ce langage a été volontairement choisi pour être peu expressif. La démarche que nous avons suivie est de commencer par un petit langage, de manière à vérifier que notre intuition selon laquelle des contrats simples sur les cas d'utilisation permettent de générer des tests pertinents était fondée. Sur cette base, il est ensuite

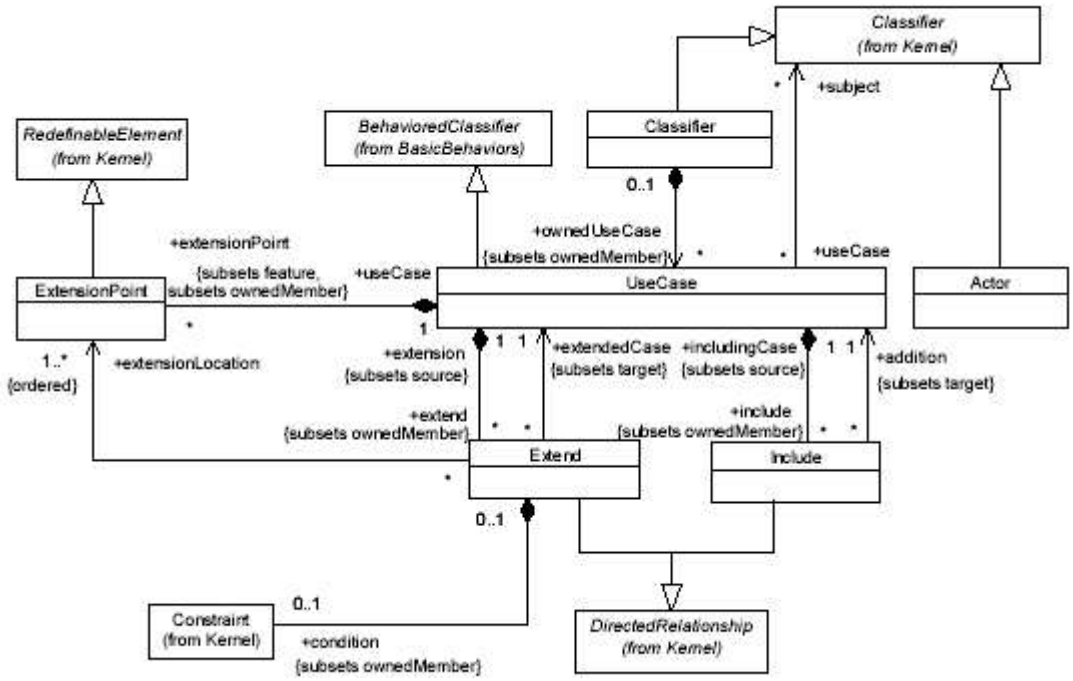


FIG. 6.2 – Cas d'utilisation en UML 2.0 : Extrait du méta modèle

```
UC useCase (p1 : TYPE1,p2 : TYPE2)
pre constraint1
post constraint2
```

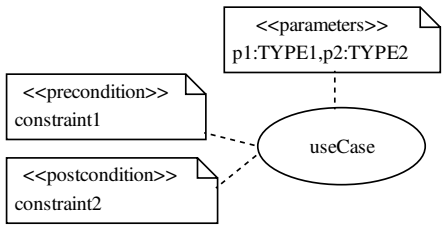


FIG. 6.3 – Exemple de cas d'utilisation sous forme textuelle et en UML

possible d'enrichir ce langage par d'autres constructions.

Il est à noter que nous n'avons pas choisi d'utiliser le langage de contraintes préconisé par UML, à savoir OCL [OMG, 2003a, Warmer and Kleppe, 1998]. En effet, OCL est un langage permettant de définir des contraintes sur des objets, or au stade de l'analyse des cas d'utilisation, aucun modèle d'objets n'est encore censé avoir été créé. En ce sens, nous ne pouvions pas utiliser OCL, et avons donc créé UCL. Certains travaux [Benattou et al., 2002] proposent des méthodes de génération de test partir de spécifications OCL. En transformant les spécifications en un modèle statique comme nous l'avons vu dans le chapitre précédent, il serait alors possible d'utiliser OCL pour spécifier des contraintes entre cas d'utilisation. Cependant, nous pensons qu'OCL est un langage lourd, « bavard » et peu lisible, et qu'il est donc difficilement utilisable par des ingénieurs « métier » pour des phases précoces d'analyse des exigences.

Les propriétés booléennes

Les propriétés manipulées portent sur les instances d'entités métier qui ont été définies dans le système. Les propriétés booléennes peuvent par exemple exprimer des informations sur le statut d'un acteur, sur les différents rôles, ou sur l'état d'une ressource. La sémantique associée aux propriétés booléennes pourra être implicitement déduite de son nom, ou explicitée par un dictionnaire attaché au système de cas d'utilisation. Ces propriétés sont donc des prédicats d'arité quelconque. L'exemple 5 illustre la notion de propriétés booléennes.

Exemple 5. Propriétés booléennes :
- *created(m : Meeting)* est une propriété qui est vraie si la réunion m est créée et fausse sinon
- *manager(u : Participant,m : Meeting)* est une propriété qui est vraie si le participant u est l'organisateur de la réunion m.

On se place dans une interprétation booléenne, et donc une propriété est soit vraie soit fausse pour des paramètres effectifs donnés, mais ne sera jamais non-définie. Ainsi, un système de cas d'utilisation possède une configuration initiale exprimant quelles propriétés sont vraies et fausses. L'état initial du système est tel que par défaut tous les prédicats qui y sont définis sont faux.

Les propriétés énumérées

Les propriétés énumérées ont été rajoutées à l'UCL pour en faciliter l'usage. Elles n'augmentent pas l'expressivité d'UCL : une transformation bijective existe entre les propriétés énumérées et les propriétés booléennes. On peut ainsi définir des propriétés dont les valeurs possibles ne sont pas booléennes comme des prédicats mais énumérées. Concrètement, on peut traduire une propriété énumérée ayant n valeurs possibles à l'aide de n propriétés booléennes. Une telle propriété permet par exemple d'associer à une réunion un type parmi les valeurs *{democratic, private, standard}*. On exprimera alors par exemple le fait qu'une réunion m est privée par l'expression : $type(m) =$ "*private*".

```
BOOLEXPR ::= DISJONCTION
DISJONCTION ::= CONJONCTION (or CONJONCTION)*
CONJONCTION ::= UNARYEXPR (and UNARYEXPR)*
UNARYEXPR ::= (BOOLEXPR)|NEGATION|FORALL|
   IMPLIES|@IMPLIES|EXISTS|PREDICATE|DIFF|EQUALITY
PREDICATE ::= IDENT(PARAM)=« STRING »|
   IDENT(PARAM)
STRING ::=IDENT*
PARAM ::= IDENT(,IDENT)*|ϵ
EQUALITY ::= IDENT = IDENT
DIFF ::= IDENT /= IDENT
NEGATION ::= not BOOLEXPR
FORALL ::= forall (LISTFORMALPARAMS){BOOLEXPR}
EXISTS ::= exists (LISTFORMALPARAMS){BOOLEXPR}
IMPLIES ::= {BOOLEXPR} implies {BOOLEXPR}
@IMPLIES ::= @pre{BOOLEXPR} implies {BOOLEXPR}
```

où IDENT est un identificateur (utilisé pour les noms de paramètres et de prédicats), et LISTFORMALPARAMS est une liste de paramètres formels, i.e. un nom et un type.
Les non-terminaux sont en majuscule et les terminaux sont soulignés.

FIG. 6.4 – Grammaire du langage UCL

Les opérateurs

Les opérateurs pouvant être utilisés pour combiner des propriétés sont les opérateurs de la logique du premier ordre : la conjonction, la disjonction, la négation, l'implication. Les quantifieurs universel et existentiel peuvent aussi être utilisés. Il est à noter qu'il est possible dans une post-condition de se référer aux valeurs des prédicats avant l'exécution du cas d'utilisation, grâce l'opérateur *@implies* : les valeurs des prédicats de la partie gauche de l'opérateur *@implies* sont les valeurs avant l'exécution du cas d'utilisation. Cette notation est inspirée de la notation *@pre* d'OCL. La grammaire des expressions booléennes que nous utilisons est donnée Figure 6.4 et précise la syntaxe exacte choisie.

Les contrats en UCL

La pré-condition d'un cas d'utilisation est la garde sur son exécution. En UCL, c'est donc une expression décrivant quelles propriétés doivent être vérifiées avant l'exécution du cas d'utilisation. La post-condition d'un cas d'utilisation exprime quant à elle les nouvelles valeurs des propriétés après exécution du cas d'utilisation. Toutes les propriétés qui ne sont pas présentes dans la post-condition d'un cas d'utilisation sont considérées comme non modifiées par ce cas d'utilisation. Il existe des restrictions sur l'expression UCL des post-conditions : les post-conditions doivent en effet être déterministes, en ce sens qu'il ne doit y avoir qu'une seule valuation possible de l'ensemble des propriétés la rendant vraie. Par exemple, une post-condition de la forme $a\ ou\ b$ est interdite. Cette restriction est encore une fois apportée pour avoir une évaluation déterministe du cas d'utilisation. Dans l'hypothèse où un ingénieur des exigences souhaite exprimer une post-condition indéterministe, nous considérons que l'indéterminisme doit être levé à l'aide d'ajout de conditions (introduction d'une implication dans la post-condition). Si l'indéterminisme ne peut pas être levé par simple introduction d'une garde logique portant sur les prédicats du système, alors c'est que cet indéterminisme

n'a pas lieu d'être introduit au niveau des cas d'utilisation.

La figure 6.5 fournit un exemple de cas d'utilisation contractualisés en UCL pour la réunion virtuelle. Un participant peut ouvrir une réunion s'il en est l'animateur, et que la réunion est créée tout en étant ni ouverte ni fermée. La réunion devient alors ouverte. Similairement, un participant peut fermer une réunion s'il en est l'animateur et que la réunion est ouverte. La réunion devient alors fermée, non ouverte, et ne contient plus de participants.

```
UC open(u : participant ; m : meeting)
pre created(m) and moderator(u, m) and not closed(m) and not opened(m)
   and connected(u)
post opened(m)

UC close(u : participant ; m : meeting)
pre opened(m) and moderator(u, m)
post not opened(m) and closed(m) and
   forall(v : participant) {not entered(v, m) and
   not asked(v, m) and not speaker(v, m) }
```

FIG. 6.5 – Exemples de cas d'utilisation contractualisés

6.2 Modèle de simulation des cas d'utilisation

Comme nous l'avons déjà expliqué, notre principe de génération de test se base sur une simulation des cas d'utilisation : on doit pour cela pouvoir simuler l'exécution de chaque cas d'utilisation, ce qui nécessite de savoir à quelle condition il est possible d'exécuter un cas d'utilisation, et quelles sont les conséquences d'une telle exécution sur le système. Du fait que les contrats des cas d'utilisation sont supposés écrits en UCL et sont donc évaluables automatiquement, ils vont pouvoir nous servir pour cette simulation.

6.2.1 Le modèle de simulation

Notre modèle de simulation comprend un système de cas d'utilisation contractualisés, la déclaration des différentes entités utilisées dans le système, et la définition d'un état initial. À tout moment de la simulation, on mémorisera une abstraction de l'état du système.

Déclaration des instances d'entités métier manipulées

Dans le modèle de cas d'utilisation que nous avons proposé, les cas d'utilisation et les prédicats utilisés pour les contractualiser sont paramétrés par des paramètres formels, qui représentent les différentes entités manipulées. Comme nous l'avons expliqué, ces

entités sont représentées par des types énumérés. Ces types doivent être déclarés avant la simulation. La déclaration se fait juste en précisant pour chaque type la liste de ses différentes instances possibles. La syntaxe est celle d'une déclaration de variable « à la Eiffel », comme le montre l'exemple 6 où sont déclarés trois participants *p1*, *p2*, et *p3* et deux réunions *m1* et *m2*.

Exemple 6. Déclaration d'entités :

```
p1, p2, p3 : PARTICIPANT
m1, m2 : MEETING
```

Ainsi, exécuter un cas d'utilisation, c'est en fait l'exécuter avec des paramètres effectifs donnés, de la même façon que quand on exécute une méthode, on l'exécute avec des paramètres effectifs (sauf exécution symbolique). Nous introduisons alors la notion de cas d'utilisation instancié.

Définition 1. Un cas d'utilisation instancié est un cas d'utilisation dont les paramètres formels ont été remplacés par des paramètres effectifs. Ainsi, $UseCase(p_1, ..., p_n)$ est une instanciation du cas d'utilisation $UseCase(P_1, ..., P_n)$ ssi $\forall i \in [1..n]\,, p_i \, est \, une \, instance \, de \, P_i$.

Si l'on reprend la déclaration d'entités de l'exemple 6, il y a par exemple six cas d'utilisation instanciés du cas d'utilisation `enter(p : PARTICIPANT, m : MEETING)` : enter(p1, m1), enter(p2, m1), enter(p3, m1), enter(p1, m2), etc.

De même, on définit la notion de prédicat instancié.

Définition 2. Un prédicat instancié est un prédicat dont les paramètres formels ont été remplacés par des paramètres effectifs. Ainsi, $Q(p_1, ..., p_n)$ est une instanciation du prédicat $Q(P_1, ..., P_n)$ ssi $\forall i \in [1..n]\,, p_i \, est \, une \, instance \, de \, P_i$.

Toujours en reprenant la déclaration d'entités de l'exemple 6, il y a 6 prédicats instanciés du prédicat `entered(p : PARTICIPANT, m : MEETING)` valant vrai si p est entré dans la réunion m : entered(p1, m1), entered(p2, m1), entered(p3, m1), entered(p1, m2), etc.

État du simulateur et définition d'un état initial

Pour débuter la simulation, nous avons besoin d'un état initial, ce qui amène à définir la notion d'état dans notre simulateur. Nous définissons l'état du simulateur comme la valuation courante de tous les prédicats instanciés du système. Concrètement, l'état est stocké sous forme de l'ensemble de tous les prédicats instanciés valués à *vrai* (on peut aussi représenter l'état courant comme la conjonction de tous les prédicats instanciés valués à *vrai*).

L'état initial est donc donné en terme de prédicats instanciés vrais à l'initialisation du système. Il est à noter que cet état peut être vide ; tous les prédicats instanciés seront donc initialement faux, la valeur par défaut d'un prédicat instancié étant fixée à *faux*.

Exécution d'un cas d'utilisation

Exécuter un cas d'utilisation, c'est tout d'abord déterminer si cette exécution est possible vis-à-vis de l'état du système de simulation, et ensuite appliquer à l'état du système de simulation les conséquences de l'exécution.

On peut exécuter un cas d'utilisation uc de pré-condition $pre(uc)$ si et seulement si $simulator\_state \Rightarrow pre(uc)$, où $simulator\_state$ est l'état courant du simulateur et est représenté comme une conjonction de tous les prédicats instanciés vrais.

Pour déterminer les effets de l'exécution d'un cas d'utilisation, on utilise sa post-condition comme oracle. La post-condition énonce une propriété vraie après exécution du cas d'utilisation. À l'exécution d'un cas d'utilisation, nous modifions donc l'état du simulateur de manière à rendre cette propriété vraie. Concrètement, pour calculer l'état du simulateur résultant de l'exécution d'un cas d'utilisation instancié cui, on résout tout d'abord l'équation $post(cui) = true$. Cette équation a une unique solution du fait des restrictions que nous avons imposées sur les post-conditions. On obtient donc une valuation pour un certain nombre de prédicats instanciés. On modifie ensuite l'état courant en fonction des valeurs de ces prédicats instanciés : si la valeur d'un prédicat instancié pi dans l'état courant est différente de celle trouvée par résolution de l'équation $post(cui) = true$, alors la valeur de pi est modifiée dans le nouvel état, sinon elle reste inchangée.

6.2.2 Simulation exhaustive et construction d'un graphe comportemental

Une simulation exhaustive du système mène à la construction d'un graphe comportemental du système sur lequel on peut facilement raisonner. Nous modélisons ce graphe par un système de transition étiqueté (ou LTS, Labelled Transition System) appelé UCTS (Use Case Transition System). Un UCTS $\mathcal{U}$ est défini par un quadruplet $(\mathcal{Q}, q_0, \mathcal{A}, \hookrightarrow)$ où :

- $\mathcal{Q}$ est un ensemble non vide d'états (chaque état étant défini comme l'état du simulateur, i.e. comme un ensemble de prédicats instanciés vrais),
- q_0 est l'état initial,
- $\mathcal{A}$ est l'alphabet des actions, une action est définie comme un cas d'utilisation instancié,
- $\hookrightarrow \subseteq \mathcal{Q} \times \mathcal{A} \times \mathcal{Q}$ est la fonction de transition.

Les états d'un UCTS représentent une abstraction des états du système en termes de prédicats instanciés, à différents stades d'exécution. Chaque transition est étiquetée par un cas d'utilisation instancié, et en symbolise l'exécution. Un exemple d'UCTS pour la réunion virtuelle est donné Figure 6.6.

Théorème. *Un* UCTS *est de taille finie, et sa taille dans le pire cas est* 2^n *états, où* n *est le nombre de prédicats instanciés possible. En bornant à* $max_{instance}$ *le nombre d'instances maximum de chaque type d'entités défini dans le système, et en bornant à* max_{param} *le nombre de paramètres maximum pour un prédicat, la taille d'un* UCTS *correspondant à un système contenant p prédicats est bornée par :* $2^{p \times \left(max_{instance}^{max_{param}}\right)}$.

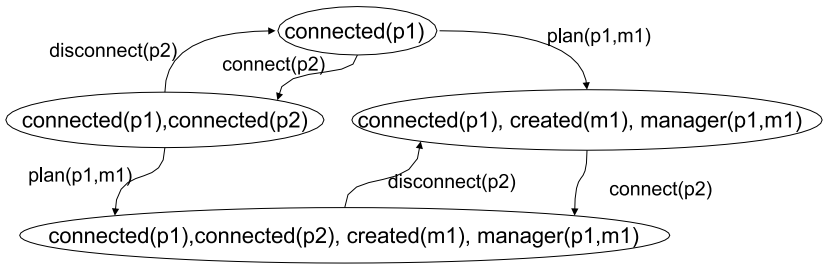


FIG. 6.6 – UCTS partiel pour la réunion virtuelle avec deux participants *p1* et *p2*, et une réunion *m1*. L'état initial est $connected(p1)$

Démonstration. Dans la mesure où il y a un nombre fini de prédicats, il y a un nombre fini de valuations pour l'ensemble des prédicats. Le nombre d'états d'un UCTS est donc fini, ce qui implique que l'UCTS est lui-même de taille finie. De plus, avec les bornes définies, chaque prédicat possède au maximum $max_{instance}^{max_{param}}$ instanciations. Il y a donc dans le système au maximum $n = p \times max_{instance}^{max_{param}}$ prédicats instanciés dans le système. L'état d'un système étant défini comme un ensemble de prédicats instanciés, le nombre possible d'états est donc au maximum 2^n. □

Si en théorie la complexité d'un UCTS est grande, en pratique elle est bien moindre, d'abord parce que tous les états possibles ne sont pas atteignables, ensuite parce que le nombre de paramètres par prédicat reste en général petit (en général inférieur à 3), et enfin parce qu'il n'est en général pas nécessaire d'utiliser un grand nombre d'instances d'entités. À titre d'exemple, la réunion virtuelle avec trois participants et une réunion possède 21 prédicats instanciés. La taille maximale de l'UCTS associé est donc 2 097 152 . Sa taille réelle est 1616 .

De plus, comme nous le verrons dans les sections suivantes, nos algorithmes de recherche de cas d'utilisation ne nécessitent pas forcément la construction exhaustive de l'UCTS grâce à l'utilisation de certains critères pertinents.

Algorithme de construction de l'UCTS

À partir de l'état initial, l'algorithme de construction d'un UCTS à partir d'un système de simulation (donné Algorithme 1) essaie successivement d'appliquer tous les cas d'utilisation instanciés, comme un jeu de puzzle. Quand il est possible d'appliquer un cas d'utilisation instancié (i.e. de l'exécuter à partir de l'état courant), alors un arc étiqueté par le cas d'utilisation instancié est créé entre l'état courant et le nouvel état issu de l'exécution du cas d'utilisation instancié (ce nouvel état est créé s'il n'existait pas déjà). L'algorithme s'arrête quand tous les états ont été explorés.

UCTS et vérification de propriétés

Le modèle d'UCTS nous permet de vérifier facilement des propriétés sur les cas d'utilisation. Nous proposons donc un mécanisme de vérification d'invariant, et un

Algorithme 1 Algorithme de construction d'un UCTS

```
algorithm buildUCTS
param initState : STATE ; useCases : SET[USECASE] ; entities: INSTANCES
var
  result : UCTS
  to_visit : STACK[STATE]
  currentState : STATE
  newState : STATE
  iUseCAses : SET[INSTANTIATED_USECASE]
init
  result.initialState := initState
  to_visit.push(initState)
  iUseCAses := useCases.computeInstances(entities)
body
  while (not to_visit.isEmpty)
  do
    currentState := to_visit.pop
    forall iuc in iUseCases such that currentState implies iuc.pre
    do
      newState := currentState.apply(iuc)
      if not result.has(newState)
      then
        result.Q := result.Q.add(newState)
        to_visit.push(newState)
      fi
      result.transitionFunction :=
      result.transitionFunction.add(currentState,iuc,newState)
    done
  done
end
return result
```

mécanisme de vérification d'existence de configuration.

Concernant la vérification d'invariant écrits en UCL, il s'agit simplement de parcourir tout l'UCTS pour s'assurer que la propriété est vérifiée pour toute exécution du sytème. En cas de violation de l'invariant, un chemin menant à la violation peut être exhibé. Par exemple, pour la réunion virtuelle, on peut s'assurer qu'il n'est pas possible qu'il y ait deux orateurs à la fois dans une même réunion en vérifiant l'invariant suivant :

```
not exists(u1, u2 : participant ; m : mtg)
{ u1/=u2 and speaker(u1, m) and speaker(u2, m) }.
```

Concernant la vérification d'existence de configuration, on parcourt l'UCTS jusqu'à rencontrer la configuration cherchée. On peut alors exhiber le chemin y conduisant. Si tout l'UCTS est parcouru sans trouver la configuration cherchée, on prouve la non existence d'une telle configuration. Par exemple, pour la réunion virtuelle, on peut s'assurer qu'il existe bien une configuration dans laquelle l'organisateur d'une réunion y est orateur par la vérification suivante :

```
exists(u1 : participant ; m : mtg)
{ speaker(u1, m) and manager(u1, m) }.
```

La vérification fournira alors un chemin menant à une telle configuration. En pratique, c'est le chemin le plus court qui mène à une telle configuration qui sera trouvé, soit pour notre exemple : `[connect(u1), plan(u1,m1, setModerator(u1,u1,m1), open(u1,m1), enter(u1,m1), ask(u1,m1), handover(u1,m1) ]`.

6.2.3 Prise en compte des relations entre cas d'utilisation UML

UML propose trois types de relations entre les cas d'utilisation : l'inclusion, l'extension, et la généralisation. Nous avons donc étudié si ces relations devaient modifier la sémantique des contrats ou les contraindre d'une façon quelconque.

La relation de généralisation

Un cas d'utilisation peut hériter d'un autre cas d'utilisation. La sémantique que nous proposons pour l'héritage de contrats est celle que Bertrand Meyer a proposée pour l'héritage de contrats portant sur les méthodes. Ainsi, quand un cas d'utilisation A hérite d'un autre cas d'utilisation B, il hérite aussi de ses contrats *pre(B)* et *post(B)*. A peut alors modifier les contrats dont il hérite, avec les contraintes qu'il ne peut qu'affaiblir la pré-condition et renforcer la post-condition. Ainsi, si A définit une pré-condition *pre(A)* et une post-condition *post(A)*, alors sa pré-condition réelle sera la disjonction $pre(A) \cup pre(B)$ et sa post-condition réelle sera la conjonction $post(A) \cap post(B)$.

FIG. 6.7 – Relation d'inclusion – figure extraite de [OMG - U2 partners, 2001]

La relation d'inclusion

Une relation d'inclusion définit le fait qu'un cas d'utilisation contient le comportement d'un autre cas d'utilisation. La figure 6.7 est extraite de la norme UML 2.0 et illustre la relation d'inclusion. On utilise la relation d'inclusion quand il y a des parties communes dans le comportement de deux cas d'utilisation ou plus. Cette partie commune est alors extraite vers un cas d'utilisation séparé, qui doit être inclus dans tous les cas d'utilisation ayant en commun cette partie. Dans l'exemple, c'est l'identification de la carte qui fait l'objet d'un cas d'utilisation séparé, et inclus dans le cas d'utilisation de retrait (*withdraw*). Dans la mesure où le but d'une telle association est la réutilisation des parties communes, la partie restante des cas d'utilisation de base est en général incomplète et dépend des parties incluses pour avoir du sens : on considère ainsi que le cas d'utilisation *retrait* n'a pas de sens sans identification de la carte.

Nous considérons dans notre modèle de cas d'utilisation que la relation d'inclusion n'est utilisée que pour décrire les cas d'utilisation opérationnels nécessaires pour réaliser un cas d'utilisation fondamental. Ainsi, si un cas d'utilisation fondamental CUf est réalisé par n cas d'utilisation opérationnels CUo_i, le comportement global de CUf peut être remplacé par les CUo_i. Concrètement, lors de la simulation des cas d'utilisation, on pourra simuler au niveau opérationnel ou au niveau fondamental. Cette modélisation impose comme contrainte que la pré-condition et la post-condition de CUf soient équivalentes à celles de la résultante des enchaînements corrects des CUo_i. Formellement, cela suppose donc que :

$$(state_{UCTS} \Rightarrow pre(CU_f)) \Rightarrow \begin{cases} \exists k \, tel \, que \, state_{UCTS} \Rightarrow pre(CUo_k) \\ \bigwedge \\ \forall \, [CUo_k..CUo_j] \; valide, \\ state_{UCTS} \blacktriangle \, [CUo_k..CUo_j] \Rightarrow post(CU_f) \end{cases}$$

où $state_{UCTS} \blacktriangle \, [CUo_k..CUo_j]$représente l'état de l'UCTS après l'application à partir de l'état courant de la séquence valide de cas d'utilisation $[CUo_k..CUo_j]$.

La relation d'extension

La relation d'extension spécifie que le comportement d'un cas d'utilisation peut être étendu par le comportement d'un autre cas d'utilisation. La figure 6.8 est extraite de la norme UML 2.0 et illustre la relation d'extension. Dans cet exemple, une transaction via l'ATM (Automated Teller Machine, équivalent anglophone du guichet automatique

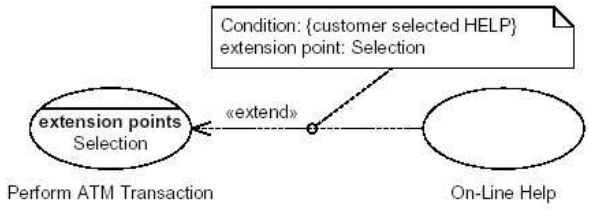


FIG. 6.8 – Relation d'extension – figure extraite de [OMG - U2 partners, 2001]

de banque) peut être étendue par une aide en ligne. L'extension a lieu à un ou plusieurs points d'extensions spécifiques définis dans le cas d'utilisation étendu. Dans l'exemple, l'extension a lieu si le client a sélectionné l'aide. Lors de l'exécution, quand un point d'extension est atteint et que la condition du point d'extension est vraie, alors tous les fragments comportementaux appropriés du cas d'utilisation étendant (ici, l'aide) vont aussi être exécutés. Si la condition du point d'extension est fausse, l'extension n'a pas lieu.

Nous proposons d'exprimer la condition d'extension en UCL. De plus, dans notre modèle, nous restreignons quelque peu l'extension, qui ne peut avoir lieu qu'en début ou fin de cas d'utilisation. Quand la condition d'extension est vraie au moment de l'application du cas d'utilisation étendu (ici la transaction ATM), alors on applique d'abord le cas d'utilisation étendant (ici, l'aide), et ensuite l'étendu. Sinon, on applique d'abord le cas d'utilisation étendu, et on réévalue la condition d'extension : si elle est vraie, on applique le cas d'utilisation étendant.

6.3 Génération d'objectifs de test

Dans cette section, nous détaillons la génération d'objectifs de test à partir du modèle de simulation et de l'UCTS décrits dans la section précédente. Nous formalisons tout d'abord la notion de séquence valide de cas d'utilisation puis d'objectifs de test, à partir de la notion d' UCTS.

Définition 3. Une séquence S de cas d'utilisation instanciés est dite valide vis-à-vis d'un système de cas d'utilisation contractualisés SU si et seulement s'il existe un chemin dans l'UCTS correspondant à SU dont la séquence des étiquettes soit indentique à S.

Un UCTS définit toutes les séquences valides de cas d'utilisation instanciés. Nous cherchons à extraire d'un UCTS des chemins pertinents pour le test. Nous définissons alors la notion d'objectif de test de la manière suivante :

Définition 4. Un objectif de test pour un système de cas d'utilisation contractualisés SU est une séquence de cas d'utilisation instanciés valide vis-à-vis de SU, et issue de la racine de l'UCTS.

Un objectif de test est donc un chemin de l'UCTS, issu de son état initial : c'est un comportement de haut niveau que l'on souhaite pouvoir exhiber. Si l'on reprend

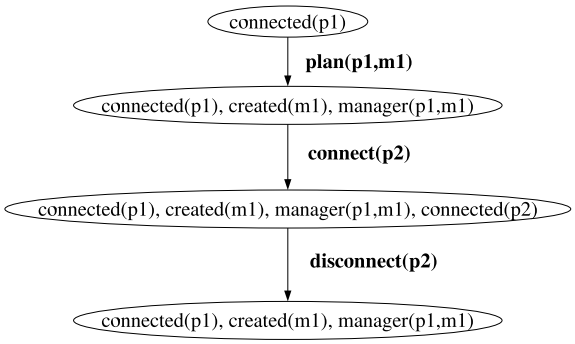


FIG. 6.9 – Illustration d'un objectif de test

l'exemple du l'UCTS partiel de la figure 6.6, un objectif de test peut par exemple représenter le fait qu'une réunion est créée par le participant p1, et qu'ensuite un participant p2 se connecte puis se déconnecte. Cela se traduit par l'objectif de test `[plan(p1,m1), connect(p2), disconnect(p2)]`, qui est illustré Figure 6.9 avec les différents états de l'UCTS parcourus.

Nous soulignons le fait qu'un objectif de test est une abstraction d'un ou plusieurs cas de test. En ce sens, pour transformer un objectif de test en cas de test, on doit apporter de l'information extérieure aux contrats des cas d'utilisation, comme nous le verrons par la suite dans le chapitre 7.

Nous souhaitons donc définir des critères de couverture permettant de sélectionner un ensemble d'objectifs de test. Nous cherchons à obtenir des ensembles d'objectifs de test qui d'une part soient de petite taille, et d'autre part contiennent des objectifs de test courts. Nous recherchons en effet un ensemble d'objectifs de test le moins redondant possible, et donc de petite taille. De plus, même dans le cadre d'une génération automatique de tests, les tests générés doivent faire l'objet au minimum d'un examen de la part du testeur, et assez souvent d'un traitement manuel. Nous pensons donc qu'il est plus raisonnable de générer un petit nombre de tests non redondants, avec pour chaque test un objectif bien ciblé. Par ailleurs, nous souhaitons obtenir des objectifs de test relativement courts, la longueur d'un objectif de test étant définie comme le nombre de cas d'utilisation instancié qui le composent. Nous pensons qu'un objectif de test court est plus compréhensible qu'un objectif de test plus long. À haut niveau d'exigences, on estime en effet qu'une séquence courte permettant d'exhiber un comportement est plus significative qu'une séquence longue parasitée de cas d'utilisation non « utiles » à l'observation du comportement.

Pour sélectionner des objectifs de test dans un UCTS, il faut définir des critères de sélection : en effet, le nombre d'objectifs de test pouvant être extraits est dans le cas général infini, du fait de l'existence de boucles dans l'UCTS. Pour établir de tels critères, nous avons recherché le meilleur compromis entre le nombre d'objectifs de test générés par un critère, et la couverture de code atteinte (la recherche de ce compromis fait l'objet d'une étude expérimentale présentée Section 8.1.2).

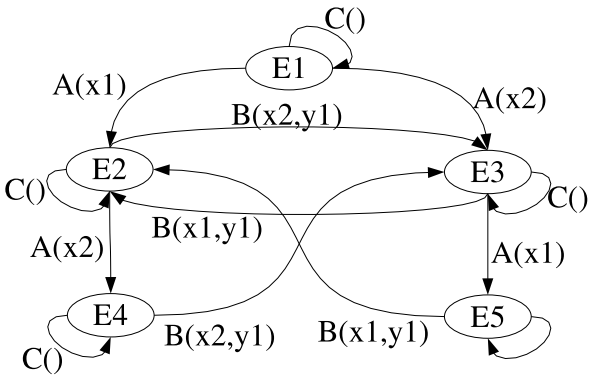


FIG. 6.10 – Exemple d'UCTS pour illustrer les critères structurels

Nous avons défini deux types de critères pour la génération d'objectifs de test : des critères de couverture structurelle de l'UCTS, et un critère sémantique sur les cas d'utilisation ; ces critères sont détaillés en sections 6.3.1 et 6.3.2. La fin de cette section définit la notion d'objectif de test de robustesse puis discute des critères permettant de les sélectionner.

6.3.1 Les critères structurels

Nous définissons dans cette section quatre critères de test structurels. Pour chacun d'entre eux, nous motivons leur définition, et en donnons une définition intuitive puis formelle. Un ordre partiel est également donné entre les critères. Nous avons défini et implémenté pour chacun de ces critères un algorithme qui le satisfait ; le principe des algorithmes est donné en fin de section. Pour illustrer tous les critères structurels, on s'appuie sur l'exemple jouet d'UCTS donné Figure 6.10, avec 5 nœuds (E1 à E5) et 13 transitions, pour un système de 3 cas d'utilisation A(x : X), B(x : X,y : Y) et C() et 5 cas d'utilisation instanciés : A(x1), A(x2), B(x1, y1), B(x2, y1) et C(). .

Critère *toutes les transitions* ($C_{transitions}$)

Ce critère vise à ce que tous les cas d'utilisations instanciés soient exécutés dans tous les contextes possibles. Dans la mesure où chaque transition de l'UCTS orienté du nœud A au nœud B représente l'exécution d'un cas d'utilisation instancié dans le contexte défini par le nœud A, il faut alors couvrir toutes les transitions de l'UCTS.

Définition 5. Un ensemble d'objectifs de test OTs satisfait le critère de couverture *toutes les transitions* pour un UCTS *ucts* si et seulement si chaque transition de l'UCTS est exercée par au moins un objectif de test de OTs. Formellement, le critère est satisfait si et seulement si :

$\forall t = (q_1, l, q_2) \in ucts. \hookrightarrow, \exists ot_i \in OTs, \exists cui \in ot_i, cui = l.$

Exemple. Pour l'UCTS de la figure 6.10, l'ensemble suivant d'objectifs de test satisfait le critère *toutes les transitions* : `{C(), [A(x1), C(), A(x2), B(x2,y1)], [A(x1),`

`B(x2, y1)], [A(x2), B(x2, y1)],[A(x2), C(), A(x1), C(), B(x1,y1)]}`.

Critère *tous les nœuds* (C_{noeuds})

Ce critère vise à ce que tous les états abstraits du système de simulation soient atteints au moins une fois. Dans la mesure où chaque nœud de l'UCTS représente l'état du simulateur, on va donc chercher à trouver des chemins passant au moins une fois par chacun des nœuds.

Définition 6. Un ensemble d'objectifs de test OTs satisfait le critère de couverture *tous les nœuds* pour un UCTS *ucts* si et seulement si chaque nœud n de l'UCTS est atteint au moins une fois par un des objectifs de test de OTs. Formellement, le critère est satisfait si et seulement si :

$\forall q \in ucts.\mathcal{Q}, \exists ot_i \in OTs, \exists cui \in ot_i, \exists t = (q_j, l, q) \in ucts. \hookrightarrow, cui = l.$

Exemple. Pour l'UCTS de la figure 6.10, l'ensemble suivant d'objectifs de test satisfait le critère *tous les nœuds* : `{[A(x1), A(x2)], [A(x2), A(x1)]}`.

Critère *tous les cas d'utilisation instanciés* (C_{cui})

Ce critère vise à ce que chaque cas d'utilisation soit au moins appliqué une fois avec chacune de ses combinaisons de paramètres effectifs.

Définition 7. Un ensemble d'objectifs de test OTs satisfait le critère de couverture *tous les cas d'utilisation instanciés* pour un UCTS *ucts* si et seulement si chaque cas d'utilisation instancié est appliqué au moins une fois par un des objectifs de test de OTs. Formellement, le critère est satisfait si et seulement si :
$\forall cui_1 \in ens_{CUI}, \exists ot_i \in OTs, \exists cui_2 \in ot_i, \exists t = (q_1, l, q_2) \in ucts. \hookrightarrow, cui = l \wedge cui_1 = cui_2$, où ens_{CUI} est l'ensemble des cas d'utilisation instanciés du système de simulation associé à *ucts*.

Exemple. Pour l'UCTS de la figure 6.10, l'ensemble suivant d'objectifs de test satisfait le critère *tous les cas d'utilisation instanciés* : `{[C(), A(x1)], [A(x2), B(x1, y1), B(x2,y1)]}`.

Critère *tous les nœuds et tous les cas d'utilisation instanciés* ($C_{noeuds-cui}$)

Ce critère vise à garantir à la fois le passage dans tous les états du simulateur et l'application de tous les cas d'utilisation instanciés. Ce critère a été introduit dans la mesure où le critère *tous les nœuds* n'assure pas que tous les cas d'utilisation soient appliqués : dès lors qu'un cas d'utilisation ne modifie pas l'état du système, alors il ne sera pas présent dans l'ensemble des objectifs de test retenus.

Définition 8. Un ensemble d'objectifs de test OTs satisfait le critère de couverture *tous les nœuds et tous les cas d'utilisation instanciés* pour un UCTS *ucts* si et seulement si chaque nœud n de l'UCTS est atteint au moins une fois par un des objectifs de test de OTs et chaque cas d'utilisation instancié est appliqué au moins une fois par un des

objectifs de test de OTs. Formellement, le critère est satisfait si et seulement si :
$\forall q \in ucts.\mathcal{Q}, \exists ot_i \in OTs, \exists cui \in ot_i, \exists t = (q_j, l, q) \in ucts. \hookrightarrow, cui = l \bigwedge$
$\forall cui_1 \in ens_{CUI}, \exists ot_i \in OTs, \exists cui_2 \in ot_i, \exists t = (q_1, l, q_2) \in ucts. \hookrightarrow, cui = l \wedge cui_1 = cui_2$, où ens_{CUI} est l'ensemble des cas d'utilisation instanciés du système de simulation associé à $ucts$.

Exemple. Pour l'UCTS de la figure 6.10, on peut constater que le critère *tous les cas d'utilisation instanciés* ne couvre pas tous les nœuds (dans notre exemple, les nœuds E4 et E5 ne sont pas atteints), et réciproquement le critère *tous les nœuds* ne couvre pas tous les cas d'utilisation instanciés (C(), B(x1, y1) et B(x2, y1) n'apparaissent pas dans l'ensemble d'objectifs de test). L'ensemble suivant d'objectifs de test satisfait le critère *tous les nœuds et tous les cas d'utilisation instanciés* : `{[C(), A(x1), A(x2)], [A(x2), A(x1)], [A(x1), B(x2, y1), B(x, y1)]}`.

Hiérarchie des critères

Pour définir la hiérarchie entre les critères structuraux, on définit la relation « est plus fort que » entre les critères.

Définition 9. Soient C_1 et C_2 deux critères de couverture. C_1 subsume ("est plus fort que") C_2 (noté $C_1 \succ C_2$) si et seulement si pour tout UCTS, et pour tout ensemble d'objectifs de test OTs satisfaisant C_1, OTs satisfait aussi C_2.

Théorème. *Le critère* toutes les transitions *subsume le critère* tous les nœuds et tous les cas d'utilisation instanciés, *qui lui-même subsume les critères* tous les nœuds *et le critère* tous les cas d'utilisation instanciés *sous l'hypothèse $\mathcal{H}$ que tous les cas d'utilisation instanciés sont présents dans l'*UCTS.

1. $C_{transitions} \succ C_{noeuds-cui}$ *sous l'hypothèse* $\mathcal{H}$
2. $C_{noeuds-cui} \succ C_{noeuds}$ *et* $C_{noeuds-cui} \succ C_{cui}$

Démonstration. Couvrir toutes les transitions d'un système de transition assure trivialement d'en couvrir tous les nœuds pour un système de transition connexe (l'UCTS est connexe par construction). De plus, une transition est étiquetée dans notre cas par un cas d'utilisation instancié. Couvrir toutes les transitions d'un système de transition assure donc également de couvrir tous les cas d'utilisation instanciés présents dans le système de transition. De par les définitions des critères, on a donc bien $C_{transitions} \succ C_{noeuds-cui}$ sous l'hypothèse $\mathcal{H}$.

La démonstration de la deuxième partie du théorème est triviale. □

La hiérarchie correspondante entre les critères est donnée Figure 6.11. Comme on le verra dans les études expérimentales du chapitre 8, les critères les plus forts ont tendance à générer un nombre déraisonnable d'objectifs de test, tout en n'étant pas plus efficaces au niveau de la couverture de code des cas de test associés.

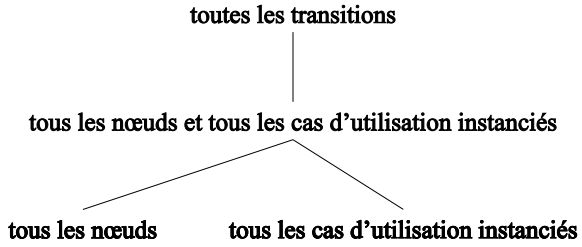


FIG. 6.11 – Relations entre les critères structuraux proposés

Algorithmes de satisfaction des critères

Pour chaque critère, nous avons défini et implémenté des algorithmes permettant de produire un ensemble d'objectifs de test les satisfaisant. Ces algorithmes sont basés sur un parcours en profondeur d'abord de l'UCTS, à partir de son état initial. Le résultat de nos algorithmes dépend de l'ordre de parcours des nœuds de profondeur égale, en ce sens nos algorithmes ne sont pas déterministes. Il suffirait cependant de fixer un ordre de parcours des nœuds pour déterminiser l'algorithme. Le choix d'un parcours en profondeur a d'abord pour but d'obtenir des objectifs de test le plus courts possible : il assure que la taille des chemins calculés est minimale, mais n'assure pas que le nombre de chemins est minimal. Si on reprend nos objectifs initiaux qui étaient d'obtenir un petit nombre de petits objectifs de test, on voit que l'on ne va ainsi satisfaire que le critère sur la taille des objectifs de test. En revanche, l'étude expérimentale qui sera présentée Section 8.1.2 sur les différents critères de test montre que certains critères aboutissent à un nombre raisonnable d'objectifs de test, comme souhaité.

Les quatre algorithmes ne diffèrent que dans le processus de marquage des nœuds et la condition d'arrêt. Nous donnons en annexe celui du critère *tous les nœuds* (Algorithme 3 page 71). Pour le critère *toutes les transitions* (resp. *tous les cas d'utilisation instanciés*), ce sont les transitions qui sont marquées (resp. les cas d'utilisation instanciés) au lieu des nœuds, et l'algorithme s'arrête quand toutes les transitions sont marquées (resp. tous les cas d'utilisation instanciés). En pratique, l'UCTS est construit à la volée et non pas a priori. Si dans le cas des critères portant sur la couverture des nœuds ou des transitions, l'intégralité de l'UCTS devra être construite, le critère *tous les cas d'utilisation instanciés* n'en nécessite pas la construction exhaustive ce qui permet de traiter des cas de taille réelle sans se heurter à l'explosion combinatoire du système de transitions.

6.3.2 Un critère sémantique

Le critère sémantique que nous proposons ne vise plus à couvrir l'UCTS correspondant à un système de contrats mais à rechercher pour chaque cas d'utilisation des objectifs de test aussi pertinents que possible. Pour ce faire, le critère qui nous a paru le plus pertinent vise à appliquer un cas d'utilisation avec les configurations les plus susceptibles d'influer sur son comportement. Nous nous attachons donc aux conditions

de satisfaction de la pré-condition de chaque cas d'utilisation. L'idée du critère que nous proposons est d'appliquer chaque cas d'utilisation dans toutes les configurations différentes qui rendent vraie sa pré-condition.

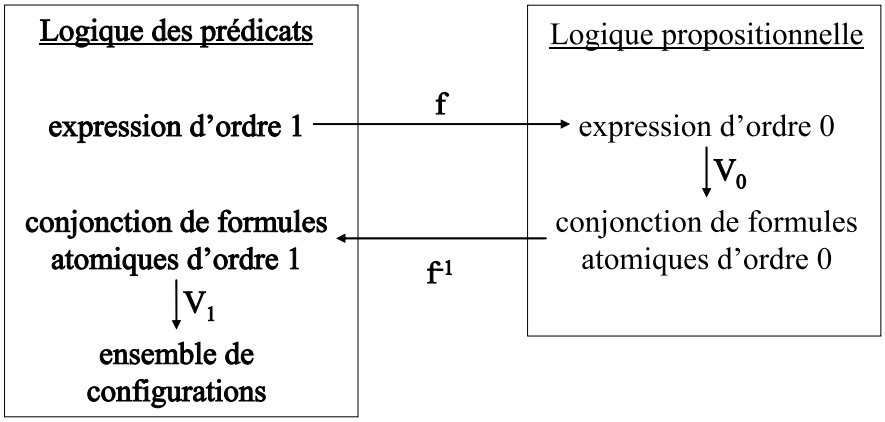


FIG. 6.12 – Critère sémantique : décomposition en sous-problèmes

Pour expliquer le critère que nous proposons, nous décomposons la recherche d'un tel critère comme exposé dans la figure 6.12 : cela nous permet de simplifier l'explication du critère, et de nous ramener à des problèmes connus dans le domaine du test structurel. Une pré-condition est une expression booléenne du premier ordre paramétrée par un nombre borné de paramètres. Soit $E(x_1, ..., x_n)$ une telle pré-condition. L'objectif est de trouver un ensemble de configurations pertinentes rendant $E(x_1, ..., x_n)$ vraie. Par « configuration », on entend « valuation de prédicats instanciés ». Les solutions recherchées seront mises sous la forme d'une conjonction de prédicats instanciés. Pour trouver ces solutions, on introduit une étape intermédiaire qui est le passage de la logique prédicative à la logique propositionnelle. Celle-ci se traduit par l'application d'une transformation (notée f dans la figure 6.12) traduisant l'expression logique du premier ordre en expression logique d'ordre 0. Par la suite, on cherche un ensemble de valuations satisfaisant $f(E(x_1, ..., x_n))$, sous la forme d'une conjonction de propositions. Cette phase de valuation est notée V_0 dans la figure 6.12. On repasse ensuite l'ensemble de conjonctions de propositions $V_0(f(E(x_1, ..., x_n)))$ dans le monde des prédicats par l'application de la fonction inverse de f. On obtient alors un ensemble de conjonctions de prédicats. On cherche alors à valuer (avec la transformation V_1 de la figure 6.12) chaque conjonction de prédicats, c'est-à-dire à obtenir des conjonctions de prédicats instanciés. Dans la mesure où les prédicats sont liés, ce n'est pas toujours possible.

Des prédicats aux propositions et réciproquement

Pour transformer une expression booléenne paramétrée en expression booléenne propositionnelle, on cherche à supprimer tous les paramètres de chaque prédicat. La fonction f se contente trivialement de transformer chaque prédicat de la forme $P(x_1, ..., x_n)$ en $P\_x_1\_...\_x_n$ et chaque expression quantifiée $\forall x_1...x_n, E(x_1, ..., x_n)$ et $\exists x_1...x_n, E(x_1, ..., x_n)$ en une proposition de nom arbitraire indicée par l'expression ini-

tiale $Q_{\forall x_1 \ldots x_n, E(x_1,\ldots,x_n)}$ et $Q_{\exists x_1 \ldots x_n, E(x_1,\ldots,x_n)}$. Ainsi on mémorise dans le nom des propositions toute l'information nécessaire à la fonction réciproque f^{-1}. La fonction réciproque est donc triviale elle aussi.

Différentes stratégies pour valuer les propositions

Après application de f, la pré-condition d'un cas d'utilisation se résume donc à une expression propositionnelle, que l'on cherche à rendre vraie. Ce type de problème est connu dans le domaine du test structurel. En effet, quand dans un programme on rencontre une structure conditionnelle gardée par une expression booléenne, il faut décider comment couvrir cette conditionnelle. Dans ce domaine, le vocabulaire adopté est celui de condition et de décision : une décision est formée d'une combinaison de conditions. La décision correspond dans notre cas à une expression propositionnelle, et les conditions sont les propositions la composant. En utilisant ce vocabulaire, on cherche donc quelles sont les valeurs des conditions à tester pour couvrir une décision en la rendant vraie. Plusieurs critères ont été proposés pour couvrir une décision (les références [Ammann et al., 2003] et [Vilkomir and Bowen, 2001] en proposent une synthèse et une formalisation), dont les critères suivants :

- Couverture des conditions multiples, c'est-à-dire la couverture de toutes les combinaisons possibles de conditions (*multiple condition coverage*, *MCC*). C'est bien sûr le critère le plus fort, mais c'est aussi celui qui génère le plus de combinaisons de conditions, puisque pour une décision comprenant n conditions, il y a 2^n combinaisons générées. On génère ainsi les 8 combinaisons données Figure 6.13 pour la décision $(a \wedge b) \vee c$.
- Couverture des décisions, c'est-à-dire que chaque décision doit être couverte par un cas de test la rendant vraie et un la rendant fausse. On peut par exemple choisir les deux combinaisons données Figure 6.13 pour la décision $(a \wedge b) \vee c$: la première combinaison value la décision à faux, la seconde à vrai.
- Couverture des conditions, c'est-à-dire que pour chaque décision, toutes les conditions doivent apparaître avec la valeur *vrai* et la valeur *faux*. On peut par exemple choisir les deux combinaisons données Figure 6.13 pour la décision $(a \wedge b) \vee c$: chaque condition y est représentée avec une fois la valeur vrai, et une fois la valeur faux.
- Couverture *MC/DC* (*Multiple Condition / Decision Coverage*), c'est-à-dire la couverture des différentes valuations de chaque décision telles que chaque condition affecte indépendamment des autres conditions la valeur de vérité de la décision. Le critère MC/DC tend à sélectionner des paires de valuations, qui ne diffèrent que dans la valuation d'une condition qui suffit à modifier l'évaluation de la décision. Le critère MC/DC a la bonne propriété de sélectionner pour chaque décision un nombre de combinaisons linéaire en le nombre de conditions. Plusieurs variantes en ont été proposées dans [Chilenski, 2001]. L'application de ce critère sélectionne par exemple les cinq combinaisons données Figure 6.13 pour la décision $(a \wedge b) \vee c$. Pour chaque condition x, on cherche une paire MC / DC telle que le premier élément de la paire value la décision à vrai, et le second élément à faux, telles que x ait une valeur différente dans les deux éléments, et telles que

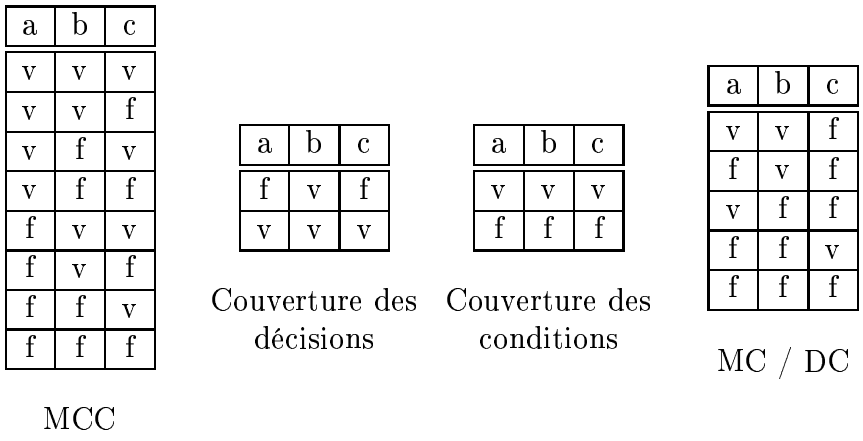


FIG. 6.13 – Exemples d'application des critères structurels avec la décision $(a \wedge b) \vee c$

les valeurs des autres conditions soient identiques dans les deux éléments. Nous avons choisi les trois paires [(v,v,f), (f,v,f)], [(v,v,f), (v,f,f)] et [(f,f,v), (f,f,f)], puis nous avons supprimé une des combinaisons (v,v,f) qui apparaissait deux fois.

Le problème que nous cherchons à résoudre est similaire, sauf que nous cherchons des critères ne sélectionnant que des combinaisons de conditions valuant la décision à *vrai*.

Nous ne pouvons donc pas adapter directement le critère *MC/DC* à notre cas précis, dans la mesure où nous ne souhaitons pas obtenir de combinaisons de conditions aboutissant à une condition fausse. En ce sens, détecter les paires *MC/DC* n'a d'intérêt que si l'on souhaite également générer des configuration amenant à l'échec de la pré-condition. Dans la mesure où la génération d'objectifs de test de robustesse peut se ramener à ce type de problème, nous discuterons de l'adaptation du critère *MC/DC* dans la section suivante.

La couverture des décisions n'a pas vraiment de sens non plus dans notre contexte, et la couverture des décisions nous a paru trop faible pour l'usage que nous en avions.

Notre choix s'est donc orienté vers l'adaptation du critère *MCC*. Ce choix était également motivé par le contexte particulier. En effet, le critère *MCC* est rédhibitoire dans le contexte de test de programme dès lors que le nombre de décisions et leur nombre de conditions associées devient grand. Dans notre contexte d'analyse de pré-condition, il n'y a qu'une seule décision examinée à la fois, et de plus le nombre de conditions est en général faible. Le nombre de solutions trouvées est également réduit dans notre contexte dans la mesure où nous ne cherchons que les solutions rendant la décision vraie. Ainsi, nous avons choisi pour la valuation des propositions (notée V_0 dans la figure 6.12) une adaptation du critère *MCC* définie comme suit.

Définition 10. Un ensemble S de valuations de propositions satisfait le critère MCC_{true} pour une expression d'ordre 0 E faisant intervenir k propositions P_k si et seulement si toutes les valuations $v = (v(P_1), ..., v(P_k))$ telles que E(v) soit satisfaite appartiennent à S : $\forall v = (v(P_1), ..., v(P_k)) \in bool \times ... \times bool$, $E(v) = true \Leftrightarrow v \in S$.

L'application de V_0 génère un ensemble de valuations, que nous considérerons non

pas sous forme de vecteurs de valeurs, mais sous forme de conjonction de propositions comme illustré dans l'exemple suivant.

Exemple. L'application de V_0 à l'expression $(a \wedge b) \vee c$ fournit l'ensemble : $\{a \wedge b \wedge c,\ a \wedge b \wedge \neg c,\ a \wedge b \wedge c,\ \neg a \wedge b \wedge c,\ \neg a \wedge \neg b \wedge c\}$.

Instanciation des prédicats

Une fois qu'a été calculé un ensemble de conjonctions de prédicats et de quantifications $f^{-1}(V_0(f(E(x_1, ..., x_n))))$, on cherche maintenant à trouver des configurations en termes de prédicats instanciés qui satisfassent ces configurations. Pour chaque conjonction C, il peut ne pas y avoir de configuration la satisfaisant (les combinaisons d'instanciations des prédicats ne sont pas toutes atteignables), il peut également y en avoir plusieurs. Nous avons alors décidé de n'en choisir qu'une.

L'instanciation des prédicats se fait par parcours de l'UCTS à la recherche du premier état dont la configuration satisfasse C, et aboutit à la valuation des différentes entités V_1.

Exemple récapitulatif

Pour récapituler la démarche, prenons un exemple. Soit le cas d'utilisation CU de pré-condition $E(x, y) = P(x, y) \vee \forall k Q(k)$.

On a alors :

$$\begin{aligned} f(E(x,y)) &= P\_x\_y \vee Q_{\forall k Q(k)} \\ V_0(f(E(x,y))) &= P\_x\_y \wedge Q_{\forall k Q(k)},\ \neg P\_x\_y \wedge Q_{\forall k Q(k)},\ P\_x\_y \wedge \neg Q_{\forall k Q(k)} \\ f^{-1}(V_0(f(E(x,y)) &= P(x,y) \wedge \forall k Q(k),\ \neg P(x,y) \wedge \forall k Q(k),\ P(x,y) \wedge \neg \forall k Q(k) \end{aligned}$$

Le résultat de l'application de V_1 dépend ensuite de l'UCTS du système complet.

Définition du critère de test *tous les termes des pré-conditions*

On peut maintenant définir notre critère sémantique, que nous nommerons *tous les termes des pré-conditions,* de la manière suivante :

Définition 11. Un ensemble d'objectifs de test OTs satisfait le critère de couverture *tous les termes des pré-conditions* pour un système de cas d'utilisation contractualisés S ssi pour la pré-condition $E(x_1, ..., x_n)$ de chaque cas d'utilisation cu de S, et pour chaque configuration c de $V_1(f^{-1}(V_0(f(E(x_1, ..., x_n)))))$, alors il existe un objectif de test ot_i dans OTs tel que ot_i amène le système dans la configuration c, puis applique le cas d'utilisation cu instancié avec la valuation V_1. Le critère est vérifié si et seulement si :

$$\begin{gathered} \forall cu \in S, \forall c \in V_1(f^{-1}(V_0(f(cu.precondition)))), \\ \exists ot_i = (cui_1, ..., cui_{k-1}, cui_k, ..., cui_j) \in OTs, \\ ((cui_1, ..., cui_{k-1}) \blacktriangle S.init = c) \bigwedge cui_k = V_1(cu), \end{gathered}$$

où $S_1 \blacktriangle S.init$ dénote la configuration du simulateur quand la séquence de cas d'utilisation instanciés S_1 est appliquée à partir de l'état initial de S.

Ce critère a pour objectif d'appliquer un cas d'utilisation à partir de toutes les configurations différentes susceptibles d'influer sur son comportement.

Nous avons défini et implémenté un algorithme (donné en annexe, Algorithme 4 page 189) visant à trouver un ensemble d'objectifs de test satisfaisant le critère *tous les termes des pré-conditions*. Cet algorithme calcule pour chaque cas d'utilisation un ensemble de configurations sous forme d'ensemble de prédicats non instanciés. Puis, pour chacune des configurations trouvées, il en recherche une instanciation dans l'UCTS par un parcours en largeur d'abord. Il ne reste alors plus qu'à calculer le chemin le plus court entre l'état initial et l'état trouvé, ce chemin forme un objectif de test. Dans le pire cas, la recherche échoue, et l'intégralité de l'UCTs aura été parcourue. L'algorithme est paramétrable dans la mesure où l'on peut stopper le parcours à une profondeur donnée pour éviter le parcours exhaustif et donc long de l'UCTS entier. En pratique, l'UCTS n'est d'ailleurs pas construit a priori mais à la volée au cours des différentes explorations.

6.3.3 Critères pour la sélection d'objectifs de test de robustesse

Les critères que nous avons définis dans la section précédente sélectionnent des séquences de cas d'utilisation instanciés valides. Ces séquences correspondent à des cas nominaux d'exécution, dans lesquels les pré-conditions sont vérifiées. Les cas de test correspondants visent donc à vérifier que le système possède bien les fonctionnalités attendues. On peut alors s'intéresser à la génération de séquences non valides, ou tout du moins « presque valides » (il ne s'agit pas de faire n'importe quoi du système, mais de tester sa réaction en présence de stimulations ciblées non conformes à sa spécification). Si le système est suffisamment spécifié et que l'implémentation respecte la spécification, alors toute séquence de cas d'utilisation instanciés n'appartenant pas à l'UCTS doit provoquer un comportement non-nominal, que nous appellerons exceptionnel. L'idée est donc de générer des séquences non valides de cas d'utilisation instanciés, à l'exécution desquelles on attendra de la part du système un message d'erreur, ou une exception (ou une faillite du système si celui-ci n'est pas robuste).

Générer de telles séquences non valides suppose que le système de contrats soit suffisamment détaillé, afin que tout comportement non spécifié soit incorrect, et que l'UCTS puisse servir d'oracle. Pour s'assurer du bon niveau de complétude, il est intéressant d'utiliser le simulateur.

L'ensemble des séquences non valides de cas d'utilisation instanciés est dans le cas général infini, et on va donc se concentrer sur un seul type de séquences non valides, ne contenant qu'un seul cas d'utilisation instancié appliqué à partir d'une configuration non conforme à sa pré-condition. Ce cas d'utilisation instancié « non valide » sera placé en fin de séquence, comme illustré Figure 6.14. On estime en effet que les conséquences de l'application d'un cas d'utilisation instancié de manière incorrecte peuvent être détectées immédiatement, par analyse des échanges de message avec l'environnement extérieur (les acteurs dans notre cas). On appellera *objectifs de test de robustesse* de

telles séquences invalides.

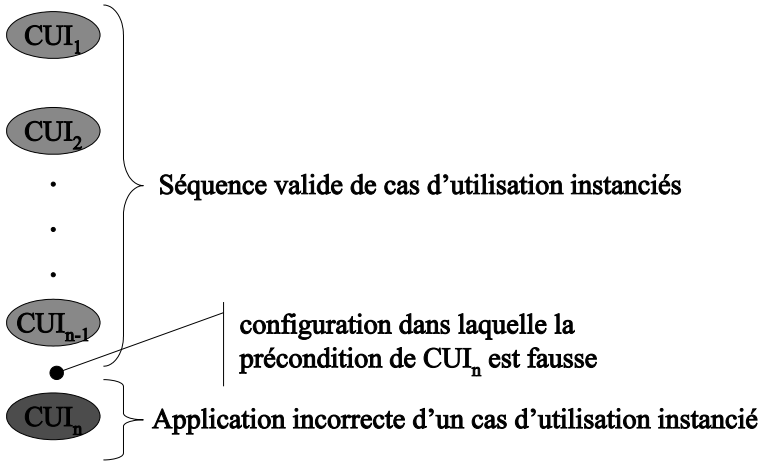


FIG. 6.14 – Objectif de test de robustesse

Définition 12. On appelle *objectif de test de robustesse* pour un système de cas d'utilisation contractualisés S une séquence $(cui_1, ..., cui_{n-1}, cui_n)$ telle que :

- $\forall i \in [1..n], cui_i \in S.ens_{CUI}$
- $(cui_1, ..., cui_{n-1})$ est une séquence de cas d'utilisation instanciés valide vis-à-vis de S
- $(cui_1, ..., cui_{n-1}, cui_n)$ n'est pas une séquence de cas d'utilisation instanciés valide vis-à-vis de S

Le principe que nous avons adopté pour sélectionner un ensemble d'objectifs de test de robustesse est similaire à celui retenu pour le critère sémantique *tous les termes des pré-conditions*. Nous cherchons en effet, pour chaque cas d'utilisation à rendre sa pré-condition fausse. On trouve pour cela des configurations rendant la pré-condition fausse, puis un chemin depuis l'état initial de l'UCTS vers cette configuration. Cela forme la partie valide de l'objectif de test de robustesse. On applique ensuite le cas d'utilisation de manière incorrecte.

Si l'on reprend le schéma de la figure 6.12, on voit que la partie qui diffère du critère sémantique *tous les termes des pré-conditions* est la phase V_0. On définit donc une nouvelle procédure $V_{0-Robustesse}$ qui produit des conjonctions de proposition rendant fausse l'expression propositionnelle donnée en entrée.

Ainsi, soit $E(x_1, ..., x_n)$ la pré-condition du cas d'utilisation étudié. $V_{0-robustesse}$ produit un ensemble de valuations satisfaisant $\neg f(E(x_1, ..., x_n))$, sous la forme d'une conjonction de propositions. On utilise pour cela les mêmes principes que pour V_0.

On peut ainsi définir le critère de robustesse permettant de sélectionner des objectifs de test de robustesse de la façon suivante :

Définition 13. Un ensemble d'objectifs de test de robustesse $OTs_{robustesse}$ satisfait le *critère de couverture de robustesse* pour un système de cas d'utilisation contractualisés S ssi pour la pré-condition $E(x_1, ..., x_n)$ de chaque cas d'utilisation *cu* de S, et pour

chaque configuration c de $V_1(f^{-1}(V_0(\neg f(E(x_1, ..., x_n)))))$, alors il existe un objectif de test de robustesse ot_i dans $OTs_{robustesse}$ tel que ot_i amène le système dans la configuration c, puis applique le cas d'utilisation cu instancié avec la valuation V_1. Le critère est vérifié ssi :

$$\forall cu \in S, \forall c \in V_1(f^{-1}(V_0(\neg f(cu.precondition)))),$$
$$\exists ot_i = (cui_1, ..., cui_{k-1}, cui_k) \in OTs,$$
$$((cui_1, ..., cui_{k-1}) \blacktriangle S.init = c) \bigwedge cui_k = V_1(cu),$$

où $S_1 \blacktriangle S.init$ dénote la configuration du simulateur quand la séquence de cas d'utilisation instanciés S_1 est appliquée à partir de l'état initial de S.

Lors de la partie décrivant cette phase V_0, nous avons abordé le fait que le critère *MC/DC* ne pouvait pas être appliqué du fait que les paires *MC/DC* n'avaient pas de sens dans un contexte où on voulait uniquement calculer des valuations rendant la décision fausse. Si l'on se place maintenant dans l'optique de générer pour un même système des objectifs de test d'une part et des objectifs de test de robustesse d'autre part, alors l'utilisation du critère *MC/DC* plutôt que *MCC* devient envisageable dans le but de réduire le nombre d'objectifs de test final, tout en conservant un ensemble pertinent. En effet, le nombre de configurations trouvées par V_0 et $V_{0-robustesse}$ à partir d'une expression propositionnelle manipulant k propositions est 2^k si on utilise *MCC* et linéaire en k si on utilise *MC/DC*.

L'application du critère *MC/DC* est envisageable en calculant les paires *MC/DC* et en utilisant l'élément des paires rendant la pré-condition vraie pour les objectifs de test et l'élément des paires rendant la pré-condition fausse pour les objectifs de test de robustesse. Cependant, nous ne l'avons pas utilisé. En effet, en pratique, le nombre d'objectifs de test générés avec *MCC* est largement inférieur à 2^k, dans la mesure où les propositions sont fortement liées, et de plus le nombre k est en général petit. Ainsi, il ne nous est pas apparu nécessaire d'utiliser *MC/DC* pour réduire le nombre d'objectifs de test. Il serait toutefois intéressant de faire des expériences avec *MC/DC* pour voir si on peut gagner en temps de génération. Appliquer les critères tels que définis (avec *MCC*) est long : la phase V_0 est rapide mais la phase V_1 peut être très longue du fait du nombre de parcours potentiellement exhaustifs de l'UCTS. Il serait intéressant d'expérimenter si l'utilisation du critère MC/DC permet de faire baisser le temps de génération par augmentation du temps de calcul de V_0 mais réduction du nombre de recherches en V_1.

6.4 Conclusion

Dans ce chapitre, nous avons proposé une approche permettant de générer automatiquement des objectifs de test et des objectifs de test de robustesse à partir de cas d'utilisation contractualisés. Nous avons pour cela :

- proposé le paramétrage des cas d'utilisation,
- défini un langage de contrainte pour les cas d'utilisation,
- proposé un modèle de simulation et une représentation de graphe comportemental

de l'application,
- proposé des critères sélectionnant dans le graphe comportemental des objectifs de test et des objectifs de test de robustesse,
- défini des algorithmes permettant de satisfaire ces critères.

Ces contributions permettent de répondre aux trois questions posées par Binder que nous avons présentées en introduction. Le modèle d'UCTS permet de répondre à la question du choix des cas de test : on choisit les cas de test parmi l'ensemble des chemins de l'UCTS. L'extraction de chemins valides répond à la question de l'ordre des tests : les objectifs de test que nous générons assurent que l'ordre d'exécution des cas d'utilisation est correct vis-à-vis de leur spécification. La question de savoir quand arrêter la phase de test à partir des cas d'utilisation est résolue par les différents critères que nous avons proposés. Comme nous le verrons dans le chapitre 8, ces critères sélectionnent un ensemble de tests plus ou moins pertinents, et certains critères seront identifiés comme assurant une bonne couverture du code.

Bien sûr, la génération automatique d'objectifs de test telle que nous l'avons proposée n'est qu'une étape dans le processus de génération de cas de test, puisqu'il est nécessaire de transformer ces objectifs de test en cas de test. Cette transformation fait l'objet du chapitre suivant.

Chapitre 7

Des objectifs de test aux cas de test

Les chapitres précédents ont détaillé notre démarche de génération d'objectifs de test à partir des exigences. Ce chapitre aborde la transformation de ces objectifs de test vers des cas de test. Rappelons que les objectifs de test se présentent sous forme de cas d'utilisation instanciés, alors qu'un cas de test doit refléter les échanges exacts entre le système sous test et le testeur. Ainsi, la problématique de transformation des objectifs de test vers des cas de test peut encore une fois être vue comme une problématique de traçabilité. L'objectif est en effet de faire le lien entre un cas d'utilisation et son implémentation dans le système : il faut pouvoir associer à un cas d'utilisation l'ensemble des échanges entre l'utilisateur et le système nécessaire pour réaliser le service décrit dans le cas d'utilisation. Nous proposons de modéliser cette information de traçabilité par des scénarios, c'est-à-dire des ensembles structurés de messages. Ces scénarios décrivent les comportements associés à un cas d'utilisation, c'est-à-dire les séquences des messages échangés entre le système et l'utilisateur pour que le système rende le service décrit.

Pour dériver les objectifs de test en cas de test, nous proposons l'approche présentée Figure 7.1.

Dans la première section de ce chapitre, nous décrirons une méthode qui constitue un premier pas vers les cas de test : cette méthode consiste globalement à substituer des scénarios aux cas d'utilisation. Des scénarios que nous appellerons *scénarios de test* sont alors obtenus. Comme nous le verrons, ces scénarios de test ne peuvent être considérés comme des cas de test que dans certains cas particuliers. Dans le cas général, il est raisonnable de penser qu'il peut manquer un certain nombre de messages ou d'informations dans les scénarios de test.

Il est alors naturel d'avoir recours à des outils de synthèse de test pour compléter les scénarios de test : ces outils utilisent un modèle comportemental du système sous test afin de pouvoir générer des cas de test. Nous expliquons Section 7.2 comment utiliser de tels outils dans notre démarche. Toutefois, malgré la résolution de certains problèmes posés par la synthèse de test, l'utilisation des outils de synthèse de test est restée décevante, même si nous restons convaincu que leur utilisation contribuera à terme efficacement à notre approche de génération automatique de test à partir des exigences.

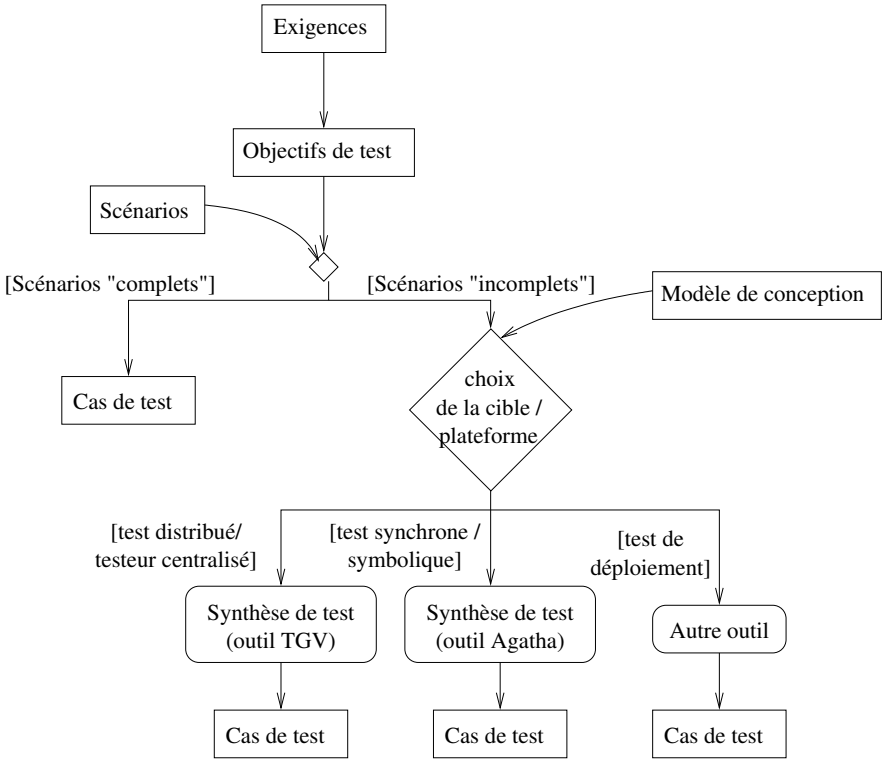


FIG. 7.1 – Des objectifs de test aux cas de test

7.1 Génération de scénarios de test

Nous proposons ici de dériver des scénarios de test à partir d'objectifs de test en utilisant des scénarios attachés à chaque cas d'utilisation. Ainsi, un cas d'utilisation est documenté non seulement par ses pré- et post-conditions comme expliqué dans le chapitre 6, mais aussi par des scénarios illustrant comment le système doit être stimulé par les acteurs de manière à réaliser le cas d'utilisation, et comment le système doit réagir à cette stimulation.

Nous appelons *scénario de test* un diagramme de séquence représentant un test. Les scénarios de test peuvent différer des cas de test dans le sens où les cas de test peuvent directement être appliqués sur un pilote de test alors que les scénarios de test peuvent être incomplets. Les scénarios de test contiennent les principaux messages échangés entre le testeur et le système sous test.

Dans cette section nous exposons tout d'abord les raisons qui nous ont amené à choisir d'utiliser des scénarios, et expliquons comment ils peuvent être représentés sous forme de diagramme de séquence ou extraits de machines à états et de diagrammes d'activité. Dans un second temps, nous détaillons comment ces scénarios sont utilisés pour générer des scénarios de test. Enfin, nous discutons par rapport à un critère de test, les différents verdicts pouvant être émis à l'exécution des scénarios de test générés.

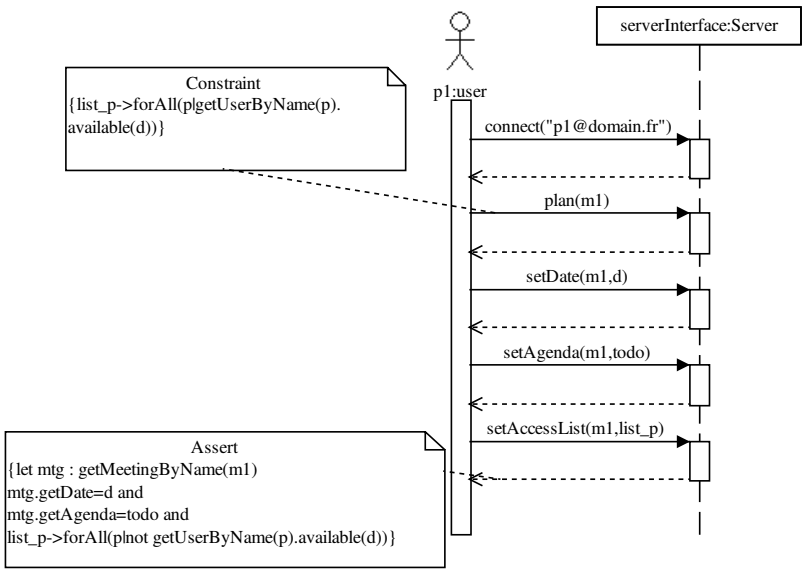


FIG. 7.2 – Un cas de test correspondant à l'objectif de test de l'exemple 7

7.1.1 Les scénarios utilisés : motivations et extraction

7.1.1.1 Motivations à l'utilisation de diagrammes de séquence

Pour bien différencier objectif de test de cas de test, prenons l'exemple suivant d'un objectif de test simple qui exhibe la planification d'une réunion :

Exemple 7. [connect(p1), plan(p1,m1)].

Un cas de test correspondant à l'exhibition de la planification d'une réunion est bien différent, comme on le voit Figure 7.2, il comprend en effet beaucoup plus d'informations. Cet exemple illustre nos trois motivations à utiliser des scénarios qui sont détaillées ci-dessous.

Motivation 1. La première motivation qui nous a orienté vers l'utilisation de scénarios sous forme de diagrammes de séquence UML est d'améliorer l'oracle inclus dans les objectifs de test. Les objectifs de test construits en utilisant les contrats des cas d'utilisation comme expliqué dans le chapitre 6 n'embarquent pas un oracle précis ; l'oracle y est juste l'attente :

- d'une exécution non-interrompue pour les objectifs de test fonctionnels,
- ou d'une erreur ou d'un avertissement pour les objectifs de test de robustesse.

De tels verdicts sont faibles dans la mesure où ils ne vérifient ni la cohérence des sorties produites par le système, ni aucune propriété sur l'état du système. Les informations utiles pour générer ce type de vérification ne sont pas extractibles des contrats des cas d'utilisation puisqu'ils doivent rester de haut niveau et indépendants du reste de

la modélisation (en ce sens qu'ils ne référencient aucun autre élément du modèle). En revanche, ce type d'information peut se trouver dans les diagrammes de séquence illustrant les cas d'utilisation. Chacun de ces diagrammes de séquence illustre comment un acteur stimule le système et comment le système répond, et permet donc d'affiner l'oracle.

Motivation 2. La seconde motivation vient de l'objectif d'obtenir des scénarios de test à partir desquels un générateur de code peut générer des squelettes de cas de test. Les objectifs de test que nous avons générés sont très éloignés des messages à échanger lors d'un test, puisqu'ils consistent juste en des cas d'utilisation instanciés.

Par exemple, considérons l'objectif de test de l'exemple 7. Un tel objectif de test ne décrit pas le protocole à suivre pour fournir au serveur les informations nécessaires à la planification : il peut exister une méthode dans le serveur avec autant de paramètres à fournir que d'informations concernant la planification, ou alors le serveur peut demander interactivement à l'organisateur les informations nécessaires, ou encore, comme c'est le cas dans notre implémentation, l'organisateur peut fournir une à une chacune des informations nécessaires avec une commande adéquate (cf. Figure 7.2).

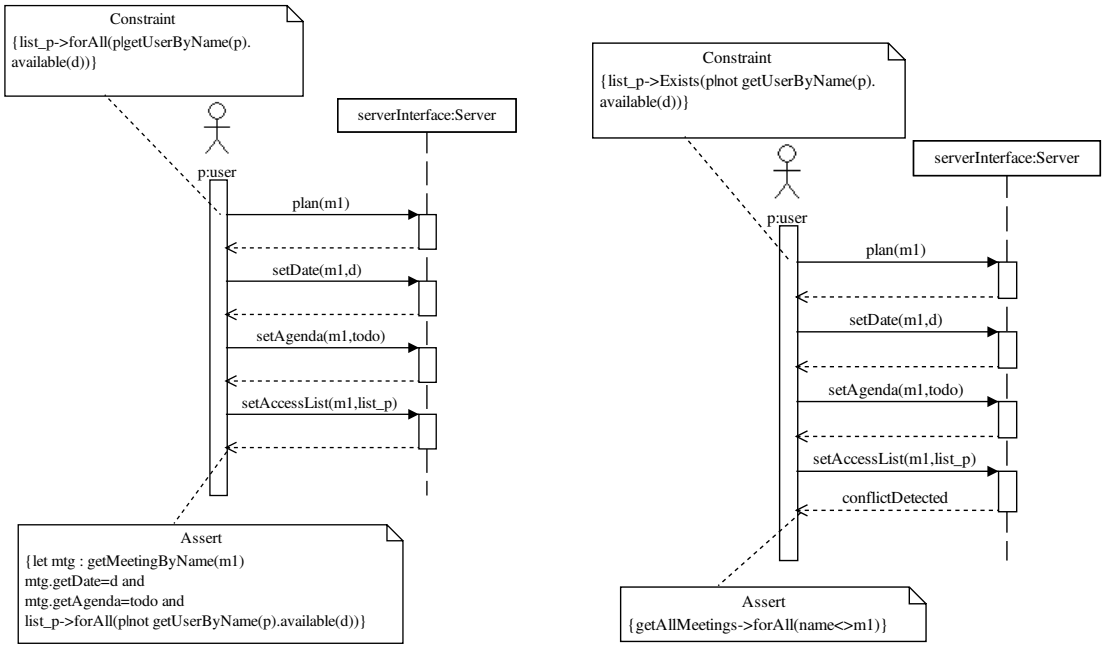
Utiliser des diagrammes de séquence nous permet donc de rapprocher les objectifs de test du modèle de conception du système : les diagrammes de séquence contiennent non seulement les échanges exacts de messages tels que prévu par le protocole de communication, mais également contiennent toutes les informations sur le type des objets impliqués. En effet, alors que les objectifs de test impliquent des entités telles que des acteurs ou des concepts métier, les diagrammes de séquence contiennent des références à des types existant dans le diagramme de classe d'analyse UML du système. Par exemple, dans l'objectif de test de l'exemple 7, *p1* est de type *Participant*. En fait, dans l'implémentation que nous avons choisie, la représentation interne d'un participant se fait via une classe nommée *User*, mais l'acteur qui représente ce participant est identifié par son adresse de courrier électronique ; c'est donc cette adresse qui doit être fournie au serveur pour identification, comme on le voit Figure 7.2.

De ce fait, d'une part de tels scénarios permettent de rapprocher suffisamment l'objectif de test des interfaces réelles du système pour pouvoir utiliser un générateur de code, et d'autre part des exigences de plus bas niveau peuvent être exprimées. Par exemple, dans le système de réunion virtuelle, il n'est pas facile ni intuitif d'exprimer avec des cas d'utilisation contractualisés le fait qu'il n'est pas possible de planifier une réunion quand un de ses participants attendu n'est pas disponible à cette date. En effet, même en rajoutant la liste des participants à la réunion en paramètre du cas d'utilisation, les exigences de haut niveau telles que nous les avons décrites ne permettent pas de manipuler de manière simple des objets complexes comme des agendas et des dates. Cela est cependant très facile et naturel en utilisant des diagrammes de séquence s'appuyant sur les structures statiques du reste du modèle, dès lors que celles-ci sont définies. Cela est illustré dans les diagrammes de séquence de la figure 7.3.

Motivation 3. Notre troisième motivation vient du fait que les scénarios et en particulier les diagrammes de séquence sont de plus en plus utilisés dans l'industrie dans

les premières phases de conception d'un logiciel. Les conclusions d'une étude sur l'utilisation des scénarios dans les projets logiciels industriels [Weidenhaupt et al., 1998] insistent sur le besoin industriel de baser les tests systèmes sur les cas d'utilisation et les scénarios, et expliquent que la plupart des projets manquent d'une approche systématique pour définir des cas de test à partir des scénarios. Notre approche est dans ce sens une proposition pour faciliter l'usage des scénarios dans les phases de validation système.

7.1.1.2 Les diagrammes de séquence utilisés



(a) Nominal (b) Exceptionnel

FIG. 7.3 – Exemples de scénarios pour le cas d'utilisation *Plan*

Chacun des diagrammes de séquence que nous considérons est attaché à un cas d'utilisation et en représente un déroulement nominal ou exceptionnel, ce qui est une manière classique de procéder [Fröhlich and Link, 2000, Cockburn, 2001]. Les diagrammes de séquence sont de niveau système, dans le sens où ils n'impliquent que l'interface du système et les acteurs. De tels diagrammes de séquence peuvent donc être construits tôt dans le cycle de développement du logiciel : dès lors que les interfaces du système ont été fixées. Cette caractéristique nous paraît primordiale méthodologiquement parlant : notre démarche est de générer des tests à partir des exigences d'un système, ces exigences regroupent les exigences textuelles, les cas d'utilisations et leurs scénarios descriptifs. Cette démarche n'est valide que si la conception des scénarios est possible

très tôt dans la conception globale. Par ailleurs, il nous semble important de nous baser sur des scénarios simples (courts et n'impliquant que les interfaces du système), dans la mesure où il nous paraît irréaliste et méthodologiquement illogique de s'appuyer sur des diagrammes de séquence décrivant le déroulement d'un cas d'utilisation en détaillant également le comportement interne du système sous test. Un exemple de scénarios tels que nous les avons décrits est donné Figure 7.3. Dans cette figure, un scénario nominal et un scénario exceptionnel sont donnés pour le cas d'utilisation *plan* du système de réunion virtuelle.

Les diagrammes de séquence peuvent mettre en jeu des paramètres : dans la mesure où ils sont attachés aux cas d'utilisation qui sont paramétrés, il est naturel de retrouver dans les diagrammes de séquence au minimum les mêmes paramètres que dans leurs cas d'utilisation. Nous ferons dans la suite la supposition qu'un scénario utilise exactement les mêmes paramètres que son cas d'utilisation. Nous discuterons dans la section 7.1.4 de la possibilité de rajouter des paramètres au diagramme de séquence. Dans les deux scénarios de la figure 7.3, les scénarios ont deux paramètres : *m1* et *p* ; *d*, *todo* et *list_p* sont des constantes, et représentent respectivement la date, l'ordre du jour, et la liste des participants invités à cette réunion. Ils pourraient être transformés en paramètres du scénario.

Les diagrammes de séquence contiennent plus d'informations que les cas d'utilisation, puisqu'ils s'appuient sur les autres éléments du modèle UML, et peuvent donc être plus précis. Ainsi, les scénarios peuvent renforcer les contrats du cas d'utilisation. Ce meilleur niveau de précision est aussi dû au fait que le scénario décrit un déroulement particulier du cas d'utilisation, qui peut donc nécessiter de renforcer les contrats du cas d'utilisation, plus général.

Le renforcement des contrats du cas d'utilisation s'effectue en ajoutant au diagramme de séquence des contraintes nécessaires à sa bonne exécution, ainsi que des assertions devant être vérifiées au cours ou à la fin de l'exécution. Ces contraintes et assertions sont exprimées en OCL [OMG, 2003a, Warmer and Kleppe, 1998], et s'appuient sur le reste du modèle, de manière à pouvoir manipuler des objets définis dans les autres vues UML, et ainsi d'affiner les contrats des cas d'utilisation. Dans les exemples de la figure 7.3, la contrainte nominale OCL vérifie que les participants sont bien disponibles à cette date, et l'assertion OCL nominale s'assure que la réunion a bien été planifiée avec les bons paramètres. Le scénario exceptionnel vérifie dans une contrainte que les participants ne sont pas disponibles à la date de la réunion, et dans une assertion que la réunion n'a pas été planifiée. Il est à noter que dans cet exemple, les contraintes et assertions sont placées en début et fin de scénario. Cependant, dans le cas général, il peut y avoir plusieurs contraintes et plusieurs assertions dans un même scénario, réparties tout au long du scénario. Le langage OCL convient parfaitement à ce type de contraintes alors qu'il ne pouvait pas être utilisé au niveau des cas d'utilisation, puisqu'on ne pouvait pas considérer qu'on disposait d'un modèle statique au moment du choix des cas d'utilisation.

Les scénarios nominaux représentent des déroulements classiques du cas d'utilisation, tandis que les scénarios exceptionnels représentent des déroulements menant à une erreur, à la levée d'une exception ou d'un message d'erreur : globalement, ces scénarios illustrent l'échec du cas d'utilisation. Dans notre contexte, les scénarios nominaux et

exceptionnels sont différenciés par les *tagged values* UML *{nominal}* et *{exceptionnel}*. Les scénarios nominaux sont utilisés pour le test fonctionnel tandis que les scénarios exceptionnels sont utilisés pour le test de robustesse.

7.1.1.3 Extraction de diagrammes de séquences à partir d'autres modèles UML

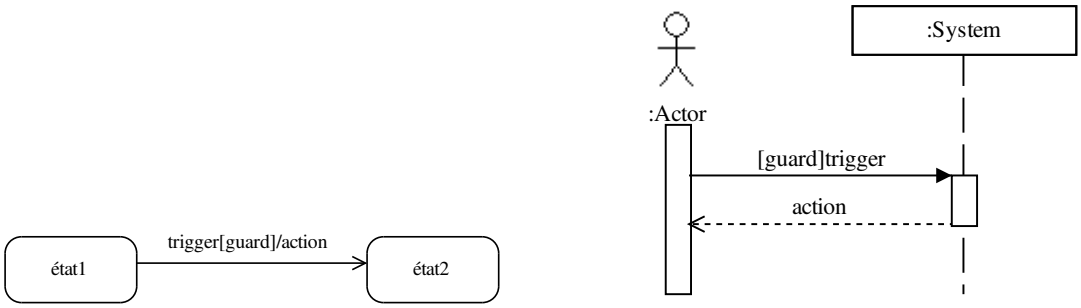
Il est parfois plus naturel de représenter les différents déroulements possibles d'un cas d'utilisation par une machine à états ou un diagramme d'activité. La notation UML prévoit ainsi de décrire les comportements d'un cas d'utilisation avec ces trois types de diagrammes. Si les scénarios ne sont pas décrits par des diagrammes de séquence, alors nous proposons d'extraire des diagrammes de séquence des machines à états et des diagrammes d'activité.

Extraction à partir des machines à états. Plusieurs méthodes [Kim et al., 1999, Offutt and Abdurazik, 1999] ont été développées pour extraire des chemins de machines à états UML, avec différents critères. Dans notre cas, nous proposons d'utiliser la technique exposée dans [Offutt and Abdurazik, 1999] pour la génération de tests unitaires avec un critère de sélection de tous les chemins élémentaires. L'application d'une telle méthode permet d'extraire d'une machine à état des chemins élémentaires, qui sont ensuite facilement transformables en diagrammes de séquence.

Pour distinguer les scénarios nominaux des scénarios exceptionnels, nous supposons que les machines à états traitées possèdent au minimum deux états finaux : un où aboutissent les comportements nominaux, et un où aboutissent les comportements exceptionnels ; de cette manière, selon que les scénarios aboutissent vers l'un ou l'autre des états finaux, ils sont distinguables.

On suppose que la machine à états est entièrement écrite du point de vue du système. Ainsi, dans une transition, le déclencheur (trigger) est un événement émis par un acteur et reçu par le système, et l'action associée est la réaction du système à ce déclencheur vis-à-vis de l'acteur. On suppose qu'aucun comportement interne au système n'est décrit. On peut donc transformer une transition en un échange de messages entre un acteur et le système, comme illustré à la figure 7.4. Il est à noter que dans le cadre d'une génération automatique des scénarios, on ne peut alors pas distinguer les acteurs, et que les diagrammes de séquence générés devront être mis à jour pour préciser les types et les noms des instances manipulées.

Extraction à partir des diagrammes d'activité. La plupart des approches préconisant de générer des tests à partir des diagrammes d'activité UML ne proposent pas de moyen automatique d'extraire des chemins d'une machine à états. Ainsi, [Kamsties et al., 2003] proposent d'extraire à la main un ensemble de scénarios de test suffisants pour couvrir toutes les branches du diagramme d'activité. *Briand et al* [Briand and Labiche, 2002] proposent de transformer le diagramme d'activité en un graphe dirigé et de le parcourir de manière à en extraire des chemins. Une telle approche peut ici être reprise pour obtenir des chemins. Donner une signification aux



(a) Transition d'une machine à états (b) échange de messages correspondant

FIG. 7.4 – Extraction de scénarios : traduction d'une transition

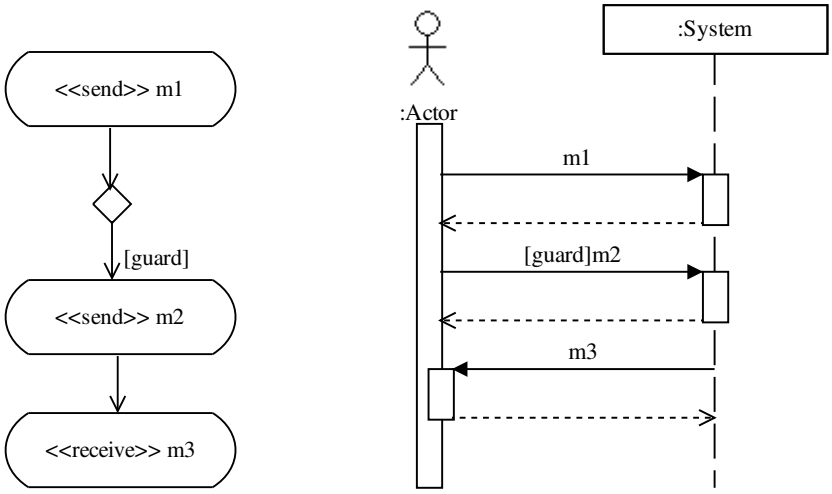
chemins en terme de diagramme de séquence ne peut se faire, comme pour les machines à états, qu'en faisant des suppositions sur le diagramme d'activité initial. On supposera donc que les actions contenues dans le diagramme d'activité sont de deux types : réception d'un événement émis par un acteur, et envoi d'un événement vers un acteur. En distinguant explicitement ces deux types d'action, un chemin du diagramme d'activité donné en termes de succession d'actions peut se dériver en un diagramme de séquence reflétant ces réceptions et envois de messages ; un exemple en est donné Figure 7.5. Comme avec les machines à états, les acteurs ne sont pas distinguables, et les scénarios extraits doivent être précisés.

Scénarios avec branchements. Les scénarios que nous manipulons sont soit nominaux, soit exceptionnels. Or il est tout à fait possible d'écrire un scénario avec un branchement, dont une branche représente un comportement nominal, et une autre branche un comportement exceptionnel. Dans ce cas, nous scindons le scénario à branchement en deux scénarios, l'un nominal, et l'autre exceptionnel, tout en conservant les gardes menant vers l'une ou l'autre des branches, comme illustré Figure 7.6.

Les scénarios à branchement dont toutes les branches sont de même type (soit nominales, soit exceptionnelles) peuvent être gardés en l'état, il faut cependant garder à l'esprit que dans la suite, un scénario sera supposé couvert dès lors qu'il aura été exécuté au moins une fois, sans pour autant en exécuter toutes les branches. Ainsi, pour obtenir une meilleure couverture des comportements, il est conseillé de le scinder en autant de scénarios que de branches, ce qui assurera la couverture de chacun d'eux.

7.1.1.4 Sémantique opérationnelle des scénarios et composition

La sémantique que nous adoptons pour les scénarios est celle des interactions d'UML 2.0, eux-mêmes inspirés des MSC et HMSC ((High-level) Message Sequence Chart) [ITU, 1996]. La sémantique est donnée en termes d'ordre partiel sur des événements :



(a) chemin d'un diagramme d'activités

(b) échange de messages correspondant

FIG. 7.5 – Extraction de scénarios à partir de diagrammes d'activité

les événements sont ordonnés séquentiellement sur une ligne de vie, et une émission de message précède sa réception.

UML 2.0 prévoit deux types de composition séquentielle entre fragments d'interactions : les compositions séquentielles faibles et fortes. La composition séquentielle forte impose que tous les événements d'un fragment aient été exécutés avant qu'un événement du fragment suivant puisse s'exécuter. La composition séquentielle faible donne une sémantique « dès que possible » : elle correspond à une fusion locale des ordres partiels (locale aux lignes de vie). Ainsi, un événement peut s'exécuter dès que son prédécesseur causal s'est exécuté : il n'y a pas de synchronisation introduite entre les deux fragments composés.

7.1.2 Génération de séquences de test

Le principe de génération de séquences de test repose sur la substitution dans les objectifs de test des cas d'utilisation instanciés par des scénarios instanciés. Des séquences de scénarios sont ainsi obtenues, et les scénarios sont ensuite composés selon une composition séquentielle forte pour obtenir un scénario de test. Nous avons choisi la composition séquentielle forte car nous considérons dans notre modèle qu'un cas d'utilisation ne peut pas démarrer avant que le cas d'utilisation précédent ne se soit terminé. Cependant, dans un système distribué, il faudrait utiliser une composition séquentielle faible : notre approche reste alors valide, son implémentation dans les outils prototypes est juste à modifier.

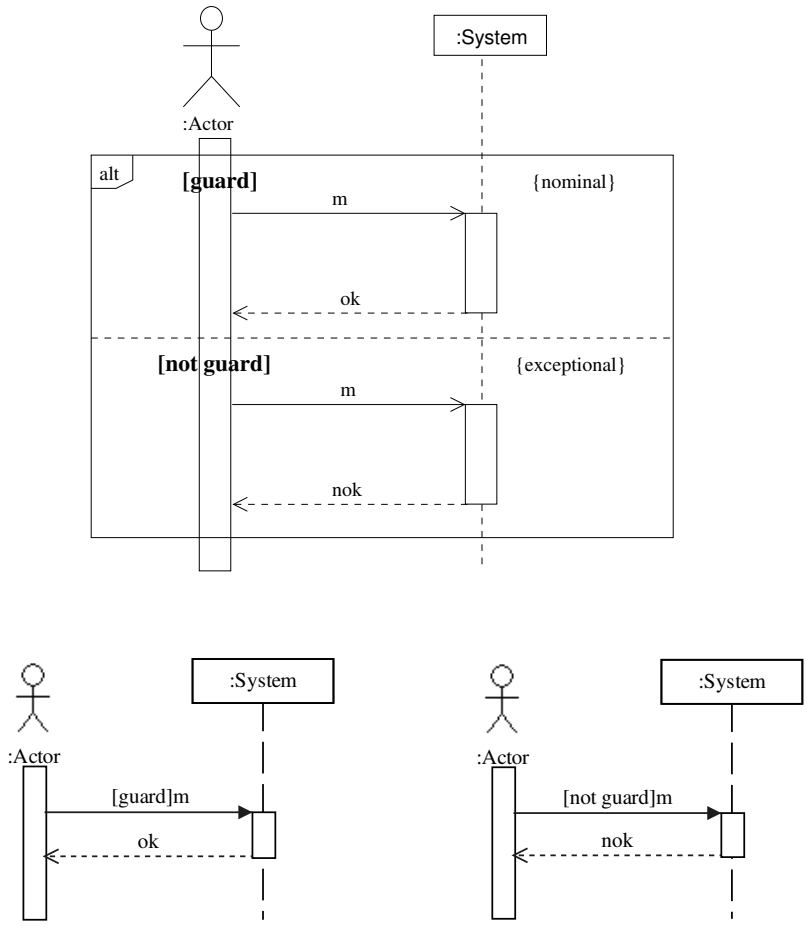


FIG. 7.6 – Décomposition des scénarios à branchement

Dans cette section, nous décrivons tout d'abord la structure du matériel de test que nous générons, puis nous détaillons la manière dont les scénarios de test sont générés.

7.1.2.1 Structure du matériel de test généré

Un diagramme de classe de la structure de test générée est donné Figure 7.7. L'outil prototype que nous avons développé pour valider notre approche est dédié aux applications Java, et nous nous appuyons donc sur le canevas de test pour Java *JUnit* [Gamma and Beck, 2004]. *JUnit* nous sert de pilote de test ; il fournit des mécanismes d'enchaînements de l'exécution de tous les tests, et offre des primitives d'assertions.

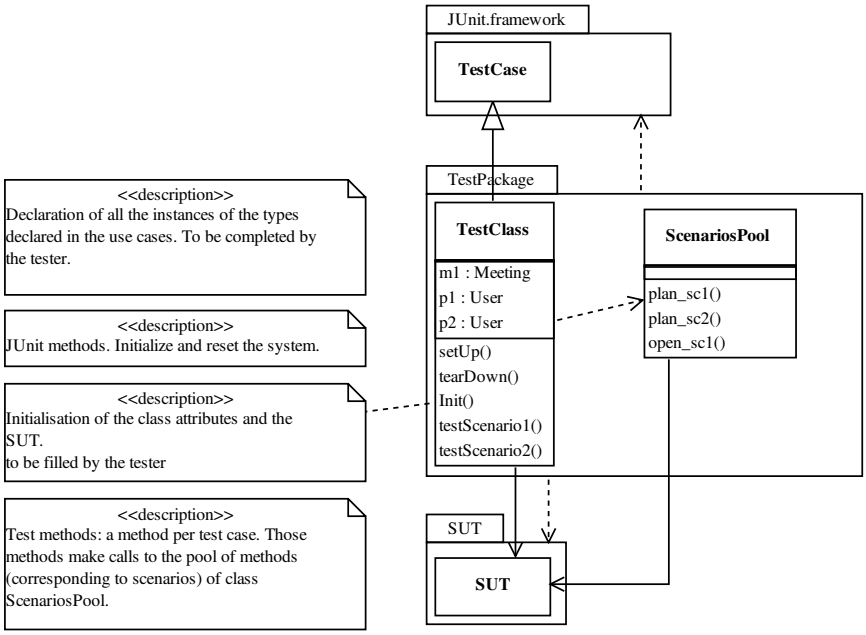


FIG. 7.7 – Structure du matériel de test généré

La classe *ScenariosPool* contient un ensemble de petits scénarios : pour chaque diagramme de séquence de chaque cas d'utilisation, une méthode est créée dans la classe *ScenariosPool*. Les paramètres formels de cette méthode sont les paramètres du scénario, c'est-à-dire les paramètres du cas d'utilisation correspondant. En revanche, ils sont maintenant typés selon ce qui est décrit dans le diagramme de séquence : on ne manipule plus les entités abstraites mais les vrais types.

Concernant l'obtention de telles méthodes Java à partir de diagrammes de séquences, les contrats OCL (contraintes et assertions) sont transformés en assertions exécutables *JUnit*, et les échanges de messages sont transformés en appels de méthode Java. Nous n'avons pas travaillé à l'automatisation de cette phase, qui nécessite d'une part un générateur de code Java à partir de diagrammes de séquence (tel que celui proposé par l'AGL *Objecteering* [Objecteering, 2004]) et d'autre part un interpréteur OCL

capable de produire du code Java correspondant, comme celui de l'université de Dresde [Interpréteur OCL, 2002]. À titre d'exemple, le code Java de la méthode correspondant au scénario nominal du cas d'utilisation de planification de la réunion virtuelle (celui de la figure 7.3) est donné Figure 7.8.

```
public static void planNominal(String Meetingname){
  try {
    for (i=0,i<list_p.size,i++){
      Assert.assertTrue((User)system.getUserByName(
                        list_p[i]).available(d))}
    system.execute(plan(MeetingName));
    system.execute(setDate(MeetingName, d));
    system.execute(setAgenda(MeetingName,todo));
    system.execute(setAccessList(list_p));
    for (i=0,i<list_p.size,i++){
      Assert.assertFalse((User)system.getUserByName(
                        list_p[i]).available(d))};
  } catch (Exception e){e.printStackTrace();}
}
```

FIG. 7.8 – Code Java de la méthode correspondant au scénario nominal de la figure 7.3

La classe principale de test est *TestClass* et a les caractéristiques suivantes :

- Elle hérite de la classe *TestCase* de *JUnit*.
- Elle contient tous les cas de test générés, chacun faisant l'objet d'une méthode de test, qui est préfixée par « test » (*JUnit* requiert de préfixer ainsi chacune des méthodes de test de manière à pouvoir utiliser l'introspection java pour lancer chacune d'entre elles). Il y a un cas de test par scénario de test généré, la méthode de génération des scénarios de test sera décrite dans la section suivante. Chacune des méthodes de test consiste en une séquence d'appels à la classe *ScenariosPool*.
- Ses attributs de classe sont les instances utilisées par le testeur dans les cas de test.
- Ses méthodes *setUp* et *tearDown* sont des méthodes que le canevas *JUnit* requiert de définir, respectivement pour initialiser le système (ici créer toutes les instances de classe) et « nettoyer » le système (ici détruire les instances créées à l'initialisation).

7.1.2.2 Construction des scénarios de test

Construire un scénario de test à partir d'un objectif de test consiste à remplacer les cas d'utilisation de l'objectif de test par leurs scénarios. On obtient ainsi une séquence de scénarios, les scénarios sont ensuite composés (selon une composition séquentielle forte) pour former un scénario de test. Dans la mesure où un objectif de test est une séquence de cas d'utilisation instanciés, il nous faut instancier les scénarios :

Définition 14. Un scénario instancié est un scénario dont les paramètres formels sont

remplacés par des paramètres effectifs.

Quand un cas d'utilisation instancié est remplacé par un de ses scénarios, le scénario choisi est instancié en utilisant les mêmes paramètres effectifs que pour le cas d'utilisation instancié qu'il remplace.

Pour définir précisément la construction des scénarios de test, nous introduisons tout d'abord quelques notations :

- On note $\{scn_{i,j}\}_{j\in 1,..n}$ l'ensemble des n scénarios nominaux attachés au cas d'utilisation cu_i et $\{sce_{i,j}\}_{j\in 1,..m}$ l'ensemble des m scénarios exceptionnels attachés au cas d'utilisation cu_i.
- La composition séquentielle forte de scénarios est dénotée par le symbole $\circ$.
- Le produit cartésien ensembliste est noté $\times$; le produit cartésien de 2 ensembles A et B est défini de la façon suivante : $A \times B = \{(a,b)|a \in A \wedge b \in B\}$.

Avec ces conventions, un scénario de test est défini à partir d'un tuple de scénarios $(sc_1, ..., sc_n)$ comme : $sc1 \circ ... \circ sc_n$ (la composition séquentielle forte des éléments du tuple). L'ensemble des tuples définissant un ensemble de scénarios de test $ST = \{st_1, ..., st_u\}$ obtenus à partir d'un objectif de test $ot = [cui_1...cui_t]$ est dénoté ST_{tuple}. L'ensemble ST_{tuple} est obtenu en appliquant un produit cartésien sur des ensembles de scénarios instanciés, comme expliqué dans les définitions suivantes, dans lesquelles les scénarios sont instanciés en utilisant la méthode *inst* prenant en paramètre un cas d'utilisation instancié.

Scénarios de test fonctionnels nominaux. Un objectif de test nominal $ot = [cui_1...cui_t]$ est transformé en l'ensemble de tuples ST_{tuple} défini par :

$$\begin{aligned} ST_{tuple} &= \prod_{i=1}^{t} \{scn_{i,j}.inst(cui_i)\}_{j\in 1,...,n} \\ &= \{scn_{1,j}.inst(cui_1)\}_{j\in 1,...,n} \times ... \times \{scn_{t,j}.inst(cui_t)\}_{j\in 1,...,n} \end{aligned}$$

La construction des scénarios de test fonctionnels nominaux peut être vue comme le remplacement l'un après l'autre des cas d'utilisation instanciés de l'objectif de test par chacun de ses scénarios nominaux. Une fois que tous les cas d'utilisation instanciés ont été remplacés, un tuple de scénarios est alors obtenu, sur lequel on applique la composition séquentielle forte pour obtenir un scénario de test.

Scénarios de test fonctionnels de robustesse. Un objectif de test de robustesse $ot = [cui_1...cui_t]$ est transformé en l'ensemble de tuples ST_{tuple} défini par :

$$\begin{aligned} ST_{tuple} &= \prod_{i=1}^{t-1} \{scn_{i,j}.inst(cui_i)\}_{j\in 1,...,n} \times \{sce_{t,j}\}_{j\in 1,...,m} \\ &= \{scn_{1,j}.inst(uci_1)\}_{j\in 1,...,n} \times ... \times \{scn_{t-1,j}.inst(cui_{t-1})\}_{j\in 1,...,n} \end{aligned}$$

$$\times \{sce_{t,j}.inst(cui_t)\}_{j\in 1,\ldots,m}$$

La construction des scénarios de test de robustesse peut être vue comme le remplacement l'un après l'autre des cas d'utilisation instanciés de l'objectif de test de robustesse par ses scénarios. Le processus de remplacement des $t-1$ premiers cas d'utilisation est le même que pour les scénarios de test fonctionnels, et le dernier cas d'utilisation instancié est remplacé par chacun de ses scénarios exceptionnels.

La figure 7.9 fournit un exemple de scénario de test ainsi construit à partir de l'objectif de test $[connect(p), plan(p, m1), connect(p2), enter(p2, m1)]$.

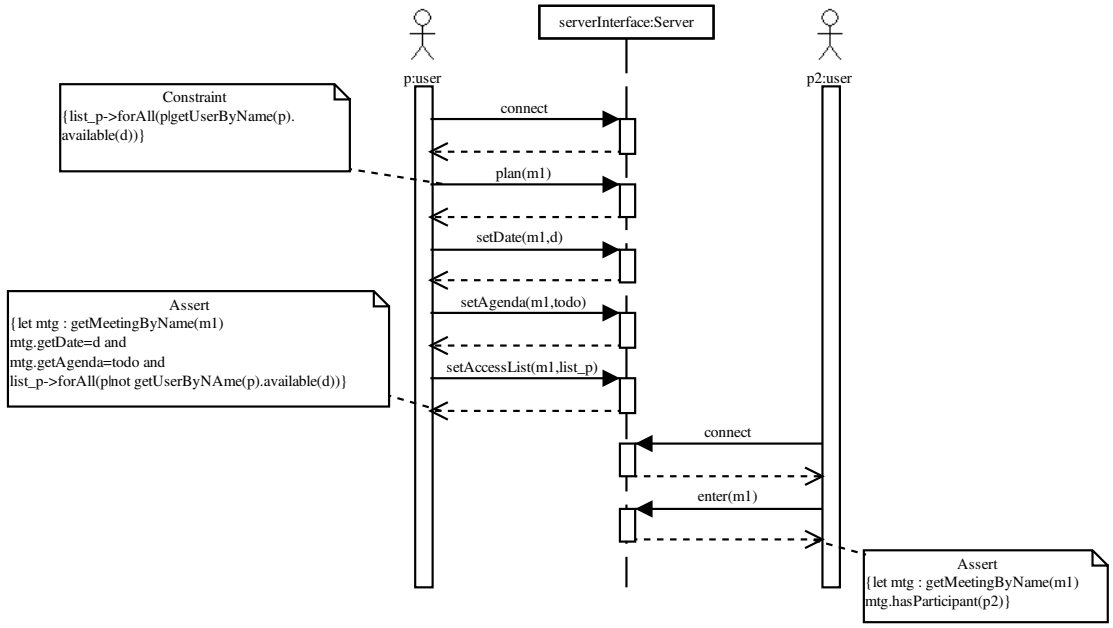


FIG. 7.9 – Un exemple de scénario de test généré

7.1.3 Analyse des verdicts vis-à-vis d'un critère de test

Nous générons les cas de test en utilisant les scénarios attachés aux cas d'utilisation. Ces scénarios sont des exigences sur le comportement du système sous test, il est donc naturel d'attendre que chacun des scénarios soit exécuté au moins par un cas de test. Nous proposons donc un critère de test intuitif en termes de scénarios.

Définition 15. Un ensemble de cas de test *CTs* satisfait le critère *tous les scénarios* (pour un système et ses exigences exprimées sous forme de cas d'utilisation et de scénarios) ssi chaque scénario (exprimé dans les exigences) est exécuté correctement dans au moins un cas de test de *CTs*. Un scénario est dit être exécuté correctement par un cas de test quand sa traduction en terme de code dans le cas de test est exécutée sans provoquer d'erreur.

Atteindre le critère *tous les scénarios* signifie que tous les comportements spécifiés dans les exigences ont été exhibés par au moins un cas de test. On ne peut déterminer si le critère *tous les scénarios* est satisfait qu'après exécution de l'ensemble des cas de test, de manière à disposer de tous les verdicts émis.

7.1.3.1 Les verdicts

Les cas de test générés peuvent émettre trois types de verdict : *pass*, *fail*, et *inconclusive*. Ils sont déterminés à partir des assertions exécutables dérivées des contrats OCL présents dans les diagrammes de séquence.

- Le verdict *Fail* est émis quand une assertion est violée pendant l'exécution du cas de test. Les assertions telles que nous les avons définies vérifient que le comportement du système sous test et son état interne sont corrects. Si ce n'est pas le cas, alors une erreur est détectée, et le rapport de test précise à quel endroit le cas de test a échoué.
- Le verdict *Pass* est émis quand le cas de test s'est exécuté correctement et totalement, aucune erreur n'est alors détectée.
- Le verdict *Inconclusive* est émis quand un cas de test doit être avorté à cause d'une contrainte non vérifiée en cours d'exécution.

Pour expliquer l'introduction du verdict inconclusif, prenons l'exemple d'un système de guichet automatique, avec les cas d'utilisation *retrait* et *dépôt*. Un objectif de test généré pour ce système pourrait être [dépôt(m1), retrait(m2)], avec m1 et m2 deux instances d'entités représentant des montants. Si on a attribué les valeurs m1=10 et m2=100, le cas de test généré va essayer de faire un retrait de 100 unités monétaires après un dépôt de 10 unités, et attendre un succès. Cependant, si le compte n'était pas suffisamment alimenté, le cas de test va, en fait, échouer, car la contrainte associée au scénario de retrait exprimant que le solde doit être supérieur au montant du retrait ne sera pas satisfaite. Cela ne veut pas dire pour autant que l'objectif de test n'est pas atteignable, ou n'a pas de sens : cela signifie simplement que les données de test fournies ne permettent pas d'exhiber le comportement attendu. C'est pour cela que le verdict inconclusif a été introduit.

Les contraintes telles que nous les avons définies ajoutent des conditions sur la bonne exécution d'un diagramme de séquence, et sont donc des gardes sur la poursuite de son exécution. En cours d'exécution, si une telle garde est évaluée à faux, alors le cas de test doit s'arrêter, cela ne signifie cependant pas que le cas de test a détecté une erreur. Cela peut signifier que la séquence instanciée choisie pour implémenter le diagramme de séquence est incorrecte. Ce cas peut provenir de sous-spécification (ou d'erreurs dans la spécification) ou des données de test utilisées pour l'instanciation. Dans le cas des scénarios de test de robustesse, cela peut aussi provenir du fait qu'un cas d'utilisation instancié a été remplacé par un de ses scénarios instanciés exceptionnels qui ne s'appliquaient pas dans le contexte de l'objectif de test. De manière à ne pas rejeter d'implémentation correcte, nous introduisons donc le verdict inconclusif, et avertissons le testeur du scénario qui n'a pas pu être joué. Le rapport de test détaille quels scénarios n'ont pas pu être exécutés, et dans quel contexte, et précise dans quels autres cas de test ce même scénario a pu être joué, de manière à guider l'analyse du testeur.

La section 7.1.3.3 propose une technique pour réduire le nombre de verdicts inconclusifs.

7.1.3.2 Atteinte du critère *tous les scénarios*

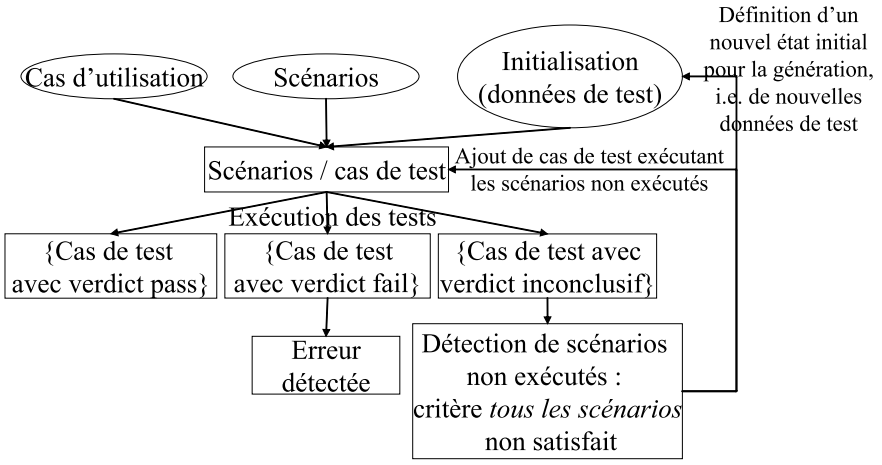


FIG. 7.10 – Analyse des verdicts

Comme illustré Figure 7.10, après l'exécution des cas de test, le critère *tous les scénarios* peut être satisfait ou pas. S'il est satisfait, alors on considérera le système conforme à ses spécification vis-à-vis de ce même critère. S'il ne l'est pas, cela signifie que l'ensemble des cas de test est à modifier (dans le cas de la présence de verdicts inconclusifs) ou qu'il y a des erreurs dans le programme sous test qu'il faut corriger avant de recommencer la phase de test (dans le cas de verdicts *fail*). Pour réduire le nombre de verdicts inconclusifs, deux solutions peuvent être utilisées. La première solution est manuelle et consiste à créer des cas de test permettant l'exécution des scénarios non exécutés. La deuxième solution consiste à regénérer de nouveaux cas de test, en modifiant l'état initial de la simulation permettant la génération d'objectifs de test. L'idée est de modifier les données de test. Cette nouvelle initialisation doit être déterminée en étudiant les assertions qui ont fait échouer le cas de test, de manière à déduire les données de test capables de les satisfaire. Cette phase est manuelle mais pourrait être automatisée en utilisant un solveur de contraintes [Marriott and Stuckey, 2000]. Par ailleurs, dans les deux solutions, il reste une phase de suppression des cas de test non pertinents. Ainsi, sur l'exemple de guichet automatique utilisé pour expliquer le verdict inconclusif, supposons que le scénario correspondant à un retrait ne soit jamais atteint, du fait que le cas de test le contenant amène à faire des retraits supérieurs au solde. Soit on générera de nouveaux objectifs avec un montant initial supérieur à 90 unités, soit on introduira un nouveau cas de test avec un dépôt de 100 unités suivi d'un retrait de 100 unités. Ainsi, le scénario de retrait pourra être exécuté.

La section suivante propose une solution pour réduire le nombre des verdicts inconclusifs dans le cas des objectifs de test de robustesse.

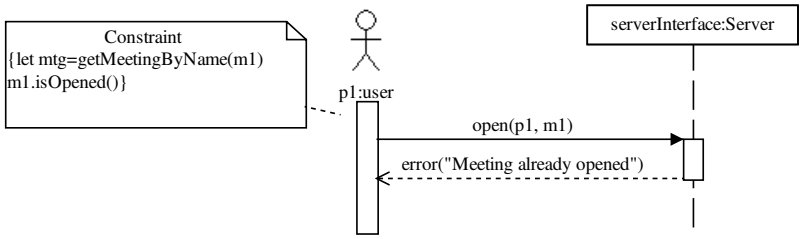


FIG. 7.11 – Exemple de cas de test inconclusif évitable

7.1.3.3 Amélioration possible pour la réduction du nombre des verdicts inconclusifs

La technique de génération de scénarios de test de robustesse décrite ci-avant est susceptible de générer un grand nombre de cas de test aboutissant à des verdicts inconclusifs.

Exemple de verdict inconclusif évitable. À titre d'exemple, étudions le cas d'utilisation *open* du système de réunion virtuelle qui échoue dans trois configurations distinctes (le participant tentant l'ouverture n'est pas le modérateur, la réunion est déjà ouverte, la réunion a été fermée), et supposons qu'on ait généré trois objectifs de test de robustesse amenant dans ces configurations (par exemple *ot1=[open(p1, m1)]*, *ot2=[open(p2, m1), open(p2, m1)]* et *ot3=[open(p2, m1), close(p2, m1), open(p2, m1)]* avec un système initialisé avec *{connected(p1), connected(p2), planned(m1), manager(p1, m1), moderator(p2, m1)}*).

Il est probable que dans les scénarios exceptionnels décrivant le cas d'utilisation en question, on ait prévu 3 scénarios *s1*, *s2*, et *s3* décrivant la réaction du système dans de tels cas.

Notre technique de génération va générer pour ces trois objectifs de tests et ces trois scénarios au moins 3*3=9 cas de test distincts, dont six non pertinents qui vont générer des verdicts inconclusifs à l'exécution. Par exemple, on va construire à partir de l'objectif de test *ot1* le scénario donné Figure 7.11, et obtenu en remplaçant le cas d'utilisation *open* par le scénario *s2*, alors que *ot1* était « prévu » pour être appliqué avec *s1*. Ce cas de test générera un verdict inconclusif à l'exécution, puisque la réunion *m1* n'est pas ouverte.

Cependant, dès lors que l'on est capable d'associer une configuration d'échec avec le scénario lui correspondant, on peut ne générer que trois scénarios de test pertinents, un par configuration d'échec.

Solution proposée. Pour exprimer un tel rapprochement entre une configuration particulière et un scénario, nous proposons que les scénarios exceptionnels puissent posséder des pré-conditions écrites en UCL, le langage de prédicats introduit pour les

contrats des cas d'utilisation. Ces pré-conditions précisent en termes de prédicats de haut niveau quelles sont les conditions décrivant la configuration permettant le déroulement particulier du cas d'utilisation décrit. De telles pré-conditions permettent de supprimer automatiquement un certain nombre de scénarios de test non pertinents. Par exemple, si le scénario *s2* possède la contrainte UCL *opened(p1)*, alors on peut statiquement décider que *s2* ne peut pas remplacer le cas d'utilisation *open* dans l'objectif de test *ot1*, puisque sa contrainte UCL n'est pas vérifiée.

7.1.4 Scénarios et cas de test

Dans les sections précédentes, nous avons supposé qu'on disposait de scénarios dont les paramètres étaient exactement les mêmes que ceux des cas d'utilisation. Dans un tel cadre, et en supposant également qu'on dispose de scénarios concrets et complets (« sans trous »), précisant bien pour chaque appel de méthode les paramètres exacts d'appels, en terme de paramètres du scénario ou de valeurs constantes, alors les scénarios de test générés par notre approche peuvent être considérés comme des cas de test.

Dans le cas où ces hypothèses ne sont pas réunies, les scénarios de test générés ne peuvent pas être considérés comme des cas de test : il peut y manquer certains échanges de messages, et certains paramètres peuvent être non instanciés ou même manquants.

Ainsi, si on se place dans le cas le plus général où l'on suppose qu'on dispose de scénarios potentiellement incomplets, les scénarios de test générés doivent être complétés pour les transformer en cas de test. Les outils de synthèse de test sont alors adaptés pour une telle complétion automatique, cela fait l'objet de la section suivante.

7.2 Utilisation d'outils de synthèse de cas de test

Pour compléter automatiquement des scénarios de test obtenus comme expliqué dans la section précédente, il est naturel de penser à l'utilisation de mécanismes de synthèse automatique de test : les outils de synthèse permettent d'explorer le graphe comportemental d'un système à partir de ses spécifications, afin d'en extraire des cas de test.

Dans cette section, nous expliquons dans un premier temps le fonctionnement des outils de synthèse qui nous paraissent les plus adaptés à l'usage que nous recherchons, nous en soulignons également quelques limitations. Comme la synthèse de test nécessite d'être fortement guidée quand les systèmes manipulés sont de taille réelle, nous introduisons des guides appelés patrons de test comportementaux, qui sont générés automatiquement. La contribution principale de cette section est de permettre une utilisation réaliste d'outils de synthèse, grâce à la génération automatique de guides à la synthèse de test.

7.2.1 Principes des outils de synthèse de test

Nous détaillons ici le couplage des deux outils prototypes UMLAUT ((UML all pUrposes Transformer) [UMLAUT, 2002] et TGV (Test Generation from transitions systems using Verification techniques)[Jard and Jéron, 2002], permettant la synthèse de test à partir de modèles statiques UML. Ce couple d'outils nous a semblé le plus adapté à notre contexte puisqu'il permettait l'utilisation de modèles UML, et que la synthèse de test était guidée par des objectifs de test. Le guidage par un objectif de test est une caractéristique très importante dans notre contexte, et n'est pas commune à tous les outils de synthèse. Ainsi, l'outil de synthèse Agatha [Lugato et al., 2002] ne proposait pas jusqu'à une période récente de synthèse guidée. Cependant, dans le cadre du projet MUTATION, un couplage entre nos outils de génération d'objectifs de test et l'outil de synthèse Agatha est en cours de développement. Le choix d'UMLAUT et de TGV est bien sûr également dû au fait qu'ils ont été développés au sein de l'équipe Pampa, ancêtre de l'équipe Triskell, et que nous avons donc plus de facilités pour les utiliser.

UMLAUT est un outil de transformation de modèles UML. Le module plus particulier qui est utilisé dans le cadre de la synthèse de test est le simulateur. Ce simulateur permet d'opérer sur un modèle UML une transformation permettant de le rendre simulable. L'interface de simulation peut être utilisée par d'autres outils, dans notre cas TGV. TGV est un outil de synthèse de test basé sur les systèmes de transitions étiquetées (IOLTS). Le principe est la construction à la volée d'un graphe comportemental, guidée par un objectif de test donné sous forme d'un IOLTS.

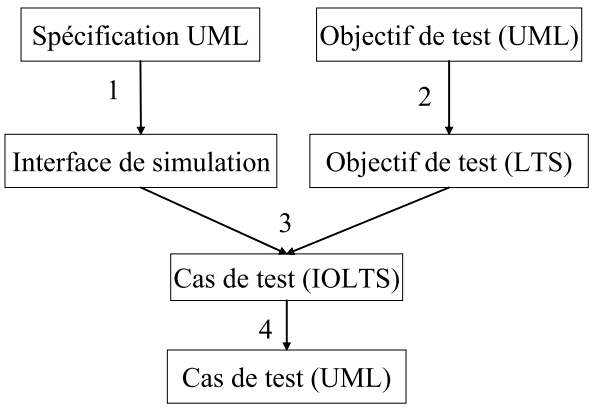


FIG. 7.12 – Synthèse de test avec UMLAUT et TGV

Le couplage UMLAUT / TGV pour la génération de test à partir de modèles UML est résumé Figure 7.12, et se compose des quatre étapes principales suivantes :

1. Dérivation d'une spécification formelle, étape au cours de laquelle une sémantique est donnée à la spécification UML du système sous test en termes de système de transitions étiqueté (LTS). Concrètement, il s'agit de la génération d'une interface de simulation à partir de la spécification, permettant à un LTS d'être construit incrémentalement (« à la volée »),
2. Dérivation formelle de l'objectif de test, i.e. transformation de diagrammes de séquence vers des LTS

3. Synthèse de test sur les modèles formels, phase au cours de laquelle un cas de test est synthétisé sous forme d'IOLTS à partir de la spécification et de l'objectif de test
4. Dérivation d'un cas de test UML, c'est-à-dire transformation de l'IOLTS synthétisé sous forme de diagramme de séquence UML.

Nous décrivons plus précisément chacune de ces étapes en annexe page 99, plus de détails sont donnés dans [Pickin, 2003, Pickin et al., 2002] pour le couplage UMLAUT / TGV et dans [Ho et al., 1999, Jard and Jéron, 2002] pour le fonctionnement exact de chacun des outils (notamment les principes des algorithmes pour TGV).

Pour faire court, les objectifs de test se composent de scénarios positifs qui expriment le comportement que le testeur souhaite exhiber, et de scénarios négatifs qui expriment les comportements que le testeur ne veut pas voir apparaître. Les deux outils combinés permettent de rechercher dans la spécification UML du système un scénario faisant apparaître les scénarios positifs désirés sans faire apparaître les scénarios négatifs. Le principe de la recherche se base sur l'exploration à la volée du graphe comportemental du système déduit du modèle UML.

7.2.2 Trois limitations et les solutions apportées

Nous abordons ici trois limitations qui nous ont semblé importantes lors de l'utilisation de cette technique de synthèse de test. Nous apportons une solution pour chacune.

Limitation 1 : méthode de conception

Une première limitation de cette approche provient du fait qu'elle n'est pas reliée à une démarche méthodologique l'insérant dans le cycle de vie du logiciel sous test : il est important de décrire à quel niveau sont décrits les objectifs de test et donc les scénarios les composant. Il est aussi important de pouvoir capitaliser les acquis.

Solution proposée : Nous avons donc proposé une démarche méthodologique permettant d'insérer cette technique dans un développement UML. Nous avons de plus introduit la notion de *préfixe de test*. Cette notion permet de capitaliser les scénarios menant à certains états-clefs du système, et points de départ de beaucoup d'objectifs de test. Nous nommons *patrons de test comportementaux* les objectifs de test ainsi composés de préfixes, d'un scénario positif et de scénarios négatifs. Nous décrivons plus précisément la méthodologie leur étant associée dans la section suivante.

Limitation 2 : traitement de l'héritage

Une autre limitation provient du traitement de l'héritage dans les objectifs de test. En effet, supposons que l'on dispose dans l'exemple de réunion virtuelle d'une hiérarchie de classes représentant toutes les réunions, dont la classe mère est *Meeting*. Supposons maintenant que l'on souhaite écrire un objectif de test où l'on ne se soucie

pas du type de réunion, par exemple, on veut s'assurer qu'il est possible d'ouvrir une réunion quelconque. On souhaiterait alors pouvoir écrire un scénario mettant en jeu une instance du type *Meeting*, sous-entendu une instance de type *Meeting* ou de ses sous-classes. Cependant, ceci n'est pas possible avec la méthode décrite ci-dessus, puisqu'un tel scénario va être dérivé en un LTS contenant une étiquette fixant statiquement le type meeting.

Solution proposée : Nous proposons donc une pré-compilation de l'objectif de test permettant de prendre en compte cette limitation. Cette pré-compilation résout une partie de la généricité des objectifs de test : il résout la généricité sur les instances d'objets pour les scénarios positifs et préfixes, ainsi que la généricité sur les types non-primitifs et les variables. La généricité restante est résolue par la synthèse de test. Pendant la pré-compilation, les vrais objets à utiliser doivent être trouvés dans le diagramme d'objets initial (pour les objets dynamiquement créés, des conventions de nom sont utilisées). L'algorithme 2 propose une version simplifiée de cet algorithme. Le principe est que chaque scénario positif est remplacé par son instanciation en utilisant le premier objet trouvé du bon type, alors que les scénarios négatifs sont remplacés par toutes leurs instanciations possibles.

Algorithme 2 Algorithme de pré-compilation de l'objectif de test

```
boolean done=false
// treatment of reject scenarios
for each bTP in ens_bTP until done
  find a declaration w=* :T with a wildcard or variable in s∈btp.reject
  for each subtype T'of T {
    for each object o :T' in OD {
      btp'=clone(btp)
      btp'.replace(w,o :T')
      ens_bTP.add(btp')}}
    done=not(ens_btp owns a btp |reject.own(* :T))}
  done=false
// treatment of the accept and prefixscenarios
for each bTP in ens_bTP until done
  find a declaration w with a wilcard or variable in s∈∪btp.accept
  case w=*.* {irrelevant}
  case (w=* :T){
    for each subtype T'of T {
      choose one object o :T' in OD
      btp'=clone(btp)
      btp'.replace(w,o.name :o.type)
      ens_bTP.add(btp')}
  case (w=obj :*){
    find o in OD |o.name=obj
    btp'=clone(btp)
    btp'.replace(w,o.name :o.type)
    ens_bTP.add(btp')}
done=not(ens_btp owns a btp | (prefix or accept).own(* :T or x :*))
}
```

Limitation 3 : efficacité

La troisième limitation concerne l'efficacité de l'approche proposée. Si les objectifs de test ne sont pas assez précis et si le système sous test est complexe (en terme de la taille de son graphe comportemental), alors une grande partie du LTS de la spécification devra être construite afin de synthétiser le cas de test, ce qui implique d'une part que la génération est longue, et d'autre part qu'elle nécessite une grande capacité mémoire.

Solution proposée : Nous proposons pour cela de générer automatiquement des patrons de test comportementaux les plus précis possible à partir des informations sur les dépendances entre cas d'utilisation dont nous disposons grâce à leurs contrats. Nous allons chercher à exprimer le plus d'objectifs de test négatifs que nous le pouvons, de manière à réduire la taille du LTS que la synthèse nécessite de construire.

7.2.2.1 Les patrons de test comportementaux et méthodologie

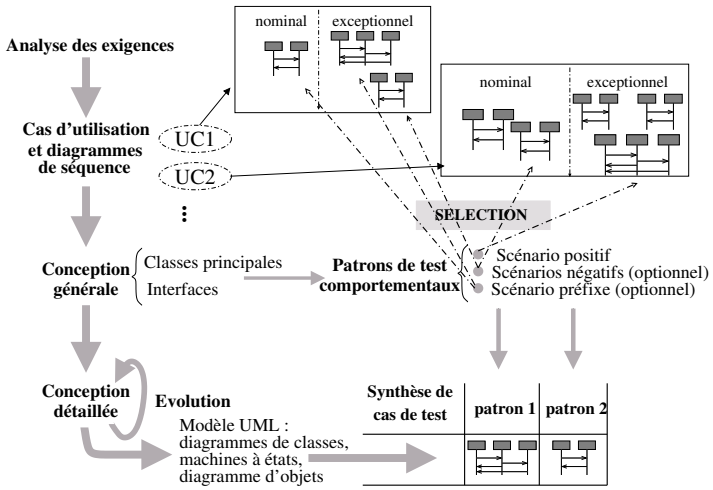


FIG. 7.13 – Méthodologie associée aux patrons de test comportementaux

Notre méthodologie de synthèse de cas de test est schématisée Figure 7.13. Dans cette méthode, on suppose que l'analyse des exigences produit un ensemble de cas d'utilisation et de scénarios. Dès lors que les interfaces du système sont fixées, alors le testeur peut créer un ensemble de patrons de test comportementaux (*Behavioral Test Patterns*) qui définissent ses objectifs de test qui peuvent être vus comme des exigences de test. Les patrons de test ont pour but de guider le plus efficacement possible la synthèse de test, notamment grâce aux scénarios négatifs.

Un patron de test comportemental est constitué :

- d'un scénario positif, qui représente ce que le testeur souhaite observer ;
- d'un préfixe, qui représente la spécification du comportement nécessaire pour placer le système dans un état dans lequel le comportement souhaité peut être observé. Le préfixe sert principalement à factoriser les parties des scénarios positifs

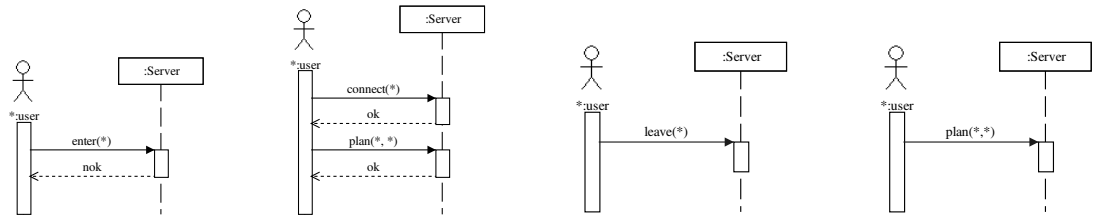
qui peuvent être communes à plusieurs patrons de test. Dans l'exemple de la réunion virtuelle, il peut ainsi être intéressant de disposer d'un scénario préfixe plaçant le système dans un état où plusieurs personnes sont connectées et une réunion est planifiée ;
- et d'un ensemble de scénarios négatifs, qui représentent l'ensemble des comportements qu'on ne souhaite pas observer, parce qu'ils ne sont pas pertinents dans le cadre de l'objectif de test.

Comme on le voit, les patrons de test comportementaux sont très proches des objectifs de test tels qu'ils sont définis dans la section 7.2.1 et l'annexe E.2. Leur intérêt réside d'abord dans la facilité de capitalisation des acquis, grâce notamment aux scénarios préfixes. La capitalisation des acquis de test est un point crucial de la méthodologie de test de lignes de produits que nous proposons dans le chapitre 9. Un autre intérêt réside dans le guidage des outils de synthèse : la présence d'un préfixe réduit l'exploration du graphe comportemental.

Construire un patron de test comportemental revient juste à sélectionner parmi les scénarios attachés aux cas d'utilisation lesquels doivent être observés, et lesquels ne doivent pas l'être. Un scénario préfixe est une composition séquentielle forte de plusieurs scénarios élémentaires attachés aux cas d'utilisation.

On peut se placer dans une approche manuelle, où cette sélection sera faite par le testeur lui-même. On verra également dans la section suivante comment cette sélection peut être faite automatiquement à partir des cas d'utilisation contractualisés.

Nous illustrons cette notion de patron de test comportemental et de synthèse de test sur la réunion virtuelle. Soit le patron de test présenté Figure 7.14. L'objectif de ce patron de test est d'exhiber le fait qu'un utilisateur peut se voir refuser l'entrée dans une réunion. On se place dans un cadre où une réunion est planifiée, et on interdit d'autres planifications de réunions, et les sorties de réunions.

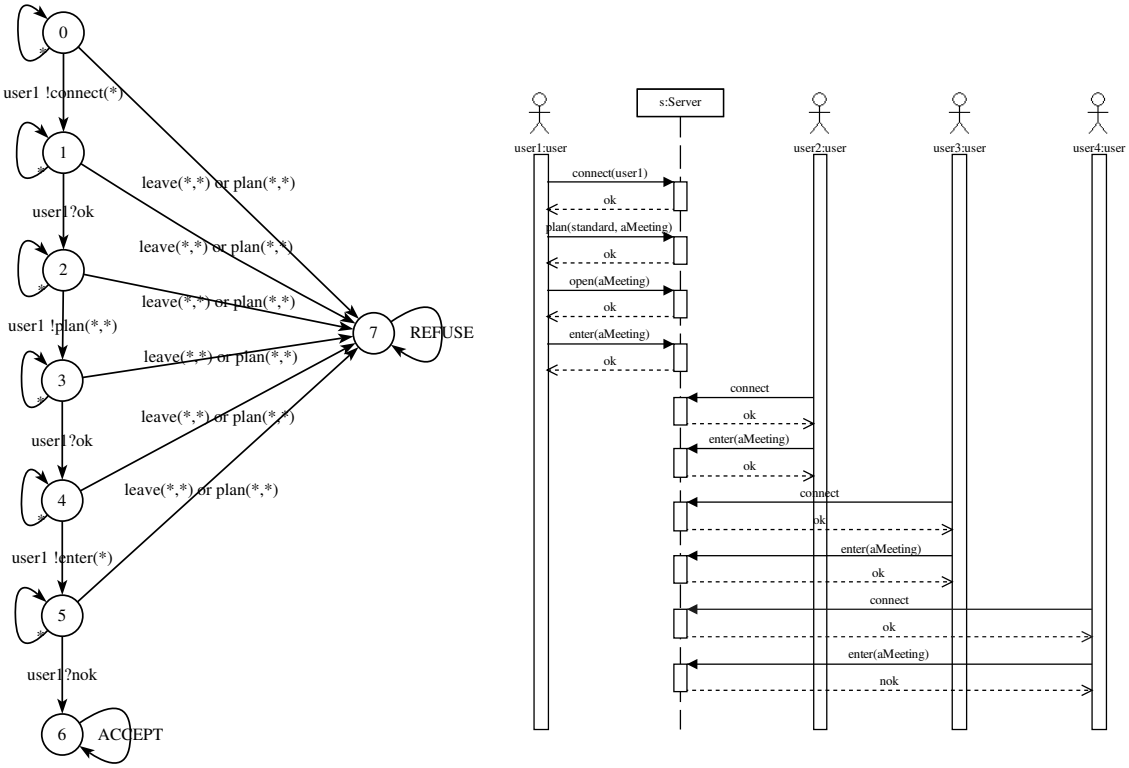


(a) Scénario positif (b) Préfixe (c) Scénario négatif 1 (d) Scénario négatif 2

FIG. 7.14 – Exemple de patron de conception

À partir d'un tel patron de test comportemental, on produit l'objectif de test TGV présenté dans la partie gauche de la figure 7.15. La partie droite de la figure 7.15 présente un exemple d'objectif synthétisé, dans lequel un utilisateur se voit refuser l'entrée dans la réunion parce que le nombre maximum de participants dans cette

réunion (fixé ici à 3) est atteint.



(a) Objectif de test TGV

(b) Exemple de cas de test synthétisé

FIG. 7.15 – Objectif de test TGV et cas de test synthétisé pour le patron de test de la figure 7.14

7.2.3 Couplage de la synthèse de test à partir des modèles et à partir des exigences

Nous présentons ici l'interaction entre la méthode de génération d'objectifs et de scénarios de test présentée chapitre 6 et section 7.1, et la méthode de synthèse de cas de test présentée ci-avant. L'idée est de générer automatiquement des patrons de test comportementaux à partir des cas d'utilisation contractualisés et de leurs scénarios.

7.2.3.1 Génération de patrons de test comportementaux

La méthode de génération d'objectifs de test présentée dans le chapitre 6 produit des objectifs de test sous forme de séquences de cas d'utilisation instanciés. L'utilisation des scénarios telle que nous l'avons présentée dans la section 7.1 permet de transformer ces objectifs de test en scénarios de test. Ici, il s'agit de générer des patrons de test

comportementaux, c'est-à-dire un scénario préfixe, un scénario objectif, et un ensemble de scénarios à rejeter.

Supposons que l'on veuille générer des patrons de test comportementaux à partir de l'objectif de test $[cui_1, ..., cui_{n-1}, cui_n]$. Pour générer le préfixe et le scénario positif, on adopte la même technique de remplacement des cas d'utilisation par les scénarios que celle présentée à la section 7.1.2.2, à ceci près que la séquence de scénario correspondant aux cas d'utilisation instanciés $[cui_1, ..., cui_{n-1}]$ formera le préfixe, et que le scénario instancié substitué au cas d'utilisation cui_n formera le scénario positif. Tous les scénarios exceptionnels instanciés seront ajoutés comme scénarios négatifs au patron de test comportemental, ainsi que tous les scénarios instanciés correspondant à des cas d'utilisation instanciés ne manipulant aucune des entités manipulées dans l'objectif de test $[cui_1, ..., cui_{n-1}, cui_n]$. Bien sûr, quand on parle de « tous les scénarios », c'est à l'exception de ceux présents dans le préfixe et dans le scénario positif. La sélection des scénarios négatifs se fonde ainsi sur les hypothèses que :

- d'une part l'échec d'un cas d'utilisation n'influe pas sur les autres cas d'utilisation, ou alors que l'effet de bord est spécifié dans les cas d'utilisation ;
- d'autre part toutes les interactions entre les entités manipulées dans le système sont spécifiées par les contrats des cas d'utilisation.

Sous ces deux hypothèses, la génération des scénarios négatifs assure que les cas de test générés ne contiennent pas un échec d'un cas d'utilisation, et ne comprennent pas de comportement n'ayant rien à voir avec les instances manipulées dans l'objectif de test. De cette façon, on réduira considérablement l'espace d'état construit pour aboutir au cas de test.

7.2.3.2 Synthèse sur le couplage

L'approche que nous proposons ici en couplant la synthèse de test à partir des modèles et à partir des exigences est résumée Figure 7.16. Les patrons de test comportementaux sont générés lors de la phase notée ① dans la figure 7.16 dès lors que l'on dispose des cas d'utilisation et de scénarios les documentant. Puis, quand la conception générale est terminée, les cas de test peuvent être synthétisés par UMLAUT et TGV. Nous préconisons des contrôles manuels du testeur dans les phases notées ① et ② Figure 7.16 : nous considérons que le testeur doit contrôler les patrons de test comportementaux afin de vérifier s'ils sont bien pertinents, et éventuellement en ajouter certains lui paraissant nécessaires ; il en est de même au niveau des cas de test synthétisés.

Ce couplage nécessite de garder une cohérence entre le modèle de simulation des cas d'utilisation et du modèle. En effet, ces deux méthodes de simulation nécessitent d'une part une énumération des types utilisés (énumération des entités pour la simulation des cas d'utilisation, et machine à états des acteurs – contenant la totalité des appels de méthode que l'acteur peut effectuer, avec les paramètres effectifs – pour la simulation du modèle UML), et d'autre part la donnée de l'état initial (sous forme de prédicats instanciés pour les cas d'utilisation, et d'un diagramme d'objets pour le modèle UML). Il est alors nécessaire d'assurer une cohérence entre les deux énumérations et les deux modélisations des états initiaux.

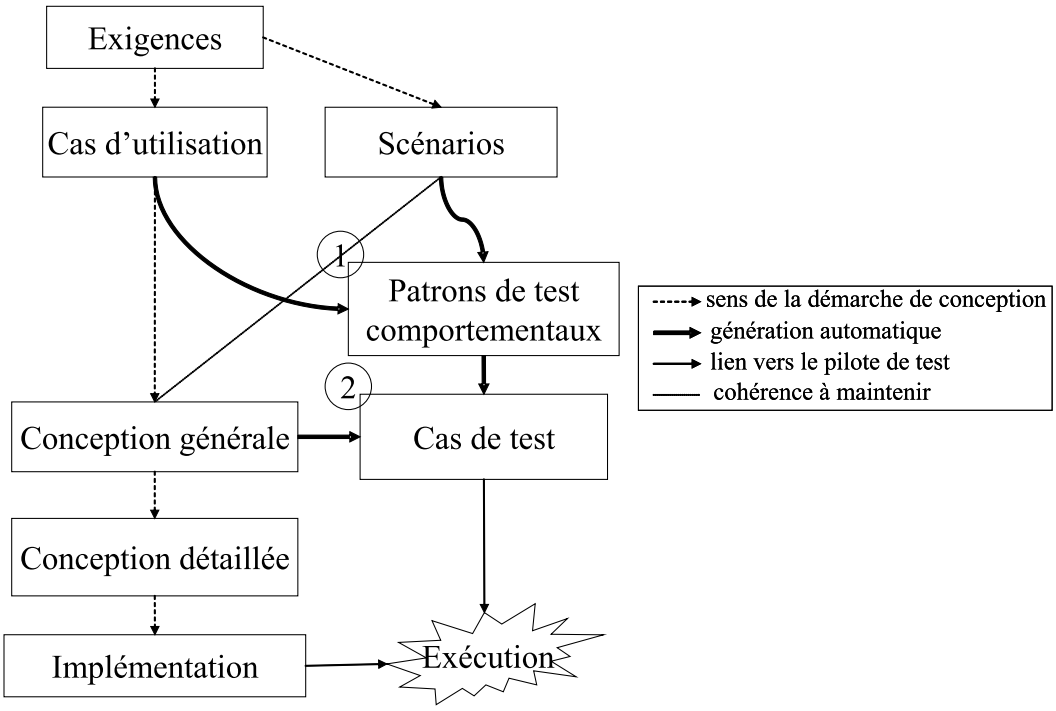


FIG. 7.16 – Génération de patrons de test comportementaux et synthèse de cas de test

Quand on utilise une telle chaîne de génération, on peut détecter des erreurs à différents niveaux. Si la synthèse de test échoue (phase ② dans la figure 7.16), l'erreur peut provenir soit de la modélisation des cas d'utilisation, soit de la modélisation UML utilisée pour la simulation (conception générale), soit dans la phase de raffinement du modèle vers le code (de la conception générale à la conception détaillée et à l'implémentation). Si un test détecte une erreur, alors cette erreur provient de la phase de raffinement du modèle vers le code.

Quand tous les cas de test produits par cette chaîne de génération ont été exécutés en produisant un verdict de succès, on considère que toutes les exigences telles qu'elles sont décrites dans les cas d'utilisation et les scénarios se retrouvent dans le code, vis-à-vis des critères de test utilisés tout au long de la chaîne : critère de sélection des objectifs de test d'une part et critère *tous les scénarios* d'autre part.

7.3 Conclusions

Dans ce chapitre, nous avons proposé une approche pour transformer les objectifs de test en cas de test. Dans un premier temps, nous avons expliqué comment utiliser les scénarios attachés aux cas d'utilisation pour transformer un objectif de test en un scénario de test. Dans le cas où l'on dispose de scénarios complets, ces scénarios de test peuvent être considérés comme des cas de test. Dans un cadre plus général, les scénarios de test devront ensuite être complétés, soit manuellement par le testeur, soit automatiquement en utilisant des outils de synthèse de test. Nous avons ainsi proposé

une méthode pour transformer les objectifs de test en patrons de test comportementaux plus précis que les scénarios de test, qui servent ensuite à guider un outil de synthèse de test. Cette approche nécessite de disposer d'un modèle UML simulable de l'application, et elle est limitée par le manque de maturité actuel des outils de synthèse de test.

En revanche, elle permet d'obtenir une chaîne complète de génération des exigences vers les cas de test.

Les principales contributions de ce chapitre sont :

- la proposition d'une approche pragmatique et peu coûteuse pour transformer un objectif de test en scénario de test, en utilisant les scénarios comme lien de traçabilité entre les cas d'utilisation et le modèle de conception,
- la proposition d'une approche plus élaborée permettant d'utiliser des techniques de synthèse de test à partir des spécifications du système sous test en termes de machines à états et de diagrammes de classes (cette approche est toutefois limitée par le manque de maturité des outils de synthèse de test),
- ainsi que des solutions pour pallier partiellement les problèmes liés à la synthèse de test à partir de modèle UML (méthodologie de test s'inscrivant dans le cycle de développement, traitement de l'héritage, coût de génération).

Ce chapitre termine la description de notre chaîne de génération de test à partir des exigences. Les chapitres suivants présentent la validation de cette approche : le chapitre 8 présente son application sur plusieurs études de cas académiques ainsi que sur une étude de cas industrielle. Le chapitre 9 présente son application pour les lignes de produits.

Chapitre 8

Expérimentations et résultats

Ce chapitre présente une validation expérimentale de la méthode de génération de test présentée dans ce manuscrit. Cette validation se compose de trois parties. La première partie est une validation sur trois études de cas académiques. La deuxième partie vise à comparer notre approche d'ajout de contrats aux cas d'utilisation pour les ordonnancer à une approche plus classiquement utilisée et basée sur les diagrammes d'activités. La troisième partie décrit les expériences faites dans le cadre de la collaboration avec Thalès Airborne System dans le projet MUTATION. Cette partie est en quelque sorte une validation industielle de notre approche.

8.1 Études de cas académiques

Cette section présente une validation académique de l'approche de test développée dans les chapitres 5 à 7, sur trois études de cas académiques : l'exemple du système de réunions virtuelles présenté Section 4.3, un serveur FTP, et un système de guichet automatique de banque.

Les expériences décrites dans cette section tendent tout d'abord à évaluer l'efficacité des tests générés. Nous utiliserons pour cela comme mesure d'efficacité la couverture de code. Ces expériences nous permettront aussi de comparer l'efficacité des différents critères de test proposés dans la section 6.3. Au cours de ces expériences, nous avons généré des scénarios de test que nous avons complétés à la main.

Nous n'avons pas utilisé la partie concernant les patrons de test comportementaux et la synthèse de test, du fait des problèmes rencontrés avec les outils de synthèse. Plusieurs raisons ont mené à cette absence de validation. Tout d'abord, l'outil UMLAUT est en totale refonte depuis deux ans. En effet, l'équipe Triskell développe UMLAUT NG, la nouvelle génération d'UMLAUT, basée sur un langage de transformation de modèles : MTL [Triskell, 2004] ; nous avons donc préféré ne pas investir dans le couplage entre la génération d'objectifs de test et UMLAUT ancienne génération, d'autant plus que cette version contient des erreurs notamment au niveau de la création dynamique d'objets, dues au problème de la représentation de l'état interne d'un système à objets. Ensuite, dans le cadre de la collaboration avec le CEA dans le projet Mutation,

nous nous sommes plutôt intéressé au couplage avec l'outil de synthèse de test Agatha [Lugato et al., 2002, Lugato et al., 2004]. Ce couplage n'a malheureusement pas encore abouti. Globalement, nous avons constaté le manque de maturité des outils de synthèse de test.

Dans cette section, nous n'utilisons pas le langage des exigences présenté dans le chapitre 5. Nous n'avons pas cherché à réécrire le cahier des charges de nos cas d'étude en LDE, nous nous sommes plutôt concentré sur les tests générés. La pertinence du LDE sera étudiée dans le cadre des expériences effectuées avec Thalès.

8.1.1 Présentation des études de cas

Nous avons cherché à estimer l'efficacité des tests générés en termes de couverture de code. Avant d'effectuer une telle analyse, nous avons fait une étude qualitative des différentes catégories de code que nous pouvions espérer couvrir, l'objectif étant de déterminer si notre approche est efficace pour couvrir le code fonctionnel. Ceci explique que nous avons dû nous limiter à trois systèmes de petite taille (de 500 à 2500 lignes de code).

Nous avons classifié le code en quatre catégories :

- Le code mort : certains de nos cas d'études contiennent du code mort. Ce code mort correspond à des accesseurs pertinents mais non utilisés; ils pourraient cependant être utilisés dans des futures évolutions du système. Le test système fonctionnel ne peut pas couvrir un tel code : il doit être testé pendant une phase de test unitaire. Pour les études qui suivent, nous avons enlevé ce code mort pour nous focaliser sur l'efficacité des tests générés sur le code atteignable.
- Le code de robustesse vis-à-vis des spécifications. Ce code s'assure que seules les spécifications requises sont présentes dans le système, il est chargé de réagir quand l'utilisateur formule des requêtes ne satisfaisant pas les spécifications. Il peut par exemple émettre des exceptions ou des messages d'erreur à la réception d'une commande incorrecte.
- Le code de robustesse vis-sà-vis de l'environnement. Ce code s'assure de la correction des entrées venant de l'environnement.
- Le code fonctionnel.

Nous avons analysé le code de nos trois cas d'études en fonction de ces quatre catégories.

Les trois cas d'études sont les suivants :

1. Un Guichet Automatique de Banque (GAB). Nous avons adapté une application déjà existante [Bjork, 2004], de manière à en découpler le cœur de l'interface graphique. Il fournit les principales fonctions suivantes : consultation, retrait, dépôt de chèques et de liquide, et transfert de compte à compte. L'implémentation que nous utilisons est composée de 850 lignes de code, et 186 instructions exécutables (hors commentaires, en-têtes de méthodes, déclaration de variables, ...).
2. Un serveur FTP déjà existant [Cueto, 2003]. Nous avons étudié une large partie de ce serveur, celle contenant les commandes les plus intéressantes. L'implémentation se compose de 500 lignes de code, dont 207 instructions exécutables.

	GAB	FTP	Virtual meeting
Code mort	0%	3,7%	9%
Robustesse vis-à-vis de l'environnement	0%	5,8%	8%
Robustesse vis-à-vis des spécifications	9,09%	17,8%	18%
Code fonctionnel	90,91%	72,7%	65%

TAB. 8.1 – Répartition du code pour nos 3 cas d'études

3. Le serveur de réunions virtuelles déjà présenté Section 4.3. Le système contient 2500 lignes de code, dont 780 instructions exécutables.

La répartition du code des 3 études de cas est présentée dans la table 8.1. Tandis que le GAB ne possède ni code mort, ni code traitant de robustesse vis-à-vis de l'environnement, le serveur FTP et le système de réunions virtuelles ont entre 9 et 17 % de ce type de code, qui ne peut pas être couvert par les cas de test générés par notre approche. Plus généralement, cela signifie qu'une approche de test basée sur les modèles d'exigences doit être complétée par une phase de test unitaire pour tester notamment le code mort et le code dépendant de l'environnement. Concernant la proportion de code dédié à la robustesse vis-à-vis de la spécification, il varie de 9% à 18%. Le code restant concerne le code fonctionnel implémentant les fonctions décrites par les cas d'utilisation. Cette proportion tend à décroître quand le système est plus grand.

8.1.2 Couverture de code et comparaison des critères

Dans cette section nous présentons les pourcentages de code couvert par les cas de test générés avec notre approche. Nous décrivons tout d'abord le protocole expérimental suivi, puis donnons et analysons les résultats obtenus.

8.1.2.1 Protocole expérimental

Pour chaque étude de cas, nous avons procédé comme suit.

1. Phase d'analyse des exigences. Nous avons pour chaque étude de cas déterminé les cas d'utilisation, ainsi que leurs contrats et leurs principaux scénarios illustratifs. Nous avons simulé les exigences exprimées à l'aide de cas d'utilisation avec le simulateur de cas d'utilisation, pour nous assurer de leur correction et dans une certaine mesure de leur complétude. Les nombres de cas d'utilisation et de scénarios utilisés sont donnés dans la table 8.2.
2. Phase de génération de test. Nous avons généré des objectifs de test pour chaque critère de test et pour chaque étude de cas. Les scénarios de test correspondants ont ensuite été générés. Il est à noter que du fait de la simplicité de nos études, les scénarios de test correspondaient, à la phase d'initialisation près, aux cas de test.
3. Phase d'exécution des tests. Nous avons ensuite exécuté tous les tests et grâce à l'outil de trace *Jtracor*[Fleurey, 2003], déduit le code couvert par chaque cas de

	GAB	FTP	Réunion virtuelle
Nombre de cas d'utilisation	5	14	14
Nombre de scénarios nominaux	5	14	14
Nombre de scénarios exceptionnels	14	33	80

TAB. 8.2 – Nombres de cas d'utilisation et scénarios utilisés

	GAB		FTP		Réunion virtuelle	
Critère	# OT	taille	# OT	taille	# OT	taille
Toutes les transitions	33	3	81	4	13841	11
Tous les nœuds	4	3	8	5	769	10
Tous les cas d'utilisation instanciés	9	2	12	3	50	5
Tous les nœuds et tous les cui	13	3	20	4	819	10
Tous les termes de pré-conditions	6	2	14	2	15	5
Robustesse	14	2	33	2	80	4

TAB. 8.3 – Statistiques sur les tests générés

test. Notons que, toujours du fait de la simplicité des études, et du fait qu'elles ne contenaient probablement pas de faute initialement, nous n'avons obtenu que des verdicts de succès (et aucun inconclusif ni aucun échec).

Les outils Nous avons utilisé les outils prototypes développés pour supporter notre approche (voir en annexe D page 191). Un premier outil traite les systèmes de cas d'utilisation contractualisés, et permet de simuler les cas d'utilisation et de générer les objectifs de test selon les différents critères de test définis. Nous avons développé un profil UML pour l'outil *Objecteering*, qui permet de générer le système de cas d'utilisation au format requis par notre outil. Ainsi, le modèle de cas d'utilisation peut être directement entré dans un modeleur UML adéquat.

Le deuxième outil que nous utilisons est connecté au premier et génère les scénarios de test et le matériel de test. Ce matériel de test reposant sur le canevas de test *JUnit*, nous pouvons ensuite directement exécuter le lanceur de test fourni par *JUnit*.

8.1.2.2 Statistiques sur les tests obtenus

Après exécution des cas de test, nous avons étudié quel pourcentage de code ils couvrent. Nous avons également étudié les tests générés, du point de vue de la taille et du nombre. La table 8.3 fournit les statistiques sur les objectifs de test générés. Cette table donne pour chaque critère de test et pour chaque étude de cas le nombre d'objectifs de test générés et leur taille moyenne, la taille étant donnée comme le nombre de cas d'utilisation instanciés dans le cas d'utilisation.

Pour chaque objectif, nous avons généré exactement un cas de test. Ceci est dû au faible nombre de scénarios utilisé, ainsi qu'au fait que nous avons utilisé des contrats

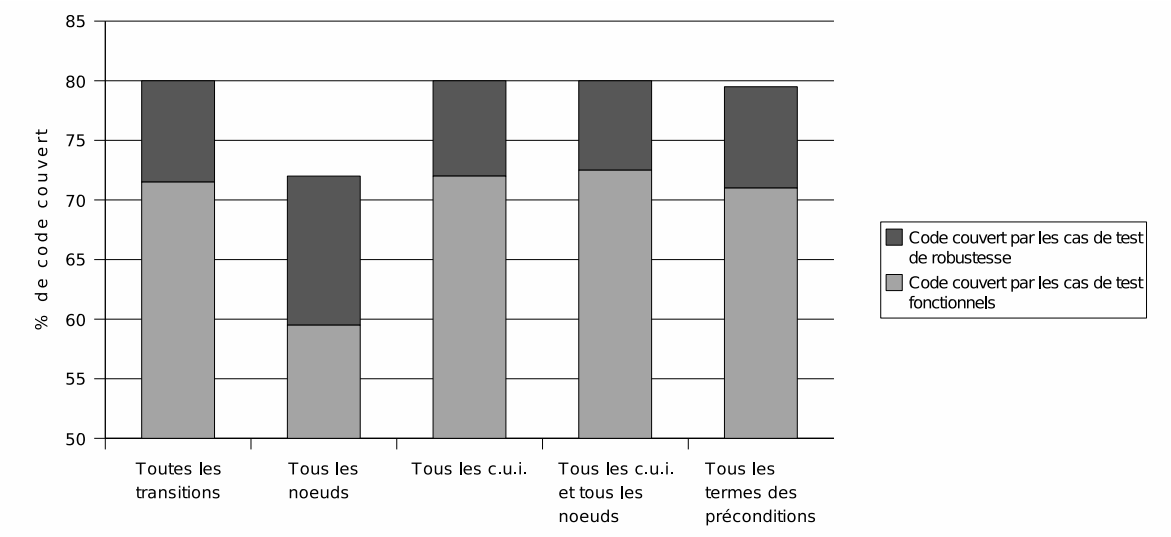


FIG. 8.1 – Comparaison des critères vis-à-vis de la couverture de code

en UCL dans les scénarios pour éviter les verdicts inconclusifs.

On peut d'ores et déjà remarquer que certains critères génèrent beaucoup plus de cas de test que d'autres. Les critères *toutes les transitions*, *tous les nœuds* et *tous les nœuds et tous les cas d'utilisation instanciés* génèrent un nombre beaucoup trop élévé de cas de test, ce qui rend ces critères inutilisables : il n'est pas acceptable que pour un système comme la réunion virtuelle, on génère 13841 cas de test, comme c'est le cas en utilisant le critère *toutes les transitions*. Comme nous l'avons expliqué dans la section 6.3, nous avons cherché à générer un petit nombre de cas d'utilisation, nous tiendrons compte de cet aspect dans notre comparaison des critères.

La deuxième remarque que l'on peut faire à la lecture de la table 8.3, c'est que l'objectif que nous souhaitions atteindre avec des objectifs de test courts est atteint, la taille moyenne des objectifs de test variant entre 2 et 11 cas d'utilisations instanciés.

8.1.2.3 Comparaison des critères

Cette section a pour but de fournir une comparaison des cinq critères fonctionnels donnés dans la section 6.3. Pour cette comparaison nous utilisons l'exemple de réunion virtuelle.

Les résultats concernant la couverture de code sont donnés Figure 8.1, sous la forme du pourcentage de code couvert par les cas de test fonctionnels. À l'exception du critère *tous les nœuds*, tous les critères atteignent 70% de couverture de code. Combinés avec le critère de robustesse, ces mêmes critères atteignent 80% de couverture de code.

Le taux de couverture atteint par le critère *tous les nœuds* est faible dans la mesure où couvrir les nœuds de l'UCTS n'assure pas que tous les cas d'utilisation sont appliqués au minimum une fois. Les cas d'utilisation dont l'application ne modifie pas l'état de l'UCTS ne sont en effet pas appliqués dans la mesure où ils apparaissent comme des

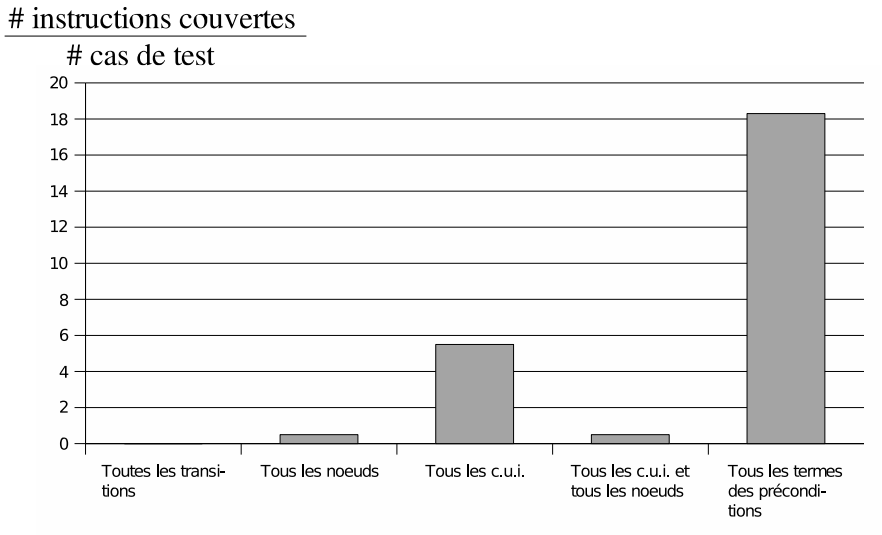


FIG. 8.2 – Comparaison des critères vis-à-vis de l'efficacité des cas de test

boucles dans l'UCTS, et que l'on recherche les chemins les plus courts pour couvrir les nœuds de l'UCTS.

Si les critères *toutes les transitions*, *tous les cas d'utilisation instanciés*, *tous les nœuds et tous les cas d'utilisation instanciés*, et *tous les termes des pré-conditions* sont équivalents pour la couverture de code, leur efficacité respective n'est pas équivalente. Dans la figure 8.2, nous proposons d'estimer leur efficacité avec le ratio entre le nombre d'instructions couvertes et le nombre de cas de test. Intuitivement, ce ratio correspond à la contribution d'un cas de test à la couverture de code. Il estime donc l'efficacité relative de chaque cas de test pour couvrir le code. Il apparait clairement dans la figure 8.2 que chaque cas de test généré avec les critères *toutes les transitions*, *tous les nœuds et tous les cas d'utilisation instanciés*, et *tous les nœuds* a en moyenne une faible efficacité. Les ensembles de cas de test correspondants sont en effet plus grand que ceux générés avec les critères *tous les termes des pré-conditions* et *tous les cas d'utilisation instanciés*, pour une couverture de code équivalente.

Il apparaît donc que les critères *tous les termes des pré-conditions* et *tous les cas d'utilisation instanciés* sont les plus efficaces.

8.1.2.4 Étude de la couverture de code

Les résultats de la table 8.4 montrent que le critère *tous les termes des pré-conditions* est efficace pour couvrir la plupart du code fonctionnel. La rapport entre le nombre de cas de test fonctionnels et le pourcentage de code couvert est fort : par exemple, avec seulement cinq cas de test pour l'étude la plus grosse (réunion virtuelle), nous couvrons 100% du code fonctionnel. En comparaison, le critère de robustesse est assez décevant. En effet, moins de la moitié du code de robustesse peut être couvert par le critère de robustesse, alors que le nombre de cas de test est plus grand que le nombre de cas de test fonctionnels. Globalement, si on considère que les cas de test sont générés directe-

	ATM	FTP	Virtual meeting
% de code fonctionnel couvert	100%	90,7%	100%
% de code de robustesse vis-à-vis de la spec. couvert	42,31%	38,6%	52%
% total de code couvert	94,76%	72,5%	80%

TAB. 8.4 – Étude de la couverture de code pour les 3 études de cas - utilisation des critères *tous les termes des pré-conditions* et *robustesse*

ment à partir des exigences, d'une manière automatique, la proportion de code globale qui est couverte est forte (entre 70% et 95%). Concernant la robustesse, l'approche proposée produit seulement des cas de test capables de violer les pré-conditions des cas d'utilisation. La génération des cas de test de robustesse pourrait être augmentée en prenant en compte les données de test de manière plus précise, par exemple en utilisant un solveur de contraintes [Marriott and Stuckey, 2000]. Un critère plus fort serait de générer des cas de test qui violent les assertions incluses dans les scénarios attachés aux cas d'utilisation, et pas seulement les pré-conditions des cas d'utilisation.

8.2 Comparaison avec une modélisation des dépendances entre cas d'utilisation à base de diagrammes d'activités

Plusieurs approches proposent d'utiliser les diagrammes d'activités pour exprimer les dépendances séquentielles existant entre les cas d'utilisation. Nous avons dans ce manuscrit proposé une expression déclarative de ces dépendances, avec des contrats. Pour comparer ces deux approches, nous proposons un modèle de diagrammes d'activités proche de celui de [Briand and Labiche, 2002] permettant de représenter les dépendances entre cas d'utilisation.

8.2.1 Des diagrammes d'activité aux diagrammes de cas d'utilisation

Les travaux de *Briand et al* [Briand and Labiche, 2002] ont inspiré l'idée d'exploiter les dépendances existant entre les cas d'utilisation pour guider la génération de test. Ils proposent d'exprimer de telles dépendances à l'aide d'un diagramme d'activité. En vue de pouvoir nous comparer avec cette approche, nous proposons un modèle permettant de traduire des dépendances exprimées sous forme de diagramme d'activité vers un modèle de cas d'utilisation contractualisés. Un tel mécanisme de traduction a également pour but que notre mécanisme de simulation puisse aussi supporter d'autres formats d'entrée que les cas d'utilisation contractualisés : l'utilisation de diagramme d'activité reste une pratique plus courante pour exprimer les dépendances entre cas d'utilisation, même si elle ne nous semble pas fiable, comme nous le verrons par la suite.

Dans cette section, nous présentons les diagrammes d'activité sur lesquels nous travaillons, puis nous détaillons la méthode de traduction vers un système de cas d'utilisation contractualisés.

8.2.1.1 Diagrammes d'activité manipulés

Rappels sur les diagrammes d'activité

Nous utilisons des diagrammes d'activité UML 2.0 plutôt que UML 1.x : leur sémantique a été largement clarifiée lors du passage à la version 2.0.

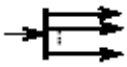
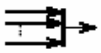
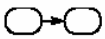
La sémantique des diagrammes d'activité 2.0 est proche de celle des réseaux de Pétri, elle est décrite en terme de jetons entrants et jetons sortants.

Les diagrammes d'activité sont utilisés pour modéliser des flots de contrôle et des flots de données (ou d'objets). Cependant, pour représenter les contraintes et le séquencement des cas d'utilisation, nous ne considèrerons que les diagrammes d'activité modélisant des flots de contrôle. En ce sens, nous ne nous intéressons qu'à un sous-ensemble de la norme UML concernant les diagrammes d'activité. La table 8.5 rappelle brièvement la notation et la sémantique des différentes constructions des diagrammes d'activité que nous traitons. Les diagrammes d'activité manipulés se composent donc principalement de nœuds d'action et de nœuds de contrôle (jonction, décision, branchement, fusion), reliés par des arcs appelés flots de contrôle. Le contrôle passe de nœuds en nœuds, symbolisé par un jeton qui peut transiter sur les flots de contrôle, en respectant les règles sémantiques données Table 8.5. Notons que la dénomination des diagrammes d'activité désigne par « flot de contrôle » un arc reliant des activités ou des nœuds de contrôle, tandis que dans la dénomination générale, un flot de contrôle représente plutôt les possibilités de passage de contrôle d'un point d'un programme à un autre, c'est-à-dire un chemin dans un diagramme d'activité. Pour ne pas confondre le sens classique de flot de contrôle et celui donné dans les diagrammes d'activité, nous mettrons le terme « flot de contrôle » en italique quand il s'agit du sens donné par les diagrammes d'activité.

Afin d'avoir un algorithme de transformation vers un système de cas d'utilisation contractualisés qui termine, on supposera que le nombre de jetons circulant dans le diagramme est fini, et qu'à chaque instant, il n'y a pas plus d'un jeton sur un *flot de contrôle*. On supposera également qu'on dispose de diagrammes d'activité dont les constructions implicites (jonction sur les entrées et branchement sur les sorties) ont été explicitées par des nœuds de jonction ou de branchement. Ainsi, chaque nœud représentant un cas d'utilisation ne possède qu'un flot entrant et un flot sortant.

Modélisation des dépendances entre cas d'utilisation avec des diagrammes d'activité

Nous nous basons sur une modélisation très proche de celle proposée par *Briand et al.* En ce sens, nous n'avons pas cherché à introduire la notion de paramètre de cas d'utilisation dans les diagrammes d'activité.

Type	Notation	Sémantique
Nœud initial (Initial node)		Point d'entrée pour invoquer une activité. Un jeton de contrôle est placé au noeud initial quand l'activité commence.
Nœud de fin d'activité (Activity final node)		Stoppe tous les flots dans une activité. Un jeton atteignant un nœud de fin d'activité fait avorter tous les flots en cours, l'activité est donc terminée et le jeton est détruit.
Nœud de fin de flot (Flow Final node)		Termine un flot. Le nœud de flot final détruit les jetons y entrant.
Nœud d'action (Action node)		Unité fondamentale de la fonctionnalité exécutable d'une activité. Une action s'exécute quand toutes les contraintes sur ses *flots de contrôle* entrants sont satisfaites (jonction implicite). L'exécution consomme les jetons de contrôle entrants puis présente un jeton sur chaque flot sortant (branchement implicite).
Nœud de décision (Decision node)		Choix parmi les flots sortants. Chaque jeton arrivant sur un nœud de décision ne peut traverser qu'un seul flot sortant. Les jetons ne sont pas dupliqués. Ce sont les gardes sur les flots sortants qui permettent le choix (les gardes doivent assurer le déterminisme du choix).
Nœud de branchement (Fork node)		Partage d'un flot en flots concurrents. Les jetons arrivant à un branchement sont dupliqués sur les flots sortants.
Nœud de jonction (Join node)		Synchronisation de plusieurs flots. Si un jeton de contrôle est offert sur chaque flot entrant, alors un jeton de contrôle est offert sur le flot sortant.
Nœud de fusion (Merge node)		Rassemblement de plusieurs flots. Tous les jetons offerts sur les flots entrants sont offerts sur le flot sortant sans synchronisation.
Flot de contrôle (Control flow)		Passage des jetons. Les jetons offerts par le nœud source sont offerts au nœud destination.
Partition d'activité (Activity Partition)		Identifie des actions ayant une caractéristique commune. Les partitions n'affectent pas le flot des jetons.

TAB. 8.5 – Diagrammes d'activité : notation et sémantique des constructions utilisées

Un diagramme illsutratif est donné à titre d'exemple pour la réunion virtuelle Figure 8.3. Chaque diagramme d'activité a une partition par acteur du système. Un cas

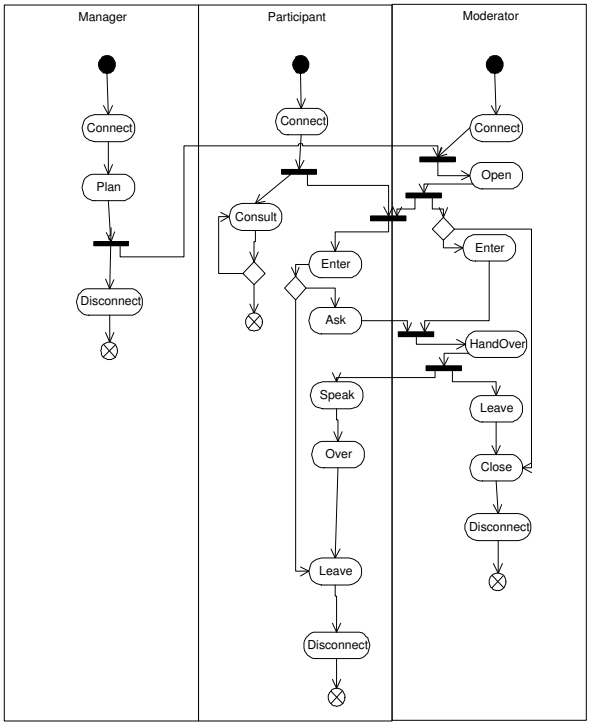


FIG. 8.3 – Diagramme d'activité simplifié pour la réunion virtuelle

d'utilisation est représenté par une action, et appartient à la partition correspondant à son acteur principal. Quand un cas d'utilisation a potentiellement plusieurs acteurs principaux (dans l'exemple de réunion virtuelle, tous les acteurs peuvent consulter), alors ce cas d'utilisation est dupliqué dans chacune des partitions correspondantes. Les contraintes entre les cas d'utilisation sont reflétées par l'utilisation de nœuds de synchronisation (branchement et jonction), ou simplement pour des relations de précédence simple par l'utilisation de *flots de contrôle*, et au besoin de nœuds de fusion. Ainsi, si l'exécution d'un cas d'utilisation nécessite l'exécution préalable des cas d'utilisation A et B, on utilisera une jonction pour se mettre en attente de la terminaison de A et B. Si on attend juste la terminaison d'un cas d'utilisation A, alors on relie juste A et le cas d'utilisation avec un *flot de contrôle*. Si on attend l'une ou l'autre des terminaisons de A ou B, on utilisera alors une fusion, etc. Les nœuds de décision sont utilisés pour refléter un choix possible de l'utilisateur.

8.2.1.2 Traduction d'un diagramme d'activité vers un système de cas d'utilisation contractualisés

Le principe de base de la traduction est de représenter la position des jetons par des prédicats. On associe à chaque *flot de contrôle* x (c'est-à-dire en fait à chaque arc orienté) un prédicat qui est vrai si et seulement si le jeton est sur le *flot de contrôle* x.

Cette manière de procéder justifie notre hypothèse selon laquelle il ne peut y avoir à tout moment qu'un seul jeton sur un *flot de contrôle*.

La simulation d'un diagramme d'activité se déroule comme suit. Initialement, on considère un jeton sur chaque nœud initial du diagramme d'activité. Puis les jetons progressent jusqu'à ce qu'une configuration bloquante soit atteinte. Dans notre cas, cette configuration sera l'attente d'une décision de l'utilisateur qui pourra choisir d'exécuter certains des cas d'utilisation du système. L'exécution d'un cas d'utilisation déclenche alors la progression de jetons, et ainsi de suite jusqu'à ce qu'il n'y ait plus de jetons dans le système, plus de jetons qui puissent progresser ou jusqu'à ce qu'un jeton arrive dans un nœud de fin d'activité.

Pour traduire un tel diagramme d'activité vers un système de cas d'utilisation contractualisés, il faut que chaque cas d'utilisation possède une pré-condition exprimant des contraintes requises sur la position des jetons pour l'exécutabilité du cas d'utilisation, et une post-condition exprimant des contraintes sur la position des jetons après exécution du cas d'utilisation. La table 8.6 résume la correspondance entre les concepts de simulation des diagrammes d'activité et des cas d'utilisation contractualisés.

Concept	Système de cas d'utilisation contractualisés	Diagramme d'activité
Initialisation	ensemble de prédicats vrais à l'initialisation	position initiale des jetons
État du système de simulation	ensemble de prédicats vrais	position des jetons
Exécution d'un cas d'utilisation (dynamique du système)	- si la pré-condition d'un cas d'utilisation est vérifiée dans le contexte de l'état du système de simulation, - alors le cas d'utilisation peut être exécuté - et l'état du simulateur progresse vers l'état résultant de l'application de la post-condition à l'état courant	- si les jetons sont positionnés de telle façon que le cas d'utilisation peut être exécuté, - alors le cas d'utilisation peut être exécuté - et les jetons progressent en fonction des flots et nœuds de contrôle positionnés en sortie du cas d'utilisation
Fin de progression	plus aucune pré-condition de cas d'utilisation n'est vraie dans le contexte de l'état courant du simulateur	les jetons ne peuvent plus bouger ou il n'y a plus de jetons, plus aucun cas d'utilisation ne peut être exécuté vis-à-vis de la position courante des jetons

TAB. 8.6 – Correspondance entre la simulation des diagrammes d'activité et des cas d'utilisation contractualisés

La traduction que nous proposons se compose de deux étapes : la combinaison de règles locales pour obtenir un premier système de contrats, et la complétion de ce premier système avec le traitement des acteurs et des nœuds de fin d'activité.

Règles élémentaires

Nous avons défini des règles locales, qui, pour chaque construction utilisée des diagrammes d'activité, précise la traduction locale sous forme de cas d'utilisation et de contrats. Ces règles sont des règles partielles, ne définissant que les cas où les nœuds de contrôle sont directement suivis et précédés par des cas d'utilisation. Ces règles doivent bien sûr être combinées entre elles pour donner les contrats finaux des cas d'utilisation. Nous détaillons tout d'abord ces règles locales avant de donner les principes de combinaison.

La table 8.7 fournit les règles locales : chaque ligne du tableau décrit une construction des diagrammes d'activité et sa traduction en système de cas d'utilisation contractualisé. On remarquera qu'il n'y a pas de règle locale pour le nœud de fin d'activité qui est traité de manière différente comme nous le verrons par la suite.

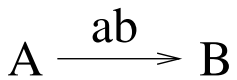
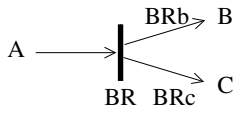
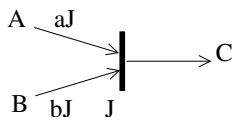
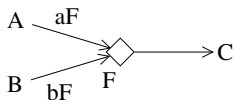
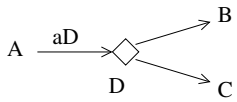
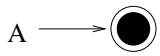
Combinaisons des règles locales

En combinant les règles locales lors de parcours du diagrammes d'activité, on peut construire un système partiel de cas d'utilisation contractualisés ; ce système partiel ne prend en compte que les contrôles entre cas d'utilisation, et pas les contraintes sur les acteurs qui seront rajoutées par la suite. Le traitement des nœuds de fin d'activité fait également l'objet d'un traitement ultérieur.

Pour chaque cas d'utilisation, on procède en deux phases :

1. On descend le flot d'exécution à partir du flot sortant, pour trouver où doivent être déposés le ou les jetons après exécution du cas d'utilisation. Cela nous permet de déterminer une partie de la post-condition prenant en compte les nouvelles positions des jetons, mais surtout de déterminer où sont suceptibles de se trouver les jetons, de manière à pouvoir calculer la pré-condition ultérieurement.
2. On remonte ensuite le flot de contrôle à partir du flot entrant, pour trouver où doivent se trouver les jetons pour l'exécution du cas d'utilisation. Cela nous permet de calculer la pré-condition (positionnement requis des jetons) et de compléter la post-condition avec la prise en compte de la suppression des jetons requis.

Les jetons peuvent s'arrêter à quatre endroits : avant un nœud de décision, avant un cas d'utilisation, avant un nœud de jonction, ou avant un nœud de fusion. Cela suppose que l'on descend les flots lors de la phase 1 jusqu'à trouver de telles positions potentielles de jetons. On calcule alors la post-condition partielle et on marque les transitions où seront déposés le ou les jetons. Lors de la phase 2, on remonte les flots de contrôle jusqu'à rencontrer des transitions marquées. Lors de la montée et de la descente, on applique les règles locales à chaque nœud de manière à calculer la pré-condition et à compléter la post-condition.

Diag. d'activité	Cas d'utilisation	Commentaires
 *Flot de contrôle*	**UC** A **pre** **post** ab **UC** B **pre** ab **post not** ab	Le *flot de contrôle* nommé ab entre 2 cas d'utilisation A et B traduit la dépendance séquentielle entre A et B. L'exécution de A dépose un jeton sur ab. L'exécution de B requiert un jeton sur ab, et le supprime.
 Branchement	**UC** A **pre** **post** BRb **and** BRc **UC** B **pre** BRb **post not** BRc **UC** C **pre** BRc **post not** BRc	Un branchement avec un arc entrant provenant du cas d'utilisation A et 2 arcs BRb et BRc sortant respectivement vers B et C traduit le fait qu'après A on peut exécuter concuremment B ou C. L'exécution de A dépose un jeton sur BRb et un jeton sur BRc. L'exécution de B (resp. de C) requiert un jeton sur BRb (resp. sur BRc) puis le supprime.
 Jonction	**UC** A **pre** **post** aJ **UC** B **pre** **post** bJ **UC** C **pre** aJ and bJ **post not** aJ **and not** bJ	Une jonction avec 2 arcs entrants aJ et bJ en provenance respective de A et B, et un arc sortant vers C traduit le fait que pour exécuter C, il faut au préalable que A et B se soient exécutés. L'exécution de A (resp. B) dépose un jeton sur aJ (resp. bJ). L'exécution de C requiert un jeton sur aJ et un jeton sur bJ, et les supprime.
 Fusion	**UC** A **pre** **post** aF **UC** B **pre** **post** bF **UC** C **pre** aF or bF **post** aF **implies not** aF **and** bF **implies** not bF	Une fusion avec 2 arcs entrants aF et bF en provenance respective de A et B, et un arc sortant vers C traduit le fait que pour exécuter C, il faut au préalable que A ou B se soient exécutés. L'exécution de A (resp. B) dépose un jeton sur aF (resp. bF). L'exécution de C requiert un jeton sur aF ou un jeton sur bF, et supprime le jeton présent.
 Décision	**UC** A **pre** **post** aD **UC** B **pre** aD **post not** aD **UC** C **pre** aD **post not** aD	Une décision avec un arc aD entrant provenant du cas d'utilisation A et 2 arcs sortant vers B et C traduit le fait qu'après A on peut exécuter B ou C. L'exécution de A dépose donc un jeton sur aD. L'exécution de B (resp. de C) requiert un jeton sur aD puis le supprime.
 Nœud initial	**UC** A **pre** init **post** not init	On introduit un prédicat particulier *init*.
 Nœud de fin de flot	**UC** A **pre** **post**	

TAB. 8.7 – Règles locales de traduction, avec les cas d'utilisation A, B et C

Ce principe aboutit à l'algorithme 5 que nous donnons en annexe page 190.

Traitement des acteurs et des nœuds de fin d'activité

Une fois que le système partiel de cas d'utilisation contractualisés est construit comme décrit ci-dessus, ce système est complété avec des informations sur les acteurs, et des nœuds de fin d'activité.

Les nœuds de fin d'activité font l'objet d'un traitement à part, dans la mesure où l'arrivée d'un jeton sur un tel nœud a des conséquences non pas uniquement localement, mais sur tout le reste de l'exécution de l'activité. Ainsi, nous avons introduit dans notre système de cas d'utilisation contractualisés un prédicat supplémentaire *end*, qui est mis à vrai quand un jeton arrive à un nœud de fin d'activité (cela est fait dans la phase 1 de la combinaison des règles locales), et qui doit être faux pour permettre l'exécution de chaque cas d'utilisation. Ainsi, on complète le système partiel en faisant la conjonction de chaque pré-condition de cas d'utilisation avec $\neg end$. De cette manière, le système ne peut plus progresser.

Concernant le traitement des acteurs, on rajoute à chaque cas d'utilisation un paramètre p correspondant à son acteur principal. On définit un prédicat d'arité 1 par acteur, prenant en paramètre un type générique correspondant à tous les acteurs. Ainsi, on peut pour chaque instance d'acteur, préciser quel rôle il a dans le système. Puis, pour cas d'utilisation du diagramme d'activité, on détermine de quelle partition A il fait partie (on rappelle que le nom des partitions correspond au nom des acteurs), et on rajoute $A(p)$ à la pré-condition. Quand il y a des cas d'utilisation dupliqués dans plusieurs partitions (c'est-à-dire partagés par plusieurs acteurs), alors le cas d'utilisation final doit être reconstruit. Supposons qu'un cas d'utilisation *uc* ait été scindé en n cas d'utilisation *uc_ actor_ 1*, ..., *uc_ actor_ n*, alors dans le système de contrats final, le cas d'utilisation *uc* sera reconstruit comme suit :

```
UC uc(p:participant)
pre (actor_1(p) and pre(uc_actor_1)) or
     ... or (actor_n(p) and pre(uc_actor_n))
post (actor_1(p) implies post(uc_actor_1)) and
     ... and (actor_n(p) implies post(uc_actor_n))
```

et ainsi on construit un système de cas d'utilisation contractualisés à partir d'un diagramme d'activité.

8.2.2 Comparaison des deux approches

Dans cette section, nous comparons les deux approches : nous proposons une comparaison qualitative et une comparaison quantitative en termes d'efficacité et de pertinence des tests générés avec l'une et l'autre des méthodes.

8.2.2.1 Comparaison qualitative

Pour effectuer notre comparaison, nous avons tout d'abord exprimé les dépendances entre les cas d'utilisation de la réunion virtuelle avec un diagramme d'activité tel que décrit dans la section 8.2. Dans cette section, nous comparons les approches en termes d'expressivité et de facilité d'utilisation, ce qui peut bien sûr paraitre subjectif.

Tout d'abord, on peut remarquer que l'avantage majeur des diagrammes d'activité est l'aspect visuel : d'un seul coup d'œil, les dépendances peuvent être visualisées sur un diagramme d'activité, alors qu'elles doivent être calculées avec un mécanisme de contrats tel que celui que nous proposons.

Cependant, les diagrammes d'activité payent cette qualité par la difficulté d'élaboration : les dépendances doivent être pensées glabalement, et non pas localement comme avec les contrats. Concrètement, nous avons eu de grandes difficultés à construire un diagramme d'activité pour le système de réunions virtuelles qui reflète exactement les mêmes dépendances que celles exprimées facilement avec le système de contrats. Par ailleurs, le diagramme d'activité est si complexe qu'il en devient illisible, à cause du grand nombre de nœuds de contrôle qui doivent être introduits pour gérer les boucles et les interactions entre acteurs.

Du fait de la difficulté de construction d'un diagramme d'activités complet, nous avons étudié deux diagrammes d'activités pour le système de réunions virtuelles : le premier, AD1, est simple, et ne comprend pas de boucles, c'est le diagramme que nous avons donné Figure 8.3. Le second, AD2, donné Figure 8.4, a été construit en complétant AD1 pour exprimer tous les comportements du système, notamment les boucles. Néanmoins, AD2 n'est pas complet, dans la mesure où certaines interactions entre acteurs sont impossibles à exprimer avec un tel formalisme. Dans notre exemple, il n'est pas possible de traduire le fait que quand un animateur ferme une réunion, alors tous les participants en sont éjectés. En effet, un jeton ne peut pas « forcer » un autre jeton à bouger vers une autre activité, indépendamment d'un nœud de contrôle.

Nous pensons de plus que la sémantique exacte des diagrammes d'activité est assez mal connue, en particulier celle des diagrammes d'activité d'UML 2.0, très proche de celle des réseaux de Pétri, mais avec des constructions qui les rendent beaucoup plus complexes. L'usage de certaines constructions très contre-intuitives nous paraît même dangereux, comme les deux règles implicites : branchement (*fork*) implicite quand deux flots sortent d'une activité, et jonction (synchronisation) implicite quand 2 flots entrent dans une activité.

Pour conclure, les diagrammes d'activité nous paraissent bien adaptés pour des systèmes très simples (ou pour documenter des comportements partiels), mais pour des systèmes complexes et réalistes, l'approche par contrats nous paraît plus adéquate en ce qui concerne à la fois la complexité et l'expressivité.

8.2.2.2 Comparaison de l'efficacité des tests générés

Pour comparer l'efficacité des tests générés avec ces deux approches, nous avons tout d'abord traduit chaque diagramme d'activité (AD1 et AD2) en un système de

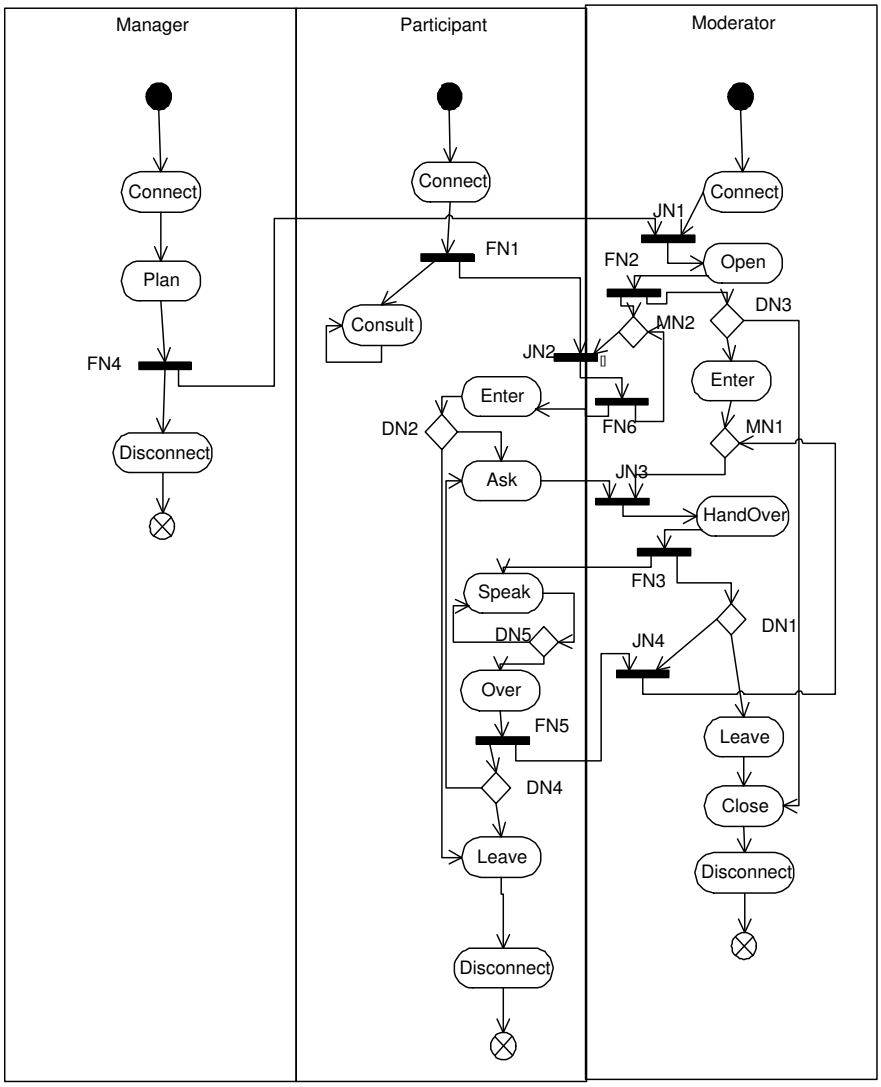


FIG. 8.4 – Diagramme d'activité AD2 pour le système de réunions virtuelles

	AD1	AD2
# d'objectifs de test générés par le critère *toutes les transitions*	346	436
Inclusion des chemins générés dans l'UCTS original	278/346 (80%)	350/436 (80%)

TAB. 8.8 – Inclusion des tests gérénés à partir des diagrammes d'activité dans l'UCTS du système de cas d'utilisation contractualisés initial

cas d'utilisation contractualisés, en adoptant la méthode décrite dans la section 8.2. Nommons SCUI1 et SCUI2 ces deux systèmes. Nous avons ensuite généré les objectifs de test pour SCUI1 et SCUI2.

Nous avons alors pu étudier la relation d'inclusion existant entre les ensembles de tests produits par les deux approches. Comme on le voit dans la table 8.8, 80% des tests générés à partir de SCUI1 (respectivement SCUI2) sont inclus dans l'UCTS construit à partir du système de cas d'utilisation contractualisés initial. L'étude des 20 % de tests restant a montré que ces tests n'étaient pas corrects : ils acceptaient des comportements incorrects du système. Cela signifie que la génération de tests à partir d'un diagramme d'activité peut aboutir à des séquences de cas d'utilisation qui sont supposées être valides, alors qu'elles contredisent la spécification. L'incorrection de ces séquences de test provient du fait que, comme nous l'avons dit dans notre comparaison qualitative, certaines interactions ne sont pas exprimables avec un diagramme d'activité. Il est à noter que dans la table 8.8, nous avons utilisé le critère *toutes les transitions*. Nous voulions utiliser le critère le plus neutre vis-à-vis du modèle de représentation des dépendances : couvrir toutes les transitions revient à couvrir chaque cas d'utilisation dans chaque contexte différent.

Exprimer les dépendances entre cas d'utilisation nous a donc mené à générer des tests acceptant (et exigeant) des comportements incorrects. C'est pourquoi nous pensons que les diagrammes d'activité ne sont pas adaptés pour la génération de test.

Pour estimer la qualité des cas de test générés avec des diagrammes d'activité pour exprimer les dépendances entre cas d'utilisation, nous les avons comparés en terme de couverture de code avec les cas de test obtenus avec le système de contrats original. Pour obtenir une comparaison significative, nous avons supprimé les 20% de cas de test incorrects. Les résultats de cette comparaison sont donnés à la figure 8.5. À cause de la faible expressivité des diagrammes d'activité, les cas de test générés aboutissent à une faible couverture de code. Un point intéressant est que le fait de compléter les diagrammes d'activité n'a pas amélioré de manière significative l'efficacité des tests générés.

Pour conclure, l'expression des dépendances entre des contrats est plus adaptée que l'utilisation de diagrammes d'activité dans la mesure où :

- les contrats sont plus expressifs que les diagrammes d'activité et donc aboutissent à une meilleure couverture de code par les tests générés,
- pour le même niveau de spécification, l'effort relatif pour écrire des contrats est

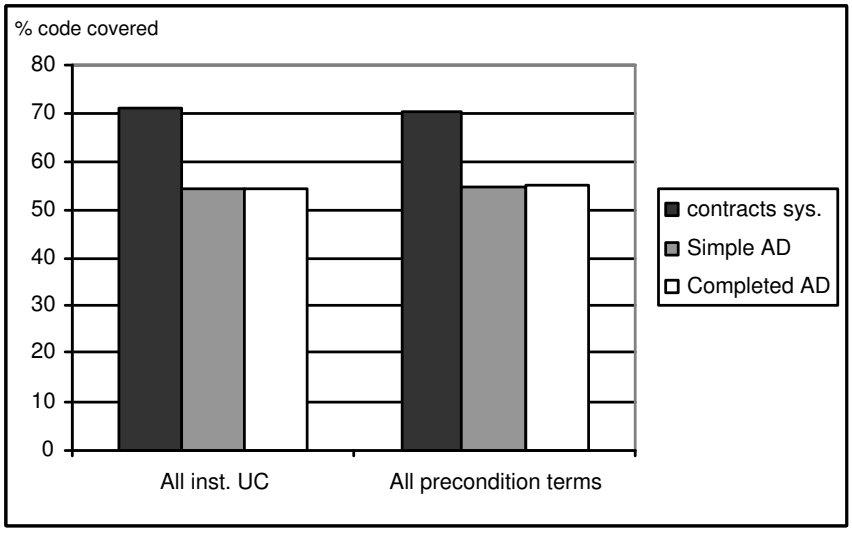


FIG. 8.5 – Comparaison de la couverture de code des tests générés à partir de diagrammes d'activité et de cas d'utilisation contractualisés

moins important que pour dessiner un diagramme d'activité,
- les diagrammes d'activité entraînent la génération d'un nombre significatif de tests invalides.

8.3 Expériences avec Thalès Airborne System

Cette section résume nos expériences dans le cadre de la collaboration avec TAS. Cette collaboration a été très enrichissante : elle nous a permis de confronter notre approche qui se veut tournée vers les préoccupations industrielles avec de réels cas d'études industriels.

Ces expériences faites avec les cas d'études de TAS sont bien sûr d'une nature très différente des études de cas académiques. La taille des composants traités (entre 5000 et 15000 lignes de code), et le fait qu'ils soient en C++ alors que nos outils de trace sont faits pour du code Java, font que nous n'avons pas pu mener les mêmes études de couverture de code. Les études présentées dans cette section seront donc plus qualitatives que quantitatives ; elles nous semblent pourtant primordiales, puisqu'elles fournissent une validation industrielle à notre approche.

Les composants étudiés sont des systèmes de navigation d'armement du Mirage 2000-9 et du Rafale. Nous avons mené des expériences sur le composant du Mirage 2000-9, tandis que TAS expérimentait nos prototypes sur le composant du Rafale.

Exigences traduites	70%
Exigences non traduites mais traduisibles après amélioration de l'outil	9%
Exigences non traduites dues aux limitations intrinsèques de l'approche	21%

TAB. 8.9 – Statistiques sur la traduction des exigences en LDE

8.3.1 Protocole expérimental

Les exigences du système de navigation d'armement du Mirage 2000-9 sont écrites dans des documents textuels, structurés en grandes sections appelées *catégories*. Chaque exigence se compose d'une ou plusieurs phrases en langue naturelle (anglais pour le mirage et français pour le rafale) et d'informations de traçabilité vers le modèle de conception (concrètement, ces informations correspondent aux noms des méthodes réalisant l'exigence).

Nous avons étudié cet ensemble d'exigences informelles, et l'avons traduit en LDE. Cette traduction est abordée dans la section 8.3.2. Nous avons ensuite utilisé nos outils prototypes (analyseur de LDE, générateur de cas d'utilisation, simulateur et générateur de tests). La section 8.3.3 présente les premières conclusions qui peuvent être tirées à l'issue de la phase de traduction et de simulation des exigences. La section 8.3.4 aborde la génération de tests.

8.3.2 Traduction des exigences en LDE

Nous avons traité 34 exigences du composant fourni, correspondant à cinq catégories sur huit.

La table 8.9 donne des statistiques sur la traduction des exigences en LDE. 70 % des exigences ont pu être traduites en LDE. La traduction de ces 70% d'exigences a été assez facile et naturelle, ce qui conforte l'idée que le LDE est facile d'utilisation. Certaines exigences n'ont pas pu être traduites en LDE. Cette impossibilité de traduction provient à 9 % de lacunes de nos outils prototypes, qui vont être comblées rapidement. Enfin, 21% des exigences n'ont pas pu être traduites du fait des limitations intrinsèques à notre approche. Ces exigences non traduites concernent essentiellement les aspects numériques, que nous ne traitons pas actuellement. Il est à noter que si nous souhaitons à terme intégrer à notre approche certains aspects numériques, toutes les exigences non traduites n'ont pas le bon niveau d'abstraction : en fait nous ne souhaitons pas être capable de toutes les traduire. Notre approche bénéficierait grandement de l'intégration des aspects temps-réels et de qualité de service, qu'il est nécessaire de pouvoir exprimer à haut niveau d'exigence.

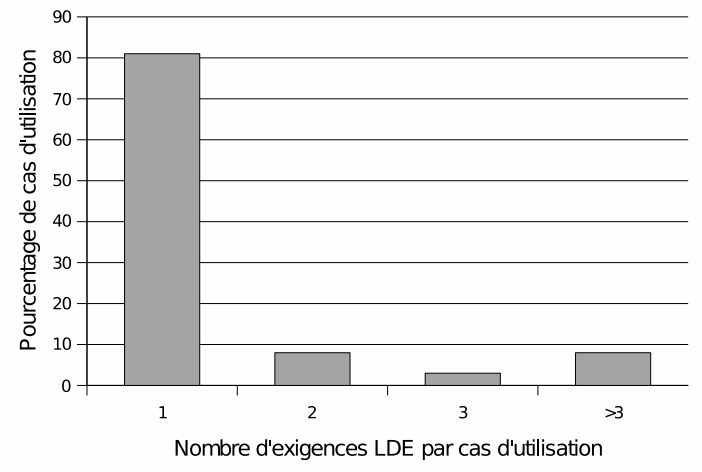


FIG. 8.6 – Distribution des cas d'utilisation en fonction du nombre d'exigences qu'ils regroupent

8.3.3 Bilan de la traduction et de la simulation des exigences traduites

L'interprétation des exigences a nécessité 16 patrons d'interprétation. Parmi ces 16 patrons d'interprétation, 8 sont des patrons communs qui font partie du cœur de l'interprétation, les 8 autres ont été ajoutés pour adhérer à la pratique de rédaction des exigences de TAS.

Nous nous sommes intéressé au nombre d'exigences nécessaires pour décrire un cas d'utilisation. Nous avons pour cela introduit un index de dispersion :

$$\text{index de dispersion} = \frac{\text{nombre d'exigences}}{\text{nombre de cas d'utilisation agrégés}}$$

Ce taux de dispersion est de 1.6 pour les exemples traités. Nous avons également étudié la distribution du pourcentage de cas d'utilisation pour lesquels un, deux, trois ou plus de trois exigences sont nécessaires pour décrire un cas d'utilisation. Cette distribution est donnée Figure 8.6. La majorité des cas d'utilisation sont décrits par une seule exigence, et 20% des cas d'utilisation sont décrits par plusieurs exigences (jusqu'à 7).

Cette distribution est un peu décevante, mais pas étonnante. En effet, nous sommes partis des exigences finales des composants de TAS. Ces exigences sont donc très détaillées, et décrivent des fonctionnalités très précises, et pas des groupes de fonctionnalités. Ainsi, les cas d'utilisation déduits sont les cas d'utilisation opérationnels du système, et pas les cas d'utilisation fondamentaux.

Après utilisation de nos différents outils prototypes (analyseur et simulateur), nous pouvons dresser les conclusions suivantes :

– L'utilisation du simulateur a montré que la définition originale du système était

nombre d'états de l'UCTS		68
nombre de transitions de l'UCTS		213
Critère	nb d' OT générés	Taille moyenne des OT
toutes les transitions	146	9
tous les nœuds	25	8
tous les cas d'utilisation instanciés	10	4
tous les nœuds et tous les c.u.i.	35	7
tous les termes des pré-conditions	11	3
robustesse	36	4

TAB. 8.10 – Statistiques sur les objectifs de test (OT) générés pour les exigences du Mirage 2000-9 traitées

sous-spécifiée. Cela a été observé du fait qu'une transition (c'est-à-dire une possible activation de service) qui était attendue n'était pas proposée par le simulateur. Après étude des exigences, il s'est avéré que l'une d'entre elles n'avaient pas été spécifiée, elle a ensuite été définie et ajoutée au système. Le simulateur a donc permis de nous assurer de la complétude des exigences.

– L'interpréteur des exigences a également permis de détecter des exigences ambiguës. Ces ambiguïtés étaient liées à une succession non parenthésée de structures temporelles.

8.3.4 Bilan sur la génération de tests

Nous avons généré les objectifs de test correspondant aux systèmes de cas d'utilisation issus de l'interprétation des exigences LDE. Les statistiques sur cette génération de test sont données table 8.10, ainsi que des statistiques sur l'UCTS correspondant. Nous n'avons généré que les objectifs de test et pas les cas de test, du fait que nous ne disposions pas de scénarios attachés aux cas d'utilisation.

Un élément intéressant est la taille de l'UCTS généré. Cette petite taille est due au fait que les cas d'utilisation du système étudié sont assez fortement séquentiels, et que d'autre part il y a peu d'instances nécessaires pour chaque entité métier manipulée (par exemple, il y a un seul pilote dans un avion de combat).

Un autre point à noter est le petit nombre de cas d'utilisation générés : pour un système de complexité moyenne, peu d'objectifs de test sont générés, ce qui était un but recherché par notre approche ; un grand nombre d'objectifs de test sont, de notre point de vue, inexploitables.

8.3.5 Retour d'expérience de TAS

Nous avons fourni à TAS nos outils prototypes (analyseur et interpréteur de LDE, simulateur et générateur de tests). Ces prototypes ont été utilisés sur les spécifications de certains composants du Mirage. Il est à noter que les prototypes fournis étaient

les tous premiers développés concernant le traitement du LDE, ce qui explique que la plupart des remarques de TAS concernent des problèmes techniques des prototypes.

Au delà des erreurs détectées dans les prototypes, les remarques de TAS concernent tout d'abord les limitations dues à l'expressivité du LDE. Certaines de ces remarques ont donné lieu à la définition de nouvelles structures syntaxiques, ainsi qu'à la définition de patrons d'interprétation permettant leur analyse sémantique.

Par ailleurs, TAS souligne l'intérêt du simulateur pour vérifier les exigences, et le fait que l'utilisation du LDE montre bien qu'il faut être strict sur les observables et leurs valeurs. Cela permet de supprimer un défaut actuel des spécifications : les changements d'appellation entre différentes exigences.

De nouvelles expérimentations sont en cours avec les nouveaux prototypes.

8.4 Conclusions

Dans ce chapitre, nous avons proposé une validation académique et une validation industrielle de l'approche de test présentée dans ce manuscrit.

La validation académique que nous avons présentée se base sur trois études de cas. Nous avons étudié pour l'une d'entre elles les différents critères de test que nous avons proposés Chapitre 6. Cela nous a permis de déterminer quels critères parmi ceux que nous avions proposés étaient les plus pertinents. Puis nous avons étudié l'efficacité de notre approche sur trois études de cas. L'efficacité de l'approche a été évaluée par une mesure de couverture de code. Nos études de couverture ont montré que sur nos trois études de cas, les tests générés par notre approche couvrent environ 80% du code, ce qui est prometteur dans la mesure où ces tests sont générés automatiquement à partir des exigences. Il serait intéressant d'utiliser plutôt une analyse de mutation pour évaluer la qualité des tests générés. Effectuer une analyse de mutation nécessite un outillage pour générer les mutants, outillage que nous n'avons pas pu mettre en œuvre au cours de ce doctorat. Nous avons généré quelques mutants manuellement pour vérifier que nos tests pouvaient les détecter, mais ce type d'expériences nous a semblé trop biaisé pour être présenté ici, du fait de la subjectivité liée à l'insertion d'erreurs et du faible nombre de mutants générés.

La validation industrielle est une analyse plus qualitative des expériences que nous avons menées sur des composants réels provenant de TAS. Elle démontre que notre approche est intéressante sur des composants de complexité moyenne. L'application de notre approche de désambiguïsation des exigences a permis de révéler des lacunes dans les spécifications dont nous disposions. Au cours de ces expériences, l'introduction d'arithmétique dans notre modèle s'est révélée primordiale pour exprimer des contraintes de qualité de service et de temps réel.

Chapitre 9

Application aux lignes de produits logicielles

Nous avons présenté dans les chapitres 6 et 7 une approche permettant de dériver des cas de test système à partir des exigences d'un logiciel, exprimées sous forme de cas d'utilisation et de scénarios. Comme nous l'avons souligné, cette approche a été fortement guidée par le contexte des lignes de produits logicielles, qui introduit de nouvelles contraintes dans la génération de test. Dans ce chapitre, nous présentons la spécialisation de notre approche pour les lignes de produits logicielles, qui permet de gérer explicitement les points de variation dans les exigences. Nous donnons tout d'abord nos objectifs et notre méthode pour le test de lignes de produits, puis nous abordons la génération d'objectifs de test puis de cas de test pour les lignes de produits.

9.1 Test des lignes de produits : objectifs et méthode

Tester une ligne de produits de façon classique est extrêmement coûteux : puisqu'il faut tester indépendamment chacun des produits de la ligne, le coût de test total pour la ligne de produits est proportionnel en son nombre de produits. Une approche prenant en compte le fait que les produits partagent un certain nombre d'exigences permet de factoriser considérablement la phase de test. C'est une telle approche de génération de test système à partir des exigences que nous présentons ici.

Les exigences d'une ligne de produits logicielle se caractérisent par la présence de parties variantes. Un produit particulier peut être vu comme étant spécifié par une liste d'exigences particulières parmi les exigences de la ligne de produits. On appelle *modèle de décision* les informations permettant de caractériser chaque produit par ses exigences spécifiques.

Un des buts du développement d'une ligne de produits logicielle est de permettre la mise sur le marché rapide de nouveaux produits d'une même ligne. Dans cette optique, tant la dérivation d'un nouveau produit que sa validation doit être rapide. Concernant la dérivation de nouveaux produits, plusieurs approches se basent sur la transformation automatique de modèles à partir d'un modèle de décision. Nous pensons que dans le

cadre de la validation de produit, la phase de test système est cruciale, puisqu'elle tend à vérifier qu'un produit est bien conforme à ses exigences. Nous avons également cherché à l'automatiser de manière à réduire les coûts de test.

Les objectifs que nous avons cherché à atteindre sont donc :

- la modélisation de la variation au niveau des exigences,
- la facilité de la maintenance des exigences,
- la génération automatique de tests spécifiques à chaque produit.

L'approche de génération de test que nous avons pour cela mise en œuvre est schématisée Figure 9.1.

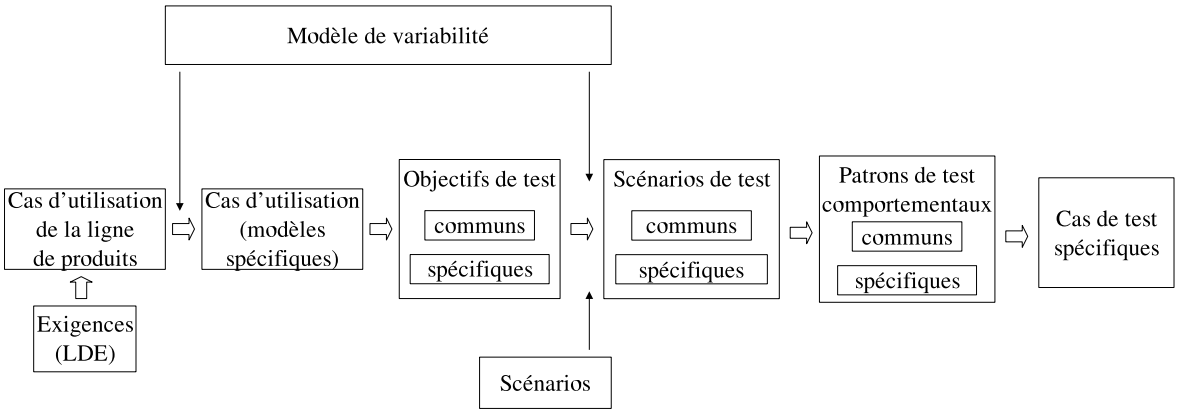


FIG. 9.1 – Génération de test pour les lignes de produits

Les exigences de la ligne de produits sont écrites avec une extension du langage LDE acceptant la spécification de parties variantes. Un modèle de cas d'utilisation est généré à partir de ces exigences. Ce modèle est commun à la ligne de produits, et reflète donc lui aussi la spécification de parties variantes ou dépendantes de variants particuliers. À partir d'un tel modèle de cas d'utilisation et d'un modèle de décision, un modèle de cas d'utilisation est généré pour chaque produit. Chaque modèle spécifique est ensuite utilisé pour dériver des objectifs de test. Les objectifs de test communs à tous les produits sont alors détectés. Ensuite, intervient la phase de génération de cas de test via la génération de scénarios de test puis de patrons de test comportementaux.

Dans la suite de ce chapitre, nous détaillons la génération d'objectifs de test pour les lignes de produits, puis l'obtention de cas de test à partir de ces objectifs. Nous commençons toutefois par introduire la version « ligne de produits » du système de réunions virtuelles de manière à pouvoir illuster ce chapitre.

9.2 Ligne de produits « réunion virtuelle »

Le système de réunions virtuelles présenté Section 4.3 a été étendu de manière à fabriquer une ligne de produits.

Nous avons rajouté deux fonctionnalités à ce système :

- celle de pouvoir superviser l'ensemble des réunions. Le superviseur du système est le seul à pouvoir en quelque sorte « espionner » le contenu de toutes les réunions du système, et consulter la liste de tous les participants connectés.
- celle de pouvoir obtenir les minutes des réunions. S'il le souhaite, l'animateur peut décider de minuter une réunion avant qu'elle ne soit ouverte. Les minutes lui seront alors envoyées à la fermeture de la réunion.

Ainsi, deux cas d'utilisation (*Spy* et *Record*) et un acteur (*supervisor*) ont été ajoutés au modèle initial. Le modèle final de cas d'utilisation pour la ligne de produits avec les contrats correspondants est donné en annexe page 175.

Nous définissons ici notre ligne de produits en introduisant ses points de variation, puis en décrivant les produits la composant.

9.2.1 Points de variation

Nous présentons ici 5 points de variation :

- la limitation ou pas du nombre de participants dans une réunion. Quand une réunion est limitée, seuls trois participants sont autorisés à y être à la fois ;
- le type de réunions qu'il est possible de planifier. Les instanciations possibles de ce point de variation correspondent à une sélection de un, deux ou trois types de réunions parmi les trois existants : standard, privée, démocratique, ce qui fait un nombre total de sept combinaisons ;
- la présence ou l'absence de la possibilité de minuter les réunions ;
- les langues supportées par le serveur (les langues possibles étant le français, l'anglais, l'espagnol et l'allemand, soit un total de quinze combinaisons) ;
- la présence ou l'absence d'un superviseur.

Ces cinq points de variation sont ici suffisants pour illustrer notre propos. Supposons toutefois que nous rajoutiions la présence ou l'absence d'un traducteur, les systèmes d'exploitation supportés, et les différentes interfaces proposées (en considérant trois systèmes d'exploitation, et deux interfaces, l'une graphique et l'autre en ligne de commande). On voit bien alors que tester un à un chacun des produits indépendamment les uns des autres est une tâche colossale, puisque cela supposerait dans notre cas le test de $(2 \times 7 \times 2 \times 15 \times 2 \times 3 \times 3) = 7560$ produits potentiels.

9.2.2 Produits

Nous considérons ici trois produits, qui pourraient être issus de considérations commerciales : l'édition de démonstration, l'édition personnelle, et l'édition entreprise. Bien d'autres produits pourraient être dérivés de la ligne, mais ces trois éditions sont suffisantes dans un but illustratif.

L'édition de démonstration. Elle ne gère que les réunions standards, et toutes les réunions y sont limitées. Il n'y a pas de superviseur, les réunions ne sont pas minutables. Seule la langue anglaise est supportée.

L'édition personnelle. Elle gère les trois types de réunions. Toutes les réunions sont limitées, et ne peuvent pas être minutées. Il n'y a pas de superviseur. Seul l'anglais est supporté.

L'édition entreprise. Elle gère les trois types de réunions. Les réunions ne sont pas limitées, et sont minutables. Les quatre langues sont supportées, et un superviseur est présent.

9.2.3 Éléments de modélisation

La modélisation que nous avons utilisée permet d'extraire facilement des produits d'une architecture globale. Nous nous sommes pour cela appuyé sur des patrons de conception [Gamma et al., 1995]. Nous avons géré les variations au niveau des réunions avec une hiérarchie d'héritage et un décorateur. Les produits sont extraits grâce à des fabriques abstraites [Jézéquel, 1999]. Un extrait du diagramme de classe UML est donné Figure 9.2.

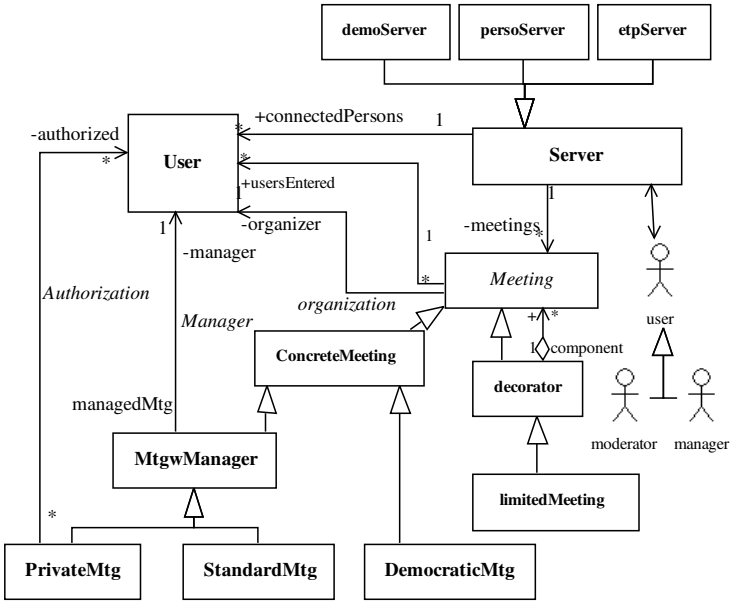


FIG. 9.2 – Extrait du diagramme de classes de la ligne de produits « réunion virtuelle »

9.3 Génération d'objectifs de test pour les lignes de produits

Dans cette section, nous détaillons la génération d'objectifs de test. Nous commençons par expliquer comment représenter la variabilité des exigences et le type de modèle

de décision que nous utilisons. Nous expliquons ensuite comment les objectifs de test sont générés.

9.3.1 Expression de la variabilité au niveau des exigences

Pour exprimer la variabilité au niveau des exigences, nous avons permis aux cas d'utilisation de refléter des parties variantes. Nous avons ainsi étendu le modèle des cas d'utilisation utilisé. Rappelons que jusqu'alors, un cas d'utilisation était constitué de paramètres, d'une précondition et d'une postcondition, ainsi que de scénarios nominaux et exceptionnels. Les scénarios intervenant dans un second temps, nous nous focalisons tout d'abord sur les cas d'utilisation, leurs paramètres et leurs contrats.

L'idée est ici de définir quelles sont les parties des exigences qui ne sont pas communes à l'ensemble de la ligne de produits, mais qui dépendent d'une certaine instanciation d'un point de variation. Il faut alors définir quel est le niveau de granularité de la définition de la variabilité, c'est-à-dire quels éléments du cas d'utilisation sont susceptibles de dépendre d'un point de variation particulier.

Une extension du LDE pour prendre en compte la variabilité au niveau des exigences textuelles est en cours d'élaboration.

9.3.1.1 Variabilité au niveau des cas d'utilisation

Le niveau de granularité le plus grossier est le cas d'utilisation lui-même. En effet, un cas d'utilisation peut ne pas être commun à l'ensemble des produits d'une même ligne, dans le cas où les produits diffèrent par l'absence ou la présence d'une fonctionnalité. C'est le cas par exemple du cas d'utilisation *Record*, qui permet de minuter les réunions et d'en obtenir les actes. Ce cas d'utilisation n'est présent que dans certains produits, dans notre cas il n'est présent que dans l'édition entreprise.

9.3.1.2 Variabilité au niveau des paramètres

La variation peut également intervenir au niveau des paramètres d'un cas d'utilisation. Ainsi certains acteurs ou certains concepts peuvent n'exister que dans certains produits, et donc les cas d'utilisation ne sont pas nécessairement paramétrés de la même façon d'un produit à un autre. Dans l'exemple de la réunion virtuelle, ce cas se produit pour le cas d'utilisation *Open*, qui permet d'ouvrir une réunion. Dans les réunions possédant un animateur, c'est l'animateur qui ouvre la réunion, tandis que les réunions démocratiques s'ouvrent spontanément. Ainsi, le cas d'utilisation *Open* peut être paramétré ou pas par son modérateur, dépendant des produits.

9.3.1.3 Variabilité au niveau des contrats.

Les contrats peuvent également varier, selon les caractéristiques du produit. Si l'on prend par exemple le cas d'utilisation *Enter* du système de réunions virtuelles, dans les cas où le produit ne supporte que les réunions limitées, la précondition va s'assurer que

la réunion n'est pas déjà pleine. En revanche, tous les produits auront une précondition commune pour le cas d'utilisation *Enter*, précisant par exemple que le participant entrant est connecté au système. Symétriquement, les postconditions des cas d'utilisation possèdent d'un produit à l'autre des parties communes et des parties variantes.

9.3.1.4 Exprimer la variabilité

La variabilité doit ainsi être définie à trois niveaux : le cas d'utilisation, ses paramètres, et ses contrats. Nous ne pensons pas que la variabilité doive être exprimée en faisant référence explicite aux différents produits de la ligne. D'une part parce qu'au stade des spécifications des exigences, le nombre de produits de la ligne, ainsi que leurs caractéristiques exactes peuvent ne pas être connus : une analyse commerciale peut être menée parallèlement pour déterminer quels produits précis vont faire partie de la ligne. D'autre part, à supposer que les caractéristiques des produits soient connues, si les exigences sont directement liées aux produits, la dérivation d'un nouveau produit demandera alors un travail de modélisation supplémentaire. C'est pour cela que nous exprimons la variabilité non pas en termes de produits mais en termes d'instanciation de points de variations. Chaque partie du cas d'utilisation susceptible de varier est définie pour un ensemble d'instanciations de point de variations donné, et donc pour tous les produits possédant la ou les instanciations de points de variations spécifiées.

9.3.1.5 Définition d'étiquettes et de valeurs étiquetées UML

Pour pouvoir spécifier la variabilité, nous introduisons une construction qui permet d'exprimer de manière homogène la variabilité dans les cas d'utilisation en s'appuyant sur les points de variation et leurs instanciations possibles. On va donc rajouter aux différents éléments du modèle concernés des étiquettes précisant la dépendance des éléments vis-à-vis des points de variations. En UML, ces étiquettes sont modélisées par des *tagged values*. Le format de ces étiquettes est le suivant : *VP{variant_list}*, où *VP* est le nom d'un point de variation, et *variant_list* est une liste de variants, i.e. d'instanciations du point de variation *VP*.

Un élément sans étiquette est par défaut valable pour tous les produits. Un élément avec une étiquette *VP{variant_list}* est valide uniquement pour les produits sélectionnés par l'étiquette, c'est-à-dire pour les produits possédant le ou l'un des variants défini(s) par *variant_list*.

Comme nous l'avons expliqué précédemment, les contrats peuvent avoir des parties communes à tous les produits, et d'autres parties variantes. Cela suppose donc que pour un même cas d'utilisation, on puisse définir plusieurs préconditions et plusieurs postconditions, avec différentes étiquettes permettant de déterminer pour quels produits elles sont valables. Quand, pour un produit, plusieurs préconditions sont valables, alors la précondition du cas d'utilisation pour ce produit est la conjonction des préconditions valables. Il en va de même pour les postconditions.

9.3.1.6 Modèle de décision

Pour déterminer quels sont les produits sélectionnés par les différentes étiquettes, on se base sur un modèle de décision qui permet de déterminer quelles sont les caractéristiques de chaque produit. Ce modèle de décision peut être vu comme une table à deux entrées : d'une part les produits et d'autre part les points de variation, et associant pour chaque couple (produit, point de variation) les instanciations du point de variation présentes dans le produit.

On se base comme c'est souvent le cas dans la littérature [Bachmann and Bass, 2001, Barry Keepence, 1999] sur trois types de points de variation :

- l'optionalité : présence ou absence d'une caractéristique repésentée par un point de variation. Les variants ou instanciations d'un point de variation optionnel sont des valeurs booléennes : vrai symbolisant la présence de la caractéristique, et faux son absence ;
- le choix : sélection d'une unique caractéristique parmi une liste de choix possibles (ou variants) définis par le point de variation. L'instanciation d'un point de variation de choix est une liste comprenant un seul variant ;
- le choix multiple : sélection d'un ensemble de caractéristiques parmi une liste de choix possibles (ou variants) définis par le point de variation. L'instanciation d'un point de variation de choix multiple est une liste de tous les variants sélectionnés (i.e. une sous-liste de tous les variants possibles associés au point de variation).

Le modèle de décision permet d'associer chaque produit avec ses caractéristiques, en donnant pour chaque point de variation les variants présents dans le produit : une valeur booléenne pour les points de variation optionnels, un variant pour les choix, et une liste de variants pour les choix multiples.

9.3.1.7 Exemple

Dans l'exemple de la réunion virtuelle, nous avons défini cinq points de variation et trois produits, dont nous avons donné les caractéristiques. Le modèle de décision que nous avons implicitement défini peut être explicité dans une table telle que la Table 9.1, dont chaque colonne représente un produit, et chaque ligne un point de variation. Chaque cellule représente donc pour un produit P et un point de variation VP donnés les variants de VP définis dans P.

Considérons maintenant le cas d'utilisation *Enter*. Pour tous les produits, sa précondition est que la réunion soit ouverte et que le participant souhaitant entrer soit connecté au système. Dans le cas de produits permettant la planification de réunions privées, il faut également s'assurer que si la réunion est privée, le participant souhaitant entrer y est autorisé. Dans le cas de produits où le nombre de participants à une réunion est limité, il faut s'assurer que la réunion n'est pas déjà saturée. Cela se traduit par trois préconditions étiquetées pour le cas d'utilisation *Enter*, ainsi que le montre la figure 9.3.

Sur cet exemple, on voit bien que définir un nouveau produit comme une nouvelle

Édition	Démonstration	Personnelle	Entreprise
Limitation des réunions	vrai	vrai	faux
Types de réunions	{Standard}	{Standard, démocratique, privée}	{Standard, démocratique, privée}
Langue	{anglais}	{anglais}	{anglais, français, allemand espagnol}
Minutage des réunions	faux	faux	vrai
Supervision	faux	faux	vrai

Table 9.1: Modèle de décision pour le système de réunions virtuelles

```
UC enter(u:participant; m:mtg)
pre connected(u) and opened(m)
pre priv(m) implies authorized(u,m) {VPMeetingType(priv)}
pre not exists (u,v,w:participant)
  {entered(u,m) and entered(v,m) and entered(w,m)
  and u/=v and v/=w and w/=u}  {VPLimitation(true)}
post entered(u,m)
```

Figure 9.3: Contrats du cas d'utilisation *Enter* pour la ligne de produits de systèmes de réunions virtuelles

combinaison de variants ne modifie pas les cas d'utilisation de la ligne de produits : il suffit juste de rajouter les informations nécessaires dans le modèle de décision.

À partir du modèle de décision donné Table 9.1 et de cette description du cas d'utilisation *Enter* pour la ligne de produits, on peut par exemple en déduire la description du cas d'utilisation *Enter* pour l'édition entreprise, qui est donnée Figure 9.4.

```
UC enter(u:participant; m:mtg)
pre connected(u) and opened(m)
  and priv(m) implies authorized(u,m)
post entered(u,m)
```

Figure 9.4: Contrats du cas d'utilisation *Enter* pour la ligne de produits de systèmes de réunions virtuelles

L'ensemble des cas d'utilisation contractualisés avec les étiquettes de variation sont présentés en annexe page 175.

9.3.2 Géneration d'objectifs de test

Comme cela est illustré dans la figure 9.1, des objectifs de test pour l'ensemble de la ligne de produits et des objectifs de test spécifiques vont être générés. Cette étape est représentée Figure 9.5.

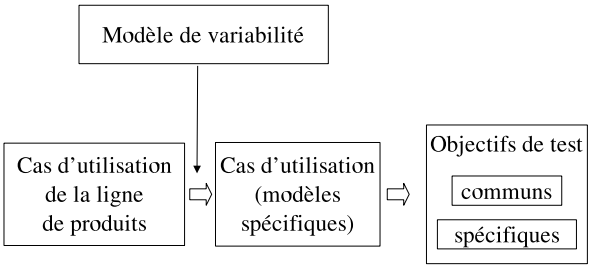


FIG. 9.5 – Génération d'objectifs de test pour les lignes de produits

On commence par générer un système de cas d'utilisation contractualisés par produit. On utilise pour cela le modèle de décision de la ligne de produits et on extrait pour chaque produit les cas d'utilisation, les paramètres et les contrats qui spécifient ce produit.

Puis on génère pour le critère souhaité un ensemble d'objectifs de test par produit, en utilisant la même méthode que celle décrite dans le chapitre 7. Ces ensembles sont ensuite analysés, de manière à détecter quels sont les objectifs communs à la ligne de produit, et quels sont les objectifs spécifiques.

Dans la section suivante, nous expliquons comment les scénarios et les cas de test sont générés à partir des objectifs de test spécifiques et génériques.

9.4 Génération de cas de test

Dans cette section, nous expliquons comment l'approche de génération de cas de test présentée dans le chapitre 7 s'applique aux lignes de produits. Cette approche utilise des scénarios des cas d'utilisation et se base sur la génération de scénarios de test, puis si les scénarios de cas d'utilisation fournis sont incomplets et abstraits, sur le couplage avec des outils de synthèse de test.

Dans cette section, nous commençons par expliquer en quoi la phase d'utilisation d'outils de synthèse de test nous paraît particulièrement cruciale dans le contexte des lignes de produits, puis nous décrivons comment les scénarios reflètent la variabilité dans les lignes de produits. Nous abordons enfin la génération de cas de tests proprement dite.

9.4.1 Impact du contexte des lignes de produits sur la génération de cas de test

Dans le chapitre 7, nous avons proposé une approche de génération de cas de test en deux temps à partir des objectifs de test. La première phase consiste à générer des scénarios de test, et la seconde à générer des patrons de test comportementaux dans l'optique d'utiliser des outils de synthèse de test.

La deuxième phase n'est pas toujours nécessaire (elle ne l'est pas si les scénarios sont complets et concrets), et est par contre coûteuse puisqu'elle nécessite de disposer d'un modèle UML simulable. Dans le contexte des lignes de produits, nous sommes intimement persuadé de l'intérêt d'une telle phase, pour les quatre principales raisons suivantes.

1. L'approche ligne de produits rend très improbable l'hypothèse d'avoir des scénarios complets et concrets. En effet, il est bien plus logique d'écrire des scénarios le plus génériques possibles, pour décrire le comportement de plusieurs produits avec un même scénario. Nous pensons que les scénarios associés aux cas d'utilisation des lignes de produits doivent mettre en jeu des paramètres plus nombreux que ceux des cas d'utilisation, et rester génériques : ne pas spécifier tous les paramètres, ni tous les échanges de messages.
   De ce fait, les scénarios de test générés par notre approche seront incomplets, et génériques.
2. L'incomplétude des scénarios de test générés peut être résolue manuellement par complétion manuelle. Cependant, si une approche manuelle est parfois envisageable, dans le contexte des lignes de produits l'effort manuel que le testeur aura à fournir pour dériver les cas de test à partir des scénarios de test est proportionnel au nombre de produits de la ligne : l'intervention manuelle va être multipliée par le nombre de produits et devient donc trop importante. Une approche automatique par utilisation d'outils de synthèse de test prend alors tout son intérêt.
3. Un des inconvénients des outils de synthèse de test est leur coût : le modèle à fournir en entrée est coûteux à construire. Cependant, les architectures des lignes

de produits sont basées sur la réutilisation. Il est donc raisonnable de penser qu'il est proportionellement moins coûteux de fournir une spécification UML détaillée pour chaque produit d'une ligne que pour un unique produit. On peut par exemple s'appuyer sur des techniques de transformation de modèles [Monestel et al., 2002] pour générer les architectures de chaque produit à partir de l'architecture globale de la ligne de produits. On peut également se baser, ainsi que nous l'avons fait pour notre exemple de réunion virtuelle, sur des patrons de conception comme le décorateur et des fabriques abstraites de manière à pouvoir configurer chaque produit à partir d'une architecture globale. Ainsi, la limitation de l'approche par synthèse de test qui était la difficulté et le coût d'obtention d'un modèle UML au format requis par UMLAUT s'atténue.

4. Enfin, comme nous le verrons, la synthèse de test permet d'obtenir à partir d'un même patron de test comportemental un cas de test spécifique pour chacun des produits de la ligne, les patrons de test deviennent donc des acquis de test réutilisables pour l'ensemble de la ligne.

9.4.2 Expression de la variabilité dans les scénarios

De même que les contrats ou les paramètres d'un cas d'utilisation varient selon les caractéristiques de chaque produit, les scénarios doivent également différer.

Tout d'abord, des scénarios valables pour tous les produits peuvent être écrits. Ensuite, on peut écrire des scénarios spécifiques à la présence de tel ou tel variant, ces scénarios seront identifiés par les mêmes étiquettes que les éléments du modèle de cas d'utilisation, comme décrit dans la section précédente. Il sera alors possible, à l'aide du modèle de décision, de déduire quels sont, pour un produit particulier, les scénarios décrivant un cas d'utilisation. Les scénarios peuvent également être représentés par des diagrammes de séquence enrichis de constructions particulières aux lignes de produits, permettant de décrire, au sein d'un même scénario, un comportement pour toute la ligne de produits. C'est l'approche préconisée par *Ziadi et al* [Ziadi et al., 2002, Ziadi et al., 2003] pour représenter des comportements dans une ligne de produits. Les auteurs proposent un certain nombre d'extensions aux diagrammes de séquence d'UML 2.0, et permettent de dériver des diagrammes de séquence spécifiques à chaque produit à partir de tels diagrammes et d'un modèle de décision proche du nôtre.

Dans tous les cas, on peut disposer de deux types de scénarios pour les cas d'utilisation : les scénarios communs à toute la ligne de produits, et ceux dépendants de variants particuliers. Il est à noter que ces scénarios sont également soit nominaux, soit exceptionnels, comme expliqué dans la section 7.1.1.

9.4.3 Génération de scénarios de test

La méthode de génération des scénarios de test se base, comme expliqué dans la section 7.1, sur la substitution des cas d'utilisation instanciés par des scénarios instanciés. Nous ne redécrirons pas ici cette méthode. Nous insistons toutefois sur le fait

que pour un produit P donné, nous n'utilisons que les scénarios lui correspondant : communs à l'ensemble de la ligne de produits et nécessitant la présence de variants que P possède. Par ailleurs, on peut noter sur la figure 9.1 que l'on génère des scénarios de test génériques et des scénarios de test spécifiques.

- Les scénarios de test communs à la ligne de produits sont obtenus à partir d'objectifs de test et de scénarios communs eux aussi à la ligne de produits.
- Les scénarios de test spécifiques à un produit particulier sont obtenus d'une part à partir des objectifs de test spécifiques, et d'autre part à partir des objectifs de test communs mais cette fois en utilisant des scénarios spécifiques.

Les scénarios de test ainsi construits sont ensuite complétés par la phase de synthèse de test.

9.4.4 Patrons de test comportementaux et lignes de produits

Il s'agit ici de générer des patrons de test comportementaux, de la même façon que décrit dans la section 7.2, et avec les mêmes restrictions sur les scénarios que dans la section précédente.

Après obtention de patrons de test comportementaux, on utilise un outil de synthèse de test pour générer des cas de test spécifiques pour chaque produit. Pour chaque patron de test commun à la ligne et pour chaque produit, on va générer des cas de test à partir des spécifications du produit (modèle UML). Pour les patrons de test comportementaux spécifiques à un produit, les cas de test seront synthétisés uniquement pour le produit ciblé.

Les cas de test sont donc automatiquement générés à partir des cas d'utilisation et des scénarios d'une part, et des spécifications comportementales des produits d'autre part.

9.4.5 Discussion méthodologique

L'approche de génération de test que nous proposons pour les lignes de produits permet :

1. la génération automatique d'un certain nombre de patrons de test comportementaux communs à la ligne de produits et spécifiques à certains produits. Cette base de patrons de test peut bien sûr être enrichie manuellement par le testeur ;
2. la capitalisation des *assets* de test à différents niveaux : objectifs de test ou patrons de test comportementaux. Ces *assets* ne seront pas modifiés, même si des modifications sont effectuées sur l'architecture ou l'implémentation d'un produit (dans la mesure où ces modifications ne sont pas incohérentes avec les exigences de départ)
3. la synthèse automatique et à la demande des cas de test pour chacun des produits, à partir des spécifications UML des produits.

Cette approche est synthétisée Figure 9.6.

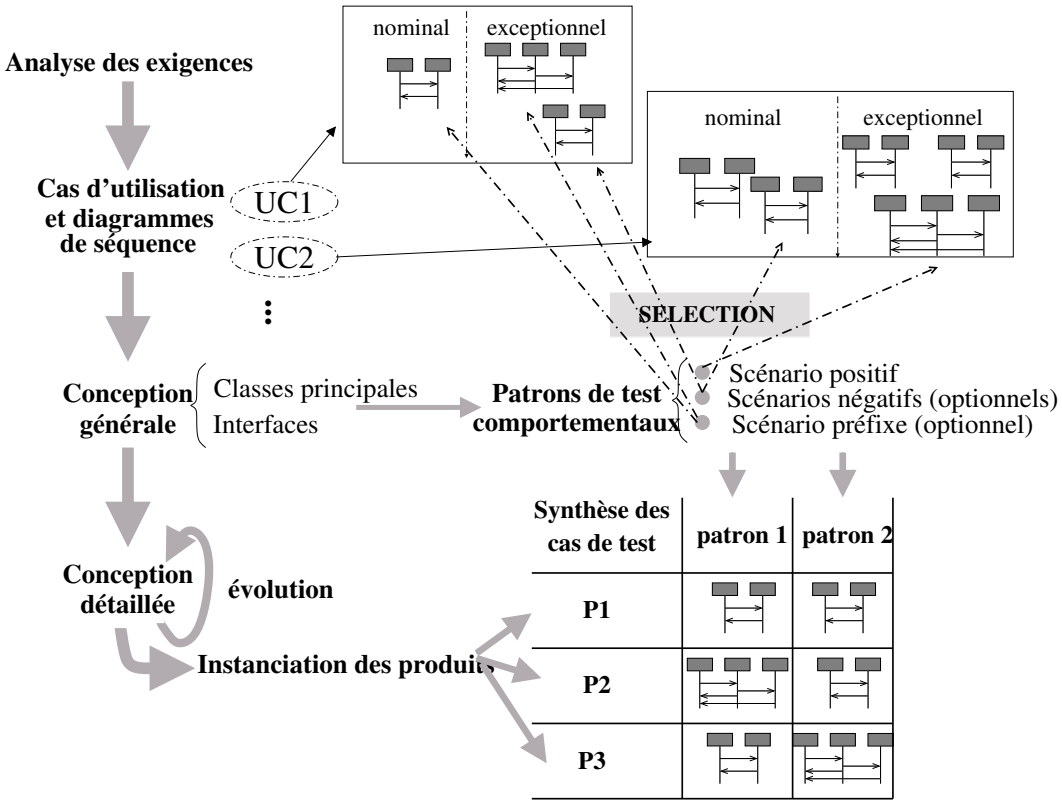


FIG. 9.6 – Approche basée sur les patrons de test comportementaux pour les lignes de produits

Durant la phase d'analyse des exigences, des cas d'utilisation sont définis pour la ligne de produits, et sont illustrés par des scénarios. Il est précisé si les cas d'utilisation et les sénarios sont présents dans tous les produits, ou seulement dans les produits possédant un variant particulier. Une telle spécification ainsi que la définition du modèle de décisions permet alors la génération automatique de patrons de test comportementaux. Parallèlement à ce processus de génération de test, l'architecture de lignes de produits est définie, et les différents produits sont instanciés. Les outils de synthèse de test vont alors produire pour chaque produit et pour chaque patron de test un cas de test spécifique. L'intérêt majeur de cette approche est que c'est l'outil de synthèse qui, disposant de la spécification opérationnelle de chaque produit, va déterminer pour chaque produit quel est le protocole exact permettant d'exhiber un cas de test satisfaisant les patrons de test comportementaux.

Nous avons généré les cas de test par cette approche pour chacun des produits définis dans notre exemple de lignes de produits de réunions virtuelles. Comme dans le chapitre 8, nous n'avons pas utilisé d'outils de synthèse, les scénarios de test générés pouvant directement être appliqués par notre pilote de test. Nous avons généré respectivement 80, 128 et 149 cas de test pour les éditions standard, personnelle et entreprise avec les critères de test *tous les termes des préconditions* et *robustesse*. Le coût en terme d'intervention des testeurs de la génération de test pour l'ensemble de la ligne de produits a été très proche du coût de génération pour l'exemple simple de réunion

virtuelle présenté Section 4.3 : le coût supplémentaire a juste consisté en la spécification des nouvelles fonctionnalités, la définition des parties variantes, et le modèle de décision. Le coût de génération de test est donc bien réduit considérablement par notre approche : il n'est plus proportionnel au nombre de produits de la ligne.

9.5 Conclusion

Dans cette section, nous avons appliqué les principes décrits dans les chapitres 6 et 7 aux lignes de produits logicielles. L'application de notre approche aux lignes de produits montre sa pertinence, dans la mesure où la technique de génération de test que nous avons proposée a été créée pour répondre aux problématiques de test des lignes de produits. Cette technique de génération de test répond parfaitement aux critères que nous avions fixés sur les méthodes de test de lignes de produits.

Les contributions principales de ce chapitre sont les suivantes :

- Nous avons proposé un modèle de cas d'utilisation permettant de prendre en compte la variabilité au niveau des cas d'utilisation eux-mêmes, de leurs paramètres, de leurs contrats, et de leurs scénarios ;
- En nous basant sur ce modèle, nous avons transposé les principes de génération de test décrits dans les chapitres précédents au contexte des lignes de produits logicielles ;
- L'approche proposée permet la capitalisation d'assets de test au niveau de la ligne de produits et au niveau des produits ;
- L'application de notre approche permet que la création d'un nouveau produit ne réclame pas une forte charge concernant le test système. L'ajout d'un nouveau produit implique d'enrichir le modèle de décision, qui permet alors la génération automatique d'objectifs de test, sans modélisation supplémentaire des exigences. Par la suite, dans le cas d'utilisation de séquences de test, le testeur doit juste les dériver en cas de test. Dans le cas d'utilisation de patrons de test comportementaux, le testeur doit juste fournir un modèle UML simulable du produit, ce qui, comme nous l'avons expliqué, n'est pas nécessairement coûteux dans le contexte des lignes de produits.

Quatrième partie

Conclusions et perspectives

Les méthodes formelles offrent une solution théorique à la génération automatique de tests à partir des spécifications d'un système. Cependant, ces approches ne sont pas directement applicables dans de nombreux contextes industriels pour un certain nombre de raisons comme le manque d'intégration aux cycles classiques de développement, les problèmes de maintenance des spécifications, et surtout le manque d'adéquation au contexte des lignes de produits logicielles. C'est pourquoi nous avons proposé une approche originale de génération de tests système à partir des exigences qui, au lieu de tirer l'industrie vers les méthodes formelles, s'inspire des pratiques industrielles (utilisation d'un langage contrôlé, et de standards de notation) et les amène vers des modèles formellement exploitables pour la génération automatique de tests.

La thèse défendue dans ce manuscrit est qu'il est possible de faire baisser le coût global du test dans des contextes complexes, en utilisant des modèles proches de ceux couramment utilisés dans l'industrie, et notamment en se basant sur les premiers modèles d'exigences des systèmes sous test. La contribution que nous apportons est une approche de génération automatique de tests à partir des exigences d'un système intégrant les contraintes suivantes : compatibilité avec les pratiques industrielles courantes, maîtrise du coût du test, adaptabilité au contexte des lignes de produits logicielles, et complexité des logiciels réels.

La compatibilité avec les pratiques industrielles courantes est assurée par l'adoption du standard de notation UML, et la méthode permettant de rédiger les exigences en s'assurant de manière incrémentale de leur cohérence. Pour cela, nous avons proposé un langage naturel contrôlé, dont la sémantique est donnée dans un modèle simulable. Les exigences sont analysées syntaxiquement et sémantiquement, puis peuvent être simulées, ce qui permet de détecter et supprimer les ambiguïtés dans les exigences, et de s'assurer qu'elles sont bien cohérentes les unes avec les autres. La simulation des exigences s'avère extrêmement utile, et permet au rédacteur de s'assurer que les exigences écrites correspondent bien à l'idée qu'il en avait, c'est-à-dire que l'interprétation sémantique proposée automatiquement par l'outil de traitement des exigences est bien la même que celle qu'il avait en tête.

Le modèle de simulation est la pierre angulaire de l'approche. Il se compose de cas d'utilisation paramétrés, et augmentés de contrats (pré- et post-conditions) exprimés dans un langage dédié. Ce modèle, sans aucun doute beaucoup moins expressif que la plupart des langages formels, a l'avantage d'avoir une sémantique opérationnelle clairement définie, et d'être d'une complexité parfaitement maîtrisable. Ce modèle de simulation permet de construire un graphe comportemental de l'application, à un fort niveau d'abstraction. Nous avons proposé des critères de test permettant de sélectionner dans ce graphe des chemins pertinents pour le test, appelés *objectifs de test*. Une évaluation expérimentale des critères de test nous a permis de déterminer quels critères, parmi ceux que nous avions proposés, sélectionnaient les objectifs de test les plus pertinents.

Les objectifs de test sont d'un niveau très abstraits, et donc potentiellement très éloignés des cas de test qui peuvent être effectivement exécutés par un pilote de test. Pour obtenir des cas de test, il est nécessaire d'avoir connaissance de l'interface réelle du système : il faut pouvoir déterminer quels échanges exacts doivent avoir lieu entre le système et les acteurs pour réaliser un cas d'utilisation particulier. Nous avons proposé

d'utiliser pour obtenir cette information des scénarios de niveau système, attachés à chaque cas d'utilisation, et exprimés sous forme de diagrammes de séquences, de machines à états, ou de diagrammes d'activités. Ces scénarios étant de niveau système, ils sont simples à concevoir dès lors que les interfaces du système sont définies. Ces scénarios sont utilisés pour générer des scénarios de test à partir des cas de test. Les scénarios de test sont plus ou moins proches des cas de test, selon la complexité du système et la complétude des scénarios décrivant chaque cas d'utilisation. Des scénarios de test proches des cas de test pourront être facilement dérivés manuellement en cas de test, pour les autres, nous proposons dans ce manuscrit une utilisation d'outils de synthèse de test.

Cette approche de génération de test assure la maitrise du coût du test ; elle permet de générer des tests à partir des exigences, en se basant sur des modèles maîtrisables, et des critères de test adéquats.

Les trois phases de notre approche ont été appliquées aux lignes de produits logicielles. Ainsi, nous générons des tests spécifiques pour chaque produit d'une ligne, à partir des exigences de la ligne de produits et d'un modèle de décision. Une telle approche automatique permet la validation rapide de nouveaux produits dérivés de la ligne.

Nous avons dans un premier temps validé notre approche sur des cas d'études académiques : une analyse du code couvert par les cas de test générés automatiquement à partir des exigences a montré qu'environ 80% du code atteignable de chaque cas d'étude était couvert. Globalement, les comportements nominaux des applications sont couverts, et une amélioration du critère de robustesse permettrait sans doute de couvrir plus de code de robustesse. Ces résultats montrent néanmoins que les tests que nous avons généré à faible coût de modélisation initiale et de manière automatique sont pertinents au niveau de l'implémentation.

Nous nous sommes également attaché à évaluer notre approche sur des cas d'études industriels. Ainsi, nous avons étudié la pertinence de notre approche sur un composant fourni par TAS. Si l'ampleur du composant et des raisons techniques ne nous ont pas permis d'évaluer la couverture de code atteinte par les tests que nous avons générés à partir des exigences, cette étude nous a montré la faisabilité de notre approche sur des composants réels. Des études complémentaires sont actuellement en cours : TAS dispose de nos outils prototypes et les utilise sur un autre composant logiciel. Le fait que notre approche puisse être appliquée sur un composant réel, et soit de plus actuellement en cours d'utilisation par TAS, est à nos yeux la meilleure validation de notre approche, et assure son applicabilité sur des logiciels de complexité réelle.

Perspectives

Ce manuscrit a permis de montrer qu'une approche à base de modèles simples permet de générer rapidement des tests pertinents, tant pour des composants classiques que pour des lignes de produits logicielles. Il reste de nombreuses pistes à explorer pour permettre d'améliorer l'approche que nous avons proposée.

Une piste de recherche à suivre pour obtenir une méthode de génération de test plus complète est l'intégration des aspects non-fonctionnels des spécifications. Nous aimerions en effet proposer des moyens d'exprimer des contraintes de qualité de service comme les temps de réponse ou les contraintes de mémoire, de manière à tester la qualité de service du produit fini vis-à-vis du niveau qui était spécifié.

Des pistes d'amélioration du langage des exigences ont été données dans la section 5.5 page 68. Nous avons proposé une approche qui permet d'utiliser pour la rédaction des exigences un langage proche du langage naturel, mais qui reste néanmoins très contraint : notre langage des exigences ne permet notamment l'utilisation que de peu de structures grammaticales. L'utilisation des résultats des nombreuses recherches sur l'analyse du langage naturel simplifierait encore l'utilisation de notre approche.

Concernant le modèle sur lequel nous nous sommes appuyé (le modèle de cas d'utilisation), nous aimerions étudier dans quels cas l'introduction d'arithmétique serait utile et non nocive à un haut niveau d'exigences.

Les objectifs de test que nous avons proposés sont un premier pas, et mériteraient d'être améliorés pour atteindre un meilleur niveau de couverture de code. Nous aimerions notamment travailler plus précisément au niveau du test de robustesse, pour générer plus finement, en utilisant les scénarios, des tests exhibant des comportements exceptionnels. Nous voudrions également étudier les résultats que fourniraient l'utilisation d'autres critères de test inspirés des critères utilisés pour le test structurel de conditionnelles.

Concernant la génération automatique de cas de test, nous avons proposé une approche pragmatique uniquement basée sur la substitution des cas d'utilisation par des scénarios, et augmentée au besoin par l'utilisation d'outils de synthèse. Nous aimerions, concernant l'approche la plus simple, étudier les apports du couplage avec un solveur de contraintes pour mieux gérer les données de test. Nous n'avons pas pu valider correctement l'intérêt du couplage avec des outils de synthèse : nous souhaiterions expérimenter l'usage des outils de synthèse Agatha et TGV, et notamment étudier le gain obtenu par l'usage de ces outils par rapport à la complexité des modèles à leur fournir. Nous souhaiterions également étudier des approches de génération de test distribué avec testeurs répartis. Nous avons commencé à étudier des modèles de vraie concurrence [Jard, 2002], et nous souhaiterions voir si nous pouvons nous appuyer sur ces modèles à un haut niveau d'exigence.

Enfin, nous souhaitons insérer notre approche de génération de test à partir des exigences dans un processus de développement guidé par les modèles, en étudiant notamment la représentation des liens de traçabilité entre les différents modèles manipulés dans notre approche.

Glossaire

Cas de test Un cas de test est directement exécutable par un pilote de test : il contient toutes les informations nécessaires à son exécution, et embarque son oracle.

Cas d'utilisation contractualisé Un cas d'utilisation contractualisé est un cas d'utilisation avec une précondition et une postcondition exprimées en UCL. Un cas d'utilisation contractualisé possède également des paramètres qui peuvent être utilisés dans les contrats.

LDE Le langage de description des exigences est un langage contrôlé, c'est-à-dire un sous-ensemble des constructions du langage naturel (l'anglais). Sa sémantique est donnée par des patrons d'interprétation.

Objectif de test Un objectif de test est la représentation d'un comportement du système sous test que l'on souhaite pouvoir exhiber. Dans notre approche, un objectif de test est représenté par une séquence de cas d'utilisation instanciés.

Patron de test comportemental Un patron de test comportemental est un ensemble de scénarios : un scénario positif précisant quel comportement le testeur souhaite exhiber, un scénario préfixe précisant comment mettre le système dans un état permettant au scénario positif de se produire, et un ensemble de scénarios négatifs précisant quels comportements le testeur ne souhaite pas voir. Tous ces scénarios peuvent être incomplets et abstraits, en ce sens qu'ils n'explicitent pas nécessairement tous les paramètres impliqués, ni toutes les instances.

Patron d'interprétation Un patron d'interprétation est une règle permettant d'associer une structure syntaxique du LDE avec une interprétation. L'interprétation est donnée dans ce manuscrit en termes de cas d'utilisation contractualisés.

Scénario de test Scénario modélisé par un diagramme de séquence UML, représentant un cas de test. Un scénario de test peut être plus ou moins éloigné du cas de test lui correspondant. Un scénario de test peut contenir des données non initialisées, et être incomplet (contenir des paramètres non définis par exemple, et représentés par des astérisques).

UCL L'UCL (Use case Constraint Language) est un langage de contrats pour les cas d'utilisation. Il permet d'exprimer des contrats interprétables. Ce langage est basé sur la logique du premier ordre.

UCTS Un UCTS (Use Case Transition System) est un système de transitions étiquetées, dont les nœuds représentent une abstraction de l'état du système en termes de prédicats instanciés, et dont les transitions représentent les cas d'utilisation instanciés qui les étiquettent.

Cinquième partie

Annexes

Annexe A

Compléments sur l'exemple de réunion virtuelle

A.1 Spécification du système de réunions virtuelles

```
# use case CONNECT
UC connect(u :participant)
pre not connected(u)
post connected(u)

# use case CONSULT
UC consult(u :participant)
pre connected(u)
post

# use case PLAN
UC plan_std(u : participant ; m : mtg ; t :type_meeting)
pre connected(u) and not created(m)
post created(m) and manager(u,m) and type(m)=t

# use case SETMODERATOR
UC setmoderator(u,v :participant ;m :mtg)
pre not opened(m) and not closed(m) and manager(u,m)
    and forall (w :participant) {not moderator(w,m)}
    and not type(m)="demo"
post moderator(v,m)

# use case SETACCESSLIST
UC setaccesslist(u,v :participant ;m :mtg)
pre not opened(m) and not closed(m) and manager(u,m)
    and not authorized(v,m)
    and type(m)="private"
post authorized(v,m)

#use case OPEN
UC open(u :participant ;m :mtg)
pre not memo(m) and (created(m) and moderator(u,m)
    and not closed(m) and not opened(m) and connected(u))
```

```
post opened(m)

#use case ENTER
UC enter(u :participant; m :mtg)
pre connected(u) and opened(m) and not exists(n :mtg){ entered(u,n) }
pre {type(m)="private"} implies {authorized(u,m)}
post entered(u,m)

#use case ASK
UC ask(u :participant; m :mtg)
pre entered(u,m) and not asked(u,m)
post asked(u,m)

#use case HANDOVER
UC handover(u,v :participant; m :mtg)
pre not type(m)="demo" and asked(v,m) and moderator(u,m) and entered(u,m)
    and not exists(w :participant) { speaker(w,m) }
post speaker(v,m) and not asked(v,m)

#use case SPEAK
UC speak(u :participant; m :mtg)
pre {not type(m)="demo"} implies {speaker(u,m)}
pre {type(m)="demo"} implies {asked(u,m)
    and not exists(w :participant){speaker(w,m) and w/=u}}
post {type(m)="demo"} @implies {speaker(u,m)}

#use case OVER
UC over(u :participant; m :mtg)
pre speaker(u,m) or (moderator(u,m) and entered(u,m)
    and exists(v :participant){speaker(v,m)})
post not speaker(u,m)
post {type(m)=demo} @implies {not asked(u,m)}


#use case LEAVE
UC leave(u :participant; m :mtg)
pre entered(u,m) and not asked(u,m) and not speaker(u,m)
post not entered(u,m) and {not exists(v :participant){u/=v and entered(u,v)}}
     implies {not closed(m) and not opened(m)}

#use case CLOSE
UC close(u :participant; m :mtg)
pre not demo(m) and opened(m) and moderator(u,m)
post not opened(m) and closed(m) and
     forall(v :participant) {not entered(v,m) and not asked(v,m) and not speaker(v,m) }

#use case DISCONNECT
UC disconnect(u :participant)
pre connected(u)
post not connected(u)
     and not exists(m :mtg) { entered(u,m) or asked(u,m) or speaker(u,m) }
```

A.2 Spécification de la ligne de produits "réunions virtuelles"

```
#VP1=meeting limitation
#VP2=meeting types
#VP3=recording
#VP4=language
#VP5=supervisor

# use case CONNECT
UC connect(u :participant)
pre not connected(u)
post connected(u)

# use case CONSULT
UC consult(u :participant)
pre connected(u)
post

# use case PLAN
UC plan_std(u : participant ; m : mtg ; t :type_meeting)
pre connected(u) and not created(m)
post created(m) and manager(u,m) and type(m)=t

# use case SETMODERATOR
UC setmoderator(u,v :participant ;m :mtg) {VP2{std,priv}}
pre not opened(m) and not closed(m) and manager(u,m)
    and forall (w :participant) {not moderator(w,m)}
    and not type(m)="demo"
post moderator(v,m)

# use case SETACCESSLIST
UC setaccesslist(u,v :participant ;m :mtg) {VP2{priv}}
pre not opened(m) and not closed(m) and manager(u,m)
    and not authorized(v,m)
    and type(m)="private"
post authorized(v,m)

#use case OPEN
UC open(u :participant ;m :mtg)  {VP2(std,priv)}
pre not memo(m) and (created(m) and moderator(u,m)
    and not closed(m) and not opened(m) and connected(u))
post opened(m)

#use case ENTER
UC enter(u :participant ; m :mtg)
pre connected(u) and opened(m) and not exists(n :mtg){ entered(u,n) }
pre {type(m)="private"} implies {authorized(u,m)}  {VP2(priv)}
post entered(u,m)

#use case ASK
UC ask(u :participant ; m :mtg)
pre entered(u,m) and not asked(u,m)
```

```
post asked(u,m)

#use case HANDOVER
UC handover(u,v :participant; m :mtg)  {VP2(std,priv)}
pre not type(m)="demo" and asked(v,m) and moderator(u,m) and entered(u,m)
    and not exists(w :participant) { speaker(w,m) }
post speaker(v,m) and not asked(v,m)

#use case SPEAK
UC speak(u :participant; m :mtg)
pre {not type(m)="demo"} implies {speaker(u,m)} {VP2(std,priv)}
pre {type(m)="demo"} implies {asked(u,m)
pre  not exists(w :participant){speaker(w,m) and w/=u}} {VP2(demo)}
post {type(m)="demo"} @implies {speaker(u,m)} {VP2(demo)}

#use case OVER
UC over(u :participant; m :mtg)
pre speaker(u,m) or (moderator(u,m) and entered(u,m)
    and exists(v :participant){speaker(v,m)})
post not speaker(u,m)
post {type(m)=demo} @implies {not asked(u,m)} {VP2(demo)}


#use case LEAVE
UC leave(u :participant; m :mtg)
pre entered(u,m) and not asked(u,m) and not speaker(u,m)
post not entered(u,m) and {not exists(v :participant){u/=v and entered(u,v)}}
     @implies {not closed(m) and not opened(m)}

#use case CLOSE
UC close(u :participant; m :mtg) {VP2(std,priv)}
pre not demo(m) and opened(m) and moderator(u,m)
post not opened(m) and closed(m) and
     forall(v :participant) {not entered(v,m) and not asked(v,m) and not speaker(v,m)}

#use case DISCONNECT
UC disconnect(u :participant)
pre connected(u)
post not connected(u)
     and not exists(m :mtg) {entered(u,m) or asked(u,m) or speaker(u,m)}

#use case RECORD
UC record(u :participant; m :mtg) {VP3(true)}
pre connected(u) and manager(u,m) and not opened(m) and not recorded(m)
post recorded(m)

#use case CHOOSE_LANGUAGE
UC chooseLanguage(u :participant; l :lang) {VP4(Fr,Sp)}
pre connected(u) and not exists (ll :lang){language(u,ll)}
post language(u,l)

#use case SPY
UC spy(u :participant) {VP5(true)}
pre not spying(u) and supervisor(u) and connected(u)
```

```
post spying(u)
```

Annexe B

Analyse des exigences avec le LDE

B.1 Grammaire du LDE

```
lde :    (r_Exigence)+ ;

/*  INITIAL CONFIGURATION  */

r_InitialConfiguration :
    INITIAL CONFIGURATION COLON
    DASH (r_InitialConfigurationItem)
    (DASH (r_InitialConfigurationItem) )* ;

r_InitialConfigurationItem :
    THERE IS r_ObjectDeclaration ;

r_ObjectDeclaration :
    ONE STRING
|   A STRING NAMED STRING ;

/*  REQUIREMENT */

r_Exigence :
    REQUIREMENT
    (r_RequirementNumber)
    (r_RequirementName)
    (r_RequirementDescription)+ ;

r_RequirementNumber :
    INT_NUMBER
    (DOT INT_NUMBER)* ;

r_RequirementName : STRING ;

r_RequirementDescription :
    COMMENT (r_RequirementDescriptionBody DOT )* ;
```

```
/* REQUIREMENT DESCRIPTION BODY */

r_RequirementDescriptionBody :
    r_RequirementIfThenElse
|   r_RequirementBasic
|   r_InitialConfigurationItem ;

/* REQUIREMENT IF THEN ELSE */

r_RequirementIfThenElse :
    IF  r_RequirementBasic
    THEN  r_RequirementBasic
    (ELSE r_RequirementBasic)? ;

/*  REQUIREMENT BASIC */

r_RequirementBasic :
    r_RequirementBasicComposite
|   r_RequirementBasicLeaf ;

/*  REQUIREMENT BASIC LEAF  */

r_RequirementBasicLeaf :
    r_ServiceActivation
|   r_ObservableProperty
|   OPENPARENTHESIS r_RequirementBasic CLOSEPARENTHESIS ;

/*  REQUIREMENT BASIC COMPOSITE  */

r_RequirementBasicComposite :
    r_RequirementBasicCompositeLogical
|   r_RequirementBasicCompositeTemporal ;

/*  REQUIREMENT BASIC COMPOSITE LOGICAL  */

r_RequirementBasicCompositeLogical :
    r_RequirementBasicLeaf (AND | OR | IMPLIES THAT)
        r_RequirementBasic ;

/*  REQUIREMENT BASIC COMPOSITE TEMPORAL  */

r_RequirementBasicCompositeTemporal  :
    r_RequirementBasicCompositeTemporalSynchronous
|   r_RequirementBasicCompositeTemporalAsynchronous ;

r_RequirementBasicCompositeTemporalSynchronous :
    r_RequirementBasicLeaf  (WHEN | AFTER | BEFORE)
        r_RequirementBasic
```

```
|    (WHEN | AFTER | BEFORE) r_RequirementBasic COMMA
   r_RequirementBasicLeaf ;

r_RequirementBasicCompositeTemporalAsynchronous :
    r_RequirementBasicLeaf  (WHILE | UNTIL)  r_RequirementBasic ;

/*  OBSERVABLE REFERENCE  */

r_ObservableReference :
    r_ObservableReferenceExplicit
|   r_ObservableReferenceImplicit ;

r_ObservableReferenceExplicit  :
    THE r_ObservableReferenceName
    OF r_ObservableReferenceOwner
    (FOR r_ObservableReferenceOwner)? ;

r_ObservableReferenceImplicit :
    r_ObservableReferenceOwner
    (FOR r_ObservableReferenceOwner)? ;

r_ObservableReferenceOwner :
    (THE | EACH | A | ONE | THIS)? STRING ;

r_ObservableReferenceName : STRING ;

/*  OBSERVABLE PROPERTY  */

r_ObservableProperty :
    r_ObservablePropertyComposite
|   r_ObservablePropertyBasic ;

/*  OBSERVABLE PROPERTY BASIC  */

r_ObservablePropertyBasic :
    r_ObservablePropertyStable
|   r_ObservablePropertyChange
|   OPENPARENTHESIS r_ObservableProperty CLOSEPARENTHESIS ;

r_ObservablePropertyStable  :
    r_ObservableReference (IS|MUST_BE) r_ObservablePropertyValue ;

r_ObservablePropertyChange :
    r_ObservableReference CHANGES
    (FROM r_ObservablePropertyValue TO r_ObservablePropertyValue)?
|   r_ObservableReference (REMAINS | BECOMES)
   r_ObservablePropertyValue ;

/* OBSERVABLE PROPERTY COMPOSITE  */
```

```
r_ObservablePropertyComposite :
    r_ObservablePropertyBasic (AND | OR)  r_ObservableProperty ;

/*  OBSERVABLE PROPERTY VALUE  */

r_ObservablePropertyValue : (NOT)? STRING ;

/*   SERVICE ACTIVATION   */

r_ServiceActivation :
    r_ServiceActivatorInstance
    (DOES | DID | STARTS TO | STOPS TO | CAN)
    r_ServiceActivationList ;

r_ServiceActivationList :
    r_ServiceActivationName
|   COLON  (DASH r_ServiceActivationName)+ ;

r_ServiceActivationName  :
    STRING ((THE | THIS | EACH | ONE | A) STRING)? ;

/*  SERVICE ACTIVATOR INSTANCE  */

r_ServiceActivatorInstance :
    (THE | EACH | THIS | A) STRING ;

/*  LDE LEXER   */
A               : "a";
AFTER           : "after";
AND             : "and";
BECOMES         : "becomes";
BEFORE          : "before";
CAN             : "can";
CHANGES         : "changes";
CONFIGURATION   : "configuration";
DID             : "did";
DOES            : "does";
EACH            : "each";
ELSE            : "else";
FOR             : "for";
FROM            : "from";
IF              : "if";
IMPLIES         : "implies";
INITIAL         : "initial";
IS              : "is";
MUST_BE         : "must be";
NAMED           : "named";
NOT             : "not";
```

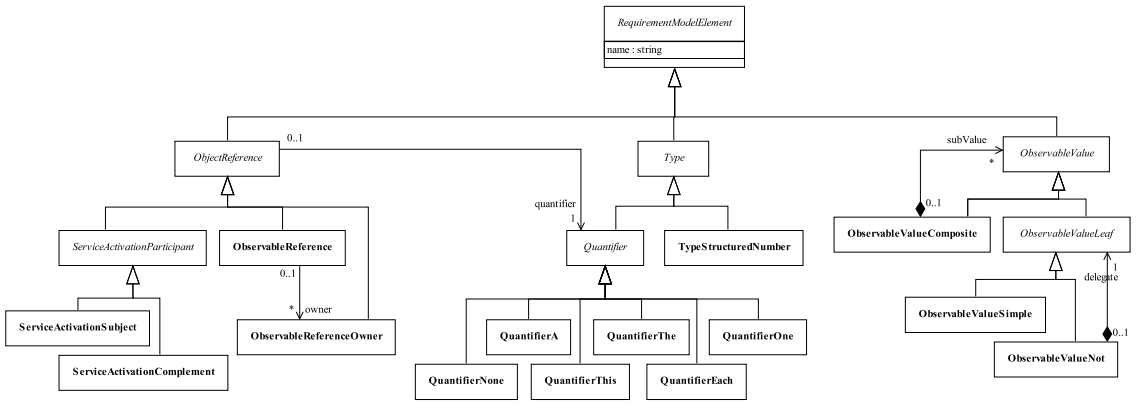


FIG. B.1 – AST LDE : références, quantifieurs, valeurs et observables

```
NOTIFICATION   : "notification";
OF             : "of";
ONE            : "one";
OR             : "or";
PRECONDITION   : "precondition";
POSTCONDITION  : "postcondition";
REMAINS        : "remains";
REQUIREMENT    : "requirement";
STARTS         : "starts";
STOPS          : "stops";
THAT           : "that";
THE            : "the";
THERE          : "there";
THEN           : "then";
THIS           : "this";
TO             : "to";
UNTIL          : "until";
WHEN           : "when";
WHILE          : "while";
WITH           : "with";
```

B.2 Compléments sur l'AST LDE

Les figures B.1 à B.4 présentent des diagrammes de classes de l'AST LDE.

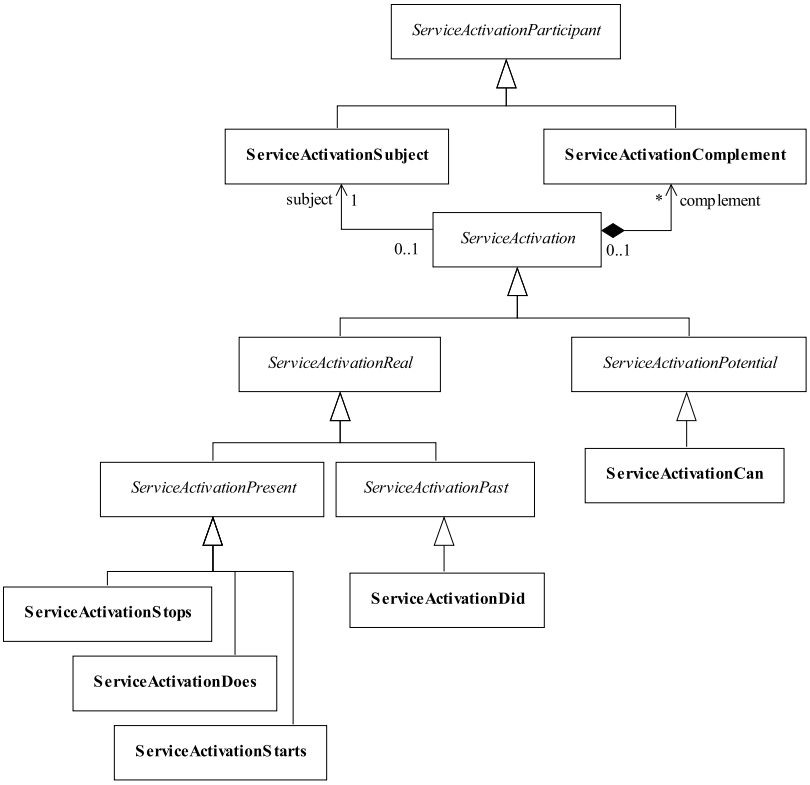


FIG. B.2 – AST LDE spécifique à l'activation de services

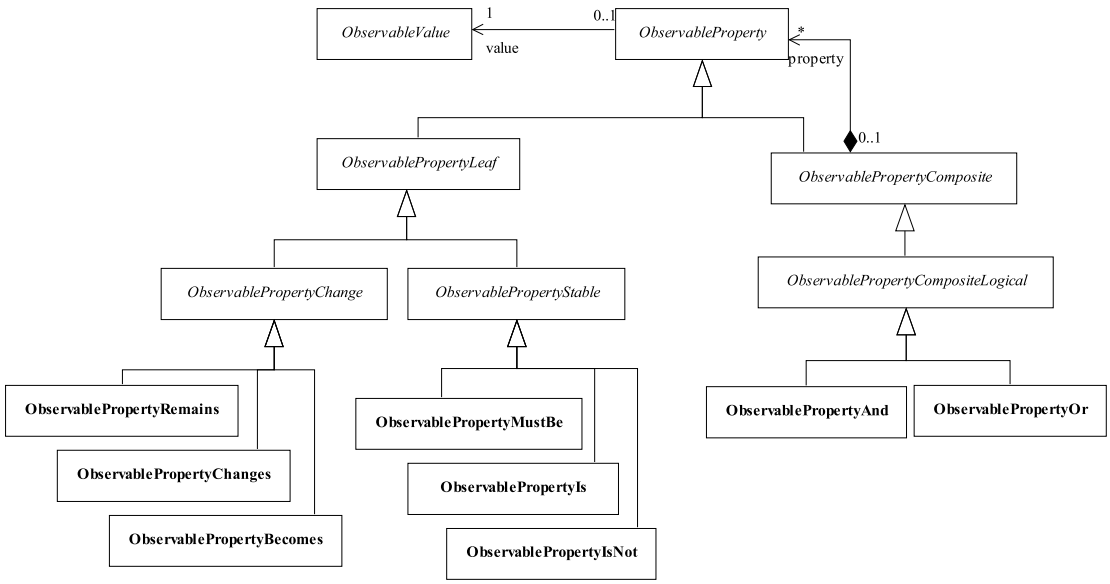


FIG. B.3 – AST LDE spécifique aux observables

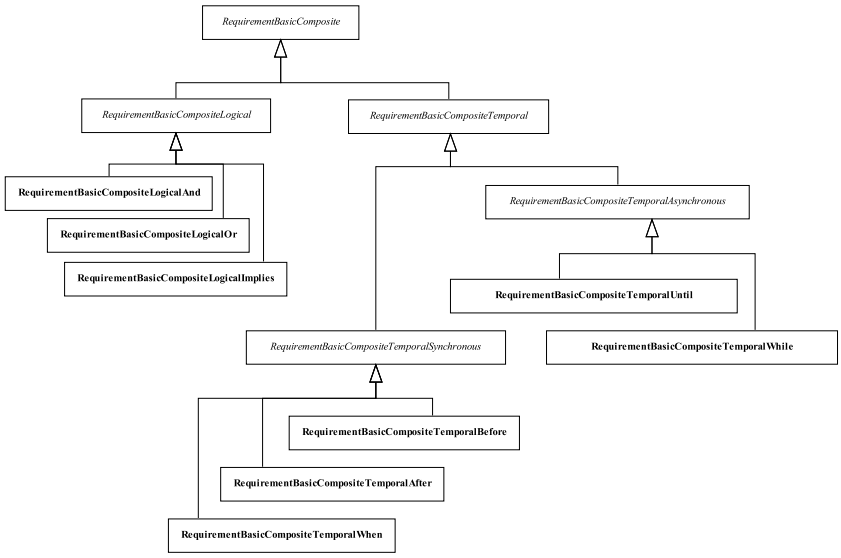


FIG. B.4 – AST LDE spécifique à la composition d'exigences

Annexe C

Des cas d'utilisation aux objectifs de test

C.1 Algorithmes de satisfaction des critères de test

Nous donnons ici les algorithmes permettant de calculer des ensembles d'objectifs de test satisfaisant les critères de test *tous les nœuds* et *tous les termes des préconditions*, définis Section 6.3.

C.2 Diagrammes d'activité vers cas d'utilisation contractualisés

Nous présentons ici l'algorithme de génération d'un système de cas d'utilisation contractualisés à partir d'un diagramme d'activité. Les principes de cet algorithme ont été présentés dans la section 8.2.

Algorithme 3 Algorithme produisant l'ensemble des objectifs de test satisfaisant le critère *tous les nœuds*

```
algorithm buildTestObjectives_with_all_nodes_criterion
param ucts : UCTS
var
  result : LIST[LIST[IUC]]
  built_paths : LIST[LIST[EDGES]]
init
  ucts.resetVisitingMarks;
  ucts.initState.set(visited)
  ∀t ∈ucts.initState.getOutgoingTransitions()
  do
    List l=new List;l.add(t);
    built_paths.add(l);
   t.destination.set(visited);
  done
body
  while (ucts.nodes_to_visit≠ ∅)
  do
   var new_built_paths : LIST[LIST[EDGES]];
     ∀p ∈built_paths
     do
       if (p.getLastElement.getOutgoingTransitions()=∅)
       then
         result.add(p.dumpIUC())
       else
         ∀t ∈p.getLastElement.getOutgoingTransitions()
         do
           if t.destination.is_visited
           then
             result.add(p.dumpIUC())
           else
            temp_p :LIST[EDGES];
            temp_p←clone(p);temp_p.addLast(t);
            new_built_paths.add(temp_p);
             t.destination.set(visited);
           fi
         done
       fi
     done
     built_paths=new_built_paths
  done
end
return result
```

Algorithme 4 Algorithme produisant un ensemble d'objectifs de test satisfaisant le critère *tous les termes des préconditions*

algorithm buildTestObjectives_with_APT_criterion
param ucts : UCTS, set_uc : LIST[USECASE]

```
body
  ∀ uc ∈set_uc
  do
    var set_b : LIST[BoolExpr]
    set_b ← getAllTrueValuations(uc.pre)
    ∀ e ∈set_b
    do
     if (getPath(e,ucts).dumpIUC())≠∅ then
     result.add(getPath(e,ucts).dumpIUC)
     fi
    done
  done
  return result
end
.
function getAllTrueValuations
param b :BoolExpr
return LIST[BoolExpr]
// return all the valuations making b true,
//under the form of boolean expressions
.
function getPath
param b:BoolExpr, ucts:UCTS
return LIST[EDGES]∪ ∅
// return the first ucts path found leading
//to a state where b is true,
//or an empty list if such a path cannot be found
```

Algorithme 5 Algorithme d'obtention d'un système partiel de cas d'utilisation contractualisés à partir d'un diagramme d'activité

```
Algo build_contracts
until all use cases are marked do {
choose a non-marked use case uc;
uc.post=phase1(uc,uc.getOutgoingActivityEdge.target);
(pre,post)=phase2(uc,uc.getIncomingActivityEdge.source);
uc.pre=pre;
uc.post=uc.post and post;
uc.mark
}

function phase1(currentNode:Node,nextNode:Node):Expr
// return a partial postcondition and marks nodes where tokens may be put
if nextNode.type="action" or nextNode.type="join" or nextNode.type="decision"
 or nextNode.type="merge"
then {-- stop recursion
  ActivityEdge e=activityEdge(currentNode,nextNode);
  e.mark;
  return e.name;}
elseif nextNode.type="fork"
then { -- forward recursion
  Expr cumulPost;
  for each e in nextNode.outgoingActivityEdges(){
    cumulPost:=cumulPost and phase1(nextNode,e.target);
  } return cumulPost}
elseif nextNode.type="finalActivity"
then { --stop recursion
  return end;}
fi

function phase2(currentNode:Node,precNode:Node):Expr*Expr
// return the precondition and a partial postcondition
local pre, post:Expr
ActivityEdge e= activityEdge(precNode,currentNode);
if e.isMarked or precNode.type="initialNode"
then {-- stop recursion
  pre=e;
  post=not e ;}
else { // backward recursion
  if precNode.type="join"
  then {
    for each e in precNode.incomingActivityEdges{
      Expr ppre, ppost;
      (ppre,ppost)=phase2(precNode,e.source);
      pre=pre and ppre
      post=post and  f.post};}
  elseif precNode.type="decision"
    then {
      Expr ppre, ppost;
      ActivityEdge e=precNode.incomingActivityEdge();
      (ppre,ppost)=phase2(precNode,e.source)
      pre=ppre;
      post=ppost;}
  elseif precNode.type="fork"
    then {
      Expr ppre, ppost;
      ActivityEdge e=precNode.incomingActivityEdge();
      (ppre,ppost)=phase2(precNode,e.source)
      pre=ppre;
      post=ppost;}
  elseif  precNode.type="merge"
    then {
      for each e in precNode.incomingActivityEdges{
        Expr ppre, ppost;
        (ppre,ppost)=phase2(precNode,e.source);
        pre=pre or ppre;
        post= post and (ppre implies ppost)}}}
return (pre,post)
```

Annexe D

Outils prototypes développés : analyseur des exigences, simulation des cas d'utilisation et génération de tests

D.1 Analyseur de LDE et transformation vers le modèle de cas d'utilisation

L'analyseur de LDE a été construit avec ANTLR [ANTLR, 2004] à partir de la grammaire du LDE. Cet analyseur a été intégré dans un plug-in Eclipse [IBM, 2004] ; il comprend des facilités comme la coloration syntaxique et la détection d'erreurs de syntaxe.

Les transformations vers le modèle statique et le modèle de cas d'utilisation sont implémentées dans ce plug-in. Ainsi, les exigences sont entrées dans un éditeur, puis les différents modèles générés sont présentés dans différentes vues Eclipse. Nous présentons Figure D.1 la vue d'édition des exigences et son navigateur associé.

D.2 Navigateur de modèles de cas d'utilisation et simulateur

La figure D.2 présente une copie d'écran de l'outil de navigation pour les cas d'utilisation. La partie gauche de la fenêtre permet de naviguer dans le modèle de cas d'utilisation, et la partie droite permet d'afficher les propriétés de l'élément de modèle sélectionné dans le navigateur (dans la figure D.2 une note de commentaire pour le cas d'utilisation *ask*). Les modèles peuvent être entrés dans le navigateur grâce à un éditeur ou bien importés depuis un fichier texte. La figure D.3 montre la fenêtre permettant l'édition et la saisie du cas d'utilisation *ask*. Cette fenêtre permet l'ajout et la suppression de paramètres, ainsi que l'édition des contrats. Pour chaque contrat, la

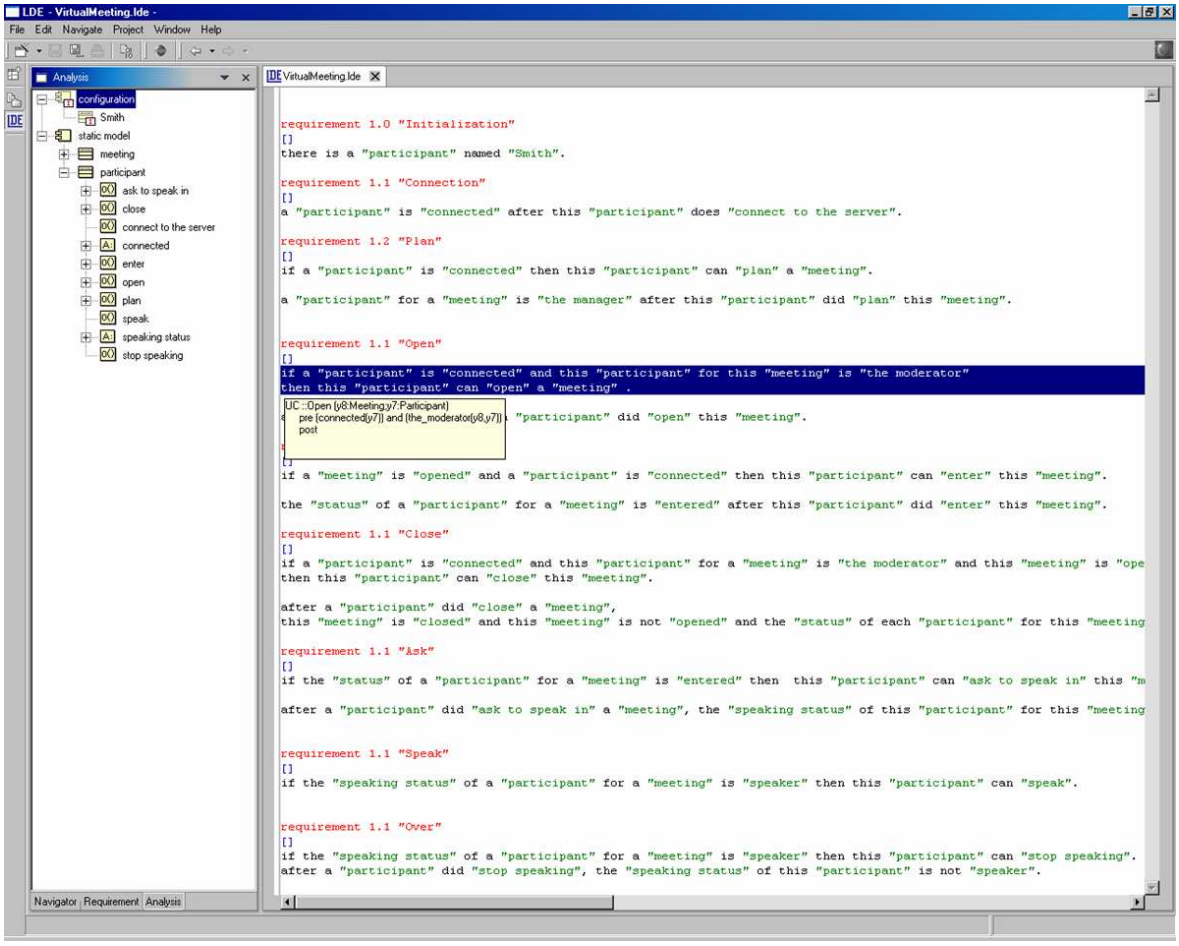


FIG. D.1 – Éditeur d'exigences et navigateur

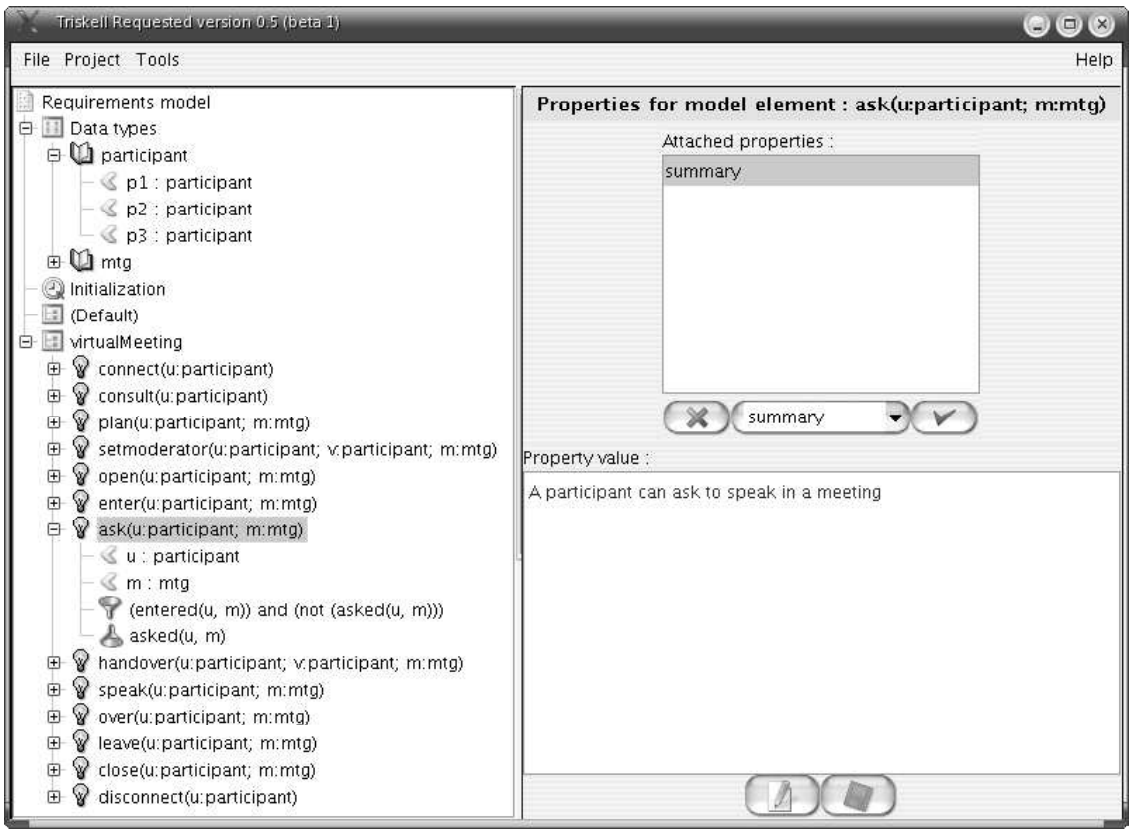


FIG. D.2 – Navigateur de modèles de cas d'utilisation

FIG. D.3 – Édition d'un cas d'utilisation

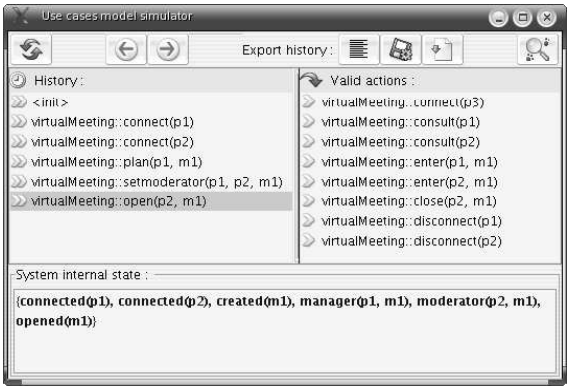


FIG. D.4 – Fenêtre de simulation

syntaxe du langage de contrats (UCL) peut être vérifiée à la demande.

Comme on peut le voir en haut de la partie gauche de la fenêtre, le navigateur fait également apparaître les types de données, c'est-à-dire les instances d'entités métiers définies pour la simulation. Il précise aussi quelle est l'initialisation du système.

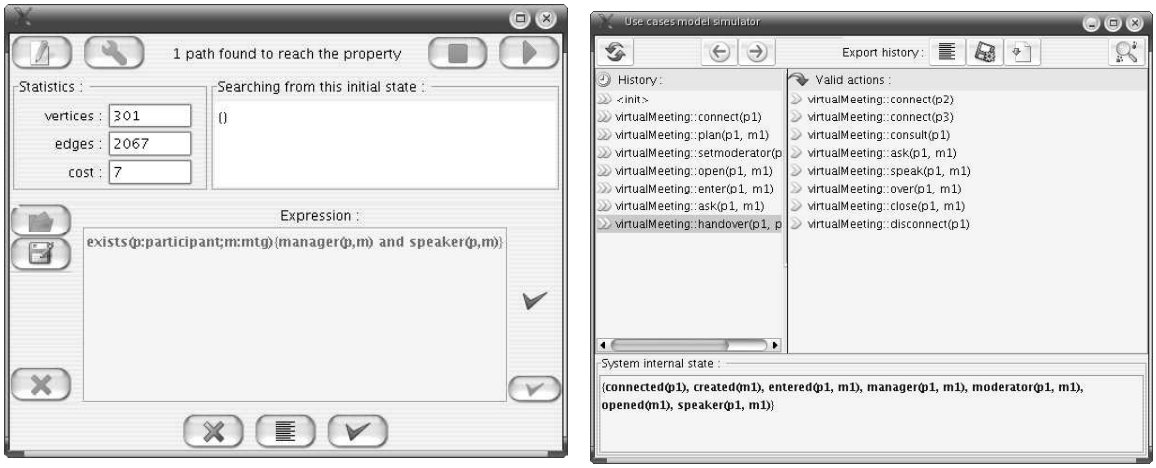
Le simulateur permet ensuite de simuler interactivement les cas d'utilisation présents dans le modèle. La figure D.4 montre la fenêtre de simulation. Le bas de la fenêtre affiche l'état interne du système de simulation. La partie droite propose les actions valides dans cet état. Quand une action est sélectionnée, elle est ajoutée à l'historique de la simulation qui est affiché partie gauche de la fenêtre.

Le simulateur donne également accès à un outil de recherche de propriétés atteignables. La fenêtre correspondante est présentée partie gauche de la figure D.5. Nous avons cherché à vérifier qu'il était possible pour l'organisateur d'une réunion d'y parler. Quand un chemin validant la propriété est trouvé, il peut ensuite être affiché dans le simulateur (ici dans la partie droite de la figure D.5).

D.3 Générateur de tests

Un outil de génération d'objectifs de test est intégré au navigateur de modèle, qui permet de générer des ensembles d'objectifs de tests pour chacun des critères de test que nous avons définis. Un scénario peut être associé à chaque cas d'utilisation, et ainsi des scénarios de test peuvent être générés avec cet outil.

Pour pouvoir utiliser l'approche proposée section 7.1, il faut utiliser un outil externe, les deux outils communiquant par sérialisation des objectifs de test.



(a) Fenêtre de recherche

(b) Affichage du chemin trouvé

FIG. D.5 – Vérification de propriété

Annexe E

Les outils de synthèse de test UMLAUT et TGV

Nous détaillons ici les quatre principales étapes de la synthèse de test avec les outils UMLAUT et TGV.

E.1 Construction d'une interface de simulation

Pré-requis sur le modèle UML. La sémantique d'une spécification UML est définie comme son graphe d'accessibilité sous la forme d'un LTS. La méthode de synthèse de test nécessite comme exigence minimale que le modèle UML contienne un diagramme de classes, une machine à états par classe principale du système, et un diagramme d'objets précisant l'état initial du système.

Une machine à états attachée à une classe spécifie le cycle de vie des objets de cette classe, c'est-à-dire qu'elle décrit comment les objets réagissent quand d'autres objets leur envoient un message, ou quand certains événements se produisent. Les classes ne possédant pas de machine à états ont implicitement un comportement spécifié par une machine à états « en marguerite », c'est-à-dire une machine à un seul état, et avec une transition bouclant sur cet état pour chacune de ses méthodes. La méthode de synthèse nécessitant un système clos, il faut également associer aux acteurs une machine à états, celle-ci pouvant aussi être une marguerite. Le comportement du système est défini par l'exécution combinée des machines à états contenues dans le système et des machines à états modélisant l'environnement du système.

Les machines à états contiennent des actions, qui peuvent être définies sur les transitions ou dans les états. Pour rendre une machine à états UML exécutable, il faut donc pouvoir décrire des actions. Jusqu'à assez récemment, aucune syntaxe particulière n'était préconisée pour les décrire. La version actuelle d'UMLAUT propose d'utiliser le langage *Eiffel* comme langage d'action. Depuis, UML 1.5 [OMG, 2003b] a défini un langage d'action, *Action Semantics*.

Les diagrammes de classes spécifient les configurations possibles des objets dans le système. Les machines à états spécifient comment les configurations peuvent évoluer. Le

troisième élément requis pour rendre un modèle UML simulable est la définition d'une configuration initiale, point de départ de la simulation. Cette configuration initiale est fournie sous la forme d'un diagramme d'objets ou un diagramme de déploiement.

Précompilation de la spécification UML. Pour faciliter la tâche de donner une sémantique formelle à une spécification UML, on la transforme automatiquement en une spécification plus simple, ne comprenant presque que des classes et des méthodes. Les machines à états sont transformées en utilisant une variante du patron de conception État (*State*) [Gamma et al., 1995] : les états sont réifiés et les sous-états sont représentés par une hiérarchie d'héritage. Des files d'événements et un aiguilleur (*dispatcher*) d'événements sont également mis en place pour les objets actifs (les événements sont eux aussi réifiés).

Compilation de la spécification UML. UMLAUT génère une interface de simulation à partir de laquelle un système de transition peut être construit. Cette interface peut alors être utilisée par TGV pour construire à la volée le sous-ensemble du système de transition nécessaire pour la synthèse de test. L'interface de simulation fournit les fonctions suivantes :

- calcul de l'état global initial,
- calcul des transitions tirables dans un état global donné,
- calcul de l'état global successeur, étant donnés un état global origine et une transition tirable à partir de cet état global,
- comparaison des états globaux.

Nous ne détaillons pas ici le mécanisme de génération de cette interface.

Transitions dans le LTS dérivé. Dans l'implémentation actuelle d'UMLAUT, la granularité des transitions est assez grossière : les états globaux correspondent à des configurations « stables » dans le système UML. Mis à part l'état courant des machines à états de chaque état, les autres informations utilisées pour définir l'état global sont les suivantes :

- la valeur des attributs de chaque objet,
- l'état des files de communication entre les objets actifs,
- la structure des liens entre les objets.

On suppose que l'appel aux objets actifs est asynchrone, ce qui implique que chaque transition du LTS est associée à une transition dans la machine à état d'un objet actif. Dans le cas d'une communication asynchrone, une transition d'une machine à états correspond à deux transitions dans le LTS (émission et réception du message).

Des LTS aux IOLTS. Le LTS généré comme décrit précédemment est complété par des informations d'entrée/sortie pour le transformer en IOLTS. Les actions d'entrées (resp. les sorties) du système sont définies comme les émissions (resp. les réceptions) des acteurs externes.

E.2 Construction de l'objectif de test sous forme d'IOLTS

Les objectifs de test de l'approche de synthèse UMLAUT / TGV sont composés d'un ensemble de scénarios dits positifs, et d'un ensemble de scénarios dits négatifs :

- Les scénarios positifs décrivent la spécification du comportement que le concepteur des tests veut tester.
- Les scénarios négatifs spécifient les comportements que le concepteur de tests veut éviter de voir apparaître, ils décrivent quels comportements ne sont pas pertinents pour le cas de test décrit.

Les scénarios négatifs permettent d'une part de spécifier plus précisément l'objectif de test (de manière pratique, il est utile de pouvoir spécifier ce qu'on ne veut pas voir apparaître), et d'autre part d'augmenter l'efficacité de l'outil de synthèse : comme on le verra par la suite, la connaissance de comportements que l'on ne souhaite pas voir apparaître permet de couper des branches dans l'exploration à la volée du graphe comportemental du système sous test.

Tous les scénarios sont supposés décrits dans le langage *O-Tela*, langage de scénario dédié au test. Nous ne détaillerons pas ici le langage *O-Tela* : il s'agit en fait de diagrammes de séquence UML dont la sémantique est donnée non pas en termes de messages mais d'ordres partiels sur les événements, et enrichis d'un certain nombre de constructions pour décrire une plus large gamme de comportements. La description du langage *O-Tela* peut être trouvée dans [Pickin et al., 2001, Pickin, 2003]. Les scénarios peuvent être volontairement incomplets, et notamment ne pas spécifier un certain nombre d'informations comme des types ou des noms d'instances. Les informations manquantes sont symbolisées par le caractère de remplacement « * » (*wildcard*).

Le système de transition étiqueté correspondant à chaque objectif de test est synthétisé de la manière suivante. Supposons que l'objectif de test est composé :

- d'un ensemble de I scénarios positifs ${seq_i^+}_{i\in 1..I}$ définissant les LTSs ${S_i^+}_{i\in 1..I}$
- d'un ensemble de J scénarios négatifs ${seq_j^-}_{j\in 1..J}$ définissant les LTSs ${S_j^-}_{j\in 1..J}$

où les LTSs sont complétés par une boucle sur chaque état vis à vis de l'alphabet A du modèle de l'application : chaque état contient une transition implicite en boucle avec pour étiquette « toute action dans l'ensemble $A \backslash \{$*les autres actions sortant de cet état*$\}$ ». Le LTS d'un tel objectif de test est alors défini comme $\bigcap_{i\in 1..I} S_i^+ \cap \neg \bigcup_{j\in 1..J} S_j^-$, où la négation dénote le complémentaire ensembliste. Les états accepteurs de ce LTS sont les états appelés *accept*, le LTS est complété par les transitions vers l'état appelé *reject*. Un exemple d'objectif de test TGV est donné en partie gauche de la figure 7.15.

E.3 Synthèse du cas de test

La synthèse de test fait appel à l'outil TGV, dont les entrées sont :

- une interface de simulation permettant de construire paresseusement le LTS re-

présentant la sémantique opérationnelle de la spécification du système entier (incluant les acteurs),
- un LTS représentant l'objectif de test,
- la listes des actions internes, des entrées et des sorties.

Les sorties sont un IOLTS représentant le cas de test, avec les verdicts suivants :

- *pass*, qui indique la terminaison des exécutions qui satisfont l'objectif de test,
- *inconclusive*, qui indique la terminaison des exécutions qui ne satisfont pas l'objectif de test mais dont le comportement est cohérent avec la spécification.

Le verdict *fail* est implicite, et est émis à la réception de n'importe quelle entrée non-explicitement spécifiée.

Le cas de test est dérivé en explorant la partie de l'espace d'état de la spécification qui est sélectionnée par l'objectif de test. Cette phase met en jeu le calcul du produit synchrone des deux IOLTS, le calcul de l'automate déterministe correspondant, et l'extraction d'un cas de test comme une image miroir du sous-graphe contrôlable de cet automate déterminisé. L'opération miroir échange les entrées et les sorties pour migrer du point de vue de la spécification vers le point de vue du testeur.

Concrètement, la synthèse de test consiste à construire et explorer à la volée le LTS du système sous test, jusqu'à trouver un chemin compatible avec l'objectif de test. La présence d'objectifs de test négatifs permet de couper certaines branches d'exploration, et en cela augmente l'efficacité du processus de synthèse.

E.4 Des IOLTS aux cas de test

Les IOLTS générés sont ensuite transformés en un scénario écrit dans le langage *Tela,* extension des diagrammes de séquence UML 1.4. Nous ne décrirons ici ni *Tela* présenté dans [Pickin et al., 2001, Pickin, 2003], ni la transformation.

Bibliographie

[Abdurazik and Offutt, 1999] Abdurazik, A. and Offutt, J. (1999). Generating test cases from UML specifications. Technical report, Department of Information and Software Engineering, George Mason University.

[Abdurazik and Offutt, 2000] Abdurazik, A. and Offutt, J. (2000). Using UML collaboration diagrams for static checking and test generation. In *Proc. of the third conference on the Unified Modeling Language*, pages 383–395.

[Abrial, 1996] Abrial, J.-R. (1996). *The B-book : assigning programs to meanings.* Cambridge University Press.

[Aertryck et al., 1997] Aertryck, L. V., Benveniste, M., and Metayer, D. L. (1997). CASTING : A formally based software test generation method. In *Proc. of IC-FEM'97, First IEEE International Conference on Formal Engineering Methods*, pages 101–110.

[Ambriola and Gervasi, 1997] Ambriola, V. and Gervasi, V. (1997). Processing natural language requirements. In *Proc of the International Conference on Automated Software Engineering 1997.*

[Ammann et al., 2003] Ammann, P., Offutt, J., and Huang, H. (2003). Coverage criteria for logical expressions. In *Proc. of the 14th internaltional Symposium on Software Reliability Engineering*, pages 99–107.

[ANTLR, 2004] ANTLR (2004). Antlr parser generator and translator generator home page. www.antlr.org.

[Antoniol et al., 2002] Antoniol, G., Briand, L., Penta, M. D., and Labiche, Y. (2002). A case study using the round-trip strategy for state-based class testing. In *Proc. of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, pages 269–279.

[Arnold, 1992] Arnold, A. (1992). *Systèmes de transitions finis et sémantiques de processus communicants.* Masson. 196 p.

[atelierB, 2004] atelierB (2004). Atelier b. www.atelierB.societe.com.

[Bachmann and Bass, 2001] Bachmann, F. and Bass, L. (2001). Managing variability in software architecture. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability*, pages 126–132.

[Barry Keepence, 1999] Barry Keepence, M. M. (1999). Using patterns to model variability in product families. *IEEE Software*, 16(4) :102–108.

[Baudry et al., 2002a] Baudry, B., Fleurey, F., Le Traon, Y., and Jézéquel, J.-M. (2002a). Automatic test cases optimization using a bacteriological adaptation model. In *Proc. of ASE'02 (Automated Software Engineering)*, pages 253–256.

[Baudry et al., 2002b] Baudry, B., Fleurey, F., Le Traon, Y., and Jézéquel, J.-M. (2002b). Genes and bacteria for automatic test cases optimization in .NET components. In *Proc. of ISSRE'02 (International Symposium on Software Reliability Engineering)*, pages 195–206.

[Beizer, 1990] Beizer, B. (1990). *Software Testing Techniques.* Van Norstrand Reinhold.

[Belina and Hogrefe, 1989] Belina, F. and Hogrefe, D. (1989). The CCITT-specification and description language SDL. *Computer Networks*, pages 311–341.

[Benattou et al., 2002] Benattou, M., Bruel, J.-M., and Hameurlain, N. (2002). Generating test data from OCL specification. In *Proceedings of the ECOOP'2002 Workshop on Integration and Transformation of UML models (WITUML'2002).*

[Bernot et al., 1991] Bernot, G., Gaudel, M.-C., and Marre, B. (1991). Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6(6) :387–405.

[Bertolino et al., 2002] Bertolino, A., Fantechi, A., Gnesi, S., Lami, G., and Maccari, A. (2002). Use case description of requirements for product lines. In *Proc. International Workshop on Requirements Engineering for Product Lines*, pages 12–19.

[Bertolino and Gnesi, 2003] Bertolino, A. and Gnesi, S. (2003). Use case-based testing of product lines. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 355–358.

[Binder, 2000a] Binder, R. (2000a). *Testing object-oriented systems.* Addison-Wesley.

[Binder, 2000b] Binder, R. (2000b). *Testing object-oriented systems*, chapter 8. Addison-Wesley.

[Bjork, 2004] Bjork, R. (2004). http ://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/.

[Bolognesi and Brinksma, 1986] Bolognesi, T. and Brinksma, E. (1986). Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1) :25–59.

[Bosch, 2000] Bosch, J. (2000). *Design and Use of Software Architectures : Adopting and Evolving a Product-Line Approach.*

[Briand and Labiche, 2001] Briand, L. and Labiche, Y. (2001). A UML-based approach to system testing. Technical report, Carleton University.

[Briand and Labiche, 2002] Briand, L. and Labiche, Y. (2002). A UML-based approach to system testing. *Journal of Software and Systems Modeling*, pages 10–42.

[Briand et al., 2003] Briand, L., Labiche, Y., and Wang, Y. (2003). An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(6).

[Budkowski and Dembinski, 1991] Budkowski, S. and Dembinski, P. (1991). An introduction to estelle : A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1) :3–24.

[Bézivin et al., 2003] Bézivin, J., Farcet, N., Jézéquel, J.-M., Langlois, B., and Pollet, D. (2003). Reflective model driven engineering. In *Proceedings of UML 2003, San Francisco*, LNCS. Springer.

[Bühne et al., 2003] Bühne, S., Halmans, G., and Pohl, K. (2003). Modelling dependencies between variation points in use case diagrams. In *Proc. of the 9th International Workshop on Requirements Engineering : Foundation For Software Quality - REFSQ´03.*

[Café, 2003] Café (2003). Projet européen café, itea ip 0004. www.esi.es/en/Projects/Cafe/overview.html.

[Chechik and Wong, 1999] Chechik, M. and Wong, A. (1999). Formal methods when money is tight. In *Proc. of the First Workshop on Economics-Driven Software Engineering Research EDSER-1.*

[Chilenski, 2001] Chilenski, J. (2001). An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical report, FAA.

[Chow, 1978] Chow, T. S. (1978). Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3) :178–187.

[Clarke et al., 2000] Clarke, E., Grumberg, O., and Peled, D. (2000). *Model Checking.* MIT Press.

[Cockburn, 1997] Cockburn, A. (1997). Structuring use cases with goals. *Journal of Object-oriented Programming*, pages 35–40 and 56–62.

[Cockburn, 2001] Cockburn, A. (2001). *"Writing Effective Use Cases"*. Addison Wesley.

[Cueto, 2003] Cueto, F. (2003). Java ftp api. http ://cqs.dyndns.org :81/javaftp/.

[Dick and Faivre, 1993] Dick, J. and Faivre, A. (1993). Automating the generation and sequencing of test cases from modelbased specifications. In *Proc.of the International Symposium Formal Methods Europe (FME'93) : Industrial Strength Formal Methods*, pages 268–284.

[D'Souza and Wills, 1999] D'Souza, D. and Wills, A. (1999). *Objects, Component, ans Frameworks with UML, The Catalysis approach*, chapter Interaction Models : Uses cases, Actions, and collaborations. Addison-Wesley.

[Elmendorf, 1974] Elmendorf, W. R. (1974). Functional analysis using cause-effect graphs. In *in Proceedings of SHARE XLIII*, pages 567–577.

[Families, 2004] Families (2004). Projet européen families, itea ip. www.esi.es/Projects/Families/famOverview.html.

[Fantechi et al., 1994] Fantechi, A., Gnesi, S., Ristori, G., Carenini, M., Vanocchi, M., and Moreschini, P. (1994). Assisting requirement formalization by means of natural language translation. *Formal Methods in System Design*, 4(3) :243–263.

[Fikes and Nilsson, 1971] Fikes, R. and Nilsson, N. (1971). STRIPS : a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4) :189–208.

[Finkelstein and Emmerich, 2000] Finkelstein, A. and Emmerich, W. (2000). The future of requirements management tools. position paper in Information Systems in Public Administration and Law.

[Fleurey, 2003] Fleurey, F. (2003). A framework to trace execution of java programs. url : http ://franck.fleurey.free.fr/JTracor/.

[Fröhlich and Link, 2000] Fröhlich, P. and Link, J. (2000). Automated test case generation from dynamic models. In *Proc. of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*.

[Fuchs et al., 1999a] Fuchs, N., Schwertel, U., and Schwitter, R. (1999a). Attempto controlled english – not just another logic specification language. In *Proceedings of the 8th international workshop on Logic-Based Program Synthesis and Transformation (Selected Papers)*, volume 1559, pages 1–20.

[Fuchs et al., 1999b] Fuchs, N., Schwertel, U., and Torge, S. (1999b). Controlled natural language can replace first-order logic. In *Proc. of Automated Software Engineering 1999*, pages 295–298.

[Fujiwara et al., 1991] Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A. (1991). Test selection based on finite-state machines. *IEEE Transactions On Software Engineering*, 17(6) :591–603.

[Gamma and Beck, 2004] Gamma, E. and Beck, K. (2004). http ://junit.org.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns : elements of reusable object-oriented software*. Addison-Wesley.

[Gargantini and Heitmeyer, 1999] Gargantini, A. and Heitmeyer, C. (1999). Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*.

[Ghosh et al., 2003] Ghosh, S., France, R., Braganza, C., Kawane, N., Andrews, A., and Pilskalns, O. (2003). Test adequacy assessment for UML design model testing. In *Proc. of the 14th International Symposium on Software Reliability Engineering*, pages 332–343.

[Glenford J. Myers, 1976] Glenford J. Myers (1976). *Software Reliability*. John Wiley & Sons, Inc.

[Gomaa and Shin, 2002] Gomaa, H. and Shin, M. (2002). Multiple-view meta-modeling of software product lines. In *Proc . of the 8th International Conference on Engineering of Complex Computer Systems*, pages 238–246.

[Goodenough and Gerhart, 1975] Goodenough, J. and Gerhart, S. (1975). Toward a theory of test data selection. *IEEE Transactions of Software Engineering*, 1(2).

[Griss, 2000] Griss, M. (2000). Implementing product-line features with component reuse. In Springer-Verlag, editor, *Proc. of the 6th international conference of Software Reuse*, pages 137–152.

[Griss et al., 1998] Griss, M., Favaro, J., and d'Alessandro, M. (1998). Integrating feature modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85.

[Halmans and Pohl, 2003] Halmans, G. and Pohl, K. (2003). Communicating the variability of a software-product family to customers. *Software and System Modeling*, 2(1) :15–36.

[Harrold and Rothermel, 1994] Harrold, M. and Rothermel, G. (1994). Performing data flow testing on classes. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 154–163.

[Helke et al., 1997] Helke, S., Neustupny, T., and Santen, T. (1997). Automating test case generation from Z specifications with isabelle. In Bowen, J. P., Hinchey, M. G., and Till, D., editors, *ZUM'97 : The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag.

[Hierons, 1997] Hierons, R. (1997). Testing from a Z specification. *Journal of Software Testing, Verification and Reliability*, 7 :19–33.

[Ho et al., 1999] Ho, W., Jézéquel, J.-M., Le Guennec, A., and Pennaneac'h, F. (1999). UMLAUT : An extendible UML transformation framework. In *Proc. of the 14th IEEE Intenational Conference on Automated Software Engineering (ASE'99)*, pages 275–278.

[IBM, 2004] IBM (2004). Open extensible ide. www.eclipse.org.

[IEEE90, 1990] IEEE90 (1990). IEEE standard glossary of software engineering technology.

[Imperato, 1991] Imperato, M. (1991). *An Introduction to Z.* Chartwell-Bratt.

[Interpréteur OCL, 2002] Interpréteur OCL (2002). Boîte à outils pour OCL. http ://dresden-ocl.sourceforge.net/index.html.

[ITU, 1996] ITU (1996). Message sequence Charts,ITU-TS recommandation Z 120.

[Jacobson et al., 1992] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992). *Object-Oriented Softaware Engineering : A Use Case Driven Approach.* Addison-Wesley.

[Jard, 2002] Jard, C. (2002). Synthesis of distributed testers from true-concurrency models of reactive systems. *Journal of Information and Software Technology.*

[Jard and Jéron, 2002] Jard, C. and Jéron, T. (2002). TGV : theory, principles and algorithms. In *Proc. of the 6th world conference on integrated design and process technology.*

[Jazayeri et al., 2000] Jazayeri, M., Ran, A., and van Der Linden, F. (2000). *Software architecture for Product Families : Principles and Practice.* Addison-Wesley.

[John and Muthig, 2002] John, I. and Muthig, D. (2002). Product line modeling with generic use cases. In *Proc. of SPLC2 workshop on techniques for exploiting commonality through varaibility management.*

[Jézéquel, 1999] Jézéquel, J. (1999). Reifying variants in configuration management. *ACM Transactions on Software Engineering and Methodology*, 8(3) :284–295.

[Kamsties et al., 2003] Kamsties, E., Pohl, K., Reis, S., and Reuys, A. (2003). Testing variabilities in use case models. In *Proc. of the fifth workshop on Product Family Engineering*, LNCS. Springer Verlag.

[Kang et al., 1990] Kang, K., Cohen, S., Hesse, J., Novak, W., and Peterson, S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University.

[Kang et al., 2002] Kang, K., Lee, J., and Donohoe, P. (2002). Feature-oriented product line engineering feasibility (FODA) study. *IEEE Software*, 19(4) :58–65.

[Kantamneni et al., 2002] Kantamneni, H., Pillai, S., and Malaiya, Y. (2002). Structurally guided black box testing. In *Proc of the International Symposium on Software Reliability Engineering.*

[Kim et al., 1999] Kim, Y., Honh, H., Cho, S., Bae, D., and Cha, S. (1999). Test cases generation from UML state diagrams. *IEE Proceedings - Software*, 146(4) :187–192.

[Kruchten, 2000] Kruchten, P. (2000). *Introduction au Rational Unified Process.* Eyrolles.

[Kuusela and Savolainen, 2000] Kuusela, J. and Savolainen, J. (2000). Requirements engineering for product families. In *Proc. of the 2000 International Conference on Software Engineering*, pages 61–69.

[Laprie, 1995] Laprie, J. (1995). *Guide de la sûreté de fonctionnement.* Cépaduès.

[Le Guennec, 2000] Le Guennec, A. (2000). Méthodes formelles avec UML. In *Proc. of CFIP'2000 : Colloque Francophone sur l'Ingénierie des Protocoles.* Hermes.

[Le Guennec, 2001] Le Guennec, A. (2001). *Génie Logiciel et Méthodes Formelles avec UML : Spécification, Validation et Génération de tests.* PhD thesis, École doctorale MATISSE, Universite de Rennes 1.

[Legeard et al., 2002] Legeard, B., Peureux, F., and Utting, M. (2002). Automated boundary testing from Z and B. In *Proc. of Formal Methods Europe (FME'02).*

[Lugato et al., 2002] Lugato, D., Bigot, C., and Valot, Y. (2002). Validation and automatic test generation on uml models : the AGATHA approach. *Electronics notes in Theorical Computer Science*, 66(2).

[Lugato et al., 2004] Lugato, D., Maraux, F., Le Traon, Y., Nebut, C., Normand, V., Dubois, H., Pierron, J.-Y., and Gallois, J.-P. (2004). Automated functional test case synthesis from thales industrial requirements. In *Proc. of the 10th IEEE Real-Time and embedded technology and Applications Symposium (RTAS).*

[Marriott and Stuckey, 2000] Marriott, K. and Stuckey, P. J. (2000). *Programming with Constraints.* MIT Press.

[McGregor, 2001] McGregor, J. (2001). Testing a software product line. Technical report, CMU/SEI.

[Meyer, 1992] Meyer, B. (1992). Applying design by contract. *Computer*, 25(10) :40–51.

[Millo et al., 1978] Millo, R. D., Lipton, R., and Sayward, F. (1978). Hints on test date selection : Help for the practicing programmer. *IEEE Computer*, 11(4) :34–41.

[Millo and Offutt, 1991] Millo, R. D. and Offutt, A. (1991). Constraint-based automatic test data generation. *IEEE Transaction on Software Engineering*, 17(9) :900–910.

[Millo and Offutt, 1993] Millo, R. D. and Offutt, A. (1993). Experimental results from an automatic test case generator. *ACM transactions on Software Engineering and Methodology*, 2(2) :109–127.

[Monestel et al., 2002] Monestel, L., Ziadi, T., and Jézéquel, J. (2002). Product line engineering : Product derivation. In *Workshop on Model Driven Architecture and Product Line Engineering, associated to the SPLC2 conference.*

[Muccini and van der Hoek, 2003] Muccini, H. and van der Hoek, A. (2003). Towards testing product line architectures. In *Proc. of the ETAPS03 workshop "Test and Analysis of Component Based Systems" ("TACOS'03")*, volume 82.

[Murray et al., 1998] Murray, L., Carrington, D. A., MacColl, I., McDonald, J., and Strooper, P. A. (1998). Formal derivation of finite state machines for class testing. In *Proc. of ZUM'98 : the Z Formal Specification Notation*, pages 42–59.

[Nebut et al., 2003a] Nebut, C., Fleurey, F., Le traon, Y., and Jézéquel, J.-M. (2003a). A requirement-based approach to test product families. In *Proc. of the 5th workshop on Product Families Engineering (PFE-05)*, LNCS. Springer Verlag.

[Nebut et al., 2003b] Nebut, C., Fleurey, F., Le traon, Y., and Jézéquel, J.-M. (2003b). Requirements by contracts allow automated system testing. In *Proc. of the 14th. IEEE International Symposium on Software Reliability Engineering (ISSRE'03).*

[Nebut et al., 2002] Nebut, C., Pickin, S., Le Traon, Y., and Jézéquel, J. (2002). Reusable test requirements for UML-modeled product lines. In *Proceedings of REPL'02 (workshop on Requirements Engineering for Product Lines)*, Essen, Germany.

[Nebut et al., 2003c] Nebut, C., Pickin, S., Le traon, Y., and Jézéquel, J.-M. (2003c). Automated requirements-based generation of test cases for product families. In *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE'03).*

[Objecteering, 2004] Objecteering (2004). Atelier de génie logiciel et modeleur UML. www.objecteering.com.

[Offutt and Abdurazik, 1999] Offutt, J. and Abdurazik, A. (1999). Generating tests from UML specifications. In *Proc. of the Second International Conference on Unified Modeling Language. Beyond the Standard (UML'99).*

[Offutt et al., 1999] Offutt, J., Xiong, Y., and Liu, S. (1999). Criteria for generating specification-based tests. In *Proc. of the Fifth IEEE International Conference on Engineering of Complex Systems.*

[OMG, 2003a] OMG (2003a). OCL. http ://www.omg.org/docs/ptc/03-08-08.pdf.

[OMG, 2003b] OMG (2003b). Unified Modeling Language Specification, version 1.5. http ://www.omg.org/docs/formal/03-03-01.pdf. in Object Management Group.

[OMG, 2004] OMG (2004). http ://www.omg.org/technology/documents/formal/mof.htm.

[OMG - U2 partners, 2001] OMG - U2 partners (2001). UML2.0 Proposal v.0.671. http ://www.u2-partners.org/artifacts.htm/.

[Ostrand and Balcer, 1988] Ostrand, T. and Balcer, M. (1988). The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6) :676–686.

[Pargas et al., 1999] Pargas, R., Harrold, M., and Peck, R. (1999). Test-data generation using data-fow information. *Journal of Software Testing, Verifications, and Reliability*, 9 :263–283.

[Parnas, 1976] Parnas, D. (1976). On the design and development of program families. *IEEE Transactions on Software Engineering.*

[Pickin, 2003] Pickin, S. (2003). *Test et composants logiciels pour les télécomunications.* PhD thesis, Université de Rennes 1.

[Pickin et al., 2001] Pickin, S., Jard, C., Heuillard, T., Jézéquel, J.-M., and Desfray, P. (2001). A UML-integrated test description language for component testing. In *Proc. UML2001 wkshp : Practical UML-Based Rigorous Development Methods*, GI-Edition - Lecture Notes in Informatics (LNI). Bonner Köllen Verlag.

[Pickin et al., 2002] Pickin, S., Jard, C., Le Traon, Y., Jéron, T., Jézéquel, J.-M., and Le Guennec, A. (2002). System test synthesis from UML models of distributed software. In *Proc. of the 22nd Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02)*, Houston, Texas.

[Rapps and Weyuker, 1985] Rapps, S. and Weyuker, E. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4) :367–375.

[Riebisch et al., 2002a] Riebisch, M., Böllert, K., Streitferdt, D., and Philippow, I. (2002a). Extending feature diagrams with uml multiplicities. In *Proceedings of the 6th Conference on Integrated Design and Process Technology.*

[Riebisch et al., 2002b] Riebisch, M., Philippow, I., and Götze, M. (2002b). UML-based statistical test case generation. In *Proc. of the Onternational Conference NetObjectDays, NODe'02*, volume 2591, pages 394–411.

[Ryser et al., 1998] Ryser, J., Berner, S., and Glinz, M. (1998). On the state of the art in requirements-based validation and test of software. Technical report, Institut für Informatik, University of Zurich.

[Ryser and Glinz, 1999] Ryser, J. and Glinz, M. (1999). A scenario-based approach to validating and testing software systems using statecharts. In CNAM, editor, *Proc. 12th International Conference on Software and Systems Engineering and their Applications.*

[Ryser and Glinz, 2000] Ryser, J. and Glinz, M. (2000). Using dependency charts to improve scenario-based testing. In *Proceedings of the 17th International Conference on Testing Computer Software (TCS2000).*

[Show and Garlan, 1996] Show, M. and Garlan, D. (1996). *Software architecture : Perspectives and on an emerging Discipline.* Prentice hall.

[Soley and OMG, 2000] Soley, R. and OMG (2000). Model-driven architecture. OMG document.

[Souter and Pollock, 2001] Souter, A. and Pollock, L. (2001). Contextual def-use associations for object aggregation. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 13–19.

[Souter et al., 1999] Souter, A., Pollock, L., and Hisley, D. (1999). Inter-class def-use analysis with partial class representations. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 47–56.

[Stidolph and Whitehead, 2003] Stidolph, D. and Whitehead, J. (2003). Managerial issues for the consideration and use of formal methods. In *Proceedings of the Twelfth International Formal Methods Engineering Symposium (FME 2003)*, pages 170–186.

[Svahnberg and Bosch, 2001] Svahnberg, M. and Bosch, J. (2001). Issues concerning variability in software product lines. In *Proc. of the International Workshop IW-SAPF-3*, volume 1951. Lecture Notes in Computer Science.

[Tahat et al., 2001] Tahat, L., Vaysburg, B., Korel, B., and Bader, A. (2001). Requirement-based automated black-box test generation. In *Proc. of the 25th Annual International Computer Software and Applications Conference.*

[Thalès et al., 2003] Thalès, INRIA, and CEA (2003). Programme de recherche CARROLL. www.carroll-research.org.

[Triskell, 2004] Triskell (2004). Model transformation language. http ://www.irisa.fr/triskell.

[UMLAUT, 2002] UMLAUT (2002). UMLAUT, Unified Modeling Language All pUrposes Transformer. http ://www.irisa.fr/UMLAUT/.

[van Gurp et al., 2001] van Gurp, J., Bosch, J., and Svahnberg, M. (2001). On the notion of variability in software product lines. In *Proceedings 2nd Working IEEE / IFIP Conference on Software Architecture (WICSA)*, pages 45–54.

[Vilkomir and Bowen, 2001] Vilkomir, S. and Bowen, J. (2001). Formalization of software testing criteria using the Z notation. In *Proc. COMPSAC 01 : 25th IEEE Annual International Computer Software and Applications Conference*, pages 351–356. IEEE Computer Society.

[Warmer and Kleppe, 1998] Warmer, J. and Kleppe, A. (1998). *The Object Constraint Language : Precise Modeling with UML.* Addison-Wesley.

[Weidenhaupt et al., 1998] Weidenhaupt, K., Pohl, K., Jarke, M., and Haumer, P. (1998). Scenario usage in system development : A report on current practice. *IEEE Software.*

[Weiss and Lai, 1999] Weiss, D. and Lai, C. (1999). *Software Product-Line Engineering : A Family-based Software Development Process.* Addison Wesley.

[Weyuker, 1979] Weyuker, E. (1979). Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing*, 8 :587–598.

[Williams, 1999] Williams, C. (1999). Software testing and the uml. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'99).*

[Xanthakis et al., 2000] Xanthakis, S., Régnier, P., and Karapoulios, C. (2000). *Le test des logiciels.* Hermès science publications.

[Ziadi et al., 2002] Ziadi, T., Hélouët, L., and Jézéquel, J.-M. (2002). Modeling behaviors in product lines. In *Proceedings of REPL'02 (workshop on Requirements Engineering for Product Lines)*, Essen, Germany.

[Ziadi et al., 2003] Ziadi, T., Hélouët, L., and Jézéquel, J.-M. (2003). Toward a UML profile for software product lines. In *Proceedings of the Fifth Internationl Workshop on Product Familly Engineering (PFE-5)*, LNCS. Springer Verlag.