Habilitation à Diriger des Recherches

présentée à

L'UNIVERSITÉ D'AUVERGNE

spécialité : Informatique

par

Sébastien SALVA

Étude sur le test basé modèle de systèmes temporisés et d'applications à base de composants.

Soutenue à Aubière le 18 mai 2012 devant la Commission d'Examen formée de :

Président de jury :

Alain Quilliot, Professeur à l'Université Blaise Pascal

Rapporteurs :

Pr. Rachida Dssouli, Professeur à CIISE, Concordia University,

Pierre de Saqui-Sannes, Professeur à l'ISAE,

Richard Castanet, Professeur à l'Université de Bordeaux,

Examinateurs :

Jean-Marc Lavest, Professeur à l'Université d'Auvergne,

Michel Misson, Professeur à l'Université d'Auvergne.

-2011-

A ma mère, qui nous a quittée bien trop tôt.

Remerciements

En préambule de ce mémoire, il semble opportun de remercier toutes les personnes qui m'ont apporté leur soutien et qui ont contribué à la réalisation de ce mémoire. Donc, que tous ceux qui m'ont permis d' être là reçoivent le témoignage de ma gratitude.

Je remercie tout d'abord les membres du jury, qui me font l'honneur de juger ce travail. Rachida Dssouli, Pierre de Saqui-Sannes et Richard Castanet ont accepté d'être rapporteurs, et ainsi de se plonger dans une version non finale du manuscrit. Jean-Marc Lavest, Michel Misson et Alain Quilliot, enfin, ont pris le temps d'examiner ce document, et ainsi de poser un autre regard sur ce travail. Je sais gré à chacun d'eux du temps qu'il m'a consacré.

Ma gratitude va ensuite, aux personnes que j'ai le plus souvent côtoyées au Laboratoire LIMOS, notamment Alain Quilliot qui m'a permis de joindre l'utile à l'agréable via des missions à l'étranger, et qui m'a guidé vers de bonnes opportunités, Michel Misson pour ses conseils avisés et pour ses bonnes remarques. Un grand merci va à Béatrice Bourdieu, qui m'a "dépannée" plus d'une fois lors de problèmes divers et administratifs. Je ne peux évidemment pas oublier mon camarade Patrice Laurençot avec qui j'ai crée des liens d'amitié et avec lequel j'ai partagé une partie de ma passion pour la recherche, des moments de galère, et bien d'autres choses encore.

Une pensée particulière est adressée aux membres du projet Webmov (et membres de LRI, GET/INT, LaBRI, SOFTEAM, Montimage) avec qui j'ai partagé des moments de travail certes mais également des moments de bonne camaraderies. Je pense également à Hacène Fouchal, mon ancien responsable de thèse, qui a continué à m'apporter des remarques ces dernières années.

Je tiens a remercier avec démesure tous mes collègues de l'IUT d'Aubière et en particulier du département informatique, pour leurs bonnes humeurs, leur soutien et pour l'ambiance collégiale qui y règne. C'est ce département qui m'a persuadé que sans enseignement, il n'y a pas de recherche possible.

J'adresse mes plus sincères remerciements à ma famille, mes parents et tous mes proches et amis pour leur soutien et leur patience.

Ce chaînage de remerciements serait bien incomplet si je n'envoyais pas mes dernières pensées à mes enfants Manon, Alexi, Maxence et à ma femme Nathalie pour leur patience, leur amour, et la correction du mémoire (le deuxième que ma femme subit).

Table des matières

Table des figures

Préface

Le test en général est aujourd'hui plutôt bien intégré dans le monde industriel, que ce soit pour la conception de systèmes critiques complexes, ou pour le développement d'applications classiques (sites Web, services, etc.). J'ai déjà pu observer, durant ma carrière, de profonds changements d'habitude qui se manifestent de nombreuses façons. Par exemple, il y a dix ans, les doctorants, présentant leurs travaux, commençaient souvent par démontrer les besoins du test en citant des catastrophes liées à une panne logicielle quelconque comme l'explosion d'un pipeline de gaz soviétique (1982), l'explosion de la fusée Ariane 5 (1996), le bug dans la division des flottants sur les processeurs Intel Pentium-II (1993), ou la mort de patients à l'institut de cancérologie de Panama (2000) [GAR05].

Aujourd'hui et depuis peu, les grands groupes industriels comme les TPE voient tout leur intérêt dans le génie logiciel, regroupant des domaines tels que la modélisation des besoins, les spécifications, la gestion de projet, les tests, etc. Ceci favorise les rapprochements entre le monde industriel et les chercheurs et ouvre vers des besoins ciblés et des problématiques nouvelles. Par exemple, on peut citer le test automatique qui voit son attrait en augmentation, mais aussi le test d'applications Web, de services Web, la génération de tests à partir de spécification UML, etc.

Ce changement peut se deviner à travers ce manuscrit en observant mes premiers travaux qui portaient sur le test de systèmes temporisés modélisés, puis les derniers travaux traitant majoritairement des composants (services Web) et compositions, et les travaux futurs basés sur les compositions dynamiques, le Cloud computing, et plus généralement sur les environnements de déploiement des applications dont l'accès est restreint.

Ce manuscrit se présente sous la forme de plusieurs chapitres décrivant ces travaux. Classiquement, le premier chapitre introduit le test du logiciel, et décrit des notations, des définitions et des modèles utilisés dans les chapitres suivants.

Le chapitre 1.4 synthétise les travaux effectués durant ces neuf dernières années après la fin de ma thèse de doctorat. Ceux-ci traitent de divers points tels que le test de systèmes

temporisés, de systèmes réactifs, d'applications Web (services, compositions BPEL, etc.), etc.

Le chapitre 2.4.3 décrit plusieurs projets de recherche, certains étant une suite logique des travaux effectués, d'autres projets étant liés à des collaborations avec des industriels. Les perspectives décrites s'orientent vers la prise en compte d'environnement partiellement ouverts dans le test, le test passif par proxy ou par objets basés sur le patron "délégué", le test de compositions dynamiques et d'applications mobiles dans des environnements hétérogènes.

Enfin, le dernier chapitre se rapporte à un CV détaillé.

Chapitre 1

Introduction

En guide d'introduction, ce chapitre présente le thème des travaux décrits dans ce manuscrit, à savoir le test formel (model based testing). Nous rappelons brièvement ce que sont le cycle de vie d'un système ou logiciel, les types de test et les approches de test classiques utilisées de près ou de loin dans ce manuscrit. Nous donnerons également quelques définitions et notations de modèles utilisés par les méthodes décrites dans les chapitres suivants. Nous terminerons par une présentation de la notion de testabilité, utilisée également par quelques travaux.

1.1 Cycle de vie du logiciel

Le cycle de vie de logiciel est un modèle de développement composé entre-autre de méthodes de conception, d'analyse de codage et de test. Ces méthodes permettent de produire une implantation conforme à un cahier des charges initial, ce dernier réunissant

des informations informelles sur le fonctionnement voulu du produit fini. Le cycle de vie peut se présenter sous différentes formes. Classiquement, il est représenté sous forme de cascades avec des phases théoriquement séquentielles : chacune exploite le résultat délivré par la phase précédente, le traite et l'offre à la phase suivante. C'est d'ailleurs sous cette forme globale que le cycle de vie est repris dans le modèle de référence CMMI (Capability Maturity Model + Integration [cmm]) qui est suivi par plusieurs sociétés de développement informatique comme ATOS-Origin, SQLI ou BNP Parisbas. Ce modèle mesure et évalue une *maturité* jusqu'à 5 niveaux en rapport aux activités menées par une organisation. Notamment, le processus CMMI-DEV 3, qui intègre le développement de systèmes et logiciels, décrit ce cycle par la Figure 1.2.

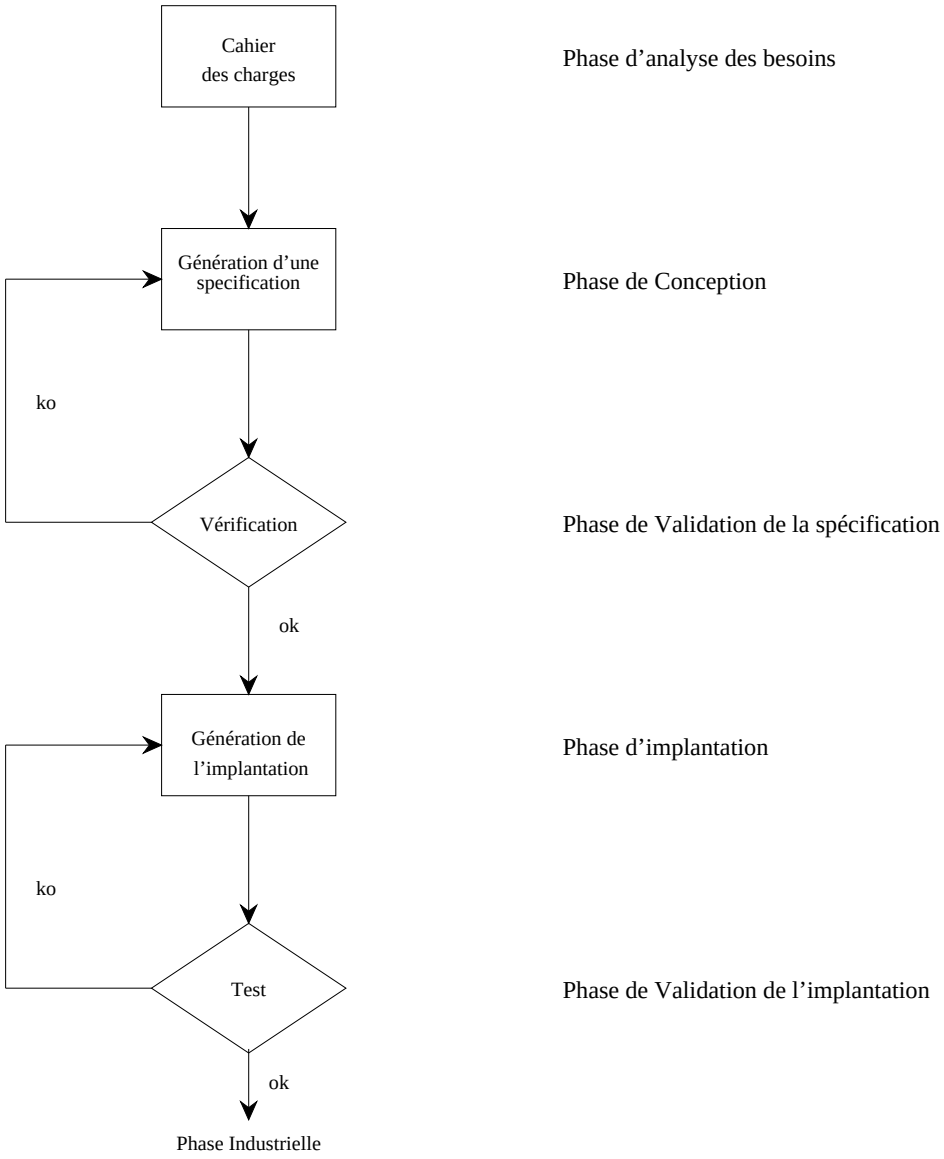


FIGURE 1.1 – Cycle de vie du logiciel

Ainsi, ce cycle peut être vu comme un processus de raffinement permettant de passer d'une spécification décrite avec un haut niveau d'abstraction à un produit final conforme à

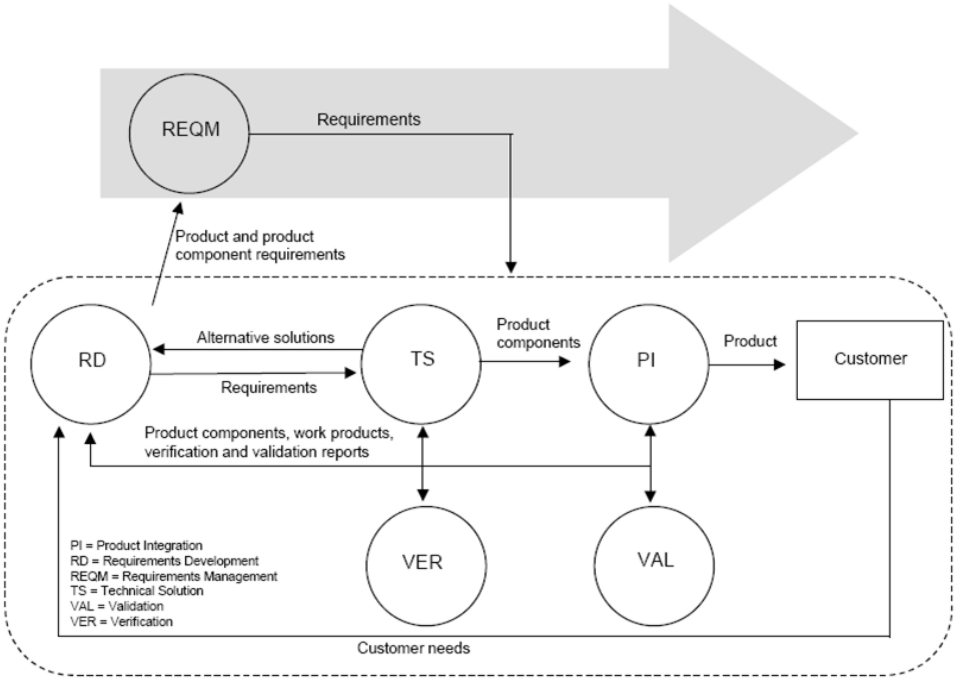


FIGURE 1.2 – Cycle de vie dans CMMI-DEV 3

ce que le concepteur ou client attendait. Lors de chaque phase, divers critères du produit, tels que la complexité, la performance, la robustesse, la facilité à tester, peuvent être analysés avant l'implantation du produit, afin de juger si ces critères sont suffisants ou non. Ainsi, à tout moment, il est possible de connaître la qualité du système.

Les différentes phases du cycle de vie sont détaillées comme suit :

- **Phase d'analyse des besoins :** elle permet l'analyse des besoins qui sont ensuite réunis dans un cahier des charges. La description de ces besoins est généralement exprimée en langage naturel.
- **Phase de conception :** le langage naturel reste ambigu et difficilement analysable. Cette phase vise alors à produire une spécification détaillée et parfois formelle à l'aide de langages de description formelle (FDT Formal Description Technique). Grâce à une suite de raffinements, la spécification décrit en détail les structures de données, l'architecture du système et l'interface liant ces différents modules.
- **Phase de Validation de la spécification :** cette opération, aussi appelée Vérification, a pour but de vérifier la cohérence de la spécification. Plus précisément, elle assure une correction et vérifie certaines propriétés comme l'existence des services proposés, l'admissibilité de ces services et leurs bons fonctionnements. Un certain nombre d'erreurs dues à la conception de la spécification peuvent ainsi être détectées.
- **Phase d'implantation :** la spécification détaillée et vérifiée est, dans cette étape, traduite sous forme de langage de programmation exécutable. Cette transformation tient compte de plusieurs paramètres tels que l'environnement du système, les interactions avec celui-ci...
- **Phase de Validation de l'implantation :** cette étape aussi appelée la phase de Test,

permet de vérifier que l'implantation satisfait un certain nombre de propriétés de la spécification. Elle permet donc la détection de défectuosités du fonctionnement de l'implantation du système. Ces tests peuvent être divers : ils peuvent s'attacher aux performances, à la robustesse, ainsi qu'à la conformité de l'implantation par rapport à sa spécification.

Le cycle peut aussi être décrit sous d'autres formes, e.g., en V ou en spirales. C'est cette dernière forme que l'on retrouve notamment dans les méthodes agiles qui semblent être à la mode en ce moment. Ces cycles ont été proposés essentiellement pour pallier au problème de réactivité du modèle en cascade vis-à-vis de la gestion des risques ou de la phase de modélisation qui peut être assouplie en l'effectuant de façon itérative. On retrouve dans ces cycles les mêmes phases que précédemment mais découpées en plusieurs morceaux et faites en continue dans la vie d'un projet. Si l'on prend comme exemple la méthode agile SCRUM, illustrée en Figure 1.3, un projet sera découpé en plusieurs tâches faites en boucles pouvant produire d'autres tâches. Ces tâches peuvent concerner la modélisation d'une partie de l'application, son développement, son test unitaire ou le test de régression de l'application.

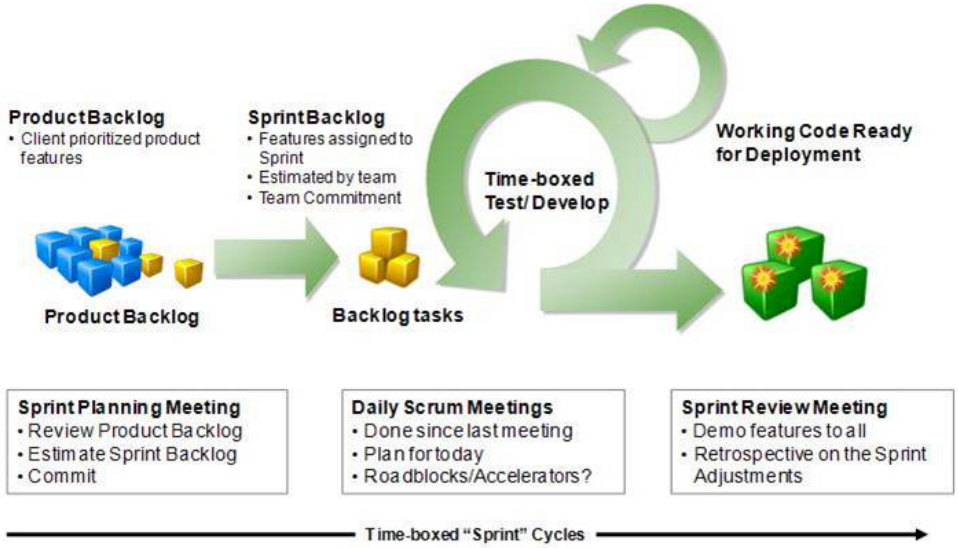


FIGURE 1.3 – Illustration synthétique de la méthode agile SCRUM

1.2 Les tests

Les phases de test ont été définies de différentes façons dans la littérature. Par exemple, Myers définit ce test comme étant *"le processus d'exécution d'un programme dans le but de trouver des erreurs"* [Mye79]. Selon Abran et al. [AW04], le test est *"une activité effectuée pour évaluer la qualité d'un produit, et pour l'améliorer, en identifiant les défauts et problèmes"*. Dans tous les cas, l'affirmation de Dijstra suivante *"Testing shows the presence, not the absence of bugs"* [Dij69] reste commune.

On remarque aujourd'hui que ces phases de test sont relatives à plusieurs sous activités importantes qui se répandent de plus en plus de façon pervasive dans le monde industriel. Ainsi, il semble pertinent de parler non pas du test mais bien des tests, ceux-ci pouvant être classés dans plusieurs catégories qui ne sont pas exclusives mais qui se complètent. Plusieurs catégories ont été proposées dans la litérature [UL07, AO08]. Nous en présentons quelques-unes ci-dessous :

- **granularité :** cette catégorie groupe les tests que l'on trouve souvent en entreprise et qui expriment la granularité utilisée par un test vis-à-vis de l'ensemble d'un projet. Ainsi, nous trouvons le test unitaire, d'intégration, fonctionnel de l'application et d'acceptation (recette),
- **accessibilité :** ce groupe rassemble les approches en boite blanche, grise et noire. Les tests en boite blanche aussi appelés tests structurels, sont effectués sur des systèmes dont la structure interne est connue et observable. Citons par exemple le test de boucle qui vise à valider toutes les boucles d'un programme en détectant les erreurs d'initialisation, d'indexage et de conditions d'arrêt. Les tests en boîte noire sont appliqués sur des systèmes ou applications où la structure interne est inconnue. Les interactions avec le système sont effectuées à travers des interfaces reliant le système avec l'environnement extérieur. Les tests en boites grises sont des tests où des parties de l'implantation sont observables et utilisables,
- **activité :** les tests peuvent être actifs ou passifs. Les tests actifs consistent à expérimenter une implantation sous test à partir de séquences de propriétés appelées *cas de test*. Une grande majorité des tests sont actifs car ils permettent d'obtenir un verdict de test "sans attendre". Cependant, ils peuvent stimuler une implantation de façon inhabituelle et surtout ils sont coûteux en temps et en génération. Le test passif "espionne" une implantation, grâce à un module qui extrait ses réactions et qui détermine si ces dernières correspondent à ce que l'on a dans la spécification. Le test passif permet une publication rapide d'une application ou d'un système, et permet de ne pas interrompre le fonctionnement normal d'une implantation de façon arbitraire,
- **caractéristique :** cette catégorie rassemble les tests effectués suivant la caractéristique testée. De façon non exhaustive, on trouve le test de robustesse, de performance, de sécurité, de l'usager ou de conformité :
  - Tests de Performance : ont pour but de vérifier un certain nombre de paramètres liés aux performances du système, tels que le débit, le temps de réponse, le nombre de connections. D'une manière simpliste, la performance est mesurée en soumettant le système à certaines conditions d'utilisation et en observant ses réactions. Ces conditions d'utilisation peuvent être minimales, moyennes ou extrêmes,
  - Tests de Robustesse : consistent à vérifier la réaction d'un système dans des conditions d'utilisations extrêmes ou bien son exécution dans un environnement dit hostile. Ces conditions ne sont généralement pas prévues dans la spécification, cette dernière référençant des conditions de fonctionnement normales. Ces tests permettent ainsi de vérifier si d'autres erreurs existent telles que des fraudes ou des erreurs d'utilisation du système.
  - Tests d'Interopérabilité : vérifient si le système développé interagit d'une façon correcte avec d'autres systèmes extérieurs en observant les fonctionnements des différents systèmes et des communications engendrées. Ces tests permettent de

vérifier un service plus global fourni aux utilisateurs.

- Tests de l'usager : sont effectués au niveau de l'usager qui manipule le système pour vérifier si ce dernier répond bien à ses besoins. Ces tests, parfois appelés beta-tests, permettent de vérifier les services les plus demandés.
- Tests de Conformité : occupent une place prépondérante dans le domaine de l'ingénierie des protocoles et du logiciel. Ils ont de ce fait été normalisés ISO dans [ISO91a]. Ceux-ci consistent à vérifier que le comportement de l'implantation est conforme à la spécification de référence. Souvent, les méthodes de test de conformité sont actives.

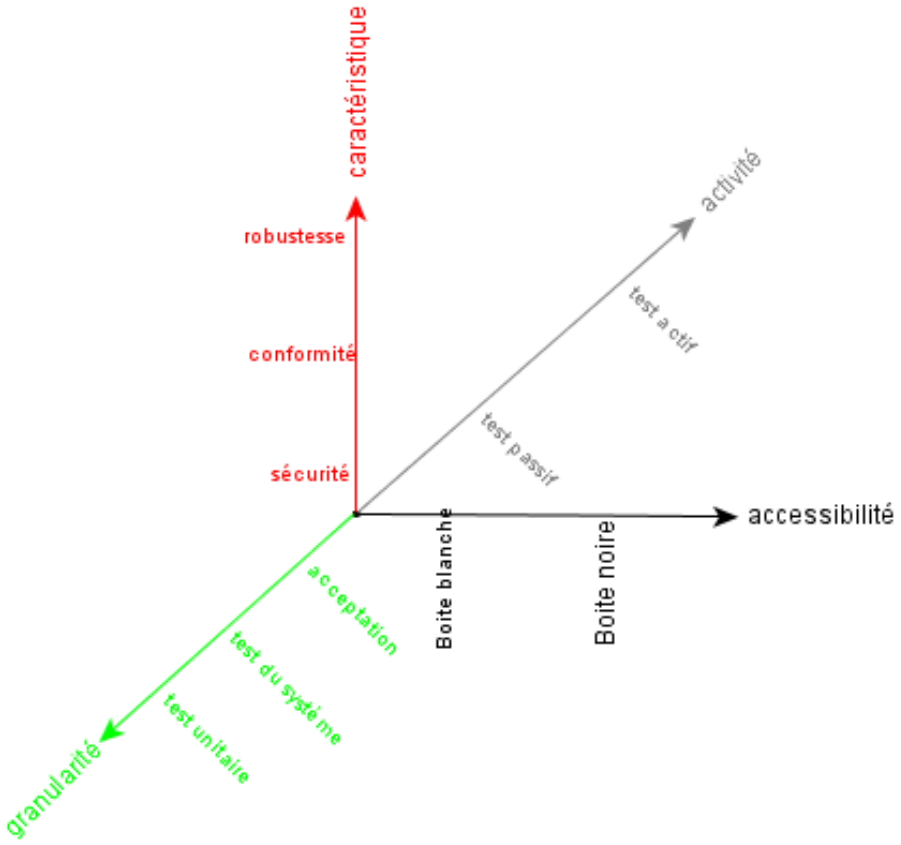


FIGURE 1.4 – Catégories de tests

Dans ce mémoire nous nous focaliserons sur le test en boite noire, avec une granularité des applications étudiées qui va du composant jusqu'aux systèmes et compositions. Nous décrirons des méthodes actives et passives pour tester la conformité, la robustesse ou la sécurité.

Afin d'augmenter l'efficacité des méthodes de test et/ou de formaliser cette efficacité, les tests sont très souvent construits à partir d'un modèle de référence, qui décrit le comportement attendu de l'implantation. On parle alors de *model-based testing*. Selon [UL07], ce processus peut se résumer en cinq étapes :

1. la formalisation du comportement attendu grâce à un modèle,

2. la génération de cas de test depuis le modèle vis-à-vis de contraintes imposées par des objectifs de test exprimant des scénarios ou des critères de test,
3. la concrétisation des cas de test de telle sorte qu'il peuvent être exécutés sur l'implantation,
4. l'exécution des cas de test sur l'implantation,
5. l'analyse des résultats pour identifier des différences entre le modèle et l'implantation.

L'analyse des résultats peut être faite de différentes façons : en observant des bugs, par rapport à un modèle de faute rassemblant l'ensemble des fautes qui peuvent être détectées. Dans ce cas, des cas de test sont construits afin de trouver des fautes sur l'implantation vis-à-vis de ce modèle.

Une autre solution est basée sur la définition initiale d'une relation entre un modèle et une implantation sous test. Cette dernière, appelée relation de test, décrit de façon non ambiguë ce qu'est une implantation correcte vis à vis du modèle (spécification). Pour vérifier cette relation, des cas de test T sont construits à partir d'une spécification S par expérimentation de l'implantation I. On définit ensuite la relation *passe*, qui décrit quand une implantation accepte un cas de test. Cette relation est souvent construite suivant les réactions que l'on observe en expérimentant des cas de test. Puis, on déduit que la relation de test est vérifiée suivant les résultats obtenus. Pour une relation *implante* ceci peut s'écrire par :

$$I \text{ implante } S \Leftrightarrow \forall t \in T, I \; passe \; t$$

D'une telle relation, on peut déduire deux implications définies dans [Pel96] : l'exactitude (soudness) qui implique que toutes les implantations correctes et quelques incorrectes passent les tests, et l'exhaustivité (exhaustiveness) qui signifie que toutes les implantations incorrectes sont détectées :

$$T \text{ est exact ssi } \forall I, I \text{ implante } S \Rightarrow \forall t \in T, I \; passe \; t$$

$$T \text{ est exhaustif ssi } \forall I, I \text{ implante } S \Leftarrow \forall t \in T, I \; passe \; t$$

Plusieurs relations de test, fondées sur la relation générale précédente, ont été proposées suivant le type de test à effectuer et le modèle. On peut par exemple citer la bissimulation [MIL80b], ou le test par refus [Phi87] et bien entendu *ioco* [Tre96c] qui est une relation de test dédiée aux LTSs.

Dans la suite, nous donnerons les définitions de quelques modèles et décrirons quelques méthodes de test utilisées dans la suite de ce mémoire.

1.3 Définition de quelques Modèles et méthodes de test

1.3.1 Définition de quelques Langages de descriptions de spécifications

Diverses modélisations sont proposées pour traduire un cahier des charges, représentant une spécification, en une description plus détaillée et plus formelle. Le but de cette

traduction en un langage de description, aussi appelé Technique de Description Formelle (TDF), est de lever toute ambiguïté sur la spécification, et permet aussi d'y appliquer plus facilement des analyses, des algorithmes et notamment des méthodes de test.

Pour décrire avec précision une spécification, un tel langage doit être construit sur des règles strictes, puis doit être normalisé pour garder une unicité de traduction. Les langages de description formelle actuellement normalisés et définis peuvent être placés dans deux catégories :

- La première rassemble des langages que nous qualifions de haut niveau, car ceux-ci modélisent une spécification avec un degré d'abstraction élevé. Certains de ces langages sont les logiques temporelles, LOTOS, ESTELLE, LDS et les Réseaux de Petri. Dans cette catégorie, il est difficile de ne pas citer UML (Unified Modeling Language) qui est un langage graphique permettant de représenter le fonctionnement d'un système ou logiciel grâce à 13 types de diagrammes. Parmi les plus connus et utilisés, nous pouvons citer le diagramme de cas d'utilisation, d'activités, de séquences, de classes, etc.,
- La deuxième rassemble des langages que nous qualifions de bas niveau. Ils peuvent être obtenus automatiquement depuis un des langages précédents, et décrivent de façon plus détaillée mais plus lourde la même spécification. Étant d'un faible degré d'abstraction, ils ne peuvent décrire, qu'avec difficulté, des notions telles que le processus ou les priorités. Ce sont ces modèles sémantiques qui sont généralement utilisés par les méthodes de test. Parmi les plus connus, citons la Machine à États Finis(FSM), les Systèmes de Transitions Étiquetées (LTS), les automates temporisés ou les systèmes symboliques à transition (STS).

Définissons quelques modèles qui seront utilisés dans la suite de ce mémoire.

Modèle Relationnel

Le modèle relationnel, décrit dans [Tar41, Mil90a] définit une spécification comme étant composée d'un espace E de variables manipulées par un programme et d'une relation R définie sur le domaine (ensemble des symboles d'entrée) et l'image (ensemble des symboles de sortie).

Une spécification S, décrite grâce à une relation $R : I \rightarrow O$, avec I l'ensemble des symboles d'entrée de cette relation et O l'ensemble des symboles de sortie, permet de représenter une spécification à un niveau d'abstraction supérieur aux automates : en effet, seuls les symboles sont mis en évidence. Ainsi, utiliser une relation permet de décrire une spécification comme une boîte de laquelle sont émis et reçus des symboles, ce qui est très proche de la représentation de l'implantation, étant donné qu'elle est vue par le test comme une boîte noire.

Machine à états finis

Une machine à états finis est un graphe décrivant les actions de la spécification. Les états de cet automate décrivent les états internes du système. Un ensemble de symboles d'entrée et un ensemble de symboles sortie sont définis pour modéliser respectivement les signaux émis vers la spécification et reçus depuis cette dernière. Comparée au modèle

relationnel, cette représentation détaille la spécification en illustrant ses états ainsi que les interactions qu'il est possible d'effectuer.

Une FSM (Finite State Machine) est un graphe défini comme suit :

Définition 1.3.1 (FSM) *Une FSM A est un quintuplet $< s_0, S, I, O, T >$ où :*

$S_0 \in S$ est l'état initial,

S est un ensemble non vide d'états,

I est l'ensemble des entrées,

O est l'ensemble des sorties,

T est un ensemble non vide de transitions tel que $T \in S \times I \times O \times S$. Le tuple <s,i,o,s'> représente une transition qui part de l'état s et va vers l'état s'. Celle-ci est étiquetée par une entrée et une sortie qui symbolisent le fait que pour passer à l'état s', il est nécessaire d'envoyer l'interaction i et de recevoir l'interaction o. Une telle transition est communément notée par $s \xrightarrow{i/o} s'$.

Une FSM $A =< s_0, S, I, O, T >$ est déterministe ssi :
$\forall s \in S$, $\forall t = s \xrightarrow{A/B} s' \in T$ avec $A \in I, B \in O, s' \in S$, $\forall t' = s \xrightarrow{C/D} s'' \in T$, avec $C \in I, D \in O, s'' \in S$, $A = C$ implique $B = D$ et $s' = s''$.

Système de transition étiqueté (LTS)

Le modèle de la machine à états finis, bien que très utilisé comporte une grande restriction quant à la description d'une spécification. En effet, il est impossible de séparer émission d'un symbole d'entrée (stimulus) et réception d'un symbole de sortie (observation). Or, bien que le fait de combiner sur une seule transition les symboles d'entrée et de sortie simplifie les calculs et les méthodes de test, il serait intéressant de pouvoir modéliser un protocole de communication, ou plus globalement une application, grâce à un automate composé de transitions étiquetées par un symbole unique.

Définition 1.3.2 *Système de Transitions Étiquetées (LTS)*

Un système de transitions étiquetées (LTS pour Labeled Transition System) A est un quadruplet $< Q, \Sigma, q_0, \rightarrow >$ où :

Q est un ensemble fini non vide d'états,

Σ est un ensemble fini non vide d'interactions (ou d'actions), τ est une action particulière, non visible représentant une action interne,

q_0 est l'état initial du système de transitions étiquetées.

$\rightarrow \subseteq Q \times \Sigma \times Q$ est la relation de transition. $< q, a, q' >$ représente une transition qui part de l'état q et va vers l'état q'. Une telle transition est communément notée par $q \xrightarrow{a} q'$.

Soit le LTS $A =< Q, \Sigma, q_0, \rightarrow >$. A est déterministe ssi :
$\forall q \in Q$, $\forall t = q \xrightarrow{a} q' \in \rightarrow$ avec $a \neq \tau \in \Sigma, q' \in Q$, $\forall t' = q \xrightarrow{b} q'' \in \Sigma$, avec $b \neq \tau \in \Sigma, q'' \in Q$, $a \neq b$ ou $q' = q''$.

Suivant le système à modéliser, l'ensemble des actions peut être partitionné en deux avec Σ_O et Σ_I les ensembles des actions de sortie et d'entrée respectivement.

Le LTS suspension est une extension directe du LTS qui prend en compte la quiescence des états, i.e. le fait de ne pas observer d'action de sortie depuis un état. La quiescence est dénotée par un nouveau symbole $!\delta$ et un LTS complété $\Delta(LTS)$. Ainsi, pour un LTS A, $\Delta(A)$ s'obtient en ajoutant une transition étiquetée par $!\delta$ bouclant pour chaque état n'ayant pas de transition sortante étiquetée par $!\delta$.

Système symbolique à transitions

Le Système Symbolique à Transitions (STS [FTW05]) est une sorte de LTS étendu par un ensemble de variables pouvant prendre des valeurs dans un domaine. Les variables sont soit utilisées avec les actions (ou symboles) pour raffiner et détailler les interactions, soit sont des variables internes au système ou à l'application. Les transitions sont aussi étiquetées par des gardes et des affectations sue ces variables. Ce modèle utilise également la notion d'états symboliques ("locations" en anglais) qui regroupent chacun un ensemble d'états obtenus par le produit cartésien d'un état symbolique et du domaine des variables.

Définition 1.3.3 *Un système symbolique à transitions STS est un tuple* $< L, l_0, V, V_0, I, \Lambda, \rightarrow >$*, où :*

- *L est l'ensemble fini des états symboliques, avec l_0 l'état symbolique initial,*
- *V est l'ensemble fini des variables internes, tandis que I est l'ensemble fini des variables externes ou d'interactions. Nous notons D_v le domaine dans lequel une variable v prend des valeurs. Les variables internes sont initialisées par l'affectation V_0, qui est supposée prendre une valeur unique dans D_V,*
- *Λ est l'ensemble fini des actions (symboles), partitionné par $\Lambda = \Lambda^I U \Lambda^O$: les entrées, commençant par ?, sont fournies au système, tandis que les sorties (commençant par !) sont observées depuis ce dernier,*
- *$\rightarrow$ est l'ensemble fini des transitions. Une transition $(l_i, l_j, a(p), \varphi, \varrho)$, depuis l'état symbolique $l_i \in L$ vers $l_j \in L$, aussi noté $l_i \xrightarrow{a(p),\varphi,\varrho} l_j$ est étiquetée par $a(p) \in \Lambda \times \mathcal{P}(I)$, avec $a \in \Lambda$ une action et $p \subseteq I$ un ensemble fini de variables externes $p = (p_1, ..., p_k)$. Nous notons type$(p) = (t_1, ..., t_k)$ le type de l'ensemble des variables p. $\varphi \subseteq D_V \times D_p$ est une garde qui restreint l'exécution d'une transition. Des variables internes sont mises à jour avec l'affectation $\varrho : D_V \times D_p \rightarrow D_V$ une fois que la transition est exécutée.*

Une extension immédiate du modèle STS est le STS *suspension*, qui décrit en plus la quiescence des états i.e. l'absence d'observation à partir d'état symbolique. La quiescence est dénotée par un nouveau symbole $!\delta$ et un STS complété $\Delta(STS)$. Pour un STS $\mathcal{S}$, $\Delta(\mathcal{S})$ s'obtient en ajoutant une transition étiquetée par $!\delta$ bouclant pour chaque état symbolique à partir duquel la quiescence peut être observée. Les gardes de ces nouvelles transitions doivent retourner vrai pour chaque valeur de $D_{V \cup I}$ qui ne permet pas l'exécution d'une transition étiquetée par une sortie.

La sémantique d'un STS, décrivant chaque action valuée et chaque état, est définie par un LTS, nommé le LTS sémantique. Celui-ci correspond à un automate valué sans variable : les états du LTS sémantique sont étiquetés par les valeurs des variables internes, les transitions sont valuées par des valeurs de paramètres.

Définition 1.3.4 *La sémantique d'un STS* $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \rightarrow >$ *est le LTS* $||\mathcal{S}|| =< Q, q_0, \sum, \rightarrow >$ *où :*

- $L = Q \times D_V$ *est l'ensemble fini des états,*
- $q_0 = (l_0, V_0)$ *est l'état initial,*
- $\sum = \{(a(p), \theta) \mid a(p) \in \Lambda, \theta \in D_p\}$ *est l'ensemble des symboles valués,*
- $\rightarrow$ *est la relation de transition* $Q \times \Sigma \times Q$ *obtenue par la règle suivante :*

$$\frac{l_i \xrightarrow{a(p),\varphi,\varrho} l_j, \theta \in D_p, v \in D_V, v' \in D_V, \varphi(v,\theta)\ true, v' = \rho(v,\theta)}{(l_i, v) \xrightarrow{a(p),\theta} (l_j, v')}$$

Intuitivement, pour une transition de STS $l_i \xrightarrow{a(p),\varphi,\varrho} l_j$, nous obtenons une transition de LTS $(l_i, v) \xrightarrow{a(p),\theta} (l_j, v')$ avec v un ensemble de valeurs de variables internes, si il existe un tuple de valeurs de paramètre θ tel que la garde $\varphi(v, \theta)$ retourne vrai. Un fois que la transition est exécutée, les variables internes prennent la valeur v' déduite de l'affectation $\varrho(v, \theta)$. Un STS suspension $\Delta(\mathcal{S})$ est associé à son LTS suspension sémantique par $||\Delta(\mathcal{S})|| = \Delta(||\mathcal{S}||)$.

Automate temporisé

Alur et Dill ont proposé dans [AD94] une modélisation des systèmes temporisés, appelée automate temporisé, qui décrit le temps d'une manière continue grâce à un ensemble d'horloges. Un automate temporisé n'est autre qu'un LTS (Définition 1.3.2) auquel est associé un ensemble d'horloges. Chaque horloge est évaluée par un réel (représentation dense du temps). Elles sont initialisées à 0 dans l'état initial de l'automate, mais peuvent être réinitialisées avec chaque transition. Ces transitions peuvent de plus contenir des contraintes sur ces horloges. Ainsi, pour qu'une transition soit franchie, toutes les horloges du système doivent satisfaire ces contraintes.

Définition 1.3.5 (Contrainte d'horloges [AD94]) *Soit* $C_{\mathcal{A}}$ *l'ensemble fini des horloges du système* $\mathcal{A}$*, et* $x_i \in C_{\mathcal{A}}$*. Une contrainte d'horloges* δ *sur* x_i *est une expression booléenne de la forme* $\delta = x_i \leq c \mid c \geq x_i \mid \neg\delta \mid \delta_1 \wedge \delta_2$ *où* $c \in \mathbb{Q}$*.*

$\Phi(C_{\mathcal{A}})$ *représente l'ensemble des contraintes d'horloges sur* $C_{\mathcal{A}}$*.*

La définition de l'automate temporisé est donc la suivante :

Définition 1.3.6 (Automate temporisé) *Un automate temporisé* $\mathcal{A}$ *est défini comme un tuple* $< \Sigma_{\mathcal{A}}, S_{\mathcal{A}}, s^0_{\mathcal{A}}, C_{\mathcal{A}}, E_{\mathcal{A}} >$*, tel que :*

- $\Sigma_{\mathcal{A}}$ *est un alphabet fini,*
- $S_{\mathcal{A}}$ *est un ensemble fini d'états,*
- $s^0_{\mathcal{A}}$ *est l'état initial,*
- $C_{\mathcal{A}}$ *est un ensemble fini d'horloges,*
- $E_{\mathcal{A}} \times S_{\mathcal{A}} \times \Sigma_{\mathcal{A}} \times 2^{C_{\mathcal{A}}} \times \Phi(C_{\mathcal{A}})$ *est un ensemble fini de transitions. Un tuple* $< s, s', a, \lambda, G >$ *représente une transition de l'état* s *vers l'état* s' *avec le symbole* a*. L'ensemble* $\lambda \subseteq C_{\mathcal{A}}$ *correspond aux horloges réinitialisées avec cette transition, et* G *est une contrainte d'horloges sur* $C_{\mathcal{A}}$*.*

Pour obtenir la valeur d'une ou plusieurs horloges, il est nécessaire de les interpréter par des fonctions appelées valuations d'horloges.

Définition 1.3.7 (Valuation d'horloges [AD94]) *Une valuation d'horloges sur un ensemble d'horloges $C_{\mathcal{A}}$ est une fonction v qui assigne à chaque horloge $x \in C_{\mathcal{A}}$ une valeur dans $\mathbb{R}+$. Nous notons l'ensemble des valuations d'horloges $V(C_{\mathcal{A}})$.*

Une valuation d'horloges satisfaisant une contrainte d'horloges G, sera notée $v \models G$.

Pour $d \in \mathbb{R}+$, $v + d$ dénote la valuation d'horloges qui assigne une valeur $v(x) + d$ à chaque horloge x. Pour $X \subseteq C_{\mathcal{A}}$, $[X \rightarrow d]$ dénote la valuation d'horloges sur $C_{\mathcal{A}}$ qui assigne d à chaque horloge $x \in X$.

Ainsi, si x et y sont deux horloges, $v(x)$ donne la valeur de l'horloge x, $v(y)$ donne celle de l'horloge y, $v(x, y)$ donne le couple de valeurs d'horloges.

Il résulte, d'après les définitions précédentes, que le comportement d'un système temporisé dépend de ses états et des valeurs d'horloges, obtenues par les valuations de $V(C_{\mathcal{A}})$. Cependant, cet ensemble de valeurs d'horloges est infini. Une relation d'équivalence sur les valuations d'horloges est donc définie dans [AD94]. Celle-ci décrit les classes d'équivalences qui rassemblent les valeurs d'horloges possédant la même partie entière, en régions d'horloges.

Pour $t \in \mathbb{R}+$, la partie fractionnelle de t est notée $fract(t)$ et sa partie entière est notée $\lfloor t \rfloor$.

Définition 1.3.8 (Région d'horloges [AD94]) *Soit $\mathcal{A} =< \Sigma_{\mathcal{A}}, S_{\mathcal{A}}, s^0_{\mathcal{A}}, C_{\mathcal{A}}, E_{\mathcal{A}} >$ un automate temporisé. Pour chaque $x \in C_{\mathcal{A}}$, soit c_x le plus grand entier c tel que $(x \leq c)$ où $(c \leq x)$ est une sous formule de contraintes d'horloges appartenant à $E_{\mathcal{A}}$.*

La relation d'équivalence $\sim$ est définie sur l'ensemble de toutes les valuations d'horloges sur $C_{\mathcal{A}}$; $v \sim v'$ ssi toutes les conditions suivantes sont vérifiées :

- *Pour tout $x \in C_{\mathcal{A}}$, soit $\lfloor v(x) \rfloor$ et $\lfloor v'(x) \rfloor$ sont les mêmes, soit $v(x)$ et $v'(x)$ sont supérieurs à c_x.*
- *Pour tout $x, y \in C_{\mathcal{A}}$ avec $v(x) \leq c_x$ et $v(y) \leq c_y$, $fract(v(x)) \leq fract(v(y))$ ssi $fract(v'(x)) \leq fract(v'(y))$.*
- *Pour tout $x \in C_{\mathcal{A}}$ avec $v(x) \leq c_x$, $fract(v(x)) = 0$ ssi $fract(v'(x)) = 0$.*

Une région d'horloges pour $\mathcal{A}$ est une classe d'équivalence de valuations d'horloges induites par $\sim$.

Donc, plus simplement, une région d'horloges est une sorte d'intervalle de temps dépendant de plusieurs horloges. Elle peut être représentée par un polyèdre ayant plusieurs sommets.

Un graphe des régions est une représentation équivalente d'un automate temporisé où les contraintes d'horloges ne sont pas décrites sur les transitions mais dans les états. Ainsi, un état d'un graphe des régions, est composé d'un état de l'automate temporisé et d'une région d'horloges. Cette représentation permet de distinguer, avec certitude, chaque intervalle de temps pendant lequel une action peut être exécutée. Un algorithme de transformation d'un automate temporisé en graphe des régions est donné dans [AD94].

Définition 1.3.9 (Graphe des régions) *Soit $\mathcal{A} = (\Sigma_{\mathcal{A}}, S_{\mathcal{A}}, s^0_{\mathcal{A}}, C_{\mathcal{A}}, E_{\mathcal{A}})$ un automate temporisé. Un graphe des régions de $\mathcal{A}$ est un automate $\mathcal{RA} = (\Sigma_{\mathcal{RA}}, S_{\mathcal{RA}}, s^0_{\mathcal{RA}}, E_{\mathcal{RA}})$ tel que :*
$\Sigma_{\mathcal{RA}} = \Sigma_{\mathcal{A}} \cup \delta$, où δ représente l'écoulement du temps.

$S_{\mathcal{RA}} \subseteq \{\langle s, [v]\rangle \mid s \in S_{\mathcal{A}} \wedge v \in V(C_{\mathcal{A}})\}$.
$s^0_{\mathcal{RA}} = \langle s^0_{\mathcal{A}}, [v_0]\rangle$ *où* $v_0(x) = 0\ \forall x \in C_{\mathcal{A}}$.
$\mathcal{RA}$ *possède comme transition* $q \xrightarrow{a}_{\mathcal{RA}} q'$, *qui part de l'état* $q = \langle s, [v]\rangle$ *vers* $q' = \langle s', [v']\rangle$ *avec le symbole* a, *ssi*

- *soit* $a \neq \delta\ \exists (s, s', a, \lambda, G) \in E_A$, $d \in \mathbf{R}^+\ tq\ (v + d) \models G$, $v' = [\lambda \mapsto 0](v + d)$,
- *soit* $a = \delta$, $s = s'$ *et* $\exists d \in \mathbf{R}^+\ tq\ v' = v + d$.

Les automates à intervalles sont un autre formalisme pour décrire des systèmes temporisés. Ce modèle, dérivé des automates temporisés [AD94], est un graphe représentant l'exécution d'un système. Chaque action est décrite par une transition étiquetée par un symbole. Le temps est représenté par un ensemble d'horloges prenant des valeurs réelles, et par des contraintes étiquetées sur les transitions représentées par des zones de contraintes.

Définition 1.3.10 (Zone de contraintes) *Une zone de contraintes d'horloges, notée* Z, *sur un ensemble d'horloges* C *est un tuple* $< Z(1), ...Z(n) >$ *d'intervalles tel que* $card(Z) = card(C)$ *et tel que* $Z(x_k)$ *soit de la forme* $[a_k\ b_k]$ *ou* $[a_k\ b_k[$, *avec* $a_k \in \mathbb{R}^+$ *et* $b_k \in \{\mathbb{R}^+, \infty\}$, *est un intervalle de temps pour l'horloge* x_k.
Soit $v = (x_1, ..., x_n) \in \mathbb{R}^{+n}$. v *satisfait* Z, *noté* $v \models Z$ *ssi* $\forall x_i \in v, x_i \in Z(i)$

Définition 1.3.11 (Automate à intervalles) *Un automate à intervalles* $\mathcal{AI}$ *est un tuple* $< \Sigma_{\mathcal{AI}}, S_{\mathcal{AI}}, s^0_{\mathcal{AI}}, C_{\mathcal{AI}}, I_{\mathcal{AI}}, E_{\mathcal{AI}} >$ *tel que :*

- $\Sigma_{\mathcal{AI}}$ *est un alphabet fini contenant des symboles d'entrées (commençant par "?") et des symboles de sorties (commençant par "!"),*
- $S_{\mathcal{AI}}$ *est un ensemble fini d'états,* $s^0_{\mathcal{AI}}$ *est l'état initial,*
- $C_{\mathcal{AI}}$ *est un ensemble fini d'horloges,* $I_{\mathcal{AI}}$ *est un ensemble fini d'intervalles,*
- $E_{\mathcal{AI}}$ *est un ensemble fini de transitions. Un tuple* $< l, l', a, \lambda, Z >$ *représente une transition de l'état* l *vers l'état* l' *étiquetée par le symbole* a. *L'ensemble* $\lambda \in C_{\mathcal{AI}}$ *correspond aux horloges réinitialisées et* $Z =< Z(1), ..., Z(n) >_{(n=card(C_{AI}))}$ *est une zone de contrainte.*

Un automate à intervalles peut se construire à partir d'un automate temporisé. Une méthode, utilisant l'algorithme de Dijkstra et dont le coût est polynomial, est décrite dans [LC97].

1.3.2 Quelques méthodes de test classiques

Nous présentons dans cette partie, de façon très succincte, quelques méthodes de test connues et citées dans la littérature, sur lesquelles se basent ou sont inspirées les approches décrites dans ce mémoire. Nous présentons ainsi :

- l'approche orientée Objectif de test qui vérifie si un ensemble de besoins (objectifs de test), appartenant ou non à la spécification, peut être vérifié sur l'implantation,
- les méthodes de test basées sur le modèle FSM notamment pour les systèmes de télécommunication. Ces méthodes (TT, UIO, W, Wp, HSI, etc.), consistent à vérifier que l'implantation ne contient pas de fautes par rapport à un modèle de fautes donné, en caractérisant, d'une manière plus ou moins forte, l'ensemble du comportement de la spécification et en vérifiant que la signature obtenue existe dans l'implantation,

- les méthodes de test basées sur le modèle LTS. Dans ce cas, des relations de test sont définies puis des cas de test sont générés pour vérifier cette relation. Bien évidemment, nous présenterons sommairement la relation *ioco*,
- le Testeur Canonique, qui détermine si une relation, appelée relation d'implantation, entre la spécification et l'implantation est vérifiée.

- un aperçu sur le test passif aussi appelé le monitoring, dont le but n'est pas de générer et d'expérimenter des cas de test mais de collecter passivement les réactions de l'implantation pour vérifier des propriétés.

1.3.3 Méthodes orientées Objectif de test

Les méthodes orientées Objectif de test [Bri87b, Tre96a, PLC98, KJM03, LZCH08a, FW08, JÓ9, AdSSR09] sont spécifiquement désignées pour ne tester qu'une partie du comportement de la spécification. Le concepteur choisit des propriétés comportementales, appelées Objectif de test, qu'il souhaite vérifier sur l'implantation.

Ces objectifs peuvent être générés pour satisfaire deux buts : vérifier que des propriétés comportementales de la spécification existent dans l'implantation, ou que d'autres propriétés n'existent pas. Ceci conduit à la définition suivante :

Définition 1.3.12 *Un objectif de test est un automate déterministe et acyclique contenant des états particuliers :*

- *Un objectif de test acceptant est composé d'un ensemble de propriétés appartenant à la spécification. Chaque état de l'objectif appartient à un ensemble d'états noté Accept, montrant que l'objectif de test doit être entièrement satisfait,*
- *Un objectif de test refusant est composé d'un ensemble de propriétés appartenant ou non à la spécification. Les états appartiennent soit à l'ensemble Accept soit à l'ensemble Refuse. Le fait qu'un état soit étiqueté par Accept montre que la propriété précédant cet état doit se vérifier dans l'implantation. S'il est étiqueté par Refuse, la propriété ne doit pas se vérifier.*

Ces objectifs de test sont généralement créés à la main, mais certains travaux proposent une automatisation de leurs choix [FJJV96].

Les cas de test sont extraits à partir d'un objectif de test qui peut être combiné avec la spécification. Dans ce dernier cas, un produit synchronisé entre la spécification et l'objectif est obtenu, en combinant des chemins de la spécification [Lau99] ou bien la spécification complète avec les propriétés de l'objectif de test. Plusieurs méthodes sont proposées pour arriver à ce but, notamment par parcours en profondeur ou largeur et par simulation. Chacune de ces méthodes possède ses propres avantages et inconvénients suivant la spécification. Une fois les cas de test générés, ceux-ci sont expérimentés sur l'implantation, lors de la phase de test. Un verdict est ensuite obtenu : si les propriétés acceptantes de l'objectif de test sont vérifiées et si les propriétés refusantes ne le sont pas alors l'objectif de test est satisfait.

Prenons l'exemple d'une spécification illustrée en Figure 1.5, décrivant un distributeur à café. Si le concepteur veut juste être conforté sur le fait que lorsqu'il met un jeton, il obtient bien son café, l'objectif de test décrit par le LTS de la Figure 1.5 lui permet de vérifier cette propriété fondamentale pour le système (et le concepteur).

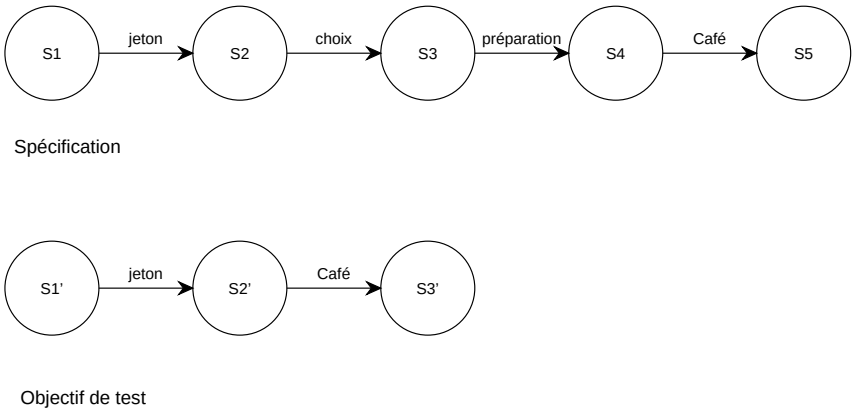


FIGURE 1.5 – Une spécification LTS décrivant un distributeur à Café, et un objectif de test

1.3.4 Méthode de test sur les FSM, Méthodes orientées caractérisation d'états

Plusieurs travaux sur les méthodologies de test basées sur le modèle FSM ont été proposés. Les méthodes les plus connues sont la méthode TT [NT81a](Transition Tour), la méthode DS [Gon80](Distinguish Sequence), la méthode UIO [SD88b] et ses extensions UIOv [VCI89], UIOp [CA92b], UIOs [SLD92] (Unique Input Output), la méthode W [Cho78] et ses extensions Wp [FBK+91], HSI [LPvB94]. D'autres peuvent être trouvées dans [TARC99, CPY00, PYvB96, Bri87b, CCKS95]. Chacune permet, suivant un degré plus ou moins élevé, la détection de fautes sur l'implantation sous test. Les fautes qui ont la possibilité d'être relevées sont réunies dans un *Modèle de fautes*. Nous rappelons rapidement le modèle de faute pour les FMSs [BDDD91] :

1. *erreur de sortie* : une implantation produit une erreur de sortie, s'il existe une transition qui ne porte pas le même symbole dans la spécification et l'implantation.
2. *erreur de transfert* : une implantation produit une erreur de transfert s'il existe une transition qui ne conduit pas au même état dans la spécification et l'implantation.
3. *Extra état (état manquant)* : une implantation est dite avoir un état extra (état manquant) si le nombre d'états de cette implantation doit être diminué (augmenté) pour obtenir le même nombre d'états que la spécification.

Le but de ces méthodes est de rechercher une sorte de signature de toutes les propriétés comportementales de la spécification et de l'intégrer dans les cas de test. Lors de la phase de test, cette signature devra être retrouvée dans l'implantation.

Ces méthodes peuvent détecter un éventail variable de fautes. Cet éventail, ou couverture de fautes, varie suivant la méthode utilisée et influence la longueur des cas de test. Les

auteurs de [FBK+91] ont présenté une comparaison de couverture de fautes de quelques unes des méthodes précédemment citées. La Figure 1.6 illustre cette comparaison.

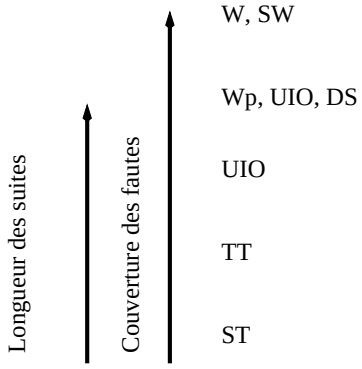


FIGURE 1.6 – Couverture de fautes de quelques méthodes de dérivation de test.

Parmi ces méthodes, la méthode W [Cho78] (et dérivées) permet la plus grande détection d'erreur. Nous nous sommes intéressés à cette méthode pour cette raison même si elle offre plusieurs inconvénients tels que le temps de calcul ou les hypothèses requises sur la spécification (automate déterministe, complètement spécifiée sur l'ensemble des symboles d'entrée et surtout minimal ce qui n'est pas toujours trivial à obtenir).

La méthode W est inspirée de la distinction de deux états d'une FSM présentée dans [Gil62]. Cette méthode utilise un ensemble particulier appelé Ensemble de Caractérisation d'états noté W.

Définition 1.3.13 *Ensemble de caractérisation d'états W.*

Soit $A < s_0, S, I, O, T >$ une FSM déterministe et minimaleCM/TP. Un ensemble de caractérisation W est composé de séquences d'entrée qui permettent la distinction du comportement de chaque paire d'états d'un automate minimal, en observant leurs réactions.

Cet ensemble permet d'obtenir l'identification unique de chaque état de la spécification. De plus, cet ensemble existe toujours, ce qui n'est pas le cas des séquences UIO. Cette méthode et ses extensions utilisent un autre ensemble de séquences, appelé Ensemble de couverture de transitions, lors de la génération des séquences de test.

Définition 1.3.14 *Ensemble de couverture de transitions P.*

Soit une FSM $A < s_0, S, I, O, T >$. L'ensemble de couverture de transitions P est un ensemble de séquences de symboles d'entrée tel que :
$\forall s_i \xrightarrow{x/y} s_j \in T, \exists p, p.x \in P$ *tel que* $s_0 \xrightarrow{p} s_i$ *et* $s_0 \xrightarrow{p.x} s_j$.

La construction peut être faite en générant un arbre couvrant de l'automate, l'ensemble P étant tous les chemins partiels de cet arbre.

Les séquences de test obtenues par la méthode W sont : $P \times \{W \cup W.I \ ... \ \cup I^{m_n}.W\}$, avec I l'ensemble des symboles d'entrée de la spécification S, n le nombre d'états de S, m le nombre d'états de l'implantation, et P l'ensemble de couverture de transitions.

La relation de conformité, entre la spécification S et l'implantation I , est définie pour la méthode W par une équivalence, appelée W-équivalence. En résumé, cette équivalence, basée sur un isomorphisme de S vers I (qui correspond à une bissimulation), prodigue que pour chaque état E de S, il doit exister un unique état E' dans I tel que E et E' réagissent de la même manière en leur appliquant l'ensemble W. En appliquant les séquences de test sur I, si celle-ci réagit en donnant les mêmes symboles de sortie que S, alors I est W-équivalent à S. Chow a montré avec cette équivalence [Cho78], que toute erreur de transfert et de sortie sera détectée dans l'implantation.

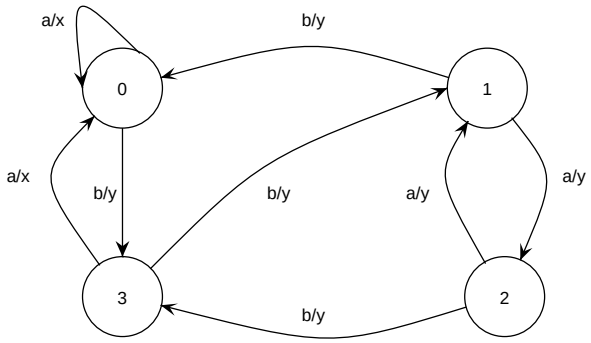


FIGURE 1.7 – Autre exemple de FSM.

En guise d'illustration, l'implantation de la figure 1.7 nous donne les suites de test suivantes : { b.b.a.b.a.a.a, b.b.a.b.b.b.a, b.b.a.a.a.a, b.b.a.a.b.b.a, b.b.b.a.a.a, b.a.a.a.a, b.b.b.b.b.a, b.a.b.b.a, b.b.a.a.a, b.b.b.b.a, a.a.a.a, a.b.b.a, b.a.a.a, b.b.b.a, a.a.a, b.b.a }

1.3.5 Méthodes de test basées sur les LTSs, méthodes orientées *ioco*

Les travaux proposés à partir du modèle LTS, bien que proches, partent de la définition de relations de test entre la spécification et l'implantation. La génération des cas de test est ensuite effectuée pour vérifier la satisfaction de la relation.

Ainsi plusieurs relations de test ont été proposées, initialement par De Nicolas et Hennecy [NH83] avec les relations *must*, *may*, $\sqsubseteq_{may}$, $\sqsubseteq_{must}$, $\sqsubseteq_{te}$ pour ne citer qu'elles. Puis, Briskma a proposé la relation *conf* pour les protocoles de télécommunication qui considère des propriétés *must* avec les traces de la spécification [Bri87b].

De façon intuitive, une trace d'un LTS représente une séquence d'actions extraites d'un chemin de ce LTS. Cette notion de trace et d'observation est reprise par Tretmans dans [Tre96c] en introduisant la notion d'observateur : un LTS p est équivalent à un LTS q si tout observateur peut faire les mêmes observations avec p et q. Il définit notamment les relations *ioco*, *ioconf* et $ioco_{\mathcal{F}}$ qui est une généralisation des deux précédentes. Ainsi, la relation *ioco* entre un LTS S et une implantation I modélisée également par un LTS est définie par :

$$I \text{ ioco } S =_{def} \forall \sigma \in Straces(S) : out(I \text{ after } \sigma) \subseteq out(S \text{ after} \sigma)$$

avec $Straces(S)$, les traces obtenues depuis le LTS suspension de S. Les définitions des opérateurs *out* (qui donne l'ensemble des sorties observées depuis un état) et $after$ (qui depuis un état mène à un autre état via une suite d'actions) peuvent être trouvées dans [Tre96c]. Jéron a réécrit cette relation de façon plus intuitive dans [JÓ9] pour les STSs déterministes par :

$$I \text{ ioco } S =^{\Delta} Straces(I) \cap NC\_Straces(S) = \emptyset$$

avec I un LTS suspension décrivant l'implantataion, S un STS déterministe décrivant une spécification, $NC\_Straces(S) = Straces(S).(\Sigma_O \cup \{\delta\})\backslash Straces(S)$ l'ensemble minimal des traces non conforme du LTS suspension S.

Tretmans donne également un algorithme de génération de cas de test permettant de vérifier la satisfaction de la relation *ioco* dans [Tre96c]. La Figure 1.8 décrit un exemple de LTS avec un tel cas de test.

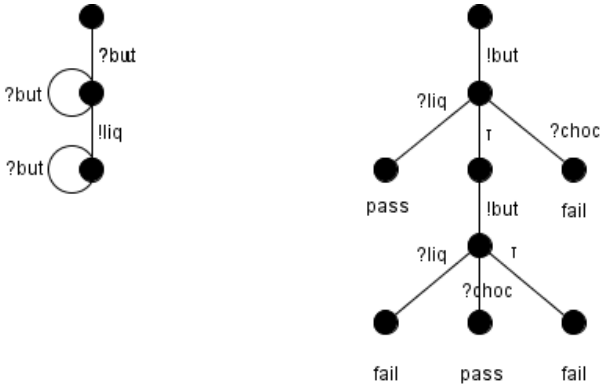


FIGURE 1.8 – Spécification (à droite) et cas de test (à gauche)

Afin de définir la relation *passe* (section 1.2), Tretmans définit également l'exécution d'un cas de test T sur une implantation I modélisée par un LTS. L'exécution de T sur I est donnée par la composition parallèle $T||\Delta(I)$ définie par le LTS $T||\Delta(I) =< Q_I \times Q_T, q0_I \times q0_T, \sum_I \cup \{!\delta\}, \rightarrow_{T||\Delta(I)} >$ où $\rightarrow_{T||\Delta(I)}$ est construit par la règle suivante :

$$\frac{a \in \sum_I \cup \{!\delta\}, q_1 \xrightarrow{a}_T q_2, q'_1 \xrightarrow{a}_{\Delta(I)} q'_2}{q_1 q'_1 \xrightarrow{a}_{T||\Delta(I)} q_2 q'_2}$$

Ainsi, l'implantation I ***passe*** T ssi $T||\Delta(I)$ ne mène pas à un état $Fail$.

De nombreux travaux, proposant des méthodes de génération de cas de test pour LTS ou STS, s'inspirent de la ioco. Parmi ceux-ci, nous pouvons notamment citer [FTW05, RMJ05, iHBvdRT03, JÓ9, OST10, BK11]

1.3.6 Testeur canonique

Un testeur canonique est un automate dérivé de la spécification qui correspond à la spécification avec entrées/sorties inversées et complétée avec son comportement incorrect. Il correspond à un cas de test.

Celui-ci est construit par rapport à la relation décrite entre la spécification et l'implantation, qui est appelée Relation d'implantation.

Définition 1.3.15 *Relation d'implantation*
Une relation d'implantation est une relation R sur LTS × LTS. Soient $I, S \in LTS$. Si $R(I, S)$ on dit que I est une implantation conforme de S.

Plusieurs relations d'implantations peuvent être trouvées, notamment dans [Bri88, Pha94]. Toutes permettent d'obtenir une plus ou moins grande couverture de fautes.

Par exemple, nous pouvons trouver dans [Pha94] des relations basées sur les traces d'un IOSM et de l'implantation. Une trace d'un IOSM S $< S_s, L_s, T, s_0 >$, notée $Tr(S)$, est donc une séquence d'entrées et de sorties observable qui, à partir de l'état initial, nous amène à un état quelconque de S. Pour $\sigma \in Tr(S)$, notons $O(\sigma, S) = \{a \in L_s \mid \sigma!a \in Tr(S)\}$ les sorties autorisées par la spécification après la trace σ.

Soit donc l'exemple de relation d'implantation décrite dans [Pha94] :

Définition 1.3.16 *Relation d'implantation R_1.*
$R_1(I, S)$ ***ssi*** $(\forall \sigma \in Tr(S))(\sigma \in Tr(I) \Rightarrow O(\sigma, I) \subseteq O(\sigma, S))$

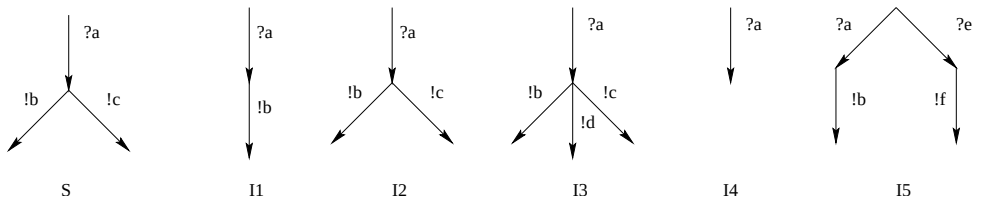


FIGURE 1.9 – Une spécification S et des implantations.

D'après la figure 1.9, nous pouvons voir que toutes les implantations sont conformes par rapport à S sauf I3. R_1 garantit que tout ce qui peut sortir de l'implantation en réponse à une entrée donnée a été prévu par la spécification.

Le testeur canonique obtenu grâce à la relation R_1 est appelé testeur canonique asynchrone [Pha94]. D'autres testeurs canoniques peuvent être obtenus suivant la relation d'implantation choisie. D'autres peuvent être trouvés dans [Bri88, Pha94].

Définition 1.3.17 *Testeur canonique asynchrone[Pha94]*
Soit S une spécification et TM(S)=$< S_s, L_s, T_s, S_{0s} >$ son automate de traces. On appelle testeur canonique asynchrone de S, une IOSM notée TCA(S) = T = $< S_t, L_s, T_t, S_{0t} >$ telle que :

1. $S_t = S_s \cup \{fail\}$
2. $L_t = L_s$
3. $s_0 t = s_0 s$
4. *les transitions du testeur sont exactement celles obtenues par les règles suivantes :*

   a) $(\forall \mu \in \{!, ?\} \times L_s)(\forall s, s' \in S_s)((s, \mu, s') \in T_s \Leftrightarrow (s, inv(\mu), s') \in T_t)$

b) $(\forall s \in S_s)(\forall a \in L_s)(\neg(\exists s')((s, !a, s') \in T_s) \Rightarrow (s, ?a, fail) \in T_t)$
avec $inv(\mu)$ la séquence obtenue en inversant les réceptions (? en !) et les émissions (! en ?).

L'application du testeur canonique sur l'implantation fournit le verdict de test suivant :

Définition 1.3.18 *Verdict du test. FAIL(T,I) ssi* $(\exists \sigma \in Tr(T))(inv(\sigma) \in Tr(I) \wedge (T, \sigma, fail))$. *$Succ(T, I)$ ssi ¬ FAIL(T,I).*

Le testeur canonique asynchrone de S est chargé de vérifier que rien de contraire à ce qui est prévu ne peut arriver. Il fournit donc une image miroir des traces de la spécification. On rajoute un mécanisme de détection des erreurs, c'est-à-dire que si le testeur reçoit une interaction qui n'est pas censée pouvoir arriver dans un état donné, il passe dans l'état **fail**.

Voici un exemple du fonctionnement du testeur canonique basé sur la spécification de la figure 1.9.

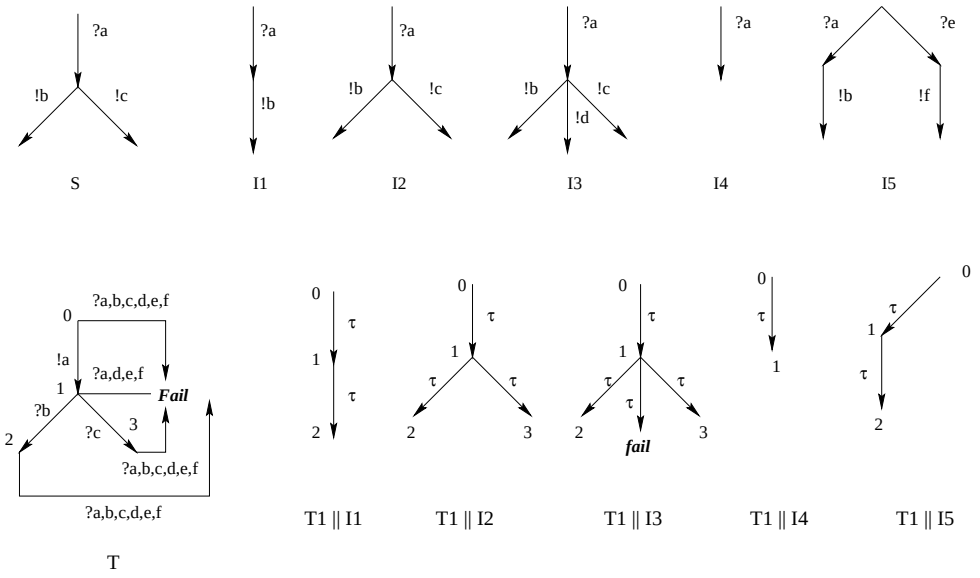


FIGURE 1.10 – Testeur canonique asynchrone.

Cet exemple montre une spécification S et ses cinq implantations. Le testeur canonique asynchrone obtenu est $T1$. Et les exécutions de ce dernier sur les implantations sont notées $T1||I1, ..., T1||I5$ (composition parallèle). La détermination du verdict revient à savoir si un état de la forme (Fail,s_i) se trouve dans le graphe de la composition du testeur $T1$ et de l'implantation.

Plusieurs autres travaux, basé sur le concept de testeur canonique, ont été proposés parmi lesquels [LC97, CJJ07, Val08, JÓ9]

1.3.7 Le Test passif

Au lieu de stimuler de façon active une implantation par le biais de cas de test, le test passif consiste à recueillir les réactions d'une implantation passivement dans le temps à travers des requêtes effectuées par un ensemble de clients.

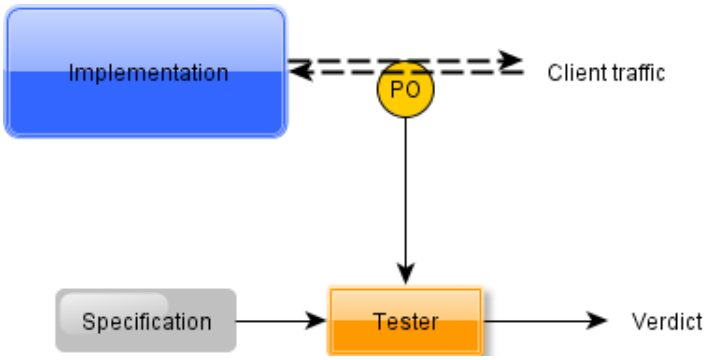


FIGURE 1.11 – Test passif

Bien que ce type de test ne soit pas exhaustif, il offre certains avantages comme la non perturbation inhabituelle d'une implantation ou l'exploitation de cette dernière directement. Plusieurs caractéristiques de test sont possibles, comme la conformité [LCH+06], la sécurité [CBML09], etc.

De façon générale, le testeur s'appuie sur un module de type sniffeur réseau pour collecter des réactions de l'implantation, comme le décrit la Figure 1.11. Ensuite, l'utilisation de ces ensembles de réactions peut être différente :

- **Satisfaction d'invariants** [BCNZ05, CGP03] : des invariants, qui sont des propriétés toujours vraies, sont construits à partir de la spécification. Puis on vérifie la satisfaction des invariants à partir des traces collectées. Cette solution permet de tester des propriétés complexes mais son inconvénient réside dans le fait que les invariants doivent être construits à la main,
- **Vérification en avant** [LNS+97, MA00, LCH+06] : les réactions observées sont données à la volée à un algorithme qui vérifie la satisfaction de la spécification en la parcourant avec les réactions observées. Par exemple, Lee et al. proposent plusieurs algorithmes dans [LCH+06] pour des protocoles de télécommunication modélisés par des spécifications symboliques. De façon intuitive, à chaque réaction observée, composée d'une action et de valeurs de variables, on vérifie que la spécification accepte le symbole et que l'union des contraintes sur les variables extraites de la spécification au fil de l'exécution sont toutes satisfaites par les valeurs collectées. Les conditions d'arrêt sont généralement la détection d'une erreur (non respect de la spécification) ou la fin des réactions observées,
- **Vérification en arrière** [ACC+04] : à partir de réactions observées, cette approche déduit un ensemble de traces en amont (en arrière) pour restreindre l'ensemble des actions déjà effectuées de la spécification. L'algorithme se fait en deux étapes. Il suit une trace donnée en arrière à partir de la configuration courante vers un ensemble de configurations potentielles de départ. Ainsi, on déduit les configurations possibles de départ qui mènent à la configuration courante. Dans une deuxième étape, l'algorithme analyse le passé de ces configurations, en arrière également, pour rechercher des configurations dans lesquelles les valeurs des variables sont connues. Enfin, une décision est prise sur la validité des traces obtenues. Une telle approche est généralement appliquée en complément à une approche de test en avant pour construire des traces plus longues et détecter plus d'erreurs.

1.4 Testabilité

Le coût de la validation d'un système dépasse souvent 70% du coût total de son développement, et ne cesse de croître de par la complexité de plus en plus accrue de ces systèmes. Il est, par conséquent, nécessaire de proposer des méthodes permettant de réduire ce coût. L'un des critères permettant de réduire le coût de la validation est la qualité de test, aussi communément appelée, testabilité. Cette notion repose sur une analyse et une évaluation du système qui permettent d'orienter les opérations de test, d'en réduire le coût, la durée et d'en assurer l'efficacité.

De nombreuses définitions de la qualité de test peuvent être trouvées dans la littérature. Quelques-unes sont décrites ci-dessous :

Définition 1.4.1 *La testabilité est l'aptitude à générer, évaluer et appliquer les tests de façon à satisfaire les objectifs prédéfinis des tests (couverture de fautes, isolation de fautes,etc.) qui sont sujets à des contraintes de coût (argent et temps) [Ben94a].*

Définition 1.4.2 *La testabilité est la capacité du logiciel à révéler ses fautes lors de l'étape de test [VM95].*

La testabilité implique des analyses et mesures, qui ont été représentées dans le cycle du logiciel par un cycle étendu. Celui-ci est résumé et illustré en Figure 1.12.

D'après ce cycle de vie, le concepteur peut évaluer la qualité de test du système, à chaque nouvelle étape de sa création. Ainsi, il peut avoir connaissance du comportement qui pourra ou ne pourra pas être testé, et d'une idée sur le coût de ce test. Il a également la possibilité d'améliorer la spécification pour n'obtenir que des propriétés qui puissent être testées, de réduire le coût de la validation du système et d'apprécier la qualité des améliorations apportées. Il peut être aussi satisfait de la qualité obtenue et continuer le processus de conception. Ces mesures peuvent éviter la génération d'implantations qui ne pourront être entièrement testées et qui demanderont une modification de la spécification puis une nouvelle génération d'implantation. Donc ces mesures évitent une perte de temps et d'argent.

La qualité de test des automates non temporisés dépend de trois notions fondamentales qui sont l'observabilité, la contrôlabilité et le coût du test. Ainsi, Freeman, dans [Fre91], décrit un système testable, comme étant un système observable et contrôlable.

Définition 1.4.3 (Observabilité [Fre91]) *Un système est dit observable si pour tout symbole d'entrée appliqué au système, un symbole de sortie différent est observé.*

Définition 1.4.4 (Contrôlabilité [Fre91]) *Un système est dit contrôlable si pour chaque symbole de sortie observé, il existe un symbole d'entrée, qui, une fois appliqué au système, force cette observation.*

Ces notions sont généralement dépendantes de plusieurs propriétés du système. Celles-ci ont été analysées, par exemple, dans les travaux [KDCK94, KGD96, DAdSS01a]. Ces analyses ont permis de définir plusieurs facteurs, appelés degrés de qualité, mesurant l'influence de ces propriétés sur la qualité de test du système. L'évaluation des degrés permet ainsi de mesurer l'observabilité et la contrôlabilité de la spécification.

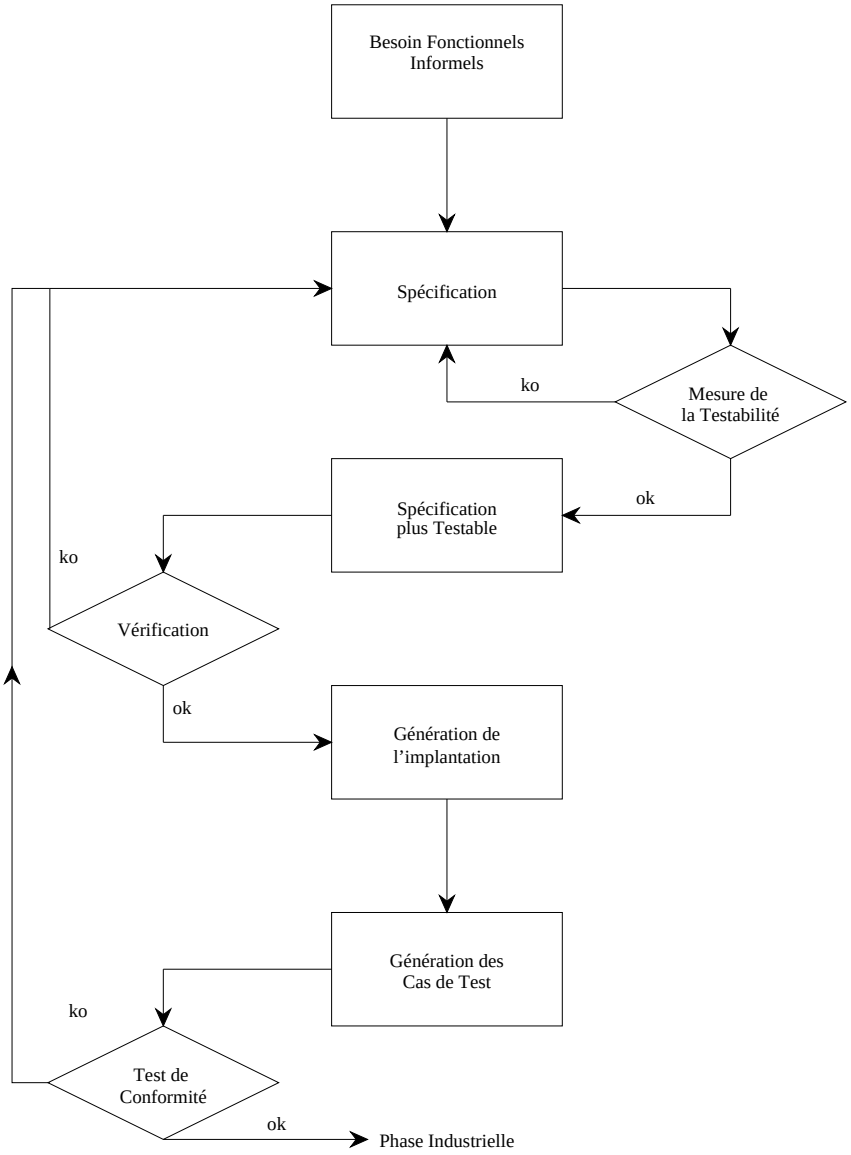


FIGURE 1.12 – Cycle de vie étendu

Chapitre 3

Synthèse des travaux

Ce chapitre représente une synthèse des travaux de recherche que j'ai menés durant ces neuf dernières années. Ce travail a été réalisé au sein du laboratoire LIMOS (Laboratoire d'Informatique, Modélisation et Optimisation des Systèmes,) et en particulier dans l'équipe SIC (Systèmes d'Information et de Communication) dont l'activité est centrée sur les réseaux, les Données, les Services et l'Interopérabilité. J'ai continué mes recherches sur le test des systèmes et protocoles de télécommunication modélisés par des automates temporisés [AD94]. La première section de ce chapitre décrit les travaux correspondants qui se focalisent sur des approches orientées caractérisation d'états et objectif de test. Ces travaux correspondent à des perspectives qui avaient été soulevées lors de ma thèse. Dans ces travaux, nous avons mis l'accent sur la réduction du coût de la génération des tests. Nous avons donc proposé une nouvelle définition de la caractérisation d'états et des méthodes orientées objectifs de test, qui sont réputées pour justement réduire le coût de génération de cas de test et le nombre de ces derniers. Néanmoins, ces méthodes de test nécessitent des objectifs de test pour fonctionner. Hors, ceux-ci sont supposés construits

à la main. Des méthodes de génération automatique d'objectifs de test ont été proposées, mais uniquement pour les systèmes non temporisés [CCKS95, HLU03a]. Nous nous sommes donc penchés sur la génération semi-automatique et automatique d'objectifs de test sur des systèmes temporisés.

En 2006, nous avons souhaité étendre nos collaborations, et avons découvert les problématiques et besoins liés aux paradigme du service et à la composition de services. J'ai donc orienté mes recherches vers cette voie ce qui a favorisé l'intégration de notre équipe dans le projet ANR Webmov (`http://webmov.lri.fr/`). L'objectif principal de ce projet est de contribuer à la conception, à la composition et la validation de services Web à travers une vue abstraite de haut niveau et d'une vision SOA (Architecture Orientée Services) basée sur une architecture logique. Ce projet traite plus particulièrement de la conception et des mécanismes de composition de services Web et également de leur validation à partir de différentes techniques de test actives et passives. A travers ce projet, nous avons contribué à l'élaboration de plusieurs méthodes de test (random testing, test de robustesse, test de sécurité, etc.) et à l'étude de la faisabilité des tests (testabilité) de services Web et de compositions ABPEL. Les problématiques sont différentes de celles rencontrées avec les systèmes temporisés. En effet, les services Web et compositions sont souvent déployés dans des environnements spécialisés (environnements SOAP) qui modifient de façon substantielle les possibilités de test. L'étude de la testabilité nous a montré les restrictions liées à cet environnement. Les modèles utilisés, pour décrire formellement ces services, sont symboliques afin de prendre en compte les notions de paramètres et de réponses typés. D'ailleurs, nous avons souvent repris le modèle STS [FTW05] (section 1.3.1) pour décrire des spécifications. Par nature, ce modèle implique un nombre parfois infini de valeurs à tester. Nous avons donc du prendre en compte cette difficulté pour restreindre la génération des cas de test. Les différents travaux obtenus sont résumés en section 2.2.

En collaboration avec le laboratoire Labri, nous avons également considéré les systèmes et protocoles de télécommunication comme étant des systèmes réactifs. Ceci a conduit à la proposition de deux travaux sur le test de robustesse, qui sont synthétisés en section 2.3.

La dernière section décrit des approches qui sortent du cadre défini par les thèmes précédents. J'ai participé à la fin du projet RNRT Platonis (`http://www-lor.int-evry.fr/platonis/` dont le but est de développer une plate-forme de validation et d'expérimentation d'applications multi-services et multi-plateformes. Ceci a conduit à la définition d'une plate-forme de test distribuée. Par curiosité et par plaisir, nous nous sommes également penchés sur la modélisation et le test d'applications Ajax, puis sur la parallélisation de flots d'invocations de services Web avec le langage OPEN-MP. Ces travaux sont décrits en section 2.4.

2.1 Les Systèmes temporisés

Par rapport aux travaux qui avaient été publiés tels que [COG98, ENDKE98, NS01a, SVD01], nous avons proposé plusieurs approches dont la contribution principale est d'éviter, tant qu'il est possible, de transformer à multiples reprises une spécification temporisée pour appliquer des méthodes de test existantes. Nous avons donc présenté des méthodes qu'il serait possible d'appliquer directement sur les spécifications initiales. Nous avons

redéfini la caractérisation d'états sur les modèles temporisés directement. Nous avons également proposé des méthodes orientées objectifs de test qui prennent en compte des propriétés *acceptantes* (qui appartiennent à la spécification) et *refusantes*.

Les méthodes orientées objectif de test [CCKS95, PLC98, FPS00, Kon01, BCF03, END03, CM04] testent des parties locales des systèmes à partir de besoins de test. Généralement, l'utilisation de telles méthodes réduit fortement les coûts du test. Cependant, ces objectifs de test doivent souvent être construits manuellement. En plus de devenir rébarbatif, cette construction peut aussi devenir un casse-tête si le système est complexe, temporisé, interopérable, etc. Des méthodes de génération automatique d'objectifs de test ont été proposées, mais uniquement pour les systèmes non temporisés [CCKS95, HLU03a]. Nous avons donc apporté plusieurs solutions pour générer des objectifs de test semi-automatiquement ou automatiquement sur des systèmes temporisés. Pour assurer et améliorer l'efficacité des objectifs, nous avons défini un langage rationnel de description qui prend en compte la notion de testabilité.

Ces différents travaux obtenus sont résumés ci dessous.

2.1.1 Méthode de test orientée caractérisation d'état

La caractérisation d'état est une technique bien connue, utilisée premièrement sur des modèles non temporisés [Cho78] puis temporisés [SVD01], qui offre l'avantage d'être la seule technique en boite noire capable de détecter des erreurs de transfert et d'état manquant (section 1.3.1).

Plusieurs méthodes [ENDKE98, SVD01, ENDK02] ont proposé d'utiliser cette technique pour générer des cas de test sur systèmes temporisés. Néanmoins, les auteurs ont souhaité garder la définition de la caractérisation d'origine sur les systèmes non temporisés et donc de transformer la spécification temporisée initiale en plusieurs autres modèles pour appliquer cette définition. Ces transformations impliquant soit un coût élevé soit une explosion de l'espace des états, nous avons proposé, dans [SL03c, SL03b], une définition et une méthode de test pour systèmes temporisés directement, qui ne nécessite pas de transformation pour générer les cas de test.

La spécification est supposée être un automate à intervalles déterministe, minimale, complète et fortement connexe.

Génération des cas de test

La génération des cas de test s'effectue en adaptant la méthode Wp [FBK+91] aux automates à intervalles et à des actions particulières appelées *actions temporisées*. En effet, pour tester une implantation temporisée, nous devons non seulement observer ses réactions mais aussi les moments où elles se produisent. Nous proposons donc de considérer des couples composés d'un symbole et d'une zone de contraintes pendant laquelle peut être émis ou reçu ce symbole.

Définition 2.1.1 (Action Temporisée) *Soit $\mathcal{AI}$ un automate à intervalles $\mathcal{AI} =< \Sigma_{\mathcal{AI}}, S_{\mathcal{AI}}, s^0_{\mathcal{AI}}, C_{\mathcal{AI}}, I_{\mathcal{AI}}, E_{\mathcal{AI}} >$. Une action temporisée $\langle A, Z\rangle$ est un couple où A est un symbole de $\Sigma_{\mathcal{AI}}$ et Z une zone de contrainte sur $C_{\mathcal{AI}}$.*
Action temporisée d'entrée : *Une action temporisée d'entrée $\langle ?A, Z\rangle$ est composée*

d'un symbole d'entrée. Elle modélise le fait que le symbole A peut être reçu par le système pour toute valeur $v = (X_1, ..., X_n)_{(n=card(C_{\mathcal{AJ}}))}$ tel que $v \models Z$.
Action temporisée de sortie : *Une action temporisée de sortie $\langle !A, Z\rangle$ est composée d'un symbole de sortie. Elle modélise le fait que A peut être envoyé par le système pour toute valeur $v = (X_1, ..., X_n)_{(n=card(C_{\mathcal{AJ}}))}$ tel que $v \models Z$.*

Trois ensembles de séquences d'actions temporisées sont utilisés pour générer les cas de test qui sont : l'ensemble de couverture d'états aussi appelé l'ensemble des préambules(Q), l'ensemble de couverture de transitions (P) et l'ensemble de caractérisation W.

Définition 2.1.2 (Ensemble de couverture d'états Q) *Un ensemble de couverture d'états Q pour un automate à intervalles $\mathcal{AJ}$ est composé, pour tout état $s_i \in S_{\mathcal{AJ}}$, d'une séquence d'actions temporisées σ qui permet d'atteindre s_i à partir de l'état initial s_0.*

Définition 2.1.3 (Ensemble de couverture de transition P) *Un ensemble de couverture de transition P pour un automate à intervalles $\mathcal{AJ}$ est composé de séquences d'actions temporisées telles que pour chaque transition $s_i \xrightarrow{A} s_j \in E_{\mathcal{AJ}}$, il existe deux séquences d'actions temporisées $\sigma \in P$ et $\sigma\langle A, Z_i\rangle \in P$ tel que σ permet d'atteindre s_i à partir de l'état initial s_0 et $\sigma\langle A, Z_i\rangle$ permet d'atteindre s_j à partir de s_0.*

La première définition de la caractérisation d'états est donnée dans [Gil62], et utilisée dans le domaine du test dans [Cho78]. Celle-ci peut se résumer par *"Un ensemble de caractérisation d'états W est composé de séquences d'entrées qui permettent la distinction de chaque paire d'états d'un automate à entrée/sortie minimal par l'observation de ses réactions de sorties"*.

Concernant les systèmes temporisés, la distinction d'une paire d'états d'un automate à intervalles dépend toujours des réactions des symboles de sortie mais aussi du moment de ces réactions. Plus précisément, cette distinction va s'opérer sur l'observation de chaque symbole de sortie et sur la zone de contraintes satisfaisant cette observation, donc sur les actions temporisées. Ainsi, notre définition montre que deux états d'un automate à intervalles sont distinguables ssi il existe une séquence d'actions temporisées qui, en étant appliquée sur chaque état, donne soit des observations différentes (symboles de sorties différents), soit les mêmes observations à des moments différents (observations à des intervalles d'horloges différents), soit les deux.

Nous pouvons maintenant définir la caractérisation d'états d'un automate à intervalles :

Définition 2.1.4 (Ensemble de caractérisation d'états W) *Soit $\mathcal{AJ}$ un automate à intervalles $\mathcal{AJ} =< \Sigma_{\mathcal{AJ}}, S_{\mathcal{AJ}}, s^0_{\mathcal{AJ}}, I_{\mathcal{AJ}}, E_{\mathcal{AJ}} >$. Soit $(s_i, s_j) \in S^2_{\mathcal{AJ}}$ une paire d'états de $\mathcal{AJ}$, avec $s_i \neq s_j$. s_i est distinguable de s_j ssi il existe une séquence $\sigma = \langle A_1, Z_1\rangle ... \langle A_n, Z_n\rangle$ tel que :*

1. $\forall\langle A_k, Z_k\rangle_{1\leq k\leq n}$, *avec A_k un symbole de sortie, on a* $s_i \xrightarrow{\langle A_1,Z_1\rangle...\langle A_{k-1},Z_{k-1}\rangle} s_k$ *et* $s_k \xrightarrow{a_k,Z_k} s_{k+1} \in E_{\mathcal{AJ}}$,
2. $\exists\langle A_k, Z_k\rangle_{1\leq k\leq n}$, *avec A_k un symbole de sortie, on a* $s_j \xrightarrow{\langle A_1,Z_1\rangle...\langle A_{k-1},Z_{k-1}\rangle} s_k$ *et* $s_k \xrightarrow{a_k,Z_k} s_{k+1} \notin E_{\mathcal{AJ}}$

L'ensemble de caractérisation W sur $S_{\mathcal{AJ}}$ est tel que $W = \{W_{s_i} \mid s_i \in S_{\mathcal{AJ}}\}$, avec W_{s_i} le plus petit ensemble de séquences d'actions temporisées permettant de distinguer chaque paire (s_i, s_j), avec $s_j \in S_{\mathcal{AJ}}/\{s_i\}$.

A partir des ensembles P, Q, et W, nous pouvons maintenant décrire les étapes de la génération des cas de test :

1. Nous générons l'ensemble de couverture d'états Q et l'ensemble de couverture de transitions P. Puis, nous construisons l'ensemble $R = P/Q$.
2. Chaque état s_i de $S_{\mathcal{AJ}}$ est caractérisé. Nous obtenons $W = \{W_{S_1}, ..., W_{S_n}\}$, avec W_{S_i} le plus petit sous-ensemble de W permettant de caractériser s_i par rapport aux autres états de $S_{\mathcal{AJ}}$.
3. Un premier ensemble de cas de test $\prod_1$ est obtenu par $\prod_1 = P.W = \{p.\sigma \mid p \in P, \sigma \in W\}$. "." représente la concaténation.
4. Un second ensemble de cas de test $\prod_2$ est obtenu avec $\prod_2 = R \otimes W = \{r.\sigma_i \mid r \in R, s_0 \xrightarrow{r} s_i$, et $\sigma_i \in W_{s_i}$.

Un algorithme de génération d'ensembles de caractérisation d'états est donné dans [SL03b].

Outil de génération d'ensemble de caractérisation d'états pour systèmes temporisés

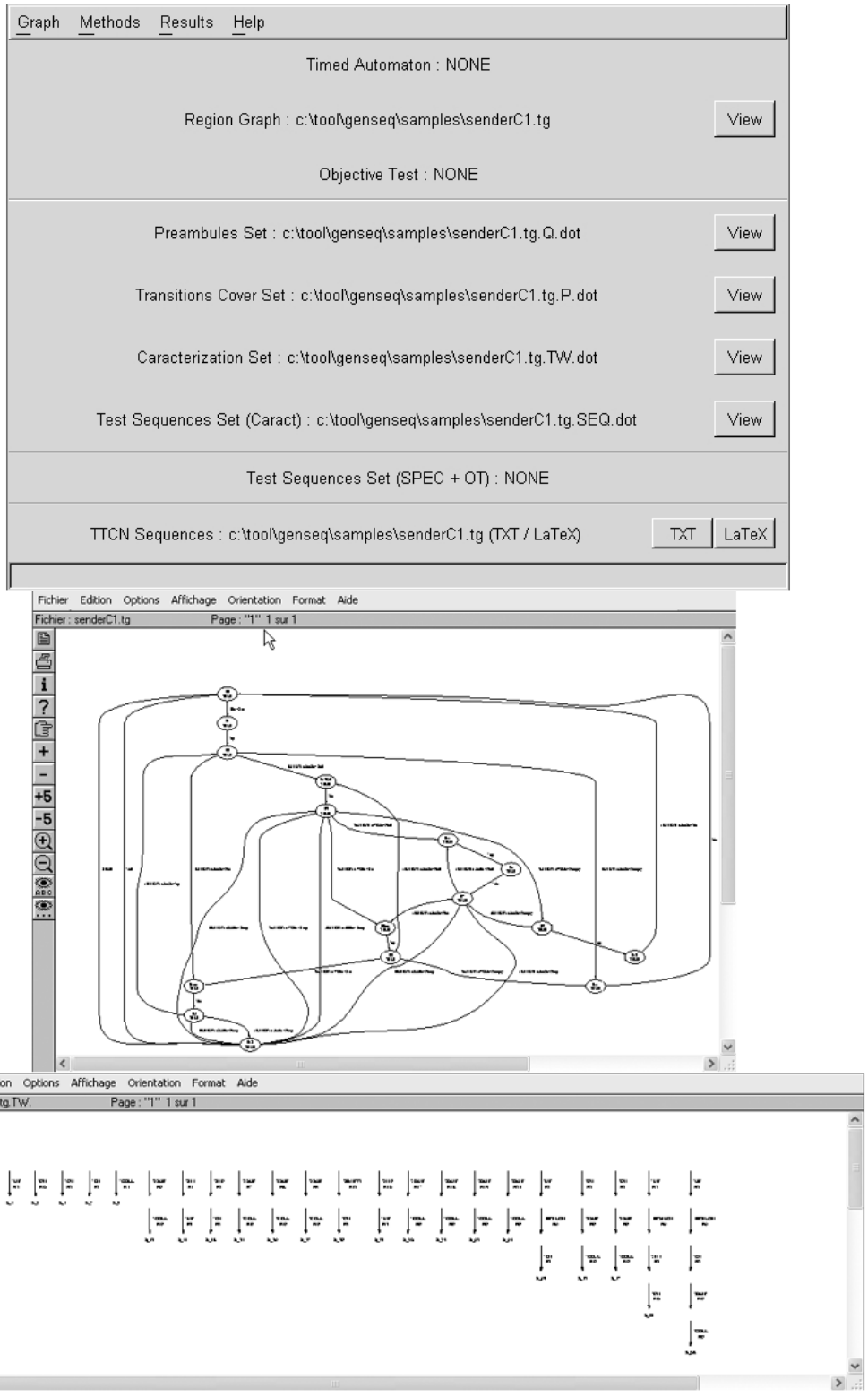
Nous avons implanté un outil de génération d'ensembles de caractérisation d'états pour graphes de régions et automates temporisés. Cet outil reprend l'algorithme décrit dans [SL03b] et le parallélise avec le langage OPEN-MP afin de générer ces ensembles plus rapidement. Quelques copies d'écran sont données en Figure2.13.

Des résultats ont été obtenus sur le protocole MAP-DSM (Figure 2.14) et sur le protocole audio de Philips (Figure 2.15). Cet outil a démontré la faisabilité à utiliser la caractérisation d'états sur des systèmes temporisés bien qu'un assez grand nombre d'hypothèses soient toujours nécessaires. Le temps de calcul nécessaire est très raisonnable (quelques minutes au plus). Il n'est donc pas impossible que nous reprenions cet outil dans des travaux futurs.

2.1.2 Méthode de test orientée objectif de test pour systèmes temporisés

Nous avons proposé des approches de génération de cas de test à partir d'objectifs de test pour les systèmes temporisés. Les objectifs de test représentent l'intention de test (voir section 1.3.3) et sont employés d'une part afin de réduire le coût de test et d'autre part pour éviter une explosion combinatoire de l'espace des états lors de la génération des cas de test.

Dans [Sal02], cette génération est effectuée sur des spécifications décrites par des automates temporisés à partir d'objectifs de test composés de propriétés acceptantes et refusantes. Dans [SL07], les cas de test sont construits à partir d'objectifs de test combinés à une approche de type caractérisation d'états sur des spécifications modélisées par


Graph Methods Results Help
Timed Automaton : NONE
Region Graph : c:\tool\genseq\samples\senderC1.tg
View
Objective Test : NONE
Preambules Set : c:\tool\genseq\samples\senderC1.tg.Q.dot
View
Transitions Cover Set : c:\tool\genseq\samples\senderC1.tg.P.dot
View
Caracterization Set : c:\tool\genseq\samples\senderC1.tg.TW.dot
View
Test Sequences Set (Caract) : c:\tool\genseq\samples\senderC1.tg.SEQ.dot
View
Test Sequences Set (SPEC + OT) : NONE
TTCN Sequences : c:\tool\genseq\samples\senderC1.tg (TXT / LaTeX)
TXT
LaTeX
Fichier Edition Options Affichage Orientation Format Aide
Fichier : senderC1.tg
Page : "1" 1 sur 1
Fichier Edition Options Affichage Orientation Format Aide
Page : "1" 1 sur 1

TA states/transitions	RG states/transitions	Nb of test cases	Test length
11/15	23/31	40	228

FIGURE 2.14 – Résultats sur le protocole MAP-DSM

	TA états/trans.	RG états/trans.	Nb cas de test	Taille cas de test
Sender	16/32	40/66	99	1265
Receiver	8/13	18/37	70	743

FIGURE 2.15 – Résultats sur le protocole Philips audio

des automates à intervalles. Nous présentons sommairement ces deux méthodes dans ce qui suit.

Génération de cas de test à partir d'objectifs acceptants et refusants

Nous proposons que les objectifs de test temporisés soient composés de propriétés *acceptantes*, appartenant à la spécification, et de propriétés *refusantes* n'appartenant pas à la spécification. Ceci permet d'effectuer plusieurs types de test (conformité, robustesse). Un tel objectif de test est défini par :

Définition 2.1.5 (Objectif de test temporisé) *Soit* $A =< \Sigma_A, S_A, s^0_A, C_A, E_A >$ *un automate temporisé. Un objectif de test temporisé* $\mathcal{TP}$ *est un automate temporisé déterministe et acyclique* $< \Sigma_{\mathcal{TP}}, S_{\mathcal{TP}}, s^0_{\mathcal{TP}}, C_{\mathcal{TP}}, E_{\mathcal{TP}} >$ *tel que chaque état de* $S_{\mathcal{TP}}$ *est étiqueté soit par ACCEPT soit par REFUSE :*
Pour une transition $s \xrightarrow{?A} s' \in E_{\mathcal{TP}}$*, étiquetée par un symbole d'entrée,* s' *est toujours étiqueté par ACCEPT.*
Pour une transition $s \xrightarrow[Ct]{!A} s' \in E_{\mathcal{TP}}$*, étiquetée par un symbole de sortie, avec* Ct *la contrainte d'horloge,*

- *si* s' *est étiqueté par ACCEPT, il existe* $s \xrightarrow[Ct']{!A} s' \in E_A$ *tel que* $\forall v \in V(C_A), Ct' \wedge Ct(v) \models true$,
- *si* s' *est étiqueté par REFUSE, soit* $s \xrightarrow[Ct']{!A} s' \notin E_A$ *soit* $s \xrightarrow[Ct']{!A} s' \in E_A$ *et* $\forall v \in V(C_A), Ct' \wedge Ct(v) \models false$.

De façon synthétique, la génération des cas de test à partir d'une spécification et d'un ensemble d'objectifs de test est obtenue par les points suivants :

Soit $A =< \Sigma_A, S_A, s^0_A, C_A, E_A >$ un automate temporisé. Pour chaque chemin P d'un objectif de test TP :

- **Recherche de chemins de la spécification :** les chemins de A, composés dans le même ordre des transitions de P finies par un état étiqueté par ACCEPT, sont extraits. Ceci donne les chemins $TS_1, ..., TS_n$. Pour cela, nous utilisons un algorithme dérivé du DFS (Depth First Path Search),
- **Traduction en graphe des régions :** $TS_1, ..., TS_n$ et P sont transformés en graphes des régions minimaux $RG_1, ..., RG_n$, $RG(TP)$,

- **Produit synchronisé temporisé :** un produit synchronisé est effectué entre chaque graphe des régions $RG_1, ..., RG_n$, avec $RG(TP)$. La définition du produit synchronisé entre les transitions et les régions d'horloges est donnée dans [Sal02],
- **Recherche des chemins faisables :** les cas de test sont les chemins faisables extraits des produits synchronisés [BCF03]. Ceux-ci sont composés d'états terminaux étiquetés par un verdict local dans $\{pass, inconclusive, fail\}$.
  Le problème de faisabilité pour un chemin $p = t_1..t_n$ donné, a pour but de déterminer s'il existe une exécution possible permettant d'atteindre la transitions t_n, et de générer les zones d'horloges sur p pour exécuter t_n. L'approche, décrite dans [BCF03], ajoute une horloge globale h et effectue une analyse d'accessibilité sur le chemin initial. Les zones d'horloges du chemin faisable obtenu peuvent être décrites soit par l'horloge globale soit par les horloges d'origine.

Le verdict de test est obtenu en exécutant chaque cas de test sur l'implantation. Cette exécution est détaillée dans [Sal02, SL07].

Définition 2.1.6 (Verdict de test) *Soit $\mathcal{A}$ une spécification, $\mathcal{TP}$ un objectif de test, I une implantation sous test et TS la suite de test. L'exécution d'un cas de test $TC \in TS$ sur I, dénoté $vl(I, TC)$ retourne un verdict local tel que :*

$$vl(I,TC) = \begin{cases} \text{PASS ssi } \forall t = s \xrightarrow{!A, PASS(R), INCONCLUSIVE(R')} s' \in TC, \text{ avec } s' \text{ étiqueté par ACCEPT, le symbole } !A \text{ est observé à n'importe quelle valuation d'horloge de } R, \\ \text{PASS ssi } \forall t = s \xrightarrow{!A, PASS(R), INCONCLUSIVE(R')} s' \in TC, \text{ avec } s' \text{ étiqueté par REFUSE, le symbole } !A \text{ est observé à n'importe quelle valuation d'horloge } v \notin R \cup R' \text{ ou } B \neq A \text{ est reçu à la place de } A, \\ \text{FAIL ssi } \forall t = s \xrightarrow{!A, PASS(R), INCONCLUSIVE(R')} s' \in TC, \text{ avec } s' \text{ étiqueté par ACCEPT, le symbole } !A \text{ est observé à n'importe quelle valuation d'horloge } v \notin R \cup R' \text{ ou } B \neq A \text{ est reçu à la place de } A, \\ \text{FAIL ssi } \forall t = s \xrightarrow{!A, PASS(R), INCONCLUSIVE(R')} s' \in TC, \text{ avec } s' \text{ étiqueté par REFUSE, le symbole } !A \text{ est observé à n'importe quelle valuation d'horloge de } R, \\ INCONCLUSIVE \text{ autrement} \end{cases}$$

Le verdict de test, noté $V(I, \mathcal{TP})$ est défini par :

$$V(I,\mathcal{TP}) = \begin{cases} PASS, \text{ ssi } \forall TC \in TS, vl(I,TC) = PASS, \\ FAIL \text{ ssi } \exists TC \in TS, vl(I,TC) = FAIL, \\ \grave{I}NCONCLUSIVE \text{ autrement} \end{cases}$$

Génération de cas de test à partir d'objectifs de test et d'ensembles de caractérisation d'états

Dans [SL07], nous montrons que la méthode précédente, comme la plupart des méthodes orientées objectif de test, ne permet pas de couvrir l'ensemble du modèle de fautes des systèmes temporisés. Notamment les fautes de transfert et d'état manquant (section 1.3.4) ne peuvent être détectées car l'état interne de l'implantation n'est pas connu. Nous proposons d'ajouter une technique de caractérisation d'état à la méthode précédente pour détecter ces erreurs.

Dans [SL07], nous supposons que la spécification est décrite par un automate à intervalles (section 1.3.1). Les objectifs de test sont définis de la même façon que précédemment (Définition 2.1.5) mais restreints à des automates à intervalles également.

La spécification $\mathcal{S}$ est supposée déterministe, minimale, complète et fortement connexe (hypothèses fortes mais cependant nécessaires pour construire, avec un temps fini, les ensembles de caractérisation d'états). La génération des cas de test est effectuée de la façon suivante :

- **Recherche de chemins de la spécification :** les chemins contenant dans le même ordre les transitions de l'objectif de test terminées par un état "ACCEPT" sont extraits, ce qui donne $TS_1(\mathcal{S}), ..., TS_n(\mathcal{S})$,
- **Produit synchronisé temporisé :** chaque chemin $TS_i(\mathcal{S})$ est ensuite synchronisé avec $\mathcal{TP}$. Cette opération retourne un automate à intervalles SP, qui inclus $\mathcal{TP}$ et qui respecte les propriétés comportementales et temporelles de $\mathcal{S}$,
- **Génération de l'ensemble de caractérisation d'états :** chaque état S_i de SP est caractérisé avec W_{S_i} (cf Section 2.1.1). Puis, nous combinons W_{S_i} avec SP : $\Pi = SP \otimes W = \{p.(s_i, s_j, a, \lambda, Z).\sigma_j \mid \forall (s_i, s_j, a, \lambda, Z) \in E_{SP}$, p est un chemin de SP de s_0 à s_i, $\sigma_j \in W_{s_j}$ si s_j est étiqueté par ACCEPT, $\sigma_j = \emptyset$ autrement$\}$. Cela correspond à la concaténation du chemin p terminé par un état s_i avec l'ensemble de caractérisation de s_i,
- **Recherche des chemins faisables :** les cas de test sont les chemins faisables de Π [BCF03].

Intuitivement, l'exécution des cas de test et le verdict sont obtenus de façon identique à la méthode précédente. L'avantage obtenu est que les fautes couvertes par cette méthode sont plus nombreuses.

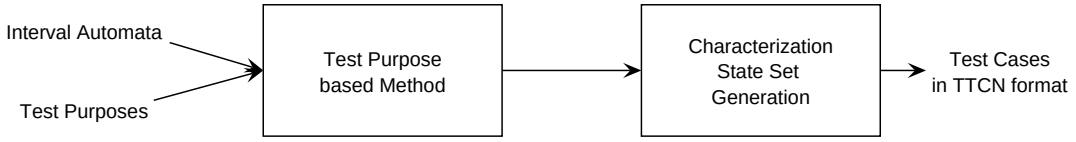


FIGURE 2.16 – L'outil TTCG

Un prototype d'outil, nommé TTCG (Timed Test Cases Generation), a été implanté. Son architecture est illustrée en Figure 2.16. Celui-ci est composé de deux parties : la première construit le produit synchronisé entre les objectifs de test et les chemins de la spécification. La seconde partie construit l'ensemble de caractérisation d'états comme nous l'avons décrit en section 2.1.1. Les cas de test sont ensuite donnés au format TTCN ou Poscript. Cet outil a été écrit en langage C avec une modélisation des zones d'horloges faites grâce à la librairie Polylib [Wil]. Nous avons effectué plusieurs générations de cas de test avec succès sur les protocoles MAP-DSM et audio de Philips.

2.1.3 Génération d'objectifs de test temporisés

De nombreuses méthodes de test ont été proposées pour les systèmes temporisés. Afin de réduire le coût de test et d'éviter une explosion combinatoire de l'espace des états lorsque la suite de test est construite, plusieurs méthodes sont orientées objectif de test

[CCKS95, PLC98, FPS00, Kon01, BCF03, END03, CM04]. Celles-ci testent des parties locales des systèmes depuis les besoins de test, appelés objectif de test. La conclusion du test est fournie en vérifiant la satisfaction de l'objectif de test sur l'implantation. Bien que ces méthodes offrent plusieurs avantages (réduction du coût, etc.), il est supposé que les objectifs de test sont construits manuellement. En plus de devenir rébarbative, cette construction peut aussi devenir un casse-tête si le système est complexe, temporisé, interopérable, etc. Ainsi, des méthodes de génération automatique d'objectifs de test ont été proposées, mais uniquement pour les systèmes non temporisés [CCKS95, HLU03a].

Nous avons proposé donc des méthodes de génération d'objectifs de test semi automatique et automatique. Nous nous sommes penchés sur les systèmes temporisés et protocoles de communication, modélisés par des automates à intervalles (section 1.3.1) [SL05, SR09b].

Dans la suite, nous présentons sommairement la définition d'un langage de modélisation d'objectifs de test qui prend en compte la testabilité et les méthodes de génération.

Définition et modélisation des objectifs de test

Pour construire des objectifs de test testables, nous avons proposé de prendre en compte la testabilité de la spécification. Pour ce faire, nous avons défini un langage rationnel composé de mots clés et d'opérateurs. Ces derniers ont l'avantage de prendre en compte une liste de degrés de testabilité et de décrire facilement un ou plusieurs objectifs de test en une seule expression. Nous considérerons aussi que les objectifs de test peuvent contenir des propriétés temporelles et comportementales, soit inclues dans la spécification et appelées alors propriétés *Acceptante*, soit des propriétés qui n'appartiennent pas à la spécification et appelées propriétés de *Refus*.

Définition 2.1.7 (Langage de description d'objectifs de test) *Soit* $\mathcal{AJ} =< \Sigma_{\mathcal{AJ}}, S_{\mathcal{AJ}}, s^0_{\mathcal{AJ}}, C_{\mathcal{AJ}}, I_{\mathcal{AJ}}, E_{\mathcal{AJ}} >$ *un automate à intervalles. Les mots Etiquette, Etat et Chemins sont définis par :*

$Etiquette ::=$	$\langle a, Z\rangle$ *avec* $a \in \Sigma_{\mathcal{AJ}}, Z \in I_{\mathcal{AJ}}$
$Etat_k ::=$	(s_k, l_k) *avec* $s_k \in S_{\mathcal{AJ}}, l_i \in \{accept, refuse\}$
$Transition ::=$	$(s_i, accept).\langle a, Z_2\rangle.(s_k, l_k)$ *avec* $s_k \in S_{\mathcal{AJ}}, l_k = accept$ *iff* $\exists (s_j, s_k, a, \lambda, Z) \in E_{\mathcal{AJ}}$ *tel que* $\forall v, v \models Z_2 \Rightarrow v \models Z$, $l_i = refuse$ *autrement, et* $s_i = s_k$
noloop	$Etat_i.noloop.Etat_j$ *représente un ensemble de chemins P depuis* s_i *vers* s_j *ne contenant pas de boucle :* $\forall Path \in P, \forall p, p', q \mid Path = p.q.p', s_i \xrightarrow{p} s_k \xrightarrow{q} s_l \xrightarrow{p'} sj \Longrightarrow s_k \neq s_l$
max	$Etat_i.max_{(x_1,...x_n)}.Etat_j$ *représente un chemin p de* $Etat_i.noloop.Etat_j$ *tel que chaque degré de testabilité* $x_i(p)$ *est le plus proche de 1.*
Pre	$Pre_{(x_1,...x_n)}.Etat_j$ *représente un chemin partant de l'état initial vers l'état* s_j. $Pre_{(x_1,...x_n)}.Etat_j = (s^0_{\mathcal{AJ}}, accept).max_{(x_1,...x_n)}.Etat_j$.
Post	$Etat_j.Post_{(x_1,...x_n)}$ *représente la réinitialisation du système. Si celui-ci possède une fonction de réinitialisation* $reset()$ *alors* $Etat_j.Post_{(x_1,...x_n)} = reset()$, *sinon* $Etat_j.Post_{(x_1,...x_n)} = Etat_j.max_{(x_1,...x_n)}.Etat_0$
$Chemin ::=$	$\emptyset \mid Transition \mid Etat_i.noloop.Etat_j \mid Etat_i.max_{(x_1,...x_n)}.$ $Etat_j \mid Pre_{(x_1,...x_n)}.Etat_j \mid Etat_j.Post_{(x_1,...x_n)} \mid Chemin^*$ $\mid Chemin \cup Chemin$
$Obj : :=$	$Pre.Chemin.Post$ *représente un objectif de test composé d'un préambule d'un chemin et d'un postambule de la spécification*

Les objectifs de test, décrits par ce langage rationnel, peuvent être ensuite facilement retranscrits sous forme d'un ou plusieurs chemins de la spécification, les expressions décrites par ce langage étant toujours composées d'états et de transitions. Ceci permet donc d'obtenir, à partir d'une expression, des objectifs de test sous forme d'automates qui soient utilisables par les méthodes de test orientées objectif de test [CCKS95, PLC98, FPS00, Kon01, BCF03, END03]. Pour prendre en compte les propriétés acceptantes et de refus, nous avons défini des objectifs de test par :

Définition 2.1.8 (Objectif de test temporisé) *Soit* $\mathcal{AJ} =< \Sigma_{\mathcal{AJ}}, S_{\mathcal{AJ}}, s^0_{\mathcal{AJ}}, C_{\mathcal{AJ}}, I_{\mathcal{AJ}}, E_{\mathcal{AJ}} >$ *un automate à intervalles. Un objectif de test temporisé* $\mathcal{TP}$ *est un automate à intervalles acycliques* $< \Sigma_{\mathcal{TP}}, S_{\mathcal{TP}}, s^0_{\mathcal{TP}}, C_{\mathcal{TP}}, I_{\mathcal{TP}}, E_{\mathcal{TP}} >$ *tel que chaque état* s' *de* $S_{\mathcal{TP}}$ *est étiqueté par :*

- *ACCEPT : si* $\exists$ *transition* $(s, s', a, \lambda, Z) \in E_{\mathcal{TP}}, \exists (s_1, s_2, a, \lambda_2, Z_2) \in E_{\mathcal{AJ}}$ *tel que* $\forall v, v \models Z \Rightarrow v \models Z_2$,
- *REFUSE : autrement*

Génération d'objectifs de test temporisés

Les méthodes suivantes permettent d'automatiser la construction d'objectifs de test en prenant en compte la notion de testabilité afin de détecter un maximum d'erreurs sur l'implantation avec les objectifs de test obtenus et de minimiser les coûts du test. Chaque méthode est composée d'une ou plusieurs expressions, obtenues par le langage défini précédemment. Ces expressions produisent les objectifs de test, qui peuvent être

traduits sous forme d'automates. Comme dans [CCKS95], deux types de méthodes sont proposées :

- les deux premières sont semi-automatiques. Dans ce cas, il est nécessaire de choisir des états ou transitions critiques avant que les objectifs de test soient construits. Cela permet d'orienter avec précision les tests.
- les 2 méthodes suivantes génèrent des objectifs de test automatiquement. Chacune cible des propriétés à tester, comme le test de services inclus et invoqués par le système et le test d'états critiques.

Génération semi-automatique d'objectifs de test à partir d'états critiques

Cette méthode construit des objectifs de test pour vérifier que chaque état d'un ensemble S_F est accessible depuis un état s_{init}. Cette méthode est très utile lorsque les états d'un système ont une signification précise.

Soit $\mathcal{AI} =< \Sigma_{\mathcal{AI}}, S_{\mathcal{AI}}, s^0_{\mathcal{AI}}, C_{\mathcal{AI}}, I_{\mathcal{AI}}, E_{\mathcal{AI}} >$ un automate à intervalles modélisant la spécification. Soit aussi un état $s_{init} \in S_{\mathcal{AI}}$ et un ensemble d'états $S_F = \{q_1, ..., q_m\} \subset S_{\mathcal{AI}}$. Les objectifs de test doivent permettre de tester si tous les états de S_F peuvent être atteints à partir de s_{init}. Les objectifs de test sont donnés par l'expression suivante :

$$Obj ::= \bigcup_{1 \leq j \leq m} Pre_{(x_1,...,x_n)}.(s_{init}, accept).noloops.(q_j, accept).Post_{(x_1,...,x_n)}$$

Nous obtenons un ou plusieurs objectifs de test qui testent si chaque état $q_j \in S_F$ peut être visité depuis s_{init} avec tous les chemins sans cycle de s_{init} à q_j. Le préambule le plus testable est utilisé pour atteindre s_{init} depuis l'état initial s_0 du système, et le postambule le plus testable est utilisé pour le réinitialiser.

Génération semi-automatique d'objectifs de test à partir de transitions critiques

Cette méthode a pour but de tester une suite de transitions (pas forcément consécutives) dans l'implantation. Elle est utile lorsque les symboles des transitions ont une signification précise. Ces transitions $(q_i, q_{i+1}, a_i, \lambda_i, Z_i)$ peuvent être du type :

- *Acceptante* : si q_{i+1} est étiqueté par "accept". Cette transition, composée de propriétés de la spécification sera testée dans l'implantation avec l'objectif de test.
- *Refusante* : si q_{i+1} est étiqueté par "refuse" et $q_i = q_{i+1}$. Si a_i est un symbole d'entrée, cette transition teste la robustesse du système, en émettant un symbole non spécifié. a_i peut aussi être un symbole de sortie, dans ce cas cette transition permet de tester que a_i ne doit pas être observé.

Soit $\mathcal{AI}$ un automate à intervalles modélisant la spécification. Soit aussi une liste de transition $T = \{(q_1, q_2, a_1, \lambda_1, Z_1)... \ (q_n, q_{n+1}, a_n, \lambda_n, Z_n)\}$. A partir de l'état initial s_0 de la spécification, les objectifs de test doivent exécuter successivement toutes les transitions de T, puis doivent réinitialiser le système. Les objectifs de test sont donc obtenus par l'expression suivante :

$$Obj ::= Pre_{(x_1,...,x_n)}.(q_1, accept). < a_1, Z_1 > .(q_2, l_2).F_2...(q_i, accept). < a_i, Z_i > .(q_{i+1}, l_{i+1}).$$
$$F_{i+1}...(q_n, accept). < a_n, Z_n > .(q_{n+1}, l_{n+1}).Post_{(x_1,...,x_n)}$$

où $F_i ::= max_{(x_1,...,x_n)}.(q_{i+1}, accept)$

Avec cette expression, la méthode génère un seul objectif de test. La première transition de T est atteinte par le préambule le plus testable. Les suivantes sont atteintes par un chemin donné par l'opérateur max dans l'expression F_i.

Objectifs de test pour le test de services :

Cette méthode essaie de reconnaître toutes les invocations de services dans la spécification et construit des objectifs de test pour les tester.

Cette méthode commence par rechercher, dans la spécification, les suites d'étiquettes $l_1, ..., l_n$ qui représentent des invocations de service. Ensuite, pour chaque symbole l_i, elle recherche les transitions qui sont étiquetées par l_i. Ainsi, nous obtenons une liste d'ensembles de transitions $L_1, ..., L_n$ tel que $L_i = \{(s, s', a, \lambda, Z) \mid a = l_i\}$. Pour chaque tuple de transitions $(t_1, ...t_n) \in L_1 \times ... \times L_n$, elle construit un objectif de test qui peut exécuter successivement chaque transition t_i. (Chacun de ces tuples représente une invocation possible d'un service dans le système.)

Considérons la suite de transitions $((s_1, s'_1, a_1, \lambda_1, Z_1), ...(s_n, s'_n, a_n, \lambda_n, Z_n)) \in L_1 \times ... \times L_n$ qui représente une invocation complète d'un service. L'objectif de test, qui aura pour but de la tester, doit atteindre successivement chaque transition pour l'exécuter. De ce fait, l'expression d'objectif de test est de la forme :

$$Obj ::= \begin{cases} \emptyset \text{ si } \forall(1 \leq i < n),\ (s'_i, accept).max_{(x_1,...,x_n)}.(s_{i+1}, accept) = \emptyset \\ Pre_{(x_1,...,x_n)}.F_1....F_{n-1}.(s_n, accept)\langle l_n, Z_n\rangle.(s'_n, accept). \\ Post_{(x_1,...,x_n)},\ \text{autrement} \end{cases}$$

où $F_i(1 \leq i < n) ::= (s_i, accept).\langle l_i, Z_i\rangle.(s'_i, accept).max_{(x_1,...,x_n)}.(s_{i+1}, accept)$

Avec cette expression, nous produisons un chemin qui invoque complètement le service à partir de l'état initial du système si chaque transition $(s_i, s'_i, a_i, \lambda_i, Z_i)$ est accessible depuis la précédente $(s_{i-1}, s'_{i-1}, a_{i-1}, \lambda_{i-1}, Z_{i-1})$. Autrement $(max = \emptyset)$, le service ne peut être complètement invoqué, donc l'objectif de test correspondant n'est pas généré.

Génération d'objectifs de test pour le test d'états critiques

Cette méthode essaie de reconnaître les états critiques du système, puis construit des objectifs de test pour tester leurs conformités ainsi que leurs robustesses.

Avec cette méthode, nous supposons qu'un critère est défini pour caractériser les états critiques, comme par exemple, les états les plus visités depuis l'état initial. Pour détecter les états critiques, nous utilisons l'algorithme de parcours en profondeur (DFS) modifié, sur la spécification $\mathcal{S}$. Tout en construisant les chemins de $\mathcal{S}$ ne contenant pas de cycle, l'algorithme donné en [SR09b] compte le nombre de fois ou est référencé chaque état.

Des objectifs de test sont ensuite construits pour tester toutes leurs transitions sortantes. Pour un état critique s, l'expression d'objectif de test est la suivante :

$$Obj ::= \bigcup_{(s,s',a,\lambda,Z)} Pre_{(x_1,...,x_n)}.(s, accept).\langle a, Z\rangle.(s', accept).Post_{(x_1,...,x_n)}$$

Cette expression produit un objectif de test pour chaque transition sortante de l'état critique s. Celui-ci a pour but d'atteindre la transition sortante avec le préambule le plus

testable, de l'exécuter, puis de réinitialiser le système avec le postambule le plus testable.

Pour tester la robustesse des états critiques, la méthode produit des objectifs de test qui ont pour but d'atteindre chacun de ces états, puis de fournir au système soit des symboles d'entrée non attendus (appartenant à l'alphabet de la spécification), soit des symboles d'entrée attendus mais avec des contraintes temporelles non permises. Pour un état critique s, les expressions d'objectif de test sont :

$Obj1 ::= \{Pre_{(x_1,...,x_n)}.(s, accept).\langle ?a, Z\rangle.(s, refuse).\langle b, Z\rangle.(s', accept).$
$Post_{(x_1,...,x_n)} \mid \forall s'', (s, s'', ?a, \lambda, Z) \notin E_S, \exists (s, s', b, \lambda, Z) \in E_S\}$

$Obj2 ::= \{Pre_{(x_1,...,x_n)}.(s, accept).\langle ?a, Z\rangle.(s, refuse).\langle ?a, Z'\rangle.(s', accept).$
$Post_{(x_1,...,x_n)} \mid \exists (s, s', ?a, \lambda, Z') \in E_S$ avec $Z = \{[a_1\ b_1]...[a_n\ b_n]\}, Z' = \{[b1\ b1 + b1 - a1]...[bn\ bn + bn - an]\}\}$

La première expression $Obj1$ produit un objectif de test pour chaque symbole d'entrée non spécifié, c'est-à-dire un symbole d'entrée appartenant au langage de la spécification mais non attendu par l'état critique s. Ces objectifs de test ont pour but d'atteindre l'état s, de produire un symbole non spécifié, puis spécifié (pour vérifier le fonctionnement correct du système). La seconde expression produit un objectif de test pour chaque symbole d'entrée que s peut accepter. La contrainte temporelle est cependant changée par l'intervalle $Z = \{[b1\ b1 + b1 - a1]...[bn\ bn + bn - an]\}$. Celui-ci est composé de valeurs supérieures à l'intervalle de temps de la spécification. En effet, pour un intervalle $[a_i\ b_i]$, nous testons avec l'intervalle $[b_i\ b_i + b_i - a_i]$ dont la durée est équivalente à celle de $[a_i\ b_i]$.

Application sur les compositions et perspectives

D'autres stratégies peuvent être proposées pour tester d'autres propriétés de la spécification : par exemple, le test de contraintes temporelles spécifiques, le test de sous-systèmes (cycles fonctionnels), ou le test de l'observabilité du système. D'autres méthodes peuvent aussi être proposées pour d'autres systèmes, comme les systèmes parallèles et distribués.

D'ailleurs, nous avons repris l'idée de génération automatique d'objectifs de test pour les compositions de service dans [Sal11b]. Dans cette approche, les objectifs sont générés directement grâce à des règles d'inférences sur des compositions de services modélisées par des STS. Les propriétés prises en compte pour générer les objectifs sont différentes des précédentes et prennent en compte les spécificités des services comme les opérations, l'environnement SOAP, les exceptions.

2.2 Les applications orientées service

Nous présentons, dans cette section, les travaux sur le test et la testabilité des applications orientées service (services Web, compositions ABPEL) que nous avons effectués depuis 2007, à travers notre implication dans le projet ANR Webmov (`http://webmov.lri.fr/`) et l'encadrement de la thèse D'Issam Rabhi.

Pour situer nos différentes contributions dans ce contexte, rappelons sommairement ce qu'est un service Web. D'après Tidwell, les services Web sont *des applications modulaires, qui se suffisent à elles-mêmes, qui peuvent être publiées, distribuées sur Internet* [Tid00].

Afin de rendre les services Web interopérables, l'organisation WS-I a proposé des profils, en particulier le profil WS-I Basic [org06]. Celui-ci est composé de quatre grands axes :

- la *description du service* présente les informations nécessaires pour invoquer un service, en définissant ses interfaces appelées (*endpoints*), les méthodes appelées opérations qui sont fournies par le service, et les paramètres/réponses de ses opérations. Cette description est donnée au format XML grâce au langage WSDL (*Web Services Description Language*), qui permet de décrire les structures des messages et de définir les types complexes utilisés par ces derniers,
- la *définition et la construction des messages XML*, sont basées sur le protocole SOAP (*Simple Object Access Protocol*) [Con03]. SOAP est utilisé pour invoquer les opérations de services Web dans un réseau en sérialisant/désérialisant les données. SOAP est aussi représenté par des processeurs SOAP, placés dans toute plate-forme de service Web (SOAP), qui traitent de la réception/transmission de messages entre les clients et les services Web. Le protocole SOAP introduit aussi la notion de faute SOAP. Comme il est défini dans [Con03], une faute SOAP est utilisée pour avertir le client qu'une erreur s'est produite (sur le réseau, le serveur ou le service). Une faute SOAP est composée d'un code, d'un message, d'une cause et éventuellement d'éléments XML donnant les détails sur l'erreur. Typiquement, une telle faute est obtenue en POO, après la levée d'une exception. Ces fautes ne sont pas nécessairement données dans les descriptions WSDL.
  SOAP peut être placé au-dessus de différentes couches transport : le HTTP qui est souvent utilisé des appels synchrones de service, ou SMTP qui est souvent utilisé pour des appels asynchrones,
- la *découverte du service* dans des registres UDDI. Les descriptions de service Web sont réunies dans ces registres UDDI (*Universal Description, Discovery Integration*) qui peuvent être consultés manuellement ou automatiquement en utilisant des API de programmation pour rechercher dynamiquement un service Web spécifique,
- La *sécurité du service*, est obtenue grâce aux protocole HTTPS et au chiffrage XML.

Les services Web peuvent aussi être de type stateless ou stateful : dans le premier cas ils ne comportent pas d'état, les opérations peuvent être appelées sans suite logique. Un service stateful possède un état interne qui évolue au fil des invocations

Dans les approches présentées ci-dessous, nous considérerons des services Web stateless ou stateful en boite noire à partir desquels seuls les messages SOAP sont observables (les requêtes de base de données sont ignorées).

Nos première contribution concerne la testabilité. De façon originale, nous avons proposé des travaux sur l'étude de la faisabilité des tests des services Web et de compositions ABPEL. Cette étude montre notamment quelles propriétés peuvent être testées à partir de signatures et de spécifications. Ces travaux ont conduit à la définition de quelques méthodes de test actives. Nous avons proposé des approches de génération de cas de test à partir de signatures. A la différence des travaux tels que [OX04, Mar06, BDTC05, BBMP09a], nous avons étudié le test de l'usager (existence des opérations, gestion des exceptions, gestion de la persistance) et le test de robustesse en prenant en compte les informations supplémentaires fournies par la couche SOAP. Ces informations permettent d'affiner le verdict en séparant les réactions de l'environnement SOAP des réactions du service. Nous avons également proposé de tester la sécurité des services Web de façon active. Notre apport dans ce domaine, notamment par rapport

aux travaux de Mallouli et al. [MMC09], est de prendre en compte des règles de sécurité complexes dans des patrons de test, qui sont ensuite traduits en objectifs de test. Une fois de plus, nous nous focalisons également sur l'environnement SOAP afin de détecter des erreurs provenant uniquement du service sous test. Cette approche par objectif permet également de réduire les coûts.

Ces travaux sont résumés ci-dessous.

2.2.1 Testabilité des services Web stateless

Avant de se pencher sur le test de services Web stateless (sans état interne), nous nous sommes penchés sur ce que pouvait être un service testable. Ceci a donné lieu à la publication [SR08b].

Bien que simple au premier abord, un service Web stateless n'est accessible qu'à travers une couche SOAP-XML qui réduit sa visibilité (observabilité et contrôlabilité). De plus, un service est connu en interprétant et en faisant confiance à sa description WSDL. Ces langages et protocoles modifient la faisabilité du test, autrement appelée testabilité. Nous avons donc étudié cette testabilité en considérant différents aspects comme le code du service, les messages SOAP qui permettent d'interagir avec le service, et la description WSDL qui modélise son interface. A partir des critères d'observabilité et de contrôlabilité, nous avons montré quelles sont les propriétés qui réduisent la testabilité et donnons les définitions d'un service Web observable, contrôlable et testable.

Afin de pouvoir raisonner sur les services Web, nous avons choisi de les formaliser sous forme de relations qui permettent de décrire les opérations et les types de paramètres d'un service à partir de sa description WSDL.

Définition 2.2.1 (Service Web) *Soit X un ensemble de variables et T un ensemble de types (Integer, String, Object, SOAPfault, etc.). Un ensemble de variables est associé à un ensemble de types par la fonction type : $X^n \rightarrow T^n$.*

Un service Web WS est un composant qui peut être invoqué à partir d'un ensemble d'opérations $OP(WS) = \{op_1, ..., op_k\}$, avec op_i définie par la relation $op_i : from(op_i) \rightarrow to(op_i)$ avec $from(op_i) = (x_1, ..., x_n) \in X^n$ et $to(op_i) = (x'_1, ..., x'_m) \in X^m$.

Soit U un ensemble de valeurs. La valuation val est la relation $val : X^n \rightarrow U^n$ donnant un ensemble de valeurs à partir d'un ensemble de variables. En particulier, la valuation de la variable x avec $type(x) = SOAPfault$, est notée $val(x) = (c, soapfault)$ avec c la cause de la faute. Pour simplifier, nous notons le type d'un ensemble de valeurs v, $type(v)$ avec $v = val(x)$ et $type(v) = type(x)$.

Nous définissons la restriction $E_{|X^n} = \{val \in E \mid val : X^n \rightarrow E\}$, qui conduit à la formalisation de l'invocation d'opérations : $inv_{op_i} : U^n_{|from(op_i)} \mapsto U^m_{|to(op_i)} \cup \{(c, soapfault)\} \cup \{\epsilon\}$. ϵ modélise le fait de ne pas recevoir de réponse. Pour simplifier, inv_{op_i} est aussi noté op_i.

Observabilité des services Web stateless

Dans [Fre91], un système est défini comme étant observable "si pour chaque entrée donnée au système, une sortie différente est observée". Cette définition correspond exactement à l'injectivité pour les relations. Un service web pourrait donc être dit observable si toutes ses opérations sont injectives.

Cependant, un service Web "baigne" dans un environnement imposé par le profil WS-I basic. Et selon ce dernier, un service Web est un composant tel que :

- le service est appelé, *via* des messages SOAP, par l'une de ses opérations *op* et par une liste de paramètres $(p_1, ...p_m)$. Ainsi, une entrée sera le couple $(op,(p_1, ...p_m))$, et la sortie $(op,(r_1, ..., r_n))$,
- chaque opération peut retourner des fautes SOAP, notamment (mais pas nécessairement) si une exception est levée au sein du service. Cette exception a généralement pour but d'indiquer qu'une erreur est survenue (problème de base de données, de thread, de socket...). Le fait d'observer le déclenchement des exceptions est donc nécessaire pour déterminer le comportement d'un service, notamment son comportement en cas d'erreurs.

Nous avons donc analysé l'injectivité des opérations de service et la gestion des exceptions en rapport avec la définition d'observabilité précédente. Ceci nous a permis de proposer une nouvelle définition de l'observabilité d'un service Web stateless :

Définition 2.2.2 *Un service Web WS est observable si pour chaque opération $op \in OP(WS)$:*

- *l'opération op, décrite dans la description WSDL peut être invoquée et retourne une réponse non nulle et lisible,*
- *op est injective,*
- *pour chaque invocation $r = op(p)$, avec $p \in from(op)$ et $r \in to(op)$ provoquant une exception, r est une faute SOAP unique, construite avec p.*

Contrôlabilité des services Web stateless

La contrôlabilité correspond à l'aptitude de forcer le système à effectuer certaines tâches et donc à retourner certaines réponses. Dans [Fre91], un système est défini comme contrôlable "si pour chaque sortie, il existe une entrée qui force l'observation de cette sortie". Cette définition correspond exactement à une relation surjective.

Pour qu'une opération de service, modélisée par une relation, soit surjective, il est nécessaire qu'elle soit déterministe et que pour chaque réponse, il existe une liste de paramètres permettant d'obtenir cette réponse. Ceci tend à proposer une définition préliminaire d'un service Web contrôlable.

Définition 2.2.3 *Un service web WS est contrôlable ssi pour toute opération $op \in OP(WS)$, op est surjective. Une opération op (relation) est surjective (en algèbre) si :*

- *op est déterministe,*
- *pour toute réponse $r \in R(op)$ (fautes SOAP incluses), il existe une invocation $r = op(p)$ avec $p \in P(op)$ tel que $r = op(p)$.*

Nous avons ensuite analysé les deux points de cette définition et avons déduit la nouvelle définition suivante :

Définition 2.2.4 *Un service web WS est contrôlable ssi pour toute opération $op \in OP(WS)$:*

1. *op est déterministe : pour toute invocation $r = op(p)$ avec $p \in P(op)$ et $r \in R(op)$, r est unique. Notamment, si op lève une exception déclenchée par les paramètres $p \in P(op)$, alors la faute SOAP retournée $r \in R(op)$ est unique,*

2. op ne lève pas d'exceptions indépendantes de l'appel.

Une grande majorité de services Web utilisent des accès à des bases de données, ou utilisent des fichiers, des threads..., et ceux-ci peuvent lever des exceptions indépendantes de l'appel du service. Ceux-ci ne sont donc pas contrôlables et pas testables.

Or si l'on considère que le test de conformité s'effectue sur une plate-forme particulière sans perturbation, nous pouvons estimer que très peu d'erreurs externes, levant ces exceptions, se produiront. Dans ce cas, cette définition est peut être trop forte. Voilà pourquoi nous avons proposé la définition de semi-contrôlabilité, pour laquelle les exceptions indépendantes de l'appel ne sont pas prises en compte. Ainsi :

Définition 2.2.5 *Un service web WS est semi-contrôlable ssi pour toute opération $op \in OP(WS)$ op est déterministe : pour toute invocation $r = op(p)$ avec $p \in P(op)$ et $r \in R(op)$, ne déclenchant pas d'exception levée à cause d'événements extérieurs, il existe une réponse unique $r \in R(op)$. WS est testable s'il est semi-contrôlable et observable.*

Les exceptions levées par des événements extérieurs aux services Web posent problème. Peut-on les enlever ? Dans la grande majorité des cas, c'est fortement déconseillé. En effet, si une erreur survient au sein du service web, il est préférable que l'utilisateur le sache, il est donc préférable que cette erreur soit observable.

Testabilité des services Web appelés en mode asynchrone

Nous avons également étudié le mode asynchrone qui permet d'appeler des services Web en mode non bloquant. De ce fait, si une réponse est donnée après plusieurs minutes ou heures, l'application cliente peut continuer à s'exécuter. En mode asynchrone, les réponses peuvent donc être reçues dans un ordre différent des appels. Nous avons obtenu la définition suivante :

Définition 2.2.6 *Un service web WS est testable si pour toute opération $op \in OP(WS)$ en mode asynchrone, pour toute invocation $r = op(p)$ il est possible d'associer la réponse $r \in to(op)$ à la liste de paramètres $p \in from(op)$.*

Ce travail a ainsi montré que le profil WS-I basic garantit l'interopérabilité des services mais aussi réduit leur testabilité. En effet, les services Web SOAP peuvent retourner des fautes SOAP et celles-ci peuvent rendre un service web indéterministe, inobservable et incontrôlable. Ces fautes SOAP sont rarement considérées dans les spécifications et dans les méthodes de test. Cependant, elles font partie intégrante de leurs comportements et peuvent fortement compliquer l'étape de test. C'est pourquoi nous avons toujours pris en compte cette notion d'environnement SOAP dans les travaux suivants, traitant du test de services.

2.2.2 Test automatique de services Web stateless

Nous avons étudié les possibilités de test automatique sur des services Web stateless (sans état, sans session) en boite noire. Ceci a donné lieu à deux travaux sur le test de l'usager [SR08a] et le test de robustesse [SR09a]. L'avantage manifeste de ces méthodes est d'automatiser les tests et leurs exécutions afin de vérifier diverses propriétés comme

l'existence des opérations, la gestion des exceptions ou le fait d'accepter certains aléas pour la robustesse.

Nous avons modélisé les services Web sous forme de relation grâce à la définition 2.2.1 donnée précédemment. Dans ces deux travaux nous avons mis l'accent sur les processeurs SOAP qui modifient (augmentent) le comportement d'un service Web. En effet, ceux-ci qui sont intégrés dans chaque Plate-forme de service Web (Apache Axis, Sun Metro, etc.) ajoutent des messages envoyés vers le Client suivant différentes interactions avec les clients ou le serveur. Notamment, les processeurs SOAP ajoutent des SOAP faults lorsqu'un service Web "plante" par exemple. Cependant, le WS-I basic profile fournit les informations permettant de différentier le comportement d'un service Web de celui d'un processeur SOAP. Ainsi, lorsqu'une exception est levée, le code du service Web doit normalement construire une faute SOAP. Dans ce cas, la cause est égale à "SOAPFaultException". Autrement, les fautes SOAP sont générées par le processeur SOAP. De ce fait, nous avons défini la robustesse d'un service Web stateless en décrivant que toutes ses opérations doivent retourner soit une réponse dont le type est celui de la spécification soit une faute SOAP dont la cause est "SOAPFaultException" :

Définition 2.2.7 *Soit WS un service Web. Une opération $op_i : from(op_i) \rightarrow to(op_i) \in OP(WS)$ est robuste, ssi $\forall v \in U^n_{from(op_i)}$, $op_i(v) \in U^m_{to(op_i)} \cup \{(SOAPFaultException, soapfault)\}$.*

Dans [SR08a], nous avons étudié le test de l'utilisateur, par les propriétés suivantes :

- *Existence des opérations* : pour chaque opération $op_i : from(op_i) \rightarrow to(op_i) \in OP(WS)$, nous construisons des cas de test pour vérifier si les opérations implantées correspondent à la spécification obtenue à partir de la description WSDL. Ainsi, les cas de test ont pour but d'appeler l'opération op_i avec des tuples de valeurs dont le type est $type(from(op_i))$. La réponse doit être de type $type(to(op_i))$ ou une faute SOAP dont la cause est différente de "Illegal Argument" ou "Client", ces causes signifiant que les types donnés ne sont pas bons ou que l'opération n'existe pas.
- *Gestion des Exceptions* : pour chaque opération $op_i : from(op_i) \rightarrow to(op_i) \in OP(WS)$, nous générons des cas de test pour tenter la levée d'exceptions en appelant op_i avec des valeurs inhabituelles bien connues pour détecter des bugs. Par exemple, pour le type "String", nous pouvons utiliser "", null, "*", "$". Nous devons recevoir une faute SOAP construite par le service Web avec la cause "SOAPFaultException". Autrement, le service Web ne gère pas correctement les exceptions, celles-ci étant traduites en fautes SOAP par les processeurs SOAP et non pas par le service,
- *Gestion de session* : nous supposons que nous avons un service Web WS qui possède des "getteurs/setteurs" décrits par les expressions $set-[opname]$ and $get-[opname]$, permettant de fournir et récupérer des données qui persistent avec l'instance du service. Nous supposons alors que le service Web fonctionne par session. Dans ce cas, nous testons l'existence d'une telle session en construisant des cas de test qui ont pour but d'appeler $set - [opname]$ avec des valeurs aléatoires puis d'appeler $get - [opname]$ et de vérifier que les données reçues sont égales à celles envoyées. Dans le cas contraire, la gestion de session est considérée comme incorrecte.

Concernant le test de robustesse automatique, que nous avons étudié dans [SR09a], nous avons recherché les aléas (actions non spécifiées) qu'il était possible d'utiliser sur une

opération $op_i : from(op_i) \rightarrow to(op_i) \in OP(WS)$. Nous avons conclu que les aléas "remplacer un type de paramètre", "ajouter un type de paramètre", "supprimer un type de paramètre" et "intervertir des paramètres" ne peuvent être utilisés car ces aléas contredisent la description WSDL du service Web et sont donc tout bonnement bloqués par le processeur SOAP. Par conséquent, le service Web n'est pas invoqué et pas testé. Le seul aléa utilisable est l'"appel d'opérations avec des données inhabituelles". Cet aléa, connu dans le test du logiciel [KKS98], a pour objectif d'appeler une opération op_i avec des valeurs de type $typ(from(op_i))$ qui sont inhabituelles. Cet aléa ne contredit pas la description WSDL et n'est pas bloqué par le processeur SOAP. Il peut donc être utilisé.

Ainsi, la méthode de test décrite dans [SR09a], vérifie l' *Existence des opérations* comme décrit précédemment et la robustesse de chaque opération d'après la définition de robustesse précédente avec l'aléa "appel d'opérations avec des données inhabituelles". Des cas de test sont donc construits pour appeler chaque opération d'un service avec des valeurs inhabituelles. En exécutant ceux-ci soit le type réponse doit être du même type que celui de la spécification soit il doit être une faute SOAP composée de la cause "SOAPFaultException".

Génération des cas de test

Le principe de génération de cas de test est proche pour les deux travaux. Nous avons construit des patrons de test abstraits qui sont complétés avec la description d'un service Web (spécification qui donne l'ensemble des opérations avec les types de paramètres et de réponses). Nous obtenons un ensemble de cas de test concrets, composés de valeurs.

Un cas de test est défini par :

Définition 2.2.8 *Un cas de test T est un arbre composé de noeuds $n_0, ..., n_m$ où n_0 est le noeud racine et où chaque noeud terminal est étiqueté par un verdict local dans $\{pass, inconclusive, fail\}$. Chaque branche est étiquetée soit par $op\_call(v)$ ou par $op\_return(r)$ tel que :*

- *v, est un tuple de valeurs avec $type(v) = type(from(op))$,*
- *soit $r = (c, soap\_fault)$ est une faute SOAP composée de la cause c soit r est un tuple de valeurs tel que $type(r) = type(to(op))$. $*$ modélise n'importe quelle valeur.*

La génération des cas de test est décrite de façon synthétique dans la Figure 2.17. La description WSDL est parsée afin de lister les opérations et les types de paramètres et réponses. Les cas de test sont construits à partir des patrons de test et d'un ensemble de valeurs prédéfinies, noté V. Nous notons $V(t)$ l'ensemble de valeurs pour le type t. Par exemple, les Figures 2.19, 2.18 and 2.20 illustrent les ensembles $V(Int)$, $V(String)$, $V(tabular)$.

Quelques patrons de test sont illustrés en Figures 2.21, 2.23 et 2.22. Le premier patron (Figure 2.21) correspond au test de l'existence des opérations, le deuxième (Figure 2.23) au test de session et le dernier au test de robustesse (Figure 2.22).

Pour simplifier la lecture, les cas "fail" sont représentés par des branches pointillées. Pour chaque cas de test obtenu, des opérations sont appelées avec des paramètres qui ont le type donné dans la description WSDL. Suivant les réponses obtenues, un noeud étiqueté par un verdict dans {pass, fail, fail/available} est atteint. Le verdict fail/available est obtenu dans le cas où une opération existe mais n'est pas robuste.

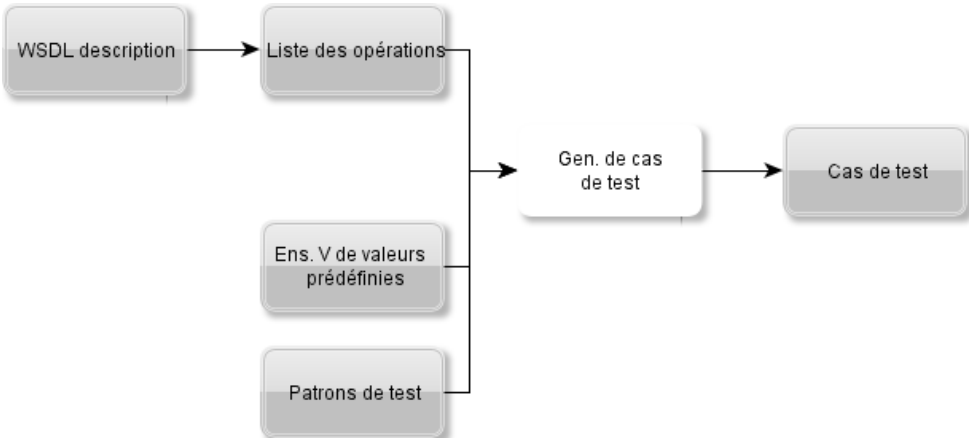


FIGURE 2.17 – Génération des cas de test

```
<type id="Int">
        <val value=null />
        <val value="0" />
        <val value="-1" />
        <val value="1" />
        <val value="MIN" />
        <val value="MAX" />
        <val value=RANDOM" /> <!-- a random
        Int-->
</type>
```

FIGURE 2.18 – V(Int)

```
<type id="String">
        <val value=null />
        <val value="" />
        <val value=" " />
        <val value="$" />
        <val value="*" />
        <val value="&" />
      <val value="hello" />
        <val value=RANDOM" /> <!-- a random
        String-->
        <val value=RANDOM(8096)" />
</type>
```

FIGURE 2.19 – V(String)

```
<type id="tabular">
        <val value=null /><!-- an empty
        tabular-->
        <val value= null null /><!--tabular
        composed of two empty elts-->
        <val value= simple-type />
</type>
```

FIGURE 2.20 – V(tabular)

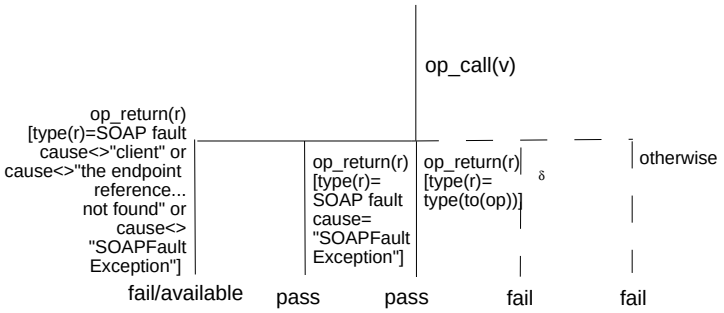


FIGURE 2.21 – Patron de test pour le test de l'existence des opérations

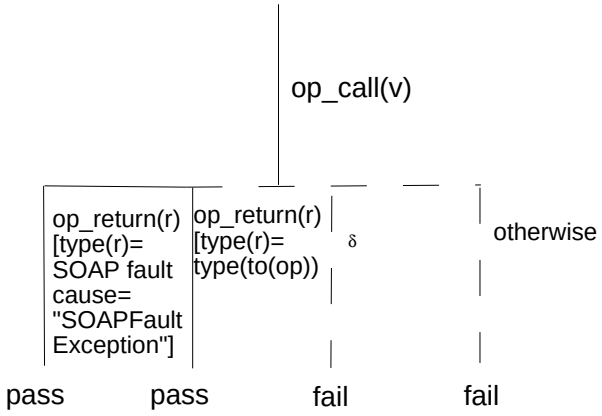


FIGURE 2.22 – Patron de test pour le test de robustesse

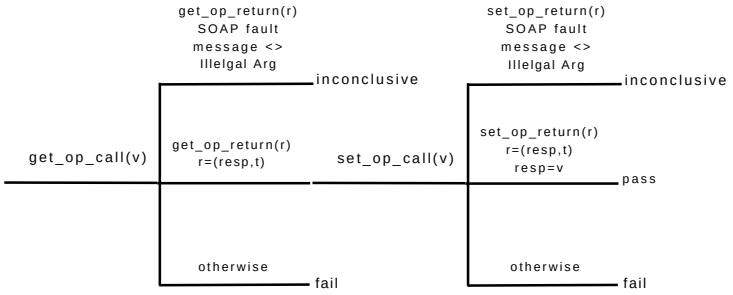


FIGURE 2.23 – Patron de test pour le test de gestion de session

Pour conclure sur un verdict de test final, le testeur exécute chaque cas de test en le parcourant (tout comme un test unitaire) : il invoque les opérations avec des valeurs de paramètres et attend l'observation d'une réponse ou de la quiescence tout en suivant la branche correspondante dans le cas de test. Lorsqu'une branche est complètement parcourue, un verdict local de cas de test est obtenu : pour un cas de test t, nous notons ce verdict local $vl(t) \in \{\text{pass} , \text{fail} , \text{inconclusive}\}$. Le verdict final du test est donné par :

Définition 2.2.9 *Soit WS un service Web et TC un suite de test. Le verdict du test, noté $Verdict(WS)/TC$, est*

- *pass, si pour tout $t \in TC, vl(t) = pass$. Le verdict pass signifie que toutes les opérations de $OP(WS)$ existent et sont robustes,*
- *inconclusive, si il existe $t \in TC$ tel que $vl(t) = inconclusive$, et s'il n'existe pas $t' \in TC$ tel que $vl(t') = fail$. Ce verdict implique que le service n'est pas robuste (à cause d'au moins une opération) mais que toutes les opérations existent,*
- *fail, s'il existe $t \in TC$ tel que $vl(t) = fail$.*

Expérimentation

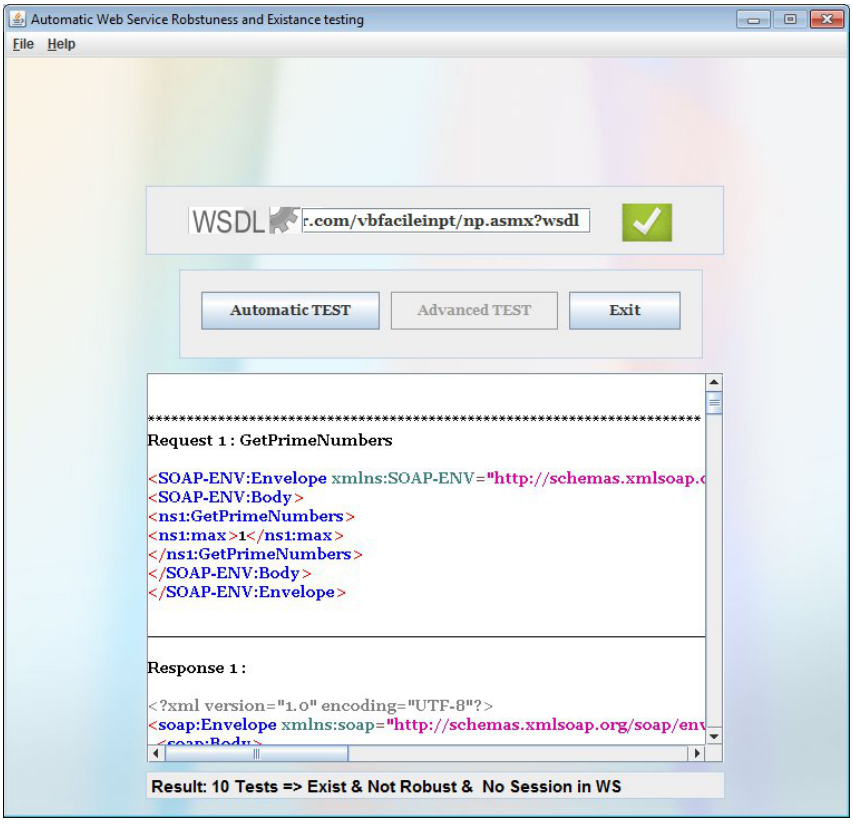


FIGURE 2.24 – Outil WS-AT

Nous avons implanté ces méthodes dans un outil appelé *WS-AT* dans le cadre du projet ANR WebMov (`http://webmov.lri.fr/`), traitant de la modélisation et du test de services Web et compositions. Cet outil est d'ailleurs disponible à cette adresse `http:`

//sebastien.salva.free.fr/WS-AT/WS-AT.html. Son écran d'accueil est illustré en Figure 2.24. Cet outil nous a permis d'expérimenter ces méthodes de test automatique sur plusieurs services Web stateless proposés par le fournisseur *Xmethods* (xmethods.org. La plupart des services ont montré des failles de robustesse (résultats illustrés en Figure 2.25). Nous avons observé que la cause principale des erreurs collectées concerne la gestion des exceptions qui est partielle ou incomplète. Cette expérimentation a confirmé les avantages suivants :

- *Efficacité :* l'outil est simple et permet de tester la majorité des services Web déployés sur Internet et accessibles. Les tests effectués sont simples également car ils ne dépendent pas d'une spécification. Cependant, de nombreuses erreurs peuvent être détectées comme l'accessibilité d'une opération, la gestion des exceptions (manque de blocs "try...catch" dans le code, fautes SOAP non gérées), la gestion de session à travers des getters/setters, la robustesse des opérations ou des problèmes d'observabilité (une opération ne répond pas). L'outil est aussi facilement évolutif car l'ensemble des valeurs de test utilisées peut être modifié facilement.
- *Coût du test :* les méthodes étant simples, le coût de test dépend uniquement du nombre de valeurs prédéfinies utilisées. Afin de réduire le nombre de cas de test construits sur le nombre de paramètres donnés par opération, nous avons implanté une génération de tuples de valeurs par la technique *Pairwise testing* [CGM03] qui consiste à générer des tuples de valeurs en considérant des combinaisons discrètes de paires de paramètres au lieu d'effectuer un produit cartésien sur tous les paramètres. Par conséquent, le test d'un service Web prend au plus quelques minutes.

Cette expérimentation a aussi montré les inconvénients suivants :

- l'ensemble V de valeurs prédéfinies a été augmenté afin de détecter plus d'erreurs. Il serait intéressant de construire dynamiquement un ensemble de paramètres le plus approprié selon le service Web testé,
- nous avons uniquement considéré les messages SOAP. Cependant, il pourrait aussi être intéressant de considérer le type de messages comme par exemple les requêtes effectuées à une base de de données,
- cette méthode couvre l'ensemble des opérations d'un service Web stateless. Dans le cas de services possédant un état (stateful), ces méthodes ne couvrent qu'une partie de la spécification. C'est pourquoi, nous avons étudié par la suite la robustesse de tels services.

2.2.3 Robustesse des services Web Stateful

Ce travail, décrit dans [SR10c, SR], entre dans la continuité des deux approches précédentes traitant de services Web stateless. Nous considérons dans ce cas que les services possèdent un état interne évoluant au fil des requêtes effectuées par des clients. En d'autres termes, nous supposons que les services Web sont stateful.

Un service Web WS est ainsi un composant réunissant un ensemble d'opérations $OP(WS)$ (Définition 2.2.1). Son état interne est formellement défini par un STS déterministe (Symbolic Transition Systems [FTW05] et section 1.3.1). La combinaison de la Définition 2.2.1 et du modèle STS permet d'exprimer la sémantique de communication d'un service Web stateful. Ainsi, une transition $(l, l', ?op_i(from(op_i)), \varphi, \varrho)$, représente une invocation d'opération, tandis que la transition $(l, l', !op\_return(to(op_i)), \varphi, \varrho)$ représente

Web Service	operations	parameters	tests	faults
primeNumbersGen	1	1	10	7
youtubeDownloader	1	1	10	9
sendSMS	2	2,4	22	0
mapIPtoCountry	2	1,1	20	4
numberToWords	1	1	10	6
localTimeByZipCode	1	1	10	0
textToBraille	2	2,2	22	20
tConvertions	2	1,2	22	11
strikeIron	1	3	12	0
svideoWs	1	3	12	12
yellowPagesLookup	1	5	12	0
codeLookup(BLZ)	1	1	9	9
textCasing	2	1,2	19	0
dateFunctions	2	3,3	20	13
koVidya	2	3,3	20	0
postML	4	1,6,5,2	40	40
ServiceObjects	5	2,3,1,4,1	50	0

FIGURE 2.25 – Résultats du test de services Web stateless

une réponse.

Concernant les aléas qu'il est possible d'exploiter pour tester la robustesse de services Web, nous avons étudié dans [SR09a] les aléas "remplacer un type de paramètre", "ajouter un type de paramètre", "supprimer un type de paramètre", "intervertir des paramètres" et "appel d'opérations avec des données inhabituelles" puis avons montré que seul le dernier est exploitable. Pour les services Web stateful, nous avons également étudié deux nouveaux aléas :

- **Remplacement du nom d'une opération** : celui-ci a pour but de modifier aléatoirement le nom d'une opération pour donner un nouveau nom qui n'existe pas dans la spécification du service Web. Cet aléa, ne respectant pas la description WSDL initiale, est rejeté par les processeurs SOAP et n'est donc pas utilisable,
- **Ajout/Remplacement d'invocation d'opération dans la spécification** : Soit l un état symbolique d'une spécification S de service Web WS, et soit les transitions sortantes $(l, l_1, ?op_1(p_{11}, ..., p_{1n}), \varphi_1, \varrho_1), ..., (l, l_k, ?op_k(p_{k1}, ..., p_{kn}), \varphi_k, \varrho_k)$ modélisant des invocations d'opérations. Si il existe une opération $op : from(op) \rightarrow to(op) \in OP(WS)$ tel que $op \notin \{op_1, ..., op_k\}$, cet aléa a pour but de remplacer/ajouter l'invocation d'une opération $op_i \in \{op_1, ..., op_k\}$ par op. Comme cet aléa ne contredit pas la description WSDL, il n'est pas bloqué par les processeurs SOAP.

Génération des cas de test

La méthode de test a pour objectif de vérifier si un service Web stateful fonctionne correctement malgré l'invocation d'opérations construites à partir des aléas "Appel d'opérations avec des données inhabituelles" et "Ajout/Remplacement d'invocation d'opération dans la spécification". Les cas de test générés sont sous forme de STS :

Définition 2.2.10 (cas de test) *Un cas de test est un STS déterministe à comportement fini* $T =< L, l_0, Var, var_0, I, S, \rightarrow >$ *où chaque état symbolique terminal est étiqueté*

par un verdict dans $\{pass, fail\}$.

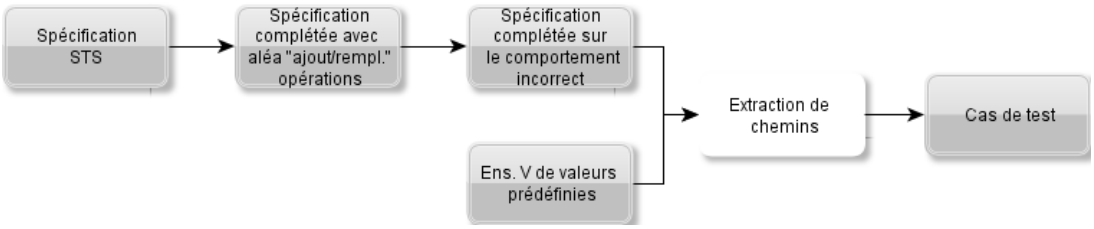


FIGURE 2.26 – Génération de cas de test de robustesse pour les services Web stateful

La génération des cas de test est décrite en Figure 2.26. Premièrement, les états symboliques de la spécification sont complétés sur l'ensemble des actions d'entrées pour injecter l'aléa "Ajout/Remplacement d'invocation d'opération dans la spécification". Celle-ci est aussi complétée pour exprimer le comportement incorrect et la quiescence des états symboliques. Ainsi, les cas de test décriront les comportements corrects et incorrects. Des chemins de cette spécification augmentée sont ensuite extraits afin de faire appel à chaque invocation de la spécification depuis son état symbolique initial. Ces derniers sont complétés avec des valeurs de variables inhabituelles avec l'utilisation de l'aléa "Appel d'opérations avec des données inhabituelles". Pendant l'extraction de chemins, une analyse d'accessibilité est effectuée sur les transitions et sur les variables pour donner une valeur aux variables indéfinies et pour vérifier si le chemin valué obtenu (séquence de transitions symboliques composées d'affectations de variables) peut être exécuté sur l'implantation sous test. Les solveurs de contraintes utilisés sont *Minisat* et *Hampi* [ES03, KGG+09]. L'ensemble des chemins valués obtenus représente l'ensemble des cas de test.

Les cas de test sont ensuite expérimentés par un testeur sur une implantation I supposée modélisée par un LTS. Pour un cas de test T le testeur exécute chaque action avant d'arriver à un état symbolique final. Nous définissons la relation *passe* avec I ***passe*** T ssi $T||\Delta(I)$ ne mène pas à un état $Fail$, avec $T||\Delta(I)$ la composition parallèle exprimant l'exécution de T sur I. La définition de la composition parallèle est donnée en section 1.3.5.

Le verdict final de la robustesse du service Web WS est donné par :

Définition 2.2.11 *Soit WS un service Web, S sa spécification et TC une suite de test. Le verdict de la robustesse de WS, noté $Verdict(WS)$ est :*
- *pass, si pour tout $t \in TC, I$ passe t,*
- *fail, si il existe $t \in TC$ tel que $\neg I$ passe t.*

Expérimentation

Nous avons implanté une partie de cette méthode dans un outil qui prend une spécification complétée et qui en extrait les cas de test comme décrit précédemment. Nous avons ensuite appliqué cet outil sur deux versions du service Web E-commerce d'Amazon (AWSECommerceService [Ama10]). Les cas de test obtenus ont ensuite été traduits par un second outil au format XML pour utiliser l'outil SOAPUI [Evi11] dont la fonction

	09/03	09/10
Number of tests	100	100
Using unusual values	75	75
Replacing/Adding operation names	25	25
Fails	34	34
Unspecified messages	28	28
Unspecified SOAP faults	6	6

FIGURE 2.27 – Résultats obtenus avec le service AWSECommerceService d'Amazon

Operation	ItemSearch	ItemLookup	CartCreate	CartAdd
Number of tests	35	25	20	20
Using unusual values	29	19	14	13
Replacing/Adding operation names	6	6	6	7
Fails	29	1	2	2
Unspecified messages	28	0	0	0
Unspecified SOAP faults	1	1	2	2

FIGURE 2.28 – Résultats détaillés

principale est l'exécution de cas de test unitaire sur services Web. Les résultats sont illustrés en Figures 2.27 et 2.28). Approximativement 30 pour-cents des tests ont relevé de problèmes de robustesse.

Avec l'aléa "Appel d'opérations avec des données inhabituelles" et bien que toutes les requêtes construites satisfont la description WSDL, nous avons obtenu des fautes SOAP composées de la cause *Client*, signifiant normalement que nos requêtes sont incorrectes. Cependant, nous pensons que ces fautes reçues peuvent être la conséquence de l'utilisation de règles de sécurité (pare-feux, WS-Policy) sur les serveurs d'Amazon. Nous avons aussi obtenu des réponses non spécifiées correspondant à des messages d'erreur incohérents. Par exemple, nous avons reçu la réponse "Your request should have at least 1 of the following parameters : AWSAccessKeyId, SubscriptionId lorsque nous avons appelé l'opération *CartAdd* avec une quantité égale à "-1", ou lorsque le type de recherche était fait sur le type "Book" au lieu de "book", alors que les deux paramètres AWSAccessKeyId et SubscriptionId étaient justes. Nous avons également collecté des erreurs similaires avec l'aléa "Ajout/Remplacement d'invocation d'opération dans la spécification" seulement lorsque l'opération *CreateCard* était remplacée par une autre. Selon ces résultats ces deux versions de service ne sont donc pas robustes.

2.2.4 Test de sécurité des services Web

Nous avons proposé de tester la sécurité des services Web stateful dans [SR10c]. L'objectif de cette approche est d'expérimenter une instance de service afin d'y détecter des vulnérabilités.

Tout comme précédemment, un service Web stateful WS (possédant un état) est un composant réunissant un ensemble d'opérations $OP(WS)$ (Définition 2.2.1 dont l'état est modélisé par un STS déterministe $\mathcal{S}$ (Symbolic Transition Systems [FTW05] et section 1.3.1). $\mathcal{S}$ est augmenté pour aussi exprimer l'environnement SOAP (ajout des fautes SOAP

comme en section 2.2.2). Ce qui donne le STS $\mathcal{S}\uparrow$.

Nous avons premièrement choisi de représenter un ensemble de patrons de test basés sur des vulnérabilités que nous avons classés selon ces critères :

- **Disponibilité :** qui représente la capacité d'un système à répondre correctement quelquesoit la requête émise. Une conséquence directe est qu'un système est disponible ssi ce dernier ne "plante" pas quelquesoit la requête,
- **Authentification :** qui a pour but d'établir ou de garantir l'identité d'un client et de vérifier qu'un client sans autorisation correcte, n'a pas les permissions correspondantes. Le processus de connection est souvent la première étape pour l'authentification d'un client,
- **Autorisation :** représente la police d'accès qui spécifie les droits d'accès aux ressources, habituellement à des utilisateurs authentifiés,
- **Intégrité :** qui contrôle que les données ne peuvent pas être modifiées de façon non autorisée.

A partir des recommandations données dans [OWA03] sur les vulnérabilités rencontrées avec les services Web, ces patrons de test sont premièrement écrits grâce au langage Nomad (Non atomic actions and deadlines [CCBS05]) car il est adapté pour exprimer des propriétés comme l'autorisation, l'interdiction, et peut prendre en compte des propriétés temporelles comme les délais. A partir de cette première formalisation de haut niveau, nous transformons chaque patron de test en STSs pour pouvoir les appliquer à la spécification. Ce qui donne un ensemble d'objectifs de test.

Pour une spécification $\mathcal{S} =< L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}}>$ nous définissons un objectif de test par un STS acyclique et déterministe $TP =< L_{TP}, l0_{TP}, V_{TP}, V0_{TP}, I_{TP}, \Lambda_{TP}, \rightarrow_{TP}>$ tel que

- $V_{TP} \cap V_{\mathcal{S}} = \emptyset$,
- $I_{TP} \subseteq I_{\mathcal{S}}$,
- $\Lambda_{TP} \subseteq \Lambda_{\mathcal{S}}$,
- $\rightarrow_{TP}$ est composé de transitions décrivant des propriétés de la spécification. Ainsi, pour toute transition $l_j \xrightarrow{a(p),\varphi_j,\varrho_j}_{TP} l'_j$, il existe une transition $l_i \xrightarrow{a(p),\varphi_i,\varrho_i}_{\mathcal{S}} l'_i$ et un ensemble de valeurs $(x_1, ..., x_n) \in D^n_{V \cup I}$ tel que $\varphi_j \wedge \varphi_i(x_1, ..., x_n) \models$ vrai.

Pour un patron de test Nomad P_i, nous notons OP_{P_i} l'ensemble des opérations données dans P_i. Nous dérivons un premier patron de test au format STS, noté tp_{P_i} et un domaine de variables $V_{tp_{P_i}}$. Ce STS est composé d'invocations génériques qui sont ensuite remplacées par des invocations réelles en les remplaçant par les opérations dans OP_{P_i}.

Prenons l'exemple d'un patron de test Nomad extrait de [SR10c]. Le patron $T1$ exprime qu'une opération est disponible ssi elle ne "plante" pas et répond avec un message SOAP quelquesoit la requête. En d'autres termes, $T1$ signifie que si une requête est "done" alors il est obligatoire ($\mathcal{O}$) d'obtenir une réponse $OutputResponseWS$.

$$T1 \longleftrightarrow \forall opReq \in \Lambda^I_{\mathcal{S}\uparrow},\ \mathcal{O}(start(output\ OutputResponseWS) | done(input\ (opReq(p), TestDom := \{Spec(opReq); RV; Inj\})))$$

Ce patron est ensuite traduit en un objectif générique illustré en Figure 2.29 qui produit plusieurs objectifs de test composés d'opérations concrètes.

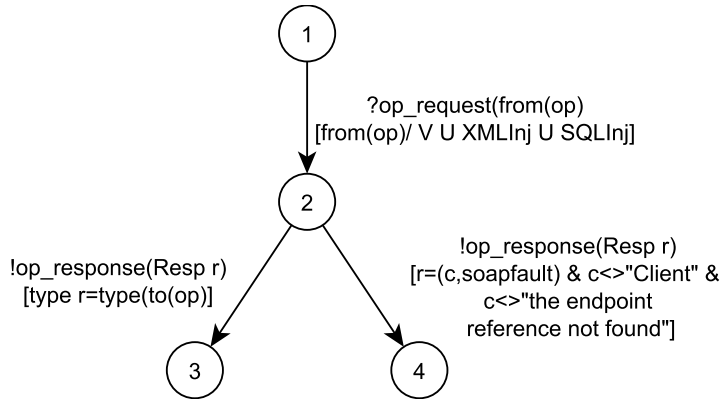


FIGURE 2.29 – Objectif de test extrait du patron de test $T1$

Relation d'implantation et génération des cas de test

L'implantation sous test est représentée par un LTS $Impl$ ($\Delta(Impl)$ son LTS suspension). L'objectif de la méthode de test est de vérifier si les traces suspensions de $Impl$ peuvent être trouvées dans les traces suspensions de la combinaison de la spécification et des objectifs de test. Ceci peut s'écrire formellement par la relation suivante :

$$Impl\ secure_{TP}\ \mathcal{S} \Leftrightarrow \forall tp \in TP, STraces(Impl) \cap NC\_STraces(\mathcal{S}^{\uparrow} \times tp) = \emptyset$$

avec TP l'ensemble des objectifs de test extraits à partir des patrons de test Nomad, $\mathcal{S}^{\uparrow} \times tp$ le produit synchronisé d'un objectif de test avec la spécification augmentée et $NC_STraces(\mathcal{S}^{\uparrow} \times tp) = STraces(\mathcal{S}^{\uparrow} \times tp).\Lambda^0 \cup !\delta \setminus STraces(\mathcal{S}^{\uparrow} \times tp)$ les traces suspension non conforme de $\mathcal{S}^{\uparrow} \times tp$. Des cas de test sont ensuite construits pour vérifier cette relation. Pour un STS $\mathcal{S}$, un cas de test est un STS déterministe $\mathcal{TC} =< L_{\mathcal{TC}}, l0_{\mathcal{TC}}, V_{\mathcal{TC}}, V0_{\mathcal{TC}}, I_{\mathcal{TC}}, \Lambda_{\mathcal{TC}}, \rightarrow_{\mathcal{TC}} >$ où les états symboliques finaux sont étiquetés par { pass, fail }.

La génération des cas de test est résumée en Figure 2.30. A partir des objectifs de test, qui représentent l'intention de test, et de la spécification augmentée avec l'environnement SOAP, nous effectuons un produit synchronisé entre chaque objectif et la spécification. Ceci donne un produit $\mathcal{P}$ dont les chemins sont ceux de la spécification combinés avec les propriétés de l'objectif de test. $\mathcal{P}$ est ensuite complété pour exprimer le comportement incorrect. Finalement, les cas de test sont sélectionnés en effectuant une analyse d'accessibilité sur $\mathcal{P}$ et en affectant des valeurs aux variables indéfinies avec les solveurs *Minisat* et *Hampi* [ES03, KGG+09] (voir section 2.2.3). Ceci donne un ensemble de cas de test TC.

Les cas de test sont ensuite expérimentés par un testeur sur une implantation $Impl$. Pour un cas de test T le testeur exécute chaque action avant d'arriver à un état symbolique final. Nous définissons la relation *passe* avec I ***passe*** T ssi $T||\Delta(I)$ ne mène pas à un état $Fail$, avec $T||\Delta(I)$ la composition parallèle exprimant l'exécution de T sur I. La définition de la composition parallèle est donnée en section 3.1.2. Le verdict final de la robustesse du service Web WS est donné par :

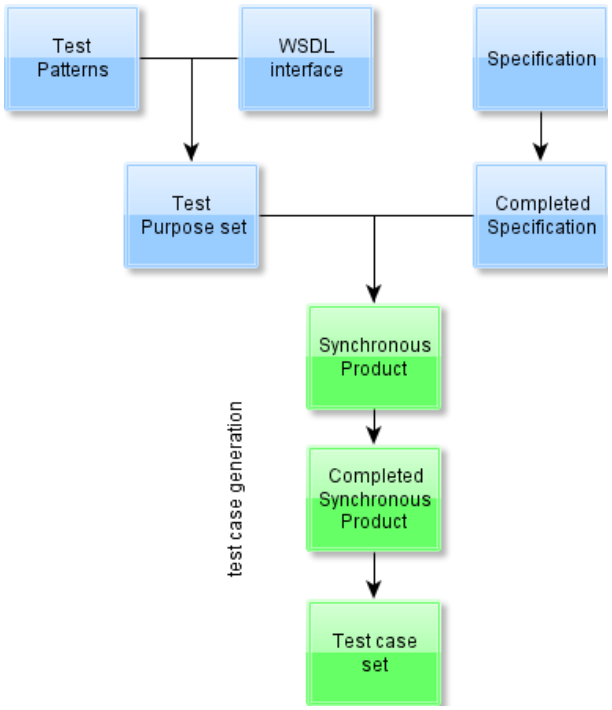


FIGURE 2.30 – Génération des cas de test

Définition 2.2.12 *Soit WS un service Web, S sa spécification et TC une suite de test. Le verdict sur la sécurité de WS, noté $Verdict(WS)$ est :*

- *pass, si pour tout $t \in TC, I$ passe t,*
- *fail, si il existe $t \in TC$ tel que $\neg I$ passe t.*

Expérimentation

Nous avons implanté une partie de cette méthode dans un outil qui prend une spécification complétée et effectue le produit synchronisé entre cette dernière et des objectifs de test. Ceux-ci sont pour l'instant donnés au format STS uniquement. L'architecture de l'outil est illustrée en Figure 2.31. Les cas de test obtenus ont ensuite été traduits par un second outil au format XML pour utiliser l'outil SOAPUI [Evi11] dont la fonction principale est l'exécution de cas de test unitaire sur services Web.

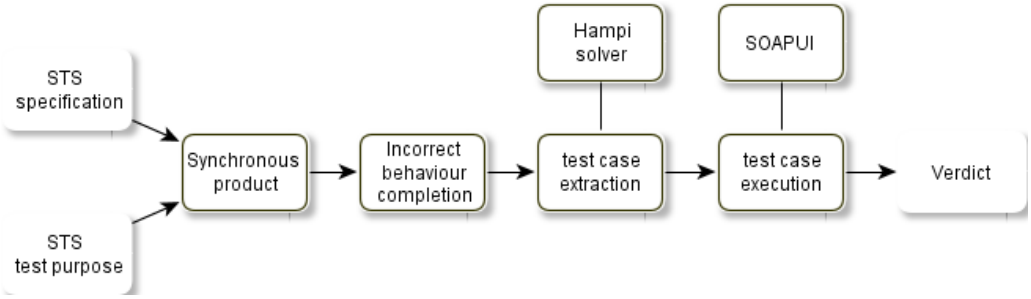


FIGURE 2.31 – Architecture de l'outil de test de sécurité

Web Service (WSDL)	test number	Availability	Authentication	Authorization
http://research.caspis.net/webservices/flightdetail.asmx?wsdl	56	0	0	1
http://student.labs.ii.edu.mk/ii9263/slaveProject/Service1.asmx?WSDL	60	0	1	1
http ://biomoby.org/services/wsdl/www.iris.irri.org/getGermplasmByPhenotype	26	0	6	0
http ://www.infored.com.sv/SRCNET/SRCWebServiceExterno/WebServSRC/servSRCWebService.asmx ?WSDL	20	10	0	0
http ://81.91.129.80/DialupWS/dialupVoiceService.asmx ?WSDL	22	0	0	2
http ://81.91.129.80/DialupWS/SecurityService.asmx ?WSDL	18	0	0	3
https ://intrumservice.intrum.is/vidskiptavefurservice.asmx ?WSDL	66	6	1	0
http://www.handicap.fr/server_hanproducts.php?wsdl	78	2	0	4
https://gforge.inria.fr/soap/index.php?wsdl	100	1	0	0
http://193.49.35.64/ModbusXmlDa?WSDL	30	2	0	0
http ://nesapp01.nesfrance.com/ws/cdiscount ?wsdl	30	2	0	2
http ://developer.ebay.com/webservices/latest/ShoppingService.wsdl	30	10	0	0

FIGURE 2.32 – Résultats d'Expérimentation

Nous avons expérimenté le service AWSECommerceService d'amazon (2009/10 version) [Ama10]. Mise à part les problèmes de disponibilités liés à la robustesse que nous avons décrits en section précédente, nous n'avons pas détecté d'autres vulnérabilités. Ce n'est pas surprenant, car ce service est constamment mis à jour depuis plusieurs années.

Nous avons également testé environ une centaine d'autres services Web. Des vulnérabilités ont été détectées pour 11 % d'entre eux avec des objectifs de test non exhaustifs par rapport aux recommandations du OWASP [OWA03]. Notamment, 6 % possèdent des problèmes d'autorisation et retournent des données confidentielles comme des login, mots de passe et données personnelles. La Figure 2.32 résume ces résultats.

Différents types de vulnérabilités ont été détectés. Par exemple, Le service Web *getGermplasmByPhenotype* n'est plus disponible lorsqu'il est appelé avec des paramètres composés de caractères spéciaux. Ici, nous suspectons l'existence de la vulnérabilité "validation de données incorrecte". Des failles d'autorisation ont été détectées avec *server_hanproducts.php* car ce service retourne des fautes SOAP composées de données confidentielles telles que des tables de base de données. Le même problème est relevé avec le service *cdiscount*. Ainsi, ceux-ci fournissent des données confidentielles. Avec le service *slaveProject/Service1.asmx*, l'attaque "brute force" peut être appliquée pour déduire des login et mots de passe. Cette expérimentation a aussi montré que d'autres facteurs peuvent conduire à un verdict "fail". Par exemple, le test du service *Ebay shopping* a montré que la quiescence peut être observée pour un tiers des opérations. En fait, au lieu de recevoir une faute SOAP, nous avons obtenu l'erreur "HTTP 503", signifiant que le service n'est pas disponible. Ceci peut être du à une erreur mais aussi à une surcharge du trafic. De tels facteurs illustrent les limites de notre méthode de test qui ne peut pour l'instant les prendre en compte. Une solution serait de combiner cette méthode avec une approche de test passif par monitoring qui pourrait détecter des vulnérabilités sur une longue période

de temps.

2.2.5 Une plate-forme pour la modélisation et le test de compositions de services

De 2007 à 2010, notre équipe du laboratoire LIMOS a activement participé au projet ANR WebMov (`http://webmov.lri.fr/`), dont les objectifs étaient de contribuer à la conception, à la composition et la validation de services Web à travers une vue abstraite de haut niveau et d'une vision SOA. Ce projet a donné lieu a plusieurs méthodes de modélisation et de test actives ou passives. Le travail présenté dans [CCM+10] présente la chaîne d'outils développés et l'application de cette chaîne à des exemples réels.

La chaîne d'outils est illustrée en Figure 2.33. Le premier outil *Modelio* permet entre autre de modéliser des compositions de services à l'aide des langages BPMN, WSDL et de schémas XML. Le squelette de la composition spécifiée peut ensuite être généré. A partir des descriptions WSDL, les services Web participants à la composition peuvent être testés (test usager, robustesse) via notre outil *WS-AT* (section 2.2.2). La spécification de haut niveau peut aussi être traduite en modèles de plus bas niveau d'abstraction (IF, STS, EIOTA) pour appliquer les outils *Testgen-IF, TGSE, SetGen* permettant le test de conformité de l'implantation à partir de sa spécification. Enfin, l'implantation peut aussi être testée passivement grâce aux outils *WSOTF* et *WS-Inject*, le premier permettant de détecter des erreurs à partir d'une spécification formelle, le second à partir d'invariants.

Nous avons explicité les résultats obtenus sur un exemple connu de composition BPEL, le TSR (Travel Reservation Service) qui est un exemple réel et complexe fourni avec l'IDE Netbeans 6.5.1 [Net09]. Les résultats ont non seulement montré la fiabilité et la robustesse de cet exemple mais aussi la complémentarité des outils à travers le test actif, passif, le test de chaque partenaire, le test de la composition en entier, etc. Afin de montrer la pertinence à utiliser cette chaîne d'outils, des erreurs ont aussi été ajoutées dans diverses spécifications. Toutes ces erreurs ont été retrouvées.

2.2.6 Étude de la testabilité des compositions ABPEL

Nous avons étudié les compositions de services Web et leur testabilité à travers les deux critères d'observabilité et de contrôlabilité, ceci dans le but de comprendre et d'appréhender les difficultés à modéliser et tester des compositions.

Nous avons choisi de modéliser les compositions grâce au langage ABPEL (Abstract Business Process Execution Language) qui permet de créer des processus centralisés d'orchestration de services. Ce langage offre une multitude de mécanismes pour la composition (partenaires, réception multiple, corrélation, gestion des exceptions, etc.) et est utilisé dans plusieurs méthodes de test [BPZ09].

BPEL (ou BPEL4WS [Con07]) définit des processus basés sur des interactions de services, appelés partenaires. BPEL fournit les éléments de déclaration (messages, rôles des partenaires, liens, etc.) et l'orchestration qui inclut différents éléments comme les états, les variables et notamment les activités. Le langage propose des activités basiques (invocation, réception,etc.) et des activités structurées (scope, flow, etc.) pour fixer la structure du processus. Il permet l'utilisation de "fault handler" pour gérer les messages d'erreur. Le code est écrit en XML qui peut être illustré sous forme de boites. Nous

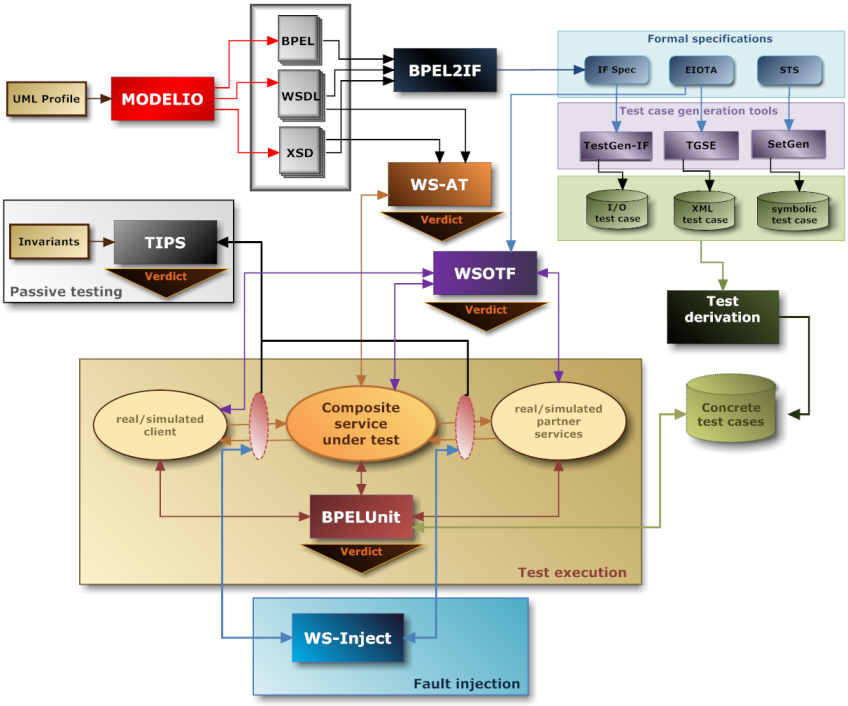


FIGURE 2.33 – Chaîne d'outils développés dans le cadre du projet ANR Webmov

donnons l'exemple du "loan approval" en Figure 2.34, qui est dérivé du code BPEL qui se trouve dans la spécification du langage BPEL [Con07]. Dans cet exemple, un client effectue une demande de prêt qui est accordée suivant le montant et les réponses des partenaires *loan assessor*, *loan approval1* et *loan approval2*.

Étude de la testabilité de spécifications BPEL

Dans [SR10b], nous avons effectué une étude préliminaire de composition ABPEL pour déterminer une solution d'estimation du niveau de testabilité.

Le langage BPEL étant grammaticalement très large (nombre important de déclarations, d'activités et de combinaisons), nous avons choisi de transformer une spécification ABPEL en STS qui est un modèle de plus bas niveau et connu. De façon intuitive, nous avons listé l'ensemble des activités et pour chacune nous avons donné une règle permettant de la transformer en transitions STS. Toutes les règles obtenues sont en deux parties : La première prend en compte l'imbrication possible entre les activités : les "fault handler" d'une activité sont propagées dans ses sous-activités et ainsi de suite. La deuxième transforme une activité en transitions. La figure 2.35 illustre deux exemples de règles pour les activités *while* et *invoke*. Pour l'activité *while*, la première règle est utilisée si le fault handler (FH) n'est pas nul. Dans ce cas, il est propagé dans les sous activités. S'il est nul, l'activité est traduite en transitions.

Nous avons également donné un algorithme transformant un processus ABPEL en STS à partir des règles de traduction d'activités ABPEL en transitions. Cet algorithme développe successivement chaque activité structurée et construit un graphe de sous activités. Ainsi, l'algorithme commence avec l'activité racine "process" qu'il traduit en un

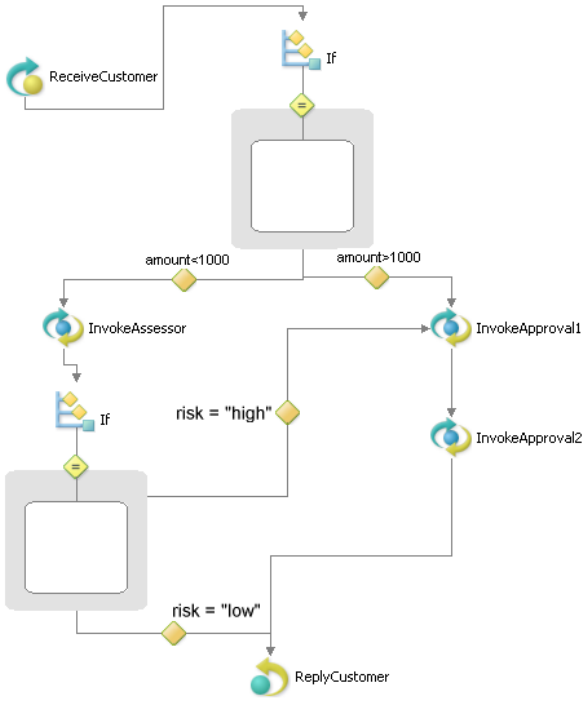


FIGURE 2.34 – Le processus BPEL Loan approval

$while(cond, act, FH, CH, TH)$	$\frac{FH=\emptyset}{e_i \xrightarrow{[cond]} e_k \xrightarrow{\tau,[act],\emptyset} e_i \wedge e_i \xrightarrow{\tau,[\neg cond],\emptyset} e_{i+1}}$ $\frac{1 \leq k \leq n \wedge FH \neq \emptyset}{while(cond,(act'_1,...,act'_n),\emptyset,\emptyset,\emptyset) \text{ avec } act'_k=((act'_{k1},...,act'_{km}),\ \overline{FH_{act_k} \cup FH, CH_{act_k} \cup CH, TH_{act_k} \cup TH})}$
$invoke(op, req, resp, partner, corr, FH, CH, TH)$	$\frac{resp \neq null, fh_k \rightarrow e_m \in FH}{e_i \xrightarrow{!(op,req,partner),\emptyset,\varrho=(c_i:=corr)} e_l \wedge \ e_l \xrightarrow{?(op,resp,partner),[c_i==corr],\emptyset} e_{i+1} \wedge e_l \xrightarrow{fh_k} e_m}$ $\frac{resp=null}{e_i \xrightarrow{!(op,req,partner)} e_{i+1}}$

FIGURE 2.35 – Exemple de règles de transformation BPELtoSTS

ensemble de transitions étiquetées par les sous activités. Ensuite, chaque sous-activité est soit traduite avec les règles de traduction, soit développée en graphe de sous-activités jusqu'à ce qu'il n'y ait plus d'activité à traduire.

Une fois l'algorithme appliqué sur une spécification ABPEL, nous obtenons un STS qui est plus facile à analyser car la testabilité de ce modèle est connue. Nous nous sommes focalisés sur l'observabilité et la contrôlabilité.

Au préalable, pour un STS S, associé à son LTS sémantique $||S||$, nous avons défini la fonction $out : Q \rightarrow \Sigma_O^*$ qui retourne l'ensemble des premières actions valuées de sortie

obtenues à partir d'un état (l, η) de $||S||$.

$out(l_1, \eta) = \{(!o_1, \eta'_1), ..., (!o_n, \eta'_n) \mid \forall 1 \leq i \leq n, \exists p = (l_1, l_2, e_1, \varphi_1, \varrho_1)...(l_i, l_{i+1}, !o_i, \varphi_i, \varrho_i)$ tel que $(e_1, ..., e_{i-1}) \in S_I^{i-1}$ et $\vee(\varphi_1(\eta), ..., \varphi_i(\varrho_{i-1}, \eta'_1))$ true$\}$.

Concernant l'observabilité, nous avons déduit deux types de dégradation pour le modèle STS :

- Dégradation d'observabilité 1 : si il existe un stimulus $(?e, \eta)$ tel que $(k, l, ?e, \varphi, \varrho) \in \rightarrow$ et $out(l, \eta) = \emptyset$,
- Dégradation d'observabilité 2 : si il existe deux stimuli $(?e_i, \eta_i) \neq (?e_j, \eta_j)$ avec $(l_i, l'_i, ?e_i, \varphi_i, \varrho_i) \in \rightarrow$, $\varphi_i(\eta_i)$ true et $(l_j, l'_j, ?e_j, \varphi_j, \varrho_j) \in \rightarrow$, $\varphi_j(\eta_j)$ true tel que $out(l'_i, \eta_i) = out(l'_j, \eta_j)$.

Concernant la contrôlabilité, nous avons étudié le déterminisme qui est l'une des propriétés dégradant le contrôle d'une implantation. Un STS $S =< L, l_0, V, V_0, I, \Lambda, \rightarrow >$ est indéterministe s'il existe un état symbolique $l \in L$ tel que :

1. il existe $(l, l_i, e, \varphi_i, \varrho_i)$, $(l, l_j, e, \varphi_j, \varrho_j)$ étiquetées par le même symbole et η avec $\vee(\varphi_i(\eta), \varphi_j(\eta))$ true,
2. il existe $(l, l_i, !o_i, \varphi_i, \varrho_i)$, $(l, l_j, !o_j, \varphi_j, \varrho_j)$ avec $!o_i \neq !o_j$ et η avec $\vee(\varphi_i(\eta), \varphi_j(\eta))$ true,
3. il existe $(l, l_i, ?e, \varphi_i, \varrho_i)$, $(l, l_j, !o, \varphi_j, \varrho_j)$ et η avec $\vee(\varphi_i(\eta), \varphi_j(\eta))$ true.

La mesure de l'observabilité est ensuite déduite en recherchant le nombre de dégradations dans le STS. Plus ce nombre est grand, moins le processus ABPEL est testable. Le cas idéal est bien sûr de ne trouver aucune dégradation.

Amélioration de la testabilité de spécifications ABPEL

Dans [SR10a, Sal11d], nous avons proposé divers algorithmes pour améliorer de façon automatique l'observabilité de spécifications ABPEL. Comme précédemment, nous avons traduit une spécification ABPEL en STS. Nous sommes parti des définitions de dégradation de l'observabilité. Grâce aux règles de traduction d'activités ABPEL en transitions STS, nous avons ensuite recherché des cas ou le code BPEL donne ces dégradations. Nous avons déduit 6 cas sous forme de propositions. A titre d'exemple, nous donnons ici la première :

Proposition 2.2.1 *Une spécification ABPEL dont le processus n'est pas terminé par une activité* Reply *(ou* one-way invoke*) n'est pas observable.*

Par la suite, pour chaque proposition, nous avons proposé un algorithme qui détecte une dégradation dans une spécification ABPEL et qui transforme de façon automatique cette dernière en supprimant la dégradation. La nouvelle spécification ABPEL est donc plus testable.

Pour la proposition précédente, l'algorithme 1 vérifie que chaque branche du processus ABPEL se termine par une activité *Reply* (invoke-only). Si une telle activité est manquante, la spécification est complétée avec une activité *Reply* modélisant l'envoi de réponse à un client qui a appelé le processus ABPEL. La réponse envoyée est composée du message "final message from $branch_i$" qui est supposé être unique et non encore utilisé dans la spécification. Par conséquent, ce message ne dégrade pas l'observabilité.

Nous avons appliqué ces algorithmes à la spécification illustrée en Figure 2.34. Initialement, celle-ci est composée de 6 dégradations qui ont été réduites à 1 en appliquant les

Algorithme 1: Ajout d'une activité "Reply"

input : Spécification ABPEL *bpel*

1 Construire $sts =< L, l_0, Var, var_0, I, S, \rightarrow >$ à partir de *bpel*;

2 **if** $\exists l_i \xrightarrow{e,\varphi,\varrho} l_f$ *avec* $e \in S_I \cup \{\tau\}$ *et* l_f *un état symbolique* **then**

3 Ajouter $Reply(resp, partner, op)$ dans *bpel* avec resp="last message from $branch_i$", partner=client, op= opération invokée par le client;

différents algorithmes donnés dans [Sal11d]. En perspective, ces travaux pourraient être étendus à d'autres critères de testabilité, comme la contrôlabilité mais aussi comme la couverture de la spécification par les tests ou le coût d'exécution des cas de test.

2.3 Test de robustesse des systèmes réactifs

A la différence des applications précédentes (services Web, compositions, protocoles), les systèmes réactifs sont des programmes qui réagissent à l'environnement à la vitesse de l'environnement, de façon synchrone et souvent déterministe.

Nous avons proposé, dans [RS08, RS09], quelques approches de test de robustesse basé modèle, de systèmes réactifs modélisés par des IOLTS (LTS dont l'alphabet sépare les actions d'entrée des actions de sortie). Nos contributions, dans ce domaine, concernent la prise en compte de spécifications dégradées décrivant le comportement minimal d'un système dans un environnement critique et l'utilisation de la relation ioco afin de montrer qu'un système est robuste. Par rapport aux travaux [FSKC07, FMP05, ROL03], ces approches permettent d'intégrer des entrées inattendues dans le modèle. Dans [RS09], le test permet aussi de vérifier que si le système est robuste alors il est ioco-conforme.

Ces approches sont résumées par la suite.

2.3.1 Test de robustesse par deux approches complémentaires

Dans [RS08], deux approches sont proposées : la première suppose que la propriété de robustesse implique la conformité, alors que la deuxième approche suppose le contraire. Ces deux approches offrent donc une très bonne complémentarité.

Test de robustesse en supposant que Système Robuste ⇒ Système Conforme

Cette première approche, très simple considère des *aléas externes*, c'est-à-dire des évènements externes non attendus comme une action d'entrée non spécifiée.

Les actions spécifiées du système et les aléas sont classés dans un tableau. Ce dernier sera utilisé pour construire les cas de test. Celui-ci décrit la spécification combinée à tous les aléas externes (ligne= état de départ, colonne= état d'arrivée, les valeurs du tableau correspondent à des actions). La méthode remplit ce tableau et génère les cas de test à partir de ce dernier.

Les étapes sont résumées par :

- le tableau de départ est construit à partir de la spécification. Un exemple de spécification est illustré en Figure 2.36.

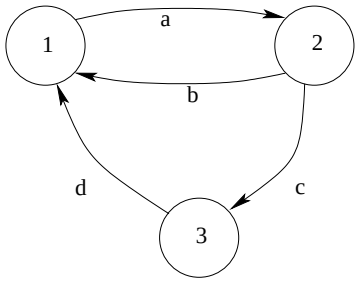


FIGURE 2.36 – Exemple d'une spécification IOLTS

	$?a$	$?c$	$\mathcal{A}_i - \{?a, ?c\}$	$!b$	$!d$	$!\delta$	$\mathcal{A}_o - \{!b, !d\}$
1	2	1	1	f	f	1	f
2	**1**	3	2	1	f	f	f
3	3	3	3	f	1	f	f

FIGURE 2.37 – Tableau modélisant une spécification complète

- la quiescence des états est ajoutée dans le tableau. Pour un état ayant des transitions étiquetées par des actions de sortie, si la quiescence est observée, un verdict local fail est obtenu,
- le tableau est rempli pour que la spécification soit complétée sur l'ensemble des actions d'entrée,
- le tableau est rempli pour que la spécification soit complétée avec son comportement incorrect sur l'ensemble des sorties (une action de sortie non spécifiée mène à un verdict local fail),
- des modifications manuelles peuvent être ajoutées. Le tableau obtenu est illustré en Figure 2.37,
- des cas de test sont générés à partir du tableau en utilisant un algorithme dérivé de la méthode ioco [Tre96c].

Test de robustesse en supposant que Système Robuste $\nRightarrow$ Système Conforme

Cette approche suppose qu'en cas de réception d'aléas, le système se place en phase dégradée où seules les actions vitales sont gardées. Ceci implique qu'un système est modélisé par une spécification nominale S et une spécification dégradée S'.

Premièrement, la relation $\leq_{rob}$ sur les IOLTS est définie pour exprimer une inclusion de comportement entre deux IOLTS $S \leq_{rob} S'$:

Définition 2.3.1 *Soit deux IOLTS S et S'. Nous notons que S' est inclus dans S dans le sens de la robustesse ssi : $S \leq_{rob} S' =_{def} \forall \sigma' \in Trace(S'), \exists \sigma \in Traces(S)$ tel que σ est composée d'actions de σ' dans le même ordre.*

En supposant que l'implantation I peut être modélisée par un IOLTS, l'objectif de la méthode est de vérifier si $I \leq_{rob} S'$ et $S \leq_{rob} S'$. Pour ce faire les cas de test sont construits par les étapes suivantes :

- une suite de test est dérivée à partir de la spécification nominale,

- des aléas sont injectés dans les cas de test. Seules les actions n'appartenant pas à la spécification dégradée S' peuvent être modifiées,
- les cas de test sont expérimentés sur l'implantation,
- les traces obtenues sont analysées pour vérifier $I \leq_{rob} S'$.

2.3.2 Méthode de test de robustesse orientée ioco

De même que précédemment, le systèmes est modélisé par un IOLTS, et le test de robustesse est appliqué en considérant des aléas externes : (1) Occurrence d'une action d'entrée connue mais non attendue à ce moment, (2) Occurrence d'une action d'entrée non connue (pas dans l'alphabet de la spécification), (3) Occurrence d'une action de sortie non attendue.

La différence de cette méthode réside dans l'utilisation de la relation de conformité ioco [Tre96c] (section 1.3.5) pour vérifier si l'implantation est robuste. Ceci offre l'avantage de montrer que les cas de test obtenus sont *exacts (sound)* et *complets (complete)* (voir section 1.2).

Cette méthode peut se synthétiser par les étapes suivantes :

- La spécification est modélisée par un IOLTS S. Le IOLTS suspension $\Delta(S)$ correspondant est construit pour prendre en compte la quiescence,
- $\Delta(S)$ est transformé pour être déterministe (s'il ne l'est pas), ce qui donne le IOLTS $Det(\Delta(S))$,
- $Det(\Delta(S))$ est complété sur l'ensemble des entrées et des sorties pour faire entrer la notion de robustesse dans le modèle. Chaque état de $Det(\Delta(S))$ est complété avec toutes les entrées connues et non connues de l'implantation par des transitions qui bouclent vers le même état. Le comportement incorrect (sorties connues ou pas) est ajouté en complétant des transitions étiquetées par des sorties menant à un état fail. Ceci donne un IOLTS $compl(Det(\Delta(S)))$,
- Les cas de test sont générés par un algorithme donné dans [RS09], à partir de $compl(Det(\Delta(S)))$. Cet algorithme a pour but de construire des cas de test "forçant" le testeur à exécuter des transitions modélisant les aléas (1) Appliquer une action d'entrée connue mais non attendue, (2) Appliquer une action d'entrée non connue. Plus précisément, l'utilisateur fournit un critère de robustesse composé de deux entiers $rob1$ et $rob2$. rob1 (resp. rob2) donne le nombre d'actions d'entrée successives que l'on doit rencontrer dans le cas de test pour l'aléa 1) (resp. (2)) lorsque $compl(Det(\Delta(S)))$ le permet.

Les cas de test obtenus permettent ensuite de vérifier si I ioco $compl(Det(\Delta(S)))$ avec I une implantation supposée décrite par un IOLTS. Nous montrons également que les cas de test sont exacts et complets par rapport à la relation ioco.

2.4 Autre

Cette section termine le chapitre de synthèse de mes travaux. Trois publications, traitant du test de systèmes distribués, d'application Ajax et de la parallélisation d'appels de services Web avec OpenMP, y sont succinctement décrits.

2.4.1 Test de systèmes distribués et mobiles

Notre équipe a participé au projet RNRT Platonis (`http://www-lor.int-evry.fr/platonis/resume.html`), qui avait pour objectif le développement et la mise en place d'une plate-forme de validation et d'expérimentation multi-protocoles et multi-services. Cette plate-forme vise le test d'interfonctionnement des services intégrant la mobilité et l'Internet (WAP), ainsi que les technologies qui permettent la mise en place de ces services (UMTS, GPRS-GSM).

Ce projet a notamment donné lieu à la publication [LS04] qui décrit une architecture de test distribuée permettant le test d'une application WAP. L'originalité de cette architecture est de synchroniser les testeurs entre eux, en complétant les cas de test.

De façon synthétique, nous avons une spécification décrite par un IOSM (Input Output State Machine) qui correspond à LTS (section 1.3.1). Des cas de test sont générés à partir d'une méthode de test orientée objectif de test [CCKS95, FJJV96]. La difficulté réside dans l'exécution de ces cas de test car l'environnement utilisé par les applications WAP est distribué sur des serveurs Web, sur des téléphones mobiles et sur l'environnement GSM. La plate-forme de test est donc nécessairement distribuée. Il faut donc éclater les cas de test, placer chaque morceaux sur le bon testeur et synchroniser ces testeurs.

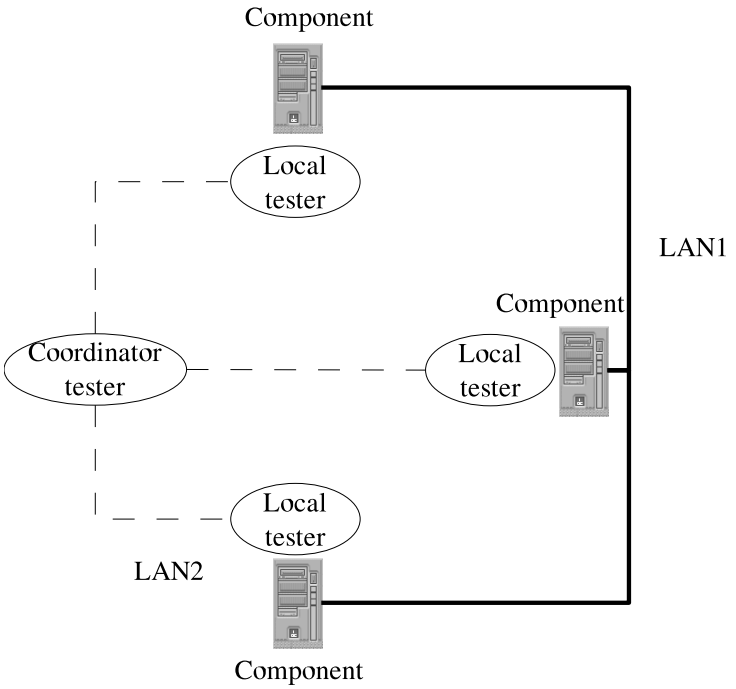


FIGURE 2.38 – Architecture de test distribuée

L'architecture proposée dans [LS04] est décrite en Figure 2.38. Celle-ci repose sur un testeur central interconnecté avec les autres testeurs par le réseau existant (illustré en lignes continues), a savoir un réseau GSM. Nous avons aussi donné un algorithme qui extrait à partir d'un cas de test ω, des cas de test partiels ω_t, pour chaque testeur local t. Cet algorithme ajoute dans chaque cas de test partiel des données de synchronisation exprimées par un ou plusieurs verrous m,odélisés par $(-sync_j^p, +sync_j^p)$:

– $+sync_j^p$ place en pause le testeur courant jusqu'à ce qu'un message de synchronisation soit reçu depuis le testeur j.
– $-sync_j^p$ modélise l'envoi d'une synchronisation au testeur j avec une valeur p.

Sans donner l'algorithme qui se trouve dans [LS04], celui-ci éclate un cas de test initial en plusieurs parties en analysant chaque action et en déduisant quel testeur doit l'exécuter. Lorsque deux actions successives ne sont pas destinées au même testeur, une synchronisation est ajoutée : $+sync_j^p$ est ajouté dans un cas de test correspondant pour verrouiller le testeur qui doit exécuter la seconde action. $-sync_j^p$ est ajouté dans le cas de test du testeur qui doit exécuter la première action.

Nous avons expérimenté cette technique sur une application WAP dont l'architecture est donnée en Figure 2.39. Celle-ci est composée de trois testeurs dont un sur un téléphone GSM. Des exemples de cas de test sont illustrés dans [LS04].

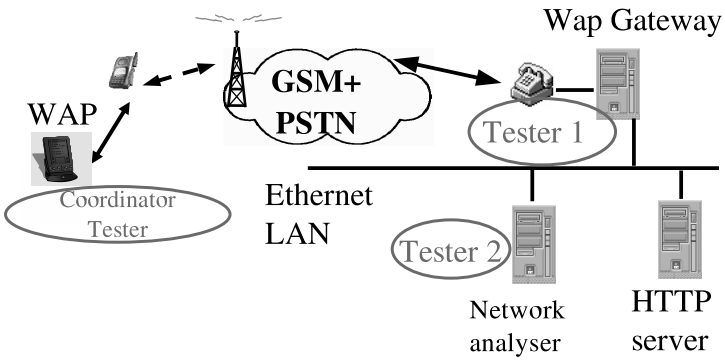


FIGURE 2.39 – Architecture de Test de l'expérimentation

2.4.2 Test d'applications Ajax

La technologie Ajax est employée aujourd'hui à large échelle dans le monde du développement Web. Celle-ci permet d'obtenir une interface Web dynamique en simulant un mode connecté alors que le protocole HTTP ne l'est pas.

Quelques méthodes et outils ont été proposés pour tester des applications Ajax [Caf07, Lar06, MTR08b, MTR08a]. Notre méthode, décrite dans [SL09], est la première à base de spécification formelle. Elle ne présente pas un solution à base de beta-test mais un test de conformité basé sur la relation ioco. La spécification est décrite de façon originale : nous avons un STS pour décrire le comportement classique de l'application Ajax et un ensemble de STS pour décrire des exécutions concrètes. La contribution notable de ce travail concerne l'outil de test qui permet de tester de façon automatique une application Ajax, sans avoir besoin de manipuler un navigateur.

Ajax (asynchronous JavaScript and XML) correspond à un groupement de technologies Web permettant d'appeler un serveur de façon asynchrone à partir d'un évènement utilisateur. Google MAP (Figure 2.40) représente un exemple classique d'application Ajax dans laquelle des lieux sont marqués sans rechargement de page à partir d'une recherche.

Les applications Ajax sont souvent composées d'une application client javascript et d'applications serveur. La première invoque une application serveur, après qu'un évène-

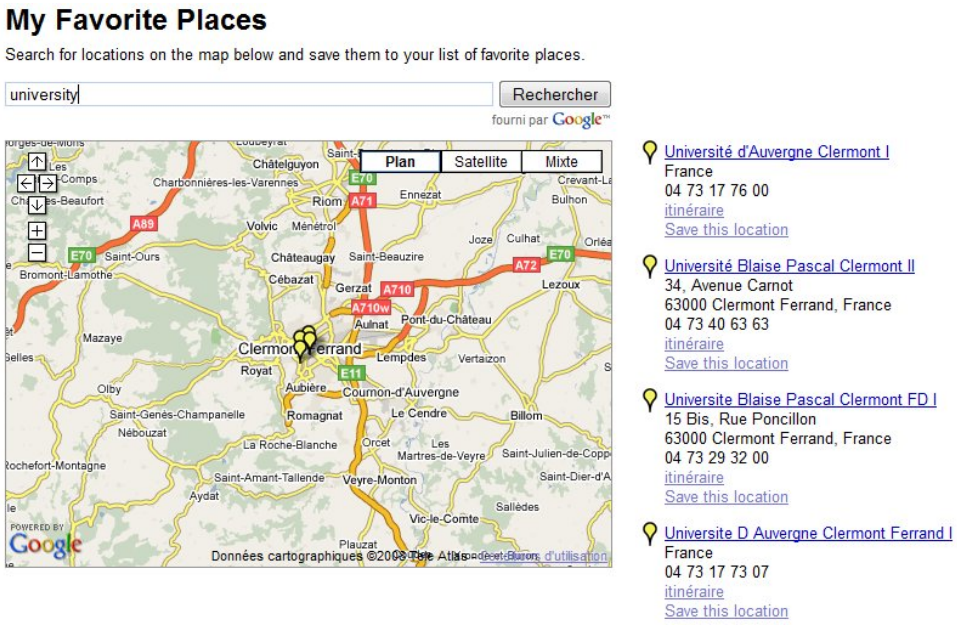


FIGURE 2.40 – Google Map Search

ment soit déclenché dans un navigateur Web, puis modifie le document lu par le navigateur après avoir reçu une réponse.

Concernant la modélisation d'applications Ajax, nous avons choisi de les modéliser par des ensembles de diagrammes de séquence UML décrivant l'interaction utilisateur, les messages émis entre le client et le serveur et les modifications obtenues dans le document. Un exemple de diagramme est donné en Figure 2.41.

Ces diagrammes sont ensuite traduits en un ensemble de STSs. Ainsi, une application Ajax est définie par :

Définition 2.4.1 *Une spécification Ajax* $AS =< S_{DEF}, S >$ *est un tuple tel que :*

- S_{DEF} *est un STS modélisant l'application Ajax,*
- S *est un ensemble de STSs tel que* $s \in S$ *représente une exécution (un diagramme de séquence UML).* s *est composé d'affectations de variables pour toutes les variables externes de* s*,*
- *pour chaque STS* $s \in S \cup \{S_{DEF}\}$*, et pour chaque transition* $t \in \rightarrow_s$ *étiquetée par une action d'entrée is,* $is =$ "$?event < i, o : DOM$" $>$ *où* i *et* o *sont des objets DOM (ensembles de données structurées décrits en XML) , avec* $i \neq \varnothing$ *et* $i.function \neq \varnothing$*.* o *peut être égal à* $\varnothing$*.*

En d'autres termes, S_{DEF} correspond à la spécification "classique" alors que l'ensemble S peut être vu comme un ensemble d'objectifs de test.

Génération des cas de test

La génération des cas de test est effectuée par les étapes suivantes :

1. la liste de transitions $(t_1, ..., t_k) \in (\rightarrow_{S_{DEF}})^k$, étiquetées par " ?event<i,o :DOM>" est premièrement extraite,

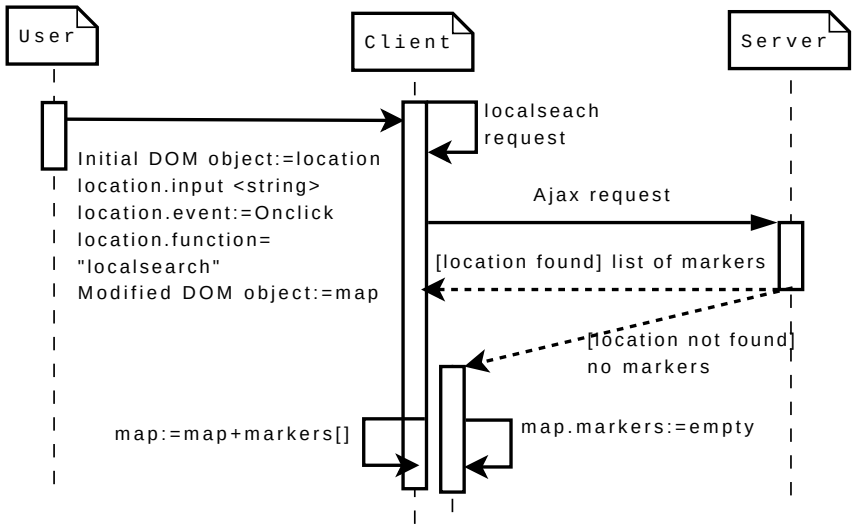


FIGURE 2.41 – Diagramme de séquence UML

2. pour chaque transition $t \in (t_1, ..., t_k)$, un ensemble de tuples de valeurs est construit à partir d'un ensemble de valeurs prédéfinies V (ensemble identique à celui décrit en section 2.2.2). $Values(t) = \{(r_1, ...r_n) \mid t$ étiqueté par $?event < i, o : DOM >$ $, (r_1, ...r_n) \in R(p_1) \times ... \times R(p_n)$ avec $\rho(i) = (p_1, ...p_n), \}$. Nous obtenons des ensembles $Values(t_1), ..., Values(t_k)$,
3. nous générons l'ensemble de STSs TS depuis S_{DEF} en injectant les tuples de valeurs précédents dans S_{DEF}. Pour chaque $(v_1, ..., v_k) \in Values(t_1) \times ... \times Values(t_k)$, nous dérivons un STS $s \in TS$ depuis S_{DEF} tel que pour chaque $t_i \in (t_1, ..., t_k)$ étiqueté par " ?event<i,o :DOM>", $\rho(i) = v_i$,
4. $TS = TS \cup S$,
5. les cas de test finaux sont construits à partir de la méthode de test orientée *ioco* pour les STSs décrite dans [FTW05].

Exécution des cas de test

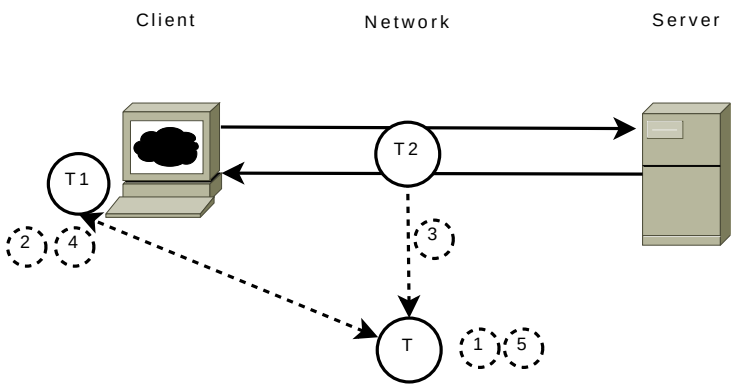


FIGURE 2.42 – Plate-forme de Test

Les cas de test sont ensuite exécutés automatiquement (sans l'aide d'une personne interagissant avec le navigateur) à l'aide de la plate-forme illustrée en Figure 2.42 qui a été implantée dans un prototype d'outil. Nous avons l'application Ajax AUT, déployée sur le coté client et le coté serveur. Le testeur est quant à lui distribué et composé des testeurs T, $T1$ et $T2$:

- **Testeur T1 :** celui-ci simule un utilisateur interagissant avec un document dans le navigateur Web. Ce testeur, écrit en Javascript est ajouté automatiquement dans le document Web et exécuté dès sa lecture. $T1$ prends une transition du cas de test étiquetée par " ?event" (1). $T1$ vérifie si l'objet DOM *init* et ses propriétés existent (fonction, évènements, valeurs, etc.). Si *init* n'existe pas, le verdict fail est retourné à T (4). Autrement, $T1$ remplit l'objet DOM *init* avec des valeurs. $T1$ appelle $init.function$ (6,7). Une fois cette fonction terminée, un objet DOM *last* a éventuellement été modifié. Dans ce cas, ($last \neq null$) ce dernier est envoyé à T (9,10),
- **Testeur T2 :** ce testeur correspond à un sniffeur XML installé entre le client et le serveur. Dès qu'un message XML est complètement lu celui-ci est envoyé au testeur T,
- **Testeur T :** ce testeur orchestre l'exécution des cas de test. T prends un liste $TS = \{s_1, ...s_m\}$ de STSs. Pour chaque STS s_k, T construit premièrement les cas de test et les parcours. Lorsqu'il atteint une transition étiquetée par " ?event", il l'envoie à $T1$ pour déclencher la fonction Ajax. Autrement, T reçoit les messages depuis $T1$ ou $T2$ ou observe la quiescence d'un état. Suivant le message, il vérifie qu'il peut franchir une transition du cas de test. Ceci est effectué jusqu'à ce que T atteigne un état final. Cet état est étiqueté par un verdict local "pass" ou "fail".

Le verdict final est donné suivant les verdicts locaux obtenus. La définition du verdict est la même que celle donnée dans la Définition 2.2.11.

Nous avons implanté cet outil qui est disponible à `http://sebastien.salva.free.fr/`. Cet outil est composé de deux parties : la première est une interface permettant d'entrer des diagrammes de séquences UML qui sont ensuite traduits en STSs. Cette partie est illustrée en Figure 2.43. La deuxième partie concerne la plate-forme de test décrite ci-dessus.

2.4.3 Parallélisation des appels de services Web avec OpenMP

Nous ne présenterons que sommairement ce travail [SBD07] qui est en marge du thème de ce mémoire. Celui-ci propose plusieurs solutions de parallélisation d'invocations de Services Web par un ensemble de clients, dont une nouvelle solution offrant des gains de performance élevés.

Nous supposons qu'il y a un flot de requêtes clientes de services sur un même serveur. Nous avons analysé plusieurs façons possibles pour paralléliser l'exécution de ces requêtes en commençant par une approche naïve qui consiste à créer un pool de tâches pour effectuer des invocations complètes en parallèle. Nous avons montré que cette granularité n'offre pas les meilleures performances surtout si le temps d'appel au service est long.

Nous avons alors proposé de découper une invocation de service en plusieurs tâches : sérialisation des données, appel d'un service, désérialisation et stockage en base de donnée. Puis, nous avons proposé une solution utilisant le paradigme du pipeline. Cette solution

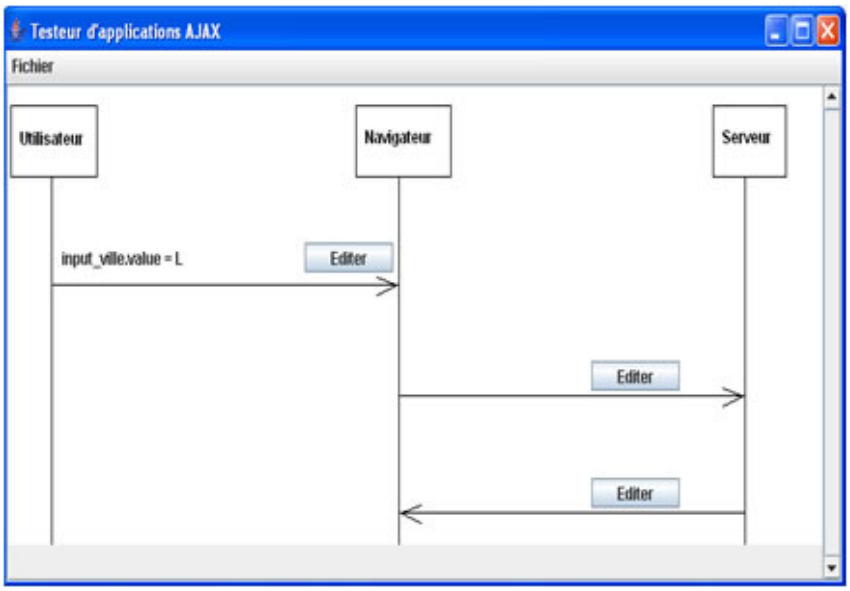


FIGURE 2.43 – Outil de modélisation

est illustrée en Figure 2.44. Le pipeline est composé de cinq étages (1 par tâche).

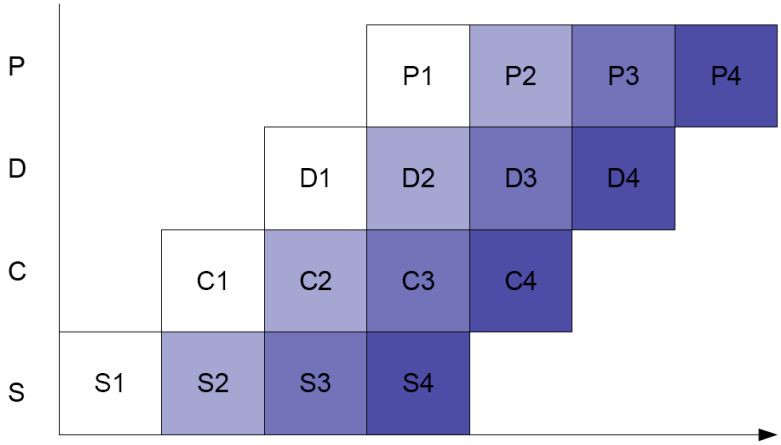


FIGURE 2.44 – Solution proposée à base du paradigme du pipeline

Cependant, les temps d'appel de service peuvent bloquer le pipeline et réduire les performances globales. Nous avons donc modifié la structure de l'étage dédié à l'appel de services. La Figure 2.45 illustre l'architecture de cet étage, qui est implanté par un pool de tâches. Une tâche est alors représentée par un objet {call identifier, URL, method, Message} qui modélise un appel de service. Un thread disponible exécute une tâche (1). Celui-ci utilise la servlet *SEND* pour appeler un service (2). La servlet *SEND* effectue une connection et envoie les données sérialisées données par l'étage précédent (3). La servlet *Response* reçoit la réponse de façon asynchrone (4). Chaque réponse est ensuite donnée à l'étage Deserialization (5).

Nous avons implanté cette méthode grâce à l'API JOMP, basée sur OPEN-MP [BK00].

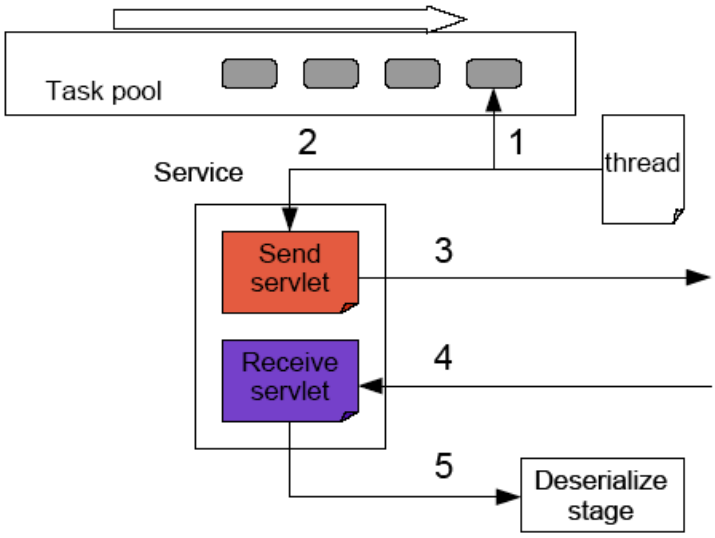


FIGURE 2.45 – Architecture de l'étage d'Appel

Nous avons effectué plusieurs expérimentations sur une partie du projet Copernic [Dir06] développé par le Ministère des Finances. Nos résultats sont donnés en Figure 2.46. Nous avons comparé l'approche naïve utilisant un pool de tâches avec notre solution pour des temps d'appels de services et un nombre de threads variés. Nous avons considéré 100 appels de services avec 4000 objets à sérialiser/désérialiser pour des temps d'appel durant 1s, 5s et 10s. Le temps de référence pour le calcul de speedup est l'invocation de service séquentielle avec un seul thread. Pour des temps d'appel faibles (1s) notre solution offre des performances 10 % supérieures à l'approche naïve du pool. Les performances augmentent de 60 % lorsque les temps d'appel sont de 10s.

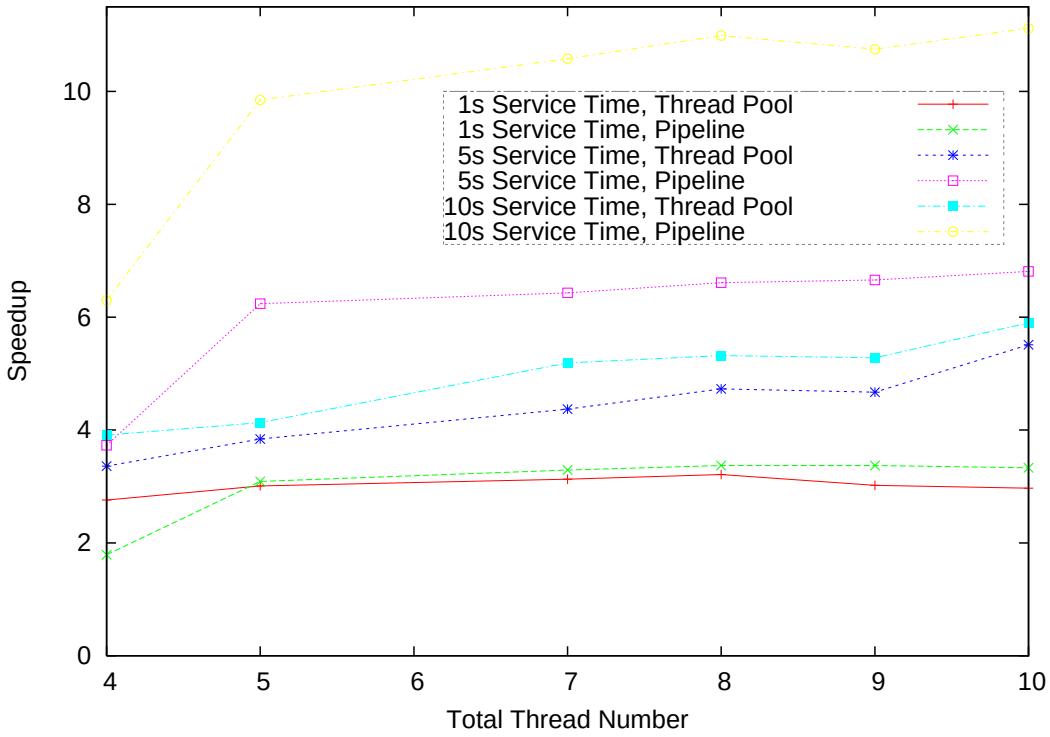


FIGURE 2.46 – Courbes de Speedup

Chapitre 4

Travaux en cours/Projets de recherche

Ce chapitre décrit quelques travaux en cours mais surtout des projets de recherche qui découlent des travaux précédents et de collaborations avec l'industrie.

Une grande partie de ce chapitre traite notamment des environnements dits partiellement ouverts que ce soit pour le test actif ou passif. Les environnements, dans lesquels sont déployés des applications ou des systèmes, sont très souvent considérés comme étant ouverts, c'est-à-dire que le système à tester est accessible et que des PO (points d'observation) ou PCO (points de contrôle et d'observation) peuvent être utilisés de façon assez simple pour expérimenter une implantation.

Cependant, on peut remarquer, dans les tendances informatiques actuelles, que l'accès aux environnements des systèmes ou logiciels peut être restreint. Ceci s'explique par des raisons de sécurité (l'accès à un système est limité par des droits d'accès, des parefeux, etc.) mais aussi par des choix technologiques. Par exemple, le paradigme du *Cloud computing*, qui est l'une des dernières tendances actuelles, induit un environnement virtualisé où les serveurs, les bases de données et même les applications sont déployés dans

un nuage abstrait sans démarque géographique précise. Les smartphones représentent un autre exemple car l'accès aux applications est difficile pour des raisons matérielles mais aussi pour sécuriser les données personnelles. Dans ces types d'environnement, installer des outils de test pour extraire les messages échangés entre applications, peut s'avérer difficile voire impossible. Cela revient donc à tester des systèmes dans des environnements que nous qualifions de partiellement ouverts. Ces environnements nous conduisent vers de nouvelles problématiques et questions générales, parmi lesquelles :

- comment peut-on interagir avec l'implantation sous test ? Il est évident que l'implantation doit pouvoir être expérimentée, mais les stimuli envoyés sont-ils modifiés par l'environnement, d'autres sont-ils créés ?
- Quelles observations peuvent être faites ? La majorité des méthodes de test, et notamment celles dites en boite grise ou noire, observent les réactions provenant de l'implantation et les emploient afin, par exemple, de valider une relation de test. Dans de tels environnements, des réactions peuvent ne pas être observées, ou bien observées de façon indirecte. Quelles sont les hypothèses à définir afin de garantir une faisabilité des tests ? Peut-on observer des réactions différemment ?
- Comment prendre en compte ces environnements dans les méthodes de test et dans les relations de test ? En effet, ces environnements jouent un rôle dans l'expérimentation de l'implantation. Les modéliser et les composer avec des spécifications peut-il suffire ? Comment ceux-ci modifient-ils les interactions et les observations ? Comment les prendre en compte dans les relations de test ?
- Comment mettre à jour les architectures de test qui ont été proposées dans la littérature ?

Ce chapitre introduit quelques réponses et propose plusieurs perspectives notamment liées à ces environnements. Premièrement, nous présentons deux approches préliminaires sur le test de conformité d'applications dans des environnements partiellement ouverts. La première approche concerne le test de compositions statiques et a pour objectif de retrouver les traces d'une composition à partir des traces des composants en décomposant des cas de test existants puis en recomposant les traces partielles obtenues. La deuxième approche cible le test de conformité effectué de façon passive. Habituellement, ce type de test est effectué par des outils dérivés de sniffeurs (TCP, XML, etc.). Cette approche définit formellement un proxy-testeur, qui correspond à un intermédiaire entre le trafic client et l'implantation.

Les deux parties suivantes décrivent deux projets de recherche, en phase de démarrage ou en cours d'élaboration, en collaboration avec les sociétés Orange Business et Openium. La première partie décrit les problématiques et perspectives liées à la modélisation et au test de compositions dynamiques de services dans des environnements partiellement ouverts. Dans ce type de composition, les acteurs ne sont pas connus à l'avance mais choisis dynamiquement à l'exécution. La dynamicité induit des problématiques fortes en terme de modélisation et de test vis-à-vis de l'abstraction qui doit être faite sur les composants potentiels et sur le comportement général de la composition. La seconde partie décrit les problématiques liées au test de sécurité d'applications mobiles que l'on trouve sur les smartphones actuellement. Les besoins de tester ce type d'application semble aujourd'hui assez évident aux vues des rapports alarmistes sur le nombre d'applications souffrant de failles de sécurité (plus de 20 % des applications sur l'Android Market par exemple [net]). Les verrous rencontrés sont, d'une part d'ordre scientifique car ces applications sont dé-

ployées sur des environnements distribués et très hétérogènes (smartphones, serveurs Web, Clouds), et d'autre part technologiques car les smartphones sont des systèmes particuliers avec des caractéristiques nouvelles comme la géo-localisation.

3.1 Test de conformité de systèmes orientés composants dans des environnements partiellement ouverts

La programmation de logiciels orientés composants est un paradigme connu et aujourd'hui omniprésent dans la plupart des sociétés de développement informatique. Ce dernier offre de nombreux avantages tels que la conception d'applications à partir de composants existants hétérogènes ou la réutilisation de code accompagnée d'une réduction de coût.

Ce paradigme a aussi donné naissance à plusieurs nouveaux concepts comme les services Web et le Cloud computing qui offre la possibilité de déployer des composants dans des environnements virtualisés et dynamiques. Ce paradigme a aussi été étudié depuis une dizaine d'années en relation avec les approches de test pour vérifier plusieurs aspects e.g., la sécurité, la robustesse, ou la conformité de différents systèmes et applications comme les applications orientées service, les systèmes distribués, ou les systèmes orientés objet.

L'évolution de la programmation orientée composant apporte toujours aujourd'hui de nouvelles problématiques en relation avec les activités de test et notamment au niveau du test de conformité, qui correspond au sujet de cette section. En effet, la majorité des approches de test nécessitent un environnement ouvert dans lequel les interactions entre les composants peuvent être observées/extraites pour produire un verdict de test. Ceci correspond typiquement à l'une des hypothèses posées dans la plupart des méthodes de test de conformité basées sur l'équivalence de traces [BFPT06, LZCH08a, iHBvdRT03, KABT10]. Cependant, cette hypothèse ne peut être maintenue lorsque sont pris en compte des environnements partiellement ouverts, environnements dont l'accès est restreint pour diverses raisons.

L'approche suivante s'attaque à cette problématique en supposant que les messages échangés entre les composants ne peuvent être extraits de l'environnement dans lequel est déployée la composition, mais que chaque composant (ou une copie exacte) peut être stimulé (appelé) directement.

De façon intuitive, à partir d'une suite de test générée par une méthode de test classique basée sur la relation ioco [FTW05], notre approche consiste à décomposer les cas de test afin d'expérimenter chaque composant de façon isolée. En exécutant les cas de test obtenus, nous retrouvons des traces partielles que nous recomposons de façon hiérarchique pour produire finalement l'ensemble des traces de la composition. Plus précisément, la décomposition des cas de test est effectuée selon la dépendance obtenue entre les méthodes des composants : ainsi, si nous avons une méthode qui invoque une autre méthode dans un cas de test initial, donc si nous avons une méthode dépendante d'une autre méthode, nous décomposons hiérarchiquement le cas de test en plusieurs parties pour appeler directement chaque méthode et pour retrouver les traces partielles correspondantes. En généralisant cette idée, nous obtenons une suite de test décomposée qui produit des traces incomplètes. Celles-ci sont ensuite ré-assemblées avec un algorithme spécifique. A partir de l'ensemble final de traces obtenu, nous pouvons vérifier si l'implantation de la composition est ioco

conforme à sa spécification.

Dans la suite, nous décrivons un état de l'art sur le test de compositions, nous définissons la modélisation de composition à partir de STSs. Puis, nous décrivons la décomposition des cas de test et la recomposition des traces. Finalement, nous discutons sur cette phase de décomposition/recomposition en relation avec la relation de test ioco et donnons plusieurs perspectives de travaux futurs.

3.1.1 Travaux existants et Motivations

Le test de composants et de compositions a donné lieu à plusieurs travaux dans différents domaines. Certains de ces travaux sont résumés dans ce qui suit.

Certains travaux considèrent des applications orientées objet. Les auteurs de [WPC01, ZB07] se sont focalisés sur des applications décrites à partir de spécifications UML. Comme plusieurs travaux traitant des applications orientées objet, les cas de test sont construits au moyen d'oracles (pré, post conditions sur l'espace des variables) représentant souvent des critères de couverture qui servent à restreindre, de façon drastique, les espaces de domaines des variables utilisées dans le code de ces applications. Gallagher et al. proposent également dans [GO09], la description technique d'un outil automatique de test aidant au test d'intégration d'applications orientées objet.

Plusieurs autres travaux se focalisent sur les aspects services et composition de services. Dans [GFTdlR06], une composition est modélisée par le langage BPEL (Business Process Execution Language). Celle-ci est ensuite traduite en PROMELA afin d'être vérifiée grâce à l'outil de model-checking SPIN. Dans [DYZ06], Les auteurs proposent une méthode qui traduit une spécification BPEL en réseaux de Petri. Ceci permet ensuite d'utiliser des outils connus comme *GreatSPN* afin d'étudier vérification, couverture de test et génération de test. Dans [BFPT06], les auteurs proposent d'augmenter les descriptions WSDL avec des diagrammes PSM (UML2.0 Protocol State Machine) qui représentent les interactions possibles entre un service et des clients. Les cas de test sont ensuite générés à partir de PSM. Une architecture de test, appelée l' Audition framework, est également définie dans [BP05]. L'approche décrite dans [LZCH08a], transforme des spécifications BPEL en un autre modèle appelé *IF*, qui permet la modélisation de contraintes temporelles. La génération des cas de test est basée sur une simulation d'exécution dans laquelle l'exploration de la spécification est guidée grâce à des objectifs de test. L' approche, décrite dans [EGGC09a], adapte l'algorithme de génération de cas de test décrit dans [GGRT06]. Ce dernier consiste à sélectionner des chemins finis d'un arbre d'exécution en simulant son parcours.

D'autres travaux se sont penchés sur les composants et les compositions en général. Kanso and al. décrivent, dans [KABT10], une théorie sur le test de conformité pour les composants, formalisés à partir d'un méta-modèle abstrait pour remplacer la plupart des formalismes orientés état. La sélection des cas de test s'effectue également via des objectifs de test. Dans [LLML10], Lei and al. proposent une méthode de test de robustesse de composants modélisés par des State Machines (automates symboliques). Dans [BK09], les systèmes composés sont décrits via des priorités et les composants sont exprimés sous forme de LTSs. De ce fait, les auteurs introduisent une relation de conformité exprimant aussi la priorité. Afin de vérifier cette relation, des cas de test sont générés à partir des LTSs. L'approche décrite dans [FRT10], a pour objectif de tester la robustesse de systèmes

composés temps réels. Pour chaque composant, des cas de test sont dérivés à partir de deux spécifications : une spécification nominale et une autre dite dégradée. Chaque composant est ensuite testé premièrement en isolation, puis le système complet est décrit comme robuste si les communications entre les composants satisfont les spécifications dégradées des composants. Grieskamp et al. proposent dans [GTC+05], une plate-forme de test de compositions symboliques. Celles-ci sont représentées sous forme d' actions machines (modèles symboliques) qui peuvent se synchroniser ensemble sur les actions pour produire une spécification composée. La relation de conformité est exprimée au moyen de *machines de conformité* qui représentent la combinaison d'une implantation (slave machine) avec une spécification (master machine). Ensuite, les machines de conformité sont explorées de façon exhaustive grâce à un ensemble d'outils nommé le XRT framework.

En résumé, dans la plupart des cas, il est supposé que toutes les réactions produites par les composants sont directement observables. Bien sûr, cette hypothèse rend le processus de test plus facile car le comportement de la composition est observable. Cependant dans les environnements partiellement ouverts cette hypothèse ne peut pas toujours être maintenue. Si nous reprenons l'exemple des Clouds, les messages échangés entre applications dans un Cloud sont dissimulés. Ceci peut s'illustrer grâce à la Figure 3.47. Dans l'architecture décrite, les flèches noires modélisent des interactions observables avec des applications clientes, alors que celles en pointillé modélisent des messages internes à la composition qui ne peuvent être observés.

Van der Bijl et al. proposent cependant une approche différente dans [iHBvdRT03]. La spécification décrit la composition en entier dans laquelle les messages échangés sont cachés lorsque les spécifications des composants sont réunies. Dans ce cas, chaque composant est supposé fonctionner correctement. Par conséquent, il devient possible de générer des cas de test afin de vérifier la relation de test ioco.

Notre approche offre un point de vue différent. Nous ne supposons pas que les composants soient testés. En effet, dans le cadre d'applications orientées service ou objet, des applications sont souvent codées à partir de composants existants non nécessairement testés, provenant de différentes sociétés. Nous supposons également avoir une spécification décrivant la composition en son entier dans laquelle tous les messages échangés entre composants sont donnés. Comme dit précédemment, nous considérons que les messages échangés entre composants ne peuvent être observés directement mais que chaque composant (ou une copie exacte) peut être invoqué.

3.1.2 Modélisation de compositions avec des STSs

Nous avons choisi de modéliser des compositions avec des STSs dont le langage est modifié (restreint) afin d'exprimer les notions de composants et d'instances. La définition des STSs est donnée dans [FTW05] et en Section 1.3.1.

Soit donc $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \rightarrow >$ un STS. Pour modéliser des compositions, nous supposons qu'une action $a \in \Lambda$, représente soit l'invocation d'une méthode m avec l'action $mReq$ ou bien la réponse à une méthode m décrite par $mResp$. Une telle méthode représente l'invocation d'un service fourni par un composant.

Nous notons $\mathcal{M}(\mathcal{S}) \subseteq \Lambda$ l'ensemble des méthodes dans Λ. Pour une transition $l \xrightarrow{a(p),\varphi,\varrho} l' \in \rightarrow$, nous supposons que le tuple de paramètres p est composé de variables spécifiques : $c \in I$ représente le composant appelé et $id \in I$ correspond à l'identification d'instance d'un

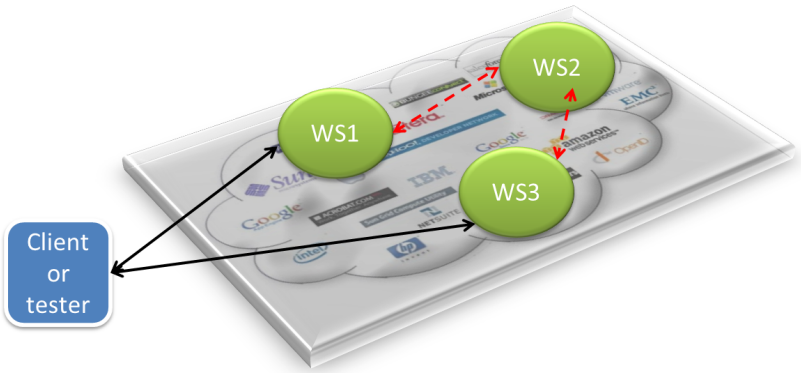


FIGURE 3.47 – Observabilité d'une composition de services dans un Cloud

composant. Un tel identifiant aide à modéliser un composant à état dans une composition i.e. un composant qui possède un état interne qui évolue pendant l'invocation de ses méthodes.

Nous définissons $component : \Lambda \times I \rightarrow D_I$ la relation qui retourne le composant référencé dans $a(p)$. $\mathcal{C}(\mathcal{S}) = \{component(a_i(p)) \mid a_i(p) \in \Lambda\}$ est l'ensemble des composants utilisés dans $\mathcal{S}$. Finalement, $session : \Lambda \times I \rightarrow D_I$ est la relation qui retourne l'identification d'instance donnée dans une action $a(p)$.

Dans la suite, nous supposons aussi que : les STSs sont déterministes, que les méthodes définies comme des relations sont bijectives et que les méthodes dépendantes sont exclusives. Ces hypothèses , qui seront expliquées dans la suite, sont requises pour décomposer des cas de test et pour les exécuter. Par conséquent, nous supposons également que les méthodes sont synchrones, i.e. une méthode retourne une réponse immédiatement ou n'en retourne pas. les méthodes asynchrones peuvent retourner une réponse n'importe quand à partir de plusieurs états différents et donc donnent souvent source à de l'indéterminisme.

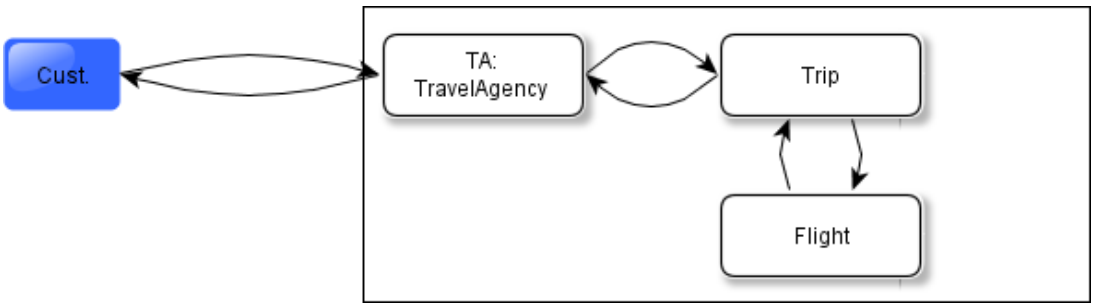


FIGURE 3.48 – Un exemple de composition

Pour illustrer ce modèle, considérons la composition décrite en Figure 3.48 où un client a la possibilité d'interagir avec une agence de voyage pour contrôler la disponibilité d'un voyage. Cette composition réunit trois composants *TravelAgency*, *Trip* et *Flight*. Son comportement est formalisé par le STS de la Figure 3.49 et la table de symboles de la Figure 3.50. Par soucis de lisibilité, les noms des composants ne sont pas donnés

par une variable c mais sont combinés directement avec les noms de méthodes. Avec cette composition, un client doit premièrement se connecter à l'agence de voyage avec la méthode*connect*, afin de récupérer une clé. Celle-ci est nécessaire pour vérifier ensuite la validité d'un voyage avec *checkTravel*. Cette méthode invoque le composant *Trip* avec *checkTrip* qui elle-même effectue une requête au composant *Flight* avec *checkFlight*. Cette méthode vérifie la disponibilité d'un vol tandis que *checkTrip* valide ou non la disponibilité d'un voyage selon le départ et la destination. Le client reçoit finalement un simple booléen qui correspond à la conjonction des résultats précédents.

L'ensemble des messages échangés entre les composants sont modélisés par des transitions exprimant les composants appelés, les méthodes et paramètres. Chaque instance de composant est modélisée par un identifiant unique e.g., ID1 pour TravelAgency. Ce composant est toujours appelé avec le même identifiant, ce qui implique dans notre cas, qu'il possède plusieurs états.

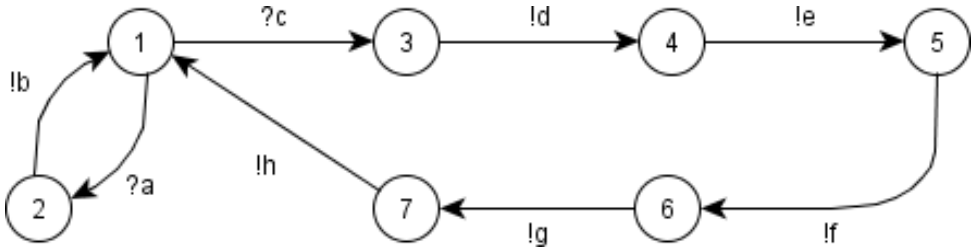


FIGURE 3.49 – Une spécification STS

A partir de cette spécification, il est tout à fait possible d'utiliser la relation ioco pour générer des cas de test. Pour les STSs, ioco a été réécrite par Rusu et al. [RMJ05] pour donner :

Définition 3.1.1 *Soit I une implantation décrite par un LTS, et $\mathcal{S}$ un STS. I est ioco-conforme à $\mathcal{S}$, noté I ioco $\mathcal{S}$ ssi $Traces(\Delta(\mathcal{S})).(\sum^O \cup \{!\delta\}) \cap Traces(\Delta(I)) \subseteq Traces(\Delta(\mathcal{S}))$.*

A partir de cette définition, on peut déduire que :

$$I \text{ ioco } \mathcal{S} \Leftrightarrow Traces(\Delta(I)) \cap NC\_Traces(\Delta(\mathcal{S})) = \emptyset$$

avec $NC\_Traces = Traces(\Delta(\mathcal{S})).\sum^O \cup \{!\delta\} \backslash Traces(\Delta(\mathcal{S}))$, les traces non conformes d'une spécification (le complément de $Traces$).

Un algorithme de génération de cas de test pour les STSs est donné dans [RMJ05]. Ces cas de test sont modélisés par des STSs ayant une structure d'arbre terminée par des états étiquetés soit par *Pass*, soit par *Fail*.

Définition 3.1.2 (Cas de test) *Un cas de test $\mathcal{T}$ est le STS déterministe $< L_{\mathcal{T}}, l0_{\mathcal{T}}, V_{\mathcal{T}}, V0_{\mathcal{T}}, I_{\mathcal{T}}, \Lambda_{\mathcal{T}} \cup \{!\delta\}, \rightarrow_{\mathcal{T}} >$ ayant un comportement fini et tel que $\{Pass, Fail\} \subseteq L$ sont des états terminaux étiquetés soit par $Pass$ soit par $Fail$.*

Soit I une implantation et $\Delta(I)$ son LTS suspension. Soit également $\mathcal{T}$ un cas de test et $T = ||\mathcal{T}||$ son LTS sémantique. L'exécution de $\mathcal{T}$ sur I est obtenue par la composition parallèle $T||\Delta(I)$ définie par le LTS $T||\Delta(I) =< Q_I \times Q_T, q0_I \times q0_T, \sum_I \cup \{!\delta\}, \rightarrow_{T||\Delta(I)} >$ où $\rightarrow_{T||\Delta(I)}$ est construit par la règle suivante :

Symb	Message	Guard	Update
?a	?TA.connectReq(String account,String id)	id==ID1	key := account
?a'	?TA.connectReq(String account,String id)	id==ID1 & account=="1234"	key := account
!b	!TA.connectResp(String id, Bool resp)	id==ID1 & resp==valid(key)	
?c	?TA.checkTravelReq(Travel trav, String id)	id==ID1 & valid(key)	t :=trav
?c'	?TA.checkTravelReq(Travel trav, String id)	id==ID1 & valid(key) & trav== {"paris-milan","2727"}	t :=trav
!d	!Trip.checkTripReq(Travel trav, String id)	id==ID2 & trav==t	
!e	!Flight.checkFlightReq(String fl, String id)	id==ID3 & fl==t.flight	f :=fl
!f	!Flight.checkFlightResp(Bool flight, String id)	id==ID3 & flight==valid(f)	b := flight
!g	!Trip.checkTripResp(Bool trip, String id)	id==ID2 & trip==(valid(t.trip)∧b)	r :=trip
!h	!TA.checktravelResp(Bool resp, String id)	id==ID1 & resp==r	

FIGURE 3.50 – Table des symboles

$$\frac{a \in \sum_I \cup \{!\delta\}, q_1 \xrightarrow{a}_T q_2, q_1' \xrightarrow{a}_{\Delta(I)} q_2'}{q_1 q_1' \xrightarrow{a}_{T||\Delta(I)} q_2 q_2'}$$

A partir de cette composition parallèle, nous pouvons définir que l'implantation I ***passe*** T ssi $T||\Delta(I)$ ne mène pas à un état $Fail$.

Notons qu'avec des spécifications déterministes, lorsque les cas de test sont composés de plusieurs états *pass*, il est alors nécessaire d'éclater ou d'exécuter plusieurs fois les cas de test afin de couvrir toutes les exécutions possibles. Pour simplifier la présentation de notre méthode, nous considérons que les cas de test sont terminés par un unique état *pass*.

Un exemple de cas de test, dérivé de la spécification de la Figure 3.49 est illustré en Figure 3.51. Celui-ci a pour but de se connecter avec le compte "1234" et de vérifier la disponibilité du voyage {"paris-milan","2727"}. Afin d'améliorer la lisibilité, nous n'avons pas étiqueté les transitions en destination de l'état Fail. Celles-ci représentent soit la réception de tout message incorrect soit l'observation de la quiescence.

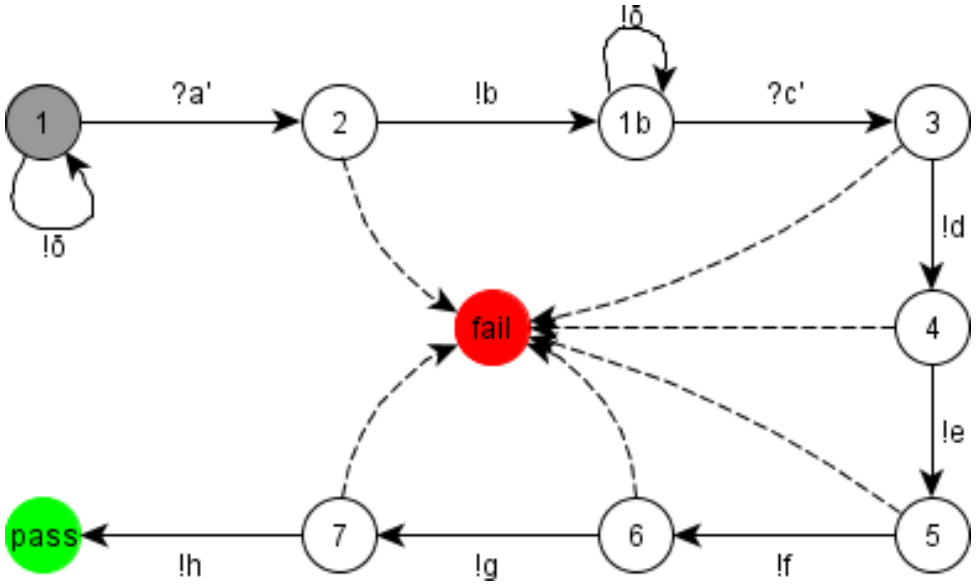


FIGURE 3.51 – Un cas de test

3.1.3 Décomposition de cas de test

La décomposition des cas de test est réalisée selon l'entrelacement (dépendance) des composants. Par suite, nous formalisons la notion de méthode de composant avec des relations, ce qui permettra de définir cette dépendance. Au préalable, nous rappelons ci-dessous des propriétés sur les relations :

Soit $R : X \to Y$ et $S : Y \to Z$ deux relations.
- $S \circ R$ ou bien $R;S = \{(x,z) \in X \times Y \mid \exists y \in Y, xRy$ et $ySz)\}$,
- R est injective : $\forall x$ et $z \in X, \forall y \in Y$, si xRy et zRy alors $x = z$,
- R est fonctionnelle : $\forall x \in X, y$ and $z \in Y$, si xRy et xRz alors $y = z$,
- R est bijective $\Leftrightarrow R$ est injective et fonctionnelle.

Définition 3.1.3 *Soit $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \to >$ un STS modélisant une composition.*

Nous associons l'invocation d'une méthode m_i, modélisée par une transition $l_i \xrightarrow{m_iReq(p),\varphi_i,\varrho_i} l_{i+1} \in \to$, avec la relation $m_iReq : D_{I\cup V} \to D_{I\cup V}$. De façon similaire, la réception d'une réponse d'une méthode, modélisé par la transition $l_i \xrightarrow{m_iResp(p),\varphi_i,\varrho_i} l_{i+1} \in \to$ est associée à la relation $m_iResp : D_{I\cup V} \to D_{I\cup V}$.

Pour une transition $l_i \xrightarrow{\delta,\varphi_i,\varrho_i} l'_i \in \to$ exprimant la quiescence, nous considérons la relation réflexive $\delta : D_V \to D_V$ tel que $\forall x \in D_V, x\delta x$.

Un chemin $l_i \xrightarrow{a(p),\varphi,\varrho} l_{i+1}...l_{j-1} \xrightarrow{a(p)',\varphi',\varrho'} l_j \in \to$, est noté $l_i \xrightarrow{\sigma=a(p);...;a(p)'} l_{j+1}$. Nous étendons cette notation avec la notion d'invocation de méthode. Ainsi, l'invocation d'une méthode $m \in \mathcal{M}(\mathcal{S})$, noté $l_i \overset{m}{\Rightarrow} l_j$, fait référence à un chemin $l_i \xrightarrow{mReq(p),\varphi,\varrho} l_{i+1}...l_{j-1} \xrightarrow{mResp(p'),\varphi',\varrho'} l_j$.

Si m ne produit pas de réponse, $l_i \overset{m}{\Rightarrow} l_{i+1}$ correspond à la transition $l_i \xrightarrow{mReq(p),\varphi,\varrho} l_{i+1}$.

Définition 3.1.4 *Soit le STS $\mathcal{S}$. Une méthode $m \in \mathcal{M}(\mathcal{S})$ est dite dépendante d'une*

relation R, notée $m \leftarrow R = mReq; R; mResp$ *s'il existe le chemin* $l_i \xrightarrow{mReq(p)\ R\ mResp(p')} l_j \in \rightarrow$ *tel que* $R \neq \delta$ *et* $R \neq \emptyset$.

Autrement, la méthode m est dite indépendante.

Nous notons également $m \nleftarrow R$ *si m est indépendante ou si* $\forall S$ *tel que* $m \leftarrow S, S \neq R$.

Une méthode peut être dépendante à une liste d'autres méthodes qui peuvent être, à leurs tours, indépendantes ou dépendantes. La spécification de la Figure 3.49 fait ressortir plusieurs dépendances de méthodes. Par exemple, la méthode *checkTravel* est dépendante de *checkTrip* qui est dépendante de *checkFlight*. Par conséquent, il est manifeste que l'on peut obtenir différents niveaux de dépendance, qui correspondent à une sorte de hiérarchie entre les méthodes. Nous formalisons cette hiérarchie grâce à un facteur appelé le *degré de dépendance*.

Définition 3.1.5 *Soit le STS* $\mathcal{S}$ *et* $m_1 \in \mathcal{M}(\mathcal{S})$ *une méthode.*

$\forall m_i \neq m_1 \in \mathcal{M}(\mathcal{S})$, *si* $m_i \nleftarrow m_1$ *alors nous fixons le degré de dépendance de* m_1 *à 1, ce qui est noté par* m_1^{d1}. *Si* $m_1 \leftarrow R$ *et* m_1^{di} *alors le degré de dépendance de R est* $i+1$, *et nous écrivons* $m_1^{di} \leftarrow R^{di+1}$. *Si* $m_1^{di} \leftarrow m_2^{di+1} ... \leftarrow m_n^{dk}$ *alors* $m_1^{di} \leftarrow m_n^{dk}$ *avec* $k > i$.

Avec la spécification de la Figure 3.49, nous pouvons maintenant écrire les dépendances suivantes : $checkTravel^{d1} \leftarrow checkTrip^{d2} \leftarrow checkFlight^{d3}$.

Cette hiérarchie dans la dépendance mène à la décomposition des cas de test. De façon intuitive, à partir d'un cas de test initial TC, nous construisons un premier cas de test décomposé TC^1 qui a objectif de tester les invocations de méthode $l_i \overset{m}{\Rightarrow} l_j$ dans TC ayant un degré m^{d1}. Ensuite, pour chaque invocation $l_i \overset{m}{\Rightarrow} l_j$ tel que $\exists R, m \leftarrow R$, nous construisons un nouveau cas de test $TC^2(l_i \overset{m}{\Rightarrow} l_j)$ pour invoquer directement les méthodes dans R ayant un degré de dépendance égal à 2. Ce processus est répété jusqu'à ce que chaque méthode dépendante ait son propre cas de test décomposé.

Cependant, cette extraction de cas de test décomposé soulève une nouvelle difficulté : les composants peuvent être de type "statefull" (ils peuvent avoir un état interne qui évolue), ce qui est décrit par un identifiant de session dans notre modélisation. Pour un tel composant, au lieu d'extraire uniquement les transitions relatives à des invocations, nous devons extraire des suites d'actions complètes i.e. des chemins à partir de l'état initial de ce composant. Ainsi, soit $TC =< L_{TC}, l0_{TC}, V_{TC}, V0_{TC}, I_{TC}, \Lambda_{TC}, \rightarrow_{TC} >$ un cas de test. Nous souhaitons extraire un cas de test décomposé depuis TC pour expérimenter un ensemble d'invocations de méthodes $\{l_i \overset{m_1}{\Rightarrow} l_j, ..., l_l \overset{m_k}{\Rightarrow} l_m\}$ avec $ID = \{session(m_i)_{1 \leq i \leq k} \mid l_i \overset{m_1}{\Rightarrow} l_j, ..., l_l \overset{m_k}{\Rightarrow} l_m \in \rightarrow_{TC}\}$. Cette extraction est faite au moyen de l'opération *keep ID* in TC dont le but est de garder les transitions de TC étiquetées par des actions ayant comme paramètre une identification de session dans ID. Le résultat est un cas de test $\mathcal{P}$, composé de transitions de TC, qui, lorsqu'il s'exécute, appelle des instances de composants dans ID uniquement. L'opération *keep* est définie comme suit :

Si $ID \subseteq \{session(a(p)) \mid a(p) \in \Lambda_{TC}\}$, *keep* ID in $TC =_{def} \mathcal{P} =< L_{TC}, l0_{TC}, V_{TC}, V0_{TC}, I_{TC}, \Lambda_{\mathcal{P}}, \rightarrow_{\mathcal{P}} >$ où $\Lambda_{\mathcal{P}}$ and $\rightarrow_{\mathcal{P}}$ sont définis par les règles suivantes :

$$R_1 : \frac{l_i \xrightarrow{a_i(p),\varphi_i,\varrho_i} l_{i'} \in \rightarrow_{TC} \wedge session(a_i(p)) \notin ID \wedge l'_i \neq \text{fail} \wedge a_i(p) \neq \delta}{l_i \xrightarrow{\tau,\emptyset,\varrho_i} l'_i \in \rightarrow_{\mathcal{P}}}$$

$$R_2 : \frac{l_i \xrightarrow{a_i(p),\varphi_i,\varrho_i} l_{i'} \in \rightarrow_{TC} \wedge a_i(p) = m_i Req(p) \wedge (session(a_i(p)) \in ID \vee l'_i = fail)}{l_i \xrightarrow{?m_i Req(p),\varphi_i,\varrho_i} l'_i \in \rightarrow_{\mathcal{P}}}$$

$$R_3 : \frac{l_i \xrightarrow{a_i(p),\varphi_i,\varrho_i} l_{i'} \in \rightarrow_{TC} \wedge a_i(p) \neq m_i Req(p) \wedge (session(a_i(p)) \in ID \vee l'_i = fail)}{l_i \xrightarrow{!a_i(p),\varphi_i,\varrho_i} l'_i \in \rightarrow_{\mathcal{P}}}$$

Ces règles produisent un STS $\mathcal{P}$ qui a le même nombre de transitions que TC. Cependant, ces règles dissimulent, avec le symbole τ, les actions de TC qui modélisent l'appel d'instances de composants qui ne sont pas référencés dans ID. De façon intuitive, utiliser de telles transitions étiquetées par τ aide à garder la structure initiale du cas de test TC. Ainsi, lors de l'exécution des cas de test, le fait de garder cette structure initiale va faciliter la recomposition des traces.

Plus précisément, la règle R_1 a pour but de dissimuler les actions de TC qui n'ont pas d'identification de session dans ID. Les transitions arrivant à fail ou étiquetées par δ ne sont pas cachées. Celles-ci n'interfèrent pas avec l'appel de composants. R_2 et R_3 signifient que si la transition du cas de test a une identification de session dans ID alors celle-ci doit être gardée. R_2 modifie également l'ensemble des actions du cas de test initial pour que les appels de méthode, qui ne sont pas dissimulés, soient traduits en entrées. Ces appels seront exécutés pour appeler chaque instance de composant dans ID.

Soient maintenant m_1 et m_2 deux méthodes telles que $m_1^{di} \leftarrow m_2^{dk}$. Pour appeler m_2, les règles précédentes extrairont, depuis un cas de test TC, toutes les transitions $l_j \xrightarrow{a_j(p),\varphi,\varrho} l_{j+1}$ qui ont le même identifiant d'instance $session(m_2)$. Si nous supposons que les deux méthodes sont appelées dans la même instance, $session(m_1) = session(m_2)$, nous obtenons une partie de cas de test $l_i \xrightarrow{m_1 Req(p),\varphi,\varrho} l_{i+1}...l_j \xrightarrow{m_2 Req(p'),\varphi',\varrho'} l_{j+1}$. Celle-ci signifie que nous appelons successivement m_1 puis m_2. Cependant, la méthode m_1 appelle déjà m_2 implicitement car $m_1^{di} \leftarrow m_2^{dk}$. Par conséquent, nous obtenons le cas de dépendance qui ne peut être décomposé. Ainsi, nous fixons l'hypothèse suivante qui implique que si une méthode m_i est dépendante de m_j alors les deux méthodes ne doivent pas appartenir à la même instance de composant.

Exclusion d'instance de méthodes : Soit un STS $\mathcal{S}$, et deux méthodes $(m_i, m_j) \in \mathcal{M}(\mathcal{S})^2$. Si $m_i^{dm} \leftarrow m_j^{dn}$ alors $session(m_i) \neq session(m_j)$.

Soit $TS = \{TC_1, ..., TC_m\}$ une suite de test initiale. Chaque cas de test $TC_n \in TS$ est décomposé avec l'algorithme 2. Ce dernier produit $TS' = \{TC'_1, ..., TC'_m\}$ où $TC'_n \in TS'$ correspond à la décomposition de TC_n. A partir d'un cas de test initial $TC_n \in TS$, cet algorithme commence par construire un ensemble d'identifiants d'instances à partir des méthodes de TC_{pass} ayant un degré de dépendance égal à 1 (lignes 2-3). Ensuite, la procédure TGGen est appelée (ligne 5). Celle-ci extrait un cas de test décomposé $TC_n^d(l_i \overset{m}{\Rightarrow} l_j)$ avec l'opération *keep* (ligne 9). Ce processus est répété pour chaque invocation de méthode $l_k \overset{m'}{\Rightarrow} l_l$ trouvée dans le chemin p tel que le degré de dépendance de m' est égal à d. Ainsi, l' algorithme construit un nouvel ensemble d'identifiants d'instances à partir des méthodes in $l_k \overset{m'}{\Rightarrow} l_l$ ayant un degré de dépendance égal à $d+1$ (ligne 13). TCGen est

appelé récursivement (ligne 14) pour produire $TC_n^{d+1}(l_k \overset{m'}{\Rightarrow} l_l)$. L'algorithme se termine lorsque chaque invocation de méthode $l_i \overset{m}{\Rightarrow} l_j$, avec m une méthode dépendante, a son propre cas de test $TC_n^d(l_i \overset{m}{\Rightarrow} l_j)$.

Par conséquent, le cas de test initial $TC_n \in TS$ est décomposé en $TC'_n = \{TC_n^1(\emptyset)\} \cup \{TC_n^d(l_i \overset{m}{\Rightarrow} l_j) \mid d > 1, m \text{ est une méthode dépendante}\}$.

A partir du cas de test illustré en Figure 3.51, nous obtenons l'ensemble de cas de test décomposé TC' donné en Figure 3.52. Le premier cas de test $TC^1(\emptyset)$ rassemble les invocations de méthodes ayant un degré de dépendance égal à 1 (*connect, checkTravel*). Le second cas de test $TC^2(l_1 b \overset{checkTravel}{\Longrightarrow} l_8)$ permet d'appeler directement la méthode *checkTrip*, composée avec *checkTravel*. Le dernier cas de test $TC^3(l_3 \overset{checkTrip}{\Longrightarrow} l_7)$ a pour but d'appeler la méthode *checkFlight*. Le nombre et l'ordre des actions sont gardés vis à vis du cas de test initial. De ce fait, lorsque ces cas de test décomposés seront exécutés, nous obtiendrons des traces partielles (incomplètes) qui seront imbriquées pour finalement produire les traces finales.

Algorithme 2: Décomposition de la suite de test

input : Une suite de test TS
output : Une suite de test décomposée TS'

1 **foreach** *cas de test* $TC_n \in TS$ **do**
2 TC_{pass} est le chemin de TC_n fini par pass;
3 $ID = \{session(m) \mid l_i \overset{m}{\Rightarrow} l_j \in TC_{pass}, m^{d1}\}$;
4 $TC'_n := \emptyset$;
5 TCGen($TC_{pass}, ID, \emptyset, 1$);
6 $TS' := TS' \cup \{TC'_n\}$;
7 TCGen(chemin STS p, ensemble d'identifiants d'instances ID, invocation de méthode $l_i \overset{m}{\Rightarrow} l_i$, degré d)
8 //extraire un cas de test décomposé avec l'opération *keep*;
9 $TC_n^d(l_i \overset{m}{\Rightarrow} l_j) :=$ *keep* ID in TC_n;
10 $TC'_n := TC'_n \cup TC_n^d(l_i \overset{m}{\Rightarrow} l_j)$;
11 **foreach** *invocation de méthode* $l_k \overset{m'}{\Rightarrow} l_l \in p$ *avec* m'^{dd} **do**
12 **if** m' *est une méthode dépendante* **then**
13 $ID = \{session(m) \mid l_q \overset{m}{\Rightarrow} l_r \in l_k \overset{m'}{\Rightarrow} l_l, m^{dd+1}\}$;
14 TCGen($l_k \overset{m'}{\Rightarrow} l_l$,$ID$, $l_k \overset{m'}{\Rightarrow} l_l, d+1$);

3.1.4 Exécution des tests et recomposition de trace

Afin de raisonner sur la notion de conformité, nous supposons avoir une implantation sous test $\mathcal{I}$, modélisée par un LTS suspension qui est supposé avoir le même alphabet que sa spécification ($\Delta(||\mathcal{S}||)$). Pendant la phase d'exécution des tests, le testeur a pour fonction de recomposer les traces $STraces(\mathcal{I})$ en exécutant les cas de test décomposés dans $TS' = \{TC'_1, ..., TC'_m\}$.

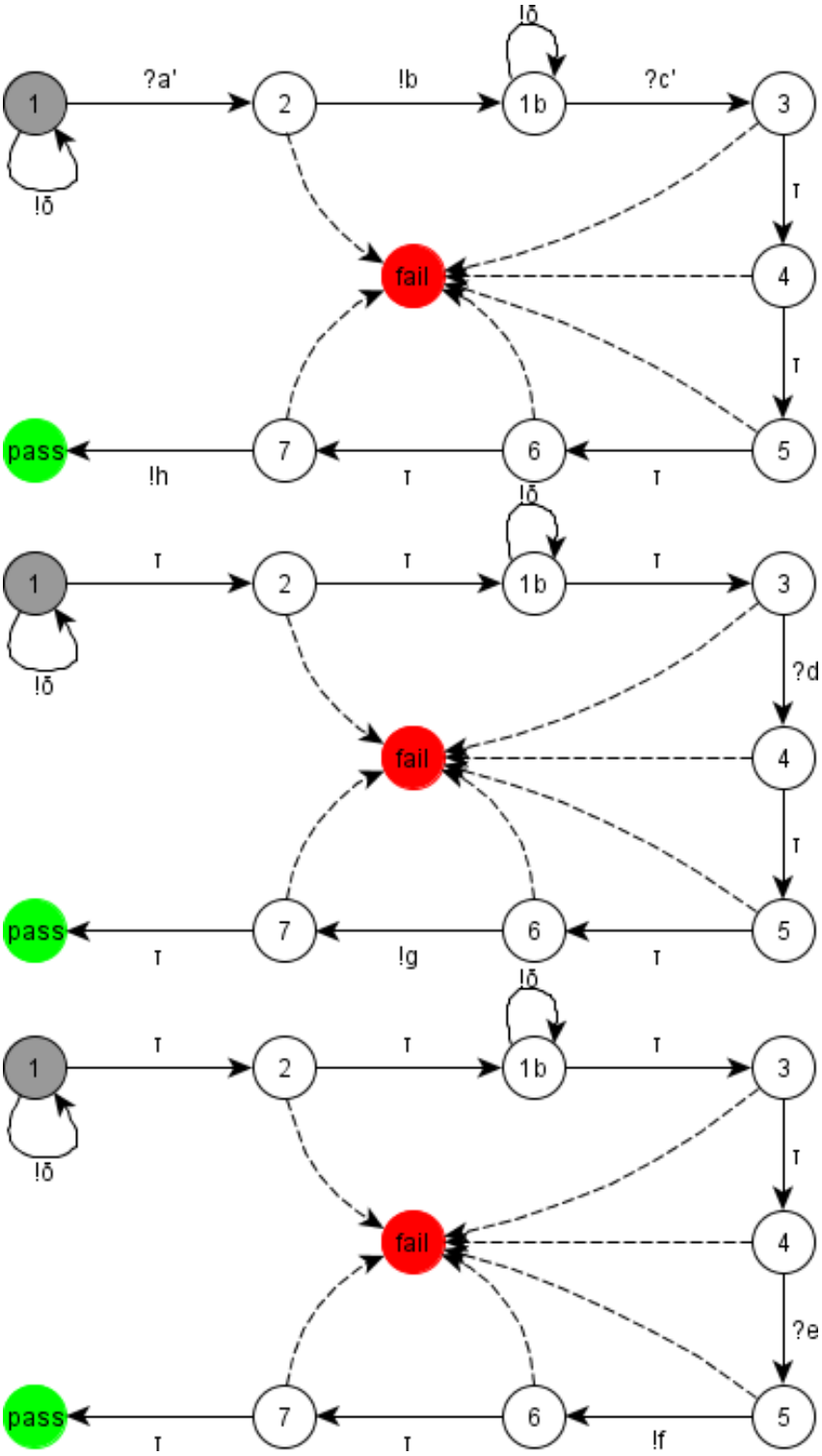


FIGURE 3.52 – Un cas de test décomposé

La recomposition de trace est effectuée en se référant toujours à la dépendance des méthodes. Cependant, cette dépendance ne peut être observée explicitement lors de l'exécution car les messages échangés entre les composants ne sont pas observables. Néanmoins, si nous supposons que toute méthode dépendante d'une spécification est bijective, alors nous pouvons affirmer qu'une méthode m est *probable dépendante à* une relation R, si m répond correctement lors du test (m retourne une réponse comme il est décrit dans la spécification).

Définition 3.1.6 *Soit le cas de test TC, composé d'une invocation de méthode $l_i \overset{m}{\Rightarrow} l_j \in \rightarrow_{TC}$ tel que $\exists R, m \leftarrow R$. Si m est bijective et si l'implantation sous test $\mathcal{I}$* passe *TC alors m est dite probablement dépendante à R.*

La relation R est une composition quelconque sur $\mathcal{M}(\mathcal{S}) \cup \{\delta\}$. Chaque relation dans $\mathcal{M}(\mathcal{S}) \cup \{\delta\}$ est bijective, ainsi R est aussi bijective. $m \leftarrow R$ correspond à $mReq; R; mResp$. Ainsi, soient $w, x, y, z \in D_{V \cup I}$ tel que w mReq x, xRy et y mResp z. $\mathcal{I}$ passe TC implique que la méthode m a retourné une réponse correcte. Supposons que $mReq(w)$ soit invoquée et retourne la réaction $mResp(z)$. $mReq$ est fonctionnelle donc il existe un unique $x' \in D_{V \cup I}$ tel que w mReq x'. $mResp$ est injective donc il existe un unique $y' \in D_{V \cup I}$ tel que y' mResp z. Par conséquent, m semble être probablement dépendante à une relation R tel que $x'Ry'$ avec $x = x'$ et $y = y'$.

L'algorithme de recomposition de trace est décrit dans l'algorithme 3. A partir d'un cas de test décomposé $TC'_n = \{TC^1_n(\emptyset)\} \cup \{TC^d_n(l_i \overset{m}{\Rightarrow} l_j) \mid d > 1, m$ est une méthode dépendante $\}$, le testeur ré-assemble une trace $trace_n(\mathcal{I})$ avec la procédure *Exec* (ligne 3). Il commence par exécuter $TC^1_n(\emptyset)$ sur $\mathcal{I}$. Par exemple, si ce cas de test est composé de l'invocation $l_i \overset{m}{\Rightarrow} l_j = l_i \xrightarrow{?mReq(p),\varphi,\varrho} l_{i+1} \xrightarrow{\tau} l_{i+2} ... l_{j-2} \xrightarrow{\tau} l_{j-1} \xrightarrow{!mResp(p),\varphi',\varrho'} l_j$, le testeur invoque la méthode m, puis ignore les actions τ et soit atten une réponse soit observe la quiescence. Les actions obtenues sont ré-injectées dans $trace_n(\mathcal{I})$ (lignes 6-8). Pour chaque invocation $l_k \overset{m'}{\Rightarrow} l_l$ inclue dans le cas de test, si $\mathcal{I}$ passe $TC^d_n(l_i \overset{m}{\Rightarrow} l_j)$, alors la méthode m' a répondu correctement. Selon la Définition 3.1.6, si $m' \leftarrow R$, m' semble probablement dépendante de R. Donc, le testeur exécute le cas de test $TC^{d+1}_n(l_k \overset{m'}{\Rightarrow} l_l)$ pour récupérer les traces de la méthode m' (lignes 10-11). Ce processus se poursuit pour chaque méthode dépendante pour finalement recomposer $trace_n(\mathcal{I})$. Si une méthode ne retourne pas une réponse correcte ($\mathcal{I}$ ne passe pas un cas de test décomposé), alors la reconstruction de trace est stoppée (ligne 13).

Considérons maintenant le STS $\mathcal{S}$, l'implantation sous test $\mathcal{I}$ et un cas de test TC. Supposons que $\mathcal{I}$ ioco $\mathcal{S}$ et que TC produise la trace $t_1 ... t_n$ en utilisant une méthode de test ioco classique. Si nous appliquons les Algorithmes 2 et 3 sur TC et $\mathcal{I}$, nous obtenons la même trace $t_1 ... t_n$. Une ébauche de preuve est donnée ci-dessous :

TC est transformé avec l'Algorithme 2 en $\{TC^1(\emptyset)\} \cup \{TC^d(l_i \overset{m_i}{\Rightarrow} l_j) \mid m_i$ est dépendante, $d > 1\}$. $TC^1(\emptyset) = p.l_1 \overset{m_1}{\Rightarrow} l_2 ... l_i \overset{m_i}{\Rightarrow} l_j ... l_k \overset{m_k}{\Rightarrow} l_n$ avec m_i^{d1} $(1 \leq i \leq k)$ (Algorithme2 et l'opération *keep*). p est un chemin qui invoque chaque composant dans $\{component(m_i) \mid 1 \leq i \leq k\}$ à partir de leurs états initiaux. Comme le degré de dépendance de chaque invocation de méthode est égal à 1, $p = \emptyset$ ici. Considérons une invocation $l_i \overset{m_i}{\Rightarrow} l_j$ avec $c = component(m_i)$ le composant appelé :

- si m_i n'est pas dépendante, alors $l_i \overset{m_i}{\Rightarrow} l_j$ correspond à l'un des chemins suivants de $TC^1(\emptyset)$: $l_i \xrightarrow{m_i Req} l_{i+1}$, $l_i \xrightarrow{\delta} l_{i+1}$, $l_i \xrightarrow{m_i Req\ m_i Resp} l_{i+2}$ ou $l_i \xrightarrow{m_i Req \delta ... \delta m_i Resp} l_j$ (

Algorithme 3: Algorithme du testeur

input : Un ensemble de cas de test décomposés TS'
output : $STraces(\mathcal{I})$

1 **foreach** *cas de test* $TC'_n \in TS'$ **do**
2 $\quad trace_n(\mathcal{I}) = t_0...t_k$ est une trace;
3 $\quad$ Exec($TC^1_n(\emptyset)$, $trace_n(\mathcal{I})$);
4 $\quad STraces(\mathcal{I}) := STraces(\mathcal{I}) \cup trace_n(\mathcal{I})$;

5 Exec(cas de test $TC^d_n(l_i \overset{m}{\Rightarrow} l_j)$, trace $trace_n(\mathcal{I})$)
6 $trace'(\mathcal{I}) = Test(TC^d_n(l_i \overset{m}{\Rightarrow} l_j))$;
7 **foreach** $t'_k \in trace'(\mathcal{I}) \neq \tau$, $i < k < j-1$ **do**
8 $\quad t_k := t'_k$;

9 **foreach** *invocation* $l_k \overset{m'}{\Rightarrow} l_l \in TC^d_n(l_i \overset{m}{\Rightarrow} l_j)$ *avec* m' *une méthode dépendante* **do**
10 $\quad$ **if** $\mathcal{I}$ *passe* $TC^d_n(l_i \overset{m}{\Rightarrow} l_j)$ *(*m' *est probablement dépendante)* **then**
11 $\quad\quad$ Exec($TC^{d+1}_n(l_k \overset{m'}{\Rightarrow} l_l)$, $trace_n(\mathcal{I})$);
12 $\quad$ **else**
13 $\quad\quad$ Stop;

nous avons émis l'hypothèse que les méthodes sont synchrones et que les STSs sont déterministes). A partir de ces chemins, l'Algorithme 3, produit respectivement les traces partielles t'_i, δ, $t'_i t'_{i+1}$ ou $t'_i \delta ... \delta t_{j-1}$ avec $t'_i = t_i$, $t'_{i+1} = t_{i+1}$ et $t'_{j-1} = t_{j-1}$ car le composant c est appelé depuis son état initial et car l'opération *keep* garde l'ordre des transitions du cas de test initial. Ainsi, nous obtenons les mêmes réactions observables que celles produites par une méthode de test ioco classique,

– si m_i est dépendante, $l_i \overset{m_i}{\Rightarrow} l_j$ correspond alors à l'un des chemins suivants de $TC^1(\emptyset)$: $l_i \xrightarrow{m_i Req} l_{i+1} \xrightarrow{\tau} l_{i+2}...l_{j-2} \xrightarrow{\tau} l_{j-1} \xrightarrow{m_i Resp} l_j$ (opération *keep*). L'algorithme 3 produit à partir de ces chemins les traces partielles $t'_i \tau ... \tau t'_{j-1}$ avec $t'_i = t_i$ et $t'_{j-1} = t_{j-1}$ car le composant c est appelé à partir de son état initial et car l'opération *keep* garde l'ordre des transitions du cas de test initial. Comme nous supposons que $\mathcal{I}$ ioco $\mathcal{S}$, $\mathcal{I}$ passe $TC^1(\emptyset)$ et $m_i Resp$ est donc probablement dépendante (Algorithme 3 ligne 10). Par conséquent, il existe un cas de test décomposé $TC^2(l_i \overset{m_i}{\Rightarrow} l_j)$ (Algorithme 2 lignes 12-14) tel que $TC^2(l_i \overset{m_i}{\Rightarrow} l_j) = p'.l_{i+1} \xrightarrow{m_{i+1} Req} l_k...l_o \overset{m_o}{\Rightarrow} l_p...l_q \overset{m_q}{\Rightarrow} l_{j-1}$, avec p' un chemin qui appelle chaque composant dans $\{component(m_o) \mid i+1 \leq o \leq q\}$ à partir de leurs états initiaux.

Nous pouvons appliquer le même raisonnement sur $TC^2(l_i \overset{m_i}{\Rightarrow} l_j)$ et pour chaque méthode dépendante et indépendante jusqu'à ce qu'il n'y ait plus de cas de test à exécuter. Comme l'opération *keep* produit des chemins composés du même nombre de transitions que le cas de test initial dans le même ordre et comme les cas de test décomposés invoquent des composants à partir de leurs états initiaux, nous obtenons finalement une trace $t_1...t_n$.

Si nous supposons maintenant que $\neg(\mathcal{I}$ ioco $\mathcal{S})$, nous pouvons obtenir une trace $t_1...t_n$

composée ou bien de réponses incorrectes ou bien de symboles τ avec l'Algorithme 3. Dans les deux cas, l'application de la trace $t_1...t_n$ sur le cas de test initial TC mène à un état Fail. En d'autres termes, $t_1...t_n \in Traces_{FAIL}(TC)$.

Ainsi, soit le STS $\mathcal{S}$, l'implantation sous test $\mathcal{I}$ et le cas de test $TC \in TS$.

$\mathcal{I}$ ioco $\mathcal{S} \Rightarrow$ l'application des Algorithmes 2 et 3 produit une trace $t \in STraces(\mathcal{I})$ tel que $t \notin Traces_{FAIL}(TC)$

$\neg(\mathcal{I}$ ioco $\mathcal{M}) \Rightarrow$ l'application des Algorithmes 2 et 3 produit une trace $t \in STraces(\mathcal{I})$ tel que $t \in Traces_{FAIL}(TC)$

3.1.5 Discussion

La complexité des algorithmes 2 et 3 reste raisonnable car elle est proportionnelle à $\mathcal{O}((k+1)n^2)$ avec k le nombre de cas de test et n le plus grand nombre de transitions dans un cas de test. A titre de comparaison, exécuter un cas de test initial dans un environnement où toutes les réactions de l'implantation peuvent être observées donne une complexité de $\mathcal{O}(kn)$. Plus précisément, à partir d'un cas de test initial TC de n transitions, nous avons au plus n invocations de méthodes. Dans le pire des cas, nous construisons un cas de test par invocation en couvrant TC deux fois (Algorithme 2). Ainsi, pour k cas de test, la complexité est égale à $\mathcal{O}(kn^2)$. Nous avons au plus $(n/2+1)$ cas de test décomposés générés à partir de TC dont la taille ne peut excéder n transitions. Par conséquent, la complexité de l'exécution des cas de test décomposés est proportionnelle à $\mathcal{O}(n^2)$.

Le second point d'attention concerne la recomposition de $STraces(\mathcal{I})$ pendant le test. Une fois que nous avons $STraces(\mathcal{I})$, nous pouvons vérifier que pour un cas de test TC, $STraces(\mathcal{I}) \cap Traces_{Fail}(TC) \neq \emptyset$ et finalement conclure si $\mathcal{S}$ ioco $\mathcal{I}$. Ceci soulève la question suivante : devrions-nous toujours nommer la relation de test ioco ? Les cas de tests sont générés par une méthode de test de type ioco, et nous reconstruisons des traces complètes de $STraces(\mathcal{I})$. Néanmoins, la recomposition de trace est effectuée en supposant que les méthodes sont probablement dépendantes alors que cette propriété n'est pas prise en considération dans la théorie ioco. Ainsi, devrions-nous redéfinir une relation plus faible ? Nous ne pensons pas pour la raison suivante : si nous avons des méthodes bijectives et $m \leftarrow R$ dans la spécification et si l'appel de la méthode m dans l'implantation répond de façon correcte, nous faisons face aux possibilités suivantes : 1) m est indépendante mais simule l'appel d'une relation R et retourne une réponse correcte, 2) m est dépendante d'autres méthodes $m \leftarrow R'$ qui ont le même comportement que R, 3) m est réellement dépendante de R. Implanter les deux premiers cas à partir d'une spécification semble être farfelu car ceux-ci correspondent à des modifications très inhabituelles de la spécification. C'est pourquoi nous pensons que seul le dernier cas est raisonnable. Néanmoins, il est manifeste que proposer une relation de test basée sur la recomposition de traces serait une perspective intéressante.

3.1.6 Perspectives sur la décomposition de cas de test dans des environnements partiellement ouverts

Les perspectives sont notamment liées aux hypothèses qui ont été formulées afin de pouvoir décomposer un cas de test puis de recomposer les traces partielles obtenues.

Nous avons supposé que les spécifications sont déterministes et les opérations synchrones. Cette hypothèse est nécessaire pour la génération des cas de test initiaux, la décomposition des cas de test et la recomposition de traces. Néanmoins des travaux sur la génération de cas de test à partir de spécifications non déterministes ont été proposés par exemple dans [JMR06]. Ainsi, nous pourrions incorporer ces travaux dans notre méthode. Nous pourrions peut être ainsi prendre en compte des méthodes asynchrones au lieu de ne considérer que des méthodes synchrones.

Nous avons supposé que les méthodes sont bijectives afin de poser qu'une méthode est probablement dépendante si elle retourne un résultat spécifié. Sans cette hypothèse, la reconstruction de traces perd tout son sens car elle repose sur la hiérarchisation induite de la dépendance entre les méthodes. Poser comme hypothèse que les méthodes sont bijectives a tout son sens : en fait, il est défini dans [KDCK94] qu'une relation est testable (et plus précisément que la faisabilité des tests est au plus haut) ssi elle est bijective. Cependant, les méthodes ou services développés dans des cas réels ne sont pas toujours liés à des relations bijectives. Peut-on donc recomposer des traces sans cette hypothèse ?

En complément, la notion de dépendance probable, qui est l'hypothèse forte de cette approche, a besoin d'être étudiée dans des travaux futurs. Il serait en effet intéressant de garantir qu'une méthode est dépendante et pas seulement probablement dépendante. Cependant, nous ne pouvons utiliser les messages internes à la composition car ceux-ci sont supposés être cachés (ce qui correspond à la réalité dans de nombreux environnements réels comme les Clouds par exemple). Une solution technique et triviale pourrait être proposée en implantant dans chaque composant une sorte d'historique accessible via une méthode supplémentaire. Mais bien entendu, ce type de solution ne peut être généralisé et ne peut s'appliquer avec des composants existants.

Enfin, nous avons considéré que les compositions sont statiques (les composants sont toujours les mêmes et connus, la spécification est statique). Or, une tendance récente, qui commence à attirer de l'engouement, concerne les compositions dynamiques dans lesquelles les composants et les méthodes sont choisis dynamiquement pendant l'exécution de la composition. Une piste à suivre, pour le test de ce type d'applications dans un environnement partiellement ouvert, est de rendre abstraits les composants, les instances de composants et les méthodes dans la spécification. Ainsi avec un STS, le nom des composants mais surtout des méthodes pourrait être donné par des variables internes. Ensuite, il faut définir la décomposition de cas de test abstraits et recomposer des traces concrètes pendant l'exécution des cas de test décomposés. La relation de test est ici nécessairement à revoir, car une implantation est conforme à sa spécification à un instant t par rapport aux composants appelés. Autrement dit, La dynamicité des composants appelés devrait être prise en compte dans la relation de test.

3.2 Le test passif dans des environnements partiellement ouverts avec des proxy-testeurs

Le test passif a pour objectif d'analyser passivement les réactions d'une implantation sans la perturber. Par rapport au test actif, il offre des avantages certains tels que la publication rapide d'une application ou d'un système, ou le test sans interrompre le fonctionnement normal d'une implantation de façon arbitraire.

Cette passivité est généralement obtenue grâce à des modules basés sur des sniffeurs, qui récupèrent les traces de l'implantation sous test. Les traces obtenues peuvent être utilisées pour vérifier que le comportement de l'implantation correspond à celui de sa spécification ou pour vérifier la satisfaction de propriétés appelées *invariants*.

Néanmoins, le test passif soulève aussi son lot d'hypothèses. En particulier, il est implicitement supposé que ces modules passifs peuvent avoir un accès complet à l'interface de l'implantation et qu'ils peuvent être donc placés dans le même environnement. Plusieurs environnements satisfont à cette hypothèse, d'autres non. De même que l'approche précédente, si l'infrastructure dans laquelle est déployée l'implantation, offre un accès restreint pour des raisons de sécurité ou techniques, les réactions de l'implantation ne pourront être lues et donc le test passif ne pourra être effectué. Si l'on reprend l'exemple du Cloud computing, les Clouds formant des environnements virtualisés dynamiques, il devient impossible de placer des modules passifs pour sniffer des flux. En effet, il n'est pas possible de savoir à l'avance où sont placées les applications et les couches basses (serveurs réels) ne sont pas accessibles.

Nous proposons une autre approche, pour le test passif, en s'orientant vers la notion de proxy et en formalisant le test par proxy-testeur. Ce dernier s'apparente à une application intermédiaire entre les applications clientes et l'implantation. Un proxy-testeur reçoit donc le trafic client, qu'il redirige ensuite vers l'implantation et vice-versa. Tout en recevant les réactions de l'implantation, un proxy-testeur va aussi être capable de détecter un comportement incorrect et ainsi de conclure à la non conformité d'une implantation. De plus, il offre un maximum de flexibilité car il peut être déployé dans le même environnement que l'implantation ou pas, à condition que le proxy-testeur puisse appeler l'implantation (comme un client). Dans la suite, nous définissons formellement la notion de proxy-testeur pour les systèmes symboliques. Nous montrons également que les proxy-testeurs proposent de nombreuses perspectives.

3.2.1 Présentation de quelques travaux sur le test passif et motivations

Plusieurs travaux, ciblant le test passif de protocoles ou de composants (services Web), ont été proposés récemment. Nous n'en citerons ici que quelques-uns. Certains de ces travaux supposent que le testeur (aussi appelé observeur) s'initialise en même temps que l'implantation, d'autres non. Dans ce dernier cas, les dernières approches considèrent généralement la combinaison d'une approche en avant et en arrière.

- **Satisfiabilité d'invariants [BCNZ05, CGP03, CMdO09] :** un invariant représente des propriétés toujours vraies. Il est construit à partir de la spécification et est plus tard vérifié sur les traces collectées. Le test passif par invariant aide à tester des propriétés complexes mais a l'inconvénient de nécessiter de construire les

invariants à la main. Cette approche a donné lieu à plusieurs travaux. Par exemple, dans [CMdO09], la méthode de test passive est basée sur la satisfiabilité d'invariants par les traces de l'implantation. Cette méthode est découpée en plusieurs étapes que l'on peut résumer comme suit : définition d'invariants à partir de la spécification, extraction de traces au moyen de sniffeurs, vérification des invariants suivant les traces obtenues.
D'autres travaux ont porté sur le test de services Web : dans ce cas, les méthodes passives sont généralement utilisées pour tester la conformité ou la sécurité. Dans [MMC+10], le niveau de confiance entre l'implantation et sa spécification est défini par des invariants également. Comme dans [CMdO09], ces invariants sont construits manuellement à partir de la spécification et les traces sont collectées par des sniffeurs. Ensuite, l'outil *TIP* exécute une analyse automatique des traces capturées pour déterminer si les invariants sont satisfaits ou non. La sécurité de compositions de services est passivement testée dans [CBML09]. Des règles de sécurité sont modélisées avec le langage Nomad qui peut exprimer l'autorisation ou l'interdiction avec des propriétés temporelles. Premièrement, un ensemble de règles est manuellement construit à partir de la spécification. Des traces de l'implantation sont extraites avec des modules placés dans chaque couche des moteurs d'exécution de services Web. Ensuite, il est vérifié que les traces satisfont les règles de sécurité,

- **Vérification en avant [LNS+97, MA00, LCH+06] :** les traces sont collectées à la volée et données à un algorithme dont le but est de vérifier si les réactions de l'implantation satisfont la spécification en la parcourant. Dans [LCH+06], Lee et al. proposent une méthode de test passive pour les protocoles filaires modélisés par des Extended Finite State Machines (EEFSM), composés de variables. Plusieurs algorithmes sur les EEFSM et leurs applications aux protocoles OSPF et TCP sont présentés. Ces algorithmes ont pour but de vérifier si des traces, composées d'actions et de paramètres, satisfont la spécification à la volée. Comme il est décrit dans la partie expérimentation, les réactions de l'implantation sont extraites grâce à des sniffeurs,
- **Vérification en avant/arrière [ACC+04] :** dans cette approche, les traces de l'implantation sont analysées vis-à-vis de la spécification, en arrière, afin de réduire les suites d'actions possibles de la spécification. L'approche est généralement scindée en deux étapes : premièrement un algorithme suit une trace donnée en arrière depuis un état valué vers un ensemble d'états valués de départ. Ensuite, un second algorithme analyse le passé de ces états valués, également en arrière, afin de rechercher des états valués dans lesquels toutes les valeurs de variable sont connues. Lorsque de tels états sont atteints, une décision est effectuée sur la validité des chemins obtenus. Dans [BDS+07], des services Web sont passivement testés en combinant le fait de recevoir des évènements (analyse en avant) et une analyse en arrière pour déduire des états et valuations de variables lorsque l'observeur est en attente d'un évènement. Plusieurs algorithmes sont donnés.

Quelques architectures de test ont été également proposées dans la littérature pour le test passif. Par exemple dans [BDSG09], des architectures sont proposées pour observer des services et des compositions de services. Celles-ci sont aussi analysées en terme de performance. Dans ce cas également, les observeurs sont placés dans l'environnement des compositions en supposant qu'il n'y a pas de restriction.

Ainsi, à notre connaissance, les méthodes passives de test proposées dans la littérature et notamment les précédentes, s'appuient sur des environnements ouverts pour extraire toutes les réactions de l'implantation sous test (messages, paquets, etc.). Nous considérons ici que l'accès à l'environnement, dans lequel est déployé l'implantation, est restreint. Nous définissons un proxy-testeur qui intercepte le traffic client et qui évite de positionner des observeurs de type sniffeurs.

3.2.2 Définition du Proxy-testeur

Dans ce qui suit, nous supposons qu'une spécification est modélisée par un STS $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \rightarrow >$, dont la définition est donnée en Section 1.3.1. Nous supposons que tous les états de cette spécification acceptent n'importe quelle entrée de l'alphabet ("input enabled", hypothèse nécessaire pour la relation ioco). Pour illustrer notre méthode, nous prenons l'exemple de spécification illustré en Figure 3.53. Celle-ci décrit une application de type bancaire qui possède deux méthodes *lg* pour se connecter sur le système bancaire et *transfer* dont le but est de permettre d'effectuer des virements bancaires vers un compte. Cette spécification utilise une méthode interne *valid* qui retourne *true* si les comptes bancaires ou les clés sont valides et *false* autrement.

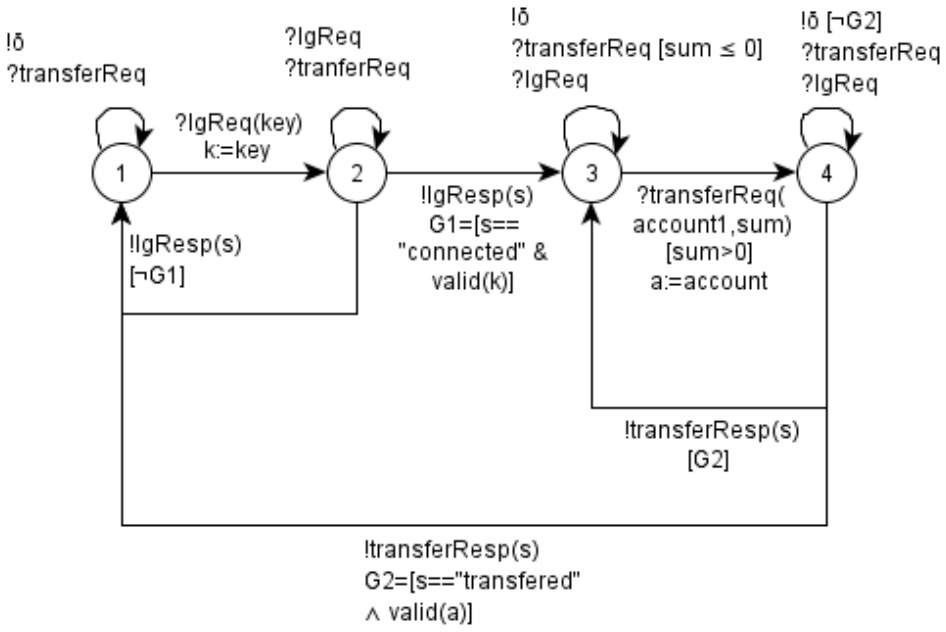


FIGURE 3.53 – Une spécification décrite par un STS suspension

Afin de réfléchir à la notion d'équivalence entre une spécification $\mathcal{S}$ et une implantation sous test I, cette dernière est habituellement supposée se comporter comme un LTS sémantique (voir Section 1.3.1 pour les définitions). Lors de la phase de test, il est ainsi possible d'extraire des exécutions et des traces (valuées) depuis l'implantation.

Définition 3.2.1 (Exécutions et traces) *Pour un STS $\mathcal{S}$, interprété par son LTS sémantique $||\mathcal{S}|| =< Q, q_0, \sum, \rightarrow >$, une exécution $q_0\alpha_0...\alpha_{n-1}q_n$ est une séquence alternée*

d'états et d'actions valuées. $RUN(\mathcal{S}) = RUN(||\mathcal{S}||)$ *est l'ensemble des exécutions trouvées dans* $||\mathcal{S}||$. $RUN_F(\mathcal{S})$ *est l'ensemble des exécutions de* $\mathcal{S}$ *terminées par un état dans* $F \subseteq Q$.

Il suit qu'une trace d'une exécution r *est définie comme étant la projection* $proj_{\sum}(r)$ *sur les actions. Ainsi,* $Traces_F(\mathcal{S}) = Traces_F(||\mathcal{S}||)$ *est l'ensemble des traces des exécutions terminées par un état dans* $F \subseteq Q$.

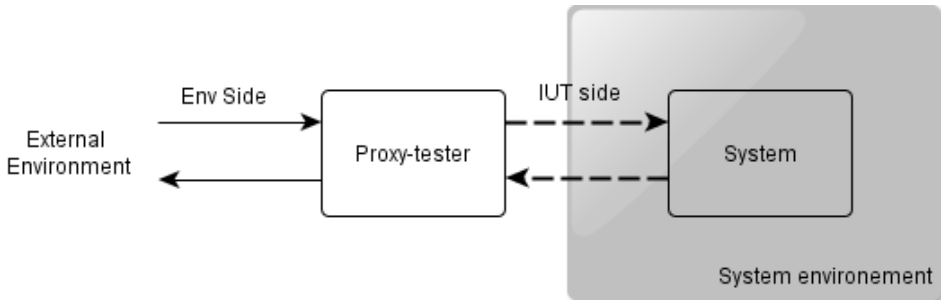


FIGURE 3.54 – Une utilisation d'un proxy-testeur

Après ces quelques définitions, passons au concept de proxy-testeur : la Figure 3.54 illustre un exemple d'utilisation. Celui-ci peut être déployé dans l'environnement de l'implantation ou en dehors, à condition qu'il puisse interagir avec l'implantation sous test. Depuis le côté client, il remplace (dissimule) l'implantation donc il est requis que le trafic client soit envoyé au proxy-testeur. Ce dernier doit donc interagir avec les applications clientes comme il est décrit dans la spécification. Il est certain qu'il ne doit pas agir à la place de l'implantation, donc chaque action reçue est renvoyée vers celle-ci. Il doit aussi être capable de recevoir des actions depuis l'implantation et de les renvoyer vers l'environnement client. Mais il doit aussi analyser ces actions pour vérifier la conformité de l'implantation.

En résumé, un proxy-testeur doit se comporter comme la spécification pour interagir avec l'environnement client et doit aussi se comporter comme une spécification miroir pour s'interconnecter avec l'implantation sous test. Pour analyser les traces correctes de l'implantation (celles trouvées dans la spécification) et les traces incorrectes, cette spécification miroir doit aussi s'accompagner du comportement incorrect. Cette seconde partie correspond généralement au testeur canonique de la spécification. Toutefois, un tel testeur canonique doit généralement être déterministe (car il est vu comme un cas de test). Or, cette hypothèse n'est pas requise ici. Pour ne pas faire d'amalgame, nous utiliserons le terme de STS réflexion.

Un STS réflexion réunit d'une part les transitions de la spécification étiquetées par les actions miroir (les entrées deviennent des sorties et vice versa) et d'autre part les transitions menant à un nouvel état symbolique $Fail$, modélisant la réception d'actions non spécifiées.

Définition 3.2.2 (STS réflexion) *Soit* $\mathcal{S} =< L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}} >$ *un STS et* $\Delta(\mathcal{S})$ *son suspension. Le STS réflexion de* $\mathcal{S}$ *est le STS* $REF(\mathcal{S}) =< L_{REF}, l0_{REF}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{REF}, \rightarrow_{REF} >$ *tel que :*

- $\Lambda^I_{REF} = \Lambda^O_{\mathcal{S}} \cup \{?\delta\}$ *et* $\Lambda^O_{REF} = \Lambda^I_{\mathcal{S}}$,
- $L_{REF}, l0_{REF}, \rightarrow_{REF}$ *sont définis par les règles suivantes :*

(keep $\mathcal{S}$ transitions) : $\dfrac{t \in \rightarrow_{\Delta(\mathcal{S})}}{t \in \rightarrow_{REF}}$

(incorrect behaviour completion) : $\dfrac{a \in \Lambda_{\mathcal{S}}^{O} \cup \{!\delta\}, l_1 \in L_{\mathcal{S}}, \varphi_a = \bigwedge_{l_1 \xrightarrow{a(p),\varphi_n,\varrho_n}_{\Delta(\mathcal{S})} l_n} \neg\varphi_n}{l_1 \xrightarrow{?a(p),\varphi_a,\emptyset}_{REF} Fail}$

Le STS réflexion de la spécification donnée en Figure 3.53, est illustré en Figure 3.55. Le STS résultant est complété sur l'ensemble des entrées. Par exemple, à partir de l'état symbolique 2, de nouvelles transitions menant à *Fail* sont ajoutées pour modéliser la réception d'actions non spécifiées ou la quiescence.

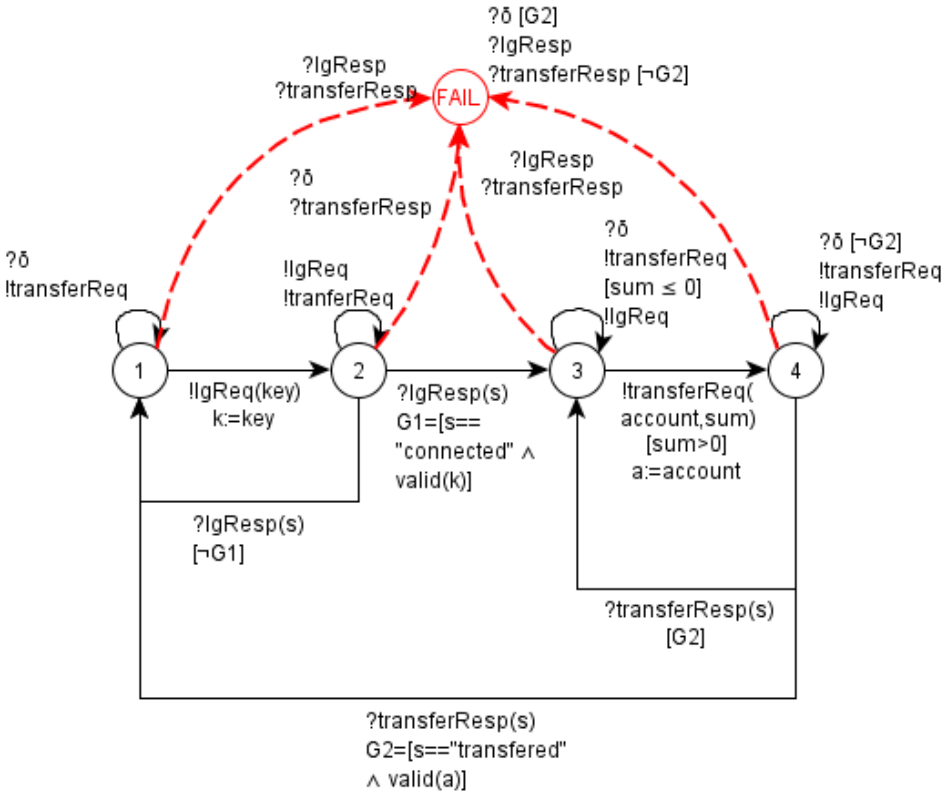


FIGURE 3.55 – Un STS réflexion

Pour combiner sans ambiguïté un STS $\mathcal{S}$ et son STS réflexion $REF(\mathcal{S})$, nous séparons les variables, états symboliques, gardes et assignations grâce à une fonction de renommage. Afin de simplifier la lecture, nous utilisons la même fonction ϕ. Par rapport à $\mathcal{S}$, les variables internes et externes de $REF(\mathcal{S})$ sont renommées par $\phi : V \cup I \rightarrow V' \cup I'$, $\phi(v) \rightarrow v'$. Les états symboliques sont renommés par $\phi L \rightarrow L'$, $\phi(l) \rightarrow l'$ et ainsi de suite.

Pour un STS $\mathcal{S}$, nous notons également $\phi(\mathcal{S}) =< \phi(L_{\mathcal{S}}), \phi(l0_{\mathcal{S}}), \phi(V_{\mathcal{S}}), \phi(V0_{\mathcal{S}}), \phi(I_{\mathcal{S}}), \Lambda_S, \rightarrow_{\phi(\mathcal{S})} >$ où $\rightarrow_{\phi(\mathcal{S})}$ est l'ensemble fini des transitions $l'_1 \xrightarrow{a(p'),\varphi',\varrho'}_{\phi(\mathcal{S})} l'_2$ avec $\varphi' = \phi(\varphi)$ une garde et $\varrho' = \phi(\varrho)$ une affectation de variables $\varrho' : V' \times p' \times \rightarrow V'$.

Dans la définition suivante de proxy-testeur, nous ajoutons également une variable interne *side* qui permet de clairement identifier les interactions de l'environnement extérieur ($side := Env$) et les interactions de l'implantation ($side := IUT$).

Nous pouvons maintenant définir la notion de proxy-testeur :

Définition 3.2.3 (proxy-testeur) *Le proxy-testeur $\mathcal{P}(\mathcal{S})$ d'une spécification $\mathcal{S} =< L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}}>$ est la combinaison de $\Delta(\mathcal{S})$ avec son STS réflexion $\phi(REF(\mathcal{S}))$. $\mathcal{P}(\mathcal{S})$ est défini par un STS $< L_{\mathcal{P}}, l0_{\mathcal{P}}, V_{\mathcal{S}} \cup V_{\phi(\mathcal{S})} \cup \{side\}, V0_{\mathcal{S}} \cup V0_{\phi(\mathcal{S})} \cup \{side := ""\}, I_{\mathcal{S}} \cup I_{\phi(\mathcal{S})}, \Lambda_{\Delta(\mathcal{S})} \cup \Lambda_{REF}, \rightarrow_{\mathcal{P}}>$ tel que $L_{\mathcal{P}}, l0_{\mathcal{P}}$ et $\rightarrow_{\mathcal{P}}$ sont construits par les règles d'inférence suivantes :*

$$\textit{(Env to IUT)} : \quad l_1 \xrightarrow{?a(p),\varphi,\varrho}_{\Delta(\mathcal{S})} l_2, l_{1'} \xrightarrow{!a(p'),\varphi',\varrho'}_{REF} l_{2'}, \; l'_1 = \phi(l_1), \varphi' = \phi(\varphi), \varrho' = \phi(\varrho)$$

$$\vdash$$

$$(l_1 l_{1'}) \xrightarrow{?a(p),\varphi,[\varrho(\{side:=Env\})]}_{\mathcal{P}} (l_2 l_{1'} a \varphi \varrho) \xrightarrow{!a(p'),[p'==p\wedge\varphi'],[\varphi'(\{side:=IUT\})]}_{\mathcal{P}} (l_2 l_{2'}),$$

$$\textit{(IUT to Env)} : \quad l_1 \xrightarrow{!a(p),\varphi,\varrho}_{\Delta(\mathcal{S})} l_2, l_{1'} \xrightarrow{?a(p'),\varphi',\varrho'}_{REF} l_{2'}, \; l'_1 = \phi(l_1), \varphi' = \phi(\varphi), \varrho' = \phi(\varrho)$$

$$\vdash$$

$$(l_1 l_{1'}) \xrightarrow{?a(p'),\varphi',[\varrho'(\{side:=IUT\})]}_{\mathcal{P}} (l_1 l_{2'} a \varphi) \xrightarrow{!a(p),[p==p'\wedge\varphi],[\varrho(\{side:=Env\})]}_{\mathcal{P}} (l_2 l_{2'}),$$

$$\textit{(to Fail)} : \quad l_{1'} \xrightarrow{b(p),\varphi,\varrho}_{REF} Fail, l_1 \in L_{\mathcal{S}}, l'_1 = \phi(l_1)$$

$$\vdash$$

$$(l_1 l_{1'}) \xrightarrow{b(p),\varphi,[\varrho(\{side:=IUT\})]}_{\mathcal{P}} Fail$$

Intuitivement, la première règle (Env to IUT) combine une transition de la spécification avec une transition de son STS réflexion étiquetée par les mêmes actions et gardes pour exprimer que si une action est reçue depuis le côté client alors celle-ci est envoyée vers l'implantation. Les deux étapes (recevoir et transférer une action) sont séparées par un nouvel état symbolique unique ($l_2 l_{1'} a \varphi \varrho$). Les transitions étiquetées par δ modélisant la quiescence sont aussi combinées : ainsi si la quiescence est observée à partir de l'implantation, la quiescence sera aussi observée par l'environnement extérieur (et vice-versa). De façon similaire, la seconde règle (IUT to Env) combine une transition de la spécification et une transition de son STS réflexion pour exprimer que si une action est reçue depuis

l'implantation, alors celle-ci est propagée vers l'environnement extérieur. La dernière règle (to Fail) complète le STS résultant avec les transitions du STS réflexion menant à l'état Fail. Pour chaque règle, les transitions sont également identifiées par la variable interne *side*.

Le proxy-testeur obtenu depuis la précédente spécification de la Figure 3.53 et de son STS réflexion (Figure 3.55) est donné en Figure 3.56. Par soucis de lisibilité, la variable *side* est remplacée par des transitions pleines ou pointillées : les transitions pleines représentent des interactions avec le côté client ($side := Env$), celles en pointillé les interactions avec l'implantation ($side := IUT$). La Figure 3.56 illustre de façon claire que le comportement initial de la spécification est toujours présent et que le comportement incorrect donné par le STS réflexion l'est également.

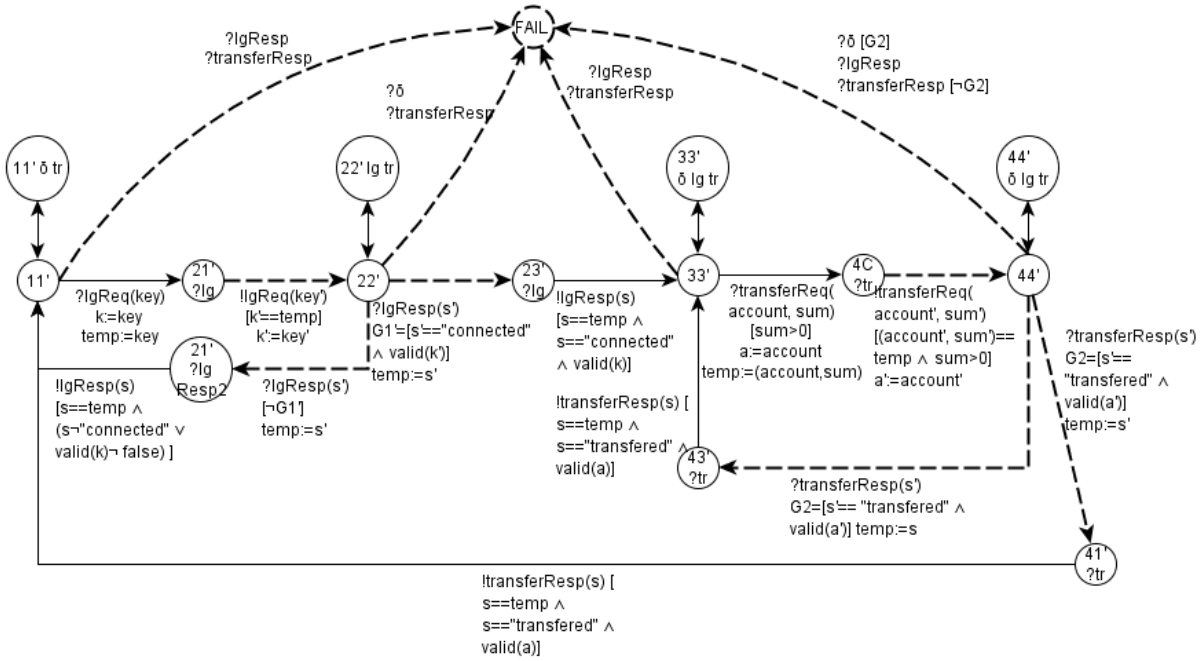


FIGURE 3.56 – Un proxy-testeur

Les transitions représentant les interactions avec le côté client sont donc séparées des autres transitions grâce à la variable interne *side*. De ce fait, nous pouvons aussi séparer les exécutions et traces du proxy-testeur.

Soit le STS $\mathcal{P} =< L_{\mathcal{P}}, l0_{\mathcal{P}}, V_{\mathcal{P}}, V0_{\mathcal{P}}, I_{\mathcal{P}}, \Lambda_{\mathcal{P}}, \rightarrow_{\mathcal{P}} >$ et son LTS sémantique $||\mathcal{P}|| = P =< Q_P, q0_P, \sum_P, \rightarrow_P >$. Nous définissons la fonction $Side : Q_P \rightarrow D_{V_{\mathcal{P}}}$ qui retourne la valeur de la variable *side* depuis un état. (les variables internes des STSs sont localisées dans les états des LTS sémantiques).

Soit $RUN(\mathcal{P})$ l'ensemble des exécutions de $\mathcal{P}$. Nous définissons $RUN^E(\mathcal{P})$ comme l'ensemble des projections $\{ proj_{q_i\alpha_i q_{i+1} | Side(q_{i+1})=E}(q_0\alpha_0 q_1 ... q_n\alpha_n q_{n+1} \in RUN(\mathcal{P}))\}$ qui dénote l'ensemble des exécutions partielles relatives à l'environnement (coté) E. Il s'ensuit que $Traces^E(\mathcal{P})$ est l'ensemble de traces partielles des exécutions partielles dans $RUN^E(\mathcal{P})$.

Pour un proxy-testeur $\mathcal{P}(\mathcal{S})$, nous pouvons écrire $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$ afin de représenter les traces non conformes, extraites du proxy-testeur, obtenues à partir de l'implantation.

Par exemple, dans le proxy-testeur de la Figure 3.56, !lgReq("a incorrect key"). $?\delta$ appartient à $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$.

Grâce à ces notations, nous pouvons aussi écrire quelques propriétés sur les traces. En particulier, nous pouvons écrire que les traces du STS réflexion sont égales aux traces produites par les transitions du proxy-testeur étiquetées par l'affectation $side := IUT$:

Lemme 3.2.1 *Soit $\mathcal{S}$ un STS et $REF(\mathcal{S})$, $\mathcal{P}(\mathcal{S})$ son STS réflexion et son proxy-testeur respectivement. Nous avons $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) = Traces_{Fail}(REF(\mathcal{S}))$.*

Preuve: Pour un STS $\mathcal{S}$, nous notons $l_1 \Rightarrow l_2$ un chemin de l_1 vers l_2.

Soit $path = l'_0 \xrightarrow{a_0,\varphi'_0,\varrho'_0} l'_1 ... l'_{n-1} \xrightarrow{a_{n-1},\varphi'_{n-1},\varrho'_{n-1}} Fail \in (\rightarrow_{REF(\mathcal{S})})^n$.

(1)$\exists! path_2 = l_0 l'_0 \Rightarrow l_1 l'_1 ... l_{n-1} l'_{n-1} \Rightarrow Fail \in (\rightarrow_{\mathcal{P}(\mathcal{S})})^{2n-1}$.

Preuve de (1)

Nous rappelons que ϕ est une fonction, ainsi $l'_1 = \phi(l_1)$ est unique.

Nous avons pour $0 \leq i \leq n-2$:

- (A1) si $a_i \in \Lambda^I_{REF}$, nous avons un chemin unique $(l_i l'_i) \xrightarrow{?a_i(p'_i),\varphi'_i,\varrho'_i(\{side:=(IUT)\})} (l_i l'_{i+1} a_i \varphi_i \varrho_i) \xrightarrow{!a_i(p_i),[p_i==p'_i \wedge \varphi_i],\varrho_i(\{side:=(Env)\})} (l_{i+1} li+1') \in (\rightarrow_{\mathcal{P}(\mathcal{S})})^2$ (règle IUT to Env),
- (B1) si $a_i \in \Lambda^O_{REF}$, nous avons un chemin unique $(l_i l'_i) \xrightarrow{?a_i(p_i),\varphi_i,\varrho_i(\{side:=(Env)\})} (l_{i+1} l'_i a_i \varphi_i \varrho_i) \xrightarrow{!a_i(p'_i),[p'_i==p_i \wedge \varphi'_i],\varrho'_i(\{side:=(IUT\})} (l_{i+1} l'_{i+1}) \in (\rightarrow_{\mathcal{P}(\mathcal{S})})^2$ (règle Env to IUT),
- (C1) pour $l'_{n-1} \xrightarrow{a_{n-1},\varphi'_{n-1},\varrho'_{n-1}} Fail$, nous avons également un chemin unique l_{n-1} $l'_{n-1} \xrightarrow{a_{n-1},\varphi'_{n-1},\varrho'_{n-1}} Fail$ (règle to Fail).

Si l'on applique (A1), (B1) ou (C1) pour $0 \leq i \leq n-2$, nous avons un chemin unique.

Soit maintenant $R = (l'_0, V'_0) a_0(p'_0), \theta_0 (l'_1, V'_1) ... (l'_{n-1}, V'_{n-1}) a_{n-1}(p'_{n-1}), \theta_{n-1} Fail \in RUN_{Fail}(\phi(REF(\mathcal{S})))$ une exécution extraite de $path$.

(2) $\exists!$ run $R_2 = q_0 \alpha_0 q_{01} \alpha_{01} q_1 ... q_{n-1} \alpha_{n-1} Fail \in RUN^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$ dérivée de R.

Preuve de (2) :

Considérons a_0. Une exécution de R est une représentation d'un chemin du STS sémantique de $\phi(REF(\mathcal{S}))$. Selon (1) il existe un unique chemin dérivé de $path$, noté $path2$. Si nous appliquons les règles de transformation en STS sémantique (Section 1.3.1), nous obtenons :

- (A2) si $a_0 \in \Lambda^I_{REF}$, nous avons $q_0 \alpha_0 q_{01} \alpha_{01} q_1 = (l_0 l'_0, V_0 \cup V'_0 \cup \{side := IUT\})$ $(?a_0(p'_0), \theta'_0)$ $((l_0 l'_1, V_0 \cup V'_1 \cup \{side := Env\}))$ $(!a_0(\phi^{-1}(p'_0)), \phi^{-1}(\theta'_0))$ $(l_1 l'_1, \phi^{-1}(V'_1) \cup V'_1 \cup \{side := \{IUT||Env\}\})$, si nous appliquons les règles de transformation en STS sémantique sur $(l_i l'_i) \xrightarrow{?a_i(p'_i),\varphi'_i,\varrho'_i(\{side:=(IUT)\})} (l_i l'_{i+1} a_i \varphi_i \varrho_i) \xrightarrow{!a_i(p_i),[p_i==p'_i \wedge \varphi_i],\varrho_i(\{side:=(Env)\})} (l_{i+1} li+1')$ avec $i = 0$ (A1),
- (B2) si $a_0 \in \Lambda^O_{REF}$, nous avons $q_0 \alpha_0 q_{01} \alpha_{01} q_1 = (l_0 l'_0, V_0 \cup V'_0 \cup \{side := Env\})$ $(?a_0(\phi^{-1}(p'_0)), \theta_0)$ $(l_1 l'_0, \phi^{-1}(V'_1) \cup V'_0 \cup \{side := IUT\})$ $(!a_0(p'_0), \phi(\theta_0))$ $(l_1 l'_1, \phi^{-1}(V'_1) \cup V'_1 \cup \{side := \{IUT||Env\}\})$ si nous appliquons les règles de transformation en STS sémantique sur $(l_i l'_i) \xrightarrow{?a_i(p_i),\varphi_i,\varrho_i(\{side:=(Env)\})} (l_{i+1} l'_i a_i \varphi_i \varrho_i) \xrightarrow{!a_i(p'_i),[p'_i==p_i \wedge \varphi'_i],\varrho'_i(\{side:=(IUT\})} (l_{i+1} l'_{i+1})$ avec $i = 0$ (B1).

En appliquant récursivement (A2) et (B2) nous avons pour $0 \leq i \leq n-2$:

- (A) si $a_i \in \Lambda^I_{REF}$, nous avons $q_i\alpha_i q_{ii+1}\alpha_{ii+1}q_{i+1} = (l_il'_i, V_i \cup V'_i \cup \{side := IUT\})$ $(?a_i(p'_i), \theta'_i)$ $((l_il'_{i+1}, V_i \cup V'_i \cup \{side := Env\}))$ $(!a_i(\phi^{-1}(p'_i)), \phi^{-1}(\theta'_i))$ $(l_{i+1}l'_{i+1}, \phi^{-1}(V'_{i+1} \cup V'_{i+1}) \cup \{side := \{IUT||Env\}\})$,
- (B) si $a_i \in \Lambda^O_{REF}$, nous avons $q_i\alpha_i q_{ii+1}\alpha_{ii+1}q_{i+1} = (l_il'_i, V_i \cup V'_i \cup \{side := Env\})$ $(?a_i(\phi^{-1}(p'_i)), \theta_i)$ $(l_{i+1}l'_i, \phi^{-1}(V'_{i+1}) \cup V'_i \cup \{side := IUT\})$ $(!a_i(p'_i), \phi(\theta_i))$ $(l_{i+1}l'_{i+1}, \phi^{-1}(V'_{i+1}) \cup V'_{i+1} \cup \{side := \{IUT||Env\}\})$,
- (C) $q_{n-1}\alpha_{n-1}Fail = (l_{n-1}l'_{n-1}, \phi^{-1}(V'_{n-1}) \cup V'_{n-1} \cup \{side := IUT\})$ $a_{n-1}(p'_{n-1}), \theta'_{n-1}$ $Fail$ (c1).

La trace $T \in Traces_{Fail}(\phi(REF(\mathcal{S})))$ de $R = (l'_0, V'_0)a_0(p'_0), \theta'_0(l'_1, V'_1)...(l'_{n-1}, V'_{n-1})$ $a_{n-1}(p'_{n-1}), \theta_{n-1}Fail$ correspond à $T = (a_0(p'_0), \theta'_0)(a_1(p'_1), \theta'_1)...(a_{n-1}(p'_{n-1}), \theta'_{n-1})$.

$proj_{\{\Sigma, Side(Q)=IUT\}}(R_2)$ est la trace $T_2 \in Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$ de R_2. Selon (A), (B) et (C), $T_2 = (a_0(p'_0), \theta'_0)(a_1(p'_1), \theta'_1)...(a_{n-1}(p'_{n-1}), \theta'_{n-1})$ and $T_2 = T$. Nous déduisons que $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) = Traces_{Fail}(\phi(REF(\mathcal{S})))$.

Comme $\phi(REF(\mathcal{S}))$ correspond à $REF(\mathcal{S})$ avec états symboliques et variables renommés, Nous écrivons également $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) = Traces_{Fail}(REF(\mathcal{S}))$. ∎

3.2.3 Conformité passive

Afin de raisonner sur la conformité, l'implantation sous test est supposée se comporter comme sa spécification et est représentée par un LTS I. $\Delta(I)$ correspond à son LTS suspension. Comme nous l'avons vu dans ce mémoire à plusieurs reprises, les méthode de test actives définissent habituellement le degré de confiance entre l'implantation I et la spécification $\mathcal{S}$ au moyen de relations de test. *ioco* est la plus connue. Dans [RMJ05], *ioco* est définie par :

Définition 3.2.4 *Soit I une implantation décrite par un LTS, et $\mathcal{S}$ un STS. I est ioco-conforme à $\mathcal{S}$, noté I ioco $\mathcal{S}$ ssi $Traces(\Delta(\mathcal{S})).(\sum^O \cup \{!\delta\}) \cap Traces(\Delta(I)) \subseteq Traces(\Delta(\mathcal{S}))$.*

A partir de cette définition, on peut déduire que :

$$I \ ioco \ \mathcal{S} \Leftrightarrow Traces(\Delta(I)) \cap NC\_Traces(\Delta(\mathcal{S})) = \emptyset$$

avec $NC\_Traces = Traces(\Delta(\mathcal{S})).\sum^O \cup \{!\delta\} \backslash Traces(\Delta(\mathcal{S}))$, les traces non conformes d'une spécification (le complément de $Traces$).

La définition du STS réflexion (Définition 3.2.2) montre que les comportements incorrects de la spécification sont reconnus dans les états Fail du STS réflexion. Ainsi, $NC\_Traces(\mathcal{S})$ est aussi équivalent à $Traces_{Fail}(REF(\mathcal{S}))$. De plus, nous avons montré précédemment que $Traces_{Fail}(REF(\mathcal{S})) = Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$.

De ce fait, nous pouvons écrire :

Proposition 3.2.1

$$I \ ioco \ \mathcal{S} \Leftrightarrow Traces(\Delta(I)) \cap Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) = \emptyset$$

De ce fait, les traces du proxy-testeur sont explicitement données dans la relation de test. Nous pouvons aussi dire que I n'est pas ioco-conforme à sa spécification lorsqu'une trace de l'implantation appartient à l'ensemble des traces partielles (avec side :=IUT) du

proxy-testeur menant à Fail. Cependant, nous pouvons aller plus loin dans la réécriture de ioco en prenant en compte l'exécution passive du proxy-testeur. Pour cela, nous devons aussi raisonner sur la composition parallèle de l'environnement avec le proxy-testeur et l'implantation.

$P =< Q_P, q0_P, \sum_P, \rightarrow_P >$ est le LTS sémantique d'un proxy-testeur $\mathcal{P}(\mathcal{S})$. Nous supposons que l'environnement client peut être modélisé par le LTS $Env =< Q_{Env}, q0_{Env}, \sum_{Env} \subseteq \sum_P, \rightarrow_{Env} >$, et $I =< Q_I, q0_I, \sum_I \subseteq \sum_P, \rightarrow_I >$ est l'implantation sous test. Le test passif de I est modélisé par la composition parallèle $||(Env, P, I) =< Q_{Env} \times Q_P \times Q_I, q0_{Env} \times q0_P \times q0_I, \sum_{Env} \subseteq \sum_P, \rightarrow_{||(Env,P,I)} >$ où la relation $\rightarrow_{||(Env,P,I)}$ est définie par les règles suivantes. Afin de faciliter la lisibilité, nous notons une transition de LTS $q_1 \xrightarrow[E]{?a} q_2$ lorsque $Side(q_1) = E$ (une variable *side* a la valeur E dans q_1).

(Env à IUT) : $$\frac{q_1 \xrightarrow{!a}_{\Delta(Env)} q_2, q_2'' \xrightarrow{?a}_{\Delta(I)} q_3'', q_1' \xrightarrow[Env\_P]{?a} q_2' \xrightarrow[IUT\_P]{!a} q_3'}{q_1 q_1' q_2'' \xrightarrow[||(Env,P,I)]{?a} q_2 q_2' q_2'' \xrightarrow[||(Env,P,I)]{!a} q_2 q_3' q_3''}$$

(IUT à Env) : $$\frac{q_2 \xrightarrow{?a}_{\Delta(Env)} q_3, q_1'' \xrightarrow{!a}_{\Delta(I)} q_2'', q_1 \xrightarrow[IUT\_P]{?a} q_2' \xrightarrow[Env\_P]{!a} q_3', q_3' \neq Fail}{q_2 q_1' q_1'' \xrightarrow[||(Env,P,I)]{?a} q_2 q_2' q_2'' \xrightarrow[||(Env,P,I)]{!a} q_3 q_3' q_2''}$$

(IUT à Fail) : $$\frac{q_2 \xrightarrow{?\delta}_{\Delta(Env)} q_3, q_1'' \xrightarrow{!a}_{\Delta(I)} q_2'', q_1' \xrightarrow[IUT\_P]{?a} Fail}{q_2 q_1' q_1'' \xrightarrow[||(Env,P,I)]{?a} Fail}$$

La déduction immédiate de la définition de $\rightarrow_{||(Env,P,I)}$ est que : $Traces^{IUT}_{Fail}(||(Env, P, I)) = Traces(\Delta(I)) \cap Traces^{IUT}_{Fail}(P) = Traces(\Delta(I)) \cap Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$. En d'autres termes, les traces non conformes de $\Delta(I)$ peuvent aussi être trouvées dans $Traces^{IUT}_{Fail}(||(Env, P, I))$. Selon la proposition précédente, nous pouvons finalement déduire :

Proposition 3.2.2

$$I \ ioco \ \mathcal{S} \Leftrightarrow Traces(\Delta(I)) \cap Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) = \emptyset$$
$$\Leftrightarrow Traces^{IUT}_{Fail}(||(Env, P, I)) = \emptyset$$

En d'autres termes, cette relation signifie que l'implantation I est non conforme à sa spécification si une trace de $||(Env, P, I)$ mène à l'un de ses états $Fail$.

L'algorithme du proxy-testeur peut maintenant être déduit facilement. Au préalable, nous supposons que le trafic client est routé vers le proxy-testeur. Nous supposons également que ce dernier a une configuration d'instance équivalente à celle de l'implantation. Par exemple, si l'implantation est de type multi-instance (une instance par client), le mode du proxy-testeur doit être identique.

Tout comme dans la relation ioco, l'environnement et l'implantation sont supposés se comporter comme des STS suspensions. Le proxy-testeur manipule un ensemble d'états instanciés (valués) chacun exprimant l'un de ses états courants. Un état unique n'est pas suffisant car l'implantation et le proxy-testeur peuvent être indéterministes et peuvent mener à différentes exécutions. Pendant son exécution, le proxy-testeur reçoit des actions valuées depuis l'environnement et l'implantation. Tout en recevant ces actions valuées, pour chaque état instancié, il couvre ses transitions jusqu'à arriver à un état Fail. Dans

ce cas, l'implantation n'est pas ioco-conforme à la spécification. L'algorithme 4 détaille son fonctionnement.

Le proxy-testeur commence à son état instancié initial $(l0_{\mathcal{P}(\mathcal{S})}, V0_{\mathcal{P}(\mathcal{S})})$ et possède un ensemble IL d'états instanciés. Après avoir reçu $e(p), \theta$ avec θ une valuation sur p (ligne 2), pour chaque état instancié dans IL, il vérifie qu'une prochaine transition peut être franchie (ligne 5) : celle-ci doit avoir le même état de départ L, la même action $e(p)$ et sa garde doit être satisfaite sur W et θ. Si cette transition mène à Fail alors l'algorithme retourne Fail (lignes 6-7). Autrement, $e(p)$ est renvoyé vers le coté IUT ou Env avec la prochaine transition du proxy-testeur t_2 selon la valeur de *side* dans ϱ_2, noté $side(t_2)$ pour simplifier dans l'algorithme (ligne 9). Le nouvel état instancié atteint est construit (ligne 11 ou 14 selon $side(t_2)$). Celui-ci est composé du dernier état symbolique atteint et d'une nouvelle valuation dérivée de θ, ϱ et ϱ_2. Une fois que chaque état instancié est couvert, le proxy-testeur attend l'évènement suivant.

Algorithme 4: Algorithme du proxy-testeur

input : Un proxy-testeur $\mathcal{P}(\mathcal{S})$
output : Faute détectée

`// initialiser le proxy-testeur avec son état instancié initial`
1 $IL := \{(l0_{\mathcal{P}(\mathcal{S})}, V0_{\mathcal{P}(\mathcal{S})})\}$;
2 **while** *Event*$(e(p), \theta)$ **do**
3 $IL' = \emptyset$;
`// vérification pour chaque état instancié`
4 **foreach** $(L, W) \in IL$ **do**
5 **foreach** $t = L \xrightarrow{e(p),\varphi,\varrho} l_{next} \in \rightarrow_{\mathcal{P}(\mathcal{S})}$ *tel que* $\theta \cup W \models \varphi$ **do**
6 **if** $l_{next} == Fail$ **then**
7 returner FAIL;
8 **else**
`// action valuée transférée vers le bon coté`
9 **if** $side(t_2 = l_{next} \xrightarrow{!e(p_2),\varphi_2,\varrho_2}_{\mathcal{P}(\mathcal{S})} l_{next2}) == IUT$ **then**
10 Exécuter($t_2 = l_{next} \xrightarrow{(!e(p_2),phi(\theta)),\varphi_2,\varrho_2}_{\mathcal{P}(\mathcal{S})} l_{next2}$) ; `// envoi de` $(!e(p'), \phi(\theta))$ `vers IUT`
11 $Q_{next} := (l_{next2}, \varrho(\theta) \cup \varrho_2(\phi(\theta)))$;
12 **else**
13 Exécuter($t_2 = l_{next} \xrightarrow{(!e(p_2),phi^{-1}(\theta)),\varphi_2,\varrho_2}_{\mathcal{P}(\mathcal{S})} l_{next2}$) ; `// envoi de` $(!e(p), \phi^{-1}(\theta))$ `vers Env`
14 $Q_{next} := (l_{next2}, \varrho(\theta) \cup \varrho_2(\phi^{-1}(\theta)))$;
15 $IL' := IL' \cup \{Q_{next}\}$;
16 $IL := IL'$;

Cet algorithme couvre passivement les transitions du proxy-testeur tout en recevant des actions valuées depuis l'environnement ou depuis l'implantation, jusqu'à ce qu'il atteigne un état Fail. Dans ce cas, le proxy-testeur a construit, depuis l'état initial, une

exécution (et donc une trace) composée d'états et d'actions valuées menant à Fail. Ainsi, nous pouvons affirmer que :

Proposition 3.2.3 *L'algorithme a retourné Fail* $\Rightarrow Traces_{Fail}^{IUT}(||(Env, P, I)) \neq \emptyset \Rightarrow \neg(I \ ioco \ \mathcal{S})$.

3.2.4 Expérimentation et discussion

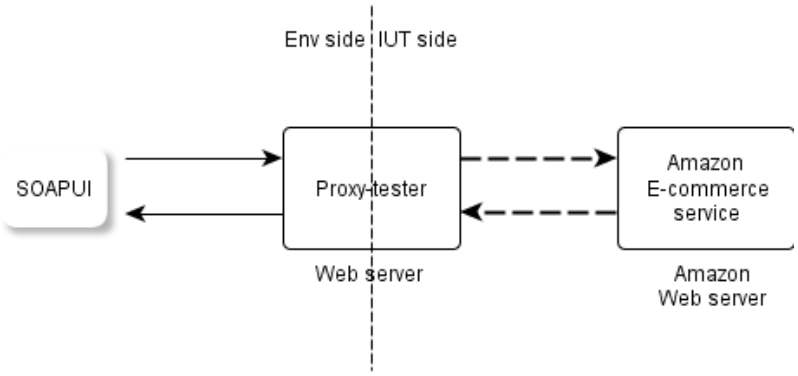


FIGURE 3.57 – Architecture d'expérimentation

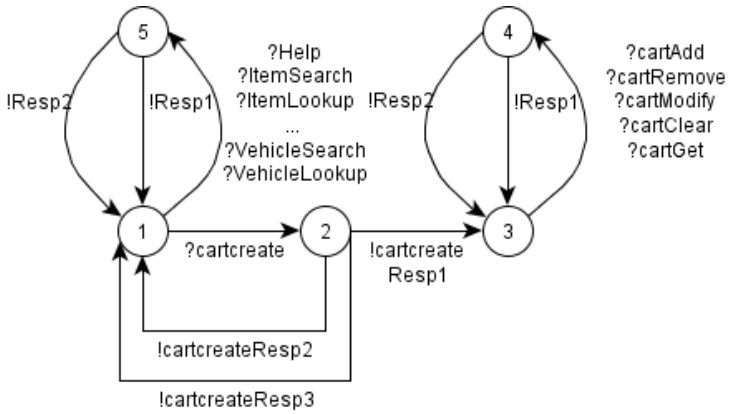


FIGURE 3.58 – Une spécification partielle du service Web Amazon E-commerce

Nous avons effectué une expérimentation préliminaire de notre méthode sur le service Web d'E-commerce d'Amazon [Ama10] dont une spécification partielle est illustrée en Figure 3.58.

Nous avons implanté un outil académique qui génère un proxy-testeur STS à partir d'une spécification décrite par un STS. Ensuite, nous avons codé le fonctionnement du proxy-testeur à partir de l'Algorithme 4, comme un service Web et l'avons déployé sur un serveur Tomcat/Axis. Pour simplifier ce cas d'étude, nous avons seulement considéré

	Nn d'états	Nb de Transitions	Nb de Transitions vers Fail
Spécification	24	67	0
testeur Canonique	25	202	133
Proxy-testeur	49	269	133

FIGURE 3.59 – Statistiques sur le test du service Web d'Amazon

le type de variable String et avons utilisé le solveur [KGG+09], qui prend comme entrée des variables de type String, pour définir des gardes et des affectations et bien sûr pour vérifier la satisfiabilité de gardes. L'architecture complète est décrite en Figure 3.57.

Comme le test passif peut être long, (les requêtes clientes provoquant des bugs peuvent ne pas apparaître de suite), nous avons aussi préparé manuellement des séquences d'actions pour simuler des requêtes clientes provoquant des bugs et nous les avons exécutées avec l'outil SOAPUI [Evi11], qui est un outil de test unitaire pour les services Web. Ces séquences ont été construites pour appeler le proxy-testeur à la place du service Web d'Amazon. Pour chaque séquence exécutée, le proxy-testeur a retourné Fail comme prévu.

Nous avons aussi effectué cette expérimentation afin de vérifier la faisabilité de notre approche. La Figure 3.59 donne des statistiques sur le nombre d'états et de transitions obtenus à partir de la spécification du service Web d'Amazon. Bien que le nombre de transitions du proxy-testeur augmente sensiblement, celui-ci est borné car il correspond à une combinaison de la spécification et de son testeur canonique qui sont finis également. En fait, le STS du proxy-testeur peut être calculé rapidement (quelques secondes à minutes au plus).

Cette expérimentation a montré que le concept de proxy-testeur offre une alternative très intéressante aux outils classiques basés sur des sniffeurs car un proxy-testeur peut détecter la non conformité de l'implantation sans avoir besoin de privilèges particuliers sur les serveurs d'Amazon et sans avoir besoin de placer un sniffeur sur chaque client.

3.2.5 Perspectives

Les proxy-testeurs peuvent offrir bien plus de possibilités que celle présentée précédemment et ouvre sur de nombreuses perspectives. Notamment, l'un des intérêts majeur des proxy-testeurs concerne leur capacité à séparer explicitement les actions qui proviennent d'un environnement extérieur (client) de celles qui proviennent d'une implantation sous test. Ci dessous, nous proposons quelques unes de ces perspectives qui seront reprises dans des travaux futurs :

- *Le test de compositions dans des environnements restreints :* de tels systèmes pourraient être passivement testés au moyen d'un ensemble de proxy-testeurs. Plusieurs solutions sont envisageables.
  La première, plutôt classique serait de générer un proxy-testeur par composant. Par exemple, si l'on reprend la composition de la Figure 3.48, l'architecture serait celle de la Figure 3.60. Si nous supposons que nous avons une seule spécification pour modéliser la composition entière, il faut alors extraire une sous-spécification par composant et générer un proxy-testeur à chaque fois. La problématique soulevée concerne la synchronisation des proxy-testeurs pour suivre le comportement entier de la composition sous test, mais les travaux sur le test de systèmes distribués

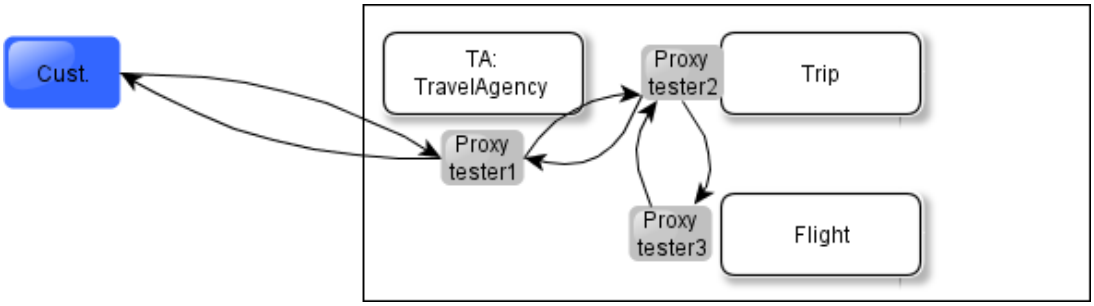


FIGURE 3.60 – Utilisation de proxy-testeurs dans une composition

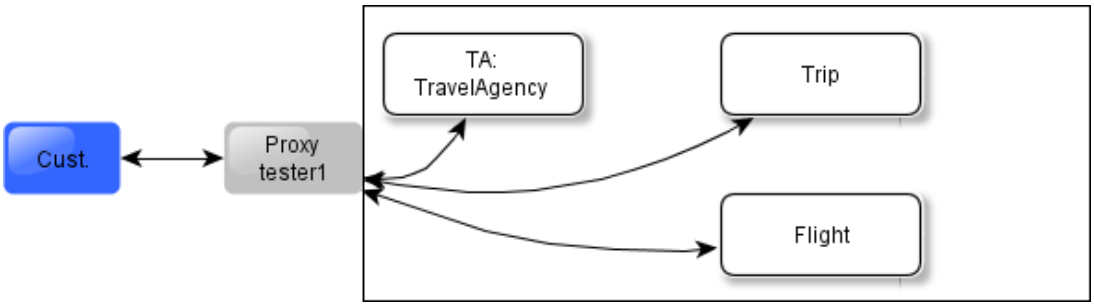


FIGURE 3.61 – Utilisation d'un seul proxy-testeur dans une composition

devraient apporter des solutions approchées. Il est aussi possible de ne pas synchroniser les proxy-testeurs : dans ce cas, ceux-ci retourneraient des traces partielles qu'il faudrait recomposer. L'approche de la Section 3.1 devrait apporter des solutions.
Une deuxième piste serait de ne considérer qu'un proxy-testeur pour toute la composition capable de recevoir tous les flux d'actions depuis le coté client et depuis chaque composant. Un tel proxy-testeur serait une partie intégrante de la composition et prévu à l'avance. Il devrait aussi être capable de re-diriger une action reçue vers la bonne destination. Ceci donnerait l'architecture de la Figure 3.61. Dans ce cas, il est inutile d'éclater la spécification en plusieurs sous-spécifications. De même, le proxy-testeur, recevant toutes les actions depuis la composition, peut produire des traces complètes. De ce fait, la relation ioco semble toujours pouvoir être utilisée. Mais bien sûr, la définition du proxy-testeur devrait alors être fortement revue et prendre en compte l'implantation, le coté client mais aussi chaque composant,

– *Combinaison des proxy-testeurs avec des méthodes de test orientées invariants :* nous avons montré qu'il est possible d'extraire les traces de l'implantation depuis le proxy-testeur. Ainsi, pour une spécification $\mathcal{S}$ et son proxy-testeur $\mathcal{P}(\mathcal{S})$, nous avons montré que $Traces^{IUT}(\mathcal{P}(\mathcal{S})) = Traces(\phi(REF(\mathcal{S})))$. De ce fait, retrouver les traces de l'implantation est assez simple. Par conséquent, les méthodes passives basées sur des invariants, que nous avons citées précédemment en Section 3.2.1, peuvent toujours être utilisées pour vérifier si les traces obtenues satisfont des invariants donnés. La combinaison de ces méthodes avec des proxy-testeurs permettrait donc de tester la satisfaction d'invariants sur un système installé dans un environnement partiellement ouvert. Prenons l'exemple d'un service Web, déployé dans un Cloud,

dont les traces auraient été collectées par un proxy-testeur. L'utilisation de l'outil, décrit dans [CBML09], permet ensuite de vérifier la satisfaction de règles de sécurité,

- *Test de sécurité et protection :* un grand avantage des proxy-testeurs est de séparer les actions provenant de l'environnement extérieur (client) et celles produites par l'implantation. Par exemple, la Figure 3.56 illustre parfaitement cette séparation : les transitions pleines ($side := Env$) correspondent à des interactions avec le coté client, celles en pointillés ($side := IUT$)à des interactions avec l'implantation.
  D'une part, une première perspective pourrait être donc de filtrer les actions reçues de l'environnement extérieur. En terme de sécurité, des travaux futurs pourraient permettre aux proxy-testeurs de reconnaître des attaques et de ne pas les retransmettre vers l'implantation. Les proxy-testeurs pourraient aussi bloquer toute action non décrite dans la spécification. Dans ce cas, les proxy-testeurs joueraient le rôle de parefeu applicatif et notamment de WAF (Web Application Firewall) sans qu'il n'y ait de règles à donner. Ils protégeraient au plus prêt des systèmes car ils seraient construits à partir de spécifications.
  D'autre part, un proxy-testeur pourrait aussi vérifier à partir des traces reçues de l'implantation, si celles-ci respectent des règles de sécurité. A la place d'utiliser des invariants, une solution serait de formaliser des règles de sécurité avec des STSs puis de combiner ces STSs avec le proxy-testeur pour modéliser et séparer les comportements sécurisés des autres.
  Ainsi, pour les deux cas, il faudrait définir deux produits synchronisés : l'un en synchronisant des règles de protection sous forme de STSs avec les transitions du proxy-testeur ayant l'affectation ($side := Env$), l'autre en combinant des règles de sécurité avec les transitions du proxy-testeur ayant l'affectation ($side := IUT$). Le proxy-testeur résultat pourrait protéger et tester la sécurité d'une application en même temps,
- *Amélioration de la qualité d'un système :* un proxy-testeur peut être aussi vu comme une sur-couche logicielle englobant une application ou un système. De ce fait, il pourrait être utilisé afin de modifier le comportement d'une implantation automatiquement. Par exemple, si cette dernière se fige ("plante"), le proxy-testeur pourrait être augmenté pour que lorsque la quiescence est observée, il retourne tout de même une réponse au côté client. Il est manifeste que ce simple ajout améliore l'observabilité du couple proxy-testeur/implantation et augmente donc sa qualité. Il serait aussi possible d'augmenter le proxy-testeur avec des règles afin de détecter certaines erreurs et de les corriger à la volée. De futurs travaux pourraient ainsi permettre de définir des règles d'amélioration à synchroniser avec un proxy-testeur pour améliorer plusieurs aspects comme sa testabilité (observabilité, contrôlabilité, etc.). Une amélioration dynamique d'un système par un proxy-testeur pourrait être aussi envisagée.

3.3 Les applications dynamiques orientées service

Cette partie traite des perspectives de travaux relatives aux compositions dynamiques orientées service. Ces perspectives représentent une suite logiques à nos travaux précédents qui s'appuient sur des composants dits "stateless" (sans état interne évolutif) et des composants dit "stateful" (qui possèdent un état interne) et sur des compositions sta-

tiques. Ces travaux ont été proposés et se feront en collaboration avec une filiale d'Orange Business (Almerys) qui soulève des problématiques liées à la composition dynamique de plusieurs versions d'une même application, par exemple dans le cadre de son application de mise en ligne des carnets de santé [(Al11].

Dans cette partie, nous considérons, de façon non restrictive, qu'une application orientée service représente une composition dans laquelle des services peuvent être choisis et invoqués dynamiquement, i.e. à l'exécution. Ce type d'application correspond à une tendance grandissante, dérivée du software engineering à base de composants, qui est notamment de plus en plus considérée en relation avec des Clouds, i.e. des environnements virtualisés. Comme précédemment, nous ne nous focaliserons pas sur les Clouds mais sur des environnements partiellement ouverts dans lesquels des applications ou systèmes sont déployés et à travers desquels l'accès est restreint.

Ces compositions dynamiques, déployées dans des environnements partiellement ouverts, amènent à des problématiques et perspectives relatives à la dynamicité et aux restrictions sur le contrôle et l'observation des applications, dues aux environnements. Ces problématiques vont donc être étudiées dans cette Section pour donner lieu à des propositions de travaux sur : la modélisation d'applications orientées service dynamiques, la composition dynamique et la cohabitation de plusieurs versions d'une même application pour que toutes ces versions soient appelées par un ensemble hétérogène de clients, et le test de ces applications.

Dans la suite, nous décrivons le contexte des compositions dynamiques, l'état de l'art succins, les problèmes soulevés et donnerons des pistes de travaux futurs.

3.3.1 Contexte

Le software engineering à base de composants est un paradigme connu qui est maintenant entré dans la plupart des sociétés de développement informatique. Ce paradigme offre de nombreux avantages tels que la réalisation d'applications avec des composants hétérogènes existants, la réutilisation d'applications accompagnée de réduction de coût, ou la modélisation claire de spécifications notamment sur les entrées nécessaires par d'autres composants. Ce paradigme a aussi donné lieu à de nouvelles tendances et concepts comme le Cloud computing.

Il est manifeste que le software engineering à base de composants a aussi été étudié en relation avec des approches de test afin de vérifier de nombreux aspects et propriétés comme la sécurité, la robustesse ou la conformité de systèmes divers tels que les applications orientées service, les systèmes distribués ou les applications orientées objet. Nous ne reviendrons pas sur le test de composants et de compositions statiques dans des environnements ouverts car nous avons présenté ce domaine précédemment, dans ce chapitre.

L'évolution actuelle du software engineering à base de composants tend premièrement vers la dynamicité des composants. La composition dynamique a le potentiel de construire des applications flexibles et adaptables en sélectionnant et en combinant des composants suivant les requêtes clientes et le contexte. Elle permet d'augmenter les fonctionnalités d'applications qui n'ont pas été envisagées au moment de la modélisation. Elle entre aussi en adéquation avec le besoin d'applications dans des environnements mobiles où les composants disponibles peuvent varier. Cette dynamicité produit de nouvelles problématiques pour le test. En effet, une grande majorité des méthodes actuelles considèrent les com-

posants comme étant statiques et connus à l'avance. Ceci permet, entre autre, de définir un environnement de test fixe et de construire des cas de test fixes également. Le choix dynamique des composants ne permet plus cela.

Les applications dynamiques orientées service sont aussi de plus en plus souvent conçues pour être déployées dans des environnements partiellement ouverts tels que les Clouds. Afin de comprendre les prochain niveaux de détails sur le Cloud computing, rappelons sommairement son fonctionnement en le segmentant en couches comme suit :

- le niveau Infrastructure as a Service (IaaS) fournit l'infrastructure matérielle, les ressources logicielles par dessus Internet (serveurs, systèmes d'exploitation, stockage, bases de données, etc.),
- le niveau Software as a Service (SaaS) peut être défini comme un modèle de distribution dans lequel des applications sont maintenues par un vendeur ou un fournisseur de services et rendues disponibles "à la demande" pour des clients connectés par un réseau, typiquement Internet. On peut trouver actuellement comme exemple Salesforce.com, NetSuite, Google's Gmail ou SPSCommerce.net,
- le niveau Platform as a Service (PaaS) fournit une infrastructure dans laquelle des développeurs peuvent développer de nouvelles applications ou étendre des applications existantes. Salesforce.com's Force.com, Google App Engine, et Microsoft Azure sont des exemples de PaaS.

Nous considérons, dans cette Section, des environnements partiellement ouverts tels que les PaaS, qui représentent des environnements virtualisés ou des services et applications (SOAP Web services, RESTful Web services, Servlets, etc.) sont dynamiquement composés et déployés. Ce type d'environnement, composé de plusieurs couches complexes, réduit l'accès direct aux applications et peut aussi modifier l'observabilité de ces dernières. Les méthodes de test doivent donc s'adapter pour prendre en compte ces environnements.

3.3.2 État de l'art

Modélisation d'applications dynamiques orientées service

La composition de services a pour objectif de construire des applications pour répondre aux besoins soit industriels soit utilisateurs. En complément au développement manuel classique souvent fastidieux et assujetti aux erreurs, la découverte et composition dynamique de services est permise grâce aux descriptions de services et aux besoins de compositions qui peuvent être fournis suivant différents niveaux d'abstractions (opération, signature, protocole d'interaction, sémantiques, etc.). Cette composition dynamique nécessite aussi des modèles formels aux caractéristiques non triviales, pour décrire les services, les compositions avec les interactions possibles, les annotations sémantiques (pour favoriser la composition dynamique des processus) [RS04, CMP06, DBN08, MP09].

Les modèles peuvent rester simples si l'on considère des services sans état ayant des interactions simples. Au delà de cette limite, lorsque les services nécessitent des protocoles particuliers ou intègrent un état évoluant au fil des requêtes, il devient nécessaire d'utiliser des modèles comportementaux comme des pre/post conditions, des systèmes à transitions, des algèbres de processus, des réseaux de Petri (voir [BBG07]).

La composition de services prend généralement un point de vue central avec un unique client qui garde une structure statique [PTD+05]. En effet, la majorité des approches abordent la composition comme un ensemble d'activités statiques où les inter-

actions de services sont anticipées avec une cohésion parfaite entre les entrées/sorties requises/fournises par les composants et protocoles.

Test de compositions avec des méthodes de test actives

Le test de composants et de compositions a été étudié dans différents travaux, dans plusieurs domaines. Nous en avons présenté quelques uns en Section 3.1.1.

Nous avons notamment montré que, dans la plupart des travaux existants, les environnements, dans lesquels sont déployées les applications, sont ouverts. Très peu de travaux traitent d'autres environnements et en particulier d'environnements où les accès à l'implantation sont restreints.

3.3.3 Problèmes soulevés : composition dynamique, composition de versions d'une même application, tests

Malgré l'engouement manifeste des paradigmes du "Cloud programming" et de la programmation orientée service, on peut toujours remarquer aujourd'hui un manque d'outils prenant en compte la modélisation de compositions dynamiques de services, et des environnements autre qu'ouverts ou vides. Les outils actuels considèrent des compositions statiques uniquement et des environnements dont l'accès est total. Sans de nouveaux outils, il devient très difficile de modéliser des applications dynamiques orientées services, de développer un cycle de vie complet pour ces dernières, et finalement de concevoir des applications fiables.

Une perspective adjacente à la composition dynamique de services, concerne la composition de plusieurs versions d'une même application. En effet, tout type d'application est généralement suivi et révisé afin d'ajouter des fonctionnalités ou de corriger des bugs. Cependant, lorsque ces applications sont déployées sur Internet ou dans des Clouds pour être utilisées par des utilisateurs divers, il n'est pas toujours possible de simplement remplacer une ancienne version d'une application par une nouvelle. La première raison directe est liée à l'hétérogénéité des clients qui ont donc besoin de plusieurs versions. C'est un problème récurrent, soulevé par de nombreuses sociétés dont *Orange Business* qui nous a fait part de ses difficultés avec son application d'accès aux carnets de santé électroniques, en cours de développement [(Al11].

De ce fait, ces versions multiples doivent être accessibles et doivent cohabiter ensemble. Habituellement, cette cohabitation est conçue manuellement, ce qui représente une tâche non négligeable. De plus, elle doit être mise à jour dès qu'une nouvelle version est proposée. Cette problématique et les travaux existants sur la composition dynamique de services soulèvent donc les questions suivantes : comment peut-on modéliser des applications dynamiques orientées service tout en prenant en compte les environnements ? Comment modéliser la cohabitation de versions multiples d'une même application ? La composition obtenue peut-elle être dynamique ? Dans ce cadre de composition dynamique, peut-on réutiliser une suite de test ? Peut-on définir des spécifications assez abstraites pour modéliser la dynamicité mais aussi assez concrète pour pouvoir être utilisée pour tester ?

Comme il a été montré précédemment dans l'état de l'art, il existe peu d'approches dédiées au test de compositions dynamiques de services dans des environnements partiellement ouverts. De façon pragmatique, quelques sociétés construisent des cas de test

unitaires qui couvrent une faible partie de ces applications. Bien entendu, cela reste très insuffisant pour justifier de leurs fiabilités. Actuellement, la grande majorité des méthodes de test manipulent des compositions statiques avec lesquelles les services sont connus (nom, URL, location, etc.) dans des environnements complètement ouverts (le comportement complet des compositions peut être facilement observé). Néanmoins, comme nous l'avons montré à plusieurs reprises dans ce chapitre, les tendances actuelles s'appuient sur des environnements virtualisés dans lesquels les applications sont de plus en plus souvent des compositions dynamiques. Ceci mène aux problématiques et questions suivantes :

- en comparaison aux méthodes de test classiques, il semble difficile, dans le contexte des compositions dynamiques de services, de construire des suites de test statiques à partir d'une spécification. Il semble probable que les cas de test doivent être construit dynamiquement, pendant l'exécution de l'implantation sous test, pour prendre en compte les invocations dynamiques de services. De ce fait, il faut répondre à ces questions : quelles propriétés d'une composition dynamique de services peuvent être testées ? Est-il possible de construire des cas de test à la volée (pendant l'exécution) ? Les relations de test, telle que la ioco [Tre96c], qui formalisent le niveau de confiance entre une implantation et une spécification, sont-elles suffisantes ? Faut-il définir des relations plus abstraites ?
- l'un des problèmes soulevés par la composition dynamique de services dans des environnements partiellement ouverts, concerne le manque d'observabilité de l'implantation. Nous avons montré en Sections 3.1 et 3.2 et que ces environnements empêchent de relever toutes les réactions d'une implantation. Dans ces conditions, il devient difficile de construire des traces. La solution de décomposition/recompostion de traces, décrite en Section 3.1, apporte une première réponse, à condition que les composants soient connus. Pour des compositions dynamiques de services, la question de construction des traces se pose à nouveau. Ainsi, peut-on observer le comportement d'une composition dynamique de services différemment ? Peut-on extraire assez de réactions observables afin de conclure sur un quelconque verdict ? Peut-on monitorer de telles applications dans des environnements partiellement ouverts.

Ces différentes problématiques ouvrent vers plusieurs perspectives et travaux futurs présentés par la suite.

3.3.4 Travaux futurs

Modélisation de compositions dynamiques pour le test

Peu de travaux apportent des solutions sur la modélisation de compositions dynamiques de services, et aucun ne prend en compte l'environnement dans lequel la composition est exécutée. Il faut également proposer des modèles qui puissent être utilisés pour le test. Une première perspective est donc de formaliser ces compositions de façon abstraite pour ne pas expliciter les composants qui ne sont pas connus, mais avec suffisamment de détails sur le comportement pour justifier la faisabilité des tests.

A propos de ce niveau d'abstraction, une première solution simpliste est de considérer le modèle STS où les composants et opérations ne sont pas explicitement donnés mais sont décrits sous forme de variables. Une exemple préliminaire est donné en Figure 3.62. Une première action consiste à rechercher un composant suivant des fonctionnalités. Puis une opération est ensuite appelée. Le nom du composant et de l'opération sont donnés par

l'action *rechercher_ composant*. Kanso et al. proposent une autre piste dans [KABT10] sur la modélisation de composants abstraits. Mais rien n'est donné sur la composition en elle-même.

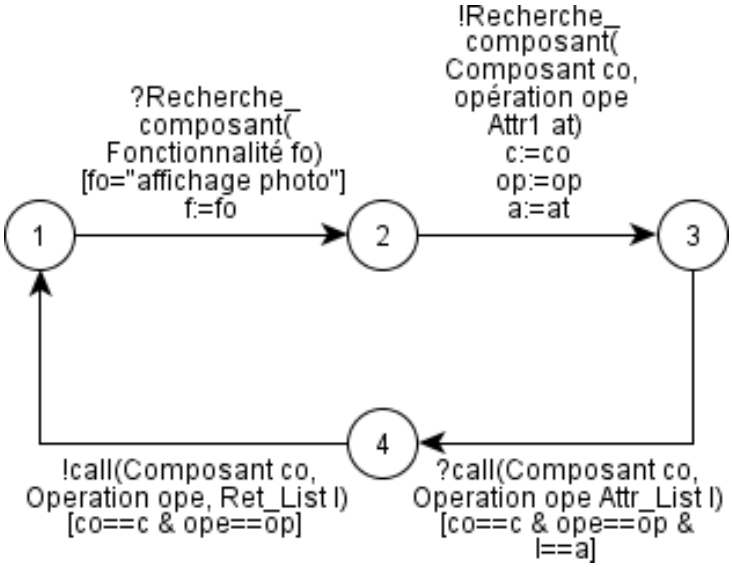


FIGURE 3.62 – STS avec abstraction sur les composants et opérations

Il serait aussi intéressant de prendre en compte l'environnement dans lequel sont déployées les compositions. Certains langages abstraits comme BMM, BPMN 2.0 et SoaML, intégrant les derniers standards de l'OMG pour les architectures d'entreprises (EA) proposent des solutions en accord avec les principes TOGAF [Fra] (The Open Group Architecture Framework) qui permet la formalisation d'architecture d'entreprises, d'applications de données, etc. Ce qu'il faut ici étudier c'est la faisabilité à tester le comportement d'une composition dynamique dans un environnement partiellement ouvert à partir d'une de ces spécifications.

Composition de plusieurs versions d'une même application

Les applications orientées service (services Web, compositions, etc.) correspondent souvent à une logique Business composée de plusieurs couches. De telles applications sont généralement continuellement révisées par des modifications majeures ou mineures en même temps que la mise à jour des applications clientes. Ceci conduit à l'implantation de plusieurs versions d'une même application et de plusieurs applications clientes associées. Les anciennes versions ne peuvent être supprimées du fait de l'hétérogénéité des clients. Toutes les versions doivent être accessibles et doivent cohabiter ensemble afin que tout client puisse invoquer la version appropriée. Actuellement, aucune approche formelle et générique offre des solutions permettant de déployer plusieurs versions d'une même application dans le même environnement. Cette composition est habituellement construite manuellement et est statique.

Nous souhaitons donc répondre à cette problématique en proposant des solutions de modélisation de composition de versions multiples d'une même application. Une première piste consiste à séparer et définir ce que sont des modifications majeures et mineures, afin

de séparer celles qui sont transparentes pour les clients de celles qui modifient l'interface des services et leurs interactions i.e. les modifications qui modifient/ajoutent/enlèvent des opérations, et/ou les données manipulées par ces opérations. Lorsque des modifications majeures impliquent la publication d'une nouvelle version, nous proposons de la composer avec les anciennes versions pour que la composition dans son ensemble soit accessible par tout type de clients. La partie précédente, traitant de modèles de composition dynamique de services, pourra apporter des choix sur le modèle adéquat. De façon plus précise, cette tâche peut se décomposer en trois parties successives :

1. **Composition statique de versions multiples d'une même application dans un environnement partiellement ouvert :** dans cette première approche, nous supposerons que la composition est statique afin de faciliter la modélisation, et de se focaliser sur la composition en elle-même, qui doit pouvoir être interrogée par tout type de client,
2. **Composition dynamique de versions multiples d'une même application :** nous supposerons ensuite que la composition est dynamique i.e. une version peut être ajoutée ou supprimée dynamiquement. Nous pensons qu'une telle composition peut être modélisée en définissant les modifications majeures d'interfaces et en s'appuyant sur les travaux existants de composition. Nous devrons cependant proposer des algorithmes supportant une adaptation entre les multiples versions entrant en conflit à l'exécution,
3. **Réutilisation des suites de test, test de régression :** Dans cette partie, nous souhaitons nous pencher sur la construction de tests sur la composition en entier. Il semble peu pertinent de reconstruire tous les tests de toutes les versions composées, à chaque fois qu'une version est ajoutée ou supprimée. Nous souhaitons proposer des méthodes pour premièrement organiser de façon hiérarchique une suite de test selon la composition des versions multiples, puis pour mettre à jour une telle suite de test seulement lorsque la composition est modifiée (en supposant, pour chaque version, que les cas de tests existent déjà). Avec de telles méthodes, il devient possible de vérifier qu'une mise à jour majeure n'introduit pas de nouveaux bugs. En d'autres termes, il devient possible d'exécuter des tests de régression afin d'assurer que tout client pourra interagir avec la composition de versions.

Test de compositions dynamiques de services

La composition dynamique de services ouvre également des perspectives sur les méthodologies de test. Quelles qu'elles soient, nous pensons que celles-ci auront comme point commun la génération de cas de test abstraits (patrons de tests) et la génération de traces abstraites lors de l'expérimentation. Sous le terme abstrait, nous sous entendons la génération partielle de ces cas de test, qui devront alors être complétés à l'exécution par le testeur avec les données (composants, variables, etc.) récoltées durant l'exécution.

Pour les méthodes de test de conformité, il faut définir de nouvelles relations de test abstraites à partir desquelles des cas de test abstraits seront générés pour les vérifier. Il sera alors fortement pertinent de comparer de telles relations de test avec des relations plus classiques avec lesquelles les cas de test sont concrets. La comparaison peut se faire en terme de couverture, de détection d'erreurs, d'observation du comportement, etc. Ces méthodes de test pourront aussi prendre en compte des caractéristiques précises que l'on

trouve dans le Cloud computing, comme la notion de rôles (service, workers, etc.) la commutation de rôle, l'appel multicast, etc.

Concernant les méthodes de test de robustesse, celles-ci pourront être basées sur ces deux techniques :

1. test de robustesse liée à la charge du réseau utilisé par la composition en montant en charge de façon extrême le réseau par l'exécution d'un grand nombre de scénarios de test utilisant de lourdes données, de façon simultanée ou répétitive ou par l'utilisation d'un simulateur de l'environnement,
2. combinaison de méthodes actives et de l'injection de fautes. Premièrement, les cas de test seront générés par des méthode de test actives, puis des scénarios basés sur ces cas de test et incluant des fautes seront injectés dans l'environnement partiellement ouvert dans le but de vérifier la robustesse de l'implantation. Il faut cependant déterminer une solution possible à l'injection de ces scénarios dans un tel environnement.

3.4 Le test de sécurité d'applications mobiles

Cette partie présente plusieurs problématiques d'actualité sur le test des applications mobiles qui l'ont peu trouver dans les smartphones aujourd'hui. Ce projet de recherche a commencé en novembre 2011 et est effectué, dans le cadre d'une Bourse Innovation Thèse, en collaboration avec l'entreprise Openium dont les activités sont centrées sur le développement de telles applications.

Ce projet de recherche a pour objectif de contribuer à la découverte et à la modélisation des vulnérabilités / attaques qui peuvent être rencontrées sur des applications pour systèmes mobiles (téléphones, tablettes, etc.) et de définir des méthodes de test de sécurité afin de détecter ces vulnérabilités. Les problématiques se trouvent essentiellement centrées dans la partie exécution des test et architecture de test.

3.4.1 Contexte

Actuellement, l'envolée des ventes de systèmes mobiles et d'applications associées n'est plus à démontrer. Par exemple, l'*Apple store* a atteint le nombre de 300,000 applications il y a plusieurs mois. Ces différentes applications permettent, par exemple, de "rester en connection", en gérant ses emails, en synchronisant son calendrier et taches, d'écouter de la musique à partir d'un Cloud, etc.

Tandis qu'une analyse récente [IDC] montre que la vente d'applications mobiles va atteindre le chiffre de $35 Billion en 2014, d'autres rapports alarmistes [net, fir] décrivent la sécurité de ces applications comme cauchemardesque en montrant qu'environ 20 % des applications proposées présentent des failles de sécurité. Parmi ces dernières, nous pouvons citer le manque de protection des données personnelles, l'accès non autorisé aux smartphones, l'utilisation d'applications malicieuses, etc. Plusieurs de ces vulnérabilités sont connues, mais d'autres sont liées à de nouvelles fonctionnalités et posent de nouvelles difficultés. Par exemple, les smarphones basés sur Android ou sur IOS, peuvent permettre la géo-localisation d'un utilisateur à travers des applications, et offrent les informations

collectées à d'autres applications sans aucun consentement. Certaines applications malicieuses effectuent des transferts de messages via SMS ou MMS sans demander aucun accord non plus.

Actuellement, et malgré ce contexte, il existe peu de méthodologies et d'outils pour tester la sécurité de ces applications. Aucun outil existant n'est basé sur des méthodes formelles. Ces outils permettent au plus un test unitaire assez restreint suivant la plateforme :

- Automated QA - TestComplete 8 : cet outil proposé pour Microsoft Windows Mobile, Symbian et Palm OS correspond à un debugger étendu qui permet d'observer certaines applications mobiles et les registres XML associés à ces applications,
- Mob4Hire - MOB TEST SUITE est un outil de test fonctionnel et utilisateur. Un ingénieur entre des cas de tests manuellement puis analyse les résultats obtenus,
- DeviceAnywhere - MonitorAnywhere, Test center : le premier outil aide à monitorer sur un ordinateur des applications mobiles. Il fournit les réactions observables obtenues, qu'un expert doit alors analyser. Des propriétés importantes, comme la couverture des tests, ne sont pas prises en compte. Le second outil permet le test unitaire manuel.

Néanmoins, on voit, à travers les sociétés de développement et même la presse spécialisée, que le besoin d'outils commence à devenir crucial. Ainsi, nous souhaitons proposer notre contribution à travers la proposition d'approches formelles de test de sécurité pour ces applications mobiles. De telles approches permettront aux entreprises et au utilisateurs de faire confiance aux applications.

3.4.2 État de l'art

La sécurité des applications et systèmes est un domaine connus où plusieurs travaux ont été proposés à propos de la modélisation des polices de sécurité, de la recherche et classification des vulnérabilités, de l'amélioration de la sécurité par suppression de vulnérabilité et du test. La modélisation de polices de sécurité est une première étape importante qui a pour objectif de décrire les permissions ou interdictions d'accès à un système. Dans la littérature, ces polices de sécurité sont souvent modélisées formellement au moyen de règles qui régulent la nature et le contexte des actions qui peuvent être effectuée sur une application, selon des rôles spécifiques (utilisateur, système, etc.) Les règles de sécurité permettent également de décrire des critères précis tels que [OWA03, GL06, ISO09, SP10] :

- **la disponibilité :** qui représente la capacité d'un système à répondre correctement quelquesoit la requête émise. Une conséquence directe est qu'un système est disponible ssi ce dernier ne "plante" pas quelquesoit la requête,
- **l'authentification :** qui a pour but d'établir ou de garantir l'identité d'un client et de vérifier qu'un client sans autorisation correcte, n'a pas les permissions correspondantes. Le processus de connection est souvent la première étape pour l'authentification d'un client,
- **l'autorisation :** représente la police d'accès qui spécifie les droits d'accès aux ressources, habituellement à des utilisateurs authentifiés,
- **l'obligation :** qui, en contraste à la permission et à l'interdiction, est souvent associée avec des temporisations qui bornent les besoins en temps de la disponibilité.

Quelques langages ont été proposés dans [OAS04, SBC05, BW07, KBM+03] afin de définir formellement ces règles. Nomad [CCBS05], qui est l'un de ces langages, exprime des propriétés telles que la permission, l'interdiction ou l'obligation en étant capable de prendre en compte les délais de réponses. Ces règles peuvent ensuite être utilisées avec des proxy ou des pare-feux pour empêcher des attaques. Mais elles peuvent aussi être employées en combinaison avec des méthodes de test [MBCB08, MMC09]. Par exemple, la méthode que nous avons proposé en [SLR10], construit des cas de test à partir de règles Nomad pour tester la sécurité des services Web en considérant ces règles comme des patrons de test.

Concernant les applications mobiles et smartphones, quelques travaux se sont penchés sur la protection des données personnelles. Ces données, stockées sur les appareils, peuvent être extraites par différent moyens e.g., des applications malveillantes ou par piratage du smartphone. Des approches préliminaires, relatives à la protection de malware par application de type antivirus, ont proposé des analyses sur les possibilités plus limitées des smartphones, en terme de puissance de calcul et de mémoire [CWYL07]. Différentes directions ont été suivies par des travaux portant sur la détection et la prévention d'applications malicieuses sur smartphones. L'un de ces travaux [YLLD09] analyse et donne des aperçus sur les limitations des smartphones et l'utilisation d'OS distincts qui nécessitent leur propre kit de développement (Google Android, IOS, Symbian OS, etc.). Ces différents points sont vus comme des restrictions qui rend plus difficile le piratage des ces appareils. Algesheimer et al. proposent, dans [ACCK01], de réguler l'accès aux données au moyen d'un contrôleur qui a pour rôle de vérifier les droits d'accès des utilisateurs. Dans le même ordre d'idée, des méthodes de cryptographie ont été proposées pour éviter la visualisation directe de données personnelles.

Concernant le test de sécurité, plusieurs approches ont été proposées. Quelques unes sont citées ci dessous :

- *Génération de test à partir de polices représentées à base de modèles :* souvent, concernant la génération de test de polices de sécurité, des cas de test abstraits sont directement générés à partir de la police. Par exemple, Le Traon et al. [LTMB07] proposent des techniques de génération de cas de test abstraits (non valués) à partir de règles de sécurité modélisées par OrBAC. Senn et al. proposent, dans [SBC05], comment formaliser des polices de sécurité pour des protocoles et comment générer des cas de test à partir du modèle obtenu. Dans [DFG+06], des règles de sécurité pour les protocoles sont testées en modélisant un comportement du réseau avec des LTSs. Ensuite des patrons de test sont injectés dans des cas de test existants afin de valider les règles. Mallouli et al. [MOC+07] proposent une approche pour générer des cas de test abstraits pour tester des polices de sécurité. Des EFSMs (Extended Finite State Machine) et des critères de test sont employés pour décrire des règles de sécurité basiques. Des objectifs de test sont ensuite construits pour tester ces règles,
- *Génération de test aléatoire :* est une technique qui construit automatiquement ou semi-automatiquement des cas de test aléatoirement avec des valeurs de variable aléatoires. Par exemple, dans [Mar06], les auteurs ont développé une approche à partir de polices écrites en XACML. Cette dernière est analysée pour générer les cas de test en sélectionnant aléatoirement des requêtes dans l'ensemble des requêtes possibles,

- *Test par Mutation :* cette technique conduit généralement à la mutation de programmes ou de polices. Dans [RAD78], des mutants représentent des codes modifiés à partir d'une police initiale. Un testeur vérifie que la police peut détecter les mutants. Dans [MFBT08], les auteurs proposent une approche pour spécifier des polices de sécurité pour des applications Java. La police est modélisée en OrBAC puis est transformée en XACML. Ensuite, la police est intégrée dans l'application. Des fautes sont injectées dans la police pour valider la police originale en effectuant des mutations sur les règles,
- *Test par injection de fautes :* La disponibilité est souvent étudiée grâce à des méthodes de test de robustesse basées sur l'injection de fautes [HTI97]. Par exemple, dans [LX03], des fautes sont injectées au niveau des couches réseaux entre le niveau application et la pile protocolaire. Le modèle de fautes considère des fautes de communication comme la suppression de messages, la duplication, etc.

Concernant le test des applications mobiles, très peu de travaux ont été proposés. Quelques uns proposent des solutions de monitoring des smartphones pour vérifier la satisfaction de propriétés basiques ou pour détecter des activités anormales provoquées par des applications. La méthode, proposée dans [SPLA07], définit une plate-forme de monitoring dans laquelle chaque appareil mobile envoie régulièrement des rapports composés de variables de l'OS. Cette méthode détecte des anomalies sur l'OS uniquement (mémoire, nombre de processus, etc.). D'autres travaux comparent le comportement des smartphones avec un fichier d'historique existant pour détecter des comportements inhabituels et alerter les utilisateurs sur les difficultés potentielles [CMMR08, MSS+08]. Cependant, ces méthodes considèrent des anomalies liées ou non à des attaques de sécurité.

3.4.3 Problématiques

Plusieurs problématiques sont apportées par le test de sécurité des applications mobiles et sont liées, entre autre, à l'environnement restreint et différent du smartphone, les caractéristiques particulières des smartphones (géo-localisation, etc.)

L'une des premières problématiques concerne le manque de connaissance des vulnérabilités et attaques que l'on peut rencontrer avec des smartphones. Plusieurs travaux ont étudié la question mais de nouvelles vulnérabilités sont trouvées tous les mois. Il est nécessaire d'au moins rassembler l'ensemble des vulnérabilités existantes et probablement de les formaliser. Comme différentes plate-formes de développement sont proposées (Google Androïd, Apple IOS, Symbian, BlackBerry Os, etc.), il semble nécessaire de se focaliser sur des concepts et propriétés comme la disponibilité, l'autorisation ou l'intégrité. Les documents, proposés par l'organisation OWASP [OWA03] seront une base à cette étude. Cette partie soulève au moins les questions suivantes : peut-on modéliser tout type de règle de sécurité vis-à-vis des vulnérabilités connues ? Ces règles peuvent-elle être indépendantes à l'OS ?

Concernant le test de sécurité d'applications mobiles, les principaux verrous se concentrent sur la génération des tests et leur exécution : ces deux points reposent sur les actions qui peuvent être produites et observées à partir des applications mobiles. Ces actions ont différentes natures et peuvent être soit des interactions utilisateur, soit des envois/réceptions de messages, ou des requêtes à des applications Web externes. Toutes ces actions hétérogènes doivent être prises en compte au niveau des spécifications, des cas de test et de

l'exécution des tests.

La phase de test doit aussi prendre en compte l'environnement distribué utilisé par les applications mobiles. Ces dernières reposent souvent sur des services Web ou des applications Web sur Clouds (couche PaaS) pour par exemple gérer des données. Ceci conduit à un environnement complexe dans lequel l'application en entier est distribuée sur plusieurs sites hétérogènes (smartphone, serveur, Cloud, etc.) Des travaux futurs doivent donc définir des architectures de test spécifiques pour le test actif et passif. En premier lieu, quels types de relation de test peut-on définir entre l'implantation, sa spécification et les règles de sécurité représentés par des modèles ? La génération des tests qui en découle doit s'appuyer sur ces modèles. Mais peut-on modéliser les différentes actions possibles qui peuvent être faites à partir d'applications mobiles ? Peut-on aussi trouver une adéquation entre les différentes règles de sécurité, les langages utilisés, la couverture de test et la complexité de génération des cas de test ?

Les méthodes de test doivent aussi être implantées en prenant en compte la puissance de calcul des smartphones. En effet, l'exécution du test ne doit pas interférer sur le fonctionnement normal des applications mobiles.

Ces problématiques tendent à proposer les perspectives décrites dans la Section suivante.

3.4.4 Perspectives et Travaux futurs

Définition formelle de règles de sécurité

Cette tâche a pour objectif de modéliser une police de sécurité au moyen de règles décrites formellement. Ces règles pourront être basées sur les vulnérabilités trouvées pour les applications mobiles. De toute évidence, il est alors nécessaire d'au moins couvrir les recommandations fournies par l'organisation OWASP.

Comme nous l'avons dit précédemment, il faut au préalable rassembler la liste exhaustive des attaques possibles. Pour cela, nous comptons sur la collaboration et l'expertise de la société Openium, qui est fortement intéressée par ce travail de recherche et dont les activités se concentrent sur le développement d'applications mobiles. Des notions telles que la géo-localisation et les délais (quiescence) doivent être prises en compte également.

Pour modéliser les règles de sécurité, quelques langages ont été introduits dans [SBC05, CCBS05, BW07] . Nous proposons de modéliser ces règles avec le langage Nomad [CCBS05] pour exprimer la façon dont une application mobile doit agir en cas de réception d'attaque e.g., brute force, l'injection SQL, etc. Nomad est basé sur une logique temporelle, étendu avec des modalités "déotic" (représentant les agents) et "aléthic" (possibilité, nécessite, interdiction).

Définition de méthodes de test

Les méthodes de test pourront être actives ou passives, la combinaison des deux types offrant souvent de bon résultats. Celles-ci doivent s'appuyer sur les critères génériques de sécurité cités précédemment, à savoir la disponibilité, l'authentification, l'autorisation, l'intégrité.

Concernant le test actif, une piste est de considérer les règles de sécurité comme des besoins de test qui seront transformés en objectifs de test. Ceux-ci peuvent ensuite être

combiner à une spécification pour donner un produit exprimant les comportements de la spécification combinés (respectant) aux règles de sécurité. Les cas de test pourront être alors extraits classiquement. Néanmoins, des caractéristiques telles que la géo-localisation semblent assez difficile à appréhender avec cette technique. D'autres approches, basées sur le test aléatoire, sont aussi à considérer car elles sont très intéressantes pour le monde industriel de part leur coût plus faible. Le test aléatoire consiste généralement à sélectionner uniformément des données de test sur le domaine des actions d'entrée d'une application. Lorsque la sélection aléatoire est basée sur des profiles opérationnels, on parle aussi de test opérationnel ou statistique et permet de faire des estimations sur la fiabilité. Une première piste serait alors de transposer la méthode décrite dans [SR] qui consiste à couvrir au moins une fois chaque action d'une spécification en construisant aléatoirement des tuples de valeurs, tout en restreignant au départ les domaines des variables utilisées afin d'éviter une explosion combinatoire du nombre de possibilités. Nous souhaitons également proposer une méthode de génération de cas de test unitaire, construits semi-aléatoirement. A partir de cas de test existants, nous souhaitons : détecter les composants associés, les tester aléatoirement de façon séparée, trouver une spécification partielle à partir des cas de test, et construire des cas de test aléatoirement à partir de cette dernière.

Nous pensons également que le test passif doit être appliqué pour tester la sécurité des applications mobiles, notamment pour tester l'autorisation. Ce test peut permettre par exemple de détecter, à plus long terme, l'intrusion ou l'envoi de données personnelles sans autorisation mais aussi analyser l'intégrité des données ou les temps de latence, etc. Dans ce cas, le testeur, qui doit se trouver sur le smartphone, doit extraire tout type de messages et doit tester la satisfaction de règles de sécurité, données par exemple sous forme d'invariant, soit sur l'ensemble des applications en cours d'exécution, soit sur une application précise. Ce type de test pourrait détecter à la volée les applications ne respectant pas les restrictions déclarées à l'utilisateur et les applications malveillantes. Ce dernier point montre tout son intérêt face aux rapports récents montrant que 20 % des applications mobiles de l'*Android Market* envoient des données personnelles à d'autres applications et que 5% de ces applications sont malveillantes (envoi d'SMS sans autorisation, appels vers d'autres smarphones, lecture des SMS, enregistrements audio, etc.). Cependant le test passif ne peut pas être effectué par des sniffeurs dans le cas des smartphones. Nous proposerons une méthode qui utilise les patrons de conception utilisés dans les framework de développement mobiles. Notamment, nous utiliserons le patron Delegate qui permet de faire des actions lors d'évènements reçus. Nous proposerons de générer automatiquement le code des délégués à partir du concept de proxy-testeur, décrit précédemment, en combinaison à des règles de sécurité.

Finalement, pour chaque approche, il sera aussi nécessaire de mesurer la couverture des tests à partir d'une spécification, et la couverture des vulnérabilités i.e. le nombre de vulnérabilités qui peuvent être détectées.

Architectures de test pour le test d'applications mobile

La notion d'architecture de test est fondamentale dans le processus car c'est elle qui décrit les accès de contrôle et d'observation. En ce qui concerne les systèmes de communication, le standard ISO ISO 9646 propose plusieurs types d'architectures de test, composées de points d'observation (PO) et de points de contrôle et d'observations (PCO), à travers lesquels un testeur peut interagir avec une implantation sous test. Ces architec-

tures peuvent être locales ou distribuées suivant le nombre de testeurs utilisés et l'accès aux POs et PCOs.

Indépendamment du matériel, il faut définir formellement des architectures de test pour les tests actifs et passifs. Nous supposerons que les applications mobiles ne fonctionnent pas seules mais font appels à des services (services Web, applications Web de type PaaS sur Clouds), comme c'est généralement le cas. De part l'environnement distribué utilisé par les applications mobiles, l'architecture devra être répartie sur les smartphones, sur les serveurs Web ou des proxy. Dans le cas où des Clouds sont utilisés, une ou plusieurs architectures spécialisées devront être proposées car l'environnement est obligatoirement partiellement ouvert (voir Sections 3.1 et 3.2.2). Une solution préliminaire serait d'utiliser le concept de proxy-testeur, comme nous l'avons décrit en Section 3.2, ce dernier permettant de récupérer facilement toutes les réactions produites par une application Web dans un Cloud.

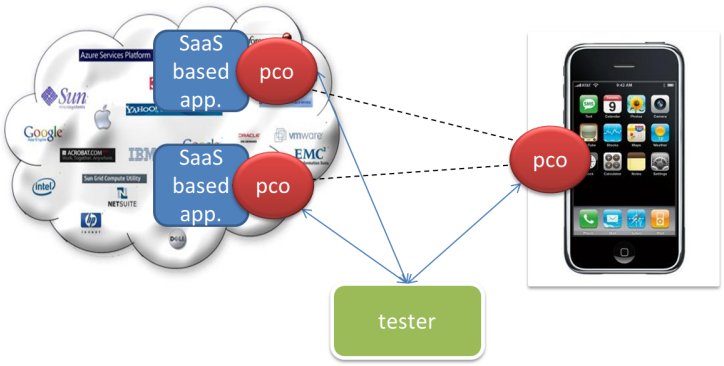


FIGURE 3.63 – Architecture mobile 1

Pour le test actif, plusieurs PO et/ou PCO sont nécessaires : un premier PCO doit être posé au niveau du smartphone pour accéder à l'application mobile sous test (interactions utilisateur, lecture des réactions, accès à des fonctionnalités du smartphone, etc.). Un PO ou PCO peut être placé sur le smartphone pour extraire les messages envoyés sur un réseau Wifi ou 3G vers des services ou applications Web. Un dernier PCO devrait être placé du coté serveur, soit directement sur un serveur Web, soit en utilisant la notion de proxy-testeur. Une architecture préliminaire est illustrée en Figure 3.63.

Dans le cas du test passif, les architectures de test doivent prendre en compte la notion de monitoring souvent implantée par des outils basés sur des "sniffeurs" réseaux. L'implantation n'est donc pas contrôlée, donc seuls des PO doivent être employés au niveau du smartphone pour extraire ses réactions. La Figure 3.64 décrit une architecture préliminaire dans laquelle le testeur vérifie que les traces fournies par les PO satisfont des règles de sécurité.

Ces types d'architecture soulèvent en premier lieu les problèmes suivants :

– Le PCO, qui permet d'interagir directement avec l'application mobile, doit donner des actions très hétérogènes comme des gestes utilisateur, des données dans des champs, etc. Un tel PCO correspond à une sur-couche similaire à l'outil *Selenium* [sel] qui permet d'effectuer le test d'interfaces Web, en ajoutant des données dans une page sans intervention humaine. Malheureusement, un tel outil n'existe actuellement pas pour les applications mobiles. Une solution est d'utiliser et d'augmenter les

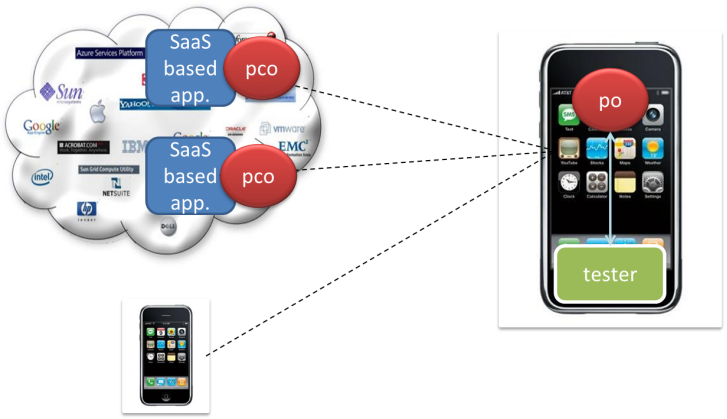


FIGURE 3.64 – Architecture mobile 2

frameworks de cas de test unitaires, trouvés dans certains environnements de développement pour smartphones (Android et IOS). Ceux-ci permettent, avec quelques restrictions, d'appeler des fonctions d'une application et de simuler des gestes (glissement, déplacement d'objets, zoom, etc.),

- nous supposons que les applications mobiles interagissent également avec des services. Simuler ces services serait une piste intéressante pour faciliter le test et pour simuler l'envoi d'attaques. La génération de tels simulateurs a déjà été étudiée dans le cadre des services Web [LZCH08b]. Cependant, il semble nécessaire d'étendre ce travail pour prendre en compte des caractéristiques nouvelles de services (notions de rôles, etc.),
- ces architectures ne pourront pas s'appuyer sur des sniffeurs pour des raisons techniques et de puissance de calcul. Elle devront être intégrées au code des applications, ce qui est pleinement innovant. Comme nous l'avons dit précédemment, nous considérerons le patron délégué comme objet de filtre des évènements. Cette approche sera combinée à la notion de proxy-testeur pour tester passivement l'application.

Chapitre 4

CV

SALVA Sébastien

37 ans, né le 8 août 1974
Nationalité française
Marié

Adresse professionnelle :
Université d'Auvergne
IUT d'Aubière - Département d'informatique
Plateau des Cézeaux
63000 Aubière

Situation Actuelle

Maître de conférences depuis septembre 2002, CNU 27ème section, IUT d'Aubière, Université d'Auvergne

Membre du laboratoire LIMOS (Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes), UMR 6158, Axe SIC

Situation Antérieure

2001 - 2002 : ATER à l'Université de Reims Champagne Ardenne

Cursus Universitaire

Novembre 2001 **Doctorat spécialité Informatique** Mention Très Honorable, Université de Reims Champagne Ardenne,
Directeur : Simon Bloch, Co-directeur : Hacène Fouchal,
Sujet : Etude de la qualité de test des systèmes temporisés.
Jury :
Rapporteurs :
J. P. Thomesse, LORIA, Professeur à l'Université de Nancy
N. Debnath, Professeur à Winona State University, USA
Examinateurs :
K. Drira, Chargé de Recherche au LAAS, Université de Toulouse
J. Zaytoon, Professeur à l'Université de Reims
Directeur de thèse :
S. Bloch, Professeur à l'Université de Reims
Codirecteur de thèse :
H. Fouchal, Maître de conférence à l'Université de Reims

Juin 1999 **DEA d'Informatique** Mention Bien, Université de Versailles (78),
Directeur : Guy Pujolle, **Sujet :** MISI : Méthodes Informatiques des Systèmes Industriels.

Juin 1998 **Maîtrise d'Informatique** Mention Bien, major de la promotion, Université de Reims Champagne Ardenne

Juin 1997 **Licence d'Informatique** Mention Bien, major de la promotion, Université de Reims Champagne Ardenne

Juin 1996 **Deug de Mathématiques**, Université de Reims Champagne Ardenne

Juin 1994 Baccalauréat série C,

Synthèse des activités

Recherche

Depuis 2002, mes activités de recherche se déroulent au sein du laboratoire LIMOS (équipe SIC). Mes travaux de recherche ont trait à la validation formelle à base de mo-

dèles :

- Validation de systèmes temporisés, test de conformité et de robustesse de systèmes modélisés sou forme d'automates temporisés à temps continu. Conception d'un outil parallèle de génération de cas de test, conception d'une architecture de test du protocole WAP (sur mobiles),
- Validation de systèmes réactifs, de services Web et de clients riches en AJAX (conformité, robustesse, sécurité). Conception d'un outil de test automatique (random testing) de services Web et de composants AJAX.

Publications :

- 7 publications dans des revues nationales et internationales avec comité de lecture
- 1 communication internationale invitée dans une conférence avec comité de lecture
- 25 communications internationales dans des conférences avec comité de lecture
- 5 communications internationales dans des conférences avec comité de lecture
- 5 présentations lors de séminaires

Interactions avec la communauté scientifique :

- Participation à des comités de lecture : ICEME 2012, IEEE ICWS 2011, IEEE AQTR08, OPODIS 2003, SNPD02, SNPD00,
- Participation à des comités de programme de conférence : ICIW 2010, 2011, 2012, CTRQ 2010, 2012, ICSEA 2012, SeNTApE'2010,
- Participation aux comités de rédaction des journal *International Journal of Computing and Information Sciences* (IJCIS), *International Journal On Advances in Internet Technology* (IJAIT), participation au comité de lecture du Journal *Software and Systems Modeling* (SOSYM) Elsevier,
- Organisateur des journées du groupe de travail MTVV (GDR GPL), 27-28 octobre 2011, Rennes,
- Membre de la commission de spécialiste 27ème Section, de l'Université d'Auvergne de 2004 à 2008 et 2011, de l'école Polytech Clermont 2012.

Encadrement de jeunes chercheurs :

- 1 thèse en co-encadrement depuis septembre 2008, soutenue en janvier 2012,
- 1 thèse en co-encadrement depuis décembre 2009, inscription non renouvelée après 2 ans,
- 1 thèse en co-encadrement depuis novembre 2011 dans le cadre d'une Bourse d'Innovation Thèse, en collaboration avec la société Openium,
- 3 Masters en formation M2 *Modèles, Systèmes, Imagerie, Robotique (MSIR).*

Projets et activités contractuelles :

- Projets nationaux : Projet RNRT Platonis, Projet ANR Webmov,
- Projet Évaluation de la robustesse des réseaux de capteurs sans fil dédiés aux applications industrielles, dans le cadre de la fédération TIMS (Technologie de l'Information de la Mobilité et de la Sureté),
- Responsable d'une bourse d'innovation en partenariat avec l'entreprise Openium

Enseignement

Mon activité d'enseignement s'effectue principalement à l'IUT d'Aubière et concerne les élèves de DUT et de Licence Professionnelle SIL. J'ai mis en place et j'assure des enseignements qui portent sur les réseaux, la programmation orientée objet, la programmation Web, la qualité logicielle, etc. Depuis mon arrivée à l'IUT d'Aubière, j'encadre tous les ans des projets et des stages d'élèves de DUT et de Licence. Depuis Septembre 2006, je suis responsable de la licence professionnelle *Développement d'applications Intranet/Internet*. J'ai également été membre élu du conseil de département depuis 2004 et responsable des stages pendant 2 ans. J'ai mis en place des partenariats avec plusieurs entreprises (Atos-Origin, Prizee, La Montagne, Michelin, etc.).

Je présente ci-dessous une liste non exhaustive des cours enseignés ces dernières années :

- **DUT :** Cours, TD, TP en réseaux (réseaux locaux, IP, TCP/UDP, etc.) et programmation Web (PHP, servlet, JSP, design pattern, JPA), projets tuteurés, stages,
- **Licence professionnelle :** Cours, TD, TP en réseaux, introduction à la programmation parallèle (OPENMP), services Web (Java AXIS), Cloud computing, Qualité logicielle (norme CMMI, analyse UML, tests, etc.), Administration système, préparation à la certification SUN, projets tuteurés, stages.
- **École d'ingénieurs ISIMA :** projets tuteurés.

4.1 Activité de Recherche

Mon activité de recherche s'effectue dans le laboratoire LIMOS (laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes), dans l'équipe SIC (Systèmes d'Information et de Communication) animée par les Professeurs Michel Misson et Farouk Toumani. Mon activité de recherche est actuellement centrée sur la validation de systèmes conduite par des modèles (test en boite noire, actif ou passif, de sécurité de conformité, de robustesse, etc.)

4.1.1 Thème de recherche

Mon activité de recherche a porté sur les thèmes suivants :

4.1.2 Méthodes de Génération de cas de test pour systèmes temporisés

J'ai effectué des travaux sur le test de systèmes et protocoles de télécommunication modélisés par des automates temporisés [Sal02, SRF02, SL03c, SL03b, SF04, SL05, SL07, SR09b]. Les travaux, proposés dans ce domaine, ont initialement suivi la voie de transformer une spécification temporisée en plusieurs autres modèles successifs afin de "masquer" les propriétés temporelles et de pouvoir appliquer des méthodologies de test existantes (caractérisation d'états, objectifs de test, etc.). Dans les travaux que nous avons proposés, nous avons souhaité mettre l'accent sur la réduction du coût de la génération des tests. Pour ce faire, au lieu de transformer une spécification initiale en plusieurs fois, nous avons apporté une nouvelle définition de la caractérisation d'états et des méthodes orientées objectif de test à appliquer, de façon directe, sur une spécification temporisée.

Les méthodes orientées objectif de test sont non exhaustives et permettent d'expérimenter des parties locales de systèmes à partir de besoins de test. Généralement, l'utilisation de telles méthodes réduit fortement les coûts du test. Cependant, ces objectifs de test doivent souvent être construits manuellement. En plus de devenir rébarbatif, cette construction peut aussi devenir un casse-tête si le système est complexe, temporisé, interopérable, etc. Des méthodes de génération automatique d'objectifs de test ont été proposées, mais uniquement pour les systèmes non temporisés. Nous avons donc apporté plusieurs solutions pour générer des objectifs de test semi-automatiquement et automatiquement sur des systèmes temporisés. Pour assurer et améliorer l'efficacité des objectifs, nous avons défini un langage rationnel de description qui prend en compte la notion de testabilité.

4.1.3 Test et testabilité de services Web et compositions

Je me suis également penché sur les problématiques et besoins liés au paradigme du service et à la composition de services vis-à-vis des tests [SR08b, SR08a, SR09a, RS09, SR10b, SR, SR10a, CCM$^+$10, SLR10, Sal11d, Sal11b, SR11a, SR11b].

Notre équipe, a premièrement intégré le projet ANR Webmov (`http://webmov.lri.fr/`) dont l'objectif principal était de contribuer à la conception, à la composition et la validation de services Web à travers une vue abstraite de haut niveau et d'une vision SOA

(Architecture Orientée Services) basée sur une architecture logique. Ce projet traite plus particulièrement de la conception et des mécanismes de composition de services Web et également de leur validation à partir de différentes techniques de test actives et passives. A travers ce projet, nous avons contribué à l'élaboration de plusieurs méthodes de test actives :

- Test aléatoire de services à partir de leur signature : nous avons proposé plusieurs méthodes de test aléatoire qui offrent l'avantage d'être automatique. Nous avons étudié le test utilisateur de service Web en se focalisant sur les propriétés telles que l'existence d'opérations de services, la gestion correcte des exceptions, ou l'existence de session au sein d'un service. Nous avons également étudié le test de robustesse de services sans état (stateless) ou avec états (stateful). Par rapport aux travaux existants, nous avons notamment pris en compte l'environnement extérieur des services Web, à savoir la couche SOAP (protocole SOAP, processeurs SOAP) qui modifie les réactions observées d'un service sous test. Ces travaux ont donné lieu a un outil de test, appelé WSAT, qui est détaillé et accessible à cette page `http://sebastien.salva.free.fr`.
- Test de sécurité de services à états (stateful) : plusieurs rapport ont montré la vulnérabilité des services Web face aux attaques telles que "XML injection", ou la "brute force". De ce fait, nous avons mis au point une méthode active permettant de tester si un service Web stateful possède des vulnérabilités. A partir de celles décrites par l'organisation OWASP, nous avons modélisé quelques patrons de test pour vérifier les propriétés de disponibilité, d'authentification, et d'intégrité. En combinant ces patrons de test avec une spécification initiale, nous obtenons des cas de test qui peuvent être expérimentés sur un service Web déployé. Nous avons implanté un outil partiel permettant de générer ces cas de test et de les exécuter. Nous avons appliqué cette méthode sur une centaine de services à partir d'un nombre de patrons de test restreints et avons détecté que 11% d'entre eux possèdent des vulnérabilités. Cette expérimentation montre non seulement les besoins à tester mais également que notre méthode est fonctionnelle.

Afin de bien cerner les problématiques à tester en boite noire des services et des compositions, nous avons étudié leur faisabilité de test, autrement appelée testabilité. Bien que simple au premier abord, un service Web stateless n'est accessible qu'à travers une couche SOAP-XML qui réduit sa visibilité (observabilité et contrôlabilité). De plus, un service est connu en interprétant et en faisant confiance à sa description WSDL. Nous avons montré que ces langages et protocoles réduisent de façon notable la testabilité. Nous avons donc étudié cette testabilité en considérant différents aspects comme le code du service, les messages SOAP qui permettent d'interagir avec le service, la description WSDL qui fournit son interface et sa spécification UML. A partir des critères d'observabilité et de contrôlabilité, nous avons montré quelles sont les propriétés qui réduisent la testabilité et donnons les définitions d'un service Web observable, contrôlable et testable.

Nous avons également étudié les compositions de services Web et leur testabilité à travers les deux critères d'observabilité et de contrôlabilité, ceci dans le but de comprendre et d'appréhender les difficultés à modéliser et tester des compositions. Nous avons choisi de modéliser les compositions grâce au langage ABPEL (Abstract Business Process Execution Language) qui permet de créer des processus abstraits d'orchestration de services. Ce langage offre une multitude de mécanismes pour la composition (partenaires, réception

multiple, corrélation, gestion des exceptions, etc.) et est utilisé dans plusieurs méthodes de test. Nous avons effectué une étude préliminaire de composition ABPEL pour déterminer une solution d'estimation du niveau de testabilité. Nous avons choisi de transformer une spécification ABPEL en STS (Symbolic Transition System) qui est un modèle de plus bas niveau et connu. De façon intuitive, nous avons listé l'ensemble des activités et pour chacune nous avons donné une règle d'inférence permettant de la transformer en transitions STS. Nous avons défini plusieurs propriétés de dégradation de testabilité et avons finalement déduit une mesure en recherchant le nombre de dégradations existantes dans le STS. Dans un second travail, nous avons proposé des algorithmes permettant de transformer automatiquement un processus ABPEL en un code plus testable en détectant, au préalable, les dégradations de testabilité dans un STS intermédiaire.

Les problématiques rencontrées sont différentes de celles des systèmes temporisés. En effet, les services Web et compositions sont souvent déployés dans des environnements spécialisés (environnements SOAP) qui modifient de façon substantielle les possibilités de test. L'étude de la testabilité nous a montré les restrictions liées à cet environnement. Les modèles utilisés, pour décrire formellement ces services, sont symboliques afin de prendre en compte les notions de paramètres et de réponses typés. Par nature, ce modèle implique un nombre parfois infini de valeurs à tester. Nous avons donc du prendre en compte cette difficulté pour restreindre la génération des cas de test.

4.1.4 Test de systèmes réactifs

A la différence des applications précédentes (services Web, compositions, protocoles), les systèmes réactifs sont des programmes qui réagissent à la vitesse de l'environnement, de façon synchrone et souvent déterministe.

A travers une collaboration avec le Labri, nous avons proposé quelques approches de test de robustesse de systèmes réactifs modélisés par des IOLTS (LTS dont l'alphabet sépare les actions d'entrée des actions de sortie) [RS08, RS09].

Une première approche considère des *aléas externes*, c'est à dire des évènements externes non attendus comme une action d'entrée non spécifiée. Les actions spécifiées du système et les aléas sont classés dans un tableau, qui sera utilisé pour construire les cas de test. Celui-ci décrit la spécification combinée à tous les aléas externes. La méthode rempli ce tableau et génère les cas de test à partir de ce dernier.

Une seconde approche complémentaire suppose qu'en cas de réception d'aléas, le système se place en phase dégradée où seules les actions vitales sont gardées. Ceci implique qu'un système est modélisé par une spécification nominale S et une spécification dégradée S'. Une relation de test $\leq_{rob}$ est premièrement définie sur les IOLTS pour exprimer l'inclusion de comportement entre deux IOLTS. En supposant que l'implantation peut être modélisée par un IOLTS, l'objectif de la méthode est de vérifier si son comportement est inclus dans celui de la spécification nominale. Une suite de test est alors dérivée à partir de la spécification nominale, puis des aléas sont injectés dans les cas de test. Seules les actions n'appartenant pas à la spécification dégradée S' peuvent être modifiées.

Une dernière approche prend en considération la relation de conformité *ioco* pour vérifier si l'implantation est robuste. Ceci offre l'avantage de montrer que les cas de test obtenus sont *sound* et *complete*. La spécification initiale est rendue déterministe, est complétée pour prendre en compte la notion de quiescence, les comportements incorrect, etc.

Puis, nous vérifions que l'implantation résultat est ioco-équivalente au modèle obtenu.

4.1.5 Test de composition dans des environnements partiellement ouverts

Récemment, je me suis focalisé sur le test de composants et de compositions dans des environnements dits partiellement ouverts que ce soit pour le test actif ou passif [Sal11e, Sal11c, Sal11a]. Les environnements, dans lesquels sont déployés des applications ou des systèmes, sont très souvent considérés comme étant ouverts, c'est à dire que le système à tester est accessible. Cependant, on peut remarquer, dans les tendances informatiques actuelles, que l'accès aux environnements des systèmes ou logiciels peut être restreint.

Ces environnements posent plusieurs problématiques. J'ai commencé à répondre à certaines d'entre elles en présentant deux approches préliminaires sur le test de conformité d'applications dans des environnements partiellement ouverts. La première approche concerne le test de compositions statiques et a pour objectif de retrouver les traces d'une composition à partir des traces des composants en décomposant des cas de test existants puis en recomposant les traces partielles obtenues. La deuxième approche cible le test de conformité effectué de façon passive. Habituellement, ce type de test est effectué par des outils dérivés d'un sniffeur (TCP, XML, etc.). Cette approche définit un proxy-testeur, qui correspond à un intermédiaire entre le trafic client et l'implantation.

D'autres travaux seront proposés sur ce thème à travers le projet de Bourse d'Innovation que nous commençons avec la société Openium sur le test des applications mobiles.

4.1.6 Autre

Notre équipe a également participé au projet RNRT Platonis (`http://www-lor.int-evry.fr/platonis/resume.html`), qui avait pour objectif le développement et la mise en place d'une plate-forme de validation et d'expérimentation multi-protocoles et multi-services. Cette plate-forme vise le test d'interfonctionnement des services intégrant la mobilité et l'Internet (WAP), ainsi que les technologies qui permettent la mise en place de ces services (UMTS, GPRS-GSM). Ce projet a notamment donné lieu à une publication qui décrit une architecture de test distribuée permettant le test d'une application WAP [LS04].

Nous nous sommes penchés sur la modélisation et le test de conformité des applications Ajax dans [SL09]. Les applications Ajax sont souvent composées d'une application client javascript et d'applications serveur. La première invoque une application serveur, après qu'un évènement soit déclenché dans un navigateur Web, puis modifie le document lu par le navigateur après avoir reçu une réponse. Nous avons choisi de les modéliser par des ensembles de diagrammes de séquence UML décrivant l'interaction utilisateur, les messages émis entre le client et le serveur et les modifications obtenues dans le document. Ces diagrammes sont ensuite transformés en spécification STS et en objectifs de test. Des cas de test sont ensuite construits vis-à-vis de la relation de test *ioco*. Ces cas de test sont expérimentés grâce à une architecture de test distribuée et hétérogène (sniffeurs, testeur javascript, etc.) Ce travail a donné lieu à un outil complet de modélisation et de test, dont l'avantage est qu'il ne nécessite pas d'intervention humaine. Cet outil est disponible à cette page `http://sebastien.salva.free.fr`.

Une collaboration avec le ministère des finances et le laboratoire LRI a donné lieu à une solution de parallélisation d'invocations de Services Web par un ensemble de clients [SBD07]. Nous supposons qu'il y a un flot de requêtes clientes de services sur un même serveur. Nous avons analysé plusieurs façons possibles pour paralléliser l'exécution de ces requêtes en commençant par une approche naïve qui consiste à créer un pool de tâches pour effectuer des invocations complètes en parallèle. Nous avons montré que cette granularité n'offre pas les meilleures performances surtout si le temps d'appel au service est long. Nous avons alors proposé de découper une invocation de service en plusieurs tâches : sérialisation des données, appel d'un service, désérialisation et stockage en base de donnée. Puis, nous avons proposé une solution utilisant le paradigme du pipeline. Nous avons implanté cette méthode grâce à l'API JOMP, basée sur OPEN-MP. Nous avons effectué plusieurs expérimentations sur une partie du projet Copernic développé par le Ministère des Finances.Nous avons comparé l'approche naïve utilisant un pool de tâches avec notre solution pour des temps d'appels de services et un nombre de threads variés. Nous avons considéré 100 appels de services avec 4000 objets à sérialiser/désérialiser pour des temps d'appel durant 1s, 5s et 10s. Pour des temps d'appel faibles (1s) nous avons montré que notre solution offre des performances 10 % supérieures à l'approche naïve du pool. Les performances augmentent de 60 % lorsque les temps d'appel sont de 10s.

4.1.7 Contrats de recherche et coopérations industrielles

- **Projet RNRT Platonis**,
  - date :2002-2004,
  - Objectif : Développement d'une plate-forme de validation et d'expérimentation multi-protocoles et multi-services (WAP sur GPRS et UMTS). Celle-ci a servie également à valider et expérimenter des protocoles et des services liés à la mobilité des terminaux du réseau. En particulier, autour du WAP et UMTS, cette plate-forme permet de vérifier que les connexions à tous les niveaux protocolaires se passent correctement et, également, que des entités différentes peuvent interagir comme prévu,
  - Partenaires : INT Évry, France Telecom R&D, LaBRI, Kaptech, LIMOS,
  - Rôle : participant
- **Projet ANR Webmov**,
  - date : 2007-2010,
  - Objectifs : L'objectif principal de WebMoV est de contribuer à la conception, à la composition et la validation de services Web à travers une vue abstraite de haut niveau et d'une vision SOA (Architecture Orientée Services) basée sur une architecture logique. Dans ce domaine, les industriels construisent les nouveaux services en composant des modules préexistants. Ces mécanismes de composition sont connus sous le nom d'orchestration,
  - Partenaires : LRI, GET/INT, LaBRI, Unicamp (brésil), SOFTEAM, Montimage, LIMOS, LRI,
  - Rôle : coordinateur LIMOS
- **Projet Bourse d'Innovation Thèse**,
  - date : 2011-2014
  - Objectif : Sécurité des transactions sur téléphones mobiles. Ce projet a pour but de

contribuer à la découverte et à la modélisation des vulnérabilités ou attaques qui peuvent être rencontrées sur des applications pour systèmes mobiles (téléphones, tablettes, etc.) et de définir des méthodes de test de sécurité afin de détecter ces vulnérabilités, les verrous technologiques se trouvant essentiellement centrés dans la partie exécution des tests et architecture de test. Les méthodes de test pourront être actives (exécution de tests) ou passives (monitoring), la combinaison des deux types offrant souvent de bon résultats. Celles-ci doivent s'appuyer sur les critères génériques de sécurité, à savoir la disponibilité, l'authentification, l'autorisation, ou l'intégrité.
  - Partenaires : LIMOS, société Openium, conseil régional
  - Rôle : coordinateur du projet.
- **Projet dans le cadre de la fédération TIMS (Technologie de l'Information de la Mobilité et de la Sureté)**,
  - date : 2007
  - Objectif : Évaluation de la robustesse des réseaux de capteurs sans fil dédiés aux applications industrielles,
  - Rôle : Coordinateur du projet

4.1.8 Encadrement

Thèse

J'ai co-encadré ou co-encadre les trois thèses suivantes avec Michel Misson (Prof.) et Patrice Laurençot (MdC).

- **Issam Rabhi**
  Septembre 2008 - janvier 2012,
  Sujet : Testabilité des services Web,
  Taux : 70 %
  Soutenue le 9 janvier 2012,
  7 publications,

- **Najib Mekouar**
  Janvier 2009, janvier 2011
  Sujet : Étude de la validation de chorégraphies de services Web,
  Taux : 90%
  Inscription en 3ème année non renouvelée suite aux manques de résultats,

- **Stassia Resondry Zafimiharisoa**
  Novembre 2011,
  Sujet : Sécurité des transactions et applications sur téléphones mobiles,
  Taux : 70 %

Master Recherche

- Mars 2008 - juillet 2008, Étude de la conformité des services Web, étudiant : S. Rihawi,
- Mars 2007 - juillet 2007, Validation d'applications Internet basées sur Ajax, étudiant : B. Boussad,
- Mars 2004 - Juillet 2004, Localisation d'erreurs dans le test de systèmes temporisés, étudiant : Y. Haydersah.

4.1.9 Rayonnement scientifique

- Participation à des comités de lecture :
  - 3rd International Conference on Engineering and Meta-Engineering (ICEME), 2012,
  - IEEE International Conference on Web Services (ICWS), 2011,
  - IEEE 15th International Conference on Automation, Quality and Testing, Robotics (AQTR), 2010,
  - 7th International Conference on Principles of Distributed Systems (OPODIS), 2003,
  - ACIS Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2002, 2000.

- Participation à des comités de programme de conférence :
  - International Conference on Internet and Web Applications and Services (ICIW), 2010, 2011, 2012,
  - International Conference on Communication Theory, Reliability, and Quality of Service (CTRQ), 2010, 2012,
  - International Conference on Software Engineering Advances (ICSEA), 2012,
  - Workshop Sensor Networks, Theory and Applications for Environmental Issues S(eNTApE), 2010.

- Participation au comité de rédaction du Journal International Journal On Advances in Internet Technology (IJAIT), 2012,

- Participation au comité de rédaction du Journal International Journal of Computing and Information Sciences (IJCIS), 2011,

- Participation au comité de lecture du Journal *Software and Systems Modeling* (SOSYM) Elsevier, 2011,

- Organisateur des journées du groupe de travail MTVV (GDR GPL), 27-28 octobre 2011, Rennes,

- Membre de la commission de spécialiste 27ème Section, Université d'Auvergne de 2004 à 2008 et 2011, école Polytech Clermont 2012,

- Participation aux GDR RGE (Réseau Grand Est, action transversale géographique

du GDR ASR), StrQds (Système Temps Réel et Qualité de Service), MTVV (méthodes de test pour la validation et la vérification),

– Présentation lors de séminaires : Séminaire Labri Bordeaux, 2010, Séminaire Labri Bordeaux, 2007 ; Séminaire Llaic 2007 : Etude du test de conformité de systèmes ; Séminaire LIFC 2002 : Etude de la qualité de test des systèmes temporisés.

4.1.10 Publications

Revues nationales et internationales

[BFPS01] Simon Bloch, Hacène Fouchal, Eric Petitjean, and Sébastien Salva. Some issues on testing real-time systems. In *International Journal in Computer Information Science (IJCIS)*, number 4. ACIS, 12 2001.

[Sal11a] Sébastien Salva. An approach for testing web service compositions when internal messages are unobservable. In Electronic Business Management Society, editor, *International Journal of Electronic Business Management (IJEBM)*, volume 9, pages 334–344, 2011.

[Sal11b] Sébastien Salva. Modelling and testing of service compositions in partially open environments. In Hermann, editor, *Studia Unformatica Universalis, special issue on "Modélisation informatique et mathématique des systèmes complexes : avancées méthodologiques"*, volume ?, page ?, 10 2011. submitted : 2011 march accepted : 2011 september.

[Sal11c] Sébastien Salva. Passive testing with proxy tester. In Science & Engineering Research Support soCiety (SERSC), editor, *International Journal of Software Engineering and Its Applications (IJSEIA)*, volume 5, pages 1–16, 10 2011.

[SF04] Sébastien Salva and Hacène Fouchal. Testability analysis for timed systems. In *International Journal of Computers and Their Applications (IJCA)*, number 1, 05 2004.

[SR08] Sébastien Salva and Antoine Rollet. Testabilité des services web. In Hermes Lavoisier, editor, *Ingénierie des Systèmes d Information RSTI série ISI, numéro spécial Objets, composants et modèles dans l ingénierie des SI*, volume 13, pages 35–58, 06 2008.

[SR11] Sébastien Salva and Antoine Rollet. A pragmatic approach for testing stateless and stateful web service robustness. In Hermann, editor, *Studia Informatica Universalis*, volume 10, page ?, 10 2011. submitted : 2010 october accepted : 2011 september.

Chapitre de livre

[Sal12] Sébastien Salva. A guided web service security testing method. In *Emerging Technologies - Innovative Concepts and Applications*, chapter 10. Intech, ISBN=979-953-307-622-4, 27 pages, 2012.

Conférences internationales

[CCM+10] Ana Cavalli, Tien-Dung Cao, Wissam Mallouli, Eliane Martins, Andrey Sadovykh, Sébastien Salva, and Fatiha Zaidi. Webmov : A dedicated framework for the modelling and testing of web services composition. In IEEE computer society press, editor, *ICWS 2010 - 8th IEEE International Conference on Web Services*, pages 377–384, Miami, USA, 07 2010.

[FPS00] Hacène Fouchal, Eric Petitjean, and Sébastien Salva. Timed testing using test purposes. In IEEE computer society press, editor, *7th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA00)*, pages 166–171, Cheju Island, South Korea, 12 2000.

[FPS01] Hacène Fouchal, Eric Petitjean, and Sébastien Salva. A user-oriented testing of real time systems. In IEEE computer society press, editor, *IEEE/IEE Real-Time Embedded Systems Workshop (RTES), satellite of RTSS*, London, UK, 12 2001.

[LS04] Patrice Laurençot and Sébastien Salva. Testing mobile and distributed systems : method and experimentation. In Springer Verlag, editor, *8th International Conference on Distribued Systems (OPODIS)*, LNCS 3544/2005, Grenoble, France, 12 2004.

[RS08] Antoine Rollet and Sébastien Salva. Two complementary approaches to test robustness of reactive systems (invited paper). In IEEE society press, editor, *17th IEEE International Conference on Automation, Quality and Testing, Robotics AQTR 2010*, pages 47–53, Cluj-Napoca, Romania, 05 2008.

[RS09] Antoine Rollet and Sébastien Salva. Testing robustness of communicating systems using ioco-based approach. In IEEE society press, editor, *Proceedings of In 1st IEEE Workshop on Performance evaluation of communications in distributed systems and Web based service architectures, in conjunction with IEEE ISCC 2009*, number 1530-1346, pages 67–72, Sousse, Tunisia, 07 2009.

[Sal02] Sébastien Salva. Testing temporal and behavior events on timed systems with timed test purposes. In Studia Informatica Universalis, editor, *6th International Conference on Distribued Systems (OPODIS)*, 12 2002.

[Sal11a] Sébastien Salva. Automatic test purpose generation for web services. In Xiaofeng Wan, editor, *Electrical Power Systems and Computers*, volume 99 of *Lecture Notes in Electrical Engineering*, pages 721–728. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-21747-0_92.

[Sal11b] Sébastien Salva. An observability enhancement method of abpel specifications. In *The 2nd International Conference on Engineering and Meta-Engineering : ICEME 2011*, Orlando, Florida USA, 03 2011.

[SBD07] Sébastien Salva, Cédric Bastoul, and Clément Delamare. Web service call parallelization using openmp. In Springer Verlag, editor, *3rd INTERNATIONAL WORKSHOP on OpenMP (IWOMP) 2007*, volume 4935/2008 of *LNCS*, pages 185–194, Tsinghua University, Beijing, China, 06 2007.

[SF01a] Sébastien Salva and Hacène Fouchal. Some parameters for timed system testability. In IEEE computer society press, editor, *ACS/IEEE International*

Conference on Computer System and Applications (AICCS01), pages 335–, Beirut, Lebanon, 06 2001.

[SF01b] Sébastien Salva and Hacène Fouchal. Timed test execution and ttcn generation. In ACIS, editor, *2nd International Conference on Software Engineering applied to Networking and Parallel/Distributed Computing (SNPD02)*, Nagoya, Japon, 08 2001.

[SFB00] Sébastien Salva, Hacène Fouchal, and Simon Bloch. Metrics for timed systems testing. In *4th International Conference on Distribued Systems (OPODIS)*, Paris, France, 12 2000.

[SL03] Sébastien Salva and Patrice Laurençot. A testing tool using the state characterization approach for timed systems. In *WRTES, satellite workshop of FME symposium*, Pisa, Italy, 09 2003.

[SL07] Sébastien Salva and Patrice Laurençot. Generation of tests for real-time systems with test purposes. In *15th International Conference on Real-Time and Network Systems RTNS07*, pages 35–44, Nancy, France, 03 2007.

[SL09] Sébastien Salva and Patrice Laurençot. Automatic ajax application testing. In IEEE computer society press, editor, *Fourth International Conference on Internet and Web Applications and Services, ICIW 2009*, pages 229–234, Venice/Mestre, Italy, 05 2009.

[SLR10] Sébastien Salva, Patrice Laurencot, and Issam Rabhi. An approach dedicated for web service security testing. In IEEE computer society press, editor, *5th International Conference on Software Engineering Advances, International Conference, ICSEA10*, pages 494–500, Nice, France, 08 2010.

[SPF01] Sébastien Salva, Eric Petitjean, and Hacène Fouchal. A simple approach for timed system testing. In *Formal Approaches to Testing of Software (FATES01), A Satellite Workshop of CONCUR01*, Aalborg, Denmark, 08 2001.

[SR08] Sébastien Salva and Antoine Rollet. Automatic web service testing from wsdl descriptions. In *8th International Conference on Innovative Internet Community Systems I2CS 2008*, volume 2011 of *Lecture Notes in Informatics (LNI)*, Schoelcher, Martinique, 06 2008.

[SR09a] Sébastien Salva and Issam Rabhi. Automatic web service robustness testing from wsdl descriptions. In *12th European Workshop on Dependable Computing, EWDC 2009*, Toulouse, France, 05 2009.

[SR09b] Sébastien Salva and Antoine Rollet. Test purpose generation for timed protocol testing. In IEEE computer society press, editor, *Proceedings of the 2009 Second International Conference on Communication Theory, Reliability, and Quality of Service, CTRQ 2009*, pages 8–14, Colmar, France, 07 2009.

[SR10a] Sébastien Salva and Issam Rabhi. A bpel observability enhancement method. In IEEE computer society press, editor, *Proceedings of the 2010 8th IEEE International Conference on Web Services, ICWS 2010*, pages 638–639, Miami, USA, 07 2010.

[SR10b] Sébastien Salva and Issam Rabhi. A preliminary study on bpel process testability. In IEEE computer society press, editor, *QuomBat2010, ICST Workshop*

on Quality of Model-Based Testing, Co-located with ICST 2010, pages 62–71, Paris, France, 04 2010.

[Sr10c] Sébastien Salva and Issam rabhi. Statefull web service robustness. In IEEE computer society press, editor, *The Fifth International Conference on Internet and Web Applications and Services, ICIW10*, pages 167–173, Barcelona, Spain, 05 2010.

[SR11] Sébastien Salva and Issam Rabhi. A test purpose and test case generation approach for soap web services. In XPS (Xpert Publishing Services), editor, *The Sixth International Conference on Software Engineering Advances ICSEA 2011*, barcelona, spain, 10 2011.

[SRF02] Sébastien Salva, Antoine Rollet, and Hacène Fouchal. Temporal and behavior characterization of states in timed systems. In ACIS, editor, *23rd ACIS Annual International Conference on Computer and Information Science (ICIS02)*, Seoul, South Korea, 08 2002.

Conférences nationales

[Sal01] Sébastien Salva. La qualité du test de conformité des systèmes temps réels. In *3eme Colloque sur la Modélisation et Simulation des Systèmes (MOSIM01)*, Troyes, France, 04 2001.

[SF02] Sébastien Salva and Hacène Fouchal. Une méthode de test des systèmes temporisés orientée objectif de test. In *RENPAR14*, Hamamet, Tunisie, 04 2002.

[SL03] Sébastien Salva and Patrice Laurençot. Génération de tests temporisés orientée caractérisation d états. In Lavoisier, editor, *Colloque Francophone de l ingénierie des Protocoles (CFIP)*, Paris, France, 12 2003.

[SL05] Sébastien Salva and Patrice Laurençot. Génération automatique d objectifs de test pour systèmes temporisés. In Lavoisier, editor, *Colloque Francophone de l ingénierie des Protocoles (CFIP)*, Bordeaux, France, 04 2005.

[SR10] Sébastien Salva and Issam Rabhi. Robustesse des services web persistants. In *MOSIM10, 8ème ENIM IFAC Conférence Internationale de Modélisation et Simulation*, Hammanet, Tunisy, 05 2010.

Communications

P10 **S. Salva** Perspectives sur le test de composants et compositions, Séminaire Labri Bordeaux,

P07 **S. Salva** Test automatique de services Web, Séminaire Labri Bordeaux,

P07b **S. Salva** Etude du test de conformité de systèmes, Séminaire Llaic Clermont-Ferrand,

P02 **S. Salva** Etude de la qualité de test des systèmes temporisés, Séminaire LIFC, Strasbourg,

P00 **S. Salva** Etude des méthodes de test temporisées et qualité de test des systèmes temporisés GDR RGE

4.2 Enseignement

Mes enseignements s'effectuent depuis 2002 à titre principal au département d'Informatique de l'Institut Universitaire Technologique d'Aubière et concernent essentiellement les élèves de DUT et de licence professionnelle.

Depuis septembre 2006, je suis responsable de la licence professionelle *Dévelopement Intranet/Internet* et co-responsable de la licence professionnelle *Informatique Répartie* en alternance. A ce titre, je m'occupe de la définition et de la coordination des enseignements, de la présidence des jurys, de la composition des dossiers administratifs, etc.

4.2.1 Responsabilités

- **Responsable des stages au département Informatique**, 2006-2010 : J'ai assumé la mise en place des stages au département (choix des stages, répartition des sujets entre options, relances). J'ai également pris des contacts avec plusieurs partenaires locaux,
- **Responsable de la formation Licence Professionnelle Développement D'application Intranet/Internet** depuis 2006, et co-responsable de la licence professionnelle *Informatique Répartie* en alternance. J'ai effectué le montage du dossier, la recherche de partenaires professionnels. Et je m'occupe entre autre de la définition et la mise à jour des contenus, de la gestion des projets, de la coordination des enseignants, des jurys, des emplois du temps, des dossiers Aeres, etc.,
- **Membre du conseil du département d'informatique** de l'IUT d'Aubière depuis 2004, membre élu.

4.2.2 Cours

Mon service total par an est d'environ 300 HTD (Heures équivalent TD). L'ensemble des cours que j'ai effectué est décrit ci-dessous. Je suis le responsable de tous les cours cités ci-dessous (définition des cours, TD, TP), excepté pour le cours d'architecture des ordinateurs.

2002 - 2011 IUT d'Aubière

DUT : Cours, TD, TP en réseaux (réseaux locaux, IP, TCP/UDP, etc.) et programmation Web (PHP, servlet, JSP, design pattern), projets tuteurés, stages.

Intitulé	Heures	Années
Réseaux (OSI, Ethernet, IP, routage, TCP, NAT, etc.) 1ère année DUT Description des couches réseaux, des algorithmes de routage. En TP, administration Linux (ipatables, apache, FTP, wireshark, SSH, etc.) voir le WIkibook `http://fr.wikibooks.org/wiki/Administration_r\%C3\%A9seau_sous_Linux`	15H CM, 28H TD, 28HTP	2002- ?
Architecture, 1ère année DUT Machine de Von Neumann, UAL, codage assembleur, etc.	14H TD, 28HTP	2002-2004
Réseaux/Web avancé, 2ème année DUT Cours sur le Wifi, xDSL, prog. Socket. Cours Web avancé : Servlet/JSP, utilisation de design pattern (MVC, sigleton, factory, etc.) ORM(BDO), injection SQL, etc.	28H CM, 56H TP	2004- ?

Licence professionnelle : Cours, TD, TP en réseaux, introduction à la programmation parallèle (OPENMP), services Web (Java AXIS), Qualité logicielle (norme CMMI, analyse UML, tests, etc.), Administration système, préparation à la certification SUN, projets tuteurés, stages.

Intitulé	Heures	Années
Réseaux : Cours sur les couches basses, Wifi, Ethernet, SDH/SONET,etc.	10H CM, 10H TD	2002- ?
Qualité logicielle : modèle CMMI, cycle de vie du logiciel, spécification formelles, tests, etc. TP sur JUNIT, fuites mémoires, SONAR	20H CM/TD	2007- ?
Services Web : Définition des Services Web SOAP, utilisation du framework Axis2, aperçu sur BPEL	8H CM,12H TP	2007- ?
Cloud Computing : présentation de l'architecture du Cloud computing (SaaS, PaaS), présentation de Windows Azure et Google App Engine, TP sur Google App Engine	4H CM, 2H TP	2010- ?
Prog. Système et Parallèle OPENMP (Processus, Thread, langage OPENMP, programmation à mémoire partagée multi-coeurs,...)	16H CM/TD, 10H TP	2003- ?
Préparation certification SUN (JAVA, niveau Programmer et plus)	>10H TD	2007- ?

Mise en place de partenariats avec plusieurs entreprises (Atos-Origin, Prizee, La Montagne, Michelin, etc.) sur la mise en place de cours, projets, stages,etc.

1998- 2002 Université de Reims

- Licence d'informatique : TD, TP de réseaux, architecture, programmation Web (PHP), programmation objet (C++), FDT

– DESS : programmation orientée objet, programmation système

Index

Bibliographie

[ACC+04] Baptiste Alcalde, Ana R. Cavalli, Dongluo Chen, Davy Khuu, and David Lee. Network protocol system passive testing for fault management : A backward checking approach. In de Frutos-Escrig and Núñez [dFEN04], pages 150–166.

[ACCK01] Joy Algesheimer, Christian Cachin, Jan Camenisch, and Gunter Karjoth. Cryptographic security for mobile code. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 2–, Washington, DC, USA, 2001. IEEE Computer Society.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[AdSSR09] Noureddine Adjir, Pierre de Saqui-Sannes, and Kamel Mustapha Rahmouni. Testing real-time systems using tina. In Núñez et al. [NBM09], pages 1–15.

[(Al11] Orange Business Service (Almerys). Sportmen health record management. 2011.

[Ama10] Amazon. Amazon e-commerce service. 2010. http ://docs.amazonwebservices.com/AWSEcommerceService/4-0/.

[AO08] P. AMMANN and J. OFFUTT. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[AW04] A. ABRAN and MOORE J. W. Guide to the software engineering body of knowledge. Washington, DC, USA, 2004. IEEE Computer Society.

[BBG07] Maurice H. Ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Formal methods for service composition. *Annals of Mathematics, Computing & Teleinformatics*, pages 1–10, 2007.

[BBMP09a] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. Ws-taxi : A wsdl-based testing tool for web services. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 326–335, Washington, DC, USA, 2009. IEEE Computer Society.

[BCF03] I. Berrada, R. Castanet, and P. Felix. A formal approach for real-time test generation. In *WRTES, satellite workshop of FME symposium*, pages 5–16, 2003.

[BCNZ05] E. Bayse, A. Cavalli, M. Nunez, and F. Zaidi. A passive testing approach based on invariants : application to the wap. In *Computer Networks*, volume 48, pages 247–266. Elsevier Science, 2005.

[BDDD91] G.v. Bochmann, G. Das, R. Dssouli, and M. Dubuc. Fault Models in Testing. In *Proceedings of the International Workshop on Testing of Communicating Systems IWTCS'91*, 1991.

[BDS+07] Abdelghani Benharref, Rachida Dssouli, Mohamed Adel Serhani, Abdeslam En-Nouaary, and Roch H. Glitho. New approach for efsm-based passive testing of web services. In Petrenko et al. [PVTG07], pages 13–27.

[BDSG09] Abdelghani Benharref, Rachida Dssouli, Mohamed Adel Serhani, and Roch H. Glitho. Efficient traces' collection mechanisms for passive testing of web services. *Information & Software Technology*, 51(2) :362–374, 2009.

[BDTC05] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. Wsdl-based automatic test case generation for web services testing. In *SOSE '05 : Proceedings of the IEEE International Workshop*, pages 215–220, Washington, DC, USA, 2005. IEEE Computer Society.

[Ben94a] R. G. Bennetts. Progress in design for test : A personal view. In *IEEE Design an Test of Computers*, 1994.

[BFPT06] A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. Audition of web services for testing conformance to open specified protocols. In R. Reussner, J. Stafford, and C. Szyperski, editors, *Architecting Systems with Trustworthy Components*, number 3938 in LNCS, pages 1–25. Springer-Verlag, 2006.

[BK00] J.M. Bull and M.E. Kambites. JOMP, an OpenMP-like interface for Java. In *Proc. of the ACM 2000 Conf. on Java Grande, San Francisco*, pages 44–53, San Francisco, 2000.

[BK09] I. B. Bourdonov and A. S. Kossatchev. Systems with priorities : Conformance, testing, and composition. *Program. Comput. Softw.*, 35 :198–211, July 2009.

[BK11] I. B. Bourdonov and A. S. Kossatchev. Specification completion for ioco. *Program. Comput. Softw.*, 37 :1–14, January 2011.

[BP05] Antonia Bertolino and Andrea Polini. The audition framework for testing web services interoperability. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 134–142, 2005.

[BPZ09] Lina Bentakouk, Pascal Poizat, and Fatiha Zaïdi. A formal framework for service orchestration testing based on symbolic transition systems. *In TESTCOM/FATES 2009 - 21th IFIP International Conference on Testing of Communicating Systems, LNCS*, 5826/2009 :16–32, 2009.

[Bri87b] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal, editor, *Proceedings of the Eighth International Conference on Protocol Specification, Testing and Verification*. North-Holland, 1987.

[Bri88] E. Brinksma. *On the design of Extended LOTOS – a specification language for open distributed systems*. PhD thesis, Department of Computer Science, University of Twente, 1988.

[BW07] Achim D. Brucker and Burkhart Wolff. Test-sequence generation with holtestgen - with an application to firewall testing. In *In TAP 2007 : Tests And Proofs, LNCS*, pages 149–168. Springer-Verlag, 2007.

[CA92b] W. Chung and P. Amer. Improved on UIO Sequence Generation and Partial UIO Sequences. In Linn and Uyar [LU92].

[Caf07] J. Mc Caffrey. Automatisation de test ajax. In *MSDN magazin*, February 2007.

[CBML09] Ana Cavalli, Azzedine Benameur, Wissam Mallouli, and Keqin Li. A passive testing approach for security checking and its practical usage for web services monitoring. In *NOTERE 2009*, June 2009.

[CCBS05] F. Cuppens, N. Cuppens-Boulahia, and T. Sans. Nomad : A security model with non atomic actions and deadlines. In *Computer Security Foundations. CSFW-18 2005. 18th IEEE Workshop*, pages 186–196, June 2005.

[CCKS95] R. Castanet, C. Chevrier, O. Koné, and B. Le Saec. An Adaptive Test Sequence Generation Method for the User Needs. In *Proceedings of IWPTS'95, Evry, France*, 1995.

[CCM+10] Ana Cavalli, Tien-Dung Cao, Wissam Mallouli, Eliane Martins, Andrey Sadovykh, Sébastien Salva, and Fatiha Zaidi. Webmov : A dedicated framework for the modelling and testing of web services composition. In IEEE computer society press, editor, *ICWS 2010 - 8th IEEE International Conference on Web Services*, pages 377–384, Miami, USA, 07 2010.

[CGM03] Myra B. Cohen, Peter B. Gibbons, and Warwick B. Mugridge. Constructing test suites for interaction testing. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 38–48, 2003.

[CGP03] Ana Cavalli, Caroline Gervy, and Svetlana Prokopenko. New approaches for passive testing using an extended finite state machine specification. *Information and Software Technology*, 45(12) :837 – 852, 2003. Testing and Validation of Communication Software.

[Cho78] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3) :178–187, 1978.

[CJJ07] C. Constant, B. Jeannet, and T. Jéron. Automatic test generation from interprocedural specifications. In *TestCom/Fates07*, number 4581 in LNCS, pages 41–57, Tallinn, Estonia, June 2007.

[CM04] A. Cavalli and S. Maag. Automated test scenarios for an e-barter system. In *ACM SAC 2004, Nicosia, Cyprus*, 2004.

[CMdO09] Ana Cavalli, Stephane Maag, and Edgardo Montes de Oca. A passive conformance testing approach for a manet routing protocol. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 207–211, New York, NY, USA, 2009. ACM.

[cmm] Capability maturity model + integration. Accessed june 2011.

[CMMR08] Alessandro Castrucci, Fabio Martinelli, Paolo Mori, and Francesco Roperti. Enhancing java me security support with resource usage monitoring. In *Proceedings of the 10th International Conference on Information and Communications Security*, ICICS '08, pages 256–266, Berlin, Heidelberg, 2008. Springer-Verlag.

[CMP06] Carlos Canal, Juan Manuel Murillo, and Pascal Poizat. Software adaptation. In *L'objet, Special Issue on Coordination and Adaptation Techniques for Software Entities, Introduction paper by the editors of the special issue*, pages 9–31, 2006.

[COG98] R. Cardel-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. In *Proc. of the 5th. Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of Lecture Notes in Computer Science, pages 251–261. SpringerVerlag, 1998.

[Con03] World Wide Web Consortium. Simple object access protocol v1.2 (soap). June 2003.

[Con07] OASIS Consortium. Ws-bpel v2.0. April 2007. http ://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.

[CPY00] A. Cavalli, S. Prokopenko, and N. Yevtushenko. Fault detection power of a widely used test suite for a system of communicating fsms. *Proceeding Testing of Communicating Systems Tools and Techniques of IFIP TC6/WG6.1/ 13th International Conference on Testing of Communicating Systems TESTCOM 2000, Ottawa (Canada),*, page 34 à 56, 2000.

[CWYL07] Jerry Cheng, Starsky H.Y. Wong, Hao Yang, and Songwu Lu. Smartsiren : virus detection and alert for smartphones. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, MobiSys '07, pages 258–271, New York, NY, USA, 2007. ACM.

[DAdSS01a] K. Drira, P. Azema, and P. de Saqui Sannes. Testability analysis in communicating systems. *Computer Networks*, 36 :671–693, 2001.

[DBN08] Marlon Dumas, Boualem Benatallah, and Hamid R. Motahari Nezhad. H.r.m. : Web service protocols : Compatibility and adaptation. *IEEE Data Eng. Bull*, pages 40–44, 2008.

[dFEN04] David de Frutos-Escrig and Manuel Núñez, editors. *Formal Techniques for Networked and Distributed Systems - FORTE 2004, 24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004, Proceedings*, volume 3235 of *Lecture Notes in Computer Science*. Springer, 2004.

[DFG+06] V. Darmaillacq, J.C Fernandez, R. Groz, L. Mounier, and J-L. Richier. Test generation for network security rules. In *Testing of Communicating Systems (TestCom)*, volume 3964, pages 341–356. LNCS, Springer, 2006.

[Dij69] E.W. Dijkstra. Software engineering techniques. In *Nato, Conference*, page 21, Rome, Italia, 1969.

[Dir06] Direction Générale des Impôts. The copernic tax project. 2006. http ://en.wikipedia.org/wiki/Copernic_tax_project.

[DYZ06] Wen-Li Dong, Hang Yu, and Yu-Bing Zhang. Testing bpel-based web service composition using high-level petri nets. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*, pages 441–444, Washington, DC, USA, 2006. IEEE Computer Society.

[EGGC09a] Jose Pablo Escobedo, Christophe Gaston, Pascale Gall, and Ana Cavalli. Observability and controllability issues in conformance testing of web service compositions. In *TESTCOM '09/FATES '09 : Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, pages 217–222, Berlin, Heidelberg, 2009. Springer-Verlag.

[END03] A. En-Nouaary and R. Dssouli. A guided method for testing timed input output automata. In *15th IFIP International Conference, TestCom 2003, Sophia Antipolis, France*, pages 211–225, May 2003.

[ENDK02] A. En-Nouaary, R. Dssouli, and Ferhat Khendek. Timed wp-method : Testing real-time systems. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, November 2002.

[ENDKE98] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *19th IEEE Real Time Systems Symposium (RTSS'98)* Madrid, Spain [RTS98].

[ES03] Niklas Een and Niklas Sörensson. Minisat. 2003. http ://minisat.se.

[Evi11] Eviware. Soapui. 2011. http ://www.soapui.org/.

[FBK+91] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite-state models. *IEEE Transactions on Software Engineering*, 17(6) :591–603, June 1991.

[fir] SecTheory firm. Holes in webos smartphones. Accessed Feb 2011.

[FJJV96] J. Cl. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV'96. LNCS 1102 Springer Verlag*, 1996.

[FMP05] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. A model-based approach for robustness testing. In *TestCom'05*, pages 333–348, 2005.

[FPS00] Hacène Fouchal, Eric Petitjean, and Sébastien Salva. Timed testing using test purposes. In IEEE computer society press, editor, *7th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA00)*, pages 166–171, Cheju Island, South Korea, 12 2000.

[Fra] The Open Group Architecture Framework. Togaf introduction. Accessed 22 Jan 2009.

[Fre91] R. S. Freedman. Testability of software components. *IEEE transactions on Software Engineering*, 17(6), june 1991.

[FRT10] Hacène Fouchal, Antoine Rollet, and Abbas Tarhini. Robustness testing of composed real-time systems. *J. Comp. Methods in Sci. and Eng.*, 10 :135–148, September 2010.

[FSKC07] A. Rollet. F. Saad-Khorchef and R. Castanet. A framework and a tool for robustness testing of communicationg software. In *In 22nd annual ACM Symposium on Applied Computing (SAC'07)*. ACM Press, 2007.

[FTW05] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing – FATES 2004*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer, 2005.

[FW08] Gordon Fraser and Franz Wotawa. Using model-checkers to generate and analyze property relevant test-cases. *Software Quality Journal*, 16(2) :161–183, 2008.

[GAR05] S. GARFINKEL. Historys worst software bugs. Wired News, 2005. http ://www.wired. com/software/coolapps/news/2005/11/69355 (last visited : 15.09.2009).

[GFTdlR06] José García-Fanjul, Javier Tuya, and Claudio de la Riva. Generating test cases specifications for compositions of web services. In Antonia Bertolino and Andrea Polini, editors, *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe06)*, pages 83–94, Palermo, Sicily, ITALY, June 2006.

[GGRT06] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *TestCom*, pages 1–18, 2006.

[Gil62] A. Gill. *Introduction to the theory of finite-state machines*. Mc Graw-Hill, New York – USA, 1962.

[GL06] Nils Gruschka and Norbert Luttenberger. Protecting web services from dos attacks by soap message validation. In *in Proceedings of the IFIP TC11 21 International Information Security Conference (SEC)*, 2006.

[GO09] Leonard Gallagher and Jeff Offutt. Test sequence generation for integration testing of component software. *Comput. J.*, 52(5) :514–529, 2009.

[Gon80] G. Gonenc. A method for the design of fault detection experiment. *IEEE transactions on Computers*, C-19 :551–558, 1980.

[GTC+05] Wolfgang Grieskamp, Nikolai Tillmann, Colin Campbell, Wolfram Schulte, and Margus Veanes. Action machines - towards a framework for model composition, exploration and conformance testing based on symbolic computation. In *Proceedings of the Fifth International Conference on Quality Software*, QSIC '05, pages 72–82, Washington, DC, USA, 2005. IEEE Computer Society.

[HLU03a] O. Henniger, M. Lu, and H. Ural. Automatic generation of test purposes for testing distributed systems. In *FATES03 (Formal Approaches for Testing Software), Canada*, pages 185–198, October 2003.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30 :75–82, April 1997.

[IDC] IDC. Idc forecasts worldwide mobile applications revenues. Accessed December 2010.

[iHBvdRT03] ir. H.M. Bijl van der, Dr.ir. A. Rensink, and Dr.ir. G.J. Tretmans. Component based testing with ioco., 2003.

[ISO91a] ISO. Conformance Testing Methodology and Framework. International Standard 9646, International Organization for Standardization — Information Technology — Open Systems Interconnection, Genève, 1991.

[ISO09] ISO/IEC. Common Criteria for Information Technology Security (CC). In *ISO/IEC 15408, version 3.1, ISO/IEC 15408*, December 2009.

[IWP94] Proceedings of the 7th international Workshop on Protocol Test System IWPTS'94 (tokyo, japan). In *Proceedings of the 7th International Workshop on Protocol Test System IWPTS'94 (Tokyo, Japan)*, Amsterdam, september 1994. North-Holland.

[IWT96] Proceedings of the 9th international Workshop on Test of Communicating Systems IWTCS'96 (darmstadt, germany). In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *Proceedings of the 8th International Workshop on Test of Communicating Systems IWTCS'96 (Darmstadt, Germany)*, Amsterdam, september 1996. North-Holland.

[JÓ9] Thierry Jéron. Symbolic model-based test selection. *Electron. Notes Theor. Comput. Sci.*, 240 :167–184, July 2009.

[JMR06] T. Jéron, H. Marchand, and V. Rusu. Symbolic determinisation of extended automata. In Springer Science and Business Media, editors, *4th IFIP International Conference on Theoretical Computer Science,Santiago, Chile*, volume 209/2006, pages 197–212, 2006.

[KABT10] Bilal Kanso, Marc Aiguier, Frédéric Boulanger, and Assia Touil. Testing of abstract components. In *Proceedings of the 7th International colloquium conference on Theoretical aspects of computing*, ICTAC'10, pages 184–198, Berlin, Heidelberg, 2010. Springer-Verlag.

[KBM+03] Anas Abou El Kalam, Salem Benferhat, Alexandre Miège, Rania El Baida, Frédéric Cuppens, Claire Saurel, Philippe Balbiani, Yves Deswarte, and

Gilles Trouessin. Organization based access control. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '03, pages 120–, Washington, DC, USA, 2003. IEEE Computer Society.

[KDCK94] K. Karoui, R. Dssouli, O. Cherkaoui, and A. Khoumsi. Estimation de la testabilité d'un logiciel modélisé par les relations. 1994. research report #921.

[KGD96] K. Karoui, A. Ghedamsi, and R. Dssouli. A study of some influencing factors in testability and diagnostics based on fsms. 1996. rapport de recherche #1048.

[KGG+09] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi : a solver for string constraints. In *ISSTA '09 : Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116, New York, NY, USA, 2009. ACM.

[KJM03] A. Khoumsi, T. Jeron, and H. Marchand. Test cases generation for nondeterministic real-time systems. In *FATES03 (Formal Approaches for Testing Software), Canada*, October 2003.

[KKS98] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *FTCS '98 : Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, page 230, Washington, DC, USA, 1998. IEEE Computer Society.

[Kon01] O. Kone. A local approach to the testing of real time systems. *The computer journal*, 44 :435–447, 2001.

[Lar06] J. Larson. Testing ajax applications with selenium. In *InfoQ magazine*, 2006.

[Lau99] P. Laurençot. *Integration du temps dans les tests de protocoles de communication.* PhD thesis, Univ. of Bordeaux 1, January 1999.

[LC97] Patrice Laurencot and Richard Castanet. Integration of Time in Canonical Testers for Real-Time Systems. In *International Workshop on Object-Oriented Real-Time Dependable Systems, California.* IEEE Computer Society Press, 1997.

[LCH+06] David Lee, Dongluo Chen, Ruibing Hao, Raymond E. Miller, Jianping Wu, and Xia Yin. Network protocol system monitoring : a formal approach with passive testing. *IEEE/ACM Trans. Netw.*, 14 :424–437, April 2006.

[LLML10] Bin Lei, Zhiming Liu, Charles Morisset, and Xuandong Li. State based robustness testing for components. *Electron. Notes Theor. Comput. Sci.*, 260 :173–188, January 2010.

[LNS+97] D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *Proceedings of the 1997 International Conference on Network Protocols (ICNP '97)*, ICNP '97, pages 113–, Washington, DC, USA, 1997. IEEE Computer Society.

[LPvB94] G. Luo, A. Petrenko, and G. v. Bochmann. Selecting Test Sequences for Partially-specified Non-deterministic Finite State Machines. In *Proceedings of the 7th International Workshop on Protocol Test System IWPTS'94 (Tokyo, Japan)* [IWP94].

[LS04] Patrice Laurençot and Sébastien Salva. Testing mobile and distributed systems : method and experimentation. In Springer Verlag, editor, *8th International Conference on Distribued Systems (OPODIS)*, LNCS 3544/2005, Grenoble, France, 12 2004.

[LTMB07] Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. Testing security policies : going beyond functional testing. In *ISSRE'07 (Int. Symposium on Software Reliability Engineering)*, 2007.

[LU92] R.J. Linn and M.U. Uyar, editors. *Protocol Specification, Testing, and Verification, XII,* Lake Buena Vista, Florida, USA. North-Holland, June 1992.

[LX03] Nik Looker and Jie Xu. Assessing the dependability of soap rpc-based web services by fault injection. In *WORDS Fall*, pages 163–170, 2003.

[LZCH08a] M. Lallali, F. Zaidi, A. Cavalli, and I. Hwang. Automatic timed test case generation for web services composition. In IEEE Computer Society Press, editor, *The 6th IEEE European Conference on Web Services (ECOWS'08)*, pages 53–63, Dublin, november 2008.

[LZCH08b] Mounir Lallali, Fatiha Zaidi, Ana Cavalli, and Iksoon Hwang. Automatic timed test case generation for web services composition. In *Proceedings of the 2008 Sixth European Conference on Web Services*, pages 53–62, Washington, DC, USA, 2008. IEEE Computer Society.

[MA00] R.E. Miller and K.A. Arisha. On fault location in networks by passive testing. In *IPCCC'2000, Phoenix, USA*, ICNP '97, pages 281–287, 2000.

[Mar06] Evan Martin. Automated test generation for access control policies. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 752–753, New York, NY, USA, 2006. ACM.

[MBCB08] Wissam Mallouli, Fayçal Bessayah, Ana Cavalli, and Azzedine Benameur. Security Rules Specification and Analysis Based on Passive Testing. In IEEE, editor, *The IEEE Global Communications Conference (GLOBECOM 2008)*, December 2008.

[MFBT08] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry, and Yves Traon. A model-based framework for security policy specification, deployment and testing. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 537–552, Berlin, Heidelberg, 2008. Springer-Verlag.

[MIL80b] R. MILNER. *A Calculus of Communicating Systems.* Springer, Berlin / Heidelberg, Germany, 1980.

[Mil90a] A. Mili. Program fault tolerance. *Prentice Hall*, 1990.

[MMC09] Wissam Mallouli, Amel Mammar, and Ana Rosa Cavalli. A formal framework to integrate timed security rules within a tefsm-based system specification. In *16th Asia-Pacific Software Engineering Conference (ASPEC'09), Malaysia*, dec 2009.

[MMC+10] Gerardo Morales, Stéphane Maag, Ana R. Cavalli, Wissam Mallouli, Edgardo Montes de Oca, and Bachar Wehbi. Timed extended invariants for the passive testing of web services. In *ICWS'10*, pages 592–599, 2010.

[MOC+07] Wissam Mallouli, Jean-Marie Orset, Ana Cavalli, Nora Cuppens-Boulahia, and Frédéric Cuppens. A formal approach for testing security rules. In *12th ACM symposium on Access control models and technologies, SACMAT*, pages 127–132, 2007.

[MP09] Annapaola Marconi and Marco Pistore. Formal methods for web services. chapter Synthesis and Composition of Web Services, pages 89–157. Springer-Verlag, Berlin, Heidelberg, 2009.

[MSS+08] Divya Muthukumaran, Anuj Sawani, Joshua Schiffman, Brian M. Jung, and Trent Jaeger. Measuring integrity on mobile phone systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, SACMAT '08, pages 155–164, New York, NY, USA, 2008. ACM.

[MTR08a] A. Marchetto, P. Tonella, and F. Ricca. A case study-based comparison of web testing techniques applied to ajax web applications. In *Int J Software Tools Technologies Transfer*, number 10. Springer-Verlag, 2008.

[MTR08b] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *ICST '08 : Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 121–130. IEEE Computer Society, 2008.

[MY01] T. Margaria and W. Yi, editors. *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems,* Genova, Italy, volume 2031 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2001.

[Mye79] Glenford. J. Myers. *The Art of Software Testing.* John Wiley & Sons, Inc., 1979.

[NBM09] Manuel Núñez, Paul Baker, and Mercedes G. Merayo, editors. *Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, 2009. Proceedings*, volume 5826 of *Lecture Notes in Computer Science*. Springer, 2009.

[net] Jupiter networks. Smobile security analysis of over 48,000 android market applications. Accessed June 2010.

[Net09] NetBeans Framework. http ://www.netbeans.org/. 2009.

[NH83] Rocco De Nicola and Matthew Hennessy. Testing equivalence for processes. In *Proceedings of the 10th Colloquium on Automata, Languages and Programming*, pages 548–560, London, UK, 1983. Springer-Verlag.

[NS01a] Brian Nielsen and Arne Skou. Automated Test Generation from Timed Automata. In Margaria and Yi [MY01], pages 343–357.

[NT81a] S. Naito and M. Tsunoyama. Fault Detection for Sequential Machines by Transition- Tours. *Proceedings of Fault Tolerant Computer Systems*, pages 238–243, 1981.

[OAS04] OASIS consortium. Ws-security core specification 1.1. 2004. http ://www.oasis-open.org/committees/tc_home.php ?wg_abbrev=wss.

[org06] WS-I organization. Ws-i basic profile. 2006. http ://www.ws-i.org/docs/charters/WSBasic_Profile_Charter2-1.pdf.

[OST10] Olaf Owe, Martin Steffen, and Arild B. Torjusen. Model testing asynchronously communicating objects using modulo ac rewriting. *Electron. Notes Theor. Comput. Sci.*, 264 :69–84, December 2010.

[OWA03] OWASP. Owasp testing guide v3.0 project. 2003.

[OX04] Jeff Offutt and Wuzhi Xu. Generating test cases for web services using data perturbation. In Software Engineering Notes, editor, *ACMSIGSOFT*, volume 29(5), pages 1–10, 2004.

[Pel96] Jan Peleska. Test automation for safety-critical systems : Industrial application and future developments. In Marie-Claude Gaudel and James Woodcock, editors, *FME'96 : Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 39–59. Springer Berlin / Heidelberg, 1996.

[Pha94] M. Phalippou. *Relation d'implantation et hypothèses de test sur des automates à entrées et sorties*. PhD thesis, Univ. of Bordeaux, September 1994.

[Phi87] I. Phillips. Refusal testing. *Theoretical Computer Science*, 50 :241–284, 1987.

[PLC98] O. Koné P. Laurencot and R. Castanet. On the Fly Test Generation for Real Time Protocols. In *International Conference on Comnputer Communications and Networks, Louisiane U.S.A*, 1998.

[PTD+05] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, Frank Leymann, and Bernd J. Krämer. Service-Oriented Computing : A Research Roadmap. In Francisco Curbera, Bernd J. Krämer, and Mike P. Papazoglou, editors, *Service Oriented Computing*, volume 05462 of *Dagstuhl Seminar Proceedings*, pages 1–29. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, May 2005.

[PVTG07] Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors. *Testing of Software and Communicating Systems, 19th IFIP TC6/WG6.1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007, Proceedings*, volume 4581 of *Lecture Notes in Computer Science*. Springer, 2007.

[PYvB96] A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Testing Deterministic Implementations from Non-deterministic FSM Specifications. In Baumgarten et al. [IWT96].

[RAD78] Frederick G. Sayward Richard A. DeMillo, Richard J. Lipton. Hints on test data selection : Help for the practicing programmer. *Computer*, 11(4) :34–41, April 1978.

[RMJ05] Vlad Rusu, Hervé Marchand, and Thierry Jéron. Automatic verification and conformance testing for validating safety properties of reactive systems. In John Fitzgerald, Andrzej Tarlecki, and Ian Hayes, editors, *Formal Methods 2005 (FM05)*, LNCS. Springer, July 2005.

[ROL03] Antoine ROLLET. Testing robustness of real-time embedded systems, September 2003.

[RS04] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004*, pages 43–54, 2004.

[RS08] Antoine Rollet and Sébastien Salva. Two complementary approaches to test robustness of reactive systems (invited paper). In IEEE society press, editor, *17th IEEE International Conference on Automation, Quality and Testing, Robotics AQTR 2010*, pages 47–53, Cluj-Napoca, Romania, 05 2008.

[RS09] Antoine Rollet and Sébastien Salva. Testing robustness of communicating systems using ioco-based approach. In IEEE society press, editor, *Proceedings of In 1st IEEE Workshop on Performance evaluation of communications in distributed systems and Web based service architectures, in conjunction with IEEE ISCC 2009*, number 1530-1346, pages 67–72, Sousse, Tunisia, 07 2009.

[RTS98] *19th IEEE Real Time Systems Symposium (RTSS'98)* Madrid, Spain, 1998.

[Sal02] Sébastien Salva. Testing temporal and behavior events on timed systems with timed test purposes. In Studia Informatica Universalis, editor, *6th International Conference on Distribued Systems (OPODIS)*, 12 2002.

[Sal11a] Sébastien Salva. An approach for testing web service compositions when internal messages are unobservable. In Electronic Business Management Society, editor, *International Journal of Electronic Business Management (IJEBM)*, volume 9, pages 334–344, 2011.

[Sal11b] Sébastien Salva. Automatic test purpose generation for web services. In Xiaofeng Wan, editor, *Electrical Power Systems and Computers*, volume 99 of *Lecture Notes in Electrical Engineering*, pages 721–728. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-21747-0_92.

[Sal11c] Sébastien Salva. Modelling and testing of service compositions in partially open environments. In Hermann, editor, *Studia Unformatica Universalis, special issue on "Modélisation informatique et mathématique des systèmes complexes : avancées méthodologiques"*, volume ?, page ?, 10 2011. submitted : 2011 march accepted : 2011 september.

[Sal11d] Sébastien Salva. An observability enhancement method of abpel specifications. In *The 2nd International Conference on Engineering and Meta-Engineering : ICEME 2011*, Orlando, Florida USA, 03 2011.

[Sal11e] Sébastien Salva. Passive testing with proxy tester. In Science & Engineering Research Support soCiety (SERSC), editor, *International Journal*

of Software Engineering and Its Applications (IJSEIA), volume 5, pages 1–16, 10 2011.

[SBC05] D. Senn, D. A. Basin, and G. Caronni. Firewall conformance testing. In *Testing of Communicating Systems (TestCom)*, volume 3502, pages 226–241. LNCS, Springer, 2005.

[SBD07] Sébastien Salva, Cédric Bastoul, and Clément Delamare. Web service call parallelization using openmp. In Springer Verlag, editor, *3rd INTERNATIONAL WORKSHOP on OpenMP (IWOMP) 2007*, volume 4935/2008 of *LNCS*, pages 185–194, Tsinghua University, Beijing, China, 06 2007.

[SD88b] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15 :285–297, 1988.

[sel] Selenium, a web application testing system. Accessed june 2011.

[SF04] Sébastien Salva and Hacène Fouchal. Testability analysis for timed systems. In *International Journal of Computers and Their Applications (IJCA)*, number 1, 05 2004.

[SL03b] Sébastien Salva and Patrice Laurençot. Génération de tests temporisés orientée caractérisation d états. In Lavoisier, editor, *Colloque Francophone de l ingénierie des Protocoles (CFIP)*, Paris, France, 12 2003.

[SL03c] Sébastien Salva and Patrice Laurençot. A testing tool using the state characterization approach for timed systems. In *WRTES, satellite workshop of FME symposium*, Pisa, Italy, 09 2003.

[SL05] Sébastien Salva and Patrice Laurençot. Génération automatique d objectifs de test pour systèmes temporisés. In Lavoisier, editor, *Colloque Francophone de l ingénierie des Protocoles (CFIP)*, Bordeaux, France, 04 2005.

[SL07] Sébastien Salva and Patrice Laurençot. Generation of tests for real-time systems with test purposes. In *15th International Conference on Real-Time and Network Systems RTNS07*, pages 35–44, Nancy, France, 03 2007.

[SL09] Sébastien Salva and Patrice Laurençot. Automatic ajax application testing. In IEEE computer society press, editor, *Fourth International Conference on Internet and Web Applications and Services, ICIW 2009*, pages 229–234, Venice/Mestre, Italy, 05 2009.

[SLD92] Y.N. Shen, F. Lombardi, and A.T. Dabuhra. Protocol Conformance Testing by Multiple uio Sequences. *IEEE Transactions on Communications*, 40 :1282–1287, 1992.

[SLR10] Sébastien Salva, Patrice Laurencot, and Issam Rabhi. An approach dedicated for web service security testing. In IEEE computer society press, editor, *5th International Conference on Software Engineering Advances, International Conference, ICSEA10*, pages 494–500, Nice, France, 08 2010.

[SP10] M. Singh and S. Pattterh. Formal specification of common criteria based access control policy. In *International Journal of Network Security*, pages 139–148, December 2010.

[SPLA07] Aubrey-Derrick Schmidt, Frank Peters, Florian Lamour, and Sahin Albayrak. Monitoring smartphones for anomaly detection. In *Proceedings of*

the 1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications, MOBILWARE '08, pages 40 :1–40 :6, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[SR] Sebastien Salva and Issam Rabhi. A preliminary study on bpel process testability. In *http ://sebastien.salva.free.fr/sr10.pdf*.

[SR08a] Sébastien Salva and Antoine Rollet. Automatic web service testing from wsdl descriptions. In *8th International Conference on Innovative Internet Community Systems I2CS 2008*, volume 2011 of *Lecture Notes in Informatics (LNI)*, Schoelcher, Martinique, 06 2008.

[SR08b] Sébastien Salva and Antoine Rollet. Testabilité des services web. In Hermes Lavoisier, editor, *Ingénierie des Systèmes d Information RSTI série ISI, numéro spécial Objets, composants et modèles dans l ingénierie des SI*, volume 13, pages 35–58, 06 2008.

[SR09a] Sébastien Salva and Issam Rabhi. Automatic web service robustness testing from wsdl descriptions. In *12th European Workshop on Dependable Computing, EWDC 2009*, Toulouse, France, 05 2009.

[SR10a] Sébastien Salva and Issam Rabhi. A bpel observability enhancement method. In IEEE computer society press, editor, *Proceedings of the 2010 8th IEEE International Conference on Web Services, ICWS 2010*, pages 638–639, Miami, USA, 07 2010.

[SR10b] Sébastien Salva and Issam Rabhi. A preliminary study on bpel process testability. In IEEE computer society press, editor, *QuomBat2010, ICST Workshop on Quality of Model-Based Testing, Co-located with ICST 2010*, pages 62–71, Paris, France, 04 2010.

[SR10c] Sébastien Salva and Issam Rabhi. Robustesse des services web persistants. In *MOSIM10, 8ème ENIM IFAC Conférence Internationale de Modélisation et Simulation*, Hammanet, Tunisy, 05 2010.

[SR11a] Sébastien Salva and Issam Rabhi. A test purpose and test case generation approach for soap web services. In XPS (Xpert Publishing Services), editor, *The Sixth International Conference on Software Engineering Advances ICSEA 2011*, barcelona, spain, 10 2011.

[SR11b] Sébastien Salva and Antoine Rollet. A pragmatic approach for testing stateless and stateful web service robustness. In Hermann, editor, *Studia Informatica Universalis*, volume 10, page ?, 10 2011. submitted : 2010 october accepted : 2011 september.

[SRF02] Sébastien Salva, Antoine Rollet, and Hacène Fouchal. Temporal and behavior characterization of states in timed systems. In ACIS, editor, *23rd ACIS Annual International Conference on Computer and Information Science (ICIS02)*, Seoul, South Korea, 08 2002.

[SVD01] J. Springintveld, F.W. Vaandrager, and P. R. D'Argenio. Testing Timed Automata. Technical Report 254, 2001.

[Tar41] A. Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6 :73–98, 1941.

[TARC99] M. Tabourier and M. Ionescu A. R. Cavalli. A gsm-map protocol experiment using passive testing. *World Congress on Formal Methods*, pages 915–934, 1999.

[Tid00] D. Tidwell. Web services, the web's next revolution. In *IBM developerWorks*, November 2000.

[Tre96a] J. Tretmans. Conformance testing with labelled transition systems : Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29 :49–79, 1996.

[Tre96c] J. Tretmans. Test generation with input, outputs, and repetitive quiescence. *Software–Concepts and Tools*, 17 :103–120, 1996.

[UL07] M. UTTING and B. LEGEARD. *Practical Model-Based Testing. A Tools Approach.* Morgan Kaufmann, San Francisco, CA, USA, 2007.

[Val08] Paul Valiant. Testing symmetric properties of distributions. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, STOC '08, pages 383–392, New York, NY, USA, 2008. ACM.

[VCI89] S. Vuong, W. Chan, and M. Ito. The UIOv-Method for Protocol Test Sequence Generation. In *2nd IWPTS International Workshop on Protocol Test Systems, Berlin*, 1989.

[VM95] J. M. Voas and K. W. Miller. Software testability : The new verification. In *IEEE Software*, May 1995.

[Wil] Doran K. Wilde. A library for doing polyhedral operations. Technical report, IRISA. http ://icps.u-strasbg.fr/PolyLib/.

[WPC01] Ye Wu, Dai Pan, and Mei-Hwa Chen. Techniques for testing component-based software. In *Proceedings of the Seventh International Conference on Engineering of Complex Computer Systems*, pages 222–232, Washington, DC, USA, 2001. IEEE Computer Society.

[YLLD09] Qiang Yan, Yingjiu Li, Tieyan Li, and Robert Deng. Insights into malware detection and prevention on mobiles phone. In *Security Technology journal*, pages 242–249, 2009.

[ZB07] Weiqun Zheng and Gary Bundell. Model-based software component testing : A uml-based approach. *ACIS International Conference on Computer and Information Science*, 0 :891–899, 2007.