Testabilité des services Web

Issam Rabhi

N° d'ordre : D.U : 2213
E D S P I C : 550

Université Blaise Pascal – Clermont II

École Doctorale
Sciences pour l'Ingénieur de Clermont-Ferrand

Thèse
présentée par

Issam RABHI

pour obtenir le grade de

Docteur d'Université
Spécialité Informatique

Testabilité des services Web

Soutenue publiquement le 09 janvier 2012 devant le jury :

Prof. FOUCHAL Hacene	Université de Reims	Rapporteur
Dr. ZAIDI Fatiha	Université Paris-Sud XI	Rapporteur
Prof. FAUVET Marie-Christine	Université de Joseph Fourier	Examinatrice
Prof. MISSON Michel	Université d'Auvergne	Directeur
Dr. SALVA Sébastien	Université d'Auvergne	Co-Directeur
Dr. LAURENÇOT Patrice	Université Blaise Pascal	Co-Directeur

Résumé

Cette thèse s'est attaquée sous diverses formes au test automatique des services Web : une première partie est consacrée au test fonctionnel à travers le test de robustesse. La seconde partie étend les travaux précédents pour le test de propriétés non fonctionnelles, telles que les propriétés de testabilité et de sécurité.

Nous avons abordé ces problématiques à la fois d'un point de vue théorique et pratique. Nous avons pour cela proposé une nouvelle méthode de test automatique de robustesse des services Web non composés, à savoir les services Web persistants (*stateful*) et ceux non persistants. Cette méthode consiste à évaluer la robustesse d'un service Web par rapport aux opérations déclarées dans sa description WSDL, en examinant les réponses reçues lorsque ces opérations sont invoquées avec des aléas et en prenant en compte l'environnement SOAP. Les services Web persistants sont modélisés grâce aux systèmes symboliques. Notre méthode de test de robustesse dédiée aux services Web persistants consiste à compléter la spécification du service Web afin de décrire l'ensemble des comportements corrects et incorrects. Puis, en utilisant cette spécification complétée, les services Web sont testés en y intégrant des aléas. Un verdict est ensuite rendu.

Nous avons aussi réalisé une étude sur la testabilité des services Web composés avec le langage BPEL. Nous avons décrit précisément les problèmes liés à l'observabilité qui réduisent la faisabilité du test de services Web. Par conséquent, nous avons évalué des facteurs de la testabilité et proposé des solutions afin d'améliorer cette dernière. Pour cela, nous avons proposé une approche permettant, en premier lieu, de transformer la spécification ABPEL en STS. Cette transformation consiste à convertir successivement et de façon récursive chaque activité structurée en un graphe de sous-activités. Ensuite, nous avons proposé des algorithmes d'améliorations permettant de réduire ces problèmes de testabilité. Finalement, nous avons présenté une méthode de test de sécurité des services Web persistants. Cette dernière consiste à évaluer quelques propriétés de sécurité, tel que l'authentification, l'autorisation et la disponibilité, grâce à un ensemble de règles. Ces règles ont été crée, avec le langage formel Nomad. Cette méthodologie de test consiste d'abord à transformer ces règles en objectifs de test en se basant sur la description WSDL, ensuite à compléter, en parallèle, la spécification du service Web persistant et enfin à effectuer le produit synchronisé afin de générer les cas de test.

Mots clefs : services Web, test de robustesse, testabilité, sécurité, génération de test.

Abstract

This PhD thesis focuses on diverse forms of automated Web services testing: on the one hand, is dedicated to functional testing through robustness testing. On the other hand, is extends previous works on the non-functional properties testing, such as the testability and security properties. We have been exploring these issues both from a theoretical and practical perspective.

We proposed a robustness testing method which generates and executes test cases automatically from WSDL descriptions. We analyze the Web service over hazards to find those which may be used for testing. We show that few hazards can be really handled and then we improve the robustness issue detection by separating the SOAP processor behavior from the Web service one. Stateful Web services are modeled with Symbolic Systems. A second method dedicated to stateful Web services consists in completing the Web service specification to describe correct and incorrect behaviors. By using this completed specification, the Web services are tested with relevant hazards and a verdict is returned.

We study the BPEL testability on a well-known testability criterion called observability. To evaluate, we have chosen to transform ABPEL specifications into STS to apply existing methods. Then, from STS testability issues, we deduce some patterns of ABPEL testability degradation. These latter help to finally propose testability enhancement methods of ABPEL specifications.

Finally, we proposed a security testing method for stateful Web Services. We define some specific security rules with the Nomad language. Afterwards, we construct test cases from a symbolic specification and test purposes derived from the previous rules. Moreover, to validate our proposal, we have applied our testing approach on real size case studies.

Key-words: Web services, robustness testing, testability, security, test case generation.

Remerciements

Tant de personnes ont rendu possible l'avènement de ce travail de thèse qu'il m'est aujourd'hui difficile de n'en oublier aucune. Je m'excuse donc par avance des oublis éventuels. Au terme de ces années de doctorat, j'éprouve une sincère gratitude envers tous ceux qui ont participé à ce travail.

Je tiens tout d'abord à remercier mes directeurs de Thèse, Michel Misson, Sébastien Salva et Patrice Laurençot, qui m'ont accueilli. A Michel d'abord, qui m'a fait l'honneur d'accepté de faire partie du jury. Mais surtout à Sébastien et Patrice pour m'avoir pendant trois ans guidé, lu, corrigé, aidé. Pour faire court, je les remercie pour m'avoir fait découvrir la recherche, pour leur humour et pour leur disponibilité.

Je remercie particulièrement Fatiha Zaidi et Hacene Fouchal pour avoir accepté d'être rapporteurs de ma thèse, et pour le temps consacré à ce travail ; et également Marie-Christine Fauvet d'être présidente de mon jury.

Ces années de doctorat ont été l'occasion de nombreuses collaborations et rencontres avec des chercheurs, des doctorants, et des professionnels de tous horizons qui ont enrichi mon travail d'une dimension humaine essentielle. J'ai une pensée particulière pour l'ensemble des membres du laboratoire LIMOS pour l'appui professionnel, mais aussi personnel, dont ils ont fait preuve tout au long de ces années de thèse. Je tiens aussi à remercier les membres de l'Université Blaise Pascal.

Je remercie très sincèrement les membres de projet WebMov.

Je dois un grand merci à tous mes chers amis et collègues qui m'ont accompagné pendant ces années. Merci à eux pour leur gentillesse et pour les discussions partagées. Ainsi que toutes les autres personnes qui m'ont apporté leur aide et que je ne peux citer ici.

Je ne pourrais jamais oublier le soutien et l'aide des personnes chères de ma nombreuse et merveilleuse famille.

Et pour finir, merci à tous ceux que je n'ai pas cités et qui ont influencé à un moment ou à un autre mon travail de thèse.

Table des matières

Liste des figures

Chapitre 1

Introduction

1.1 – Présentation du contexte

De nos jours, les systèmes informatiques sont devenus de plus en plus complexes et hétérogènes avec un marché toujours plus exigeant et évolué en matière de performance, d'efficacité, d'innovation, de compétitivité et de productivité. Pour survivre dans un tel contexte, les systèmes informatiques se doivent d'être évolutifs et adaptables.

L'architecture orientée services permet d'être une solution très efficace pour ces systèmes, en améliorant leur qualité et en simplifiant leur intégration dans l'infrastructure informatique de l'entreprise, via l'utilisation de composants réutilisables et distribués. Ces composants, basés sur l'infrastructure d'internet, sont appelé services Web.

Un service Web est une technologie qui permet aux entreprises de faire communiquer leurs systèmes d'informations avec ceux de leurs clients ou partenaires, via internet. Cette technologie basée sur le standard XML est totalement indépendante du matériel, ou du langage de programmation utilisé ce qui permet de facilement la mettre en œuvre. D'ailleurs, le nombre de services Web utilisés par les entreprises est en constante évolution.

Afin de produire des applications fiables, les entreprises qui utilisent les services Web doivent suivre des processus de qualité et de certification logiciels, comme le CMMI (Capability Maturity Model Integration) [15]. Pour garantir la fiabilité, ces processus de qualité intègrent dans leur méthodologie un ensemble d'activités de tests.

Le test de services Web apporte cependant de nouveaux challenges, spécialement lorsque nous considérons le profil WS-I basic [79].

En effet, les services Web n'appartiennent pas directement à l'application qui les invoque. Ils sont accessibles à travers plusieurs couches (SOAP, HTTP/HTTPS, couches client, processeur SOAP), ce qui réduit leur visibilité. D'autre part, un service Web peut être lui-même composé avec d'autres services existants ce qui réduit la faisabilité du test, autrement appelée testabilité.

Cette thèse a donc pour but:

- d'étudier principalement le test de services Web,
- de proposer de nouvelles méthodes de test automatique, qui prennent en compte l'environnement SOAP des services Web pour les valider au mieux,
- de déterminer les propriétés qui réduisent la faisabilité du test de services Web composés, ceux-ci étant décrits en utilisant le formalisme BPEL,

- d’évaluer des facteurs de la testabilité et de proposer des solutions afin d’améliorer cette dernière.

1.2 – Le cycle de vie du logiciel

Le développement d’un logiciel se fait suivant un cycle appelé le cycle de vie du logiciel.

Ce cycle est composé de méthodes de conception, d’analyse de codage et de test, permettant de produire une implémentation conforme à un cahier des charges initial décrivant des informations de base sur le fonctionnement voulu du produit à réaliser.

Le cycle de vie est décomposé en phases de développement séquentielles : chacune exploite le résultat délivré par la phase précédente, la traite et la remet à la phase suivante. 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 ce que le concepteur 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.

Les différentes phases du cycle de vie d’un logiciel (Figure 1) sont détaillées comme suit :

- Phase d’analyse des besoins : l’objectif essentiel est d’analyser les besoins du client qui sont ensuite utilisés dans un cahier des charges,

- Phase de conception : l’objectif de cette phase est la conception du logiciel selon les spécifications identifiées dans la phase précédente,

- Phase de validation de la spécification : cette phase (appelée Vérification) consiste à vérifier certaines propriétés comme l’existence des services proposés et leurs bons fonctionnements. Ainsi, elle permet de détecter certain nombre d’erreurs dues à la conception de la spécification.

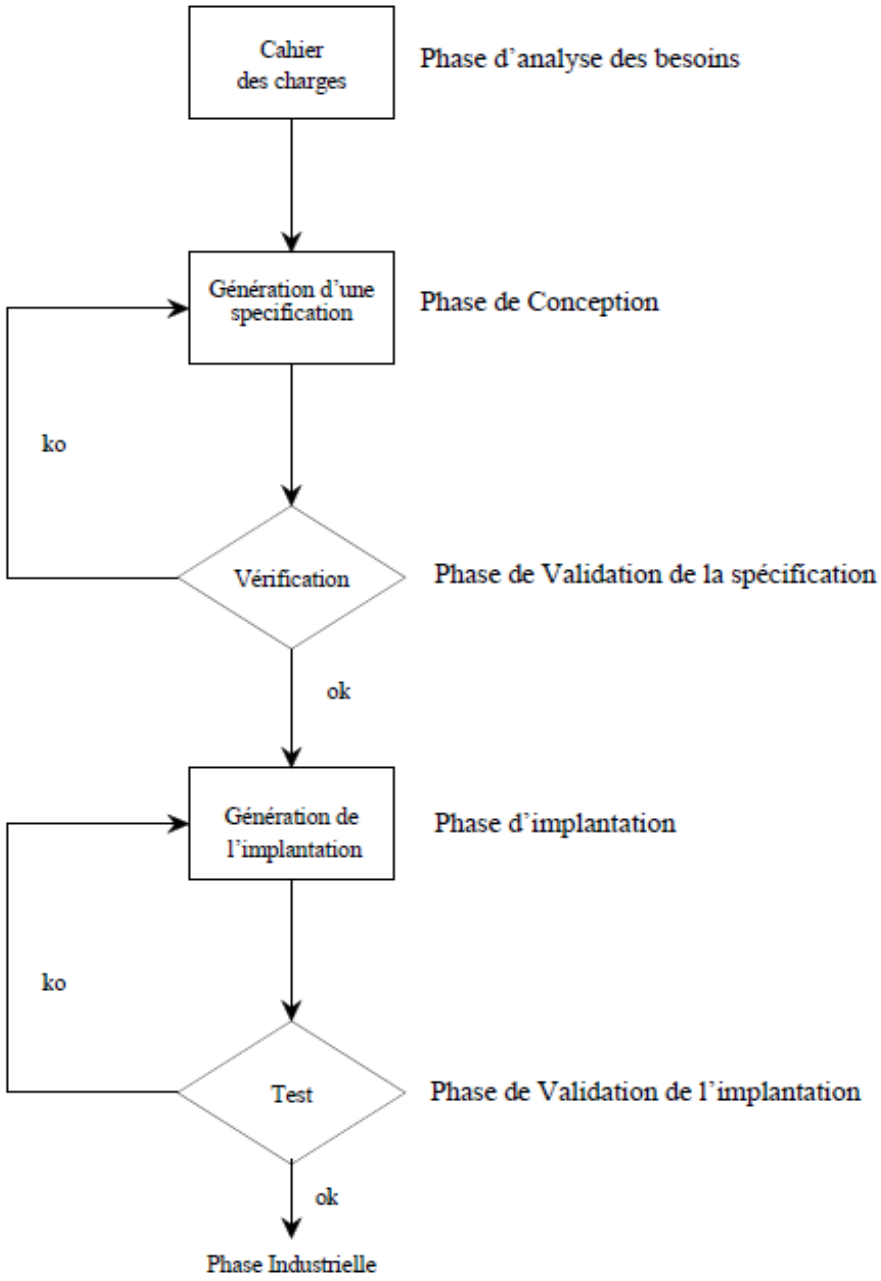


Figure 1: Cycle de vie du logiciel

- Phase d'implantation : cette étape aussi appelée phase de programmation, consiste à la rédaction du code source, c'est-à-dire aux instructions de programme en tenant compte de plusieurs paramètres tels que l'environnement du système, les interactions avec celui-ci...

- Phase de test : elle consiste à vérifier que les fonctions offertes par l'implantation correspondent à ses besoins. Ces tests peuvent être divers : liés aux performances, à la

robustesse, ainsi qu'à la conformité de l'implantation par rapport à sa spécification. C'est dans le cadre de ces types de test que s'inscrit cette thèse.

1.3 – Plan de la thèse

Ce manuscrit est organisé comme suit :

Chapitre 2 — Service Web : historique, concept et technique

Ce chapitre présente l'historique, les concepts et techniques des services Web ainsi que les mécanismes de composition de services. Ce chapitre décrit particulièrement les services Web composés en BPEL.

Chapitre 3 — Les méthodologies de test

Ce chapitre propose un état de l'art détaillé sur les modèles, méthodes et approches définies pour le test.

Le Chapitre 3 est consacré en premier lieu à la définition de modèles de description formelle. Il détaille ensuite les architectures de tests et les outils de test les plus connu. Enfin, il expose quelques méthodes de test appliquées dans le domaine de services Web : robustesse, conformité, sécurité et testabilité.

Chapitre 4 — Méthodologies de test de services Web

Le Chapitre 4 présente notre méthode de test de services Web non composés, à savoir les services persistants et ceux non persistants. Ce chapitre décrit deux méthodes automatiques de test de robustesse en prenant en compte l'environnement SOAP et une méthode de test de sécurité des services persistants.

Chapitre 5 — Testabilité de services Web composés

Le Chapitre 5 détaille notre méthodologie de test de services Web composés. Il présente, d'abord, une étude sur la testabilité des services Web composés. Il expose aussi des solutions afin d'améliorer la testabilité.

Chapitre 6 — Conclusion et perspectives

Ce dernier chapitre conclut la thèse en donnant des perspectives envisageables.

Chapitre 2

Service Web : historique, concept et technique

2.1 – Introduction

Les services Web puisent leur origine dans l'informatique distribuée et dans l'avènement du Web. Afin d'élucider les motivations qui ont poussé à la conception et la mise en place de ces dispositifs, nous faisons un retour dans le passé afin de repérer l'évolution (la chronologie) des langages de programmation, des concepts, des architectures et des organisations associées aux applications informatiques.

2.2 – L'apparition des services Web : évolution des architectures (Object, Composant et Service)

Au commencement de la programmation traditionnelle, le produit logiciel était fondé sur l'approche procédurale dont le concept est basé sur le traitement des données via des procédures implémentant des algorithmes. Deux langages majeurs ont marqué cette période : le Cobol et le C. La programmation procédurale est encore très utilisée aujourd'hui. Le noyau LINUX est un exemple de programme écrit dans ce style.

Le manque de liens particuliers entre les procédures et les structures de données a mené rapidement à des difficultés en cas de modification ou bien de réutilisation de la structure des données. D'un autre coté, l'absence de la modularité et l'abstraction a rendu le travail d'équipe difficile.

Ainsi, et depuis une vingtaine d'années, les développeurs de logiciels sont passés à la programmation orientée objets (POO).

La programmation par objets, pour sa part, tourne autour d'une unique entité appelée « objet » qui regroupe un petit ensemble de données représentant leurs propriétés. Cette entité est plus ou moins concrète.

Par rapport à la programmation procédurale, l'approche objet se caractérise par le regroupement dans une même classe de la description des attributs et des méthodes (opérations). Ce mécanisme de regroupement (appeler encapsulation) augmente le potentiel de réutilisation de classe et améliore aussi leur autonomie.

En outre, la gestion des erreurs ne se produit pas via les valeurs retour des fonctions, un autre mécanisme a été mis en place: les exceptions. La prise en compte d'exception est d'une importance capitale, car elle permet d'éviter la propagation d'une erreur vers les niveaux supérieurs. En effet, lorsqu'une erreur ou bien une action inattendue n'est pas traitée dans son unité de déclaration, elle se propage vers son unité appelante et elle se retrouve en dehors de sa portée. Le mécanisme d'exception se traduit par un signal qui traite le cas des erreurs en séparant traitement du cas normal d'un programme et traitement des cas exceptionnels.

Globalement, la programmation orientée objet a fait une arrivée des plus remarquées en apportant de nouveaux concepts tels que l'héritage et le polymorphisme. Aussi, elle convient mieux aux équipes de programmation importantes, car elle est réutilisable, évolutive et offre un autre angle de traitement des programmes qui est peu liée au problème à traité, donc plus stable (et réutilisable) pour divers problèmes.

Malgré ses atouts importants, l'approche objet souffre au moins de deux insuffisances majeures. En effet elle n'est pas suffisamment abstraite et pose le problème du niveau de granularité pour le cas de la modélisation des systèmes complexes.

La prise en compte des limitations de l'approche objet a mené à la création d'une nouvelle approche de programmation par « composant » [33], [62].

Les composants sont des entités logicielles fondées sur une interface contractuelle et une sémantique bien définie. Ces composants interagissent via une infrastructure permettant d'assurer la communication entre différentes entités au sein d'un même système ou à travers un réseau moyennant des composants distribués [11], [63]. L'architecture de composants basée sur le concept d'objet informatique distribué a engendré un développement important d'applications distribuées, plus étendues et complexes. Actuellement, il existe trois architectures majeures prenant en compte ses composants: les EJB (Enterprise Java Beans) de SUN [21], .NET/COM+ de Microsoft [16], et le CCM (CORBA Component Model) de l'OMG [53].

La mise en œuvre de ces trois architectures rencontre des difficultés particulières dans le cadre d'un réseau ouvert tel qu'Internet. Ces architectures disposent chacune de ses propres infrastructures et de ses propres règles de fonctionnement, ce qui limite l'interaction entre ces différents composants.

Les progrès techniques de ces dernières années ont permis de faciliter l'utilisation de composants grâce à une nouvelle forme de composants distribués basée sur l'infrastructure d'internet, appelée services Web.

A l'instar des technologies précédentes, les services Web proposent une architecture par composants qui permet aux entreprises, de faire communiquer leurs systèmes d'informations avec ceux de leurs clients ou partenaires, moyennant l'usage de fonctionnalités distantes. En plus, cette technologie offre des mécanismes qui permettent de traiter le problème de l'hétérogénéité des composants ce qui permet de créer des applications complexes, dans des domaines divers et sans contrainte d'infrastructure.

En effet, les services Web reposent sur les technologies standards de l'infrastructure d'Internet qui sont totalement indépendante du matériel, ou du langage de programmation

utilisé ce qui permet de facilement les mettre en œuvre. Alors que, les autres architectures par composants reposent chacune sur une infrastructure particulière. L'interopérabilité constitue donc une caractéristique importante des services Web. D'ailleurs, le nombre de services Web utilisés par les entreprises est en constante évolution.

Définition :

Un service Web est défini par « *Un système logiciel identifié par une URI, dont les interfaces publiques et les associations sont définies et décrites en XML. Sa définition peut être découverte par d'autres systèmes logiciels. Ces systèmes peuvent alors interagir avec le service Web selon les modalités indiquées dans sa définition, en utilisant des messages XML transmis par des protocoles internet* » (D'après le W3C http://www.org/TR/ws-gloss/).

Les services Web sont « des applications modulaires, qui se suffisent à elles-mêmes, qui peuvent être publiées, distribuées sur Internet » [64].

Expliquer le fonctionnement des services Web nécessite alors de présenter à la fois leur architecture et les technologies fondamentales sur lesquelles ils se fondent.

2.3 – L'Architecture Orientée Service (SOA)

Actuellement, sous le vocable de SOA, se développe un style d'architecture orientée service permettant de construire des systèmes informatiques évolutifs et adaptables, en améliorant leur qualité et en simplifiant leur intégration dans l'infrastructure informatique de l'entreprise, par recours à des composants réutilisables appelés services.

Dans la suite, nous présentons les concepts sur lesquels repose l'architecture SOA.

2.3.1 – Service

Le terme de service est inspiré par analogie avec des services simples et réutilisables du monde réel tel que le service de distribution de billets automatique.

Au sens SOA, un service est un programme autonome, réutilisable, indépendant des langages de programmation et qui peut s'exécuter sur n'importe quelle plateforme.

Ce composant clef du SOA expose un ensemble d'opérations (fonctionnalités) mises à disposition par un fournisseur à l'attention d'un client selon un ou des contrats prédéfinis. L'interaction entre le client et le fournisseur est faite par le biais d'un bus de services (médiateur) qui peut être dédié ou être entièrement prise en charge par internet. Dans ce dernier cas, il s'agit des services Web.

La Figure 2 montre un modèle fonctionnel de l'architecture SOA.

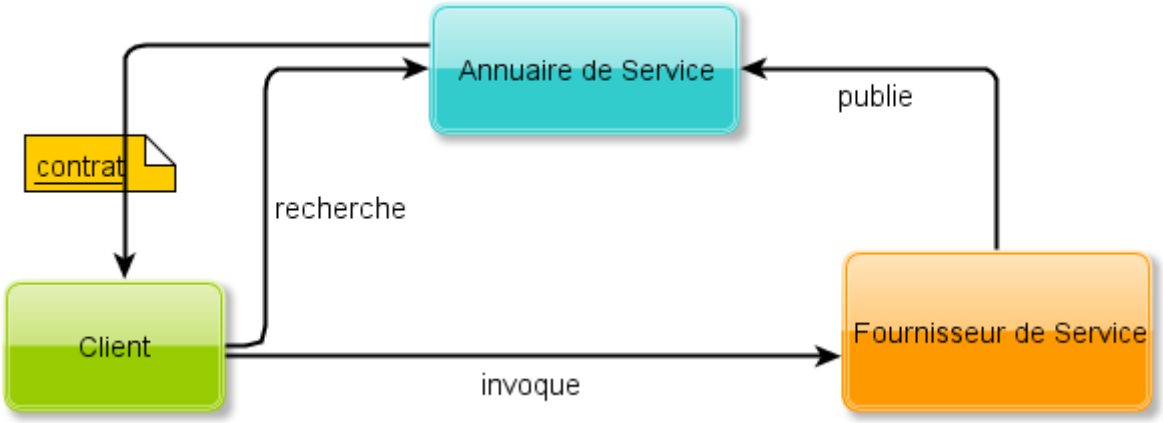


Figure 2: Modèle fonctionnel de l'architecture SOA

2.3.2 – Les concepts de SOA

Les concepts de SOA sont à la mode dans le monde des systèmes d'information. Ces concepts énoncent diverses promesses:

- La réutilisation, qui signifie la possibilité de réutiliser des services par plusieurs consommateurs. L'architecture SOA propose des services génériques en dissociant leurs fonctionnalités de leurs implantations afin qu'ils puissent être réutilisés [49],
- La composition des services qui est liée à la réutilisation. La composition est possible grâce à la modularité des services et permet d'assurer une fonctionnalité plus importante, en s'appuyant sur la composition de services (section 2.5),
- L'autonomie des services, ce principe de service joue un rôle important afin d'assurer la réutilisation et la composition des services. L'autonomie peut être définie par la capacité des services à manipuler ses ressources, contrôler son état et exécuter ses fonctionnalités, sans appui extérieur [22]. Cela signifie la non-dépendance du service par rapport à ceux qui veulent l'utiliser,
- L'interopérabilité des services, c'est à dire leur interconnexion sans programmation spécifique.

Les services Web présentent l'implémentation la plus commune de SOA. Ceux-ci sont de plus en plus employés dans les entreprises pour construire des applications orientées Business ou Web, en améliorant leur qualité et en simplifiant leur intégration dans l'infrastructure informatique de l'entreprise. Quelles sont les technologies prometteuses utilisées par les services Web?

2.4 – Les technologies des services Web

La majorité des grands sites Web (Amazon, eBay…) qui proposent des services Web aux développeurs, offrent simultanément deux catégories de services Web : SOAP et REST.

- Les services Web REST, exposent entièrement ces fonctionnalités comme un ensemble de ressources (URI) identifiables et accessibles par la syntaxe et la sémantique du protocole HTTP,
- Les services Web SOAP (appelés aussi WS-*), exposent ces mêmes fonctionnalités sous la forme de services exécutables à distance.

Afin de rendre les services Web SOAP interopérables, l'organisation WS-I propose de définir les services Web en introduisant des profils, en particulier le profil WS-I Basic [79]. Celui-ci est composé de quatre grandes parties: la description de l'interface du service Web grâce au langage WSDL (Web Services Description Language)[17], la sérialisation des messages transmis via le protocole SOAP (Simple Object Access Protocol)[61], l'indexation des services Web dans des registres UDDI (Universal Description, Discovery Integration)[74] et la sécurité des services Web, obtenue essentiellement grâce à des protocoles d'authentification et de cryptage XML.

L'une des particularités des services réside dans le recours permanent à une activité de mise à jour. Cette activité permet aux services d'affronter ses concurrents, en assurant des fonctionnalités de qualité et à jour.

2.4.1 – WSDL

Face aux modifications ou améliorations possibles au niveau du service, et conformément au profil WS-I Basic, un service Web doit publier la description de ses fonctionnalités à travers un fichier WSDL [17]. Ce dernier décrit les signatures des fonctionnalités proposées par le service (noms des fonctions, les différents types décrivant les paramètres des fonctions et les types de réponses retournées). Les détails techniques propres au fournisseur de service Web ne sont pas dévoilés aux consommateurs, ce qui permet d'assurer un faible couplage entre le fournisseur de service et les différents consommateurs de services.

Dans la mesure où il s'agit d'un document XML, le fichier WSDL se divise en plusieurs balises. Celles-ci présentent la description des éléments permettant de mettre en place l'accès à un service Web (Figure 3), en regroupant les informations suivantes: les interfaces du service (endpoints), les opérations fournies par le service, et les paramètres/réponses de ces opérations (types). Cette description, montre aussi comment les messages doivent être structurés en définissant les types complexes utilisés par ces derniers.

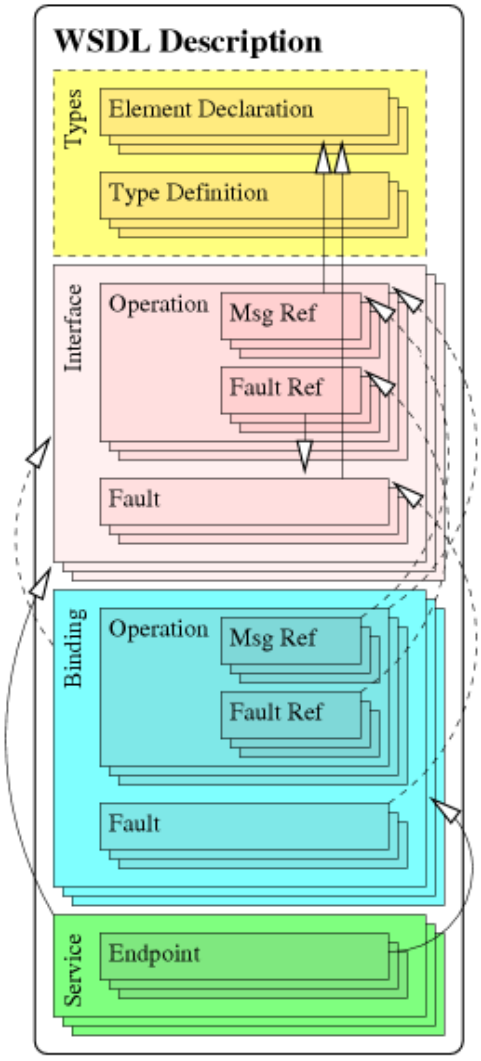


Figure 3: Description WSDL

Les types complexes peuvent être définis moyennant des XSD.

XSD:

Le XSD, acronyme de XML Schema Definition, est un langage de description de formats de document XML permettant de définir de façon structurée le type de contenu et la syntaxe d'un document XML à travers la définition de balises.

Ces balises sont ensuite utilisées dans des fichiers XML pour décrire des objets, ou pour valider un document XML, c'est-à-dire vérifier si le document XML respecte les règles décrites dans le document XSD [19]. Dans le cas des services Web, les XSD peuvent être utilisés pour définir les paramètres complexes (d'entrées ou de sortie) des différentes opérations fournies par ces services. Le standard XSD a été approuvé par le W3C [70].

Le WSDL est très souvent utilisé en combinaison avec le SOAP.

2.4.2 – SOAP

SOAP, développé par IBM et Microsoft, est une recommandation W3C qui le définit comme étant un protocole destiné à l'échange de messages structurés, permettant d'invoquer des applications sur des réseaux distribués [61].

Ce protocole SOAP reposant sur XML a pour but de mettre en place un mécanisme d'échanges de messages valable sur une variété de protocoles. De plus, il est conçu pour être indépendant du modèle de programmation de l'application et du système d'exploitation.

SOAP est utilisé pour invoquer les opérations (les méthodes) de services Web dans un réseau en sérialisant/désérialisant les données (les paramètres d'opération et les réponses). SOAP est mis en œuvre avec d'autres protocoles: le protocole HTTP qui est souvent utilisé pour des appels synchrones de service, ou le SMTP qui est souvent associé avec les appels asynchrones à travers des messages SOAP.

Un message SOAP est constitué de trois parties :

- d'une enveloppe, élément racine du message SOAP, définissant le contexte du message, son destinataire et son contenu. Elle contient un élément d'en-tête optionnel.
- de règles de codage afin d'exprimer les types de données définis par l'application,
- d'un protocole permettant la représentation des appels de procédure et des réponses à distance tel que le RPC (Remote Procedure Call).

2.4.3 – UDDI

UDDI, souvent appelé les "pages jaunes" des services Web, est un annuaire de services basé sur XML, destiné aux services Web et qui permet aux clients et aux applications de trouver et utiliser facilement et dynamiquement les services Web à travers le Web [13]. Pour cela, les descriptions de service Web sont réunies dans ces registres UDDI [74] qui peuvent être consultés manuellement ou automatiquement en utilisant des API de programmation pour rechercher dynamiquement un service Web spécifique. UDDI est une spécification mise au point par le Consortium de standards OASIS [65].

2.5 – La composition de services Web

La composition de services est un des enjeux principaux du SOA. En effet, les caractéristiques essentielles du SOA telles que le couplage faible entre les services, l'indépendance par rapport aux aspects technologiques et la mise à l'échelle favorisent la composition des services Web.

On dira qu'un service Web est composé lorsque son invocation engendre des interactions avec d'autres services Web.

Un service Web peut être lui-même composé avec d'autres services existants, afin de réaliser une tâche plus complexe, ou même de simplifier le travail lors du développement en utilisant des services déjà développés. Il existe deux stratégies générales pour composer ces services Web: la chorégraphie et l'orchestration.

- La chorégraphie définit la collaboration et les échanges de messages point à point entre plusieurs partenaires et plusieurs services Web ainsi elle propose une vision globale des interactions,
- L'orchestration, elle, définit l'enchaînement des services selon un scénario prédéfini, elle propose une vision centralisée décrivant la manière par laquelle les services peuvent agir entre eux.

Actuellement, plusieurs langages ont été proposés pour composer les services tels que le BPEL, WS-CDL, WSCI, XLANG, WSFL.

Par définition, un service Web est dit composé lorsque son invocation implique des interactions avec d'autres services Web. Cependant un service Web ne peut pas agir sur son environnement. Donc la composition doit être assurée par n processus (appelés aussi médiateurs) avec n=1 pour le cas d'orchestration et n>1 pour le cas de la chorégraphie. Un processus permet d'invoquer les services Web, sauvegarder leurs réponses et gérer les interactions avec d'autres processus. Son comportement est décrit grâce à un workflow adapté pour le service composé.
Un workflow est un ensemble d'instructions (flux d'information) décrivant les tâches (les invocations, les conditions, les affectations, les délais, les exceptions…) à accomplir entre les différents acteurs d'un processus (services Web et processus), et fournit à chacun des acteurs les informations nécessaires pour la réalisation de sa tâche. Un workflow est exécuté par un (ou plusieurs) processus et permet généralement d'automatiser une série de tâches accomplies par les différents acteurs en précisant leur rôle. Par exemple, dans le langage BPEL(Business Process Execution Language), le BPEL process présente le processus permettant d'orchestrer les services Web. La structure du fichier BPEL est la même que celle du workflow. Ce dernier est implémenté grâce aux différentes activités, que nous détaillerons dans la section suivante.

2.5.1 – BPEL

Le langage BPEL, nommé aussi BPEL4WS ou WS-BPEL, est un standard OASIS [78] d'orchestration de services Web. Ce langage est largement considéré comme étant le plus abouti du marché dans le cadre de la mise en œuvre des architectures orientées services.

BPEL définit le processus, l'enchaînement et l'ordonnancement des actions qui seront exécutées par le moteur d'orchestration, agissant comme une machine virtuelle capable d'exécuter le code BPEL.

Grâce à sa définition riche d'activités et ses mécanismes évolués, BPEL permet de:

- exécuter en parallèle des activités et de synchroniser des flots,
- gérer des exceptions, en particulier, des fautes et des événements,
- gérer la corrélation des messages, pour le cas des communications asynchrones,
- décrire des transactions contextuelles et de longue durée.

Les activités, offertes par ce langage, sont classées en deux catégories : activités basiques (Figure 4) et activités structurées (Figure 5).

Activités basiques
<reply> permet l'envoie d'un message comme réponse à une invocation d'un partenaire.
<invoke> permet d'invoquer une opération d'un service partenaire. Elle peut servir à la fois pour un appel synchrone (invoke-response) ou asynchrone (invoke oneway).
<receive**>** permet d'attendre un message d'un service partenaire.
<empty> une activité vide.
<throw> permet de gèrer une erreur interne (faute) au processus BPEL.
<exit> termine l'instance d'un processus BPEL.
<assign> permet de copier les données dans les variables du processus BPEL.
<wait> permet d'attendre un certain temps.

Figure 4: Description des activités basiques

Activités structurées
<flow> permet l'acheminement parallèle d'un ensemble d'activités et de spécifier un ordre partiel d'exécution.
<sequence> permet de définir un ordre d'exécution séquentiel pour une collection d'activités.
<scope> permet de regrouper un ensemble activités dans un seul bloc, afin qu'il soit traité par un même gestionnaire de fautes, d'événements, de terminaison et de compensation.
<while> permet une exécution en boucle de l'activité associée tant que sa condition booléenne est satisfaite au début de chaque itération.
<repeatUntil> permet aussi une exécution en boucle de sa sous-activité mais jusqu'à la satisfaction de sa condition.
<if> définit un choix.
<pick> attend l'arrivée d'un message particulier ou qu'une alarme soit déclenchée. Quand l'un des deux se produit, sa sous-activité est ainsi exécutée.

Figure 5: Description des activités structurées

2.5.2 – ABPEL

ABPEL (Abstract BPEL) [1], est une variante abstraite de BPEL, permettant de décrire le comportement extérieur d'un processus.

ABPEL ne présente pas les détails internes du processus comme les activités *assign* ou *empty* et n'est pas exécutable. Il décrit seulement les interactions effectuées entre les différents partenaires participants au processus d'affaires.

Chapitre 3

Les méthodologies de test

3.1 – Introduction

Si l'on se réfère aux fondamentaux, c'est-à-dire à la norme IEEE [36] (Standard Glossary of Software Engineering Terminology), le terme de « test » apparaît comme:
« *l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus* ».
Pour pouvoir tester un système donné, ce dernier doit d'abord être spécifié.
Dans ce chapitre, nous présenterons dans un premier temps un survol des modèles formels qui permettent de décrire les spécifications formellement. Nous décrirons dans la suite quelques types, méthodes, architectures et outils de test. Enfin nous détaillerons une partie des méthodes de test appliquées aux services Web.

3.2 – Description de modèles : FSM, LTS, STS et UML

Nous rappelons, dans cette section, les modèles formels de bas niveau permettant de représenter les comportements attendus (spécifications) des systèmes à analyser. Cette pratique de formalisation, appelée aussi TDF (Techniques de Description Formelles), concerne principalement les préoccupations relative à :

- avoir une spécification formelle non ambiguë,
- appliquer facilement des algorithmes et des méthodes de test sur une spécification,

Plusieurs modèles ont été proposés parmi lesquels il existe des modèles de type algèbre de processus, automates, etc. Nous nous intéressons ici aux modèles de type automate qui permettent de représenter les spécifications par une succession d'actions (évènements) d'entrées et sorties. Les modèles relatifs aux travaux de cette thèse et décrit par suite sont : les FSM (machines à états finis) [45], LTS (systèmes de transitions étiquetés) [73] et les STS (systèmes de transitions symbolique) [26].

3.2.1 – FSM

Parmi les modèles formels, le modèle de machine à états finis ou FSM est le plus connu et le plus utilisé.

Un FSM est un automate dont chaque état décrit un état interne du système. Les transitions sont étiquetées par des entrées et des sorties décrivant respectivement les actions reçus par le système et émis depuis ce dernier.

Formellement, un FSM est un graphe défini par un quintuplet $F = <So, S, I, O, T>$ où :

- $So \in S$ est l'état initial,
- S est un ensemble non vide d'états,
- I est un ensemble fini d'entrées,
- O est un ensemble fini de sorties,
- T est un ensemble non vide de transitions tel que $T \in S \times I \times O \times S$. Une transition $t \in T$ est représentée par le tuple $<s,i,o,s'>$. Ce dernier indique que si l'automate est dans l'état s et qu'il rencontre l'entrée i, il passe dans l'état s' en fournissant la sortie o. La transition t est notée par : $s \xrightarrow{i/o} s'$.

Différentes variantes des FSMs sont utilisées dans la littérature [57] : FSM déterministe, FSM non-déterministe, CFSM (Communicating Finite State Machine), EFSM (Extended Finite State Machine), etc.

Un FSM est dit déterministe s'il existe, pour un état et une entrée donnée, une seule sortie et un seul état suivant possible.

Formellement, un FSM $F =< So, S, I, O, T >$ est déterministe si :

$\forall s \in S, \forall t = s \xrightarrow{A/B} s' \in T$ avec $A \in I, B \in O, s \in S, \forall t = s \xrightarrow{C/D} s'' \in T$, avec $C \in I, D \in O, s \in S, A = C \Rightarrow B = D$ et $s = s''$.

3.2.2 – LTS

Un LTS [50] est un système de transitions étiquetées, c'est-à-dire un automate incluant des états reliés par des transitions étiquetées représentant des actions. Ces actions peuvent être de type entrée (input), sortie (output) ou évènement interne (actions non exécutées sur une interface du système comme la mise à jour d'une variable).

Le point commun entre les modèles LTS et celles de FSM est qu'ils modélisent les systèmes via un ensemble d'états et d'actions associées aux transitions entre états. La principale différence se situe donc dans l'étiquetage des transitions. En effet, à la différence des modèles FSM, les LTS permettent de distinguer la réception d'une action d'entrée (stimulus) d'une émission d'une action de sortie en les modélisant séparément. Ceci permet de décrire les comportements asynchrones de spécifications pouvant exécuter plusieurs entrées et/ou plusieurs sorties successivement.

Formellement, un LTS est un quadruplet $< Q, Qo, L, T >$ tel que :

- Q est un ensemble fini non vide d'états,
- Qo est l'ensemble des états initiaux du système de transitions étiquetées,
- L est un ensemble d'étiquettes (représentant des actions ou des interactions),
- $T \subseteq Q \times (L \cup \{\tau\}) \times Q$ est un ensemble de transitions (τ représente une action interne). Une transition $t \in T$ est représentée par le tuple <q,a,q'>. Cette relation représente une transition qui part de l'état q et va vers l'état q' sur l'action a. Une telle transition est noté par : $q \xrightarrow{a} q'$.

Diverses variantes des LTS ont également été proposées [10] : IOLTS (Input-Output Labelled Transition Systems) [75], IOTS (Input-Output Transition Systems) [72], MIOTS (Multiple Input-Output Transition Systems) [34], etc.

Comme précédemment, nous allons définir ce qu'est un LTS déterministe.

Soit le LTS $A =< Q, Qo, L, T >$. A est déterministe si:

$\forall\, q \in Q, \forall\, t = q \xrightarrow{A} q' \in T$ avec $A \neq \tau \in L$, $q' \in Q$, $\forall\, t' = q \xrightarrow{B} q'' \in T$, avec $B \neq \tau \in L$, $q'' \in Q$, $A \neq B$ ou $q' = q''$.

3.2.3 – STS

Un *symbolic transition systems* (STS) [26] est une extension des LTS où les transitions sont dites symboliques.

Les STS incorporent explicitement la notion de données et permettant de modéliser de manière concise des systèmes de transitions ayant un nombre infini d'états (dont chaque état caractérise un ensemble éventuellement infini de variables).

Un $STS < L, L_0, Var, var_0, I, S \rightarrow >$ est constitué d'un alphabet de symboles S et d'un ensemble de variables Var initialisé par var_0. Var est mis à jour en tirant une transition incluse dans $\rightarrow$. Chaque transition $(l_i, l_j, s, \varphi, \rho) \in \rightarrow$ partant de l'état l_i vers l_j étiquetée par le symbole s, peut effectuer une mise à jour des variables avec ρ ($var_i := var_i + \rho$ indique une mise à jour pour var_i de valeur ρ) et possède une garde φ sur Var décrivant la condition de franchissement d'une transition et qui doit être satisfaite pour tirer cette transition.

Nous notons $(x_1, \ldots, x_n) \vDash \varphi$, la satisfaction de la garde φ suivant les valeurs $(x_1, \ldots, x_n)$, x_i correspondant à la valuation de la variable var_i.

Le principal avantage lié à l'utilisation des STS est leur format compact qui permet d'améliorer la lisibilité et l'expressivité des variables.

3.2.4 – UML

Le langage UML (Unified Modeling Language) est principalement adopté par l'OMG comme un standard de modélisation orientée objet. Dans ses dernières versions 2.x, UML peut être utilisé pour décrire les machines à états et pour modéliser le comportement discret d'un système à travers un LTS. Les versions 2.x rassemblent plusieurs types de machines à états : machine à états comportementale « behavioral state machine » et machine à états protocolaire « protocol state machine ». Nous notons aussi qu'un diagramme UML peut-être couplé avec un langage de contrainte comme OCL (Object Constraint Language). Ce dernier est un langage d'expression permettant de donner des descriptions précises en complétant les diagrammes.
L'intérêt d'utiliser ces formalismes (FSM, LTS, STS, UML) réside dans l'automatisation des techniques de test que nous détaillerons dans la section suivante.

3.3 – Types de test

Nous présentons dans cette section un modèle de fautes en explicitant sommairement les principales erreurs possibles, quelques techniques de test et une classification de ceux-ci proposée par J. Tretmans.

3.3.1 – Modèle de fautes

Le test de protocoles est basé sur la détection d'erreurs en envoyant des séquences de tests directement sur les interfaces de communication du système (l'implantation).
Nous rappelons quelques types d'erreurs, détectés via les descriptions formelles cités précédemment.
– erreur de sortie : une erreur de sortie se produit, si à la réception d'une entrée correspondante, l'implémentation émet une sortie différente de celle attendue ou prévue dans la spécification,
– erreur de transfert : une erreur de transfert se produit si pour une entrée donnée, l'‘implémentation atteint un état différent de celui décrit dans la spécification,
– état manquant/extra état : cette erreur se produit si l'implémentation n'a pas le même nombre d'états que la spécification,
– erreur de dépassement de délai : cette erreur se produit dans le cas d'une attente plus longue que celle prévue dans la spécification,

– erreur d'entrée/sortie temporelle : cette erreur se produit si une entrée/sortie prévue à un certain moment arrive hors délai,

Nous notons que les techniques de description formelles, que nous avons présentées précédemment, ne sont pas dédiées à l'aspect temporel.

Par définition, avec LTS et STS, un état est quiescent si aucune sortie n'est possible dans cet état. La quiescence est modélisée par une transition spéciale étiquetée δ, cette transition peut correspondre aussi un blocage sur l'ensemble de sortie après un timeout *t*.

Pour détecter ces erreurs, de nombreuses méthodes de test existent dans la littérature.

3.3.2 – Techniques de tests et classification

Plusieurs classifications de tests ont émergé. Nous détaillons dans cette section celle proposée par J. Tretmans selon trois axes. Cette classification est illustrée dans la Figure 6.

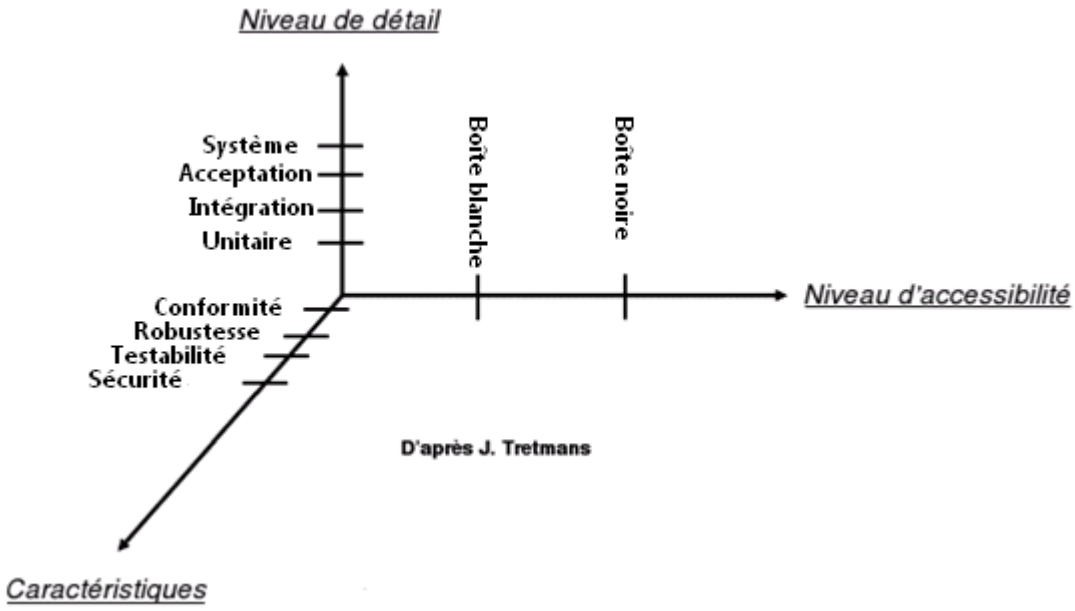


Figure 6: Classification de tests (Tretmans)

Dans le niveau de détail, quatre niveaux de test se présentent:

– test unitaire, qui vérifie les fonctions une par une, en isolation du reste du système,

– test d'intégration, qui vérifie le bon enchaînement des interactions entre systèmes intégrés,

– test système, qui vérifie le comportement global du système lors de la phase d'assemblage,

– test d'acceptation, qui permet de déterminer si un système satisfait ou non des critères d'acceptation.

Le niveau d'accessibilité est regroupé en deux catégories :

– les tests boîte blanche : qui sont applicables sur des systèmes dont la structure interne est connue. Par exemple, les boucles d'un code source peuvent être testées (les initialisations, les conditions etc.). Ce type de test est appelé aussi test structurel,

– les tests boîte noire : qui sont utilisés sur des systèmes dont la structure interne n'est pas connue. Par conséquent, les testeurs ne peuvent avoir accès qu'aux interfaces. Pour vérifier qu'une opération fonctionne correctement, le test s'appui sur les informations retournées par celle-ci.

Nous notons qu'une combinaison des deux approches précédentes est appelée: tests en boîte grise.

Le troisième axe expose des caractéristiques à tester, nous en citerons quelques exemples dans les sections ci-dessous : la robustesse, la conformité, la testabilité et la sécurité.

Test de robustesse

La robustesse est considérée comme étant «*la capacité d'un système à fonctionner correctement en présence d'entrées invalides ou d'environnement stressant*» (IEEE Standard Glossary of Software Engineering Terminology 1999). Ceci implique qu'un système est robuste s'il agit conformément à sa spécification en présence d'entrées inusuelles et anormales.

Le test de robustesse consiste à tester la réaction d'un système dans des conditions inattendues (erreurs de manipulation, conditions aux limites, malveillances, etc) et qui ne sont généralement pas prévues dans la spécification du système. Ce test de la robustesse permet donc d'évaluer la stabilité et la fiabilité du système.

Test de conformité

Le test de conformité, est l'une des méthodes de test, qui est normalisée par la norme ISO9646 qui définit le cadre et la méthodologie générale de tests de conformité.

Le test de conformité peut être formalisé à travers une relation d'équivalence entre une implémentation et une spécification, appelée relation de conformité. Celle-ci consiste à vérifier qu'une implémentation satisfait les comportements initialement prévus par sa spécification de référence.

Supposons que *Spec* est une spécification décrite par une FSM et *Imp* une implantation décrite aussi par le même formalisme. Une relation de conformité entre *Imp* et *Spec* notée par, *Imp conf Spec*, et satisfaite si et seulement si :

- *Imp* et *Spec* acceptent les mêmes symboles d'entrées *E*,

- *Imp* et *Spec* se comportent de la même manière face aux différentes entrées *E*, c'est-à-dire en appliquant E, nous devront observer le même ensemble de symboles de sortie depuis *Spec* et *Imp*.

Une exécution d'un test consiste à la mise en parallèle d'un testeur *T* (fabriqué à partir de la spécification) et de l'implémentation *Imp* en laissant le système évolué via la communication entre *T* et *Imp*. Plusieurs observations sont envisageables par le test (blocages, refus, etc). S'il existe un état s tel que V(s) = pass, l'exécution est réussie.

Cette relation de conformité repère les erreurs d'implantation grâce à des séquences d'actions (de test et de comparaison), pour déterminer si les réactions du système sont ceux prévus. Cela se passe en deux étapes.

- La première étape réside à obtenir des séquences de test, en appliquant sur la spécification de méthodes de génération de séquences de test,
- La deuxième étape est l'étape de test qui consiste à vérifier si les comportements de l'implémentation sont identiques à celles de la spécification en appliquant ces séquences de test au système réel, et ce afin d'obtenir un verdict.

L'application de séquences de test sur l'implémentation retourne trois types de verdict :

- **PASS** qui signifie que le comportement observable prévu dans la spécification a été identique avec l'implémentation après l'application d'une séquence de test,

- **INCONCLUANT** qui ne permet pas de conclure si l'implémentation est conforme ou pas à sa spécification,

- **FAIL** ce dernier verdict, indique la présence d'erreurs dans l'implémentation rapport à la spécification.

Grâce à ces verdicts, nous pouvons conclure sur la conformité de l'implémentation par rapport à sa spécification.

Test de sécurité

Les systèmes d'informations sont définis par l'ensemble des ressources, données et services matérielles et logicielles de l'entreprise. Celles-ci doivent être accessibles à la consultation par les personnes autorisées, elles doivent aussi pouvoir être modifiées, partagées ou sauvegardées. Le système d'information représente l'épine dorsale de l'entreprise, et il est nécessaire de le sécuriser.

La sécurité des systèmes informatiques consiste à s'assurer que les ressources d'une organisation sont uniquement utilisées dans le cadre prévu.

La sécurité informatique vise généralement cinq principaux objectifs (Voir Figure 7)
-L'authentification: consiste à comparer des informations d'identité fournies à des informations stockées,
- L'autorisation: chaque client à un accès aux données selon ces droits d'accès.
- La confidentialité: consiste à garantir que seules les personnes autorisées aient accès aux informations échangées. Cette propriété est assurée avec le cryptage,
- L'intégrité: vise à s'assurer que chaque bit envoyé par l'expéditeur est reçu par le bénéficiaire sans duplication. Cette propriété est assurée avec la signature,
- La disponibilité: signifie que le système doit répondre une fois appelé.
Ces concepts de base de la sécurité peuvent être classés en deux types : le premier regroupe les concepts qui garantissent la sécurité de la couche de transport, tels que l'intégrité et la confidentialité et le deuxième type englobe les trois autres concepts qui touche plus les données.

	chiffrement	Signature	intégrité	Echange authentification	Contrôle d'accès
Authentification	X	X		X	
Autorisation					X
Confidentialité	X				
Intégrité	X	X	X		
Disponibilité				X	X

Figure 7: Objectifs de base de la sécurité

Voici quelques exemples d'attaques connues :

- Le Déni de Service (DoS) reste la forme d'attaque la plus difficile à contrer. Ces attaques cherchent à réduire la disponibilité d'un service. Le but est atteint en dépassant les limites des ressources du réseau, d'un système d'opération ou d'une application,

- Une injection SQL est une exploitation d'une faille de sécurité d'une application utilisant une base de données, en injectant des requêtes SQL non attendues par le système et pouvant compromettre notamment des perte de données, usurpation d'identité, contournement de règles de gestion et altération de données,

- Attaques sur mot de passe : les attaquants mettent en place divers méthodes pour saisir les mots de passe : la force brute, les programmes de type Cheval de Troie, IP spoofing et les détecteurs de trames.

Ces attaques qu'elles que soient leur nature, menacent la survie de n'importe quel système, réseau, logiciel ou matériel.

La testabilité

La testabilité d'un système caractérise sa capacité à être testé à travers un ensemble de critères qui évaluent sa capacité à relever ses fautes, l'accessibilité de ses composants et les coûts de son test. L'analyse de la testabilité passe par deux étapes :

- la qualification du système, qui consiste à calculer les facteurs de testabilité,
- la planification des tests, qui consiste à spécifier le plan de test.

La Figure 8 illustre le niveau de la testabilité dans le cycle de vie de logiciel.

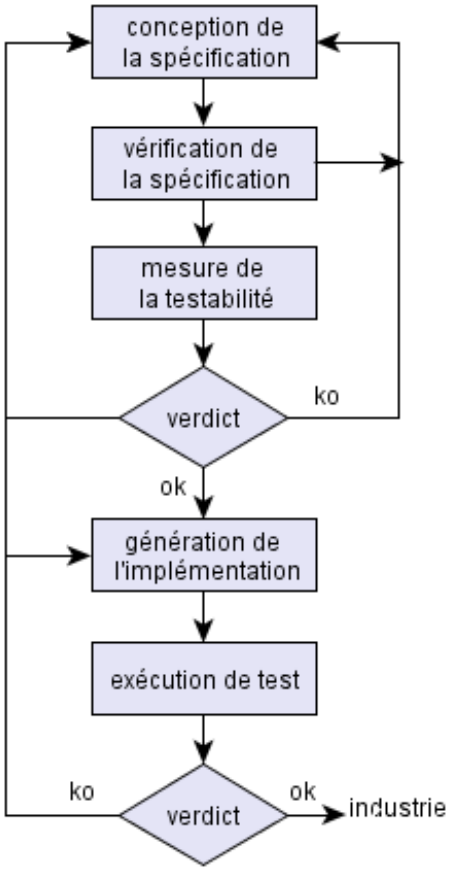


Figure 8: Testabilité et cycle de vie

Il existe diverses caractéristiques globales de testabilité, dont l'accessibilité, l'observabilité, la contrôlabilité et le coût d'exécution.

Nous détaillons ci-dessous l'observabilité et la contrôlabilité selon Freedman[28]:

Observabilité : Un système est observable, si et seulement si pour toute entrée *e*, il existe une sortie *s* différente des autres. Pour un modèle relationnel, cela peut se décrire comme l'indique la Figure 9. Celle-ci décrit un exemple de système non observable car pour les entrées *e1* et *e2* (resp. *e3* et *e4*) il existe une même sortie *s1* (resp. *s3*).

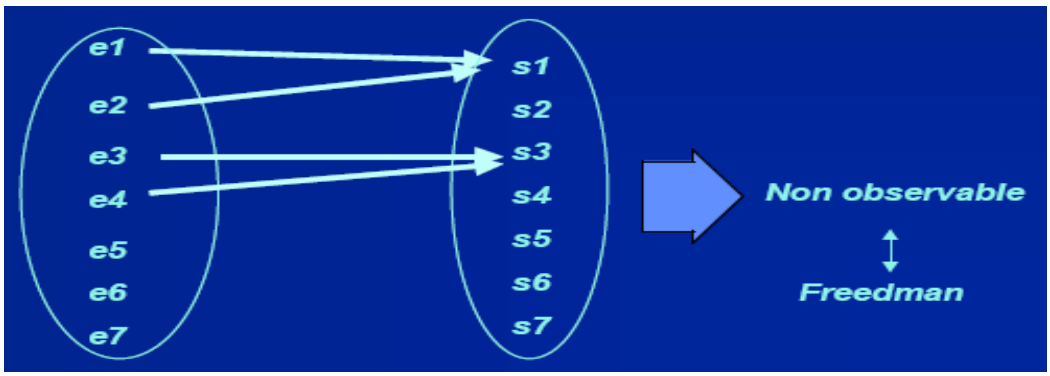


Figure 9: Un système non observable

Contrôlabilité : Un système est contrôlable, si et seulement si pour toute sortie *s*, il existe une entrée *e* qui force cette sortie.

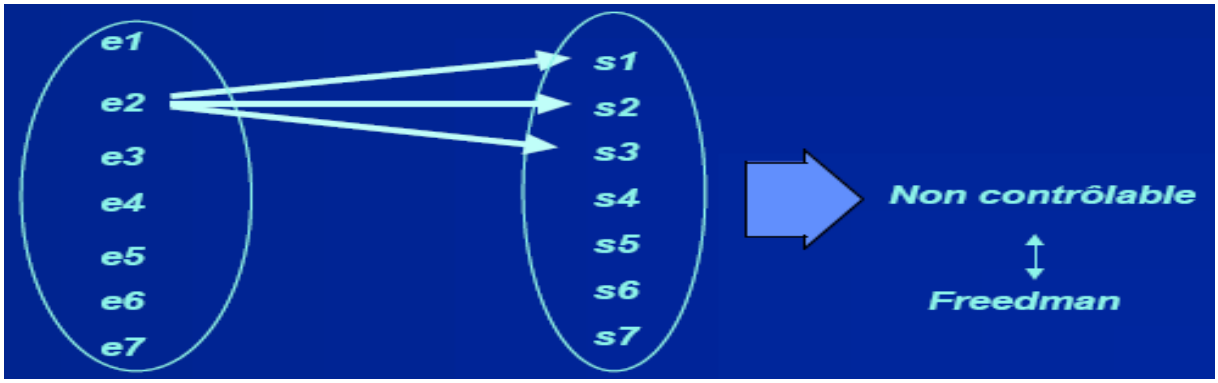


Figure 10: Un système non contrôlable

La Figure 10 décrit un système non contrôlable car il existe une même entrée *e2* qui a forcée les sorties *s1*, *s2* et *s3*.

Un système est testable s'il est observable et contrôlable. La Figure 11 présente un modèle relationnel décrivant un système testable, puisque pour chaque sortie *s* il existe une entrée *e* qui force cette sortie (contrôlabilité) et pour toute entrée *e* il existe une sortie *s* différente (observabilité).

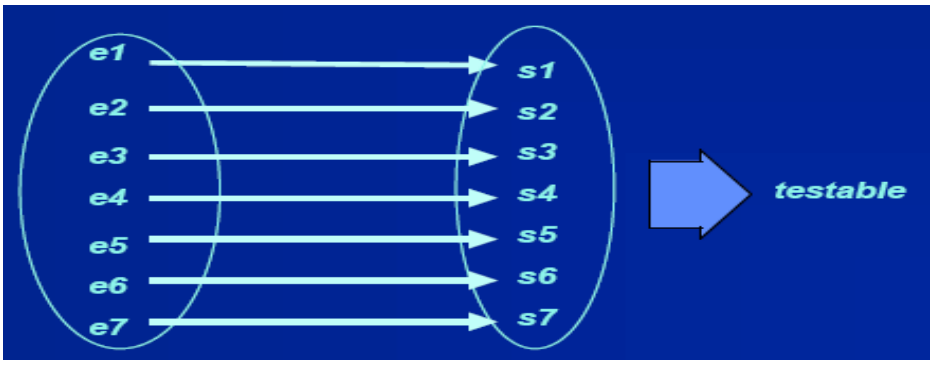


Figure 11: Un système testable

Comment peut-on évaluer et tester les systèmes informatiques et ses environnements? Quelles techniques ou méthodes appliquées? Nous rapportons dans la section suivante quelques éléments de réponse.

3.4 – Description de quelques méthodes de test

Cette section présente quelques méthodes de test utilisées dans nos travaux: relation de conformité, objectif de test et le test canonique.

3.4.1 – Relation de conformité : IOCO

Il est possible de définir ou d'étendre des relations de conformité afin de s'adapter aux besoins : selon l'architecture de test, l'environnement de test etc.

La relation IOCO, décrite dans [73], présente la définition de la relation de conformité la plus utilisée actuellement. Il s'agit d'une relation entre implémentations et spécifications. Nous utilisons la relation IOCO comme une équivalence de traces pour déterminer si l'implémentation est conforme à la spécification donnée.

Nous détaillons dans cette section la relation IOCO. Pour cela, nous définissons une trace.

trace est une trace d'exécution représentant les observations possibles faites par un testeur sur une implémentation, ou encore une séquence finie d'événements observés lors des exécutions des cas de test. La relation IOCO s'exprime simplement par inclusion de traces.

Etant donnée Imp (une implémentation) et Spec* (une spécification incluant éventuellement des blocages), deux IOLTS. Nous notons out(*Spec** *after* σ) les sorties autorisées par la spécification *Spec** après la trace σ. Soit la définition suivante :

$$\text{Imp IOCO Spec}* = \forall \sigma \in \text{trace(Spec}*\text{)}: \text{out(Imp after } \sigma) \subseteq \text{out(Spec}* \text{ after } \sigma)$$

En clair, Imp est ioco-conforme à Spec $*$ si et seulement si:

- si Imp produit la sortie x après la trace σ, alors Spec $*$ doit contenir x après σ.
- si Spec * ne peut pas produire de sortie après la trace σ, alors *Imp* ne peut pas produire de sortie après σ.

Selon la relation IOCO, l'implémentation sous test est conforme à sa spécification, si et seulement si l'exécution de test est réussie. Une implémentation est non conforme si une exécution d'un cas de test comporte au moins une action de sortie non spécifiée. Donc, dans ce cas l'exécution de test n'a pas réussie.

Exemple Soit la relation IOCO illustrée par la Figure 12.

La première implémentation Imp1 est bien conforme à la spécification Spec : Imp1 IOCO Spec, puisque pour tout état, les sorties de Imp1 sont incluses dans celles de Spec. En effet, IOCO permet de restreindre les sorties de l'implémentation par rapport à celles prévues dans la spécification de référence. Cependant, pour le deuxième implémentation Imp2 nous avons ¬(Imp2 IOCO S), car la sortie !y après la séquence !x_!z_!x n'est pas autorisée dans la spécification Spec.

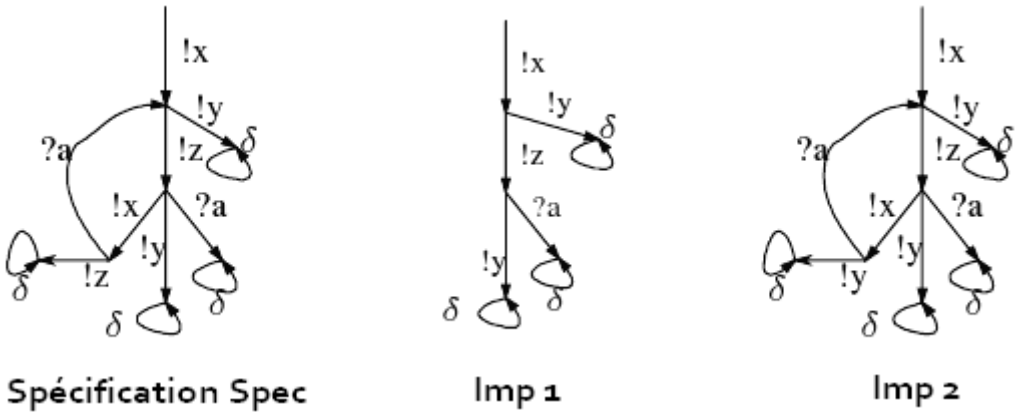


Figure 12: Exemple IOCO

3.4.2 – Testeur Canonique

Un testeur canonique peut être défini comme un automate dérivé de la spécification et qui présente l'ensemble des tests suffisants pour vérifier la conformité des implémentations. Celle-ci est appelée une relation d'implémentation. Le testeur canonique *T* et l'implémentation sous test *Imp* sont exécutés en parallèle, et un blocage de l'exécution donne un échec de test.

Une relation d'implémentation est définie par une relation *R* (LTS × LTS). Étant donnés Imp, *Spec* ∈ LTS^2. Nous notons out(σ, *Spec*) les sorties autorisées par la spécification *Spec* après la trace σ .Soit donc :

R (Imp, Spec) ssi (∀σ ∈ trace(Spec))(σ ∈ trace(Imp) ⇒ out(σ, Imp) ⊆ out(σ, Spec)).

Cela signifie que l'implémentation *Imp* est bien conforme à *Spec*.

Exemple Dans la Figure 13, les implémentations *Imp1* et *Imp3* sont conformes à la Spécification *Spec*. Par contre *Imp2* n'est plus conforme à Spec à cause de la sortie !f.

Donc la relation R assure que tout les sorties de l'implémentation en réponse à une entrée donnée ont été attendues par la spécification de référence.

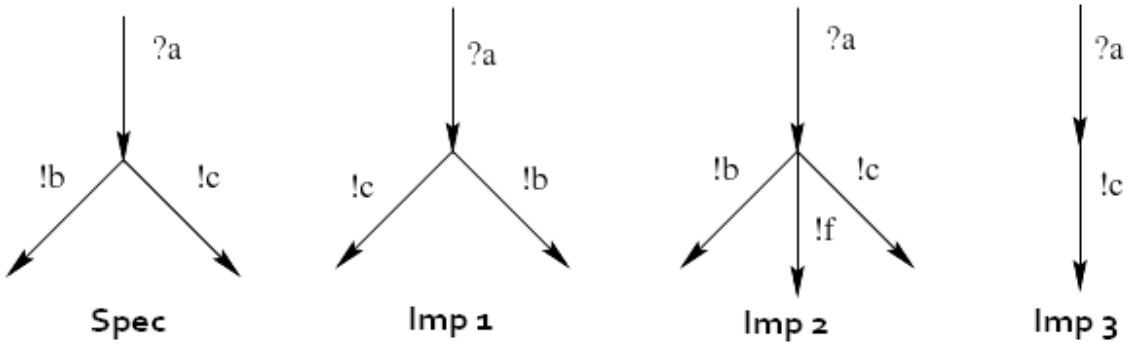


Figure 13: Une spécification et ses implémentations

3.4.3 – Objectif de test

A la différence des deux méthodes précédente, un objectif de test ne consiste pas à comparer le comportement global de l'implémentation par rapport à sa spécification. Des propriétés, appelées objectif de test, sont définies par une séquence d'évènements.
A partir d'un objectif de test donné, un ou des cas de test sont générés. Ces derniers décrivent une séquence d'évènements visant à observer une propriété définie par l'objectif de test, ce qui permet de réduire l'exploration de la spécification en testant qu'une partie du comportement de la spécification.
Formellement, un objectif de test est souvent décrit comme un automate déterministe et acyclique dont les transitions sont étiquetées par des interactions de la spécification avec son environnement. Les états finaux sont des états d'acceptation (avec le mot clé accept), indiquant que l'objectif de test doit être entièrement satisfait, ou des états de refus (avec le mot clé reject) montrant que la propriété ne doit pas se vérifier.
Les objectifs de test sont souvent créés à la main [73,41], puis un produit synchrone est réalisé entre chaque objectif et la spécification afin de ne garder que les chemins acceptables, c'est-à-dire ceux pouvant être utilisés pendant la phase de test.
D'autres travaux ont été réalisés pour la génération automatique des objectifs de test [14, 30, 71].

3.5 – Architectures de test

Afin d'appliquer les séquences de tests, il existe plusieurs modes d'accès aux implémentations sous test. Suivant ces modes d'accès (observation et/ou contrôle), plusieurs architectures de

test sont définies. La norme ISO 9646 [38] définit quelques architectures, dont les plus connus sont les architectures locales et distribuées.

3.5.1 – Architecture locale

Dans l'architecture locale le testeur et l'implémentation sous test sont exécutés sur une même machine. Pour ce type d'architecture deux testeurs y sont considérés :

- Le Testeur supérieur qui communique directement avec le service de niveau N (ASP N) à travers les points d'accès à l'implantation et qui sont appelés des PCO (Point de Contrôle et d'Observation).
- Le Testeur Inférieur qui communique avec le service de niveau N-1 (ASP N-1) au moyen des PCO.

La Figure 14 illustre cette campagne de test.

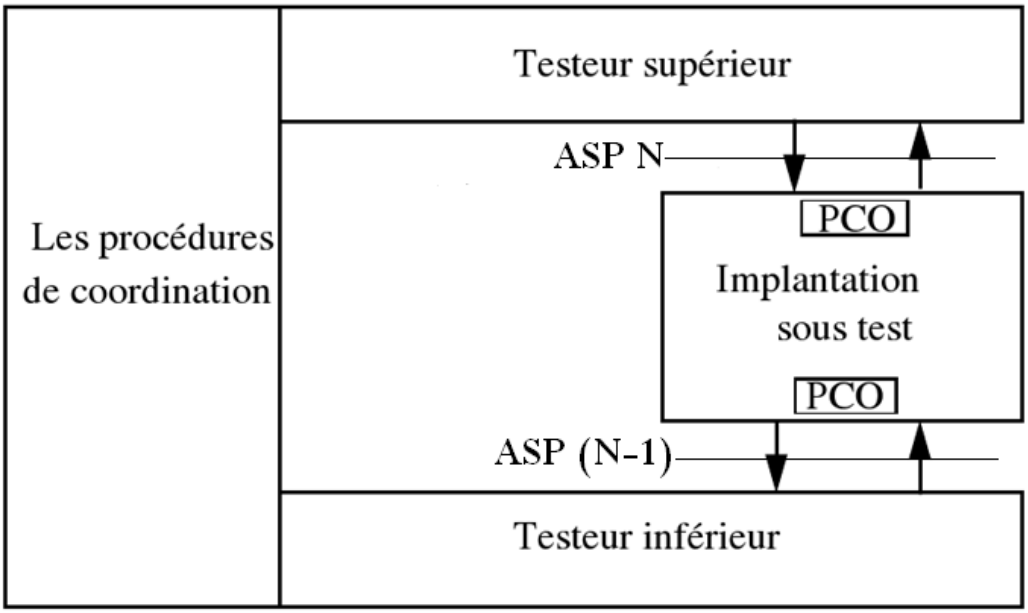


Figure 14: Architecture locale

L'implémentation sous test est alors directement « stimulée » par les générateurs de tests. Les 3 modules sont synchronisés grâce à des procédures de coordination permettant l'échange d'informations.

L'avantage de cette architecture est qu'elle est directe mais elle est rarement utilisée.

3.5.2 – Architecture distante

Comme son nom l'indique, l'architecture distante est conçue pour les systèmes distribués, répartis et distants.

Considérons l'architecture distante de la Figure 15 dans lequel le testeur inférieur devient cette fois externe au système.

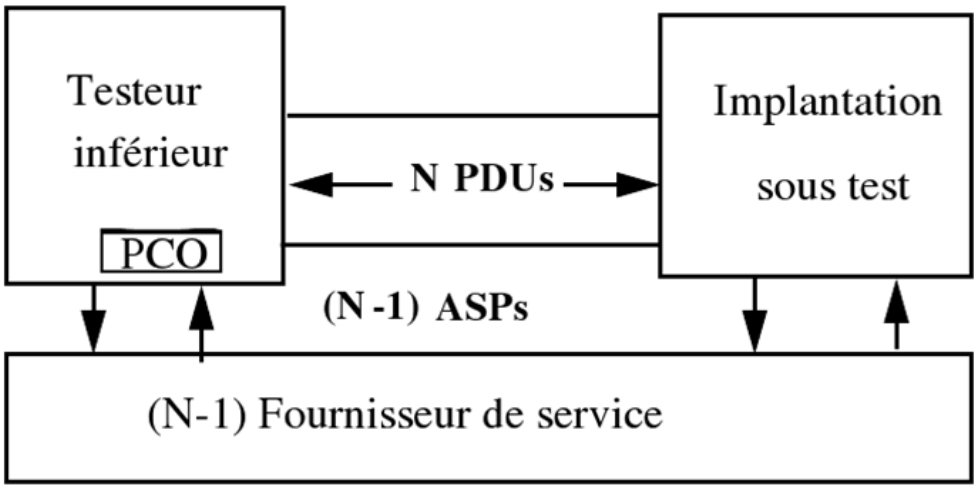


Figure 15: Architecture distante

La communication entre le testeur inférieur et le système est assurée grâce au service fourni par la couche N-1. Seul le PCO au dessous du testeur inférieur est disponible.
Aucun testeur supérieur n'est présent dans cette architecture. Cependant, pour la spécification des cas de test, certaines fonctions du testeur supérieur peuvent être utilisées si nécessaire.
Cette architecture a des limites de contrôle et d'observation.

3.6 – Outils de test

Dans cette section, nous détaillons quelques outils de génération de test, basés sur les méthodologies citées précédemment.

3.6.1 – TVEDA

L'outil TVEDA [58] est un outil élaboré au sein du Centre National d'Etudes des Télécommunications, et qui permet de produire des séquences de tests permettant de vérifier la conformité des systèmes.
Cet outil est composé d'une interface qui accepte les langages Estelle [39] ou SDL [40]. A partir de ces langages, des objectifs de tests peuvent alors être générés automatiquement. Puis les tests correspondant, décrites grâce au langage TTCN [37], sont générés par construction partielle du système de transition de la spécification.

3.6.2 – TGV

TGV (Test Generation with Verification technology) [6, 25, 24] est un outil de génération automatique de tests de conformité à travers une approche orientée objectif de test.

TGV a été développé en collaboration entre l'équipe Pampa de l'IRISA à Rennes et le laboratoire Verimag Grenoble.

Cet outil permet de générer des séquences de test, décrites sous forme TTCN, en combinant la spécification avec un objectif de test donné.

En premier lieu TGV construit une spécification sous forme d'IOLTS, décrivant le comportement observable dans son environnement. Ensuite, la construction des cas de test est guidée par l'objectif de test à l'aide d'un produit synchronisé. Les cas de test générés sont des IOLTS avec des états finaux "pass", "fail" et "inconclusive" représentant les verdicts.

3.6.3 – Agatha

Agatha est un outil réalisé par CEA [29]. A l'origine dédié aux spécifications Estelle et SDL, il permet aussi de générer des séquences de tests à partir du langage UML en passant par un modèle intermédiaire, appelé EIOLTS.

La génération de tests se passe via une simulation symbolique de l'ensemble d'états, ce qui permet de couvrir chaque chemin symbolique.

Dans l'exécution symbolique les paramètres d'entrées ne sont pas considérés numériquement mais sous forme de constantes symboliques. AGATHA permet alors de générer un arbre d'exécution symbolique couvrant tous les comportements de la spécification.

L'outil a été appliqué à de nombreux cas industriels avec succès.

3.6.4 – Outils de Webmov

Dans le cadre du projet Webmov [77], plusieurs outils de test de services Web ont été proposés, tel que WSInject [8]. Ce dernier consiste à perturber la communication entre l'application cliente et le service Web en injectant des fautes (d'interface et de communication) dans les requêtes SOAP envoyées au service Web, afin de tester la robustesse de ce dernier suivant le délai de réponse.

La méthode d'injection de fautes est basée sur un ensemble de règles. Celles-ci sont dédiées à la suppression, modification et à la duplication des messages SOAP. L'ensemble de fautes injectées est configurable via un fichier. Nous notons que l'outil est capable de tester à la fois les services Web simples et composés.

3.7 – État de l'art des tests appliqués aux services Web

L'ingénierie des services Web regroupe un ensemble de protocoles et d'activités permettant de développer des systèmes de communication de qualité. La complexité de tels systèmes nécessite d'utiliser des techniques de méthodes de test permettant d'évaluer leur qualité et fiabilité.

Nous présentons dans cette section un ensemble de travaux sur le test de services Web.

Dans cette partie, nous nous intéressons aux tests de robustesse, de conformité, de sécurité et aussi à la testabilité des services Web.

3.7.1 – Test de robustesse des services Web

Différents travaux ont abordé le test de robustesse.

Dans [54] et [46], la robustesse des services Web est testée en appliquant des mutations sur les messages de requête SOAP, et en analysant les réponses obtenues. De plus, si le service ne répond pas, il est considéré comme non robuste.

Dans [2], les services Web sont automatiquement testés en utilisant la description WDSL. Les cas de test sont construits pour analyser des types de données reçues et tester les dépendances d'opérations (analyse sur des types de paramètres donnés et les types de réponses reçues).

Dans [7], les auteurs testent l'interopérabilité des services Web en proposant d'augmenter la description WSDL via un diagramme PSM (UML2.0 Protocol State Machine), dont le but est de modéliser les interactions possibles entre un client et un service Web. Les cas de test sont générés à partir des PSM.

Les auteurs de [51] proposent une méthode automatique de test de robustesse des services Web. A partir de la description WSDL, la méthode utilise le framework "Axis 2" pour générer une classe cliente composée d'une liste de méthodes permettant d'invoquer toutes les opérations. Ensuite, les cas de test sont générés avec l'outil Jcrasher à partir de la classe précédente. Enfin, l'outil Junit est utilisé pour exécuter les cas de test.

Dans [76], la robustesse est testée en appliquant des injections de fautes au niveau des paramètres. Le service est robuste s'il retourne uniquement des réponses satisfaisant la description WSDL. La méthode présentée dans [31] discute sur une technique d'évaluation de la performance des services Web en prenant en compte la gigue d'un réseau.

L'article [4] présente l'outil WS-TAXI, obtenu en intégrant "SOAPUI" qui est un outil de test manuel permettant d'effectuer des tests sur des opérations de services Web, et "TAXI" qui permet de définir des instances XML à partir d'un fichier XSD. L'outil WS-TAXI permet

ainsi de générer des cas de test pour tester automatiquement des opérations en traitant toute la structure du service Web comme un fichier XSD.

3.7.2 – Test de conformité des services Web

Nous présentons ici quelques travaux sur le test de conformité des services Web simples et composés. Dans [27], les auteurs proposent une approche formelle pour la modélisation et le test de la coordination des services Web. La spécification de cette coordination de services est décrite en UML (diagrammes de composants). Elle présente des invocations successives des différentes opérations d'un même service. Cette spécification est traduite en un modèle LTS. Les cas de test sont ensuite générés suivant la relation de conformité IOCO.

Les auteurs de [42] proposent une approche de génération automatique de cas de test visant à vérifier la conformité d'une spécification d'un service composé vis à vis de son implémentation. Cette spécification est fournie par des diagrammes d'activités UML, Abstract BPEL ou BPMN. Ensuite, elle est transformée en oWFNs : une classe spéciale des réseaux de Pétri. oWFNs est utilisée pour la génération de cas de test. Par défaut, les auteurs ont considéré que l'implémentation d'un service composé est conforme à sa spécification si elle n'exclut aucun partenaire.

L'article [3] présente une méthode permettant le test de la conformité et l'interopérabilité pour une chorégraphie globale. Cette dernière définie les règles d'interactions à respecter afin de garantir l'interopérabilité.

Dans [12], les auteurs définissent une relation de conformité entre la composition en chorégraphie (WS-CDL) et celle en orchestration (WS-BPEL). Deux langages formels ont été définis : CLp et OL, décrivant respectivement la chorégraphie et l'orchestration. La vérification de conformité a été définit via une bi-simulation.

3.7.3 – Sécurité des services Web

Plusieurs modèles de sécurité dédiés au Services Web ont été conçus : nous citons par exemple le (WS-Security) qui définit une série d'extensions SOAP permettant d'assurer la sécurité des services Web via l'intégrité, la confidentialité et l'authentification des messages. Cependant, ces modèles annoncent un niveau d'abstraction élevé, ce qui ne présente pas une solution directe pour la communauté de test de services Web.

Dans ce contexte, quelques travaux ont été proposés.

Dans [32], les auteurs présentent une méthode qui consiste à valider les requêtes SOAP en filtrant les messages SOAP contenant des données malveillantes, et spécialement les attaques de type DoS. Les auteurs proposent aussi quelques solutions techniques permettant une

validation stricte de la description WSDL : utilisation des schémas XML, remplacement des valeurs "unbounded" par des nombres constants.

De son coté [47] propose une approche de test passif qui permet de collecter les traces des messages SOAP et de les analyser en intégrant des modules dans le moteur BPEL.

Cette approche se base sur le langage formel Nomad pour définir les règles de sécurité.

L'article [48], présente une méthode de test de sécurité avec des règles de sécurité temporelles exprimant l'obligation et l'interdiction, en se basant toujours sur Nomad. La spécification est ensuite augmentée par ces règles et l'outil « testGenIF » génère les cas de test demandés.

3.7.4 – Testabilité des services Web

La testabilité des services Web est un domaine de recherche qui n'est pas encore pleinement exploré. Cependant, quelques travaux ont été proposés sur la testabilité des services Web.

Dans [69], l'article présente une des premières études sur la testabilité des services Web. Il propose une méthode automatique d'évaluation de la testabilité de services Web existant via leur description WSDL. Deux facteurs de testabilité ont été étudiés : l'observabilité et la contrôlabilité. Le premier facteur est présenté comme une relation d'injectivité alors que le deuxième peut être considéré comme relation de surjectivité (cf. 3.2.2).

L'article [59] présente une méthode d'analyse pour les services Web. Cette méthode définie l'observabilité suivant les informations qui peuvent être observé durant l'exécution du service Web. Cette observation prend en compte les fautes SOAP retournées par le service Web.

Dans [23], l'article présente quelques problèmes d'observabilité et de contrôlabilité dans le cas des services Web composés. L'observabilité est définie suivant la possibilité du testeur d'observer la communication depuis des points d'observabilité. De même, la contrôlabilité est définie suivant la possibilité du testeur de contrôler la communication depuis des points de contrôle. Dans ce cadre, et en partant d'une spécification symbolique IOSTS, une méthode de test de conformité basé sur IOCO est proposée.

Chapitre 4

Méthodologies de test de services Web

4.1 – Introduction

Du fait des nombreux avantages qu'ils procurent, les services Web s'inscrivent clairement parmi les techniques de développement dites émergentes et qui sont de plus en plus utilisées aujourd'hui pour construire des applications orientées Web. De part leurs natures, ces services Web doivent être fiables, robustes et sécurisés.

Un des challenges apportés par le test de services Web concerne l'environnement SOAP dans lequel ceux-ci "baignent". Celui-ci complique le contrôle et l'observation des messages émis par et vers le client. Ainsi, n'importe quel message habituellement observable, comme une réponse classique, est encapsulé puis émis vers le client via le protocole SOAP. D'après le protocole SOAP 1.2, les exceptions utilisées en programmation orientée objet sont traduites en éléments XML appelés fautes SOAP.

Nous nous intéressons, dans ce chapitre, au test de robustesse et de sécurité de services Web simples (non composés). Nous considérons le cas de figure où seule la spécification WSDL du service est connue. Donc nous étudions un service comme une boîte noire à partir desquelles seuls les messages SOAP (requêtes et réponses) sont observables via les interfaces.

4.2 – Robustesse

Nous avons effectué nos travaux sur deux types de services non composés, en tenant compte de la notion de session : les services Web persistants et non persistants.

Dans ce qui suit, nous allons introduire notre méthodologie de test sur les services Web non persistants.

4.2.1 – Services Web non persistants

Beaucoup de services Web sont actuellement proposés dans des registres UDDI. Pour la plupart d'entre eux, les renseignements sur leurs structures comme les connexions aux bases de données et le code interne du service Web, ne sont pas donnés.

Ainsi, en gardant à l'esprit ces caractéristiques, dans [66], nous avons proposé une étude sur la robustesse des services Web non persistants, en utilisant seulement leurs descriptions WSDL.

Notre méthode consiste à évaluer automatiquement la robustesse d'un service Web non persistant par rapport aux opérations déclarées dans sa description WSDL, en examinant les réponses reçues lorsque ces opérations sont invoquées avec des aléas qui sont des valeurs inhabituelles et aléatoires connus dans le test logiciel.

Pour cela, nous considérons un service Web comme étant :

Définition : Un service Web WS est un composant qui peut être appelé via un ensemble d'opérations $OP(WS) = \{op_1, \dots, op_k\}$, avec op_i défini par une relation $op_i : (param_1, \dots, param_m) \rightarrow (resp_1, \dots, resp_n)$ où $(param_1, \dots, param_m)$ sont les types de paramètres et $(resp_1, \dots, resp_n)$ sont les types de réponses de l'opération op_i .

Pour une opération op, nous définissons $P(op)$ l'ensemble des tuples de paramètres de op, $P(op) = \left\{ \begin{matrix} (p_1, \dots, p_m) \mid \\ p_i \text{ est une valeur de type } param_i \end{matrix} \right\}$. De même, R(op) est l'ensemble des tuples de réponses,

$R(op) = \left\{ \begin{matrix} (r_1, \dots, r_n) \mid \\ r_j \text{ est une valeur de type } resp_j \end{matrix} \right\} \cup \{r \mid r \text{ est de type faute SOAP}\} \cup \{\in\}$.

$\in$ représente une réponse vide (ou pas de réponse).

Chaque paramètre/réponse peut être de type simple (integer, float, String...) ou complexe (arbre, tableau, faute SOAP, objet composé de types simples ou complexes...) et chaque type peut être fini (Integer,...) ou infini (String,...).

Un exemple de service Web est illustré en Figure 16 par des diagrammes UML. Ceux ci décrivent deux opérations : "getPerson" qui retourne un objet "Person" en donnant un paramètre de type "String" et l'opération "divide" qui retourne le résultat de la division euclidienne de deux entiers.

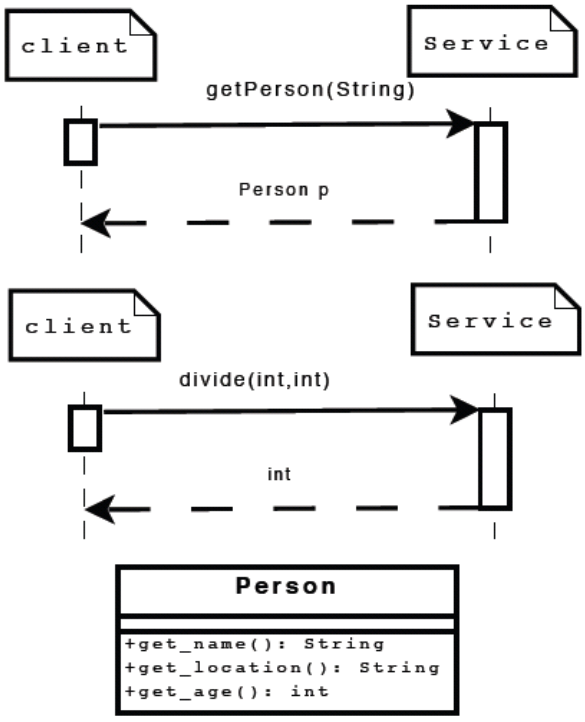


Figure 16: Exemple de service Web

Étude de la robustesse de service Web

Tout comme dans (IEEE Standard Glossary of Software Engineering Terminology 1999), nous considérons qu'une opération est robuste lorsque celle-ci ne se termine pas de façon anormale lorsqu'elle est invoquée par des aléas. Ceci implique qu'un service Web robuste doit agir conformément à sa spécification en présence d'aléas.

Étant donné que les services Web sont imbriqués dans un environnement SOAP, lorsque nous appelons une opération nous faisons en fait un appel à un couple (processeur SOAP, service Web) et nous obtenons une réponse de ce dernier.

Un processeur SOAP est en général seulement un élément d'une architecture plus complète d'un framework comme Apache Axis ou Sun Metro JAXWS.

Nous avons analysé et déduit que ces processeurs SOAP affectent l'observabilité dans le cas où une opération se termine anormalement. En effet, certains processeurs SOAP (comme celui d'Axis2 par exemple) génèrent eux-mêmes une faute SOAP à la place du service Web dans certains cas. Un tel comportement complique alors le test car on peut estimer que l'opération réagit correctement et est robuste alors que c'est une autre partie logicielle qui répond. De plus, certains processeurs SOAP génèrent des fautes SOAP dans certains contextes et d'autres ne le font pas. Donc nous avons besoin de séparer le comportement du processeur SOAP de celui du service Web.

Le profil WS-I basic permet de différencier les fautes SOAP générées par le processeur SOAP des fautes SOAP construites par un service Web. Lorsqu'une exception est levée, un service Web doit générer lui-même une faute SOAP. Dans ce cas, la cause de la faute SOAP est composée par "SOAPFaultException". Autrement, les fautes SOAP sont générées par les processeurs SOAP.

Ceci est formalisé dans la définition suivante:

Définition

Soit WS un service Web. Une opération op_i : $(resp_1, \dots, resp_n) \rightarrow (param_1, \dots, param_m)$ ∈ OP(WS), est robuste ssi ∀ v ∈ P(op), r = op(v) avec :

– r = $(r_1, \dots, r_n)$ tels que $r_i = resp_i$,

– ou r est une faute SOAP composée de la cause "SOAPFaultException".

En conséquence, il nous semble intéressant d'analyser l'observabilité des services Web avec ces aléas afin d'en déterminer les types les plus pertinents pour le test de robustesse.

Analyse du comportement de service Web en présence d'aléas

Les services Web sont uniquement accessibles via une couche SOAP qui réduit leur observabilité. Donc, nous avons analysé le comportement des services Web avec la présence d'aléas, afin de déterminer si ceux-ci peuvent être réellement exploités pendant la phase de test.

Nous avons en effet observé que de nombreux aléas classiques étaient bloqués par le processeur SOAP et donc complètement inutiles.

En résumé, nous avons déduit que tout aléa utilisé pour appeler une opération qui ne respecte pas à la lettre la description WSDL du service associé, est bloqué par le processeur SOAP, et n'est donc pas donné à l'opération.

De ce fait, le test ne pas être effectué. Ainsi, les aléas suivants « Remplacement de types de paramètres », « Inversion de types de paramètres », « Ajout/injection de types de paramètres », « Suppression de types de paramètres » sont bloqués par le processeur SOAP et par conséquent inutiles.

Le processeur SOAP retourne alors toujours la même faute SOAP commençant par "Client" qui signifie que les valeurs des paramètres données sont incorrectes.

Le seul aléa pertinent est alors l'« Invocation par des valeurs inhabituelles » : cet aléa est bien connu dans le test du logiciel [44], et est utilisé pour appeler une opération op_i : $(resp_1, \dots, resp_n) \rightarrow (param_1, \dots, param_m)$ avec des valeurs $(p_1, \dots, p_m) \in$ P(op), tel que chaque paramètre p_i a le type $param_i$.

Mais ces valeurs prédéfinies sont inhabituelles. Par exemple null, "", "$", "*" sont des valeurs inhabituelles de "string". Celles-ci sont acceptées par le processeur SOAP car elles vérifient la description WSDL. Ces valeurs inhabituelles peuvent être utilisées pour le test de la robustesse de service Web.

Par conséquent, l'unique aléa permettant de tester des services Web en boîte noire est l' "invocation des opérations de services Web avec des valeurs inhabituelles".

Après avoir analysé le comportement des services Web en présence de différents aléas, nous détaillons par la suite notre méthode de test.

Méthode de test de services non persistants

Notre méthode vise à tester ces deux caractéristiques : l'existence et la robustesse.

– *Existence de toutes les opérations de service Web* :

Pour toute opération op_i : $(resp_1, \dots, resp_n) \rightarrow (param_1, \dots, param_m) \in$ OP(WS), nous construisons des cas de test pour vérifier si l'appel de l'opération correspond à sa description

dans le fichier WSDL. Ainsi, les cas de test invoquent l'opération op avec plusieurs valeurs $(p_1, \dots, p_m) \in \mathrm{P}(op)$.

Nous considérons qu'une opération op est existante si elle renvoie une réponse r_i, avec r pouvant être une réponse classique $(r_1, \dots, r_n)$ tel que le type de chaque valeur r_i correspond à $resp_\mathrm{i}$ avec $resp = (resp_1, \dots, resp_n)$, ou alors pouvant être une faute SOAP dont la cause est différente de "Client" et de "the endpoint reference not found". Cette première cause signifie que l'opération est appelée avec un mauvais type de paramètre. La deuxième cause, signifie que le nom de l'opération n'existe pas. Sinon, op n'est pas existante comme il est indiqué dans le fichier WSDL.

– *Robustesse de toutes les opérations de service Web* :

Pour toute opération $op_i : (resp_1, \dots, resp_n) \rightarrow (param_1, \dots, param_m) \in \mathrm{OP(WS)}$, nous construisons des cas de test afin de vérifier si l'appel d'opération op se termine correctement malgré l'utilisation d'aléas. Selon notre analyse sur les aléas, le plus pertinent des aléas correspond à l'invocation de op avec des valeurs inhabituelles $(p_1, \dots, p_m) \in \mathrm{P}(op)$ où op_i a le type $param_\mathrm{i}$.

Pour cela, nous construisons des cas de test en utilisant l'aléa « valeurs inhabituelles ». Un ensemble de valeurs prédéfinies, noté V, est donc employé. Celui-ci est composé pour chaque type de données, d'une liste XML de valeurs spécifiques qui ont été choisies afin d'effectuer des appels avec des valeurs inhabituelles. Nous utilisons des valeurs connues dans le test de logiciel, qui sont supposées avoir un taux de déclenchement d'erreurs élevé (Kropp et al. 1998).

Nous notons V(t) l'ensemble des valeurs de type t qui sont utilisées pour invoquer une opération de service Web. Par exemple, la Figure 17 décrit quelques valeurs utilisées pour le type "String" et pour le type "tabular de "simple-type". Pour un "tabular" de "String", nous utiliserons donc les valeurs suivantes: "tabular" vide, "tabular" avec éléments vides et plusieurs "tabular" construite avec V (String).

Lors du test, une opération robuste op doit retourner soit une réponse "classique", soit une faute SOAP dont la cause est égale à "SOAPFaultException".

Nous considérons qu'un service Web n'est pas robuste si aucune réponse n'est observée ou si un autre type de fautes SOAP construit par le processeur SOAP est reçu. Afin de tester la robustesse des opérations, nous définissons l'hypothèse suivante sur les services Web.

```
<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>
```

(a) 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>
```

(b) V(tabular)

Figure 17: Exemple de valeurs inhabituelles

Hypothèse d'observabilité d'une opération de service Web : Nous supposons que chaque opération du service Web, décrite dans des fichiers WSDL, renvoie des réponses non nulles. Cette hypothèse nous permet d'observer une opération qui « crashe » et ne retourne pas de message.

Avant de décrire la génération de cas de test, nous définissons un cas de test par :

Définition Soit WS un service Web et $op_i : (resp_1, \dots, resp_n) \rightarrow (param_1, \dots, param_m) \in$ OP(WS) une opération de WS. Un cas de test T est un arbre composé de nœuds $n_0, \dots, n_m$ où n_0 est le nœud racine et chaque nœud de fin est étiqueté par un verdict dans $\{ pass, inconclusive, fail \}$.

Une branche est étiquetée soit par *op_call(v)*, soit par *op_return(r)* avec :

– $v \in P(op)$, une liste de valeurs de paramètres utilisée pour invoquer *op*,

– $r = (c, soap\_fault)$ une faute SOAP composée de la cause *c* ou alors $r = (r_1, \dots, r_m)$ la liste des réponses où $r_j = (v_j, t_j)$ avec v_j une valeur et t_j le type de v_j .

Nous notons aussi * un symbole correspondant à n'importe quelle valeur. (*, *t*) est une réponse avec un type *t*.

La génération des cas de test est illustrée en Figure 18.

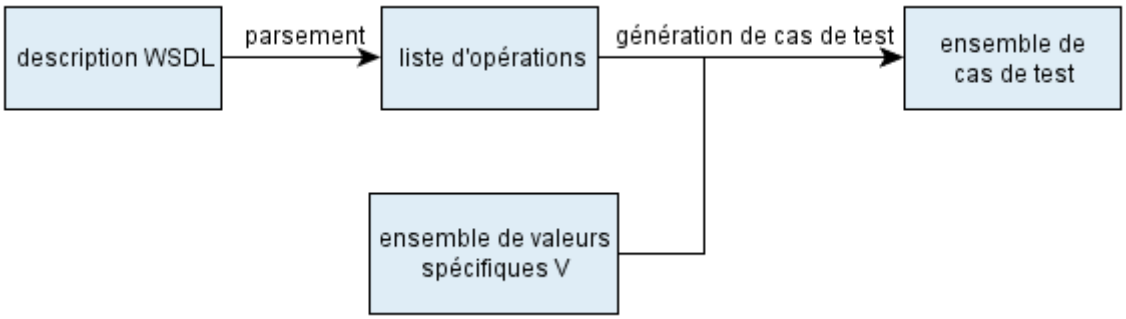


Figure 18: Génération des cas de test

Pour un service Web WS, les cas de test sont générés grâce aux étapes suivantes :

1) la description WSDL est analysée, ce qui donne une liste d'opérations L = $\{op_1, \dots, op_l\}$,

2) pour chaque opération op_i : $(resp_1, \dots, resp_n) \rightarrow (param_1, \dots, param_m) \in$ L, nous construisons à partir de l'ensemble V la liste de tuple Value(op) = {(V_1,…,V_m) ∈ V ($param_1$) x …x V ($param_m$)}. Si le type de paramètre est complexe (tabular, objet,...), les valeurs sont construites. Nous utilisons également une heuristique pour évaluer et éventuellement pour réduire le nombre de tests selon le nombre de tuples Value(op),

3) pour chaque opération op_i : $(resp_1, \dots, resp_n) \rightarrow (param_1, \dots, param_m) \in$ L, nous construisons le cas de test TC(op)=$\cup_{V \in Value(op)}$ { n_0.op_call(v): n_1:op_return(r_1).pass, n_0.op_call(v).n_1.op_return(r_2).pass, n_0.op_call(v).n_1.op_return(r_3).inconclusive}

où r_1 = (*, t) avec t = ($resp_1$, … , $resp_n$), r_2 = (c, soap_fault) cause="SOAPFaultException";

r_3 = (c, soap_fault) cause $\notin$ {"client", "SOAPFaultException", "the endpoint reference not found"}. Toute autre branche correspond à un cas d'échec fini par "fail",

4) la suite de tests est obtenue par TC = $\cup_{op \in L}$ { $TC(op)$}

Pour illustrer notre méthode, nous présentons les cas de test par deux figures : l'une décrivant le test de l'existence d'opération (Figure 19) et l'autre décrivant le test de robustesse (Figure 20). Dans TC, chaque arbre invoque une opération avec les valeurs de paramètres en fonction de la description WSDL.

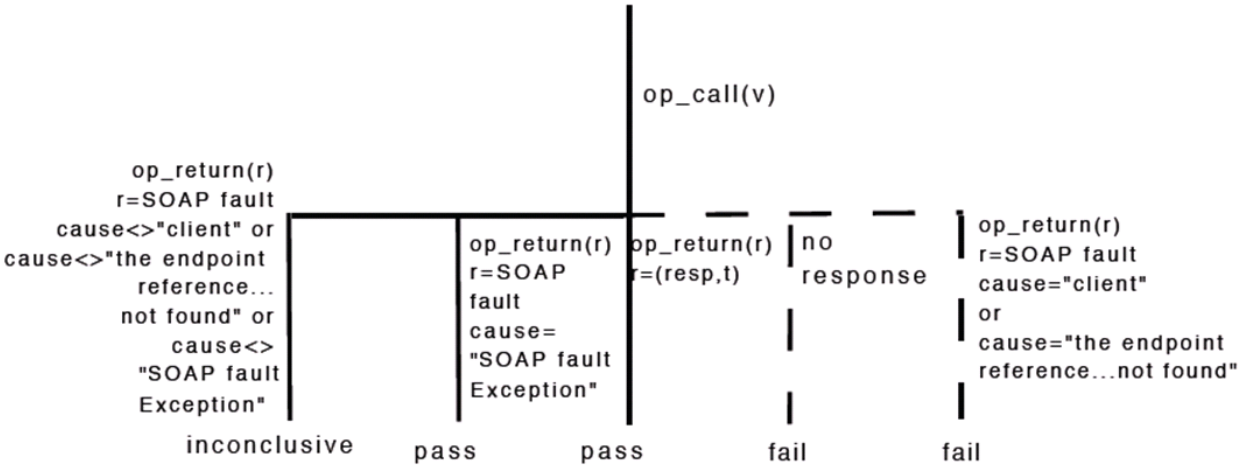


Figure 19: Schéma de cas de test pour le test d'existence d'opération

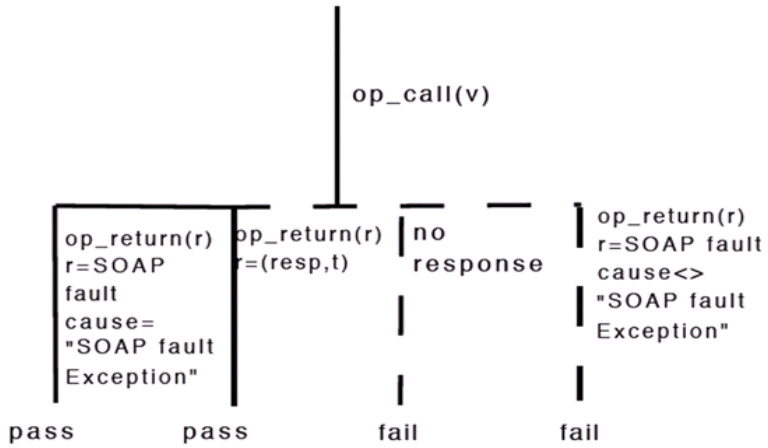


Figure 20: Schéma de cas de test pour le test de la robustesse d'opération

Dans la Figure 19, si la réponse n'est pas une faute SOAP et si son type est bien décrit dans le fichier WSDL, le verdict local est "pass". Si la réponse est une faute SOAP dont la cause est égale à "SOAPFaultException" alors l'opération gère elle-même les exceptions et le verdict local est "pass". Si la réponse est une faute SOAP dont la cause n'est pas dans {"SOAPFaultException","client", "the endpoint reference not found"} dans ce cas l'opération existe mais elle ne gère pas les exceptions. Si une opération se termine anormalement et qu'une faute SOAP est retournée par le processeur SOAP, alors cette opération n'est pas robuste et dans ce cas le verdict local est "inconclusive". Sinon, le verdict est "fail".

Exécution des cas de test

Les cas de test sont générés et exécutés avec l'architecture de test illustrée par la Figure 21.

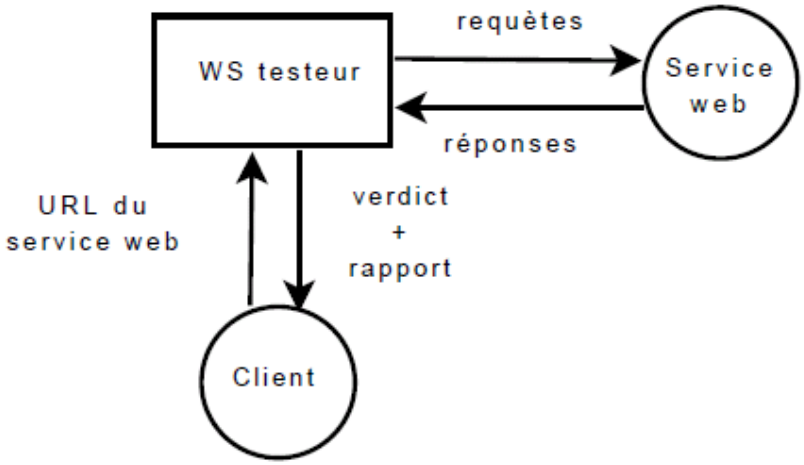


Figure 21: Architecture de test

A partir de l'url du service Web introduit par le client, le testeur construit une suite de tests pour chaque opération figurant dans la description WSDL. Les valeurs utilisées dans cette suite de tests sont choisies selon les types de paramètres de chaque opération, en se basant sur l'ensemble des valeurs inhabituelles. Après l'exécution des cas de test, un verdict final avec un rapport sont transmis au client.

Ce framework a été mis en œuvre dans un outil appelé WS-AT qui a été développé dans le cadre du projet Webmov [77] (voir Figure 22).

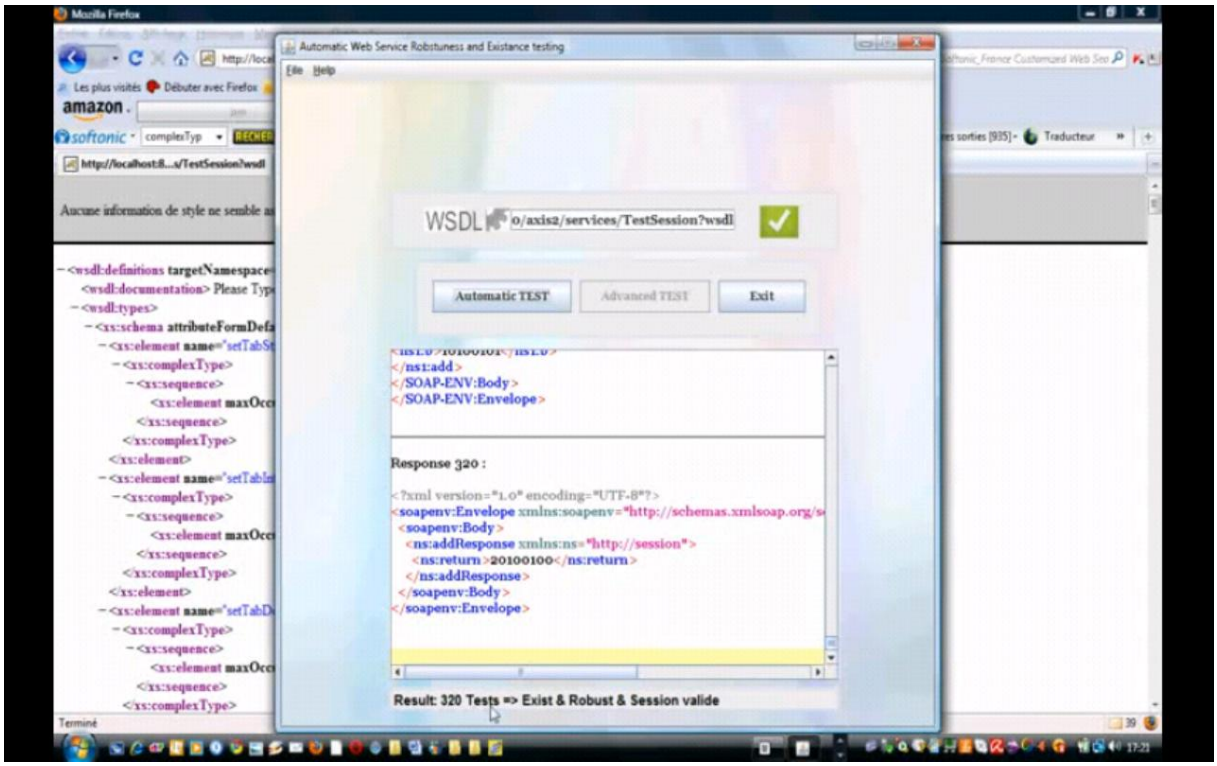


Figure 22: Capture d'écran de l'outil WS-AT

Notre testeur correspond à un service Web qui reçoit l'URL associée au service Web à tester. Celui-ci construit les cas de test décrits précédemment, puis appelle successivement le service Web. Une fois les cas de test exécutés, l'analyse des réponses obtenues commence, puis arrive le verdict de test.

Pour donner le verdict final, le testeur exécute chaque cas de test en parcourant l'arbre des cas de test. Si une branche est complètement exécutée alors un verdict local est obtenu. Sinon, un verdict local "fail" est retourné. Soit t un cas de test et vl(t) ∈ {pass ; fail ; inconclusive} le verdict local associé.

Le verdict final est donné par :

Définition Soit WS un service Web et TC un cas de test. Le verdict de test sur TC, noté Verdict(WS)=TC est :

– "pass", si pour tout t ∈ TC; vl(t) = pass. Le verdict "pass" signifie que toutes les opérations de WS existent et sont robustes,

– "inconclusive", s'il existe t ∈ TC tel que vl(t) = inconclusive, et il n'existe pas de t′ ∈ TC tels que vl(t′) = fail. Ce verdict signifie que le service Web n'est pas robuste mais l'ensemble de ces opérations existent,

– "fail", s'il existe t ∈ TC tels que vl(t) = fail.

Bénéfice de l'approche

Dans notre méthode, comme dans [2], nous vérifions si chaque opération d'un service Web décrite dans le fichier WSDL existe. Par rapport aux travaux de [54] et [51], nous analysons et déterminons les aléas qui peuvent être réellement utilisés. Nous séparons également le comportement du service Web de celui du processeur SOAP, ce qui augmente la détection des erreurs de robustesse. Par rapport à [51], notre architecture de test effectue des appels de service Web directement, ce qui permet d'analyser les réponses SOAP et en particulier les fautes SOAP. La méthode présentée dans [51] ne teste pas directement les services Web, mais plutôt les clients qui appellent les méthodes les services web.

Nous avons expérimenté avec succès cette méthode à quelques services Web déjà déployés et nous avons détecté des résultats de robustesse pour la plupart d'entre elles. Les résultats obtenus sont présentés dans la Figure 23.

ws	nombre d'opérations	nombre de paramètres	nombre de tests	fail
PrimeNumbersGenerator	1	1	10	7
YoutubeDownloader	1	1	10	9
sendSMS	2	2,4	22	0
MapIPtoCountry	2	1,1	20	4
Number To Words	1	1	10	6
LocalTimeByZipCode	1	1	10	0
TextToBraille	2	2,2	22	20
TConversions	2	1,2	22	11
StrikeIron	1	3	12	0
VideoWs	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 23: Résultat de test de robustesse

Dans la plupart des cas, les opérations ne déclenchent pas d'exceptions. Cette expérimentation a confirmé les avantages suivants :

– Efficacité : l'outil est facile d'utilisation puisque le testeur peut tester automatiquement la plupart des services Web déployés sur Internet avec seulement l'URL de description WSDL (bien sûr pour ceux qui n'utilisent pas d'authentification). Formellement, la couverture de test de notre méthode est tout à fait simple : nous la vérifions après l'appel d'une opération avec des aléas, si celui-ci ne se "crash" pas. Toutefois, la méthode permet de détecter de nombreux problèmes, comme l'accessibilité d'opération (l'opération n'existe pas, ne gère pas le bon paramètre), les problèmes de gestion d'exception (absence du code "try ... catch", des fautes SOAP non envoyées), et les problèmes d'observabilité (opérations ne répondant pas). Cette méthode est aussi extensible puisque l'ensemble prédéfini de valeurs peut être mis à jour facilement,

– Coût de test : la méthode est extensible puisqu'elle ne fonctionne pas par des séquences d'appels.

Ainsi, le coût de test dépend seulement du nombre de valeurs prédéfinies utilisées pour le test. Cependant, cette expérimentation a également montré quelques inconvénients :

– l'ensemble V de paramètres utilisés pour le test a été amélioré pour détecter plus d'échecs. Mais, il serait plus intéressant de proposer une analyse dynamique pour construire la liste de paramètres la plus appropriée pour chaque service Web,

– pour éviter l'explosion de cas de test, les paramètres sur V sont choisis aléatoirement. Une meilleure solution serait de choisir ces paramètres selon la description des opérations,
– nous avons supposé que les opérations du même service Web sont indépendantes. Or, avec des opérations dépendantes, le service Web créé des sessions avec des clients, devient persistant dans le serveur Web et prend différents états pendant la session. Notre outil peut détecter des problèmes de robustesse, mais n'est pas totalement exhaustif car l'ensemble des états n'est pas couvert. Par exemple, si une opération vérifie qu'une session existe avant d'employer des valeurs de paramètres, la méthode ne peut pas parcourir les états internes du service.

4.2.2 – Méthodologies de test des services Web persistants

Dans [68] et [67], nous distinguons les services Web persistants (appelés aussi stateful) de ceux non persistants. En effet, lors de la conception et le développement des services Web, il arrive de devoir introduire, dans le cas de service persistant, la notion de session. Nous rappelons que la session permet de conserver des informations au travers de plusieurs invocations provenant d'un même client jusqu'à la libération de cette session. Donc, un tel service est gardé en mémoire à travers une session et possède un état interne qui évolue au fil des séquences d'appels.
Par exemple: les Services Web utilisant les *shopping carts* ou bien commençant par une étape d'identification sont persistants.
Nous formalisons la notion d'état grâce au modèle STS car ce dernier permet d'incorporer explicitement la notion de données, contrôler la spécification, modéliser de manière concise des systèmes de transitions infinis et donner une réponse possible au problème de l'explosion combinatoire.
Un exemple de spécification est donné en Figure 24. Celle-ci décrit un service Web avec lequel il est possible de s'identifier, de chercher des livres et de les acheter. La base de données associée est donnée en Figure 26.

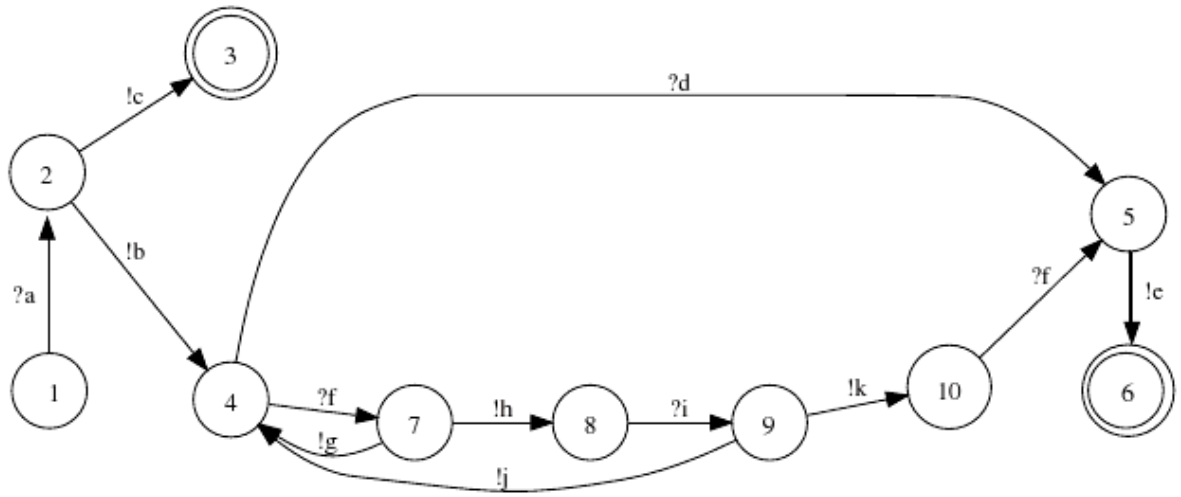


Figure 24: Spécification d'un service Web persistant

?a	login<String s>
!c	login_return<String s2> [S2=="invalid" && s ≠ BD.account.name]
!b	login_return<String s2> [S2=="valid" && s == BD.account.name]
?d	disconnect()
!e	disconnect_return<String r> [r=="disconnected"]
?f	list_book<String title>
!g	list_book_return<String r> [r==soapFault("invalid book")&& title ≠ BD.book.title]
!h	list_book_return<String isdn> [isdn==BD.book. isdn(title) && title == BD.book.title]
?i	buy<String i> i:=isdn
!j	buy_return<boolean b> [b==false && fund ≤ book.isdn(isdn).price
!k	buy_return<boolean b> [b==true && BD.account.name(s).fund > book.isdn(isdn).price]

Figure 25: Table des symboles de spécification

account		
name	fund	
"name"	10	
book		
title	isdn	price
"fundation"	"0553293354"	20

Figure 26: Table des symboles de spécification

Analyse de la robustesse des services Web persistants

Nous étendons, dans [68] et [67], le test de robustesse aux services Web persistants ayant différents états internes au fil des appels. Tout comme précédemment, certains aléas sont inutiles car bloqués par les processeurs SOAP. Nous avons étudié les aléas suivants: Modification du nom d'une opération, Remplacement/Ajout du nom de l'opération. Etant donné un service Web WS et STS sa spécification:

• *Modification du nom d'une opération*: cet aléa correspond à la modification aléatoire du nom d'une opération op ∈ OP (WS) tel que le nom modifié *op_modif* n'est pas celui d'une opération existante (*op_modif* ∉ OP (W S)). En appliquant ce type d'aléa, nous obtenons toujours une faute SOAP composée de la cause "Client". Celle-ci est construite par les processeurs SOAP et signifie que l'appel client est erroné. L'appel est donc bloqué par les processeurs SOAP et n'est pas propagé vers le service web. Le test ne pouvant pas être effectué, nous considérons que cet aléa est inutile,

• *Remplacement du nom /Ajout d'une opération*: soit *l* un état de la spécification STS avec les transitions sortantes (l , l_1 , "op_1 (p_{l1} , ..., p_{m1})", φ_1 , ρ_1), ... ,

(l , l_n , "op_n (p_{ln} , ..., p_{mn})", φ_n , ρ_n) modélisant des appels d'opérations. S'il existe une opération op_i : $(resp_1, \dots, resp_n) \rightarrow (param_1, \dots, param_m) \in$ OP (WS) qui n'est pas appelée depuis *l* (op ∉ {op_1 , ..., op_n }), cet aléa a pour but de forcer l'appel de l'opération op à la place d'une autre. En appelant op, on ne peut bien sûr qu'obtenir une réponse d'op. Si dans notre exemple de la Figure 24, nous remplaçons l'opération "login" par "buy", nous n'obtiendrons pas une réponse de "login" mais de "buy". On ne peut donc pas juste remplacer/ajouter un nom. Ainsi, si l'opération op est robuste, selon la définition vue dans le paragraphe 3.1, la réponse attendue est soit une réponse "classique" ($r_1, \dots, r_n$) avec r_i de type $resp_i$, soit une faute SOAP dont la cause est "SOAPFaultException".

Cet aléa implique donc la modification de la spécification pour la compléter sur les réponses attendues. Cette complétude est détaillée lors de la génération des cas de test.

Il existe bien sûr d'autres aléas comme le remplacement du nom du service appelé ou la modification aléatoire des messages SOAP. Ces aléas sont généralement utilisés pour le test de services Web composés afin d'observer le comportement global des partenaires inclus dans la composition. Ceux-ci ne sont pas des plus intéressants pour tester un service Web unique.

A noter que le profil WS-I basic ne permet pas la surcharge d'opération donc cet aléa n'est pas étudié.

Après avoir analysé le comportement des services Web persistants en présence d'aléas, nous présentons par la suite notre méthode de test.

Méthode de test de services persistants

L'analyse précédente sur la robustesse des opérations nous montre que les aléas les plus pertinents sont le "Remplacement du nom /Ajout d'une opération" et l'"Utilisation de valeurs inhabituelles". Ces deux aléas sont donc utilisés pendant la génération des cas de test. La méthode est illustrée en Figure 27.

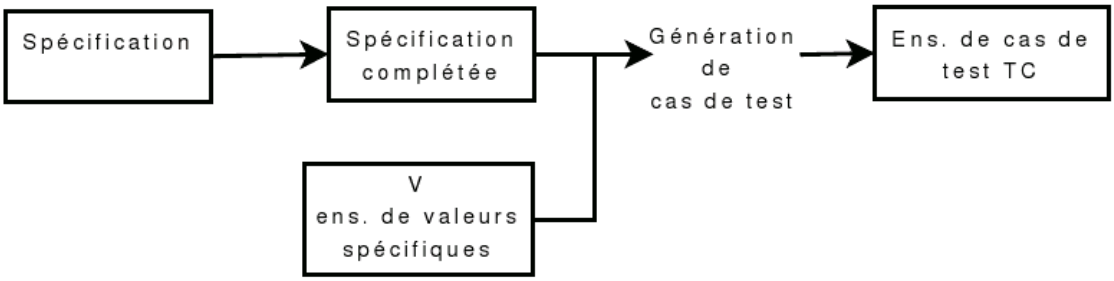


Figure 27: Génération des cas de test

La spécification est premièrement complétée sur les opérations qui peuvent être appelées en utilisant l'aléa "Remplacement du nom /Ajout d'une opération". Elle est aussi complétée afin de modéliser les comportements incorrects, le problème de la quiescence des états (états bloqués après un timeout) ainsi que l'ensemble des réponses incorrectes.

Les cas de test sont ensuite générés en utilisant l'aléa "Utilisation de valeurs inhabituelles".

Complétion de la spécification

Comme énoncé précédemment, nous complétons la spécification pour y injecter l'aléa "Remplacement du nom /Ajout d'une opération" et pour décrire tous les comportements corrects et incorrects du service Web. Soit donc le service Web WS et sa spécification STS $= < L, l_0, Var, var_0, I, S \rightarrow >$. Le STS est complété grâce aux étapes suivantes:

1. remplacement des conditions sur base de données par des valeurs réelles,
2. ajout du verdict "pass" dans les états finaux, ce qui illustre que pour atteindre cet état final, un comportement correct a été exécuté,

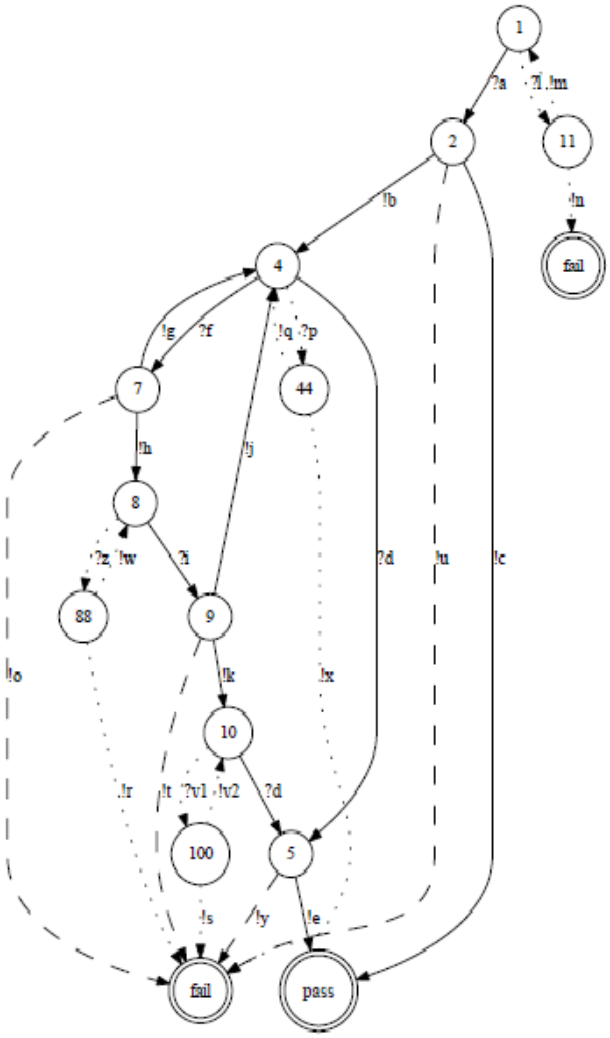
3. complétion sur les appels d'opérations: $\forall (l, l', op(p_1, ..., p_m), \varphi, \rho) \in \rightarrow$ nous ajoutons $\forall op_i \neq op \in OP(WS)$, $l \xrightarrow{op_i} l' \notin \rightarrow$ et $l' \xrightarrow{op_i\_return(r1)\ [r1 \neq ((c1, soapfault), c1 \neq SOAPFalutException)]} l \notin \rightarrow$. Ceci correspond à l'aléa "Remplacement du nom /Ajout d'une opération". D'après la définition de robustesse, si à partir d'un état symbolique une opération non spécifiée est appelée et que la réponse obtenue est soit composée d'une valeur "classique" soit d'une faute SOAP dont la cause est "SOAPFaultException" alors le service Web a accepté l'appel et a répondu de façon robuste. Il revient ensuite à l'état précédent l'appel,

4. complétion sur la réception de réponses incorrectes: $\forall\ l \in$ L tel que l a des transitions sortantes $(l, l_1, op\_return(r_1), \varphi_1, \rho_1), \ldots, (l, l_n, op\_return(r_n), \varphi_n, \rho_n)$ nous ajoutons:

(1) (l, fail, δ, ∅, ∅), (2) (l, fail, op_return(r), ¬(φ_1 ∨...∨ φ_n), ∅).

Cette complétion modélise le manque de robustesse suivant les réponses observées. (1) Si l'état est quiescent (bloquant), le service Web est figé et l'opération appelée n'est pas robuste. (2) Si l'opération appelée ne retourne pas une réponse spécifiée, alors le comportement du service Web est incorrect en présence d'aléas et donc l'opération n'est pas robuste.

En partant de la spécification des Figures 25 et 26, la spécification complétée est obtenue en Figure 28. Les transitions modélisées par des pointillés représentent l'appel d'opérations non spécifiées. Celles illustrées avec des traits hachurés correspondent aux réponses modélisant un comportement erroné.



!b	login_return<String s2> [S2=="valid" && s == "name"]
!g	list_book_return<String r> [r==soapFault("invalid book")&& title ≠ "fundation"]
!h	list_book_return<String isdn> [isdn=="0553293354" && title == "fundation"]
!j	buy_return<boolean b> [b==false && fund ≤ 20]
!k	buy_return<boolean b> [b==true && fund > 20]
?l	list_book<String title>|disconnect()|buy<String isdn>
!m	list_book_return(r)|disconnect_return(r)|buy_return(r) r=String|r=Boolean'|r=(c,soapFault) c="RemoteException"
?n	δ|list_book_return(r)|disconnect_return(r)| buy_return(r) r=(c,soapFault) c≠"RemoteException"
?p	login<String s>|buy<String isdn>
!q	login_return(r)|buy_return(r) r=String|r=(c,soapFault) c="RemoteException"
!x	δ|login_return(r)|buy_return(r) r=(c,soapFault) c≠"RemoteException"
?z	list_book<String title>|disconnect()|login<String s>
!w	list_book_return(r)|disconnect_return(r)| login_return(r) r=String|r=(c,soapFault) c="RemoteException"
!r	δ|list_book_return(r)|disconnect_return(r)| login_return(r) r=(c,soapFault) c≠"RemoteException"
?v1	list_book("title")|buy("isdn")|login("name")
!v2	list_book_return(r)|buy_return(r)|login_return(r) r=String|r=Boolean|r=(c,soapFault) c="RemoteException
!s	δ|list_book_return(r)|buy_return(r)|login_return(r) r=(c,soapFault) c≠"RemoteException"
!u	δ|login_return(r) [(S2 ≠ "invalid" || s =="name") && (S2 ≠ "valid" ||s ≠ "name")]
!y	δ|disconnect_return(r) [r≠"disconnected"]
!o	δ|list_book_return(r) [(r≠soapFault("invalid book")||title=="fundation") && (r≠"0553293354" || title ≠ "fundation")]
!t	δ|buy-return(r) [(b ≠ false || fund > 20)&&(b ≠ true || fund ≤ 20)]

Figure 28: Spécification complétée

Avant de décrire leur génération, nous définissons un cas de test par :

Définition Soit WS un service Web modélisé par STS $= < L, l_0, Var, var_0, I, S \rightarrow >$. Un cas de test T est un arbre déterministe et acyclique où chaque nœud terminal est étiqueté par un verdict dans {pass, fail}. Une branche est étiquetée soit par (op(v), φ, ρ) soit par (op_return(r), φ, ρ) soit par δ avec:

- v ∈ P(op), une liste de valeurs utilisée pour invoquer op,
- r = (c, soap_fault) une faute SOAP composée de la cause c,
- r = (r_1 , …, r_m) une réponse valide,
- φ et ρ une condition et une mise à jour sur l'ensemble des variables de Var respectivement,
- δ représentant la quiescence d'un état, (état bloquant après un timeout).

Les cas de test sont construits grâce à l'algorithme illustré dans la Figure 29.

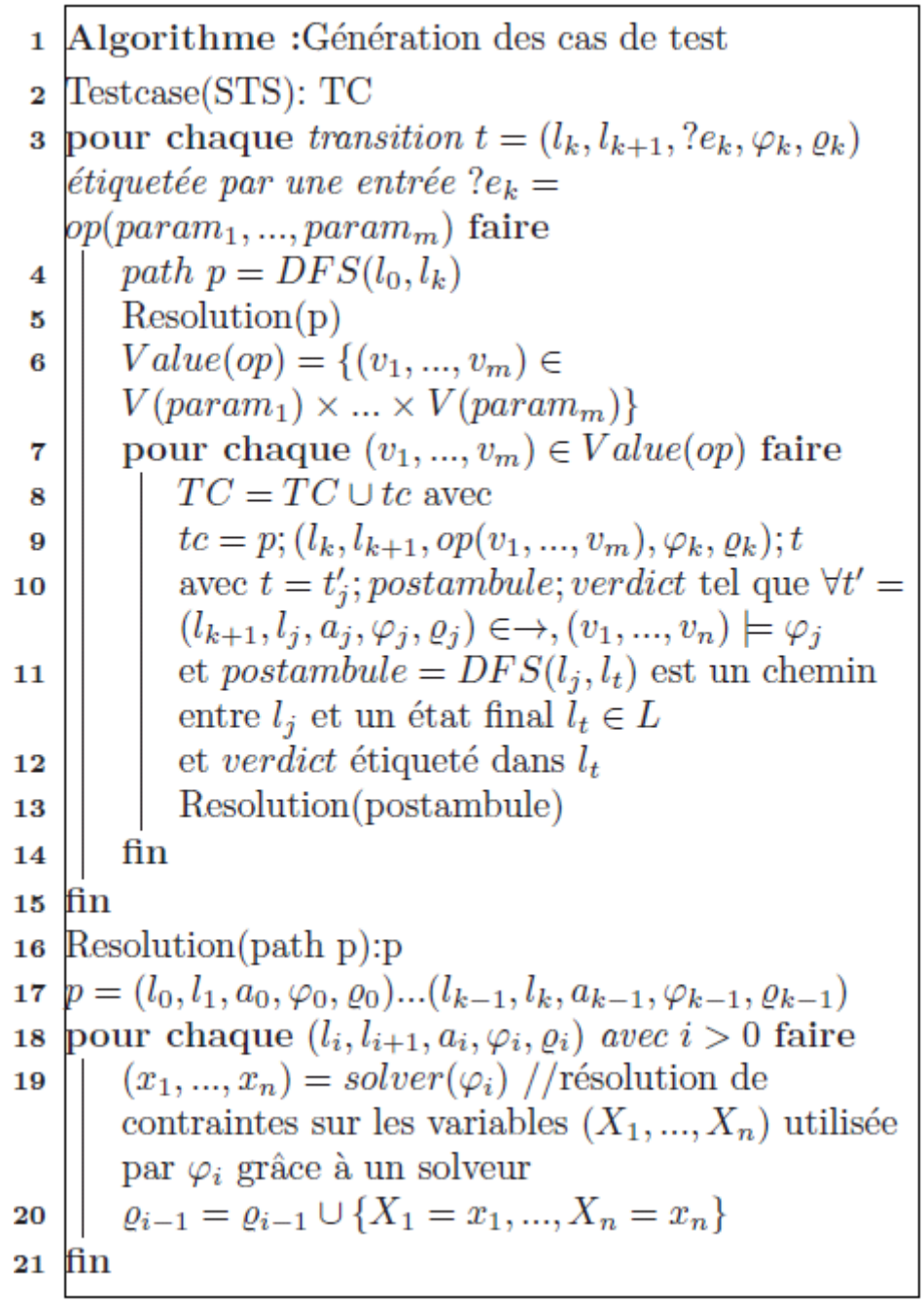


Figure 29: Génération des cas de test

Celui-ci génère des chemins dans lesquels sont injectés les aléas décrits précédemment. Pour une transition modélisant un appel d'opération op, l'algorithme construit un préambule permettant d'atteindre cette transition (lignes 4-5). Un solveur de contraintes est utilisé pour obtenir des valeurs permettant sa complète exécution. Nous ajoutons op(v_1 , …, v_m) au préambule pour appeler l'opération op avec des valeurs inhabituelles (v_1 , …, v_m) (ligne 9).

Puis, nous concaténons toutes les traces exécutables (dont les conditions peuvent être satisfaites) atteignant un état final étiqueté par un verdict (lignes 10-11).

Les solveurs de contraintes permettent de rechercher des valeurs satisfaisant les conditions d'un chemin de la spécification et donc satisfaisant son exécution. Nous utilisons les solveurs [20] et [43]. Ceux-ci fonctionnent sous forme de serveurs externes et peuvent être facilement appelés par l'algorithme de génération de cas de test. Le solveur [43] traite les types "String" et le solveur [20] peut notamment traiter les types "booléen" et "Integer". Dans le cas où le type de donnée est complexe (objet, ...) la résolution des contraintes doit être faite manuellement. Les Figures 30 et 31 illustrent des exemples de cas de test obtenus à partir de la spécification complète de la Figure 28.

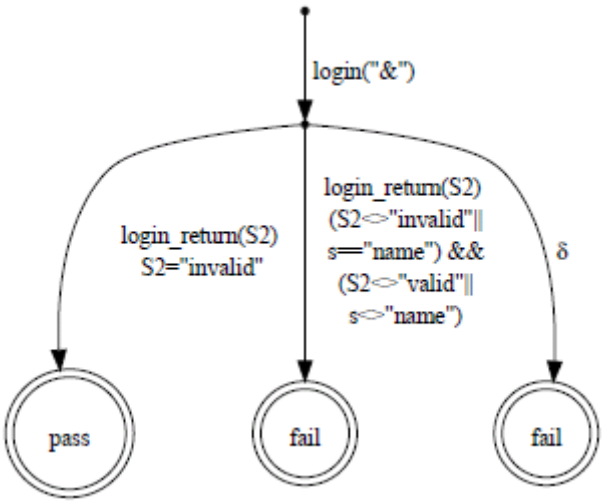


Figure 30: Cas de test : utilisation d'un aléa

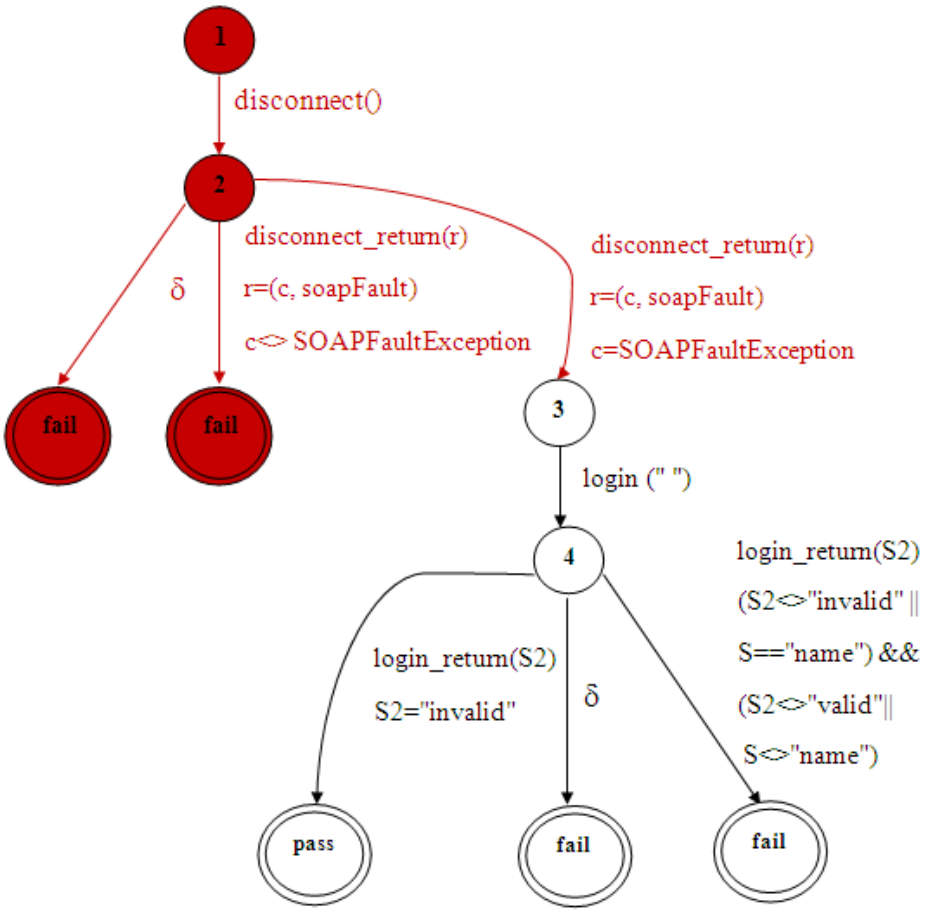


Figure 31: Cas de test : remplacement d'une opération

Dans le premier cas de test, l'opération "login" est appelée avec la chaine "&". Si la réponse obtenue est "invalid" (seule réponse possible avec la valeur donnée) alors le verdict est "pass", autrement on obtient "fail". Dans le deuxième cas de test, l'opération "disconnect" est appelée à la place de "login". Le service est robuste s'il répond "classiquement" ou avec une faute SOAP composée de la cause "SOAPFaultException" et s'il est toujours possible d'utiliser l'opération "login" pour atteindre un état final "pass" de la spécification (fonctionnement normal après l'utilisation de l'aléa).

Autrement, le verdict est "fail" et le service Web n'est pas robuste.

Exécution des cas de test

Les cas de test sont générés puis exécutés grâce à la méthodologie de test illustrée en Figure 27. Celle-ci a été implantée dans un outil académique. Le testeur correspond à un service Web qui reçoit l'URL du service Web à tester et la spécification STS. Le testeur construit les cas de test comme décrit en section 4.2 puis les exécute successivement sur le service Web

persistant. Une fois les cas de test exécutés, l'analyse des réponses obtenues est effectuée pour produire un verdict final de test.

Un cas de test étant modélisé par un arbre, le testeur l'exécute en le parcourant et en appelant les opérations correspondantes. Si une branche est complètement exécutée alors le verdict local "pass" ou "fail" est retourné. Sachant qu'une opération non robuste, peut ne pas retourner de réponse, nous avons considéré que chaque nœud d'un cas de test est quiescent si aucune réponse n'est reçue dans les 60s. Dans ce cas, le verdict "fail" est donné (en suivant une branche étiquetée par δ).

Bénéfices de l'approche

Par rapport aux travaux précédents dont le thème est la robustesse, nous considérons un service Web persistant dont l'état interne est modélisé par un STS.

A la différence des travaux traitant de la robustesse de systèmes "classiques", nous considérons les spécificités des services Web et notamment la couche SOAP et la gestion des exceptions (fautes SOAP). Par rapport aux travaux de [54], [51], [76] sur le test de robustesse des services web, nous analysons et déterminons les aléas qui peuvent être réellement utilisés. Nous séparons également le comportement du service Web de celui du processeur SOAP, ce qui augmente la détection des erreurs de robustesse. Notamment, nous considérons, comme robuste, un service Web retournant des fautes SOAP construites par lui même et non générées par le processeur SOAP. Par rapport à [51], notre architecture de test effectue des appels de service Web directement, ce qui permet d'analyser les réponses SOAP, et en particulier les fautes SOAP.

Notre méthode, comme toutes celles basées sur des types complexes, possède un inconvénient inhérent aux solveurs de contraintes.

En effet, la génération des cas de test s'effectue automatiquement grâce à ces solveurs de contraintes qui peuvent manipulés plusieurs types simples. Il n'existe cependant pas de solveurs sur des types complexes. Dans ce cas, la seule solution est de construire à la main ces valeurs. Cependant, nous avons observé qu'une grande majorité des services Web existants restent basés sur des types simples, ce qui facilite la tâche des solveurs en limitant l'espace de domaine des variables.

Plusieurs perspectives futures peuvent être envisagées notamment par rapport à l'ensemble de valeurs V.

Celui-ci peut en effet être modifié mais reste statique lors de la construction des tests. Il pourrait être plus intéressant de proposer une analyse dynamique pour construire une liste de

valeurs inhabituelles la plus appropriée pour chaque service Web. Pour éviter l'explosion des cas de test, les paramètres sur V sont choisis aléatoirement. Une meilleure solution serait de choisir ces paramètres selon la description de l'opération ou bien d'analyser les valeurs relevant le plus d'erreurs pendant les tests et d'y apposer une priorité (coefficient de pondération).

4.3 – Méthodologie de test de sécurité pour les services Web

« Être sécurisé, c'est être à l'abri de tout danger et risque » : cette définition de la sécurité des individus s'applique aussi sur celle des systèmes informatiques. Cette dernière consiste à assurer ou bien protéger ses ressources matérielles ou logicielles en couvrant l'ensemble de politiques et procédures permettant d'éviter les intrusions (confidentialité), les incohérences (intégrité) et les pannes (disponibilité). Nous notons qu'une politique de sécurité est un plan d'actions définies dont le but est de maintenir un certain niveau de sécurité grâce à un ensemble de règles.

Dans ce cadre, nous nous intéressons particulièrement à la sécurité des services Web. Ceux-ci sont confrontés à l'existence de multiples attaques qui ont été dévoilées par l'organisation OWASP [56]. Parmi celles-ci, citons deux exemples :

- *injection SQL* : est un type d'exploitation d'une faille de sécurité d'un service Web interagissant avec une base de données, en injectant du code SQL malveillant dans les requêtes SOAP afin d'être exécuté par le service Web. L'exemple ci-dessous montre une injection SQL basique, visant à récupérer les comptes administrateurs.

```
<Itemid> 1 and 1=0 union select username,password from admin --
 </Itemid>
<ItemName>book</ItemName>
```

- *injection XML* : Tout comme l'injection SQL, il est également possible d'injecter du XML afin de construire des requêtes SOAP avec des Tag XML ouverts et les imbriqués les un dans les autres, de façon à avoir une requête de taille géante. Celle-ci sera chargée dans la mémoire du serveur Web déployant le service Web. Cette attaque permet d'exploiter la faiblesse de la structuration du schéma de WSDL. La création de

cet ensemble est basée sur des mots clés spécifiques à XSD, tels que l'attribut *maxOccurs* et l'élément *Any.*

- Dans le cas ou l'attribut maxOccurs prend la valeur « *unbounded* », il est possible d'envoyer des requêtes SOAP avec une infinité de balises,
- L'élément *Any* autorise l'ajout des éléments XML non déterminés et appartenant à d'autres espaces de noms.

Nous présentons, dans la suite de ce chapitre, une méthode de test de sécurité pour les services Web persistants. Notre méthode vise à tester des propriétés liées aux services Web telles que la disponibilité, l'autorisation et l'authentification, grâce à un ensemble de règles qui correspondent à des patterns de test définis via des requêtes malveillantes basées sur des paramètres aléatoires et sur des injections SQL et XML.

Afin de réaliser cette méthode, nous avons utilisé dans un premier temps le langage Nomad[18], pour définir cet ensemble de règles.

4.3.1 – Langage Nomad

Divers langages permettant d'exprimer formellement des propriétés de sécurité (SPL[60] et TPL[35]), ont été proposés. Parmi ces langages nous avons choisi le langage Nomad (Non atomic actions and deadlines).

En effet, Nomad est un langage formel incluant des aspects temporels et est adapté pour la description de propriétés de sécurité telle que l'obligation, l'interdiction, la permission, la nécessité et la possibilité.

Avec ce langage, il est, par exemple, possible d'exprimer l'interdiction pour un client d'envoyer deux requêtes successives dans un délai t. Soit la règle correspondante :

$F(start(\text{SendReq(param)}) \mid 0^{<=-t}\ done(\text{SendReq(param)}))$

Pour bien comprendre cette règle décrite avec le modèle formel de Nomad, nous détaillons les formules et les actions définies par ce dernier:

- Deux formules de base ont été définies par Nomad : start(A) et done(A). La première signifie « débuter l'action A » et la deuxième « action A fini»,
- Une action peut prendre deux formes : atomique ou basique

- Action atomique : désigne l'une des actions suivantes : affectation d'une variable, assignation d'une horloge, réception d'un message d'entrée, envoie d'un message de sortie, création/destruction d'un processus,
- Action non-atomique : composée par des actions atomiques : (A;B), pour exprimer que « A est immédiatement suivie par B » et (A || B), décrivant que « A et B sont exécutées simultanément».

Nous détaillons aussi quelques propriétés sur les formules et les actions:

- Si α et β sont des formules, alors (α | β) est une formule exprimant que α est vrai dans le contexte de la formule β,
- *O*(α | β) est une formule exprimant l'obligation, c'est-à-dire que α doit être vrai dans le contexte de la formule β,
- La formule *F*(α | β) exprime l'interdiction, ainsi il est interdit que α soit vrai dans le contexte de la formule β,
- Si α et β sont des formules alors ¬α, (α ∩ β) et (α ∪ β) sont des formules,
- Si α est une formule alors $O^{<=t}$ α(A), exprime que α est vrai t unités de temps dans le passé si t ≤ 0, (α sera vrai après t unités de temps si t ≥ 0), est aussi une formule.

Par conséquence, nous formalisons, avec Nomad, quelques patterns de test de sécurité qui seront intégrées à la méthode de test.

4.3.2 – Les règles de disponibilité

Nous rappelons que la disponibilité de service Web signifie que ce dernier doit répondre une fois appelé. Cette première propriété de sécurité consiste à tester en partie la robustesse du service Web. Ainsi, nous allons vérifier si une réponse est retournée par le processeur SOAP ou bien par le service Web lui-même (Cette réponse retournée peut être une réponse normale ou bien une faute SOAP).

R1 ↔ *O*(*start*(OutputResponseWS(r))| *done*(inputRequest(Spec(op) ∪ V ∪ Inj))

Cette règle concerne la réception d'une réponse après chaque appel à une opération op avec

- OutputResponseWS(r) ↔ OutputResponse(r) ∪ OutputResponseFault(r),
- OutputResponse(r) est une réponse normale de valeur r, tel que (type(r) = type(to(op)) si l'opération appelée est op),

- OutputResponsefault(r) ↔ (c, soapfault) c ≠ "Client" ∩ c ≠ "the endpoint reference not found" est une faute SOAP, dont la cause est différente de ”Client” et de ”the endpoint reference not found”. La première cause signifie que l’opération est appelée avec des mauvais types de paramètres tandis que la deuxième signifie que le nom d’opération n’existe pas,
- inputRequest : modélise l’ensemble des opérations de service Web OP(WS), dont chaque opération op est invoquée avec des valeurs construites par recours à l’ensemble de valeurs Spec(op), V et Inj. Spec(op) rassemble les valeurs de la spécification utilisées avec l’opération op. L’ensemble V est composé de valeurs aléatoires et spécifiques connues dans le test de logiciel (Figure 17),
- Inj ↔ XMLInj ∪ SQLInj correspond à l’ensemble de valeurs prédéfinies, permettant d’exécuter une des deux types d’injections XML ou SQL. De la même manière, XMLInj et SQLInj sont deux ensembles de valeurs spécifiques pour le type "String". Par exemple, les injections XML sont introduites à l’aide de mots-clés spécifiques à XSD, tels que *maxOccurs* qui peut être utilisé dans une tentative d’attaque de type DoS (*Denial of Service*). Plus de détails sur les injections XML et SQL peuvent être trouvés dans [55].

La disponibilité est également assurée à condition que le délai de réponse soit limité. Ceci peut être exprimé en Nomad avec la règle R2.

R2 ↔ *F*(*start*(outputResponseWs(r)) | $O^{<=-Tmax}$ *done*(inputRequestAv(V))).

Celle-ci ne nécessite pas de tests spécifiques si nous prenons en compte la notion de quiescence (absence de réponse observée après un timeout) au cours du processus de test.
Ainsi, si la quiescence est observée après un délai Tmax de l’invocation de l’opération, nous considérons que la règle R2 n’est pas satisfaite.

4.3.3 – Les règles d’authentification

L’authentification vise à garantir l’identité des Clients et de vérifier que si ces derniers n’ont pas la permission alors ils ne peuvent pas être authentifiés. Le processus de connexion représente souvent la première étape d’authentification des Clients. Nous proposons ici deux règles classiques relatives à ce processus de connexion : R3 et R4.

R*3* ↔ *O(start(*OutputResponseWS*(*r*)) | done(*inputAuth*(*Spec*(*op*) ∪ V)))*

Cette règle désigne l'obligation de retourner un résultat d'authentification à chaque fois qu'une requête SOAP d'authentification est envoyée au service Web avec :

- OutputResponseWS(r) ↔ OutputResponse(r) ∪ OutputResponseFault(r),
- inputAuth est l'ensemble d'opérations consacré au processus d'authentification. Ces opérations sont appelées avec des valeurs extraites de la spécification (Spec (op)) ou de l'ensemble de valeurs V.

La règle R4 est dédiée à l'attaque «force brute». Celle-ci vise à décrypter ou à trouver des paramètres d'authentification en parcourant l'espace de recherche de valeurs possibles.
Une des contre-mesures efficace consiste à interdire une nouvelle tentative de connexion après avoir échouer «N» fois par le même utilisateur. La règle correspondante peut être écrite en Nomad avec:

R4 ↔ *O*(*start*(outputResponseWS(*rlforbid*)) | (*done* (inputAuth(V); outputResponseWS (rlfail);inputAuth(V)(nfois))))

- inputAuth est l'ensemble des opérations consacrées au processus d'authentification. Ces opérations sont appelées par les valeurs construites à partir de V,
- OutputResponseWS(r) ↔ OutputResponse (r) ∪ OutputResponseFAult(r) est une réponse d'opération. Celle-ci peut être composée du message rlfail, qui décrit qu'une tentative de connexion a échouée ou par le message rlforbid qui indique que toute nouvelle connexion tentative est interdite. Ces messages doivent être extraits de la spécification et appartenir à l'ensemble OutputResponse.

4.3.4 – Les règles de test d'autorisation

L'autorisation représente la politique d'accès et précise les droits d'accès aux ressources, généralement pour des utilisateurs authentifiés. Pour cela, nous définissons deux règles qui visent à vérifier que l'utilisateur, demandant des données confidentielles, est bien authentifié.

La première règle d'autorisation R5, vérifie que le service Web a retourné un message « autorisation refusée », si une demande de données confidentielles est reçue depuis un utilisateur non authentifié. Cette règle peut être écrite avec:

R5 ↔ *O (start(*outputResponseWS*(*rfail*))* | *(¬ done(*outputResponseAuth*(*rtrue*));* done*(*inputRequestConf*(*Spec*(*op*)* ∪ V *))))*

- outputResponseAuth(rtrue) est la réponse d'une opération d'autorisation qui est composée de message r décrivant une tentative de connexion avec succès,
- inputRequestConf est l'ensemble d'opérations utilisé pour demander des données confidentielles. Les opérations de inputRequestConf sont appelées avec les valeurs extraites de la spécification (Spec (op)) ou bien de l'ensemble de V,
- outputResponseWS(rfail) ↔ OutputResponse(rfail) ∪ OutputResponseFAult (r) décrit une réponse d'opération où le message rfail correspond à un accès refusé. rfail doit être extrait de la spécification.

La dernière règle R6 est dédiée à la réception de données confidentielles moyennant des requêtes contenant des injections XML ou SQL.
Cette règle vérifie la réception d'un message d'erreur pour le cas d'une requête contenant une injection XML ou SQL :

R6 ↔ *O* (start (outputResponseWS (rfail)) | done (inputRequest (Inj)))

Notre objectif est de détecter des vulnérabilités dans des Services Web déployés grâce à ces règles de sécurité données.
A partir de ces règles, la méthode de test, illustrée en Figure 32, construit un ensemble d'objectifs de test abstrait (prérequis pour la construction des tests) pour chaque règle.

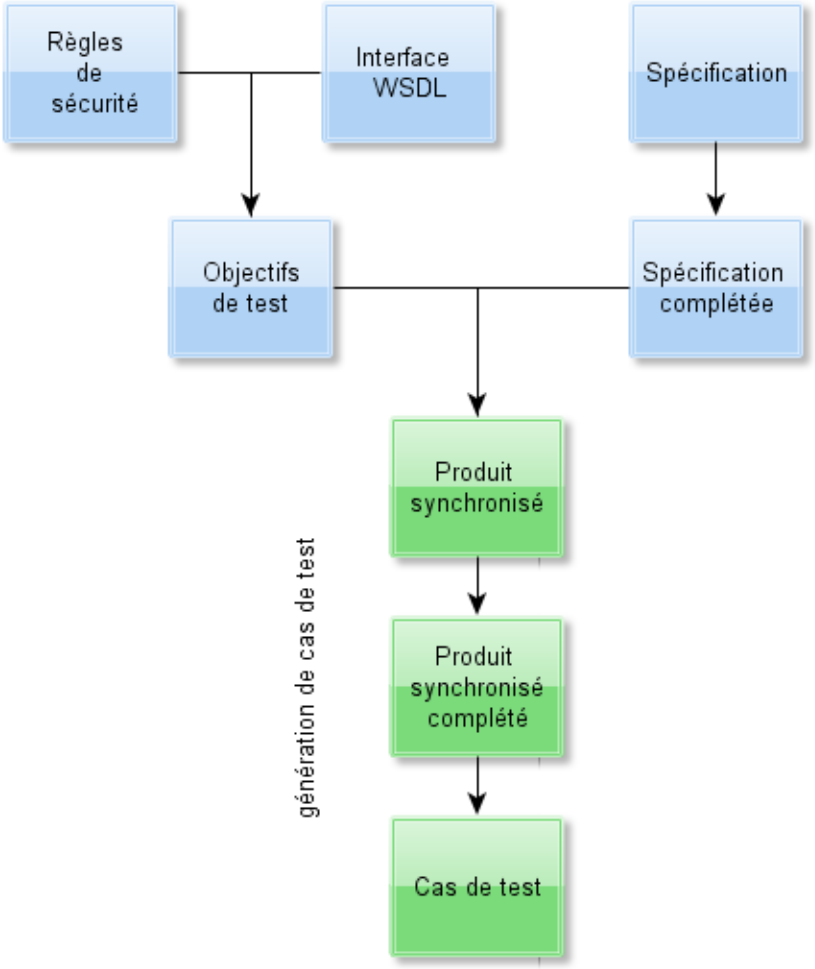


Figure 32: Méthode de test

Ensuite, nous proposons de compléter ces règles à travers le fichier de description WSDL, ce qui permet de déterminer l'ensemble d'opérations et des données de test. Les objectifs de test complétés seront modélisés par un STS.

En parallèle, la spécification de service Web persistant qui est décrite aussi en STS est complétée sur les entrées afin qu'elle accepte les appels de toutes les opérations et ceci, à partir de chaque état symbolique. En effet, peu de spécifications de services Web persistant sont complétée sur les entrées par défaut, cependant le WS-I basic profil statue que toute opération peut être appelée à tout moment.

L'ensemble de tests est ensuite généré à partir de la spécification complétée et des objectifs de tests obtenus en utilisant les règles précédentes. Chaque objectif de test est d'abord synchronisé avec la spécification sur l'ensemble des symboles, sur l'ensemble des contraintes et aussi au niveau affectations des variables.

Il en résulte un STS *Prod*, dont les chemins sont ceux de la spécification complète combinée avec les propriétés des objectifs de test. *Prod* est ensuite complété afin de modéliser les comportements incorrects. Chaque chemin, extrait à partir de Prod grâce à une analyse d'accessibilité, donne un cas de test final. Fondamentalement, les cas de test vont décrire à la fois le comportement correct de service Web, ce qui conduira à un verdict *pass*, mais aussi le comportement incorrect, ce qui conduira à un verdict *fail*. La Figure 33 décrit l'algorithme de la génération des cas de test.

Algorithme 1 : génération de cas de test

1 CasDeTest(STS) : TC ;
Entrée : Un Prod STS, une opération op_{tp}
Sortie : Une suite de cas de test TC
2 Pour tout $transition\ t = (l_k, l_{k+1}, ?e_k, \varphi_k, \varrho_k)$ *étiquetés par une entrée* $?e_k = ?op_{tp}(p_1, \dots, p_n)$ *avec* φ_k *est composé par la condition* $\left[\frac{(p_1,\dots,p_n)}{X_R}\right]$
Faire
3 **Répéter**
4 $path\ p = DFS(l_0, l_k)$;
5 $\varrho_0 = résoudre(p)$;
6 **Jusqu'à** $\varrho_0 \neq \emptyset$
7 **si** $\varrho_0 == \emptyset$ **alors**
8 Aller à la prochaine transition
9 $Value(op_{tp}) = \{(v_1, \dots, v_n) |\ v_i \in X_R(String)\ \boldsymbol{si}\ type(p_i) = "String", v_i \in V(type(p_i))\ \boldsymbol{si}\ type(p_i) \neq "String"\}$;
10 **Pour tout** $(v_1, \dots, v_n) \in Valeur(op_{tp})$ **faire**
11 $t_c \coloneqq \emptyset$;
12 ϱ_0 est la variable d'initialisation de t_c
13 $\varphi_{t_c} \coloneqq [p_1 \coloneqq v_1, \dots, p_n \coloneqq v_n]$;
14 **Pour tout** $transition\ t' = \left(l_{k+1}, l_{k+2}, !\ op_{tp_{return(r_1,\dots,r_m)}}, \varphi_{k+1}, \varrho_{k+1}\right)$ **faire**
15 $t_c \coloneqq t_c \cup p.\left(l_k, l_{k+1}, ?\ op_{tp}(p_1, \dots, p_n), \varphi_k \cup \varphi_{tc}, \varrho_k\right).t'$;
16 $TC \coloneqq TC \cup tc$;

Figure 33: Génération des cas de test

Celui-ci génère les chemins dans lequel les tuples de valeurs utilisés pour les appels sur l'opération de service Web, sont ajoutés. Une analyse d'accessibilité est effectuée à la volée

afin de s'assurer que le cas de test peut être complètement exécuté. Des exemples de cas de test finaux extraits du produit synchronisé sont illustrés dans la Figure 36.

L'algorithme construit un préambule et effectue une analyse d'accessibilité pour vérifier si la transition T est accessible (lignes 2-8). Un ensemble de valeurs X_R est construit selon les types de paramètres de op (ligne 9). Lorsque les valeurs correspondent au format XML ou SQL injections, nous vérifions que les types de paramètres sont équivaut à "String".

Si celles-ci sont complexes (tableaux, objet, etc), nous les composons avec d'autres types pour obtenir les valeurs finales. Nous utilisons également une heuristique permettant d'estimer et de réduire éventuellement la quantité de tests selon le nombre de tuples dans Valeurs(op_{tp}). Les cas de test TC sont réinitialisés, ses variables sont initialisées avec ρ_0.

Ensuite, l'algorithme ajoute les transitions suivantes (l_{k+1}, l_f , $o_{tp\_}return(r_1, \dots, r_m), \varphi_{k+1}$, ρ_{k+1}) avec l'emplacement marqué par un verdict {pass, fail} (lignes 14-16). Nous obtenons un arbre STS, qui décrit une invocation d'opération et les différents emplacements finaux qui peuvent être atteints après cette invocation. Le TC est finalement ajouté à l'ensemble de test.

Pour illustrer notre méthode nous présentons un cas simple de test.

La Figure 34 présente un objectif de test de la règle R1 appliqué à l'opération itemSearch.

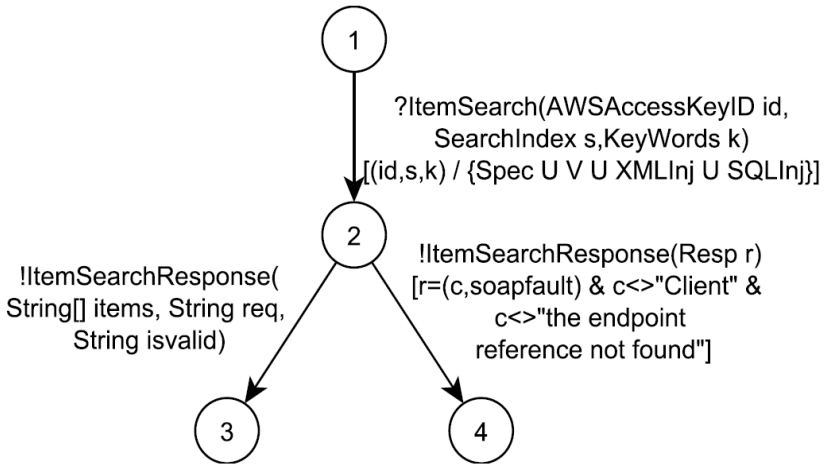


Figure 34: Item Search

Le STS illustré dans la Figure 35, présente la complétion de la spécification d'Item Search, dont les transitions pointillées représentent la complétion sur l'appel d'opérations.

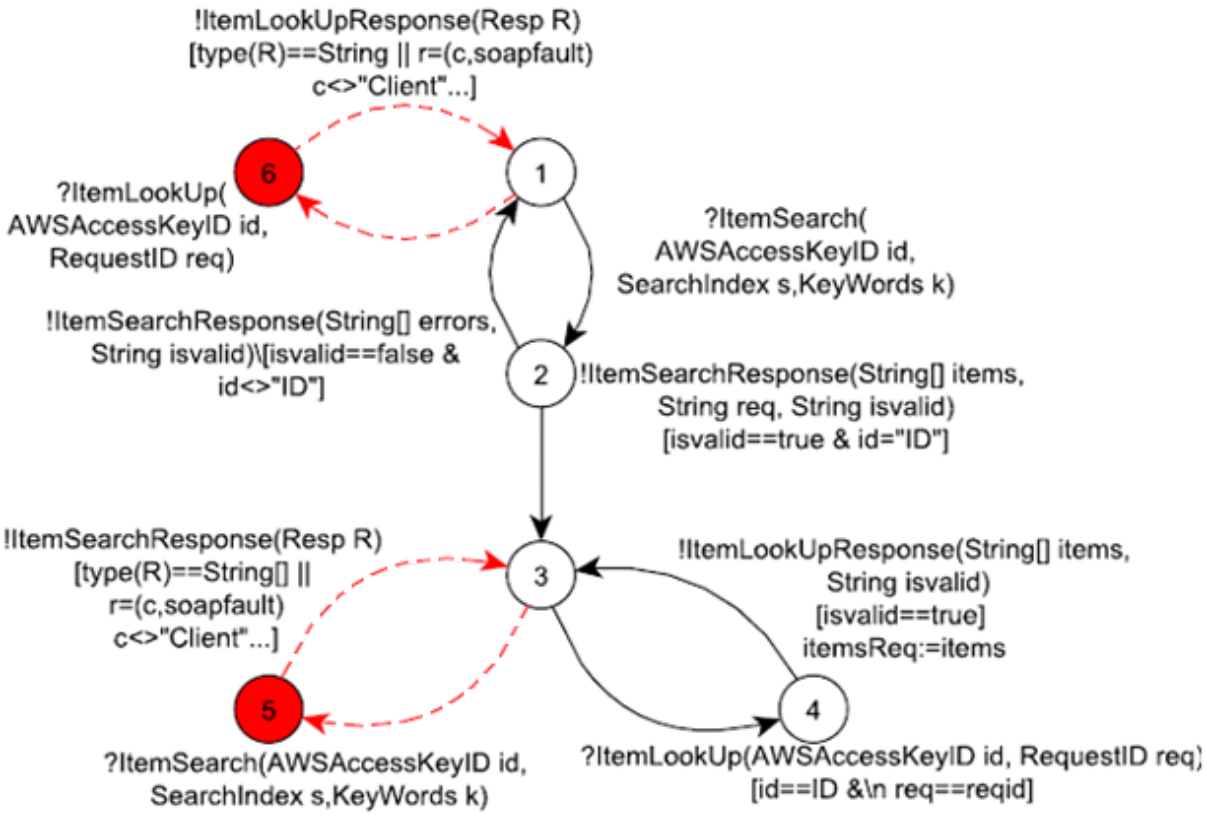


Figure 35: Item Search Complétée

La Figure 36 présente le produit synchronisé complété.

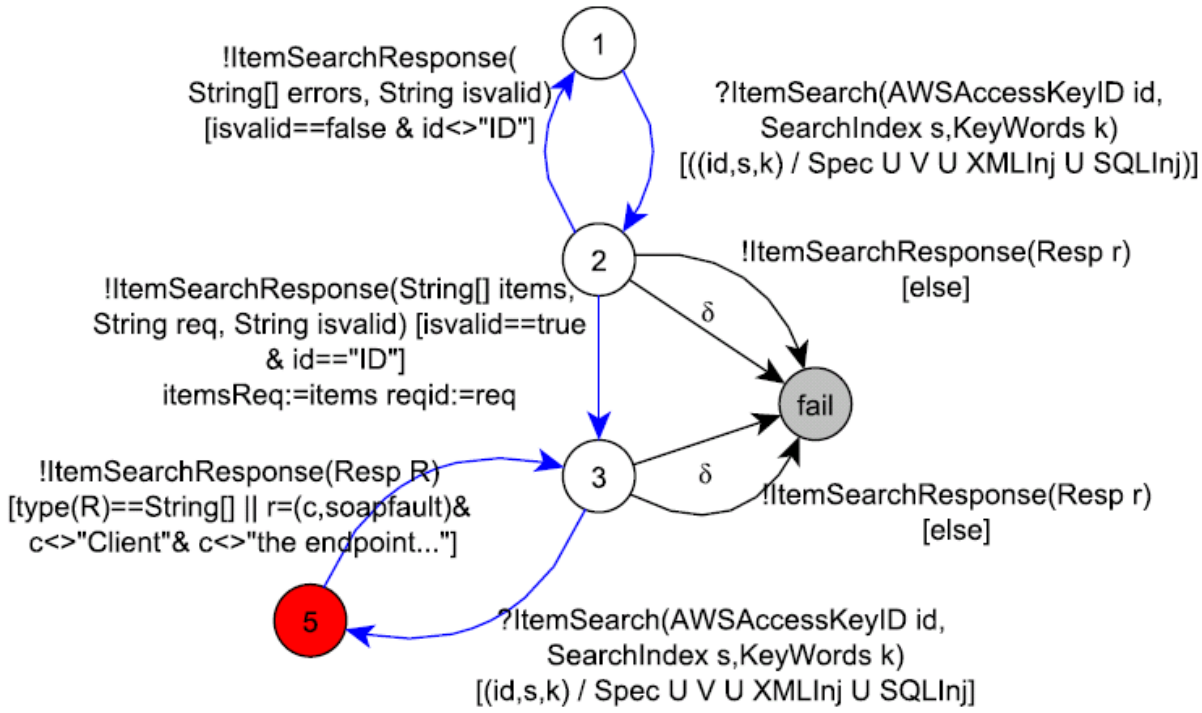


Figure 36: Produit synchronisé complété

Finalement, la Figure 37 présente les cas de tests finaux extraits du produit synchronisé complété.

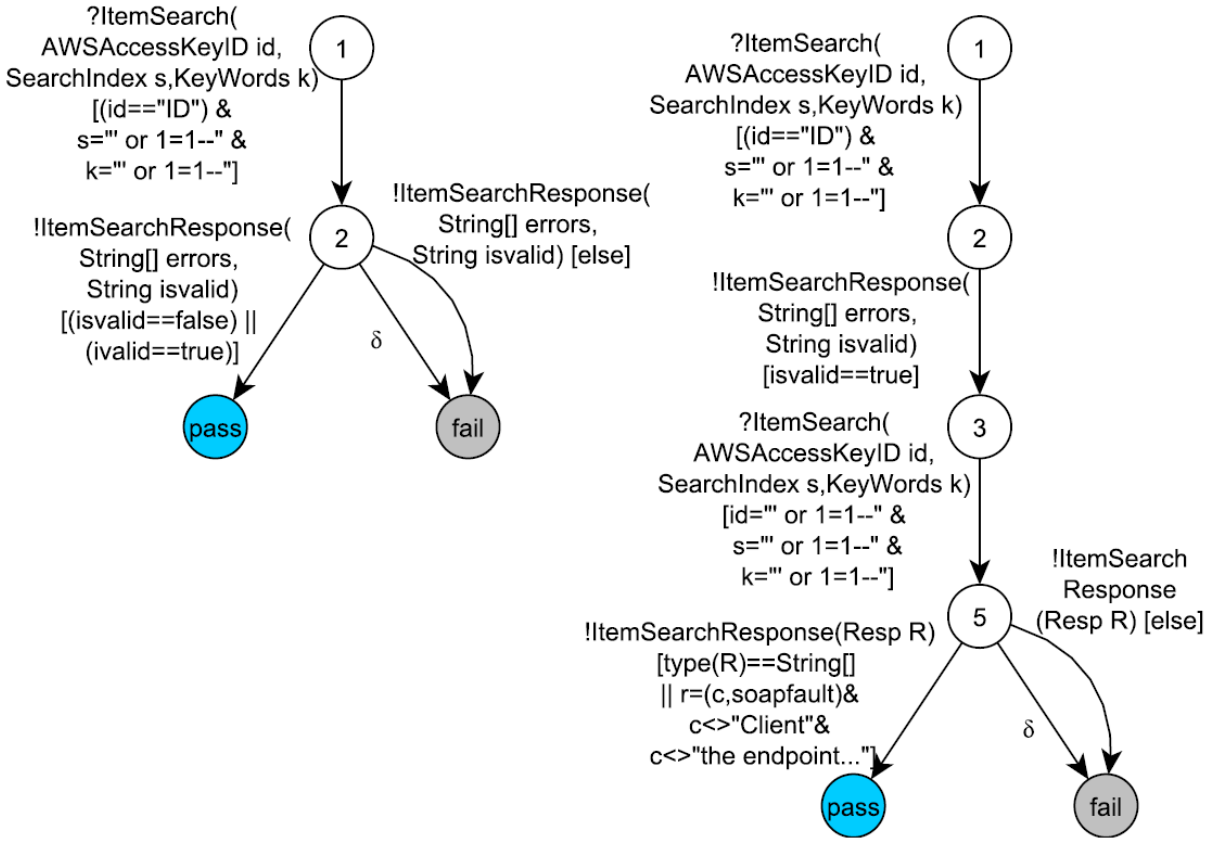


Figure 37: Cas de test finaux

4.3.5 – Expérimentation

Nous avons expérimenté notre méthode, avec succès, sur environ 100 services. 11% ont montré des vulnérabilités de sécurité (Figure 38). Ce chiffre pourrait être facilement augmenté avec un nombre supérieur de règles.

Différents types de vulnérabilités ont été détectés. Par exemple, le service Web http://biomoby.org/services/wsdl/www.iris.irri.org/getGermplasmByPhenotype n'est plus disponible lorsque celui-ci est appelé avec des caractères spéciaux. Aussi, des problèmes d'autorisation ont été détectés avec http://www.handicap.fr/serveur_hanproducts.php?wsdl. Ce dernier renvoi des réponses SOAP contenant des données confidentielles, telles que les noms de table de la base de données et aussi des valeurs.

Avec http://student.labs.ii.edu.mk/ii9263/slaveProject/Service1.asmx?WSDL, il est possible d'appliquer l'attaque «force brute» ce qui permet d'extraire les logins et mots de passe.

Web Service (WSDL)	test number	Availa-bility	Authen-tication	Autho-rization
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

Figure 38: Résultats d'expérimentation

Notre liste des règles de sécurité peut être augmentée afin de révéler plus de vulnérabilités. D'autres concepts de sécurité peuvent être pris en compte tels que l'intégrité.

Chapitre 5

Testabilité de services Web Composés

5.1 – Introduction

L'un des mécanismes intéressants qu'offrent les services Web est celui de la composition. Ce dernier permet de combiner des services Web plus élémentaires afin de produire un service complexe offrant une fonctionnalité plus importante.

Plusieurs types de compositions sont proposés (cf. 2.5), nous intéresserons surtout à l'orchestration de services Web qui correspond à une collaboration suivant une vision centralisée (autour d'un orchestrateur) par l'échange de séquences de messages.

Dans la suite, nous étudierons la testabilité de services Web composés, et plus particulièrement ceux décrits en langage BPEL.

Le présent chapitre est structuré comme suit : d'abord, nous introduisons l'état de l'art et les motivations qui nous ont conduits à tester la composition en BPEL. Ensuite nous détaillons les problématiques relatives au BPEL auxquelles nous répondons avec un ensemble de propositions.

5.2 – Le test de compositions de services Web

Grâce à son vocabulaire riche et ses mécanismes évolués (Figure 4 et Figure 5), le langage BPEL est considéré aujourd'hui comme une technologie permettant la mise en œuvre d'orchestrations de services.

BPEL permet d'invoquer des services Web suivant deux modes : synchrone et asynchrone. Les invocations des opérations des services Web peuvent être réalisées en parallèle ou en séquence. BPEL fournit des mécanismes basiques permettant de définir des conditions, déclarer des variables et de leurs affecter les valeurs retournées par les services invoqués. Aussi, d'autres mécanismes plus évolués permettent la gestion des erreurs et la gestion d'aspect temporaire.

Nous notons que la structure d'un processus BPEL favorise les imbrications récursives des activités moyennant les activités structurées.

La nature de BPEL en terme de diversité de ses fonctionnalités avec son mécanisme d'imbrication d'activités, offre plus de choix et de facilité aux développeurs, cependant elle rend aussi l'analyse de la composition compliquée et difficile à analyser.

Notamment le chaînage et l'imbrication des activités et des messages peuvent entraîner une difficulté au niveau de la testabilité.

Ainsi, il nous semble utile de proposer des méthodes permettant d'analyser et d'améliorer la spécification et la testabilité des compositions de services Web écrites en BPEL.

En effet, différentes méthodes de test des services Web composés en BPEL ont été proposées ([52], [5], [9],…), mais aucune d'entre elles ne portent sur la testabilité.

Dans ce qui suit, nous proposons une méthode d'amélioration d'observabilité des spécifications BPEL. En premier lieu, nous considérons un modèle formel pour la représentation de la description BPEL. Ensuite, et à partir des problèmes d'observabilité détectés, nous en déduisons quelques propositions d'amélioration d'observabilité. Celles-ci peuvent être utilisées pour écrire directement des spécifications ABPEL plus testables et ainsi évaluer leur observabilité et contrôlabilité.

Enfin, nous définissons des algorithmes d'amélioration de l'observabilité.

5.3 – Étude de la testabilité de spécification ABPEL

5.3.1 – Architecture de test

Comme première étape, nous nous intéressons aux architectures de test qui prennent une place prépondérante sur l'observabilité. Deux types d'architectures sont généralement utilisés avec les services Web. Nous allons considérer l'architecture de test en boite grise. Celle-ci consiste à tester un service Web composé avec une connaissance de sa structure interne, c'est-à-dire de la séquence de messages échangés avec ses différents partenaires.

Avec l'approche en boîte grise, nous proposons deux types d'architectures qui diffèrent par le nombre de PCO (Figure 39).

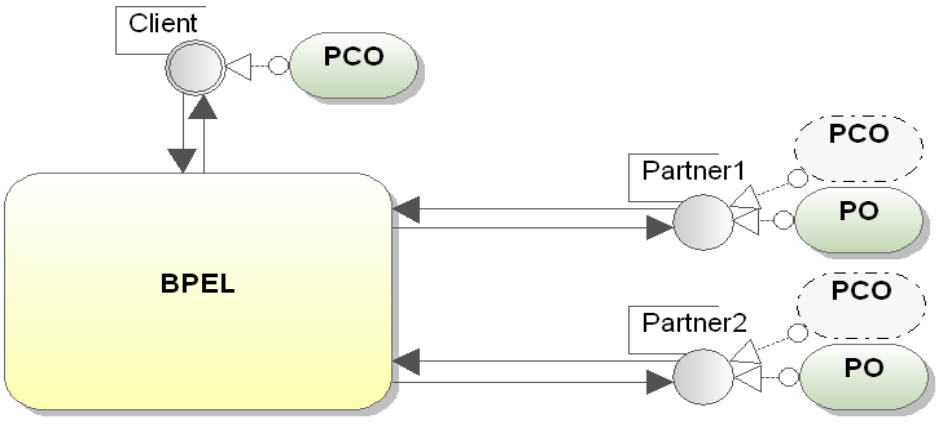


Figure 39: Architectures proposés

- La première architecture est illustrée avec les PCO et PO en trais continu. Elle consiste à ajouter un seul PCO entre le client et le processus BPEL et un PO pour chaque partenaire. Dans ce cas les interactions des services sont observables mais incontrôlables,

- La deuxième architecture est illustrée avec les trais pointillés. Dans cette architecture, nous ajoutons des PCO, coté partenaires, en simulant les services Web. Ainsi le problème d'incontrôlabilité des partenaires est résolu. Cette architecture est efficace malgré une augmentation de la difficulté pour la simulation et aussi du coût du test.

Dans la suite, nous détaillons la transformation de spécification ABPEL en STS.

5.3.2 – Transformation de spécification ABPEL en STS

Dans le cadre de l'étude de la testabilité de spécification BPEL, nous avons défini un ensemble de règles de transformation d'une description ABPEL en STS. Cette transformation prend en compte les messages échangés, la gestion des exceptions, la terminaison des activités ainsi que la corrélation des messages.

Cette transformation vise à aplatir les activités imbriquées, propager les *fault handlers* à chaque activité imbriquée dans une autre et éliminer les activités non pertinentes pour l'analyse de testabilité telle que *assign* et *empty*.

Nous utiliserons ensuite cette spécification formelle STS pour analyser les degrés de testabilité.

Dans cette transformation, nous considérons la plupart des concepts de BPEL : les messages, les liens partenaires, les activités basiques et structurées, la propagation et la gestion des fautes, les activités <scope>… .

Mais nous ne traitons pas la concurrence c'est à dire que nous ne prenons pas en considération l'activité « flow» ni aussi l'activité « event handler ».

Nous commençons par introduire une définition formelle de BPEL. Nous détaillerons ensuite la transformation des concepts majeurs de BPEL en STS.

Définition formelle de spécification BPEL nous considérons qu'une description BPEL est définie avec :

BPEL= SA U BA, avec SA l'ensemble des activités structurées et BA l'ensemble des activités basiques. On a :

- une activité structurée sa = ((sa_1,…, sa_n), FH_{sa}, CH_{sa}, TH_{sa}) ∈ SA est composée d'une liste (sa_1,…, sa_n), d'activités dans (SA U BA)^n, d'un ensemble de *fault handler* noté FH ∈ SA, d'un *compensation handler* noté CH ∈ SA et d'un ensemble de *termination handler* noté TH ∈ SA. Nous notons aussi que les activités *compensation handler* et *termination handler* sont directement appelées par un *fault*

handler lorsqu'une faute est déclenchée. Celles-ci sont utilisées pour réinitialiser ou mettre fin au processus actuel. Les activités FH, CH et TH peuvent être vides (optionnelles),

- une activité basique ba = (ba_1, FH_{ba}, CH_{ba}, TH_{ba}) ∈ BA est composée d'une action ba_1 (invoke, receive, ...), d'un *fault handler* FH ∈ SA, d'un *compensation handler* CH ∈ SA et d'un *termination handler* TH ∈ SA. FH, CH et TH peuvent être vides,
- une activité *fault handler* fh(catch(? f_1, act_1),…, catch (? f_n, act_n), catchall (act)) → e ∈ SA est une activité structurée déclenchée lors de la réception des *WSDL fault* ? f_1,…,? f_n. e présente l'état atteint par la faute f_h une fois celle-ci terminée. Nous notons également FH→ e comme l'ensemble des activités *fault handler* où chaque activité fh_i → e ∈ FH→ e atteint l'état e une fois qu'elle est terminée. Chaque faute est elle-même composée par une liste de variables (faultName, faultElement, faultMessageType).

Notre algorithme de transformation de ABPEL vers STS consiste à convertir successivement et de façon récursive chaque activité structurée en un graphe de sous-activités.

Nous commençons par la première activité du processus ABPEL qui est transformée en un ensemble de transitions STS. Ensuite, chaque activité, rencontrée sur une transition, est transformée jusqu'à ce qu'il n'y ait plus d'autres activités. Afin de transformer les activités basiques et structurées en transitions STS, l'algorithme se réfère à l'ensemble de règles donné dans la Figure 40.

BPEL	STS transformation rules
$assign(var\_update)$ or $empty$	$e_i \xrightarrow{\tau,\emptyset,\varrho} e_{i+1}, \varrho = var\_update$
$receive(op, resp, partner, corr, FH, CH, TH)$	$\frac{fh_k \in FH}{e_i \xrightarrow{?(op,resp,partner,corr),[c_i==corr],\varrho(resp)} e_{i+1} \wedge e_i \xrightarrow{fh_k} e_{n+1}}$
$reply(op, req, partner, corr, FH, CH, TH)$	$e_i \xrightarrow{!(op,req,partner,corr),\emptyset,(c_i:=corr)} e_{i+1}$
$throw(!f, FH, CH, TH)$	$\frac{fh_k \in FH}{e_i \xrightarrow{!f,\emptyset,\emptyset} e_i 1 \xrightarrow{fh_k} e_i fk}$
$if((cond_1, act_1), ..., (cond_n, act_n), FH, CH, TH)$ or $switch$	$\frac{1 \leq l \leq n \wedge FH = \emptyset}{e_i \xrightarrow{\tau,[cond_l],\emptyset} e_{il} \xrightarrow{act_l} e_{i+1l}}$ $\frac{1 \leq l \leq n \wedge FH \neq \emptyset}{if(((cond1, act'_1), ..., (cond_n, act'_n)), \emptyset, \emptyset, \emptyset) \text{ with } act'_k = ((act'_{k1}, ..., act'_{km}), FH_{act_k} \cup FH, CH_{act_k} \cup CH, TH_{act_k} \cup TH)}$
$pick((mess_1, act_1), ..., (mess_n, act_n), onalert(t, act_{n+1}), FH, CH, TH)$ with $mess_k = (op_k, resp_k, partner_k, corr_k)$	$\frac{1 \leq k \leq n \wedge FH = \emptyset}{e_i \xrightarrow{?mess_k,\emptyset,\varrho(resp_k)} e_{ik} \xrightarrow{act_k} e_{i+1k}}$ $\frac{}{e_i(reset\_timer(t)) \xrightarrow{\tau,[t>=Time],\emptyset} e_{in+1} \xrightarrow{act_{n+1}} e_{i+1n+1} \wedge e_i \xrightarrow{\tau,[t<TIME],\emptyset} e_i}$ $\frac{1 \leq k \leq n+1 \wedge FH \neq \emptyset}{pick(((m_1, act'_1), ..., (m_n, act'_n), (onalert(t, act_{n+1}')), \emptyset, \emptyset, \emptyset) \text{ with } act'_k = ((act'_{k1}, ..., act'_{km}), FH_{act_k} \cup FH, CH_{act_k} \cup CH, TH_{act_k} \cup TH)}$
$while(cond, act, FH, CH, TH)$	$\frac{FH = \emptyset}{e_i \xrightarrow{[cond]} e_k \xrightarrow{\tau,[act],\emptyset} e_i \wedge e_i \xrightarrow{\tau,[\neg cond],\emptyset} e_{i+1}}$ $\frac{1 \leq k \leq n \wedge FH \neq \emptyset}{while(cond, (act'_1, ..., act'_n), \emptyset, \emptyset, \emptyset) \text{ with } act'_k = ((act'_{k1}, ..., act'_{km}), FH_{act_k} \cup FH, CH_{act_k} \cup CH, TH_{act_k} \cup TH)}$
$invoke(op, req, resp, partner, corr, FH, CH, TH)$	$\frac{resp \neq null, fh_k \rightarrow e_m \in FH}{e_i \xrightarrow{!(op,req,partner),\emptyset,\varrho=(c_i:=corr)} e_l \wedge e_l \xrightarrow{?(op,resp,partner),[c_i==corr],\emptyset} e_{i+1} \wedge e_l \xrightarrow{fh_k} e_m}$ $\frac{resp = null}{e_i \xrightarrow{!(op,req,partner)} e_{i+1}}$
$fh(catch_1(?f_1(n_1, e_1, m_1), act_1), ..., catch_n(?f_n((n_n, e_n, m_n)), act_n), catchall(?f(n_{n+1}, e_{n+1}, m_{n+1}), act))$	$\forall 1 \leq l \leq n, e_i \xrightarrow{?f_l,\emptyset,\varrho_l} e_{il} \xrightarrow{act_l} e_{ilf}$ $e_i \xrightarrow{?f[f \neq f_1, ..., f \neq f_n],\varrho_n} e_{in+1} \xrightarrow{act} e_{n+1f}$ $\varrho_l = (faultName_l = n_l, faultElt_l = e_l, faultMess_l = m_l)$
$scope((act_1, ..., act_n), FH, CH, TH)$ or $process$ or $sequence$	$\forall 1 \leq l \leq ne_l \xrightarrow{act_l} e_{l+1}$ $\forall 1 \leq k \leq n, act'_k = ((act'_{k1}, ..., act'_{km}), FH_{act_k} \rightarrow e_{k+1} \cup FH \rightarrow e_{n+1}, CH_{act_k} \cup CH, TH_{act_k} \cup TH)$

Figure 40: Règles de transformation de ABPEL en STS

Algorithme de transformation de spécification ABPEL en STS

1. Nous construisons le STS initial composé de la transition $e_1 \xrightarrow{P((p_1,\dots,p_n),\, FH_p \to e_{n+1}, CH_p, TH_p)} e_{n+1}$, où P est le processus initial de ABPEL. $FH_p \to e_{n+1}$ signifie que si un *fault handler* fh $\in FH_p \to e_{n+1}$ est lancé, alors l'état e_{n+1} est atteint et le *fault handler* est terminé. Nous définissons SA = P((p_1,…, p_n), $FH_p \to e_{n+1}$, CH_p, TH_p).
2. Soit la transition $e_1 \xrightarrow{SA} e_{n+1}$, avec SA = P((sa_1,…, sa_n), FH → e_{n+1}, CH, TH). Nous développons SA et construisons les transitions STS correspondantes :

   - Si SA est une activité « while » (ou «if», «pick»), nous transformons SA en SA'= ((sa'_1,…, sa'_n), ∅, ∅, ∅) en se basant sur l'une des règles donnée en Figure 40, composée de la condition FH ≠ ∅. Avec cette règle nous avons étendu l'ensemble de *fault handler* de la première activité structurée dans chacune de ses sous-activités imbriquées. Par exemple, l'ensemble de *fault handler* d'une activité « scope » est étendu dans ses sous activités comme par exemple « while » « if » ou « invoke ». Ensuite nous utilisons une des règles de transformation pour transformer SA' en STS (avec la condition FH ≠ ∅).
   - Si SA est une activité structurée différente de « while » (ou «if», «pick»), alors pour chaque $sa_i \in (sa_1, \dots, sa_n)$:
     - Si $sa_i \in$ BA, nous utilisons la règle correspondante,
     - Si $sa_i \in$ SA, nous construisons la transition $e_i \xrightarrow{sa_i} e_{i+1}$ avec sa_i =((sa_{i1},…, sa_{im}), $FH_{sa_i} \to e_{i+1}$ U FH → e_{n+1}, CH_{sa_i} U CH, TH_{sa_i} U TH). Comme précédemment, nous étendons les *fault handler* de la première activité structurée dans chacune de ses sous activités imbriquées. Nous notons le *fault handler* $FH_{sa_i} \to e_{i+1}$ lié à l'activité sa_i qui se termine à l'état e_{i+1}. Si un *fault handler* de sa_i : fh → $e_{i+1} \in FH_{sa_i} \to e_{i+1}$ est capturé, une fois exécuté, le processus doit repartir de l'état e_{i+1}.

3. Tant qu'il existe une activité non développée $e_i \xrightarrow{SA} e_j$, nous utilisons 2.

Nous avons appliqué cette transformation sur l'exemple *loan Approval* illustré dans la Figure 41. Cet exemple décrit un service de prêt permettant d'accepter l'emprunt d'un montant

amount. Après la réception d'une demande de prêt, et suivant le montant réclamé et le risque lié au client, le service *loan Approval* décide d'accepter ou de refuser la demande.

Le service *loan Approval* peut utiliser trois partenaires :

- un service d'évaluation de risques appelé *loan Assessor*. Ce dernier est utilisé pour obtenir une évaluation du risqué lié au client,
- deux services d'approbation de prêt *loan Approval1* et *loan Approval2* qui sont invoqués successivement, dans le cas d'un haut risque.

Nous obtenons le STS de la Figure 42, qui préserve la sémantique de la spécification ABPEL. Par exemple, la première activité "reply" est traduite par la première transition d'emplacement A1. L'activité *fault handler* commence à l'emplacement A10. Les transitions sortantes sont étiquetées par "?FaultId" et modélisent l'activité catch.

Ci-dessous, nous fournissons quelques définitions et notations que nous allons utilisés après pour l'analyse de la testabilité.

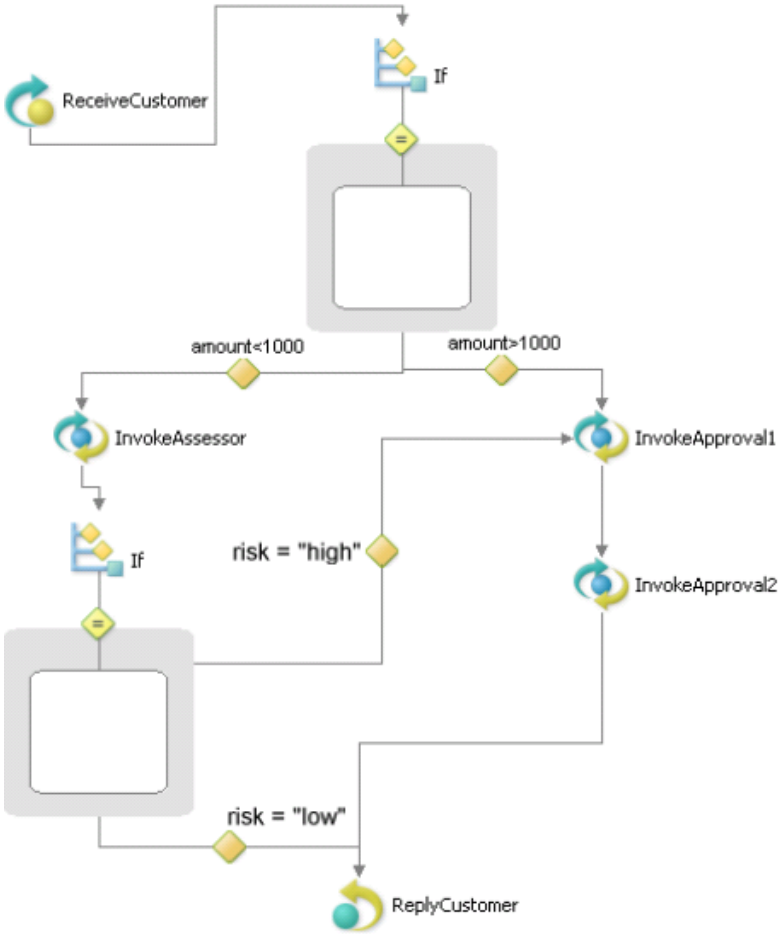


Figure 41: Exemple loan Approval

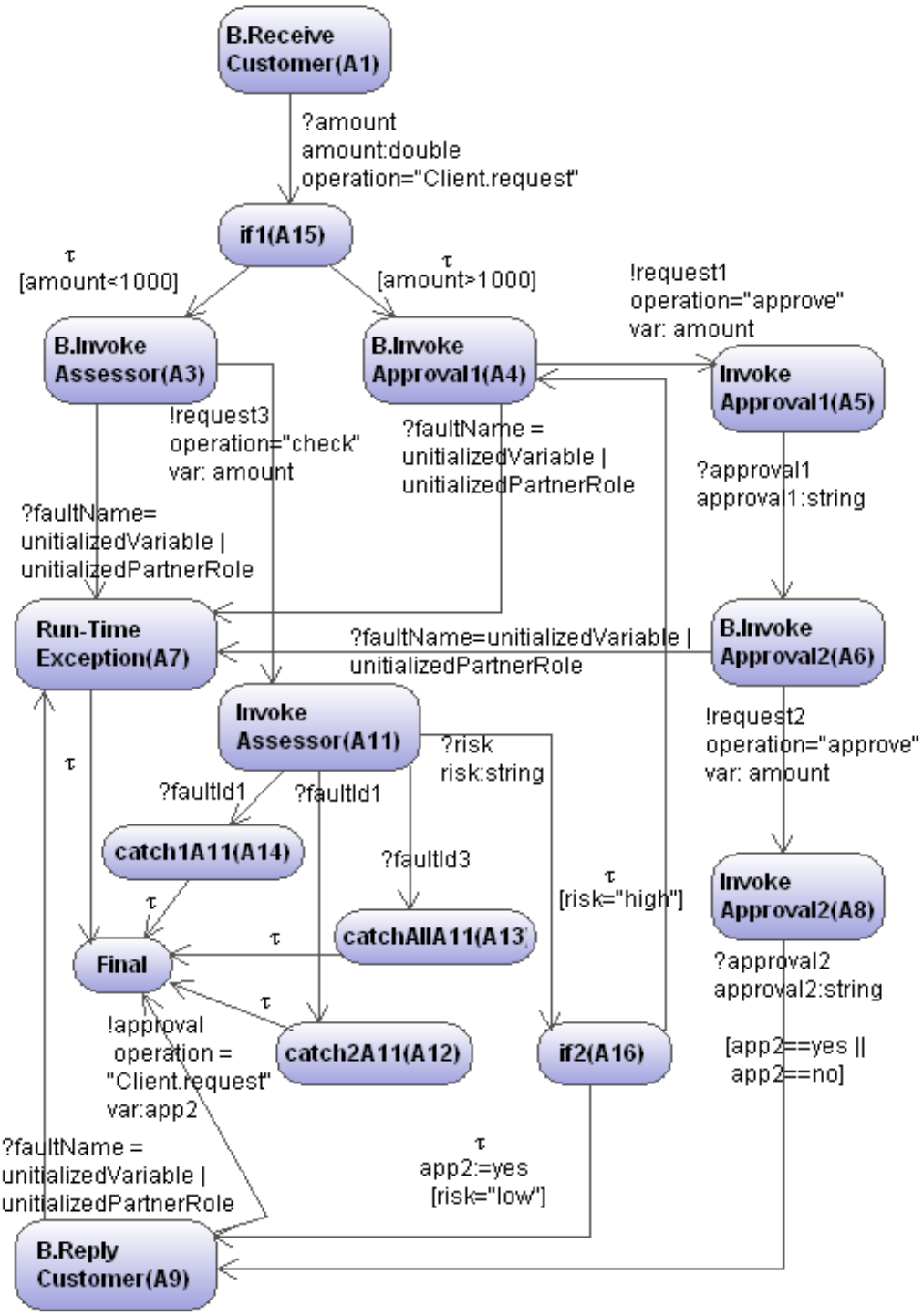


Figure 42: Un STS modélisant une spécification ABPEL

Après avoir défini la transformation de ABPEL en STS, nous étudions la testabilité de spécification ABPEL en se focalisons sur une liste de dégradations qui peuvent affecter l'observabilité.

Propriétés de dégradation d'observabilité

Afin de formuler ces dégradations nous définissons la notion de stimulus (resp. de réaction) [26] qui est formée d'une paire (s,η), avec

- $s \in S_i$ un symbole d'entrée (resp. $s \in S_o$ est celui de sortie)
- η la conversion des interactions de variable de s sous forme d'atomes (*ground terms* en anglais).

Une réaction (s,η) signifie que η est observable dans le message s. Nous considérons qu'une réaction est observée à partir d'un processus BPEL, soit quand une faute est captée (avec l'activité "throw"), soit avec les activités "reply" et "invoke", quand une opération d'un partenaire est appelée avec les valeurs des paramètres (η).

Nous notons aussi out(l,η) l'ensemble des premières réactions observées à partir de l'emplacement l avec la valeur η. out(l_1 ,η) ={(! o_1 , η'_1),.., (! o_n, η'_n) | ∀ 1 ≤ i ≤ n, ∃ p = $(l_1, l_2, e_1, \varphi_1, \rho_1) \cdots (l_i, l_{i+1}, !o_i, \varphi_i, \rho_i)$ tel que $(e_i \ldots, e_{i-1}) \in S_i^{i-1}$ et $(\varphi_1(\eta) \Lambda, \ldots, \Lambda \varphi_i(\rho_{i-1}, \eta'_1))$ est vrai} .

Dans la suite, nous supposons que les partenaires faisant partie de l'orchestration BPEL, sont observables.

L'observabilité vise à évaluer l'état interne d'un système à travers ses sorties observées. Selon la définition donnée à la section 3.3.5, un système est observable si pour toute entrée il existe une sortie différente.

Ainsi, pour un STS = $< L, L_0, Var, var_0, I, S \rightarrow >$, nous définissons deux propriétés de dégradation d'observabilité qui prennent en compte les stimuli et les réactions. La première propriété signifie que l'observabilité est dégradée si aucune réaction n'est observée. La seconde décrit le cas de différents stimuli qui produisent les mêmes réactions.

- propriété de dégradation 1 : s'il existe un stimulus (?e,η) tel que $(k, l, ?e, \varphi, \rho) \in \rightarrow$ et out(l, η) = ∅,
- propriété de dégradation 2 : s'il existe deux stimuli $(?e_i, \eta_i) \neq (?e_j, \eta_j)$ avec $(l_i, l'_i, ?e_i, \varphi_i, \rho_i) \in \rightarrow$, $\varphi_i(\eta_i)$ est vrai et $(l_j, l'_j, ?e_j, \varphi_j, \rho_j) \in \rightarrow$, $\varphi_j(\eta_j)$ est vrai tel que out(l'_i , η_i) = out(l'_j , η_j).

A partir de la première propriété de dégradation STS, nous en déduisons les deux dégradations de code ABPEL: «l'absence d'une activité « reply » à la fin de processus ABPEL p» et « l'utilisation de conditions dans le cas des activités qui ne peuvent pas toujours être satisfaites ».

Proposition 1 : Une spécification ABPEL qui ne se termine pas par l'activité « reply » (one-way invoke) est non observable.

Preuve : les deux activités « reply » et one-way « invoke » produisent la transition STS $(l_j , l_{j+1}, ! o_j , \varphi_j , \rho_j)$

1. Prenons un processus ABPEL composé uniquement d'actions internes (par exemple avec les activités « assign » ou « empty ») et qui ne se termine pas par l'activité « reply ». Un processus ABPEL commence toujours par une activité « receive » [78]. La transformation de ABPEL vers STS entraine une transition $(l , l' , ? e , \varphi , \rho)$ étiquetée par un symbole d'entrée. Donc $\forall$ η, out(l', η) =$\emptyset$, ABPEL est non observable.
2. Considérons un processus ABPEL composé d'interactions internes des partenaires (le cas habituel), et qui ne se termine pas l'activité « reply ». Nous avons supposé que les partenaires associés sont observables. Par conséquence, ils produisent toujours des réponses une fois appelés. Une activité «invoke » se traduit par deux transitions STS, dont la première est étiquetée par un symbole de sortie et la deuxième est étiquetée par un symbole d'entrée. La dernière activité « invoke » dans le processus ABPEL produit les transitions de STS $(l_i , l_{i+1}, ! o , \varphi , \rho)$, $(l_{i+1} , l_{i+2}, ? e, \varphi_{i+1} , \rho_{i+1})$.Si cette dernière activité n'est pas suivie par un « reply », il n'y a pas de chemin composé de symboles de sortie. Par conséquent, aucune réaction ne peut être observée, $\forall$ η, out(l_{i+2}, η) =$\emptyset$, ABPEL n'est pas observable.

Proposition 2 : une spécification ABPEL composée d'une activité « invoke » (ou « receive ») suivie d'une activité « if » if(($cond_1$, act_1),…, ($cond_n$, act_n)) qui ne peut pas être toujours satisfait ($\exists$ η, $\bigvee_{1 \le i \le n}$ $cond_i$ (η) est faux), est non observable.

Preuve : une activité « invoke » est transformée en deux transition de STS $(l , l' , ! o , \varphi , \rho)$, $(l' , l'' , ? e, \varphi' , \rho')$. Une activité « receive » est traduite seulement par la dernière transition. L'activité « if » donne la transition $\forall$ $(1 \le i \le n)$ $(l'' , l_i , \tau , \varphi_i , \emptyset)$ avec φ_i = $cond_i$.

En supposant qu'il existe η tel que $\bigvee_{1 \le i \le n} (\varphi_i(\eta))$ est faux. Aucune transition ne peut être franchie, l'état symbolique l'' est bloqué (quiescent). Ainsi, il existe un stimulus (?e, η) tel que out(l'' , η) =$\emptyset$. Par conséquence BPEL est non observable.

Les quatre propositions de dégradation de la testabilité de ABPEL, que nous présentons, résultent de la deuxième dégradation. Avec le modèle STS, cette dégradation est déclenchée s'il existe deux transitions $(l_i, l_{i+1}, ?e_i, \varphi_i, \rho_i)$, $(l_j, l_{j+1}, ?e_j, \varphi_j, \rho_j)$ à partir desquelles la même réaction (!o, η) est observée. Ce cas peut être dérivé pour la spécification ayant:
« deux différentes activités « catch » suivies par la même invocation», « une activité « catchall », déclenchée par de multiples fautes et suivie par la même invocation», « une activité « pick » avec des multiples branches « onmessage » et suivie par la même invocation », ou « un bloc d'activités internes liés par deux activités «receive».

Proposition 3 :
Une spécification ABPEL composée d'un couple d'activité « catch » (catchall) non identiques ($catch_i$, $catch_j$), $catch_i \neq catch_j$, suivie par deux activités « invoke» utilisant la même opération et les valeurs des paramètres, est non observable.
Preuve : Dans la spécification WS-BPEL [78], deux activités « catch » sont considèrées comme identiques si elles traitent les mêmes fautes. Chaque activité « catch » est déclenchée par une faute ?f = (faultName, faultElement, faultMessageType). Deux différentes activités « catch » ($catch_i$, $catch_j$), sont transformées en deux transitions STS $(l_i, l_{i+1}, ?fault_i, \emptyset, \rho_i)$ et $(l_j, l_{j+1}, ?fault_j, \emptyset, \rho_i)$, avec $?fault_i \neq ?fault_j$. Si $catch_i$ et $catch_j$ sont suivies par la même opération, en utilisant les même valeurs de paramètres η, le STS est composé de deux transitions $(l_{i+1}, l_{i+2}, !o, \emptyset, \rho_{i+1})$ et $(l_{j+1}, l_{j+2}, !o, \emptyset, \rho_{j+1})$. Par conséquence, il existe deux stimuli (l_{i+1}, η_i), (l_{j+1}, η_j) tel que out(l_{i+1}, η_i) = out(l_{j+1}, η_j), =(! o , η). Dans ce cas le ABPEL est non observable. Pour les propositions restantes nous ne détaillons pas les preuves (même principe).

Proposition 4 : une spécification ABPEL composée de l'activité « catchall », déclenchée par de multiples fautes et suivie d'une activité « invoke » dont l'appel à l'opération est indépendant des fautes déclenchées, est non observable.

Proposition 5 : une spécification ABPEL composée d'une activité « pick » avec des multiples branches « onmessage », suivie par des activités « invoke » en utilisant les mêmes opérations et les valeurs des paramètres, n'est pas observable.

Proposition 6 : une spécification ABPEL composée d'un bloc de n ≥ 0 activités internes (pas d'activités "invoke", "reply", "throw") lié par deux activités «receive», est non observable.
Ces propositions de dégradation de l'observabilité seront utilisées pour la définition des algorithmes d'amélioration d'observabilité.

5.3.3 – Amélioration d'observabilité de ABPEL

Nous proposons quelques algorithmes d'amélioration d'observabilité de ABPEL qui suppriment des dégradations présentées auparavant et un outil associé. Ce dernier analyse une spécification ABPEL, détecte les dégradations d'observabilité, selon les propositions 1-6 en les supprimant. L'architecture de l'outil est présentée dans la Figure 43.

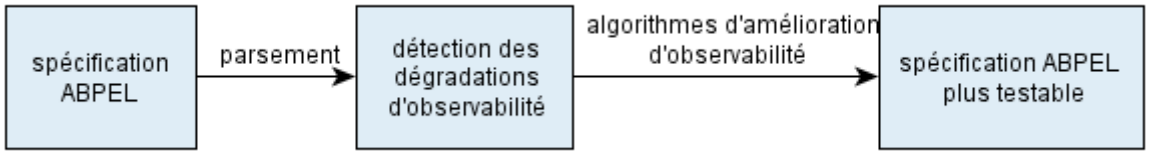


Figure 43: Outil d'amélioration d'observabilité de ABPEL

La modification de la spécification ABPEL peut exiger la mise à jour de certaines descriptions de partenaire WSDL ou de certaines parties du code source. C'est pourquoi nous notons que cette méthode d'amélioration est semi-automatique. Les différentes étapes d'amélioration sont détaillées ci-dessous.

- Ajout d'activité "reply": nous vérifions que chaque branche de ABPEL se termine par une activité "reply" (invoke-only) (proposition 1). S'il en manque une, nous complétons la spécification par une activité "reply", modélisant une réponse au client qui a appelé le processus ABPEL. L'algorithme correspondant est présenté dans la Figure 44. La réponse envoyée au client est composé du message "final message from $branch_i$" qui est supposé être un message de sortie unique et pas encore utilisé dans la spécification (ligne 3). (L'observabilité n'est pas dégradée avec l'ajout de ce message de sortie),

Algorithme 1 : Ajout d'activité"reply"

Entrée : une spécification *ABPEL*

1 Calculer $sts = \langle L, l_0, Var, var_0, I, S, \rightarrow \rangle$ à partir de *abpel* ;

2 si $\exists\, l_i \xrightarrow{e,\varphi,\varrho} l_f \; avec \; e \in S_I \cup \{\tau\} et \; l_f \; est \; une \; position \; finale$ **alors**

3 Ajouter *reply(resp, partner, op)* dans *bpel* avec resp= "$le\ dernier\ message\ de\ branch_i$"
Partner=client, op=opération du client utilisée pour appeler le processus BPEL ;

Figure 44: Ajout d'activité « reply »

- Ajout d'activité "if": d'après la proposition 2, une activité "if" $\big((cond_1, act_1), \ldots, (cond_n, act_n)\big)$, qui ne peut pas toujours être satisfaite $(\exists \eta \bigvee_{1 \le i \le n} cond_i(\eta) \; est \; faux)$, peut dégrader la spécifications de l'observabilité. Nous complétons une telle activité, en ajoutant une branche conditionnelle "<else>", représentant la disjonction de toutes autres conditions. Cette nouvelle branche se termine par une activité "reply" pour ne pas dégrader l'observabilité (un processus BPEL doit se terminer par une activité "reply"). Plus précisément, l'algorithme donné dans la Figure 45, ajoute une branche décrivant une nouvelle exception "FLT:$badCondition_k$" (ligne 4). Cette exception est capturée par une nouvelle activité "catch" qui est ajoutée dans le bloc courant "scope" (lignes 5-6).

Cette activité est composée d'une "reply" pour l'envoi de l'exception "FLT:$badCondition_k$ " au client.

Algorithme 2 : Ajout d'activité "if"

Entrée: une spécification *ABPEL*

1 pour toute activité "invoke" I suivi d'une activité "if" $if_k\left((conf_1, act_1), \dots, (cond_n, act_n)\right)$ **faire**

2 Calculer V tel que $\forall v \in V \ \bigvee_{1 \le i \le n}(cond_i(v))$ faux avec les solveurs de contraintes;

3 **si** $V \neq \emptyset$ **alors**

4 Ajouter "$< else >< throw\ faultName =$ FLT: badCondition$_k$ / $>< /else >$" ;

5 Ajouter cette activité "catch" dans un faulthandler d'une activité "scope" contenant if_k:<catch faultName="$FLT: badCondition_k$" > **REPLY** < /catch>";

6 tel que REPLY= reply(resp, partner, op) avec partner=client, op= opération du client utilisée pour appeler le processus BPEL, resp="FLT : $badCondition_k$" ;

Figure 45: Ajout d'activité « if »

- Distinction d'activité « catch » : Nous proposons de distinguer deux appels de partenaires (pour produire des réactions différentes), en complétant op avec une nouvelle variable et en prenant une valeur différente selon le stimulus. L'algorithme correspondant est donné en Figure 46.

Algorithme 3 : distinction de l'activité "catch"

Entrée : une spécification *ABPEL*

1 **si** il existe $catch_i\ (?fault_i, act_i), catch_j(?fault_j, act_j)$ dans abpel, suivies par le même appel d'opération op et avec les types de paramètres $(p_1, \dots, p_m)$ **alors**

2 calculer $sts = \langle L, l_0, Var, var_0, I, S, \rightarrow \rangle$;

3 **pour tout** $paths\ p_i = l_0 \xrightarrow{e_0,\varphi_0,\varrho_0} l_1, \dots, l_i \xrightarrow{?fault_i,\varphi_i,\varrho_i} l_{i+1} \xrightarrow{!op,\emptyset,\varrho_{i+1}} l_{i+2}$ et $p_j = l_0 \xrightarrow{e'_0,\varphi'_0,\varrho'_0} l'_1, \dots, l'_j \xrightarrow{?fault_j,\varphi'_j,\varrho'_j} l'_{j+1} \xrightarrow{!op,\emptyset,\varrho_{j+1}} l'_{j+2}$ **faire**

4 calculer en utilisant les solveurs l'ensemble de valeurs V_1 moyennant $(p_1, \dots, p_m)\ tel\ que\ \forall\ v\ \in V_1, \left(\varphi_0(v) \wedge\ \varphi_1(\varrho_0) \wedge, \dots, \wedge \varphi_i(\varrho_{i-1})\right), true$;

5 calculer V_2 à partir de p_j ;

6 **si** $V_1 \cap V_2 \neq \emptyset$ **alors**

7 Ajouter à la première activité invoke un paramètre de type string "fault" avec la valeur "?$fault_i$";

8 Ajouter à la deuxième un paramètre de type string "fault" avec la valeur "?$fault_j$";

Figure 46: Distinction de l'activité «catch »

5.3.4 – Analyse de l'exemple Loan

L'exemple *loan Approval* et sa transformation en STS sont donnés en Figure 40 et 41.
Nous constatons, dans cet exemple, trois types de problèmes sur l'observabilité:

1) Le premier cas est lié à l'état symbolique A2. Dans le cas de stimuli (τ, amount=1000) aucune réaction n'est observée.
2) Le deuxième cas détecté est au niveau de l'état symbolique A9 (reply Customer). Dans le cas de stimuli (?risk, risk='low') et (?approval2, app2=yes), nous obtenons la même réaction (!approval, app2=yes). Ceci entre en contradiction avec la définition d'observabilité.
3) Le troisième problème d'observabilité est lié aux états symboliques A12, A13 et A14. Dans ce cas aussi nous avons des entrées différentes modélisant l'activité « catch » avec la même sortie τ. Ce qui implique un problème d'observabilité.

Grâce à notre outil, nous avons expérimenté les algorithmes précédents sur notre exemple *loan Approval*. L'exemple amélioré est montré dans la Figure 47.

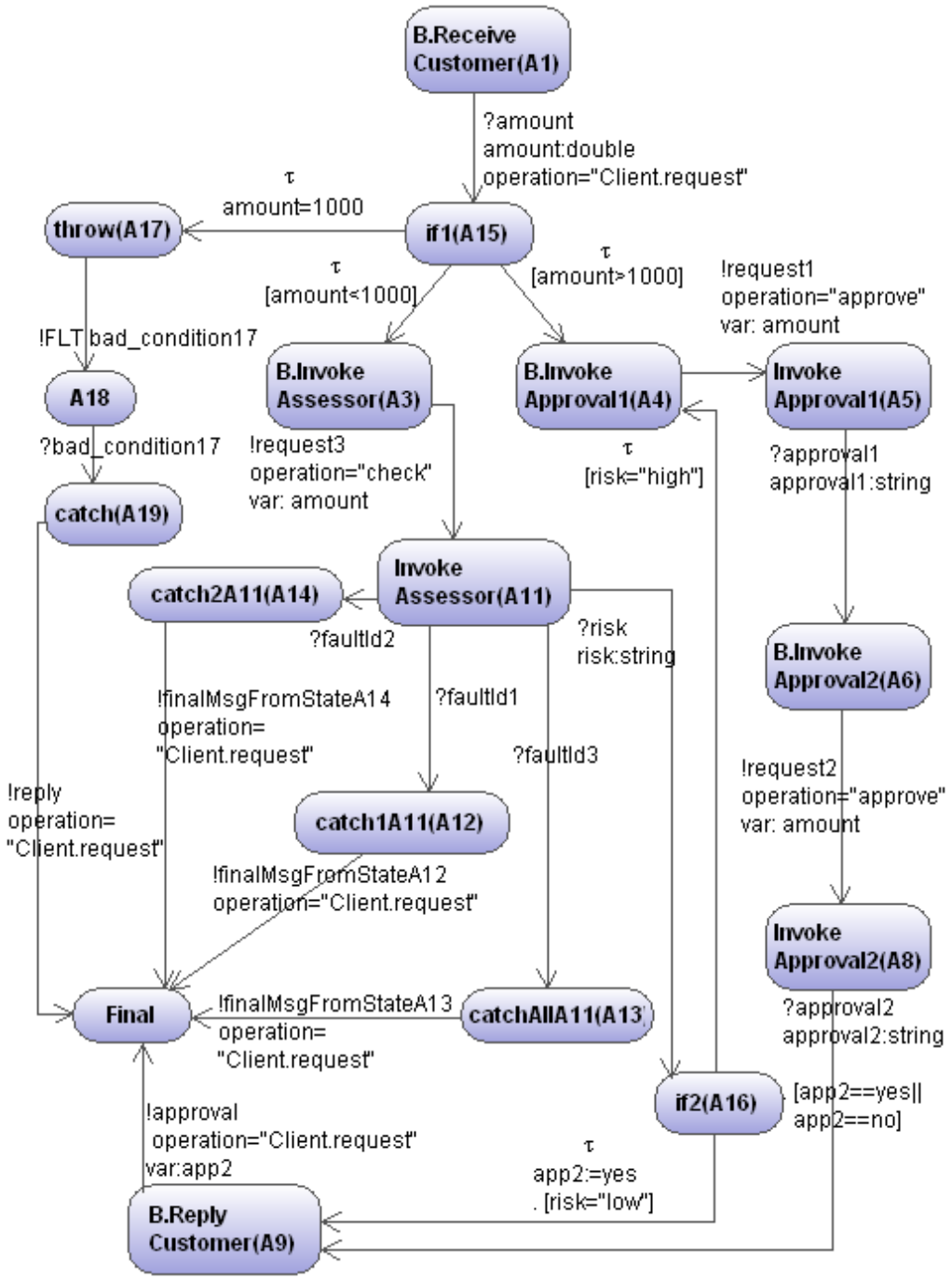


Figure 47: Exemple amélioré

Nous avons abordé, dans ce chapitre, les services Web composés avec le langage BPEL, nous avons décrit quelques problèmes de testabilité et précisément ceux liés l'observabilité. Pour cela, nous avons proposé une approche permettant de transformer la spécification de ABPEL en STS afin d'aplatir les activités imbriquées dans un modèle formel connu, ce qui réduit la complexité de BPEL et facilite l'analyse de ses activités. Ensuite, nous avons détaillé quelques propositions permettant de réduire les problèmes liés à l'observabilité. Enfin, et en

se basant sur ces propositions, nous avons défini quelques règles, sous format d'algorithmes, permettant l'amélioration de l'observabilité.

D'autres caractéristiques globales de testabilité comme la contrôlabilité peuvent être envisageables. Dans le dernier cas, il serait possible de mesurer le degré de contrôlabilité de la spécification ABPEL, en prenant en compte certaines propriétés comme le cas des activités indéterministes. Par exemple, avec une activité « Faulthandler » composée de deux *catch* identiques, la spécification ABPEL devient alors non contrôlable.

Chapitre 6

Conclusion et perspectives

6.1 – Conclusions

La présente thèse s'est attaquée sous diverses formes au test automatique : une première partie est consacrée au test fonctionnel à travers le test de robustesse des services Web non composés. La seconde partie étend les travaux précédents pour le test de propriétés non fonctionnelles dans le domaine de la composition de services, telles que les propriétés de testabilité et de sécurité.
Ces tests automatiques ne peuvent s'effectuer qu'avec une combinaison habile de techniques, comme par exemple l'utilisation d'oracle. Nous avons pour cela proposé des méthodes permettant de modéliser des systèmes à état grâce aux systèmes symboliques ayant un espace d'états comportementaux infinis.

Nous nous sommes intéressés au cours cette thèse à l'amélioration des techniques existantes concernant la création des tests de robustesse pour les services Web. L'approche développée dans le chapitre 4, consiste à évaluer automatiquement la robustesse d'un service Web par rapport aux opérations déclarées dans sa description WSDL. En examinant les réponses reçues lorsque ces opérations sont invoquées avec des aléas (exemple : les valeurs inhabituelles, des paramètres erronés), nous avons remarqué que de nombreux aléas sont bloqués par les processeurs SOAP. Nous avons dû alors séparer le comportement du processeur SOAP de celui du service Web. En effet, les processeurs SOAP peuvent retourner des fautes SOAP, celles-ci n'étant pas considérées dans les spécifications et par conséquent, non prises en compte dans les méthodes de test. Cependant, elles font partie intégrante des services et peuvent fortement compliquer le test et les résultats obtenus. Un outil (WS-AT) a été développé et expérimenté sur des exemples réels de services Web prenant en compte ces différentes problématiques.

Nous avons également proposé, au cours du chapitre 4, une méthode de test dédiée aux services Web persistants en se basant sur un exemple d'Amazon AWS E-commerce. Notre méthode de test de robustesse prend en compte l'environnement SOAP, permet de compléter la spécification (STS) du service Web afin de décrire l'ensemble des comportements corrects et incorrects possibles. Ensuite, en utilisant cette spécification complétée, les services Web sont testés en y intégrant des aléas. Un verdict est ensuite rendu.

Nous avons clôturé le chapitre 4 avec une méthode de test de sécurité de services Web persistants. Cette méthode consiste à évaluer quelques propriétés de sécurité, tel que l'authentification, l'autorisation et la disponibilité, grâce à un ensemble de règles. Ces règles ont été crée, avec le langage formel Nomad. La méthodologie de test consiste d'abord à transformer ces règles en objectifs de test (avec STS) en se basant sur la description WSDL, ensuite à compléter, en parallèle, la spécification du service Web persistant et enfin à effectuer le produit synchronisé afin de générer les cas de test.

Dans le chapitre 5, nous avons attaqué les problèmes de testabilité des services Web composés avec le langage BPEL. Nous avons décrit précisément les problèmes liés à l'observabilité qui sont intéressant au vu de la nature de ce langage en termes de diversité de fonctionnalités. Pour cela, nous avons proposé une approche permettant, en premier lieu, de transformer la spécification ABPEL en STS. Cette transformation consiste à convertir successivement et de façon récursive chaque activité structurée en un graphe de sous-activités, afin d'aplatir les activités imbriquées et aussi propager les fautes (*fault handlers)* à chaque activité imbriquée dans une autre. Ensuite avons proposé des algorithmes d'améliorations permettant de réduire ces problèmes de testabilité.

6.2 – Travaux en cours et perspectives

Terminons cette thèse, en introduisant quelques travaux envisageables et extensions possibles dans le domaine du test de services Web.

Au cours de cette thèse, nous avons supposé que les messages envoyés et reçus par les services Web sont seulement des messages SOAP. Ainsi, nous avons seulement considéré les interfaces fournies par les descriptions de WSDL qui concernent exclusivement la communication envers les clients. Cependant, des services peuvent être reliés à d'autres ressources, comme par exemple des bases de données. En fait, ces autres messages ne sont pas étudiés dans ce travail, ni même dans la plupart des autres travaux concernant le test des services Web. Ainsi, dans les travaux futurs, nous envisageons de considérer le service Web non plus en tant que boîte noire, mais plutôt en tant que boîte grise à partir de laquelle on pourrait observer n'importe quel genre de messages.

Dans le chapitre 4, nous avons proposé une méthode de test de sécurité de services Web persistants qui consiste à évaluer quelques propriétés de sécurité d'un service Web persistant

grâce à un ensemble de règles. Cette méthode a été crée à l'aide de requêtes de test qui examinent les réponses reçues lorsque des opérations sont invoquées avec un ensemble statique de valeurs (valeurs inhabituelles, injection SQL et injection XML). Plusieurs perspectives peuvent être envisagées, notamment par rapport à l'ensemble de valeurs utilisées. Il serait intéressant, par exemple, de proposer une analyse dynamique permettant de construire une liste de valeurs la plus adaptée possible pour chaque service Web. Une meilleure solution serait de choisir ces paramètres en fonction de la description de l'opération, en utilisant les valeurs qui ont un taux d'erreurs important dans les tests passés et de leur attribuer ainsi des coefficients de pondération.

Notre méthode de test de sécurité est conçue pour les services Web persistants. Cependant, la plupart des services Web sur Internet sont déployés avec l'URL de description WSDL. Donc une extension dédiée aux services non persistants peut être aussi envisagée. Celle-ci permet d'automatiser la construction des tests en considérant seulement les interfaces fournies par le fichier WSDL, ce qui permet de déterminer l'ensemble d'opérations vulnérables sans avoir recours à d'autres spécifications.

Au niveau de l'étude de la testabilité du langage BPEL, d'autres facteurs de testabilité nous semblent aussi intéressants à étudier comme l'accessibilité : étant donné que le langage BPEL est défini par un ensemble d'activités, il est important de déterminer celles qui sont inaccessibles.

Certaines activités peuvent utiliser des variables partagées ce qui peut provoquer des incohérences dans leur comportement ou des blocages et donc rendre inaccessible certaines parties spécifiques de la spécification BPEL. Ainsi, la considération de BPEL en boîte grise et l'analyse des domaines de variables impliquées dans les activités conditionnelles (« switch », « if », « for », « while »,…), présente une extension possible dans le test d'accessibilité de BPEL.

Beaucoup d'autres critères de qualité peuvent aussi être étudiées comme:

- Le temps d'exécution, qui évalue le coût des tests selon les temps d'exécutions minimales et maximales.
- Le degré de contrôle servant à estimer le coût moyen (ou max) pour exécuter les tests sur un système donné et qui consiste ici à évaluer les efforts nécessaires pour atteindre chaque état de la spécification BPEL.
- La résistance aux charges et la capacité de traiter des requêtes en parallèles sont aussi des critères de qualité qui peuvent être intéressant à étudier.

Il est également possible de réduire ce temps d'exécution en traduisant les activités séquentielles et indépendantes en activités parallèles grâce à l'activité «flow».

Nous avons attaqué la composition décrite en orchestration BPEL dont les interactions sont dirigées de point vers multipoint (et inversement). Il serait aussi intéressant de tester la composition en chorégraphie dont les échanges sont en mode point à point. Dans ce dernier cas, l'utilisation d'une plateforme distribuée est nécessaire afin de pouvoir intercepter les messages SOAP reçus et aussi de contrôler ceux émis par les différents partenaires. Cette plateforme serait composée de PCOs qui devront être aussi synchronisés afin de pouvoir vérifier la chronologie des échanges. Une possibilité consiste à utiliser le protocole NTP (Network Time Protocol) qui permet d'avoir un temps global synchronisé entre tous les partenaires (services Web et coordinateur).

Bibliographie

[1] Andrews T. et al.. Business Process Execution Language for Web Services (WSBPEL). 2005

[2] Bai X., Dong W., Tsai W.-T. & Chen Y. (2005). Wsdl-based automatic test case generation for Web services testing, SOSE '05: Proceedings of the IEEE International Workshop, IEEE Computer Society, Washington, DC, USA, pp. 215–220

[3] Baldoni M. and Baroglio C. and Chopra A. K. and Desai N. and Patti V. and Singh. M. P.. Choice, Interoperability, and Conformance in Interaction Protocols and Service Choreographies, 2009

[4] Bartolini C., Bertolino A., Marchetti E. & Polini A. (2009). Ws-taxi: A wsdl-based testing tool for Web services, ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation, IEEE Computer Society, Washington, DC, USA, pp. 326–335

[5] Beek M. H. T., Bucciarone A., and Gnesi S.. A Survey on Service Composition Approaches : From Industrial Standards to Formal Methods. Technical Report 2006-TR-15, Consiglio Nazionale delle Ricerche, 2006

[6] Belinfante A., Frantzen L., and Schallhart C.. Tools for test case generation. InM. Broy, B. Jonsson, J.-P. Katoen,M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, number 3472 in Lecture Notes in Computer Science No 3472, chapter 14. Springer Verlag, 2005

[7] Bertolino, A., Frantzen, L., Polini, A. & Tretmans, J. (2006). Audition of Web services for testing conformance to open specified protocols, in R. Reussner, J. Stafford & C. Szyperski (eds), Architecting Systems with Trustworthy Components, number 3938 in LNCS, Springer-Verlag, pp. 1–25

[8] Bessayah F., Cavalli A., Maja W., Martins E., and Valenti A.W., A Fault Injection Tool for Testing Web Services Composition, TAIC PART 2010, Windsor, UK, September 2010

[9] Breugel F. V. and Koshkina M.. Models and verification of BPEL, 2006. http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf, visité le 29/09/2011

[10] Brinksma E. and Tretmans J. . Testing transition systems : An annotated bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000, volume 2067 of Lecture Notes in Computer Science, pages 196–205. Springer, 2000

[11] Brown. P.J. The Stick-e Document: a framework for creating Context aware applications. In Electronic Publishing 96, (1996), Document Manipulation and Typography, 24-26 September 1996, Pala-Alto, California, USA, pp. 259-272

[12] Busi N.et al.Choreography and Orchestration: a synergic approach for system design. In Proc. of 3rd. International Conference of Service-Oriented Computing ICSOC'05, 2005

[13] Chauvet J. M., Services Web avec SOAP, WSDL, UDDI, ebXML…, Eyrolles, Paris, 2002, http://www.librairiedialogues.fr/livre/120684-services-web-avec-soap-wdsl-uddi-ebxml-jean-marie-chauvet-eyrolles, visité le 29/09/2011

[14] Clatin M., Groz R., Phalippou M., and Thummel R.. Two approaches linking a test generation tool with verification techniques. In A. Cavalli and S. Budkowski, editors, Proceedings of IWPTS'95 (8th Int. Workshop on Protocol Test Systems, Evry, France). INT, sep 1995.

[15] CMMI, Capability Maturity Model Integration, Version 1.2. http://www.sei.cmu.edu/cmmi/, visité le 29/09/2011

[16] COM+ (Component Services), http://msdn.microsoft.com/library/ms685978(VS.85).aspx, visité le 29/09/2011

[17] Christensen E., Curbera F., Meredith G., Weerawarana S., Web Services Descriptor Language (WSDL) 1.1, http://www.w3.org/TR/2001/NOTE-wsdl-20010315, visité le 29/09/2011

[18] Cuppens F., Cuppens-Boulahia N., Sans. T., Nomad : A Security Model with Non Atomic Actions and Deadlines, CSFW, 2005, p. 186-196.

[19] David C. Fallside and Priscilla Walmsley. XML Schema part 0: Primer second edition. W3C recommendation, World Wide Web Consortium, October 2004, http://www.w3.org/TR/xmlschema-0/, visité le 29/09/2011

[20] Een N., Sörensson. N.. SAT 2003. Minisat. http://minisat.se, visité le 29/09/2011

[21] EJB (Enterprise JavaBeans Technology), http://www.oracle.com/technetwork/java/javaee/ejb/index.html, visité le 29/09/2011

[22] Erl Thomas: SOA principles of Service Design, p. 294, 2007

[23] Escobedo J.P., Gaston C., Gall P.L., and Cavalli A, Observability and controllability Issues in Conformance Testing of Web Service Compositions, TestCom/FATES'09 pp 217-222, 2009

[24] Fernandez J. C., Jard C., Jeron T., and Viho G., An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology, *Science of Computer Programming*, 29, 1997, p. 123–146.

[25] Fernandez J. C., Jard C., Jeron T., and Viho G.. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 348–359, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[26] Frantzen, L., Tretmans, J. & Willemse, T. (2005). Test Generation Based on Symbolic Specifications, in J. Grabowski & B. Nielsen (eds), Formal Approaches to Software Testing (FATES), number 3395 in Lecture Notes in Computer Science, Springer, pp. 1-15 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.6550&rep=rep1&type=pdf, visité le 29/09/2011

[27] Frantzen L., Tretmans J. & de Vries R. (2006). To-wards model-based testing of Web services, in A. Bertolino & A. Polini (eds), in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006), Palermo, Sicily, ITALY, pp. 67–82

[28] Freedman R.S., "Testability of Software Components," IEEE Transactions on Software Engineering, vol. 17, no. 6, pp. 553-564, June, 1991

[29] Gallois J.-P., Gaston C., Lapitre A. (CEA) AGATHA, un outil de simulation symbolique, AFADL 2004, Juin 2004, Besançon (France)

[30] Gaston C., Le-Gall P., Rapin N., and Touil A.. Symbolic execution techniques for test execution techniques for test purpose definition. In Proceedings of Testing of Communicating Systems, pages 1–18. Springer, 2006

[31] Gorbenko A. & al. (2008). The threat of uncertainty in service-oriented architecture, SERENE ’08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems. ACM.

[32] Gruschka N. and Luttenberger N., Protecting Web Services from DoS Attacks by SOAP Message Validation, in Proceedings of the IFIP TC11 21 International Information Security Conference (SEC}, 2006

[33] Heineman G. T.& Council W. T.. Component-Based Software Engineering, Putting the Pieces Together. Amsterdam : Addison Weysley, 2001.

[34] Heerink L.. *Ins and outs in refusal testing*. PhD thesis, University of Twente, Institute for programming Research and Algorithmics, May 1998.

[35] Herzberg A., Massy Y., Mihaelij J., Naord D., Ravid Y., Access Control Meets Public Key Infrastructure, Or : Assigning Roles to Strangers , IEEE Symposium on Security and Privacy, 2000, p. 2-14

[36] IEEE Standard for Software Test Documentation, 1998 (ISBN 0-7381-1444-8)

[37] ISO. Conformance Testing Methodology and Framework. International Standard 9646, International Organization for Standardization – Information Technology– Open Systems Interconnection, Genève, 1991.

[38] ISO/IEC 9646-1:1994 http://www.iso.org/iso/fr/catalogue_detail.htm?csnumber=17473, visité le 29/09/2011

[39] ISO. Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard IS-8807. ISO, Geneve, 1989.

[40] ITU. CCITT., "Specification and Description Language(SDL)." Recommendation Z.100, 1992.

[41] Jeannet B., Jéron T., Rusu V., and Zinovieva E.. Symbolic test selection based on approximate analysis. In Proceedings of International Conference on Tools and Algorithms

for the Construction and Analysis of Systems (TACAS'05), volume 3440 of Lecture Notes in Computer Science, pages 349–364. Springer, 2005

[42] Kaschner K. and Lohmann N.. Automatic Test Case Generation for Services. In Monika Solanki, Barry Norton, and Stephan Reiff-Marganiec, editors, 3rd Young Researchers Workshop on Service Oriented Computing (YR-SOC 2008), June 2008

[43] Kiezun A., Ganesh V., Guo P. J., Hooimeijer P. & Ernst M. D. (2009). Hampi: a solver for string constraints, ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis, ACM, New York, NY, USA, pp. 105–116

[44] Kropp N. P., Koopman P. J. & Siewiorek D. P. (1998). Automated robustness testing of off-the-shelf software components, FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, IEEE Computer Society, Washington, DC, USA, p. 230

[45] Lee D. and Yannakakis M.. Principles and Methods of Testing Finite State Machines : a survey. Proceedings of the IEEE, 84(8) :1090-1123, Aug 1996.

[46] Looker N., Munro M., Xu J. (2004). Ws-fit: A tool for dependability analysis of Web services, in Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts (COMPSAC'04). IEEE Computer Society Press, Vol. 02.

[47] Mallouli W. and Bessayah F. and Cavalli A. and Benameur A.. Security Rules Specification and Analysis Based on Passive Testing, The IEEE Global Communications Conference (GLOBECOM 2008), 2008

[48] Mallouli W., Lallali M., Morales G. and Cavalli A.R.: Modeling and Testing Secure Web-Based Systems: Application to an Industrial Case Study, The fourth International Conference on Signal-Image technology & Internet-Based Systems (SITIS 2008), Bali, Indonesia, November 30 - December 03, 2008.

[49] Manes A. T.: Service-Oriented Architecture: Developing the Enterprise Roadmap, Version: 2.0, 2006. p. 22 http://i.i.com.com/cnwk.1d/html/itp/burton_service_orient.pdf, visité le 29/09/2011

[50] Marc Phalippou. *Relations d'implantation et hypothèses de test sur des automates à entrées et sorties*. PhD thesis, Université de Bordeaux I, 1994

[51] Martin E. & Xie T. (2006). Automated test generation for access control policies, Supplemental Proc. 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)

[52] Morimoto S.. A Survey of Formal Verification for Business Process Modeling. In Proceedings of the 8th international conference on Computational Science, Part II (ICCS '08), pages 514–522, Berlin, Heidelberg, 2008. Springer-Verlag

[53] Object Management Group. "Telecom Task Force Roadmap". OMG Document telecom/01/04/02. Avril 2001

[54] Offutt J. & Xu W. (2004). Generating test cases for Web services using data perturbation, in S. E. Notes (ed.), ACMSIGSOFT, Vol. 29(5), pp. 1–10

[55] OWASP , https://www.owasp.org/index.php/Testing_for_XML_Injection_(OWASP-DV-008), visité le 29/09/2011

[56] OWASP, https://www.owasp.org/index.php/Web_Services, visité le 29/09/2011

[57] Petrenko. A. Fault model-driven test derivation from finite state models : Annotated bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000, volume 2067 of Lecture Notes in Computer Science, pages 196–205. Springer, 2000

[58] Phalippou M.. Tveda : an experiment in computer-aided test case generation from formal speci_cations of protocols. In Note Technique NT/LAA/SLC/347, CNET, January 1991

[59] Pucel X., Bocconi S., Picardi C., Dupré D. T., and Massuyès L.T., Diagnosability Analysis for Web Services with Constraint-based Models, DX-07, Nashville, TN, USA, 2007

[60] Ribeiro C., Zuquete A., Ferreira P., Guedes P., SPL : An Access Control Language for Security Policies with Complex Constraints », In Proceedings of the Network and Distributed System Security Symposium, 2001

[61] SOAP, "Simple object access protocol v1.2 (soap)." World Wide Web Consortium, June 2003

[62] Szyperski C. Component Software Beyond Object-Oriented Programming. Second Edition. London : Addison-Wesley / ACM Press, 2002, 589 p.

[63] Szyperski C. Component Software. New York : ACM Press, 1997, 411 p

[64] Tidwell, D. Web services, the web's next revolution, IBM developerWorks, 2000 http://www.cn-java.com/download/book/wsbasics-a4.pdf, visité le 29/09/2011

[65] Tom Bellwood, IBM UDDI Version 2 API Specification. UDDI Version 2.04 API, Published Specification, Dated 19 July 2002, http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm, visité le 29/09/2011

[66] Salva S. and Rabhi I., Automatic Web service robustness testing from WSDL descriptions, In 12th European Workshop on Dependable Computing, EWDC 2009, may 2009, Toulouse, France

[67] Salva S. and Rabhi I., Robustesse des Services Web persistants, In MOSIM10, 8ème ENIM IFAC Conférence Internationale de Modélisation et Simulation, may, 2010, Hammanet, Tunisie

[68] Salva S. and Rabhi I., Statefull Web service robustness, In The Fifth International Conference on Internet and Web Applications and Services, ICIW10, p. 167-173 , IEEE Computer Society Press, May 9 - 15, 2010 , Barcelona, Spain

[69] Salva S. and Rollet A., Testabilité des services web, In Ingénierie des Systèmes d Information RSTI série ISI, numéro spécial Objets, composants et modèles dans l ingénierie des SI, vol. 13, nb. 3, p. 35-58, Hermes, Lavoisier, 2008

[70] Schéma XML, http://www.w3.org/XML/Schema, visité le 29/09/2011

[71] Touil A.. Exécution symbolique pour le test de conformité et le test de raffinement, Thèse de doctorat, laboratoire IBISC, université d'Évry Val d'Essonne 2006

[72] Tretmans J.. A Formal Approach to Conformance Testing. PhD thesis, University of Twente, Enschede, The Netherlands, 1992

[73] Tretmans J. (1996). Test generation with input, outputs, and repetitive quiescence, Software - Concepts and Tools 17: 103-120

[74] UDDI, http://uddi.xml.org/uddi-org, visité le 29/09/2011

[75] Verhaard Louis, Tretmans Jan, Kars Pim, and Brinksma Ed. On asynchronous testing. In Gregor von Bochmann, Rachida Dssouli, and Anindya Das, editors, *Protocol Test Systems*, volume C-11 of *IFIP Transactions*, pages 55–66. North-Holland, 1992

[76] Vieira M., Laranjeiro N. & Madeira H. (2007). Assessing robustness of web-services infrastructures, In Proc. of the Int. Conf. On Dependable Systems and Networks (DSN'2007)

[77] WebMov http://webmov.lri.fr, visité le 29/09/2011

[78] WS-BPEL 2.0 http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf, visité le 29/09/2011

[79] WS-I, "Ws-i basic profile." WS-I organization, 2006