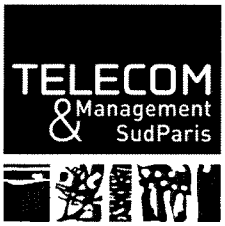
A testing methodology for the validation of web applications

Gerardo Morales



Thèse de doctorat de l'INSTITUT NATIONAL DES TELECOMMUNICATIONS dans le cadre de l'école doctorale S&I en co-accréditation avec l' UNIVERSITE D'EVRY-VAL D'ESSONNE

Spécialité :
Informatique

Par
Gerardo MORALES

Thèse présentée pour l'obtention du grade de Docteur de l'INSTITUT NATIONAL DES TELECOMMUNICATIONS

Une méthodologie de test pour la validation des applications Web

Soutenue le 13/07/2010 devant le jury composé de :

Prof. Abdelhamid MELLOUK	Université de Paris XII	Président
Dr. Ismael RODRIGUEZ	Universidad Complutense de Madrid	Rapporteur
Prof. Eliane MARTINS	Universidade Estadual de Campinas	Rapporteur
Dr. Kequin LI	SAP France	Examinateur
Dr. Stéphane MAAG	Institut Télécom Sud-Paris	Co-encadrant de thèse
Prof. Ana CAVALLI	Institut Télécom Sud-Paris	Directrice de thèse

Thèse n° 2010TELE0014

Contents

List of Figures

Acknowledgment

I achieved this work thanks to the help and support of many persons. First of all I want to thank Professor Ana Cavalli for giving me the opportunity to come to France and to do this PhD work under her direction.

I am also very grateful with Dr. Stéphane Maag for his guidance and patience as supervisor of this work. I want to express also my gratitude to Montimage and Edgardo Montes de Oca for their support during this work.

I want to give special thanks to my wife Marcela for her love, for reading this thesis (several times), and for being cheerful in the good times and supportive in the bad ones.

I would like to particularly express my gratitude to the members of the jury of the defense of this work. Thank you Professor Martins and Dr. Rodríguez for reading, evaluating, and helping me with your experience and suggestions. Thank you Professor Mellouk and Dr. Li for accepting to be part of this work and sharing with me your experience in the area and knowledge.

I am also appreciative towards the LOR department, thank you Jopez, Fayçal, Bakr, Mazen, IkSoon, Anis and the rest of the current LOR team. Thank you Wissam, Bachar and Mounir for all the great moments and adventures we lived at the Lab and for your support in technical and personal aspects. Thank you Brigitte and Jocelyne for helping me in all the administrative issues and for your kindness.

Finally COMA I want to thank my family and close friends: my parents Bernardo and Marie-Claire; to Alfredo, Maria Angelina, Jean Pierre, Bea, Luc, Rober, Michou and the rest of the Cadorets. Also to Dafné, Xmucané, Clara, Stephany, Bernardo, Barbara, Sofia and the rest of my family in Guatemala.

I hope they find here the expression of my deep gratitude and appreciation.

To my beloved wife, Marcela

Abstract

Abstract

Nowadays, Web based systems gained in importance as they are dominating the synchronous and asynchronous collaboration between institutions and their remote users. Web applications have evolved in complexity to become powerful communication tools between institutions and users. Moreover, in recent years, Web applications have evolved towards the Service Oriented Architecture (SOA). This allows the reuse of existing services to build new and more complex applications in less time.

The objective of this thesis is to ensure the proper behavior of the functional aspects of Web based systems. To achieve this goal, we proposed two different test approaches: the active approach and the passive approach. Our goal is to automatically generate a suite of active test scenarios that will be applied on a system under test to examine its compliance with respect to its functional specification, and, when interrupting the normal flow of operation is problematic, to observe the system under test with passive testing.

The goal of this work is developing a method and a set of tools to test Web based systems by using the active and passive testing approaches. Concerning the active testing approach, we presented a methodology to cover the end-to-end testing process (from building the model until the test execution). This work tackles the gap between, on the one hand, generating abstract test cases from abstract models and, on the other hand, developing methods allowing concretizing these tests and automatically applying them on real applications. Then, concerning the passive testing approach, we presented a methodology and a new tool for observing the behaviour of the communications of the Web application with external Web services (for SOA based Web applications) in order to check whether the observed behaviour is correct. All the methodologies and tools presented in this work are applied on two industrial case studies, Mission Handler and Travel Reservation Service, in order to validate our contributions in active and passive testing respectively.

Résumé

De nos jours, les systèmes basés sur le Web sont devenus de plus en plus importants en tant que éléments de base pour la collaboration synchrone ou asynchrone entre utilisateurs distants. D'une part, les applications Web (e.g. e-government ou e-banking) ont évolué en complexité pour devenir un moyen de communication puissant entre les petites/grandes institutions et les internautes, et entre les internautes eux-mêmes. D'autre part, ces dernières années, les applications Web et plus généralement, toutes les applications, ont évolué vers une nouvelle architecture qui substitue un seul grand système par des systèmes multiples qui collaborent entre eux. Cette architecture est orientée services et se nomme SOA pour " Service Oriented Architecture ". Elle permet la réutilisation de services existants pour construire de nouveaux services Web plus complexes en moins de temps.

Certains institutions utilisent des systèmes basés sur le Web pour administrer des tâches critiques relatives à leurs métiers ou pour gérer des données sensibles et confidentielles (e.g. des données utilisateurs, des données bancaires etc.). Un comportement erroné de ce type de système peut être exploitable par des actions malveillantes ou des attaques pour l'accès à ces données, par des utilisateurs ou des systèmes internes ou externes.

L'objectif de cette thèse est d'assurer le bon comportement des aspects fonctionnels des systèmes basés sur le Web. Pour atteindre cet objectif, nous nous basons dans ce manuscrit, sur deux approches différentes de test: l'approche active et l'approche passive. Le principe du test actif consiste à générer automatiquement une suite de scénarios de tests qui sera appliquée sur un système sous test pour en étudier sa conformité par rapport à ses besoins fonctionnels. Quand au test passif, il consiste à observer passivement le système sous test, sans interrompre le flux normal de ses opérations.

Pour l'approche active, nous proposons une méthodologie qui permet de générer automatiquement des séquences de test afin de valider la conformité d'un système par rapport à la description formel du comportement du système. Le comportement est spécifié en utilisant un modèle formel basé sur des machines à états finis étendues temporisées (TEFSM). La génération automatique des tests est ensuite effectuée en utilisant des outils développés dans notre laboratoire et permet d'obtenir des cas de tests exécutables qui permettent au moteur de test d'interagir avec une application Web réel.

Dans l'approche passive, nous spécifions des propriétés fonctionnelles à tester sous la forme d'invariants temporisées. Nous analysons ensuite les traces d'exécution d'un Web service composé afin d'élaborer un verdict sur sa conformité par rapport au comportement souhaité du système.

Plusieurs algorithmes et outils sont fournis dans ce manuscrit pour effectuer le test actif et passif des systèmes Web. Nous avons appliqué nos méthodologies à divers systèmes (le Mission Handler et le Travel Reservation Service) pour illustrer les approches proposées sur des systèmes réels.

Résumé En Français

0.1 Introduction

De nos jours, les systèmes basés sur Web constituent une solution puissante et facile pour permettre une collaboration synchrone ou asynchrone entre des utilisateurs distants et utilisant des plateformes hétérogènes. Les concepts d'e-commerce, e-banking, e-gouvernement et e-learning sont devenus d'importants canaux de communication et les applications Web ont évolué dans la complexité de devenir des outils puissants pour gérer une collaboration fructueuse en ligne. Par ailleurs, dans les dernières années de grands progrès ont été réalisés dans les technologies Web, le concept d'une application autonome de grande taille a évolué vers le concept d'applications multiples spécialisées et interconnectées (appelé Web services). L'architecture qui utilise ce concept est appelée Service Oriented Architecture (SOA). Elle permet la réutilisation d'applications existantes et par conséquent, la possibilité de construire plus efficacement des applications Web en moins de temps.

Parallèlement à cette évolution, la sophistication et la complexité des plates-formes basées sur le Web sont de plus en plus importantes. Par conséquent, il devient de plus en plus difficile et coûteux d'assurer le bon déploiement des systèmes et de leur fiabilité. Cette situation devient pire encore en raison de l'absence de normalisation dans les clients Web (navigateurs).

Certaines institutions utilisent des technologies Web pour gérer les tâches critiques ou de administrer des données sensibles (par exemple les banques). Toutefois, les erreurs fonctionnelles dans ces systèmes peuvent être exploitées pour effectuer des intrusions, des attaques ou d'accéder à des données sensibles. Ceci explique l'effort que réserve les développeurs pour concevoir les meilleurs tests possibles.

Le test de logiciels consomme en général entre 30 et 60 pour cent de coût global du développement [UL06]. En outre, en raison de l'utilisation croissante de la SOA dans les systèmes modernes, il y a un besoin émergeant afin de développer de nouvelles méthodologies de test qui peut être appliquées sur des systèmes distants ou vus comme des boites noires.

Il y a également un besoin croissant pour réduire le temps et l'argent dépensés pour tester des systèmes modernes basés sur le Web puisqu'ils deviennent de plus en plus complexes à tester. Beaucoup d'efforts ont été fait dans la recherche et l'industrie pour automatiser l'exécution des cas de test pour ce type de systèmes. Et il existe un grand intérêt dans l'utilisation du test basé sur des modèles (Model-Based Testing) afin de pousser encore plus loin le niveau d'automatisation et assurer ainsi une meilleure couverture des tests et développer des systèmes mieux testés et plus fiables. Il est donc intéressant de concevoir de nouveaux outils permettant d'automatiser aussi la génération de tests pour valider les systèmes Web.

0.1.1 Motivations

Aujourd'hui, différentes organisations adoptent des applications Web pour se connecter avec des systèmes distants. Ceci a l'avantage d'avoir plus de souplesse dans l'utilisation des outils de collaboration synchrone et asynchrone entre les utilisateurs du Web [CMPV05a].

Certains travaux ont été réalisés pour tester les applications Web, les méthodes et outils existants, comme celles présentées dans [HK06], permettent l'automatisation de l'exécution de test. Les cas de tests sont générées manuellement et re-exécutés plus tard dans le but d'accélérer les tests de régression. Il existe cependant des efforts à fournir afin d'automatiser la génération de cas de tests pour les applications Web.

Dans notre point de vue, la meilleure façon d'éviter l'intervention humaine dans la génération de test est de se baser sur le Model Based Testing (MBT). MBT nécessite une description abstraite du système sous test qui soit lisible par une machine et qui soit sans ambiguïté. Cette description constitue le modèle du système à tester à savoir un modèle formel de l'implémentation sous test (IUT).

Malheureusement, aujourd'hui la génération de test pour les systèmes basés sur le Web est généralement manuelle. Cela est dû à l'absence d'instruments bien adaptés pour spécifier formellement les systèmes basés sur le Web, une étape nécessaire pour la génération automatique de cas de test. En regardant de près dans ce domaine, on peut constater un écart entre le monde de la recherche et l'industrie. D'un côté, la communauté des chercheurs a mis au point pendant des décennies les fondements de préciser la théorie des systèmes utilisant des modèles mathématiques abstraits, puis de générer des cas de tests abstraits pour ces modèles. D'un autre côté, l'intérêt de la communauté industrielle se concentre sur l'exécution des cas de test sur les implémentations du monde réel. Il y a un besoin à combler cette lacune et de compléter la chaîne, y compris les spécificités requises des systèmes Web de la phase de modélisation, afin de générer automatiquement et instancier des cas de test exécutables qui peuvent stimuler directement le système sous test.

En outre, les applications Web évoluent vers une nouvelle génération basés sur l'architecture orientée services, ce qui porte de grandes possibilités de construire des outils puissants en moins de temps à l'aide des systèmes hétérogènes et distants. Dans la plupart des cas, les applications SOA utilisent des systèmes tiers. Dans ce contexte distribué, le test actif peut être assez difficile à réaliser car il doit manipuler/contrôler des systèmes Web distants.

De plus, pour réaliser un test complet d'une application Web moderne, il est nécessaire d'utiliser une approche active de tests pour tester ses fonctionnalités à partir de son interface utilisateur, mais en même temps il y a un intérêt dans l'application d'une approche passive pour les tests pour vérifier de conformité de la communication de l'application Web avec les services Web distants.

Ce travail de thèse explore des méthodologies et des outils existants pour effectuer différents types de tests sur les systèmes basés sur le Web. Les principales motivations sont les suivantes: (i) proposer des méthodes et des outils nouveaux pour combler le fossé entre les communautés de la recherche et de l'industrie concernant le test de systèmes basées sur le Web. (ii) proposer de nouveaux concepts, méthodologies et outils pour tester des applications Web qui utilise l'architecture SOA de collaborer avec les services Web

distants.

0.1.2 Contributions

Le travail réalisé au sein de cette thèse s'inscrivent dans le cadre du projet E-LANE de l'Union européenne [1] qui vise à construire de nouvelles technologies Web à l'appui d'e-learning et le projet national ANR Webmov [2] qui traite des tests fonctionnels et de robustesse des services Web.

Dans ce travail, nous proposons deux approches différentes afin de tester les aspects fonctionnels des systèmes basées sur le Web. La première approche est basée sur les techniques de tests actifs et nous proposons un cadre pour générer automatiquement des séquences de test pour valider les aspects fonctionnels d'une application Web à partir de son interface utilisateur. Le comportement fonctionnel du système est spécifié en utilisant une technique de description formelle basée sur les machines à états finis étendue (EFSM). Ensuite, la génération automatique de tests est effectuée en utilisant des outils dédiés à produire des suites de tests dans un langage exécutable capable d'interagir avec de vraies applications Web via son interface utilisateur.

La deuxième approche est basée sur une technique de test passif. Nous analysons les traces recueillies de la communication entre le système sous test et des services Web externes afin d'en déduire un verdict de leur conformité à l'égard de ses exigences fonctionnelles. Cette approche propose un nouvel outil, une nouvelle méthodologie et introduit les invariants étendues en aspects temporisés, une amélioration de concepts existants effectivement utilisés dans le réseau de techniques de test passif.

Ces contributions ont donné lieu à plusieurs publications dans des chapitres de livres, conférences et ateliers internationaux, comme [CMM07], [MMC08], [MLMC08], [CLM+08], [MLM09] et [GMBW].

0.1.3 Plan du document

Ce manuscrit de thèse est organisé en deux chapitres:

1. La première partie présente notre contribution sur le test actif des applications Web en utilisant MBT.

2. Le deuxième chapitre présente notre contribution dans le test passif de systèmes Web hétérogènes à l'aide des services Web composés.

Le premier chapitre présente notre première contribution. Ce travail vise à obtenir la spécification formelle et les tests actifs d'applications Web via son interface utilisateur. Notre cadre permet de générer de manière automatique un ensemble de séquences de test, puis de les exécuter afin de valider leur conformité par rapport à la description

[1] Plus d'informations dans http://git.ucauca.edu.co/e-lane/index-en.html
[2] Plus d'informations dans http://webmov.lri.fr/

fonctionnelle du système. Son comportement est spécifié en utilisant une technique de description formelle basée sur la machine d'état finis étendue (EFSM) [BDAR98]. Ensuite, la génération automatique de tests de cas de tests abstraits est effectuée en utilisant l'outil TestGen-IF (développées dans notre laboratoire) et la dérivation des cas de test exécutable s'effectue à l'aide de notre outil GeneraTCL. Enfin, nous présentons le *Mission Handler*, une étude de cas du monde réel pour démontrer la fiabilité de notre approche.

Après avoir couvert les méthodes pour effectuer des tests actifs sur les systèmes basées sur le Web, nous étendons ce travail dans le deuxième chapitre en tenant compte des nouvelles technologies basées sur l'architecture SOA. Il est présenté une nouvelle méthode et un nouvel outil pour réaliser les tests de des systèmes hétérogènes et réparties par une méthode de test passif. Ce chapitre présente une amélioration des structures mathématiques qui sont utilisés pour effectuer des tests passifs et introduit les invariantes étendues.

Enfin, le dernier chapitre conclut le manuscrit de thèse en résumant nos principales propositions et contributions dans le domaine des tests formels pour les systèmes basées sur le Web et présente quelques perspectives pour notre travail.

0.2 Test Actif

Avec le développement continu d'applications Web de plus en plus complexes, il devient crucial d'exécuter d'une manière formelle une série de cas de test pour assurer sa fiabilité fonctionnelle et de s'assurer que les application Web fonctionnent toujours même après des modifications appliquées à leur implémentation.

Le processus complet détaillant les étapes du test actif des applications Web est décrit dans la figure 1. La première contribution se fait dans la partie principale du processus, la modélisation formelle. L'ensemble du processus de génération de test est basé sur le modèle formel de l'application Web. La méthode utilisée pour obtenir un tel modèle spécifique pour l'application Web est présentée dans ce document.

La partie suivante du processus, est la simulation du modèle et la génération automatique de cas de tests abstraits (tests sans les détails de l'implémentation). Dans l'étape suivante, ces cas de test seront utilisés pour tester une application réelle Web qui mène à la deuxième contribution dans des tests actifs; l'instanciation des cas de tests abstraits en cas de test concrets. L'instanciation permet les cas de test générés d'être exécuté sur une application Web réelle Ce document explique le processus de concrétisation de cas de test et la traduction en code exécutable qui gère communications HTTP. Il explique également comment les résultats définis dans le modèle formel sont utilisés pour générer la logique qui attribuera le verdict du cas de test.

Ensuite, une étude de cas nommée " Mission Handler " illustre notre méthodologie. Finalement, les résultats de nos méthodologies et les résultats de l'exécution des cas de test sur le gestionnaire de la mission sont présentés.

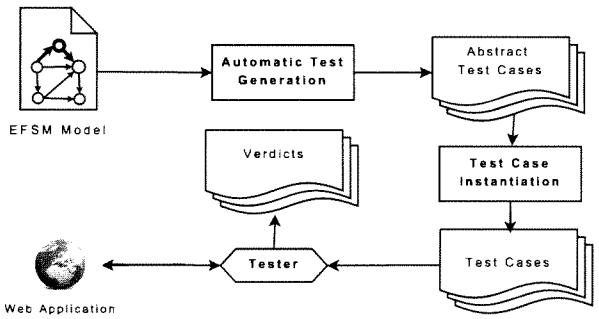


Figure 1: Méthodologie de génération de tests

0.2.1 Modélisation des applications Web en utilisant TEFSM

En théorie, l'objectif de la modélisation est de fournir une meilleure compréhension commune des systèmes. En outre, l'objectif de la modélisation d'une application Web est de fournir un cahier des charges d'exploitation à partir d'un point de vue fonctionnel. Il permet de donner une représentation compréhensible de l'implémentation où certains détails sont extraits du comportement global du système.

Un modèle formel peut également être utilisé comme entrée pour des outils de validation et vérification (tels que les simulateurs interactifs ou aléatoire). Cette méthode de modélisation est inspirée des oeuvres qui utilisent des techniques formelles pour modéliser des applications Web en utilisant SDL EFSM décrit dans [SM03a], en tenant compte des aspects temporelles comme le travail dans [MNR08].

Pour atteindre l'objectif de modélisation, nous nous appuyons dans cette approche sur le modèle TEFSM soutenu par le langue IF [BGO+04] qui fournit les principaux concepts de la conception de systèmes avec contraintes du temps. En outre, plusieurs outils permettant la simulation et la génération de séquences de test (un ensemble de cas de test) existent et sont open source. Un modèle TEFSM d'un système se compose d'un ensemble de processus, chacun désigne un TEFSM qui peut communiquer entre eux via des canaux FIFO.

Après avoir défini le type de structure "mathématiques" qui seront utilisés pour décrire une application Web (TEFSM), il est nécessaire de définir une méthodologie pour construire le modèle. Nous avons besoin d'obtenir un modèle qui simplifie le processus pour atteindre la génération automatique des cas de test.

Par exemple, dans certaines approches en utilisant SDL pour modéliser des applications Web [SM03a], ses utilisateurs font partie du modèle.

Nous avons choisi de modéliser le comportement de l'application Web du point de vue de l'utilisateur. De cette façon, l'utilisateur fait partie de l'environnement qui interagit avec le modèle. En choisissant ce, nous éviter de fournir au modélisateur la liberté d'ajouter des politiques de sécurité (ou même des processus logique) du côté des utilisateurs plutôt que sur le côté de l'application.

Exigences des Modéles pour le test Web

Les méthodes qui nous servent pour construire le modèle du système Web sont basés sur le respect de ces exigences fondamentales:

1. Le modèle doit être lisible et compréhensible par une machine, ce qui signifie un modèle sans ambiguïtés, où l'intervention humaine est nécessaire. L'utilisation de modèles mathématiques formels tels que la TEFSM est alors pertinente.

2. Le modèle doit contenir le comportement des fonctionnalités (les données et les contraintes temporelles), qui sera testé.

3. Le modèle spécifie les interactions de l'IUT avec l'environnement comme une boîte noire. De cette façon, le système est spécifié du point de vue de l'utilisateur qui interagit avec l'IUT à travers son interface Web.

4. Les utilisateurs devraient faire partis de l'environnement et ne fait pas partie de l'TEFSM.

0.2.2 Résumé cas de test de production

Précédente ment le document a porté sur la méthodologie pour construire un modèle formel d'une application Web, les sections qui suivent présentent la génération automatique de cas de tests abstraits de la spécification IF. Cette génération se fait en utilisant l'outil *Testgen-IF*, développé dans notre laboratoire. Néanmoins, pour tester une application réelle Web, le cas de tests abstraits doit être instancié pour obtenir un test concret et exécutable.

Le cas de test abstrait est la base utilisée pour construire le scénario de test instancié (ou cas de test réel) qui seront utilisés pour tester un système réel.

Génération de tests avec TestGen-IF

TestGen, IF est un outil conçu pour générer automatiquement des séquences de test en fonction des objectifs de test. Nous avons également définir manuellement, par regroupement autour d', adéquat valeurs de l'intervalle pour les variables de données afin de réduire la taille de graphe d'accessibilité et d'éviter les problèmes d'explosion de l'État.

Une série de cas de test abstraites sont générés sur la base des IF spécification de l'application Web et les objectifs des tests pour chaque règle à l'aide TestGen-IF. Ces cas de test sont ensuite filtrés en fonction des actions (entrée, de sortie et des retards) sur le système sous test.

Le cas de test abstraite résultante est le chemin de l'exploration d'arbres d'accessibilité partielle décrite dans un format d'entrée/sortie comme illustré à la figure 2. Par exemple, la première ligne de cette figure montre l'entrée / sortie de trace lorsque l'utilisateur se connecte au système. Il montre que le signal d'entrée *login* avec les variables *user* et *userp*, la sortie est le signal *htmchunk* avec la variable *text_to_find_0*; les deux signaux appartiennent à au procesus *webint* du modèle IF.

```
1. {webint}0 i ?login{user,userp} !htmchunk{text_to_find_0}
2. {webint}0 i ?new_mission{} !htmchunk{text_to_find_1}
3. {webint}0 i {webint}0 ?search_m{boston,sgd,srd} !urlexpr{url_to_compare_1}
```

Figure 2: Chemin de l'exploration du graphe d'accessibilité dans format d'entrée/sortie.

0.2.3 Instanciation de cas de test

Comme il est expliqué précédemment, le modèle d'un système est l'abstraction d'une implémentation réelle. Cela signifie: une simplification du système réel où tous les détails inutiles [3] sont extraites afin de laisser seulement une simple description de son comportement. De cette façon, un modèle est facilement compris et évalués.

Afin d'exécuter les cas de test générés à une véritable application Web, il est obligatoire de les instancier. L'instanciation d'un scénario de test abstrait est composée de deux sous-processus: la concrétisation (l'ajout des détails) et la traduction des scripts exécutables. Le test final est un script qui contient les détails de l'implémentation tels que: l'URL où les pages sont instanciés et les valeurs réelles pour les variables des signaux. Ce processus est appelé l'instanciation cas de test.

Pour être en mesure de tester une application Web, nous avons besoin d'un outil capable de communiquer via HTTP (ou HTTPS) avec l'implémentation sous test (IUT). L'instanciation est réalisée à l'aide de notre outil *GeneraTCL*, qui est présenté dans la section suivante.

GeneraTCL

Le outil *GeneraTCL* , illustré dans la figure 3, sert à concrétiser les cas de tests abstraits et de les traduire vers un script exécutable capable d'interagir avec l'IUT. Dans le processus de concrétisation, certains détails de l'implémentation (comme le nom d'utilisateur et mot de passe d'un utilisateur réel) sont ajoutés aux cas de tests abstraits. Ces détails sont nécessaires pour effectuer le testeur interaction-IUT. Ensuite, le processus de la traduction en code exécutable, les traces concrètes sont remplacés par:

- TCLwebtest code, qui permet la communication HTTP avec les IUT,
- code TCL, pour construire la structure logique (par exemple une structure *if-then-else* pour fixer le verdict) du cas de test.

Après l'exécution de ces deux processus, la sortie est prévu un test concret et exécutable, soit un cas de test instancié.

[3] initialisations variables ou des requêtes SQL ne font pas partie du comportement de l'application Web du point de vue de l'utilisateur. Par conséquent, ils sont des détails sans importance pour générer des scénarios de test noir boîte.

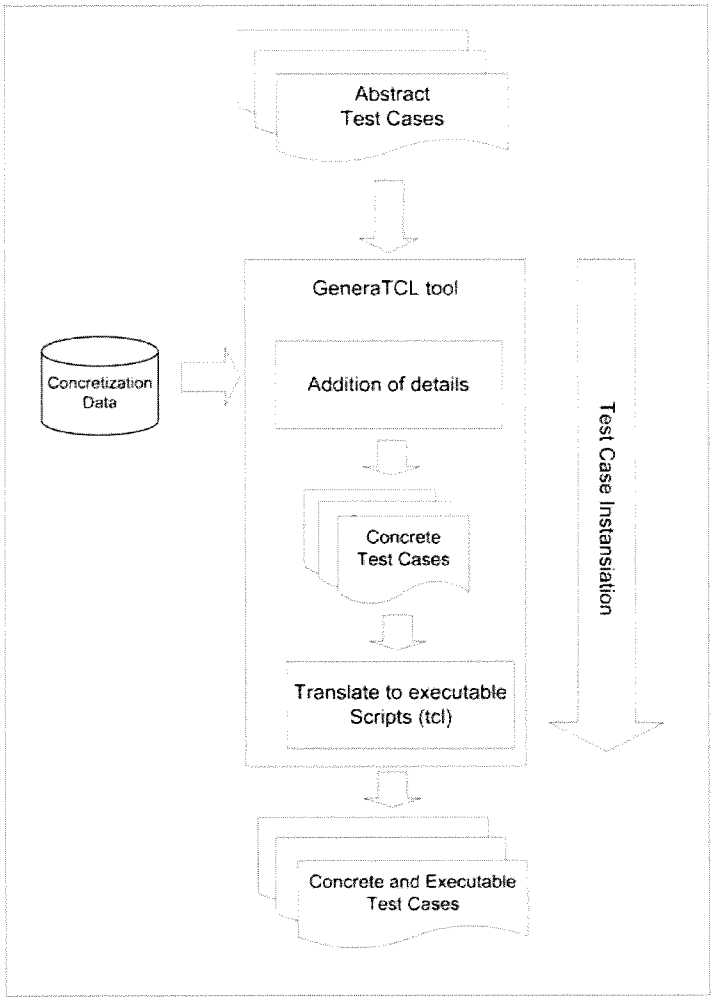


Figure 3: Architecture de base de l'outil GeneraTCL

0.2.4 Méthodologie d'instanciation du GeneraTCL

Pour pouvoir donner les détails sur la méthodologie et les algorithmes de GeneraTCL, nous devons d'abord expliquer la structure des cas de tests abstraits qui sont utilisés comme entrées par GeneraTCL. Ces cas de tests abstraits (généré par l'outil TestGen-IF) sont fournis sous forme de traces entrée/sortie qui représentent:

- Retards: un retard représente un montant de temps que le testeur doit attendre, avant d'effectuer toute action d'entrée.
- Signaux d'entrée: le testeur a de stimuler l'application Web en appliquant un ensemble des requêtes HTTP(S).
- Signaux de sortie: le testeur a accès aux réponses du système Web pour analyser et

de vérifier s'il est conforme à la réaction attendue comme décrit dans la spécification formelle du système.

Ces données sont traitées par *GeneraTCL* de différentes manières, l'instanciation de ces trois types de données sont effectuées de la manière suivante:

Concrétisation du rétard

Le *delay* type de données n'ont pas besoin de concrétisation. Il est directement traduit en un script exécutable dans le cas de tests abstraits. La traduction est illustré plus loin dans cette section.

Concrétisation d'entrée

Pour concrétiser les entrées résumé fourni par l'outil *TestGen-IF*, il est important de connaître les types d'éléments HTML qui correspondent aux signaux d'entrée du système Web à l'essai. Dans ce travail, le système Web sera limitée à trois types de contributions d'un utilisateur via un navigateur régulier: (1) une URL situé dans la barre d'adresse, (2) un lien dans le corps de la page ou (3) la présentation d'un formulaire dans le corps de la page. Des actions telles que le glisser-déposer et d'autres fonctionnalités Ajax ne sont pas considérés dans ce travail.

Le processus de concrétisation de notre méthodologie consiste à cartographier les signaux dans les trois types d'entrées que l'application Web peut recevoir. Il est important de souligner que certains signaux peuvent être mappés à une seule interaction unique avec l'application Web, par exemple, suivant un lien. D'autres signaux sont assignés à un ensemble d'interactions, par exemple, soumettant un formulaire.

Par exemple, compte tenu de la soumission du formulaire, il ya plusieurs interactions qui doivent être effectuées par le testeur, c.à d. en remplissant les champs de texte, en sélectionnant les boutons radio, cases à cocher le choix et enfin la soumission du formulaire. Dans ces cas, le signal est mappé dans un élément HTML (par exemple un formulaire) et aussi à chaque paramètre de signal (par exemple les champs de texte).

Pour effectuer la cartographie, nous proposons deux tableaux contenant les informations requises des signaux d'entrée et leurs paramètres pour les transformer plus tard en un script exécutable. Les deux, le *signal_info_table* et le *parameter_info_table* sont illustrés dans les tableaux 1 et 2. Pour accéder à l'information qu'ils contiennent, nous pouvons tirer profit de la norme SQL. Pour chaque signal, la table *signal_info_table* stocke les données suivantes:

- *signal:* le nom du signal d'entrée dans la spécification IF,
- *html_element:* le type de l'élément HTML qui correspond au signal,
- *html_name:* le nom ou l'id de l'élément HTML. Par exemple, dans le cas d'un lien, c'est l'adresse URL où pointe le lien.

Table 1: Signal_Info_Table Example

signal_name	html_element	html_name
login	form	/register
signup	link	/register/newuser
logout	link	/signout
htmchunk	regtext	void

Puis, pour chaque paramètre d'un signal d'entrée, les informations stockées dans le tableau 2 est:

- *parameter:* nom du signal de paramètre dans la spécification IF,
- *of_signal:* nom du signal d'entrée qui utilise cette variable,
- *html_element:* le type de l'élément HTML qui correspond au paramètre,
- *type:* le type de la variable attendue par les html_element, par exemple, integer, string, etc.
- *html_name:* le nom ou l'identifiant du html_element.

Table 2: Parameter_Info_Table Example

parameter	of_signal	html_element	type	html_name
user	login	textfield	string	email
password	login	textfield	string	password

La dernière partie du processus de concrétisation est de remplacer le nom des variables dans les traces des valeurs que les utilisations de implémentation réelle, par exemple, pour remplacer user1 par gmorales, un nom d'utilisateur réel de la implémentation. La table de substitution est illustré dans le tableau 3.

Table 3: Substitution_Table Example

variable_name	replacement
user	gmorales
password	m!lat0
text_to_find_0	Welcome

Méthodologie instanciation d'entrée

Cette partie de l'instanciation d'entrée utilise des tables 1, 2 et 3 pour faire la correspondance entre les signaux d'entrée et quelques détails de la demande réelle Web. Dans cette méthode, nous utilisons les notations introduites dans le tableau 4.

GeneraTcl utilise en entrée le fichier contenant les cas de tests abstraits, ce fichier est analysé ligne par ligne. Pour chaque ligne, la méthode illustrée à la figure 4 est appliquée pour obtenir l'instanciation des signaux d'entrée. D'ailleurs, laissez-nous définir les Ω

Table 4: Notations utilisées dans la méthode

Notation	Meaning
sg	Signal des cas de tests abstraits (entrées et sorties)
param	Un paramètre de *sg*
SIT.*element*	Un *element* d'un *Signal_Info_Table*
PIT.*element*	Un *element* d'un *Parameter_Info_Table*

Figure 4: Organigramme décrivant la méthodologie de l'instansiation entrée.

fonction permet de définir des pointeurs vers les tables SIT (tableau 1) et du PIT (tableau 2).

Notez que l'utilisation de la méthode d'instanciation d'entrée, pour chaque ligne du scénario de test abstrait l'entrée est capturé, puis classés (lien ou un formulaire), puis

traduit à la TCLwebtest qui interagit avec une application réelle Web.

Enfin nous avons mis les vraies valeurs aux variables des cas de tests abstraits. Ceci est fait en remplaçant les textes dans les cas de tests abstraits en utilisant le tableau de substitution illustrée dans le tableau 1.3. Par exemple, si dans le cas de tests abstraits, nous avons l'entrée *?connexion (user, userp)*, alors on peut remplacer *user* et *userp* avec le vrai nom et mot de passe de l'utilisateur, l'obtention d'un résultat *?connexion(gmorales, m! LT0)*.

Concrétisation de sortie

La sortie prévue dans le cas de tests abstraits prévoit la sortie attendue de l'application Web (IUT) après une interaction avec le testeur. Après cela, nous nous intéressons à l'analyse de la réponse que nous avons obtenus et aussi de connaître l'état actuel vis-à-vis du modèle. La sortie doit fournir des informations que le testeur sera en mesure d'analyser tels que:

- Un ou plusieurs morceaux de HTML, nous pouvons trouver dans la page Web en cours,
- L'URL en cours qui se trouve dans la barre d'adresse du navigateur.

Méthodologie d'Instanciation

Traduction d'un code exécutable

Retard

Le retard se transforme directement le code TCL '*après n* 'où *n* est en millisecondes (ms). Par exemple, la note si dans le cas de tests abstraits " *délai de 10* (Nous considérons cela comme un retard de dix minutes) est trouvée, il sera traduit dans le code TCL' *après 10 * 1000 * 60.*

Entrée

Nous avons besoin de traduire les entrées concrets vers *TCLwebtest* script afin d'interagir avec l'application Web.

Le TCLwebtest actions sont construites dynamiquement par le *GeneraTCL* outil et peut être divisée en trois catégories: suivant un lien, soumettre un formulaire ou en définissant une URL dans le navigateur. La méthode de traduction des entrées est présentée dans un pseudo-code de l'algorithme 0.2.4 (lignes 6 à 36). En effectuant cet algorithme les parties suivantes du cas de test seront construits:

- Le script du préambule de test: une séquence d'entrées (opérations) qui conduira le système à un état où le scénario de test peut être exécuté. Au cours de ce préambule,

Algorithm 1 Instantiation Methodology

```
1: for each (act_i ∈ TC) do
2:    /*(where i ∈ N, 0 < i < n + 1 such that n is the number of actions and delays in TC)*/
3:    case (act_i = delay n) do
4:       TCL_script: after n;
5:    end case
6:    case (act_i = input sg_i(in_1, in_2, ..., in_k)) do
7:       if (html_element(sg_i)= url) then
8:          TCL_script: do request url;
9:       end if
10:      if (html_element(sg_i)= link) then
11:         TCL_script: follow link;
12:      end if
13:      if (html_element(sg_i)= form) then
14:         TCL_script: form find ~n html_name(sg_i);
15:         for (each parameter x_j of sg_i) do
16:            /*(where j ∈ N, 0 < j < k + 1)*/
17:            TCL_script: field find ~n html_name(x_i);
18:            case (html_element(x_j) = textfield) do
19:               TCL_script: field fill in_j;
20:            end case
21:            case (html_element(x_j) = textarea) do
22:               TCL_script: field fill in_j;
23:            end case
24:            case (html_element(x_j) = checkbox) do
25:               if (in_j = 1) then
26:                  TCL_script: field check html_name(x_j);
27:               else
28:                  TCL_script: field uncheck html_name(x_j);
29:               end if
30:            end case
31:            case (html_element(x_j) = radiobutton) do
32:               TCL_script: field select in_j;
33:            end case
34:         end for
35:         TCL_script: submit form;
36:      end if
37:   end case
38:   case (act_i = output sg_i(out_1, out_2, ..., out_k)) do
39:      if (html_element(sg_i)= regurl) then
40:         TCL_script: assert {[response url] == html_name(sg_i)};
41:      end if
42:      if (html_element(sg_i)= regtext) then
43:         for each (parameter x_j of sg_i) do
44:            TCL_script: assert {[field get_value find ~n html_name (x_j)]== out_j} ;
45:         end for
46:      end if
47:      call deduce_verdict procedure;
48:   end case
49: end for
```

les sorties du système ne sont pas analysés. Par exemple, dans un système de Voyage, de tester la création d'une mission, l'utilisateur doit être authentifié par ce système.

- Le script qui stimulent le système pour le tester..

Sortie

La dernière étape de la méthodologie consiste à développer des scripts qui analysent la réponse (ou réaction) de l'application Web. Ce script attribue également le verdict (réussite ou échec). Fondamentalement, il vérifie si l'IUT a fait ce qu'il devait faire ou non.

Les sorties de l'IUT peuvent être classés en deux catégories:

- Sorties Observables : TCLwebtest est essentiellement dédiés à l'analyse d'application Web accessible par l'interface utilisateur Web. Il offre des fonctionnalités de base d'analyse HTML et les commandes pour la manipulation des éléments HTML des pages Web. La réaction du système peut être fournie dans une ou plusieurs pages HTML du système basé sur le Web. En général, il s'agit d'un message de notification qui stipule que l'action souhaitée réussi ou échoué, par exemple une authentification. Parfois, cette réaction peut être plus difficile à découvrir, par exemple rechargement de la page en cours ou de naviguer vers une autre page de l'application Web. Dans tous les cas, nous devons définir deux tableaux 1 et 2 pour la sortie, ainsi que leurs paramètres et de suivre la même méthodologie développée dans le cas d'instanciation d'entrée. Pour analyser le système des pages Web, nous utilisons *response* et *find* qui appartiens au TCLwebtestet pour localiser les éléments HTML que vous voulez tester. Ensuite, nous profitons de la commande *assert* pour comparer les valeurs affichées page Web et les paramètres du signal de sortie, et en déduire le verdict adéquate.

  Il est important de souligner que les commandes TCLWebtest ne sont utilisées que pour communiquer avec une application Web via HTTP, il ne contient pas de commandes logiques telles que *if-then-else*, *while*, etc. Par conséquent, les résultats sont convertis en structures TCLwebtest et logique TCL afin d'analyser les réponses de l'application Web.

- Les sorties non observables: le système peut réagir à une opération de l'utilisateur en effectuant une action qui est non observables du point de cet utilisateur de vue (et par conséquent de l'appareil d'essai). Par exemple, on peut considérer *Ajout / modification / suppression* de l'information dans une base de données spécifique ou l'envoi d'un email de notification à un utilisateur spécifique. Cette affaire n'a pas été abordée dans ce travail.

0.2.5 Exécution des cas de tests

L'exécution des cas de test est effectué en utilisant un outil de test dédié proposé par la communauté OpenACS [Ope09]. Cet outil est appelé outil ACS-automatisé-Test, il permet l'exécution de scénarios de test instancié interagir avec l'application sur le Web à l'essai, et affiche aussi le verdict de chaque cas de test (illustré dans la figure 5 . L'outil automatisé ACS-Test-est, en soi, une application Web, mais nous ferons référence à elle comme *le testeur* pour éviter les confusions entre ce système et l'application testée Web.

Table 5: Instantiated Test Case

Signal and Parameters	type	Instantiated
?login{user,userp}	input	TCLwebtest::form find ~a "register" TCLwebtest::field find "user" TCLwebtest::field fill "gerardo.morales@montimage.com" TCLwebtest::field find ~n "password" TCLwebtest::field fill "m!lat0" TCLwebtest::form submit

!htmchunk{text_to_find_0}	output	if {[catch {tclwebtest::assert text "Created!"} errmsg]} {
		aa_error "The text: Welcome was not found in the body of the page"
		set verdict_msg "Text not found: Welcome"
		set response 0
		} else {
		aa_log "The text: Welcome was found"
		}

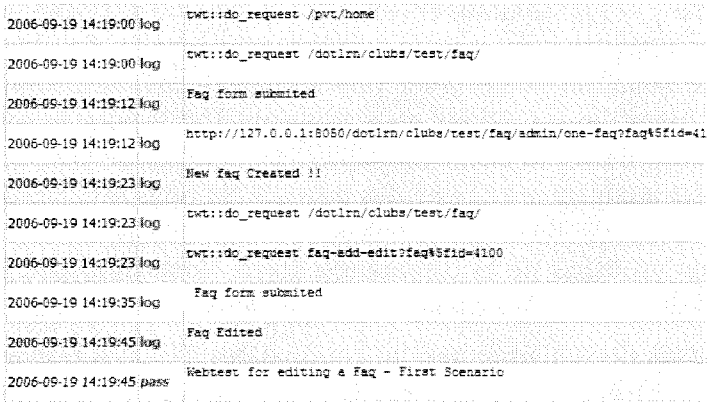


Figure 5: Capture d'écran du test automatisé ACS.

0.3 Test Pasif

La majorité des applications Web modernes utilisent des services qui sont fournis par des applications externes. Cela permet la réutilisation des fonctions existantes et la division de "l'expertise" des applications. A titre d'exemple nous pouvons prendre l'étude de cas, le *Mission Handler*. La communication avec les compagnies aériennes à la recherche et de réservation de vols n'est pas effectuée par le Mission Handler elle-même, cela se fait par une application externe qui est détenue par une agence de Voyage. Merci à ce "diviser pour conquérir" approche, les développeurs peuvent se concentrer sur un seul problème à résoudre: "Comment gérer une mission dans l'entreprise" et l'utilisation de services externes pour résoudre des questions secondaires comme l'obtention d'informations des vols et réservation de billets.

En général, les services (comme celle qui apporte à l'organisme Voyage) peuvent être utilisé par plus d'une application permettant la réutilisation de code. Il est également important de souligner que les développeurs de nouvelles applications Web peuvent utiliser le savoir-faire des services extérieurs et de réaliser pour construire des systèmes complexes en un temps réduit. Toutefois, lors de l'essai d'une application Web en utilisant une

approche boîte noire, les données que nous avons besoin de recueillir afin de mettre un verdict peut être enrichie si l'on observe et de tester plusieurs interfaces (et pas seulement l'interface utilisateur). Cela est possible si l'on observe également l'interface de l'application Web sert à communiquer avec d'autres systèmes via des services Web. En d'autres termes, en testant aussi les communications de l'application Web avec des applications externes (appelés services) tout en testant l'interface utilisateur, nous pouvons obtenir un processus de test amélioré de l'IUT.

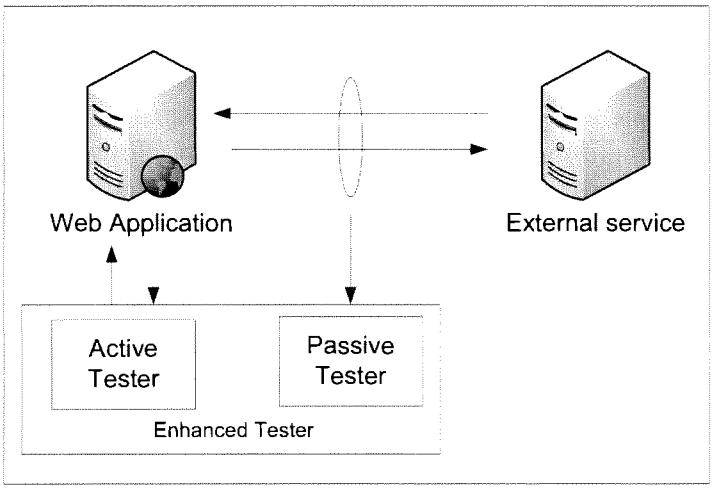


Figure 6: Amélioration de l'essai de test combinant active et passive

Un service Web est définie par le World Wide Web Consortium W3C que "*un système logiciel conçu pour permettre l'interopérabilité interaction machine-to-machine sur un réseau. Il possède une interface décrite dans un format par machine* (en particulier WSDL). *D'autres systèmes d'interagir avec le service Web de la manière prescrite par sa description en utilisant le protocole SOAP des messages, généralement transmis via le protocole HTTP avec une sérialisation XML en conjonction avec d'autres normes liées au Web.*"

Étant donné l'existence d'un forte croisasse en la taille et complexité des services Web, la nécessité de tester leur fiabilité et le niveau de sécurité sont de plus en plus crucial. La technologie des services Web, qui est basé sur le SOA, est difficile à tester car il repose sur des systèmes distribués qui compliquent le comportement d'exécution.

Dans ce contexte distribué, tester en utilisant méthodologies de test actif peut être assez difficile à réaliser car il faut contrôler les systèmes distantes déployés dans machines externes au l'IUT. Le test actif repose généralement sur la comparaison entre le comportement d'une implémentation et ses caractéristiques formelles (qui ne sont pas forcément disponibles) en vérifiant si elles sont équivalentes. Séquences de test sont communément automatique ou semi-automatiquement généré à partir des modèles formels qui représentent des critères de test, les hypothèses et les objectifs de test. Ces séquences (avec un format exécutable) sont effectués par l'établissement de points de contrôle et d'observation (BCP, les interfaces

d'exécution) définis par les testeurs et peut perturber le comportement naturel des services Web qui sont déjà déployés.

Pour éviter ce problème, il ya un intérêt dans l'application de l'approche passive tests pour vérifier la conformité des services Web. Le test passif consiste à observer les échanges de messages (entrées et sorties) d'une implémentation sous test pendant quelque temps et ensuite d'analyser les traces de messages pour détecter les erreurs ou défauts [LNS+97]. Le terme *pasif* signifie que les tests ne perturbent pas le fonctionnement naturel de l'IUT. Le dossier des paquets observé est appelé une trace. Ce sera comparé à des biens provenant de la norme ou proposées par les experts de service Web.

Les techniques de test passif sont appliquées, en particulier, parce qu'ils sont non-intrusive considérant que les techniques des tests actifs requièrent la mise en place des importants architectures de test où les testeurs doivent être en mesure de contrôler l'implémentation à certains points spécifiques. Ce n'est pas toujours possible, surtout quand il n'y a pas un accès direct à des partenaires à distance de services Web. Nous avons donc défini et appliqué une approche fondée sur les invariantes pour valider les implémentations de services Web.

Dans le travail que nous présentons ici, le concept d'invariants déjà définie dans [BCNZ05] est étendu pour prendre des données et des contraintes de temps en considération. Ensuite, nous définissons une nouvelle méthodologie et des nouveaux algorithmes pour le test passive basée sur ces invariants étendu. Les principales contributions de ce partie sont les suivantes:

- Les invariantes étendues qui permettent de spécifier formellement les propriétés fonctionnelles que les services Web doivent respecter sont définies. Ces propriétés sont liées à des paquets échangés entre les systèmes et peut traiter de la partie données de ces paquets et / ou avec des contraintes de temps. La syntaxe et la sémantique de ces invariantes étendues sont présentées.

- Nous présentons notre approche passive des tests de conformité pour les services Web par rapport à leurs exigences fonctionnelles. Cette approche est particulièrement bien adaptée quand il est difficile de contrôler à distance les partenaires, requis par des tests actifs, et permet de ne pas perturber le fonctionnement naturel des services Web qui sont déjà déployés.

- Nous présentons l'analyse l'expérience et les résultats obtenus de l'application de notre outil de test passif appelé TIPS sur un service de reservations de voyages, le Travel Reservation Service. Cet outil effectue une analyse dynamique des traces d'exécution, la vérification des invariants formels étendu. À notre connaissance, aucun autre travail ne s'attaque à la fois les invariants formels qui comprennent des données et des contraintes de temps, et le test des traces d'exécution en fonction de ces invariants, faisant de cette nouvelle approche et innovantes.

0.3.1 Aperçu de la méthodologie

Les tests de conformité utilisant l'approche passive vise à vérifier la conformité d'une implémentation du service Web à travers un ensemble de propriétés (invariants) et suivi des traces (extrait de la fonction en cours d'exécution Web à travers des sondes appelées aussi des points d'observation (PO)). La procédure de test de conformité passive suit ces quatre étapes:

- Étape 1: formulation Propriétés. Les propriétés pertinentes de service Web à tester sont prévues par les normes ou par des experts des services Web. Ces propriétés d'exprimer les principales exigences relatives à la communication entre les partenaires Web dans le cadre de services Web.

- Étape 2: Propriétés comme des invariants. Propriétés être formulée au moyen d'invariants qui expriment des propriétés locales, c'est à dire liés à une entité locale. En outre, les propriétés pourraient être formellement vérifiées sur la spécification du système formel (comme un automate temporisé) si elle est disponible, en s'assurant qu'ils sont corrects vis-à-vis des exigences.

- Étape 3: Extraction des traces d'exécution. Afin d'obtenir de telles traces, PO différents sont mis en place au moyen d'un analyseur de réseau installé sur un des serveurs. Les traces capturées sont stockées dans un format XML. Le bon de commande doit être réglé dans le réseau utilisé pour échanger des messages entre les systèmes, par exemple, si notre application communique Web avec 3 différents services extérieurs, le PO doit être dans le carte d'interface réseau, où l'application Web est en cours d'exécution.

- Étape 4: Test des invariants sur les traces. Les traces sont traitées de manière à obtenir des informations concernant des événements particuliers ainsi que des données pertinentes (par exemple la source et l'adresse de destination, l'origine des données pour initialiser une variable, etc.) Au cours de ce processus, le test des propriétés attendu est effectuée et un verdict est émis (Pass, Fail ou incertaine). Un verdict non conclusif peuvent être obtenues si la trace ne contient pas suffisamment d'informations pour permettre à un Pass ou Fail verdict. Pour réduire ce type de jugements, la trace doit comprendre l'état d'ouverture, ce qui est possible si le sniffer est lancé avant que l'échange de paquets entre les systèmes en cours de test.

0.3.2 Outil de test pasif

Sur la base de la méthodologie de notre travail, nous avons développé le conseils sur les outils [COML08] en collaboration avec Montimage [4] et inspiré d'un premier proto-

[4]Montimage est une PME française travaillant sur le domaine de la vérification formelle. http://www.montimage.com

type développé par IT Sudparis [5]. Les conseils sur les outils vise à tester un système passif de communication déployés dans le cadre de test pour vérifier si elle respecte un ensemble de propriétés. Dans le cas de conseils, d'invariants de décrire l'ordre correct des messages échangés entre les entités du système avec les conditions sur les données communiquées et le temps. test passif consiste à observer d'entrée et de sortie des événements de l'implémentation du système en run-time et de détecter les dysfonctionnements éventuels ou d'erreurs.

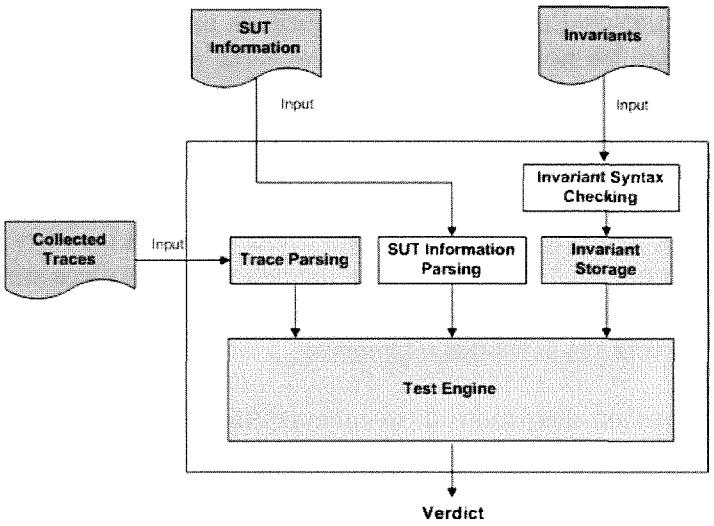


Figure 7: Architecture de TIPS pour le contrôle de service Web

- Information sur le service Web à l'essai qui est observé. Cette information représente les données d'intérêt (par exemple pour les noms de paquets du protocole sur le terrain) qui sont pertinentes pour l'analyse automatisée des traces capturées.

- Les invariants définies au format XML. Par des invariants, nous entendons ici la combinaison de conditions qui doivent être respectés par le système. Le non-respect d'un invariant peut entraîner une erreur dans les services Web.

- Et les traces de la communication en format XML (capturés en utilisant Wireshark[6]).

Pour utiliser l'outil TIPS, la première étape consiste à définir les invariants et les données d'intérêt. Cela peut être fait par un expert du service Web à l'essai qui comprend en détail les services étudiés. Les invariants sont ensuite vérifiées par rapport à leur schéma XML.

[5]Institut TELECOM est un groupe d'écoles d'ingénieurs françaises dans le domaine des télécommunications. http://www.institut-telecom.fr/

[6]http://www.wireshark.org/

La prochaine étape consiste à capturer les traces de communication en utilisant un sniffer (nous avons utilisé Wireshark) et de les analyser en utilisant l'outil TIPS.

Dans le cas de services Web, ce que nous analysons sont des paquets du protocole SOAP. Une requête SOAP est un basé sur XML Remote Procedure Call (RPC) a envoyé en utilisant le protocole de transport HTTP. La charge utile du paquet SOAP est un document XML qui spécifie l'appel en cours et les paramètres sont passés.

Dans le corps d'un message SOAP, espaces de noms XML sont utilisés pour qualifier des éléments et noms d'attributs dans les parties du document. Les noms d'éléments peuvent être globales (référencé dans le message SOAP) ou local. Les noms d'éléments locaux sont fournis par les espaces de noms et sont utilisés dans le cadre notamment du message où ils se trouvent.

0.4 Conclusion

Aujourd'hui, différentes organisations profitent des avantages de systèmes basées sur le Web. Par exemple, les avantages pour des applications Web sont bien connus et exploitées depuis les années 90. Ils aident à surmonter les barrières géographiques et d'acquérir d'autres avantages tels que la flexibilité dans l'utilisation des outils de collaboration synchrone et asynchrone entre les utilisateurs. Plus récemment, un nouveau type de systèmes Web (appelé les services Web) qui s'appuient sur l'architecture orientée services est utilisé pour faciliter l'interopérabilité et la collaboration des systèmes hétérogènes. En outre les systèmes basés sur le Web sont de plus en plus complexes et critiques, ce qui pousse l'industrie et la recherche vers la recherche de solutions pour assurer le bon fonctionnement et la stabilité de ce type de systèmes.

Un processus de bout en bout pour tester les applications Web doit être composé de plusieurs étapes qui doivent être cohérentes pour être en mesure de collaborer et d'obtenir un processus de test complet. Ce processus doit partir la modélisation des systèmes Web, la génération automatique de tests jusqu'à l'exécution de scénarios de test.

Ce travail de thèse a proposé des méthodes pour réaliser toutes les étapes nécessaires pour achever le processus de tests de bout en bout. Nous avons présenté un processus pour modéliser des applications Web de leur point de vue fonctionnel. Notre approche consiste à construire modèle IF qui décrit le système fonctionnel pouvant inclure des restrictions temporelles. Ensuite, nous avons présenté une approche pour gérer les cas de test à partir de ce modèle en utilisant TestGen-IF [COML08], un outil développé dans notre laboratoire. Puis, la méthodologie pour la concrétisation et la traduction des cas de test en code exécutable est présentée. Cette étape présente une méthodologie complète et l'outil GeneraTCL utilisé pour transformer les cas de tests abstraits en cas de test concrets qui peuvent être exécutés sur une application Web réelle en utilisant des communications HTTP. Enfin, nous avons appliqué les cas de test générés à une application Web industrielle.

Après avoir présenté la manière d'aborder le test des applications Web via son interface d'utilisateur en utilisant une approche de test actif, nous avons proposé une approche passive test pour inspecter les communications entre les systèmes basées sur le Web. Nous

avons également fourni une nouvelle approche pour tester la collaboration entre les systèmes de sites Web différents, et a introduit l'outil de TIPS.

Enfin, les méthodologies et outils ont été appliqués à une étude de cas du monde réel (Web services de réservation de vols) pour démontrer l'efficacité de cette nouvelle approche.

Part I

Introduction

0.5 Context

Web based systems became the most important foundation for synchronous and asynchronous collaboration between users. The concepts of e-banking, e-government and e-learning have become important channels of communication and Web applications have evolved in complexity to become powerful tools to handle online collaboration. In addition, in the last few years major progress has been achieved in Web technologies, the concept of one big stand-alone application is changing towards the concept of multiple specialized interconnected applications (called Web services). This concept is called Service Oriented Architecture (SOA). It allows the reuse of existent applications, hence, the possibility of building more effective Web applications in less time.

Along with this evolution, the sophistication and complexity of Web based platforms are growing. Therefore, it is becoming more and more difficult and expensive to assure the good deployment of systems and their stability. This situation becomes even worse because of the lack of standardization in Web clients (browsers).

Some of the institutions that rely on Web technologies handle mission critical tasks or manage sensitive data (e.g. Banks). However the functional misbehavior of these systems can be used to perform intrusions, attacks or gain access to sensitive data.

Testing has a main role when the Web based systems are mission critical to assert the system usability, stability and the respect of security functions. However, software testing consumes between 30 and 60 percent of the overall development [UL06]. Moreover, because of the increasing use of SOA in modern systems, there is a growing need in testing methods that can be applied on systems where it is not possible to have control over remote deployed Web systems.

There is an increasing need to reduce the time and money spent to test modern Web based systems as they become more complex to test. Many efforts have been done in the research and commercial fields to automatize the execution of test cases to this kind of systems. And it exist a big interest in using Model Based testing to push even forward the automation level and achieve obtaining reliable methods and tools allowing automatize also the test generation.

0.6 Motivations

Nowadays, different organizations are adopting Web applications to get connected with remote sites, clients or users. This has the advantage of gaining flexibility in using tools for synchronous and asynchronous collaboration between users [CMPV05a]. The dependence on Web services and applications of millions of users in a wide variety of business domains makes the assurance of their reliability and availability critical.

Some works have been done for testing Web application, existing methods and tools like the ones presented in [HK06], [WX09] allow the automation of test case execution. Recorded test cases are manually generated and re-executed (played) further on in order to speed up regression testing. It exists however just few efforts that try to automatize the generation of active test cases for Web applications. These tools use for example a combination of semi formal models like UML and a constrained language like OCL instead of one formal dedicated model.

However, the only way to avoid human intervention in the test generation is through Model Based Testing (MBT). MBT requires a machine readable and unambiguous model of the system to be tested i.e. a formal model of the implementation under test (IUT).

Unfortunately, nowadays the test case generation for Web based systems is generally manual. This is due to the lack of well suited tools to formally specify Web based systems; a required step for automatic test case generation. Looking closely into this field, we can notice a gap between the academia and the industry. In one hand, the research community has developed for decades the theory foundations to specify systems using abstract mathematical models and then to generate abstract test cases for these models. On the other hand however, the interest of the industrial community focuses on the execution of the test cases on real world implementations. There is an urge to fill this gap and complete the chain including the required Web systems specificities into the modeling phase in order to automatically generate and instantiate executable test cases that can directly stimulate the system under test.

Moreover, Web applications are changing towards a new generation based on the Service Oriented Architecture; this brings great possibilities to build powerful tools in less time using heterogeneous remote systems. In most of the cases, SOA based applications use third party systems. In this distributed context, formal active testing can be rather difficult to achieve since it needs to control remote deployed Web systems.

Finally, to achieve a complete testing of a modern Web application it is needed to use an active testing approach to test the usability of its user interface, but at the same time there is an interest in applying passive testing approach for the conformance checking of the communication of the Web application with remote Web services.

This PhD work explores methodologies and existent tools to perform different kind of tests over Web based systems. The main motivations are: (i) To propose new methods and tools to fill the gap between the communities of research and industry concerning the testing of Web based systems. (ii) To propose new concepts, methodologies and tools to test Web applications that uses the SOA architecture to collaborate with remote Web services.

0.7 Contributions

The work achieved within this thesis are part of the European @lis E-LANE project [7] that aims to build new Web technologies to support e-learning courses and the national ANR Webmov project [8] that deals with functional and security testing of Web services.

In this work we propose two different approaches to test the functional aspects of Web based systems. The first approach is based on active testing techniques and we propose a framework to automatically generate test sequences to validate the functional aspects and usability of a Web application. The functional behavior of the system is specified using a formal description technique based on Extended Finite State Machines (EFSM). Then, the automatic test generation is performed using dedicated tools to produce test suites in a specialized executable language to interact with real world Web applications via its user interface.

The second approach is based on passive testing techniques. We analyze the collected traces of the communication between the system under test and external Web services in order to deduce verdicts of their conformance with respect to its functional requirements. This approach proposes a new tool, a new methodology and introduces the timed extended invariants, an enhancement of existing concepts actually used in the network passive testing techniques.

These contributions have given rise to several publications in book chapters, international conferences and workshops like [CMM07], [MMC08], [MLMC08], [CLM+08], [MLM09] and [?].

0.8 Plan of the document

This thesis manuscript is organized into three chapters:

1. The first chapter presents the state of art on Model Based Testing (MBT) and some works done in testing of Web based systems.

2. The second chapter introduces our contribution on active testing of Web applications using MBT.

3. Finally, the third chapter presents our contribution in the passive testing of the interoperability of heterogeneous systems using Web services.

In the first chapter we present the basic concepts of Model Based testing. We also present different languages and mathematical structures to build a formal description of a Web based systems. We hereafter present the concepts of verification and validation. Then we provide some details about active testing approach. We define conformance testing, automatic test generation and test cases execution. We also present some concepts about

[7] More information in http://git.ucauca.edu.co/e-lane/index-en.html
[8] More information in http://webmov.lri.fr/

passive testing approaches and mainly the forward and backward checking techniques as well as the invariant based passive testing. Some related works and tools about the testing of Web based systems are also presented.

Chapter number two presents our first contribution. This work aims to obtain the formal specification and the active testing of Web applications through its user interface. Our framework enables to generate in an automatic manner a set of test sequences and then to execute them in order to validate the conformance of the functional description of the system. Its behavior is specified using a formal description technique based on extended finite state machine (EFSM) [BDAR98]. Then, the automatic test generation of abstract test cases is performed using the TestGen-IF tool (developed in our laboratory) and the derivation of executable test cases is accomplished using our GeneraTCL tool. Finally, we present the *Travel Reservation Service* a real world case study to demonstrate the reliability of our approach.

After covering the methodologies to perform active testing on Web based systems, we extend this work in chapter three taking into account the new technologies based on the SOA architecture. It is presented a new methodology and a new tool to achieve the testing of distributed and remote heterogeneous systems by a passive testing approach. This chapter presents an enhancement of mathematical structures that are being used to perform passive testing and introduces the timed extended invariants.

Finally, chapter four concludes the thesis manuscript by summarizing our main propositions and contributions in the field of formal testing for Web based systems and present some perspectives for our work.

Part II

State of Art

Chapter 1

State of the Art

Contents

"The proper route to an
understanding of the world
is an examination of our errors about it."
Errol Morris

1.1 Introduction

Nowadays, different organizations are increasing the use of Web applications to overcome geographical barriers and gain other advantages such as flexibility in exploiting tools for synchronous and asynchronous collaboration between users [CMPV05b]. In this same context, in the last few years major progress has been achieved in the design of Web based platforms. It is also needed to take into account that modern Web applications are becoming increasingly complex and mission critical.

Along with this evolution, the sophistication and complexity of these platforms are growing. Therefore, it is not enough to perform superficial testing of usability by monitoring human-machine interaction through the user interface to cover the system functionality. To test in an exhaustive way a complex system such as an e-learning or a banking platform, it is necessary to adopt well known approaches of testing, for example, those used in the field of telecommunications.

Testing has to assert the system usability, among the guarantees that testing should give, is that a user cannot gain permissions over the system if he is not supposed to have them, this assures the basic security rules. Another guarantee should be that the system does what it is supposed to do, to assure its usability.

In general, software testing consumes between 30 and 60 percent of the overall development [UL06]. The need to reduce time and money investment in Web testing is rapidly pushing the development of automated testing tools as *Selenium* or *Squash for Web*. Model based testing pushes the level of automation even further by automating not only the test execution, but also the test cases generation. Model based testing relies on *formal methods* to build the model of the implementation that will be tested (the implementation under testing or IUT).

Model based testing may be used to generate test cases that covers a large scope of test types (see Figure 1.1), taking into account the black box testing of the functional aspects of a system based on the conformance of the implementation with its model.

The formal methods are based on a rigorous mathematical concepts used to describe and analyze the behavior of a system. The main advantage of using formal languages is the aptitude to automatically verify the correctness of a system. A model built based on

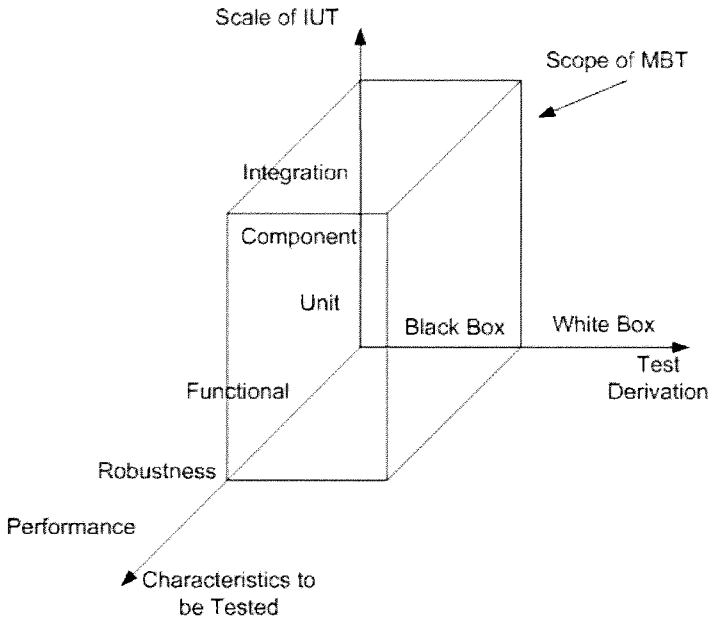


Figure 1.1: Model Based Testing Scope

a formal method is known as a formal model, which is a machine readable model (with no ambiguities).

In general terms it is commonly admitted to classify the different testing techniques into two main testing families: the active testing and the passive testing. Each of these families encompasses various approaches, each approach contains different techniques. The basic concepts of these two approaches are introduced in this chapter and the methodologies based in these testing approaches will be presented in further chapters of this work.

1.2 Modeling Systems

By definition, a model is an abstraction of something real. In our case, by modeling a system, we abstract away irrelevant or potentially confusing details that are useless to describe the system from the user's point of view. The main goal of modeling is to obtain a simple specification of the real system from its behavioral point of view, so it allows the design and viability of a system to be understood, evaluated, and criticized quicker.

As formal models are precise as a programming language, they can be interpreted, executed and transformed by machines. Therefore these operational models can be used as input to existing validation tools, such as random simulators, test generation engines or model checkers.

1.2.1 Notations for Formal Models

There are a large number of formal notations used to model the functional behavior of a system, the work described in [vL00] classifies these notations in five different families: (1) State Based notations that use *pre* and *post* conditions; (2) Operational notations, that describe the system as a set of processes running in parallel as CCS or LOTOS; (3) History based notations, that use the allowable traces of the system to build its model, e.g. Message sequence charts; (4) Functional notations that describe the system as a collection of mathematical functions and (5) Transition based systems, that focus in the transition between states of the system, e.g. Finite State Machines. To describe Web based systems, we need to deal with timed constraints, e.g. timeouts. Fortunately, among the different model notations, some of them have been extended to handle timed constraints.

During this chapter, different modeling notations are introduced. The transition based systems are presented in more details since they correspond to those that have been most investigated during this work for modeling and testing Web based systems.

Petri Nets

Petri Nets belong to the operational notation family of the [vL00] classification. Petri net is a mathematical modeling language for describing discrete distributed systems. A Petri net is a directed bipartite graph composed of transitions (i.e. discrete events that may occur), places (i.e. conditions), and directed arcs (that describe which places are pre- and/or post conditions for which transitions). Petri nets were proposed in 1962 by Carl Adam Petri [Pet62].

Process Algebras

The Process Algebras [Mil89a] whose the notation is classified as operational, are a set of related approaches based on syntax rules and mathematical operators to formally model concurrent systems. Process Algebras provide a tool for the high-level description of interactions, communications and synchronizations between a collection of independent agents or processes. They also provide algebraic laws that allow process descriptions to be manipulated and analyzed, and permit formal reasoning about equivalences between processes. Leading examples of process calculi include CSP [GR88] (Communicating Sequential Processes), CCS [Yi91] (Calculus of Communicating Systems), ACP (Algebra of Communicating Processes), and LOTOS [ISO89] (Language of Temporal Ordering Specification).

The Calculus of Communicating Systems is an algebraic calculus designed to describe and analyze the behavior of different systems running in parallel. Originally proposed by [Mil89b], it has generated a lot of variants, among which we find the LOTOS of ISO, which is more precisely used in the domain of data communication protocol specifications.

LOTOS has been widely used for the specification of large data communication systems. It is mathematically well-defined and expressive: it allows the description of concurrency, non-determinism, synchronous and asynchronous communications. It supports various levels of abstraction and provides several specification styles. There are tools (e.g.

the EUCALYPTUS toolset [Gar96]) to support specification, verification, code and test generation [CKM93].

Finite State Machine

Classified as transition based models in [vL00], it is usually drawn visually as a graph, the finite state machines or finite state automaton is a model of computation consisting of a set of states, a start state, an input alphabet and a transition function that maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states depending on the transition function.

There are many variants, for instance, machines having actions (outputs) associated with transitions (Mealy machine) or states (Moore machine). Formally the FSM are defined as:

Definition 1 *A Finite State Machine is a 6-tuple* $< S, I, O, \delta, \lambda, s_0 >$ *where:*

- S *is a set of states, where* $s_0 \in S$ *is the initial state;*
- I *corresponds to a finite set of input events;*
- O *corresponds to a finite set of output events;*
- δ: $S \times I \rightarrow S$ *is the state transition function. We can extend* δ *to* δ^*: $S \times I^* \rightarrow S$ *where* I^* *is the set of all finite input sequences including the empty sequence* ϵ.
- λ: $S \times I \rightarrow O$. *We can extend* λ *to* λ^*: $S \times I^* \rightarrow O^*$ *where* I^* *is the set of all finite input sequences including the empty sequence* ϵ *and* O^* *is the set of all finite output sequences including the empty sequence* ϵ.

A further distinction is between a *deterministic* and a *non-deterministic* FSM.

- In a *deterministic FSM*, for each state there is exactly one transition for each possible input.
- In a *non-deterministic*, there can be none or more than one transition from a given state for a given possible input.

Extended Finite State Machines

The Extended Finite State Machine (EFSM) [LY96] is an FSM where the input and output data are explicitly specified as parameters and presents local variables. Their transitions are associated with a predicate and an action. The predicate depends on parameters of

input data and on values of current local variables. Actions are the result of the transition execution, where variable values may be changed.

In a conventional finite state machine, the transition is associated with a set of input Boolean conditions and a set of output Boolean functions. Nevertheless, in an EFSM model, the transition can be expressed by an "if statement" consisting of a set of trigger conditions. If trigger conditions are all satisfied, the transition is fired, bringing the machine from the current state to the next state and performing the specified data operations.

Using this formal model, we can represent not only the control portion of a system but also properly model the data portion, the variables associated as well as the constraints which affect them.

Definition 2 *An Extended Finite State Machine M is a 6-tuple $M = < S, s_0, I, O, \vec{x}, Tr >$ where:*

- *S is a finite set of states;*
- *s_0 is the initial state;*
- *I is a finite set of input symbols (possibly with parameters);*
- *O is a finite set of output symbols (possibly with parameters);*
- *$\vec{x}$ is a vector denoting a finite set of variables;*
- *and Tr is a finite set of transitions.*

A transition tr is a 6-tuple $tr =< s_i, s_f, i, o, P, A >$ denoted also $(i/o, P, A)$ where:

- *s_i and s_f are the initial and final state of the transition;*
- *i and o are the input and the output;*
- *P is a predicate (a Boolean expression) on variables $\vec{x}$;*
- *and A is an ordered set (sequence) of actions.*

Figure 1.2 illustrates a simple EFSM of three states S=$\{S_0, S_1, S_2\}$, an alphabet of two inputs I =$\{A, B\}$, three outputs O=$\{X, Y, Z\}$, two tasks A=$\{T, T', T''\}$ and two predicates P=$\{P, P'\}$. A transition is fired when an input I occurs and the predicate P is respected. For example, starting from state S_0, when the input A occurs, the predicate P is checked. If the predicate's condition holds, the machine performs the task T, triggers the output X and the system loop returning to state S_O. If P is not satisfied ($\overline{P}$), the output Z is triggered and the task T'' is performed and passes to the state S_2.

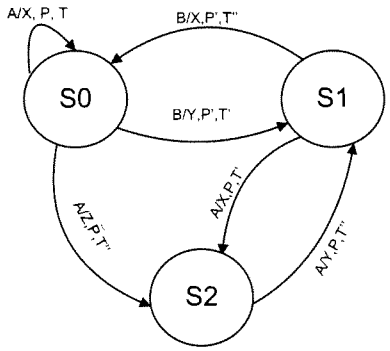


Figure 1.2: Example of a simple EFSM.

Timed Extended Finite State Machine

A communicating extended timed automata [BGO+04] called also Timed and Extended Finite State Machine (TEFSM) is an FSM that has been extended to manage timed issues and handle variables values in its transitions.

Definition 3 *A TEFSM M is a 7-tuple $M = < S, s_0, I, O, \vec{x}, \vec{c}, Tr >$ where:*

- *S is a finite set of states;*
- *s_0 is the initial state;*
- *I is a finite set of input symbols (messages possibly with parameters);*
- *O is a finite set of output symbols (messages possibly with parameters);*
- *$\vec{x}$ is a vector denoting a finite set of variables;*
- *$\vec{c}$ is a vector denoting a finite set of clocks;*
- *and Tr is a finite set of transitions.*

A transition tr is a 4-tuple $tr =< s_i, s_f, G, Act >$ where:

- *s_i and s_f are respectively the initial and final state of the transition;*
- *G is the transition guard which is composed of predicates (Boolean expression) on variables $\vec{x}$ and clocks $\vec{c}$;*
- *and Act is an ordered set (sequence) of atomic actions including inputs, outputs, variable assignments, clock setting, process creation and destruction.*

The actions associated with the transitions occur simultaneously and take no time to be executed. In the other hand, the time progress takes place in some states before executing the selected transitions [BGO+04, BGMO04, BFG+99].
The Figure 1.3 illustrates the graphical representation of a TEFSM.

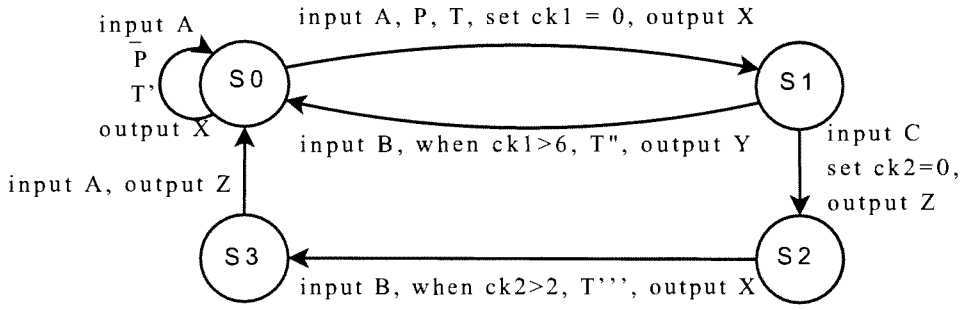


Figure 1.3: Example of a Simple TEFSM with Four States.

1.2.2 Specification Languages

State-transition system based languages are the group of formal languages that has been the most investigated in the last years for communicating systems. Basically, researches have been done in order to apply finite state machines and its extensions in industrial applications. From *Synchronous* languages, State charts [Har87] is the mostly used, as SDL [ITU92] (Specification and Description Language) is for the group of *Asynchronous* languages. State charts has been used mainly in the automotive industry, while SDL has been largely used in the protocol specification of communication systems. We can also cite the effort that has been made in applying UML [Gro00] in the real-time domain [TM02]. Due to the variety of tools and supports, it has been easily applied in industrial applications.

We present in the following a brief description of two asynchronous languages: SDL and IF (for Intermediate Format) [BGO+04]. This choice is due to the fact that we will rely on these languages during the remaining of the thesis.

Specification and Description Language SDL

The Specification and Description Language SDL standardized by ITU-T [ITU92] is widely used to specify communicating systems and protocols. This language has evolved according to user needs. It provides different concepts needed by designers to specify systems more and more complex. SDL is based on the semantic model of Extended Finite State Machine (EFSM). Its goal is to specify the behavior of a system from the representation of its functional aspects. The description of the functional aspects is provided at different abstraction levels. The most abstract is the one describing the system, while the lowest is the specification of abstract machines composed by signals, channels, tasks, etc. Two kinds of properties may describe these functional aspects: the architectural and behavioral

properties. The first one denotes the architecture of the system that is the connection and organization of elements like blocks, processes, etc. The second one describes the behaviors of the entities after an interaction with the environment. These reactions are described by tasks (transitions between states) and are based on the EFSMs.

Verification on local variable values imposes a condition (predicate) on moving to the next state. The actions associated with a transition include: verification on local variable (that can impose conditions or predicates to move to the next state), the execution of tasks (assignment or informal text), procedure calls, dynamic creation of processes in order to include new processes instances into a system (SDL contains the concepts of 'type' and 'instance of type'), arming and disarming timers, etc.

IF Language

The Intermediate Format (IF) language can be considered as a common representation model for other existing languages. It was originally developed to sit between languages as SDL, Promela [GMP04] or Lotos [ISO89]. It has been extended to deal with UML notation as well [CCD+]. IF is based on communicating timed automata, and is used to describe and validate asynchronous systems.

In IF, a system is a set of processes communicating asynchronously through a set of buffers. Each process is a TEFSM that models the behavior of a given component. A process can send and receive messages to and from any buffer.

The semantic of time is similar to the one of communicating extended timed automata. That is:

- A time behavior of a system can be controlled through clocks.
- The time progresses in some state before selecting and executing some transitions.
- Transitions take zero time to be executed.

In order to control the time progress or the waiting time in states, IF implements the notion of urgency in the transitions. A transition may have priority over others, or may be delayed. In this context, a transition may be described as following:

```
deadline {eager, delayable, lazy};
provided <expression>;
when <constraint>;
input <signal (expression)>;
{statement};
{action};
if <expression> then {statement} endif;
while <expression> do {statement} endwhile;
nextstate <state_id>;
stop;
```

In the sample above, "Eager", "Delayable" and "Lazy" concerns the priority of the transitions related to the progress of time, where:

- **Eager**: the transition has priority over the time. The time can not evolve except if the transition is fired. In other words, these transitions must be executed as soon as they are enabled and waiting is not allowed.
- **Delayable**: the time has priority over the transition. This means that the time may evolve until the time constraint becomes true. When it is enabled, waiting is allowed as long as time progress does not disable it.
- **Lazy**: the transition and the time have the same priority. In this case it does not matter what comes first: the transition may be fired or the time may evolve. These transitions are never urgent. When a lazy transition is enabled, the transition may be executed or the process may wait without any restriction.

Several tools may interact with IF. Some concern the automatic transformation of system specifications into IF Format (as *SDL2IF* or *UML2IF*). Other tools have tackled the system analysis and verification using the IF format such as TReX [ABS01]. Other possibilities are the simulation of the system (IF-2.0[1] and IFx[1]), or even the test generation using TGV tool.

This subsection introduced different formalisms that can be used to model the behavior of a system. The main goal of this work is to use a model of a system to perform validation and verification of the system under testing, terms defined in the next subsection.

1.3 Verification and Validation

The validation of a system consists in guaranteeing that its specification and implementation correspond to its expected behavior. In practice it is divided in two phases: verification and testing [CA97, RN07].

Verification is used to guarantee that systems will work correctly in all circumstances. It is done through the development of automatic (or semi-automatic) techniques in order to detect errors and incoherence in a software system specification. However, the verification process is only possible on specifications that are made based on a certain level of formalism. Nowadays the two main approaches that have been used are Model Checking and Symbolic Execution. During the verification, the main analyzed properties are:

- Safety (absence of deadlock, unspecified reception, blocking cycles, etc). Deadlock takes place when a state of the system, reachable from the initial state cannot trigger a transition anymore.
- Promptness (live lock), a state is known as alive if it can be reached starting from all the states of the global system.

[1] http://www-omega.imag.fr/

- Reachability, i.e., ensuring that all the system states are reachable at least by one path from the initial state.

The testing phase will be exposed in details in the next sections. The type of errors that can be discovered by performing verification and validation of a system are explained in the following section.

1.4 The Different Error Types

The goal of conformance testing is to check that an implementation which represents the system under test performs all what is described in its specification (what is supposed to be correct). In consequence, in order to perform the conformance testing, an implementation and a specification are needed. To minimize wrong interpretations in the specification, we describe it based on a formal language or formal model.

The conformance testing is then similar to a comparison between two elements. When the elements are not corresponding, the implementation is said to be not conform to the specification. The possible kinds of errors are classified following the three types below:

- output error: the output of a transition in the implementation differs from the corresponding one in the specification,
- transfer error: the ending state of a transition in the implementation differs from the corresponding one in the specification,
- mix error: the output and ending state of a transition in the implementation differ from the corresponding ones in the specification.

1.5 Active Testing Approach

Active Testing consists in applying a set of test scenarios on a system under test (SUT) to check its conformance according to its specified requirements. It is one of the most important phases of the system building, the most expensive and time-consuming. Systems are error-prone and the main goal of testing is identifying the errors in a system implementation.

Ideally, the testing phase cannot be concluded until the whole system is analyzed, all the errors identified and corrected. In practical terms, the real question is: is it possible to test a whole system?

- A first drawback consists in how to generate all possible test cases. For a human being, and dealing with a complex system, it is impossible to think through all possible situations, single and combined, to generate a complete test sequence, even worst, executing them.

- A second limitation is the time: it is not straight forward to determine how long the process of testing a system will take. The experience and studies have shown that testing may consume up to 50% of the time and the project resources [TB99] in a system development. When errors are found, it is expected that the implementation is changed, and the tests redone. Not only the tests that had previously detected faults, but ideally, all sets of tests should be re-executed in order to be sure that new errors were not introduced. This scenario may lead a system development to take longer time that previously expected.

In industrial applications, *time-to-market* is one of the main reasons for reducing the time of the system development. Even when timing requirements or other safety aspects are not taken into account, the selection of which tests should be applied is not a direct process. These contexts bring other practical questions, such as:

- Is it possible to reduce the time spent in testing?
- How to select the test cases that must be generated and executed?

Automation of testing activities seems to get around these problems. Automation may help in making the testing process faster, in making it less susceptible to errors and more reproducible [TB99].

1.5.1 Conformance Testing

Conformance testing aims at verifying whether a system behavior (the components behaviors, the delays and the messages exchange) corresponds to its specification. Different strategies may be used to realize this kind of test, such as white-box, black-box and gray-box.

- Structural and white-box testing are synonyms. They correspond to test the code of the implementation, observing the exchange of messages but also considering and analyzing the internal actions, data structures, loops, etc. There are tools specially designed for testing generation and/or execution of this phase.
- Functional or black-box testing correspond to observe the inputs and outputs exchanged among the components under test, without considering the internal actions developed inside these elements. The verdict is given based on the observed events.
- Gray-box testing corresponds to an intermediate approach between the white and black-box. We consider that some internal actions and features are known. One of the strategies is to observe the events (black-box) to detect errors and then to analyze the code (white-box) to identify its cause.

In a common active testing approach there are two main steps:

- The (automatic) generation of a set of test cases based on the system specification.

- The processing of these test cases on the implementation. The conformance verdict is deduced after the analysis of the system reaction to these stimuli (tests).

The strength of the active testing is in ability to focus on particular aspects of the implementation. Indeed, the test cases can be limited to a specific type of errors, or to an important state of the system. On the other hand, the test case choice and production can turn out to be complicated. Besides, testing actively a system can disturb its normal functioning. For this reason, active testing is usually performed on a system implementation before its commercialization.

The Figure 1.4 describes the steps for the active testing general methodology. Based on a formal specification of the system under test, we generate automatically a set of test cases that we provide to the active tester unit. This tester will apply the test scenarios on the system implementation to stimulate it and collect then its reaction (output messages). These messages are analyzed: the tester checks whether they correspond to the desired outputs described in the specification. If it is the case, the verdict would be 'pass', otherwise, it is 'fail'. In some cases (for instance, when the system is non-deterministic), the verdict can be 'inconclusive'.

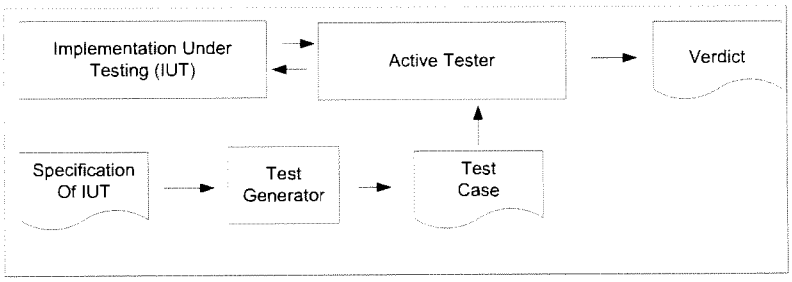


Figure 1.4: Active Testing Methodology.

By using formal specifications, the automatic generation of conformance test cases is possible. However, it is necesary to have some hypothesis concerning the formal model as datailed in the following.

1.5.2 FSM Based Testing: Hypothesis

In the conformance testing, we consider that we have a complete information about the specification machine `Spec`. In addition, we have an implementation machine `Imp` that we can only observe its I/O (for Input/Output) behavior. We would like to know whether the implementation machine `Imp` is conform (i.e., is equivalent) to the specification machine `Spec`. That is, we would like to carry out a test in order to determine whether the machine `Imp` is a correct implementation of the machine `Spec` by applying a sequence of input symbols to the machine `Imp` and observing its outputs.

In order to get around this problem, some basic assumptions are usually made in the literature:

- √ Specification machine `Spec` is strongly connected.
- √ The machine `Imp` is reduced.
- √ The implementation machine `Imp` does not change during the tests execution and has the same input alphabet as the machine `Spec`.
- √ The machine `Imp` has no more states than the machine `Spec`.
- √ The implementation machine `Imp` has reliable reset.

If the implementation is tested as a black-box, the strongest conformance relation that can be tested is trace-equivalence.

Definition 4 *Two FSMs are trace-equivalents if they cannot be told apart by any input sequence. That is, both the specification and the implementation will generate the same outputs (or trace) for all specified input sequences.*

In order to check whether two machines are trace-equivalent, it is sufficient to show that:

1. There is a set of implementation states that are isomorphic to the states of the specification. In order to check for isomorphic states, we need to uniquely characterize each state of a machine.
2. Every transition in the specification has a corresponding isomorphic transition in the implementation.

These hypotheses are the same if we deal with an EFSM or TEFSM based specification. The approach used in black-box test derivation methods is to check trace-equivalence between the specification and the implementation by verifying that the implementation has isomorphic states (with regard to the specification) and then performing transition testing. In order to check for isomorphic states, we need to uniquely characterize each state of a machine (this can be done using tequiniques as the Distinguish Sequense DS [UZ93] or the Unique Input/Output UIO [DHHG06]) and then make the relation between the implementation machine states with the ones of the specification machine.

1.5.3 Automated Test Generation

In practice, test generation is in general performed by humans after studying the specification [Tre01]. In formal contexts, a model represents the system specification and it allows an automatic test case generation. Model-based testing is a technique where a model is considered as an input to generate test suites based on dedicated algorithms. These algorithms allow the process of generating and selecting test cases.

In a test generation process, there are two important concepts that must be considered, that are *soundness* and *completeness*. We define these concepts based on [BFS05]:

- Soundness: Generated test cases should never cause a fail verdict when executed on a correct implementation.
- Completeness: For each possible implementation that does not conform to the specification model, it is possible to generate a test case that causes a fail verdict with respect to that system under test (SUT).

A number of methods have been proposed for test case derivation based on FSMs and the verification of the arrival state. We mention for example the transition tour method [NT81], the distinguishing sequences [UZ93], or UIO-method [SD88] etc. These methods have been extended to EFSMs and Timed automata [Vie07].

One of the strategies to generate test cases is the test purposes based generation. This strategy has been used to avoid the state explosion problem. It selects parts of the specification for being tested, and ideally, the critical properties of the specification that must be validated. The strategy is to generate scenarios that correspond to the given properties. As advantage, critical parts are tested and we avoid the state explosion problem.

In this context, the test coverage considers a complete test generation if all sets of chosen properties are satisfied in the test suite. The model checking technique has been applied for test case generation using test purposes. The main idea is to use a reachability property to determine if a coverage item can be fulfilled and, next, to construct a property that becomes true when a coverage item is fulfilled [Hes06]. The principle used is the same used to generate a witness trace that represents a collection of test cases that fulfills the requested criterion.

Hereafter, we present a set of test generation and/or execution tools. Some of them were originally conceived for dealing with EFSM-based specifications or more specifically for SDL specifications. Other tools are still prototypes, but we consider relevant and necessary to distinguish them as well. Concerning test generation, most of them are based on the test derivation algorithm based on the ioco theory, proposed in [TB99]. This is the case for TGV, TVEDA and TorX tools. Next, we will only present the tools that we will use later within this thesis.

TVEDA

TVEDA [MRMT95] is a test purpose-based tool, developed by the R&D center of France Telecom. Its main goal is to support automatic conformance testing of protocols. It deals with Estelle or SDL specification languages and the resulting test suite is represented in TTCN3 format [TTC]. The aim of the tool is to find a path from one EFSM-state to another while satisfying given conditions on the variables. The main drawback in a state-based testing is the state explosion when building the complete state graph of the specification. To overcome it, TVEDA implements a reachability analysis.

According to [BFS05], the reachability analysis is done through a simulator/verifier. It is used as a test selection strategy to produce the test suite. First, the EFSM is reduced. All the parts which do not concern reaching the demanded target are excluded, that means those elements which do not affect firing conditions of transitions. Then, the simulator works following three heuristics:

1. A limited exhaustive simulation is done. The limitation is about 30000 explored states.

2. A second exhaustive simulation may be started in order to reach other transitions not found in the step 1. During the exploration, the current path is incremented with the next taken branch.

3. Finally, the simulator tries to use an already computed path, which brings the specification to a state which is close to the start state of a missing transition. Another simulation is initiated until the target transition is reached.

TGV

Test Generation with Verification technology (TGV) [JJ05] is a tool for the generation of conformance test suites for protocols. It is the result of a collaboration of the project Pampa (from IRISA institute) with the project Spectre (from INRIA) in the VERIMAG research center. The specification languages accepted by TGV include LOTOS, SDL (using the ObjectGeode tool[2]), UML [CCD+] or IF [BGM02].

TGV takes two Input Output Label Transition System as input: one for the specification and the other formalizing the behavioral part of a test purpose. From both a synchronous product is computed, in which the states are marked as "Accept" and "Refuse" based on information from the test purpose. Briefly, test cases are generated by selecting accepted behaviors, i.e., it performs a selection of traces leading to "Accept" states [BFS05].

A complete test graph, i.e. that contains all test cases corresponding to the test purpose or individual test cases, can be generated. In the complete version: "Pass" verdicts are based on traces that reach "Accept". Traces not leading to an "Accept" state are truncated and an "Inconclusive" verdict is added. Finally, "Fail" verdicts are generated from observations not explicitly presented in the complete test graph [BFS05]. The result IOLTS, containing the test cases, is translated into TTCN3 format [JGW03].

TVEDA and TGV have been partially incorporated into the commercial tool TestComposer, which is part of the ObjectGeode from Telelogic.

TestGen-IF

TestGen-IF [COML08] is a test generation tool developed by TSP that uses as the main input an IF model of the IUT. The major advantage of this tool is that it avoids the

[2]http://www.telelogic.com/products/additional/objectgeode/index.cfm

exhaustive generation of the reachability graph of a specified system by building only partial graphs, allowing solving combinative explosion constraints.

TestGen-IF is based on one of the flagship algorithms of TestGen-SDL, named Hit-or-Jump [CDRZ99]. In particular it allows the automatic generation of test sequences. This is very significant mainly for the features integration in an operating platform. In addition to the generation of test sequences from test objectives, TestGen-IF also offers many other functionalities such as:

- The transformation of a possibly partial reachability graph in an Aldebaran [FGK+96] automaton.
- The parsing and performing the lexical analysis in the purpose of debugging all the output files of the IF simulator.

1.5.4 Test Cases Execution

The test generation tool provides in general a set of test cases described in an abstract formalism. They can be produced in Aldebaran [FGK+96], TTCN3 [TTC], MSC [msc96] or other standard notations facilitating their portability and their automatic execution.

In order to execute the generated test cases on a real application/system/network, they need to be transformed into an executable script or language capable of communicating with the implementation under test. We call this stage the test cases instantiation. Unfortunately, there are some applications (e.g. the *Vocal Service*[3] case study that we studied and presented in [CMM+06]) where the automatic tests execution encounters many difficulties. For instance, it requires the emulation of the input tests, starting from text to speech or recorded sound files and to observe the outputs by using techniques of voice recognition. Hence, many tests must be carried out manually, which is expensive in terms of time and resources.

1.6 Passive Testing Approach

The passive testing consists in observing the input and output events of an implementation in run-time. It does not disturb the natural run-time of a protocol or service, that is why it is called passive testing (sometimes also referred to as monitoring). The record of the event observation is called an event trace. This event trace will be analyzed according to the specification in order to determine the conformance relation between the implementation and the specification. We should keep in mind that when an event trace conforms to the specification, it does not mean that the whole implementation conforms to the specification. In the other hand, if a trace does not conform to the specification, then the implementation neither.

The passive testing has the huge advantage to not influence the system under test. Contrary to active testing, there is no injected messages (no system stimulation). That

[3]The Vocal Service is a France Telecom telephone service

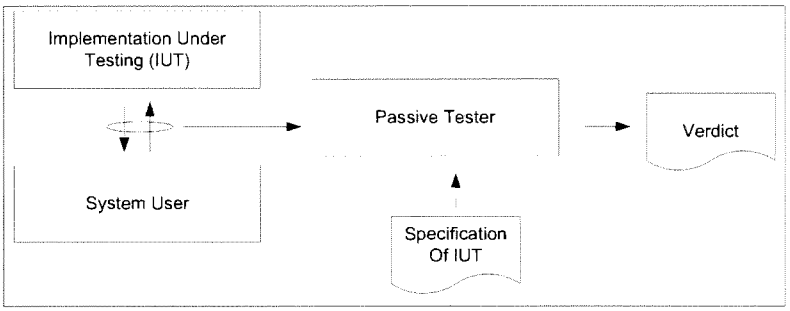


Figure 1.5: Passive Testing Methodology.

is crucial because these messages modify the environment and may trouble or, in a worse scenario, cause the crash of the tested protocol or service. The passive testing approach seems then to be an ideal tool for doing a control directly on the implementation in natural run-time conditions (monitoring). In addition, in this approach the tests can be run during the whole protocol life time in contrast with the active testing test campaigns that must be run on a limited duration. The Figure 1.5 describe the passive testing methodology. A trace analysis can produce a "pass" verdict if the trace is conforming to the system specification (or properties) and a "fail" verdict otherwise. The "inconclusive" verdict can be delivered if the trace is not long enough to allow a complete analysis.

Many methodologies have been explored these last years to perform passive testing. In the following three subsections, we present some of the most important ones.

1.6.1 Passive Testing Interval Determination

This methodology tries to map the system collected trace (composed of input/output sequence) to its formal EFSM based specification. It seeks for a transition (or a set of transitions) of the specification where the couple input/output matches and where the predicate on variables is true or can not be evaluated (in the case of a trace that does not provide enough information about the system variables values). The values of variables are then deducted from the formal specification.

However, this methodology suffers from a possible loss of information. For example, assuming that x has previously been assessed at a value of 3, the Figure 1.6 shows that, in the absence of information about y, we have to choose two transitions leading to different values of x making it undefined.

Considering these deficiencies, a more efficient algorithm for errors detection was developed and presented in [LCH+02]. This algorithm is based on three aspects:

- Intervals to denote the values of variables, denoted $R(v) = [a, b]$ for variable v.
- Assertions or predicates on variables, denoted $asrt(\vec{x})$ on the vector $\vec{x}$ of variables.
- Candidate Configuration Sets (CCS) to formalize the analyzed environment of the

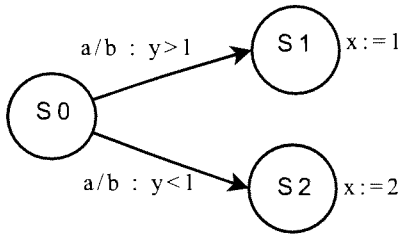


Figure 1.6: An EFSM Example.

system under test. A CSS is the triplet $(s, R(\vec{x}), asrt(\vec{x}))$ where s is the current state of the specification.

This algorithm tries to determine the values of the variables not using only a single value assignment (in the form of an interval) but a set of possible values for each variable. It refines as much possible the intervals in which the variables take their values.

The intervals

The intervals are a beginning of an answer to the information loss problem. Indeed, based on this concept, a variable is allowed to have more than one possible value. A variable v whose value is contained between two integers a and b will be defined by the interval $R(v)$ so that: $R(v) = [a; b]$. In the particular case where v is a constant value a, we have: $R(v) = [a; a]$. We then say that this variable v is decided. Three operations on intervals are possible:

- the sum of two intervals. We have: [a;b]+[c;d]=[a+c;b+d]
- the subtraction of two intervals. We have: [a;b]-[c;d]=[a-d;b-c]
- the multiplication of an interval by an integer:
  $w \times [a; b] = [w \times a; w \times b]$ if $w \geq 0$
  $w \times [a; b] = [w \times b; w \times a]$ if $w \leq 0$

The assertions

An assertion $asrt(\vec{x})$ is a Boolean formula on the vector of variables $\vec{x}$ that must be true at the current state of the analysis. The assertions are used to record constraints on variables. These constraints can arise from transition predicates and actions. In this way, if a transition is fired then its predicate is added to the assertion as well as updates (i.e. actions) that contain undecided variables in the right member of the equality. For instance, the action $x_2 \leftarrow x_1 + 1$ updates x_2. In this case, every term of $asrt(\vec{x})$ containing x_2 must be deleted and the term $x_2 \leftarrow x_1 + 1$ must be added to $asrt(\vec{x})$. As soon as we discover the value of x_1 we deduct trivially the x_2 one.

The Candidate Configuration Sets

A Candidate Configuration Set (CCS) is triplet $(s, R(\vec{x}), asrt(\vec{x}))$, where:

- s is the current specification state,
- $R(\vec{x})$ is the set of constraints on the variables (i.e. intervals),
- $asrt(\vec{x})$ is an assertion on $\vec{x}$.

The candidate configurations model the states where the system under test is. They also show the different constraints on variables. For instance, the following configuration $(S_1, R(x) = [1; 5], (x < 2) \wedge (x > 2))$ means that the system is in the state S_1 and that the value of x is contained between 1 and 5 but not equal to 2.

The algorithm uses two lists Q_1 and Q_2, Q_1 being the set of current possible CCS and Q_2 the possible CCS of the previous step. From Q_1 and an event e, we must obtain the corresponding transitions. A transition t will be fired if it exists a configuration in Q_1 whose constraints (the intervals of the variables and the assertion) are compatibles with the predicate p of t.

1.6.2 Backward Checking Technique

In [ACC+04], the authors propose a new approach that can be considered as the opposite to the one presented earlier in [LCH+02]. Based on the fact that the end of a trace correspond to a system state, we can already discover from this stage some information about the variables values if we look to the past of the trace.

The algorithm Backward-Checking operates in two stages. The first step is to go (back) in the trace W from its end to its beginning, mapping W to the specification machine. The aim is to reach all possible configurations X that can generate the trace W, i.e. all triplets $(s, R(\vec{x}), asrt(\vec{x}))$ candidate configurations set from which W could begin.

The second step is to explore all possible paths from X, to verify that W is reachable from the initial configuration of the specification.

In fact, validating a trace W is based on finding a path p that connects a configuration c and an element of X. p validates W if there exists a set of predicates and actions that can confirm the correction of the element of X. This approach results in a complexity that is at worst equal to the total parsing of the system specification, i.e. the full exploration of its graph accessibility.

1.6.3 The Invariant Based Approach

The foundations of the invariant-based passive testing technique can be found in [ACN03] and improved in [BCNZ05] where several modifications were realized. The leading idea of the invariant-based testing is simple. The specification is studied in order to disclose particular properties.

Invariants are properties that the implementation under test expects to satisfy. They are used to express constraints over exchanged messages between the system entities (in this case, remote Web services). Basically, they allow expressing that the occurrence of an event must be necessarily preceded or followed by a sequence of events. Once invariants are found, we just check them on the implementation traces. We remark that this is a two-step process:

- The invariant research (on the specification),
- The application of the invariants on the implementation trace.

Unfortunately, the search for invariants is a grueling task. When we attempt to model a temporal property with invariants we are confronted to an obstacle: the trace size should be long enough to encompass the whole invariant. For example, let consider the invariant that says 'when we have the input i_1 then in the future we will have the input i_n'. Let also consider the recorder trace ends with i_1/o_1. Then the invariant checking cannot be applied. An Input/Output invariant contains two parts:

- The test, which is an input or an output symbol.
- The preamble, which is the sequence that must be first found in the trace before trying to check the test.

There exist three types of invariants answering these requirements:

- The output invariants, when the test is an output symbol. The property has the form "immediately after the sequence preamble, we must always have the output test". Here are some examples of output invariants:
  - $(\underbrace{i_1}_{preamble} / \underbrace{o_1}_{test})$: which means "$i_1$ is always followed by o_1".
  - $(\underbrace{i_1/o_1)(i_2}_{preamble} / \underbrace{o_2}_{test})$: which means "immediately after the sequence (i_1/o_1) and the input i_2 we always have o_2". This example reflects to be an invariant of length 2 because its preamble sprawl on two I/O couples.
- The input invariants, when the test is an input symbol. The property has the form "immediately before the sequence preamble we must always have the input test". Here are some examples of input invariants:
  - $(\underbrace{i_1}_{test} / \underbrace{o_1}_{preamble})$: which means "$o_1$ is always immediately preceded by i_1".
  - $(\underbrace{i_1}_{test} / \underbrace{o_1)(i_2/o_2}_{preamble})$: which means "immediately before the sequence $o_1(i_2/o_2)$, we always have the input i_1".

- The succession invariants, that materialize complex properties such as loop problems. Its drawback is that there is nowadays no known technique to automate the search of this kind of invariants. We present an example of succession invariant use:
  - $(\underbrace{i_1/o_1)(i_2/}_{preamble}\underbrace{o_2}_{test})$
  - $(\underbrace{i_1/o_1)(i_2/o_2)(i_2/}_{preamble}\underbrace{o_2}_{test})$
  - $(\underbrace{i_1/o_1)(i_2/o_2)(i_2/o_2)(i_2/}_{preamble}\underbrace{o_3}_{test})$
  - the set of the three invariants creates the succession invariant. This invariant force the transition holding (i_2/o_2) to be fired twice before the transition with (i_2/o_3) is obligatorily fired. This kind of sequence is used to allow few attempts for a given operation (in our example two attempts) in a protocol before it returns a failure (o_3 can represent this failure).

With this formalism, it becomes possible to extract properties from the specification. The invariants are a powerful approach based on the properties of the specification but as we have already evoked it, research of such properties is not an easy task. If we delegate this task to a human it is likely to take a big amount of time. On the other hand, the existing algorithms like the ones described in [BCNZ05] are sensitive to non determinism of the EFSM specification for extraction of invariant with a length strictly greater than one. In addition, this method does not allow the detection of all types of errors and is destined to be used complementarily with other methods and then acts like an error filter.

1.7 Testing Web based systems

1.7.1 Active Testing of Web based systems

As the use of the Web applications is growing, the interest on testing these systems and number of works in industry and the academia is increasing as well. Among the recent works it can be quoted [ACD07], work in which the authors highlight the importance of the model quality to achieve the automatic generation of test cases. They also survey and compare several analysis modeling methods used in Website verification and testing. In the early stage of our work we took as a reference the work presented in [SM03b] that gives a slight overview of how a Web system can be modeled using SDL. This approach can be modified to use any TEFSM based language instead of SDL to describe the behavior of a Web page.

The work presented in [LqMW+06], proposes a methodology to perform Model based testing. To model the system they propose to use a UML class diagram for defining the Web application navigation model (WANM). They can describe the navigation relations between

the pages, the links between them and the forms in each page. Although, this method is interesting to describe the static relations between elements, it is not the case to describe the behavior of dynamic systems. This work also introduces the MDWATP tool, which uses as input a WANM model in order to generate test cases. Nevertheless, the methodology used to obtain and execute the test cases is not complete. Concerning the testing needed to help maintaining the stability of Web applications, numerous works are using the regression testing approach. Regression testing is the process of validating modified software to detect whether new errors have been introduced into previously tested code, and provide confidence that modifications are correct [GHK+98]. This testing technique is important to apply to the open source Web applications because of its continuous developing speed and changeable user demands.

In [ROM08], a model-based testing method is discussed where state charts are proposed for web application testing; however the description level of test execution automation is rather low in this method. In [AOA05], a technique called FSMWeb is presented. It copes with the state space explosion problem by using a hierarchical collection of aggregated FSMs. The bottom level FSMs are formed from Web pages and parts of Web pages called logical Web pages, and the top level FSM represents the entire system under test. In our work, we manage the state explosion issue by using the Hit-or-Jump algorithm described in [CDRZ99].

Some other works aim to optimize regression testing on Web applications, for example, the work presented in [XXC+03] proposes a method based on Slicing to avoid the re-execution of all the regression test cases of a Web application and select the test cases that will interact with the part of the Web application modified by an insertion of new code.

1.7.2 Tools for Active Testing for Web applications

The Web testing is defined as the testing of a Web application via the HTTP interface. The simplest but more inefficient way to achieve Web testing is to let a human interact with the Web application to examine if the system does what it should do. Several tools and methodologies have been developed to achieve an automatic testing on Web applications that do not have a formal specification.

TclWebtest

Tclwebtest [Tcl09] is a tool to write tests for Web applications. It provides an API for issuing HTTP requests and processing results. It assumes specific response values, while taking care of the details such as redirects and cookies. It has the basic HTML parsing functionality, to provide access to elements of the resulting HTML page that are needed for testing, mainly links and forms.

The execution of a test case written in *tclwebtest* will simulate a user that is interacting with the Web application through a Web browser. Using the links and forms it is possible to add, edit or delete data of the Web application by executing the test case script. Figure

1.7 illustrates the *tclwebtest* code for logging in into the dotLRN platform by requesting the register page, then filling the e-mail and password, and submitting.

```
::twt::do_request "http://dotlrn.org/register"
tclwebtest::form find ~n login
tclwebtest::field fill "user@mymail.com"
tclwebtest::field find ~n password
tclwebtest::field fill "mypassword"
tclwebtest::form submit
```

Figure 1.7: Example of tclwebtest code.

Selenium

Selenium is a web testing tool which uses simple scripts to run tests directly within a browser. It uses JavaScript and iframes to embed the test automation engine into the browser [HK06]. This allows the same test scripts to be used to test multiple browsers on multiple platforms.

Selenium gives the user a standard set of commands such as open (a URL), click (on an element), or type (into an input box); it also provides a set of verification commands to allow the user to specify expected values or behavior. The tests are written as HTML tables and run directly in the browser, with passing tests turning green and failing tests turning red as the user watches the tests run. Because Selenium is JavaScript-based and runs directly in the browser (the user can see the test running), it overcomes some of the problems encountered by users of HttpUnit [Htt09] or Canoo WebTest [Web09], particularly problems related to testing JavaScript functionality. An execution of a Selenium test on a real Web application is illustrated in the Figure 1.8.

Interaction Recorders

Projects as Selenium IDE [Too09a] or TCLwebtest Recorder [sR09] aims to semi-automatize the generation of Web test cases. Their goal is, as illustrated in Figure 1.9, to record the interactions of a human (an expert of the system) with the IUT. This trace of interactions can be then easily modified into an executable script capable of interacting with the Web application and reproduce (or replay) the trace of interactions. This script can be slightly changed to obtain a test case, in order to generate test cases in a semi automatic way. Finally, the interaction with the IUT will be managed by a machine testing and simulating the expert human user. This method is performed to generate in a semi automatic way test cases for non-regression testing [CMM07].

In Web testing, the IUT is tested via its interface as a black box. Therefore, the main point of reference to describe the system is the user's point of view. Among the tools that can capture the interactions of a user with a Web application to generate test cases, we can mention Selenium IDE [Too09b] and TclWebtest [sR09].

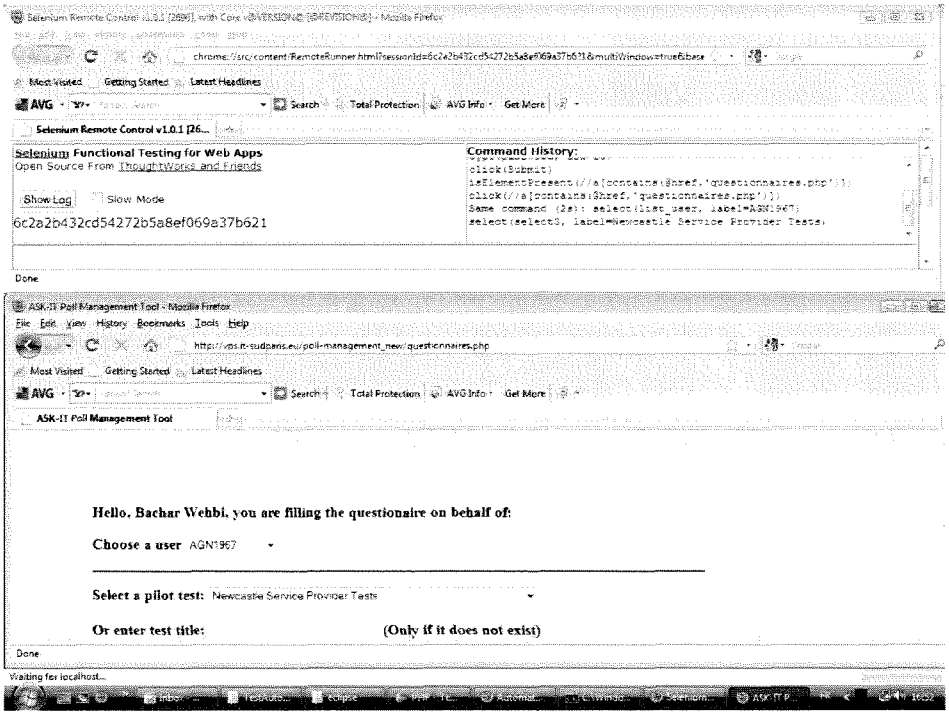


Figure 1.8: Execution of Selenium test cases on a Web application.

Selenium IDE

Selenium IDE (originally called the Recorder), which allows users to navigate their applications through Firefox and record their actions into a trace. This trace is saved in selenium script, and it can be then automatically transformed to PHP or Python to then re execute the trace of interactions. This traces can be used as the base for test cases.

TclWebtest Recorder

Even if *tclwebtest* is very intuitive, hand development of test cases for an exhaustive platform test requires a big amount of time and effort. To reduce this needed time, a plug-in called Tclwebtest-Recorder (TwtR) [sR09] for Mozilla Firefox has been created in 2006. Using the TwtR a trace written in *tclwebtest* of all the interactions that a user had with the Web application is provided. Then, test cases may be coded by adding variables and applying some changes to the code of this trace.

The TwtR is a tool based in the interaction recording as Selenium IDE, by using this tool it is possible to obtain the trace in a language based in TCL script language, more specifically in *tclwebtest* code. TwtR produces a static trace that does not contain all the interactions between the user and the Web application, but only the stimulations of the user to the Web application. This static trace can be used as a base for the creation of the functional testing process as presented in [CMM07]

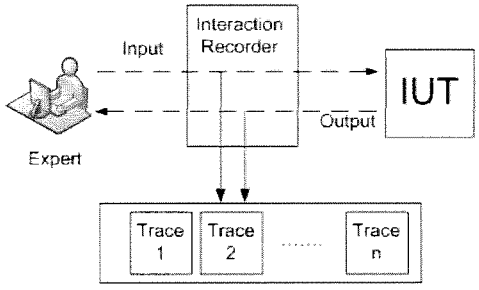


Figure 1.9: Generation of non-regression test cases by recording the interaction with IUT.

1.7.3 Passive testing of Web based systems

In the last years, the number of works aiming to test Web based systems using the monitoring approach has increased significantly. This can be due to the fact that most of the modern systems are migrating towards the SOA or SOA like architectures. Therefore, the machine-to-machine communications using specific interfaces and channels provide a new and interesting point of observation for the testers.

Among the recent works, we can mention the work of [BDSG09] that proposes to extend the architecture used to deploy Web services (SOA) with an *observer element*. The authors proposes to use FSM+ model (FSM that can take into account some timed constraints) to be used as the reference behavior to set verdicts over the tests.

In [AMN09a], the authors propose a framework to test the correctness of a timed system log by using a passive testing approach. The model used to describe the behavior of the system is a TEFSM, and they are inspired in the invariant approach taking into account some timed constraints. In this work, they present *PASTE*, the tool used to check the correctness of a system with respect to the specification through the use of invariants. *PASTE* could be easily adapted to test any communicating system. The flexibility of this tool for being adapted to different case studies is described in several works. For example, in [AMC+09] it is applied to test the correctness of a MANET routing protocol (OLSR). The same tool is also used in the work [AMN09b] that combines the classic invariant approach with probabilities and present the "Stochastic Invariants" aiming to proof the correctness of non deterministic systems.

Recently, a new tool based on the methodologies of [YD07], the TGSE is presented in [CFCB09] allows the coverage of transitions based on guard conditions (by examining variable values). In this approach, the authors model an execution trace as a test purpose (i.e. also referred as a TEFSM) by giving real values to each input message. After modeling the Web service (they use a composition of Web services) specification and the trace as TEFSMs and declaring their synchronization rules, they use TGSE to verify this system. If the TGSE output is a sequence, it means that this trace is a valid trace of the interactions among the systems.

The methodologies and tools that will be presented in the following chapters to perform passive testing are based on the invariant approach presented in [BCNZ05]. We also use

as reference of the behavior of the system a TEFSM like the approach used by the *TGSE* and *PASTE* frameworks.

1.7.4 Conclusion

The growing need of assuring the good deployment of the functional aspects in Web based systems is pushing the development of new methods and tools to test them. There are numerous efforts aiming to propose new methodologies and tools to test Web applications using the active testing approach and in less quantity it exist as well some works intending to test the Web applications and services by using the passive testing approach.

Following the active testing approach, there are some tools capable of interacting with Web applications like Selenium and Tclwebtest, but the generation of the test cases is done manually. Some efforts to semi-automatize the test case generation was done in the last years allowing to record a trace of interactions user-application [sR09] [CMM07] [Too09a], and then it is possible to manually edit this traces to obtain test cases.

Model based testing can push even forward these solutions allowing automatize the whole workflow of test, from the generation to the execution. We present in the following chapters a new methodology and propose the combination of different existent testing software and a new tool to perform the workflow of active testing on Web applications.

There are also a few efforts in the research field aiming to use passive testing in Web based systems. The invariant approach [ACN03] [BCNZ05] can be used to test collected traces of the exchange of messages between Web applications and external systems. Among the existent tools that use invariants, they do not take into account the time constraints.

In the following chapters we describe how to extend the invariants to be able to manage time issues and how it is possible to generate them using a TEFSM description of the Web application. It is also presented a new tool to perform timed passive testing on the communication between a Web application and an external system using timed extended invariants.

Part III

Formal Testing of Web Based Systems

Chapter 2

Active testing of Web based systems

Contents

"An error doesn't become a mistake
until you refuse to correct it."
Orlando A. Battista

2.1 Introduction

Along with the continuous development of big and complex Web applications, it becomes crucial to execute in a scheduled way a set of test cases to assure its functional stability and to make sure that the Web application still runs without minding the modifications applied to the implementation.

This chapter describes the complete process (described in Figure 2.1) of performing active testing on a Web application. The first contribution is done in the primary part of the process, the formal modelization. The entire process of test generation is based on the Web application formal model, facilitating afterwards the automatic execution of generated test cases. The methodology to obtain such a model specific for Web application is presented in Section 2.2.

The next part of the process, illustrated in Section 2.3, is the simulation of the model and the automatic generation of abstract test cases (tests without the details of the real implementation). In the following step, these test cases will be used to test a real Web application that leads to the second contribution in active testing; the instantiation of the abstract test cases into concrete test cases (Section 2.4). The instantiation allows the generated test cases to be executed in a real Web application implementation. This chapter explains the process of concretization of test cases and the translation to executable code that handles HTTP communications. It also covers how the outputs defined in the formal model are used to generate the logic that will assign the verdict of the test case.

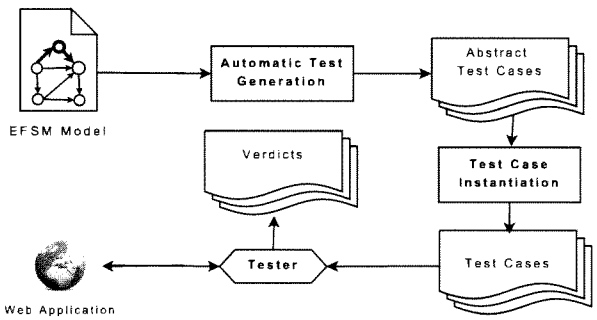


Figure 2.1: Methodology of test generation

Then, a case study named Mission Handler illustrating our methodology is presented in Section 2.5. This Web application is a prototype developed by Montimage [1] and inspired on the requirements of an industrial Web application, presented by Orange Group in the French ANR Project Politess [2]. Finally, the results of our methodologies and the execution results of the test cases on the Mission Handler are presented.

2.2 Modeling Web applications using TEFSM

In theory, the goal of modeling is to provide a better common understanding of systems. Furthermore, the objective of modeling a Web application is to provide an operational specification from a functional point of view. Hence, it is to provide an understandable representation of the implementation where some details are abstracted to leave just a simple description of the system behavior.

A formal model can also be used as input to existing validation and verification tools, such as interactive or random simulators, model-checkers or test generation engines. This modeling methodology is inspired on works that use formal techniques to model Web applications using EFSM described in SDL [SM03a], taking into account the timed issues as the work in [MNR08].

To achieve the modeling goal, we rely in this approach on the TEFSM model supported by the IF language [BGO+04] which provides the main concepts to design real-time systems. Moreover, several tools allowing its simulation and the generation of test sequence (a set of test cases) exist and are open source. A TEFSM model of a system consists of a set of processes; each one denotes a TEFSM that can communicate with each other via FIFO channels.

After defining the type of "mathematical" structure that will be used to describe a Web application (TEFSM), it is needed to define a methodology to build the model. We need to obtain a model that simplifies the processes to achieve the automatic test cases generation.

For example, in some approaches using SDL to model Web applications [SM03a], its users are part of the model.

We have chosen to model the behavior of the Web application from the point of view of the user. In this way the user is part of the environment that interacts with the model. By choosing this, we avoid providing to the modeler the freedom to add the security policies (or even logic processes) in the user side rather than in the application side.

[1] Montimage is a French SME specialized in software development and research. It was created in 2003 by a group of research engineers from Alcatel and Ericsson, experts in the development of software for telecommunications and treatment of critical and massive data.

[2] The ANR POLITESS project investigated methods to ensure a correct implementation of security policies for distributed information systems.

2.2.1 Modeling requirements for Web test

The methods that we use to build the model of the Web application are based on the respect of these fundamental requirements:

1. The model must be readable and understandable by a machine, that means a model without ambiguities where human intervention is needed. The use of formal mathematical models such as the TEFSM is then relevant.

2. The model must contain the behavior of the functionalities (data and timed constraints) that will be tested.

3. The model must contain the behavior of the functionalities (data and timed constraints) used in the test preamble.

4. The model specifies the interactions of the IUT with the environment as a black box. In this way the system is specified from the point of view of the user interacting with the IUT through its Web interface.

5. The users should be part of the environment and not part of the TEFSM.

2.2.2 Model definition of states, inputs and transitions

To describe a Web application from the user point of view, the TEFSM states are defined as the Web pages the user can visit. Following this approach, the changes from one page to another are defined with the transition function of the TEFSM presented in the previous chapter.

The execution of the change from one page to another is triggered by inputs of the Web application. For instance, if the user clicks on a link, he will go from one page to another; that is, the change between pages is triggered by the link. Likewise, in the model the transition between two states is triggered by an input. Consequently, the links and form submissions are defined as inputs of the model.

For example, let us imagine the standard login page to Web applications. In this page, we find a form to provide the *user* and the *password* to login and a link to register as a new user of this Web application. The login page (q_1) will be represented as a state having two possible inputs (login or register). Each input will trigger a transition that will lead the user from the actual state (page) to a new one.

As illustrated in Figure 2.2, the system has two inputs, and depending on them, we have three possible paths:

- If the input login(usr,pass) is received and the *user* and *password* matches; then the user will be guided to the welcome page (q_2).

- If the input login(usr,pass) is received and the *user* and/or *password* do not match; then the user will be redirected to the same login page q_1.

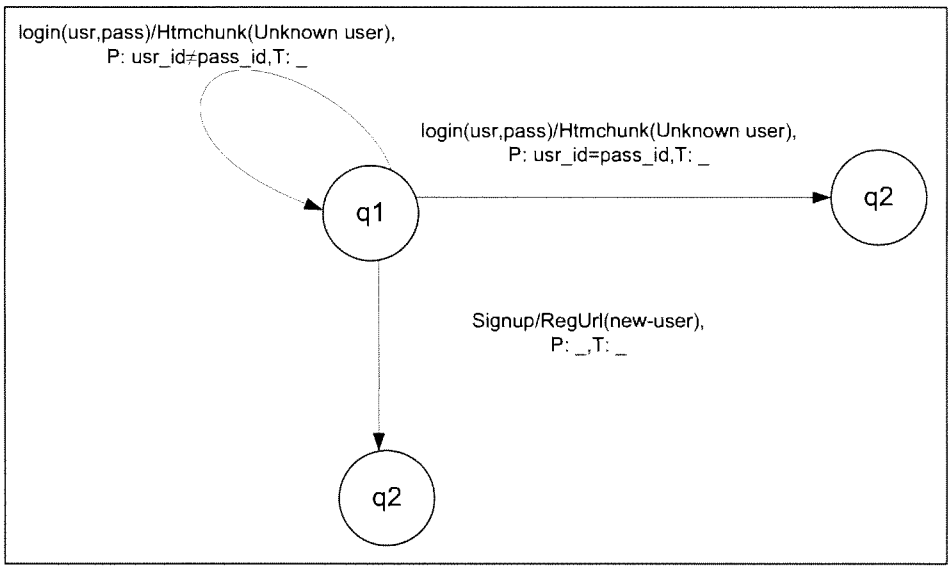


Figure 2.2: Example of modeling the login of a Web application.

- If the received input is the link to signup, then the system will guide the user towards the page q_3 to fill its information and then to become a new user of the Web application.

2.2.3 Model definition of outputs

From the user point of view, the output of a Web application when an input is received, is a response Web page (or a modification of the existent page if we are using Ajax).

In general, to build the logical part that assigns the verdict, we use the element or elements that we seek in the response Web page. These elements are the point of reference to determine if the system did what it was expected. For instance, if we provide as an input the *login(user,password)* signal, we expect the welcome page of the system. If the Web application output is an "ERROR 404, page not found" Web page, or any other page where we did not find the expected welcome message, we could say that the system did not behave as we expected. That is, the system is not conforming to its specification.

To build the TEFSM model (described in IF language) it is possible to define as the output the complete HTML that we are expecting after an input, but it is not very flexible as a solution. Another solution is to define the output as "what are we seeking after the input". Using our methods of test generation, we can seek: a specific text, a piece of HTML, the URL or a combination of these three elements of the page the system returned after a provided input.

For example, Figure 2.2 illustrates that after clicking in the link *signup*, it is expected that the next page represented by the state q3 has as URL "new-user".

A good modeling of the output is mandatory in order to generate a good test sequence,

the output is the key element that our *GeneraTCL* tool (introduced in Section 2.4.1) will use to create the logical structure to assign the test case verdict.

2.2.4 Model building methodology

The methodology to build the model of a Web application contains 3 steps:

1. Create one state for every Web page of the Web application.
2. Create the transitions between the states. Based on the TEFSM definition 3, a transition is triggered when an input is received (described as an atomic action Act) and the guard G composed of predicates on variables $\vec{x}$ and clocks $\vec{c}$, is satisfied.

   In other words, the transition from one state to a new one is triggered when a input is received (when the user clicks a link or submits a form) and if the conditions (if any) are satisfied.
3. Assign an output (described as an atomic action Act in definition 3) to the transition as explained in Section 2.2.3.

To illustrate the modelization we can use the Speed Game, a little Web application that has five Web pages:

- Page A: a Web page with one link whose role is to fire the clock *c1* and to lead the user to the page B when clicked,
- Page B: a Web page with one link pointing to the page *redir*,
- Page *redir*: a Web page made in PHP without user interface. The role of this Web page is to redirect the user to the page C if the clock ck1 $<$ 5, or to page D if the clock ck1 $\geq$ 5.
- Page C: Is a Web page with one link that is pointing to the page A
- Page D: Is a Web page with one link that is pointing to the page A

To proceed with the modelization it is important to remark that from the user point of view, the system (illustrated in Figure 2.3) has just four pages: A,B,C and D. Since the page *redir* has no User interface; it is not observable by the user. Therefore, the TEFSM model of the Speed Game (illustrated in Figure 2.4) has just 4 states.

Based on the definition 3, the transition between states is represented with the format $<s_i,s_f,Act,G>$ where

- s_i and s_f are the initial and final state of the transition respectively;
- Act is an ordered set (sequence) of atomic actions including inputs, outputs, variable assignments, clock setting, process creation and destruction, and

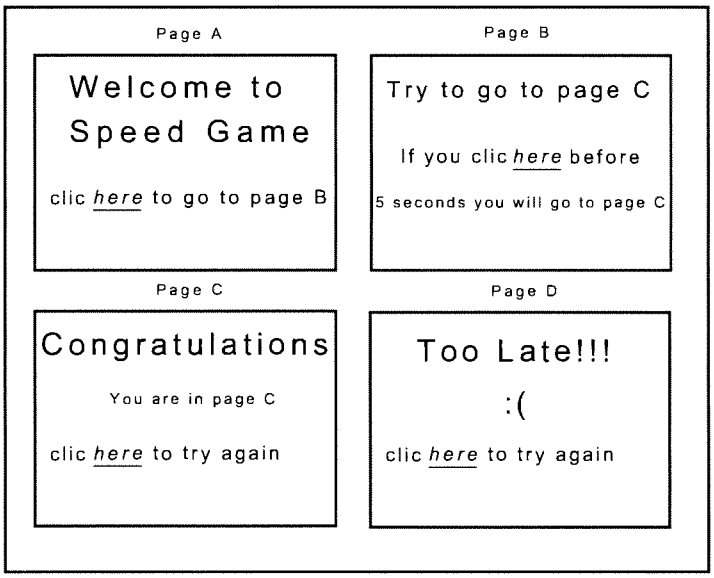


Figure 2.3: Speed Game Web application

- G is the transition guard which is composed of predicates (Boolean expression) on variables $\vec{x}$ and clocks $\vec{c}$;

Figure 2.5 illustrates the Speed Game using the simplified graphic notation that is used in this work representing the TEFSM models described in IF language. Until now, all the *Act* of the TEFSM are represented, excepting the outputs. The addition of the outputs is imperative because the verdict of the test cases is based on them.

The outputs are used as a point of reference to define if the IUT reacts as expected. An observable output (from the user point of view) can be either an element in the displayed HTML or the URL of the new state after the transition. To follow the modeling methodology we must add now an output on each transition. In this example the outputs are just pieces of text that are used as signatures for each observable Web page. For instance after the input L:c, we have two possible transitions: For the transition leading the system to C we can set as output the text to find "*Congratulations*"; in the other hand, for the transition that leads to D, we can set as output the text to find "*Too Late*". The final diagram of this part of the model is presented by Figure 2.6

2.3 Abstract test cases generation

The previous section covered the methodology to build a formal model of a Web application, the following sections present the automatic generation of abstract test cases from the IF specification. This generation is done using the *Testgen-IF* tool developed in our laboratory. Nevertheless, to test a real Web application, the abstract test case must be

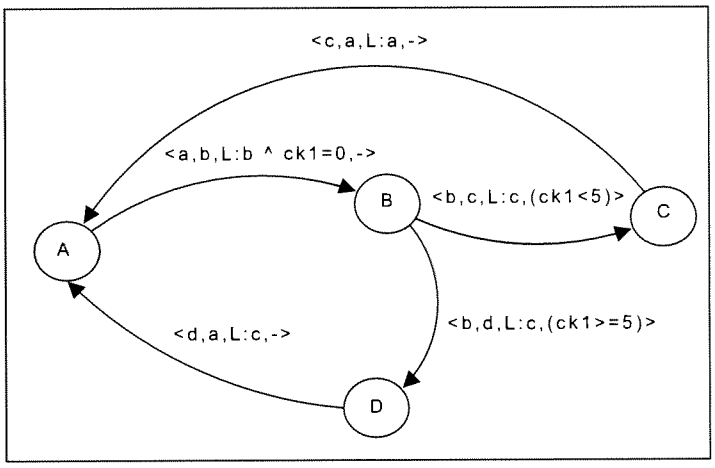


Figure 2.4: TEFSM of the Speed Game Web application

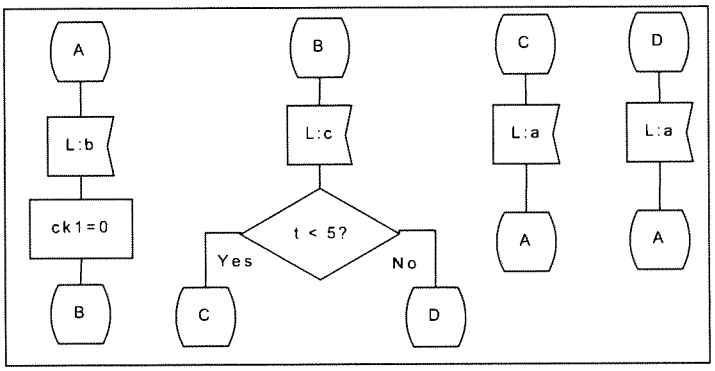


Figure 2.5: Speed Game represented with a simplified graphic notation

instantiated in order to obtain a concrete and executable test case. The instantiation methodology is presented later in Section 2.4.

The abstract test case is the basis used to build the instantiated test case (or real test case) that will be used to test a real system.

2.3.1 TestGen-IF Algorithm

TestGen-IF implements a timed test generation algorithm based on the Hit-or-Jump exploration strategy [CDRZ99]. This algorithm efficiently constructs test sequence with high fault coverage, avoiding the state explosion and deadlock problems encountered in exhaustive or exclusively random searches respectively. It allows producing a partial accessibility graph of the implementation under test (IUT) specification in conjunction with the IF simulator [BGO+04].

At any moment, a local search is conducted from the current state in a neighborhood of the accessibility graph. If a state is reached and one or more test objectives are satisfied (a

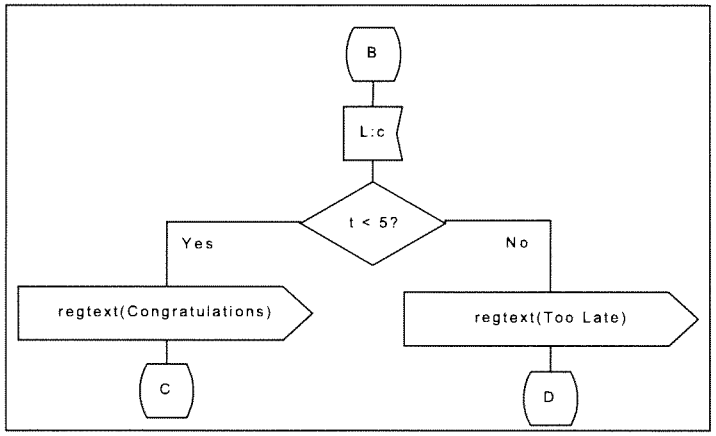


Figure 2.6: Transitions triggered with L:c with its Outputs

Hit), the set of test objectives is updated and a new partial search is conducted from this state. Otherwise, if a search depth limit is reached without satisfying any test purpose, a partial search is performed from a random graph leaf (a Jump). This algorithm terminates when all the test objectives are satisfied or when no transition is left to explore. The abstract test case is the path constructed on the fly from the initial state of the IUT specification, containing all the hit and jump states.

2.3.2 TestGen-IF Architecture

The active testing tool is illustrated in Figure 2.7. The test objectives represent the timed system objectives to be tested (see Section 2.3.3). The automatic test generation procedure is combined with the IF specification and the test objectives. It is up to the tester to choose the exploration strategy of the generated partial graph: in depth (DFS) or in breath (BFS) [CMM+06]. During this generation, when a test purpose is satisfied, a message is displayed to inform the user. The number of test objectives already found and the number of those missing are also provided. Based on this approach, an abstract test case is generated. A test suite is composed of a finite set of test cases (or scenarios) described in an input/output format. It is used to stimulate the implementation under test (IUT) to validate its reaction.

2.3.3 Fixing the Test Objectives

Test objectives Formulation

In order to formulate timed test objectives using TestGen-IF tool, several options are permitted:

- State constraint objectives: expressing that a system can be in a specific state;
- Action constraint objectives: corresponding in particular to signals actions (e.g.,

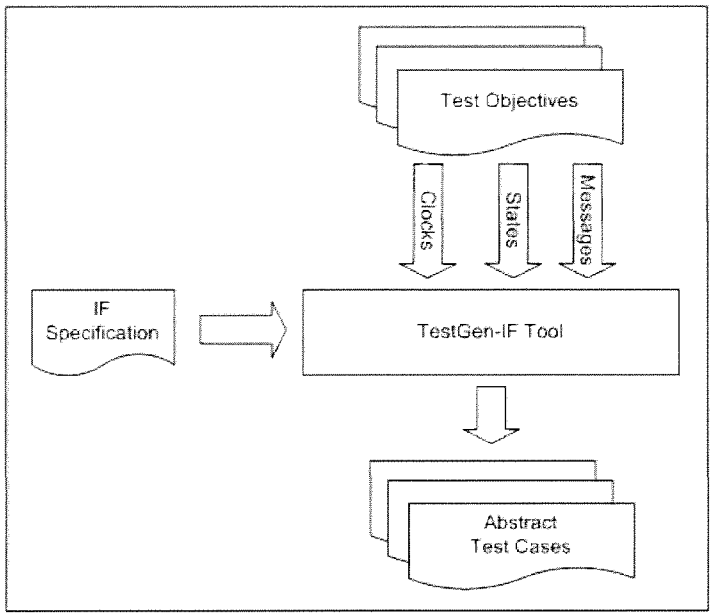


Figure 2.7: Basic architecture of the TestGen-IF tool.

input signal, output signal) and describe that an action can be executed in a state (optionally) at a specific time;

- Clock constraint objectives: expressing that a clock can have a specific value, optionally coupled with a specific state.

For instance, the constraint "*action = input sg in state = s when clock c = d*", describes that the signal *sg* has to be received in the state *s* when the clock $c = d$. Timeouts and deadlines can be usually described by clock constraint, whereas the flow requirements can be described using state and action constraints.

2.3.4 Test Generation with TestGen-IF

TestGen-IF is a tool conceived to automatically generate test sequences according to the test objectives. We also manually define, by grouping around, adequate interval values for data variables in order to reduce the accessibility graph size and to avoid state explosion problems.

A set of timed abstract test cases are generated based on the IF specification of the Web application and the timed test objectives for each rule using TestGen-IF. These test cases are then filtered according to the actions (input, output and delays) relating to the system under test.

The resultant abstract test case is the path of the partial accessibility tree exploration described in an input/output format as illustrated in Figure 2.8. For instance, the first line of this figure shows the input/output trace when a user does login to the system. It

illustrates that for the input signal *login* with the variables *user* and *userp*, the output is the signal *htmchunk* with the variable *text_to_find_0*; both signals belong to the *webint* process of the IF model.

```
1. {webint}0 i ?login{user,userp} !htmchunk{text_to_find_0}
2. {webint}0 i ?new_mission{} !htmchunk{text_to_find_1}
3. {webint}0 i {webint}0 ?search_m{boston,sgd,srd} !urlexpr{url_to_compare_1}
```

Figure 2.8: Path of exploration of the accessibility graph in Input/Output format.

2.4 Test Case Instantiation

As it is explained earlier, the model of a system is the abstraction of a real implementation. This means: a simplification of the real system where all the irrelevant details [3] are abstracted to leave just a simple description of its behavior. In this way, a model is easily understood and evaluated.

In order to execute the generated test cases to a real Web application, it is mandatory to instantiate them. The instantiation of an abstract test case is composed of two subprocesses: the concretization (addition of details) and the translation to executable scripts. The final test case is a script that contains the details of the implementation such as: the URL where the pages are instantiated and real values for the variables of the signals. This process is called the test case instantiation.

To be able to test a Web application, we need a tool capable of communicating via HTTP (or HTTPS) with the implementation under test. The instantiation is performed using our *GeneraTCL* tool that is introduced in the following section.

2.4.1 GeneraTCL Tool

The *GeneraTCL* tool, illustrated in the Figure 2.9, is applied to concretize the abstract test cases and then translate them into an executable script able to interact with the IUT. In the concretization process, some details of the implementation (as the username and password of a real user) are added to the abstract test cases. These details are needed to perform the interaction tester-IUT. Then, in the translation to executable code process, the concrete Input/Output traces are replaced by:

- TCLwebtest code, that allows HTTP communication with the IUT,
- TCL code, to build the logical structure (e.g *if-then-else* structure to set the verdict) of the test case.

[3] Variable initializations or SQL queries are not part of the behavior of the Web application from the point of view of the user. Therefore they are irrelevant details to generate black box test cases.

After the execution of these two processes, the provided output is a concrete and executable test case, i.e. an instantiated test case.

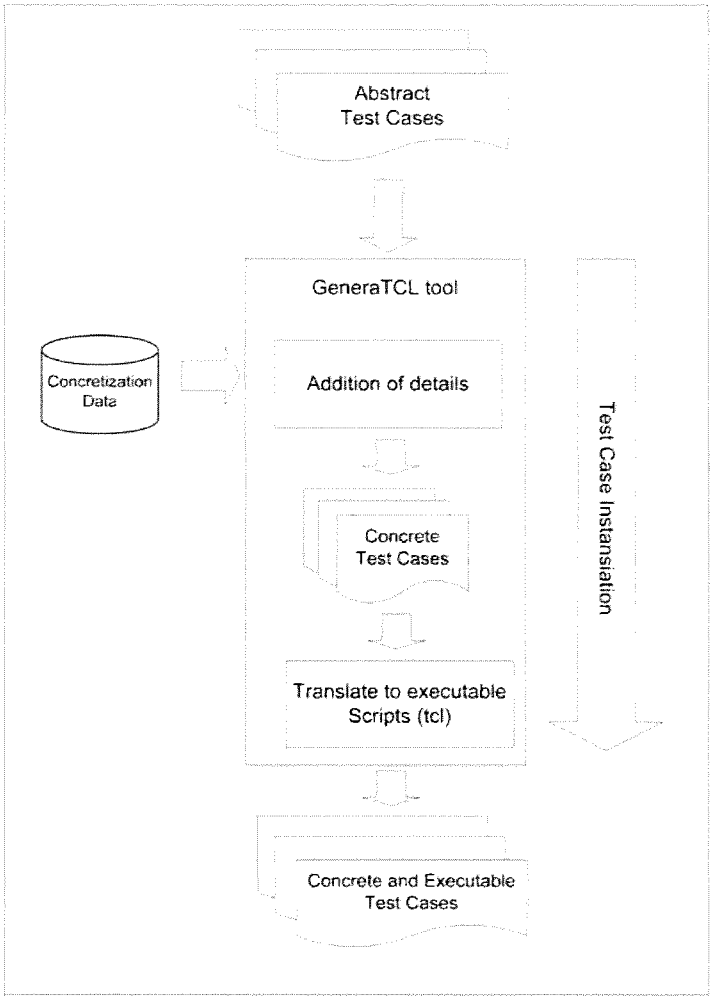


Figure 2.9: Basic Architecture of the GeneraTCL tool

2.4.2 GeneraTCL Instantiation Methodology

In order to detail the methodology and algorithms of GeneraTCL, we first have to explain the structure of the abstract test cases that are used as input by GeneraTCL. These abstract test cases (generated by TestGen-IF tool) are provided as traces in input/output format and are sets of:

- Delays: a delay represents an amount of time that the tester has to wait for, before performing any input action.

- Input signals: the tester has to stimulate the Web application by applying a set of HTTP(S) requests, called inputs.
- Output signals: the tester has to access the Web system answers to analyze it and to check if it conforms to the expected reaction as described in the formal specification of the system.

These data are treated by *GeneraTCL* in different ways, the instantiation of these three types of data are performed in the following way:

Delay Concretization

The *delay* type of data need no concretization. It is directly translated into an executable script in the abstract test case. The translation is illustrated later in this Section.

Input Concretization

To concretize the abstract inputs provided by *TestGen-IF* tool, it is important to know the types of HTML elements that correspond to input signals of the Web system under test. In this work, the Web system will be limited to receive just three types of inputs from a user via a regular browser: (1) a URL set in the address bar, (2) a link in the body of the page or (3) the submission of a form in the body of the page. Actions such as drag-and-drop and other Ajax functionalities are not considered in this work.

The concretization process of our methodology consists in mapping the signals into the three types of inputs that the Web application can receive. It is important to highlight that some IF signals can be mapped just to one single interaction with the Web application, e.g. following a link. Other signals are mapped to a set of interactions, e.g. submitting a form.

For example, considering the form submission, there are several interactions that must be performed by the tester, i.e. filling text fields, selecting radio buttons, selecting check-boxes and finally submitting the form. In these cases the signal is mapped into a HTML element (e.g. a form) and also to each signal parameter (e.g. text fields).

To perform the mapping, we propose two tables containing required information of the input signals and their parameters to transform them later into an executable script. Both, the *signal_info_table* and the *parameter_info_table* are illustrated in Tables 2.1 and 2.2. To access the information they contain, we can take advantage of standard SQL queries. For each signal, the table *signal_info_table* stores the following data:

- *signal:* the name of the input signal in the IF specification,
- *html_element:* the HTML type of the element that corresponds to the signal,
- *html_name:* the name or id of the HTML element. For example, in the case of a link, this is the URL where the link is pointing.

Table 2.1: Signal_Info_Table Example

signal_name	html_element	html_name
login	form	/register
signup	link	/register/newuser
logout	link	/signout
htmchunk	regtext	void

Then, for each parameter of an input signal, the information stored in the Table 2.2 is:

- *parameter:* name of the parameter signal in the IF specification,
- *of_signal:* name of the input signal that uses this variable,
- *html_element:* the HTML type of the element that corresponds to the parameter,
- *type:* the type of the variable expected by the html_element, e.g. integer, string, etc.
- *html_name:* the name or id of the html_element.

Table 2.2: Parameter_Info_Table Example

parameter	of_signal	html_element	type	html_name
user	login	textfield	string	email
password	login	textfield	string	password

The last part of the concretization process is to replace the name of the variables in the traces by values that the real implementation uses, e.g. to replace user1 by gmorales, a real username of the implementation. The substitution table is illustrated in Table 2.3

Table 2.3: Substitution_Table Example

variable_name	replacement
user	gmorales
password	m!lat0
text_to_find_0	Welcome

Input Instantiation Methodology

This part of the input instantiation uses Tables 2.1, 2.2 and 2.3 to do the mapping between the input signals and some details of the real Web application. In this methodology we use the notation introduced in Table 2.4.

GeneraTcl uses as input the file containing the abstract test cases, this file is parsed line by line. For each line, the methodology illustrated in Figure 2.10 is applied to obtain the instantiation of the input signals. Besides, let us define the Ω function used to set pointers to the tables SIT (table 2.1) and PIT (table 2.2).

Table 2.4: Notations used in the Methodology

Notation	Meaning
sg	Signal of the abstract test cases (inputs and outputs)
param	A parameter of *sg*
SIT.*element*	An *element* of the *Signal_Info_Table*
PIT.*element*	An *element* of the *Parameter_Info_Table*

Definition 5 *(Active row) Let N and M be a set of finite integers, and D a finite set of strings. Let $n \in N$, $m \in M$ be two integers and $d \in D$ be a string. Let the table $T_{m,n}$ with name na be a $m \times n$ matrix (a table with m rows and n columns) where each column co_i is identified with the element $el_{1,i}$. We define the function Ω: $na \times N \times M \rightarrow D$ as the function allowing to provide the row number (m) where the element d exists the column i of the table na.*

$\Omega(na, el_{1,i}, d) = j$ *if and only if* $el_{j,i} = d$

Note that using the input instantiation methodology, for each line of the abstract test case the input is captured, then classified (as link or form) and then translated to the TCLwebtest that will interact with a real Web application.

Finally we set the real values to the variables of the abstract test cases. This is done by replacing texts in the abstract test cases using the Substitution_Table illustrated in Table 1.3. For instance, if in the abstract test case we have the input *?login(user,userp)*, then we can replace *user* and *userp* with the real username and password of the user, obtaining as a result *?login(gmorales,m!lt0)*.

Output Concretization

The output provided in the abstract test case provides the expected output of the Web application (IUT) after an interaction with the tester. After this, we are interested in analyzing the response we obtained and also to know the current state vis-à-vis the model. The output must provide information that the tester will be able to analyze such as:

- One or several chunks of HTML we can find in the current Web page,
- The current URL that is in the address bar of the browser.

Output Concretization Methodology

For each *sg*

1. Assign the SIT.html_element attribute to identify what we are seeking in the response Web page of the IUT.

   The concretizations of the output start with the mapping between the model variables and the real information of the Web application. To distinguish if the output is a piece

of HTML or a URL, we introduce two new *html_ elements* in the *Signal_ Info_ Table*: *regtext* (it means that the *param* is a text or html chunk to be found) and *regurl* (it means that the *param* is a URL that needs to match a value).

2. After adding the type of output (regtext or regurl) as an attribute of *sg*, by using the *substitutions_ table* it is possible to replace the variable by a real value. For example change *text_ to_ find_ 0* by *"Welcome to the speed site"*

Translation to Executable Code

Delay

The delay is transformed directly to the TCL code '*after n*' where n is in millisecond (ms). For instance, if in the abstract test case '*delay 10*'[4] is found, it will be translated to the TCL code '*after 10*1000*60*'. (See Algorithm 2.4.2, lines 3 to 5)

Input

We need to translate the concrete inputs to *TCLwebtest* script in order to interact with the Web application.

The TCLwebtest actions are built dynamically by the *GeneraTCL* tool and can be divided into three categories: following a link, submitting a form or setting a URL in the browser. The inputs translation methodology is presented in pseudo-code in Algorithm 2.4.2 (lines 6 to 36). By performing this algorithm the following parts of the test case will be built:

- The script of the test preamble: a sequence of inputs (operations) that will lead the system to a state where the test case can be executed. During this preamble, system outputs are not analyzed. For example, in a Travel system, to test the creation of a mission, the user needs to be authenticated by that system.

- The script which will stimulate the system to test it.

Output

The last step of the methodology consists in developing the scripts that analyze the response (or reaction) of the Web application (Algorithm 2.4.2, lines 38 to 49). This script also assigns the verdict (pass or fail). Basically it checks whether the IUT did what it was supposed to do or not.

The IUT outputs can be classified in two categories:

[4]We consider this as a delay of ten minutes

Algorithm 2 Instantiation Methodology

Require: An abstract test case TC, signal_info_table, parameter_info_table and substitution_table tables. Let act be a delay or an observable action in TC and let $sg(d_1, d_2, ..., d_k)$ be a signal instance of $sg(x_1, x_2, ..., x_k)$. d_j is denoted in_j if sg is an input signal and out_j if sg is an output signal. $(0 < j < k + 1)$

```
1: for each (act_i ∈ TC) do
2:    /*(where i ∈ N, 0 < i < n + 1 such that n is the number of actions and delays in TC)*/
3:    case (act_i = delay n) do
4:       TCL_script: after n;
5:    end case
6:    case (act_i = input sg_i(in_1, in_2, ..., in_k)) do
7:       if (html_element(sg_i)= url) then
8:          TCL_script: do request url;
9:       end if
10:      if (html_element(sg_i)= link) then
11:         TCL_script: follow link;
12:      end if
13:      if (html_element(sg_i)= form) then
14:         TCL_script: form find ~n html_name(sg_i);
15:         for (each parameter x_j of sg_i) do
16:            /*(where j ∈ N, 0 < j < k + 1)*/
17:            TCL_script: field find ~n html_name(x_i);
18:            case (html_element(x_j) = textfield) do
19:               TCL_script: field fill in_j;
20:            end case
21:            case (html_element(x_j) = textarea) do
22:               TCL_script: field fill in_j;
23:            end case
24:            case (html_element(x_j) = checkbox) do
25:               if (in_j = 1) then
26:                  TCL_script: field check html_name(x_j);
27:               else
28:                  TCL_script: field uncheck html_name(x_j);
29:               end if
30:            end case
31:            case (html_element(x_j) = radiobutton) do
32:               TCL_script: field select in_j;
33:            end case
34:         end for
35:         TCL_script: submit form;
36:      end if
37:   end case
38:   case (act_i = output sg_i(out_1, out_2, ..., out_k)) do
39:      if (html_element(sg_i)= regurl) then
40:         TCL_script: assert {[response url] == html_name(sg_i)};
41:      end if
42:      if (html_element(sg_i)= regtext) then
43:         for each (parameter x_j of sg_i) do
44:            TCL_script: assert {[field get_value find ~n html_name (x_j)]== out_j} ;
45:         end for
46:      end if
47:      call deduce_verdict procedure;
48:   end case
49: end for
```

- Observable outputs: TCLwebtest is basically dedicated for testing Web application accessible by Web user interface. It offers some basic HTML parsing functionalities and commands for the manipulation of the HTML elements of Web pages. The reaction of the system can be provided in one or many HTML pages of the Web-based system. In general, it is a notification message that stipulates that the desired

action succeeded or failed, for instance an authentication. Sometimes, this reaction can be more difficult to discover e.g. reloading the current page or navigating to another Web application page. In all cases, we need to define both Tables 2.1 and 2.2 for the output, as well as their parameters and follow the same methodology developed in the input instantiation case. To analyze system Web pages, we use the TCLwebtest *response* and *find* commands to locate the HTML elements we want to test. Then we relay on the TCLwebtest *assert* command to compare the displayed Web page values and the output signal parameters, and deduce the adequate verdict.

It is important to highlight that *TCLwebtest* commands are used only to communicate with a Web application via HTTP, it does not contain logical commands such as *if-then-else*, *while*, etc. Therefore the outputs are translated to *TCLwebtest* and TCL logical structures to analyze the responses of the Web application.

- Non observable outputs: the system may react to a user operation by performing an action which is non-observable from this user's point of view (and as a result of the tester). For example, we can consider *adding/editing/deleting* of information in a specific data base or sending a notification email to a specific user. This case has not been tackled in this work.

For the first line of the abstract test case of Figure 2.8 *"?login{user,userp} !htmchunk{text_to_find_0}"* we show the results of the instantiation in Table 2.5. For this instantiated test case, we need to add some code at the beginning of the test case to set the *response* variable equal to 1. If at the end of the execution of the test case *response* was not changed to 0, then its verdict is Pass.

Table 2.5: Instantiated Test Case

Signal and Parameters	type	Instantiated
?login{user,userp}	input	TCLwebtest::form find ~a "register" TCLwebtest::field find "user" TCLwebtest::field fill "gerardo.morales@montimage.com" TCLwebtest::field find ~n "password" TCLwebtest::field fill "m!lat0" TCLwebtest::form submit

!htmchunk{text_to_find_0}	output	if {[catch {tclwebtest::assert text "Created!"} errmsg]} {
		aa_error "The text: Welcome was not found in the body of the page"
		set verdict_msg "Text not found: Welcome"
		set response 0
		} else {
		aa_log "The text: Welcome was found"
		}

2.4.3 Test Cases Execution

The execution of the test cases is performed using a dedicated testing tool proposed by the OpenACS community [Ope09]. This tool is called ACS-Automated-Testing tool; it allows executing the instantiated test cases interacting with the Web-based application under test, and also displaying the verdict of each test case (illustrated in the Figure 2.11). The

ACS-Automated-Testing tool is, in itself, a Web application but we will refer to it just as *the tester* to avoid confusions between this system and the tested Web application.

2.5 Real Case Study

2.5.1 Mission Handler Description

The implementation of our testing methodology was performed on a prototype implementation, the Mission Handler Web application. This implementation was deployed to validate the work presented in [MLMC08]. This Web application is used to handle the missions of the employees of an enterprise as a workflow.

Use Case Diagram

As it is illustrated in Figure 2.12, when an employee needs to travel for a mission, he can search flights, select them and then wait for the authorization of a superior who must validate the mission. There are two types of users: (1) normal users and (2) mission validators, who may also be an employee.

The Mission Handler is connected to an external system, the travel agency, whose role is to send all the information concerning the flights and make the booking with the airlines. Communication between the Mission Handler and the travel agency system is implemented using Web services based on SOAP.

After the creation of a mission, the validators must validate the missions within a period of three days; if not, the system starts to send daily reminders to the validator. This kind of timed constrained features are well described by the IF language. Therefore, this case study let us test timed systems using our methodologies of modelization, testing generation and instantiation.

Mission Creation Activity Workflow

The main feature of the Mission Handler is the mission creation, as illustrated in Figure 2.13. The employee creates the mission following the steps: (1) request a new mission (2) give to the Mission Handler the city, departure and returning date concerning the mission and (3) choose the flights.

The mission is then created and pending of its validation (approval or denial).

2.5.2 Implementation of the Mission Handler

The Mission Handler Web implementation illustrated in Figure 2.14 is a prototype that we developed based on the use of the OpenACS framework [Ope09]. The Mission Handler runs over AOLserver 4.5, whose logical part uses TCL as scripting language and PorsgreSQL as database. For the experimentations the Mission Handler was instantiated in http://gmorales.net:8000

2.5.3 Modelization

The functional model of the Mission Handler was developed using the methodology described in Subsection 2.2. The resultant model before the addition of outputs is introduced in Figure 2.15, an 11 states TEFSM described in IF language.

2.5.4 Generation and Instantiation of the Test Cases

Using as an input the IF Model that describes the functional aspects of the Mission Handler, the abstract test cases were generated using the TestGen-IF tool and based on the test objectives listed in the appendix A. Figure 2.16 gives the abstract test case for the creation of a new mission. This test case contains the logic aspects, i.e. the actions that must be executed to achieve the interactions with the feature to be tested. However we need to add details of the real implementation (concretize) like the name of the links, forms and pages that the tester will interact with.

To achieve this concretization, the abstract test case and Tables 2.6 and 2.7 are used as inputs by *GeneraTCL*. The information in the *Signal_ info_ table* is used to provide the HTML types of inputs and outputs of the test cases.

2.5.5 Concretization of Delay

There is no need to add details to the delays of the abstract test cases at it is described in Subsection 2.4.2, therefore the delays skip the concretization step and pass directly to the translation step.

Concretization of Inputs

The inputs that are in the abstract test cases represent the inputs that the Web application will receive, i.e. the stimulation of the tester to the IUT. The types of inputs of a Web application can be links, forms or URLs. If an input is of type *link*, then just the *Signal_ Info_ Table* is needed. If it is a form, details concerning the type of sub elements (or parameters) need to be added. A form may contain different types of elements like text areas, radio buttons or checkboxes among others. Therefore, in that case, we need to search all its parameters and parameter types in the *Parameter_ Info_ Table*

After adding the details of input types it is mandatory to link each input with an identifier. For instance, the name of the form that will be filled or the text of the link that will be clicked. TclWebtest can handle different identifiers for a link or form, these identifiers are *html_ name* in the *Signal_ Info_ able*. Moreover, it is also needed to link each parameter with an identifier, main goal of the *html_ name* column of the *Parameter_ Info_ Table*.

Concretization of Outputs

The outputs in the abstract test cases represent what we expect to have after each interaction. This could be a keyword among the HTML or the URL of the page that the IUT

Table 2.6: Signal_Info_Table

signal_name	html_element	html_name
login	form	/register
signup	link	/register/newuser
new_mission	link	/startmission
validate_miss	link	/confirmvalidation
change_profile	link	/pvt-home
delegate	link	/delegate
logout	link	/signout
return	link	/travel
search_m	form	/findflights
go_flight	link	/findflights-2
return_flight	link	/createmission
htmchunk	regtext	void
urlresp	regurl	void

Table 2.7: Parameter_Info_Table

parameter	of_signal	html_element	type	html_name
user	login	textfield	string	email
password	login	texfield	string	password
city	search_m	texfield	string	city
date1	search_m	texfield	string	departure_go
date2	search_m	texfield	string	departure_return

provides after its stimulation with an input.

Each one of the outputs is concretized as regtext (a text among the HTML) or regurl (the URL of the page).

For example, in Figure 2.16, in the first line of the trace, the output *htmchunk(text_to_find_1)* is given. The output *htmchunk* is defined in Table 2.6 as type *regtext*, that means that this output is interpreted as executing regular expression of a specific text in the HTML of the page. This specific text is *text_to_find_1*

For example, after the concretization, the input/output trace "?new_mission{} !htmchunk{text_to_find_1}" is interpreted as the following set of ordered actions:

1. Search and follow the link that has as a reference href="startmission" (startmission is the URL of the signal new_mission in the Table 2.6),
2. Parse the page that was received after clicking the link,
3. Search in the HTML of the received page the text "text_to_find_1",
4. If the text was not found, send "error" and set verdict as "Fail"

It is important to mention that the last action can set the verdict as Fail and stop the execution of actions.

The last step of the concretization consists in doing simple text substitutions in the actions using Table 2.8. In this step, the *text_to_find* or the *url_to_compare* will be replaced by the real text the tester will seek in the returned Web page. Information that will be added to the actions in this case study, includes:

Table 2.8: Substitution_Table Example

variable_name	replacement
user	gerardo.morales@montimage.com
password	b0t3s
admin_user	edgardo@montimage.com
admin_password	m!s4
date_go_1	01/02/2009
date_return_1	06/02/2009
city1	boston
text_to_find_1	Welcome!
text_to_find_2	"Mission created"
url_to_compare_1	http://gmorales.net:8000/travel

- The real text that will be sought in the page e.g. the text in the HTML that is displayed by the browser
- The real URL of the page
- The real data to interact with the system such as
  - Users
  - Passwords
  - Data that will be used to fill the forms

Finally, as a result of the trace concretization, a set of ordered actions is generated. If an expected output is not found, the execution of the actions will be interrupted and the verdict will be set as "Fail".

Translation to TCL/TCLwebtest

After the concretization, we obtain a set of ordered actions, but so far it is still not possible to make a system communicate with the real implementation. To let this happen, the actions must be translated into HTTP post and gets, or into a scripting language enabled to handle this HTTP based communication. As explained in Section 2.4.1, TCLwebtest handles this kind of communications and the logical structures are built in TCL language.

GeneraTCL, following the algorithm 2.4.2 of page 56, generates the TCLwebtest and TCL code using as input the set of concrete actions defined in the concretization process. The instantiated test case of the abstract one (Figure 2.16) is illustrated in Figure 2.17.

2.5.6 Execution of Test Cases

As a result of the execution of the designed test cases on the prototype, we obtained positive verdicts for nine test objectives while four of them were violated (Fail verdict). The list of the test objectives used in this case study is enumerated in Annexe A.

For example: if a potential traveler requests for a first mission and then waits 2 minutes, he is not allowed by the system to request another mission. We analyzed the implementation of the Web-based system and noticed that an error was made in the code. Instead of 2 minutes, the Travel system waited much longer before allowing a second mission request.

The Mission Handler application was analyzed to detect the four sources of error. Once the mistakes were corrected, all the test cases were applied again on the Web application. This time, all the verdicts were Pass which demonstrates the efficiency of our methodology.

2.6 Conclusion

This chapter has presented the complete process to perform active testing over Web applications using model based testing. We have presented some new methodologies and tools that in combination with TestGen-IF, consolidate a complete testing framework for Web applications. The first contribution that was presented is the methodology to build a well formed model of the tested Web application. It is explained how to obtain the model that represents the behavior of the system and how to define the states, inputs, transitions and outputs of the model.

The second contribution presented in this chapter concerns the concretization and translation of the test cases to executable code. This step introduces a complete methodology and the GeneraTCL tool used to render the abstract test cases into concrete test cases that can be executed on a real Web application using HTTP communication.

Finally, it is illustrated how to use the testing framework with a complete real case study, the Mission Handler Web application. The case study section introduces the Mission Handler and illustrates all the steps needed to perform the testing. An overview of the functionalities and results of the testing framework have also been provided.

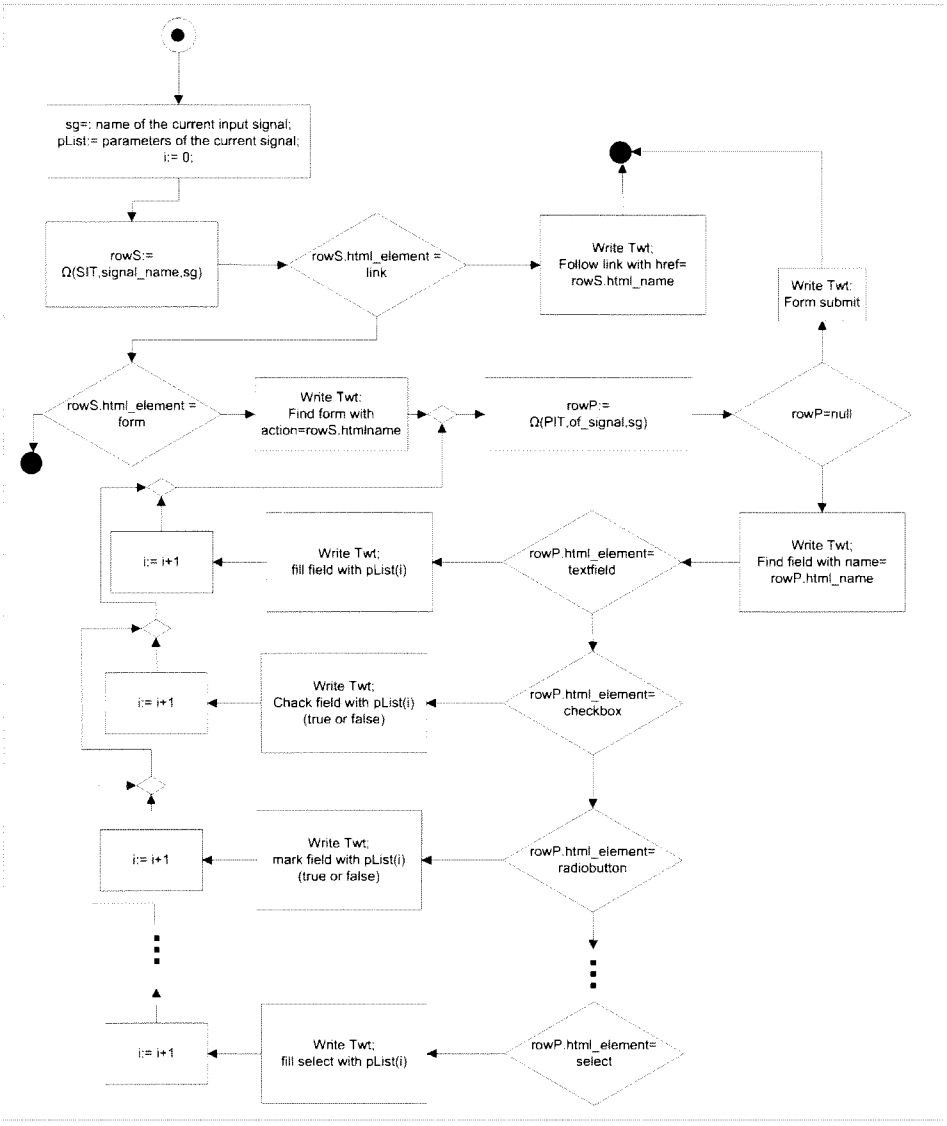


Figure 2.10: Flowchart describing the methodology of the Input instansiation.

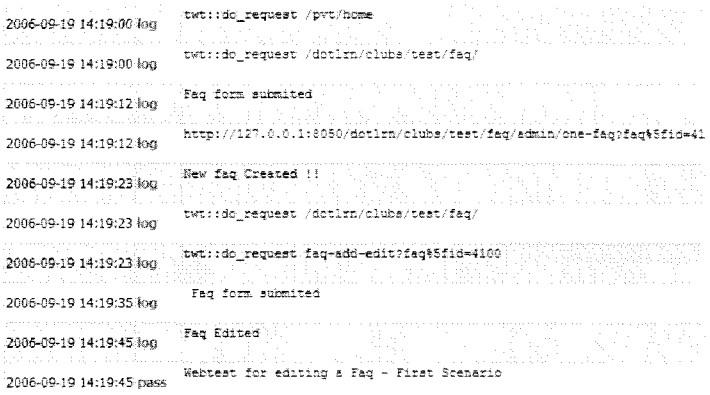


Figure 2.11: Screenshot of the ACS Automated Testing.

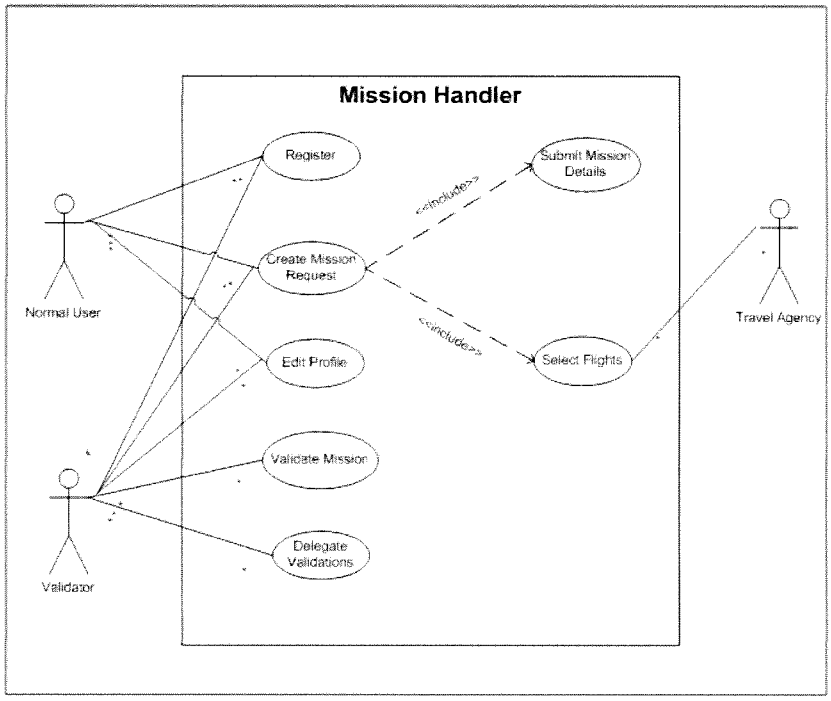


Figure 2.12: Use Case of the Mission Handler.

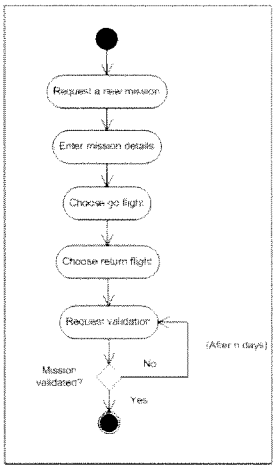


Figure 2.13: Workflow of the creation of a mission.

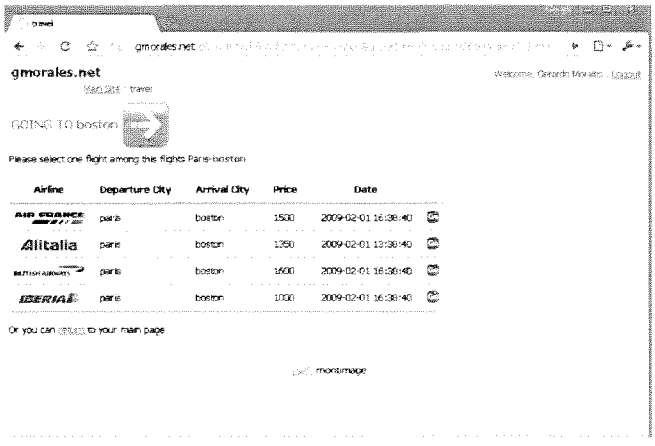


Figure 2.14: Screenshot of the Mission Handler Web application.

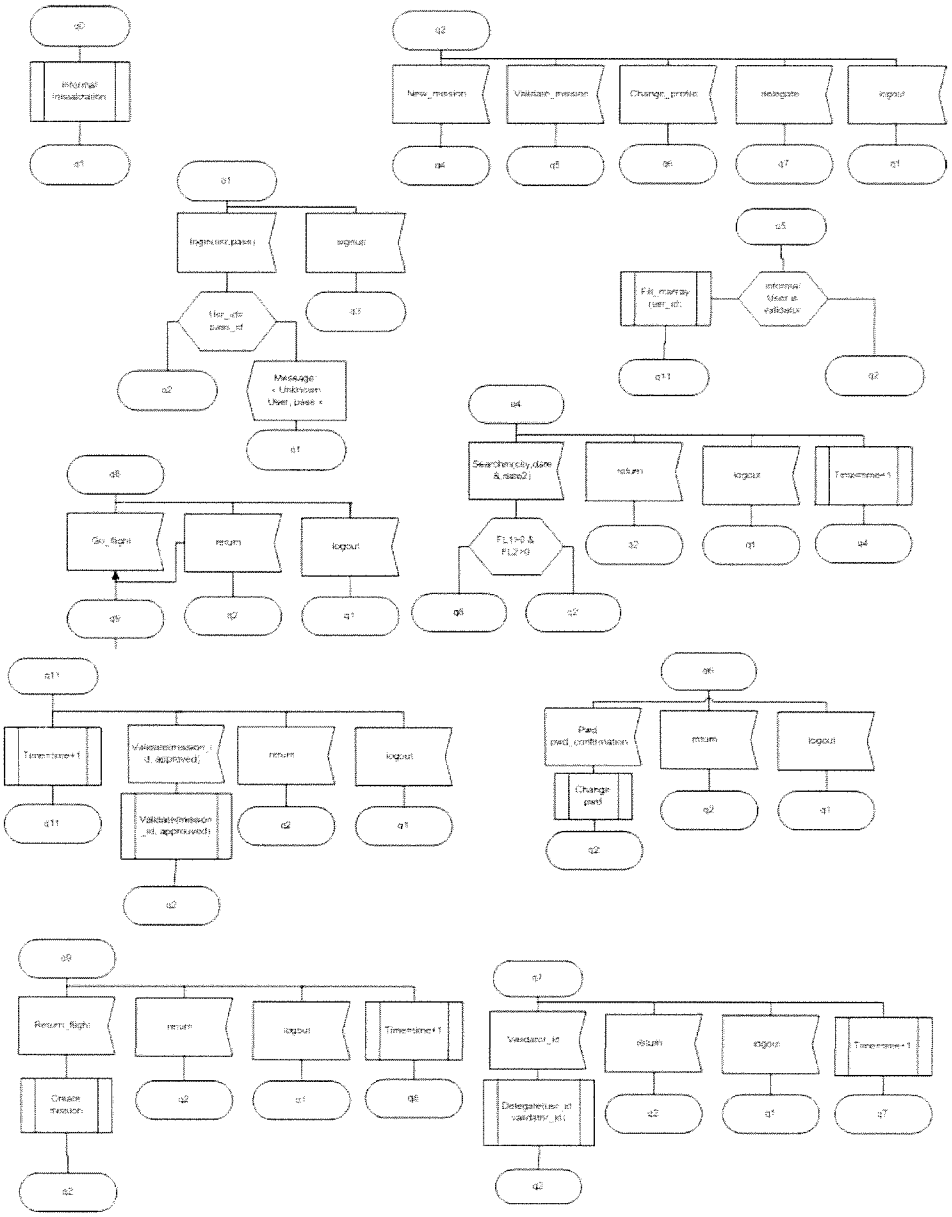


Figure 2.15: Model of the Mission Handler before adding the Outputs

1. {webint}0 i ?login{user,userp} !var2usr{loged_in}
2. {webint}0 i ?new_mission{} !htmchunk{text_to_find_1}
3. {webint}0 i {webint}0 ?search_m{boston,sgd,srd} !urlexpr{url_to_compare_1}
4. {webint}0 i {webint}0 ?go_flight{} !htmchunk {text_to_find_2}
5. {webint}0 i {webint}0 ?return_flight{} {webint}0 !htmchunk{text_to_find_3}
6. delay = 10080
7. {webint}0 i ?new_mission{} !htmchunk{text_to_find_5}

Figure 2.16: Path of the exploration of the test case "New Mission".

```
aa_register_case -cats {web smoke} -libraries TCLwebtest TCLwebtest_new_mission {
Test the creation of a mission
} {
aa_run_teardown -test_code{
set response 1
set verdict "The test case was executed without finding errors"
TCLwebtest::cookies clear
TCLwebtest::do_recuest "http://gmorales.net:8000/travel
TCLwebtest::form find ~a "register"
TCLwebtest::field find ~n "email"
TCLwebtest::field fill "gerardo.morales@montimage.com"
TCLwebtest::field find ~n "password"
TCLwebtest::field fill "b0t3s"
TCLwebtest::form submit
TCLwebtest::link follow ~u "startmission"
TCLwebtest::form find ~a "find-flights"
TCLwebtest::field find ~n "city"
TCLwebtest::field fill "boston"
TCLwebtest::field find ~n "departure-go"
TCLwebtest::field fill "01022009"
TCLwebtest::field find ~n "departure-return"
TCLwebtest::field fill "05022009"
TCLwebtest::form submit
TCLwebtest::link follow ~u "find-flights-2"
TCLwebtest::link follow ~u "createmission"
if {[catch {TCLwebtest::assert text "Created!"} errmsg]} {
aa_error "The text Created!  was not found in the body of the page"
set verdict_msg "Text not found:  Created!"
set response 0
}else{
aa_log "The text Created!  was found"
}
TCLwebtest::user::logout
aa_display_result -response $response -explanation $verdict_msg
}}
}
```

Figure 2.17: Example of TCLwebtest Code generated by GeneraTcl

Chapter 3

Passive testing of Web based systems

Contents

"A few observation and much reasoning
lead to error; many observations and
a little reasoning to truth."
Alexis Carrel

3.1 Introduction

Most of the modern Web applications use services that are provided by external applications. This allows the reuse of existent functions and the division of the "expertise" of the applications. As an example we can take the case study of the previous chapter, the Mission Handler. The communication with the airlines to search and book flights is not performed by the Mission Handler itself, this is done by an external application that is owned by a travel agency. Thanks to this divide-and-conquer approach, the developers can focus on a single problem to solve: "How to handle a mission in the enterprise" and using external services to solve secondary issues as obtaining information of the flights and booking tickets.

In general, the services (as the one that provides the travel agency) can be used by more than one application allowing the re-use of code. It is also important to highlight that the developers of new Web applications can use the know-how of external services and achieve to build complex systems in a reduced time. However, when testing a Web application using a black box approach, the data that we need to collect in order to set a verdict can be enriched if we observe and test more interfaces (not only the user interface). This is possible if we observe also the interface of the Web application used to communicate with other systems through Web services. In other words, by testing also the communications of the Web application with external applications (called services) while testing the user interface, we can obtain an enhanced testing process of the IUT. This approach is illustrated in Figure 3.1.

A Web service is defined by the World Wide Web Consortium W3C as "*a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format* (specifically WSDL [ECW01]). *Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*"

As the Web service implementations grow in size and complexity, the necessity for testing their reliability and level of security are becoming more and more crucial. The Web services technology, which is based on the SOA, is hard to test because it relies on distributed systems that complicate the runtime behavior.

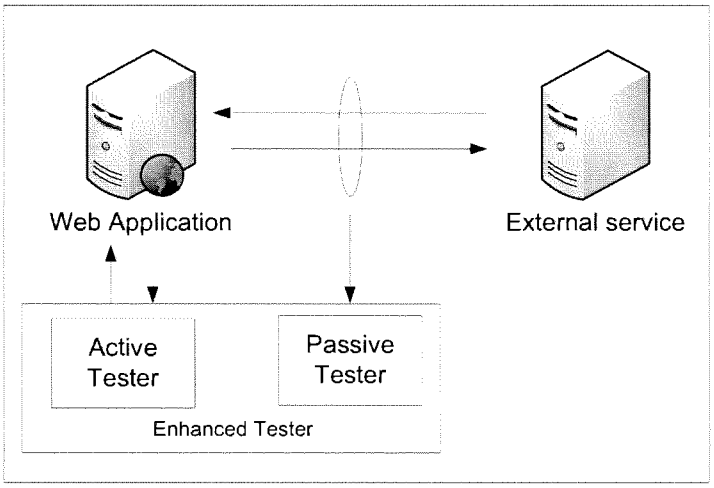


Figure 3.1: Enhanced test combining active and passive testing

In this distributed context, formal active testing can be rather difficult to achieve since it needs to control remote deployed Web systems. Active testing usually relies on the comparison between the behavior of an implementation and its formal specification (which is not necessarily available) by checking whether they are equivalent. Test sequences are commonly automatically or semi-automatically generated from formal models that represent test criteria, hypothesis and test objectives. These sequences (with an executable format) are performed by establishing Points of Control and Observation (PCO, execution interfaces) defined by the testers and may disturb the natural behavior of already deployed Web services.

To avoid this issue, there is an interest in applying passive testing approach for the conformance checking of Web services. Passive testing consists in observing the exchanges of messages (inputs and outputs) of an implementation under test during run-time and then to analyze the traces of messages to detect errors or faults [LNS+97]. The term *passive* means that the tests do not disturb the natural operation of the IUT. The record of the observed packets is called a trace. This will be compared to properties derived from the standard or proposed by the Web service experts.

The passive testing techniques are applied, in particular, because they are non-intrusive whereas the active testing techniques require the set-up of important testing architectures where the testers need to be able to control the implementation at some specific points. This is sometimes not feasible, especially when there is no direct access to the remote Web service partners. We therefore define and apply an invariant-based passive testing approach to validate Web services implementations.

In the work we present here, the concept of invariants already defined in [BCNZ05] is extended to take data and time constraints into consideration. Then we define a methodology with new algorithms for formal passive testing based on these extended invariants. The main contributions of this chapter are the following:

- The timed extended invariants that allow to formally specify functional properties that Web services have to respect are defined. These properties are related to the exchanged packets between systems and may deal with the data portion of these packets and/or with time constraints. The syntax and semantics of these timed extended invariants are presented in this chapter.

- We present our passive conformance testing approach for Web services with respect to their functional requirements. This approach is particularly well adapted when it is difficult to control remote partners, required by active testing, and permits to avoid perturbing the natural running of already deployed Web services.

- We present the experiment analysis and obtained results of applying our passive testing tool called TIPS (stands for Testing Invariants for Protocols and Services) on a travel reservation Web service. This tool performs a dynamic analysis of the execution traces, checking the formal extended invariants. To our knowledge, no other work tackles both the formal invariants that include data and time constraints, and the test of the execution traces according to these invariants, making this approach novel and innovative.

The rest of this chapter is organized as follows. Section 3.2 presents the concept of invariant and especially the syntax and semantics of timed extended invariants. In Section 3.3, we present our passive testing methodology as well as its related tool and algorithm. Section 3.4 presents our experiment on a travel reservation Web service. We apply on it our passive testing tool and analyze the obtained results. In the last section, the conclusions and perspectives are provided.

3.2 Timed extended invariants

3.2.1 Preliminaries

Invariants are properties that the implementation under test expects to satisfy. They are used to express constraints over exchanged messages between the system entities (in this case, remote Web services). Basically, they allow to express that the occurrence of an event must be necessarily preceded or followed by a sequence of events. In [BCNZ05], a first introduction to invariants was provided. These invariants are described according to a finite state machine based specification of the system under test.

Simple invariants

A trace in the format input/output such as i_1/o_1, ..., i_{n-1}/o_{n-1}, i_n/O is a simple invariant for an FSM M if each time that the trace i_1/o_1, ..., i_{n-1}/o_{n-1} is observed, if we obtain the input i_n, then we necessarily get (after this input) an output belonging to O where $O \in \mathcal{O}$. In addition to sequences of input and output symbols, we allow the wildcard characters ?

and $*$. The meaning of ? is the standard one on the pattern matching community (that is to replace any symbol). However, the meaning of $*$ is slightly modified. For example, the intuitive meaning of an invariant as i/o,$*$, i'/O is that if we detect the transition i/o then the first occurrence of the input symbol i' is followed by an output belonging to the set O. In other words, $*$ replaces any sequence of symbols not containing the input symbol i'.

Definition 6 *Formally, a simple invariant Inv is defined according to the following EBNF notation (Extended Backus-Naur Form):*
$Inv ::= i\ /O \mid *,\ Inv \mid a\ /\ z,\ Inv$
Where $i \in I$, $a \in I \cup \{?\}$, $z \in O \cup \{?\}$ *and* $O \in \mathcal{O}$.

Intuitively, the previous EBNF expresses that an invariant is a sequence of symbols where each component, but the last one, is either a pair a/z, with a being an input action or the wildcard ? and z being an output action or the wildcard character ?, or the wildcard character $*$. The last component of an input action followed by a set of output actions.

Example:

The notation of simple invariants allows us to express several interesting properties. For example, we can test that each time that a user requests a disconnection, he is in fact disconnected by using the following invariant:

$$Req\_Disconnect/\{Disconnected\}$$

The idea is that each occurrence of the symbol $Req\_Disconnect$ is followed by the output symbol *Disconnected*. This invariant has the same distinguishing power as the invariant:

$$*,Req\_Disconnect/\{Disconnected\}$$

Obligation Invariants

Obligation invariants may be used to express properties where the occurrence of an event must necessarily be preceded by a sequence of events.

Definition 7 *An obligation invariant is defined according to the following EBNF:*
$Io ::= a/\bar{O}/*,\ Io \mid a/z\ ,Io$
where $a \in I \cup \{?\}$, $z \in O \cup \{?\}$ *and* $O \in \mathcal{O}$.

Let us remark that, in contrast with simple invariants, we do not force the first symbol of the last pair of the obligation invariant to be an input action (it can also be the wildcard character ?).

Example:

The intuitive meaning of an invariant such as:

$$I{=}Req\_Page/Req\_Ack,*,\ ?\ /\ \overline{\{Page\_Sent\}}$$

is that if the event *Page_Sent* is observed in the trace then we must have that a page had been requested before (*Req_Page*) and that the server has acknowledged the reception of this request (*Req_Ack*).

Definitions of Timed Extended Invariants

The definitions of both types of invariants (simple and obligation invariants) described previously rely on an FSM-based model of the IUT. This model is not well adapted to express complex functional behavior of Web services that involve data portion and time constraints. To solve this problem, we propose to extend the formalization of invariants by adding these two aspects (data and time), and rely thus on a specification of the SUT described using a Timed Extended Finite State Machine model ([LZC07, BGO+04]).

Notice that only the system inputs and outputs can be observable in the collected system traces. These actions represent the events that we take into account in the timed extended invariants. Based on the definition of a TEFSM, we adapt the formalization of the messages collected within a trace and the events described within the invariants.

In the following definitions a captured packet is a network packet exchanged between the entities under test (in our case, distributed Web services). This packet is captured by a sniffer that adds some extra data (for example the timestamp of when it was captured), i.e. captured packet = network packet + extra data.

Definition 8 *(Collected trace) A collected trace is a set of ordered captured packets.*

- *A trace* $T = \bigcup_{i=1}^{n} p_i$ *where* n *is the number of the captured packets,* p_1 *is the first packet captured in the trace and* p_n *the last one.*
- *Each packet* p_i *has a rank* r_i *that corresponds to its position in the trace* T*.*
- $\forall p_i \in T$, $p_i = \bigcup_{j=1}^{m_i} f_{i,j}$ *where* $f_{i,j}$ *is a field of the packet* p_i *and* m_i *is the number of fields of the packet* p_i*. Each field* $f_{i,j}$ *of the packet* p_i *has a value* $v_{i,j}$*.*
- $\forall p_i \in T$, $\exists f_{i,j} \in p_i$ / $f_{i,j} = t_i$ *where* t_i *is the timestamp when* p_i *was captured.*
- $\forall r_i$, r_j /r_i *is rank of* p_i *and* r_j *is rank of* p_j*, if* $r_i > r_j$ *then* $t_i > t_j$

Definition 9 *(Value function* ϕ*) Using the elements of Definition 8, let* T *be a collected trace of* n *packets,* F *the set of fields of all the packets* p_i *of the trace* T*,* V *the domain of values and* P *a set of packets.* $V = \mathbb{R} \cup S \cup \{NULL\}$ *where* S *is a finite set of strings. We define the function:* ϕ*:* $P \times F \rightarrow V$ *as the function allowing to provide the value of a field in a specific packet of the trace* T*:*

$$\phi(p_i, f_{m,n}) = \begin{cases} v_{i,n} \text{ if } f_{m,n} \in p_i \\ NULL \text{ if } f_{m,n} \notin p_i \end{cases}$$

3.2.2 Definition of timed extended invariants

An invariant is an if/then property. It allows expressing the desired behavior of a communicating system and describes the correct order of messages collected in the trace with potential time constraints. If specific conditions on the exchanged messages hold, then the occurrence of a set of events must happen. An event is a set of conditions on some field values of exchanged packets.

Definition 10 *(Conditions) Conditions are predicates on packets' fields values. Let* p_i *and* $p_i\prime$ *be two captured packets,* V *be the domain of values,* $f_{i,j}$ *be a field of the packet* p_i*,* $f_{i\prime,j\prime}$ *be a field of* $p_i\prime$ *and* $x \in V$*. We say that a condition can be defined according to* c_s *(simple condition) or to* c_c *(complex condition)*

$$c_s ::= \phi(p_i, f_{i,j}) \ op \ x$$

We say that the packet p_i *satisfies* c_s *iff* $\phi(p_i, f_{i,j}) \ op \ x$ *is true.*

$$c_c ::= \phi(p_i, f_i) \ op \ \phi(p_i\prime, f_{i\prime,j\prime})$$

We say that packet p_i *satisfies* c_c *iff*

$$\exists p_i\prime \in T, \exists f_{i\prime,j\prime} \in p_i\prime / \ \phi(p_i, f_j) \ op \ \phi(p_i\prime, f_{i\prime,j\prime}) \text{ is true.}$$

op is an element of $O_T = O_R \cup O_S$ *where* $O_R = \{\leq, \geq, =, \neq, \in\}$ *and* $O_S = \{$*contain, not contain*$\}$*.*
The operator must respect these rules:

- *If* $x \in \mathbb{R}$ *then* $op \in O_R$*.*
- *If* $x \in S$ *then* $op \in O_T$ *and:*
  - *the operators that are in* O_R *manages the elements in* V *with lexicographic priority*
  - *the operators that are in* O_S *are used when a field's value has a string type and may include or not a set of characters or words.*

Definition 11 *(Basic event) A timed extended event* e_j *is a set of conditions on relevant fields of captured packets.* $e_j = \bigcup_{k=1}^{m_j} c_{j,k}$*,* m_j *being the number of conditions.*

Definition 12 *(Event satisfaction) Let* p_i *be a packet and* e_j *an event with* m_j *conditions and* $c_{j,k}$ *the* k^{th} *condition of* e_j

A packet p_i *satisfies an event* e_j *iff*
$\forall \ k \in [1, m_j]$*,* $c_{j,k}$ *is true*

A packet can satisfy a set of events (may be empty). In this case, we say that this

packet is the instantiation of the event.

Definition 13 *(Abstention of having an event)*
If e is an event, then $\neg e$ is also an event. $\neg e$ is satisfied if there is no packet in the collected trace that satisfies the event e.

Definition 14 *(Complex Events: Follows) Let $n \in \mathbb{N}^* \cup \{-1\}$, $t \in \mathbb{R}^{+*} \cup \{-1\}$ and e_1 and e_2 be two basic events. Let $\{p_1 ... p_2\}$ be an ordered set of packets where p_1 is the first packet and p_2 the last one. $(e_1; e_2)_{n,t}$ is a complex event denoted also by Follows((e_1, e_2),n,t). It is composed of two basic events.*
$\{p_1, ..., p_2\}$ satisfies $(e_1; e_2)_{n,t}$ $\Leftrightarrow$

- *p_1 satisfies e_1 and*
- *p_2 satisfies e_2 and*
- *time(p_1) < time(p_2) < time(p_1)+t if ($t \neq -1$) and*
- *rank(p_1) < rank(p_2) < rank(p_1)+n if ($n \neq -1$).*

In other words, $\{p_1, ... , p_2\}$ satisfies $(e_1; e_2)_{n,t}$ iff p_2 follows p_1 and they are separated by at most n packets and t units of time.

Definition 15 *(Complex Events: Preceede) Let $n \in \mathbb{N}^* \cup \{-1\}$, $t \in \mathbb{R}^{+*} \cup \{-1\}$ and e_1 and e_2 be two basic events. Let $p_1 ... p-2$ be an ordered set of packets where p_1 is the first packat and p_2 the last one. $(e_1 \bar{;} e_2)_{n,t}$ is a complex event also denoted by Precede($(e_1 \bar{;} e_2)$,n,t). It is composed of two basic events.*
$\{p_1, ..., p_2\}$ satisfies $(e_1 \bar{;} e_2)_{n,t}$ $\Leftrightarrow$

- *p_1 satisfies e_1 and*
- *p_2 satisfies e_2 and*
- *time(p_1)-t < time(p_2) < time(p_1) if ($t \neq -1$) and*
- *rank(p_1)-n < rank(p_2) < rank(p_1) if ($n \neq -1$).*

In other words, $\{p_1, p_2\}$ satisfies $(e_1 \bar{;} e_2)_{n,t}$ iff p_2 precedes p_1 and they are separated by at most n packets and t units of time.

Definition 16 *(Complex Events: AND) Let $n \in \mathbb{N}^* \cup \{-1\}$, $t \in \mathbb{R}^{+*} \cup \{-1\}$ and e_1 and e_2 two basic events.Let $p_1 ... p-2$ be an ordered set of packets where p_1 is the first packat and p_2 the last one. $(e_1 \wedge e_2)_{n,t}$ is a complex event denoted also by AND((e_1,e_2),n,t). It is composed of two basic events.*
$\{p_1,...,p_2\}$ satisfies $(e_1 \wedge e_2)_{n,t}$ $\Leftrightarrow$
$\{p_1,...,p_2\}$ satisfies $(e_1; e_2)_{n,t}$ or $\{p_1,...,p_2\}$ satisfies $(e_2; e_1)_{n,t}$

Definition 17 *(Complex Events: OR) Let* $n \in \mathbb{N}^* \cup \{-1\}$, $t \in \mathbb{R}^{+*} \cup \{-1\}$ *and* e_1 *and* e_2 *two basic events.* $(e_1 \vee e_2)_{n,t}$ *is a complex event denoted also by OR((*e_1*,*e_2*),n,t). It is composed of two basic events.*
p_1 *satisfies* $(e_1 \vee e_2)_{n,t} \Leftrightarrow p_1$ *satisfies* e_1 *or* p_1 *satisfies* e_2

Definition 18 *(Timed Extended Invariant) Let* $When \in \{Before, After\}$, $n \in \mathbb{N}^* \cup \{-1\}$, $t \in \mathbb{R}^{+*} \cup \{-1\}$ *and* e_1 *and* e_2 *two events (basic or not). A timed extended invariant is an IF-THEN expression that allows expressing a property regarding the messages exchanged in a captured trace* $P = \{p_1, ..., p_{n\prime}\}$. *It has the following syntax:*

$$e_1 \xrightarrow{When,n,t} e_2$$

This property expresses that if the event e_1 *is satisfied (by one or several packets* p_i, $i \in \{1, ..., n\prime\}$*), then event* e_2 *must be satisfied (by another set of packets* p_j, $j \in \{1, ..., n\prime\}$*) before or after (depending on the When value) at most n packets if* $n \neq -1$ *and t units of time if* $t \neq -1$.

It is important to highlight that in the invariant it is possible to disable n and t by setting its value to -1.

If $(When = AFTER)$, we say that the invariant is a *simple invariant.* Otherwise, if $(When = BEFORE)$, we have an *obligation invariant.*

Example:

According to this last definition, invariants can express complex properties including data and time constraints. We can for instance express for the HTTP protocol that if an OK message (the code 200) is received, then a POST request must have been received before. This is described by the following obligation invariant:

$$\begin{gathered} e_1 = \{\phi(p_1, http.response.code) = 200, \\ \phi(p_1, tcp.srcport) = 80, \\ \phi(p_1, tcp.dtsport) = 1490\} \\ \xrightarrow{BEFORE,-1,6} \\ e_2 = \{\phi(p_2), http.request.method) = POST, \\ \phi(p_2, tcp.srcport) = \phi(p_1, tcp.dtsport), \\ \phi(p_2, tcp.dtsport) = \phi(p_1, srcport)\} \end{gathered}$$

Notice that in this invariant, we do not mention the number of packets that are between the OK and the POST messages, on the other hand, the delay of time between the messages must be 6 seconds.

3.3 Conformance passive testing approach for Web services

3.3.1 Methodology overview

Conformance testing in the passive testing approach aims to test the correctness of a Web service implementation through a set of properties (invariants) and monitored traces (extracted from the running Web service through probes called also Points of Observation (PO)). The conformance passive testing procedure follows these four steps:

- Step 1: Properties formulation. The relevant Web service properties to be tested are provided by the standards or by Web service experts. These properties express the main requirements related to communication between the Web partners in the context of Web services.
- Step 2: Properties as invariants. Properties have to be formulated by means of invariants that express local properties; i.e. related to a local entity. Moreover, the properties could be formally verified on the formal system specification (as a timed automata) if it is available, ensuring that they are correct vis-à-vis the requirements.
- Step 3: Extraction of execution traces. In order to obtain such traces, different PO are set up by means of a network sniffer installed on one of the servers. The captured traces are stored in an XML format. The PO must be set in the network used to exchange messages between the systems, for example, if our Web application communicates with 3 different external services, the PO needs to be in the Network Interface Card where the Web application is running.
- Step 4: Test of the invariants on the traces. The traces are processed in order to obtain information concerning particular events as well as relevant data (e.g. source and destination address, origin of data to initialize a variable, etc.). During this process, the test of the expected properties is performed and a verdict is emitted (Pass, Fail or Inconclusive). An inconclusive verdict may be obtained if the trace does not contain enough information to allow a Pass or Fail verdict. To reduce this type of verdicts, the trace should include the state of initiation; this is possible if the sniffer is launched before the exchange of packets between the systems under testing.

3.3.2 Passive testing tool

Based on the methodology of our work, we developed the TIPS tool [COML08] in collaboration with Montimage[1] and inspired from an early prototype developed by IT[2]. The TIPS tool aims at passively testing a deployed communicating system under test to verify if it

[1]Montimage is a French SME working on the field of formal testing. http://www.montimage.com

[2]Institut TELECOM is a group of French engineering schools in the telecommunication field. http://www.institut-telecom.fr/

respects a set of properties. In the case of TIPS, invariants describe the correct order of exchanged messages among system entities with conditions on communicated data and time. Passive testing consists in observing input and output events of the system implementation in run-time and detecting potential misbehaviors or errors.

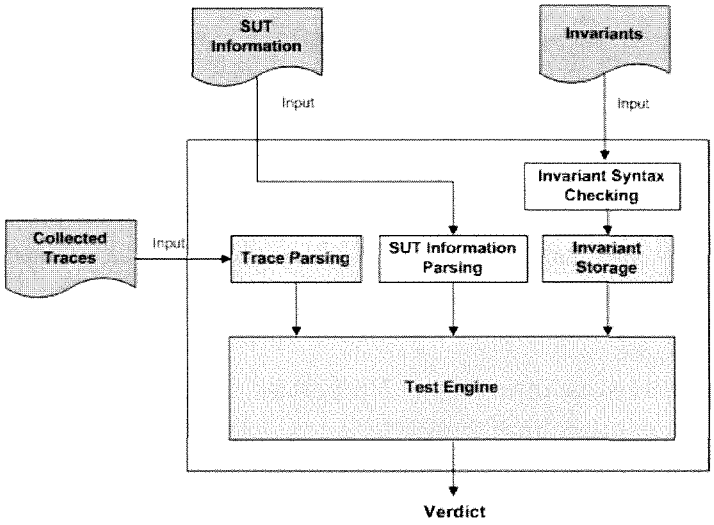


Figure 3.2: TIPS architecture for Web service checking

Figure 3.2 illustrates the components of the TIPS tool. It has three different inputs:

- Information on the Web service under test that is being observed. This information represents data of interest (for example protocol packets field names) that are relevant to the automated analysis of the captured traces.
- The invariants defined in XML format. By invariants we mean here the combination of conditions that must be respected by the system. The non-respect of an invariant may imply an error in the Web services.
- And the communication traces represented in XML format (captured using Wireshark[3] for instance).

In order to use the TIPS tool, the first step consists in defining the invariants and the data of interest. This can be done by an expert of the Web service under test that understands in detail the studied service. The invariants are then verified with respect to their XML schema.

The next step consists in capturing the communication traces using a sniffer (we used Wireshark) and analyzing them using the TIPS tool.

In the case of Web services, what we analyze are SOAP protocol packets. A SOAP request is an XML-based Remote Procedure Call (RPC) sent using the HTTP transport

[3]http://www.wireshark.org/

protocol. The payload of the SOAP packet is an XML document that specifies the call being made and the parameters being passed.

Within the body of a SOAP message, XML namespaces are used to qualify element and attribute names within the parts of the document. Element names can be global (referenced throughout the SOAP message) or local. The local element names are provided by namespaces and are used in the particular part of the message where they are located. Thus, SOAP messages use namespaces to qualify element names in the separate parts of a message. Namespaces also identify the SOAP envelope version and encoding style.

3.3.3 Trace collection in XML

PDML introduction

Wireshark can be used to capture protocol packets and convert them to an XML format called Packet Details Markup Language (PDML) specified by the 'Politecnico Di Torino' [pro09]. The PDML specification is a way to organize this information list all the packets within a capture file, detailing the most important information for every protocol that has been found into the packet and for each field that has been decoded within the protocol.

The PDML document is delimited by the <pdml> and </pdml> tags. This tag contains a set of <packet> elements, which contains the decoded packet. Both <pdml> and <packet> do not have attributes. An example of PDML is illustrated in Figure 3.3

As in a packet we can have one or more used protocols, a <packet> element can have one or more <proto> elements. The <proto> element keeps the name of one protocol. A <proto> element must have one or more <fields> child elements nested to include all the data that is exchanged using this protocol.

PDML trace

In <proto> we have as an attribute the name of the protocol, when the value of this attribute is "geninfo", it exist the field "timestamp" (<field name=timestamp .. >) , among the nested fields of the protocol providing the time when the packet was sniffed by Wireshark. This value is used to specify invariants with time constraints. We should note here that this time can be considered as the real emission/reception time of the sniffed packets since the sniffer is usually installed in the same local network of the IUT.

Another possible name of <proto> is "ip", in this protocol it exists the fields "ip.src" and "ip.dst" that give the source and destination ip addresses of the message. This is useful to determine who is sending a SOAP message and who is the destination. In some cases it is also necessary to use the source and destination ports given by the "tcp" protocol, element that contains the "tcp.srcport" and "tcp.dstport" fields.

In the case of a request, the <proto> of name "http" contains the following fields:

- "http.request.method": that allows determining the the request method (POST or GET)

- "http.request.uri": that allows determining the WSDL description used

and in the case of a reply:

- "http.response.code": that allows determining that it is an OK reply

Finally, the <proto name="xml"> element contains the fields "xml.tag" and "xml.cdata" corresponding to the request data sent by the client of the Web service or the reply data sent from the server implementing the Web service to the client that made the request.

```
<packet>
 <proto name="http" showname="Hypertext Transfer Protocol" size="642" pos="68">
 <field name="" show="POST /webservice/VehicleReservationService HTTP/1.1" size="53" pos="68"
 value="504f...">
 <field name="http.request.method" showname="Request Method:  POST" size="4" pos="68" show="POST"
 value="504f5354"/>
 <field name="http.request.uri" showname="Request URI: /webservice/VehicleReservationService" size="37"
 pos="73" show="/webservice/VehicleReservationService" value="2f776..."/>
 <field name="" show="\r\n" size="2" pos="708" value="0d0a"/>
 <field name="http.request" showname="Request:  True" hide="yes" size="0" pos="68" show="1"/>
 </proto>
</packet>
```

Figure 3.3: PDML structure

3.3.4 Invariants Storage

The invariants are written in XML format and are represented in an If/then structure presented in Figure 3.4. Basicaly, if in the trace we find the events of the Trigger Context, then we need to verify that the events of the Verdict Clause are part of the trace as well.

```
<if>
 Trigger Context
</if>
<then>
 Verdict Clause
</then>
```

Figure 3.4: Invariant storage

Within an invariant, the <if> tag identifies the triggering events, we call this part of the invariant the *Trigger Context*. The events that need to be verified on the trace are found in the <then> tag, we call this part of the invariant the *Verdict Clause*.

The invariant in Figure 3.5 expresses that if a system receives a reply message, then this means that it sent a request message 10 seconds before. In this invariant we have the *Trigger Context* from lines 5 to 15, this means that if in the collected trace, we find all the events described from lines 6 to 14, the process of verifying the invariant over the trace is triggered.

In the line 6 an event is defined, it is the reference event and the id of the event is the 001. Then, lines 7 to 11 defines the first condition of the event e001 expresses the reception

```
1.  ...
2.
3.  <invariant name="INVARIANT 1">
4.  <!- Trigger Context start here ->
5.  <if>
6.  <event reference= "true" id= "e001">
7.  <condition>
8.  <variable>Type</variable>
9.  <operation>=</operation>
10.  <value>REPLY</value>
11.  </condition>
12.
13.  ...
14.  </event>
15.  </if>
16.  <!- Trigger Context end here ->
17.  <!- Verdict Clause start here ->
18.  <then type="BEFORE" max_skip="-1" max_time="10">
19.  <event id= "032">
20.  <condition>
21.  <variable>Type</variable>
22.  <operation>=</operation>
23.  <variable>REQ</value>
24.  </condition>
25.  <condition>
26.  <variable>Source</variable>
27.  <operation>=</operation>
28.  <variable type=="e001">Destination</value>
29.  </condition>
30.
31.  ...
32.  </event>
33.  </then>
34.  <!- Verdict Clause end here ->
35.  </invariant>
36.
37.  ...
```

Figure 3.5: XML format for defining an invariant

of a reply message. Then the *Verdict Clause* (lines 18 to 33) contains the events that will be searched in the trace once the verification process is triggered.

Among the conditions of the events in the *Verdict Clause* we can mention from lines 25 to 29 a condition expressing that the source of the received message must be equal to the destination of the message described in the event e001.

Values <val> can be strings or numbers and operations can be the ones defined in Section 3.2.

3.3.5 Passive testing algorithm

This section presents the algorithm that allows the deduction of a verdict by analyzing the system trace with respect to a set of predefined obligation invariants of type "BEFORE". The obtained verdict for an invariant can be either: Pass, Fail or Inconclusive meaning respectively that all events where satisfied, that at least one event was not satisfied or that it is not possible to give a verdict due to the lack of information in the trace.

The algorithm used by TIPS analyzes the traces in, at most, time complexity of $O(N^2)$ or to be more precise: the number of packets that need to be analyzed is N^2 x I x T where N = the number of packets in the trace, I = the number of invariants and T = the average time spent in analyzing an invariant on a packet. This can be reduced to N x I x T if we store information of each condition for each packet in a hash table making it necessary to inspect each packet only once. This will be done for the future version of the tool that will thus be able to perform correctly for on-line invariant analysis.

The behavior of the obligation algorithm (Algorithm 3.3.5 on page 86) is illustrated (as an abstraction) by the flowchart of Figure 3.6. In this flowchart we consider the trace T $=\{p_1 ... p_k\}$. The pointers Current packet (CrPkt) and Reference packet (RefPkt) are used to cover T. Note that as the obligation invariant uses the logic "if a happens then b should happens before a", the trace is then covered backwards starting from p_k and finishing in p_1.

In this algorithm, the *Trigger Context* of the invariant is satisfied when the tester (TIPS) detects the packet containing the action a. After this, it will seek for b, if it founds the action b, the Verdict Clause is satisfied and then the verdict is Pass, otherwise it can be Fail or Inconclusive.

Before explaining the algorithm behavior in details we need to define the functions *next* and *previous* that let us cover the analyzed trace.

Definition 19 *(Next function) Using the elements of Definition 8, let r_i be the rank of the packet p_i in the trace T. We define the function:*

$$\omega(p_i) = \begin{cases} r_j \text{ where } j = i+1 \text{ if } p_j \in T \\ NULL \text{ if } p_j \notin T \end{cases}$$

Definition 20 *(Previous function) Using the elements of Definition 8, let r_i be the rank of the packet p_i in the trace T. We define the function:*

$$\alpha(p_i) = \begin{cases} r_j \text{ where } j = i\text{-}1 \text{ if } p_j \in T \\ NULL \text{ if } p_j \notin T \end{cases}$$

Definition 21 *(Affect function) Using the elements of Definition 8, let r_i be the rank of the packet p_i in the trace T. We define the function: $\sigma(r_i) = p_i$*

$$\sigma(r_i) = \begin{cases} p_i \text{ if } p_i \in T \\ NULL \text{ if } p_i \notin T \end{cases}$$

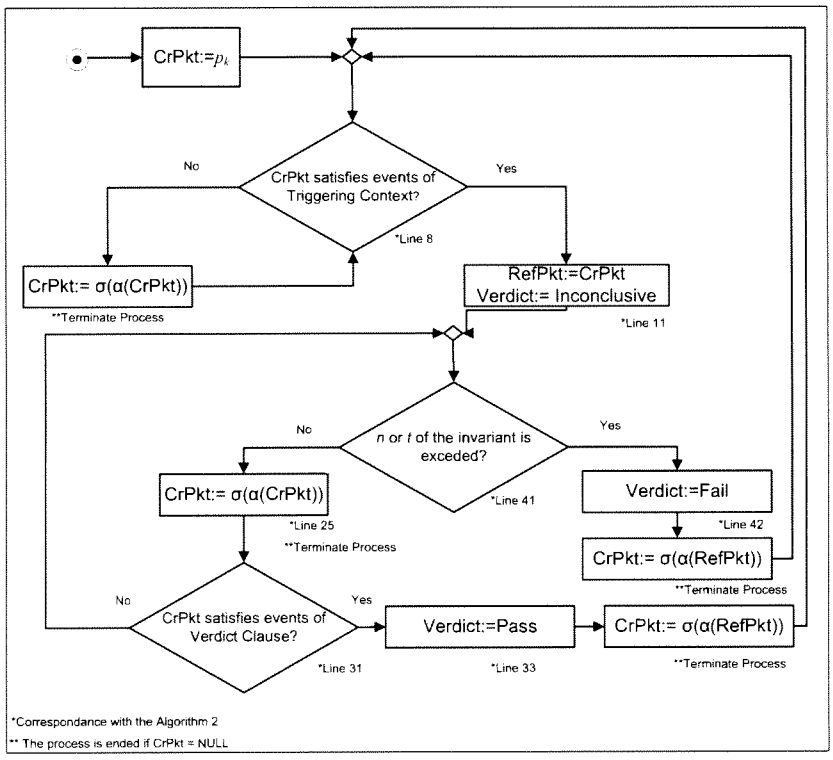


Figure 3.6: Abstraction of the obligation algorithm

The algorithm that allows checking simple invariants on a captured trace is a variant of this proposed algorithm. The difference is located in the way of parsing the trace. In the case of simple invariants, we begin to parse from the beginning of the file and move forward until the end of the trace.

Algorithm 3 TIPS Obligation invariant analysis

Require: INPUT: Invariants (I_1,...,I_k), trace (P_1,...,P_N). Each invariant I_i is a set of events denoted ($e_{\{i,R\}}$,$e_{\{i,1\}}$, ... $e_{\{i,n(i)\}}$) where $e\{i,R\}$ is the triggering event, (ei, 1, ..., $e_{\{i,n(i)\}}$) are the events that need to be satisfied before $e_{\{i,R\}}$ and $n(i)$ is the number of these events.
OUTPUT: Verdicts (V_1,...,V_k)

```
1:  for each packet P_t in trace (from t=N to t=1) do
2:     /*the algorithm parses the trace from the end */
3:     r = P_t
4:     /*r is the reference packet */
5:     set all current event of invariants E_1, ..., E_k to first event e_1
6:     for invariant I_i(from i=1 to i=k) do
7:        V_i = skip
8:        if (r satisfies e_{i,R}) then
9:           /* r satisfies an event e if all the conditions of e are satisfied
10:          by the packet r */
11:          V_i = INCONCLUSIVE
12:          current I_i= e_{i,1}
13:          /* current Ii indicates the event that needs to be inspected
14:          at a given moment (this means that, for this invariant,
15:          the events before have been satisfied,
16:          the others not yet) */
17:       end if
18:    end for
19:    for each invariant I_i (from i=1 to k) do
20:       if V_i = INCONCLUSIVE then
21:          continue
22:       end if
23:    end for
24:    for each P_{t-s} (from s=1 to s=t-1) do
25:       c=P_{t-s}
26:       /* c is the current packet */
27:       for each invariant I_i (from i=1 to i=k) do
28:          if (V_i= PASSED) or (V_i= FAILED) or (V_i= SKIP) then
29:             continue
30:          end if
31:          if (c satisfies current I_i) then
32:             if (current I_i = e_{i,n(i)}) then
33:                V_i = PASSED
34:                continue
35:             else
36:                current I_i = next.current I_i
37:                /* current Ii has been satisfied so the next time
38:                around we need to inspect the next
39:                event for this invariant */
40:             end if
41:          else if ((timestamp(c) - timestamp(r)) > max_time) or (rank(c) - rank(r)) > max_skip) then
42:             V_i = Failed
43:             continue
44:          end if
45:       end for
46:    end for
47:    for each invariant I_i (from i=1 to i=k) do
48:       if V_i neq SKIP then
49:          print V_i and information about r
50:       end if
51:    end for
52: end for
```

3.4 Case Study

3.4.1 Travel Reservation Service description

The Travel Reservation Service (TRS) is an example of real-life service for travel organization provided within the Netbeans IDE 6.5.1 platform [Net09]. It acts as a logical

aggregator of three other Web services:

- Airline Reservation Service (ARS)
- Hotel Reservation Service (HRS)
- Hotel Reservation Service (VRS)

The TRS (illustrated in Figure 3.7) is a system based in the Business Process of the Service Oriented Architecture.

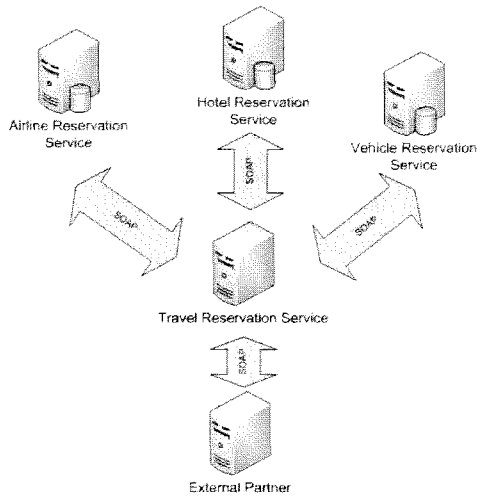


Figure 3.7: Communication of the TRS with its Web services partners

- The process assumes that an External Partner initiates the process by sending a message that contains a partial travel itinerary document.
- The client's travel itinerary may have:
  - No pre-existing reservations, or
  - A combination of pre-existing airline, vehicle and/or hotel reservations.
- The TRS examines the incoming client itinerary and processes it for completion.
- If the client itinerary does not contain a pre-existing airline reservation, the TRS passes the itinerary to the ARS, to add the airline reservation.
- The ARS passes back the modified itinerary to the TRS.
- The TRS conducts similar logic for both vehicle and hotel reservations. In each case it will delegate the actual provisioning of the reservation to the VRS and HRS.

- Finally, the TRS passes the completed itinerary back to the original client, completing the process.

As illustrated in Figure 3.7, the TRS is stimulated by an External Partner system which is the only actor that interacts with the Travel Reservation Service. The External Partner system can be exploited by a human client through, for instance, a Web application.

The TRS orchestrates the different partner services. Accordingly, the external partner side functionality of building a travel itinerary leads to a process workflow. This process workflow is illustrated in the UML use case diagram of Figure 3.8

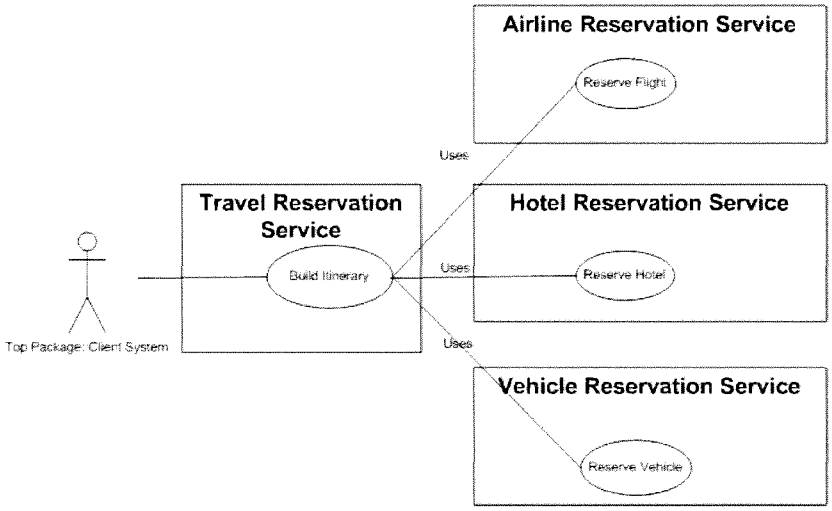


Figure 3.8: Use case diagram of the TRS

The *Build Itinerary* function allows the External Partner to build an itinerary for his traveling needs. This partner initiates the process by sending a message that contains a partial travel itinerary document. The Travel system receives the itinerary request and contacts its service partners (ARS, VRS and HRS) in the case, respectively, an airline, vehicle or hotel reservation is needed. After the reservations are done, the system sends back the completed itinerary to the External Partner.

3.4.2 TRS dynamic view

The dynamic view of the system illustrate how the architectural elements defined in the static view interact with each other. The behavior of the TRS and the communications with the partner Web services are described with the Business Process Modeling Notation (BPMN) [OAS09] notation in Figure 3.9.

3.4.3 Invariants specification

Starting from the TRS requirements, a set of 12 test objectives (TO) has been defined. These objectives are related to the order of the Web partners' interactions or to data or

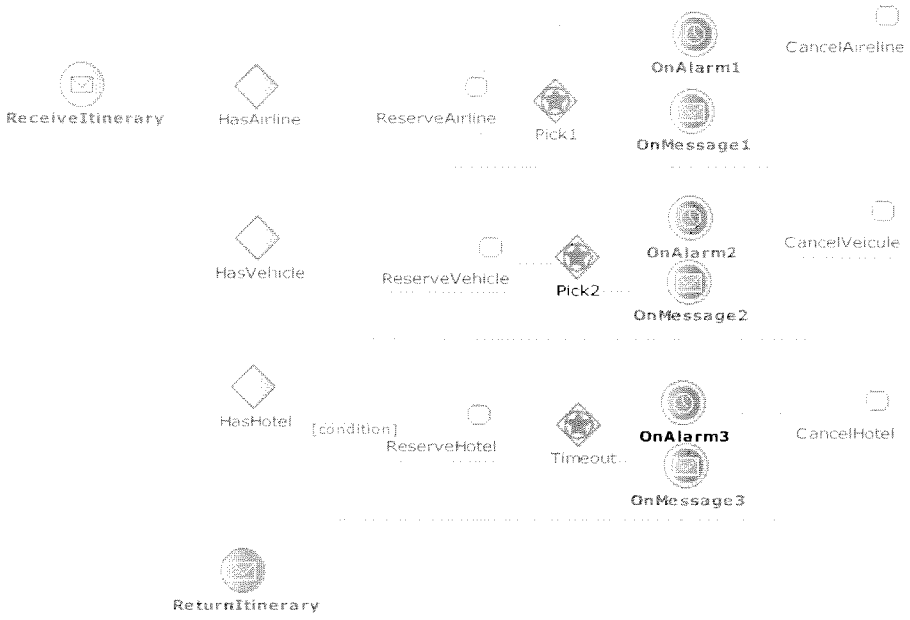


Figure 3.9: Behavior of the TRS described in BPMN

time constraints. Here we present a selection of 3 test objectives:

- TO1. A partial itinerary is sent to the TRS. It does not contain any previous reservation (no airline, no vehicle and no hotel). The TRS system contacts the airline reservation service partner, then the VRS partner, then the hotel reservation service partner, before sending back the complete itinerary.

- TO7. A partial itinerary is sent to the TRS. It already has a vehicle reservation and a hotel reservation but no airline reservation. The TRS system contacts the airline reservation service partner, before sending back the complete itinerary.

- TO9. If the TRS receives a message with an itinerary from VRS, the TRS should have contacted the VRS asking for a vehicle reservation.

- TO12. The process does not receive any response from the HRS within 20 seconds, a request for canceling the reservation is sent to this partner. No hotel item is included in the completed reservation.

The 12 test objectives are in the Annexe B of this work. These test objectives are specified according to the syntax and semantics of the Timed Extended Invariants defined in Section 3.2. Figure 3.10 represents a basic obligation invariant for the test objective TO9, which expresses that if the vehicle reservation service sends an itinerary with a

```
1:  <invariant name="INV1:Vehicle Request/Reply">
2:  <if>
3:  <event reference="TRUE" verdict="TRUE">
4:  <condition>
5:  <variable>tcp.srcport</variable>
6:  <operation>=</operation>
7:  <value>36550</value>
8:  </condition>
9:  <condition>
10:  <variable>tcp.dstport</variable>
11:  <operation>=</operation>
12:  <value>18181</value>
13:  </condition>
14:  <condition>
15:  <variable>data</variable>
16:  <operation>contains</operation>
17:  <value>"Vehicle"</value>
18:  </condition>
19:  </event>
20:</if>
21:<then type="BEFORE" max_skip="-1" max_time="10">
22:  <event verdict="TRUE">
23:  <condition>
24:  <variable>tcp.dstport</variable>
25:  <operation>=</operation>
26:  <value>8080</value>
27:  </condition>
28:  <condition>
29:  <variable>data</variable>
30:  <operation>contains</operation>
31:  <value>Itinerary</value>
32:  </condition>
33:  </event>
34:  </then>
35:</invariant>
```

Figure 3.10: Invariant for the test objective 9

vehicle reservation (the Trigger Context from lines 2 to 20), this means that a request for that has been previously performed (the verdict clause from lines 21 to 34).

Note that the TCP port numbers are configuration data that is defined by the user when deploying the Web services. Note also that for simplicity, these Web services were deployed on only one machine. Normally, it is needed to indicate the IP address to differentiate Web services.

3.4.4 Trace Capture

In this section, the demonstration of TIPS is applied to the travel reservation Web service. The traces of the composed Web service have been captured running the implementation as a black box through the PO installed in the TRS server to capture the communications with the external systems (HRS, VRS and ARS). Wireshark was used to save the traces in XML format. The main advantage of using the selected PO is that it allowed observing all exchanges between the composed Web service and its partners. Several itineraries have been simulated according to different scenarios and different traces have been extracted. In the Figure 3.11, a selection from the collected trace file is provided.

```
...
<packet>
 <proto name="geninfo" pos="0" showname="General information" size="6523">
 <field name="num" pos="0" show="30" showname="Number" value="1e" size="6523"/>
 <field name="len" pos="0" show="6523" showname="Packet Length" value="197b" size="6523"/>
 <field name="caplen" pos="0" show="6523" showname="Captured Length" value="197b" size="6523"/>
 <field name="timestamp" pos="0" show="Jul 24, 2009 18:36:38.128050000" showname="Captured Time"
 value="1248453398.128050000" size="6523"/>
 </proto>
 <proto name="frame" showname="Frame 30 (6523 bytes on wire, 6523 bytes captured)" size="6523" pos="0">
 ...
 </proto>
...
</packet>
```

Figure 3.11: Travel reservation service trace in XML format (a selection only)

Data of interest are defined by an expert of the SUT and are stored in "trace-format.properties" file illustrated in Table 3.1

Table 3.1: Data of interest for Travel Reservation Service

Field name	attribute of interest
timestamp	show
tcp.srcport	show
data	value

3.4.5 Results and analysis

Applying TIPS to the traces is rather fast (less than 1 sec. for a trace of 1000 packets). Some of the obtained verdicts were PASS and INCONCLUSIVE. Some of the invariants

were never tested since they were not found in the collected traces. Indeed, the collection of a trace from a running system should be long enough to be sure to cover all the Web service scenarios. In other cases, a simulation of the scenarios must be made. For example, to be able to test the 11th test objective, the hotel reservation service must not answer the itinerary request (or answer it 20 seconds after it receives the request). This kind of situation is not very frequent in a real system deployment and to test it (i.e test the invariant describing this test), we would need to wait for a sufficiently long time or we can simulate the non answer by shutting down the hotel Web service. The complexity of TIPS ($O(N^2)$) is illustrated in Figure 3.12 as a function depending on the length of the trace.

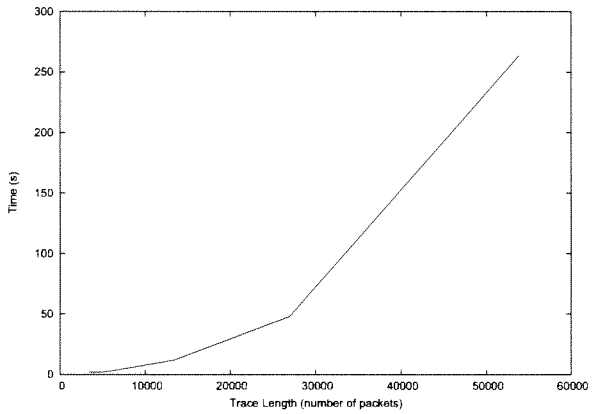


Figure 3.12: Processing time as a function of the length of the trace

During this experiment we finally tested the invariants at least once. The obtained verdicts were all PASS which illustrates the correctness of the Web service.

To prove the efficiency of our TIPS passive testing tool, we manually edited the file containing traces of travel reservation Web service. We have, for instance, deleted a packet that violates the invariant number 1. The verdict given by the tester was FAIL: the violation of the invariant has been detected (and provides the packet characteristics that violates the invariant) which indicates the correctness of TIPS.

3.5 Conclusion

This chapter has explained the importance of performing tests to assure the good interoperability of Web applications and its Web partners. It is also explained why an interesting way to tackle the testing of Web services is through passive testing, especially when they are already deployed or are closed implementations.

It has also presented the third contribution of this work, the presentation of a new approach for testing the collaboration between two systems, focusing in particular on the communication between SOA partners. We have proposed and evaluated an extensible and more flexible approach where properties called timed extended are checked on a collected trace.

Nowadays TIPS uses a collected trace to perform the testing, however as further work it, could be improved and adapted to analyze packets in real time and then detect functional violations on the fly.

Part IV

Conclusions and Perspectives

"Education is the preparation of
the mind for future work, hence men should be educated
with special reference to the work."
Samuel Butler

Conclusions

Nowadays, different organizations are increasing the use of Web based systems. Such an example are the for example Web applications advantages that are well known and exploited since the latest 90's. They help to overcome geographical barriers and gain other advantages such as flexibility in using tools for synchronous and asynchronous collaboration between users. More recently, a new type of Web systems called Web services that relies on Service Oriented Architecture is being used to facilitate the interoperability and collaboration of heterogeneous systems. Moreover Web based systems are becoming increasingly complex and mission critical, and this is pushing the industry and research institutions towards the search of solutions to assure the usability and stability of these kind of systems.

The end-to-end process to test Web applications is composed of different stages. Obviously, these stages must be coherent to be able to have a smooth coupling and obtain a complete testing process, starting from the modeling up to the execution of test cases.

Among the most popular methods used on is to generate test cases in a semi-automatic way using as a base a recorded trace of human-IUT interactions. In the research field it exist few methodologies that achieve the performing of automatic abstract test cases generation. Besides, there is also a lack when considering the concretization of these abstract test cases to be able to effectively interact with real Web applications.

This thesis work has proposed methodologies to achieve all the testing stages considering all steps to complete the end-to-end testing process. In chapter two, we have presented a frame to model Web applications from their functional point of view. Our approach consists in building an IF specification that takes the system functional which may contain temporal restrictions. Afterward, we have presented an approach to derive test cases from this IF specification using TestGen-IF tool [COML08] developed in our laboratory. Then, the methodology that deals with the concretization and translation of the test cases to executable code is presented. This step introduces a complete methodology and the GeneraTCL tool used to render the abstract test cases into concrete test cases that can be executed on a real Web application using HTTP communication. Finally we have applied the generated test cases to an industrial Web application, the Mission Handler, based on a Web application provided by France Telecom.

After presenting how to tackle the testing of Web applications via its user interface using an active testing approach, chapter three has proposed a passive conformance testing

approach to inspect the communications among Web based systems. Nowadays, several Web applications invoke external applications using Web technologies (called Web services). In this work we illustrate that passive testing is relevant when Web services are already deployed or are closed implementations. We have also provided a new approach for testing the collaboration between different Web systems, and introduced the TIPS tool. We have proposed and evaluated an extensible and more flexible approach where properties called timed extended invariants are checked on a collected trace.

Our approach is flexible enough to detect new errors, such as functional errors and security flaws. The invariant approach can be combined with active testing to deduce a verdict after a Web service stimulation. Finally the methodologies and tools were applied to a real world case study (A travel reservation Web service) to demonstrate the effectiveness of this novel approach.

Perspectives

This work has presented a methodology to build a formal model of a Web application, automatically generate abstract tests and instantiate them to obtain executable code capable of interacting with a real Web system. However, as a first experiment, we have used a Web application based on HTML 4.0 pages. Thus, our work can be extended to cover needs of modern Web applications that contain user interface interactive elements such as in new HTML 5.0 tags to present multimedia. These Web applications have event centric interfaces such as the drag and drop or the right click generated by Ajax. This can represent a challenge mainly because:

- The lack of scripts and testing engines like Tclwebtest-ACS automated-testing being able to successfully interact with UIs based on Ajax. The major achievements until nowadays have been presented by Selenium [Too09a], but there are not yet stable technologies to perform this task.

- The capability of the tester to observe if a multimedia element of HTML 5.0 (such as <audio>) is working as it should be and assign a verdict.

The challenge might also be tackled by investigating the automatic analysis of non-observable system reactions (not included in HTTP communication and/or the user interface). In this case, there is a need to define a white/gray box based testing approach where we could have for instance a direct access to the Web application data base.

For the instantiation of abstract test cases part, in this stage of the work, only one specific value is assigned to each signal and parameter to instantiate the test case, therefore concretization of the test can be considered as simplistic. However, this stage can be enhanced to allow the extraction of more concrete test cases for each abstract one. This can be accomplished by using a data strategy to assign different values to the signals and parameters and improve the verdict assignation by adding data constraints.

The end-to-end active testing process (modelization/generation/instantiation/execution) can be integrated in a sole platform with a unique GUI to facilitate its utilization. This

work is being done by Montimage on the new tool "TEGsuite" that allows to manage test projects that may include different actors (designers, developers, testers, etc).

Concerning the passive testing approach, the methodology and the TIPS tool are in an early stage. The methodology can be extended and refined by including the XML schemas used to define the type of data exchanged between the different Web services. This study should enable invariants to express XML based requests to have a quicker access to the exchanged data. Another recommended enhancement is to fix the observation of TIPS to a layer (applicative, protocol, etc) and examine the contents of the exchanged XML data using Xpath. This can also improve the performance of the tool.

Nowadays, TIPS uses a collected trace to perform offline testing. However, as a major improvement, this tool could be adapted to analyze packets in real time and thus detect functional violations on the fly.

Finally, in complex SOA architectures with different collaboration between Web services (a Web partner can be for example a composed Web service itself), a study is recommended to define strategies to achieve the possibility of having distributed observation points. This could be an interesting challenge due to the need of correlation of collected messages and potential PO synchronization before the analysis of distributed traces.

Appendix

A Test Objectives of the Mission Handler

A1 Functional based test objectives

1. The MH can be accessed just by the internal personal of the company
2. All the users that access the MH must be authenticated by the server.
3. All the users registered to the MH has a personal profile created.
4. All the travelers can modify some information of its profile (just its own profile).
5. All the travelers can create a new Travel Request (TR)
6. A validator user can not validate its own mission; for each mission a validator and the traveler must be different users.
7. A validator user can delegate its validation rights to another user.
8. A user can receive one or more delegation of rights.
9. All validators can see the details of the missions that he must validate.
10. A validator user can also have the role of a traveler.

A2 Test Objectives related to timed constraints

1. A traveler can not create more one mission request for a period of 30 minutes.
2. If a validator don't validate a mission one week after the mission request is created, the validator will be reminded to validate the mission in his next login.
3. The MH will do automatic logout 20 minutes after detecting "no activity" of a user.

B Test Objectives of the TRS

B1 Test objectives considering the orchestration order

1. A partial itinerary is sent to the Travel Reservation service. It does not contain any previous reservation (no airline, no vehicle and no hotel). The Travel Reservation service system contacts the Airline Reservation service partner, then the Vehicle Reservation service partner, then the Hotel Reservation service partner, before sending back the complete itinerary.

2. A partial itinerary is sent to the Travel Reservation service. It already has an airline reservation but no vehicle and no hotel reservations. The Travel Reservation service system contacts the Vehicle Reservation service partner, then the Hotel Reservation service partner, before sending back the complete itinerary.

3. A partial itinerary is sent to the Travel Reservation service. It already has a vehicle reservation but no airline and no hotel reservations. The Travel Reservation service system contacts the Airline Reservation service partner, then the Hotel Reservation service partner, before sending back the complete itinerary.

4. A partial itinerary is sent to the Travel Reservation service. It already has a hotel reservation but no airline and no vehicle reservations. The Travel Reservation service system contacts the Airline Reservation service partner, then the Vehicle Reservation service partner, before sending back the complete itinerary.

5. A partial itinerary is sent to the Travel Reservation service. It already has an airline reservation and a vehicle reservation but no hotel reservation. The Travel Reservation service system contacts the Hotel Reservation service partner, before sending back the complete itinerary.

6. A partial itinerary is sent to the Travel Reservation service. It already has an airline reservation and a hotel reservation but no vehicle reservation. The Travel Reservation service system contacts the Vehicle Reservation service partner, before sending back the complete itinerary.

7. A partial itinerary is sent to the Travel Reservation service. It already has a vehicle reservation and a hotel reservation but no airline reservation. The Travel Reservation service system contacts the Airline Reservation service partner, before sending back the complete itinerary.

8. A complete itinerary is sent to the Travel Reservation service. It already has an airline, a vehicle and a hotel reservation. The Travel Reservation service system sends back the complete itinerary to the external system.

9. If the TRS receives a message with an itinerary from VRS, the TRS should have contacted the VRS asking for a vehicle reservation.

B2 Test objectives related to time constraints

1. In case the process doesn't receive any response from the Airline Reservation partner service within 20 seconds, a request for canceling the reservation is sent to this partner. No airline item(s) is included in the completed reservation.

2. In case the process doesn't receive any response from the Airline Reservation partner service within 20 seconds, a request for canceling the reservation is sent to this partner. No airline item(s) is included in the completed reservation.

3. In case the process doesn't receive any response from the Vehicle Reservation partner service within 20 seconds, a request for canceling the reservation is sent to this partner. No vehicle item(s) is included in the completed reservation.

4. In case the process doesn't receive any response from the Hotel Reservation partner service within 20 seconds, a request for canceling the reservation is sent to this partner. No hotel item(s) is included in the completed reservation

Bibliography

[ABS01] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. Trex: A tool for reachability analysis of complex systems. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 368–372. Springer, 2001.

[ACC+04] Baptiste Alcalde, Ana R. Cavalli, Dongluo Chen, Davy Khuu, and David Lee. Network protocol system passive testing for fault management: A backward checking approach. In David de Frutos-Escrig and Manuel Núñez, editors, *FORTE*, volume 3235 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2004.

[ACD07] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. A survey of analysis models and methods in website verification and testing. In *ICWE*, pages 306–311, 2007.

[ACN03] José Antonio Arnedo, Ana R. Cavalli, and Manuel Núñez. Fast testing of critical properties through passive testing. In Dieter Hogrefe and Anthony Wiles, editors, *TestCom*, volume 2644 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 2003.

[AMC+09] Cesar Andres, Stephane Maag, Ana Cavalli, Mercedes G. Merayo, and Manuel Nunez. Analysis of the olsr protocol by using formal passive testing. *Asia-Pacific Software Engineering Conference*, 0:152–159, 2009.

[AMN09a] Cesar Andres, Mercedes G. Merayo, and Manuel Nunez. Formal correctness of a passive testing approach for timed systems. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:67–76, 2009.

[AMN09b] Cesar Andres, Mercedes G. Merayo, and Manuel Nunez. Passive testing of stochastic timed systems. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:71–80, 2009.

[AOA05] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with fsms. *Journal of Software and Systems Modeling*, 2005.

[BCNZ05] Emmanuel Bayse, Ana R. Cavalli, Manuel Núñez, and Fatiha Zaïdi. A passive testing approach based on invariants: application to the wap. *Computer Networks*, 48(2):235–245, 2005.

[BDAR98] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. A guided incremental test case generation procedure for conformance testing for CEFSM specified protocols. In *IWTCS*, Tomsk, Russia, August 1998.

[BDSG09] Abdelghani Benharref, Rachida Dssouli, Mohamed Adel Serhani, and Roch Glitho. Efficient traces' collection mechanisms for passive testing of web services. *Inf. Softw. Technol.*, 51(2):362–374, 2009.

[BFG+99] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, Laurent Mounier, and Joseph Sifakis. IF: An Intermediate Representation for SDL and its Applications. In *SDL Forum*, pages 423–440, 1999.

[BFS05] A. Belinfante, L. Frantzen, and C. Schallhart. Tools for test case generation. In *Model-based Testing of Reactive Systems: Advanced Lectures*. Springer-Verlag, 2005.

[BGM02] Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In *CAV*, pages 343–348, 2002.

[BGMO04] Marius Bozga, Susanne Graf, Laurent Mounier, and Iulian Ober. IF Validation Environment Tutorial. In *SPIN*, pages 306–307, 2004.

[BGO+04] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF Toolset. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 237–267. Springer, 2004.

[CA97] Ana Cavalli and Ricardo Anido. Verification and testing techniques based on the finite state machine model. Technical Report 970902, Institut National des Telecommunications, September 1997.

[CCD+] Cavarra, Crichton, Davies, Hartman, Jeron, and Mounier. Using UML for automatic test generation. TACAS 2002.

[CDRZ99] Ana Cavalli, David, Christian Rinderknecht, and Fatiha Zaïdi. Hit-or-Jump: An Algorithm for Embedded Testing with Applications to IN Services. In *Formal Methods for Protocol Engineering And Distributed Systems*, pages 41–56, Beijing, China, october 1999.

[CFCB09] Tien-Dung Cao, Patrick Félix, Richard Castanet, and Ismail Berrada. Testing web services composition using the tgse tool. *Services, IEEE Congress on*, 0:187–194, 2009.

[CKM93] A. Cavalli, Sung Un Kim, and Patrick Maigron. Improving conformance testing for LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Umit Uyar, editors, *Proc. of the 6th International Conference on Formal Description Techniques FORTE'93*, pages 367–384, 1993.

[CLM+08] A. Cavalli, M. Lallali, S. Maag, G. Morales, and F. Zaidi. *Emergent Web Intelligence*, chapter Modeling and Testing of Web Based Systems. Studies in Computational Intelligence. Springer Verlag, 2008. To appear, 39 pages.

[CMM+06] Ana R. Cavalli, Stéphane Maag, Wissam Mallouli, Mikael Marche, and Yves-Marie Quemener. Application of Two Test Generation Tools to an Industrial Case Study. In *TestCom*, pages 134–148, 2006.

[CMM07] Ana Cavalli, Stephane Maag, and Gerardo Morales. Regression and performance testing of an e-learning web application: dotlrn. In *SITIS '07: Proceedings of the 2007 Third International IEEE Conference on Signal-Image Technologies and Internet-Based System*, pages 369–376, Washington, DC, USA, 2007. IEEE Computer Society.

[CMPV05a] Ana Cavalli, Stéphane Maag, Sofia Papagiannaki, and Georgios Verigakis. From uml models to automatic generated tests for the dotlrn e-learning platform. *Electron. Notes Theor. Comput. Sci.*, 116:133–144, 2005.

[CMPV05b] Ana R. Cavalli, Stéphane Maag, Sofia Papagiannaki, and Georgios Verigakis. From uml models to automatic generated tests for the dotlrn e-learning platform. *Electr. Notes Theor. Comput. Sci.*, 116:133–144, 2005.

[COML08] Ana Rosa Cavalli, Edgardo Montes De Oca, Wissam Mallouli, and Mounir Lallali. Two complementary tools for the formal testing of distributed systems with time constraints. In *DS-RT '08: Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 315–318, Washington, DC, USA, 2008. IEEE Computer Society.

[DHHG06] Karnig Derderian, Robert M. Hierons, Mark Harman, and Qiang Guo. Automated unique input output sequence generation for conformance testing of fsms. *The Computer Journal*, 49:2006, 2006.

[ECW01] G. Meredith E. Christensen, F. Curbera and S. Weerawarana, 15 March, 2001. Web Services Description Language (WSDL) 1.1.http://www.w3.org/TR/wsdl.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbat, L. Mounier R. Mateescu, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *The 8th Conference on Computer-Aided Verification, CAV'96*, New Jersey, USA, August 1996. Springer Verlag.

[Gar96] H. Garavel. An overview of the eucalyptus toolbox. In *Proceedings of the COST247 Workshop*, 1996.

[GHK+98] Todd Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. In *Proceedings of theInternational Conference on Software Engineering (ICSE 1998)*, pages 188–197, Kyoto, Japan, April 1998.

[GMBW] Ana Cavalli Wissam Mallouli Edgardo Montes de Oca Gerardo Morales, Stephane Maag and booktitle = 8th IEEE International Conference on Web Services ICWS year = 2010 Bachar Wehbi, title = Timed Extended Invariants for the Passive Testing of Web Services.

[GMP04] Maria Del Mar Gallardo, Pedro Merino, and Ernesto Pimentel. A generalized semantics of promela for abstract model checking. *Formal Asp. Comput.*, 16(3):166–193, 2004.

[GR88] G.M.Reed and A.W. Roscoe. A timed model for communicating sequential processes. Theoretical Computer Science 58, 249 - 261, 1988.

[Gro00] Object Management Group. Unified modelling language 1.3 specification, March 2000.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Hes06] Anders Hessel. *Model-Based Test Case Selection and Generation for Real-Time Systems.* PhD thesis, Uppsala Universitet, Sweden, 2006.

[HK06] Antawan Holmes and Marc Kellogg. Automating functional tests using selenium. In *AGILE '06: Proceedings of the conference on AGILE 2006*, pages 270–275, Washington, DC, USA, 2006. IEEE Computer Society.

[Htt09] HttpUnit. http://httpunit.sourceforge.net/. 2009.

[ISO89] ISO. *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour.* International Standard IS-880, 1989.

[ITU92] ITU. *Recommendation Z.100 : CCITT Specification and Description Language (SDL)*, 1992.

[JGW03] G. Rethy I. Schieferdecker A. Wiles J. Grabowski, D. Hogrefe and Colin Willcock. An Introduction to The Testing and Test Control Notation (TTCN-3). In *Computer Networks 42(3)*, pages 375–403, 2003.

[JJ05] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms. *STTT*, 7(4):297–315, 2005.

[LCH+02] David Lee, Dongluo Chen, Ruibing Hao, Raymond E. Miller, Jianping Wu, and Xia Yin. A formal approach for passive testing of protocol data portions. In *ICNP*, pages 122–131. IEEE Computer Society, 2002.

[LNS+97] D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *ICNP '97: Proceedings of the 1997 International Conference on Network Protocols (ICNP '97)*, page 113, Washington, DC, USA, 1997. IEEE Computer Society.

[LqMW+06] Nuo Li, Qin qin Ma, Ji Wu, Mao zhong Jin, and Chao Liu. A framework of model-driven web application testing. In *COMPSAC (2)*, pages 157–162, 2006.

[LY96] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - a Survey. *IEEE Transactions on Computers*, 84:1090–1123, August 1996.

[LZC07] Mounir Lallali, Fatiha Zaidi, and Ana Cavalli. Timed modeling of web services composition for automatic testing. In *SITIS '07: Proceedings of the 2007 Third International IEEE Conference on Signal-Image Technologies and Internet-Based System*, pages 417–426, Washington, DC, USA, 2007. IEEE Computer Society.

[Mil89a] R. Milner. Communication and concurrency. Prentice-Hall, 1989.

[Mil89b] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[MLM09] A. Mammar M. Lallali, W. Mallouli and G. Morales. *Emergent Web Intelligence: Advanced Semantic Technologies*, chapter Modeling and Testing Secure Web Applications. Atlantis Press, 2009. To appear.

[MLMC08] Wissam Mallouli, Mounir Lallali, Gerardo Morales, and Ana Rosa Cavalli. Modeling and Testing Secure Web-Based Systems: Application to an Industrial Case Study. In *The fourth International IEEE Conference on Signal-Image technology & Internet-Based Systems (SITIS)*, Bali, Indonesia, November 30 - December 03 2008.

[MMC08] Wissam Mallouli, Gerardo Morales, and Ana Rosa Cavalli. Testing Security Policies for Web Applications. In *the 1st International ICST workshop on Security Testing (SECTEST'08)*, Lillehammer, Norway, April 09, 2008.

[MNR08] Mercedes G. Merayo, Manuel Nú nez, and Ismael Rodríguez. Formal testing from timed finite state machines. *Comput. Netw.*, 52(2):432–460, 2008.

[MRMT95] M.Clatin, R.Groz, M.Phalippou, and Richard Thummel. Two approaches linking a test generation tool with verification techniques. *8th Int. Workshop on Protocol Test Systems*, 1995.

[msc96] *IUT-T Rec. Z. 120 Message Sequence Charts, (MSC).* Geneva, 1996.

[Net09] NetBeans Framework. http://www.netbeans.org/. 2009.

[NT81] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transitions tours. In *IEEE Fault Tolerant Computer Systems*, 1981.

[OAS09] OASIS. http://www.bpmn.org/. 2009.

[Ope09] OpenACS Community. http://www.openacs.org/. 2009.

[Pet62] A. A. Petri. *Kommunikation mit Automaten.* Ph. D. thesis, Universitat Bonn, 1962.

[pro09] Politecnico Di Torino Analyser project. http://www.nbee.org/doku. 2009.

[RN07] Ismael Rodríguez and Manuel Núñez. A formal methodology to test complex heterogeneous systems. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *ATVA*, volume 4762 of *Lecture Notes in Computer Science*, pages 394–409. Springer, 2007.

[ROM08] Hassan Reza, Kirk Ogaard, and Amarnath Malge. A model based testing technique to test web applications using statecharts. In *ITNG '08: Proceedings of the Fifth International Conference on Information Technology: New Generations*, pages 183–188, Washington, DC, USA, 2008. IEEE Computer Society.

[SD88] K. K. Sabnani and A. T. Dahbura. A Protocol Test Generation Procedure . In *Computer Networks and ISDN System*, volume 15, pages 285–297, 1988.

[SM03a] Joe Abboud Syriani and Nashat Mansour. Modeling web systems using sdl. In *ISCIS*, pages 1019–1026, 2003.

[SM03b] Joe Abboud Syriani and Nashat Mansour. Modeling web systems using sdl. In *ISCIS*, pages 1019–1026, 2003.

[sR09] Åsmund Realfsen. http://www.km.co.at/km/twtr. 2009.

[TB99] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *Proceedings of the 7th European International Conference on Software Testing, EuroSTAR'99*, November 1999.

[Tcl09] TclWebTest Tool. http://tclwebtest.sourceforge.net/. 2009.

[TM02] C. Teyssie and Z. Mammmeri. Analyse des approches d'extension de SDL pour la specification de systemes et reseaux temps reel. In *RTS Embedded System 2002*, Paris, March 2002.

[Too09a] Selenium Tools. http://seleniumhq.org/. 2009.

[Too09b] Selenium Tools. http://seleniumhq.org/projects/ide/. 2009.

[Tre01] Jan Tretmans. Testing techniques. Lecture Notes. Formal Methods and Tools Group, Faculty of Computer Science, University of Twente, 2001.

[TTC] ETSI. TTCN-3. *TTCN-3 - Core Language.*

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

[UZ93] Hasan Ural and Keqin Zhu. Optimal length test sequence generation using distinguishing sequences. *IEEE/ACM Trans. Netw.*, 1(3):358–371, 1993.

[Vie07] Elisangela Vieira. *Automated Model-Based Test Generation for Timed Systems.* PhD thesis, INT, Evry, France, 2007.

[vL00] Axel van Lamsweerde. Formal specification: a roadmap. In *ICSE - Future of SE Track*, pages 147–159, 2000.

[Web09] Canoo WebTest. http://webtest.canoo.com/webtest/manual/webtesthome.html. 2009.

[WX09] Xinchun Wang and Peijie Xu. Build an auto testing framework based on selenium and fitnesse. *Information Technology and Computer Science, International Conference on*, 2:436–439, 2009.

[XXC+03] Lei Xu, Baowen Xu, Zhenqiang Chen, Jixiang Jiang, and Huowang Chen. Regression testing for web applications based on slicing. In *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, page 652, Washington, DC, USA, 2003. IEEE Computer Society.

[YD07] Y. Yan and P. Drague. Monitoring and Diagnosing Orchestrated Web Service Processes. In *Proceedings of the 2007 IEEE International Conference on Web Services, Salt Lake City, Utah, USA*, 2007.

[Yi91] W. Yi. CCS + Time = an interleaving model for real time systems. Automata, Languages and Programming, 18 (LNCS 510, Springer-Verlag, Berlin) 217-228, 1991.