Symbolic test case generation for testing orchestrators in context

José Pablo Escobedo del Cid

Thèse de doctorat de Télécom SudParis dans le cadre de l'école
doctorale S&I en co-accréditation avec
l'Université d'Évry-Val d'Essone

Spécialité:
Informatique

Par :
José Pablo Escobedo Del Cid

Thése prsentée pour l'obtention du diplôme de Docteur de
Telecom SudParis

Symbolic Test Case Generation for Testing Orchestrators in Context

---

Soutenue le 25 novembre 2010 devant le jury composé de:

Hierons, Rob	Rapporteur	Prof. à l'Université de Brunel
Castanet, Richard	Rapporteur	Prof. à LaBRI
Rusu, Vlad	Examinateur	Chercherur de recherche INRIA
Gaston, Christophe	Examinateur	Chercheur à CEA LIST DILS
Cavalli, Ana	Co-directrice de thèse	Prof. à Télécom SudParis
Le Gall, Pascale	Co-directrice de thèse	Prof. à l'Université d'Évry-Val d'Essone

Thèse n° 2010TELE0027

Symbolic Test Case Generation for Testing Orchestrators in Context

(Génération Symbolique des Cas de Test pour Tester d'Orchestrateurs en Contexte)

Résumé de la thèse de

José Pablo Escobedo

Telecom SudParis

Supervisé par:

Ana Cavalli et Pascale Le Gall

LES services Web sont des logiciels qui offrent des fonctionnalités à des autres machines à travers l'Internet. Les services Web sont basés sur l'Architecture Orientée Services (SOA, pour son nom en anglais). Ils peuvent être invoqués en utilisant des standards pour le Web (normalement, *SOAP* [GHM+07], *UDDI* [CHvRR04], *XML* [BPSM+08], *HTTP* [FGM+99], *WSDL* [CCMW01]). Dans les dernières années, l'utilisation des services Web a augmenté à cause de la flexibilité qu'ils offrent, ainsi comme de l'intégration des systèmes hétérogènes. En plus, SOA ajoute de la valeur dans le sens où les services peuvent être réutilisés et partagés; cela fait les systèmes plus flexibles et adaptables en cas où il y a des changements dans les processus des entreprises, et améliore l'intégration des systèmes [HS05]. Aussi, des nouvelles façons d'utiliser les services Web se sont développées, en combinant pour créer des services plus complets et complexes. Le processus de réutiliser les services Web pour en créer des nouveaux s'appelle *composition des services Web*, et son but principal est de permettre la réutilisation des fonctionnalités proposées par les services Web. C'est pour cette raison que l'architecture SOA a été bien acceptée par les entreprises partout dans le monde: elle aide à réduire le coût et le temps qu'il faut pour créer des solutions, et c'est la composition des services Web que nous étudions dans cette thèse. Plus spécifiquement, on veut assurer leur correct comportement en utilisant des techniques du test pour détecter des possibles erreurs.

Parmi les différents façons de créer des compositions des services Web, les deux qui sont les plus utilisées sont ([Pel03]):

- *Chorégraphies des services Web* [MKB09], où chaque service Web dans la composition interagis au même niveau, en connaissant quand et comment se communiquer avec le reste des services Web.
- *Orchestrations des services Web* [GJW04], où il y a un composant central qui guide le processus, en prenant des décisions selon les réponses des services Web.

Il y a aussi une autre façon de faire des compositions des services Web, qui est basée sur la Web sémantique, et qui n'est pas une orchestration mais plutôt une chorégraphie. Parmi les trois types de faire des compositions mentionnés, les orchestrations sont les plus utilisés (au moment de l'écriture de ce document). Une des raisons est ce que la façon la plus utilisée de créer des orchestrations est en utilisant le standard WS-BPEL (pour Web Services Business Process Execution Language), qui a été développée par OASIS[1] (qui est un consortium qui guide le développement et l'adoption des standards pour la communauté global informaticienne). En plus, WS-BPEL est basé sur XLANG (Microsoft, 2001) et WSFL (IBM, 2001), et plusieurs entreprises connues ont participé lors de ça création (Microsoft, IBM, SAP, Oracle, Adobe Systems, parmi autres).

Dû à la grande acceptation de cet nouvelle technique de combiner des services Web pour créer des nouveaux, plus complètes et complexes, des nouvelles façons d'assurer leur correct comportement ont été aussi développées [vBK06]. En plus, les orchestrations ont des caractéristiques spéciales qui n'ont pas présentes dans des autres types des systèmes (comme les systèmes unitaires), et les approches du test habituels (i.e., techniques du test unitaire utilisées avec des systèmes isolés) ne les prennent pas en compte. Dans cet thèse, on s'intéresse à cet problème pour finalement apporter un nouvel approche (boîte noire) du test pour aider avec la validation et vérification des orchestrations.

On va parler maintenant un peu plus sur les caractéristiques des orchestrations. Leur composant central qui guide tout le processus s'appelle *l'orchestrateur*. Tous les services Web involucrés dans l'orchestration interagissent que avec l'orchestrateur et jamais entre eux. De la même façon, tout utilisateur de l'orchestration (qui normalement le fait à travers d'une application Web) interagisse avec l'orchestration que à travers l'orchestrateur. C'est pour ce raisons que l'orchestrateur joue une rôle très important dans l'orchestration, parce qu'il prend des décisions selon les réponses du Web services et les entrées de l'utilisateur. On considère les orchestrations comme un type particulier des systèmes à base des composants, où il y a un composant cen-

[1] http://www.oasis-open.org/

tral (l'orchestrateur) qui interagisse avec l'environnement (l'utilisateur) et joue le rôle d'interface du système.

Contexte de Notre Travail

L'INTÉRÊT de notre travail consiste précisément en tester des orchestrateurs, mais d'un point de vue particulier: en le testant dans son situation d'usage. Tester un système veut dire de l'exécuter pour trouver des erreurs. Cet activité exige une grande quantité du temps dans les entreprises (50-70%) [NS08], et cela provoque des délais dans les projets. Pourtant, l'activité du test doit être le plus automatique possible pour réduire le temps utilisé pour la faire. En plus, cela doit être fait de façon que cela assure, le plus possible, que le système après avoir être testé, fonctionne correctement (même si, comme l'on discute plus loin, on ne peut pas être sûr qu'il n'y aura pas des erreurs). Pour des systèmes où il ne peut pas y avoir des erreurs (comme c'est les cas pour les systèmes dans les trains, avions, etc.), on utilise l'aide des approches formels. Or, des modèles mathématiques sont utilisés pour représenter la spécification des systèmes (à la place des langages avec des ambiguïtés, comme par exemple, le français) et des techniques formelles sont utilisées pour tester l'implémentation de la spécification d'un système (le système mis au test, que dans notre cas est l'orchestrateur de l'orchestration). Cet façon de tester des systèmes en utilisant des modèles est connu comme *test à base des modèles*.

Une activité en relation directe avec le test à base de modèles est connue comme *contrôle des modèles*.

Dans quelques mots, le contrôle des modèles est une activité automatique qui, donné le modèle d'un système et une propriété donnée dans une formalisme logique (en pratique, une variation de la logique temporelle), systématiquement vérifie si le modèle donné satisfait la propriété [Kat99]. Dans cet thèse, on ne fait pas des contrôle des modèles avec les modèles qu'on utilise. On suppose que les modèles sont correctes mais, dans le contexte d'un processus de validation et vérification complète, faire du contrôle des modèles est une activité complémentaire que doit être fait avec le travail présenté dans cet document.

En ce qui concerne le formalisme choisit pour représenter les spécifications, il y a plusieurs parmi lesquels ont peut choisir [LVD09], chacun en offrant différents avantages et désavantages. On peut mentionner quelques différents types des modèles

qui on été utilisées pour faire du test, comme les modèles à base des états (comme JML [L$^+$99]), les modèles à base des transitions (comme FSM [LZCC07], LTS [Tre08]), les modèles opérationnels (comme Petri Nets [YJC10]), parmi autres. Le modèle choisit dépends des caractéristiques du système qu'on veut capturer, aussi comme la familiarité qu'on a avec le modèle.

Tester un système veut dire d'interagir avec lui (selon les objectifs du test) pour détecter des erreurs par rapport à sa spécification. Donc, à partir des modèles des systèmes, on peut extraire ces objectifs de test qui le système soumis au test (SUT) doit respecter. Ceci est fait parce que c'est impossible de tester *tous* les comportements qui sont décrits par les modèles qui représentent les spécifications des systèmes (sauf pour des exemples triviales). Cet processus peut être très lent, même infini. Pour cela, les objectifs du test prennent en compte qu'un sous ensemble de tous les comportements possibles du système, et pourtant, l'activité du test ne peut pas être complète: peut importe l'approche utilisé, l'activité du test peut que détecter les erreurs, mais pas leur absence [Dij79].

Les objectifs du test sont sélectionnés pour tester des aspects spécifiques du système, selon le type du test qu'on veut réaliser. Parmi les types du test, on peut mentionner:

- Tester la vitesse du système (test de performance).
- Tester la réaction du système lorsque le nombre de requettes augment (test du stress).
- Tester la réaction du système lors des entrées erronées (test de robustesse).
- Tester combien du temps on peut compter avec le correct comportement du système (test du dépendance).

Dans ce document, on s'intéresse à tester si le SUT fait ce qu'il doit faire. Ce type du test est connu comme *test de conformité* et, plus spécifiquement, a comme bût de déterminer si le comportement d'un SUT est correcte par rapport à son spécification (qui est donnée sous la forme d'un modèle). La spécification sur laquelle on parle est une spécification des comportements du système, i.e., elle décrit le comportement du système en fonction des opérations qu'il peut effectuer. Dans le reste de ce document, lorsqu'on parle sur la spécification d'un système, on parle de son spécification des comportements.

Aussi, normalement lors de faire du test de conformité, la spécification du SUT n'est

pas connue, i.e., depuis le point de vue du testeur, le système est une *boîte noire*. Cela veut dire que, vu qu'on teste des systèmes d'information, le testeur n'a pas accès au code du SUT.

Pour le cas où le testeur a accès, l'activité du test est connue comme *test au boîte blanche*, et normalement a comme objectif d'assurer la couverture de différentes parties du code. Dans le test au boîte noire, lest tests sont obtenus entièrement à partir de la spécification du système (dans la pratique, en utilisant des objectifs du test) qui décrit les comportements attendus de la boîte noire [UL07].

Finalement, un système peut être testé à différent niveaux :

- isolé (test unitaire),
- chaque composant est testé séparément (test de composants),
- le système est testé de manière que ses différents composants interagissent correctement (test d'intégration),
- le système est testé lorsqu'il interagisse avec des autres systèmes (qui lui enferment ou contient) (test en contexte),
- ou bien le système peut être testé dans sa totalité (test de systèmes).

Notre approche consiste donc à faire du test en contexte des orchestrateurs. Lors de tester des systèmes en contexte [Kho04], le (sous)système qu'on veut tester est *encastré* dans le système en sa totalité et en train d'interagir avec des autres composants, qui constituent son contexte: le système dans sa totalité est composé du SUT et du reste des composants. Par rapport à ce contexte, il y a plusieurs hypothèses qui son faites. Las plus connues sont que les composants dans le contexte du SUT se comportent correctement, i.e., ils n'ont pas des erreurs parce qu'ils conforment par rapport à leur spécifications (qui sont supposées d'être disponibles), et que la communication parmi les composants marche correctement. En plus, normalement le SUT est testé que à travers son contexte [ACJY03].

Notre Approche

MAINTENANT on va dire plus sur comment faire du test de conformité en contexte (des boîtes noires) pour déterminer si un orchestrateur se comporte correctement selon sa spécification. Comme on a dit avant, les orchestrations

sont des types de systèmes avec des caractéristiques spéciales, où l'orchestrateur guide le processus. Ainsi, le système qui est soumis au test est composé de l'orchestrateur et les services Web (ou composants) qui interagissent avec lui. Ces services Web constituent le contexte de l'orchestrateur, et on a comme objectif de assurer la conformité de l'orchestrateur par rapport à son spécification lorsqu'il interagisse avec son contexte. Pourtant, pour pouvoir appliquer un approche classique du test pour ce type de systèmes, on devrait prendre en compte la spécification de l'orchestrateur et de tous les services Web avec celui-ci interagisse.

En faisant cela, on peut obtenir une représentation formel de tout le système. Dû à qu'on utilise des modèles mathématiques pour représenter chaque spécification (de l'orchestrateur et de chaque service Web), la spécification de tout le système peut être obtenue en utilisant des manipulations mathématiques avec les modèles (comme le produit cartésien, etc.). Par contre, il peut y avoir deux problèmes en faisant cela: le premier est que la spécification de tous les composants (l'orchestrateur ou services Web) ne son pas toujours disponibles (et en plus dans une notation mathématique). Cela peut se produire, par exemple, car les services Web sont développés par des sociétés externes qui ne veulent pas donner des détails sur le comportement interne de leur produits.

Le deuxième problème est connu sous le nom de *explosion combinatoire* et est provoqué parce que la combinaison de différents représentations formels de les spécifications des composants qui forment partie du système peut donner comme résultat un modèle très grand, qui est difficile de manipuler.

Donc, la première hypothèse qu'on fait est que la spécification de l'orchestrateur est *toujours* disponible et, même si l'on travaille avec l'approche boîte noire, que le SUT *peut être* représenté par un modèle formel (que l'on peut obtenir, par exemple, en interagissant avec lui).

Pour attaquer les problèmes décrits précédemment (explosion combinatoire) et la non disponibilité des spécifications des services Web, on propose un approche qui consiste de tester les orchestrateurs en prenant en compte que leur spécifications mais pas celles de reste des composants dans le système, même si les orchestrateurs interagissent avec les services Web pendant qu'on les teste. Plus précisément, on prend en compte la description de l'interface des services Web, qui est normalement disponible sous la forme d'un fichier WSDL, mais qui décrit les servies d'un point de vue syntactique et ne décrit pas leur comportement (du point de vue sémantique).

Notre approche est illustré dans la Figure 1: le contexte de l'orchestrateur est l'utilisateur

et les services Weg WS_1 et WS_2; et on veut déterminer si l'orchestrateur est conforme par rapport à son spécification lorsqu'il interagisse avec WS_1 et WS_2.

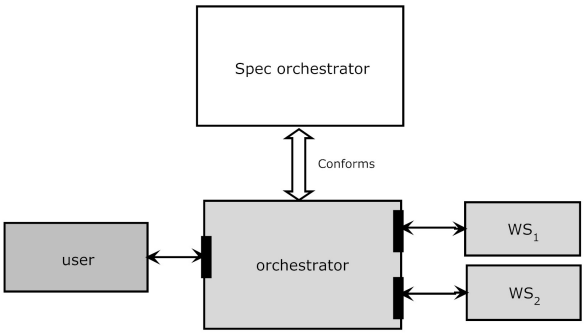


Figure 1: Teste de boîte noire basé sur modèles pour déterminer la conformité de l'orchestrateur

Dans cette thèse, en prenant en compte que la spécification de l'orchestrateur, on propose un approche pour tester les orchestrateurs en utilisant la technique de test de boîte noire, basée sur des modèles, dans le cas particulier où les orchestrateurs interagissent avec le reste des services Web dans le système pendant qu'on les teste. Ceci représente une cas spéciale du test en contexte, où le testeur peut contrôler ou piloter l'orchestration en utilisant l'orchestrateur.

Ceci est précisément l'une des contributions de notre travail, parce que la plupart des travails qui on été fait pour tester des orchestrations et des systèmes à base de composants, simulent le contexte de l'orchestrateur, ou font l'hypothèse que les spécifications de tous les services Web sont disponibles ([CFCB10, LZCH08, BvdMFR06, vRT04]), mais, comme on a dit avant, ceci n'est pas toujours le cas. Aussi, on a choisit la relation de conformité *ioco* pour tester les orchestrateurs en contexte [Tre96].

ioco veut dire, en anglais, conformité des entrées/sorties, et son idée de base est que n'importe quel SUT est conforme par rapport à son spécification si, après toute séquence des interactions qui on été observés par le testeur en interagissant avec le SUT (et qui est valide selon la spécification, i.e., est spécifiée), n'importe quel observation après cet séquence est aussi spécifiée. S'il y a une observation faite par le testeur qui n'est pas spécifiée, donc on peut conclure que le système qui est soumis au test n'est pas conforme par rapport à son spécification.

Depuis l'année 1996, plusieurs travaux ont été basés sur cet relation de conformité [RBJ00, GGRT06, vdBRT04, FTW06], et a été aussi bien acceptée et utilisée dans des différents

domaines d'application (par exemple, en prenant en compte le temps [FT07]). En plus, cet relation de conformité a comme avantage le fait d'autoriser des spécifications non déterministes. En fait, **ioco** autorise des situations où il y a de la sous-spécification parce qu'elle ne force pas aux implémentations de considérer toutes les sorties possibles (depuis un état), mais seulement quelques unes. Par rapport aux entrées, si une entrée n'est pas spécifiée, n'importe quel observation faite dans le SUT après cet entrée est valide aussi.

En utilisant que la spécification de l'orchestrateur, on complément l'activité du test des orchestrateurs en prenant en compte un autre problème qui est commun lors de travailler avec des orchestrations: commet pouvons-nous être sûrs que les services Web qu'on choisit sont compatibles avec l'orchestrateur ? Dans cette thèse, on montre comment générer des comportements pour tester les services Web, et qui sont extraits de la spécification de l'orchestrateur. Ces comportements sont précisément les comportements attendues pour les services Web de la part de l'orchestrateur. En plus, ces comportements extraits de la spécification de l'orchestrateur peuvent être vus comme une spécification partiel pour les services web. En fait, les services Web peuvent offrir plus des fonctionnalités que celles qui sont utilisées par l'orchestrateur. Dans notre approche, on choisit de ne prendre en compte tous les fonctionnalités qui ne son pas utilisées par l'orchestrateur et on se focalise seulement dans celles qui sont utilisées.

Donc, on propose un approche pour tester le comportement correcte des orchestrations en testant l'orchestrateur en contexte et en testant aussi la compatibilité des services Web, tout ça en prenant en compote que la spécification de l'orchestrateur. En procédant comme ça, on attaque deux problèmes communs à l'heure de tester des systèmes avec des plusieurs composants: la manque de la disponibilité des spécifications des composants et l'explosion combinatoire. En plus, on attaque cet dernier problème en utilisant *l'exécution symbolique* [Kin75, Cla76], qui a comme idée principale celle de exécuter des programmes en utilisant des symboles à la place de donnés concrets, et de générer une structure en forme d'arbre pour décrire tous les comportements possibles du programme d'un manière symbolique. En effet, dans notre approche, lest objectifs du tests sont des sous-arbres des arbres symboliques, et on utilise un algorithme pour tester la conformité du SUT qui est basé sur ces techniques symboliques.

Structure du Document

CETTE thèse est structurée dans deux parties. Dans la Partie 1, on présente la partie théorique qui est la base de notre approche, ainsi comme l'état de l'art et l'introduction aux types des systèmes sur lesquels on travaille dans le reste de la thèse.

- Nous commençons au le Chapitre 2 en introduisant l'état de l'art par rapport aux types des systèmes que nous analysons: les orchestrations des services Web. Nous présentons les différentes façons de créer des compositions des services Web, les standards utilisés par les orchestrations et les systèmes à base des composants. Nous présentons aussi l'architecture SOA, qui est la base des orchestrations, et le langage le plus utilisé pour créer celles-ci: WS-BPEL (pour son nom en anglais Web Services Business Process Execution Language). Finalement, nous introduisons l'exemple que nous allons utiliser dans le reste de la thèse.

- Dans le Chapitre 3 nous introduisons la relation de conformité que nous utilisons: **ioco**. In introduise la notation que nous utilisons pour représenter les spécifications des systèmes: les systèmes à base des transitions étiquetées (LTS). Nous utilisons les LTSs pour modeler les spécifications des systèmes et les SUTs. Plus spécifiquement, nous utilisons les IOLTS, qui ne sont plus que des LTS où les étiquettes sont divisées entre entrées et sorties. Nous commençons par présenter les techniques utilisées pour tester des systèmes unitaires et après nous continuons avec les systèmes à base de composants. Plus spécifiquement, nous présentons les résultats obtenus dans [vdBRT03] lorsqu'ils utilisent la relation de conformité **ioco** avec des systèmes à base des composants. Finalement nous présentons aussi des autres travaux qui ont été faits pour tester des systèmes similaires.

- Nous continuons dans le Chapitre 4 en présentant la version modifiée de **ioco** et que nous définissons pour tester des orchestrateurs en contexte. Cette nouvelle relation de conformité nous permet de prendre en compte des informations par rapport au contexte des orchestrateurs, et on fait cela à partir des modifications sur les spécifications des orchestrateurs (qui sont représentées par des IOLTs). Ces modifications prennent en compte le statu des services Web qu'interagissent avec l'orchestrateur. Dans ce chapitre, on introduise une classification pour les différentes situations des services Web. Plus spécifiquement, nous classifions les canaux utilisés par les services Web et nous disons qu'un canal est:

  - **Contrôlable**, si le testeur joue le rôle du service Web, ou bien s'il contrôle complètement les informations qui sont envoyés sur le canal.

  - **Observable**, si le testeur ne contrôle pas les informations qui sont envoyés sur le canal, mais l'architecture du test est tel qu'il peut observer ces informations.
  - **Caché**, si le testeur ne peut ni observer ni contrôler les informations qui sont envoyés sur le canal.

  La plus part du temps, lorsque nous travaillons avec des orchestrations, les services Web sont simulés, mais cela n'est pas toujours le cas. Donc, dans la suite du chapitre, on montre comment modifier les spécifications pour prendre en compte ces statu. Inspirés par les travaux dans [vdBRT03], on définit aussi un autre relation de conformité, qu'on utilise quand il y a des actions dans le SUT qui ne peuvent pas être vus par le testeur (à cause de l'architecture du test). Nous finissons le chapitre en présentant nos résultats dans la forme des deux théorèmes, qui ont été définis pour répondre la question suivante: s'il y a un erreur détecté lorsqu'on teste l'orchestrateur en contexte, quelles sont les hypothèses qui doivent être faites pour déterminer que l'erreur est dû à l'orchestrateur et pas à son contexte?.

- Nous finissons la Partie 1 avec le Chapitre 5, qui complètement le Chapitre 4, c'est à dire, nous définissons une relation de conformité qui peut être utilisé pour tester la compatibilité des services Web par rapport à comportements qui sont attendus de la part de l'orchestrateur. Pour faire cela, on génère les comportements attendus pour les services Web à partir de la spécification de l'orchestrateur, en utilisant des transformations mathématiques comme le miroir et la projection. Plus spécifiquement, si tous les services Web sont compatibles par rapport à l'orchestrateur (selon cet relation de compatibilité), alors il n'y aura pas des situations de blocage mortel dans le système.

Dans la Partie II, nous montrons comment exploiter les résultats de la Partie 1, où les spécifications sont exprimés d'un manière concise en utilisant des systèmes à base des notations symboliques.

- Dans le Chapitre 6 nous introduisons les techniques d'exécution symboliques, commençant avec le formalisme *symbolique* que nous utilisons pour modeler les spécifications des orchestrateurs. Après, on présente l'exécution symbolique, qui consiste à exécuter un programme (dans ce cas, les spécifications des systèmes) en utilisant des symboles à la place des données numériques. Le résultat est un arbre qui représente tous les comportements possibles du système.

- Dans le Chapitre 7, nous appliquons applique les techniques symboliques introduites dans le Chapitre 6 pour tester des orchestrateurs en contexte, c'est à dire, on adapte le Chapitre 4 en utilisant des techniques d'exécution symboliques. Dans ce chapitre on introduit aussi un algorithme basé sur des règles pour tester les orchestrateurs. Cet algorithme est basé dans la relation de conformité introduit dans le Chapitre 3 et est guidé par des objectifs du test. L'algorithme exécute essentiellement 3 actions:

  1. il calcule les valeurs pour envoyer au SUT à chaque fois qu'il est nécessaire de le faire de la part du testeur et pour couvrir l'objectif du test,
  2. il observe les réactions du SUT,
  3. et il émet un verdict le plus tôt possible. En plus, l'algorithme émet des verdicts sur des comportements du système que sont dûs à des interactions avec les services Web.

- Dans le Chapitre 8 nous appliquons les techniques symboliques pour générer des comportements pour les services Web, c'est à dire, on adapte le Chapitre 5 dans l'optique symbolique. Nous montrons que les comportements qui sont générés (à partir des techniques de miroir et projection appliquées à des arbres symboliques) peuvent être utilisés pour déterminer si les services Web n'amènent pas à l'orchestration dans une situation de blocage mortel.

- Nous finissons la Partie 3 en présentant notre prototype, qui implémente l'algorithme à base des règles et qui permet de tester des orchestrations des services Web décrites dans WS-BPEL.

References

[ACJY03] R. Anido, A. Cavalli, L. Lima Jr., and N. Yevtushenko. Test suite minimization for testing in context. *Softw. Test., Verif. Reliab.*, 13(3):141–155, 2003.

[BPSM+08] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C, 2008. http://www.w3.org/TR/REC-xml/.

[BvdMFR06] N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, and J.E. Rooda. A Model-based Integration and Testing Method to Reduce System Development Effort. *Electronic Notes in Theoretical Computer Science*, 164(4):13–28, 2006. Proc. of the Second Workshop on Model Based Testing (MBT 2006).

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *WSDL (Version 1.1)*. W3C, March 2001. http://www.w3.org/TR/wsdl.

[CFCB10] T.D. Cao, P. Félix, R. Castanet, and I. Berrada. Online Testing Framework for Web Services. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 363–372, Washington, DC, USA, 2010. IEEE Computer Society.

[CHvRR04] L. Clement, A. Hately, C. von Riegen, and T. Rogers. *UDDI (Version 3.0.2)*. OASIS, October 2004. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

[Cla76] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[Dij79] E. Dijkstra. Structured programming. pages 41–48, 1979.

[EGGC09a] J.P. Escobedo, P. Le Gall, C. Gaston, and A. Cavalli. Examples of testing scenarios for web service composition. Technical Report 09003_LOR, TELECOM & Management SudParis, 2009. http://www.it-sudparis.eu/.

[EGGC09b] J.P. Escobedo, P. Le Gall, C. Gaston, and A. Cavalli. Observability and controllability issues in conformance testing of web service composition. In *Testing of Communicating Systems and Formal Approaches to Software Testing (TESTCOM/FATES)*, volume 5826 of *LNCS*, pages 217–222. Springer, 2009.

[EGGC10] J.P. Escobedo, P. Le Gall, C. Gaston, and A. Cavalli. Testing web service orchestrators in context:a symbolic approach. In *Proc. of Software Engineering Formal Methods (SEFM) '10*. IEEE Computer Society, 2010.

[FGM+99] R. T. Fielding, J. Gettys, J. C. Mogul, H. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, 1999. http://tools.ietf.org/html/rfc2616.

[FT07] L. Frantzen and J. Tretmans. Model-Based Testing of Environmental Conformance of Components. In *Formal Methods of Components and Objects (FMCO)*, number 4709 in LNCS, pages 1–25, 2007.

[FTW06] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In *Intl. Workshops FATES/RV*, volume 4262 of *LNCS*, pages 40–54, 2006.

[GGRT06] C. Gaston, P. Le Gall, N. Rapin, and A. Touil. Symbolic Execution Techniques for Test Purpose Definition. In *Testing of Communicating Systems (TESTCOM)*, volume 3964 of *LNCS*, pages 1–18, 2006.

[GHM+07] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. Nielsen, A. Karmarkar, and Y. Lafon. *SOAP (Version 1.2)*. W3C, April 2007. http://www.w3.org/TR/soap12-part1/.

[GJW04] J. Gortmaker, M. Janssen, and R. Wagenaar. The advantages of web service orchestration in perspective. In *ICEC '04: Proceedings of the 6th international conference on Electronic commerce*, pages 506–515, New York, NY, USA, 2004. ACM.

[HS05] M. Huhns and M. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[Kat99] J.-P. Katoen. Concepts, Algorithms, and Tools for Model Checking, 1999.

[Kho04] A. Khoumsi. Test cases generation for embedded systems, 2004.

[Kin75] J. C. King. A new approach to program testing. In *Intl. Conf. on Reliable Software*, pages 228–233. ACM, 1975.

[L+99] G. Leavens et al. *The Java Modeling Language (JML)*. 1999. http://www.eecs.ucf.edu/ leavens/JML/.

[LVD09] N. Lohmann, E. Verbeek, and R. Dijkman. Petri net transformations for business processes — a survey. pages 46–63, 2009.

[LZCC07] M. Lallali, F. Zaïdi, C., and 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.

[LZCH08] M. Lallali, F. Zaïdi, A. Cavalli, and I. Hwang. Automatic timed test case generation for web services composition. *European Conference on Web Services (ECOWS)*, 0:53–62, 2008.

[MKB09] S. Mitra, R. Kumar, and S. Basu. A Framework for Optimal Decentralized Service-Choreography. In *ICWS*, pages 493–500, 2009.

[NS08] S. Noikajana and T. Suwannasart. Web Service Test Case Generation Based on Decision Table (Short Paper). In *QSIC '08: Proceedings of the 2008 The Eighth International Conference on Quality Software*, pages 321–326, Washington, DC, USA, 2008. IEEE Computer Society.

[Pel03] C. Peltz. Web services orchestration and choreography. In *Computer*, pages 46–52. IEEE Computer Society, 2003.

[RBJ00] V. Rusu, L. Bousquet, and T. Jéron. An Approach to Symbolic Test Generation. In *IFM '00: Proceedings of the Second International Conference on Integrated Formal Methods*, pages 338–357, London, UK, 2000. Springer-Verlag.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[Tre08] Jan Tretmans. Formal methods and testing. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, pages 1–38, Berlin, Heidelberg, 2008. Springer-Verlag.

[UL07] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 edition, 2007.

[vBK06] F. van Breugel and M. Koshkina. Models and Verification of BPEL, 2006.

[vdBRT03] H.M. van der Bijl, A. Rensink, and J. Tretmans. Component based testing with ioco, 2003.

[vdBRT04] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. In *Workshop on Formal Approaches to Testing of Software (FATES)*, volume 2931 of *LNCS*, pages 86–100, 2004.

[vRT04] H.M. van der Bijl, A. Rensink, and G.J. Tretmans. Compositional testing with ioco. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing (FATES)*, volume 2931 of *LNCS*, pages 86–100, Berlin, 2004. Springer Verlag.

[YJC10] T. Yoo, B. Jeong, and H. Cho. A Petri Nets based functional validation for services composition. *Expert Syst. Appl.*, 37(5):3768–3776, 2010.

Abstract

In the last years, the Service-Oriented Architecture (SOA) has gained popularity in the industry due to the advantages it offers: independence (low coupling), reusability and sharing of services. Web services are an example of SOA, which are pieces of software offering functionalities trough the Internet. Due to their acceptance, new ways of using Web services have also emerged. The most popular is the one known as *Web service Orchestration*. Orchestrations are the result of combining different Web services in order to create new, more complete and complex ones. In this type of systems, there is a central Web service guiding the whole process: the *orchestrator*.

In this thesis, we focus on conformance testing of orchestrators *in context*: that is, orchestrators interacting with Web services in the orchestration during the testing phase. Moreover, we restrict to cases where only the specification of the orchestrator is known. That is a usual practical case, since Web services are often developed by companies that do not publish their specifications. Our objective is to determine if, when an error is detected in the orchestration at the testing in context phase, the error is due to a non conformance of the orchestrator. A first contribution of this thesis is to define conformance relations (based on the conformance relation **ioco**) allowing us to characterize conformance in context. Moreover, we provide a classification of the different situations when testing the orchestrator, depending on the nature of the testing architecture. The usage of our conformance relations is strongly coupled to those testing architectures. For the different testing architectures, we state theorems allowing us to relate conformance in context and usual unit conformance in the **ioco** framework. We then show how to put the formal work in practice by means of *symbolic techniques* and of a rule-based online testing algorithm, allowing us to detect non conformance of orchestrators while interacting with orchestrators in context. The main symbolic technique we use is the symbolic execution, whose basic idea is to execute programs using symbols instead of concrete data as input values, and to derive tree-like structures in order to describe all possible computations in a symbolic way.

A directly derived and complementary work is to define Web service testing techniques which allow one to determine if a given Web service can be used together with an orchestrator without leading the resulting orchestration into deadlock situations. Since we only take into account the specification of the orchestrator, this is done only according to the the Web service behaviors expected *from* the orchestrator.

A final contribution of this thesis is the implementation of a prototype which allows us to test implementations of orchestrators in their context of usage. This prototype implements the rule-based online testing algorithm, as well as the rest of technical operations to apply it in a testing architecture aiming at performing this kind of tests with this kind of systems.

Keywords: **ioco**, SOA, test in context, orchestrators, symbolic testing, test case generation.

Résumé

Les Architectures Orientées Services (SOA) se sont imposées depuis quelques années dans la plupart des entreprises par les nombreux avantages qu'elles procurent: faible couplage, réutilisation et le partage des services. Puisqu'ils constituent des logiciels offrant des fonctionnalités via Internet, les services Web s'inscrivent clairement comme une instance de l'architecture SOA. Afin de faciliter encore leur usage, de nouvelles façons d'utiliser les services Web ont émergé. La plus connue d'entre elles est celle connue sous le nom d'orchestration de services Web qui consiste à combiner différents services Web afin d'en créer de nouveau par le biais d'un service central Web, l'orchestrateur, guidant l'ensemble du processus.

Dans cette thèse, nous nous intéressons au test de conformité des orchestrateurs testés dans leur contexte, cest-à-dire interagissant avec des services Web. De plus, nous considérons le cas de figure où seule la spécification de l'orchestrateur est connue. En effet, les services Web sont souvent développés par des sociétés tiers ne publiant pas nécessairement leurs spécifications. Notre objectif est de déterminer si une erreur détectée dans l'orchestration, i.e. l'orchestrateur dans le contexte des services Web invoqués, peut s'expliquer par une non-conformité de l'orchestrateur. Dans le cadre de la relation de conformité connue sous le nom de **ioco**, nous avons précisé quels sont les liens rigoureux entre la conformité de l'orchestrateur au sens **ioco**, et la conformité de l'orchestrateur en contexte. Nous avons effectué cette étude en tenant compte des différentes situations en lien avec l'architecture de test. Ces différentes situations sont reliées aux facilités d'accès et/ou de contrôle des communications échangées entre l'orchestrateur et les services Web. Nous adaptons ensuite ce travail dans le cas où les spécifications des orchestrateurs sont donnés sous la forme de systèmes de transitions symboliques à entrées/sorties. Nous fournissons un algorithme de génération de séquences de test sous la forme d'un ensemble de règle: sous l'hypothèse que le système sous test est constitué de l'orchestrateur et des services Web invoqués au sein de l'orchestration, des verdicts de non-conformité des orchestrateurs sont émis.

Nous avons aussi tiré parti de la donnée de la spécification de l'orchestrateur pour inférer les comportements des services Web tels qu'attendus par l'orchestrateur. Les comportements inférés peuvent servir d'objectifs de test de telle sorte qu'une erreur de conformité mise à jour sur les services Web en question révèlerait alors une situation de blocage (dead-lock) dans une orchestration constitué d'un orchestrateur conforme et des implémentations sous test des services Web.

Nous avons implémenté un prototype de génération de séquences de test des implémentations des orchestrateurs considérés dans leur contexte d'utilisation constitué des services Web. A l'aide de techniques d'exécution symbolique et de résolution de contraintes, ce prototype met en oeuvre des opérations de transformations des arbres d'exécution symbolique des spécifications afin de se placer du point de vue de l'orchestrateur en contexte, et implémente l'algorithme à base de règles.

Mots-clés: ioco, SOA, test en contexte, orchestrateurs, test symbolique, génération des cas de test.

Contents

List of Figures

Chapter 1

General Introduction

Web services are pieces of software offering functionalities to other (remote) machines over the Internet that work based on the Service-Oriented Architecture (SOA). They can be invoked by means of Web related standards (usually, *SOAP* [Gudgin 2007], *UDDI* [Clement 2004], *XML* [Bray 2008], *HTTP* [Fielding 1999], *WSDL* [Christensen 2001]). In the recent years, the usage of Web services has increased due to the flexibility and interoperability among heterogeneous platforms and operative systems that they provide. SOA adds value in terms of low coupling, re-usability of services and sharing; it makes the systems flexible and adaptive in case of changes in the business process and improves the integration of heterogeneous systems [Huhns 2005]. Besides, new ways of using Web services have emerged, by combining them in order to create more complete (and complex) services. This process of re-using and combining Web services is called *Web service composition*, and its main objective is to allow the re-usability of the functionalities proposed by the Web services. This is why this architecture has been widely accepted by the enterprises all over the world: it helps reducing the cost and time to create business processes, and this is the type of systems we work with in this thesis, more specifically, we aim at ensuring their correct behavior by using testing techniques.

Among the different ways of creating new Web service compositions, the two that are the most used are ([Peltz 2003]):

- *Web Service Choreographies* [Mitra 2009], where every Web service involved in the composition interacts at the same level, knowing when and how to communicate with the rest of Web services.

- *Web Service Orchestrations* [Gortmaker 2004], where there is a central component guiding the process in a centralized way, taking decisions based on the answers it gets from the Web services.

There is also the composition of Web services based on the Semantic Web[1], which is not an orchestration but more like a choreography. From the three previous ways of composing Web services, orchestrations are the most popular among enterprises. One of the reasons is that the most used standard for describing an orchestration is the *Web Services Business Process Execution Language* (WS-BPEL) [Alves 2007] (whose current version is 2.0), that is maintained by OASIS[2] (which is a consortium

[1] http://semanticweb.org
[2] http://www.oasis-open.org/

that drives the development, convergence and adoption of open standards for the global information society). Besides, it is based on XLANG (Microsoft, 2001) and WSFL (IBM, 2001), and many well-known enterprises participated in its creation (Microsoft, IBM, SAP, Oracle, Adobe Systems, among others).

Due to the high acceptance of this new technique of combining Web services in order to create new, more complex ones, new ways to assure their correct behavior have been developed [van Breugel 2006]. Besides, orchestrations have special characteristics that are not present in other types of systems (specially, stand-alone systems), and usual testing approaches (i.e., unit testing techniques used with stand-alone systems) do not to take them into account. We address this problem in this thesis, in order to provide new (black-box) testing approaches to help in the validation and verification of orchestrations. Let us examine some characteristics of the orchestrations. Its central component which guides the whole process is known as the *orchestrator*. All the Web services involved in the orchestration interact only with the orchestrator and never between them. Any user (by means of a Web application) interacts with the orchestration also through the orchestrator. Thus, the orchestrator has a very important role in the orchestration, taking decisions according to the answer it gets from the Web services and the inputs sent by the user. We consider orchestrations as particular component-based systems, where there is a central component (the orchestrator) that interacts with the environment (user) and acts as an interface of the system.

Context of our Work

The interest of our work consists precisely in *testing* orchestrators from a particular point or view: by doing it in or out its context of usage. Now, testing consists in executing a system in order to detect errors. Software testing is an activity that consumes great software and time costs in the industry (50-70%) [Noikajana 2008], and so causes software delivery delay. Therefore, the testing activity has to be as much as possible an automated process in order to reduce the time employed in this task. Besides, it has to be done in a way to *ensure*, as most as possible, that the system, once it has been tested, will work correctly. For systems where no failures are allowed, we *have* to make use of formal rigorous approaches in order to provide the desired level of assurance. Thus, mathematical models are used to represent the specification of systems (instead of using an ambiguous language like, for instance, English) and formal techniques are used to test the implementation of a system's specification (the system under examination, that in our case is the orchestrator within an orchestration, and is usually referred to as the *system under test*). This way of testing systems by using models is known as *model-based testing*. Besides, in order to ensure that a model correctly represents the behaviors of a system, a technique called *model checking* has been developed and is currently an active field of work [Fraser 2009]. Briefly, model checking is an automated technique that, given

a model of a system and a property stated in some appropriate logical formalism (in practice, a variation of temporal logic), systematically checks the validity of this property [Katoen 1999]. In this thesis, we do not perform any model checking techniques on the models we use. We assume that our models represent the specifications correctly, but, in the context of a complete validation and verification process, performing model checking is a complementary activity that should be performed together with the work presented in this document. Regarding the formalism chosen to model systems, there are many to choose from [Lohmann 2009], each one offering different advantages (and disadvantages). We can mention some different types of models that have been used for testing purposes, like state-based models (like JML [Leavens 1999]), transition-based models (like FSM [Lallali 2007], LTS [Tretmans 2008]), operational models (like Petri Nets [Yoo 2010]), among others. The chosen model depends on the characteristics of the system that one wants to capture, as well as, of course, the degree of familiarity that one has with a given notation.

From the models of the specifications one can then extract key behaviors (known as *test purposes*) that the system under test has to respect. This is done because one cannot verify *all* the behaviors described in the model of the specification, since that would be a very long, even infinite, task. It follows that test purposes can only take into account a finite subset of all possible behaviors of the system, and therefore, it can never be complete: testing can only show the presence of errors, but not their absence [Dijkstra 1979]. Thus, test purposes are selected in order to test specific aspects of a system: they can aim at testing how fast the system can perform its tasks (performance testing); how does it react when the number of requests made to the system increases greatly (stress testing); how does it reacts when wrong inputs are sent to it (robustness testing); how long can we rely on the correct functioning of the system (reliability testing); and finally, the kind of test we are interested in, does the system do what it should do? (conformance testing), or, more specifically, does the behavior of system under test comply with its specification? More precisely, the specification referred here is actually a behavioral one, i.e., it describes the behavior of the system by means of the functions or operations that it can perform (as opposed to its performance, usability, or reliability). In the rest of this thesis, whenever we refer to the specification of a system, we are referring in fact to its behavioral specification. Conformance testing is the approach we take in this thesis. Besides, the usual configuration when performing conformance testing is that the system under test is *inaccessible*, that is, it is perceived by the tester as a *black-box*. It means that, since we are testing software-based systems, the tester does not have access to the code of the system under test. For the case where we do have access, the testing activity is known as *white box-testing*, and usually consists in ensuring the coverage of the different parts of the code. In black-box testing, tests are designed completely from the system's specification (in practice, by means of a model and test purposes) which describes the expected behavior of the black box [Utting 2007]. Finally, a system can be tested in different scales: in isolation (unit testing); each

component in the system can be tested separately (component testing); the system can be tested in order to ensure that several components work correctly (integration testing); the system is tested when interacting with some other pieces of software (that can encapsulate it or play the role of stubs) (testing in context); or the system can be tested as a whole (system testing).

Our approach consists in performing testing in context for orchestrators. While performing test in context [Khoumsi 2004], the targeted system under test is *embedded* within a system and interacting with some other components of it, which constitute precisely its context: the whole system is composed of the targeted system under test and of all the components in its context. Different assumptions can be made regarding the components in the context. The most classical hypotheses are that components in the context are supposed to be faulty-free, i.e., they conform to their specifications (that are thus available), and that communication with the components works correctly. Besides, the usual approach is that the tester interacts with the component under test only by means of its context [Anido 2003].

Our Approach

Now, let us say more about applying black-box model-based conformance testing in context in order to determine if an orchestrator behaves accordingly to its specification. As introduced previously, orchestrations are a special kind of systems where the orchestrator guides the whole process. Thus, the system that is under test is composed of the orchestrator plus the Web services (or components) interacting with it. Those Web services constitute the context of the orchestrator, and we aim at ensuring the conformance of the orchestrator with respect to its specification while interacting with its context. Therefore, in order to apply a classical testing approach to this type of systems, one would take into account the specifications of the orchestrator and of all the Web services interacting with it. Then, we would obtain a model representing the specification of the entire system. Since we use mathematical models to represent each specification (of the orchestrator and of each of the Web services), the specification of the entire system can be obtained by means of mathematical manipulations over all those models (like cartesian product, union of sets, etc.). Nevertheless, there may be two problems when doing this: first, the specification of all the components in the system (orchestrator and Web services) is not always known (especially, under the form of a formal notation). This can happen, for instance, because there can be some Web services that are developed by third parties who do not want to expose the internal behaviors of their products. The second problem is the one known as the *state explosion* problem, and it is caused because *combining* the different models of the specification of the components in the system may result in an extremely large model, which becomes difficult to manipulate [Valmari 1998]. The main hypothesis that we will make in our approach is to assume that the specification of the orchestrator *is always* available, and, in order to tackle the previous problems, we propose an approach to test

the orchestrators by taking only into account its specification but not the ones of the rest of Web services, even if orchestrators will interact with Web services during the testing process. More strictly, we do consider the interface's specification of the Web services, (that is often available in practice by means of their associated WSDL file), however, such a WSDL file does not provide a behavioral specification, since WSDL describes the services that the Web service offers but says nothing about their internal behavior.

Thus, in this work, by taking only the specification of the orchestrator, we propose an approach to test the orchestrators studying a black-box model-based conformance testing approach in the particular case where the orchestrators are interacting with the rest of the system. This clearly represents a specialized case of testing in context, where the tester can only control the orchestration via the orchestrator. This is precisely one of the contributions of our work, since most of the works concerning testing orchestrations and component based systems either simulate the context of the orchestrator, or make the assumption that the specification of all the Web services (or components) is available ([Cao 2010, Lallali 2008, Braspenning 2006, van der Bijl 2003b]). Besides, we have chosen the **ioco** conformance relation in order to test orchestrators in context [Tretmans 1996b]. **ioco** (which stands for *input/output conformance*) choses to work on the basis of labeled transition systems, and its basic idea is to test the conformance of a system under test with respect to its specification by examining if, after any sequence of interactions observed in the system under test (and that are valid according to the specification, i.e., specified) any further observation is also specified. If there is an observation in the system under test that is not specified, then we can conclude that it does not conform to its specification. Since 1996, several works have been based on this conformance relation [Rusu 2000, Gaston 2006, van der Bijl 2004], which has been widely accepted and extended on several domains of application [Schmaltz 2008, Frantzen 2007, Frantzen 2006b]. Besides, it has the advantage of authorizing nondeterministic specifications, because it does not require the implementations to consider all the specified outputs but only some of them. Moreover, **ioco** authorizes situations where there is underspecification, because if an input is not specified, the implementation has the freedom to do *whatever* it wants.

In order to address the fact of not taking the specification of the Web services into account, we complement the testing of orchestrators in context by dealing with another common problem when testing orchestrations: how can we be sure to select the Web services that are compatible with the orchestrator? By exploiting the specification of the orchestrator, we show how to elicit behaviors which are expected from the Web services by the orchestrator. These elicited behaviors can be seen as a partial specification for the Web services, since it is the orchestrator that makes use of them. A Web service could offer more functionalities than the ones requested by the orchestrator; in our approach we choose to neglect those functionalities and focus only on the ones of interest for the orchestrator.

Thus, we propose an approach to test the correct behavior of orchestrations by testing the orchestrator in context and by testing the compatibility of Web services, all of this by taking into account only the specification of the orchestrator. By doing so, we aim at tackling two common problems when testing systems with multiple components: the explosion problem and the lack of availability of specifications.

Plan of the Document

The thesis is structured in two parts. In Part I we present the theoretical framework which grounds our approach.

- We begin in Chapter 2 by taking a look into the state of the art regarding the type of systems we are interested in: Web services orchestrations.
- In Chapter 3 we introduce the conformance relation that we use. We first present the formalism that we use in order to model the specification and implementation of the orchestrators. Moreover, we present some results obtained in [van der Bijl 2003a] when using the chosen conformance relation with component-based systems, as well as some other related works.
- We continue in Chapter 4 by presenting the modified version of the conformance relation, that we define in order to test orchestrators in context. This conformance relation allows us to take into account pieces of information about the context of the orchestrators by means of some modifications performed over the specification of the orchestrator. Inspired by the results obtained in [van der Bijl 2003a], we define another version of the conformance relation, that is used when there are some actions in the system under test that cannot be observed. We end this chapter by presenting our results by means of two theorems that aim at answering to the question: if there is an error detected while testing the orchestrator in context, under which hypotheses does it mean that the error precisely belongs to the orchestrator?.
- We finish Part I with Chapter 5, which is the complementing work of Chapter 4, that is, we present an approach to test the compatibility of Web services with respect to an orchestrator. More specifically, we show how to test if a given Web service does not lead an orchestrator into a deadlock state.

In Part II we show how to exploit our results of Part I, where specifications are expressed in a concise way by means of Symbolic Transition Systems.

- In Chapter 6 we introduce the symbolic execution techniques starting with the *symbolic* formalism we use to model the specification of the orchestrators. These models are symbolically executed generating a structure that represents all the valid behaviors of the system.

- Chapter 7 is the symbolic version of Chapter 4, showing how to use the symbolic execution technique in order to test the conformance of orchestrators in context. Here we also present a rule-based algorithm (which is based on the conformance relation introduced in Chapter 3) to test orchestrators in context in which generation of test data sequences is guided by test purposes to be covered. The algorithm performs essentially 3 actions: (1) it computes input values to be sent to the system under test each time it is necessary in order to cover test purposes, (2) it observes system under test reactions and (3) it computes a verdict as soon as possible.

- Chapter 8 is the symbolic version of Chapter 5, consisting on eliciting behaviors that can be used to test the compatibility of Web services with the orchestrator. We show how to generate test cases in order to determine if Web services do not lead the orchestration into a deadlock situation.

- We finish Part II by presenting our prototype tool, which implements the rule-based algorithm allowing to test Web service orchestrations in the form of WS-BPEL process implementations.

Part I

Test in Context of Component-based Systems via its Interface

Chapter 2

Web Service Orchestrations

Contents

2.1 Introduction

Web services are pieces of software offering specific operations (which together are called services) through the Web. The Web service architecture is the architectural model used to make the Web services available to the service consumers so they can use them in a standard way. This architecture is in turn based on the *Service-Oriented Architecture* (SOA), which allows applications to use functionalities (called services) from other applications operating in heterogeneous environments. Web services are currently used all over the Internet, and in recent years they started to be combined in order to obtain more complex services. This re-usage and combination of existing Web services in order to create a new, more complex one, is called *Web service composition*. Moreover, among the different types of existing Web service compositions, we are especially interested in the one known as *orchestration*. An orchestration of Web services is a Web service composition where there is a central Web service, called *orchestrator*, guiding the composition. From a practical point of view, Web service orchestrations can be seen as component-based systems with some particularities, as discussed later in this chapter.

There are different technologies for creating new services using of existing ones. Among the most known and used ones today there is the *Web Services Business Process Execution Language* WS-BPEL standard [Alves 2007]. WS-BPEL, allows

one to create business processes by making use of available Web services. A business process delivers a service supposed to by used by third parties.

In this chapter we introduce all of the concepts necessary to understand what a Web service composition is. We start by introducing the Service-Oriented Architecture in Section 2.2. Web services are then introduced in Section 2.3. The notion of *Web service orchestration* is described in Section 2.4 (as well as component-based systems with similar characteristics). In Section 2.5, we discuss techniques to specify orchestrators, which is a key aspect of our work. We conclude this chapter with Section 2.6.

2.2 Service-Oriented Architecture

Service-Oriented Architecture (SOA) [Huhns 2005] refers to an architecture based on components offering services (these components are themselves called services) and clients using them. In this context, a service is an application providing access to some of its functionalities to other applications. Moreover, the services are supposed to be independent one from the other. Then, the main objective of Service-Oriented Architecture is to allow system designers to reuse functionalities, which is why this architecture has been widely accepted by enterprises all over the world. Besides the great advantage of re-using services, SOA can also be used in the opposite way; that is, to decompose a complex service into multiple more simple ones, so that a client can combine them later in order to have the complex service as a result. Therefore, SOA adds value in terms of low coupling, re-usability of services and sharing; it makes the systems flexible and adaptive in case of changes in the business process and improves the integration of heterogeneous systems. Loose coupling among services means that the mutual dependencies are minimized by the use of standardized interfaces. One of the main advantages of SOA is that its configuration can change dynamically as needed and without affecting the process's results.

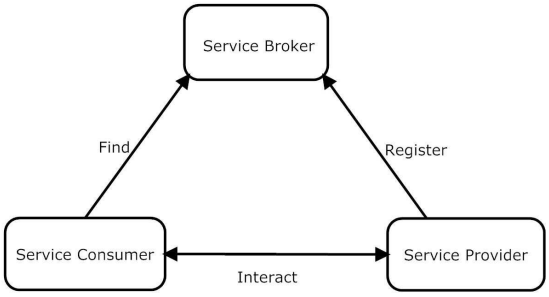


Figure 2.1: Service-Oriented Architecture example.

The most common example of SOA instantiation is the one provided by the Web service architecture. As shown in Figure 2.1, a Service-Oriented Architecture

is typically composed of three basic parts:

a) The service consumer, which is the final user making use of the service. In the case of Web services, in order to use the service, they must have an interface with which the user can interact. This interface is known as *Web application*.

b) The service broker, which is the *place* where the service consumer can search for a service. In the case of Web services, such broker is known as *Universal Description, Discovery and Integration* (UDDI)[Clement 2004]. The UDDI can be seen like a *yellow pages* service for phone numbers. In order to find a telephone number, there has to be a place where one can search for it (by using the name of the person). Thus, in order to find a Web service, it has to be listed in the UDDI (by using the services).

c) The service provider, which, in the case of the Web service architectures are the Web services themselves. Web services can be seen as (remote) pieces of software offering some basic functionalities. They can be anywhere over the Internet and are meant to provide a service to a *machine* and not directly to the user, as commonly misunderstood. In order to be found by the clients, Web service providers must publish the services they offer in the UDDI. In practice, however, it may happen that the Web service providers never publish their services so anyone can find it. This can happen if Web services are not meant to be public. In this case the service consumers must know how to access them by other ways —typically the case of the private enterprises.

2.3 Web services

Web services are pieces of software offering services throughout the Internet or any other type of network. According to the World Wide Web Consortium[1], a Web service *is a software system designed to support inter-operable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically* WSDL *[Christensen 2001]). Other systems interact with the Web service in a manner prescribed by its description using* SOAP *[Gudgin 2007] messages, typically conveyed using* HTTP *[Fielding 1999] with an* XML *[Bray 2008] serialization in conjunction with other Web-related standards.*[2] Let us say more about those Web-related standards:

- **WSDL**. Stands for *Web Service Description Language* and is a format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.
- **SOAP**. Is a format for the XML-based information which can be used for exchanging structured and typed pieces of information between peers in a decentralized, distributed environment.

[1] http://www.w3.org/
[2] http://www.w3.org/TR/ws-gloss/

- **HTTP**. Stands for *HyperText Transfer Protocol*, and is the protocol used in the Internet. It is an application-level protocol for distributed, collaborative, hypermedia information systems.
- **XML.** Stands for *eXtensible Markup Language*, and is a text format that has an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.
- **WSDL-S**. Even if not explicitly mentioned in the definition of Web services, Web Service Semantics (WSDL-S) [Akkiraju 2005] is a recent standard that defines a mechanism to associate semantic annotations with Web services that are described using WSDL, and it was created because the current WSDL standard operates at the syntactic level and lacks the semantic expressiveness needed to represent the requirements and capabilities of Web services.

Remark 1 *At the time of writing this thesis, the REST [Fielding 2000] description was gaining a lot of acceptance around the Internet. However, no specifications have been published for creating compositions of Web services described in REST.*

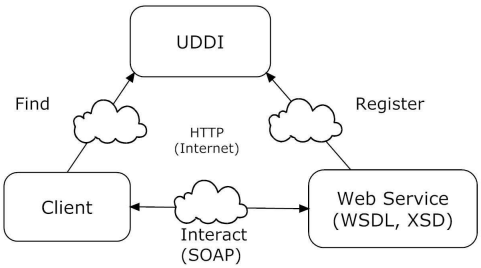


Figure 2.2: Web service architecture example.

Figure 2.2 depicts the typical Web service architecture. Web services are usually invoked over the Internet, but they can be also used in any other kind of network (LAN, VPN, etc.). The client in this case can be a Web application or another Web service. The service broker instance is the UDDI and the client interacts with the Web service by sending and receiving SOAP messages according to its WSDL description file.

2.4 Web Service Compositions: Orchestrations

Web services can be combined in order to provide a new, more complex service. As discussed in Chapter 1, in this thesis we focus on the Web service compositions known as orchestrations. An orchestration is a Web service composition where there is a central component guiding the whole process, called the *orchestrator*. In fact, one cannot avoid thinking of an orchestration as an orchestra, the instrumental

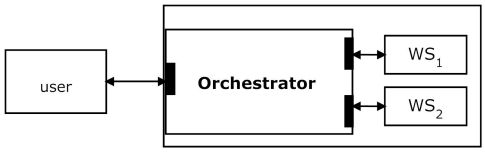


Figure 2.3: Orchestration's structure example.

ensemble, where the conductor guides all the musicians. Figure 2.3 depicts a typical structure of an orchestration. All the Web services (WS_1 and WS_2) involved in the orchestration communicate only with the orchestrator, which in turn is the only one that communicates with the user. It is the orchestrator who invokes the Web services when needed, and waits for their answers. If the Web services do not answer in time, do not answer at all, or there is an error when interacting with a Web service, the orchestrator can take some actions. Thus, the whole process can be reduced to the invocation of Web services by the orchestrator, who in turn makes decisions based on the answers it receives. The orchestrator is usually instantiated when invoked the first time, and the instance is destroyed when it finishes executing all of its tasks.

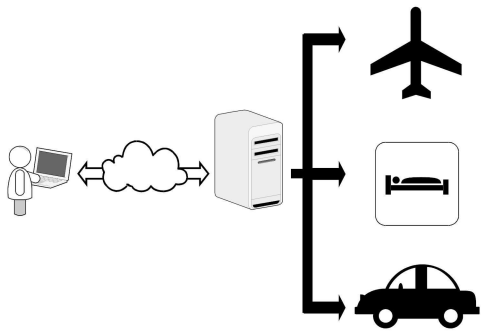


Figure 2.4: Online Travel Agency orchestration example.

Example 2.4.1 *A typical example when introducing an orchestrator is the one of the* Online Travel Agency *(OTA) depicted in Figure 2.4: a user wants to plan his vacation and he decides to use the online travel agency OTA. By using OTA, the user can reserve his flight, accommodation and transport; all of this from a single, centralized Web application. Now, technically speaking, each one of those services is provided by three different Web services, and what the OTA offers is just the composition of those three Web services. Thus, OTA handles the different situations when, for instance, for the dates the user wants, only the flight and hotel are available, or when the hotel service sends the answer too late.*

2.4.1 Web Services Business Process Execution Language

WS-BPEL is a language for specifying business process behaviors based on Web services. A business process delivers a service in the context of a work organization. Briefly, WS-BPEL defines how the interaction between Web services is coordinated to achieve a business goal. A business process described with WS-BPEL takes advantage of the interoperability available between applications implemented using Web standards (SOAP, WSDL, UDDI). Then, WS-BPEL serves as a standard process integration model, allowing the interaction between different Web services in a loosely coupled, platform independent way, and also giving a formal description of the message exchange protocols used by the partners in the interactions; taking into account different concepts like data-dependent behavior and recovery from exceptional conditions. Interactions with different partners occur through Web services interfaces and are encapsulated in what is called *partnerLinks*. One can think of a partnerLink as the the communication channel (link) used to interact with a Web service (partner).

Example 2.4.2 *Figure 2.5 depicts the simplified version of the WS-BPEL description of the OTA example. We can notice that there are definitions for: the partnerLinks, that is, the Web services with which the orchestrator will interact; the variables that are going to be used to store and exchange data with the partners; and the behavior of the orchestrator.*

WS-BPEL, in the end, creates a new Web service, and has also an associated WSDL file which exposes its operations, the data types and the links with the partners with which it interacts. WS-BPEL offers basic and structured activities in order to describe the process behavior. Among the basic activities there are the *receive*, *reply*, and *invoke* ones, as well as the *assign* activity. Among the structured activities there are the *if* activity, as well as the *sequence*, *while*, and *repeatUntil* ones. Finally, WS-BPEL offers the option to handle the possible errors that may arise when invoking the Web services. These errors may be due to the lack of response of the Web services (by using timeouts), wrong answers from the Web services, error message answers from the Web services, etc.

2.4.1.1 Discussion

In the WS-BPEL related literature, it is common to refer to the WS-BPEL description of a business process as an *orchestration*. However, strictly speaking, WS-BPEL describes only the behavior of the central component, that is, the orchestrator. In the sequel, we differentiate between the term orchestrator, to refer to description of the business processes, and orchestration to refer to the whole system: the orchestrator *and* the Web services. Furthermore, in the sequel we no longer work with orchestrators described in WS-BPEL, but in a symbolic formalism ($IOSTS$, for Input/Output Symbolic Transition Systems), and we assume that is possible to translate most usual Web service description languages into it (as it has been shown for WS-BPEL in [Bentakouk 2009]).

```
<process name="OTA">
    <partnerLinks>
        <partnerLink name="client"/>
        <partnerLink name="FlightService"/>
        <partnerLink name="HotelService"/>
        <partnerLink name="CarService"/>
    </partnerLinks>
    <variables>
        <variable name="dates"/>
        <variable name="flightOK"/>
        <variable name="hotelOK"/>
        <variable name="carOK"/>
        <variable name="finalStatus"/>
    </variables>

    <sequence name="main">
        <receive name="receiveInput" partnerLink="client" variable="dates"/>
        <invoke name="invokeFS" partnerLink="FlightService" inputVariable="dates"/>
        <switch>
            <case condition="getVariableData('flightOK') = 'OK'" >
                <assign>
                    <from expression="string('Flight: Reserved')"/>
                    <to variable="finalStatus"/>
                </assign>
            </case>
            <otherwise>
                <assign>
                    <from expression="string('Flight: Not reserved')"/>
                    <to variable="finalStatus"/>
                </assign>
            </otherwise>
        </switch>
        // The same process repeats itself for the Hotel and Car Services
        <invoke partnerLink="client" inputVariable="finalStatus"/>
    </sequence>
</process>
```

Figure 2.5: Simplified WS-BPEL code for the OTA example.

As we describe later in this document, our objective is to provide an approach to test orchestrators in general, which can be simply seen as a central components guiding the process execution of other components of a system.

2.4.2 Component-based Systems with Interfaces

Orchestrations, in a more general way, can be seen as component-based systems which have some special characteristics: there is a central component guiding the system, every component communicates only with the central one, and the user interacts with the system by means of the central component. WS-BPEL is the most common way to describe orchestrations; nevertheless, these type of systems can be described in several ways. Figure 2.6 depicts the structure of a component-based system with an interface. In this figure there are three components interacting with the orchestrator: $Comp_1$, $Comp_2$, and $Comp_3$.

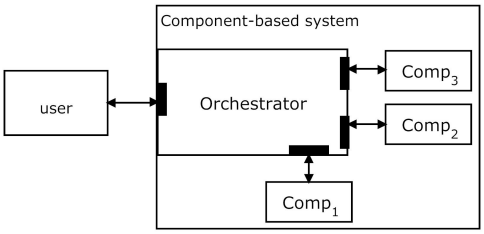


Figure 2.6: Component-based system with an interface.

The architecture is similar for the case of Figure 2.4, but in this case the *user* does not necessarily interact with the system via the Internet, nor does the orchestrator with the rest of the components. This communication can happen by means of other communication channels like dedicated networks or any other physical link. Thus, the *user* cannot see the interactions of the *orchestrator* with the rest of the components, $Comp_1$, $Comp_2$ and $Comp_3$. From his point of view, the system behaves exactly the same as if there were no remote components. The orchestrator, and not the user, has to deal with the different situations where the components do not react as required or do not react at all. It may also happen that there are errors not in the components but in the communication channels represented in the figure by the double-oriented arrows. In our work we, do not make a separation between the component and the communication channel used to communicate with the orchestrator: the orchestrator can interact with the Web service *only* by means of the communication channel, that, we suppose, works correctly. For example, telling that a Web service sends a value to the orchestrator means that the orchestrator receives that value on its port connected to the communication channel.

2.5 Modeling Orchestrators

In this section we discuss different formalisms used to model orchestrators. For illustrative purposes, Figure 2.7 depicts a model of the OTA example. It represents its behavioral description in the form of states and transitions representing the different progressions in the system, as well as the communication with the Web services. The model represents only the behavior of the orchestrator: the transition from state q_0 to q_1 represents the user sending the dates d to OTA, the transitions after that one represent the exchange of messages between OTA and the Web services, sending them the date d and receiving their answers a, to finally compute the answer f that is sent back from OTA to the user (transition from $q7$ to $q0$).

There are many different formalisms that can be used to model the orchestrator of our example, as well as any other type of system. They can be modeled by means of:

1. *Automata* [Pu 2006, Wombacher 2004],
2. *Finite State Machines* [Lallali 2008, Chow 1978, Nakajima 2006],

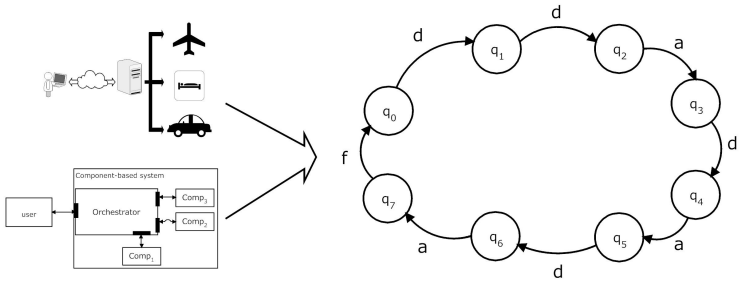


Figure 2.7: Modeling orchestrators.

3. *Petri Nets* [Yoo 2010, Lohmann 2007, Dumas 2005, Yang 2005],
4. *UML* diagrams [Cambronero 2007, Li 2005],
5. *Process Algebra* [Cámara 2006, Viroli 2004], etc.[3]

In our work we chose to work with *Input/Output Symbolic Transition Systems* (*IOSTS*) [Gaston 2006], for which we associate semantics in terms of *Input/Output Labeled Transition Systems* (*LTS*). *IOSTS*s have already been used for testing purposes of Web services orchestrations and WS-BPEL. More specifically, in [Bentakouk 2009], authors have made a detailed work on modeling most of the activities and notions in WS-BPEL with Symbolic Transition Systems (a variation of our *IOSTS*s), taking various notions of WS-BPEL into account, like events (inputs and outputs) over partner links, time related activities, etc. Authors also provide a detailed table showing how to translate each WS-BPEL activity into its *IOSTS* equivalent representation. Thus, we can rely on the fact that *IOSTS*s are a good option for working with orchestrations and WS-BPEL.

2.6 Conclusion

In this chapter we have introduced the basic concepts of Web service systems (orchestrations). Orchestrations have a central component called orchestrator guiding the entire system. In an orchestration, Web services do not interact directly with each other, but only with the orchestrator. The user interacts only with the orchestrator. Our goal is to test orchestrators.

In the next chapter we examine techniques that are used to test orchestrations and component-based systems.

[3]Surveys of various formalisms, specifically related to Web service compositions, can be found in [Lohmann 2009, van Breugel 2006].

Chapter 3

Conformance Testing

Contents

3.1 Introduction

The type of systems that we consider in this thesis is the one known as *orchestrations*. Orchestrations are composed of a central component called *orchestrator*, playing the role of an interface between users and other components called Web services. In this thesis we are interested in testing the orchestrators of such systems. In this chapter we introduce the conformance relation that we use to test orchestrations. The conformance relation is called **ioco** [Tretmans 1996b] and is used to perform black-box testing of *reactive systems*. A reactive system is a system that continuously interacts with its environment.

Conformance testing aims at determining if an implementation of a given system is correct with respect to a specification of reference. The specification is assumed to be given in a formal language, and the implementation under test also needs to be represented in a formal way. Implementations are physical, real objects, that are in principle not amenable to formal reasoning. This representation of a real object by means of a formal, mathematical one, is called a *model*. We make the assumption that the implementations under test *can* be modeled. This hypothesis is known as *test hypothesis* [Heymer 2007, Tretmans 1996b].

ioco[Tretmans 1996b], stands for input/output conformance and refers to the implementation relation (notion of correctness) on which the theory has been built. The chosen formalism to model specifications and implementations is known as *labeled transition systems* (LTS). LTSs are structures consisting of states with transitions, labeled with actions, between them. In our work, however, we do not

work directly with LTSs but with $IOLTS$s, which are an specialization of LTSs that distinguish between messages that are sent and messages that are received. This will prove pertinent when testing orchestrations. Nevertheless, $IOLTS$s can behave as particular LTSs defined in [Tretmans 1996b] and [van der Bijl 2005], so we can assume that the results provided by those works can also be directly applied with $IOLTS$s.

In Section 3.2, we present the technical preliminaries of the approach, namely, the LTSs and the $IOLTS$s structures and operations that we use in order to test orchestrators. Then, we present the **ioco** conformance relation. We begin by introducing the case of unit testing and then we introduce the case of testing component-based systems, since, as we discussed in the previous chapter, orchestrations can be considered as an especial type of component-based systems, thus, we can also use **ioco** to test them. In Section 3.3, we examine some of the related works. We finish the chapter with the conclusions in Section 3.4.

3.2 Conformance Testing: ioco

In this section we present the conformance relation **ioco** [Tretmans 1996b], that is the basis of our work. The terminology of **ioco** stands for *input output conformance*. In Section 3.2.1, we introduce the automata known as *Labeled Transition Systems* (LTS) [Tretmans 1996b] and their specializations for the **ioco** theory. Labeled transition systems are basically automata with labels over their transitions. Those labels denote actions. LTS are specialized to be efficient at representing Systems Under Test (SUT) and specifications, with the goal of defining a conformance relation relying only on properties concerning *traces* of the SUT, that is, sequences of emissions and receptions built by a tester interacting with the SUT. **ioco** is thus dedicated to black-box testing. In Section 3.2.2, we introduce the **ioco** theory as formal basis for black-box unit testing approaches. In Section 3.2.3, we present some results ([van der Bijl 2003b]) concerning component-based systems in the frame of the **ioco** theory, and we discuss the practical impact of those results in particular regarding to our goal of testing orchestrations in context.

3.2.1 Labeled Transition Systems

LTSs are automata whose transitions are associated with *labels* denoting observable actions of the system. Those actions may typically be communication messages or observable internal actions. We use the label τ to denote an internal action. The states of an LTS are abstractions of real states of the modeled system, and firing a transition results on a state evolution associated to the occurrence of the action labeling the transition.

Definition 3.2.1 (LTS) *Let L be a so-called set of labels.*
A Labeled Transition System (LTS) over L *is a tuple $(Q, init, Tr)$ where:*

- Q *is a set of* states
- $init \in Q$ *is the* initial state
- $Tr \subseteq Q \times L \times Q$ *is a set of* transitions

Notation 3.2.1.1 *In the following, for any LTS $\mathbb{G} = (Q, init, Tr)$ over L, we use the notations $Q_{\mathbb{G}}$, $init_{\mathbb{G}}$, and $Tr_{\mathbb{G}}$ in order to refer, respectively, to the set of states Q, the initial state $init$, and the set of transitions Tr.*

In the same way, for any transition $tr \in Tr_{\mathbb{G}}$ of the form (q, a, q'), we use the notations $source(tr)$, $target(tr)$ and $act(tr)$ in order to refer, respectively, to the q, q' and a.

We represent an LTS in the standard way, that is, by means of a directed, edge-labeled graph where nodes represent states and edges represent transitions. Transitions are represented with an arrow $\rightarrow$ from their source state to their target state.

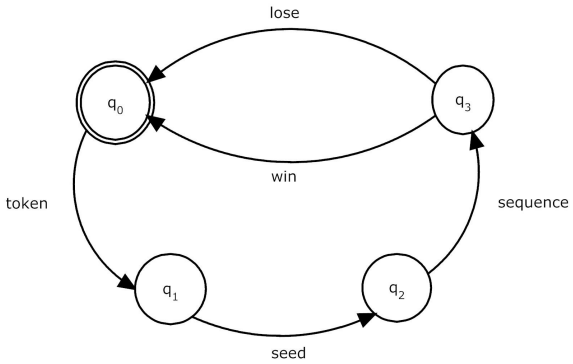


Figure 3.1: LTS for the Slot Machine example.

Example 3.2.1 *Let us introduce the* Slot Machine *example. Figure 3.1 depicts the specification of its orchestrator by means of an LTS. The set of states is $\{q_0, q_1, q_2, q_3\}$, where q_0 is the initial state (and depicted differently from the rest). The set of labels is: $\{token, seed, sequence, win, lose\}$.*

The functional behavior of a Slot Machine is the following: it receives a token from the user and, according to a randomly generated sequence, gives him a prize or not. In our example, we consider that the Slot Machine is composed of two components: the slot machine's interface (SM) and the Sequence Generator service (SG). Moreover, SM is the orchestrator of the system, which means that it controls the whole process by interacting with the user and with SG, acting and taking decisions

based on their inputs as introduced in Section 2.4. More precisely, the orchestrator (SM) behaves as follows:

The user enters a token into the corresponding interface of the Slot Machine, SM. SM will then randomly generate both a seed based on the token, and the winner sequence based on that seed. Next, it will send the seed to SG, which in turn will randomly generate the sequence (in this case is just one number) for the user. SG will then send the user's sequence back to SM so the later can compare the user's sequence with respect to the winner sequence. Finally, it will send the final message back to the user through the system's interface. This message can say 'you win*', if the user's sequence is equal to the winner sequence, or 'you* lose*' otherwise.*

In the figure, messages are represented by labels. Thus, for instance, the transition $q_0 \xrightarrow{token} q_1$ *represents the user sending the token to the slot machine's interface SM, and the transition* $q_1 \xrightarrow{seed} q_2$ *represents SM sending the seed to the Sequence Generator service SG.*

Behaviors of LTSs are characterized from their associated *traces*. Traces are possible successions of communication actions that are specified by an LTS. In order to give the formal definition of traces, we introduce the notion of *paths* of an $IOLTS$. A path of an LTS is a succession of transitions that can be fired sequentially.

In the following, for any set S, we note S^* the set of words over S, which contains the empty word ε, and such that for any a in S and m in S^*, $a.m$ is in S^* (by convention, $a.\varepsilon$ is a, and '.' (dot) denotes the concatenation of words).

Definition 3.2.2 (Paths of $\mathbb{G}$) *Let L be a so-called set of labels, and $\mathbb{G} = (Q, init, Tr)$ be an LTS over L.*

The set of paths *of $\mathbb{G}$ is the set $Path(\mathbb{G}) \subseteq Tr^*$, whose elements, together with their associated* target *states, $target(p)$, are defined as follows:*

- *the empty word ε is in $Path(\mathbb{G})$, and its target state, $target(\varepsilon)$, is $init_{\mathbb{G}}$*
- *for any $p \in Path(\mathbb{G})$, for any $tr \in Tr$, $p.tr \in Path(\mathbb{G})$ if and only if $target(p) = source(tr)$, and in this case $target(p.tr) = target(tr)$*

Example 3.2.2 *Let us consider the LTS introduced in Example 3.2.1. An example of a path of that LTS is the sequence of transitions going from state q_0 to state q_3 and then from q_3 back to q_0, that is, the sequence:*

$(q_0, token, q_1).(q_1, seed, q_2).(q_2, sequence, q_3).(q_3, win, q_0).$

In this example, we chose the case where the user sequence is equal to the winner sequence.

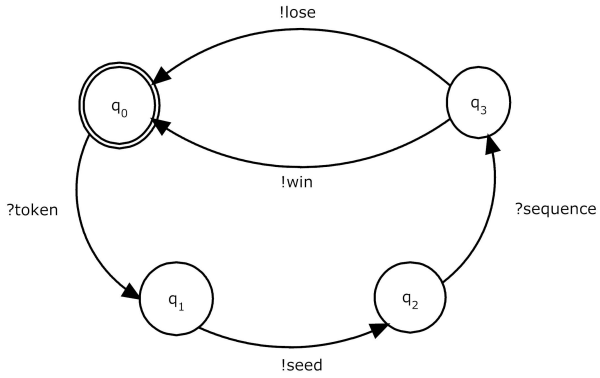


Figure 3.2: LTS for the Slot Machine example with input/output labels.

The *trace of a path* is the sequence of communication actions (inputs or outputs) of the path. In the sequel, we introduce a distinguished symbol τ as a particular label denoting an internal action. That symbol is not necessarily present in sets of labels associated to LTSs.

Definition 3.2.3 (Traces of $\mathbb{G}$) *Let L be a so-called set of labels, and $\mathbb{G}$ be an LTS over L.*

The set of traces of $\mathbb{G}$, *denoted $Traces(\mathbb{G})$, is the subset of L^*, defined as $\{traces(p) \mid p \in Path(\mathbb{G})\}$, where $traces(p)$ is defined as follows:*

- *if p is ε then $traces(p) = \varepsilon$*
- *if p is of the form $p'.tr$, with $act(tr) = \tau$, then $traces(p)$ is $traces(p')$*
- *if p is of the form $p'.tr$, with $act(tr) \neq \tau$, then $traces(p)$ is $traces(p').act(tr)$*

Example 3.2.3 *Let us consider the path of example 3.2.2:*

$(q_0, token, q_1).(q_1, seed, q_2).(q_2, sequence, q_3).(q_3, win, q_0)$.

Its trace is the sequence of actions of the transitions the path, that is, the actions of the transitions going from state q_0 to q_3 and from q_3 back to q_0 through the transition with the label win:

$token.seed.sequence.win$.

As in [Tretmans 1996b, van der Bijl 2003b], we also use the distinction between *inputs* and *outputs*. Outputs, identified by the symbol !, denote values sent form the system to the environment. Inputs, identified by the symbol ?, correspond to values sent from the environment to the system.

In [Tretmans 1996b, van der Bijl 2003b], sets of labels L are simply partitioned into two subsets L_I and L_U, whose elements are respectively inputs and outputs.

For the sake of readability, an element of L_I (respectively of L_U) is denoted $?a$ (respectively $!a$) to signify that it denotes an input (respectively an output).

Example 3.2.4 *Figure 3.2 depicts the LTS of the orchestrator (interface) of the Slot Machine example (Ex. 3.2.1), where the set of labels is partitioned into inputs and outputs. This partition allows us to better understand the behavior of the system: SM can receive inputs in transitions* $q_0 \xrightarrow{?token} q_1$ *and* $q_2 \xrightarrow{?sequence} q_3$*, while it can emit outputs in transitions* $q_1 \xrightarrow{!seed} q_2$*,* $q_3 \xrightarrow{!win} q_0$*, and* $q_3 \xrightarrow{!lose} q_0$*.*

Furthermore, the user and SG constitute the environment of SM, and we can distinguish the sense *of the messages: whether they are sent from the environment or emitted to it.*

We slightly adapt the definition of inputs and outputs to also denote the communication channels used to exchange messages. That information will be useful in the sequel because we need to identify whether or not internal communications between components of a system are observable. From a practical point of view, such a message is observable if the communication channel it uses is instrumented in order to allow one to observe messages transmitted on it. To reflect that fact at the theoretical level, we need to denote channels. Therefore, an input (respectively, output) is of the form $c?v$ (respectively, $c!v$) and denotes the reception (respectively, the emission) of the value v through the channel c. In the sequel of this thesis, we suppose that a set M of values is given.

Definition 3.2.4 (Communication actions) *Let C be a set whose elements are called communication channels.*

The set of communication actions over C*, denoted $Act(C, M)$, is the set $I(C, M) \cup O(C, M) \cup \{\tau\}$, where:*

- *elements of $I(C, M)$ are of the form $c?v$, with $c \in C$ and $v \in M$, and are called* inputs
- *elements of $O(C, M)$ are of the form $c!v$, with $c \in C$ and $v \in M$, and are called* outputs

We now proceed to define the abstraction-based model that we use in this thesis. That model is name *IOLTS* (for *Input/Output Labeled Transition System*) and is an adaptation of the *IOTS* models introduced in [Tretmans 1996b, van der Bijl 2005], which is done in order to reflect our point of view on the messages.

Definition 3.2.5 (*IOLTS*) *Let C be a set of communication channels.*

An Input/Output Labeled Transition System (*IOLTS*) over C *is a labeled transition system over $Act(C, M)$.*

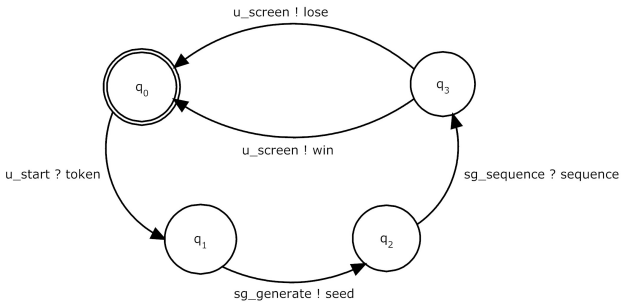


Figure 3.3: $IOLTS$ $\mathbb{G}$ for the Slot Machine example.

Notation 3.2.5.1 *$IOLTS$ are just particular LTSs as introduced in Definition 3.2.1, thus, the notations concerning LTS can also be adapted for $IOLTS$. That is, in the following, for any $IOLTS$ $\mathbb{G} = (Q, init, Tr)$ over C, we use the notations $Q_{\mathbb{G}}$, $init_{\mathbb{G}}$ and, $Tr_{\mathbb{G}}$ in order to refer, respectively, to the set of states Q, the initial state $init$, and the set of transitions Tr.*

In the same way, for any transition $tr \in Tr_{\mathbb{G}}$ of the form (q, a, q'), we use the notations $source(tr)$, $target(tr)$, and $act(tr)$ in order to refer, respectively, to q, q', and a.

Example 3.2.5 *Figure 3.3 depicts the $IOLTS$ $\mathbb{G}$ of the orchestrator of the Slot Machine example, introduced in Example 3.2.1. By using this notation, we can distinguish the set of communication channels:*

$\{u\_start, sg\_generate, sg\_sequence, u\_screen\}$.

Moreover, since we also partition the set of labels into inputs and outputs, we can notice that the communication channels u_start and u_screen are used to communicate with the user (u), and the communication channels $sg\_generate$ and $sg\_sequence$ are used to communicate with the Sequence Generator service (SG). If we restrict M to the set of values $\{token, seed, sequence, win, lose\}$, we can get the following set of communication actions: $\{(u\_start?token), (sg\_generate!seed), (sg\_sequence?sequence), (u\_screen!win), (u\_screen!lose)\}$, which is precisely the ones used by $\mathbb{G}$[1].

For testing reasons, it is important to identify under which situations a system can be silent. Definition 3.2.5 of $IOLTS$ does not take into account these situations. We make the classical hypothesis ([Tretmans 1996b]) that when there is a transition from which no outputs or internal actions τ are possible, i.e., when the system cannot proceed autonomously without inputs from its environment, then silence is mandatory. Moreover, it is the only circumstance under which the silence is

[1] In practice, for the set of communication actions, we should consider all possible combinations of communication channels and values of M. We restrict it for the sake of readability.

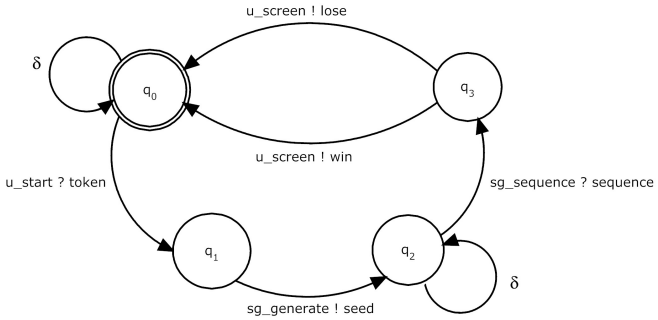


Figure 3.4: Quiescence enrichment for the Slot Machine example, $\mathbb{G}_\delta$.

allowed. In order to reflect this hypothesis, $IOLTS$ are enriched with new transitions representing those silent situations. These transitions are qualified as *quiescence* transitions.

Definition 3.2.6 (Quiescence enrichment for $\mathbb{G}$) *Let C be a set of communication channels and $\mathbb{G}$ be an $IOLTS$ over C.*

The enrichment by quiescence *of $\mathbb{G}$ is the LTS, denoted $\mathbb{G}_\delta$, over the set of labels $Act(C, M) \cup \{\delta\}$ such that:*

- $Q_{\mathbb{G}_\delta} = Q_{\mathbb{G}}$
- $init_{\mathbb{G}_\delta} = init_{\mathbb{G}}$
- $Tr_{\mathbb{G}_\delta} = Tr_{\mathbb{G}} \cup Tr_\delta$ *where Tr_δ is defined by:*

  for any $q \in Q_{\mathbb{G}_\delta}$, $(q, \delta, q) \in Tr_\delta$ if and only if there is no $(q, act, q') \in Tr_{\mathbb{G}}$ with act is τ or of the form $c!v$

The notations concerning $IOLTS$s can also be adapted for $IOLTS$s enriched with quiescence.

Example 3.2.6 *Based on the $IOLTS$ $\mathbb{G}$ of Figure 3.3, Figure 3.4 depicts the quiescence enrichment of $\mathbb{G}$ according to Definition 3.2.6. Note that in states q_0 and q_2, there are no output nor τ-transitions leaving from them, so the system has to be quiescent: it could happen that the user does not send any token to SM or that the sequence is not sent from SG. In such cases, SM (the orchestrator) cannot evolve by itself since no internal actions nor outputs can be fired, and so, we make that lack of reaction visible.*

It is important to note that this situation of quiescence is an accepted *behavior of the system. Figure 3.4 represents the specification of an orchestrator, then if the implementation is quiescent in any of those states, then it would be a* valid *behavior. This would not be the case if the implementation was quiescent state in, for example, state q_1.*

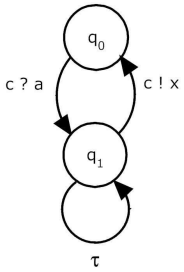


Figure 3.5: Not strongly responsive $IOLTS$.

In practice, the action δ denotes the expiration of a timer whose duration is specified by an expert. Loops characterized by quiescent transitions represent successive observations of the timer expiration without observing any outputs from the system.

Before going any further, it is important to give one hypothesis we make regarding the $IOLTS$s that we use. For technical reasons, we only consider *strongly responsive* $IOLTS$s as in [van der Bijl 2003a]. An $IOLTS$ is strongly responsive if it always eventually enters in a quiescent state, i.e., if it does not have infinite τ or output transitions.

Definition 3.2.7 (Strongly responsive $IOLTS$) *Let $\mathbb{G} = (Q, init, Tr)$ be an $IOLTS$ over C.*

Let $Live(\mathbb{G})$ be the greatest subset of $Path(\mathbb{G})$ verifying that for all $p \in Live(\mathbb{G})$, $\exists\, tr \in Tr$, with $act(tr) \in O(C, M) \cup \{\tau\}$, and such that $p.tr \in Live(\mathbb{G})$.

$\mathbb{G}$ is strongly responsive *if and only if $Live(\mathbb{G}) = \emptyset$*

Let us consider Figure 3.5. It depicts an $IOLTS$ which is not strongly responsive. The system could enter in an infinite τ loop from q_1 that cannot be distinguishable from a quiescent situation. Since from state q_1 there is always the possibility to fire the τ transition, the quiescent situation as introduced in Definition 3.2.6 cannot be allowed in that state. However, from the tester point of view, the system could be perceived as silent. That is why we consider that the $IOLTS$s that we work with are strongly responsive. From hereon, even if not specified, all considered $IOLTS$s are assumed to be strongly responsive.

As in [Tretmans 1996b], and in order to take into account the quiescent situations, we define the notion of *suspension traces* of an $IOLTS$, which is no more than the set of traces of an $IOLTS$ enriched by quiescence. Moreover, the set of traces defines the behavior of the $IOLTS$. This behavior is also known as *semantics*.

Definition 3.2.8 (Semantics of an $IOLTS$) *Let $\mathbb{G}$ be an $IOLTS$ over C.*

The set of suspension traces of $\mathbb{G}$, *denoted $STraces(\mathbb{G})$, is $Traces(\mathbb{G}_\delta)$. The* semantics of $\mathbb{G}$ *is defined by $STraces(\mathbb{G})$.*

Example 3.2.7 *Let us consider the $IOLTS$ enriched by quiescence of example 3.2.6. An example for a suspension trace of $\mathbb{G}_\delta$ is the one that goes from the state q_0 (taking quiescence into account) to state q_3 (again taking quiescence into account) and then back to q_0:*

$\delta^*.u\_start?token.sg\_generate!seed.\delta^*.sg\_sequence?sequence.u\_screen!win.$

3.2.2 Unit Testing

ioco defines, in a formal way, a notion of correctness of an implementation of the specification of a system, SUT (for System Under Test), with respect to the specification. Therefore, we need a mathematical representation of the SUTs and their specifications.

Specifications are nothing more than $IOLTS$s. Before defining an SUT, we introduce the technical definition of *input complete $IOLTS$*: an $IOLTS$ is input complete if it is completely specified for input actions, i.e., if every input in $I(C, M)$ is accepted by any state of the transition system.

Definition 3.2.9 (Input complete $IOLTS$) *Let $\mathbb{G}$ be an (possibly enriched by quiescence) $IOLTS$ over C.*

$\mathbb{G}$ is an input complete *$IOLTS$ if and only if for any $q \in Q_{\mathbb{G}}$, for any $a \in I(C, M)$, there exists $tr \in Tr_{\mathbb{G}}$ of the form (q, a, q')*

In fact, an input complete $IOLTS$ is a reformulation in our technical context of the IOTS notion introduced in [Tretmans 1996b, van der Bijl 2003b]. As in [Tretmans 1996b, van der Bijl 2003b], we make the assumption that an SUT can be seen as an $IOLTS$ that we do not know, but for which we can discover associated traces by interacting with it. That is, we do not need to have the $IOLTS$ of the SUT, we just assume that it could by modeled by it. As in [Tretmans 1996b, van der Bijl 2003b], we assume that this $IOLTS$ is input complete to specify that it never refuses any inputs (sent by the tester). It does not mean that the SUT will *use* it (functionally). We assume such $IOLTS$s to be enriched by quiescence to reflect possible silence of the system under test. Finally, since we are using a black-box approach, we assume that there are no internal actions τ in the SUT. From the point of view of the tester, the SUT is a black-box that is characterized by the sequence of outputs that he can observe according to the inputs that he sends. It does not make any sense to talk about internal actions of the SUT, since, even if the SUT performs such internal actions, they are not perceived by the tester.

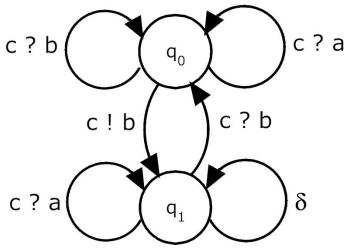


Figure 3.6: SUT: input complete, quiescence-enriched, and without τ-transitions.

Definition 3.2.10 (SUT System Under Test) *A* System Under Test over C *is an input complete $IOLTS$ enriched by quiescence, satisfying:*
$\forall tr \in Tr_{SUT}, act(tr) \neq \tau$

Let us consider Figure 3.6. It depicts an SUT for the signature over the set of communication channels $\{c\}$, and under the assumption that $M = \{a, b\}$. We can notice that: in both states, the SUT accepts all inputs (it is input complete); when it can not evolve by itself (in state q_1, because it is waiting for the input b) it goes into a quiescent state (it is quiescence enriched); and that, since it is defined by the interactions with the tester, it has no internal actions (τ-transitions).

Let us now introduce the **ioco** conformance relation. The definition is based on [Tretmans 1996b] but defined here over $IOLTS$s.

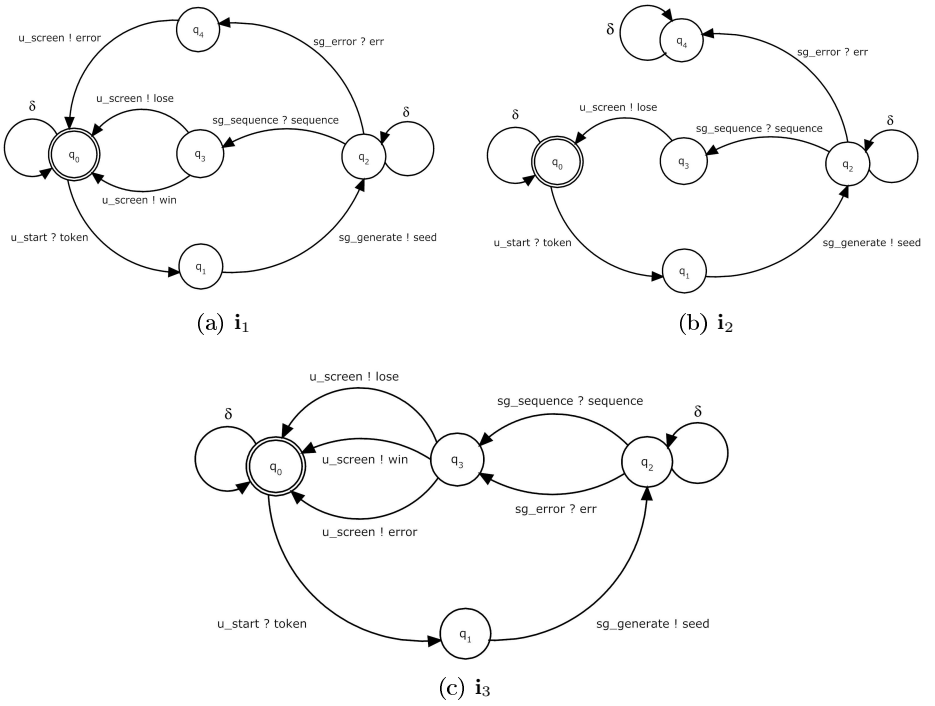
Definition 3.2.11 (ioco conformance relation) *Let $\mathbb{G}$ be an $IOLTS$ and SUT be an system under test, both of them over C.*

SUT ***ioco*** $\mathbb{G}$ *if and only if* $\forall \sigma \in STraces(\mathbb{G})$, $\forall o \in O(C, M) \cup \{\delta\}$
$\sigma.o \in Traces(SUT) \Rightarrow \sigma.o \in STraces(\mathbb{G})$

Intuitively, an SUT conforms to $\mathbb{G}$ if and only if after any specified trace of input/output actions sequence that have been built by interacting with the SUT, any observation made on the SUT is specified in $\mathbb{G}$.

Example 3.2.8 *Figure 3.7 depicts different implementations (SUTs) for the Slot Machine example*[2]*. Based on the $IOLTS$ $\mathbb{G}_\delta$ of Figure 3.4, we can notice that* $\mathbf{i}_1$ *of Figure 3.7* ***ioco*** *$\mathbb{G}_\delta$, because after any specified sequence of events, the outputs emitted by* $\mathbf{i}_1$ *are also specified. Even if the implementation chooses to handle the possible situations where there is an error in the implementation of the Sequence Generator service SG, it does not interfere with the fact that after any specified sequence of events, any output in* $\mathbf{i}_1$ *is also specified.* $\mathbf{i}_2$ *also* ***ioco*** *$\mathbb{G}_\delta$. Even if the implementation never gives a prize to the user, and even if it also takes the*

[2] We consider that the implementations are input complete. This is not explicitly shown in the figure for the sake of readability



(a) $\mathbf{i}_1$ (b) $\mathbf{i}_2$

(c) $\mathbf{i}_3$

Figure 3.7: **ioco** examples.

possible case of error into account but does nothing, every output of $\mathbf{i}_2$ *after any specified trace is also specified. Finally,* $\mathbf{i}_3$ *does not* ***ioco*** *(~~**ioco**~~)* $\mathbb{G}_\delta$*, because after the specified sequence* $u\_screen?1.sg\_generate!6.sg\_sequence?4$*, the non-specified* $u\_screen!error$ *can be observed.*

Finally, **ioco** has been used as the basis of different approaches. Among them we can mention taking time into account ([Schmaltz 2008]), environmental conformance ([Frantzen 2007]), and adapting it to the context of symbolic models ([Jeannet 2005]). **ioco** has also been applied to test Web services ([Frantzen 2009, Frantzen 2006a]). Some of these woks are discussed in Section 3.3, but mentioned here in order to accentuate the wide acceptance of the **ioco** theory. Besides, several tools have also implemented an **ioco**-based test generation algorithm, among which there are TGV [Jard 2005], TestGen [He 1999] and TorX [Tretmans 2003].

3.2.3 Compositional Testing

The previous definition of **ioco** (Definition 3.2.11) is mainly targeted to perform conformance testing of stand-alone systems. In this section we present the results of the works provided by [van der Bijl 2003b]. for **ioco** applied to component-based systems. Component-based systems are systems which are in turn composed of more

systems, called components. Together, components constitute a new more complex system offering new functionalities. The integration of components can be modeled mathematically by synchronizing their actions and internalizing their common actions. This synchronization and internalization may be typically defined as two separate operations: a *product*, which represents the system resulting of communications between components, and the *hiding* operation, which restricts observability of internal actions.

Definition 3.2.12 (Product between two $IOLTS$s) *Let $\mathbb{G}$ and $\mathbb{H}$ be two $IOLTS$ respectively over C_1 and C_2. The* synchronous product *of $\mathbb{G}$ and $\mathbb{H}$, denoted $\mathbb{G} \bigotimes \mathbb{H}$, is an $IOLTS$ $(Q, init, Tr)$ over $C_1 \cup C_2$ such that:*

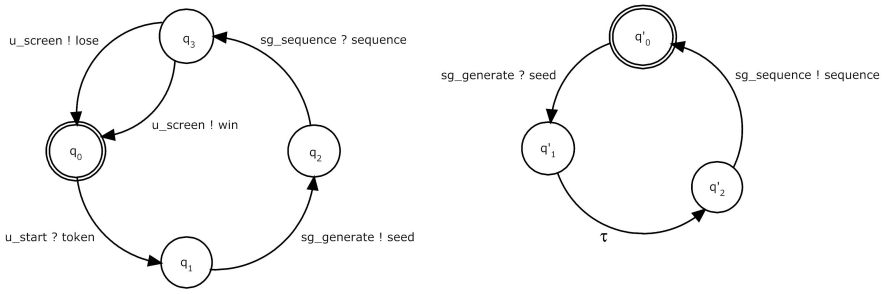
- $Q = Q_{\mathbb{G}} \times Q_{\mathbb{H}}$
- $init = (init_{\mathbb{G}}, init_{\mathbb{H}})$
- *Tr is defined as follows:*
  - *If $(q_1, c!v, q_1') \in Tr_{\mathbb{G}}$ and $(q_2, c?v, q_2') \in Tr_{\mathbb{H}}$, such that $c \in C_1 \cap C_2$, then $((q_1, q_2), c!v, (q_1', q_2')) \in Tr$*
  - *If $(q_1, c?v, q_1') \in Tr_{\mathbb{G}}$ and $(q_2, c!v, q_2') \in Tr_{\mathbb{H}}$, such that $c \in C_1 \cap C_2$, then $((q_1, q_2), c!v, (q_1', q_2')) \in Tr$*
  - *For any $(q_1, a, q_1') \in Tr_{\mathbb{G}}$ where a is τ or is of the form $c?v$ or $c!v$ with $c \notin C_1 \cap C_2$, then, for any $q_2 \in Q_{\mathbb{H}}$, $((q_1, q_2), a, (q_1', q_2)) \in Tr$*
  - *For any $(q_1, a, q_1') \in Tr_{\mathbb{H}}$ where a is τ or is of the form $c?v$ or $c!v$ with $c \notin C_1 \cap C_2$, then for any $q_2 \in Q_{\mathbb{G}}$, $((q_1, q_2), a, (q_1', q_2)) \in Tr$*

As shown in Definition 3.2.12, the transitions of the resulting $IOLTS$ are defined according to the following intuitions:

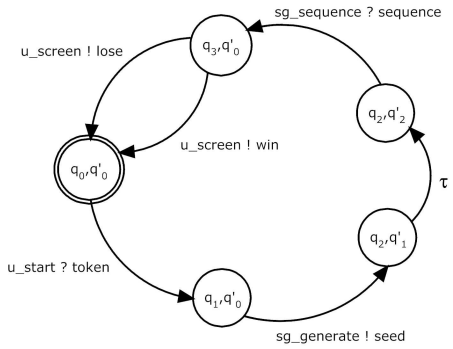
- When a component emits through a channel, and when the other component is ready to receive, the communication occurs and is considered as an emission at the system level.
- Actions that do not correspond to communications between components execute asynchronously.

Example 3.2.9 *Figure 3.8(a) depicts two $IOLTS$ $\mathbb{G}$ and $\mathbb{F}$. $\mathbb{G}$ represents the specification of the slot machine's interface, whereas $\mathbb{F}$ represents the specification of the Sequence Generator service. The $IOLTS$ of Figure 3.8(b) depicts the resulting $IOLTS$, $\mathbb{G} \bigotimes \mathbb{F}$, after applying Definition 3.2.12:*

1. *Transition $(q_0, u\_start?token, q_1)$ of $\mathbb{G}$ does not synchronize with $\mathbb{F}$.*
2. *Transition $(q_1, sg\_generate!seed, q_2)$ of $\mathbb{G}$ synchronizes with the transition of $\mathbb{F}$ $(q_0', sg\_generate?seed, q_1')$.*



(a) Two $IOLTS$: $\mathbb{G}$ and $\mathbb{F}$.



(b) $\mathbb{G} \otimes \mathbb{F}$

Figure 3.8: Product example.

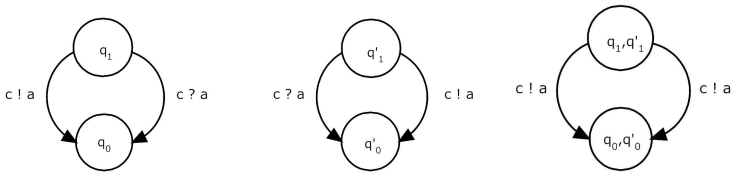
3. *The internal action* τ *of* $\mathbb{F}$ *in transition* (q'_1, τ, q'_2) *does not synchronize with* $\mathbb{G}$.

4. *Transition* $(q_2, sg\_sequence?sequence, q_3)$ *of* $\mathbb{G}$ *synchronizes with the transition of* $\mathbb{F}$ $(q'_2, sg\_sequence!sequence, q'_0)$.

5. *Finally, transitions* $(q_3, u\_screen!lose, q_0)$ *and* $(q_3, u\_screen!win, q_0)$ *of* $\mathbb{G}$ *do not synchronize with* $\mathbb{F}$.

It may happen that even if both $IOLTS$ are strongly responsive, the resulting $IOLTS$ is not. Consider Figure 3.9: the two $IOLTS$s depicted in Figure 3.9(a) are strongly responsive, whereas their product, depicted in Figure 3.9(b), is not. Thus, we consider only $IOLTS$s for which the synchronous product is strongly responsive.

The hiding operation consists in internalizing certain communication actions by transforming them into the action τ.



(a) Two strongly responsive $IOLTS$.

(b) Resulting not strongly responsive $IOLTS$

Figure 3.9: Product according to Definition 3.2.12.

Definition 3.2.13 (Hiding operation over an $IOLTS$) *Let $\mathbb{G} = (Q, init, Tr)$ be an $IOLTS$ over C, and let $\mathcal{C} \subseteq C$.*

$Hide(\mathcal{C}, \mathbb{G})$ is the $IOLTS$ over $C \setminus \mathcal{C}$ defined as follows:

- $Q_{Hide(\mathcal{C},\mathbb{G})} = Q_{\mathbb{G}}$
- $init_{Hide(\mathcal{C},\mathbb{G})} = init_{\mathbb{G}}$
- $Tr_{Hide(\mathcal{C},\mathbb{G})}$ *is defined as follows:*
  - *for any transition $(q, a, q') \in Tr_{\mathbb{G}}$ with a of the form $c \triangle v$, with $\triangle \in \{?, !\}$ and $c \in \mathcal{C}$, $(q, \tau, q') \in Tr_{Hide(\mathcal{C},\mathbb{G})}$*
  - *for any transition $(q, a, q') \in Tr_{\mathbb{G}}$ with a of the form $c \triangle v$, with $\triangle \in \{?, !\}$ and $c \notin \mathcal{C}$, or $a = \tau$, $(q, a, q') \in Tr_{Hide(\mathcal{C},\mathbb{G})}$*

Example 3.2.10 *Figure 3.10 depicts the hiding operation applied to the $IOLTS$ $\mathbb{G} \bigotimes \mathbb{F}$ of Figure 3.8. In this example, $\mathcal{C} = \{sg\_generate, sg\_sequence\}$, thus, communication actions going through those channels are perceived as internal actions (τ-transitions).*

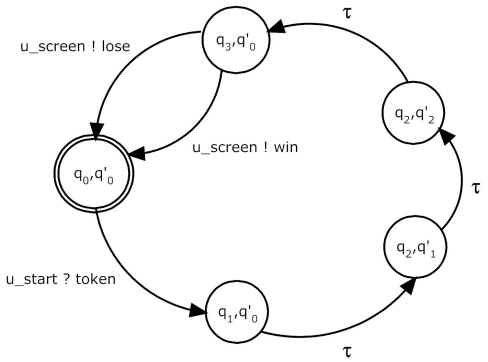


Figure 3.10: Hiding operation of Figure 3.8(b) according to Definition 3.2.13.

The extension of the **ioco** testing framework to component based systems intends basically to address the state explosion problem induced by the product of the specifications of components ($IOLTS$s). Briefly, in [van der Bijl 2003b], the authors address the following question: if single components of a system conform to their specifications, what can be said concerning conformance of the whole system according to its specification? The answer is that the composed system conforms to its specification with the hypothesis that the product and hiding operation are well implemented and that specifications of the components are input complete. This results in two theorems:

The first one states that, if the specifications of the components are modeled by input complete $IOLTS$s, and if the implementations of the components conform to their specifications, then the synchronous product of the implementations also conforms to the synchronous product of the specifications.

Theorem 3.2.1 *Let $\mathbb{S}_1$ (respectively $\mathbb{S}_2$) be an $IOLTS$ over C_1 (respectively C_2), and $\mathbb{I}_1$ (respectively $\mathbb{I}_2$) be an SUT over C_1 (respectively C_2).*

Then,

$$\mathbb{I}_1 \ \textbf{\textit{ioco}} \ \mathbb{S}_1 \wedge \mathbb{I}_2 \ \textbf{\textit{ioco}} \ \mathbb{S}_2 \Rightarrow \mathbb{I}_1 \bigotimes \mathbb{I}_2 \ \textbf{\textit{ioco}} \ \mathbb{S}_1 \bigotimes \mathbb{S}_2$$

The second theorem shows that, if the specification of a system is modeled by an input complete $IOLTS$, then, if an implementation conforms to its specification, the result is preserved trough the hiding operation.

Theorem 3.2.2 *Let $\mathbb{S}$ be an input complete $IOLTS$ defined over C, let $\mathbb{I}$ be an SUT over C, and let $\mathcal{C} \subseteq C$.*

Then,

$$\mathbb{I} \ \textbf{\textit{ioco}} \ \mathbb{S} \Rightarrow Hide(\mathcal{C}, \mathbb{I}) \ \textbf{\textit{ioco}} \ Hide(\mathcal{C}, \mathbb{S})$$

However, from the previous theorems, authors of [van der Bijl 2003a] show that if the input complete hypothesis is omitted, the results of Theorems 3.2.1 and 3.2.2 do not hold anymore.

Underspecification can be present in the specification as underspecified inputs or outputs. The underspecification of outputs is always explicit. For any state all possible outputs are specified. The underspecification of inputs is implicit. It is not mandatory to consider all possible inputs at a given state (and if any input is not considered, the resulting $IOLTS$ is not input complete). The idea behind the underspecification of inputs is that the specification does not characterizes how the implementation deals with those inputs: after an underspecified input in the implementation, *anything* can happen. More specifically, after any state reached after an unspecified input, every action from the label set is correct. This situation is called a *chaotic* behavior [van der Bijl 2003a].

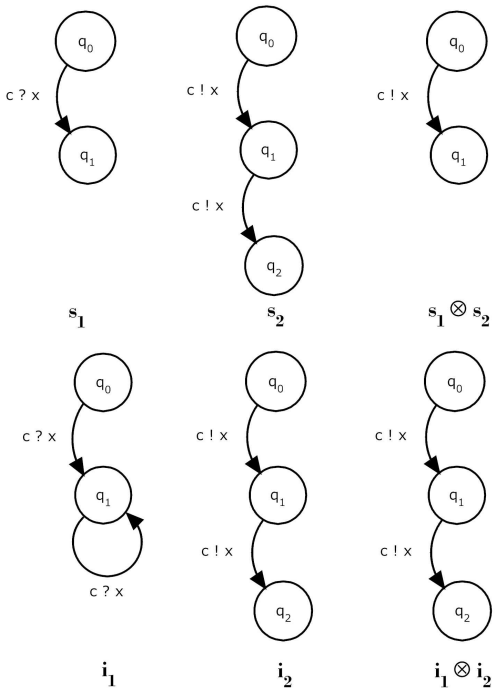


Figure 3.11: Underspecification situation for the product.

To help explaining the problems that arise due to the underspecification of inputs when working with component-based systems, let us consider Figure 3.11. The three upper images depict specifications of two given components, $\mathbf{s}_1$ and $\mathbf{s}_2$, and their product, $\mathbf{s}_1 \otimes \mathbf{s}_2$. The three lower images depict two possible implementations $\mathbf{i}_1$ and $\mathbf{i}_2$ of, respectively, $\mathbf{s}_1$ and $\mathbf{s}_2$, and their product $\mathbf{i}_1 \otimes \mathbf{i}_2$. For the sake of readability, quiescence enrichment (specifications and implementations) and input completeness (implementations) are not depicted in the figure. We can notice that, according to Definition 3.2.11, $\mathbf{i}_1$ **ioco** $\mathbf{s}_1$ and $\mathbf{i}_2$ **ioco** $\mathbf{s}_2$. However, we can notice that $\mathbf{i}_1 \otimes \mathbf{i}_2$ ~~**ioco**~~ $\mathbf{s}_1 \otimes \mathbf{s}_2$, since there exists an output $c!x$ in $\mathbf{i}_1 \otimes \mathbf{i}_2$ after $c!x$, while there is no output after $c!x$ in $\mathbf{s}_1 \otimes \mathbf{s}_2$.

A consequence of the definition for the product is that underspecification of inputs of some components may induce underspecification of internal communications at the system level. Since those internal actions are denoted as outputs, an occurrence of such a communication at the implementation level breaks **ioco**. Considering again Figure 3.11, even though $\mathbf{s}_2$ can output a second x, it cannot do so in $\mathbf{s}_1 \otimes \mathbf{s}_2$, because $\mathbf{s}_1$ cannot input the second x. $\mathbf{s}_1$ is underspecified for $c?x$ in q_1: even though the implementation $\mathbf{i}_1$ is allowed to accept a second $c?x$ as input, the correct behavior after the second x is not specified.

Now, concerning the hiding operation, let us consider the Figure 3.12. The left side image depicts the specification of a given system, $\mathbf{s}$, and a possible im-

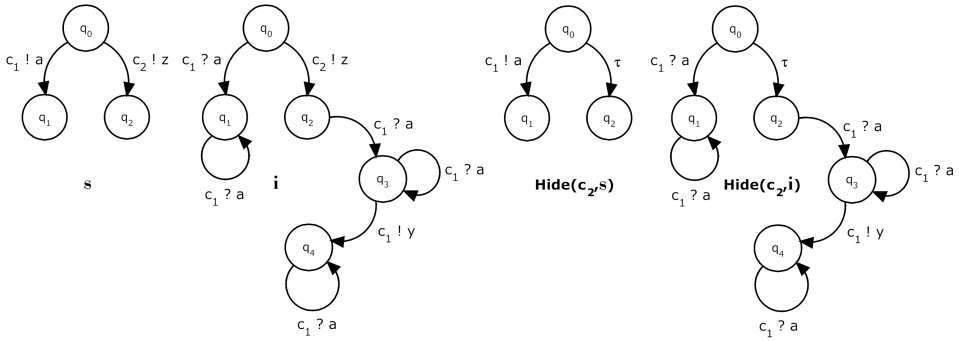


Figure 3.12: Underspecification and hide operation example.

plementation, **i**, both with the input set $\{c_1?a\}$ and output set $\{c_1!x, c_1!y, c_2!z\}$. For the sake of readability, quiescence enrichment (specifications and implementations) and input completeness (implementations) are not depicted in the figure. We can notice that **i ioco s.** The right side image depicts the result of hiding c_2 in both **s** and **i:** we get $Hide(\{c_2\}, \mathbf{s})$ and $Hide(\{c_2\}, \mathbf{i})$. We can notice that $Hide(\{c_2\}, \mathbf{i})$ ~~**ioco**~~ $Hide(\{c_2\}, \mathbf{s})$, because the implementation $Hide(\{c_2\}, \mathbf{i})$ may emit $c_1!y$ after $c_1?a$, while in $Hide(\{c_2\}, \mathbf{s})$ this action is forbidden.

Analysis shows that **s** is underspecified for $c_1?a$ in q_2 (i.e., after $c_2!z$) because it says nothing about how an implementation should behave after the trace $c_2!z.c_1?a$. The proposed implementation has an unspecified output $c_1!y$ after $c_2!z.c_1?a$ (which does not break **ioco**). However, if c_2 becomes unobservable due to hiding, then traces $c_2!z.c_1?a$ and $c_1?a$ collapse and become indistinguishable. In this case, $Hide(\{c_2\}, \mathbf{s})$ specifies that after $c_1?a$, only δ is allowed; however, $Hide(\{c_2\}, \mathbf{i})$ still has the unspecified output $c_1!y$. Thus, hiding creates confusion about what part of the system is underspecified.

The discussion above explains why implementations and specifications are *supposed* to be input complete in Theorems 3.2.1 and 3.2.2.

In order to relax the input complete hypothesis, authors in [van der Bijl 2003a] propose to adapt the conformance relation **ioco.** This results on the conformance relation **uioco**: the idea is to restrict the set of traces over which we reason when testing the conformance of the implementations. This set of traces is denoted $UTraces$ and is the same as $STraces$ but without the underspecified traces.

Definition 3.2.14 (Specified traces of $\mathbb{G}$) *Let $\mathbb{G}$ be an IOLTS over C.*

The set of specified traces of $\mathbb{G}$, *denoted $UTraces(\mathbb{G})$, is the set of all $\sigma \in STraces(\mathbb{G})$ such that there do not exist two paths p and p_1 of $\mathbb{G}_\delta$ such that:*

- *σ can be decomposed in $\sigma_1.a.\sigma_2$ with $a \in I(C,M)$*
- *σ is the trace of p*
- *σ_1 is the trace of p_1*
- *for any $tr \in Tr_{\mathbb{G}_\delta}$ with $source(tr) = target(p_1)$, $\sigma_1.a$ is not the trace of $p_1.tr$*

Then, **uioco** is simply **ioco** restricted to $UTraces$.

Definition 3.2.15 (uioco conformance relation) *Let $\mathbb{G}$ be an IOLTS and SUT be an system under test, both of them over C.*

SUT ***uioco*** $\mathbb{G}$ *if and only if* $\forall \sigma \in UTraces(\mathbb{G})$, $\forall o \in O(C,M) \cup \{\delta\}$ $\sigma.o \in Traces(SUT) \Rightarrow \sigma.o \in STraces(\mathbb{G})$

By using only the specified traces of the $IOLTS$s, authors in [van der Bijl 2003a] relax the input complete hypothesis of Theorem 3.2.3.

Theorem 3.2.3 *Let $\mathbb{S}_1$ (respectively $\mathbb{S}_2$) be an IOLTS over C_1 (respectively C_2), and $\mathbb{I}_1$ (respectively $\mathbb{I}_2$) be an SUT over C_1 (respectively C_2).*

Then,

$$\mathbb{I}_1 \textit{ uioco } \mathbb{S}_1 \wedge \mathbb{I}_2 \textit{ uioco } \mathbb{S}_2 \Rightarrow \mathbb{I}_1 \bigotimes \mathbb{I}_2 \textit{ uioco } \mathbb{S}_1 \bigotimes \mathbb{I}_2$$

By using Theorem 3.2.3, we see that in order to assert that a component-based system behaves correctly (under the hypothesis that the communication between the components is correctly modeled by the synchronous product), it is only necessary to test each component with respect to its specification: in our context, the orchestrator and each Web services, which can be naturally considered as components of a system (orchestration). Note that the result is no useful to conclude something on the whole system if an error is discovered during the unit testing phase of $\mathbb{I}_1$ or $\mathbb{I}_2$.

3.3 Related Work

The component-based systems testing framework proposed in [van der Bijl 2003b] is technically close to our approach and was carefully examined throughout the chapter. In this section, we present a brief overview of contributions which are technically less close to our proposal than [van der Bijl 2003b], but which focus on various aspects of Web service systems testing, as well as component-based ones. Finally, we

discuss the differences between problematics addressed by those contributions and those addressed by our approach.

Among the works concerning testing of Web service orchestrations, we can mention [Cao 2010, Lallali 2008, Li 2005, Mayer 2006, Yuan 2006, Cao 2009]. They use different notations and focus on different aspects of the orchestrations (like taking time into account), but what is important to us is that they all simulate Web services supposed to interact with the orchestration. In order to simulate the Web services, it is necessary to have their specification, or at least some behavioral description. Besides that, and strictly speaking, simulating all the Web services interacting with the orchestrator corresponds to performing some kind of unit testing on the orchestrator. For instance, in [Cao 2010], authors perform conformance testing for WS-BPEL orchestrators (taking time into account) by performing an online testing approach. The specification of the orchestrator is modeled by means of a Timed-Extended FSM and from which test purposes can be extracted. They define a conformance relation reasoning over the traces of the system under test and of its specification, and developed a prototype tool in order to perform an online testing algorithm on the orchestrators.

Another similar work for testing orchestrators (and more similar to our approach) is the one proposed in [Bentakouk 2009] (which is examined also in Section 6.1): authors test the orchestrator in context, with the assumption that the tester does not have access to the Web services interacting with the orchestrator. In our case this corresponds to the case where the Web services are hidden. Authors provide a complete translation from WS-BPEL to Symbolic Transition Systems (which are introduced in Chapter 6), that are used to model the specification of the orchestrator (given in WS-BPEL). Moreover, this same translation is the one we adopt in our approach when testing instances of orchestrators (this work is presented in Chapter 9). By using symbolic techniques, authors define an online testing algorithm (also implemented in a tool) to perform functional testing of the orchestrators.

Regarding component-based systems (and besides the works already examined of [van der Bijl 2003b, van der Bijl 2003a]), in [Braspenning 2006], authors propose an approach to perform model-based integration testing. The idea is to use models of the components in order to test if they work together correctly. They take into account the specification of the components in order to generate such models (by means of process algebra) and they assume that those specifications are available. Thus, the entire system is modeled by means of parallel compositions of all the models of the components and it is submitted to validation and verification with model checking tools. When a component is implemented, authors propose a model-based testing approach in order to check if the implementation of the component conforms to the model used in the model-based integration testing step. By doing so, the implemented component *will* interact correctly with the entire system. Thus, with their approach, authors help at reducing the time when performing integration testing of components, which is usually done until all the implementations of the components in the system are available. By performing integration testing before

finishing the implementation of the component, less time is waisted fixing the errors. This work is similar to ours for the case when all the Web services are controlled by the tester.

Another similar approach is the one proposed in [Faivre 2007], where authors work with Input/Output Symbolic Transition Systems and define the correctness of a component-based system by means of testing each individual component of the system using the **ioco** conformance relation. In order to do that, authors assume that the specification of every component in the system is available. Then, authors propose to derive test purposes a given component of the system from the behaviors of the components that encapsulate them.

The works presented before are similar to ours for the case of testing orchestrators in context. Now we take a look at works that are similar to the one we present in Chapter 5, where we give an approach to test if a given Web service behaves as expected from the orchestrator. This is done by extracting the behaviors of the Web service (or component) to test from the specification of the orchestrator (or another component(s)) of the system.

In [Frantzen 2007], authors introduce the **eco** conformance relation to test the environmental conformance of components. Authors assume that there are two types of specifications for the components: the interface and the behavioral ones, and that the interface's specification of a component can be divided in two: the one of the services it offers and the one of the services it invokes. In their approach, authors propose to test components only with the knowledge of the interfaces specifications to determine if a component interacts correctly with its environment. Moreover, the specification of the services that a component invokes is obtained from the environment's specification. For testing the component according to the services it provides (usually this information is provided in the interface specification of the component), authors propose to use **uioco**. Then, **eco** aims at testing if a component behaves as expected from the environment via the specification of the services that the environment invokes from the component under test. Thus, they inverse the inputs and outputs of the interface's specification of the environment, which is given by means of an LTS. In this way, they test the conformance of the implementation of a component with respect to what the environment expects from it, i.e., if the outputs that the component produces can be accepted by the environment, and if the outputs produced by the environment can be accepted by the component. Authors also assume that the implementation of the component under test can be modeled as an input complete $IOLTS$ (that they call IOTS). Then, test cases are generated from the environment in order to check if after any trace in $UTraces$ of the environment's LTS, whether all outputs produced by the implementation of the component (its IOTS) are included in the set of inputs of the environment. Besides, since the component under test is tested according to the environment, in **eco** there is no notion of quiescence situations, since the implementation of a component is not forced to request a service just because the environment is ready to accept such a request. Thus, in their approach, quiescence is not an observation.

In [Bertolino 2008], authors give an approach to test Web services supposed to interact with other Web services (not exactly in the scope of a Web service orchestration). In order to test them, authors simulate the rest of Web services taking into account their specifications, functional and extra-functional, which they suppose are available. Then, they use Symbolic Transition Systems to model the interfaces of services (functional specifications) to be simulated, and use the **eco** environmental conformance to ensure the generation of functionally correct responses and requests from the simulated Web services.

Another similar approach, in the sense that it is the Web services that are tested, is the one proposed in [Sinha 2006], where authors deal with the lack of knowledge of the behavior of the Web services by including in the test activity the WSDL-S standard, which gives more information about the behavior of the Web service. Then, they use Extended FSM in order to model the specifications of the Web services and to generate test cases which can be used to test the functional conformance of the Web services. This work can be complementary to our approach introduced in Chapter 5.

In [Bravetti 2007], authors test the conformance of services with respect to Web service choreographies by using what they call services contracts as specifications for the Web services. Authors check whether the Web services behave correctly according to the specification of the choreography. In order to do that, they use both the interface specification of the Web services (WSDL) and what is known as their *service contracts*, which can roughly be seen as a more complete specification than the WSDL of the Web services. Then, they perform the projection techniques (which are used also in our approach in Chapter 5) and, together with the information of the service contracts, they give an approach (using process calculus) to check if Web services conform with respect to what is expected from them by the choreography.

Finally, in [Bertolino 2009] authors present the PLASTIC validation framework, which is rather a combination of existing methods in order to test Web services (strictly speaking, the proposed methodology could be applied to any instantiation of the SOA, but it focuses on the Web services technology). This framework (which include some other tools) allow them to perform offline testing (testing Web services in a laboratory) by relying on the **ioco** and **eco** conformance relations, and assuming that behavioral specifications (augmented WDSL ones) are available for the Web services. They can also perform online testing (testing Web while being used) while simulating the needed services in such a way that they yield the correct functional and extra-functional behavior with respect to a given specification.

3.3.1 Discussion

In the first part of the works listed in previous section, we analyzed the ones which are close to our approach for testing orchestrators. However, most of them address the problem in a different way, especially by assuming that the specification of all the components or Web services in the system is given, and that they can be simulated. However, we test orchestrators by taking into account the fact that they

may potentially be plugged to contexts consisting of a collection of remote Web services while being tested, and that we have less information of the system: as detailed in next chapter, we only take into account the WSDL of the Web services and the specification of the orchestrator. With respect to [Bentakouk 2009], the main difference regarding our approach is is that we use $IOLTS$ for the theoretical part of our approach and we do not focus on WS-BPEL; besides, our approach can be more parametrized: the communication channels with the Web services can not only be hidden, they can be observable or controllable. Finally, we also elicit behaviors in order to test Web services.

Regarding the elicitation of behaviors in order to test Web services, in the second part of the presented works we examined some of the related works which share the notion of testing components or Web services from the knowledge of the specification of the system. This specification reflects the usage of a given component in the system and so it can be used to test the given component. The closest work to our approach is [Frantzen 2009], since in our case the environment could be seen as the orchestrator. However, we use symbolic techniques (presented in Part II), and we have the notion of test purposes (key behaviors that are tested on the SUT) that they do not. [Bravetti 2007] is also very close to our approach. The main difference with them is that we use symbolic techniques and that the systems we consider are not choreographies but orchestrations. Regarding the rest of works, the main difference is that we assume to have less information about the Web services, and that the type of systems that they work with are not the same as ours (orchestrators).

3.4 Conclusion

In this chapter we have presented the conformance relation **ioco**, that is the one that we use as the basis in the rest of this thesis. We started by introducing the LTSs automata, and then the $IOLTS$s, which are the adaptation of LTSs that we use, and that include the notions of inputs and outputs messages, as well as communication channels. We have also presented some operations and notions defined over the $IOLTS$s that are going to be used in the next chapters in order to test orchestrators in context, and to elicit behaviors for the Web services from the orchestrator's specification.

We have started by introducing **ioco** for testing stand-alone systems, and then we have also shown that, if we take into account several components (and therefore the synchronous product and hiding operations when modeling the specification of the entire system), some situations may arise. Therefore, we introduce the **uioco** conformance relation, which differs form **ioco** by the fact that the former one only takes the specified traces when reasoning about the conformance of implementations of component-based systems with respect to their specifications. Thus, we have shown that in a component-based system, it suffices only to test the components with respect to their specifications to be sure that the component-based system will behave correctly. In our case, it would mean to test the orchestrator

and the Web services with respect to their specifications to be sure that the orchestration will behave correctly. However, the previous results can be obtained only if the specifications are input complete, and under the assumptions that both the components and the composed system are correctly implemented.

Finally, we have presented some related works, showing that, with the exception of [Bentakouk 2009], and for the best of our knowledge, there are no works that have been done to test orchestrators while being used. Moreover, in our approach we only take into account the specification of the orchestrator. We take into account only the partial specification of the Web services (WSDL) and choose to *ignore* their behavioral specification. This reduces largely the explosion problem and allows us not to make the same assumptions about the specifications and the implementation of the components and their composition. Since the orchestrator is the one in charge of guiding the whole process, it is also possible, under certain hypothesis, to give some results about the correctness of the orchestrator being tested while interacting with the Web services.

Regarding our work done to test if Web services are compatible with the orchestrator, more similar works have been done, specially the ones of [Frantzen 2009] and [Bravetti 2007], but none of them deal with orchestrations by using the symbolic techniques that we use. Besides, we assume that we only have the specification of the orchestrator and the interface description of the Web services.

In the next chapters we show our approach for testing orchestrators in context and eliciting behaviors for the Web services from orchestrator's specifications.

Chapter 4

Testing in Context for Orchestrators

Contents

4.1 Introduction

In this chapter we present an approach to test orchestrators in context. That is, orchestrators potentially interacting with some Web services. We consider that we only know the specification of the orchestrator and not the behavioral specifications of the Web services (we only have the information of their interface via their associated WSDL).

The situation of the Web services supposed to be present in the orchestration can range between two extremes according to the testing architecture: (a) The orchestrator does not communicate with any remote Web services. The tester simulates all the Web services behaviors; (b) All the Web services are remotely connected to the orchestrator, the tester does not control any of the Web services. Thus, the orchestrator is tested by interacting with the whole orchestration. In Case (a), the tester has full control of the orchestrator, which eases the testing process. In Case (b), the testing process discovers errors on a system composed of the orchestrator and the Web services. That is, the errors can be due to the interactions between the orchestrator and the Web services. Messages exchanged between the orchestrator and the Web services are not controllable, which makes the testing process more complex than in Case (a). Between those two extremes, there can be the case where the

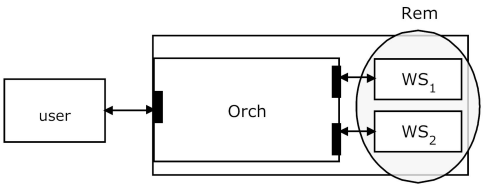


Figure 4.1: Orchestration example.

tester simulates only some Web services while making use of the implementations of the rest. Moreover, Case (b) can be subdivided into two sub-cases by considering observability issues concerning messages exchanged between the orchestrator and the Web services. Case ($b.1$): in the testing architecture, some of these messages can be *observable*, which means that the orchestration is instrumented in order to allow the tester to observe those messages. We suppose that such an instrumentation is realized on the computer where the orchestrator is deployed. Case ($b.2$): some other messages may be unobservable (we say *hidden*). That situation may result from technical limitations concerning accessibility to media used to connect the Web services and the orchestrator.

By taking the above presented facts into account, the system with which the tester interacts is no longer a stand-alone one, but can be seen as composed of two parts: the local one, composed of the orchestrator and the simulated Web services; and the remote one, composed of the remote implementation of the Web services. This particularity that the system with which the tester interacts is not the system to be tested, together with the fact that we only have a specification of the system to be tested (i.e., the orchestrator), leads us to slightly adapt the **ioco** theory for our purpose. We define new conformance relations to study correctness of orchestrators in context and give theorems to relate our testing in context approach to usual unit testing in the **ioco** framework. The theorems basically state that, under certain hypothesis, if there is an error in the orchestrator in context, then the error is in the orchestrator itself.

We begin this chapter by introducing an informal presentation of our approach in Section 4.2. Then, in Section 4.3, we show how to adapt the $IOLTS$ framework in order to model the SUTs in context, as well as how to obtain partial specifications for orchestrators in context. In Section 4.4, we define two basic conformance relations in context, which are an adaptation of **ioco** and **uioco**, and, based on them, we give the two theorems mentioned above. We conclude the chapter in Section 4.5.

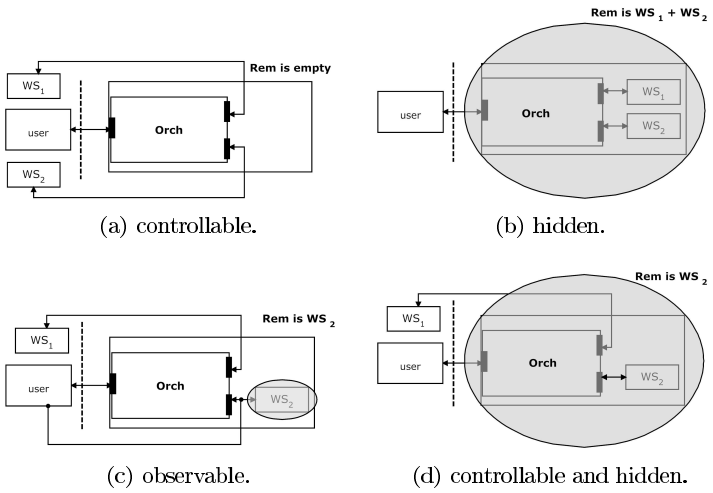
4.2 Informal Presentation of our Approach

In this section we give an informal description of our approach. We introduce some notions that are formally defined in the next sections.

We begin by describing some characteristics of the type of systems that we are interested in testing. Figure 4.1 depicts an orchestration. $Orch$ represents the or-

chestrator and WS_1 and WS_2 represent two Web services with which $Orch$ interacts. The $user$ of the system interacts with the orchestration which is composed of the orchestrator and the Web services. Moreover, he interacts with the system only by means of the orchestrator, so it can be seen as the interface of the system. From the point of view of $Orch$, WS_1 and WS_2 behave as a single remote system (that we call Rem) and with which $Orch$ interacts through two different communication channels represented by the two arrows in the figure. For the user, the interaction between $Orch$ and the Web services is not visible. Thus, we say that the user interacts with $Orch$ in the context of Rem ($Orch[Rem]$).

In our approach, like we did in Section 2.5, we suppose that $Orch$ and Rem can be modeled by means of $IOLTS$s. Thus, $Orch$ in context of Rem can intuitively be seen as a restriction of all the possible behaviors of $Orch$ due to the interaction with Rem. We can characterize this restriction by means of an product between $Orch$ and Rem (see Definitions 4.3.1 and 4.3.3), which results in an LTS, $Orch[Rem]$, on which the messages sent by Rem to $Orch$ have a special characteristic: they are inputs from the point of view of $Orch$ but they are not controlled by the user. We call those inputs *observations*.



(a) controllable. (b) hidden.

(c) observable. (d) controllable and hidden.

Figure 4.2: Communication channel status.

It is important to note that we do not propose an approach to test the system $Orch[Rem]$, but one to test $Orch$, where $Orch$ is interacting with Rem. Let us discuss more deeply about the nature of Rem. As we mentioned in the Introduction, our approach takes into account different situations for the Web services. Figure 4.2 depicts these different situations. The first case (Figure 4.2(a)) corresponds to the situation where the tester simulates the Web services. In fact, this architecture is the usual for unit testing of $Orch$. If a given Web service is simulated by the tester, we say that the communication channels initially used by $Orch$ to interact with that Web service are **controllable.** In this case, Rem is empty, and corresponds

to an $IOLTS$ with no transitions. The second case (Figure 4.2(b)) corresponds to the situation where the Web services are present and no instrumentation allows the tester to observe their interactions with $Orch$. Channels used for those interactions are said to be **hidden.** Rem consists of WS_1 and WS_2. In Figure 4.2(c), WS_1 is simulated and WS_2 is not. The channel initially plugged to WS_1 is accessible by the tester, an the one effectively plugged to WS_2 is said to be **observable**, since it can be accessed by the tester. Rem consists of WS_2. Finally, Figure 4.2(d) depicts the same situation than in Figure 4.2(c), except that the channel connected to WS_2 is hidden. Again, Rem is restricted to WS_2.

The previous classification allows us to partition the set of communication channels C used by $Orch$ into three sub-sets C_c, C_h and C_o, for the set of channels controllable, hidden and observable, respectively. Since the hidden channels (the set C_h) are not observable by the tester, this last one does not, strictly speaking, interact with $Orch$ in the context of Rem, $Orch[Rem]$, but with $Orch[Rem]$ *where* the messages sent through the channels in C_h are not observable. The tester can observe the messages sent through channels in C_o, and the tester can, of course, always observe the messages sent though C_c. If we take all this information and restrictions into account, we get a system with which the tester interacts, that is no longer $Orch[Rem]$ but rather the one that we note $Obs(Orch[Rem], C_o)$, which corresponds to $Orch[Rem]$ where only messages emitted through C_o are observable (as far as we only deal with messages exchanged with Web services, since messages emitted through C_c are also observable). Thus, our objective is to give an algorithm that allows us to test $Obs(Orch[Rem], C_o)$ in such a way that the detection of an error induces the non-conformance of $Orch$ with respect to its specification. That is, we test $Orch$ not in isolation but by interacting with $Orch$ in the context of Rem *and* taking into account the observability restrictions.

In order to achieve our objective, we need one more element: the specification of $Obs(Orch[Rem], C_o)$. As said before, we only use the specification of $Orch$, which we assume is always available. This specification is an $IOLTS$ that we note $\mathbb{O}rch$ in the sequel. From this specification, we build the one of $Obs(Orch[Rem], C_o)$ that we call *partial specification* of $Orch$ in the context of Rem, and that we note $Obs(\mathbb{O}rch)$. We achieve this in three steps:

I. We identify under which situations the system $Orch[Rem]$ can be quiescent. However, since we are considering a system composed of $Orch$ and Rem, we have to identify the situations where $Orch[Rem]$ can be quiescent due to quiescence of $Orch$: this corresponds to the case introduced in Definition 3.2.6; and those where $Orch[Rem]$ can be quiescent due to quiescence of Rem.

II. We transform the inputs of $Orch$ sent by Rem into observations. This simple transformation takes the inputs of $Orch$ of the form $c?t$, where c is a communication channel used by Rem, and transforms it by $\overline{c}!t$.

III. We hide the communications through the hidden channels in C_h.

By now, we have all the elements that we need in order to adapt the **ioco** conformance relation for testing orchestrators in context: the SUT, which in this case is $Obs(Orch[Rem], C_o)$, and its partial specification, $Obs(\mathbb{O}rch)$. With these elements we define a theorem which highlights that, under the hypothesis that the transition systems are strongly responsive, then, if there is an error in $Obs(Orch[Rem], C_o)$, it reveals an error in $Orch$ itself. Since introducing the hiding operation can make unobservable some underspecified inputs on hidden channels, we make two versions of the theorem: one for the cases where all the communication channels are either controllable or observable, and another one that takes into account the hidden case.

4.3 Orchestrators and Their Specifications in Context

In this section we introduce the formal definition of the operations introduced in the previous section. First we show how to model $Orch[Rem]$ by taking into account the observability restrictions in order to obtain $Obs(Orch[Rem], C_o)$, and then we show how to get its partial specification.

4.3.1 Orchestrators in Context

To reflect the point of view of the orchestrator, we slightly extend the $IOLTS$ framework into so-called *IOLTS with internal actions*. The main idea is to distinguish the various actions of the orchestrator as they relate to the system's user or Web services. So, actions are composed of: *inputs* of the form $c?v$, denoting local requests from the user; *outputs* of the form $c!v$, denoting results sent locally to its user or to the remote part; the τ unobservable action; internal actions of the form $\overline{c}!v$ (where c is a channel used by $Orch$ to communicate with Web services) corresponding to values sent by Rem to $Orch$, and called, as discussed in Section 4.2, *observations*; and *quiescence actions*, used to characterize the quiescence of an orchestrator induced by the lack of communication with Web services. Such an action of the form $\delta[c_1, \ldots, c_n]$, where $c_1, \ldots, c_n$ are channels used to communicate with Web services, denotes the quiescence of an orchestration induced by a lack of reactiveness of Web services using some channels in $\{c_1, \ldots, c_n\}$ to communicate with the orchestrator.

Definition 4.3.1 ($IOLTS$ with internal actions) *Let C be a set of communication channels and $C_r \subseteq C$ be a set of so-called* internal channels.

An $IOLTS$ over C with internal actions over C_r *is a LTS $\mathbb{G}$ over $Act(C, M) \cup Int(C_r, M) \cup \Delta(C_r)$ where:*

- $Int(C_r, M)$ *contains* internal actions *of the form $\overline{c}!v$ with $c \in C_r$ and $v \in M$*
- $\Delta(C_r)$ *contains* silent actions $\delta[w]$ *where $w \in {C_r}^*$*

Notation 4.3.1.1 *In the following, the set of communication channels over which internal actions are defined is not relevant, we simply use the terminology $IOLTS$* with internal actions.

The notations concerning $IOLTS$s can also be adapted for $IOLTS$s with internal actions.

In the following, for any $IOLTS$ with internal actions over the set of labels $Act(C, M) \cup Int(C_r, M) \cup \Delta(C_r)$, we use the notations $Int_{\mathbb{G}}$ and $\Delta_{\mathbb{G}}$ to refer, respectively, to $Int(C_r, M)$ and $\Delta(C_r)$.

In the following, the silent action $\delta[]$ is denoted δ for the sake of simplicity, and has the usual meaning of quiescence.

In the sequel we need to extend the hiding operation, introduced in Definition 3.2.13 in Section 3.2, for the case of $IOLTS$s with internal actions, since the set of communication actions is no longer the same. Moreover, this hiding operation will be used when defining the conformance relation in order to test orchestrators where there are hidden communication channels.

Definition 4.3.2 (Hiding operation for $IOLTS$s with internal actions) *Let C be a set of communication channels, let $C_r \subseteq C$ be a set of so-called* internal channels*, let $\mathbb{G}$ be an $IOLTS$ over C with internal actions over C_r, and let $\mathcal{C} \subseteq C_r$.*

$Hide(\mathcal{C}, \mathbb{G})$ is the $IOLTS$ over $C \setminus \mathcal{C}$ with internal actions over $C_r \setminus \mathcal{C}$ defined as follows:

- $Q_{Hide(\mathcal{C},\mathbb{G})} = Q_{\mathbb{G}}$
- $init_{Hide(\mathcal{C},\mathbb{G})} = init_{\mathbb{G}}$
- $Tr_{Hide(\mathcal{C},\mathbb{G})}$ *is defined as follows:*
  - *for any transition $(q, a, q') \in Tr_{\mathbb{G}}$ with a of the form $c\triangle v$ or $\overline{c}\triangle v$, with $\triangle \in \{?, !\}$ and $c \in \mathcal{C}$, $(q, \tau, q') \in Tr_{Hide(\mathcal{C},\mathbb{G})}$*
  - *for any transition $(q, a, q') \in Tr_{\mathbb{G}}$ with a of the form $c\triangle v$ or $\overline{c}\triangle v$, with $\triangle \in \{?, !\}$ and $c \notin \mathcal{C}$, or $a = \tau$, or $a \in \Delta_{\mathbb{G}}$, $(q, a, q') \in Tr_{Hide(\mathcal{C},\mathbb{G})}$*

As explained in Section 4.2, from the point of view of the orchestrator, $Orch$, the set of remote Web services can be modeled by a single SUT[1] that communicates through channels of the subset C_r of the set of communication channels C used by $Orch$. The SUT, denoted Rem, characterizes a set of traces that contains inputs corresponding to values sent by $Orch$ to the Web services, and outputs denoting the reaction of the Web services. In order to give a formal representation of $Orch$ in the context of Rems, we use a dedicated product.

[1] Let us recall that an SUT is an input-complete $IOLTS$ enriched with quiescence, and in practice does not contain any τ-transitions.

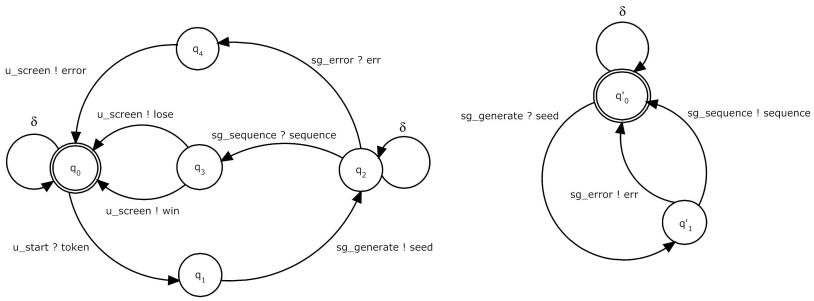
Definition 4.3.3 (*Orch* **in context of** *Rem*) *Let Orch be a SUT over C, let* $C_r \subseteq C$*, and let Rem be a SUT over* C_r*.*

Orch in the context of *Rem is the IOLTS over C, denoted Orch*[*Rem*]*, with internal actions over* C_r*, where:*

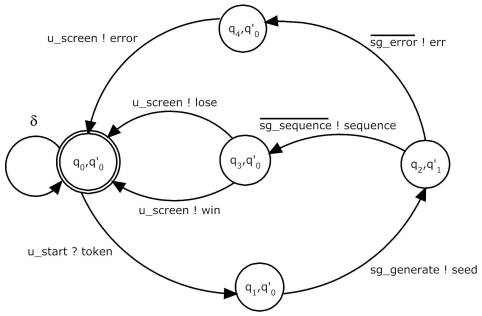
- $Q_{Orch[Rem]} = Q_{Orch} \times Q_{Rem}$
- $init_{Orch[Rem]} = (init_{Orch}, init_{Rem})$
- $Tr_{Orch[Rem]}$ *is defined as follows:*
  - *If* $(q_1, c!v, q_1') \in Tr_{Orch}$ *and* $(q_2, c?v, q_2') \in Tr_{Rem}$ *then* $((q_1, q_2), c!v, (q_1', q_2')) \in Tr_{Orch[Rem]}$
  - *If* $(q_1, c?v, q_1') \in Tr_{Orch}$ *and* $(q_2, c!v, q_2') \in Tr_{Rem}$ *then* $((q_1, q_2), \overline{c}!v, (q_1', q_2')) \in Tr_{Orch[Rem]}$
  - *For any* $(q_1, a, q_1') \in Tr_{Orch}$ *where a is* τ *or is of the form* $c?v$ *or* $c!v$*, with* $c \notin C_r$*, then for any* $q_2 \in Q_{Rem}$*,* $((q_1, q_2), a, (q_1', q_2)) \in Tr_{Orch[Rem]}$
  - *For any* $(q_1, \delta, q_1) \in Tr_{Orch}$*, for any* $(q_2, \delta, q_2) \in Tr_{Rem}$ *then* $((q_1, q_2), \delta, (q_1, q_2)) \in Tr_{Orch[Rem]}$

The first item of previous definition corresponds to values sent by *Orch* to *Rem*. Such values are outputs of *Orch* and thus denoted as outputs. The second item corresponds to values sent by *Rem* to *Orch*. These messages are not controlled by the tester, and from the point of view of *Orch* they are inputs; however, from the point of view of the tester they are outputs of the whole system since he does not control them. In order to differentiate them from values sent by *Orch*, we model them as observations of the form $\overline{c}!v$. The third item states that the transitions of *Orch* introducing the internal action or the communication actions with the user, execute asynchronously with respect to *Rem*. Finally, the fourth item corresponds to quiescent situations. Quiescence of *Orch*[*Rem*] occurs whenever no outputs or observation can occur. Since *Orch* and *Rem* are *SUT*s, they are input complete. Therefore, due to the first and second items, each time *Orch* or *Rem* may emit an output, *Orch*[*Rem*] emits either an output or an observation. For this reason, quiescence is allowed if and only if quiescence is observed for *Orch* and for *Rem*.

Example 4.3.1 *Consider Figure 4.3. Based on the Slot Machine example (Example 3.2.5), let us assume that the left hand image of Figure 4.3(a) depicts one possible implementation of the slot machine's interface (it is supposed to also be input complete, but it is not depicted for the sake of readability), Orch; while the right hand one depicts one possible implementation corresponding to the Sequence Generator service, Rem (it is also assumed to be input complete). Then, together they compose one possible implementation of the Slot Machine, as depicted by Figure 4.3(b) and which results after applying Definition 4.3.3 to the SUTs Orch and Rem, i.e., Orch*[*Rem*]*. The inputs from Rem are represented as observations for two reasons: first, inputs from Rem should not be treated the same as inputs from*



(a) *Orch* and *Rem*



(b) *Orch* in context of *Rem*

Figure 4.3: Slot Machine example: *Orch* in context of *Rem*.

the user; second, from the point of view of the tester, all communications between Rem and Orch are perceived as outputs from the system.

Now, it may happen that $Orch[Rem]$ is not fully observable by the tester. Indeed, as discussed in Section 4.2, some communication channels used by the Web services may be hidden. In order to reflect these situations, we partition the set of channels used by the orchestrator.

Definition 4.3.4 (Partitioned set of communication channels) *Let S be a set of communication channels.*

A partition for a set of channels C *is the given of three sets C_h, C_o, and C_c, such that*

- $C = C_o \cup C_h \cup C_c$
- $C_o \cap C_h = C_h \cap C_c = C_o \cap C_c = \emptyset$

Elements of C_o are called observable channels of C, elements of C_h are called hidden channels of C, and elements of C_c are called controllable channels of C. We denote $C_r = C_o \cup C_h$ the set of remote channels

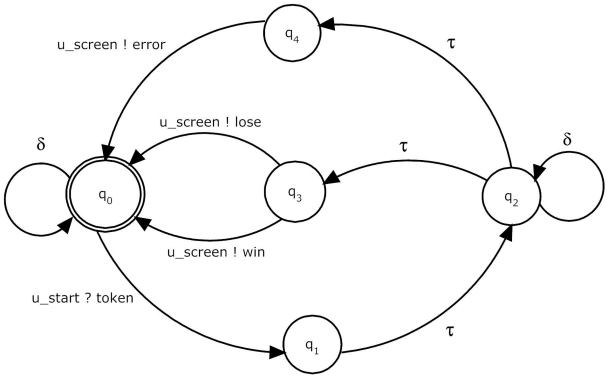


Figure 4.4: Partial observation of $Orch[Rem]$.

Channels of C_c are those used by the tester to interact with $Orch[Rem]$. This set contains, of course, the channels used by the user to communicate with the system, but also all the communication channels intended to communicate with the Web services whose functional role is simulated by the tester. Channels of C_o are those plugged to Web services for which messages transmitted through them can be intercepted and observed by the testing architecture. Finally, channels of C_h are those plugged to Web services and such that the testing architecture does not allow the tester to observe the messages transmitted through them. In the following, any set of channels C is considered as partitioned into C_h, C_o, and C_c even if it is not specified.

$Orch[Rem]$ can be modeled in its testing architecture by hiding in it the hidden channels just as it is done in Definition 4.3.2.

Definition 4.3.5 (Partial observation of $Orch$ in context of Rem) *Let $Orch$ be a SUT over C, and let Rem be a SUT over C_r.*

The partial observation of $Orch[Rem]$ through C_o, *denoted* $Obs(Orch[Rem], C_o)$, *is* $Hide(C_h, Orch[Rem])$

Example 4.3.2 *Figure 4.4 depicts the partial observation of $Orch[Rem]$ through $C_o = \emptyset$, and $C_h = \{sg\_generate, sg\_sequence, sg\_error\}$. That is, the Sequence Generator service SG is not simulated by the tester, and the communication channels used by the slot machine's interface SM to communicate with it are hidden. In this case, the tester cannot see what happens while SM interacts with SG, since he can only observe the outputs of SM. However, if an error occurs, it can be due to SM or to SG.*

Let us make some few comments concerning partial observations of orchestrators. In the sequel, partial observations are used as systems under test. The main difference with usual SUTs (as introduced in Definition 3.2.10 in Section 3.2), is

that a partial observation of an orchestrator may contain τ-transitions. We forbid τ-transitions in SUTs in order to make the product of Definition 4.3.3 easier. Besides, since we *assume* that the SUT can be obtained by interacting with the system under test, it makes no sense to model any internal action, since, even if the system does perform them, they are not visible from outside the system. However, that is not a restriction since τ-transitions in SUTs are not observable from the tester point of view, and for any $IOLTS$ with τ-transitions it is possible to define another one characterizing the same traces but with no τ-transitions. Here, as we discuss in the sequel, having τ-transitions in partial observations of orchestrators will no cause any particular technical problem, since we only manipulate it by means of its associated traces. Note that since both $Orch$ and Rem are input complete respectively on C and C_r, $Orch[Rem]$ is input complete over $C \setminus C_r$, and that set exactly characterizes the set of channels which are controlled by the tester (i.e., $C \setminus C_r = C_c$). Finally, regarding the quiescent situations, $Orch[Rem]$ is also enriched by quiescence, but, as said before, quiescence can be observed only with both $Orch$ and Rem are quiescent.

4.3.2 Specifications for Orchestrators in Context

In this section we show how to build specifications of partial observation of orchestrators in context as introduced in Definition 4.3.5. As discussed in the previous sections, our only prerequisite is the knowledge of the orchestrator's specification. Thus, we build the partial specification in four steps, taking into account the status of the communication channels used by the orchestrator to communicate with the Web services. As motivated by Section 4.2 and as we did in Definition 4.3.4, we also partition the set of communication channels C used by $\mathbb{O}rch$ into three sub-sets: C_c, C_h and C_o, for the set of channels controllable, hidden and observable, respectively, and we define the sub-set of channels used by $\mathbb{O}rch$ to communicate with the Web services as $C_r = C_o \cup C_h$. In the sequel of this chapter, we assume that such partitions are defined.

Taking this classification into account, in Definition 4.3.6 we enrich the specification with quiescence due to the lack of reaction by the Web services. In Definition 4.3.7 we enrich the specification with quiescence as defined in Section 3.2, Definition 3.2.6. We differentiate those two quiescences because the algorithm presented in Chapter 7 handles them differently, and their observations may yield different verdicts. Now, as discussed in Section 4.2, messages sent to the orchestrator by the Web services have the particular status of observations, and we apply this transformation in Definition 4.3.9. In Definition 4.3.10 we define the specification of an orchestrator in context in a testing architecture where we hide the messages on hidden channels.

Definition 4.3.6 (Internal quiescence) *Let $\mathbb{O}rch$ be an IOLTS over C.*

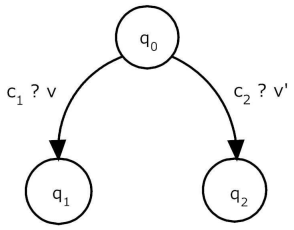
The internal quiescence enrichment of $\mathbb{O}rch$ *is the IOLTS with internal actions over C_r, denoted $\mathbb{O}rch_{\delta_i}$, such that:*

- $Q_{\mathbb{O}rch_{\delta_i}} = Q_{\mathbb{O}rch} \cup Q_\delta$
- $init_{\mathbb{O}rch_{\delta_i}} = init_{\mathbb{O}rch}$
- $Tr_{\mathbb{O}rch_{\delta_i}} = Tr_{\mathbb{O}rch} \cup Tr_\delta$,
  where Q_δ and Tr_δ are defined as follows:

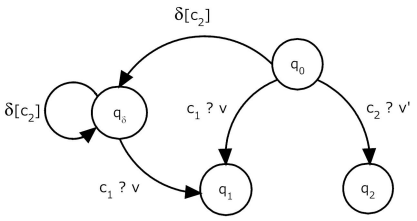
  for any $q \in Q_{\mathbb{O}rch}$ for which:

  - *there is no $(q, act, q') \in Tr_{\mathbb{O}rch}$ such that act is τ or is of the form $c!v$*
  - *there exists $(q, c?v, q') \in Tr_{\mathbb{O}rch}$ with $c \in C_r$*

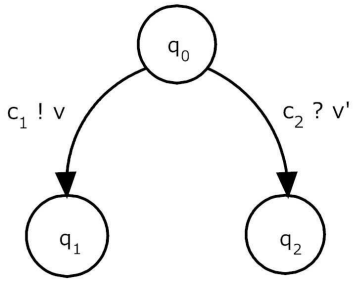
if we note $\{c_1, \cdots, c_n\}$ the greatest set of channels of C_r for which there exists $(q, c_j?v, q_j) \in Tr_{\mathbb{O}rch}$ with $1 \leq j \leq n$, then, there exists a unique state $q_\delta \in Q_\delta$ such that: $(q, \delta[c_1 \cdots c_n], q_\delta) \in Tr_\delta$, $(q_\delta, \delta[c_1 \cdots c_n], q_\delta) \in Tr_\delta$, and for any $(q, c?v, q') \in Tr_{\mathbb{O}rch}$ with $c \notin C_r$, we have $(q_\delta, c?v, q') \in Tr_\delta$



(a) *IOLTS* with only input transitions.



(b) Internal quiescence enrichment (Definition 4.3.6) of *IOLTS* of Figure 4.5(a).



(c) *IOSTS* with no internal quiescence.

Figure 4.5: Quiescence enrichment examples.

In order to illustrate Definition 4.3.6, let us consider the *IOLTS* depicted in

Figure 4.5(a), with $c_1 \notin C_r$ and $c_2 \in C_r$, and let us enrich it with internal quiescence. We obtain the $IOLTS$ depicted in Figure 4.5(b). The transition from q to q_δ reflects the quiescent situation induced by the absence of reaction of the Web service supposed to interact with the orchestrator through c_2. The loop on q_δ reflects the fact that the user will observe continuously this quiescent situation until it performs the input v on the channel c_1, which is depicted by the transition going from q_δ to q_1.

Let us note that for the $IOLTS$ of Figure 4.5(c), there is no quiescence enrichment for the state q. If no message is received, the system is supposed to emit the message v through c.

Now, Definition 4.3.7 is just an adaptation of the classical quiescence enrichment as introduced in Section 3.2, Definition 3.2.6.

Definition 4.3.7 (Full quiescence) *Let $\mathbb{O}rch$ be an $IOLTS$ over C.*

The full quiescence enrichment of $\mathbb{O}rch$ for C_r *is the $IOLTS$ over C with internal actions over C_r, denoted $\mathbb{O}rch_{\delta_f}$, such that:*

- $Q_{\mathbb{O}rch_{\delta_f}} = Q_{\mathbb{O}rch_{\delta_i}}$
- $init_{\mathbb{O}rch_{\delta_f}} = init_{\mathbb{O}rch}$
- $Tr_{\mathbb{O}rch_{\delta_f}} = Tr_{\mathbb{O}rch_{\delta_i}} \cup Tr_{\delta_e}$,
  where Tr_{δ_e} is defined as follows:

  for any $q \in Q_{\mathbb{O}rch}$, $(q, \delta[], q) \in Tr_{\delta_e}$ if and only if there is no $(q, act, q') \in Tr_{\mathbb{O}rch}$ such that act is τ, act is an output or act is an input of the form $c?v$ with $c \in C_r$

Notation 4.3.7.1 *In the following, for the sake of readability, we note δ for $\delta[]$.*

Note that actions of the form $c?v$ with $c \in C_r$ have to be considered as internal actions from the tester's point of view. Therefore states enriched with full quiescence cannot have outgoing transitions labeled by such an action. Quiescence due to the absence of reaction in such channels is taken into account in Definition 4.3.6.

We have given two successive definitions in order to take into account the two types of quiescence that can be found in an orchestrator in context, but this has been done in that way merely for pedagogical reasons, because the distinction between internal and external quiescence is important for our algorithm defined later in Section 7.4. We can, however, give a single definition in order to identify under which situations an $IOLTS$ with internal actions can be quiescent. The resulting structure takes into account both internal and external quiescences. This definition is the following one.

Definition 4.3.8 (Quiescence in context) *Let $\mathbb{G}$ be an IOLTS over C.*

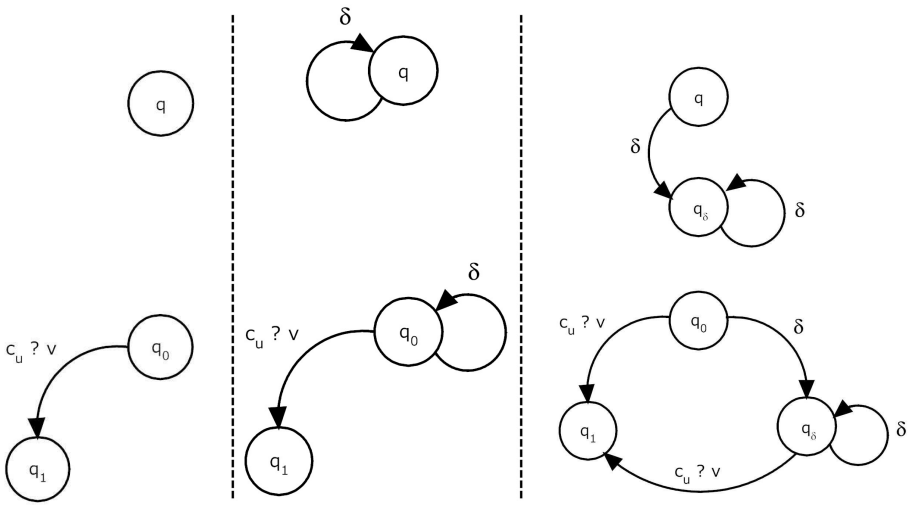
The quiescence in context enrichment of $\mathbb{G}$ *is the IOLTS over C with internal actions over C_r, denoted $\mathbb{G}\Delta$, such that:*

- $Q_{\mathbb{G}_\Delta} = Q_\mathbb{G} \cup Q_\delta$
- $init_{\mathbb{G}_\Delta} = init_\mathbb{G}$
- $Tr_{\mathbb{G}_\Delta} = Tr_\mathbb{G} \cup Tr_\delta$,
  where Q_δ and Tr_δ are defined as follows:

  for any $q \in Q_\mathbb{G}$ for which there is no $(q, act, q') \in Tr_\mathbb{G}$ such that act is τ or is of the form $c!v$:

  let us note C_q the set $\{c \in C_r \mid \exists\ (q, c?v, q') \in Tr_\mathbb{G}\}$. If C_q is not empty we note it $\{c_1, \ldots, c_n\}$. We define m as $c_1 \cdots c_n$ if C_q is not empty and as the empty word ε otherwise. Then, there exists a unique state $q_\delta \in Q_\delta$ such that $(q, \delta[m], q_\delta) \in Tr_\delta$, $(q_\delta, \delta[m], q_\delta) \in Tr_\delta$, and for any $(q, c?v, q') \in Tr_\mathbb{G}$ with $c \notin C_r$, we have $(q_\delta, c?v, q') \in Tr_\delta$

Notation 4.3.8.1 *In the following, for the sake of readability, we note δ for $\delta[]$.*



(a) Two *IOLTS*s. (b) Full quiescence Def. 4.3.7. (c) Quiescence in context Def. 4.3.8.

Figure 4.6: Quiescence equivalence.

Let us consider Figure 4.6. The column in Figure 4.6(a) depicts two basic *IOSTS*s:

- The upper one is the trivial $IOLTS$ consisting of only one state q, and no transitions. The second one characterizes a single transition from q_0 to q_1. The channel c_u is supposed to be in $C \setminus C_r$.

- Figure 4.6(b) depicts their full quiescence enrichment according to Definition 4.3.7: for the upper $IOLTS$, since there are no possible actions to be taken from q, quiescence can be observed and there is no way out; for the lower $IOLTS$, since $c_u \in C \setminus C_r$, we can also observe quiescence.

- Figure 4.6(c) depicts the quiescence enrichment of the $IOLTS$s according to Definition 4.3.8: for the upper $IOSTS$, since there is no transition leaving from q whose action is τ or of the form $c!v$, then m is the empty word and there is no way to leave the quiescence situation. For the lower $IOLTS$, the first condition is also true and m is also the empty word (since $c_u \notin C_r$), so quiescence is allowed. However, since $c_u \notin C_r$, it is possible to leave the quiescent situation from q_δ by an input $c_u?v$.

On these examples, we see that both quiescence enrichments of Definitions 4.3.7 and 4.3.8 are the same.

As glimpsed before, distinguishing between quiescent actions of respective forms $\delta[c1 \cdots c_n]$ and $\delta[]$ (or more simply just δ) is useful for our algorithm, since it gives an indication about a potential cause of the quiescence. However, at the semantical level, δ and $\delta[c_1 \cdots c_n]$ represent a same situation: the quiescence of the system under test. Therefore, from now, when we work at the semantical level (i.e., on sets of traces), we collapse the two notations $\delta[c_1 \cdots c_n]$ and δ, which do not have to be differentiated. We will keep the distinction when working at a symbolic level (in Part II), so that our algorithm (which operates at a symbolic level) can differentiate between those two forms of quiescence.

At the $IOLTS$ level, we authorize to identify occurrences of any $\delta[c_1 \cdots c_n]$ in any trace as an occurrence of δ. Also, in the sequel, for any transitions of the form $tr = (q, \delta[c_1 \cdots c_n], q')$, we allow indifferently to say that $act(tr)$ is $\delta[c_1 \cdots c_n]$ or δ, since those two actions will always be treated in the same way in all definitions of Sections 4.3–5.4.

Until now, all inputs and outputs are treated from the point of view of $Orch$. That is, inputs and outputs from and to the user are treated in the same way as those of Rem. However, we need to be able to differentiate between them. We then convert inputs that are not controllable by the user into observations in the sense of Definition 4.3.1.

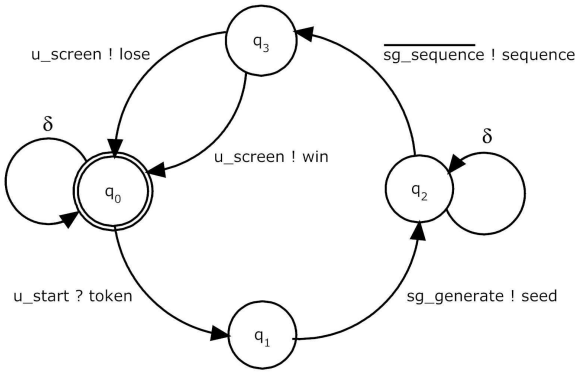


Figure 4.7: $IT(\mathbb{O}rch)$ for the Slot Machine example of Figure 3.3 in Section 3.2.

Definition 4.3.9 (Remote Input/Output transformation of $\mathbb{O}rch$) *Let $\mathbb{O}rch$ be an $IOLTS$ over C.*

The remote input/output transformation of $\mathbb{O}rch$ *is the $IOLTS$ with internal actions over C_r, denoted $IT(\mathbb{O}rch)$, such that:*

- $Q_{IT(\mathbb{O}rch)} = Q_{\mathbb{O}rch_{\delta_f}}$
- $init_{IT(\mathbb{O}rch)} = init_{\mathbb{O}rch_{\delta_f}}$
- $Tr_{IT(\mathbb{O}rch)}$ *is such that for all* $tr \in Tr_{\mathbb{O}rch_{\delta_f}}$,
  - *if tr is of the form $(q, c?v, q')$, with $c \in C_r$, then $(q, \overline{c}!v, q') \in Tr_{IT(\mathbb{O}rch)}$*
  - *if tr is of the form (q, a, q'), where a is not of the form $c?v$ with $c \in C_r$, then $(q, a, q') \in Tr_{IT(\mathbb{O}rch)}$*

Note that in Definition 4.3.9, we could have equivalently used $\mathbb{G}_\Delta$ in the place of $\mathbb{O}rch_{\delta_f}$.

Example 4.3.3 *Let us recall the specification of the orchestrator of the Slot Machine example, depicted in Figure 3.3 of Section 3.2. Figure 4.7 depicts its remote input/output transformation according to Definition 4.3.9, with the assumption that $C_o = \{sg\_generate, sg\_sequence\}$. In this case, the reception of the sequence generated by the Sequence Generator service is an observation from the point of view of the tester. Thus, we reflect this situation on the specification of $Orch$.*

Finally, in order to get the partial specification of $Orch$, $Obs(\mathbb{O}rch)$, we *hide* the communication actions going through the hidden channels. In order to do that, we apply the hiding operation of Definition 3.2.13 (Section 3.2) over the set C_h to the $IOLTS$ obtained after applying Definition 4.3.9, $IT(\mathbb{O}rch)$.

Definition 4.3.10 (Partial specification of $Orch$ in context of Rem) *Let $\mathbb{O}rch$ be an IOLTS over C.*

The partial specification of $Orch[Rem]$ through C_o, *denoted* $Obs(\mathbb{O}rch)$, *is defined as:*
$Hide(C_h, IT(\mathbb{O}rch))$

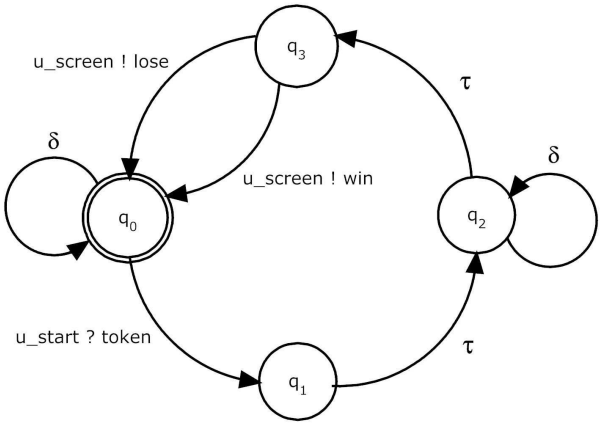


Figure 4.8: Slot Machine's partial specification.

Example 4.3.4 *Figure 4.8 depicts the partial specification $Obs(\mathbb{O}rch)$ for the Slot Machine example by applying Definition 4.3.10 to the IOLTS of Figure 4.7. By abuse, we consider in this example that $C_h = \{sg\_generate, sg\_sequence\}$. Thus, those communication channels are hidden and the messages going through them are not perceived by the tester.*

We now have the two elements needed to define the conformance relation in context: $Obs(Orch[Rem], C_o)$ and its partial specification, $Obs(\mathbb{O}rch)$.

4.4 Adaptation of ioco for Testing Orchestrators in Context

In this section we present how do we adapt the **ioco** and **uioco** conformance relations in order to use them in the context of $IOLTS$s with internal actions. We call these conformance relations $\overline{\textbf{ioco}}$ and $\overline{\textbf{uioco}}$.

Based on the conformance relations we introduce, we then define two theorems allowing us to relate conformance of orchestrators in context to usual unit conformance of orchestrators. The first theorem concerns cases where all internal channels are observable, and is based on $\overline{\textbf{ioco}}$. The second one concerns cases where some channels are hidden, and is based on $\overline{\textbf{uioco}}$.

Section 4.4.1 concerns adaptations of **ioco** and **uioco**, while Sections 4.4.2 and 4.4.3 concern, respectively, the theorems discussed above.

4.4.1 Adaptation of ioco and uioco to *IOLTS*s with Internal Actions

We begin by adapting **ioco** for *IOLTS*s with internal actions. This adaptation is straightforward, the only change is in the set of traces that we consider.

Definition 4.4.1 ($\overline{\textbf{ioco}}$) *Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two IOLTSs over C with internal actions over C_r.*

$\mathbb{G}_2$ $\overline{\textbf{ioco}}$ $\mathbb{G}_1$ *if and only if* $\forall \sigma \in Traces(\mathbb{G}_1)$, $\forall o \in O(C,M) \cup \{\delta\}$
$\sigma.o \in Traces(\mathbb{G}_2) \Rightarrow \sigma.o \in Traces(\mathbb{G}_1)$

Notice that $\overline{\textbf{ioco}}$ is the trivial restriction of **ioco** where observations are treated as particular inputs: indeed, as **ioco** defines no restriction about inputs that may follow a trace in the system under test, $\overline{\textbf{ioco}}$ also does not define any restriction on inputs or observations that may follow a trace of the system under test (here modeled as $\mathbb{G}_2$). The restrictions on outputs are the same than in **ioco**.

In order to define $\overline{\textbf{uioco}}$, we first introduce a notion of *UTraces* (as in Section 3.2.3) restricted to transitions with observations.

Definition 4.4.2 (Specified traces of an *IOLTS* with internal actions) *Let $\mathbb{G}$ be an IOLTS with internal actions over C_r.*

The set of specified traces of $\mathbb{G}$, *denoted* $UTraces(\mathbb{G})$, *is the set of all* $\sigma \in Traces(\mathbb{G})$ *such that there do not exist two paths p and p_1 of $\mathbb{G}$ such that:*

- *σ can be decomposed in $\sigma_1.a.\sigma_2$ with a of the form $c?v$ or $\overline{c}!v$*
- *σ is the trace of p*
- *σ_1 is the trace of p_1*
- *for any $tr \in Tr_{\mathbb{G}}$ with $source(tr) = target(p_1)$, $\sigma_1.a$ is not the trace of $p_1.tr$*

Again, we notice that observations are treated as particular inputs with regard to Definition 3.2.14 of Section 3.2.3. This is natural because observations are indeed inputs that only have the particularity to be uncontrollable by the tester.

We now define $\overline{\textbf{uioco}}$, which is a trivial adaptation of **uioco** as $\overline{\textbf{ioco}}$ is a trivial adaptation of **ioco**.

Definition 4.4.3 ($\overline{\textbf{uioco}}$) *Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two IOLTSs over C with internal actions over C_r.*

$\mathbb{G}_2$ $\overline{\textbf{uioco}}$ $\mathbb{G}_1$ *if and only if* $\forall \sigma \in UTraces(\mathbb{G}_1)$, $\forall o \in O(C,M) \cup \{\delta\}$
$\sigma.o \in Traces(\mathbb{G}_2) \Rightarrow \sigma.o \in Traces(\mathbb{G}_1)$

4.4.2 Conformance Testing of Orchestrators with no Hidden Channels

Taking into account only the cases where all the communication channels used between $Orch$ and Rem are either controllable or observable, that is, the system under test of Definition 4.3.3, $Orch[Rem]$, we show that if $Orch$ **ioco** conforms to $\mathbb{O}rch$, then $Orch[Rem]$ $\overline{\textbf{ioco}}$ $IT(\mathbb{O}rch)$.

Let us recall that in the end, our goal is to evaluate the conformance of $Orch$ with respect to $\mathbb{O}rch$ by interacting with $Orch[Rem]$ (here consisting of $Orch[Rem]$ —see discussion in Section 4.2). In this perspective, Theorem 4.4.1 is very useful since it means that if one finds an error in $Orch[Rem]$ (i.e., $Orch[Rem]$ $\underline{\textbf{ioco}}$ $IT(\mathbb{O}rch)$), then, necessarily the error comes from $Orch$ and not from the remote Web services in Rem (i.e., $Orch$ ~~**ioco**~~ $\mathbb{O}rch$).

Theorem 4.4.1 *Let $\mathbb{O}rch$ be an IOLTS over C, and let $Orch$ and Rem be two SUTs respectively over C and C_r.*

$$Orch \textbf{ ioco } \mathbb{O}rch \Rightarrow Orch[Rem] \ \overline{\textbf{ioco}} \ IT(\mathbb{O}rch)$$

Proof of Theorem 4.4.1.

Next definition introduces a function allowing us to associate any trace of any $IOSLTS$ with internal actions to a usual suspension trace by transforming observations into inputs, and internal quiescence actions into usual quiescence actions.

Definition 4.4.4 *Let C be a set of channels and M be a set of values. Let us note $\overline{Act}(C,M) = Act(C,M) \cup Int(C_r,M)$, and let us note $\widetilde{Act} \subseteq (\overline{Act}(C,M) \cup \{\delta\})^*$ the set of traces of $\overline{Act}(C,M)$ such that $\sigma \in \widetilde{Act}$ if and only if:*

for any decomposition $\sigma_1.a.\sigma_2$ of σ, with a of the form $\overline{c}!v$, then, if we note σ_1 as $\sigma_0.b$, we have $b \neq \delta$.

The externalization function of C_r in C *$Ext : \widetilde{Act} \rightarrow (Act(C,M) \cup \{\delta\})^*$ is such that $Ext(\varepsilon) = \varepsilon$, and for any trace $\sigma.a \in \widetilde{Act}$ we have:*

- *if a is of the form $c?v$ or $c!v$ or δ then $Ext(\sigma.a)$ is $Ext(\sigma).a$,*
- *if a is of the form $\overline{c}!v$ then $Ext(\sigma.a)$ is $Ext(\sigma).c?v$.*

Proposition 4.4.1.0.1 *Ext is an injective function.*

Proof.

By induction:

Basic case: *There exists a unique σ such that $Ext(\sigma)$ is ε: σ is ε,*

Inductive step: *Let σ' be $\sigma.a$, and let us suppose that there exists a unique σ'' such that $Ext(\sigma'') = \sigma$. Let us show that there exists a unique σ''' such that $Ext(\sigma''') = \sigma'$. Trivially, σ''' is $\sigma''.b$, where b is of the form $\overline{c}!v$ if a is of the form $c?v$, with $c \in C_r$, and b is a otherwise.*

Note that if σ' is of the form $\sigma_1.\delta.c?a$, with $c \in C_r$, then, there is no σ''' such that $Ext(\sigma''') = \sigma'$, because σ' could not belong to $\widetilde{Act}$.

□

The next definition allows one to associate any path of an orchestrator in context, to the corresponding path of the orchestrator considered as a unit.

Definition 4.4.5 *Let $Orch$ and Rem be two SUTs respectively over C and C_r. We define the function $ExtPath : Path(Orch[Rem]) \rightarrow Path(Orch)$ as follows:*

- *if p is ε $ExtPath(p)$ is ε*
- *if p is of the form $p_p.tr$ where tr is of the form $((q_1, q_2), a, (q'_1, q'_2))$ then $ExtPath(p_p.tr) = ExtPath(p_p).tr'$, where $tr' = (q_1, c?v, q_2)$ if a is of the form $\overline{c}!v$, and tr' is (q_1, a, q_2) otherwise*

Proposition 4.4.1.0.2 *With notations of Definition 4.4.5, for any $\sigma \in Traces(Orch[Rem])$, for any $p \in Path(Orch[Rem])$, such that $\sigma = traces(p)$, we have $Ext(\sigma) = traces(ExtPath(p))$.*

Proof.

We give the proof by induction on the form of σ:

Basic case: *If σ is ε, then p is necessarily a (possibly empty) sequence of transitions whose associated actions are τ. Thus, $ExtPath(p)$ is also necessarily a (possibly empty) sequence of transitions whose associated actions are τ. Therefore, $traces(ExtPath(p))$ is ε. Now, from Definition 4.4.4, since σ is ε, we have $Ext(\sigma) = \varepsilon$. Thus, for all p such that $\sigma \in traces(p)$, we have $Ext(\sigma) = traces(ExtPath(p))$.*

Inductive step: *We note σ as $\sigma'.a$. Since $\sigma = traces(p)$, p is necessarily of the form $p_p.tr.p_s$ where $traces(p_p) = \sigma'$, $act(tr) = a$, and p_s is a possibly empty sequence of transitions whose associated actions are τ.*

Let us suppose that:

(H): $Ext(\sigma') = traces(ExtPath(p_p))$,

and let us reason on the form of a:

Case (a)**:** *If a is of the form $\overline{c}!v$, then $Ext(\sigma) = Ext(\sigma').c?v$, and $ExtPath(p'.tr)$ is of the form $ExtPath(p').tr'$, where tr' is a transition such that $act(tr') = c?v$. By Definition 3.2.3 of traces and from (H), we deduce $Ext(\sigma) = traces(ExtPath(p_p.tr))$. Moreover, since p_s is a possibly empty sequence of transitions whose associated actions are τ, we also deduce from Definition 3.2.3 that $Ext(\sigma) = traces(ExtPath(p_p.tr.p_s))$, that is, $Ext(\sigma) = traces(ExtPath(p))$.*

Case (b)**:** *If a is not of the form $\overline{c}!v$, then $Ext(\sigma) = Ext(\sigma').a$, and $ExtPath(p'.tr)$ is of the form $ExtPath(p').tr'$, where tr' is a transition such that $act(tr') = a$. The remaining of the proof is the same than in Case (a).*

□

Next Lemma sates that if a trace is a trace of $Orch[Rem]$, then its externalization is a suspension trace of $Orch$.

Lemma 4.4.1.1 *Let $Orch$ and Rem be two SUTs respectively over C and C_r. For any $\sigma \in \widetilde{Act}$, we have:*

$$\sigma \in Traces(Orch[Rem]) \Rightarrow Ext(\sigma) \in Traces(Orch)$$

Proof.

It suffices to show that for any $\sigma \in \widetilde{Act}$, if $\sigma \in Traces(Orch[Rem])$, then there exists $p' \in Path(Orch)$ such that $Ext(\sigma) = traces(p')$.

This is obvious because, since $\sigma \in Traces(Orch[Rem])$, there exists $p \in Path(Orch[Rem])$ such that $\sigma = traces(p)$. Now, from Proposition 4.4.1.0.2 we have $Ext(\sigma) = traces(ExtPath(p))$. We conclude by defining p' as $ExtPath(p)$.

□

We now adapt the function $ExtPath$ of Definition 4.4.5 to the case of orchestration's specifications.

Definition 4.4.6 *For any IOLTS $\mathbb{O}rch$ over C, we define $ExtPath_S : Path(IT(\mathbb{O}rch)) \rightarrow Path(\mathbb{O}rch_\delta)$ as follows:*

- $ExtPath_S(\varepsilon) = \varepsilon$
- *$ExtPath_S(p.(q, act, q'))$, where act is τ or of the form $c?v$ or $c!v$, is:* $ExtPath_S(p).(target(ExtPath_S(p)), act(tr), target(tr))$
- *$ExtPath_S(p.(q, act, q'))$, where act is of the form $\overline{c}!v$ is:* $ExtPath_S(p).(target(ExtPath_S(p)), c?v, target(tr))$
- *$ExtPath_S(p.(q, act, q'))$, where act is of the form δ is:* $ExtPath_S(p).(target(ExtPath_S(p)), \delta, target(ExtPath_S(p)))$

Intuitively, $ExtPath_S$ differs from $ExtPath$ to take into account that transitions labeled with some δ in $IT(\mathbb{O}rch)$ have[2]:

- either some state q (q being a state of $\mathbb{O}rch_\delta$) as source, and some state q_δ (introduced at the quiescence enrichment step) as target state,
- or define loops on some step q_δ defined at the quiescence enrichment step.

In order to identify the corresponding paths in $\mathbb{O}rch_\delta$, we have to map those transitions to those of the form (q, δ, q) in $\mathbb{O}rch_\delta$, as it is done in the last item of Definition 4.4.6.

Proposition 4.4.1.1.3 *We note $Ext(\widetilde{Act})$ the subset of $Act(C,M)^*$ such that $\sigma \in Ext(\widetilde{Act})$ if and only if $\exists \sigma' \in \widetilde{Act}$ with $\sigma = Ext(\sigma')$, and we note $\widetilde{ExtPath_S} : Path(IT(\mathbb{O}rch)) \rightarrow \{p \in Path(\mathbb{O}rch) \,|\, traces(p) \in Ext(\widetilde{Act})\}$*

$\widetilde{ExtPath_S}$ is surjective.

Proof.

Any path $p = t_1 \cdots t_n$ of $\{p \in Path(\mathbb{O}rch) \,|\, traces(p) \in Ext(\widetilde{Act})\}$ can be associated with one path $p' = t'_1 \cdots t'_n$ of $Path(IT(\mathbb{O}rch))$, satisfying $ExtPath_S(p') = p$, where p' is such that:

- *for all maximal sub sequences $tr_l \cdots tr_m$ of p such that for all $i \in [l, \cdots, m[$,$act(tr_i)$ is δ, and $act(tr_m)$ is of the form $c?v$, then,the sub - sequence $tr'_l \cdots tr'_m$ is such that:*
  - *either $tr'_l \cdots tr'_{m-1}$ is $tr_l \cdots tr_{m-1}$, and $tr'_l = \cdots = tr'_{m-1}$, and $tr_l = \cdots = tr_{m-1}$, and tr'_m is tr_m (c is necessarily in $C \setminus C_r$),*
  - *or for all $i \in [l, \cdots, m[$, $act(tr'_i)$ is of the form δ, $source(tr'_l) = source(tr_l)$, $target(tr'_l)$ is the state $source(tr_l)_\delta$ (as defined in Definition 4.3.7), for all $j \in]l, \cdots m[$, we have $source(tr'_j) = target(tr'_j) = source(tr_l)_\delta$, and $source(tr'_m) = source(tr_l)_\delta$, $target(tr'_m) = target(tr_m)$, and $act(tr'_m)$ is $act(tr_m)$ (c is necessarily in $C \setminus C_r$).*
- *for all $k \leq n$ such that tr'_k does not occur in a sequence as described above, we have tr'_k is tr_k if $act(tr_k)$ is not of the form $c?v$ with $c \in C_r$, and tr'_k is $(source(tr_k), \overline{c}!v, target(tr_k))$ otherwise.*

□

Proposition 4.4.1.1.4 *With notations of Definition 4.4.6, for any $\sigma \in Traces(IT(\mathbb{O}rch))$, for any $p \in Path(IT(\mathbb{O}rch))$ such that $\sigma = traces(p)$, we have $Ext(\sigma) = traces(\widetilde{ExtPath_S}(p))$*

[2]See Definitions 4.3.7 and 4.3.9

Proof.

We give the proof by induction on the form of σ:

Basic case: *If σ is ε, then p is necessarily a (possibly empty) sequence of transitions whose associated actions are τ. Thus, $\widetilde{ExtPath_S}(p)$ is also necessarily a (possibly empty) sequence of transitions whose associated actions are τ. Therefore, $traces(\widetilde{ExtPath_S}(p))$ is ε. Now, from Definition 4.4.4, since σ is ε, we have $Ext(\sigma) = \varepsilon$. Thus, for all p such that $\sigma \in traces(p)$, we have $Ext(\sigma) = traces(\widetilde{ExtPath_S}(p))$.*

Inductive step: *We note σ as $\sigma'.a$. Since $\sigma = traces(p)$, p is necessarily of the form $p_p.tr.p_s$; where $traces(p_p) = \sigma'$, $act(tr) = a$, and p_s is a possibly empty sequence of transitions whose associated actions are τ.*

Let us suppose that:

(H): $Ext(\sigma') = traces(\widetilde{ExtPath_S}(p_p))$,

and let us reason on the form of a:

Case (a): *If a is of the form $\overline{c}!v$, then $\widetilde{Ext}(\sigma) = Ext(\sigma').c?v$, and $\widetilde{ExtPath_S}(p'.tr)$ is of the form $\widetilde{ExtPath_S}(p').tr'$, where tr' is a transition such that $act(tr') = c?v$. By Definition 3.2.3 and from (H), we deduce $Ext(\sigma) = traces(\widetilde{ExtPath_S}(p_p.tr))$. Moreover, since p_s is a possibly empty sequence of transitions whose associated actions are τ, we also deduce from Definition 3.2.3 that $Ext(\sigma) = traces(\widetilde{ExtPath_S}(p_p.tr.p_s))$, that is, $Ext(\sigma) = traces(\widetilde{ExtPath_S}(p))$.*

Case (b): *If a is not of the form $\overline{c}!v$, then:*

- *either a is of the form δ, and in this case we have $Ext(\sigma) = Ext(\sigma').\delta$, and $\widetilde{ExtPath_S}(p'.tr)$ is of the form $\widetilde{ExtPath_S}(p').tr'$, where tr' is a transition such that $act(tr') = \delta$. The remaining of the proof is the same than in Case (a),*
- *or a is not of the form δ, and in this case we have $Ext(\sigma) = Ext(\sigma').a$, and $\widetilde{ExtPath_S}(p'.tr)$ is of the form $\widetilde{ExtPath_S}(p').tr'$, where tr' is a transition such that $act(tr') = a$. The remaining of the proof is the same than in Case (a).*

□

Lemma 4.4.1.2 *For any $\sigma \in \widetilde{Act}$, we have:*
$Ext(\sigma) \in STraces(\mathbb{O}rch) \Rightarrow \sigma \in Traces(IT(\mathbb{O}rch))$.

Proof.

Since $Ext(\sigma) \in STraces(\mathbb{O}rch)$*, there exists a path* p *in* $\mathbb{O}rch$ *such that* $Ext(\sigma) = traces(p)$*, and* $p \in \{p \in Path(\mathbb{O}rch) | traces(p) \in Ext(\widetilde{Act})\}$

$\widetilde{ExtPath_S}$ *is surjective, thus there exists* $p' \in Path(IT(\mathbb{O}rch))$ *such that* $\widetilde{ExtPath_S}(p') = p$

From Proposition 4.4.1.1.4, $Ext(traces(p')) = traces(\widetilde{ExtPath_S}(p'))$

Since $\widetilde{ExtPath_S}(p') = p$ *and* $Ext(\sigma) = traces(p)$*,*
we have $Ext(traces(p')) = Ext(\sigma)$

Now, since Ext is injective, we have
$traces(p') = \sigma$*, and thus,* $\sigma \in Traces(IT(\mathbb{O}rch))$*.*

□

Proof of Theorem 4.4.1*:*
We want to prove:

$$Orch \ ioco \ \mathbb{O}rch \ \Rightarrow Orch[Rem] \ \overline{ioco} \ IT(\mathbb{O}rch)$$

We suppose:

(1)*:* $Orch \ ioco \ \mathbb{O}rch$*,*

which means by definition:

(2)*:* $\forall \sigma \in STraces(\mathbb{O}rch)$*,* $\forall o \in O(C, M) \cup \{\delta\}$

$$\sigma.o \in Traces(Orch) \implies \sigma.o \in STraces(\mathbb{O}rch)$$

Now, from Lemma 4.4.1.1,

(3)*:* $\forall \sigma \in \widetilde{Act}$*,*

$$\sigma \in Traces(Orch[Rem]) \Rightarrow Ext(\sigma) \in Traces(Orch)$$

From (3)*, we deduce:*

(4)*:* $\forall \sigma \in \widetilde{Act}$*,* $\forall o \in O(C, M) \cup \{\delta\}$

$$\sigma.o \in Traces(Orch[Rem]) \Rightarrow Ext(\sigma.o) \in Traces(Orch)$$

Now, clearly $Ext(\sigma.o) = Ext(\sigma).Ext(o)$*, thus, from* (4)*, we have:*

(5)*:* $\forall \sigma \in \widetilde{Act}$*,* $\forall o \in O(C, M) \cup \{\delta\}$

$$\sigma.o \in Traces(Orch[Rem]) \Rightarrow Ext(\sigma).Ext(o) \in Traces(Orch)$$

Since $Ext(o) \in O(C,M) \cup \{\delta\}$*, and* $Ext(o) = o$*, we deduce from* (2) *and* (5)*:*

(6)*:* $\forall \sigma \in \widetilde{Act}$*,* $\forall o \in O(C,M) \cup \{\delta\}$

$$\sigma.o \in Traces(Orch[Rem]) \Rightarrow Ext(\sigma).Ext(o) \in STraces(\mathbb{O}rch)$$

Now, traces of $IT(\mathbb{O}rch)$ *are traces of* $\widetilde{Act}$*. So, from* (6)*, we have:*

(7)*:* $\forall \sigma \in Traces(IT(\mathbb{O}rch))$*,* $\forall o \in O(C,M) \cup \{\delta\}$

$$\sigma.o \in Traces(Orch[Rem]) \Rightarrow Ext(\sigma).Ext(o) \in STraces(\mathbb{O}rch)$$

Now,since $Ext(\sigma.o) = Ext(\sigma).Ext(o)$*, from* (7)*, we have:*

(8)*:* $\forall \sigma \in Traces(IT(\mathbb{O}rch))$*,* $\forall o \in O(C,M) \cup \{\delta\}$

$$\sigma.o \in Trace(Orch[Rem]) \Rightarrow Ext(\sigma.o) \in STraces(\mathbb{O}rch)$$

From Lemma 4.4.1.2 we have:

(9)*: For all* $\sigma \in Traces(IT(\mathbb{O}rch))$*,*

$$Ext(\sigma.o) \in STraces(\mathbb{O}rch) \Rightarrow \sigma.o \in Traces(IT(\mathbb{O}rch))$$

Thus, from (8) *and* (9)*, we conclude:*

$\forall \sigma \in Traces(IT(\mathbb{O}rch))$*,* $\forall o \in O(C,M) \cup \{\delta\}$

$$\sigma.o \in Traces(Orch[Rem]) \Rightarrow \sigma.o \in Traces(IT(\mathbb{O}rch))$$

□

4.4.3 Conformance Testing of Orchestrators with Hidden Channels

In order to obtain a similar theorem to Theorem 4.4.1 when taking into account hidden actions over some set of channels C_h, we add the restriction that the $IOLTS$ $\mathbb{O}rch$ be a so-called *input complete for* C_h, that is: for any state $q \in Q_{\mathbb{O}rch}$, for any $c \in C_h$ and $v \in M$, there exists $(q, c?v, q') \in Tr_{\mathbb{O}rch}$. Moreover, we also must reason with the $\overline{\textbf{uioco}}$ conformance relation: i.e., the result of Theorem 4.4.2 does not hold when replacing $\overline{\textbf{uioco}}$ by $\overline{\textbf{ioco}}$ for the same reason that in Theorem 3.2.3: underspecification of inputs or observations induce underspecification in the internal actions if underspecified inputs or observations are hidden.

Theorem 4.4.2 *Let $\mathbb{O}rch$ be an IOLTS over C and input complete on C_h, and let $Orch$ and Rem be two SUTs respectively over C and C_r.*

Then,
$Orch$ ***ioco*** $\mathbb{O}rch \Rightarrow Obs(Orch[Rem], C_o)$ $\overline{\textbf{\textit{uioco}}}$ $Obs(\mathbb{O}rch)$

Proof of Theorem 4.4.2.

We know from Theorem 4.4.1 that

$$Orch\ ioco\ \mathbb{O}rch \Rightarrow Orch[Rem]\ \overline{ioco}\ IT(\mathbb{O}rch)$$

Let us show that if $\mathbb{O}rch$ is input complete over C_h:

$$Orch\ ioco\ \mathbb{O}rch \Rightarrow Obs(Orch[Rem], C_o)\ \overline{uioco}\ Obs(\mathbb{O}rch)$$

Let us define a forgetful function over traces and paths of $IT(\mathbb{O}rch)$ or $Orch[Rem]$. For $\sigma \in Traces(IT(\mathbb{O}rch)$ or in $Traces(Orch[Rem])$, we define $U_{C_h}(\sigma)$ (or simply $U(\sigma)$) by:

- if $\sigma = \varepsilon$, then $U(\sigma) = \varepsilon$
- for $\sigma = \sigma'.a$, if a is of the form $c!t$ or $\bar{c}!t$, with $c \in C_h$, then $U(\sigma) = U(\sigma')$, else $U(\sigma) = U(\sigma').a$

We define in a similar way the application U_{Orch} (resp. $U_{\mathbb{O}rch}$) over paths of $Orch[Rem]$ (resp. of $IT(\mathbb{O}rch)$). Transitions with an action over C_h simply become τ-transitions.

By remarking that $IT(\mathbb{O}rch)$ inherits from the hypothesis of input-completeness of $\mathbb{O}rch$ over C_h, let us now give the proof of the theorem:

proof of Theorem 4.4.2:

Let $\sigma.o$ be a trace of $Traces(Obs(Orch[Rem], C_o))$, with o an output and such that σ belongs to $UTraces(Obs(\mathbb{O}rch))$. Let us note $p^{\mathbb{O}}_{\sigma}$ such that $U(traces(p^{\mathbb{O}}_{\sigma})) = \sigma$.

Let us note κ a trace of $Orch[Rem]$ verifying that $U(\kappa) = \sigma.o$, and the last transition carries the o action (that is, κ is written $\kappa'.o$). Such a trace exists necessarily since $\sigma.o$ is a trace of $Obs(Orch[Rem], C_o)$.

Let us introduce for any trace ρ defined over the signature of $IT(\mathbb{O}rch)$:

$$Path_{\mathbb{O}rch}(\rho) = \{p \in Path(IT(\mathbb{O}rch)) \mid traces(p) = \rho\}$$

Let us demonstrate by an inductive reasoning that for each sub-trace ρ that is a prefix of σ, i.e., σ can be written $\rho.\pi$, then $Path_{\mathbb{O}rch}(\rho)$ is not empty.

Basic case: *The case $\rho = \varepsilon$ is clear.*

Inductive step: *Let us consider ρ of the form $\mu.a$. By hypothesis, $Path_{\mathbb{G}}(\mu)$ contains a path p_μ verifying $traces(p_\mu) = \mu$. Let us reason by case on the form of a:*

Case (a): *If a is of the form $\overline{c}!t$, then as $IT(\mathbb{O}rch)$ is input-complete over C_h, then each path of $Path_{\mathbb{O}rch}(\mu)$ can be extended as a path of trace $\mu.a$, and thus $Path_{\mathbb{O}rch}(\rho)$ is not empty.*

Case (b): *If a is of the form $c!t$, then as $Orch[Rem]$ $\overline{ioco}$ $IT(\mathbb{O}rch$, and as μ is a trace of both $Traces(IT(\mathbb{O}rch))$ and $Orch[Rem]$, then $\mu.a$ is a trace of $Traces(IT(\mathbb{O}rch))$, and thus $Path_{\mathbb{O}rch}(\mu.a)$ is not empty.*

Case (c): *If a is of the form $c?t$: let us suppose that paths of $Path_{\mathbb{O}rch}(\mu)$ cannot be extended with a transition whose action is $c?t$, then let us note p_1 one of such a paths that cannot be extended. Then, $U(p_\sigma^{\mathbb{O}})$ and $U(p_1)$ are two paths of $Obs(\mathbb{O}rch)$ verifying that there exists a trace σ_2 such that $traces(U(p_\sigma^{\mathbb{O}})) = traces(U(p_1)).c?t.\sigma_2$, and verifying that $U(p_1)$ cannot be extended with a transition carrying a $c?t$ action. This contradicts the fact that σ has been chosen among $UTraces(\mathbb{O}rch)$. It means that all paths of $Path_{\mathbb{O}rch}(\mu)$ can be extended as paths of the trace $\mu.a$. Thus $Path_{\mathbb{O}rch}(\rho)$ is not empty.*

Thus, we have established that for any trace σ of $Traces(Obs(Orch[Rem], C_o)) \cap UTraces(Obs(\mathbb{O}rch))$, for a possible trace κ' of $Orch[Rem]$ corresponding to σ, verifying that $U(\kappa') = \sigma$, then under the hypotheses that $Orch$ ioco $\mathbb{O}rch$, and that $\mathbb{O}rch$ is input complete over C_h, there necessarily exists a path p_σ in $Path(IT(\mathbb{O}rch))$ verifying $traces(p_\sigma) = \kappa'$.

As by Theorem 4.4.1, $Orch[Rem]$ $\overline{ioco}$ $IT(\mathbb{O}rch)$, then, as κ' is a common trace of $Orch[Rem]$ and $IT(\mathbb{O}rch)$ that is followed by an output o in $Orch[Rem]$, necessarily $\kappa'.o$ is also a trace of $IT(\mathbb{O}rch)$, for which the trace $U(\kappa'.o) = U(\kappa').o$ is a suspension trace of $Obs(\mathbb{O}rch)$.

Thus, we get:

$$\forall \sigma \in Traces(Obs(Orch[Rem], C_o)) \cap UTraces(Obs(\mathbb{O}rch)),\ \forall o \in O(C, M)$$

$$\sigma.o \in Traces(Obs(Orch[Rem], C_o)) \Rightarrow \sigma.o \in Traces(Obs(\mathbb{O}rch))$$

That is:

$$Obs(Orch[Rem], C_o)\ \overline{uioco}\ Obs(\mathbb{O}rch)$$

□

4.5 Conclusion

In this chapter we have defined a formal framework allowing us to relate conformance in context of orchestrators to their usual unit conformance. This is done in the **ioco** framework. In order to do so, we have taken into account the different possible status of the communication channels used by the orchestrator to communicate with the Web services. For any given channel, if it is is controlled by the tester, then it is said to be *controllable*; if it is not accessible at all, it is said to be *hidden*; and if it is not controlled but the tester can see the information going through it, then it is said to be *observable*.

By using this classification, and by means of some operations, we have shown how to obtain $Orch[Rem]$, and $Obs(Orch[Rem], C_o)$. The former one is the *assumed* $IOLTS$ with internal actions that represents the implementation of the orchestrator, $Orch$, in the context its remote part, Rem, when all the communication channels used by $Orch$ to communicate with Rem are either controllable or observable; and the later one is the *assumed* $IOLTS$ with internal actions representing $Orch$ in the context of Rem when there are hidden channels. Both $IOLTS$ with internal actions are obtained by some operations and modifications on the ones assumed to model their implementations.

Besides, he have also shown how to define the partial specification for an orchestrator in context. We achieve this by means also of the classification of the communication channels and of basically applying three operations over $\mathbb{O}rch$: full quiescence enrichment and remote input/output transformation for the observable and controllable cases (producing as a result $IT(\mathbb{O}rch)$), plus the hiding operation when there are some hidden channels (producing as a result $Obs(\mathbb{O}rch)$).

Finally, we have presented the conformance relations that we define in order to test orchestrators in context: $\overline{\textbf{ioco}}$ and $\overline{\textbf{uioco}}$. For these two relations we state two theorems allowing us to ensure that non conformance in context implies non conformance at the unit level. The first theorem makes use of $\overline{\textbf{ioco}}$ and concerns the case where all communication channels used for interactions between the orchestrator and the Web services are controllable or observable, while the second theorem makes use of $\overline{\textbf{uioco}}$, and concerns the case where some of the communication channels are hidden. Both theorems state that if we are able to build a test architecture so that we can test the conformance of $Orch$ in context of Rem, if an error is detected during this testing phase, such an error reveals in fact a non conformance of the orchestrator with respect to its specification in the sense of **ioco** (or **uioco**, depending on the testing architecture).

Chapter 5

Eliciting Web Service Behaviors

Contents

5.1 Introduction

FINDING the right Web service to ensure the right behavior of an orchestration is a difficult task, especially when the behavioral specification of the Web service is not available. Even if it is available, the composed system could be so large that one could face the *explosion problem* [Valmari 1998]. In this chapter we give the semantics of a foundation for an approach to test the compatibility of the Web services with respect to orchestrators (the approach itself is described in Sections 8.2,8.3) by taking into account only the expected behaviors of the Web services as they are expressed in the orchestrator's specification.

More specifically, we aim a testing if the Web services (when composed with an orchestrator) do not lead the whole orchestration into a deadlock situation. We begin by eliciting from the set of traces of the orchestrator a set of traces characterizing intended behaviors of the Web services. This set will serve as the basis of the approach presented in Chapter 8. In order to elicit such behaviors from the orchestrator's specification, we need to identify the situations where the Web services are allowed to go into a quiescent situation without leading the orchestration into a deadlock state. A Web service can be quiescent only if the orchestrator is not expecting an answer from it in order to move forward and if this answer is the only way for the orchestrator to move forward.

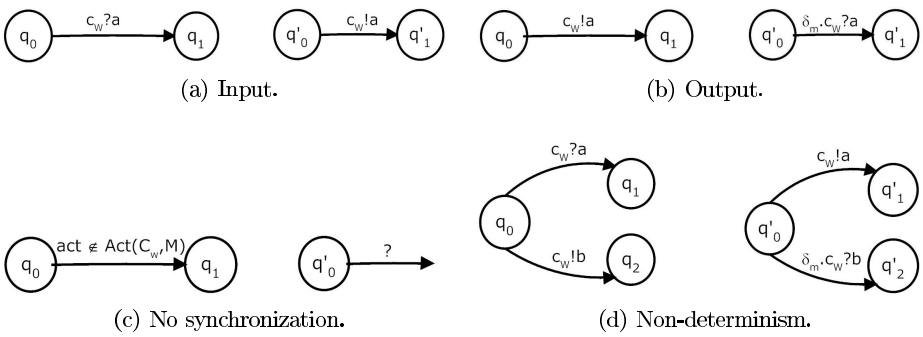
Once the valid quiescent situations of the Web services are identified, we proceed to transform the behaviors from the point of view of the orchestrator to the point

of view of the Web services. From these elicited behaviors, we the define a correctness relation between the Web services and a structure representing those elicited behaviors. That correctness relation reflects a compatibility property between the concerned Web service and the orchestrator.

Finally, the work presented in this chapter is similar to the one presented in [Angelis 2010], where authors use specifications of orchestrators to extract numerical behaviors that can be projected on Web services interfaces. The goal is to define unit test cases for Web services participating in the orchestration. They do this by taking into account only the information provided by the orchestrator's specification and the interface description of the Web services (WSDL). However, they use numerical model-based testing techniques, and in our approach we use symbolic techniques, as shown in Chapter 8.

We start this chapter with Section 5.2 by informally examining the different interaction situations of Web services with the orchestrator in order to identify when a Web service may remain quiescent while not causing deadlock situations. In Section 5.3, we introduce the technical operations used in Section 5.4 to identify intended behaviors of the Web services. Then, we define the correctness relation discussed above. Section 5.5 is a conclusion of the chapter.

5.2 Motivation



(a) Input. (b) Output.

(c) No synchronization. (d) Non-determinism.

Figure 5.1: Different status for Web services with respect to the orchestrator.

Orchestrators interact with Web services. From the orchestrator's point of view, we have no knowledge of the intended behaviors of the Web services. However, it is possible to define some fundamental compatibility constraints between an orchestrator and the Web services with which it interacts regardless of their functional roles. First of all, each Web service should be able to receive any message that the orchestrator sends to it. Concerning messages sent by Web services to the orchestrator, it may happen that those messages occur on channels that are not used by the orchestrator, because Web services are by nature pieces of software of generic usage that may be used by several orchestrators and that may provide more information

than required by a given orchestrator. Such messages will be simply ignored by the orchestrator of interest. The only important point is that messages *required* by the orchestrator are actually sent by the Web services. In the sequel of this section we discuss, intuitively, in which cases such a message is *required*. From the perspective of a user of a system composed of an orchestrator communicating with Web services, a basic requirement is that the whole system does not fall into a deadlock state (or just deadlock). Therefore, a message will be required whenever its absence would cause the system to deadlock. In Figure 5.1 we consider an orchestrator and a connected Web service, both of them denoted as $IOLTS$s. For each case (a), (b), (c), and (d), the left part (respectively right part) of the figure represents sequences of actions leading from a state q_0 (respectively q'_0) of the orchestrator (respectively of the Web service) to some states q_1, or q_1 and q_2 (respectively q'_1, or q'_1 and q'_2), depending on the considered case. q_0 is a state reached after some execution of the orchestrator and q'_0 is the corresponding state reached by the Web service. We consider four cases: for each of them the left part specifies a sequence of actions performed by the orchestrator and the corresponding right part denotes the required behavior of the Web service.

(a) In the case of Figure 7.1(a), the orchestrator requires a message a from the Web service to reach q_1. Moreover, the state of the orchestrator cannot change by some other sequence of actions. If the Web service does not send a from q'_0, that would lead to a deadlock situation to the system composed by the orchestrator and the Web services. Therefore, the Web service *is required* to send a and thus cannot be quiescent.

(b) In the case of Figure 7.1(b), the orchestrator sends the message a to the Web service. In that case the Web service may remain quiescent until it receives the message a. This quiescence is depicted by δ_m, where δ is the quiescence action as introduced in Section 3.2, and δ_m represents an arbitrary long finite sequence of δ.

(c) In the case of Figure 7.1(c), the orchestrator performs an action act which does not denote an interaction with the Web service. In this case, the intended behavior of the Web service depends on the actions that will follow act and thus we cannot infer any required behaviors for the Web service at this step.

(d) The case of Figure 5.1(d) denotes a situation where actions of the cases of Figures 7.1(a) and 7.1(b) are non-deterministically possible concerning the orchestrator, and in this situation the Web service may react as in the case of Figure 7.1(a) or as in the case of Figure 7.1(b).

A last case, which is not depicted, corresponds to a situation where no actions allow the orchestrator to evolve. In this situation the Web service may remain quiescent. However, regardless of that situation, an analysis of the four cases described above shows that authorizing the Web service to be quiescent may only be decided

when the orchestrator sends a message to the Web service. For this reason, we will impose that any orchestrator always finally invokes all its associated Web services unless its execution terminates. This corresponds to the *fairness invocation of Web services* property introduced later in Section 5.4 (Definition 5.4.1). This necessary restriction is reasonable with respect to real life orchestrators which generally characterize finite sequences of interactions with Web services.

5.3 Technical Preliminaries

We begin by introducing two technical operations that will be useful in order to elicit behaviors of Web services from behaviors of the orchestrator. These operations are defined over the traces of the $IOLTS$ representing the specification of the orchestrator.

The first operation is used to, given a particular set of communication channels, restrict the behaviors of the $IOLTS$ by projecting the associated traces on this chosen set of communication channels. The idea is to *remove* all actions that have *nothing to* do with the channels of interest.

Definition 5.3.1 (Projection) *Using notations of Definition 3.2.4, let* C *and* C *be two sets of communication channels verifying* $\mathsf{C} \subseteq C$*, and let* S *be a set of finite sequences of communication actions in* $(Act(C, M) \cup \{\delta\})^*$.

We note $S\downarrow_{\mathsf{C}}$ *the set* $\{sq\downarrow \mid sq \in S\}$*, where* $sq\downarrow$ *is defined as follows:*

- *if* $sq = \varepsilon$ *then* $sq\downarrow = \varepsilon$
- *if* $sq = sq'.a$*, then*
  - *for* $a \notin Act(\mathsf{C}, M) \cup \{\delta\}$*,* $sq\downarrow = sq'\downarrow$
  - *for* $a \in Act(\mathsf{C}, M) \cup \{\delta\}$*,* $sq\downarrow = (sq'\downarrow).a$

Example 5.3.1 *Let us consider the IOLTS of Figure 3.3 (Section 3.2), which corresponds to the IOLTS of the orchestrator of the Slot Machine example. The following is a sequence of communication actions* sq *of that IOLTS: the one going from state* q_0 *to* q_3 *and then back to* q_0 *through the transition labeled with* $u\_screen!win$*, that is, the sequence:*

$u\_start?token.sg\_generate!seed.sg\_sequence?sequence.u\_screen!win.$

If we define $\mathbf{C}$ *to be the set of communication channels:*

$\mathbf{C} = \{sg\_generate, sg\_sequence\}$,

then, the projection of sq *over* $\mathbf{C}$ *is the sequence:*

$sg\_generate!seed.sg\_sequence?sequence.$

That is, we simply remove *the communication actions that are not of our interest.*

Now we introduce a mirroring operation ([Jeannet 2005]) whose basic idea is to reverse all the communications' status: emissions become receptions and reciprocally. This is done so that the traces that are extracted from the specification of the orchestrator are no longer *perceived* from the perspective of the orchestrator, but from the one of the Web service.

Definition 5.3.2 (Mirroring) *Let C be a set of communication channels.*

The mirroring operation $Mir : (Act(C,M) \cup \{\delta\})^* \rightarrow (Act(C,M) \cup \{\delta\})^*$ is defined by:

- $Mir(\varepsilon) = \varepsilon$
- *if sq is of the form $sq'.c!v$, then $Mir(sq) = Mir(sq').c?v$*
- *if sq is of the form $sq'.c?v$, then $Mir(sq) = Mir(sq').c!v$*
- *if sq is of the form $sq'.\delta$, then $Mir(sq) = Mir(sq').\delta$*

Example 5.3.2 *Let us consider the IOLTS of Figure 3.3 (Section 3.2), which corresponds to the IOLTS of the orchestrator of the Slot Machine example. The following is a sequence of communication actions sq of that IOLTS: the one going from state q_0 to q_3 and then back to q_0 through the transition labeled with $u\_screen!win$, that is, the sequence:*

$u\_start?token.sg\_generate!seed.sg\_sequence?sequence.u\_screen!win.$

Then, the mirroring of sq is the sequence:

$u\_start!token.sg\_generate?seed.sg\_sequence!sequence.u\_screen?win.$

That is, we simply reverse *the sense of the communication actions.*

With these basic operations, we proceed to show how to elicit the required behaviors for the Web services from the specification of the orchestrator.

5.4 Correctness of Web Services with respect to Orchestrators

In this section we characterize the notion of correctness of Web services with respect to an orchestrator. We begin by identifying in the orchestrator's specification, $\mathbb{O}rch$, communication channels which are used to interact with Web services. Then, we identify under which circumstances the Web services are allowed to be quiescent.

Finally, we define the set of traces that are used to state the correctness property, and we state that property.

5.4.1 Web Services Communication Channels and Fairness Invocation

An orchestrator's specification $\mathbb{O}rch$ is just an *IOLTS* defined over a set C of distinguished channels: they are either communication channels with users, or communication channels dedicated to communicate with Web services. We suppose that a set $\mathbb{W}$ of so-called *Web service names* is given, which allows us to differentiate between the channels used to communicate with each of the Web services. The set of communication channels C is then of the form $C = C_c \cup \bigcup_{w\in\mathbb{W}} C_w$ (let us remind that C_c is the set of communication channels under the control of the user), and satisfies that the channels are associated exclusively to one remote part (user or Web service): for all $w \neq w' \in \mathbb{W}$, $C_c \cap C_w = \emptyset$ and $C_w \cap C_{w'} = \emptyset$. Note that concerning the partitions of the form C_c, C_h, and C_o introduced in Definition 4.3.4 in Section 4.3.1, $\bigcup_{w\in\mathbb{W}} C_w$ forms a partition of $C_r = C_h \cup C_o$. In the sequel, any set of communication channels C over which an orchestrator is defined, is supposed to be of the form $C_c \cup \bigcup_{w\in\mathbb{W}} C_w$ as described above.

As introduced in Section 5.2, we impose the *fairness invocation of Web services* on the Web services communicating with the orchestrator. This property is formally defined as follows.

Definition 5.4.1 (Fairness invocation of Web services property) *With notations of Definition 3.2.2, let $\mathbb{O}rch$ be an IOLTS over C.*

$\mathbb{O}rch$ satisfies the fairness invocation of Web services *property if and only if, for all $w \in \mathbb{W}$, such that $C_w \neq \emptyset$, we have:*

for any $p \in Path(\mathbb{O}rch)$, there does not exist an infinite sequence of transitions $(tr_i)_{i\in\mathbb{N}}$ with $source(tr_1) = target(p)$, and for all $i \in \mathbb{N}$, $target(tr_i) = source(tr_{i+1})$, and such that for all $i \in \mathbb{N}$, we have $act(tr_i) \notin O(C_w, M)$

5.4.2 Web Service Quiescence

We now show how to enrich the traces of an orchestrator's specification $\mathbb{O}rch$ with the quiescence symbol δ in order to specify under which circumstances a Web service may remain quiescent while interacting with the orchestrator. The underlying intuition is that the absence of reaction of the Web service should not cause any deadlock in the whole system's (the orchestration: orchestrator and Web services) execution. Let us note that even though we use the same symbol δ, it should not be understood in the same way as the one introduced in Section 3.2.1, which actually represents the situations where the system under test can be quiescent with respect to its specification; whereas in Definition 5.4.2, δ refers to the quiescence of some

Web services as it is observed from the orchestrator's point of view (*i.e.*, δ is added in the traces of the orchestrator's specification).

Definition 5.4.2 (Traces of a path with w-quiescence) *Let $\mathbb{O}rch = (Q, init, Tr)$ be an IOLTS over C, let w be a Web service of $\mathbb{W}$, and let $p \in Path(\mathbb{O}rch)$ be a path of $\mathbb{O}rch$.*

The set $SQ_w(p)$ of traces of p with w-quiescence *is minimally defined by:*

- *if $p = \varepsilon$, then $SQ_w(\varepsilon) = \{\varepsilon\}$*
- *if there does not exist tr'' such that $p.tr'' \in Path(\mathbb{O}rch)$, then $sq.\delta_m \in SQ_w(p)$ for any $sq \in SQ_w(p)$ and $\delta_m \in \{\delta\}^*$*
- *if $p = p'.tr \in Path(\mathbb{O}rch)$ with $tr \in Tr_{\mathbb{O}rch}$, then for any $sq' \in SQ_w(p')$ we have:*
  - *$sq'.act(tr) \in SQ_w(p)$*
  - *if $act(tr) \in O(C_w, M)$, then $sq'.\delta_m.act(tr) \in SQ_w(p)$ for any $\delta_m \in \{\delta\}^*$*

As discussed in Section 5.2, the second item of Definition 5.4.2 authorizes Web service quiescence for paths that cannot be continued, and the third item builds traces from consecutive actions of paths of the specification while authorizing Web service quiescence before the Web service is invoked.

Example 5.4.1 *Let us consider again our Example 3.2.5 of the Slot Machine. According to the IOLTS $\mathbb{G}$ $(\mathbb{O}rch)$ of Figure 3.3, the following is path p of $\mathbb{O}rch$:*

$(q_0, u\_start?token, q_1).(q_1, sg\_generate!seed, q_2).\ (q_2, .sg\_sequence?sequence, q_3).$
$(q_3, u\_screen!win, q_0).$

Then, the following is a trace, sq, with w-quiescence of p:

$u\_start?token.\delta^*.sg\_generate!seed.sg\_sequence?sequence.u\_screen!win.\delta^*.$

Note that this w-quiescence is different from the one introduced in Section 3.2. First, this quiescence is introduced over the traces of $\mathbb{O}rch$ and not over $\mathbb{O}rch$ itself. Second, even if we use the same symbol δ, this quiescence represents the fact that the Web service (the Sequence Generator service SG, in this case) can remain quiescent without leading the orchestration into a deadlock state as discussed in Section 5.2. In this case, SG can remain quiescent only before it is invoked (before executing the transition labeled by $sg\_generate!seed$), and most important, when executing the transition $sg\_sequence?sequence$, it is not allowed *to remain silent, since it would lead the orchestration into a deadlock situation because the orchestrator cannot evolve by itself: it* needs *the answer from SG.*

5.4.3 Traces for Web Services According to Orchestrators

The traces of $\mathbb{O}rch$ enriched with Web service quiescence, and from which we can extract the desired Web service behaviors, are defined as the set of all the paths with w-quiescence of $\mathbb{O}rch$.

Definition 5.4.3 (Traces of $\mathbb{O}rch$ with w-quiescence) *Let $\mathbb{O}rch$ be an $IOLTS$ over C*

The set of traces of $\mathbb{O}rch$ with w-quiescence *is defined as:*
$SQ_w(\mathbb{O}rch) = \bigcup_{p \in Path(\mathbb{O}rch)} SQ_w(p)$

We can now proceed to perform the transformations over the intended behaviors of the Web services from the point of view of the orchestration into suspension traces from the point of view of the Web services. Technically speaking, this is done by combining the projection operation removing useless actions, and the mirroring operation exchanging the role of emissions and receptions.

Definition 5.4.4 (w-traces according to $\mathbb{O}rch$) *Let $\mathbb{O}rch$ be an $IOLTS$ over C.*

The set $STr(\mathbb{O}rch, w)$ of w-traces according to $\mathbb{O}rch$ *is the set* $Mir(SQ_w(\mathbb{O}rch)\downarrow_{C_w})$

Example 5.4.2 *Following our example of the Slot Machine, if we consider the w-quiescence trace sq of Example 5.4.1, and if we define the set of communication channels $C_w = \{sg\_generate, sg\_sequence\}$ (which are the communication channels used by the slot machine's interface to interact with the Sequence Generator service), the w-trace according to $\mathbb{O}rch$, $Mir(SQ_w(\mathbb{O}rch)\downarrow_{C_w})(sq)$, is:*

$\delta^*.sg\_generate?seed.sg\_sequence!sequence.\delta^*.$

Which is the result of applying the mirror, projection and Web service quiescence to the sequence of communication actions obtained from the orchestrator of the Slot Machine example (with $C_w = \{sg\_generate, sg\_sequence\}$):

$u\_start?token.sg\_generate!seed.sg\_sequence?sequence.u\_screen!win.$

Thus, this trace can be used to verify if the implementation of the Sequence Generator service is compatible with $\mathbb{O}rch$.

We now define our correctness relation for Web services with respect to orchestrators. This relation is characterized by a property relating the Web services seen as usual SUTs and the w-traces according to the orchestrator, where w refers to the concerned Web service.

Definition 5.4.5 (Compatibility relation) *Let $\mathbb{O}rch$ be an IOLTS over C, let $w \in \mathbb{W}$,and let WUT be an SUT over C_{WUT}, verifying $C_w \subseteq C_{WUT}$.*

We note $Traces(WUT)|_{C_w}$ the subset of traces of WUT whose elements cannot be decomposed as $\sigma_1.c?v.\sigma_2$, where $\sigma_1, \sigma_2 \in (Act(C_w, M) \cup \{\delta\})^*$ and $c \notin C_w$.

WUT is compatible with $\mathbb{O}rch$ if and only if:

- $C_{WUT} \cap C_c = \emptyset$
- *for all $w' \neq w$, $C_{w'} \cap C_{WUT} = \emptyset$*
- *for any $\sigma \in (Traces(WUT)|_{C_w}) \downarrow_{C_w} \cap STr(\mathbb{O}rch, w)$,*

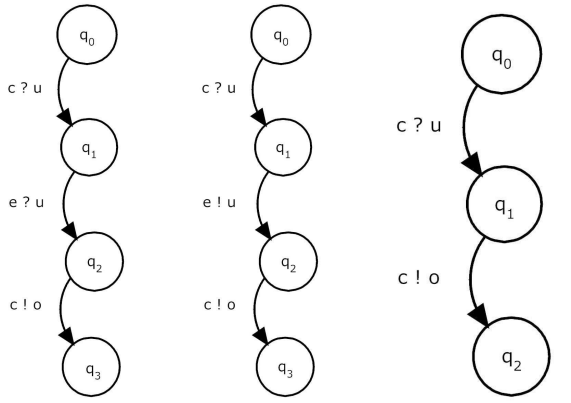
  $\exists o \in Act(C_w, M) \cup \{\delta\}$,
  $\sigma.o \in (Traces(WUT)|_{C_w}) \downarrow_{C_w} \Rightarrow \sigma.o \in STr(\mathbb{O}rch, w)$

Note that the compatibility condition given in Definition 5.4.5 is characterized as the usual **ioco** relation between a restriction of WUT's traces and $STr(\mathbb{O}rch, w)$. The restriction of traces of WUT, denoted $Traces(WUT)|_{C_w}$, simply contains WUT behaviors that result from invocations of the orchestrator: all traces containing invocations trough some communication channels that do not belong to C_w are just not considered. Then, the traces $Traces(WUT)|_{C_w}$ are projected on the set of communication channels C_w ($(Traces(WUT)|_{C_w}) \downarrow_{C_w}$) in order to make disappear all occurrences of outputs on channels that do not belong to C_w.

Let us consider two WUTs over a set of communication channels $\{c, e\}$ as depicted in Figures 5.2(a) and 5.2(b) (for the sake of readability, input completion and quiescence enrichment are omitted form the figures. Input completion would simply add loops on all states for all inputs that do not appear in a transition of some state).

Now, in Figure 5.2(c) we depict, abusively in the form of an $IOLTS$, the set of traces $STr(\mathbb{O}rch, w)$ of some orchestrator defined over C. w may be either WUT_1 or WUT_2 (both are defined over the set $\{c, e\}$ of communication channels). We have that $STr(\mathbb{O}rch, w)$ is $\{m.c?u.c!o.m' \mid m \in \{\delta\}^*, m \in \{\delta\}^*\}$ for both WUT_1 and WUT_2.

The usage of the operation $|_C$ removes all traces on WUT_1 that contains invocations on the channel e, since they do not have to be compared with traces of $\mathbb{O}rch$ ($\mathbb{O}rch$ never specifies what happens after such invocation). Thus, $Traces(WUT_1)|_C$ is $\{m.c?u.m' \mid m \in \{\delta\}^*, m \in \{\delta\}^*\}$. Applying $\downarrow_C$ on $Traces(WUT_1)|_C$ has no effect since elements of $Traces(WUT_1)|_C$ do not contain any occurrence of actions of the form $e?v_1$ (by definition of $|_C$) or $e!v_2$ (v_1 and v_2 are any elements of M). Thus, $(Traces(WUT_1)|_C) \downarrow_C = \{m.c?u.m' \mid m \in \{\delta\}^*, m \in \{\delta\}^*\}$ and, since it contains $c?u.\delta$ which does not belong to $STr(\mathbb{O}rch, WUT_1)$, we have that WUT_1 is not compatible with $\mathbb{O}rch$. Even though the presence of the output $c!o$ in WUT_1 would



(a) WUT_1. (b) WUT_2. (c) $STr(\mathbb{O}rch, w)$.

Figure 5.2: Compatibility relation example.

have let us suppose that WUT_1 interacts correctly with $\mathbb{O}rch$, in fact, the output $c!o$ results from two successive invocations: $c?u$ and $e?u$, while $\mathbb{O}rch$ specifies that $c!o$ can be observed after only one invocation $c?u$.

Now, considering WUT_2, we have that $Traces(WUT_2)|_C$ is $Traces(WUT_2)$, and the application of $\downarrow_C$ makes disappear the occurrences of $e!u$ in $Traces(WUT_2)|_C$. That is, $(Traces(WUT_2)|_C) \downarrow_C = \{m.c?u.c!o.m' \mid m \in \{\delta\}^*, m \in \{\delta\}^*\}$, which is exactly $STr(\mathbb{O}rch, WUT_2)$. Therefore, WUT_2 is compatible with $\mathbb{O}rch$.

In Chapter 8, we show how to adapt our testing framework in order to evaluate the correctness relation given in Definition 5.4.5.

5.5 Conclusion

In this chapter we have shown how to elicit behaviors from $\mathbb{O}rch$ in order to state a compatibility relation for Web services supposed to be plugged to the orchestrator. Those behaviors are precisely the ones specified by the orchestrator. First, we need to identify the valid quiescence situations of the Web services. We assume that the orchestrators we consider invoke all the Web services involved in the orchestration. This property is called *fairness invocation* of Web services and is necessary because the situations where the Web services are allowed to be quiescent are precisely determined when the orchestrator is interacting with them. Once these situations have been identified, we proceed to transform the traces from $\mathbb{O}rch$ by means of the mirroring and projection operations. We have defined the compatibility relation as a relation between traces of a Web service under test and traces reflecting behaviors of the Web services expected by the orchestrator. Then, the compatibility relation is formalized in a similar way to the **ioco** relation.

This work complements the one presented in Chapter 4, where the orchestrator

itself is tested in context to determine if it conforms to its partial specification. In this way, and under some hypothesis, we can test the conformance of the orchestrator with respect to its partial specification and the compatibility of the Web services (for which we do not have their specification), with respect to their elicited behaviors (expected behaviors) from the orchestrator's specification. Thus, we have shown that for orchestrations, which are a special type of component-based systems where there is a central component guiding the entire system, and because of this central component, we can perform some interesting tests that allow us to check the conformity of the orchestrator and the compatibility of the Web services with the orchestrator.

Part II

Symbolic Approach for Testing Orchestrators in Context

In this part of the thesis, and hereon, we no longer work with $IOLTS$s but with generic representations of them: *Input/Output Symbolic Transition Systems* $IOSTS$. $IOSTS$s are presented with detail later in this document, but here we focus in explaining why we make the transition from $IOLTS$s to $IOSTS$s. $IOSTS$s are concise representations of usual $IOLTS$s, introducing symbolic data to characterize internal states, to express firing conditions of transitions and to denote messages exchanged through communication channels. Thus, $IOSTS$s are a way to encode $IOLTS$s in such a way that it allows to explicit the relation and dependencies between the data.

We use $IOLTS$s for the theorization of our approach mainly because **ioco** is defined by using them as their models of the specifications and implementations (more strictly, it uses LTSs, which are similar structures regarding the $IOLTS$s). Thus, choosing the $IOLTS$s allows us to use the results obtained with LTS and **ioco** ([Tretmans 1996b, van der Bijl 2003b, van der Bijl 2003a]) and adapt them to define the formal basis of our approach, as presented in Part I. Besides, there are various works based on LTSs and its variations ([Jéron 2004, Tretmans 1996a]), so we can rely on the fact that they are well suited for formal works with conformance testing.

We use $IOSTS$s hereon mainly because:

- By using them we can reduce the situations of non-determinism. $IOSTS$s, as defined in the following chapter, introduce more precise information in the transitions. Especially, they introduce the notion of *guards*, which are conditions of the transitions. Thus, when there are two transitions leaving from the same state, in several cases, one can know which or why one of the two transitions is executed.

- $IOSTS$s introduce the concept of attribute variables. That is, instead of enumerating all possible real data when modeling systems, an $IOSTS$ uses these attribute variables (called also symbols). This abstraction of real data helps reducing the number of situations when the model reaches a large number of states. This problem is also known as *state explosion problem* [Valmari 1998].

- The testing algorithm we define (based on the one presented in [Gaston 2006]) is given in a symbolic way, based on symbolic execution techniques. Thus, by using $IOSTS$s to model the specifications of the systems under test, and by using the symbolic techniques presented later, we can use the algorithm to test the conformance of implementations with respect to their specifications.

- $IOSTS$s are a general abstraction of the $IOLTS$s, and as presented in the following chapter, for each $IOSTS$ there is a unique $IOLTS$ associated with it. Semantics of an $IOSTS$ is also given by means of an $IOLTS$. It means that both automata are closely related, and that the operations performed over $IOSTS$s could also be expressed with $IOLTS$s and vice-versa. Thus, the results of Part I apply also for $IOSTS$.

From the above mentioned facts, $IOSTS$s are a good option for modeling systems. Moreover, the prototype we developed uses the $IOSTS$s and the symbolic techniques as its basis, and it would have been more difficult to develop it by using only $IOLTS$s, since the models of the systems under text would have been significantly large.

Finally, theoretical results are advantageously presented in an LTS framework because this framework is simpler and because the size of models does not matter to state theoretical results. On the contrary, in practice, for defining models and for applying tools and algorithms, the size and algorithm efficiency become of primary interest. This mainly explains the interest of $IOSTS$s with respect to $IOLTS$s. Besides, this approach of developing the theory with the LTS framework and taking it to the symbolic framework has already been done, as well as adaptations of **ioco** to STS. Those approaches are similar to ours, the main difference consists in the way that test purposes are defined. In [Rusu 2000, Jeannet 2005], test purposes are given by means of automata. In [Frantzen 2006b], inputs are computed according to the observations until a verdict is emitted. Our approach is based on [Gaston 2006], where test purposes are selected from the unfolded specification of the system.

Chapter 6

Input/Output Symbolic Transition Systems

Contents

6.1 Introduction

In this chapter we present what is known as the the symbolic execution techniques that we use in order to test orchestrators in context and the compatibility of Web services. By using *IOSTS*s and *symbolic execution* techniques [Păsăreanu 2009, King 1975, Clarke 1976], we can *translate* the work done in Part I from the numeric version (based on *IOLTS*s) to the symbolic one (based on *IOSTS*s). The basic idea of the symbolic execution is to execute programs or specifications using symbols instead of concrete data as input values, and to derive *symbolic executions* (tree-like structures) in order to describe all possible computations, behaviors or situations of the program in a symbolic way.

*IOSTS*s introduce the notion attribute variables to represent concrete data and this is why they are extremely well fitted to be used together with the symbolic execution technique. The fact of using attribute variables instead of concrete data greatly reduces the state explosion problem [Valmari 1998], and introduces interesting advantages when performing the testing activity, as we show in next chapters.

The symbolic execution trees resulting from the symbolic execution of an *IOSTS* has some interesting properties that are worth mentioning: first, each symbolic state in the tree-like structure can have any number of child states, each one representing different situations that can happen in the system. Then, each state has a unique underlying execution path leading to it. It means that there is a concrete sequence of inputs that can be given to the system that will reach that path. Second, every time there is a decision to make, different branches of the tree represent the different decisions that can be taken and, what is more interesting, the decisions that have to be taken in order to reach a given state of one branch of the tree are different from the ones that have to be taken in order to reach any other state in any other branch. These decisions are stored in states of the tree (usually, when speaking about a tree structure, it has no states but nodes; however, in our work, we took the decision to call them states), and can be thought of as accumulators of properties which the inputs must satisfy in order to perform an execution that follows a particular path. Besides storing the decisions that are taken, we also store the history of transitions that were executed in the form of symbolic traces.

The symbolic execution tree represents all the possible behaviors of the system in a symbolic way, thus, if one wants to *execute* one of those behaviors in the system, it suffices to find concrete data for the different symbols in the corresponding branch satisfying the conditions accordingly, so one can guide the system to a specific state.

Finally, the symbolic execution has already been used to test different types of systems. In [Bentakouk 2009], authors use symbolic execution techniques in an approach similar to ours (their test purposes are a unique branch of the three, while in our case, we use more general test purposes represented by subtrees of the symbolic execution). Among other works using the symbolic execution technique for testing purposes (not only black-box model-based testing but also structural testing) we can refer to [Khurshid 2003, Xie 2005, Tillmann 2006, Inkumsah 2008].

In this chapter we begin with Section 6.2 by introducing the *IOSTS*s, for which we present their syntax and semantics. Then, we show how to apply symbolic execution techniques on *IOSTS*s in Section 6.3, where we also present some operations that will prove useful in the generation of test cases. We conclude the chapter with Section 6.4.

6.2 Symbolic Transition Systems

In this section we introduce the syntax of the *IOSTS*s as well as their semantics. We begin by introducing the data upon which the *IOSTS*s are built. This data is characterized by means of usual multi-sorted first-order logic. In practice, those data essentially comprise the Presburger arithmetic, and some enumerated types for ensuring constraint solving capacities.

Based on these data, we introduce the rest of concepts related to an *IOSTS*: their signature, the communication actions, and so on. Finally, as motivated by the introduction to Part II, we define how to obtain a unique *IOLTS* from a given

IOSTS.

6.2.1 First order Logic for *IOSTS*s

6.2.1.1 Datatypes

A **data type signature** is a couple $\Omega = (S, Op)$ where S is a set of type names and Op is a set of operation names, each one provided with a profile $s_1 \cdots s_{n-1} \rightarrow s_n$ (for $i \leq n$, $s_i \in S$). In the following, even if it is not explicitly defined, all data types are related to a given signature Ω.

Example 6.2.1 *Let us consider the signature $\Omega_{\mathbb{N}} = (S_{\mathbb{N}}, F_{\mathbb{N}})$ associated to the specification of elementary arithmetic.*

- $S_{\mathbb{N}} = \{nat, bool\}$
- $F_{\mathbb{N}} = \{0 :\rightarrow nat,$
  $succ : nat \rightarrow nat,$ *(successorship)*
  $+ : nat \times nat \rightarrow nat,$ *(addition)*
  $<: nat \times nat \rightarrow bool,$ *(inequality less than)*
  $true :\rightarrow bool,$
  $false :\rightarrow bool\}$

Let $V = \coprod_{s \in S} V_s$ be a set of **typed variable names**. The set of **Ω-terms** with variables in V is denoted $T_\Omega(V) = \bigcup_{s \in S} T_\Omega(V)_s$ and is inductively defined as usual over Op and V. The **application** $Type : T_\Omega(V) \rightarrow S$ is the function such that for each $t \in T_\Omega(V)_s$, $Type(t) = s$. In the following, we overload the notation $Type$ by defining $Type(X) = s$ for any set $X \subseteq V_s$. $T_\Omega(\emptyset)$ is simply denoted T_Ω.

Example 6.2.2 *Using the signature $\Omega_{\mathbb{N}} = (S_{\mathbb{N}}, F_{\mathbb{N}})$, let us consider the typed variable names $V = V_{nat} \cup V_{bool}$, with $V_{nat} = \{x, y\}$ and $V_{bool} = \emptyset$. The following are some $\Omega_{\mathbb{N}}$-terms with variables in V ($T_{\Omega_{\mathbb{N}}}(V)$):*

$$0, x, y, succ(0), succ(x), succ(y), +(0, 0).$$

The following are some examples of couples of the form (t, s), with $Type(t) = s$, according to $Type : T_{\Omega_{\mathbb{N}}}(V) \rightarrow S$:

$$(0, nat),\ (succ(x), nat),\ (+(0, 0), nat).$$

Notation 6.2.0.1 *In the following, for the sake of simplicity, we adopt an infix notation instead of the prefix one. For instance, we note $0 + 0$ instead of $+(0, 0)$.*

A **Ω-substitution** is a function $\sigma : V \to T_\Omega(V)$ preserving types, that is, associating to each variable v of type s, a term t in $T_\Omega(V)$ also of type s. In the following, we note $T_\Omega(V)^V$ the set of all Ω-substitutions of the variables V. Any substitution σ may be canonically extended to terms (with $\sigma(f(t_1 \dots t_n)) = f(\sigma(t_1)) \dots f(\sigma(t_n))$). The **identity Ω-substitution** over the variables V, Id_V, is defined as $Id_V(v) = v$ for all $v \in V$.

Example 6.2.3 *Based on Examples 6.2.1 and 6.2.2, we can define the following* $\Omega_\mathbb{N}$*-substitution* $\sigma : V \to T_{\Omega_\mathbb{N}}(V)$ *such that* $\sigma(x) = x+1$ *and* $\sigma(y) = y+1$*, and is noted* $[x \to x+1, y \to y+1]$.

For example, we have then $\sigma(x+y) = (x+1)+(y+1)$.

The set $Sen_\Omega(V)$ of all typed equational **Ω-formulae** contains the truth values $true$, $false$ and all formulae built using the equality predicates $t = t'$ for $t, t' \in T_\Omega(V)_s$, and the usual connectives $\neg, \vee, \wedge$.

Example 6.2.4 *Based on the signature and terms of Examples 6.2.1 and 6.2.2, we can define the following formulae:*

- $x = y$
- $\neg(0 = succ(x))$
- $x + 0 = x$
- $x + succ(y) = succ(x + y)$

6.2.1.2 Semantics

A **Ω-model** is a set M whose elements are associated with a type in S, and we note $M_s \subseteq M$ the subset of M whose elements are associated with s. Each $f : s_1 \cdots s_n \to s \in Op$, is interpreted as a function $f_M : M_{s_1} \times \cdots \times M_{s_n} \to M_s$.

We define **Ω-interpretations** as applications ν from V to M preserving types and extended to terms in $T_\Omega(V)$. M^V is the set of all Ω-interpretations of V in M.

A **model M satisfies a formula** φ, denoted by $M \models \varphi$, if and only if, for all interpretations ν, $M \models_\nu \varphi$, where $M \models_\nu t = t'$ is defined by $\nu(t) = \nu(t')$, and where the truth values and the connectives are handled as usual. Given a model M and a formula φ, φ is said *satisfiable* in M if there exists an interpretation ν such that $M \models_\nu \varphi$.

M denotes a given Ω-model and values of M are qualified as **concrete**.

In the rest of this thesis, we suppose that data types of our $IOSTS$s correspond to the generic signature $\Omega = (S, Op)$, and are interpreted in a fixed model M. Ω contains the usual datatypes *bool*, *integer*, and *string*, provided with classical operations as + (addition), − (subtraction), and ∗ (multiplication). This is left implicit in the sequel.

6.2.2 Syntax of the $IOSTS$s

Based on the datatypes presented in Subsection 6.2.1, we define the *attribute variables*, which are typed variables used to store data inside states of the transition systems that we use. We note this set as $A = \bigcup_{s \in S} A_s$. Thus, messages in an $IOSTS$ can be either attribute variables or terms over them, and which are sent over what we define as *communication channels*. We note C the set of communication channels that are used by a given $IOSTS$. As for $IOLTS$, messages sent through communication channels can be of two types:

- Outputs; which are emissions of values from the system to its environment over a communication channel in C, denoted $c!t$, where c is the name of the channel and t is the value emitted and defined as a term of $T_\Omega(A)$.
- Inputs; which are receptions over a channel in C, denoted as $c?x$ where c is the name of the channel and x is the attribute variable where the received value is stored.

$IOSTS$ are defined over a *signature*, which is a structure that contains all the variables and communication channels that can be used by an $IOSTS$.

Definition 6.2.1 ($IOSTS$ signature) *Let A be a set of typed variables and C be a set of communication channels.*
An $IOSTS$ signature Σ is defined as the tuple (A, C)

Remark 2 *In the following, we consider that all $IOSTS$ are defined over a signature Σ.*

Example 6.2.5 *Let us consider the signature $\Sigma_{sm} = (A_{sm}, C_{sm})$ that is used later in our examples, with the set of attribute variables and communication channels given by:*

$A_{sm} = \{token, seed, win\_seq, seq\}$.

$C_{sm} = \{u\_start, sg\_generate, sg\_sequence, u\_screen\}$.

Usage of the set of attributes variables and communication channels is introduced later in Example 6.2.7. Here, it is important to notice that communication channels of the form u_start, where the u in the name of the communication channel states that it is used for interactions with the user by means of the operation $start$, or of the form $sg\_generate$, where sg states that the communication channel is used to interact with the Web service sg by means of the operation $generate$.

We now define the *communication actions* that can be used by an $IOSTS$. Communication actions can be inputs or outputs sent by communication channels, or they can be *internal actions*. Internal actions represents the fact that the system is executing an operation that does not involve any communication with the exterior. As for $IOLTS$s, the internal action is generically denoted by τ.

Definition 6.2.2 (Communication actions) *Let Σ be an IOSTS signature. The set of* communication actions *over Σ is defined as $Act(\Sigma) = I(\Sigma) \cup O(\Sigma) \cup \{\tau\}$, where:*

- $I(\Sigma) = \{c?x \mid x \in A, c \in C\}$
- $O(\Sigma) = \{c!t \mid t \in T_\Omega(A), c \in C\}$

Example 6.2.6 *Based on the signature Σ_{sm} of Example 6.2.5, we can give some communication actions in $Act(\Sigma_{sm})$:*

$sg\_generate!seed$, which represents an output sent to a component in the system by means of the term with a single variable.

$u\_screen!'lose'$, which represents an output sent to the user by means of a constant. This constant (and another one introduced later, $'win'$) is used later in Example 6.2.7 to denote constant strings *that represent messages sent to the user. Since constants never change of value, we did not include them in the set of attribute variables A_{sm}: they belong to the set of terms Ω.*

*IOSTS*s are structures composed of *states*, an *initial state*, and *transitions* going from one state to another. Transitions are composed of: *guards*, which are conditions that have to be satisfied in oder to fire the transition; *communication actions*, introduced in Definition 6.2.2; and *affectations*, representing the modifications on the attribute variables when firing the transition.

Definition 6.2.3 (*IOSTS* over Σ) *Let $\Sigma = (A, C)$ be an IOSTS signature. An* IOSTS *over Σ is a tuple $\mathbb{G} = (Q, init, Tr)$ where:*

- *Q is a set of* state names
- *$init \in Q$ is the* initial state
- *$Tr \subseteq Q \times Sen_\Omega(A) \times Act(\Sigma) \times T_\Omega(A)^A \times Q$ is a set of* transitions

Notation 6.2.3.1 *In the following, for any IOSTS $\mathbb{G}$ of the form $(Q, init, Tr)$ over Σ, we use the notations $Q_\mathbb{G}$, $init_\mathbb{G}$, and $Tr_\mathbb{G}$ in order to refer, respectively, to Q, $init$, and Tr.*

In the same way, for any transition $tr \in Tr_\mathbb{G}$ of the form $(q, \varphi, act, \rho, q')$ we use the notations $source(tr)$, $guard(tr)$, $act(tr)$, $sub(tr)$, and $target(tr)$ in order to refer, respectively, to q, φ, act, ρ, and q'. If $sub(tr)$ does not affect any variable in A, we note it Id_A, which stands for the identity function over the set A.

We represent an *IOSTS* in the standard way, that is, by a directed, edge-labeled graph where nodes represent states and edges represent transitions. Transitions are represented with an arrow $\rightarrow$ representing the flow of the communication from their source state to their target state. Moreover, if $guard(tr) = true$, then it is omitted in the graph; if $sub(tr) = Id_A$, then it is also omitted in the graph.

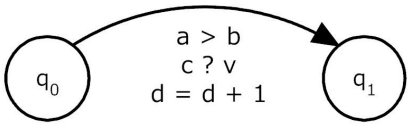


Figure 6.1: One transition of an $IOSTS$.

Before giving an example of an $IOSTS$, let us say more about the notion of a *transition*. Basically, a transition represents the change of a state in the system. This change of state can be due to a reception or emission of a value (output $c!t$ or input $c?x$), or to the execution of an internal action (τ) that does not involve any communication with the environment. Besides of the *communication action*, a transition also has important information associated with it, namely, the necessary condition that has to be valid in order to execute the transition (the *guard*), as well as the attribute variables that are modified from one state to another. Let us consider the transition tr depicted on Figure 6.1. Here, the set of attribute variables is $\{d, v\}$, a and b are terms of Ω, and c is a communication channel of C. In order to go from state q_0 to state q_1, the condition $a > b$ ($guard(tr)$) has to be satisfied. Only when the value of the attribute variable a is greater than the one of b the transition *can* be fired. If this condition is true, then the system can receive a value over the communication channel c that will be stored on the attribute variable v. Once the reception over the variable v is performed, the affectation is executed (d is incremented in one unit) and the system is finally in state q_1. Note that the order of *execution* of the elements of the transition is: first we check $guard(tr)$; then a new value for the variable v can be received over c; and finally, the affectation is executed and the system is in state q_1.

Thus, $IOSTS$s have important information of the modeled system, since, besides the information provided by the transitions, they represent its status by means of the variables in each state, and show, in each state, the affectations that are made over such variables.

Example 6.2.7 *Let us recall the Slot Machine example introduced in Part I in Chapter 3: the system is composed of the orchestrator (slot machine's interface, SM) and of one remote Web service (the Sequence Generator service, SG). The user interacts with SM by sending it a token, then SM interacts with SG in order to determine a sequence for the user, which is finally notified whereas he or she won (because his or her assigned sequence was equal to the winner sequence) or not.*

Figure 6.2 depicts the $IOSTS$ $\mathbb{G}$ for the Slot Machine example. Its signature is the one introduced in Example 6.2.5 and its set of communication actions the one that can be obtained from $\Sigma_{sm} = (A_{sm}, C_{sm})$. The set of states is $\{q_0, q_1, q_2, q_3\}$. Transition $q_0 \xrightarrow[\substack{seed=random(token)\\ win\_seq=random(seed)}]{u\_start?token} q_1$ denotes the reception of a value sent from the user through the communication channel u_start and stored in the attribute vari-

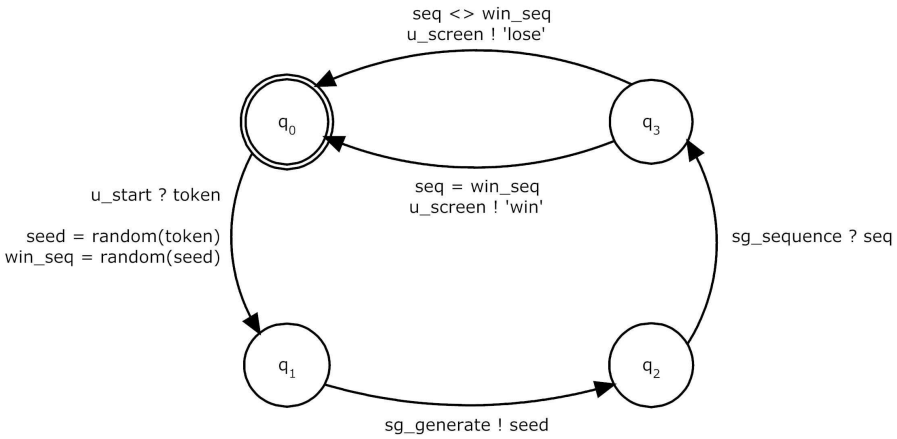


Figure 6.2: $IOSTS$ for the slot machine example.

able token; as well as the modification of the attribute variables seed and win_seq according to the received value and by means of a dedicated computation function $random(integer)$.

Its structure is similar to its respective IOLTS of Figure 3.3 in Section 3.2.1, but we can notice that the IOSTS is more complete*, in the sense that it also takes into account the firing conditions of each transition, it represents the status of the variables in each state, and shows the affectations that are made over such variables. Finally, as glimpsed before, we consider the* $random(integer)$ *function as a dedicated computation function over its argument (for instance,* $random(x) = x * 2 + 1$*).*

6.2.3 Behaviors of the $IOSTS$s

The behavior of an $IOSTS$, also called its *semantics*, is defined by the notion of the interpreted traces that can be generated from it. Traces are possible successions of communication actions that are specified by an $IOSTS$. However, those succession of communication actions are to be interpreted in order to get real values. Therefore, we give a series of definitions that are needed in order to define the behavior of an $IOSTS$. We start by defining the notion of *concrete actions*, which are the interpretations of the communication actions.

Definition 6.2.4 (Concrete actions) *Let* $\Sigma = (A, C)$ *be an IOSTS signature.*

Se set of concrete actions *over* Σ *is the defined as* $Act_M(\Sigma) = I_M(\Sigma) \cup O_M(\Sigma) \cup \{\tau\}$*, where:*
$I_M(C) = \{c?v \mid c \in C, v \in M\}$
$O_M(C) = \{c!v \mid c \in C, v \in M\}$

The value v is the interpretation of the received or emitted terms.

Example 6.2.8 *Based on Figure 6.2, we take the communication action of the transition going from q_0 to q_1, that is, u_start?token. One possible interpretation, i.e., a* concrete action *of the attribute variable token, is the concrete value* 1*, that would give the concrete action u_start?1, corresponding to the fact that the environment (the user) sends the value* 1 *that is to be stored in the attribute variable token.*

Traces of an $IOLTS$ are built from sequences of transitions. The semantics of an $IOSTS$ is then built upon the semantics that we give to the transitions.

Definition 6.2.5 (Semantics of a transition) *Let $\mathbb{G} = (Q, init, Tr)$ be an $IOSTS$ over Σ.*
The semantics of a transition *$tr \in Tr$ of the form $(q, \varphi, act, \rho, q')$ is the relation $Run(tr) \subseteq M^A \times Act_M(\Sigma) \times M^A$, such that $(\nu_i, act_M, \nu_f) \in Run(tr)$ if and only if:*

- *if act is of the form $c!t$, then $M \models_{\nu_i} \varphi$, $\nu_f = \nu_i \circ \rho$ and $act_M = c!\nu_i(t)$*
- *if act is of the form $c?x$, then $M \models_{\nu_i} \varphi$, there exists ν_a such that $\nu_a(z) = \nu_i(z)$ for every $z \neq x$, $\nu_f = \nu_a \circ \rho$, and $act_M = c?\nu_a(x)$*
- *if act is of the form τ then $M \models_{\nu_i} \varphi$, $\nu_f = \nu_i \circ \rho$ and $act_M = \tau$*

Notation 6.2.5.1 *In the following, $Run(tr)$ stands for the* run of a transition *and, for any run r of $Run(tr)$ of the form (ν_i, act_M, ν_f), we use the notations $source(r)$, $act(r)$, and $target(r)$ in order to refer, respectively, to ν_i, act_M, and ν_f.*

The application ν_i is the interpretation of variables *before* executing the transition, and ν_f is the interpretation of the variables *after* the execution of the transition. act_M is the interpretation of either the value sent or received in the communication action of the transition or the internal action τ.

Example 6.2.9 *Based on Figure 6.2, let us consider again the transition going from q_0 to q_1. A possible run for the transition is the case where the attribute variables before executing the transition have the values:*

$[token \to 0,\ seed \to 0,\ win\_seq \to 0,\ seq \to 0]$,

and with the final values (after executing the transition):

$[token \to 1,\ seed \to 6,\ win\_seq \to 4,\ seq \to 0]$ *(the values of seed and win_seq depend on the definition of the function random, which is not described here).*

Paths are sequences of transitions beginning at the initial state of the $IOSTS$.

Definition 6.2.6 (Paths of an $IOSTS$) *Let $\mathbb{G} = (Q, init, Tr)$ be an $IOSTS$ over Σ.*
The set of paths*, denoted, by abuse, $Path(\mathbb{G})$, contains all the finite sequences $tr_1 \cdots tr_n$ of transitions of Tr such that:*

- *$source(tr_1) = init$*
- *for every i, $1 \leq i < n$, $target(tr_i) = source(tr_{i+1})$*

We define, for any $q \in Q$, the set $Path(q) \subseteq Path(\mathbb{G})$, such that $p \in Path(\mathbb{G})$, $p = tr_1 \cdots tr_n$, $p \in Path(q)$ if and only if $target(tr_n) = q$

Notation 6.2.6.1 *In the following, for a path $p = tr_1 \cdots tr_n$, the length of the path p is denoted $length(p) = n$.*

Note that the set $Path(q)$ contains all the paths such that the target state of the last transition of the path is q.

Example 6.2.10 *Based on the $IOSTS$ of Figure 6.2, a finite of that $IOSTS$ is the one composed by the transitions going from state q_0 to q_2. That is, the path composed by the transitions:*

$(q_0, true, u\_start?token, \rho, q_1).(q_1, true, sg\_generate!seed, Id_A, q_2)$*, with*
$\rho : [seed \rightarrow random(token), win\_seq \rightarrow random(seed)]$.

This path is also an element of the set $Path(q_2)$.

The *run of a path* is the sequence of runs of the transitions in the path, where the target state shares the variable interpretation with the source state of the consecutive transitions.

Definition 6.2.7 (Runs of paths) *Let $\mathbb{G}$ be an $IOSTS$ over Σ.*

The set of runs of a path *p, denoted $Run(p)$, for a path $p = tr_1 \cdots tr_n$ in $Path(\mathbb{G})$, are sequences $r_1 \cdots r_n$ such that:*

- *for all $i \leq n$, r_i is a run of tr_i, $r_i \in Run(tr_i)$*
- *for all $i < n$, $target(r_i) = source(tr_{i+1})$*

We define the runs of paths *of $\mathbb{G}$, denoted $RP(\mathbb{G})$, as*
$RP(\mathbb{G}) = \bigcup_{p \in Path(\mathbb{G})} Run(p)$

For the same testing reasons that for $IOLTS$s (Section 3.2, Definition 3.2.7), we make the hypothesis that the $IOSTS$ we use are *strongly responsive*, i.e., they always eventually enter in a quiescent state, or, what is the same, they do not have infinite τ transitions or outputs.

Definition 6.2.8 (Strongly responsive $IOSTS$) *Let $\mathbb{G}$ be an $IOSTS$.*

Let $Live(\mathbb{G})$ be the greatest subset of $Path(\mathbb{G})$ verifying that for all $p \in Live(\mathbb{G})$, $\exists\, tr \in Tr$, with $act(tr) \in O(\Sigma) \cup \{\tau\}$ such that $p.tr \in Live(\mathbb{G})$.

$\mathbb{G}$ is strongly responsive *if and only if $Live(\mathbb{G}) = \emptyset$*

$Live(\mathbb{G})$ of Definition 6.2.8 represents the set of behaviors of $\mathbb{G}$ that can continue infinitely without any intervention from the exterior. Since in the test activity it is necessary to interact with the system, and more especially, observe its reactions, in our work we test only systems that can be modeled by means of strongly responsive $IOSTS$s, or, what is the same, only $IOSTS$s whose subset $Live(\mathbb{G})$ is the empty set.

The interpretations of the communication actions of the paths are called *concrete traces*, and are defined as follows.

Definition 6.2.9 (Concrete traces) *Let $\mathbb{G}$ be an $IOSTS$ over Σ, and let $p \in Path(\mathbb{G})$.*

The set of concrete traces *of a path p, denoted $traces(p)$, is the set $\bigcup_{r \in Run(p)} \{traces(r)\}$, where $traces(r)$ is inductively defined as follows:*

- *if r is ε, then $traces(r)$ is ε*
- *if p is of the form $p'.tr$ and r is of the form $r'.a$, where $r' \in Run(p')$ and $a \in Run(tr)$, then:*
  - *if $act(a)$ is τ, we have $traces(r)$ is $traces(r')$*
  - *if $act(a)$ is not τ, we have $traces(r') = traces(r).act(a)$*

Example 6.2.11 *If we take the path in Figure 6.2 going from states q_0 to q_3, and then again to q_0 by the transition with the communication action $u\_screen!'win'$, a possible sequence of concrete traces of that path is:*

$u\_start?1.\ sg\_generate!6.\ sg\_sequence?4.\ u\_screen!'win'.$

Finally, the behaviors of an $IOSTS$, also called its *semantics*, are defined as the set of all the concrete traces that can be obtained from its paths.

Definition 6.2.10 (Semantics of an $IOSTS$) *Let $\mathbb{G}$ be an $IOSTS$ over Σ. The semantics of $\mathbb{G}$ is defined as:*

$$Traces(\mathbb{G}) = \bigcup_{p \in Path(\mathbb{G})} traces(p)$$

For testing reasons, as we did in Part I, what is interesting for us is the set of suspension traces which can be built from the traces of the system. Suspension

traces are the ones that are built by taking into account the quiescent situations of the system, that is, the situations where it cannot evolve by itself. Thus, suspension traces represent the *real* semantics of the systems, especially for testing purposes. In our approach, we define such traces by means of the symbolic execution techniques, as introduced in Definition 6.3.9. Until now, we have defined the automata under which we base the symbolic execution: the $IOSTS$s; as well as the traces that are defined by them in a natural way.

6.2.4 From $IOSTS$s to $IOLTS$s

As discussed in the introduction to Part II, in this section we show the relation between the $IOLTS$s and the $IOSTS$s in a more systematic way. More specifically, we show how to *build* an $IOLTS$ from an $IOSTS$. Basically, it suffices to interpret the attribute variables introduced in the $IOSTS$s (in the states and in the transitions) and define enumerated states and transitions with those interpretations. Thus, $IOSTS$s are just intentional definitions of $IOLTS$, and from any given $IOSTS$ we can obtain an $IOLTS$ by applying the following definition.

Definition 6.2.11 ($IOLTS$ associated to an $IOSTS$) *Let $\mathbb{G}$ be an $IOSTS$ over $\Sigma = (A, C)$.*
The associated $IOLTS$ of $\mathbb{G}$, *denoted $\mathbb{G}_{LTS} = (Q_{\mathbb{G}_{LTS}}, init_{\mathbb{G}_{LTS}}, Tr_{\mathbb{G}_{LTS}})$ over $Act(C, M)$, is such that:*

- $\forall q \in Q_{\mathbb{G}}$, *the set of* numerical states of q *is defined as* $num(q) = \{(p, \nu) \mid p \in Path(q) \wedge \nu \in M^A\}$
- *the set of* numerical states of $\mathbb{G}$ *is defined as* $num(\mathbb{G}) = \bigcup_{q \in Q} num(q)$

Then:

- $Q_{\mathbb{G}_{LTS}} = num(\mathbb{G}) \cup \{init_{\mathbb{G}}\}$
- $init_{\mathbb{G}_{LTS}} = init_{\mathbb{G}}$
- $Tr_{\mathbb{G}_{LTS}}$ *is defined as follows:*
  - *for all* $\nu \in M^A$, $(init_{\mathbb{G}_{LTS}}, \tau, (\varepsilon, \nu)) \in Tr_{\mathbb{G}_{LTS}}$
  - *for all* $tr = (q, \varphi, act, \rho, q') \in Tr_{\mathbb{G}}$, *for all* $(p, \nu) \in num(q)$ *such that* $M \models_\nu \varphi$, *then* $((p, \nu), act_M, (p.tr, \nu')) \in Tr_{\mathbb{G}_{LTS}}$, *where* act_M *and* ν' *are defined as follows:*
    - *if act is of the form* $c!t$ *(resp.* τ*) then* $\nu' = \nu \circ \rho$, *and* $act_M = !c\_\nu(t)$ *(resp.* $act_M = \tau$*)*
    - *if act is of the form* $c?x$ *then there exists* ν^a *such that* $\nu^a(z) = \nu(z)$ *for all* $z \neq x$, $\nu' = \nu^a \circ \rho$, *and* $act_M = ?c\_\nu^a(x)$

Note that $\mathbb{G}_{LTS}$ has a tree-like structure: it has an initial state (root) $init_{\mathbb{G}_{LTS}}$ and for any state of $Q_{\mathbb{G}_{LTS}}$ there is at most one path leading to it.

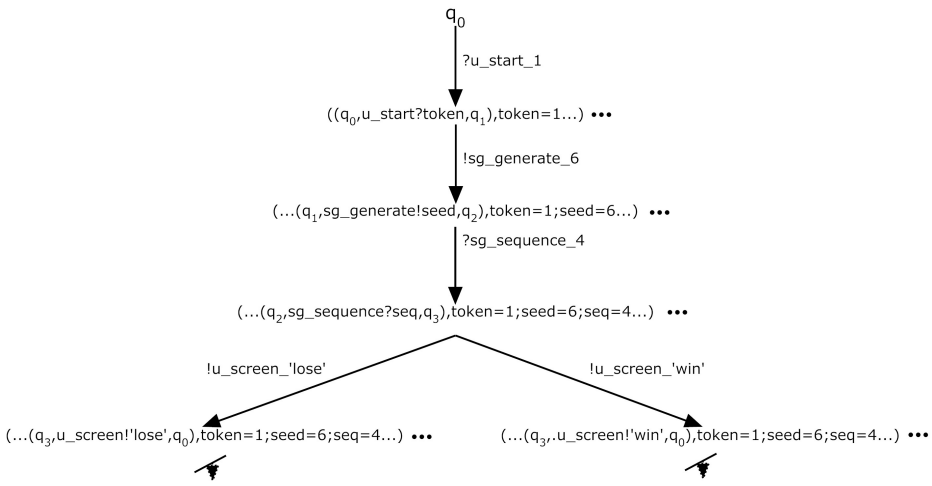


Figure 6.3: $IOLTS$ $\mathbb{G}_{LTS}$ for the $IOSTS$ of Figure 6.2.

The first two items in Definition 6.2.11 indicate how to build the set of states of $\mathbb{G}_{LTS}$. For each state in $q \in Q_{\mathbb{G}}$, we can define its set of respective numerical states by defining the set of pairs whose first element is a path leading to q, and the second one can be any interpretation of the attribute variables.

The first item in the construction of $Tr_{\mathbb{G}_{LTS}}$ denotes the construction of the states that can be reached from the initial state. Such set of states is the set of all the possible interpretations of the attribute variables in A. The second item shows how to build the transitions between the set of numerical states. Basically, if the communication action in $tr \in Tr_{\mathbb{G}}$ is an output of the form $c!t$, then the numerical transition has as final states all the possible interpretations of the attribute variables in the affectation of the transition. If the communication action is an input of the form $c?x$, then the transition has as final states the resulting interpretation of both attribute variables in the affectation of tr and the variables in the communication action.

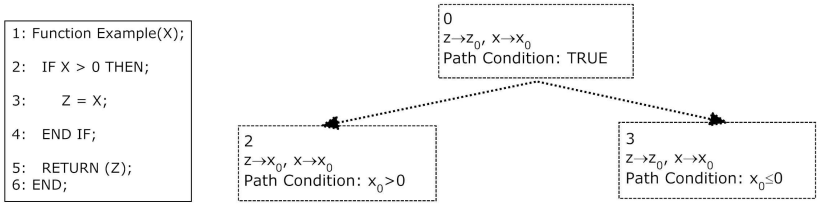
Remark 3 *The traces that can be obtained from the LTS $\mathbb{G}_{LTS}$ are, by construction, the traces that can be obtained from the IOSTS $\mathbb{G}$, since the traces of an IOSTS are constructed by interpreting the attribute variables of the paths in $Path(\mathbb{G})$. Thus, any trace in $\mathbb{G}_{LTS}$ is also a concrete trace in $\mathbb{G}$, and reciprocally.*

Example 6.2.12 *Figure 6.3 depicts one part of the associated IOLTS $\mathbb{G}_{LTS}$ to the IOSTS $\mathbb{G}$ of Figure 6.2. Note that, for readability reasons, not all the information nor every interpretation is depicted, and the tree was cut when reaching the initial state for the second time. Also, the final structure is similar to the IOLTS for the Slot Machine example, of Figure 3.3 (Section 3.2), but here $\mathbb{G}_{LTS}$ has a tree*

structure, so the loops are presented as repeated successions of transitions. In other words, $\mathbb{G}_{LTS}$ is the unfolded version of the $IOLTS$.

6.3 Symbolic Execution

Symbolic execution [King 1975] is a technique that consists in executing a program by using arbitrary symbols instead of real data. Thus, computational operations involving conditions, assignments, etc., receive symbols as inputs and produce symbolic formulae as outputs. This technique has the advantage, first, of reducing the state explosion problem, and second, of fitting extremely well with the $IOSTS$s: the $IOSTS$s are then executed by using symbols instead of concrete data. The idea is to generate a tree-like structure which represents *all* the behaviors accepted by the $IOSTS$ in a symbolic way.



(a) Program code. (b) Symbolic execution.

Figure 6.4: Symbolic execution of a simple program example.

To better understand the idea of the symbolic execution, let us consider the Figure 6.4. Figure 6.4(a) shows the code of a function $Example(X)$. Figure 6.4(b) shows its associated symbolic execution, where for each variable in the program, a symbol is introduced to denote its initial value. Note that it has a tree-like structure, since each time a decision has to be taken (for example, when encountering a while or an if instruction), a new branch is added to the tree. This branch represents the decision that was taken or condition that has to be met in order to reach the states of the branch (or states of the system). Thus, each state has an associated *path condition*, which is the set of conditions that have to be met in order to reach that state. In order to reach the left branch, x_0 must be greater than 0. Besides, in the tree-like structure, we can store more relevant information. In the figure, we can see that in each state of the tree we store the symbols representing the value of the variables (symbolic values) and the *path condition* of the state.

We begin this section by introducing the syntax-related notions of the symbolic execution, as well as how to obtain it from a given $IOSTS$. Then we define the behaviors associated with a symbolic execution, and we finish the section by introducing two operations (quiescence enrichment and τ-reduction, Definitions 6.3.9 and 6.3.10) that are going to be used in the next chapters in order to test orchestrators in in context and elicit behaviors to test Web services.

6.3.1 Generating a Symbolic Execution Tree

We start by introducing the values, called *fresh variables*, that we use when applying the symbolic execution technique to $IOSTS$s. Those values are in fact symbols, and we note the set of fresh variables by F, where $F \cap A = \emptyset$. In the sequel, we assume that this set F is given even if not specified, as well as the corresponding $IOSTS$ signature defined over it.

We introduce now the notion of *symbolic traces*. Symbolic traces are used in the symbolic execution as a way to store information about the exchanged communication actions, as well as the δ situations.

Definition 6.3.1 (Symbolic traces) *Let $\Sigma_F = (F, C)$ be a signature.*

The set of symbolic traces *over F denoted, by abuse, $Traces(\Sigma_F)$, is the set of all finite sequences over $Act(\Sigma_F) \cup \{\delta\}$*

Symbolic traces can be interpreted as *concrete* ones, denoting exchanges of concrete values by introducing, for each symbol, a concrete data.

Definition 6.3.2 (Concrete traces) *Let $Traces(\Sigma_F)$ be a set of symbolic traces.*

Given an interpretation $\nu : F \to M$ and a symbolic trace $ts \in Traces(\Sigma_F)$, the interpretation of *ts, denoted by $\nu(ts)$, is the trace obtained by replacing in ts:*

- *any action of the form $c?z$ by $c?\nu(z)$*
- *any action of the form $c!t$ by $c!\nu(t)$*

Notation 6.3.2.1 *In the following, for any $\nu \models \pi$, $\nu(ts)$ is called an* interpreted trace*.*

Symbolic states are used to store pieces of information concerning the execution, namely the symbolic trace related to the current state of the symbolic execution, the reached state of the orchestrator's specification, the symbolic values assigned to the attribute variables, and the constraints on those symbolic values after the execution. Moreover, the constraints which are stored within the symbolic states are called *path conditions*, and can be seen as the accumulator of properties which the inputs must satisfy in order for an execution to follow the particular associated path.

Definition 6.3.3 (Symbolic states) *Let $\mathbb{G}$ be an $IOSTS$ over Σ.*

A symbolic state *η is a quadruple (ts, q, σ, π) where $ts \in Traces(\Sigma_F)$, $q \in Q$, $\sigma \in T_\Omega(F)^A$, and $\pi \in Sen_\Omega(F)$.*

We note $\mathcal{S}$ for the set of symbolic states over $\mathbb{G}$ and F

Notation 6.3.3.1 *In the following, for any $\eta \in \mathcal{S}$ of the form (ts, q, σ, π), we use the notations $ts(\eta)$, $state(\eta)$, $sub(\eta)$, and $pc(\eta)$ in order to refer, respectively, to ts, q, σ, and π.*

When generating the symbolic execution tree, it may happen that some symbolic states are not reachable. That is, the path condition of the symbolic state is not satisfiable. Thus, we define the set of *satisfiable symbolic states*.

Definition 6.3.4 (Satisfiable symbolic states) *Let $\mathbb{G}$ be an $IOSTS$ over Σ, and let $\mathcal{S}$ be the set of all symbolic states over $\mathbb{G}$ and F.*

$\mathcal{S}_{sat}$ is the set of symbolic states of the form (ts, q, σ, π) for which there exists an interpretation $\nu : F \to \mathcal{M}$ such that $\nu \models \pi$

Example 6.3.1 *Let us consider the symbolic state η of the form (ts, q, σ, π), such that $\pi = (a > b) \wedge (b > c) \wedge (c > a)$. As π is clearly unsatisfiable if interpreted over the usual integers, we discard this type of symbolic states.*

The symbolic execution of an $IOSTS$ results from the symbolic execution of its transitions. Symbolic executions of transitions are denoted as triples (η, sa, η'), where η is the symbolic state from which the transition is executed, sa is the symbolic interpretation of the communication action introduced in the transition, and η' is the symbolic state reached by the execution.

Definition 6.3.5 (Symbolic execution of a transition) *Let $\mathbb{G} = (Q, q_0, Tr)$ be an $IOSTS$ over $\Sigma = (A, C)$.*

For any $tr \in Tr$ and $\eta \in \mathcal{S}$, such that $source(tr) = state(\eta)$, a symbolic execution *of tr from η is a triple $(\eta, sa, \eta') \in \mathcal{S} \times Act(\Sigma_F) \times \mathcal{S}$ such that:*

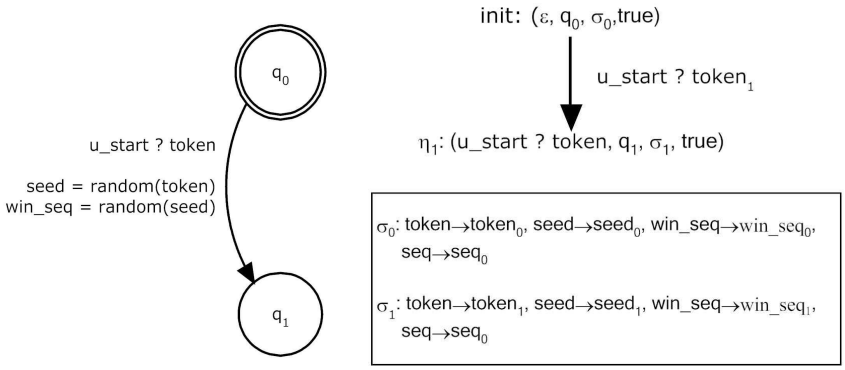
- *if $act = \tau$, then $sa = \tau$, and*
  $\eta' = (ts(\eta), target(tr), sub(\eta) \circ sub(tr), \pi(\eta) \wedge sub(\eta)(guard(tr)))$
- *if $act = c!t$, there exists a fresh variable z of F, not occurring in η such that $sa = c!z$, and*
  $\eta' = (ts(\eta).sa, target(tr), sub(\eta) \circ sub(tr),$
  $\pi(\eta) \wedge sub(\eta)(guard(tr)) \wedge z = sub(\eta)(t))$
- *if $act = c?x$, there exists a fresh variable z of F, not occurring in η such that[a] $sa = c?z$ and*
  $\eta' = (ts(\eta).sa, target(tr), sub(\eta)_{[x \mapsto z]} \circ sub(tr), \pi(\eta) \wedge sub(\eta)(guard(tr)))$

[a] $\sigma_{[x \mapsto z]}$ is the application σ' verifying $\sigma'(x) = z$ and $\forall y \neq x$, $\sigma'(y) = \sigma(y)$.

Note that, according to the usual definition of symbolic execution of $IOSTS$ ([Gaston 2006]), the symbolic action corresponding to a communication action of the form $c!t$ is directly $c!sub(\eta)(t))$. In our approach, however, we chose to use a

new symbolic variable each time that a value is emitted ($c!z$, with $z = sub(\eta)(t)$)). This modification does not affect the semantics of the symbolic execution but has the inconvinient of introducing more fresh variables. However, it proves useful when eliciting behaviors from the symbolic execution (Chapter 8): outputs are transformed into inputs and vice-versa.

Notation 6.3.5.1 *In the following, str denotes a triple* (η, sa, η') *and we use the notations* $source(str)$, $act(str)$, *and* $target(str)$ *in order to refer to, respectively,* η, sa, *and* η'.



(a) One transition of an *IOSTS*. (b) Symbolic execution of the transition.

Figure 6.5: Symbolic execution of one transition example.

Example 6.3.2 *Let us consider Figure 6.5. Figure 6.5(a) depicts a transition of an IOSTS (the IOSTS of the Slot Machine example of Figure 6.2). Figure 6.5(b) depicts its symbolic execution. Note that there are two symbolic states (Definition 6.3.3)* $init$ *and* η_1.

$init$ *contains the information* before *executing the transition: the symbolic trace* $ts(init)$ *is* ε *since it is the initial state, its associated state* $state(init)$ *of the IOSTS is* q_0, *the affectation* $sub(init) = \sigma_0$ *is the assignation of a value to each attribute variable, and the path condition* $pc(init)$ *is* $true$ *for the same reason that it is the initial state (there is no condition needed to be satisfied in order to reach this state).*

The symbolic transition is labeled with the communication action of tr, *but substituting the variables with their corresponding symbols. In this case, the communication action* $act(tr) = u\_start?token_1$, *representing the fact that we are receiving a new value (symbol) to be stored on the variable* $token$. *The symbolic target state* $target(tr)$ *is* η_1, *which stores the same information as* $init$ *except for the fact that now the symbolic trace is now* $st(\eta_1) = u\_start?token_1$ *and its affectation* $sub(\eta_2) = \sigma_1$ *takes into account the fact that the value of the variable* $token$ *is no longer the symbol* $token_0$ *but* $token_1$. *The path condition* $pc(\eta_1)$ *is the same as* $pc(init)$ *because* $guard(tr) = true$.

Intuitively, as for the case of programs, the symbolic execution of an $IOSTS$ can be seen as a tree whose edges are symbolic states and vertexes are labeled by symbolic communication actions. The root is a symbolic state made of the $IOSTS$'s initial state, the empty symbolic trace, the path condition $true$ (there is no constraint to begin the execution), and of an arbitrary initialization σ_0 of variables of A associating to each attribute variable a new and different fresh variable of F. Vertexes are computed by choosing a source symbolic state η already computed and by symbolically executing a transition of the $IOSTS$ whose source is the state introduced in η. The symbolic communication action is computed from the transition's communication action and from the symbolic values associated to attribute variables in η. A target symbolic state is then computed: it stores the target state of the transition, the complete symbolic trace that has been executed in order to reach the state (and which is computed by using the symbolic trace of the source state and appending the new trace), a new path condition derived from the path condition of η and from the transition guard of the transition, and the new symbolic values associated to attribute variables.

Definition 6.3.6 (Symbolic execution of an $IOSTS$) *Let $\mathbb{G} = (Q, q_0, Tr)$ be an $IOSTS$ over $\Sigma = (A, C)$.*

The symbolic execution*, denoted $SE(\mathbb{G})$, of $\mathbb{G}$ is the restriction $(init, \mathcal{R}_{sat})$ to $\mathcal{S}_{sat}$ of the couple $(init, \mathcal{R})$ where:*

- *$init = (\varepsilon, q_0, \sigma_0, true)$ such that $\forall x \in A$, $\sigma_0(x) \in F$ and σ_0 is injective*
- *$\mathcal{R} \subseteq \mathcal{S} \times Act(\Sigma_F) \times \mathcal{S}$ is such that for all $\eta \in \mathcal{S}$, and $tr \in Tr$ with $source(tr) = state(\eta)$, there exists exactly one symbolic execution of tr from η in $\mathcal{R}$. Moreover, for $(\eta_1, c\triangle z, \eta_1')$ and $(\eta_2, d\triangle w, \eta_2')$ in $\mathcal{R}$ with $\triangle \in \{!, ?\}$, we have $z \neq w$*

Remark 4 *The symbolic execution is unique, up to the choice of the involved fresh variables.*

Example 6.3.3 *Figure 6.6 depicts the symbolic execution for the $IOLTS$ of Figure 6.2. For readability reasons, affectations and symbolic traces are not directly shown inside the symbolic states but in separate frames. The symbolic execution is shown until symbolic states η_4 and η_5, where the initial state of $\mathbb{G}$ is re-visited for the second time.*

In this case, there is only one decision that had to be made, and is the one regarding the value of the variable seq. If its value (seq_1) is equal to the value of the variable win_seq (win_seq_1), then the right branch is taken. The left branch represent the state of the system where $seq \neq win\_seq_1$. As illustrated by the figure, symbolic states are structures which accumulate important information that will prove useful in the testing activity.

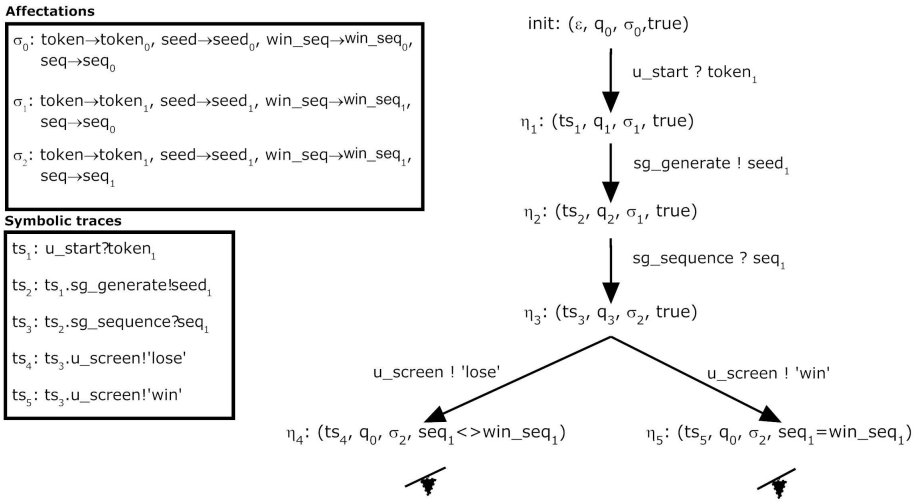


Figure 6.6: $SE(\mathbb{G})$ based on the $IOSTS$ of Figure 6.2.

Note that the constants 'lose' and 'win' are not symbols, and we would have had to introduce one symbolic variable for each of them in order to use them in the example of the symbolic execution, as well as a condition in oder to keep the value of the variable constant. For the sake of readability, we abuse and use directly the constants. This is also done in the rest of the related examples.

Paths of a symbolic execution are the sequences of symbolic transitions that start at the initial state.

Definition 6.3.7 (Paths of $SE(\mathbb{G})$) *Let $SE(\mathbb{G}) = (init, \mathcal{R}_{sat})$ be a symbolic execution.*

The set of paths *of $SE(\mathbb{G})$, denoted by abuse $Path(SE(\mathbb{G}))$, contains all the finite sequences $str_1 \cdots str_n$ of transitions of $\mathcal{R}_{sat}$, such that:*

- $source(str_1) = init$
- *for every i, $1 \leq i \leq n$, $target(str_i) = source(str_{i+1})$*

Notation 6.3.7.1 *In the following, if (ts, q, σ, π) occurs as a state of $SE(\mathbb{G})$, then we say that (ts, q, σ, π) belongs to $SE(\mathbb{G})$, and we note $(ts, q, \sigma, \pi) \in SE(\mathbb{G})$.*

Example 6.3.4 *Let us consider the symbolic execution of Figure 6.6. A finite path of that tree is the sequence of symbolic transitions going from the root state $init$ until the symbolic state η_5, that is, the sequence:*

$$(init, u\_start?token_1, \eta_1).(\eta_1, sg\_generate!seed_1, \eta_2).(\eta_2, sg\_sequence?seq_1, \eta_3).(\eta_3, u\_screen!'win', \eta_5).$$

We define the behaviors (also known as *semantics*) of a symbolic execution by means of its symbolic traces.

Definition 6.3.8 (Semantics of a symbolic execution) *Let* $SE(\mathbb{G}) = (init, \mathcal{R}_{sat})$ *be a symbolic execution.*

The semantics of $SE(\mathbb{G})$ *is defined as:*

$$Traces(SE(\mathbb{G})) = \bigcup_{\substack{\eta=(ts,q,\sigma,\pi)\in SE(\mathbb{G}) \\ \nu\models\pi}} \nu(ts(\eta))$$

Remark 5 *Note that the traces that can be obtained from* $SE(\mathbb{G})$ *are, by construction, the traces that can be obtained from the IOSTS* $\mathbb{G}$*, since traces of an IOSTS are constructed by interpreting the attribute variables of the paths in* $Path(\mathbb{G})$*, and the concrete traces of* $SE(\mathbb{G})$ *are no more than the interpretation of the traces obtained from traversing (i.e., the symbolic execution) the IOSTS.*

6.3.2 Operations over the Symbolic Execution

The first operation is the quiescence enrichment of the symbolic execution. The idea behind this enrichment is the same as for *IOLTS* introduced in Section 3.2.1 (Definition 3.2.6), that is, the situations where the system can remain silent are identified and made observable in the symbolic execution. A system can remain silent only if it can not evolve by itself, i.e., if from a given state in the symbolic execution there are only input transitions. Strictly speaking, if there are any transitions that can be fired and whose action is not an input, then the quiescence situation is also allowed only if the negation of the disjunction of all path conditions of such transitions is satisfiable.

Definition 6.3.9 (Quiescence enrichment of a symbolic execution) *Let* $\mathbb{G}$ *be an IOSTS, and* $SE(\mathbb{G}) = (init, \mathcal{R}_{sat})$ *be its symbolic execution.*
If, for any $\eta \in \mathcal{S}_{sat}$*, we note:*

- $react(\eta) = \{str \mid str \in \mathcal{R}_{sat}, str = (\eta, act, \eta'), act \notin I(\Sigma_F)\}$
- $f_\eta = \bigwedge_{str\in react(\eta)} \neg pc(target(str))$ *if* $react(\eta) \neq \emptyset$ *and* true *otherwise*

Then, the enrichment by quiescence *of* $SE(\mathbb{G})$ *is* $SE(\mathbb{G})_\delta = (init, \mathcal{R}_\delta)$*, where* $\mathcal{R}_\delta$ *is the least relation* $(\mathcal{R}_{sat} \cup \mathcal{R}_q)$ *defined by:*

- $\mathcal{R}_q \subset \mathcal{S}_{sat} \times \{\delta\} \times \mathcal{S}_{sat}$ *is such that for all* η *in* $\mathcal{S}_{sat}$*, we have:*

$$(\eta, \delta, \eta') \in \mathcal{R}_q$$
$$\text{with } \eta' = (ts(\eta).\delta, state(\eta), sub(\eta), \pi(\eta) \wedge f_\eta)$$

According to Definition 6.3.9, f_η is the auxiliary formula used to define if quiescence is allowed or not: if there are only input communication actions for the

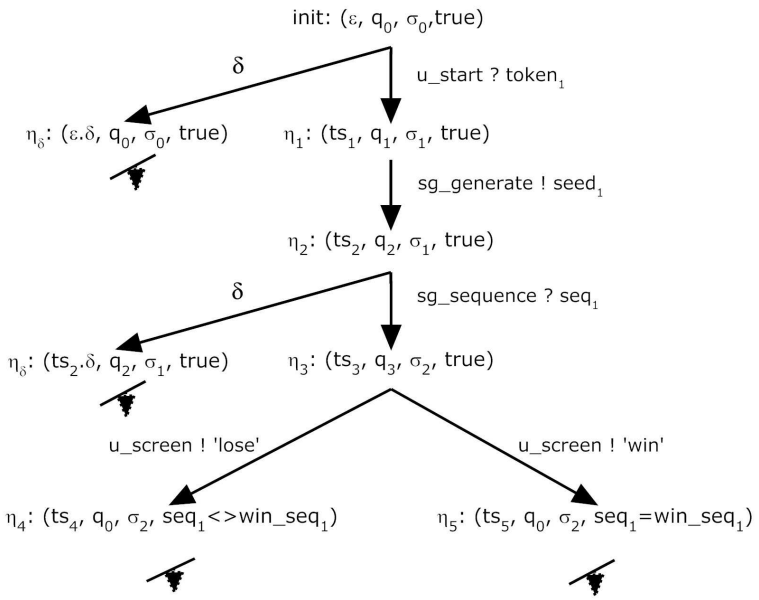


Figure 6.7: $SE(\mathbb{G})_\delta$ based on the symbolic execution of Figure 6.6.

transitions, then f_η is *true* (this means that if the system cannot evolve because it is waiting for some intervention from the environment, it is allowed to be quiescent), otherwise, it is the conjunction of the negation of all the path conditions of the target states of the transitions which do not have an input as their communication action (if this condition is satisfiable, it means that it may happen that the system cannot evolve by itself). Thus, quiescence is allowed only if f_η, together with the path condition of the source state of the transition being examined ($\pi(\eta)$), is satisfiable.

Example 6.3.5 *Figure 6.7 denotes the quiescence enrichment of the IOSTS $\mathbb{G}$ of Figure 6.6. In this case, the quiescence situation is allowed only in states init and η_2: in state init, the slot machine's interface is waiting for the token from the user, and in state η_2 it is waiting for the value of seq from the Sequence Generator service; in both cases, it cannot evolve if it does not receive a message from them. For the rest of symbolic states, if we apply the Definition 6.3.9, the respective conditions are not satisfiable, meaning that in those states the system is supposed to react. Finally, for readability reasons, we cut the tree as in Example 6.3.3 and also for the newly added quiescent states.*

The other operation we perform is the one called τ-reduction. It consists on *removing* the internal actions (τ) from the symbolic execution. This operation will prove useful for writing the algorithm of test case generation in the next chapter. Indeed, the τ-reduction operation simplifies the tree-like structure while preserving traces.

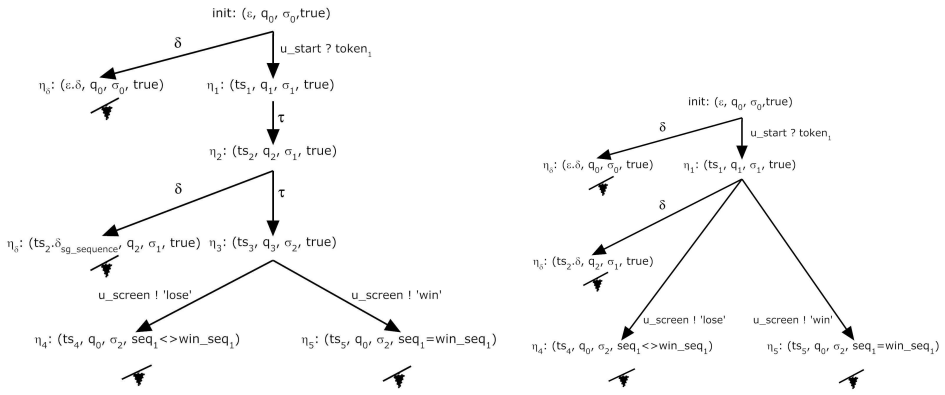
Definition 6.3.10 (τ-reduction of a symbolic execution) *Let $SE(\mathbb{G}) = (init, \mathcal{R}_{sat})$ be a symbolic execution.*

The τ-reduction of $SE(\mathbb{G})_\delta$ *is* $SE(\mathbb{G})_\tau = (init, \mathcal{R}_\tau)$*, where* $\mathcal{R}_\tau$ *is the least relation such that for any sequence* $(\eta_1, sa_1, \eta_2) \cdots (\eta_n, sa_n, \eta_{n+1})$ *of transitions in* $\mathcal{R}_\delta$ *it satisfies:*

- *there exist no elements of the form* (η, τ, η_1) *in* $\mathcal{R}_\delta$
- $sa_n \neq \tau$ *and* $\forall 1 \leq i < n$, $sa_i = \tau$

then $(\eta_1, sa_n, \eta_{n+1}) \in \mathcal{R}_\tau$

Note that even if some transitions are removed from $SE(\mathbb{G})_\delta$ after applying Definition 6.3.10, the symbolic execution still has relevant information about them: the symbolic traces and the path conditions are not affected by the τ-reduction.



(a) $SE(\mathbb{G}_1)_\delta$ with τ-transitions. (b) τ-reduction (Def. 6.3.10).

Figure 6.8: τ-reduction example.

Example 6.3.6 *If we apply the* τ*-reduction to Figure 6.6, it remains the same, since there are no* τ *transitions in* $SE(\mathbb{G})_\delta$*. Thus, we consider the symbolic execution* $SE(\mathbb{G}_1)_\delta$ *of Figure 6.8(a), where we replace some communication actions of the transitions by* τ*. Figure 6.8(b) depicts the* τ*-reduction of* $SE(\mathbb{G}_1)_\delta$*.*

6.3.3 Stop Criteria

Symbolic executions are infinite structures but, in practice, we cannot work with infinite structures, so we need to define ways to stop the generation of the symbolic execution tree structure. This notion is known as *stop criteria*. Besides the stop criteria called *restriction by inclusion* given in [Gaston 2006] and that can be perfectly applied to our symbolic execution, we introduce a more simple criterion called *root re-visited or depth reached*. Thus, we have:

1. **Restriction by inclusion criterion**, which is not detailed here (for details, refer to [Gaston 2006]).

2. **Root re-visited or depth reached criterion**, whose idea is to stop the symbolic execution when the initial state of the respective $IOSTS$ is re-visited or when a certain depth has been reached.

6.4 Conclusion

In this chapter we have introduced the $IOSTS$s structures, which are the basis of the work presented in the rest of this thesis. $IOSTS$s are symbolic characterizations of the $IOLTS$s presented in Part I, and introducing symbolic data to characterize internal states, to express firing conditions of transitions and to denote messages exchanged through channels. In fact, we have shown that there is a unique $IOLTS$ canonically associated to each $IOSTS$. $IOSTS$s are used to model specifications of orchestrators, and then are symbolically executed in order to generate a tree-like structure that represents all the valid behaviors that any implementation of the specification should offer. $IOSTS$s are executed by using the so-called symbolic execution technique, whose main idea is to execute programs or specifications by using symbols instead of concrete data. This symbolic execution tree structure will serve as the basis when testing the conformance of orchestrations with respect to their specifications. We have also shown the operations that we will apply on those symbolic execution structures and that will prove useful in the following chapters.

With the introduction of the symbolic technique, we are ready to show how to apply the work presented in Part I to the $IOSTS$s and their symbolic executions in order to test orchestrators in context, and to elicit behaviors to test the compatibility of Web services. In fact, the next two chapters can be seen as the symbolic counterpart of Chapters 4 and 5.

Chapter 7

Algorithm for Testing Orchestrators in Context

Contents

7.1 Introduction

With the assumption that the specification of the orchestrator $\mathbb{O}rch$ is available, we propose a symbolic approach to test the orchestrator by taking into account its different situations or contexts of usage. Those contexts may vary from the case where all the Web services are simulated or controlled by the tester, to the case where the tester has no access at all to any of the Web services involved in the orchestration.

Thus, we show how to adapt the specification of the orchestrator by taking the classification of the communication channels into account. The specification is given in the form of the symbolic execution of $\mathbb{O}rch$ and represents all the possible and valid behaviors of the system. We modify the symbolic execution according to the status of the communication channels by means of different transformations of the symbolic execution tree: the quiescence enrichment, remote input/output transformation and hiding operations (already presented for $IOLTS$ in Chapters 3 and 4, and here they are adapted for the $IOSTS$s). After applying these operations

and removing form the symbolic tree the internal communications (τ-reduction) the resulting structure is the partial specification of the orchestrator in context. Once we have this structure, we proceed to define the system under test in context: $Obs(Orch[Rem], C_o)$, like introduced in Section 4.3, as well as the behaviors that we want to test in it. These behaviors are known as *test purposes*, and correspond to finite sub-trees of the partial specification, each path of such a sub-tree characterizing a class of executions to be covered by some generated test cases.

Then, we define a rule-based algorithm which is based on the one of [Gaston 2006]. The goal of the algorithm is to check if a partial observation of an orchestrator in context conforms to its partial specification. Our algorithm is defined in the context of the $\overline{\textbf{ioco}}$ conformance relation but, as discussed in this chapter, can be easily extended to the $\overline{\textbf{uioco}}$ one (both of those relations already introduced in Section 4.4). The algorithm is defined in the form of a set of rules, which follows the intuition behind the **ioco** conformance relation, that is, the IUT implementing the SUT is stimulated and for each observation we verify if such observation is accepted by the specification or not and, furthermore, if the SUT remains in the path of behavior that the tester wants to verify.

We begin this chapter with Section 7.2 by introducing the classification of the communication channels according to their status. Then, taking that classification into account, we show how to modify the symbolic execution in order to generate the partial specification of the orchestrator in context. We finish the section by introducing the notion of system under test in context for the case of orchestrations. We continue en Section 7.3 by showing how to extract test purposes from the partial specification. In Section 7.4 we present our rule-based algorithm, which is introduced first for the controllable an observable cases ($\overline{\textbf{ioco}}$) and then for the hidden case ($\overline{\textbf{uioco}}$). Section 7.5 presents the conclusion of the chapter.

7.2 Orchestrators in Context

In this section we show how to transform the specifications in the form of symbolic executions of an $ISOST$, $SE(\mathbb{O}rch)$, into partial specifications of orchestrators in context, $Obs(\mathbb{O}rch)$. These transformations are basically done according to the status of the communication channels used to interact with the Web services. Thus, we transform the symbolic execution of the orchestrator's specification so that the traces associated to the resulting structure will correspond to the traces of the underlying $IOLTS$ modified by the operations described in Chapter 4: enrichment by quiescence, remote input/output transformation, and hiding operator; and with the introduction for the symbolic case of the τ-reduction operation (Definition 6.3.10). Let us briefly recall that the quiescence enrichment aims at identifying the situations where the orchestrator can remain silent; the remote input/output transformation aims at transforming the inputs sent from the remote Web services to the orchestrator into observations, when such inputs are sent over observable channels from the point of view of the tester; the hiding operator makes unobservable the communi-

cation actions going through hidden channels (this is achieved by considering such actions as internal ones); and the τ-reduction operation removes the τ transitions from the symbolic execution tree while preserving important information. We naturally apply these operations consecutively at the symbolic level by starting from the symbolic execution tree $SE(\mathbb{O}rch)$. Finally, we also show how we assume that the orchestrator in context under test can be modeled.

7.2.1 Web services Status and Communication Channels

As already introduced in Section 4.2, in this section we briefly recall the classification of the communication channels according to their status. We distinguish the cases where the internal interactions between the Web services and the orchestrator are controllable (Web services are simulated), hidden (no access by the test architecture) or observable (collected by the test harness). Figure 7.1 depicts this classification in the context of test architectures. In the figure, **CP** stands for Control Point, and represents the interface of the test architecture used to send and receive information through the controllable communication channels. **OP** stands for Observation Point, and represents the interface of the test architecture used to observe the information going through the observable communication channels. Thus, a channel is said to be:

- **Controllable** if the tester may send inputs and observe outputs on it.
- **Hidden** if the tester cannot access the communication channel.
- **Observable** if the tester can access the communication channel so that communications can be observed.

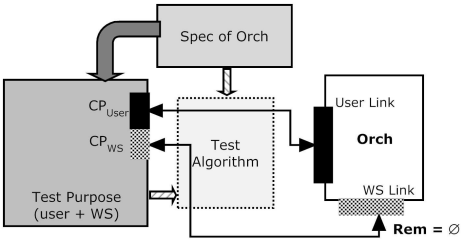
Following the same intuitions introduced in Section 4.2, we partition the set of communication channels C introduced in Definition 6.2.1 (Section 6.2) into three sub-sets:

(1.) C_c for the controllable channels,

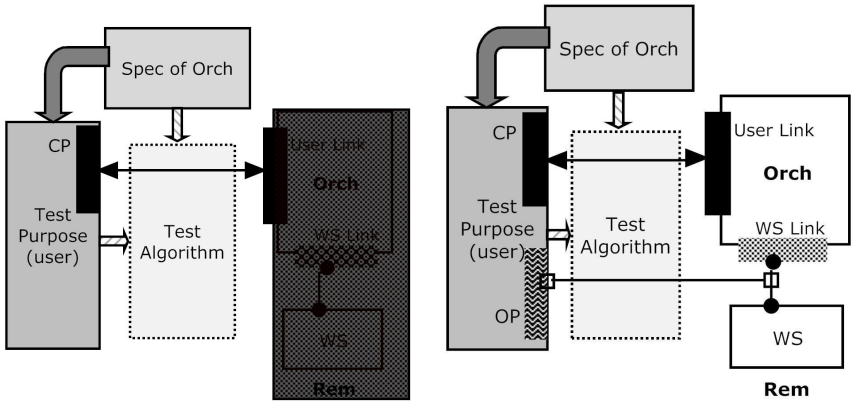
(2.) C_h for the hidden channels, and

(3.) C_o for the observable channels.

We also use the notations $Orch$ to refer to the orchestrator implementing the specification $\mathbb{O}rch$, and Rem to refer to the remote part of the system corresponding to the Web services interacting with $Orch$. Besides, the set of communication channels used to interact with Rem is denoted C_r. In this way, where there are hidden or observable communication channels in C_r, and the user no longer interacts only with $Orch$, but with $Orch$ in the context of Rem, denoted $Orch[Rem]$, where there are some channels in C_r that can be accessed by the tester and some other ones that are completely hidden. In the same way, the set of communication channels used by $\mathbb{O}rch$ is also partitioned.



(a) Controllable.



(b) Hidden. (c) Observable.

Figure 7.1: Classification of the communications on channels according to their status.

7.2.2 Partial Specifications for Orchestrators in Context

According to the classification of channels presented in Subsection 7.2.1, we now show how to transform the specification of an orchestrator in the form of a symbolic execution tree, $SE(\mathbb{O}rch)$, into an specification of the orchestrator in context. Therefore, we introduce a series of modifications that are performed as a sequence of operations over symbolic execution structures. The resulting structure will then serve as the basis of our algorithm in order to test the conformance of the orchestrators in context.

We begin by modifying the enrichment by quiescence introduced in Definition 6.3.9, since now that we are taking into account the situation of the communication channels, we need to include the quiescence caused by the lack of response of the Web services. This corresponds to the internal quiescence for the $IOLTS$ of Definition 4.3.6, Section 4.3, adapted here to the symbolic level and with the difference that, since we are now working with the tree structures, once we reach a quiescence state due to a Web service, we *do not allow* any further inputs from it, since from the user's point of view it is in a blocked situation. If a Web service enters into a quiescence situation, then it needs external intervention in order to

react. Thus, we define the operation *enrichment by full quiescence*, which in turn is composed of the *external quiescence* and that corresponds to the one that indicates that the orchestrator can go into a quiescence situation due to the fact that it cannot evolve by itself, without intervention from the exterior, and of the *internal quiescence*, which is the quiescence situation provoked by the lack of synchronization or response from the Web services.

Definition 7.2.1 (Enrichment by full quiescence) *Let* $SE(\mathbb{O}rch) = (init, \mathcal{R}_{sat})$ *be a symbolic execution.*
If, for any $\eta \in \mathcal{S}_{sat}$*, we note:*

- $react(\eta) = \{str \mid str \in \mathcal{R}_{sat}, str = (\eta, act, \eta'), act \notin I(\Sigma_F)\}$
- $f_\eta = \bigwedge_{str \in react(\eta)} \neg pc(target(str))$ *if* $react(\eta) \neq \emptyset$ *and* true *otherwise*
- *for any* $c \in C_h \cup C_o$, $In(c, \eta) = \{str \mid str \in \mathcal{R}_{sat}, str = (\eta, c?x, \eta')\}$
- $f'_{\eta,c} = \bigvee_{str \in In(c,\eta)} pc(target(str))$ *if* $In(c, \eta) \neq \emptyset$ *and* false *otherwise*

Then, the enrichment by full quiescence *of* $SE(\mathbb{O}rch) = (init, \mathcal{R}_{sat})$ *is the denoted, by abuse, as the couple* $SE(\mathbb{O}rch)_\delta = (init, \mathcal{R}_\delta)$ *where* $\mathcal{R}_\delta$ *is the least relation* $(\mathcal{R}_{sat} \cup \mathcal{R}_{pq} \cup \mathcal{R}_{ls}) \setminus \mathcal{R}_{junk}$ *defined by:*

External quiescence: $\mathcal{R}_{pq} \subset \mathcal{S}_{sat} \times \{\delta\} \times \mathcal{S}_{sat}$ *is such that for all* η *in* $\mathcal{S}_{sat}$*,*
we have $(\eta, \delta, \eta') \in \mathcal{R}_{pq}$
with $\eta' = (ts(\eta).\delta, state(\eta), sub(\eta), \pi(\eta) \wedge f_\eta \wedge \bigwedge_{c \in C_h \cup C_o} \neg f'_{\eta,c})$

Internal quiescence: $\mathcal{R}_{ls} \subset \mathcal{S}_{sat} \times \{\delta\} \times \mathcal{S}_{sat}$ *is such that for all* η *in* $\mathcal{S}_{sat}$*,*
we have $(\eta, \delta, \eta') \in \mathcal{R}_{ls}$
with $\eta' = (ts(\eta).\delta_c, state(\eta), sub(\eta), \pi(\eta) \wedge f_\eta \wedge f'_{\eta,c})$*,*
and in this case, for all $c \in C_h \cup C_o$*, for any* $str' \in In(c, source(str))$ *with* $str \in In(c, \eta)$ *then* $str' \in \mathcal{R}_{junk}$

According to Definition 7.2.1, f_η is the auxiliary formula used to determine if quiescence due to the impossibility of the orchestrator to move forward is allowed or not: if there are only input communication actions for the transitions, then f_η is *true*, otherwise, it is the conjunction of the negation of all the path conditions of the target states of the transitions which do not have an input as their communication action.

$f'_{\eta,c}$ is the auxiliary formula used to determine if quiescence due to the Web services is allowed or not: if the orchestrator is waiting for some inputs of any remote Web service (through communication channels in C_h or in C_o), then it is allowed to remain quiescent, and $f'_{\eta,c}$ is the disjunction of all the path conditions of the target states of the transitions which have an input (from a remote Web service) as their communication action. If there is none input expected from any

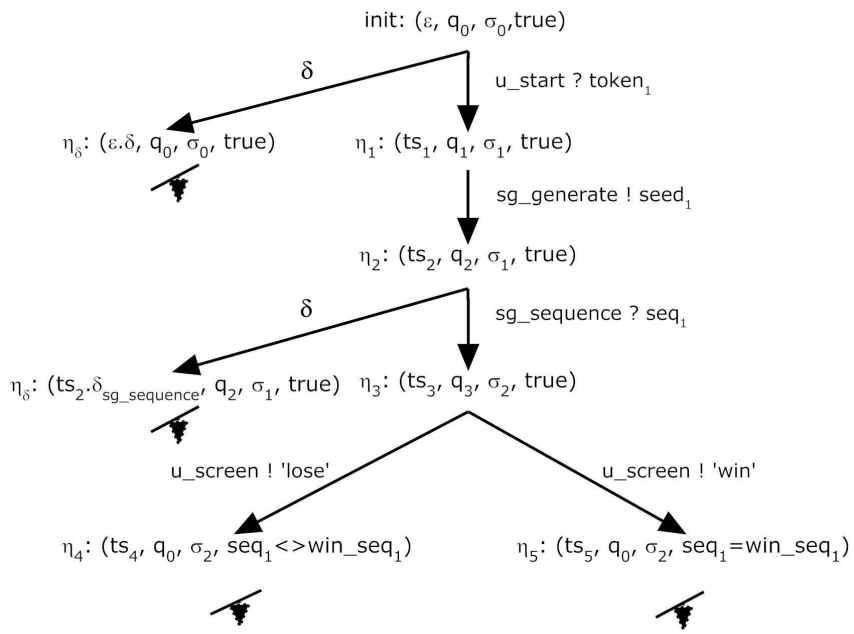


Figure 7.2: Full quiescence $SE(\mathbb{O}rch)_\delta$ for the symbolic execution $SE(\mathbb{O}rch)$ of the Slot Machine example (Figure 6.6).

Web service, $f'_{\eta,c}$ is $false$ (the orchestrator cannot be quiescent due to the lack of reaction of the Web services).

Thus, internal quiescence is allowed only if the conjunction of f_η with the path condition of the source state of the transition being examined ($\pi(\eta)$), and with the negation of $f'_{\eta,c}$, is satisfiable. External quiescence is allowed only if the conjunction of f_η with the path condition of the source state of the transition being examined ($\pi(\eta)$), and with $f'_{\eta,c}$, is satisfiable.

Finally, note that, for the internal quiescence, no more receptions are allowed on the channel c if a quiescence situation has already been stated on that channel.

Example 7.2.1 *Figure 7.2 depicts the full quiescence enrichment for the symbolic execution $SE(\mathbb{O}rch)$ of the Slot Machine example, that is, of the symbolic execution of Figure 6.6. Note that in state η_3, the orchestrator is waiting for a response from the Sequence Generator service SG, and it may happen that SG never answers. Thus, the orchestrator is allowed to enter into a quiescence state. Finally, the resulting structure is the same as the one depicted in Figure 6.7 in Section 6.3 for the* usual *quiescence, however, here the quiescence situation added in the node η_3 is due to the* internal *quiescence, and thus, it does not represent the same situation. As it is the case of Figure 6.6, the resulting tree-like structure was cut for readability reasons.*

As a reminder, we should have had to introduce symbolic variables for the con-

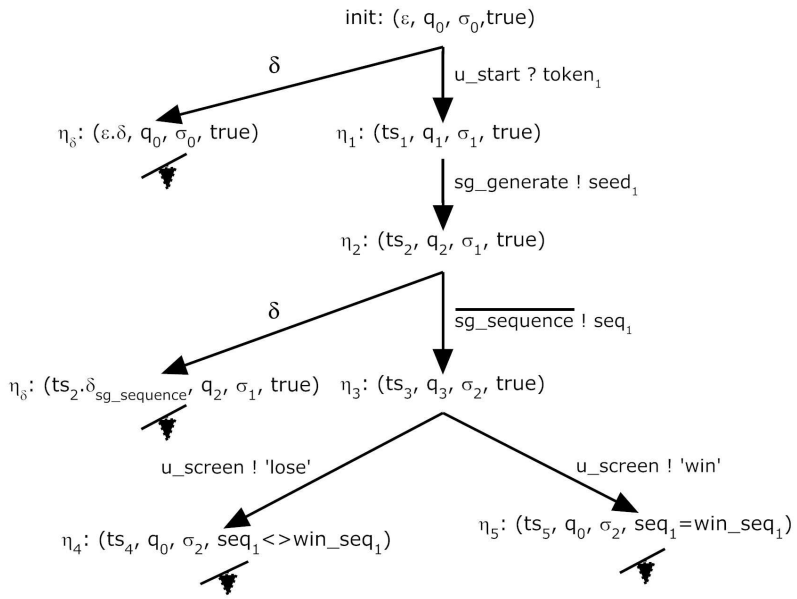


Figure 7.3: WS input transformation $IT(\mathbb{O}rch)$ for $SE(\mathbb{O}rch)_\delta$ of Figure 7.2.

stants 'lose' and 'win', but we did not do it for the sake of readability.

The next operation consists on transforming the inputs from remote Web services into *observations*. This is the symbolic counterpart of Definition 4.3.9, and the idea behind it is that inputs from remote Web services should not be treated as inputs from the user, since they are not controlled by the user. In fact, from the point of view of the tester, those inputs are actually observations.

Definition 7.2.2 (WS input transformation) *Let $SE(\mathbb{O}rch) = (init, \mathcal{R}_{sat})$ be a symbolic execution.*

The WS input transformation *of $SE(\mathbb{O}rch)_\delta$ is defined as $IT(\mathbb{O}rch) = (init, \mathcal{R}_{IT})$, where $\mathcal{R}_{IT}$ is defined as follows:*

for all (η, sa, η') in $\mathcal{R}_\delta$, if sa is of the form $c?z$ with $c \in C_o$, then $(\eta, \overline{c}!z, (ts(\eta).\overline{c}!z, state(\eta'), sub(\eta'), \pi(\eta')))$ is in $\mathcal{R}_{IT}$, else (η, sa, η') is in $\mathcal{R}_{IT}$

Example 7.2.2 *Figure 7.3 depicts the WS input transformation applied to the IOSTS $\mathbb{O}rch$ of Figure 7.2. Inputs from the Sequence Generator service are transformed into observations, since it is like that that the tester perceives them (with the assumption the communication channels are observable). Moreover, we note those observations with an overline over the communication channel in order to know that they actually come from the Web services and not from the user.*

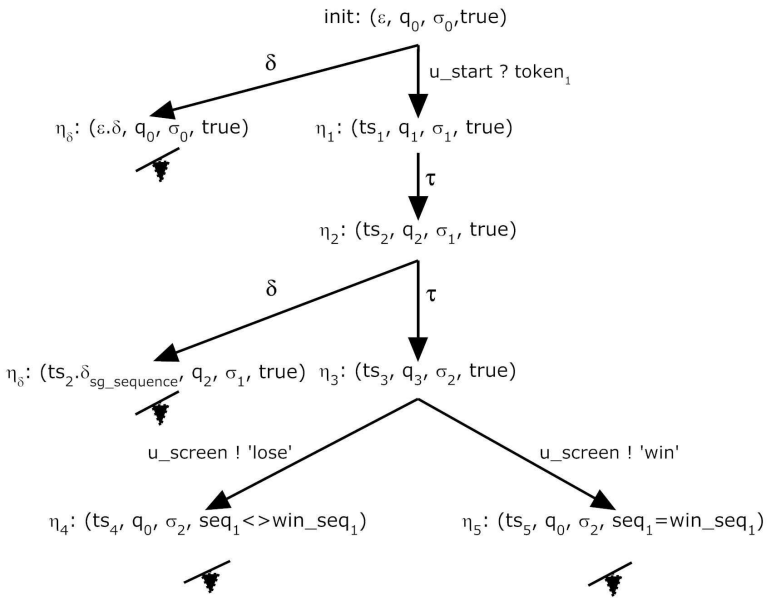


Figure 7.4: Hiding operator $HO(\mathbb{O}rch)$ for $SE(\mathbb{O}rch)_\delta$ of Figure 7.2.

In order to take the hidden channels into account, we first define the symbolic version of the hide operation introduced in Definition 3.2.13, Section 3.2. This operation transforms the communications actions through the hidden channels into internal ones, since from the point of view of the tester, they are non distinguishable one from the other (for helping in that case, we keep the traces).

Definition 7.2.3 (Hiding operator) *Let $SE(\mathbb{O}rch) = (init, \mathcal{R}_{sat})$ be a symbolic execution.*

The hiding operator of $SE(\mathbb{O}rch)$ *is defined as $HO(\mathbb{O}rch) = (init, \mathcal{R}_{HO})$, where $\mathcal{R}_{HO}$ is defined as follows:*

for all (η, sa, η') in $\mathcal{R}_{IT}$, if sa is of the form $c\Delta t$ with $c \in C_h$ and $\Delta \in \{!, ?\}$, then $(\eta, \tau, (ts(\eta), state(\eta'), sub(\eta'), \pi(\eta'))$ is in $\mathcal{R}_{HO}$, else (η, sa, η') is in $\mathcal{R}_{HO}$

Example 7.2.3 *Figure 7.4 depicts the hiding operator applied to the symbolic execution of Figure 7.2, with the assumption that the communication channels used to interact with the Sequence Generator service SG are hidden. Thus, first the WS input transformation (Definition 7.2.2) is preformed but, since there are no observable communication channels, the structure remains the same. Then, the hiding operator is performed (Definition 7.2.3) and the resulting structure is the one depicted in Figure 7.4.*

From the perspective of the tester, in this case SG becomes indistinguishable from the orchestrator under test since the communication actions going through such channels are no longer observable to the tester but perceived as internal actions.

The partial specification of the orchestrator in context, that we call *observable behaviors* of $SE(\mathbb{O}rch)$, is nothing more than the result of applying successively the quiescence enrichment, the WS input transformation operation, the hiding operation and the τ-reduction (Definition 6.3.10, Section 6.3) on the symbolic execution of the $IOSTS$ of the orchestrator's specification $\mathbb{O}rch$.

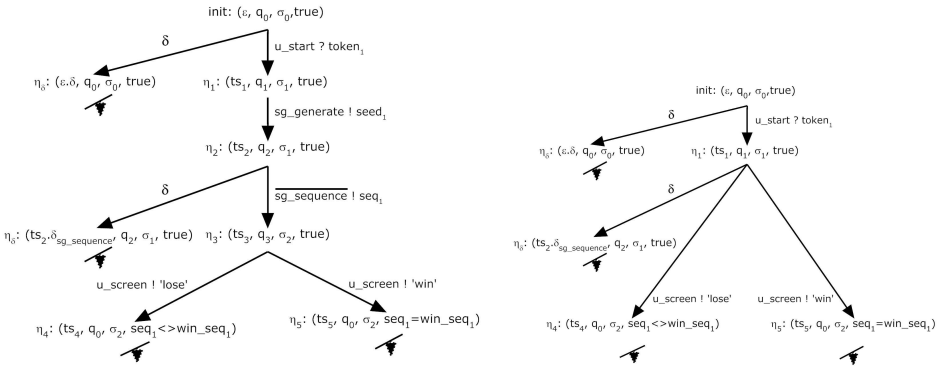
Definition 7.2.4 (Observable Behaviors of $SE(\mathbb{O}rch)$) *Let $SE(\mathbb{O}rch) = (init, \mathcal{R}_{sat})$ be a symbolic execution.*

The observable behaviors of $SE(\mathbb{O}rch)$ *is defined as $Obs(\mathbb{O}rch) = (init, \mathcal{R}_{Obs})$, where $\mathcal{R}_{Obs}$ is the least relation such that for any sequence $(\eta_1, sa_1, \eta_2) \cdots (\eta_n, sa_n, \eta_{n+1})$ of transitions in $\mathcal{R}_{HO}$ it satisfies:*

- *there exist no elements of the form (η, τ, η_1) in $\mathcal{R}_{HO}$*
- *$sa_n \neq \tau$ and $\forall 1 \leq i < n$, $sa_i = \tau$*

then $(\eta_1, sa_n, \eta'_n) \in \mathcal{R}_{Obs}$.

We define the reachable states *of $Obs(\mathbb{O}rch)$, denoted $\mathcal{S}_{reach}$, as the set of symbolic states in $\mathcal{S}_{sat}$ such that, for a given $\eta \in \mathcal{S}_{sat}$, $\eta \in \mathcal{S}_{reach}$ if and only if there exists a sequence $(\eta_1, sa_1, \eta_2) \ldots (\eta_{n-1}, sa_{n-1}, \eta_n)$ with $\eta_1 = init$, $\eta_n = \eta$, and for all $i < n$, $(\eta_i, sa_i, \eta_{i+1}) \in \mathcal{R}_{Obs}$*



(a) Observable case. (b) Hidden case.

Figure 7.5: Observable behaviors of $SE(\mathbb{O}rch)$.

Example 7.2.4 *Figure 7.5 depicts the observable behaviors of the symbolic execution of Figure 6.6. Both trees are the result of applying the quiescence enrichment, WS input transformation, hiding and τ-reduction operations consecutively on*

$SE(\mathbb{O}rch)$. Figure 7.5(a) depicts the observable behaviors of $SE(\mathbb{O}rch)$ with the assumption that the communication channels used by the slot machine's interface to communicate with the Sequence Generator service are observable. In this case, since there are no τ-transitions, the hiding and τ-reduction operations do not affect the tree. Figure 7.5(b) depicts the observable behaviors of $SE(\mathbb{O}rch)$ with the assumption that the communication channels used by the slot machine's interface to communicate with the Sequence Generator service are hidden. Thus, from the point of view of the tester, the communication actions going through those channels are internal actions (τ), which are removed from the tree when applying the τ-reduction operation.

The *semantics* of $Obs(\mathbb{O}rch)$ is defined by the traces of its *reachable* symbolic states.

Definition 7.2.5 (Semantics of $Obs(\mathbb{O}rch)$) *Let $Obs(\mathbb{O}rch)$ be the observable behaviors of $SE(\mathbb{O}rch)$.*

The semantics *of $SE(\mathbb{O}rch)$, denoted, by abuse, $Traces(SE(\mathbb{O}rch))$, is defined as*

$Traces(SE(\mathbb{O}rch)) = \bigcup_{\eta \in \mathcal{S}_{reach}} ts(\eta)$

7.2.3 SUT in context

In order to model the system under test, we simply use the same notion of SUT presented in Subsection 4.3.1. That is, if the underlying $IOLTS$ of the orchestrator's specification $\mathbb{O}rch$ is built upon the signature $Act(A, C)$, then SUT is of the form $Obs(Orch[Rem], C_o)$ (Definition 4.3.5, Section 4.3), where $Orch$ is defined over A and C, and Rem is defined over A and $C_o \cup C_h$. Thus, $Obs(Orch[Rem], C_o)$ is the system which takes all the information of the status of the communication channels into account and represents the system under test in its context of usage.

7.3 Symbolic Test Purposes

The symbolic execution tree represents all the valid behaviors that an implementation of the orchestrator can perform, and the partial specification (observable behaviors) represents these behaviors taking into account the different status of the communication channels with the Web services. Then, if the tester wants to test if a given implementation of an orchestrator in context behaves as supposed to, it suffices to take the behaviors described by the partial specification and execute them on the implementation system under test. One could think of executing *every* behavior of $Obs(\mathbb{O}rch)$ in $Obs(Orch[Rem], C_o)$. However, this process would represent an infinite process due to the nature of the symbolic execution trees, and thus we select *some* important behaviors that $Obs(Orch[Rem], C_o)$ should respect. These behaviors are called *test purposes* and are usually selected by some expert of the

system, someone that knows what are the crucial behaviors of the system that need to be tested. In this section, we show how to technically define such test purposes. In our case, test purposes correspond to finite sub-trees of the partial specification, each path of such a sub-tree characterizing a class of executions to be covered by some generated test cases.

We begin by introducing the concept of a *target state*. A target state is the final observation of the desired behavior to test. The target states must be reached by a symbolic transition whose communication action is an output. That is, an observable event from the point of view of the tester.

Definition 7.3.1 (Target state) *Let* $Obs(\mathbb{O}rch) = (init, \mathcal{R}_{Obs})$ *be the observable behaviors of* $SE(\mathbb{O}rch)$.
A target state *is a state* $\eta' \in Obs(\mathbb{O}rch)$ *such that there exists* $(\eta, c!z, \eta') \in \mathcal{R}_{Obs}$

Example 7.3.1 *A target state can be, for instance,* η_5 *in* $Obs(\mathbb{O}rch)$ *of Figure 7.5 (in both cases). That is, we want to test the case where the sequence is a winner sequence and the user gets the prize.*

Now we define the notion of a *complete* desired behavior to test, i.e., a *test purpose*. A test purpose is nothing more than a sub-tree of the symbolic execution whose all paths end on a target state.

Definition 7.3.2 (Test purpose) *Let* $Obs(\mathbb{O}rch) = (init, \mathcal{R}_{Obs})$ *be the observable behaviors of* $SE(\mathbb{O}rch)$.

A test purpose, *denoted* $\mathcal{TP}$, *for* $\mathbb{O}rch$, *is any finite sub-tree* $(init, \mathcal{R}_{\mathcal{TP}})$ *of* $Obs(\mathbb{O}rch)$, *where* $\mathcal{R}_{\mathcal{TP}} \subseteq \mathcal{R}_{Obs}$, *and whose all paths end on a target state.*

Notation 7.3.2.1 *In the following, we use the notations* $\eta \in \mathcal{TP}$ *to say that* η *is the source or the target state of some symbolic transition in* $\mathcal{R}_{\mathcal{TP}}$.

Example 7.3.2 *If we take* η_5 *in* $Obs(\mathbb{O}rch)$ *of Figure 7.5(a) as the only target state, the corresponding test purpose* $\mathcal{TP}$ *for* $Obs(\mathbb{O}rch)$ *is the one depicted in Figure 7.6. In this case, the test purpose represents the entire behavior of the orchestrator (inputs, outputs, and observations, as well as the internal conditions and affectations) that has to be met in order to get to reach target state: the user gets the prize (*η_5*).*

Target states are symbolic states and so they have an associated path condition. In our rule-based algorithm, we use the logical disjunction of all the path conditions of all the target states of a test purpose in order to verify if at any given moment (no matter which state $\eta \in Obs(SE(\mathbb{O}rch))$ we consider), we are at least in the path of a target state (i.e., if this condition is satisfiable). This condition is called *target condition*.

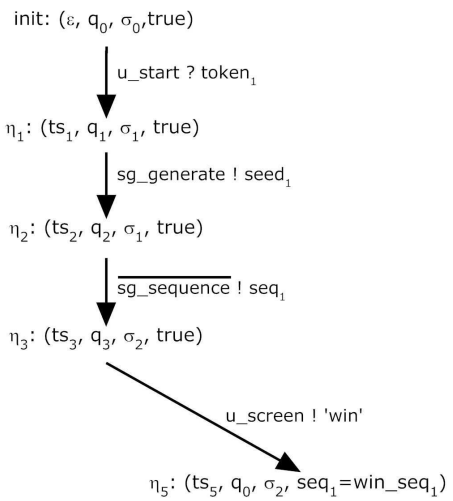


Figure 7.6: Test purpose $\mathcal{TP}$ for $Obs(\mathbb{O}rch)$ of Figure 7.5(a). $Accept(\mathcal{TP}) = \eta_5$.

Definition 7.3.3 (Target condition) *Let $Obs(\mathbb{O}rch)$ be the observable behaviors of $SE(\mathbb{O}rch)$. Let us note*[a]
$Accept(\eta, \mathcal{TP}) = \{\eta' \in \mathcal{R}_{Obs} \mid \exists m \in Act(\Sigma_F)^*, \eta \xrightarrow{m} \eta' \text{ and } \eta' \in Accept(\mathcal{TP})\}$

The target condition of $\mathcal{TP}$ *for any $\eta \in \mathcal{R}_{Obs}$, denoted $targetCond(\eta, \mathcal{TP})$, is defined as the formula:*
$\bigvee_{\eta' \in Accept(\eta, \mathcal{TP})} pc(\eta')$

[a] Where m is a sequence of communication actions of the form $m_1 \cdots m_n$, such that there exists a path in the symbolic execution tree of the form $\eta \xrightarrow{m_1} \cdots \xrightarrow{m_n} \eta'$

Example 7.3.3 *Following our example, if we consider Figure 7.5(a) and we assume $Accept(\mathcal{TP}) = \{\eta_5\}$ (so we get $\mathcal{TP}$ depicted in Figure 7.6), then, for instance, the target condition of η_1 is $seq_1 = win\_seq_1$, which is the disjunction of all target states that can be reached from it (η_5). No matter in what state of the tree we are, if this condition is satisfiable, it means that we can still* reach *the target state.*

Test purposes are usually selected by experts of the system that know the key behaviors that have to be tested. However, we mention some criteria in order to select test purposes in an automated way, the first two already defined in [Gaston 2006]:

1. **All paths of length n criterion**, which consists on selecting as test purposes all paths of the symbolic execution of length n that end with an output. In this way, the tester can make a trade-off between the size of the test purposes and the testing cost.

2. **k-inclusion criterion**, which consists on selecting as test purposes a subpart

of a symbolic execution with no redundant behaviors, where k denotes the number of basic behaviors that are verified not to be redundant.

3. **Root re-visited or depth reached**, which, as for the symbolic execution, consists on selecting as test purposes the paths of the symbolic execution of a given depth or the paths that are cut when the initial state of the associated $IOSTS$ appears for the second time.

This last criterion is particularly well adapted for testing orchestrators, since revisiting the initial state can be thought of as ending the process (in WS-BPEL, for instance, the instance of the orchestrator is destroyed when the final state is reached).

7.4 Rule-based Algorithm

In this section we define our rule-based algorithm to test the conformance of orchestrators in context, in which generation of test data sequences is guided by test purposes to be covered. This algorithm is based on the one presented in [Gaston 2006] and extended with two verdicts that take into account the interactions with the Web services.

We start this section by introducing some key notions of the algorithm and then we give the verdicts that the algorithm can emit. Then we define the rule-based algorithm and we provide a detailed example for the case where the communication channels are controllable or observable. Finally, we present the adjustments that have to be made to the algorithm in order to also take into account the hidden channels.

7.4.1 Key Notions of the Algorithm

Our algorithm aims at executing a symbolic execution tree-like structure, called a test purpose, on a system under test in order to produce test verdicts. The test purpose and the system under test are synchronized by coupling emissions and receptions. The main advantage of characterizing test purposes from a symbolic execution of $\mathbb{O}rch$ is that the testing process can be expressed as a simultaneous traversal, of both, the symbolic execution and the test purpose. Verdicts are emitted according to the fact that the observed behavior, in the form of a sequence of inputs (stimuli) and outputs (observations), does or does not belong to the test purpose *and* to the symbolic execution.

We start by introducing the notion of a *context*, denoted (η, f). A context denotes a state that can be reached from another state taking into account all previous observations/stimuli encountered to reach it. It is composed of a symbolic state $\eta \in \mathcal{S}_{reach}$, and a formula f whose variables are in F and expressing constraints induced by the sequence of previously encountered inputs/outputs. As there may

be many contexts compatible with a sequence of observations/stimuli, we use sets of contexts, denoted SC.

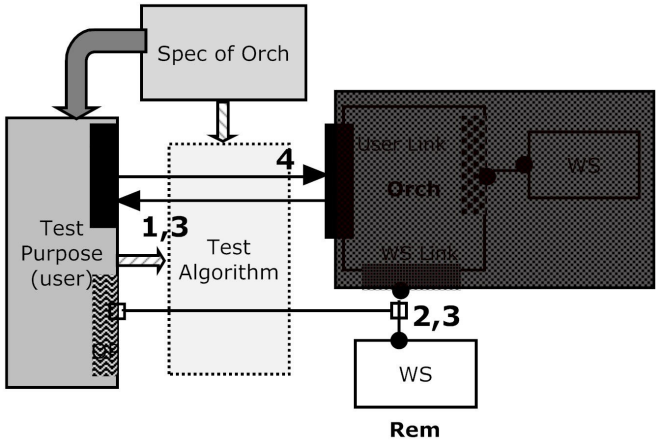


Figure 7.7: Classification of events ev.

The second concept is the notion of an *event*. An event can be an observation or a stimuli and is a value sent or received through a communication channel. We use the notation ev to denote an event, and we note:

1. $Orch.obs(ev)$ to say that ev is of the form $c!v$, with $c \in C_c \cup C_o$ and $v \in M$, or is the quiescent action δ.

2. $WS.obs(ev)$ to say that ev is of the form $\bar{c}!v$, corresponding to an observation of a message sent by some Web service, $c \in C_o$.

3. $obs(ev)$ to say that ev is of one of the two forms previously described.

4. $stim(ev)$ to say that ev is of the form $c?v$ with $c \in C_c$, corresponding to a stimulus made by the tester.

This classification of events is as depicted in Figure 7.7, where numbers in the figure correspond to the ones of the list just introduced.

Moreover, any $v \in M$ occurring in such events can be symbolically denoted by a term $t \in T_\Omega(\emptyset)$, where t is a term with no variables (t is said to be a ground term) such that $\nu(t) = v$ for all Ω-interpretation $\nu : F \to M$. In the following, values in M will simply be denoted by any corresponding ground term. Events ev will be abusively considered as element of $Act(\Sigma_F)$, up to this direct correspondence between values and ground terms.

Given a set of context SC, we define the function that generates the set of symbolic states that can be reached from any symbolic state in SC after the occurrence of an event ev.

Definition 7.4.1 (Function $Next(ev, SC)$) *Let $Obs(\mathbb{O}rch) = (init, \mathcal{R}_{Obs})$ be the observable behaviors of $SE(\mathbb{O}rch)$, SC be a finite set of contexts, and $ev \in Act(\Sigma_F)$.*

If ev is of the form $c\triangle t$, with $\triangle \in \{?, !\}$, then $(\eta', f') \in Next(ev, SC)$, with $\eta' = (ts(\eta).c\triangle t, q', \pi', \sigma')$, if and only if:

- *there exists $(\eta, f) \in SC$ such that $(\eta, c\triangle u, \eta') \in \mathcal{R}_{Obs}$*
- *f' is $f \wedge (t = u)$ and $f' \wedge \pi'$ is satisfiable*

If ev is of the form δ, then $(\eta', f') \in Next(ev, SC)$ if and only if there exists $(\eta, f) \in SC$ such that $(\eta, \delta, \eta') \in \mathcal{R}_{Obs}$, and f' is f, and $f' \wedge \pi'$ is satisfiable

Remark 6 *If there exists such a context (η', f') for all contexts (η, f) in SC, then $Next(ev, SC)$ is said to be* conservative.

Due to the discussion presented in Section 3.2.3, when taking hidden communication channels into account (**uioco**), we require $\mathbb{O}\diagdown\approx$ to be input complete on those channels. In order to define test purposes for those situations (as introduced later in Section 7.4.4), we require that, for any ev of the form $WS.obs(ev)$, $Next(ev, SC)$ is conservative.

We now define three more notions that are going to be used in our algorithm when computing a verdict. The first one is the set $Skip(SC)$, which consists of the set of contexts whose symbolic state is in $\mathcal{TP}$ and whose formula in conjunction with the $targetCond(\eta, \mathcal{TP})$ formula is satisfiable, that is, the set contexts from which a target state can eventually be reached.

Definition 7.4.2 ($Skip(SC)$) *Let SC be a set of contexts.*

$$Skip(SC) = \{(\eta, f) \mid (\eta, f) \in SC, \eta \in \mathcal{T}P, \\ (targetCond(\eta, \mathcal{TP}) \wedge f)\ is\ satisfiable\}$$

The second notion is the set $Pass(SC)$, which consists on all the contexts in $Skip(SC)$ whose symbolic states are in $Accept(\mathcal{TP})$.

Definition 7.4.3 ($Pass(SC)$) *Let SC be a set of contexts.*

$$Pass(SC) = \{(\eta, f) \mid (\eta, f) \in Skip(SC), \eta \in Accept(\mathcal{T}P)\}$$

The third notion is the set $Report(SC)$, which is no more than the set of symbolic traces and the conjunction of formulae in SC. This set provides important information that will help the tester in determining what might have happened in the system in order to reach a given verdict.

Definition 7.4.4 ($Report(SC)$) *Let SC be a set of contexts.*

$$Report(SC) = \{(ts, \pi \wedge f) \mid ((ts, q, \sigma, \pi), f) \in SC\}$$

7.4.2 Rules and Verdicts

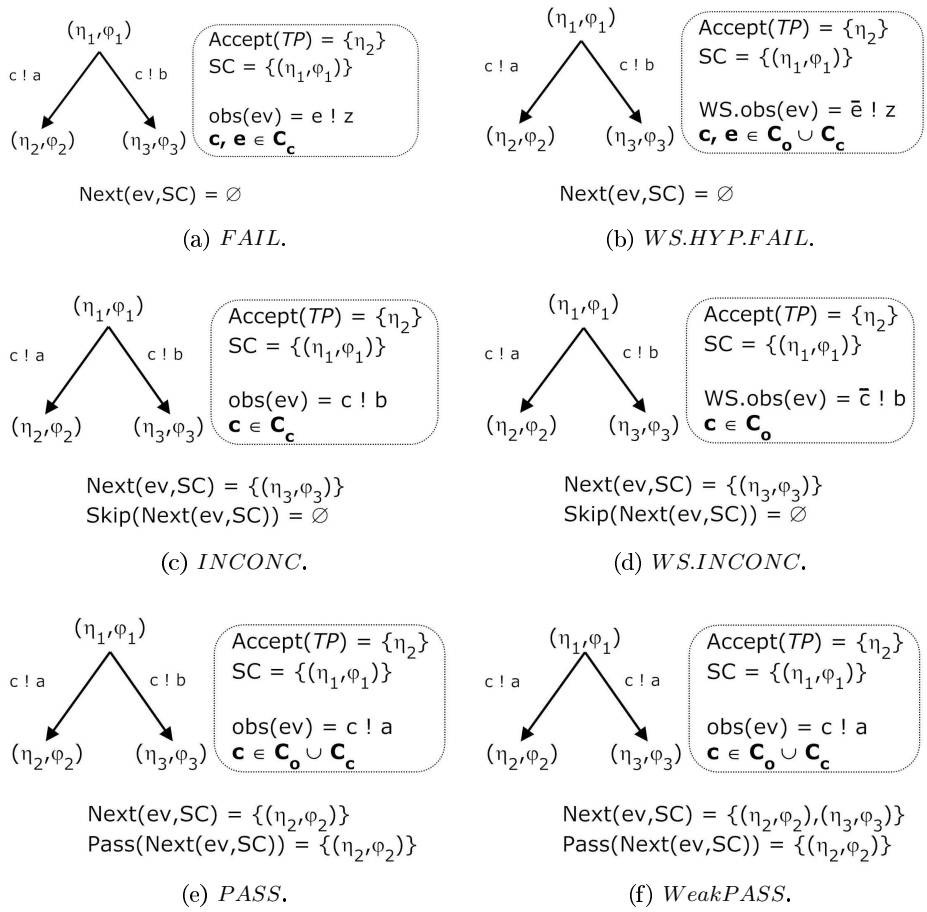


Figure 7.8: Verdicts of the algorithm.

The algorithm is given as a set of rules, each one emitting a verdict or building a new context, according to the observation of an event ev of the form introduced in Section 7.4.1. In practice, most of the times we build a new context and stimulate the system under test. The algorithm ends with the emission of a verdict. Each verdict is described by means of inference rules holding on sets of contexts. The rules are of the form:

$$\frac{SC}{Result} \quad cond(ev),$$

where SC is a set of contexts, $Result$ is either a set of contexts or a verdict, and $cond(ev)$ is a set of conditions including the observation ($obs(ev)$, $Orch.obs(ev)$ or $WS.obs(ev)$) or the stimulus ($stim(ev)$), which justifies the evolution of the set of context. The idea behind the rules, as depicted in Figure 7.8, is that, given the

current set of contexts SC, if $cond(ev)$ is verified, then the algorithm may achieve a step with ev as elementary action. Six verdicts can be defined (four previously defined in [Gaston 2006], and two new ones regarding Web service observations):

1. $FAIL$, when the behavior belongs neither to $\mathcal{TP}$ or to $Obs(\mathbb{O}rch)$ (**Rule 2** for $Orch.obs(ev)$, given in next section —Sect. 7.4.3). In this case, the set $Next(ev, SC)$ is the empty set, meaning that the observation does not lead $Obs(Orch[Rem], C_o)$ to any specified state in $Obs(\mathbb{O}rch)$. Thus, this is an invalid behavior of $Obs(Orch[Rem], C_o)$. This verdict is depicted in Figure 7.8(a): the value z is observed on the channel e, which is an unexpected (invalid) observation of $Obs(Orch[Rem], C_o)$.

2. $WS.HYP.FAIL$, when a reception on an observable channel is observed, but is not specified in $Obs(\mathbb{O}rch)$ (**Rule 2.WS** for $WS.obs(ev)$, given in next section). As for the previous rule, an unspecified state in $Obs(\mathbb{O}rch)$ is reached, however, in this case the observation does not come from $Orch$ but from a remote Web service in Rem (thus, the observation is done on an observable communication channel). Thus, this is an unexpected behavior of the Web service. However, we cannot conclude that there is a failure in the Web service since we are not taking its specification into account, and it may be the case that from the Web service's perspective, it is in fact a valid behavior. This is why we give the verdict $WS.HYP.FAIL$, which stands for *Web Service Hypothetical FAILure*. This verdict is depicted in Figure 7.8(b): the value z is observed (sent to the orchestrator) on a channel used by the Web service where the orchestrator was not expecting anything.

3. $INCONC$, when the behavior belongs to $Obs(\mathbb{O}rch)$ but not to $\mathcal{TP}$ (**Rule 3** for $Orch.obs(ev)$, given in next section). In this case, the observation does lead $Obs(Orch[Rem], C_o)$ to an specified state in $Obs(\mathbb{O}rch)$ (the set $Next(ev, SC)$ is not empty), but this state is not in $\mathcal{TP}$ (the set $Skip(Next(ev, SC))$ is empty). The verdict stands for *inconclusive* because even if one is sure that the target state cannot be reached from the new set of context ($Next(ev, SC)$), the trace is compatible with the specification. Thus, this does not correspond to an invalid behavior even if it does not belong to the test purpose. This verdict is depicted in Figure 7.8(c). Even if in the figure there is no way to reach the target state from η_3, it is still a valid behavior according to $\mathbb{O}rch$.

4. $WS.INCONC$, when a reception on an observable channel happens, belonging to $Obs(\mathbb{O}rch)$ but not to $\mathcal{TP}$ (**Rule 3.WS**, given in next section). As for the previous case, the set $Next(ev, SC)$ is not empty but the set $Skip(Next(ev, SC))$ is, however, in this case the observation does not come from $Orch$ but from Rem. This verdict is depicted in Figure 7.8(d).

5. $PASS$, when the behavior belongs to a path of $\mathcal{TP}$ ending by a target state and not to any other path of $Obs(\mathbb{O}rch)$ (**Rule 4**, given in next section). In this case, the observation makes the sets $Next(ev, SC)$ and $Pass(Next(ev, SC))$ to be the

same, thus, it means that the only reachable state(s) taking ev into account are precisely the target state(s) (usually is only one state that is reached, but it could be more). This is a successful test and the verdict is depicted in Figure 7.8(e).

6. $WeakPASS$, when the behavior belongs to a path of $\mathcal{TP}$ ending by an accept state and to at least one other path of $Obs(\mathbb{O}rch)$ (**Rule 5**, given in next section). In this case, the sets $Next(ev, SC)$ and $Pass(Next(ev, SC))$ are not empty but they are not the same, meaning that there are some states in $Next(ev, SC)$ that can be reached with ev but that are not target states, and this is mainly due to the non-determinism of $Obs(Orch[Rem], C_o)$ when there is an observation that can lead to at least two different states. Even if the $IOSTS$s and symbolic executions use conditions and guards, it may happen that the same condition (or the lack of it) allows more than one state to be reached with a single observation. This verdict is depicted in Figure 7.8(f).

7.4.3 Algorithm for Observable and Controllable Cases

If all the communication channels are either observable or controllable, sequences of stimuli and observations built by the interaction between the algorithm and the SUT are modeled as elements of $STraces(SUT)$, and the algorithm is defined in Algorithm 2. For readability reasons, we use the notation $Next$ for $Next(ev, SC)$. Note also that for the verdicts $FAIL$ and $WS.HYP.FAIL$, we include the observation that provoked the verdict.

Let us explain **Rule 4**: in this case, the condition is actually composed of two other ones: $Pass(Next(ev, SC)) = Next(ev, SC)$ and $Next(ev, SC) \neq \emptyset$. This means that, in order to apply the action $PASS, Report(Next)$, this condition has to be satisfiable by taking into account the event $obs(ev)$ and by using the set of contexts SC. That is, the set of contexts resulting of applying the function $Next(ev, SC)$ (of Definition 7.4.1), must be the same as the resulting set of contexts of applying the Definition 7.4.3 (with $SC = Next(ev, SC)$, i.e., $Pass(Next(ev, SC))$). Only if this condition is satisfiable, the verdict $PASS$, jointly with the $Report(Next(ev, SC))$, can be emitted.

Example 7.4.1 *Let us present a guided example of our rule-based algorithm taking into account that* $Obs(\mathbb{O}rch)$ *is the one depicted in Figure 7.5(a), and that the test purpose* $\mathcal{TP}$ *is the one depicted in Figure 7.6. Both trees are (re)depicted together in Figure 7.9. The* target state *is* η_5*, so* $Accept = \{\eta_5\}$*, and the* target condition *is* $seq_1 = win\_seq$*.*

1. *The orchestrator is waiting for an event.* ***Rule 6*** *is applied. The tester computes an stimulus that will guide the SUT* $(Obs(Orch[Rem], C_o))$ *to the desired target state. For instance, the tester computes the stimulus* $stim(u\_start?1)$*. Thus, the set of contexts* $SC = \{(init, true)\}$*,* $Next(ev, SC) = \{(\eta_1, true)\}$ *and the set* $Skip(Next(ev, SC)) = \{(\eta_1, true)\} \neq \emptyset$ *(since the target condition is satisfiable). The tester sends the value* 1*, which we know is the value of the*

Algorithm 1: Rule-based algorithm to test the conformance of orchestrators in context.

Rule 1: The observation is compatible with $\mathcal{TP}$ but no target state is reached.

$$\frac{SC}{Next}\ obs(ev), Skip(Next) \neq \emptyset, Pass(Next) = \emptyset$$

Rule 2: The $Orch$ observation is not expected in $Obs(\mathbb{O}rch)$.

$$\frac{SC}{FAIL, Report(SC), ev}\ Orch.obs(ev), Next = \emptyset$$

Rule 2.WS : The WS observation is not specified in $Obs(\mathbb{O}rch)$.

$$\frac{SC}{WS.HYP.FAIL, Report(SC), ev}\ WS.obs(ev), Next = \emptyset$$

Rule 3: The $Orch$ observation is specified in $Obs(\mathbb{O}rch)$ but not in $\mathcal{TP}$.

$$\frac{SC}{INCONC, Report(Next)}\ Orch.obs(ev), Next \neq \emptyset, Skip(Next) = \emptyset$$

Rule 3.WS: The WS observation is specified in $Obs(\mathbb{O}rch)$ but not in $\mathcal{TP}$.

$$\frac{SC}{WS.INCONC, Report(Next)}\ WS.obs(ev), Next \neq \emptyset, Skip(Next) = \emptyset$$

Rule 4: All symbolic states in $Next$ are in $Accept(\mathcal{TP})$.

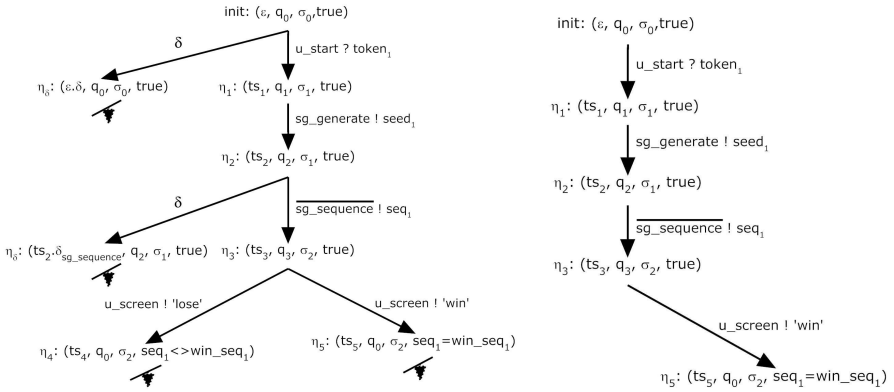
$$\frac{SC}{PASS, Report(Next)}\ obs(ev), Pass(Next) = Next, Next \neq \emptyset$$

Rule 5: Some next contexts have states in $Accept(\mathcal{TP})$, but not all of them.

$$\frac{SC}{WeakPASS, Report(Next)}\ obs(ev), Pass(Next) \neq \emptyset, Pass(Next) \neq Next$$

Rule 6: Stimulate the SUT

$$\frac{SC}{Next}\ stim(ev), Skip(Next) \neq \emptyset$$



(a) $Obs(\mathbb{O}rch)$. (b) $\mathcal{TP}$.

Figure 7.9: Inputs of the algorithm for the controllable case.

symbolic variable $token_1$ *and, as we know the random algorithm used by the implementation (or at least the one specified by* $\mathbb{O}rch$*), we also know that the values computed for* $seed_1$ *and* win_seq_1 *are, respectively* 6 *and* 4.

2. *An observation happens, that is,* $Orch.obs(ev) = obs(sg\_generate!6)$. *It means that the orchestrator sends the value* 6 *to the Sequence Generator service through the communication channel* $sg\_generate$. *So,* $token_1 = 1$, $seed_1 = 6$ *and* $win\_seq_1 = 4$*; thus, the new sets of contexts are now:* $SC = \{(\eta_1, true)\}$, $Next(ev, SC) = \{(\eta_2, true)\}$, $Skip(Next(ev, SC)) = \{(\eta_2, true)\}$ *(since the target condition is satisfiable) and* $Pass(Next(ev, SC)) = \emptyset$. *Therefore,* ***Rule 1*** *is applied.*

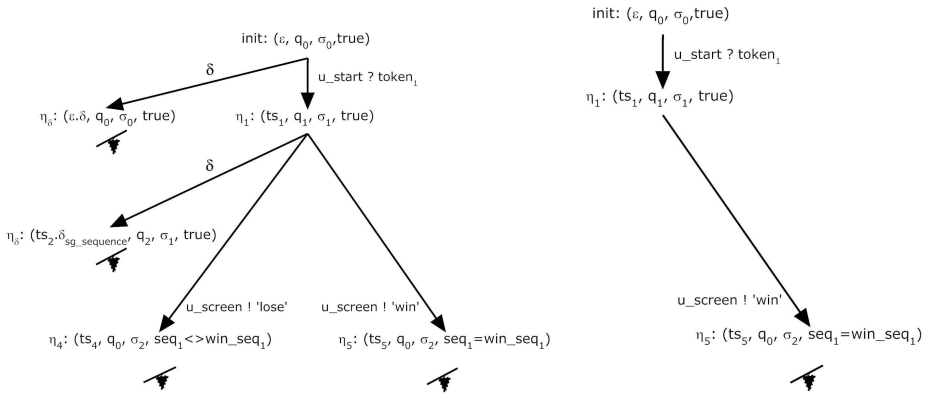
3. *An observation happens, that is,* $WS.obs(sg\_sequence?4)$. *It means that the Sequence Generator service sends the value* 4 *to the slot machine's interface through the communication channel* $sg\_sequence$. *So,* $token_1 = 1$, $seed_1 = 6$, $win\_seq_1 = 4$, *and* $seq_1 = 4$*; thus, the new sets of contexts are, in this step:* $SC = \{(\eta_2, true)\}$, $Next(ev, SC) = \{(\eta_3, true)\}$, *which is the same as the set* $Skip(Next(ev, SC)) = \{(\eta_3, true)\}$ *(since the target condition is satisfiable), and* $Pass(Next(ev, SC)) = \emptyset$. *Therefore,* ***Rule 1*** *is applied. Note that in this state, quiescence was accepted, but the Sequence Generator service reacted before entering into a quiescence situation. Also, we suppose that the Sequence Generator service sends the value* 4, *which is in fact the winner sequence. However, in a real case scenario and since the Sequence Generator service is not controlled by the tester, this is only an assumption and several observations would have to be made in order to test this path of the tree.*

4. *An observation happens, that is,* $Orch.obs(ev) = obs(u\_screen!'win')$. *It means that the orchestrator sends the value 'win' to the user service through the communication channel* u_screen *(in fact, this value should be also stored in a*

new symbolic variable, but as explained before, we abusively skip this step for the sake of readability). So, $token_1 = 1$, $seed_1 = 6$, $win\_seq_1 = 4$, *and* $seq_1 = 4$; *thus:* $SC = \{(\eta_3, true)\}$, $Next(ev, SC) = \{(\eta_5, seq_1 = win\_seq_1)\}$, $Skip(Next(ev, SC)) = Pass(Next(ev, SC)) = \{(\eta_5, seq_1 = win\_seq_1)\}$. *Therefore,* ***Rule 4*** *is applied and the verdict* $PASS$ *is emitted, jointly with the collected information* $Report(Next(ev, SC)) = \{$ $(u\_start?token_1.\ sg\_generate!seed_1.\ sg\_sequence?seq_1.\ u\_screen!'win'\}$, $token = 1 \wedge seed = 6 \wedge win\_seq = 4\ wedgeseq = 4)\}$.

7.4.4 Algorithm for the Hidden Case

If we take the hidden channels into account, in order to extend the algorithm to **uioco**, we consider events as elements of $Act(C \setminus C_h, M)$, and we simply have to check that, at each step of an application of a rule involving an event $WS.obs(ev)$ from a set of context SC, we have that $Next(ev, SC)$ is conservative. If it is not the case, the algorithm should stop and send a verdict expressing that the characterized sequence of actions does not belong to $UTraces(\mathbb{O}rch)$. Similarly **Rule** 6 should only be applied for events ev such that $Next(ev, SC)$ is conservative.



(a) $Obs(\mathbb{O}rch)$. (b) $\mathcal{TP}$.

Figure 7.10: Inputs of the algorithm for the hidden case.

Example 7.4.2 *Let us consider Figure 7.10. Figure 7.10(a) depicts the* $Obs(\mathbb{O}rch)$ *with the assumption that the communication channels are hidden, and Figure 7.10(b) depicts a test purpose for that* $Obs(\mathbb{O}rch)$. *In our previous example, the property that* $Next(ev, SC)$ *is conservative holds (for each event that was observed in the Web service), so apart from the fact that* $Obs(\mathbb{O}rch)$ *and* $\mathcal{TP}$ *are different, the rest remains the same the same. That is, from the steps in Example 7.4.1, only the steps 2 and 3 are omitted, the rest are the same.*

Finally, our algorithm is based on the one presented in [Gaston 2006, Touil 2006], but extended by the addition of two new verdicts regarding interactions with the Web services, and with minor changes in the definition of the sets used together with the algorithm. Also, the inputs of the algorithm changed, specially for the symbolic execution. In this case, $Obs(\mathbb{O}rch)$ has more information than the symbolic execution without any modifications. However, the key behavior of the algorithm was not changed so we can naturally assume that the theorem of correctness and completeness given in [Touil 2006] can also be obtained with our version of the algorithm. The theorem states that[1]:

Correctness: If $Obs(Orch[Rem], C_o)$ conforms to $\mathbb{O}rch$, no matter what test purpose we consider, there is no way to compute the verdict $FAIL$ in the algorithm.

Completeness: If $Obs(Orch[Rem], C_o)$ does not conform to $\mathbb{O}rch$, then there exists a test purpose for which the algorithm produces the verdict $FAIL$.

7.5 Conclusion

In this chapter we have presented a symbolic approach to test orchestrators in their context of usage. By taking the different status of the communication channels used by the Web services to interact with the orchestrator into account, we proceeded to generate the observable behaviors, $Obs(\mathbb{O}rch)$, of the orchestrator in context by performing consecutively on the symbolic execution the quiescence enrichment, WS input transformation, hidden and τ-reduction operations. We have shown how to select the behaviors to test in $Obs(Orch[Rem], C_o)$ which are called test purposes. By using the test purposes and the observable behaviors of the orchestrator in context, we have defined the rule-based algorithm to test the conformance of orchestrators in context. This algorithm is based on $\overline{\textbf{ioco}}$ for the observable and controllable cases, and on $\overline{\textbf{uioco}}$ for the observable, controllable and hidden cases. The algorithm stimulates $Obs(Orch[Rem], C_o)$ and emits a verdict as soon as possible, jointly with a report that proves useful for the tester when determining the cause of the verdict.

In the next chapter, we conclude Part II with the symbolic counterpart of Chapter 5, which complements this work by means of generating behaviors from $SE(\mathbb{O}rch)$ in order to test the Web services to determine if they are compatible with the orchestrator.

[1] See [Touil 2006] for a proof.

Chapter 8

A Method for Testing Web Service's Compatibility

Contents

8.1 Introduction

The symbolic execution represents the valid behaviors for an implementation of the *orchestrator*. In this chapter we take advantage of the fact that the orchestrator is the one that guides the whole system to extract some behaviors from its symbolic execution in order to test the compatibility of *Web services* involved in the process. We do this mainly because it is common not to have the complete and precise behavioral specification of all the Web services involved in the orchestration. Thus, we aim at testing the compatibility of the Web service behaviors as they are expected from $\mathbb{O}rch$. In fact, the Web services may offer more services and functionalities than the ones requested from $\mathbb{O}rch$, but by studying the behaviors of $\mathbb{O}rch$, we do not consider all of them. In order to generate the behaviors from the symbolic execution of $\mathbb{O}rch$, we introduce two operations: projection and mirror, which are they symbolic version of the ones presented in Chapter 5. Roughly speaking:

The projection operation only keeps the interactions between $\mathbb{O}rch$ and the targeted Web services.

The mirror operation inverses receptions and emissions so that behaviors are considered from the point of view of the Web service instead of $\mathbb{O}rch$.

As for the numerical case (Section 5.4), we also identify the situations where the Web services are allowed to be quiescent without leading the orchestrator into

a deadlock state. The resulting structure is the set of all the valid behaviors of the Web service as expected from the orchestrator.

With the assumption that the orchestrator invokes all the Web services involved in the orchestration (fairness invocation of Web services property), we can consider the behaviors described above as test cases that can be used together with our algorithm presented in Section 7.4 in order to test Web services. More specifically, we use the compatibility relation introduced in Section 5.4 to test Web services. We say that if a Web service meets the compatibility relation with respect to the behaviors that $\mathbb{O}rch$ expects from it, then it meets the deadlock-free property. This also means that the Web service cannot lead the orchestration into a deadlock state.

We begin this chapter in Section 8.2 by introducing the technical preliminaries of our work, including the mirror and projection operations at the symbolic level. With these operations, and together with the quiescence enrichment for the Web services, we show how to extract the behaviors for the Web services from the ones of the orchestrator. In Section 8.3 we show how the extracted behaviors can be used to test the deadlock-free property of a given Web service. Section 8.4 concludes the chapter.

8.2 Web service Behaviors Inferred from Orchestrators

In this section we show how to obtain Web service behaviors from the symbolic execution of $\mathbb{O}rch$ (the specification of the orchestrator), $SE(\mathbb{O}rch)$. In order to do that, we begin by defining the technical preliminaries which are a series of notions and operations that are then used to define the executable behaviors for Web services. Thus, those behaviors are extracted from $SE(\mathbb{O}rch)$ after some modifications.

8.2.1 Technical Preliminaries

As for the numerical case presented in Section 5.4, we suppose that a set $\mathbb{W}$ of so-called *Web service names* is given, which allows us to differentiate between the channels used to communicate with the user and with each of the Web services. Thus, we partition the set of communication channels C (introduced in Definition 6.2.1, Section 6.2) in such a way that C is now of the form $C_c \cup \bigcup_{w \in \mathbb{W}} C_w$, and satisfying: for all $w \neq w' \in \mathbb{W}$, $C_c \cap C_w = \emptyset$ and $C_w \cap C_{w'} = \emptyset$. We assume that a corresponding set of fresh variables F_w is associated to each Web service, and, for every $w \neq w'$, $F_w \cap F_{w'} = \emptyset$.

Each Web service, has then, an associated sub-signature, defined over its corresponding set of communication channels.

Definition 8.2.1 (Web service sub-signature) *Let $\mathbb{W}$ be a set of Web service names, and let C be a set of channels partitioned as $C_c \cup \bigcup_{w\in\mathbb{W}} C_w$, F_w a set of fresh variables, $\Sigma_F = (F_w, C)$ be an $IOSTS$ signature, and $w \in \mathbb{W}$ be a Web service name.*
The sub-signature of w *is defined as:*
$\Sigma_F^w = (F_w, C_w)$

In the sequel, we suppose that a Web service sub-signature Σ_F^w is given.

As discussed in Section 5.2, authorizing the Web service to be quiescent may only be decided when the orchestrator sends it a message. For this reason, we impose that any orchestrator always finally invokes all its associated Web services unless its execution terminates. This corresponds to what we call the *fairness invocation of Web services*. This necessary restriction is reasonable with respect to real life orchestrators which generally characterize finite sequences of interactions with Web services.

Definition 8.2.2 (Fairness invocation of Web services property) *Let $SE(\mathbb{O}rch) = (init, \mathcal{R}_{sat})$ be a symbolic execution.*

The fairness invocation of Web services *property is such that:*

for any w such that $C_w \neq \emptyset$, there does not exist an infinite sequence of consecutive symbolic transitions $(str_n)_{n\in\mathbb{N}}$ with $str_n \in \mathcal{R}_{sat}$, $source(str_0) = init$, and for all $i \in \mathbb{N}$, $target(str_i) = source(str_{i+1})$, and such that for all $i \in \mathbb{N}$, we have $act(str_i) \notin O({\Sigma_F}^w, F)$

We now present the necessary operations to transform the behaviors from the point of view of the orchestrator to the one of the Web services. We achieve this by defining the symbolic version of the projection and mirroring operations, introduced in Definitions 5.3.1 and 5.3.2 in Section 5.3 for the $IOLTS$s. The projection operation's idea is to *remove* all the interactions in the symbolic execution that do not involve the Web service under test.

Definition 8.2.3 (Projection of $SE(\mathbb{G})$) *Let $SE(\mathbb{G}) = (init, \mathcal{R}_{sat})$ be a symbolic execution, and let $A \subseteq Act(\Sigma_F)$ be a subset of the communication actions over Σ_F.*

The projection of $SE(\mathbb{G})$ over A, *denoted $SE(\mathbb{G}) \downarrow_A$, is the couple $(init, \mathcal{R}_A)$, where $\mathcal{R}_A$ is the least relation so that for any sequence $(\eta_1, sa_1, \eta_2) \cdots (\eta_m, sa_m, \eta_{m+1})$ of transitions in $\mathcal{R}_{sat}$, we have:*

- *if there exists (η, act, η_1) in $\mathcal{R}_{sat}$, then $act \in A$*
- *$sa_m \in A$ and $\forall 1 \leq i < m$, $sa_i \notin A$*

then $(\eta_1, sa_m, \eta_{m+1}) \in \mathcal{R}_A$

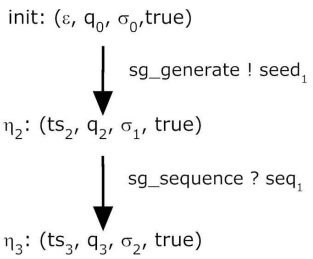


Figure 8.1: Projection example.

Example 8.2.1 *Figure 8.1 depicts the projection of the symbolic execution $SE(\mathbb{G})$ of Figure 6.6 presented in Section 6.3 (corresponding to the Slot Machine example), and with the assumption that it is projected over the subset of communication actions $A = \{F, \{sg\_generate, sg\_sequence\}\}$. Thus, all the information in $SE(\mathbb{G})$ that has nothing to do with the communication actions in A is just* ignored *in the resulting structure $SE(\mathbb{G}) \downarrow_A$, that is, all the interactions between the user and the slot machine's interface (since in this case there is only one Web service in the orchestration).*

The mirror operation transforms all communication actions as if they were seen from the point of view of a Web service. It basically consists on transforming inputs into outputs and vice-versa.

Definition 8.2.4 (Mirror of a symbolic execution) *Let $SE(\mathbb{G}) = (init, \mathcal{R}_{sat})$ be a symbolic execution.*

The mirror of *$SE(\mathbb{G})$, denoted $Mir(SE(\mathbb{G}))$, is the couple $(init, Mir(\mathcal{R}_{sat}))$, where for each $str \in \mathcal{R}_{sat}$ of the form (η, sa, η'), $Mir(str)$ is defined as follows:*

- *if sa is of the form $c!z$, then $Mir(str) = (\eta, c?z, \eta')$*
- *if sa is of the form $c?z$, then $Mir(str) = (\eta, c!z, \eta')$*
- *else, $Mir(str) = str$*

Example 8.2.2 *Figure 8.2 depicts the mirror of the symbolic execution $SE(\mathbb{G})$ of Figure 6.6 presented in Section 6.3. The idea of this operation is simple: to transform inputs into outputs and vice-versa, so the interactions between the slot machine's interface and the Sequence Generator service SG are represented from SG's point of view. In this example, transforming the interactions with the user makes no sense, and that is why we use the mirroring operation together with the projection one, as explained in next section.*

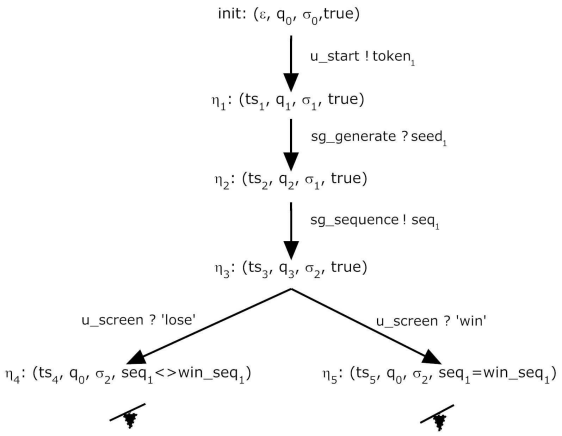


Figure 8.2: Mirror example.

8.2.2 Executable Behaviors for Web services

We start by introducing the *Web service quiescence* enrichment over $SE(\mathbb{O}rch)$. This quiescence enrichment differs from the one introduced in Definition 6.3.9 in Section 6.3, which refers to the situation where the orchestrator may remain silent; and from the one introduced in Definition 7.2.1 in Section 7.2, which refers to the quiescence due to lack of response of Web services. The quiescence introduced here aims at identifying the situations where a *Web services* can be quiescent without leading the orchestrator into a deadlock state. Thus, it can be quiescent if the orchestrator is not expecting anything from it and also, strictly speaking, if the orchestrator can not execute any action.

Definition 8.2.5 (Enrichment by Web service quiescence) *Let $SE(\mathbb{O}rch) = (init, \mathcal{R}_{sat})$ be a symbolic execution, and $w \in \mathbb{W}$ be a Web service name.*

The enrichment by w-quiescence of $SE(\mathbb{O}rch)$, *denoted* ${SE(\mathbb{O}rch)_\delta}^w$, *is the restriction to $\mathcal{S}_{sat}$ of the couple $(init, \mathcal{R}_{sat} \cup \Delta)$, where $\Delta \subseteq \mathcal{S} \times \{\delta\} \times \mathcal{S}$ is defined as $\Delta_1 \cup \Delta_2$, and where Δ_1 and Δ_2 are respectively defined as follows:*

- *for any $\eta \in \mathcal{S}$, let us note $React_o^w(\eta)$ the set of all $(\eta, sa, \eta') \in \mathcal{R}_{sat}$ such that $sa \in O(\Sigma_F^w)$, and $\Pi_o(\eta)$ the formula* false *if $React_o^w(\eta) = \emptyset$, and $\bigvee_{(\eta,sa,\eta') \in React_o^w(\eta)} pc(\eta')$ otherwise,*
  then $(\eta, \delta, (ts(\eta).\delta, state(\eta), sub(\eta), pc(\eta) \wedge \Pi_o(\eta)) \in \Delta_1$
- *for any $\eta \in \mathcal{S}$, let us note $React_{all}^w(\eta)$ the set of all $(\eta, sa, \eta') \in \mathcal{R}_{sat}$, and $\Pi_{all}(\eta)$ the formula* true *if $React_{all}^w(\eta) = \emptyset$, and $\bigwedge_{(\eta,sa,\eta') \in React_{all}^w(\eta)} \neg pc(\eta')$ otherwise,*
  then $(\eta, \delta, (ts(\eta).\delta, state(\eta), sub(\eta), pc(\eta) \wedge \Pi_{all}(\eta)) \in \Delta_2$

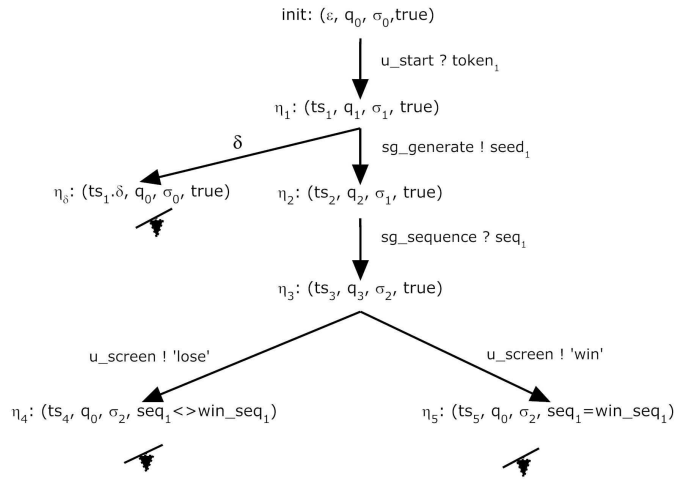


Figure 8.3: Enrichment by Web service quiescence for the Slot Machine example (Figure 6.6).

The auxiliary formula $\Pi_o(\eta)$ is used to determine if Web service quiescence can be allowed in the sense of the discussion presented in Section 5.2, where we concluded that a Web service can be quiescent only when the orchestrator sends it a message. $\Pi_o(\eta)$ is then the disjunction of all the path conditions of the target states of precisely the transitions denoting the emission of a message to a Web service. If there is none, $\Pi_o(\eta)$ is $false$ (quiescence due to interactions with the Web services is not allowed since there are no communications with them). Thus, Web service quiescence is allowed only if the path condition of the target states in conjunction with $\Pi_o(\eta)$ is satisfiable.

In the same way, the auxiliary formula $\Pi_{all}(\eta)$ is used to determine if Web service quiescence can be allowed due to the impossibility to move forward by the orchestrator, that is, the negation of all the path conditions of all the target states of the transition under examination. If this situation happens, the Web service is allowed to be quiescent. If there are no states that can be reached from the current one (which is not usually the case), $\Pi_{all}(\eta)$ is $true$ (Web service quiescence can be allowed). Thus, Web service quiescence is also allowed if the path condition of the target states of the examined transitions in conjunction with $\Pi_{all}(\eta)$ is satisfiable.

Example 8.2.3 *Figure 8.3 depicts the enrichment by Web service quiescence for the Slot Machine example. That is, Definition 8.2.5 was applied to the symbolic execution $SE(\mathbb{G})$ of Figure 6.6, where $\mathbb{G}$ is in fact $\mathbb{O}rch$. We can see that in state η_1, the Sequence Generator service SG is waiting for the request of the slot machine's interface SM. This is the only situation where it can be silent. In η_3, SM is waiting for a sequence form SG, and the quiescence is not allowed, since if it does not send the requested value, SM cannot evolve by itself, and thus it would enter in a deadlock*

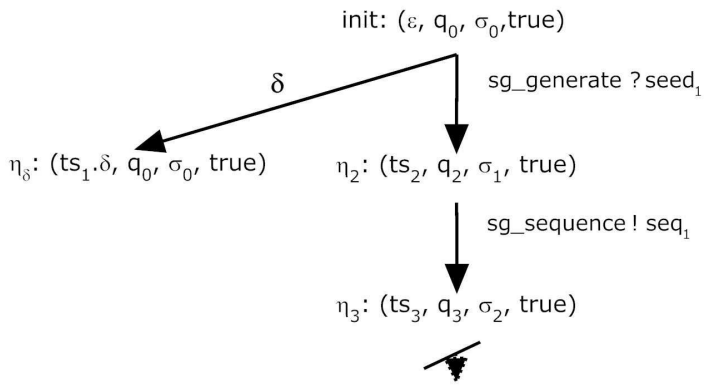


Figure 8.4: w-trace structure according to Figure 8.3.

situation.

We are now ready to define the w-traces that represent behaviors that can be used to test if a given Web service is compatible with an orchestrator. These behaviors are nothing more than the result of applying the projection and mirror operations on $SE(\mathbb{O}rch)_\delta{}^w$ for a given Web service name w. Besides, we explicitly remove any possible quiescent situation that is not allowed for the Web service.

Definition 8.2.6 (w-trace structure according to the orchestrator) *Let $SE(\mathbb{O}rch)$ be a symbolic execution, let $w \in \mathbb{W}$ be a Web service name, and let us consider the symbolic structure $(init, \mathcal{R}_{sat}) = Mir(SE(\mathbb{O}rch)_\delta{}^w \downarrow_{C_w})$. Let us define $\mathcal{R}_{junk}$ the set of all transitions $(\eta', c!z, \eta'')$ such that there exists in $\mathcal{R}_{sat}$ a transition of the form (η, δ, η').*

The symbolic structure of w-traces according to $\mathbb{O}rch$ *is defined as:* $(init, \mathcal{R}_w) = (init, (\mathcal{R}_{sat} \backslash \mathcal{R}_{junk}))$

According to Definition 8.2.6, w-traces according to $\mathbb{O}rch$ are the result of applying the mirror and projection operation to $SE(\mathbb{O}rch)$ enriched with Web service quiescence with respect to a given Web service w, that is, $Mir(SE(\mathbb{O}rch)_\delta{}^w \downarrow_{C_w})$. However, due to the nature of those operations, there may be some situations where we can have a trace like the following: $\sigma.\delta.c!z$, that is, according to that trace, w may emit a value after quiescence is observed. As already mentioned, a quiescent situation is allowed only if the system cannot emit and output or execute an internal action. This trace breaks our requirements for allowing quiescence. Thus, we *remove* such traces by means of the auxiliary set of symbolic states $\mathcal{R}_{junk}$.

Example 8.2.4 *Figure 8.4 depicts the w-trace structure according to the orchestrator of the Slot Machine example. This structure is nothing more than the result of enriching $SE(\mathbb{O}rch)$ with Web service quiescence, removing the transitions from*

the symbolic execution of Figure 8.3 that do not interact with SG, from converting inputs into outputs and vice-versa, and explicitly removing any possible quiescence situations where SG cannot be quiescent (which in this case there was none). Then, this trace represents a behavior that the Sequence Generator service has *to provide in order to correctly interact with the slot machine's interface. The rest of behaviors of the Sequence Generator service are not in need to be tested. More precisely, if SG specifies other behaviors, they do not concern interactions with* $\mathbb{O}rch$ *(the specification of the slot machine's interface), and then we are not interested in testing them.*

8.3 Testing the Deadlock-free Property

The resulting symbolic execution structure $(init, \mathcal{R}_w)$ defines the behaviors of the Web service w that are inferred from the orchestrator's specification, including the situations where the Web service w can be quiescent. These behaviors precisely capture the requirements upon Web services that the whole system does not meet a deadlock situation.

In Section 7.4, we have defined a rule-based algorithm to test orchestrators in context. The same algorithm can be used to test Web services against $(init, \mathcal{R}_w)$. However, in this case, all the communication channels are observable, thus the verdicts associated to Web services are useless. Thus, we can obtain a simplified version of the algorithm with only 4 verdicts: $WeakPASS$, when the behavior belongs to the test purpose and to, at least, one path of the symbolic execution which is not in the test purpose; $PASS$, when the behavior belongs to the test purpose and not to any path of the symbolic execution which does not belong to the test purpose; $INCONC$ (for inconclusive), when the behavior belongs to the symbolic execution and not to the test purpose; and finally, $FAIL$, when the behavior belongs neither to the test purpose nor to the symbolic execution.

Testing a Web service (using our algorithm introduced in Section 7.4) against $(init, \mathcal{R}_w)$ amounts to test the deadlock-free property. Any $FAIL$ verdict discovered for a given system SUT will mean that if the corresponding SUT is used to implement the Web service w, then a situation of deadlock can occur.

Example 8.3.1 *If we consider again Figure 8.4, we can define the test purpose depicted in Figure 8.5 (with* $Accept = \{\eta_3\}$*) and use it, jointly with our algorithm, to test the compatibility of any implementation of the Sequence Generator Web service with respect to* $\mathbb{O}rch$ *(slot machine's interface).*

8.4 Conclusion

This chapter complements the symbolic approach for testing orchestrators in context. We have shown how to test if the Web services do not lead the orchestration (involving an orchestrator specified by $\mathbb{O}rch$) into a deadlock state. In order to do

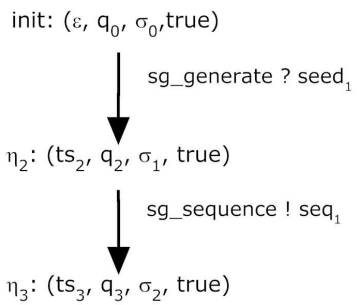


Figure 8.5: Test purpose for a Web service.

that, we have presented the notions of the signature associated with each of the Web services interacting with the orchestrator, the mirror and projection operations, as well as identified the situations where the Web service can be quiescent without leading the orchestration into a deadlock state. This state is avoided if every time that the orchestrator is waiting for an answer of the Web service (and only for that answer) in order to move forward, then the Web services are *forced* to react, otherwise, it can remain silent. The resulting traces (w-trace structure according to the orchestrator) can then be used together with our algorithm presented in Section 7.4. The test purposes in this case are sub-trees of the w-trace structures.

Thus, we have shown that even if we only take into account the specification of the orchestrator, since orchestrations are a special type of component-based system where the orchestrator is not only the interface of the system but also the component that guides the system, it is possible, under some hypotheses, to test partially Web services involved with the orchestrator.

In the next chapter we show how we have implemented a prototype of our algorithm to test orchestrators in context.

Chapter 9

Prototype for Test Case Generation

Contents

9.1 Introduction

Based on the symbolic approach introduced in Chapter 7, in this chapter we present the implementation of the rule-based algorithm to test orchestrators in context. The algorithm performs essentially 3 actions: it computes input values to be sent to the implementation under test IUT, each time it is necessary, in order to cover test purposes; it observes IUT reactions; and it computes a verdict as soon as possible.

In order for the algorithm to work we need: the specification of the orchestrator by means of an $IOSTS$, the symbolic execution of the $IOSTS$, and the adaptation of the symbolic execution regarding the status of the Web services for testing orchestrators in context. Besides, an instance of the orchestrator's specification is also needed, i.e., the implementation under test. This IUT is described by means of the WS-BPEL specification. In this chapter we present the prototype that was developed to cover all of the previous points. External tools are used together with

the prototype for different purposes: first, a constraint solver is needed when generating concrete data from the symbolic execution in order to perform the tests on the IUT. Regarding the IUT, an editor of the WS-BPEL specification is needed, as well as a tool to deploy the WS-BPEL process instances.

The specification of the orchestrator is given by means of its WS-BPEL description. That is, WS-BPEL is used both to describe the behavioral specification of the orchestrator, and to define the instance of that specification, i.e., the IUT. Thus, the first thing we need to do is to represent the WS-BPEL specification by means of an $IOSTS$. In order to do that, we introduce the notion of *multiple communication channels* in Section 9.2. In Section 9.3, we present the implementation of the rule-based algorithm, which is the core module of the prototype. In Section 9.4, we present the constraint solver that we use, JaCoP[1], the JDeveloper WS-BPEL editor[2] used to design WS-BPEL processes, the WS-BPEL engine SOA[3] used deploy the WS-BPEL process instances, as well the rest of modules of the prototype and its general behavior. In Section 9.5, we present a complete example of usage of the prototype by making use of the Slot Machine example. We conclude this chapter with Section 9.6.

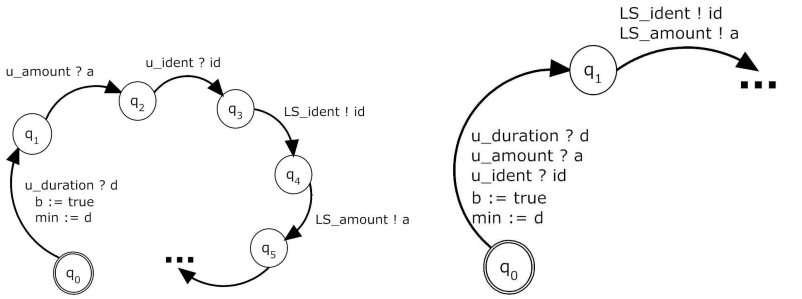
9.2 Multiple Communication Channels

We introduce the notion of *multiple communication channels* in order to better deal with the exchange of values defined in WS-BPEL, which can be composed of different parts. In WS-BPEL it is possible to send *message structures* in the <*receive*>, <*reply*> and <*invoke*> activities. However, with our $IOSTS$, we do not have the notion of an structured message, therefore, we use the multiple communication channels notion to send and receive multiple values in one single transition. A similar notion of multiple value exchanges in a single transition has been already used with Symbolic Transition Systems [Frantzen 2006b].

In fact, the usual presentation of sequences of the form $q \xrightarrow{c?x} q' \xrightarrow{d?y} q''$ presents the disadvantage of possibly allowing internal actions between the transitions, interfering, by doing so, with the test process (for instance, in the sequence previously given, quiescence could be allowed in transition $q' \xrightarrow{d?y} q''$). Thus the need to introduce multiple communication channels, which can be thought also as a sequence of transitions that have an atomic behavior: if one transition is executed, all the transitions (involved with the structured message) are executed. Then, the sequence of transitions $q \xrightarrow{c?x} q' \xrightarrow{d?y} q''$ becomes just $q \xrightarrow[d?y]{c?x} q''$. By doing this, first, we better represent the way WS-BPEL manipulate data, and second, it is a notation easy to adapt to the symbolic execution. Now, this notion of multiple communication channels could have been introduced before (in the definition of

[1] http://jacop.osolpro.com/
[2] http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html
[3] http://www.oracle.com/us/technologies/soa/index.html

communication actions, for instance), but in order not to grow heavy the notations, we preferred to keep the usual single emission/reception representation. Here, at the prototype level, and in order to better work together with IUTs, we introduce it in a natural way.



(a) Single channels.

(b) Multiple communication channels.

Figure 9.1: Notion of multiple communication channels for $IOSTS$s.

For the case of $IOSTS$s, the notion of multiple communication channels is represented in Figure 9.1. Figure 9.1(a) depicts how we would represent the exchange of messages composed of different fields in our approach. Figure 9.1(b) depicts how we actually represent the exchange of composed messages in the prototype. This simple idea is used for the prototype to work correctly together with the WS-BPEL process instances, in such a way that every time a structured message is sent or received, this can be modeled as a single transition.

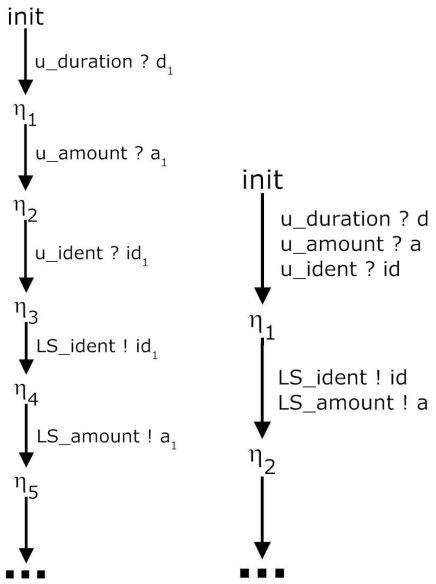
Figure 9.2(a) depicts the symbolic execution of the $IOSTS$ of Figure 9.1(a). Figure 9.2(b) depicts the symbolic execution of the $IOSTS$ with multiple communication channels of Figure 9.1(b). The only subtlety is then to ensure that the symbolic execution of the transition depicted in Figure 9.1(b) does not introduce any interference between the different receptions.

Thus, the notion of multiple communication channels allows us to model the specification of the orchestrators in a more structurally brief way, as well as facilitating the test process.

9.3 Implementation of the Rule-based Algorithm

In this section we present the pseudo code of the implementation of the rule-based algorithm introduced in Section 7.4. This implementation is the most important module of the prototype and it was implemented in the Java programming language.

Algorithm 3 presents the pseudo code of the implementation, which follows the intuition of the **ioco** conformance relation. Its inputs are: the specification of the orchestrator in context under test $Obs(\mathbb{G})$, a test purpose $\mathcal{TP}$, and a parameter *someTime* representing an amount of time that the test algorithm waits for an



(a) Single channels. (b) Multiple communication channels.

Figure 9.2: Multiple communication channels in symbolic executions.

observation before trying to send an stimulus. If this parameter is small, the stimuli are prioritized over observations and vice-versa. Regarding the observations on the IUT, the queue *Observations* is used to store them[4].

The function *Update(Obs*($\mathbb{G}$)*,NextStimulus,SC,Next,Skip,Pass)* updates the sets *Next, Skip* and *Pass* according to the set of contexts SC (which is also updated) and to the event *NextStimulus*, which is either an element of the queue *Observations* or an input computed by the *Update()* function and stored in $NextStimulus.ev()$[5] (the sets $Next$, $Skip$, and $Pass$ are updated according to, respectively, Definitions 7.4.1, 7.4.2, and 7.4.3). The behavior of the *Update()* function is the following one: for each child of each symbolic state occurring in some context of the current set of contexts (SC), if the the transition of $Obs(\mathbb{G})$ leading to the child state is labeled by an action that can be synchronized with the computed event (observation or stimulus), then a new context is computed accordingly and is added to the set *Next* (if the constraint induced by the sequence of actions is compatible with the path condition of the child state). If the child state is also in $\mathcal{TP}$, and $targetCond$ is satisfiable (according to Definition 7.3.3, $targetCond$ is the logical disjunction of the path conditions of all the symbolic states in $\mathcal{TP}$), then it is added to the set

[4]We assume that the queue cannot receive an infinite number of events, this means that the IUT is supposed to be strongly responsive.

[5]Note that such an input is only used if *Observations* is empty at the next step and is useless otherwise.

Skip. If the child node is a *target* state of $\mathcal{TP}$, then it is added to the set *Pass*. Finally, a new stimulus is computed in such a way that, if possible, the IUT remains in the path of $\mathcal{TP}$, and is stored in $NextStimulus.ev()$ (the stimulus is computed by making use of $Obs(\mathbb{G})$ and $\mathcal{TP}$).

The behavior of the test algorithm can be seen as traversing simultaneously the two trees $Obs(\mathbb{G})$ and $\mathcal{TP}$, by stimulating and observing the IUT, so that a verdict can be emitted as soon as possible according to the fact that the observed behavior of the IUT does or does not belong to both trees and is or not in the path of $\mathcal{TP}$. Rules of Section 7.4 are applied and, as long as *Result* remains a set of contexts (**Rule 1** for $obs(ev)$ and **Rule 6** for $stim(ev)$), the process continues. The algorithm ends when a verdict is emitted.

Algorithm 2: Pseudo-code of the Java-based prototype of the rule-based algorithm of test case generation.

```
   Data: Obs(𝔾), Test Purpose TP ≠ Root(Obs(𝔾)), someTime
   Result: One of the verdicts described in Sec.7.4
 1: begin
 2:     Init(Obs(𝔾), SC, Next, Skip, Pass)
 3:     Update(Obs(𝔾), NextStimulus.ev(), SC, Next, Skip, Pass)
 4:     while (true) do
 5:         timer ← reset()
 6:         while (Observations ≠ empty) do
 7:             ev ← Observations.getFirst()[a]
 8:             timer ← reset()
 9:             Update(Obs(𝔾), ev, SC, Next, Skip, Pass)
10:             if (Next = ∅ && ev.sender() ∉ WS) then
11:                 return FAIL, Report, ev                          /* Rule 2 */
12:             if (Next = ∅ && ev.sender() ∈ WS) then
13:                 return WS.HYP.FAIL, Report, ev                   /* Rule 2.WS */
14:             if (Next ≠ ∅ && Skip = ∅ && ev.sender() ∉ WS) then
15:                 return INCONC, Report                            /* Rule 3 */
16:             if (Next ≠ ∅ && Skip = ∅ && ev.sender() ∈ WS) then
17:                 return WS.INCONC, Report                         /* Rule 3.WS */
18:             if (Pass ≠ ∅ && Pass = Next) then
19:                 return PASS, Report                              /* Rule 4 */
20:             if (Pass ≠ ∅ && Pass ≠ Next) then
21:                 return WeakPASS, Report                          /* Rule 5 */
22:             wait(someTime)                  /* Prioritize obs. vrs. stim. */;
23:         if (timer < Timeout && NextStimulus.ev() ≠ null) then
24:             IUT.stimulate(NextStimulus.ev())                     /* Rule 6 */
25:             timer ← reset()
26:             Update(Obs(𝔾), NextStimulus.ev(), SC, Next, Skip, Pass)
27:         else if (timer > Timeout) then
                /* Timeout reached: Quiescent state                  */
28:             NextStimulus.ev() ← Quiescence
29:             IUT.stimulate(NextStimulus.ev())
30:             Update(Obs(𝔾), NextStimulus.ev(), SC, Next, Skip, Pass)
31: end
```

[a] **Rule 1** is implicitly applied for each observation every time that a verdict is not reached: the algorithm continues processing the queue.

According to the pseudo-code of Algorithm 3, after initialization in lines 2 and 3,

a *while* loop begins and will finish until a verdict is emitted (line 4). After initializing the timer, the first step of the algorithm consists in processing the observations sent from the IUT to the environment (the tester, or more specifically, the test harness) and stored in the queue *Observations* (line 6). Each observation of the queue is examined: the time is reset, sets *SC, Next, Skip* and *Pass* are updated (lines 7 to 21), and the rules described in Section 7.4 are applied, showing how the verdicts are produced. Besides, jointly with the verdict, a *Report* is provided. This report consists of the trace of the symbolic state(s) present in *SC* when the verdict is emitted. For the $WS.HYP.FAIL$ and $FAIL$ verdicts, the observation that caused the verdict is also given. If no verdict is found, the algorithm waits for *someTime* amount of time before trying to read again from the queue *Observations* (line 22). When there are no more observations in the queue then, if the *timer* has not reached *Timeout* and if there is a stimulus capable to guide the IUT to a target state, then this stimulus is sent, the timer is reset and the sets and next stimulus are updated (lines 23 to 26). Finally, if there is not a stimulus, we wait for *Timeout* amount of time to state that the IUT is in a quiescent state. If this happens, quiescence is *emitted* and the sets are updated according with the observation of the quiescent situation (lines 27 to 30).

9.4 Technical Aspects and Instrumentation of the Prototype

In order to test a WS-BPEL process instance, we use external tools to deal with the constraint solving of symbolic conditions in $Obs(\mathbb{G})$ (so we can send concrete data to the IUT), as well as to handle the edition and deployment of the implementation of the orchestrators as WS-BPEL process instances. In this section we present the external tools that are used together with the prototype as well as the rest of the prototype's modules that were developed.

9.4.1 External Tools

9.4.1.1 Constraint Solver: JaCoP

The symbolic execution presents all the possible behaviors of the implementation of an orchestrator in a symbolic way. Thus, if we want to *execute* one of those behaviors in the implementation, it suffices to take the path of the symbolic execution representing the desired behavior, *interpret it* and send the concrete data to the IUT. In order to interpret the symbols in the symbolic execution tree, we use a *constraint solver* [Hoffman 2005]. The purpose of a constraint solver is, as its name suggests, to propose solutions for a given constraint, in our case it is used to propose real data for the symbols of the symbolic execution, provided with a formula considered as a constraint. Those data has to satisfy the conditions over the symbols.

For instance, suppose that we have the following symbolic transition:

$$((\varepsilon, q_0, a \rightarrow a_0, true)\ , c?a,\ (c?a, q_1, a \rightarrow a_1, (a_1 * 3) > (a_1 - 3)))$$

This transition is only fired if the condition $(a_1 * 3) > (a_1 - 3)$ is satisfiable. Thus, the tester needs to find *some* values that make the constraint satisfiable in order to send the solution through the channel c to the IUT. In this case, the tester has to solve the condition $(a_1 * 3) > (a_1 - 3)$ and send the corresponding value of a. This is a trivial case and one can compute the value of a easily (for instance, $a = 4$). However, as we move forward in the symbolic execution tree, the path condition of the symbolic states gets more and more complex. For this reason, as it is classical, we use an external constraint solver.

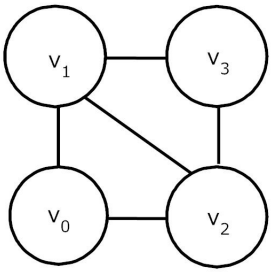


Figure 9.3: Constraint example for JaCoP.

In our case, the chosen constraint solver is JaCoP, which stands for *Java Constraint Programming Solver*. The main reason we chose JaCoP is because it is open source and is developed in Java. JaCoP admits only integer and boolean variables, so when using it to solve constraints over enumerated types or strings, messages are mapped to integer numbers. Figure 9.4 shows an example of JaCoP to solve the following constraint: there is a graph with 4 nodes, labeled v_1, v_2, v_3, and v_4, as depicted in Figure 9.3. The condition is to find a color (number) for each node in such a way that there are no two adjacent nodes sharing the same color.

As we notice in Figure 9.4, values are initialized and one variable is created for each node in the graph ($v[0]$ for node v_0, and so on, lines 8-13). Then, the conditions are set in lines 15 to 19, and finally JaCoP is requested to find a solution (lines 21 to 28): if it finds one, it gives the value for each variable, and if it does not, the program prints **** No*. In this case, the solution is fond and the result is printed out: *Solution: v0=1, v1=2, v2=3, v3=1.*

9.4.1.2 WS-BPEL Editor: JDeveloper

In order to design the WS-BPEL processes we use the Oracle's JDeveloper editor, together with its corresponding plugin for WS-BPEL. JDeveloper is a free integrated development environment that simplifies the development of Java-based SOA applications and user interfaces. The WS-BPEL editor allows to design the WS-BPEL process in both graphical and textual ways, as shown in Figure 9.5. One of the main reasons we chose JDeveloper is because it is free and can be easily integrated with Oracle's SOA.

```
1  import JaCoP.core.*;
2  import JaCoP.constraints.XneqY;
3  import JaCoP.search.*;
4  import java.util.*;

5  public class Main {

6      static Main m = new Main ();

7      public static void main (String[] args) {
8          Store store = new Store(); // define FD store
9          int size = 4;
10         // define variables
11         Variable[] v = new Variable[size];
12         for (int i = 0; i < size; i++)
13             v[i] = new Variable(store, "v" + i, 1, size);
14         // define constraints
15         store.impose( new XneqY(v[0], v[1]) );
16         store.impose( new XneqY(v[0], v[2]) );
17         store.impose( new XneqY(v[1], v[2]) );
18         store.impose( new XneqY(v[1], v[3]) );
19         store.impose( new XneqY(v[2], v[3]) );

20         // search for a solution and print results
21         Search label = new DepthFirstSearch();
22         SelectChoicePoint select = new InputOrderSelect(store, v, new IndomainMin());
23         boolean result = label.labeling(store, select);

24         if ( result )
25             System.out.println("Solution: " + v[0]+", "+v[1] +", "+v[2] +", "+v[3]);
26         else
27             System.out.println("*** No");
28     }
29 }
```

Figure 9.4: Example of JaCoP for solving constraints.

9.4.1.3 WS-BPEL Engine: Oracle's SOA

In order to deploy the WS-BPEL process instances, we chose Oracle's SOA. Among the tools we tried, Oracle's SOA was the one that better fitted to our needs. It is easy to install and user friendly. Besides, it offers the possibility to simulate some of the message exchanges, to introduce human tasks and to audit all the communications. These characteristics are useful because we can simulate the Web services when needed, and find out where the error was when there is failure. For instance, if the Web service is controllable, we can replace it by a human task. Figure 9.6 shows the interface of Oracle's SOA.

9.4.2 Rest of Modules and Behavior of the Prototype

In order to test a WS-BPEL process implementation, the prototype performs the following tasks (as depicted in Figure 9.7):

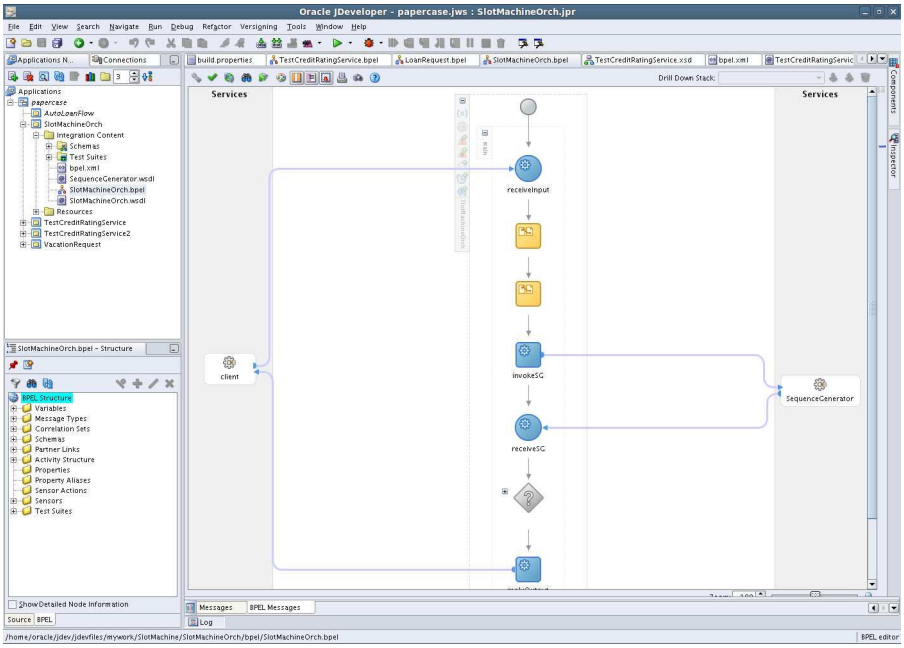


Figure 9.5: JDeveloper BPEL plugin.

1. It receives as input the specification of the WS-BPEL process instance by means of an $IOSTS$, $\mathbb{O}rch$. The first module of the prototype is then the one that allows the tester to manually enter this $IOSTS$. As explained in Section 2.5, we take as inspiration the the work of [Bentakouk 2009] to represent WS-BPEL as $IOSTS$s.

2. Then, another module symbolically executes $\mathbb{O}rch$ and generates $SE(\mathbb{O}rch)$. We stop the generation of the tree when we reach a given (variable) depth or when we visit the root state for the second time, as explained in Section 6.3. After that, this module asks the tester for the status of the different communication channels (controllable, observable, or hidden) and applies the transformations described in Chapter 7 in order to obtain $Obs(\mathbb{O}rch)$.

3. Finally, the module introduced in Section 9.3 firsts asks the tester for the test purposes and then executes the Algorithm 3 presented before, and each time a constraint has to be solved, JaCoP is invoked; if the communication is supposed to happen with a controllable Web service, we simulate the Web service by a SOA's human task. If the communication is supposed to happen with a hidden Web service, we emulate the Web service which becomes an embedded part of the IUT. If the communication is supposed to happen with an observable Web service, then the inputs sent by the Web services are just given as observations to the test algorithm. Also, quiescence situations are manually indicated by the tester.

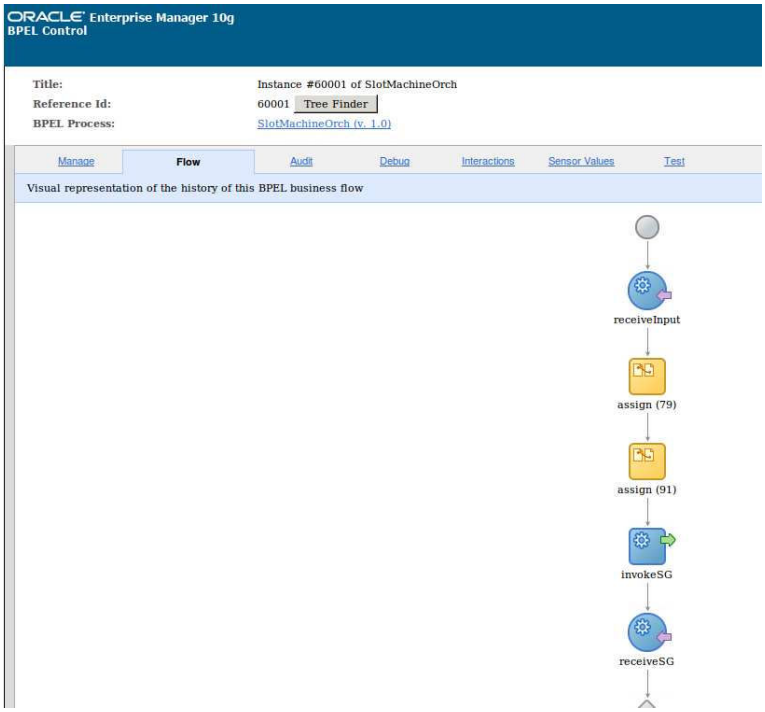


Figure 9.6: Oracle's SOA BPEL console.

9.4.2.1 Technical Details of the Prototype

The modules of the prototype make a total of 3,794 lines of code in Java (according to the Ubuntu package sloccount[6]). The execution time depends on the situation of the communication channels and the interaction with the tester. The time for solving a condition increases according to the domain of the variables and to the number of constraints in the condition, however, the prototype was not submitted to any performance requirement. The use case has been validated manually.

9.5 Usage of the Prototype: A Complete Example

In this section we show the complete process when testing an orchestrator in context. We take the Slot Machine example and we show one by one the steps to follow when using the prototype.

1. First, we consider the specification of the orchestrator given by means of a WS-BPEL description. For the Slot Machine example, Figure 9.8 shows a simplified version of its WS-BPEL description. We can notice that there are two partnerLinks (lines 2-5), one to communicate with the user and the other one to

[6]http://packages.ubuntu.com/dapper/sloccount

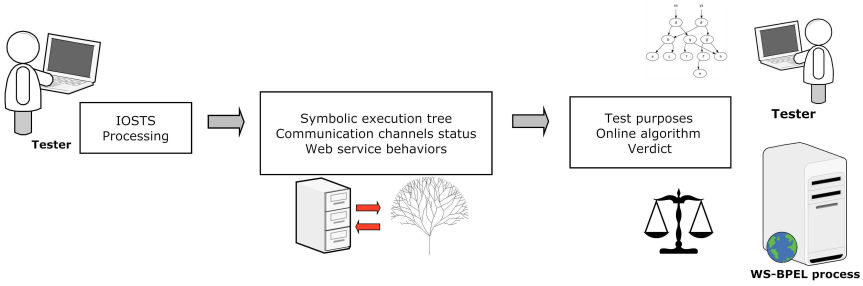


Figure 9.7: Modules of the prototype.

communicate with the Sequence Generator service, there is also the definition of the variables that are used in the business process execution (lines 6-12). Finally, the behavior of the orchestrator (slot machine's interface) is given (lines 13-52): first, the user invokes the orchestrator by sending it a token (line 14); then, the variables are initialized according to the token (lines 15-26) and the seed is sent to SG (line 27). The generated user's sequence is received from SG (line 28) and is compared with the computed winner sequence. If they are the same, the final message to be sent to the user is set to 'win' (lines 30-39) and to '$false$' otherwise (lines 40-50). The process ends by sending the final message to the user (line 51).

2. The next step is to represent the WS-BPEL description of Figure 9.8 by an $IOSTS$, denoted $\mathbb{O}rch$. Then, its text version is given as an input file to the prototype (see Figure A.1 in Appendix A). Its manual representation is (re)depicted in Figure 9.9. Taking as reference Figure 9.8 we can notice that the attribute variables and communication channels correspond to the ones used in the WS-BPEL description. For instance, the WS-BPEL variable $token$ is received through the partnerLink $user$ by invoking the operation $start$; thus, the name u_start is used in $\mathbb{O}rch$ to represent it. Also, the flow of the WS-BPEL business process is naturally represented in $\mathbb{O}rch$.

3. Once $\mathbb{O}rch$ has been given as an input, the prototype is compiled and executed.

4. The first output is the symbolic execution $SE(\mathbb{O}rch)$ in graphical and text mode, generated as defined in Section 6.3 (Definition 6.3.6) and depicted in Figure A.2 in Appendix A. The symbolic execution tree generation stops according to the criterion *root re-visited or depth reached*, that is, when a cycle of the slot machine is completed and the message is sent to the user. As for $\mathbb{O}rch$, $SE(\mathbb{O}rch)$ has been already introduced, but we depict it here again in Figure 9.10 to make the reading easier.

5. The second output is the full quiescence enrichment of $SE(\mathbb{O}rch)$, both in graphic and text mode, as depicted in Figure A.3 in Appendix A. This quiescence enrichment is done according to Definition 7.2.1 in Section 7.2. Its manual version is

```
 1 <process name="SlotMachineOrch">
 2     <partnerLinks>
 3         <partnerLink name="user"/>
 4         <partnerLink name="SequenceGenerator"/>
 5     </partnerLinks>
 6     <variables>
 7         <variable name="token"/>
 8         <variable name="seed"/>
 9         <variable name="sequence"/>
10         <variable name="winnersequence"/>
11         <variable name="finalMessage"/>
12     </variables>
13     <sequence name="main">
14         <receive name="receiveInput" partnerLink="user" variable="token" operation="start"/>
15         <assign>
16             <copy>
17                 <from expression="random(token)"/>
18                 <to variable="seed"/>
19             </copy>
20         </assign>
21         <assign>
22             <copy>
23                 <from expression="random('seed')"/>
24                 <to variable="winnersequence"/>
25             </copy>
26         </assign>
27         <invoke name="invokeSG" partnerLink="SequenceGenerator" inputVariable="seed"
   operation="generate"/>
28         <receive name="receiveSG" partnerLink="SequenceGenerator" variable="sequence"
   operation="sequence"/>
29         <switch>
30             <case condition="getVariableData('sequence') = getVariableData('winnersequence')" >
31                 <sequence>
32                     <assign>
33                         <copy>
34                             <from expression="string('win')"/>
35                             <to variable="finalMessage"/>
36                         </copy>
37                     </assign>
38                 </sequence>
39             </case>
40             <otherwise>
41                 <sequence>
42                     <assign>
43                         <copy>
44                             <from expression="string('lose')"/>
45                             <to variable="finalMessage"/>
46                         </copy>
47                     </assign>
48                 </sequence>
49             </otherwise>
50         </switch>
51         <invoke partnerLink="user" inputVariable="finalMessage" operation="screen"/>
52     </sequence>
53 </process>
```

Figure 9.8: WS-BPEL code for the Slot Machine example.

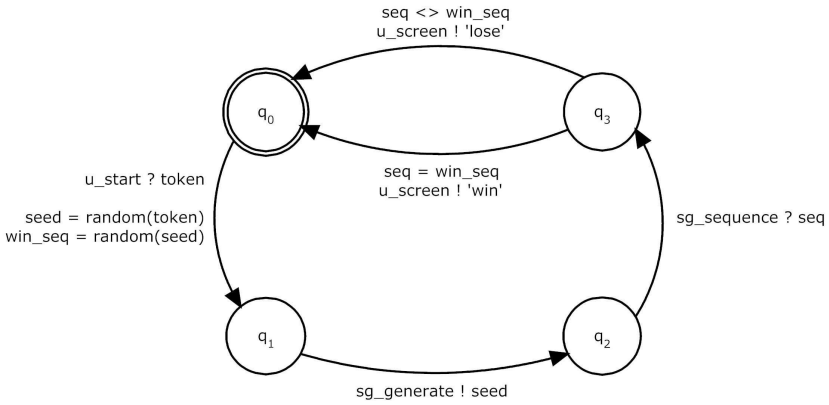


Figure 9.9: $\mathbb{O}rch$ for the slot machine example.

depicted in Figure 9.11. For the moment, in the prototype, for each quiescence state that is generated, we stop the symbolic tree.

6. The next step consists in preparing the partial specification of $\mathbb{O}rch$, $Obs(\mathbb{O}rch)$, according to the status of the communication channels of the Sequence Generator service SG. We start by showing the hidden case. After indicating the hidden status of the communication channels to the prototype, the modified symbolic execution $HO(\mathbb{O}rch)$ (obtained by applying Definition 7.2.3 in Section 7.2 to the WS transformation of $SE(\mathbb{O}rch)_\delta$ of Figure 9.11) is calculated and given in its text an graphical versions. The resulting structure computed by the prototype is depicted in Figure A.5 in Appendix A. A more clear, manual version, is depicted in Figure 9.12. There are two transitions denoting interactions trough hidden communication channels, $sg\_generate$ and $sg\_sequence$. Since observations are no possible in those channels, they are treated as internal actions and the communication actions are replaced by τ.

7. The next output is the τ-reduction of $HO(\mathbb{O}rch)$, both in text an graphical mode, as depicted in Figure A.6 in Appendix A. This reduction is done according to Definition 6.3.10 in Section 6.3, and which basically *removes* the transitions labeled with τ. Its manual representation is depicted in Figure 9.12. In fact, since the channels used by the slot machine's interface to interact with SG are hidden, from the tester's perspective the system just receives a *token* and gives the answer '*win*' or '*lose*'. The resulting structure is $Obs(\mathbb{O}rch)$, which is the specification that we will consider as a reference to test the WS-BPEL implementation under test IUT of the Slot Machine.

8. The following step consists in asking the tester to enter the target state(s). Such states are identified by the state's label. Each symbolic state in the path of the target state is also marked, computing $\mathcal{TP}$ at the same time. This step is depicted in Figure A.7 in Appendix A.

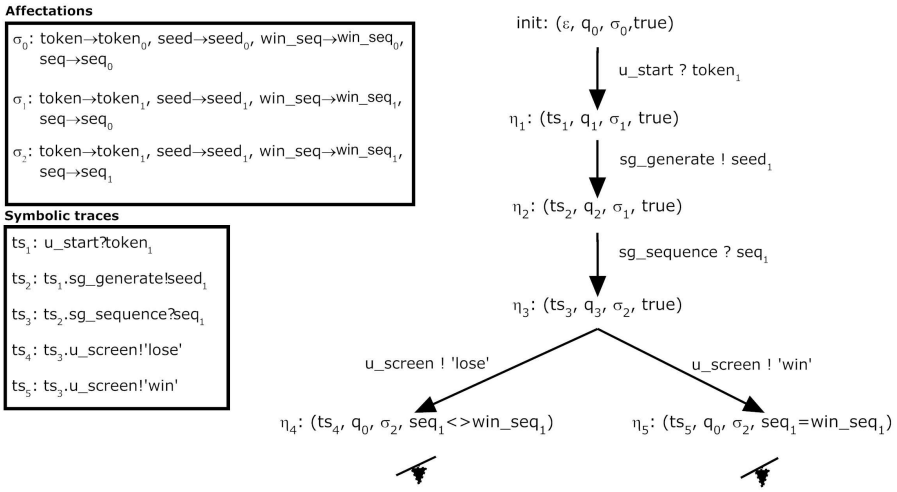


Figure 9.10: $SE(\mathbb{O}rch)$ of the Slot Machine example.

9. Taking $Obs(\mathbb{O}rch)$ and $\mathcal{TP}$ as inputs (assuming that the target state is η_5 and that the *someTtime* parameter discussed in Section 9.3 is set to 5 seconds), the online test algorithm is executed by interacting with the tester, requesting him or her to enter the respective observations and stimuli observed and sent to the IUT. The algorithm stops until a verdict is given. One part of the execution of the test algorithm of the prototype for the Slot Machine example is given in Figure A.8 in Appendix A, and the emission of a verdict is given in Figure A.9 in Appendix A.

9.5.1 Example of Two Verdicts

Here, the $WS.INCONC$ verdict is depicted in Figure 9.14: the value 4 (for the symbolic variable $token_1$) is sent to the IUT through the communication channel (respectively partherLink) u_start (respectively *user*, operation *start*) and the observed event is a quiescence state (in this example, we assumed that the $Timeout$ parameter is 30 seconds, and SG does not answer in that time). Since we know the behavior of the $random()$ function, we also know the values of the *seed* and the win_seq variables. The quiescence situation is a valid behavior of $Obs(\mathbb{O}rch)$, however, it is due to the lack of reaction of SG and this observation is not in the path of $\mathcal{TP}$; therefore the verdict $WS.INCONC$ is given, together with a *Report* that contains the history of the communications (internal and external) as well as the values of the variables before emitting the verdict.

Moreover, if we use the simplified notation $SC \xrightarrow[action]{Rule} SC'$ to denote the execution of rule *Rule* due to the action *action* (which can be an observation or a stimulus), such that applying the rule *Rule* makes the set of contexts to change

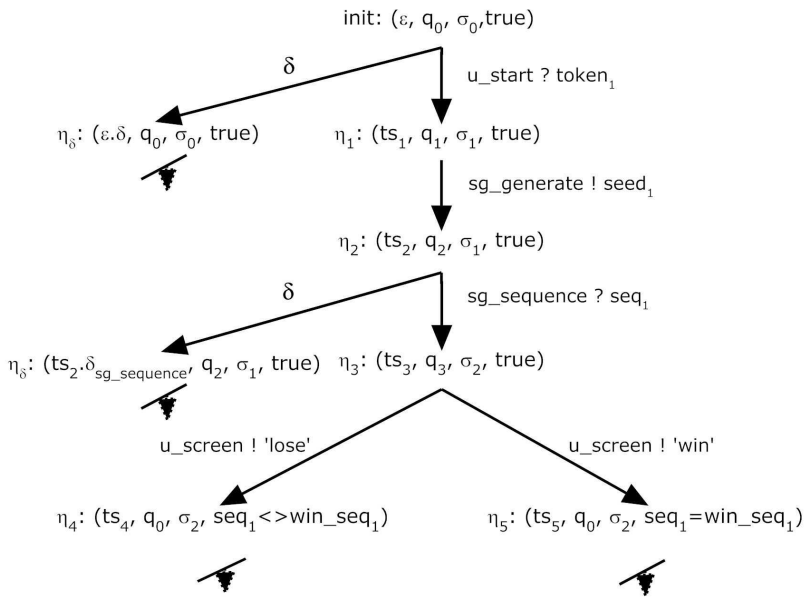


Figure 9.11: Full quiescence $SE(\mathbb{O}rch)_\delta$ for the Slot Machine example.

from SC to SC', then, for this example, we have the following sequence of applications of rules:

$$\emptyset \xrightarrow[u\_start?4]{Rule\ 6} \{(\eta_1, true)\} \xrightarrow[\delta]{Rule\ 3.WS} WS.INCONC$$

In order to give an example of the verdict $WS.HYP.FAIL$, we need to modify the Slot Machine example with at least one more interaction with SG. Thus, we assume that the slot machine's interface SM after receiving the *token* through the communication channel u_start from the user, instead of directly requesting the sequence to SG, first asks for the amount of the prize (also to SG), and only after receiving the answer, SM asks for the sequence. Finally, is not reasonable to refer to $WS.HYP.FAIL$ if we cannot observe the communication channels of the Web service, so we assume that they are observable. The new $Obs(\mathbb{O}rch)$ for illustrating this verdict is depicted in Figure 9.15(a). Taking η_7 as the new target state (the equivalent of η_5 of Figure 9.13, i.e., the user wins the prize), Figure 9.15(b) depicts the $WS.HYP.FAIL$ verdict obtained because, after sending the value 4 over the channel u_start, the value 7 is observed in the communication channel $sg\_sequence$: this means that 7 is the value of the symbolic variable seq_1 instead of the one for $amount_1$ that should be observed over the communication channel sg_prize. This behavior is not expected by $Obs(\mathbb{O}rch)$; therefore the verdict $WS.HYP.FAIL$.

For this example, the sequence of applications of rules according to our algorithm

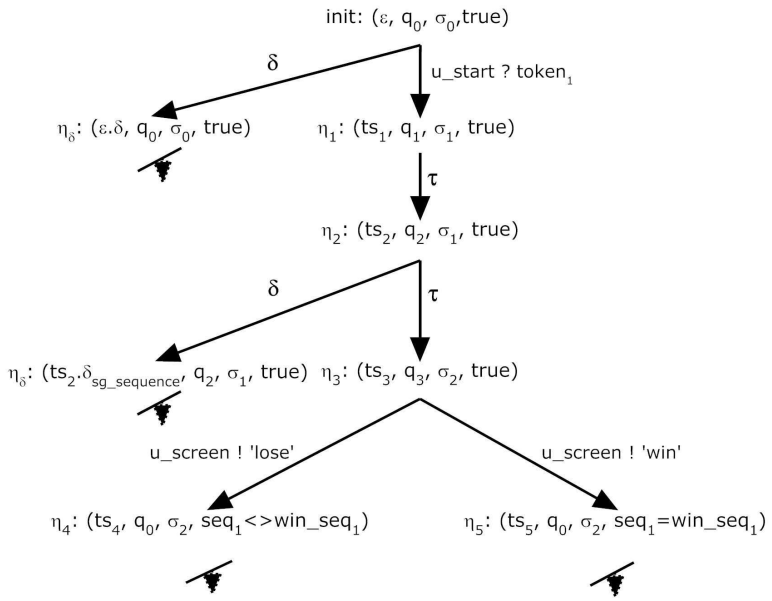


Figure 9.12: Hiding operator $HO(\mathbb{O}rch)$ of the Slot Machine example.

is:

$$\emptyset \xrightarrow[u\_start?4]{Rule\ \ 6} \{(\eta_1, true)\} \xrightarrow[sg\_prize!7]{Rule\ \ 2.WS} WS.HYP.FAIL$$

The controllable case is not shown since it corresponds to perform unit testing on $Obs(\mathbb{O}rch)$. Scenarios for the rest of verdicts are depicted in Appendix A.

9.6 Conclusion

In this chapter we have presented the technical details of the prototype for testing orchestrators in context, which implements the rule-based algorithm introduced in Chapter 7. However, besides the implementation of the rule-based algorithm, the $IOSTS$ representing the orchestrator's specification is also needed in order to symbolically execute it and generate the set of valid behaviors of the orchestrator in context, that is, taking the status of the communication channels into account. An instance of the orchestrator is also needed in order to apply our algorithm as well as a constraint solver to send concrete data to WS-BPEL instances. In this chapter we have presented all the modules of the prototype that were implemented in order to deal with all the previous requirements, as well as the external tools chosen to be used together with the prototype. Finally we have also shown how to use the prototype by means of a complete example.

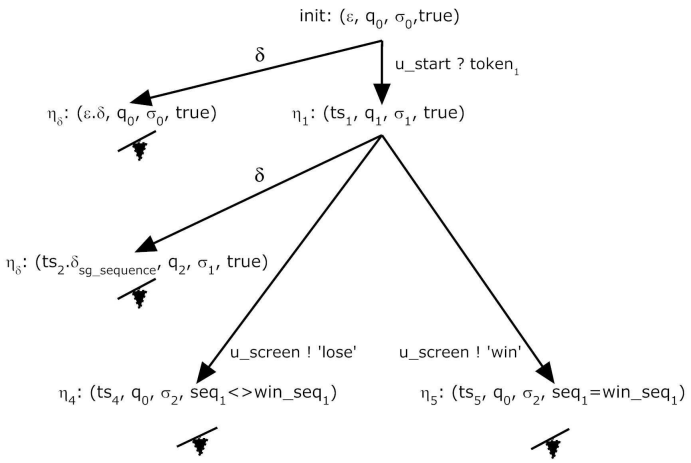


Figure 9.13: $Obs(\mathbb{O}rch)$ of the Slot Machine example.

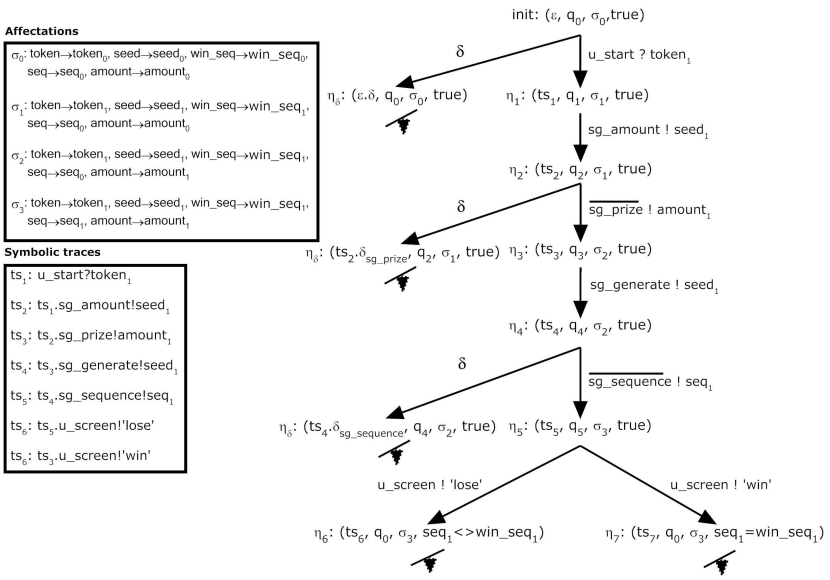
1. stimuli: u_start→4

2. obs: δ

3. Verdict: WS.INCONC, **{**u_start?token$_1$.sg_generate!seed$_1$,token=4^seed=2^win_seq=6^seq=seq$_0$**}**

Figure 9.14: $WS.INCONC$ verdict for the Slot Machine example with hidden communication channels.

At present time, elicitation of behaviors in order to test the compatibility of Web services (Chapter 8) has not been yet implemented in the prototype. However, this implementation is easy since, as we can deduce by Section 8.3, we just need to enrich $SE(\mathbb{O}rch)$ with Web service quiescence and then perform the projection and mirroring operations (Definitions 8.2.5, 8.2.3 and 8.2.4 in Section 8.2.3), and use it as the specification (which is not the specification of the Web service, but the one of the expected behaviors of $\mathbb{O}rch$) in order to test the Web service implementations.



(a) $Obs(\mathbb{O}rch)$ for the modified version of the Slot Machine example.

1. stimuli: u_start→4

2. obs: sg_sequence→7

3. Verdict: WS.HYP.FAIL, {u_start?token1.sg_amount!seed1,token=4^seed=2^win_seq=6^seq=seq0^amount=amount0}, sg_sequence!7

(b) $WS.HYP.FAIL$ verdict.

Figure 9.15: Modified Slot Machine example to illustrate the $WS.HYP.FAIL$ verdict.

Chapter 10

Conclusion and Future Works

An orchestrator is the main component in the type of systems known as orchestrations. In this type of systems, the orchestrator is the one who guides the entire process in a centralized way, whereas the user and Web services (or components) interact with the system via the orchestrator. In this thesis we have shown an approach to test orchestrators in their context of usage in such a way that, if an error is found in this testing phase, then, under certain hypotheses, the error indicates an unconformance of the orchestrators with respect to their specifications. We have also shown an approach to determine if Web services are or not compatible with a given orchestrator. For both testing approaches, we take into account only the specification of the orchestrator, and we ignore the ones of the rest of Web services involved in the orchestration.

Conclusions

In the first part of the thesis, we have started by setting the formal basis of our work, which takes as reference the **ioco** conformance relation applied to component-based systems. The **ioco** conformance relation basically states that a system under test conforms to its specification only if any observation in the system under test after any specified trace is also specified. By making use of the $IOLTS$s in order to model the specification of the orchestrators under text, we adapt it by a series of operations that reflect its situation of usage. One first contribution is precisely the explicit distinction of such situations. Since we test the orchestrator while interacting with the Web services, it may happen that the tester has complete control over the Web service (like, for instance, simulating it). We say in this case that the communication channels used to interact with the Web services (or, what is the same from this point of view, the Web services) are **controllable.** If the Web services are not accessible at all by the tester, we say that the communication channels are **hidden**. Finally, an intermediate case can occur: the Web service may not be controlled by the tester, but the communication channels are accessible in such a way that pieces information going through them can be observed. We say in this case that the communication channels are **observable.**

With the previous classification, we proceed to generate the partial specification (or specification in context) of the orchestrator. We build this partial specification by means of the following technical operations: full quiescence, to identify where the system is silent (due to the impossibility to move forward by itself or because

of the lack of reaction of the Web services); WS input transformation, to make a distinction between the inputs sent by the user to the orchestrator and those sent by the Web services over observable channels (since those inputs are actually observations); and the hiding operator, to make communication actions going through hidden channels to be treated as internal actions (since it is like that that they are perceived by the tester). By assuming that the system under test can be modeled as a special *IOLTS* (enriched by quiescence, denoting the inputs from the remote Web services through observable channels as special observations, without τ-transitions, and input-complete), we define a conformance relation in context, that we call **ioco**, and that can be used to test the conformance of orchestrators in context, that is, by taking into account the status of the communication channels used to interact with the Web services. Inspired by [van der Bijl 2003b], we define another variation of **ioco**, called **uioco**, and that, when there are hidden communication channels, does not take all the traces of the partial specification when reasoning about the conformance of the orchestrator in context: it takes into account only the fully specified traces (as opposed to **ioco**, that takes into account traces where there are underspecified inputs).

We finish this discussion by presenting two theorems (one based on **ioco** and the other one based on **uioco**) that provide an answer to the main concern of our work: if an error is detected while testing the orchestrator in context, the error is in fact due to the orchestrator. This result holds under the hypotheses that the system under test can be decomposed into the implementation of the orchestrator and the implementation of the remote Web services, and that communication between them is captured by the product of their models.

Then, we have shown how to elicit behaviors for testing the deadlock-free property of Web services interacting with the orchestrator (the deadlock-free property guarantees that a Web service does not lead the orchestration into a deadlock state). We do not consider the specification of the Web services and we elicit those behaviors from the orchestrator, that is, only the behaviors that are expected from the orchestrators are taken into account. By doing this, we reduce the state explosion problem and we give a solution to the common problem of not having the specifications of the Web services in hand for testing purposes. These behaviors are extracted from the traces of the *IOLTS* of the orchestrator's specification after some modifications: enrichment by Web service quiescence, which identifies under which situations the Web service can be quiescent without leading the orchestration into a deadlock state; and mirror and projection operations, which basically transform the behaviors from the point of view of the orchestrator to the point of view of the Web services. With these elicited behaviors, a given Web service can be tested by using the classical **ioco**, but in this case not in order to test the conformance of the Web services with respect to their specification, but with respect to the expected behaviors of the orchestrator (in other words, we test them *in order to verify* the deadlock-free property).

In the second part of the thesis, once we have grounded the theory of our approach, we move to the field of the symbolic techniques. $IOSTS$s are symbolic characterizations of $IOLTS$s and are thereon used to model the specifications of the orchestrators. Moreover, the semantics of an $IOSTS$ can be given in the form of an $IOLTS$, and therefore the results obtained in the first part of the thesis, also apply on them. We use the symbolic techniques because they are better fitted to model orchestrators, since they include more information about the modeled system, introducing firing conditions on the transitions and making explicit the relation between the data. Besides, the symbolic execution works very well in conjunction with the $IOSTS$s, and symbolic execution's resulting structures represent all the possible behaviors of the orchestrator in a symbolic way, reducing the state explosion problem. Symbolic executions are then submitted to a series of transformations, in a similar way that we did for the numerical case (working with $IOLTS$s) and based on the same hypothesis (classification of communication channels), but with the main difference that in this case we also apply the τ-reduction operation, which *removes* the internal actions from the specification. The final structure represents the possible behaviors of the orchestrator in context, and thus can be seen as the partial specification of the orchestrator in context. This structure is given as an input to our rule-based test algorithm, together with the desired behaviors to test in the system under test (test purposes) which are in fact subtrees of the partial specification. This algorithm is used to test the conformance of a system under test with respect to its specification and follows the idea of **ioco** (more specifically, of $\overline{\textbf{ioco}}$) by stimulating the system under test according to the test purposes extracted from the specification, and emitting a verdict according to the observations as soon as possible. More important, another contribution in this thesis is the introduction of two verdicts that are directly related to the activities of the Web services involved in the orchestration; Web services of whom we do not consider their specification. Those verdicts are: $WS.HYP.FAIL$, which means that an error was detected and that it is possibly due to a failure in a Web service (however, since we do not take into account the specification of the Web services, we cannot be sure —the only thing we know is that it did not answer according to what the orchestrator was expecting); and $WS.INCONC$, which indicates that the Web service reacted in such a way that it is not an observation that is part of the test purpose, but that it is still a valid observation according to the specification.

Then, we have defined a methodology to generate test cases in order to test Web services to determine if they are deadlock-free with respect to the orchestrator, meaning that they do not lead the orchestration into a deadlock state. The test cases are elicited from the specification of the orchestrator, and can be used together with our rule-based test algorithm. In this case, the system under test are the implementations of the Web services that interact with the orchestrator. Any $FAIL$ verdict would mean that, if the Web service under test is used to interact with the orchestrator, then a situation of deadlock can occur.

In the last chapter of this thesis we have presented another aspect of our contributions: the prototype tool which implements the rule-based testing algorithm described before as well as some other modules in order to test an instance of an orchestrator described in WS-BPEL. In order to do this, external tools are also used to design and deploy the WS-BPEL process instances and to solve the constraints of the symbolic execution trees. Thus, we have shown that our theoretical work can be applied by means of the symbolic execution techniques in order to test implementations of orchestrators in their context of usage. More important, we have shown that, since orchestrations are especial systems where there is a central component guiding the whole process, by taking into account only the specification of this central component, we can still give interesting results about its conformance: we can test the conformance of orchestrators while interacting with the Web services and generate test cases in order to determine if a given Web service can interact with the orchestrator without leading it into a deadlock state.

Perspectives

Regarding the perspectives of the work presented in this document, first, it would be interesting to take time delays into account. In order to do that, we have to take as a basis a conformance relation which also takes time into account (for instance, **tioco** [Schmaltz 2008]). Then, we could extend our results by taking time into account so we can take also into account the WS-BPEL activities that are related with timeouts. This is an important aspect of WS-BPEL and orchestrations in general, in the sense that it is common to define actions that have to be taken when a component does not react in a given period of time. From a certain perspective, our notion of quiescent situations in the system aims at dealing with those situations where a Web service does not react, but not every time that this happens the orchestration has to go into a quiescent state: for instance, WS-BPEL provides ways to handle these situations (by means of event handlers), so the process execution can continue and different actions can be taken, like invoking another Web service if the first one does not react on time.

There are also some other activities in WS-BPEL that would be interesting to take into account. Even if our purpose was not to focus on WS-BPEL, there are some notions like fault handling and error recovery that are common to every type of systems.

All the previous perspectives can also be implemented in the prototype tool. This prototype successfully implements our rule-based testing algorithm, but it would be interesting to improve it and automatize the testing activity. Currently, the interaction between the prototype and the WS-BPEL process instances is done by hand. Improving the technical aspect of the prototype by including time and automatizing the test harness is an interesting work to do. This automation can be achieved by giving as an input to the prototype the WSDL of the Web services that

interact with the orchestrator, and use automatically generated SOAP messages to communicate with them.

Finally, a more general perspective would be to apply our approach for the case of integration testing. The idea is to consider component-based systems with an interface, that is, the user interacts with the whole system by means of the component acting as the interface *but* that does not guide the whole system. Then, we can consider the specification of the interface component and of *some* of the rest of components but not all of them: we want to avoid the state explosion problem. In fact, taking into account the specification of the *hidden* components would be a way to start (in order to avoid the number of inconclusive verdicts related to the remote components —Web services, in our approach), adding more and more specifications until we reach a certain limit (for instance, a given size of the model). Then, we could apply the techniques presented in this thesis in order to test the conformance of the components whose specification is being considered. Our elicitation techniques could also be applied in order to test if the rest of components are compatible (not leading the whole process into a deadlock state) with the ones that are under examination.

Appendix A

Appendix

A.1 Outputs and User Interface of the Prototype

1. *IOSTS*

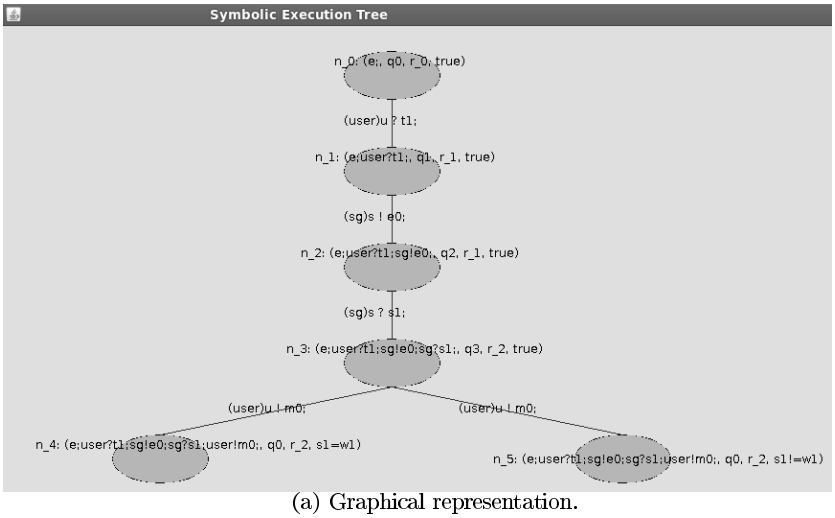
   As we can notice in Figure A.1, the constraints are given in a special format so then they can be parsed and solved by JaCoP. Also, since JaCoP only deals with integer an boolean variables, we map the '*win*' and '*lose*' messages to integer constants. Besides, we have chosen to deal with the boolean variables also as integers, 1 for $true$ and 0 for $false$.

```
IOSTS iosts = new IOSTS(rootState);

iosts.setVariables(new LinkedList<String>());

// initialize variables
iosts.addVariable("t");
iosts.addVariable("s");
iosts.addVariable("w");
iosts.addVariable("e");
iosts.addVariable("m");

// q1
LinkedList<Transition> q1childTrs = new LinkedList<Transition>();
State q1State = new State("q1", q1childTrs);

// q0 -> q1
LinkedList<String> q0q1comAction0 = new LinkedList<String>();
q0q1comAction0.add("u;?;t:v");
LinkedList<String> q0q1Affectation0 = new LinkedList<String>();
q0q1Affectation0.add("w;6:c,-,t:v");
Transition q0trans0 = new Transition("true",q0q1comAction0,q0q1Affectation0,"user","user",q1State
rootState.addChildTransition(q0trans0);

// q2
LinkedList<Transition> q2childTrs = new LinkedList<Transition>();
State q2State = new State("q2", q2childTrs);

// q1 -> q2
LinkedList<String> q1q2comAction0 = new LinkedList<String>();
q1q2comAction0.add("s;!;e:v");
Transition q1trans0 = new Transition("true",q1q2comAction0,null,"sg","ws",q2State,false,"!");
q1State.addChildTransition(q1trans0);

// q3
LinkedList<Transition> q3childTrs = new LinkedList<Transition>();
State q3State = new State("q3", q3childTrs);

// q2 -> q3
LinkedList<String> q2q3comAction0 = new LinkedList<String>();
q2q3comAction0.add("s;?;s:v");
Transition q2trans0 = new Transition("true",q2q3comAction0,null,"sg","ws",q3State,false,"?");
q2State.addChildTransition(q2trans0);

// q3 -> q0 two times
LinkedList<String> q3q0comAction0 = new LinkedList<String>();
q3q0comAction0.add("u;!;m:v");
Transition q3trans0 = new Transition("s:v;=;w:v",q3q0comAction0,null,"user","user",rootState,fals
q3State.addChildTransition(q3trans0);
```

Figure A.1: Prototype's text version of $\mathbb{O}rch$ of the Slot Machine example.

2. **Symbolic Execution**

Figure A.2 depicts the symbolic execution of the Slot Machine example as given by the prototype. The upper (resp.lower) image depicts its graphical (resp. textual) representation. Its manual, and more clear, version, is depicted in Figure 6.7. Important information associated to each state is printed in the graphical version, however, to have a more clear representation of this information, it is also printed out in the text version.



(a) Graphical representation.

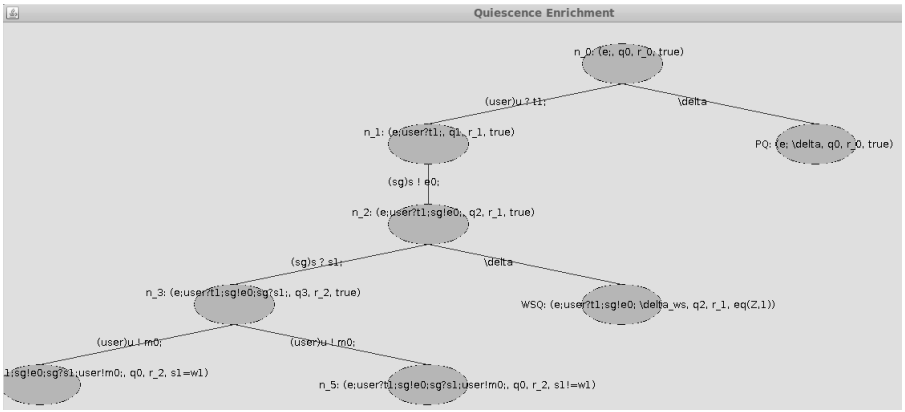
```
1 n_0, PARENT : nullpc: true, sa: tau
2  n_1, PARENT : SEtree@b6ece5pc: true, sa: (user)u ? t1;, channel type: user, chan
3   n_2, PARENT : SEtree@14693c7pc: true, sa: (sg)s ! e0;, channel type: ws, channe
4    n_3, PARENT : SEtree@3a6727pc: true, sa: (sg)s ? s1;, channel type: ws, channe
5     n_4, PARENT : SEtree@665753pc: s1=w1, sa: (user)u ! m0;, channel type: user,
6     n_5, PARENT : SEtree@665753pc: s1!=w1, sa: (user)u ! m0;, channel type: user,
```

(b) Text version.

Figure A.2: $SE(\mathbb{O}rch)$ for the Slot Machine example.

3. **Quiescence Enrichment**

Figure A.3 depicts the quiescence enrichment of $SE(\mathbb{O}rch)$ depicted in Figure A.2. As for the previous case, both graphical and text versions are shown. Quiescence is represented by the word *delta*. Thus, quiescence is allowed in two states of the symbolic execution: this situations represent the impossibility of the orchestrator (the slot machine's interface) to move forward by itself, since it is waiting for an input from the user or the Sequence Generator service.



(a) Graphical representation.

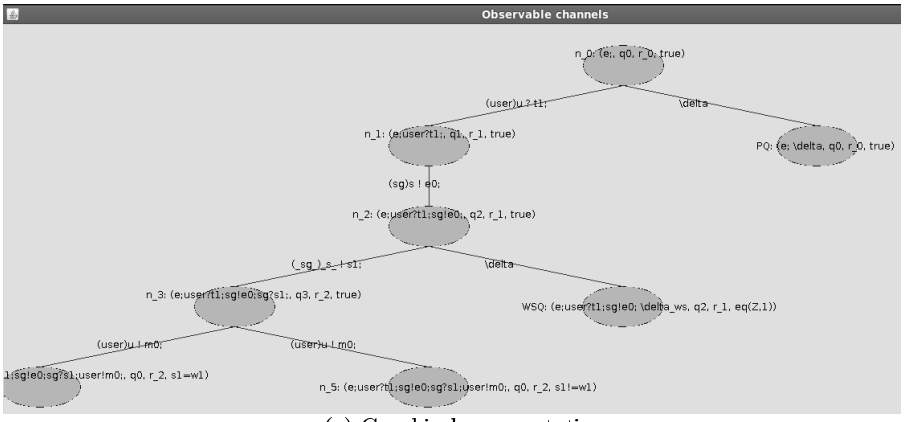
```
 Quiescence enrichment finished.
 Tree after Quiescence Enrichment:
1 n_0, PARENT : nullpc: true, sa: tau
2  n_1, PARENT : SEtree@b6ece5pc: true, sa: (user)u ? t1;, channel type: user, channel status: con
3  PQ, PARENT : SEtree@b6ece5pc: true, sa: \delta, channel type: pq, channel status: con
4   n_2, PARENT : SEtree@14693c7pc: true, sa: (sg)s ! e0;, channel type: ws, channel status: con
5    n_3, PARENT : SEtree@3a6727pc: true, sa: (sg)s ? s1;, channel type: ws, channel status: con
6    WSQ, PARENT : SEtree@3a6727pc: eq(Z,1), sa: \delta, channel type: wsq, channel status: con
7     n_4, PARENT : SEtree@665753pc: s1=w1, sa: (user)u ! m0;, channel type: user, channel status: con
8     n_5, PARENT : SEtree@665753pc: s1!=w1, sa: (user)u ! m0;, channel type: user, channel status: con
```

(b) Text version.

Figure A.3: Full quiescence enrichment of Figure A.2.

4. **WS Input Transformation**

Figure A.4 depicts the WS input transformation of Figure A.3. In this case, the communication channels used by the slot machine's interface to interact with the Sequence Generator Web service are supposed to be observable. If an input is of the form $c?a$, where c is a communication channel used to interact with a Web service, then, after applying the WS input transformation of Definition 7.2.2, in the prototype it is represented by $\_c\_!a$.



(a) Graphical representation.

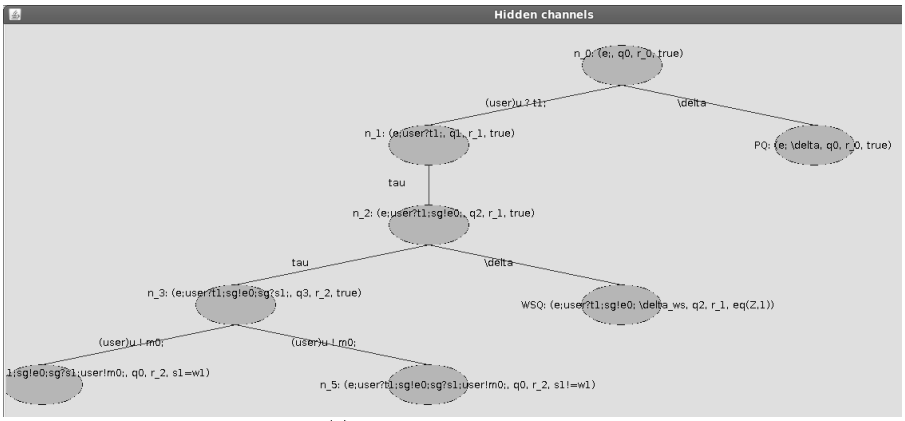
```
 Please enter the list of OBSERVABLE channels sepparated by colons:
sg
[sg]
Channel in transition (sg)s ! e0; marked as obs
Channel in transition (sg)s ? s1; marked as obs
Channel transformed into an observation in transition: (_sg_)s_ ! s1;
Tree after WS input transformation:

1 n_0, PARENT : nullpc: true, sa: tau
2  n_1, PARENT : SEtree@b6ece5pc: true, sa: (user)u ? t1;, channel type: user, channel status: con
3  PQ, PARENT : SEtree@b6ece5pc: true, sa: \delta, channel type: pq, channel status: con
4   n_2, PARENT : SEtree@14693c7pc: true, sa: (sg)s ! e0;, channel type: ws, channel status: obs
5    n_3, PARENT : SEtree@3a6727pc: true, sa: (_sg_)s_ ! s1;, channel type: ws, channel status: obs
6    WSQ, PARENT : SEtree@3a6727pc: eq(Z,1), sa: \delta, channel type: wsq, channel status: con
7     n_4, PARENT : SEtree@665753pc: s1=w1, sa: (user)u ! m0;, channel type: user, channel status: con
8     n_5, PARENT : SEtree@665753pc: s1!=w1, sa: (user)u ! m0;, channel type: user, channel status: con
Siblings: {0=1, 1=2, 2=1, 3=2, 4=2}
```

(b) Text version.

Figure A.4: Remote input transformation of Figure A.2.

5. **Hiding Operator**

The hiding operator of Figure A.3 is depicted by Figure A.5. In this case, the communication channels with the Sequence Generator Web service are supposed to be hidden. In the prototype, the internal action τ is represented by the word *tau*. Thus, the two transitions representing communications with the Sequence Generator service are labeled with *tau*.



(a) Graphical representation.

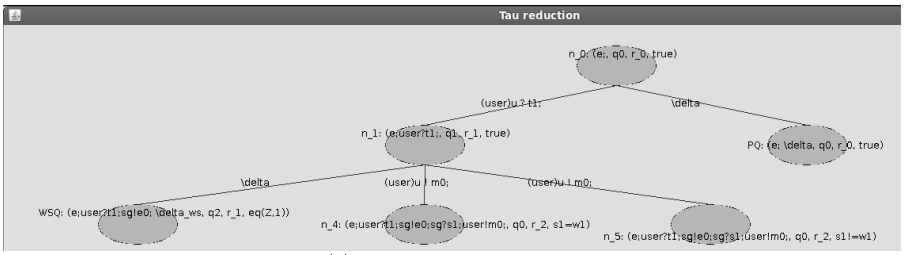
```
 Please enter the list of HIDDEN channels sepparated by colons:
sg
[sg]
Channel in transition (sg)s ! e0; marked as hid
Channel in transition (sg)s ? s1; marked as hid
Channel transformed into a hide operation in transition: tau
Channel transformed into a hide operation in transition: tau
Tree after Hiding Operation:

1 n_0, PARENT : nullpc: true, sa: tau
2  n_1, PARENT : SEtree@b6ece5pc: true, sa: (user)u ? t1;, channel type: user, channel status: con
3  PQ, PARENT : SEtree@b6ece5pc: true, sa: \delta, channel type: pq, channel status: con
4   n_2, PARENT : SEtree@14693c7pc: true, sa: tau, channel type: ws, channel status: hid
5    n_3, PARENT : SEtree@3a6727pc: true, sa: tau, channel type: ws, channel status: hid
6    WSQ, PARENT : SEtree@3a6727pc: eq(Z,1), sa: \delta, channel type: wsq, channel status: con
7     n_4, PARENT : SEtree@665753pc: s1=w1, sa: (user)u ! m0;, channel type: user, channel status: con
8     n_5, PARENT : SEtree@665753pc: s1!=w1, sa: (user)u ! m0;, channel type: user, channel status: con
Siblings: {0=1, 1=2, 2=1, 3=2, 4=2}
```

(b) Text version.

Figure A.5: $HO(\mathbb{O}rch)$ of Figure A.3.

6. τ-reduction

Figure A.6 depicts the τ-reduction applied to Figure A.5. Transitions with communication actions *tau* (τ) are just removed from the tree. The resulting structure is then the partial observation of the slot machine's interface ($Obs(\mathbb{G})$). This structure is the one that the prototype takes as a reference for selecting test purposes and for reasoning about the conformance of the IUT.



(a) Graphical representation.

```
Node removed from tree: n_2
Node removed from tree: n_3
Tree after Tau Reduction:

1 n_0, PARENT : nullpc: true, sa: tau
2  n_1, PARENT : SEtree@b6ece5pc: true, sa: (user)u ? t1;, channel type: user, channel status: con
3  PQ, PARENT : SEtree@b6ece5pc: true, sa: \delta, channel type: pq, channel status: con
4   WSQ, PARENT : SEtree@14693c7pc: eq(Z,1), sa: \delta, channel type: wsq, channel status: con
5   n_4, PARENT : SEtree@14693c7pc: s1=w1, sa: (user)u ! m0;, channel type: user, channel status: con
6   n_5, PARENT : SEtree@14693c7pc: s1!=w1, sa: (user)u ! m0;, channel type: user, channel status: con
Siblings: {0=1, 1=2, 2=3}
```

(b) Text version.

Figure A.6: τ-reduction of Figure A.5.

7. **Choosing Target State**

Target states are chosen by typing their labels. When searching for the target state in the tree (if it is found) all the intermediate symbolic states are marked so the prototype knows that they are part of the test purpose (they are in the set $\mathcal{TP}$). This process is depicted in Figure A.7.

```
 Enter target state and press enter:
n_5
found!
Node marked as target state. Path to node parked as TP.
Test purpose path:
 n_0 ;   n_1 ;    n_2 ;     n_3 ;      n_5 ; Target state successfully marked!
```

Figure A.7: Choosing a target state in the prototype.

8. **Executing the Test Algorithm**

The test algorithm is executed by interacting with the tester: the tester enters the observations performed on the system under test, as well as the stimuli sent to it. Quiescence situations are also manually entered. This process is depicted in Figure A.8.

```
 About to perform online test algorithm...
The queue is now [SEtree@b6ece5]
Working with node : SEtree@b6ece5
Withy tag : n_0
Proceeding with rule NEXT, celar sets
Enter observations in the form: channelName,varName,value;[channelName,varName,value;...] (enter T for timeout0, Q to 'emmit' quiescence): user,t1,1
Observations [java.lang.String;@5e0602 to be synched
(Re)staring: The queue is []
Working with node n_0
Working with child node (Next candidate) : n_1
About to compare obs t1 with node info t1
Observation synched... perform constraint solving
adding to child input vars 1: t1 with value 1
Depth First Search DFS8

Solution : [tr=1, t1=1, w1=1]
Nodes : 2
Decisions : 2
Wrong Decisions : 0
Backtracks : 0
Max Depth : 2

Solution found, adding into NEXT
Depth First Search DFS8
No of solutions : 2
Last Solution : [tr=1, t1=1, w1=1]
Nodes : 8
Decisions : 8
Wrong Decisions : 0
Backtracks : 0
Max Depth : 8

TargetCond satisfiable; search for values
adding note n_1 to SKIP
checking if it is a target state (node n_1).... false
Found a next context. Adding node to queue : SEtree@14693c7
With label : n_1
Continue algorithm? (N for quit)
```

Figure A.8: Test algorithm for the Slot Machine example.

9. **Emission of a Verdict**

The algorithm continues it execution until a verdict is found. The emission of a verdict ($PASS$) is depicted in Figure A.9.

```
Depth First Search DFS15
No of solutions : 2
Last Solution : [s1=1, w1=1, tr=1, t1=1, w1=1]
Nodes : 5
Decisions : 5
Wrong Decisions : 0
Backtracks : 0
Max Depth : 5

TargetCond satisfiable; search for values
adding note n_5 to SKIP
checking if it is a target state (node n_5).... true
adding note n_5 to PASS
Found a next context. Adding node to queue : SEtree@1af9e22
With label : n_5
 obs es true y timeoutO es false y SynchWithObs es true
Continue algorithm? (N for quit)n
The queue is now [SEtree@1af9e22]
Working with node : SEtree@1af9e22
Withy tag : n_5
And the verdict is:
Pass
jopez@kangourou:~/work/tesis/implementation/java/stree$
```

Figure A.9: Verdict $PASS$ for Slot Machine example.

A.2 Verdicts of the Prototype

Our test algorithm aims at emitting a verdict for the system under test. As introduced in Section 7.4, there are 6 possible verdicts: 2 related to interactions with Web services, and 4 related to the orchestrator itself. Moreover, in this section we only present the 4 verdicts regarding the orchestrator, since the other two were already exemplified in Section 9.5.

1. $FAIL$

   The sequence of execution of rules for the modified Slot Machine example (introduced in Section 9.5) in order to reach a $FAIL$ verdict is:

   $\emptyset \xrightarrow[u\_start?4]{Rule\ 6} \{(\eta_1, true)\} \xrightarrow[u\_screen!'lose']{Rule\ 2} FAIL$, and the verdict is given because $Next(u\_screen!'lose', \{(\eta_1, true)\}) = \emptyset$.

   1. stimuli: u_start→4

   2. obs: u_screen→'lose'

   3. Verdict: FAIL, **{**u_start?token$_1$,token=4^seed=2^win_seq=6^seq=seq$_0$^amount=amount$_0$**}**, u_screen!'lose'

   Figure A.10: $FAIL$ verdict for the modified Slot Machine example.

2. $INCONC$

   The sequence of execution of rules for the Slot Machine example to reach an $INCONC$ verdict is:

$\emptyset \xrightarrow[u\_start?4]{Rule\ 6} \{(\eta_1, true)\} \xrightarrow[u\_screen!'lose']{Rule\ 3} INCONC$, and the verdict is given because $Next(u\_screen!'lose', \{(\eta_1, true)\}) = \{(\eta_4, seq_1 <> win\_seq_1)\}$, but $Skip(Next(u\_screen!'lose', \{(\eta_1, true)\})) = \emptyset$.

1. stimuli: u_start→4

2. obs: u_screen→'lose'

3. Verdict: INCONC, {u_start?token$_1$,token=4^seed=2^win_seq=6^seq=seq$_0$}

Figure A.11: $INCONC$ verdict for the Slot Machine example.

3. $PASS$

The sequence of execution of rules for the Slot Machine example in order to reach a $PASS$ verdict is:

$\emptyset \xrightarrow[u\_start?4]{Rule\ 6} \{(\eta_1, true)\} \xrightarrow[u\_screen!'win']{Rule\ 4} PASS$, and the verdict is given because $Next(u\_screen!'win', \{(\eta_1, true)\}) = \{(\eta_5, seq_1 = win\_seq_1)\}$, and $Pass(Next(u\_screen!'lose', \{(\eta_1, true)\})) = \{(\eta_5, seq_1 = win\_seq_1)\}$.

1. stimuli: u_start→4

2. obs: u_screen→'win'

3. Verdict: PASS, {u_start?token$_1$.sg_generate!seed$_1$.sg_sequence?seq$_1$,token=4^seed=2^win_seq=6^seq=6}

Figure A.12: $PASS$ verdict for the Slot Machine example.

4. $WeakPASS$

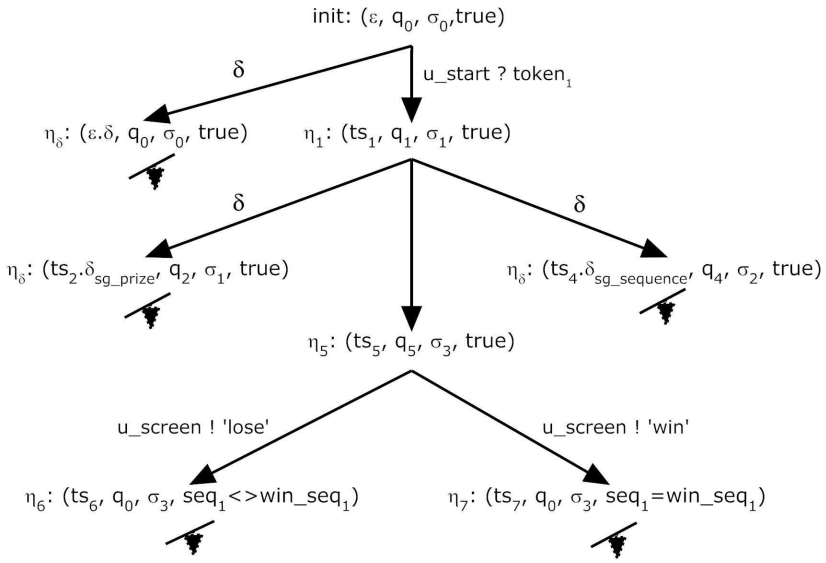
For the $WeakPASS$ verdict, we can refer to [Gaston 2006]. In order to obtain a $WeakPASS$ verdict, there has to be non-determinism in the system, which is not the case for the Slot Machine example. It would apply, for instance, in an ATM where the message *transaction denied* can be due to the fact that the user is requesting an amount of money which is superior to the amount that he or she has in his or her account, or because the total amount of money withdrawn by the user is superior to the daily allowed amount to be withdrawn. In this case, if the target state is, let us say, the first case where the amount of the request is superior to the one the user has in his or her account, and if the tester observes the message *transaction denied*, it may happen that this observation leads to the target state *or* to the other case, where the daily amount has been reached. Therefore the verdict is $WeakPASS$.

Nevertheless, in order to make our example as complete as possible and to avoid introducing a whole new example, let us consider the example of the modified

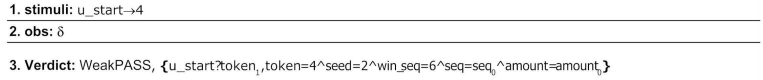
Slot Machine, with the assumption that the communication channels are hidden. The (once more) new $Obs(\mathbb{O}rch)$ is depicted in Figure A.13(a). If our target state is to detect the quiescence situation due to the lack of response of SG when it is asked for the amount of the prize, then we get the $WeakPASS$ verdict, as depicted in Figure A.13(b). This verdict is due to the fact that we cannot discriminate between the two quiescence situations: one can be due to the lack of response of SG when requesting it for the amount of the prize, and the second one can be due to the lack of response of SG when requesting it for the user's sequence.

The sequence of execution of rules for this version of the modified Slot Machine example in order to reach a $WeakPASS$ verdict is:

$\emptyset \xrightarrow[u\_start?4]{Rule\ 6} \{(\eta_1, true)\} \xrightarrow[\delta]{Rule\ 5} WeakPASS$, and the verdict is given because $Pass(Next(\delta, \{(\eta_1, true)\})) = \{(\eta_\delta, true)\}$, but $Next(\delta, \{\eta_1\}) = \{(\eta_\delta, true), (\eta_\delta, true)\}$.



(a) $Obs(\mathbb{O}rch)$ of the modified Slot Machine example with hidden channels.



(b) $WeakPASS$ verdict.

Figure A.13: $WeakPASS$ verdict for the modified Slot Machine example with hidden channels.

The verdicts $WS.HYPFAIL$ and $WS.INCONC$ are not presented here since they were already presented in Section 9.5.

Bibliography

[Akkiraju 2005] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth and K. Verma. WSDL-S (version 1.o). W3C, 2005. http://www.w3.org/Submission/WSDL-S/. 14

[Alves 2007] A. Alves and et al. SEE MOUNIR. Web Services Business Process Execution Language Version 2.0. OASIS, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. 1, 11

[Angelis 2010] F. De Angelis, A. Polini and G. De Angelis. *A Counter-Example Testing Approach for Orchestrated Services.* In ICST '10: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, pages 373–382, Washington, DC, USA, 2010. IEEE Computer Society. 74

[Anido 2003] R. Anido, A. Cavalli, L. Lima Jr. and N. Yevtushenko. *Test suite minimization for testing in context.* Softw. Test., Verif. Reliab., vol. 13, no. 3, pages 141–155, 2003. 4

[Bentakouk 2009] L. Bentakouk, P. Poizat and F. Zaïdi. *A Formal Framework for Service Orchestration Testing based on Symbolic Transition Systems.* In Testing of Communicating Systems and Formal Approaches to Software Testing (TESTCOM/TFATES), The Netherlands, 2009. 16, 19, 40, 43, 44, 90, 153

[Bertolino 2008] A. Bertolino, G. De Angelis, L. Frantzen and A. Polini. *Model-Based Generation of Testbeds for Web Services.* In Testing of Communicating Systems and Formal Approaches to Software Testing (TESTCOM/FATES), numéro 5047 de LNCS, pages 266–282, 2008. 42

[Bertolino 2009] A. Bertolino, G. De Angelis, L. Frantzen and A. Polini. *The PLASTIC Framework and Tools for Testing Service-Oriented Applications.* In A. De Lucia and F. Ferrucci, editeurs, Proceedings of the International Summer School on Software Engineering – ISSSE 2006-2008, numéro 5413 de Lecture Notes in Computer Science, pages 106–139. Springer, 2009. 42

[Braspenning 2006] N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak and J.E. Rooda. *A Model-based Integration and Testing Method to Reduce System Development Effort.* Electronic Notes in Theoretical Computer Science, vol. 164, no. 4, pages 13–28, 2006. Proc. of the Second Workshop on Model Based Testing (MBT 2006). 5, 40

[Bravetti 2007] M. Bravetti and G. Zavattaro. *Towards a Unifying Theory for Choreography Conformance and Contract Compliance.* In Software Composition, LNCS, pages 34–50, 2007. 42, 43, 44

[Bray 2008] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C, 2008. http://www.w3.org/TR/REC-xml/. 1, 13

[Cámara 2006] J. Cámara, C. Canal, J. Cubo and A. Vallecillo. *Formalizing WS-BPEL Business Processes Using Process Algebra.* Electron. Notes Theor. Comput. Sci., vol. 154, no. 1, pages 159–173, 2006. 19

[Cambronero 2007] M.-E. Cambronero, J. J. Pardo, G. Díaz and V. Valero. *Using RT-UML for modelling web services.* In SAC, pages 643–648, 2007. 19

[Cao 2009] T.D. Cao, P. Felix, R. Castanet and I. Berrada. *Testing Web Services Composition Using the TGSE Tool.* IEEE Congress on Services, vol. 0, pages 187–194, 2009. 40

[Cao 2010] T.D. Cao, P. Félix, R. Castanet and I. Berrada. *Online Testing Framework for Web Services.* In ICST, pages 363–372, 2010. 5, 40

[Chow 1978] T.S. Chow. *Testing Software Design Modeled by Finite-State Machines.* Software Engineering, IEEE Transactions on, vol. SE-4, no. 3, pages 178 – 187, may. 1978. 18

[Christensen 2001] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana. WSDL (version 1.1). W3C, March 2001. http://www.w3.org/TR/wsdl. 1, 13

[Clarke 1976] L. A. Clarke. *A system to generate test data and symbolically execute programs.* IEEE Transactions on Software Engineering, vol. 2(3), pages 215–222, 1976. 89

[Clement 2004] L. Clement, A. Hately, C. von Riegen and T. Rogers. UDDI (version 3.0.2). OASIS, October 2004. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm. 1, 13

[Dijkstra 1979] E. Dijkstra. *Structured programming.* pages 41–48, 1979. 3

[Dumas 2005] M. Dumas, C. Ouyang and A. Rozinat. *Choreography Conformance Checking: An Approach based on BPEL and Petri Nets (extended version). BPM Center Report BPM-05-25, BPMcenter.org*, 2005. 19

[Escobedo 2009a] J.P. Escobedo, P. Le Gall, C. Gaston and A. Cavalli. *Examples of Testing Scenarios for Web Service Composition.* Rapport technique 09003_LOR, TELECOM & Management SudParis, 2009. http://www.it-sudparis.eu/.

[Escobedo 2009b] J.P. Escobedo, P. Le Gall, C. Gaston and A. Cavalli. *Observability and Controllability Issues in Conformance Testing of Web Service Composition.* In Testing of Communicating Systems and Formal Approaches to Software Testing (TESTCOM/FATES), 2009.

[Escobedo 2010] J.P. Escobedo, P. Le Gall, C. Gaston and A. Cavalli. *Testing Web Service Orchestrators in context:a symbolic approach.* In Proc. of Software Engineering Formal Methods (SEFM) '10. IEEE Computer Society, 2010.

[Faivre 2007] A. Faivre, C. Gaston and P. Le Gall. *Symbolic Model Based Testing for Component oriented Sytems.* In Testing of Communicating Systems (TESTCOM), volume 4581/2007 of *LNCS*, pages 90–106, 2007. 41

[Fielding 1999] R. T. Fielding, J. Gettys, J. C. Mogul, H. Nielsen, L. Masinter, P. Leach and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. 1999. `http://tools.ietf.org/html/rfc2616`. 1, 13

[Fielding 2000] T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000. 14

[Frantzen 2006a] L. Frantzen, J. Tretmans and R. d. Vries. *Towards Model-Based Testing of Web Services.* In A. Polini, editeur, International Workshop on Web Services - Modeling and Testing (WS-MaTe), pages 67–82, Italy, 2006. 32

[Frantzen 2006b] L. Frantzen, J. Tretmans and T.A.C. Willemse. *A Symbolic Framework for Model-Based Testing.* In Intl. Workshops FATES/RV, volume 4262 of *LNCS*, pages 40–54, 2006. 5, 88, 146

[Frantzen 2007] L. Frantzen and J. Tretmans. *Model-Based Testing of Environmental Conformance of Components.* In Formal Methods of Components and Objects (FMCO), numéro 4709 de LNCS, pages 1–25, 2007. 5, 32, 41

[Frantzen 2009] L. Frantzen, M. N. Huerta, Z. G. Kiss and T. Wallet. *On-The-Fly Model-Based Testing of Web Services with Jambition.* In R. Bruni and K. Wolf, editeurs, Intl. Workshop on Web Services and Formal Methods (WS-FM), numéro 5387 de LNCS, pages 143–157, 2009. 32, 43, 44

[Fraser 2009] G. Fraser, F. Wotawa and P. Ammann. *Testing with model checkers: a survey.* Softw. Test., Verif. Reliab., vol. 19, no. 3, pages 215–261, 2009. 2

[Gaston 2006] C. Gaston, P. Le Gall, N. Rapin and A. Touil. *Symbolic Execution Techniques for Test Purpose Definition.* In Testing of Communicating Systems (TESTCOM), volume 3964 of *LNCS*, pages 1–18, 2006. 5, 19, 87, 88, 104, 110, 111, 114, 124, 125, 129, 134, 177

[Gortmaker 2004] J. Gortmaker, M. Janssen and R. Wagenaar. *The advantages of web service orchestration in perspective.* In ICEC '04: Proceedings of the 6th international conference on Electronic commerce, pages 506–515, New York, NY, USA, 2004. ACM. 1

[Gudgin 2007] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. Nielsen, A. Karmarkar and Y. Lafon. SOAP (Version 1.2). W3C, April 2007. `http://www.w3.org/TR/soap12-part1/`. 1, 13

[He 1999] J. He and K. Turner. *Protocol-inspired hardware testing.* In Proc. Testing Communicating Systems XII, pages 131–147. Kluwer Academic Publishers, 1999. 32

[Heymer 2007] S. Heymer and J. Grabowski. *Formal Methods and Conformance Testing - or - What are we testing anyway?* 2007. 21

[Hoffman 2005] C. M. Hoffman and R. Joan-Arinyo. *A brief on constraint solving.* vol. 2, no. 5, pages 665–663, 2005. 150

[Huhns 2005] M. Huhns and M. Singh. *Service-Oriented Computing: Key Concepts and Principles.* IEEE Internet Computing, vol. 9, no. 1, pages 75–81, 2005. 1, 12

[Inkumsah 2008] Kobi Inkumsah and Tao Xie. *Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution.* In Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pages 297–306, September 2008. 90

[Jard 2005] C. Jard and T. Jéron. *TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems.* Int. J. Softw. Tools Technol. Transf., vol. 7, no. 4, pages 297–315, 2005. 32

[Jeannet 2005] B. Jeannet, T. Jéron, V. Rusu and E. Zinovieva. *Symbolic Test Selection based on Approximate Analysis.* In TACAS, volume 3440 of *LNCS*, pages 349–364, 2005. 32, 77, 88

[Jéron 2004] T. Jéron. *Contribution à la génération automatique de tests pour les systèmes réactifs.* Habilitation à diriger les recherches, Université de Rennes 1, March 2004. 87

[Katoen 1999] J.-P. Katoen. *Concepts, Algorithms, and Tools for Model Checking*, 1999. 3

[Khoumsi 2004] A. Khoumsi. *Test cases generation for embedded systems*, 2004. 4

[Khurshid 2003] S. Khurshid, C.S. Pasareanu and W. Visser. *Generalized Symbolic Execution for Model Checking and Testing.* In Hubert Garavel and John Hatcliff, editeurs, TACAS, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003. 90

[King 1975] J. C. King. *A new approach to program testing.* In Intl. Conf. on Reliable Software, pages 228–233. ACM, 1975. 89, 102

[Lallali 2007] M. Lallali, F. Zaïdi, C. and 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. 3

[Lallali 2008] M. Lallali, F. Zaïdi, A. Cavalli and I. Hwang. *Automatic Timed Test Case Generation for Web Services Composition.* European Conference on Web Services (ECOWS), vol. 0, pages 53–62, 2008. 5, 18, 40

[Leavens 1999] G. Leavens *et al.* The Java Modeling Language (JML). 1999. `http://www.eecs.ucf.edu/~leavens/JML/`. 3

[Li 2005] Z. Li, W. Sun, Z. B. Jiang and X. Zhang. *BPEL4WS unit testing: framework and implementation.* In International Conference on Web Services (ICWS), volume 1, pages 103–110. IEEE Computer Society, 2005. 19, 40

[Lohmann 2007] N. Lohmann. *A Feature-Complete Petri Net Semantics for WS-BPEL 2.0.* In K. va Hee, W. Reisig and K. Wolf, editeurs, Proceedings of the Workshop on Formal Approaches to Business Processes and Web Services (FABPWS'07), pages 21–35. University of Podlasie, June 2007. 19

[Lohmann 2009] N. Lohmann, E. Verbeek and R. Dijkman. *Petri Net Transformations for Business Processes — A Survey.* pages 46–63, 2009. 3, 19

[Mayer 2006] P. Mayer and D. Lübke. *Towards a BPEL unit testing framework.* In Workshop on Testing, analysis, and verification of web services and applications (TAV-WEB), pages 33–42. ACM, 2006. 40

[Mitra 2009] S. Mitra, R. Kumar and S. Basu. *A Framework for Optimal Decentralized Service-Choreography.* In ICWS, pages 493–500, 2009. 1

[Nakajima 2006] S. Nakajima. *Model-Checking Behavioral Specification of BPEL Applications.* Electr. Notes Theor. Comput. Sci., vol. 151, no. 2, pages 89–105, 2006. 18

[Noikajana 2008] S. Noikajana and T. Suwannasart. *Web Service Test Case Generation Based on Decision Table (Short Paper).* In QSIC '08: Proceedings of the 2008 The Eighth International Conference on Quality Software, pages 321–326, Washington, DC, USA, 2008. IEEE Computer Society. 2

[Peltz 2003] C. Peltz. *Web services orchestration and choreography.* In Computer, pages 46–52. IEEE Computer Society, 2003. 1

[Pu 2006] G. Pu, X. Zhao, S. Wang and Z. Qiu. *Towards the Semantics and Verification of BPEL4WS.* Electronic Notes in Theoretical Computer Science, vol. 151, no. 2, pages 33 – 52, 2006. Proceedings of the International Workshop on Web Languages and Formal Methods (WLFM 2005). 18

[Păsăreanu 2009] C. Păsăreanu and W. Visser. *A survey of new trends in symbolic execution for software testing and analysis.* Int. J. Softw. Tools Technol. Transf., vol. 11, no. 4, pages 339–353, 2009. 89

[Rusu 2000] V. Rusu, L. Bousquet and T. Jéron. *An Approach to Symbolic Test Generation.* In IFM '00: Proceedings of the Second International Conference on Integrated Formal Methods, pages 338–357, London, UK, 2000. Springer-Verlag. 5, 88

[Schmaltz 2008] J. Schmaltz and J. Tretmans. *On Conformance Testing for Timed Systems.* In FORMATS '08: Proceedings of the 6th international conference on Formal Modeling and Analysis of Timed Systems, pages 250–264, Berlin, Heidelberg, 2008. Springer-Verlag. 5, 32, 166

[Sinha 2006] A. Sinha and A. Paradkar. *Model-based functional conformance testing of web services operating on persistent data.* In TAV-WEB '06: Proc. of the 2006 workshop on Testing, analysis, and verification of web services and applications, pages 17–22, New York, NY, USA, 2006. ACM. 42

[Tillmann 2006] N. Tillmann and W. Schulte. *Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution.* IEEE Softw., vol. 23, no. 4, pages 38–47, 2006. 90

[Touil 2006] A. Touil. *Exécution symbolique pour le test de conformité et le test de raffinement.* PhD thesis, Université dÉvry-Val-D'Essonne, December 2006. 134

[Tretmans 1996a] J. Tretmans. *Conformance testing with labelled transition systems: Implementation relations and test generation.* In Computer Networks and ISDN Systems, volume 29, pages 49–79. Elsevier B.V., 1996. 87

[Tretmans 1996b] J. Tretmans. *Test Generation with Inputs, Outputs and Repetitive Quiescence.* Software - Concepts and Tools, vol. 17, no. 3, pages 103–120, 1996. 5, 21, 22, 25, 26, 27, 29, 30, 31, 87

[Tretmans 2003] G. J. Tretmans and H. Brinksma. *TorX: Automated Model-Based Testing.* In A. Hartman and K. Dussa-Ziegler, editeurs, First European Conference on Model-Driven Software Engineering, Nuremberg, Germany, pages 31–43, December 2003. 32

[Tretmans 2008] Jan Tretmans. *Model Based Testing with Labelled Transition Systems.* In Formal Methods and Testing, pages 1–38, 2008. 3

[Utting 2007] M. Utting and B. Legeard. Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, 1 édition, 2007. 3

[Valmari 1998] A. Valmari. *The State Explosion Problem.* In Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the

Advanced Course on Petri Nets, pages 429–528, London, UK, 1998. Springer-Verlag. 4, 73, 87, 89

[van Breugel 2006] F. van Breugel and M. Koshkina. *Models and Verification of BPEL*, 2006. 2, 19

[van der Bijl 2003a] H.M. van der Bijl, A. Rensink and J. Tretmans. *Component Based Testing with ioco*, 2003. 6, 29, 36, 38, 39, 40, 87

[van der Bijl 2003b] H.M. van der Bijl, A. Rensink and J. Tretmans. *Compositional Testing with ioco*. In Formal Approaches to Software Testing (FATES), pages 86–100, 2003. 5, 22, 25, 30, 32, 36, 39, 40, 87, 164

[van der Bijl 2004] M. van der Bijl, A. Rensink and J. Tretmans. *Compositional Testing with ioco*. In Workshop on Formal Approaches to Testing of Software (FATES), volume 2931 of *LNCS*, pages 86–100, 2004. 5

[van der Bijl 2005] H.M. van der Bijl, A. Rensink and J. Tretmans. *Action Refinement in Conformance Testing*, 2005. eemcs1564. 22, 26

[Viroli 2004] M. Viroli. *Towards a Formal Foundation to Orchestration Languages*. Electron. Notes Theor. Comput. Sci., vol. 105, pages 51–71, 2004. 19

[Wombacher 2004] A. Wombacher, P. Fankhauser and E. Neuhold. *Transforming BPEL into annotated deterministic finite state automata for service discovery*. In 2nd International Conference on Web Services (ICWS 04), pages 316–323, Los Alamitos, California, USA, July 2004. IEEE Computer Society Press. 18

[Xie 2005] T. Xie, D. Marinov, W. Schulte and D. Notkin. *Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution*. In TACAS, pages 365–381, 2005. 90

[Yang 2005] Y. Yang, Q.P. Tan, J. Y. and F. Liu. *Transformation BPEL to CP-nets for verifying Web services composition*. page 6 pp., aug. 2005. 19

[Yoo 2010] T. Yoo, B. Jeong and H. Cho. *A Petri Nets based functional validation for services composition*. Expert Syst. Appl., vol. 37, no. 5, pages 3768–3776, 2010. 3, 19

[Yuan 2006] Y. Yuan, Z. Li and W. Sun. *A Graph-Search Based Approach to BPEL4WS Test Generation*. In Intl. Conf. on Software Engineering Advances, page 14, 2006. 40