Génération automatique de tests pour des modèles avec variables ou récursivité.

Camille Constant

N° d'ordre: 3754

THÈSE

Présentée devant

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

Camille Constant

Équipe d'accueil : VerTeCS - IRISA
École Doctorale : Matisse
Composante universitaire : IFSIC

Titre de la thèse :

Génération automatique de tests
pour modèles avec variables ou récursivité

soutenue le 24 novembre 2008 devant la commission d'examen

M. :	Thomas	Jensen	Président
Mme, M. :	Pascale	Le Gall	Rapporteurs
	Richard	Castanet	
MM. :	Bertrand	Jeannet	Examinateurs
	Thierry	Massart	
	Thierry	Jéron	
M. :	Yves-Marie	Quemener	Invité

Remerciements

J'aimerais tout d'abord remercier Bertrand Jeannet qui m'a encadrée pendant une partie de ma thèse et sans qui tout cela n'aurait pu être possible. Son soutien, ses conseils et sa franchise m'ont permis d'aller de l'avant et de développer toute une partie de mes travaux de recherche.

J'aimerais également remercier Thierry Jéron, directeur de thèse et chef de l'équipe Vertecs, qui a su se montrer patient et disponible en particulier lors de la rédaction de mon manuscrit et de la préparation de la soutenance.

Je remercie Pascale Le Gall et Richard Castanet qui ont bien voulu rapporter cette thèse pour leur remarques constructives et leur gentillesse. Merci également à Thierry Massart d'avoir bien voulu faire le déplacement depuis Bruxelles pour assister à ma soutenance et à Thomas Jensen pour avoir accepté de présider le jury. Enfin, je souhaiterais remercier Yves-Marie Quemener d'avoir supervisé cette thèse pour France Telecom R&D et d'avoir été membre invité du jury de thèse.

Un grand merci à l'ensemble de l'équipe Vertecs (et à Patricia) pour leur aide et leur soutien mais également pour leur convivialité et la bonne ambiance qu'ils font régner durant les pauses.

Merci à Vlad pour m'avoir orientée lors de mon travail sur la combinaison entre vérification et test et pour les publications qui en ont découlé.

J'aimerais remercier en particulier Lydie pour les agréables discussions que nous avons eues mais également pour avoir si bien géré les missions et l'organisation autour de la soutenance.

Je remercie également énormément Hervé pour m'avoir supportée quelle que soit mon humeur, pour toute l'aide et le soutien apportés mais également pour toutes les discussions que nous avons eues et tous les livres et vidéos échangés.

Merci à Florimond pour sa patience en particulier lors de l'installation de STG, à Nathalie pour nos discussions entre filles (j'aurais aimé que tu arrives plus tôt au sein de l'équipe), à Christophe pour les discussions sur l'enseignement et pour m'avoir bien souvent fait rire, à Tristan pour m'avoir rassurée lors des baisses de moral, à Jérémy pour avoir supporté de longues discussions sur de nombreux sujets, à Hatem pour sa générosité, à Gwenaël pour ses idées, et enfin à Valéry et François-Xavier pour avoir rendu le DEA si agréable par l'ambiance qui régnait dans notre bureau.

Je souhaiterais également remercier tous les collègues de l'IRISA et d'ailleurs, trop nombreux pour les citer, dont les conversations furent et sont fort enrichissantes et agréables.

J'aimerais remercier de manière toute particulière Eric Ossieux, enseignant de danse modern'jazz à l'Université Rennes 1 et ami, qui m'a permis d'exprimer par la danse ce que je ne pouvais exprimer autrement. J'ai, grâce à lui, non seulement gagné en technique mais également en assurance au fil des représentations que nous avons pu donner dans l'Ouest. Merci également aux danseuses qui ont participé à cette aventure pour leur entrain, leur dynamisme et leur amitié (Nolwenn, Edith, Marion, Laurie et tant d'autres).

Je remercie également tous mes amis de Rennes et de Gironde pour leur soutien, leur générosité et leur bonne humeur. Merci à tous ceux qui m'ont supportée quotidiennement ou presque (Gaylord, Gaël, Floriane, Gaël, Laëtitia, Peggy, Olivier, Xavier, etc.) et en particulier merci à Christian pour avoir écouté tous mes états d'âme sans jamais se plaindre. Merci également à ceux que je voyais moins souvent mais toujours avec autant de plaisir : Amélie, Solène, Antoine, Jiji et en particulier Maelle dont l'amitié et le soutien ont été d'une grande importance pour moi. Merci à mes amis d'enfance de Gironde (Emilie, Cyril, Mylène, Guillaume, Estelle, Damien et Julien entre autres) pour être toujours là malgré le temps et la distance et pour tous les fous rires partagés. Merci également à Ludovic, venu assister à ma soutenance, que j'ai toujours autant de plaisir à voir.

Je tiens particulièrement à remercier Alexandre pour sa patience, sa gentillesse, sa générosité et sa compréhension. Merci de me supporter et me soutenir tous les jours, quelle que soit la situation. Merci également à ses parents, Jacques et Catherine, pour leur accueil chaleureux, leur soutien et leur générosité.

Enfin, je souhaiterais remercier toute ma famille pour leur patience et leur soutien. Merci en particulier à ma cousine Nathalie pour sa disponibilité et les heures passées au téléphone qui m'ont fait tant de bien. Merci également à mon frère Olivier pour toute l'aide et l'expérience apportées durant ma thèse (et avant). Pour finir, merci à mes parents pour m'avoir toujours soutenue et aidée tant dans mon travail que dans ma vie personnelle. Merci d'être toujours là pour nous.

Table des matières

Introduction

Les systèmes informatiques sont désormais omniprésents : ils sont utilisés dans bien des domaines, tels que les transports, la communication, la finance, etc. Ces systèmes deviennent de plus en plus complexes et leur fiabilité est essentielle, en particulier pour les systèmes critiques, systèmes dont une erreur ou une panne peut avoir de graves conséquences tant sur le plan humain que matériel ou écologique. Il est donc important qu'un logiciel ait bien le comportement attendu. La phase de validation est l'une des phases les plus coûteuses lors de la livraison d'un logiciel, tant en terme de temps que de personnel car celle-ci est souvent effectuée de manière artisanale. Il serait donc intéressant d'automatiser au maximum les techniques de validation utilisées. Il en existe différentes permettant de vérifier la correction d'un logiciel, telles que la vérification formelle ou le test. La vérification va permettre de prouver que le modèle d'un système satisfait des propriétés, tandis que le test va permettre de détecter un maximum d'erreurs de l'implémentation lors de son exécution.

Nous nous intéresserons en particulier au test, une des techniques de validation les plus utilisées dans le monde industriel. Celui-ci ne peut pas prouver la correction d'une implémentation mais peut améliorer la confiance qu'on a en elle. Concrètement, tester un logiciel consiste à l'exécuter en lui fournissant des entrées précises et à observer ses sorties. Il faut ensuite comparer ces sorties avec celles attendues, décrites dans un système de référence (cahiers des charges, spécifications, etc.), et en déduire ainsi si l'implémentation est correcte. Ceci est appelé le problème de l'oracle.

Différents types de test. Le test permet de vérifier différentes caractéristiques du système. Il est par exemple possible de tester la robustesse, la performance ou les fonctionnalités d'un logiciel. Tester la robustesse consiste à vérifier que l'implémentation a un comportement acceptable en présence d'entrées invalides, de dysfonctionnements internes, de conditions de stress, de pannes, etc. Le test de performance permet quant à lui de mesurer les temps de réponse du système en fonction de sa sollicitation. On peut par exemple déterminer à quel point on peut stimuler le système, ou si celui-ci ne perd pas en performance au fil du temps. Nous nous intéresserons dans ce document au test fonctionnel. Comme son nom l'indique, celui-ci consiste à vérifier si les exigences fonctionnelles du cahier des charges sont bien respectées.

Il existe différents degrés d'accessibilité au système (implémentation) sous test :

nous pouvons n'en connaître que l'interface (système "boîte noire" dont le code nous est inconnu) ou en connaître toute la structure (système "boîte blanche", dont on connaît tout le code source). En fonction de ce degré d'accessibilité, nous pouvons distinguer différents types de test : les tests de type boîte blanche, utilisés dans le cadre des tests unitaires comme celui des tests d'intégration, et les tests de type boîte noire, utilisés plutôt dans le cadre des tests systèmes, mais également pour les tests unitaires et d'intégration.

Le test structurel, par exemple, est un test de type boîte blanche. Il a pour modèle une représentation du code source (graphe de flot de contrôle, graphe def/use[1], etc.) et permet de tester des comportements précis de l'implémentation grâce à une sélection des tests effectuée suivant un critère de couverture structurel. Il est par exemple possible de tester un programme suivant le critère de couverture de tous les sommets (c'est-à-dire toutes les instructions), de tous les arcs (toutes les décisions), de tous les chemins sur les graphes de flot de contrôle, ou suivant le critère de couverture de toutes les définitions ou toutes les utilisations de variables sur un graphe def/use. Il existe deux types d'approche pour le test structurel : l'approche dynamique [DN84, TFW91, Kor90] qui consiste à tester le programme au fur et à mesure de son exécution et l'approche statique [Kin76, Cla76, BBS+79] effectuant une analyse du programme avant l'exécution de celui-ci.

Test de conformité. Nous nous intéresserons, dans la suite de ce document, au test de type fonctionnel boîte noire s'effectuant sur un système réactif, un système réagissant aux stimuli de son environnement. Nous nous focaliserons plus précisément sur le test de conformité, test pour lequel la spécification sert de référence : c'est elle qui décrit les comportements attendus de l'implémentation. Elle sert donc d'oracle. Le but du test de conformité est de vérifier que le comportement de l'implémentation réelle du système, appelée implémentation sous test (IUT pour *Implémentation Under Test*) est correct/conforme au comportement spécifié. La relation reliant la spécification et l'implémentation est appelée *relation de conformité*. L'IUT est en général du code exécutable ou du matériel dont le code source nous est inconnu. Nous allons donc nous baser sur les interactions de l'IUT avec son environnement pour établir la correction de son comportement.

L'environnement va être simulé par un *testeur* exécutant des *cas de test* sur l'implémentation. Un cas de test est un test élémentaire, tiré de la spécification, composé, par exemple, pour un système réactif, d'une ou plusieurs séquences d'interactions (entrées/sorties). Un ensemble de cas de test est appelé une *suite de test*. Le testeur distingue deux types d'interactions : celles qui sont contrôlables (les entrées de l'IUT) et celles qui sont observables (ses sorties). Le comportement de l'IUT est observé à travers des interfaces appelées PCO pour *Points de Contrôle et d'Observation*. Le testeur fournit donc des entrées (actions contrôlables) à l'IUT et observe ses sorties à travers les PCO. Suivant la réaction du système après une interaction, le testeur peut

[1] Un graphe def/use est un graphe de flot de contrôle sur lequel sont ajoutées des informations sur les définitions et utilisations des variables.

alors soit continuer le test soit émettre un *verdict* (voir figure 1).

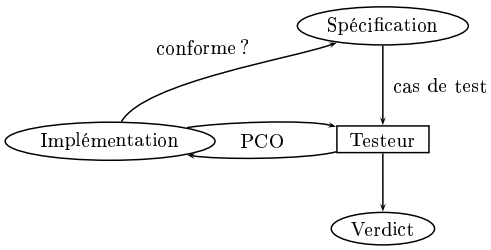


FIG. 1 – Test de conformité.

L'échec d'un seul cas de test suffit à prouver la non-conformité de l'IUT. Ceci est vrai même si l'implémentation n'est pas déterministe du point de vue de l'observation, c'est-à-dire, si, pour une même séquence d'entrées, l'implémentation n'a pas toujours le même comportement. Deux exécutions d'un même cas de test peuvent alors produire deux verdicts différents.

Propriétés des suites de test. Idéalement, une suite de test devrait être capable de rejeter toute implémentation non-conforme (propriété d'*exhaustivité*), et uniquement celles-ci (propriété de correction[2], de *non-biais*). La propriété de non-biais assure qu'il y a non-conformité lorsqu'une implémentation est rejetée par la suite de test ; cependant, une implémentation non-conforme peut ne pas être rejetée. La propriété d'exhaustivité assure quant à elle que toute implémentation non-conforme sera rejetée ; cependant, une implémentation conforme pourrait faire échouer le test. La propriété de *complétude* est la conjonction de ces deux propriétés. Malheureusement, en pratique, une suite de test peut être correcte (propriété de non-biais satisfaite), mais il est difficile, voire impossible, d'assurer qu'elle soit complète. Dans le domaine du test, nous ne pouvons pas prouver la non-existence d'une faute, mais nous pouvons prouver son existence.

Sélection. Puisque nous ne pouvons pas tester tous les comportements possibles de l'implémentation par rapport à ceux de la spécification, une solution consiste à sélectionner un comportement particulier de celle-ci (appelé *objectif de test*). L'objectif de test permet de focaliser le test sur une fonctionnalité précise, ce qui générera un ou plusieurs cas de tests. Il existe d'autres techniques de sélection telles que celles basées sur les critères de couverture (voir par exemple [ZHM97]), mais nous nous focaliserons sur celle de l'objectif de test dans la suite de ce manuscrit.

Le principe de la sélection consiste à réduire l'ensemble infini de cas de test nécessaire à la propriété de complétude, tout en gardant vraie cette propriété à la limite : le mode de sélection doit permettre de sélectionner tout cas de test de cet ensemble infini. Avec le mode de sélection par objectif de test, la satisfaction de celui-ci (verdict *Pass* produit)

[2]Cela correspond au terme *soundness* en anglais.

signifie que ce comportement précis de l'IUT est conforme à celui de la spécification. Si le verdict *Fail* est produit, l'implémentation n'est pas conforme (voir figure 2).

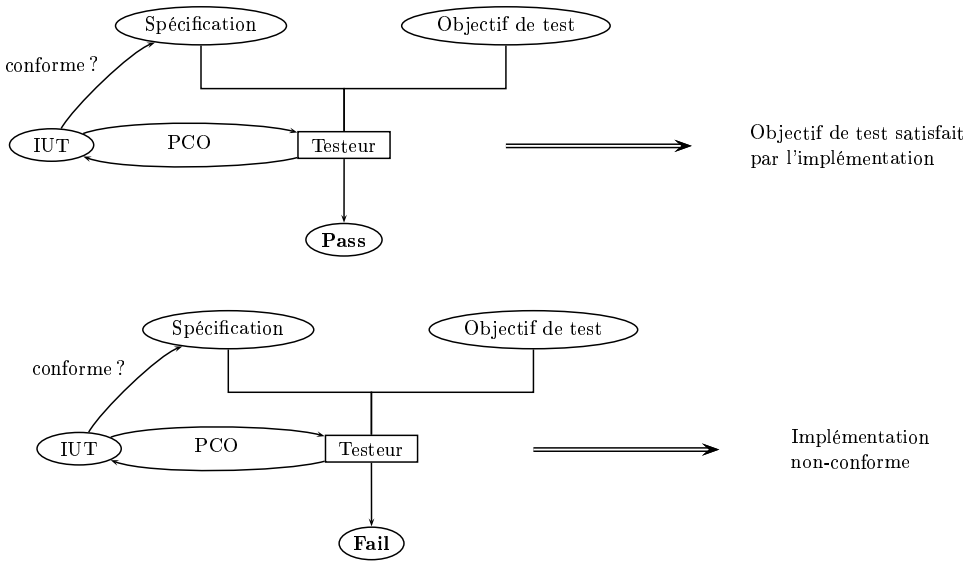


FIG. 2 – Sélection par objectif de test.

Formalisation. La génération des cas de test (ou génération de tests) se fait encore couramment de manière artisanale. Notre but est d'automatiser la génération : cela serait à la fois plus pratique et moins coûteux. Pour cela, il est nécessaire de formaliser le test de conformité [ISO92, BAL+90].

Tout d'abord, il nous faut formaliser la spécification. Celle-ci doit être assez claire et précise pour qu'il n'y ait aucune ambiguïté sur le comportement attendu du système. Elle dérive souvent d'un cahier des charges, document où sont relatées les exigences que doit satisfaire le système. La spécification doit être donnée dans un langage dont la sémantique peut être décrite dans un modèle mathématique. Par exemple, si la spécification est décrite par un langage de description formelle comme LDS (*Langage de description et de spécification*, SDL en anglais), la sémantique peut être donnée par un système de transitions. Il existe d'autres langages de description formelle tels que Lustre ou Lotos par exemple.

Les implémentations sous test ne sont pas des objets mathématiques. Néanmoins, afin de raisonner formellement, nous allons supposer que toute implémentation est modélisable. Notons que le modèle des implémentations peut être différent de celui des spécifications. Il est également nécessaire de formaliser la relation de conformité liant la spécification et l'IUT par une relation mathématique (inclusion de comportement ou équivalences par exemple).

Il est aussi nécessaire de formaliser les interactions entre le cas de test et l'IUT lors de l'exécution des tests, l'observation liée à l'exécution et les verdicts émis. Reste à

décrire l'algorithme de génération automatique de tests en fonction de tous ces éléments et de l'objectif de test pour la sélection.

Motivation. Cette thèse a été en partie supportée par France Telecom (renommé Orange) Recherche et Développement. L'étude de cas du service vocal QuiDonc fournie par cette entreprise est à l'origine des deux axes de recherche entrepris au cours de cette thèse. QuiDonc est un système réactif contenant des variables et une récursivité bornée. Nous nous sommes donc d'abord intéressés à la génération de tests pour des modèles de systèmes de transitions à entrées/sorties avec variables (les ioSTS pour *Input/Output Symbolic Transitions System*), combinée à de la vérification. Nous avons ensuite étudié la génération de tests sur un modèle de système de transitions à entrées/sorties récursif. Le but ultime de ces deux axes de recherche étant d'aboutir à un algorithme de génération de tests sur des modèles de systèmes récursifs avec variables.

La théorie du test de conformité est basée sur l'hypothèse que la spécification est correcte, qu'elle est l'exacte représentation du comportement attendu du système. Or, cette hypothèse peut être fausse. Il serait donc intéressant, avant de générer des cas de test à partir de cette spécification, de s'assurer de la correction de celle-ci. Pour cela, nous allons utiliser la vérification formelle, technique de validation souvent mise en opposition avec le test de conformité.

La vérification formelle consiste à s'assurer qu'un système est correct par rapport aux propriétés que celui-ci devrait satisfaire. Elle permet *a priori* de prouver exhaustivement une propriété sur la spécification ou de prouver sa violation en produisant un contre-exemple. Le test et la vérification peuvent donc être complémentaires, la vérification permettant d'abord de s'assurer de la correction de la spécification par rapport aux propriétés, et le test permettant ensuite de détecter des erreurs de l'implémentation par rapport à cette même spécification. Cependant, pour des modèles expressifs, certains problèmes de vérification sont indécidables, ou simplement trop complexes, et la vérification peut alors n'être que partielle. Le test pourra dans ce cas prouver que la propriété est satisfaite ou violée par la spécification. Cette complémentarité va également permettre de sélectionner les cas de test à partir de propriétés de sûreté (rien de mauvais n'arrivera), ou d'accessibilité (quelque chose d'attendu arrivera), modélisées par un *observateur*, et non à partir d'un objectif de test.

Cependant, cette complémentarité n'est pas totalement satisfaisante. En effet, le but serait de s'assurer que l'implémentation satisfait ces propriétés. Or, en supposant que la spécification satisfait les propriétés, la non-conformité (respectivement la conformité, si on pouvait l'établir) ne permet pas de conclure à la violation (respectivement à la satisfaction) des propriétés par l'implémentation. Il manque donc un lien formel entre vérification et test de conformité, lien que nous nous attachons à étudier dans la deuxième partie de ce document.

Nous étendons, dans la troisième partie de ce document, la génération de tests par l'expressivité du modèle de spécification en nous focalisant sur les spécifications interprocédurales récursives. Elles permettent une représentation plus compacte de systèmes

récursifs que les modèles non-récursifs, et plus expressive qu'un modèle avec une seule procédure. Notre méthode de génération de tests, appliquée à ce modèle, est basée sur une analyse exacte de co-accessibilité permettant de décider si et comment un objectif de test pourra être atteint. Cependant, l'incapacité des cas de test à connaître leur propre pile ne permet pas d'utiliser la totalité des résultats de l'analyse. Nous discuterons de ce problème d'observation partielle et de ses conséquences dans cette troisième partie.

Structure du document. Nous présenterons dans la partie I la théorie du test de conformité fondée sur la relation de conformité ioco de Jan Tretmans [Tre96] pour des systèmes de transitions. Cette théorie et le modèle des systèmes de transitions à entrées/sorties (les ioLTS pour *Input-Output Labelled Transitions System*) seront présentés au chapitre 1. La génération automatique de tests (sélection comprise) basée sur ce modèle et cette relation seront ensuite détaillées au chapitre 2.

Nous présenterons dans la partie II une méthodologie combinant vérification et test de conformité. Celle-ci permettra, pour des propriétés de sûreté ou d'accessibilité, d'une part de détecter la violation de la propriété de sûreté (respectivement la satisfaction d'une propriété d'accessibilité) sur l'implémentation, mais aussi de compléter la vérification en détectant la violation (respectivement la satisfaction) sur la spécification, le tout pendant l'exécution du test. Nous nous basons alors sur le modèle des systèmes de transitions symboliques à entrées/sorties (les ioSTS) comportant des variables à domaine infini. Ce modèle, présenté au chapitre 3, a pour sémantique les ioLTS à espace d'états infini, ce qui rend les problèmes d'accessibilité typiques de la vérification indécidables sur celui-ci. La vérification et la sélection des cas de test seront présentées au chapitre 4. Des expérimentations sur l'outil de génération symbolique de tests STG [CJRZ02, PJJ07] seront ensuite illustrées au chapitre 5.

Le modèle des ioSTS étend celui des ioLTS par l'ajout de variables. Ces deux modèles permettent de représenter des programmes à une seule procédure. Nous nous penchons, dans la partie III de ce document, sur la génération de tests à partir de spécifications interprocédurales récursives. Le modèle des ioPDS (pour *input/output PushDown System*) que nous étudions dans cette partie étend celui des ioLTS par une pile portant sur un alphabet fini. Ce modèle, ainsi que la méthode de génération de tests basée sur celui-ci seront présentés au chapitre 6 : un cas de test récursif sera généré à partir d'une spécification récursive et d'un cas de test non-récursif. Cette même méthode sera ensuite présentée au chapitre 7, mais elle sera cette fois basée sur des transformations de programmes. Les choix techniques seront guidés par la théorie du test et par les propriétés dues aux modèles des ioPDS et des ioLTS, mais la génération en elle-même sera définie en termes de concepts de langages de programmation. Notre algorithme de sélection sera enfin présenté au chapitre 8. La sélection du cas de test, basée sur les mêmes principes que pour le modèle des ioLTS, ne pourra être optimale à cause du problème d'observation partielle de la pile. Un moyen de minimiser l'impact de ce problème sera alors proposé. Enfin, des expérimentations sur un analyseur interprocédural développé par Bertrand Jeannet (InterprocStack) seront présentées au chapitre 9.

Première partie

Préliminaires sur la génération de tests de conformité à la ioco

Chapitre 1

Théorie du test sur les systèmes de transitions

La théorie du test de conformité basée sur le modèle des systèmes de transitions sera présentée dans ce chapitre. Celle-ci servira de référence aux algorithmes et modèles présentés ultérieurement dans ce document. L'architecture du test de conformité avec sélection par objectif de test est présentée figure 1.1. Nous en détaillerons certains aspects dans ce chapitre. Le modèle des ioLTS sera d'abord présenté section 1.1, puis nous nous intéresserons à la relation de conformité liant l'implémentation à sa spécification (section 1.2). Enfin, nous discuterons du modèle des cas de test, de la formalisation de son exécution sur l'implémentation et des propriétés attendues des suites de test (section 1.3.1). La partie génération de test et sélection, sera détaillée dans le chapitre 2.

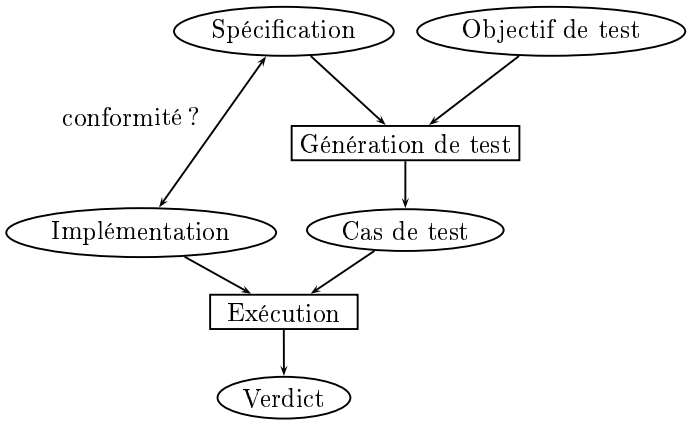


FIG. 1.1 – Architecture du test de conformité avec sélection par objectif de test.

1.1 Modèle des ioLTS

Un des modèles utilisés dans le domaine de la génération automatique de tests est celui des systèmes de transitions étiquetées à entrées/sorties (ioLTS : *Input-Output Labelled Transitions System*) représentant les comportements des spécifications, des implémentations et des cas de test.

1.1.1 Syntaxe

Les ioLTS permettent de distinguer les événements du système contrôlables par l'environnement (ses entrées) des événements seulement observables (ses sorties). Cela caractérise la communication entre l'environnement simulé par le testeur et l'implémentation.

Définition 1.1 (ioLTS) *Un ioLTS est un quadruplet* $M = \langle Q^M, q_0^M, \Lambda^M, \rightarrow_M \rangle$ *pour lequel :*

- Q^M *est un ensemble non vide d'états ;*
- q_0^M *est l'état initial ;*
- Λ^M *est l'alphabet d'actions partitionné en trois ensembles distincts* $\Lambda^M = \Lambda_?^M \cup \Lambda_!^M \cup \Lambda_\tau^M$ *où* $\Lambda_?^M$ *et* $\Lambda_!^M$ *sont respectivement les alphabets d'entrées et de sorties et* Λ_τ^M *est l'alphabet d'actions internes non-observables ;*
- $\rightarrow_M \subseteq Q^M \times \Lambda^M \times Q^M$ *est la relation de transition.*

Le modèle des ioLTS est un raffinement des systèmes de transitions LTS (*Labelled Transition System*, [AN82]) pour lequel les entrées et les sorties sont distinguées. Les entrées seront représentées par des actions suivies du symbole " ?" et les sorties par des actions suivies par " !". De plus, nous distinguerons les actions dites *visibles* de l'alphabet, c'est-à-dire les entrées (contrôlables par l'environnement) et les sorties (observables par celui-ci), des actions internes, non-visibles.

Exemple 1.1 *Considérons le protocole d'un digicode à deux chiffres. L'utilisateur entre deux chiffres. S'ils correspondent au code du dispositif, l'accès à la ressource est autorisé sinon l'accès est définitivement fermé.*

La spécification du digicode est modélisée par un ioLTS figure 1.2. L'utilisateur entre d'abord un chiffre (entrée Digit ?) puis un second chiffre (même entrée) après une action interne τ_1*. Si le code entré est correct, la sortie CodeValide ! est émise suivie d'un certain nombre d'actions internes* τ_2*, sinon AccesFerme ! est produit.*

1.1.2 Notations et opérations

Nous allons maintenant introduire un ensemble de notations pour les ioLTS.

Soit un ioLTS $M = \langle Q^M, q_0^M, \Lambda^M, \rightarrow_M \rangle$. Nous noterons $q \xrightarrow{\lambda}_M q'$ (ou simplement $q \xrightarrow{\lambda} q'$ quand il n'y a pas d'ambiguïté sur l'ioLTS utilisé) pour $(q, \lambda, q') \in \rightarrow_M$, ainsi

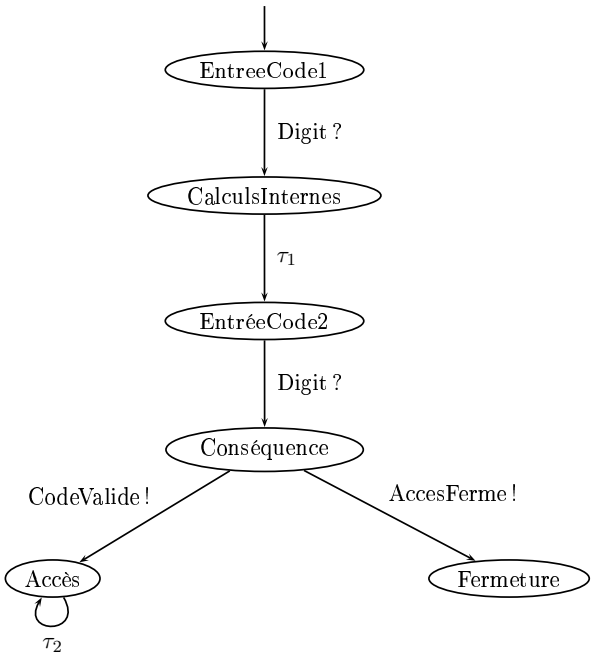


FIG. 1.2 – Spécification du Digicode à 2 chiffres.

que $q \xrightarrow{\lambda}_{\mathrm{M}}$ pour $\exists q' \ : \ q \xrightarrow{\lambda}_{\mathrm{M}} q'$.

Afin de ne parler que des actions visibles du système de transitions, nous utiliserons les notations suivantes pour lesquelles $q, q', q_1, ..., q_n \in Q^{\mathrm{M}}, \lambda, \lambda_1, ..., \lambda_n \in \Lambda^{\mathrm{M}}, \sigma \in \Lambda^{\mathrm{M}*}$:

- $q \stackrel{\varepsilon}{\Rightarrow} q'$ signifie $\exists q_0 = q, q_1, ..., q_n = q', \forall i \in [0, n-1], \exists \tau_i \in \Lambda_\tau$ tels que $q_i \stackrel{\tau_i}{\rightarrow} q_{i+1}$;
- pour $\lambda \in \Lambda^{\mathrm{M}}_? \cup \Lambda^{\mathrm{M}}_!, q \stackrel{\lambda}{\Rightarrow} q'$ signifie $\exists q_1, q_2 : q \stackrel{\varepsilon}{\Rightarrow} q_1 \stackrel{\lambda}{\rightarrow} q_2 \stackrel{\varepsilon}{\Rightarrow} q'$;
- $q \stackrel{\lambda_1 ... \lambda_n}{\Rightarrow} q'$ signifie $\exists q_0, ..., q_n : q = q_0 \stackrel{\lambda_1}{\Rightarrow} q_1 ... \stackrel{\lambda_n}{\Rightarrow} q_n = q'$;
- $\Gamma_{\mathrm{M}}(q) \triangleq \{\lambda \in \Lambda^M \mid q \xrightarrow{\lambda}_M\}$ (l'ensemble des actions tirables en q). On distingue :
  - $In_{\mathrm{M}}(q) \equiv \{\lambda \in \Lambda^{\mathrm{M}}_? | q \stackrel{\lambda}{\Rightarrow}_{\mathrm{M}}\}$: le sous-ensemble des entrées tirables depuis q ;
  - $Out_{\mathrm{M}}(q) \equiv \{\lambda \in \Lambda^{\mathrm{M}}_! | q \stackrel{\lambda}{\Rightarrow}_{\mathrm{M}}\}$: le sous-ensemble des sorties tirables depuis q.

Ces dernières notations peuvent être étendues aux ensembles d'états. Soit $Q \subseteq Q^{\mathrm{M}}$:
- $\Gamma_{\mathrm{M}}(Q) = \bigcup_{q \in Q} \Gamma_{\mathrm{M}}(q)$;
- $In_{\mathrm{M}}(Q) = \bigcup_{q \in Q} In_{\mathrm{M}}(q)$;
- $Out_{\mathrm{M}}(Q) = \bigcup_{q \in Q} Out_{\mathrm{M}}(q)$.

Nous allons maintenant décrire le comportement d'un ioLTS.

Définition 1.2 (Exécution) *Un fragment d'exécution d'un ioLTS M est une séquence alternée d'états et d'actions $q_1\lambda_1 q_2 \dots \lambda_{n-1} q_n$ telle que $\forall i, q_i \xrightarrow{\lambda_i} q_{i+1}$. Une exécution ρ est un fragment d'exécution qui démarre dans un état initial : $\rho = q_0\lambda_0 q_1 \dots \lambda_{n-1} q_n \in q_0.(\Lambda.Q)^*$.*

Soit un ensemble $F \subseteq Q$, l'exécution ρ est acceptée par F si $q_n \in F$. Nous noterons $Runs(M)$ l'ensemble des exécutions de M, et $Runs_F(M)$ l'ensemble des exécutions acceptées par F.

On dit d'un état qu'il est *accessible* s'il est le dernier état d'une exécution. Pour une séquence $\sigma = \lambda_1\lambda_2 \dots \lambda_n$ d'actions valuées, on notera $q \xrightarrow{\sigma} q'$ pour

$$\exists q_1, \dots q_{n+1} \in Q.\ q = q_1 \xrightarrow{\lambda_1} q_2 \xrightarrow{\lambda_2} \cdots q_n \xrightarrow{\lambda_n} q_{n+1} = q'.$$

Pour un ensemble d'états $Q' \subseteq Q$ d'un ioLTS, on notera $q \xrightarrow{\sigma} Q'$ s'il existe un état $q' \in Q'$ tel que $q \xrightarrow{\sigma} q'$.

De plus, on dit d'un état q qu'il est *co-accessible* depuis q' s'il est le premier état d'une exécution menant à q'.

1.1.2.1 Langages définis sur les ioLTS

Nous aurons besoin par la suite de raisonner en termes d'ioLTS mais aussi en termes de langages. Nous distinguerons les langages suivant l'alphabet sur lequel ils portent : si les actions internes (considérées comme des ϵ-transitions d'automates) sont prises en compte, nous parlerons de langage d'un ioLTS, si nous ne considérons que les actions visibles, nous parlerons alors de *traces*. De plus, un ioLTS peut contenir des états particuliers nommés des états accepteurs. Nous noterons un langage :

- $L(M) \triangleq \{\sigma \in \Lambda^{M*} | q_0^M \xrightarrow{\sigma}_M\}$: l'ensemble des séquences d'actions possibles (actions internes incluses) de l'ioLTS $M = \langle Q^M, q_0^M, \Lambda^M, \rightarrow_M \rangle$;
- $L(M, F) \triangleq \{\sigma \in \Lambda^{M*} | q_0^M \xrightarrow{\sigma}_M q, q \in F\}$: pour un ioLTS M d'état initial q_0^M, le langage reconnu pour un ensemble $F \subseteq Q^M$ est l'ensemble des séquences d'actions menant à un état de cet ensemble F.

Lorsque nous nous intéressons au comportement visible du système, nous parlerons alors de *traces*. Une trace peut être définie par rapport à une exécution de la manière suivante.

Définition 1.3 (Trace) *La trace d'une exécution ρ est la projection de ρ sur $\Lambda_! \cup \Lambda_?$, qui sont des actions visibles, observables par l'environnement (contrairement à Λ_τ). L'ensemble des traces d'un ioLTS M sera noté $Traces(M) \triangleq proj_{\Lambda_! \cup \Lambda_?}(Runs(M))$.*

Une trace est reconnue par F si elle est la projection sur $\Lambda_! \cup \Lambda_?$ d'une exécution reconnue. L'ensemble des traces reconnues d'un ioLTS M pour un ensemble F est noté $Traces(M, F) \triangleq proj_{\Lambda_! \cup \Lambda_?}(Runs_F(M))$.

Nous remarquerons que pour un ioLTS M ne contenant pas d'action interne ($\Lambda_\tau^{\mathrm{M}} = \emptyset$), $\mathit{Traces}(M) = L(M)$ et $\mathit{Traces}(M, F) = L(M, F)$ pour $F \subseteq Q^{\mathrm{M}}$.

L'ensemble des états accessibles depuis un état (ou un ensemble d'états) après une trace est noté :

- q *after* $\sigma \triangleq \{q' \in Q \mid q \stackrel{\sigma}{\Rightarrow} q'\}$;
- Q *after* $\sigma \triangleq \bigcup_{q \in Q} q$ *after* σ.

Exemple 1.2 *Dans l'exemple du digicode vu précédemment, nous avons alors les langages suivants :*

$$\begin{aligned} L(S) &= Digit \cdot \tau_1 \cdot Digit \cdot (CodeValide \cdot \tau_2^* + AccesFerme)\,; \\ Traces(S) &= Digit^2 \cdot (CodeValide + AccesFerme)\,; \\ Traces(S,\ Acces) &= Digit^2 \cdot CodeValide. \end{aligned}$$

Préfixe. Soient $\omega, \omega' \in \Lambda^*$ des mots sur un alphabet Λ. On notera $\omega' \leq \omega$ le fait que ω' soit un préfixe de ω. En d'autres termes, $\exists \omega'' \in \Lambda^*$ tel que $\omega = \omega'\omega''$. Si $\omega' \leq \omega$ mais $\omega' \neq \omega$, ω' est un préfixe strict de ω qu'on notera alors $\omega' < \omega$. L'ensemble des préfixes de ω sera noté :

$$\mathit{pref}_{\leq}(\omega) \triangleq \{\omega' \in \Lambda^* |\ \omega' \leq \omega\}.$$

Pour un langage $L \subseteq \Lambda^*$, on note $\mathit{pref}_{\leq}(L)$ la clôture par préfixe de L :

$$\mathit{pref}_{\leq}(L) \triangleq \bigcup_{\omega \in L} \mathit{pref}_{\leq}(\omega).$$

Nous noterons $\mathit{pref}_{<}(\omega)$ l'ensemble de préfixes stricts de ω et $\mathit{pref}_{<}(L)$ l'ensemble des préfixes stricts de L.

Un langage de L est dit *préfixe-clos* si $L = \mathit{pref}_{\leq}(L)$, ce qui est le cas du langage d'un ioLTS $L(M)$ ainsi que de ses traces $\mathit{Traces}(M)$.

1.1.2.2 Définitions classiques

Nous parlerons d'ioLTS *complet* si pour tout état, toutes les actions de l'alphabet sont tirables.

Définition 1.4 (ioLTS complet) *Un ioLTS $M = \langle Q^M, q_0^M, \Lambda^M, \rightarrow_M \rangle$ est dit complet si : $\forall q \in Q^M, \Gamma_M(q) = \Lambda^M$.*

Nous parlerons d'ioLTS complet en entrée (respectivement en sortie) si l'ioLTS est complet sur l'alphabet des entrées (respectivement des sorties) : $\forall q \in Q^{\mathrm{M}}, \mathit{In}_{\mathrm{M}}(q) = \Lambda_?^{\mathrm{M}}$ ($\forall q \in Q^{\mathrm{M}}, \mathit{Out}_{\mathrm{M}}(q) = \Lambda_!^{\mathrm{M}}$).

Un ioLTS est dit *déterministe* si deux transitions de même étiquette ne peuvent être tirées depuis un même état. Nous utiliserons dans cette partie une définition plus forte (voir la définition 1.5), simplifiant la génération de tests. Un ioLTS déterministe ne contiendra alors aucune action interne. La figure 1.3 illustre ce qui n'est pas permis dans un ioLTS déterministe en représentant deux configurations différentes non-déterministes.

Définition 1.5 (ioLTS déterministe) *Un ioLTS* $M = \langle Q^M, q_0^M, \Lambda^M, \rightarrow_M \rangle$ *est déterministe si* $\Lambda_\tau^M = \emptyset$ *et* $\forall q, q_1, q_2 \in Q^M$ *et* $\lambda \in \Lambda^M$, $q \xrightarrow{\lambda} q_1$ *et* $q \xrightarrow{\lambda} q_2$ *implique* $q_1 = q_2$.

FIG. 1.3 – Non-déterminisme.

Le produit synchrone entre deux ioLTS va également être utile à la génération de tests (voir le chapitre 2 à la sous-section 2.2.2). Ce produit permet la synchronisation sur les actions visibles des ioLTS et laisse évoluer indépendamment les actions internes.

Définition 1.6 (Produit synchrone) *Soient* $M_i = \langle Q^i, q_0^i, \Lambda^i, \rightarrow_i \rangle$, $i = 1, 2$ *deux ioLTS, avec* Λ_τ^i *les actions internes de* M_i. *Le produit synchrone* $M_1 \times M_2$ *de* M_1 *et* M_2 *est un ioLTS* $\langle Q, q_0, \Lambda, \rightarrow \rangle$ *tel que :*

- $Q = Q^1 \times Q^2$,
- $q_0 = (q_0^1, q_0^2)$,
- $\Lambda = \Lambda^1 \cap \Lambda^2 \cup (\Lambda_\tau^1 \cup \Lambda_\tau^2)$,
- $\rightarrow$ *est la plus petite relation dans* $Q \times \Lambda \times Q$ *satisfaisant*

$$(q_1, q_2) \xrightarrow{\lambda} \begin{cases} (q_1', q_2') \text{ si } \lambda \in \Lambda \setminus (\Lambda_\tau^1 \cup \Lambda_\tau^2) \ \wedge \ q_1 \xrightarrow{\lambda}_1 q_1' \ \wedge \ q_2 \xrightarrow{\lambda}_2 q_2' \\ (q_1', q_2) \text{ si } \lambda \in \Lambda_\tau^1 \ \wedge \ q_1 \xrightarrow{\lambda}_1 q_1' \\ (q_1, q_2') \text{ si } \lambda \in \Lambda_\tau^2 \ \wedge \ q_2 \xrightarrow{\lambda}_2 q_2' \end{cases}$$

Nous déduisons alors directement de la définition 1.6 le lemme suivant.

Lemme 1.1 $Traces(M_1 \times M_2) = Traces(M_1) \cap Traces(M_2)$.
$Traces(M_1 \times M_2, F_1 \times F_2) = Traces(M_1, F_1) \cap Traces(M_2, F_2)$.

La génération d'un cas de test va également nécessiter des analyses d'ioLTS.

1.1.2.3 Analyses d'accessibilité et de co-accessibilité

Ces analyses, utiles lors de la sélection du cas de test (voir section 2.2), sont **exactes** sur les ioLTS dont le nombre d'états est **fini**.

Analyse avant : calcul des états accessibles. L'objectif de l'analyse avant est de calculer l'ensemble *reach* des états accessibles depuis un état ou un ensemble d'états. On se restreint ici au calcul de $reach(q_0)$, c'est-à-dire l'ensemble des états accessibles depuis l'état initial d'un ioLTS. Formellement,

$$reach(q_0) = \{q' \in Q \mid \exists \sigma \in \Lambda^*,\ q_0 \xrightarrow{\sigma} q'\}.$$

On a donc

$$reach(q_0) = \{q_0\} \cup \left[\bigcup_{i \geq 1} post^i(q_0)\right],$$

avec

$$post(q_0) = \{q \in Q \mid \exists \lambda \in \Lambda,\ q_0 \xrightarrow{\lambda} q\}$$

l'ensemble de états successeurs de q_0 et $post^i(q_0) = post(post^{i-1}(q_0))$.
Nous pouvons également décrire $reach(q_0)$ par un plus petit point fixe (*least fix-point* en anglais, abrégé en *lfp*) :

$$reach(q_0) = lfp(\lambda X.q_0 \cup post(X)),$$

avec, pour $P \subseteq Q$, $post(P)$, l'ensemble des états de Q successeurs des états de P. Nous pouvons également formaliser la restriction sur un alphabet $\Lambda' \subseteq \Lambda$ de l'ensemble des états de Q successeurs des états de P :

$$post_{\Lambda'}(P) = \{q \in Q \mid \exists \lambda \in \Lambda',\ \exists q' \in P,\ q' \xrightarrow{\lambda} q\}.$$

Analyse arrière : calcul des états co-accessibles L'objectif de l'analyse arrière est de calculer l'ensemble *coreach* des états menant à un état ou un ensemble d'états, noté **Accept** par cohérence avec la section 2.2. Nous calculons donc ici $coreach(\mathbf{Accept})$, c'est-à-dire l'ensemble des états co-accessibles depuis l'ensemble **Accept**. Formellement,

$$coreach(\mathbf{Accept}) = \{q \in Q \mid \exists \sigma \in \Lambda^*,\ q \xrightarrow{\sigma} \mathbf{Accept}\}.$$

On a donc

$$coreach(\mathbf{Accept}) = \{\mathbf{Accept}\} \cup \left[\bigcup_{i \geq 1} pre^i(\mathbf{Accept})\right],$$

avec

$$pre(\mathbf{Accept}) = \{q \in Q \mid \exists \lambda \in \Lambda,\ q \xrightarrow{\lambda} \mathbf{Accept}\}$$

l'ensemble des états de Q ayant pour successeur **Accept** et $pre^i(\mathbf{Accept}) = pre(pre^{i-1}(\mathbf{Accept}))$.
Nous pouvons également décrire $coreach(\mathbf{Accept})$ par un plus petit point fixe :

$$coreach(\mathbf{Accept}) = lfp(\lambda X.\mathbf{Accept} \cup pre(X)),$$

avec, pour $P \subseteq Q$, $pre(P)$ l'ensemble des états de Q ayant un successeur dans P. Nous pouvons également formaliser la restriction sur un alphabet $\Lambda' \subseteq \Lambda$ de l'ensemble des états de Q ayant un successeur dans P :

$$pre_{\Lambda'}(P) = \{q \in Q \mid \exists \lambda \in \Lambda',\ \exists q' \in P,\ q \xrightarrow{\lambda} q'\}.$$

1.2 Implémentation, spécification et relation de conformité

Le but du test de conformité est de comparer les comportements de l'implémentation à ceux décrits par sa spécification. Les comportements de la spécification sont modélisés par un ioLTS S. Afin de raisonner formellement sur la conformité entre implémentation et spécification, nous faisons alors l'hypothèse de test que les comportements possibles de l'implémentation sont aussi modélisables par un ioLTS inconnu I.

Nous supposons que l'implémentation est complète en entrée faiblement. Un ioLTS est dit complet en entrée faiblement (*weak input complete* en anglais) si dans tout état, toute entrée est tirable après d'éventuelles actions internes : $\forall q \in Q, In(q \ after \ \epsilon) = \Lambda_?$. De plus, les alphabets visibles de la spécification sont ceux de l'implémentation : $\Lambda^I_? = \Lambda^S_?$ et $\Lambda^I_! = \Lambda^S_!$. Nous supposons ainsi que l'alphabet de la spécification contient toutes les entrées et sorties possibles de l'alphabet de l'implémentation et vice-versa.

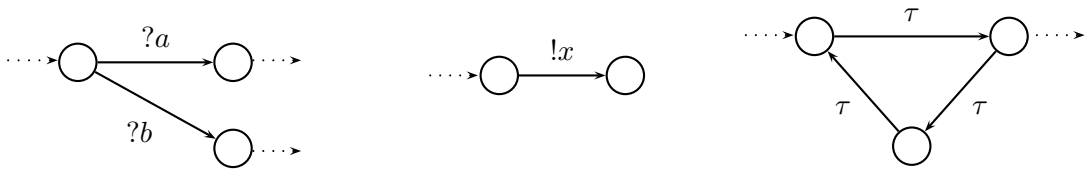
1.2.1 Blocages et suspension

Dans le cadre du test de conformité, il est généralement supposé que l'environnement peut observer non seulement les sorties du système, mais également l'absence de sorties : dans un état donné, le système n'émet aucun événement que l'environnement puisse observer. On parlera de silence (*quiescence*) [Tre99] pour caractériser ce type de blocage. Afin de détecter de tels blocages sur une implémentation boîte noire, nous utiliserons des temporisateurs (*timers*) qui sont armés en attente d'une réponse de l'implémentation. On fait alors l'hypothèse que si la valeur du temporisateur est suffisamment grande, l'expiration du temporisateur est assimilée à un blocage de l'implémentation.

Nous distinguerons différents types de blocage : le blocage de sortie (*outputlock*), le blocage complet (*deadlock*) et le blocage vivant (*livelock*) (voir figure 1.4). Un état est en blocage de sortie s'il ne possède aucune transition tirable d'action interne ou de sortie. Dans ce cas, l'état n'a que des actions tirables par des entrées. Si cet état n'a aucune transition tirable, alors l'état est dans un cas particulier du blocage de sortie, appelé blocage complet. Le *livelock* se produit lorsqu'il existe un cycle d'actions internes. Même si le système continue à être actif en interne, l'environnement ne peut rien observer pendant l'exécution éventuellement infinie de ce cycle. Les définitions de ces trois types de blocage sont données par la définition 1.7.

Définition 1.7 (Blocages) *Soit un ioLTS fini $M = \langle Q^M, q_0^M, \Lambda^M, \rightarrow_M \rangle$, il existe trois types de blocage définis comme suit sur ce modèle :*

- *le blocage de sortie (ou outputlock) : un état q de M est en blocage de sortie si et seulement si $\forall \lambda \in \Lambda^M_\tau \cup \Lambda^M_!, q \not\xrightarrow{\lambda}_M$;*



(a) Blocage de sortie, *Outputlock* (b) Blocage complet *Deadlock* (c) Blocage vivant *Livelock*

FIG. 1.4 – Trois types de blocage

- *le blocage complet (ou deadlock) : un état q de M est en blocage complet si et seulement si* $\forall \lambda \in \Lambda^M, q \not\xrightarrow{\lambda}_M$;
- *le blocage vivant (ou livelock) : un état q de M est en blocage vivant si et seulement si* $\exists \sigma \in (\Lambda_\tau^M)^+, q \xrightarrow{\sigma}_M q$.

Le testeur devrait être capable de distinguer les silences (*quiescences*) spécifiés de ceux qui ne le sont pas. Une spécification peut contenir des blocages de sortie, mais également des *deadlocks* (par exemple, si le système est un calcul ne terminant pas forcément, la non-terminaison de celui-ci peut être un *deadlock* autorisé). Il faut donc détecter les blocages possibles dans les comportements de la spécification afin de distinguer les blocages admissibles. À cet effet, le blocage peut être rendu explicite sur la spécification par une opération appelée suspension. Cette opération transforme un ioLTS M en un ioLTS M^δ, appelé ioLTS suspendu de M.

Définition 1.8 (Suspension d'un ioLTS) *La suspension d'un ioLTS* $M = \langle Q^M, q_0^M, \Lambda^M, \rightarrow_M \rangle$ *donne l'ioLTS* $M^\delta = (Q_{M^\delta}, q_0^{M^\delta}, \Lambda^{M^\delta}, \rightarrow_{M^\delta})$ *avec* $Q_{M^\delta} = Q_M$, $\Lambda^{M^\delta} = \Lambda^M \cup \{\delta\}$ *($\delta \in \Lambda_!^{M^\delta}$, i.e δ est une sortie),* $q_0^{M^\delta} = q_0^M$ *et la relation de transition* $\rightarrow_{M^\delta}$*, obtenue à partir de* $\rightarrow_M$ *par ajout de boucles* $q \xrightarrow{\delta} q$ *pour tous les états q de blocage.*

L'opération de suspension consiste à ajouter dans chaque état de blocage une boucle d'action étiquetée δ. Cette action est considérée comme une sortie puisqu'elle est observable mais non contrôlable. Les traces d'un ioLTS suspendu sont appelées les traces de suspension [Tre99], c'est-à-dire des traces pouvant contenir des silences entre les actions (voir définition 1.9).

Définition 1.9 (Traces suspendues) *Les traces suspendues d'un ioLTS M sont l'ensemble des traces de son automate de suspension M^δ, où δ est considérée comme une action observable :*

$$STraces(M) \triangleq Traces(M^\delta).$$

Exemple 1.3 *La suspension de la spécification du digicode modélisée figure 1.2 est représentée figure 1.5. Nous remarquons des états en blocage de sortie (états EntreeCode1 et EntreeCode2), un état en blocage vivant (Accès) et un état en blocage complet (Fermeture). Une boucle étiquetée par la sortie δ est donc créée sur chacun de ces états.*

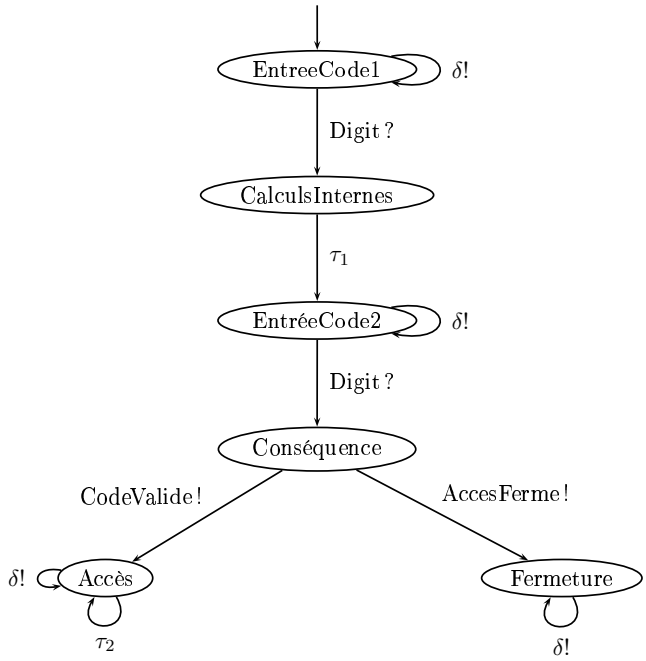


FIG. 1.5 – Suspension de la spécification du Digicode à 2 chiffres.

Une fois la spécification suspendue, toutes les actions visibles autorisées sont mises en évidence. Nous supposerons que l'implémentation contient également des silences que nous pouvons observer.

1.2.2 Relation de conformité ioco

La relation de conformité permet de définir en quoi une implémentation est conforme à sa spécification. Plusieurs sont possibles suivant les observations permises par le test. Les observations considérées ici sont les traces suspendues de la spécification. Par conséquent, la relation de conformité utilisée est celle de Jan Tretmans [Tre96] ioco, dont la définition est la suivante.

Définition 1.10 (ioco) *Soient une spécification $S = \langle Q^S, q_0^S, \Lambda^S, \rightarrow_S \rangle$ et une implémentation $I = \langle Q^I, q_0^I, \Lambda^I, \rightarrow_I \rangle$ complète faiblement en entrée et de même alphabet que S. I est ioco-conforme à sa spécification S, (I ioco S), si*

$$\forall \sigma \in STraces(S),\ Out(q_0^{I^\delta}\ after\ \sigma) \subseteq Out(q_0^{S^\delta}\ after\ \sigma).$$

Une implémentation complète faiblement en entrée et d'alphabet compatible est conforme à sa spécification pour ioco si après toute trace suspendue de S les sorties de l'implémentation (blocages inclus) sont incluses dans celles de la spécification.

Nous pouvons également exprimer la relation de conformité ioco par rapport aux traces, comme cela a été prouvé dans [JMRT04] :

$$I \text{ ioco } S \Leftrightarrow [STraces(S) \cdot \Lambda^{S}_{\delta!} \setminus STraces(S)] \cap STraces(I) = \emptyset, \tag{1.1}$$

avec $\Lambda^{S}_{\delta!} = \Lambda^{S}_{!} \cup \{\delta\}$.

La définition 1.10 prend en compte le fait que la spécification peut partiellement décrire le comportement en entrée du système. En effet, l'implémentation a le droit d'accepter toutes les entrées, même celles qui ne sont pas admissibles dans la spécification, sans que cela n'induise une non-conformité entre la spécification et l'implémentation. Dans ce contexte, la spécification peut être vue comme une modélisation d'un ensemble de services que le système doit réaliser. Une implémentation de ce système peut réaliser plus de services que ceux initialement attendus. Ces nouvelles fonctionnalités ne doivent en rien influer sur la conformité.

Exemple 1.4 *Différentes implémentations sont données à la figure 1.6. Nous avons représenté par l'action ♯? l'ensemble des entrées qui ne sont pas portées par d'autres transitions. L'implémentation I_1 qui permet la trace $Digit? \cdot Digit? \cdot AccesFerme!$ est conforme à la spécification (même figure à gauche) puisque cette trace est une trace de la spécification. Par contre, l'implémentation I_2 n'est pas conforme à S car sa trace ($Digit? \cdot AccesFerme!$) n'est pas admissible par la spécification. Enfin, la trace $Digit? \cdot Digit? \cdot Digit?$ (implémentation I_3) ne permet pas de conclure que l'implémentation est non-conforme à la spécification. L'entrée $Digit?$ n'est pas prévue par la spécification après la trace $Digit? \cdot Digit?$, mais la spécification pouvant n'être que partielle, toute entrée non-spécifiée de l'implémentation est considérée comme conforme.*

Afin de formaliser le test de conformité d'une IUT par rapport à sa spécification, nous allons donner une définition formelle d'un cas de test et de son exécution sur l'implémentation en section 1.3. Nous allons ensuite décrire les verdicts produits et les propriétés attendues des cas de tests.

1.3 Cas de test, exécution et propriétés

L'implémentation sous test est un système réactif boîte noire dont on ne peut pas vérifier le contenu mais dont on peut tester les réactions via ses interactions avec un autre système réactif, le cas de test. La relation de conformité ioco étant basée sur les traces de la spécification et de l'implémentation, le cas de test doit pouvoir observer les traces de l'IUT. Le cas de test est donc modélisé, comme la spécification et l'implémentation, par un ioLTS dont la définition est donnée en sous-section 1.3.1. Seront ensuite définies

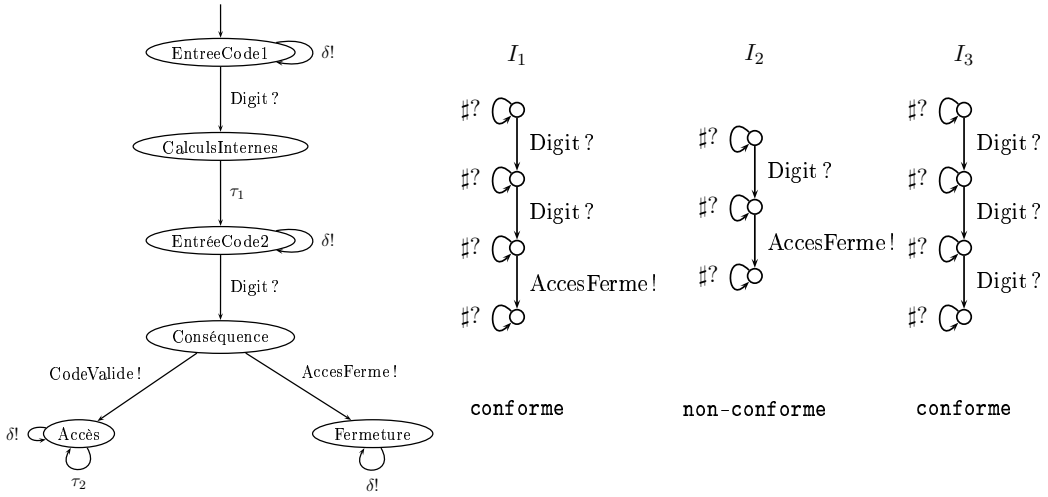


FIG. 1.6 – Conformité d'implémentations par rapport à la spécification S du Digicode.

l'exécution de ce cas de test sur l'implémentation ainsi que la notion de verdict (sous-section 1.3.2). Enfin, nous définirons les propriétés que nous attendons des cas de test générés en sous-section 1.3.3.

1.3.1 Cas de test

Les implémentations réactives considérées ici ne sont pas toujours contrôlables par l'environnement : elles peuvent produire différentes sorties pour une même entrée (on parle alors de *non-déterminisme observable*). Les cas de test décrivant les interactions entre testeur et implémentation doivent donc en tenir compte. Comme la spécification et l'implémentation, nous modélisons un cas de test par un ioLTS (TC) mais celui-ci est muni de verdicts et de quelques propriétés de structure supplémentaires. Le cas de test contient trois ensembles d'états sans successeur **Pass**, **Fail** et **Inconc** (pour inconclusif) qui caractérisent les verdicts. Il n'émet que des entrées de la spécification ($\Lambda_!^{TC} = \Lambda_?^S$) et doit s'attendre à recevoir toutes les sorties possibles de l'implémentation ($\Lambda_?^{TC} = \Lambda_!^{I^\delta}$), en d'autres termes il est complet en entrée. Les états dans **Fail** et **Inconc** ne sont atteints que par des entrées et de tout état, un verdict est accessible.

Définition 1.11 *Un cas de test est un ioLTS déterministe* $TC = \langle Q^{TC}, q_0^{TC}, \Lambda^{TC}, \rightarrow_{TC} \rangle$ *muni de trois sous-ensembles distincts d'états sans successeur* **Pass**, **Fail** *et* **Inconc** *de* Q^{TC} *caractérisant les verdicts. Un cas de test est défini comme suit :*

- *l'alphabet du cas de test est le miroir de celui de la spécification suspendue (et donc de l'implémentation) :* $\Lambda_?^{TC} = \Lambda_!^{S^\delta}$ *et* $\Lambda_!^{TC} = \Lambda_?^{S^\delta}$ *;*
- TC *est complet en entrée excepté dans les états de verdict puisque le cas de test ne doit refuser aucune sortie de l'implémentation :*

$$\forall q \in Q^{TC} \setminus (\mathbf{Pass} \cup \mathbf{Inconc} \cup \mathbf{Fail}),\ \forall \lambda \in \Lambda_?^{TC},\ q \xrightarrow{\lambda}_{TC} ;$$

– *les états* **Fail** *et* **Inconc** *ne sont directement accessibles que par des entrées :*

$$\forall (q, \lambda, q') \in \rightarrow_{TC},\ q' \in \mathbf{Inconc} \cup \mathbf{Fail} \Rightarrow \lambda \in \Lambda_?^{TC}\ ;$$

– *de tout état, un état de verdict est accessible :*

$$\forall q \in Q^{TC},\ \exists \sigma \in \Lambda^{TC*},\ q \xrightarrow{\sigma}_{TC} \mathbf{Pass} \cup \mathbf{Inconc} \cup \mathbf{Fail}.$$

1.3.2 Exécution

L'exécution d'un cas de test TC sur une implémentation I est modélisée par la composition parallèle $I^\delta || TC$ avec synchronisation sur les actions communes. Nous en donnons une définition formelle dans la définition 1.12.

Définition 1.12 *Soient les ioLTS* $I^\delta = (Q_I, q_0^I, \Lambda_! \cup \{\delta\} \cup \Lambda_? \cup \Lambda_\tau^I, \rightarrow_{I^\delta})$ *et* $TC = (Q_{TC}, q_0^{TC}, \Lambda_? \cup \Lambda_! \cup \{\delta\}, \rightarrow_{TC})$. *L'exécution du cas de test* TC *sur l'implémentation* I^δ *est modélisée par la composition parallèle* $TC||I^\delta = \langle Q_I \times Q_{TC},\ q_0^I \times q_0^{TC},\ \Lambda_! \cup \{\delta\} \cup \Lambda_?,\ \rightarrow_{TC||I^\delta} \rangle$, *avec* $\rightarrow_{TC||I^\delta}$ *défini par la règle :*

$$\frac{\lambda \in \Lambda_! \cup \{\delta\} \cup \Lambda_? \quad q_1 \overset{\lambda}{\Rightarrow}_{I^\delta} q_2 \quad q_1' \overset{\lambda}{\rightarrow}_{TC} q_2'}{(q_1, q_1') \overset{\lambda}{\rightarrow}_{TC||I^\delta} (q_2, q_2')}.$$

Les traces de la composition parallèle de deux ioLTS est l'intersection des traces de chacun des ioLTS, nous avons donc :

$$\mathit{Traces}(TC||I^\delta) = \mathit{Traces}(TC) \cap \mathit{Traces}(I^\delta).$$

Toute exécution d'un cas de test sur une implémentation émettra un verdict. Cependant, à cause du non-déterminisme observable de l'implémentation, deux exécutions d'un même cas de test peuvent produire deux verdicts différents. Nous parlerons donc de possibilité de verdict : possibilité de rejet de l'implémentation, possibilité de passer le test et possibilité de produire un inconclusif.

Définition 1.13 (Possibilités de verdicts) *Soient une implémentation* I *et un cas de test* TC *:*

$$\begin{array}{lcl} TC\ \mathit{mayfail}\ I & \triangleq & \exists \sigma \in \mathit{Traces}(TC||I^\delta),\ \mathit{verdict}(\sigma) = \mathit{Fail} \\ TC\ \mathit{maypass}\ I & \triangleq & \exists \sigma \in \mathit{Traces}(TC||I^\delta),\ \mathit{verdict}(\sigma) = \mathit{Pass} \\ TC\ \mathit{mayinconc}\ I & \triangleq & \exists \sigma \in \mathit{Traces}(TC||I^\delta),\ \mathit{verdict}(\sigma) = \mathit{Inconc} \end{array}$$

1.3.3 Propriétés

Nous avons défini dans la sous-section précédente les verdicts émis par l'exécution du cas de test sur l'implémentation, mais nous n'avons pas fait le lien entre ces verdicts et la relation de conformité. Intuitivement, le verdict *Fail* implique la non-conformité (propriété de non-biais) et inversement, toute non-conformité devrait être signalée par le verdict *Fail* (propriété d'exhaustivité). Les verdicts *Pass* et *Inconc* n'ont pas de lien

direct avec la relation de conformité mais avec la sélection par objectif de test. Le verdict *Pass* signifie que le test n'a pas échoué sur cette exécution et que l'objectif de test est atteint. Le verdict *Inconc* signifie que pour cette exécution, l'objectif de test ne peut plus être atteint et que jusqu'à présent, aucune non-conformité n'a été détectée.

Définition 1.14 (Non-biais, exhaustivité et complétude) *Soient une implémentation I, une spécification S et un cas de test TC.*

(1) Un cas de test TC est dit non-biaisé pour S et ioco si et seulement si :

$$\forall I,\ I \text{ ioco } S \ \Rightarrow\ \neg(TC \text{ mayfail } I).$$

Une suite de test est dite non-biaisée si tous ses cas de test sont non-biaisés.

(2) Une suite de test TS est dite exhaustive pour S et ioco si et seulement si :

$$\forall I,\ \neg(I \text{ ioco } S) \ \Rightarrow\ \exists TC \in TS,\ TC \text{ mayfail } I.$$

(3) Une suite de test est dite complète si elle est non-biaisée et exhaustive.

Comme expliqué dans l'introduction, la propriété de non-biais ne suffit pas à produire des tests satisfaisants puisque ceux-ci peuvent ne pas rejeter une implémentation non-conforme. De même, la propriété d'exhaustivité seule est insuffisante car une implémentation conforme peut faire échouer le test. Or, il est impossible d'avoir une suite de test complète puisqu'une suite de test (finie) ne peut pas être exhaustive. Prenons par exemple une spécification avec un cycle dont l'ensemble des traces est du type $a \cdot b^*$. Un cas de test contenant ce même cycle peut être généré afin de tester l'ensemble des traces $a \cdot b^*$. Il faudrait alors exécuter une infinité de fois ce cas de test afin de tester toutes les traces possibles reconnues par la spécification (ab, abb, $abbb$, etc.) et ainsi obtenir l'exhaustivité[1]. Une façon de limiter ce problème consiste à obtenir l'exhaustivité de la méthode permettant de générer le cas de test. Ainsi, si l'implémentation est non-conforme, nous pouvons toujours produire un cas de test la rejetant.

Nous pouvons déduire des définitions de la relation de conformité ioco 1.10 (plus précisément de l'équation 1.1) et des propriétés de non-biais et d'exhaustivité 1.14 les propriétés sur les traces suivantes ([JJR07]) :

$$\text{TS est non-biaisée} \quad \Leftrightarrow \quad \bigcup_{TC \in TS} \mathit{Traces}(TC, \mathit{Fail}) \subseteq \mathit{STraces}(S) \cdot \Lambda^{\mathrm{S}}_{\delta!} \setminus \mathit{STraces}(S)\,;$$

$$\text{TS est exhaustive} \quad \Leftrightarrow \quad \bigcup_{TC \in TS} \mathit{Traces}(TC, \mathit{Fail}) \supseteq \mathit{STraces}(S) \cdot \Lambda^{\mathrm{S}}_{\delta!} \setminus \mathit{STraces}(S).$$

Ces propriétés sont dues au fait que l'ensemble des traces $\mathit{STraces}(S) \cdot \Lambda^{\mathrm{S}}_{\delta!} \setminus \mathit{STraces}(S)$ constitue l'ensemble des traces non-conformes minimales. Cet ensemble de traces correspond aux traces du **testeur canonique** menant au verdict de non-conformité (voir

[1] Ceci n'est vrai que si l'implémentation est déterministe ou non-déterministe mais à équité bornée. L'hypothèse de l'équité bornée est satisfaite si, dans tout état, une implémentation non-déterministe fournit toutes les différentes sorties possibles sur un nombre d'exécutions borné.

sous-section 2.1.2). Le testeur canonique permet, pour une spécification et une relation données, de détecter toute implémentation non-conforme. Cette notion a été introduite par Brinksma dans [Bri88].

Nous avons ainsi un lien direct entre les propriétés de non-biais et d'exhaustivité des suites de test et les inclusions de traces des cas de test et de la spécification. Afin de générer des cas de test satisfaisant ces propriétés pour la relation de conformité ioco, nous construirons ce que nous définirons comme le testeur canonique puis nous effectuerons une sélection dessus. Ceci est détaillé au chapitre 2.

Chapitre 2

Génération de tests basée sur les systèmes finis de transitions

Une méthode de génération automatique de tests basée sur la relation de conformité ioco et sur le modèle des ioLTS finis est présentée dans ce chapitre. Cette méthode découle des travaux de Jan Tretmans et de ses collègues de l'Université de Twente, du point de vue des modèles et de la théorie du test (voir chapitre 1). Elle a été développée au sein de l'équipe Pampa puis de l'équipe VerTeCS à Rennes, par Thierry Jéron et ses collègues ainsi que par l'équipe Vérimag à Grenoble. L'architecture de cette méthode est représentée à la figure 2.1.

Après une présentation des opérations nécessaires à la génération du testeur canonique dans la section 2.1 (déterminisation et complétion en sortie), le mode de sélection par objectif de test sera détaillé en section 2.2 (modélisation de l'objectif de test et sélection).

2.1 Génération du testeur canonique

Toute la génération de tests est basée sur la spécification. Afin que les cas de test générés à partir de celle-ci puissent être exécutés efficacement sur l'implémentation, il est nécessaire que ceux-ci soient déterministes et complets en entrée. De plus, il est préférable que toutes les actions permises par la spécification soient représentées, les blocages y compris. Les opérations de déterminisation et de complétion de la spécification se font donc sur la spécification suspendue (voir en sous-section 1.2.1). Le résultat de ces opérations forme ce que nous appellerons le testeur canonique. Le principe d'un testeur canonique est de pouvoir détecter toute implémentation non-conforme par rapport à une relation donnée. Cette relation est ici la relation de conformité ioco : après toute trace de S, l'implémentation ne peut émettre que des sorties autorisées par la spécification. La complétion en sortie permettant de mettre en évidence, sur la spécification, les sorties menant à une non-conformité, le testeur canonique $Can(\mathcal{S}) = \Sigma^{!}(det(S^{\delta}))$ permet alors de détecter toute implémentation non-conforme.

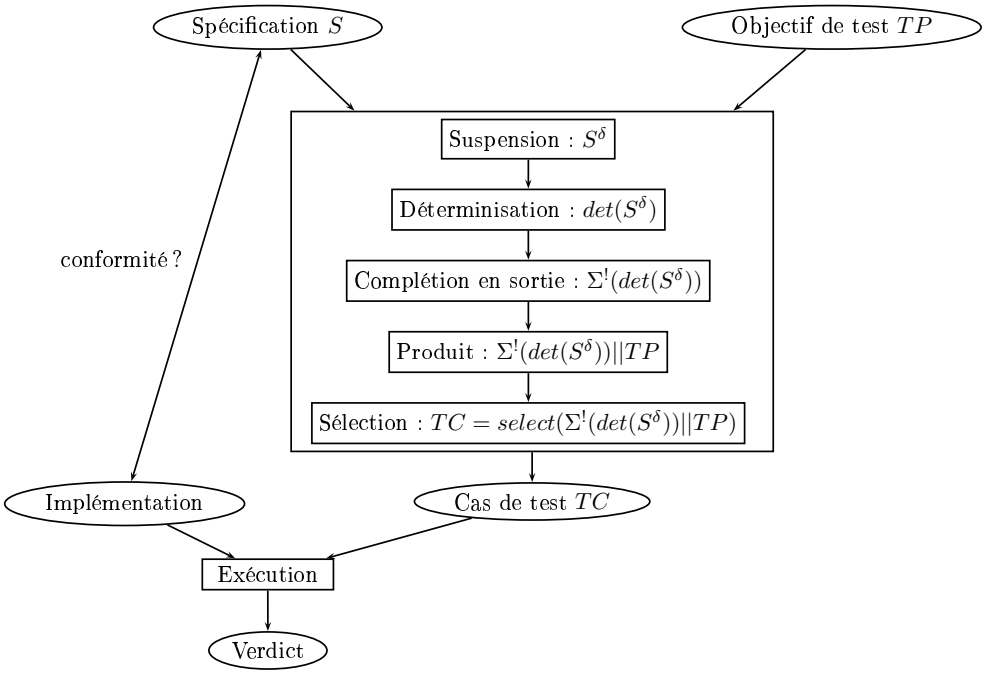


FIG. 2.1 – Architecture de la génération de test et de la sélection par objectif de test

2.1.1 Déterminisation

Le test de conformité se basant sur l'observation des comportements visibles, les actions internes ne nous intéressent plus. Il est donc nécessaire de déterminiser la spécification afin que le cas de test généré soit déterministe. Un cas de test permet de tester un comportement du système. Si celui-ci n'était pas déterministe, une exécution de ce cas de test pourrait rendre un verdict contraire à une autre exécution de ce même cas de test (pour une même trace de TC). La définition d'un ioLTS déterministe a été donnée au chapitre précédent en définition 1.5.

Dans le cas général, à partir d'un ioLTS M quelconque, nous pouvons construire l'ioLTS déterministe correspondant $det(M)$ dont le comportement observable est le même que celui de M : $Traces(M) = Traces(det(M))$.

Définition 2.1 (Déterminisation) *L'ioLTS déterministe d'un ioLTS* $M = \langle Q, q_0, \Lambda, \rightarrow \rangle$ *est* $det(M) = \langle 2^Q, q_0 \text{ after } \varepsilon, \Lambda \setminus \Lambda_\tau, \rightarrow_{det} \rangle$ *dont la relation de transition* $\rightarrow_{det}$ *est la plus petite relation définie par :* $P \xrightarrow{\lambda}_{det} P'$ *si* $P' = P$ *after* λ, *pour* $P, P' \in 2^Q$ *et* $\lambda \in \Lambda_? \cup \Lambda_!$.

Exemple 2.1 *La déterminisation de la spécification suspendue du digicode est représentée figure 2.2. Les transitions d'actions internes ont été supprimées et pour plus de*

simplicité, nous avons renommé le méta-état {CalculsInternes, EntréeCode2} par EntréeCode2.

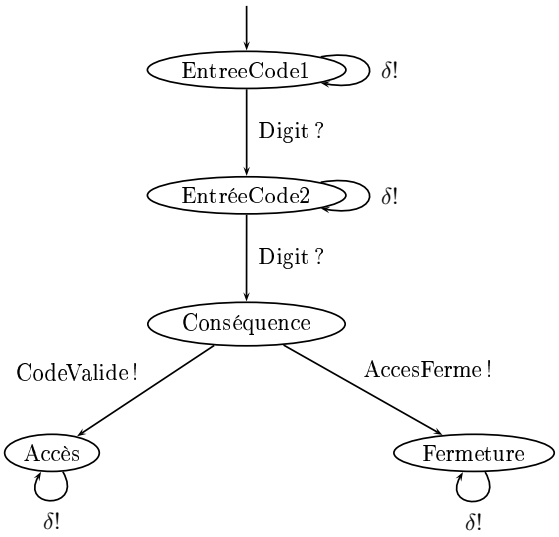


FIG. 2.2 – Déterminisation de la spécification suspendue du Digicode à 2 chiffres.

Une fois la spécification suspendue et déterminisée, l'étape suivante consiste à rendre explicite les traces non-conformes.

2.1.2 Complétion en sortie

Le test de conformité à la ioco consiste à détecter si une sortie de l'implémentation n'est pas autorisée par la spécification. Pour cela, le cas de test doit être capable de distinguer les sorties permises par la spécification de celles qui ne sont pas autorisées et mènent donc à la non-conformité. Si une sortie non-autorisée est observée, alors le cas de test doit rejeter l'implémentation en produisant le verdict de non-conformité *Fail*. Il est donc nécessaire de compléter la spécification par les sorties non-permises par celle-ci. Ces sorties mèneront à un ensemble d'états particuliers **Fail** représentant le verdict de non-conformité. Cette opération est appelée la complétion en sortie (voir définition ci-dessous).

Définition 2.2 (Complétion en sortie) *Soit un ioLTS déterministe suspendu* $M = \langle Q^M, q_0^M, \Lambda^M, \rightarrow_M \rangle$ *avec* $\Lambda^M = \Lambda_?^M \cup \Lambda_!^M \cup \{\delta\}$. *Après complétion par l'état* **fail**, *on obtient un ioLTS déterministe suspendu* $\Sigma^!(M) = \langle Q, q_0, \Lambda, \rightarrow \rangle$ *tel que :*

- $Q = Q_M \cup \{\textbf{fail}\}$;
- $q_0 = q_0^M$;
- $\Lambda = \Lambda^M$;
- $\rightarrow = \rightarrow_M \ \cup \{q \xrightarrow{\lambda} \textbf{fail} \mid \lambda \in \Lambda_! \cup \{\delta\},\ q \not\xrightarrow{\lambda}_M\}$.

Remarquons que si la spécification n'était pas déterministe, la complétion en sortie pourrait rendre des traces autorisées par la spécification non-conformes. En effet, cette opération pourrait ajouter à tort des transitions menant à **Fail** : si, après une séquence d'actions visibles σ, on arrive en q et $q \overset{\lambda!}{\not\to} \wedge q \overset{\tau}{\to} q' \overset{\lambda!}{\to} q''$ (seule une transition portant une action interne est permise en q) alors la complétion en sortie ajouterait la transition $q \overset{\lambda!}{\to}$ **fail**, ce qui rendrait l'observation de $\lambda!$ après σ non-conforme alors que cette sortie est autorisée. Il est donc important de n'appliquer cette opération que sur une spécification déterministe.

Exemple 2.2 *La complétion en sortie de la spécification suspendue déterministe du digicode est représentée figure 2.3. Pour plus de clarté dans la représentation de l'ioLTS nous ne distinguerons pas chacun des états* **fail***, nous les noterons tous* **Fail**. *Des transitions de sorties ont été tirées vers* **Fail** *depuis les états pour lesquels ces sorties n'étaient pas spécifiées.*

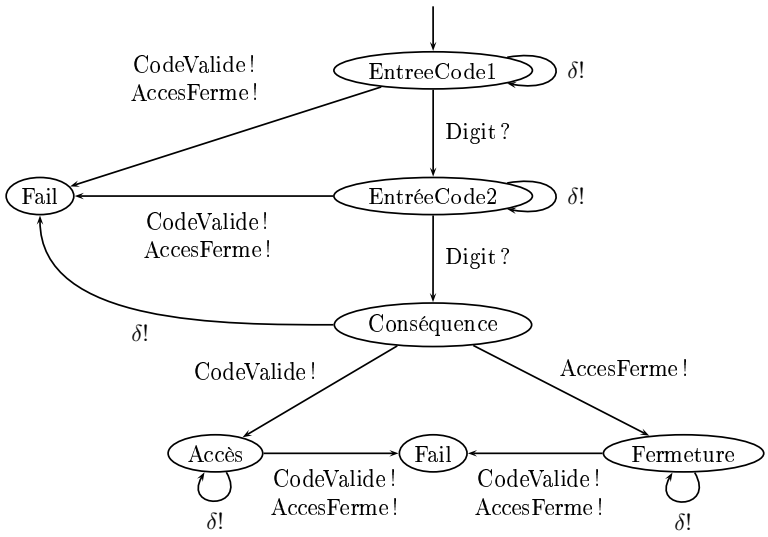


FIG. 2.3 – Complétion en sortie de la spécification suspendue déterministe du Digicode.

Lorsque la spécification déterministe suspendue S^δ est complétée en sortie, l'ensemble de ses traces correspond alors à l'ensemble suivant :

$$Traces(\Sigma^!(det(S^\delta))) = pref_{\leq}(STraces(S) \cdot \Lambda^{\mathrm{S}}_{\delta!}).$$

De plus, grâce à la complétion en sortie d'un ioLTS déterministe suspendu, l'ensemble des traces menant à l'ensemble d'états **Fail** correspond à l'ensemble des traces menant à la non-conformité :

$$Traces(\Sigma^!(det(S^\delta)), \mathbf{Fail}) = STraces(S) \cdot \Lambda^{\mathrm{S}}_{\delta!} \setminus STraces(S).$$

Nous obtenons donc la relation suivante :

$$I \text{ ioco } S \Leftrightarrow \mathit{Traces}(\Sigma^{!}(\mathit{det}(S^{\delta})), \mathbf{Fail}) \cap \mathit{STraces}(I) = \emptyset.$$

Nous pouvons alors définir la spécification déterministe suspendue complétée en sortie comme notre testeur canonique puisqu'elle permet de détecter toute implémentation non-conforme :

$$\Sigma^{!}(\mathit{det}(S^{\delta})) = \mathit{Can}(\mathcal{S}).$$

Nous pouvons donc réécrire la relation précédente comme suit :

$$I \text{ ioco } S \Leftrightarrow \mathit{Traces}(\mathit{Can}(\mathcal{S}), \mathbf{Fail}) \cap \mathit{STraces}(I) = \emptyset.$$

Le testeur canonique est un cas de test, mais celui-ci ne privilégie aucun comportement particulier. Or, nous préférons sélectionner des comportements précis à tester et ainsi optimiser le nombre de tests à effectuer, grâce, par exemple, à un objectif de test.

2.2 Sélection par objectif de test

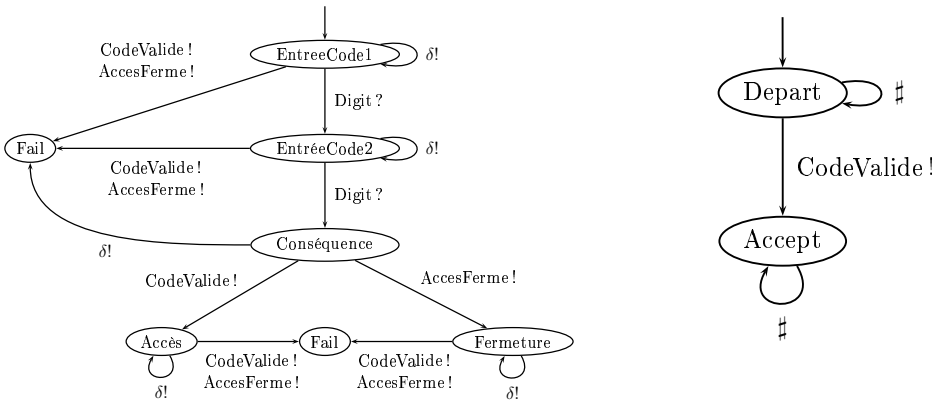
La sélection a pour but de guider les tests pour tester des comportements précis. Elle permet également d'obtenir un verdict quel que soit le résultat du test (échec ou succès). Un objectif de test, présenté en sous-section 2.2.1, permet de représenter un comportement à tester (ou à éviter). Le produit synchrone (voir sous-section 2.2.2) est ensuite effectué entre cet objectif de test et la spécification (déterminisée, suspendue et complétée en sortie). Enfin, la sélection est effectuée sur ce produit par calcul des états accessibles depuis l'état initial et co-accessibles depuis le verdict attendu (voir sous-section 2.2.3).

2.2.1 Objectif de test

Un objectif de test est un ioLTS particulier dont la définition est donnée définition 2.3. Il modélise des comportements intéressants précis de la spécification et une condition d'arrêt du test.

Définition 2.3 (Objectif de test) *Un objectif de test TP (pour Test Purpose) est un ioLTS $\langle Q^{TP}, q_0^{TP}, \Lambda^{TP}, \rightarrow_{TP} \rangle$ complet et déterministe muni d'un ensemble d'états puits particuliers :* **Accept** $\subseteq Q^{TP}$. *Un état q est un état puits s'il boucle pour toute action ($\forall \lambda \in \Lambda^{TP}, q \xrightarrow{\lambda}_{TP} q$). L'objectif de test est défini par rapport au comportement visible d'une spécification, son alphabet est donc l'alphabet visible de la spécification, blocages inclus : $\Lambda^{TP} = \Lambda^{S}_{?} \cup \Lambda^{S}_{!} \cup \{\delta\}$.*

Exemple 2.3 *Un objectif de test pour la spécification du digicode est représenté figure 2.4. Nous voulons tester si l'implémentation peut émettre la sortie $CodeValide!$. Nous avons représenté par l'action ♯ l'ensemble des actions (incluant l'action spécifique de blocage δ) qui ne sont pas portées par d'autres transitions.*



(a) Spécification $\Sigma^!(det(S^\delta))$ (b) Objectif de test TP

FIG. 2.4 – La spécification suspendue déterministe complète en sortie du Digicode et son objectif de test.

L'ensemble des traces de l'objectif de test est l'ensemble des traces possibles sur l'alphabet de la spécification ($\mathit{Traces}(TP) = \Lambda_S^*$), mais pour un sous-ensemble de ces traces (représentant un comportement précis), un état de l'ensemble **Accept** est atteint. L'ensemble de ces traces acceptées est donné par $\mathit{Traces}(TP, \mathbf{Accept}^{TP})$.

Le langage des traces acceptées est dit clos par extension : puisque les états de l'ensemble **Accept** sont des états puits, tout prolongement d'une séquence acceptée est accepté. Nous nous intéresserons principalement par la suite à l'ensemble des traces *minimales* acceptées, ce qui nous permet d'arrêter la construction d'un cas de test dès que le préfixe d'une trace de la spécification est accepté.

Afin de connaître les traces du testeur canonique pouvant satisfaire l'objectif de test, le produit synchrone de ces deux objets est réalisé.

2.2.2 Produit synchrone

Le produit synchrone (voir définition 1.6) va permettre de sélectionner les traces du testeur canonique acceptées par l'objectif de test. Dans notre cas, le produit synchrone se fait entre deux objets déterministes (il n'y a aucune action interne). De plus, l'objectif a un ensemble d'états important **Accept** dont il faut garder la visibilité. Le produit synchrone $\mathit{Can}(\mathcal{S}) \times TP$ entre une spécification suspendue déterministe $\mathit{Can}(\mathcal{S}) = \langle Q^{Can(\mathcal{S})}, q_0^{Can(\mathcal{S})}, \Lambda^{Can(\mathcal{S})}, \rightarrow_{Can(\mathcal{S})} \rangle$ et un objectif de test $TP = \langle Q^{TP}, q_0^{TP}, \Lambda^{TP}, \rightarrow_{TP} \rangle$ est un donc ioLTS $\langle Q, q_0, \Lambda, \rightarrow \rangle$ tel que :

- $Q = Q^{Can(\mathcal{S})} \times Q^{TP}$ avec $\mathbf{Fail} = \mathbf{Fail}^{Can(\mathcal{S})} \times Q^{TP}$ et $\mathbf{Accept} = (Q^{Can(\mathcal{S})} \setminus \mathbf{Fail}^{Can(\mathcal{S})}) \times \mathbf{Accept}^{TP}$, [1]

[1] S'il y avait un état $(\mathbf{Fail}, \mathbf{Accept})$, alors celui-ci serait considéré comme un état **Fail**. En effet,

- $q_0 = (q_0^{Can(\mathcal{S})}, q_0^{TP})$,
- $\Lambda = \Lambda^{Can(\mathcal{S})} (= \Lambda^{TP})$ avec $\Lambda_\tau^{Can(\mathcal{S})} = \emptyset$,
- $\rightarrow$ est la plus petite relation dans $Q \times \Lambda \times Q$ satisfaisant $(q_1, q_2) \stackrel{\lambda}{\rightarrow} (q_1', q_2')$ si $q_1 \stackrel{\lambda}{\rightarrow}_{Can(\mathcal{S})} q_1' \ \wedge \ q_2 \stackrel{\lambda}{\rightarrow}_{TP} q_2'$.

Le produit préserve tous les comportements possibles du testeur canonique puisque l'objectif est complet :

$$STraces(Can(\mathcal{S}) \times TP) = STraces(Can(\mathcal{S})).$$

De plus, le produit de deux ioLTS déterministes est déterministe.

Nous remarquons également que les traces acceptées par le produit sont les traces du testeur canonique acceptées par l'objectif de test. Nous avons donc :

$$Traces(Can(\mathcal{S}) \times TP, \mathbf{Accept}) = STraces(Can(\mathcal{S})) \cap Traces(TP, \mathbf{Accept}^{TP}).$$

Nous appellerons graphe de test l'ioLTS résultant des opérations :

$$test = Can(\mathcal{S}) \times TP = \langle Q^{test}, q_0^{test}, \Lambda_!^{S^\delta} \cup \Lambda_?^{S^\delta}, \rightarrow_{test} \rangle.$$

Exemple 2.4 *Le produit entre la spécification suspendue déterministe complète en sortie $\Sigma^!(det(S^\delta))$ et l'objectif de test TP est représenté figure 2.5. Ce produit a pour conséquence la composition des états de la spécification avec ceux de l'objectif. Dans notre représentation, nous n'avons pas composé l'état* **Fail**. *Nous avons fait ce choix car la complétion peut être faite après le produit sans que cela n'ait d'incidence.*

2.2.3 Sélection

Le but de la sélection est d'extraire les traces de la spécification acceptées par l'objectif de test. Pour cela, nous allons extraire du produit un ioLTS qui ne conservera que les états utiles à une trace acceptée. Les états utiles sont les états qui peuvent être parcourus de l'état initial à l'état accepteur. Nous ne nous intéresserons ici qu'aux traces minimales, c'est-à-dire les traces s'arrêtant au premier état **Accept** rencontré.

La sélection va consister à calculer l'ensemble des états co-accessibles depuis **Accept**, et à supprimer, suivant la nature de l'événement y menant, les sous-graphes ne pouvant pas atteindre **Accept**. Une analyse d'accessibilité depuis l'état initial est ensuite effectuée afin de raffiner le sous-graphe obtenu. Cette sélection se fait sur un ioLTS avec un nombre d'états finis grâce aux analyses d'accessibilité et de co-accessibilité exactes vues en 1.1.2.3.

---

l'information la plus importante est la non-conformité. Le fait qu'une trace non-conforme satisfasse l'objectif de test ne nous intéresse pas.

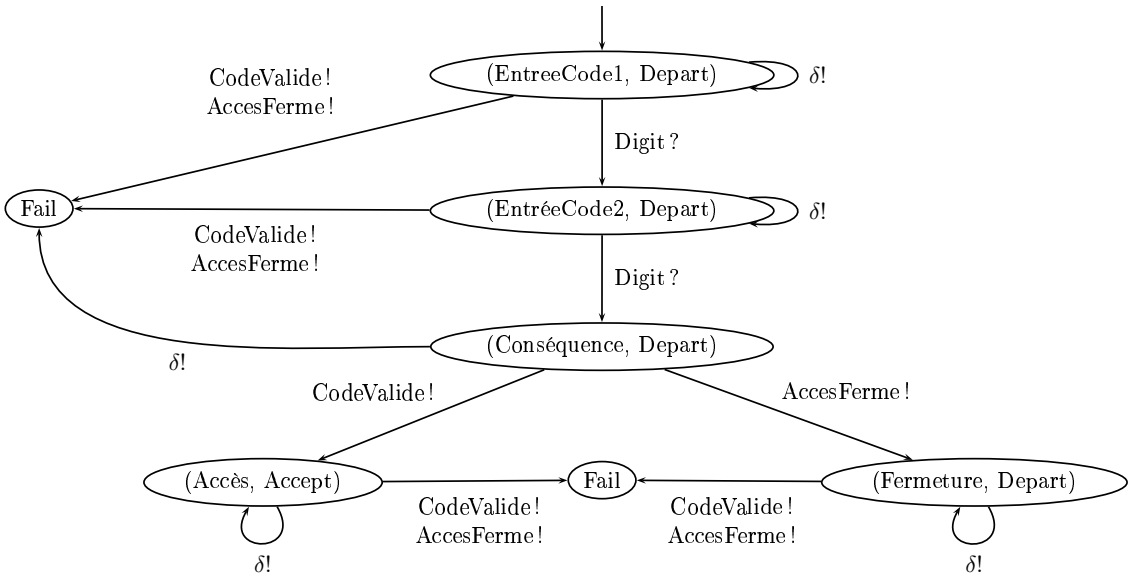


FIG. 2.5 – Produit entre $\Sigma^{!}(det(S^{\delta}))$ et l'objectif de test TP.

De plus, nous constatons que les préfixes des traces menant à **Accept** sont par définition les traces co-accessibles depuis cet état :

$$pref_{\leq}(Traces(test, \mathbf{Accept})) = Traces(test, coreach(\mathbf{Accept})),$$

ce qui va nous permettre de signaler, grâce à un nouveau verdict *Inconc*, qu'une exécution ne pourra pas satisfaire l'objectif de test, si celle-ci sort des traces co-accessibles depuis **Accept**.

Construction d'un cas de test Un cas de test (voir définition 1.11) est construit à partir du graphe de test *test* par l'opération miroir (l'alphabet des entrées devient celui des sorties et réciproquement) et par le calcul des états co-accessibles et accessibles. Les sous-graphes ne pouvant pas mener à **Accept** (**Pass**) sont supprimés en fonction de la nature de l'événement menant au sous-graphe :

- si cet événement est une sortie du testeur (une entrée de la spécification), alors cette transition ainsi que le sous-graphe sont supprimés. Intuitivement, le testeur contrôle ses sorties, il peut donc décider de ne pas stimuler l'implémentation par une sortie s'il sait qu'elle ne mènera pas au verdict *Pass*.
- si cet événement est une entrée du testeur (qui ne mène pas directement à **Fail**), alors seule cette transition est conservée (le sous-graphe est supprimé). La destination de cette transition est alors un nouvel état appelé **Inconc** signifiant que **Accept** ne peut plus être atteint mais que la conformité n'est pas violée. Dans cette situation, le verdict sera donc *Inconc* (*inconclusif*).

Définition 2.4 (Construction d'un cas de test) *Soit* $test = \Sigma^{!}((det(S))^{\delta} \times TP)$, *un graphe de test. Un cas de test est un ioLTS* $TC = \langle Q^{TC}, q_0^{TC}, \Lambda^{TC}, \rightarrow_{TC} \rangle$, *muni de trois ensembles d'états sans successeur* **Pass**, **Inconc** *et* **Fail** *et défini comme suit :*

- $\Lambda^{TC} = \Lambda^{test}$, *avec* $\Lambda^{TC}_{?} = \Lambda^{test}_{!}$ *et* $\Lambda^{TC}_{!} = \Lambda^{test}_{?}$ *(opération miroir) ;*
- $Q^{TC} = \Upsilon \cup \mathbf{Inconc} \cup \mathbf{Fail}$ *son ensemble d'états avec :*
  - $\Upsilon = coreach(\mathbf{Accept}) \cap reach(q_0^{test})$ *l'ensemble des états co-accessibles depuis* **Accept***et accessibles depuis l'état initial. Parmi ces états, est défini l'ensemble des états* **Pass**, *tel que* $\mathbf{Pass} \triangleq \Upsilon \cap \mathbf{Accept}$.
  - **Inconc** *l'ensemble des états à partir desquels* **Accept** *ne peut plus être atteint :* $\mathbf{Inconc} = post_{\Lambda^{TC}_{?}}(\Upsilon) \setminus \Upsilon$ *(nous sortons de l'ensemble des états co-accessibles depuis* **Accept** *par une entrée menant à* **Inconc***).*
  - **Fail** *l'ensemble des états* $\mathbf{Fail}^{test}$ *du graphe de test accessibles en une transition depuis un état appartenant à* Υ *:* $\mathbf{Fail} = post_{\Lambda^{TC}_{?}}(\Upsilon) \cap \mathbf{Fail}^{test}$.
- *l'état initial est* $q_0^{TC} = q_0^{test}$ *si* $\Upsilon \neq \emptyset$. *Si l'ensemble des états accessibles depuis l'état initial et co-accessibles depuis* **Accept** *était vide, nous aurions* $Q^{TC} = \emptyset$ *;*
- $\rightarrow_{TC} = \rightarrow_{\Upsilon} \cup \rightarrow_{\mathbf{Inconc}} \cup \rightarrow_{\mathbf{Fail}}$ *la relation de transition avec :*
  - $\rightarrow_{\Upsilon} = \rightarrow_{test} \cap ((\Upsilon \setminus \mathbf{Pass}) \times \Lambda^{TC} \times \Upsilon)$ *l'ensemble des transitions de test depuis l'état initial pouvant mener à* **Pass** *(comme nous recherchons les chemins minimaux, nous enlevons les transitions depuis* **Pass***).*
  - $\rightarrow_{\mathbf{Inconc}} = \rightarrow_{test} \cap ((\Upsilon \setminus \mathbf{Pass}) \times \Lambda^{TC}_{?} \times \mathbf{Inconc})$ *l'ensemble des transitions menant immédiatement à* **Inconc** *depuis un état de* Υ, **Pass** *mis à part.*
  - $\rightarrow_{\mathbf{Fail}} = \rightarrow_{test} \cap ((\Upsilon \setminus \mathbf{Pass}) \times \Lambda^{TC}_{?} \times \mathbf{Fail})$ *l'ensemble des transitions menant immédiatement à* **Fail** *depuis un état de* Υ, **Pass** *mis à part.*

Notons que les algorithmes de génération de test présentés dans [JJ04] et utilisés dans l'outil TGV sont semblables à ceux-ci. La principale différence est que les opérations se font à la volée pour une plus grande efficacité.

Exemple 2.5 *Le cas de test sélectionné à partir du produit de la figure 2.5 est représenté figure 2.6. La trace de l'objectif mène bien à l'état* **Pass**. *Les traces non-conformes sont explicites, le verdict de non-conformité Fail sera donc émis lors de l'exécution. Si l'implémentation produisait AccesFerme après que le testeur lui a fourni deux entrées Digit, alors le cas de test saurait qu'il est inutile de continuer le test et le verdict Inconc serait émis.*

Par la construction de test présentée à la définition 2.4, nous obtenons un cas de test comme décrit en définition 1.11. Nous pouvons alors exécuter ce cas de test sur l'implémentation sous test comme présenté à la sous-section 1.3.2 et obtenir un des verdicts *Fail*, *Pass* ou *Inconc*. De plus, nous conservons les propriétés de non-biais et d'exhaustivité à la limite (voir la sous-section 1.3.3) par cet algorithme.

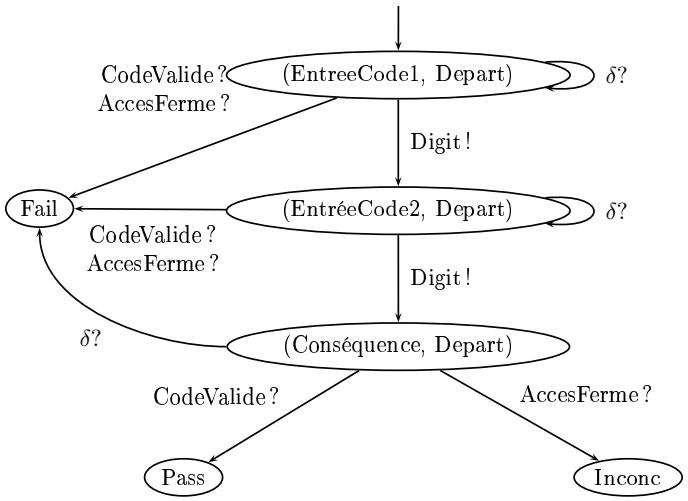


FIG. 2.6 – Cas de test généré à partir de la spécification S et de l'objectif de test TP.

Ce cas de test est non-biaisé par construction. En effet, il a été prouvé que le testeur canonique $Can(\mathcal{S})$ était non-biaisé (seules les implémentations non-conformes seront rejetées). Cette propriété est conservée à la fois par le produit synchrone entre $Can(\mathcal{S})$ et l'objectif de test TP et par la sélection puisque ces transformations n'ajoutent aucun cas de détection de la non-conformité. Tous les cas de test générés sont donc non-biaisés.

Un ensemble fini des cas de test générés par cet algorithme n'est pas exhaustif, par contre, l'ensemble des cas de tests que l'algorithme peut produire l'est. Le fait que l'exhaustivité soit préservée à la limite vient de la construction suivante. Par définition de la relation de conformité ioco, pour toute implémentation non-conforme, il existe une trace $\sigma.a$ appartenant à $Traces(Can(\mathcal{S}), \mathbf{Fail}) \cap STraces(I)$, le préfixe σ étant une trace de la spécification suspendue S^{δ} tandis que $\sigma.a$ est une trace non-conforme appartenant à $Traces(Can(\mathcal{S}), \mathbf{Fail})$. La spécification ayant été suspendue, aucun silence n'est possible, il existe donc obligatoirement une sortie b après la trace σ telle que $\sigma.b$ appartienne à $STraces(S)$ ainsi qu'à $STraces(Can(\mathcal{S}))$ mais pas à $Traces(Can(\mathcal{S}), \mathbf{Fail})$. Il est alors possible de construire un objectif de test TP reconnaissant cette trace ($\sigma.b$ mène alors à **Accept**). Soit TC le cas de test généré à partir de cet objectif. Par construction, ce cas de test peut alors détecter que la trace $\sigma.a$ n'est pas conforme, elle appartient à $Traces(Can(\mathcal{S}), \mathbf{Fail}) \cap STraces(I)$ (le verdict *Fail* est émis). Cela signifie que TC peut rejeter l'implémentation I.

Conclusion

Nous avons décrit dans cette partie les principes de la génération de tests de conformité pour des systèmes réactifs basée sur le modèle des ioLTS finis. Après avoir présenté ces principes dans le cadre général lors de l'introduction, nous avons expliqué, au chapitre 1, certains aspects de la théorie du test par rapport au modèle des ioLTS : la relation de conformité utilisée, les propriétés des suites de test, etc. De cette théorie découle la méthode de génération de test décrite au chapitre 2. Nous avons présenté les différentes opérations effectuées lors de cette génération et montré que les cas de test qui en résultent satisfont certaines propriétés essentielles du test assurant la correction des verdicts.

Malheureusement, le modèle des ioLTS implique une génération de tests énumérative. Par exemple, les valeurs des entrées doivent être énumérées sur leur domaine. Dans le cas de la spécification du digicode, nous avons fait le choix d'abstraire les valeurs possibles d'entrée par une seule entrée ($Digit?$), mais si nous avions voulu un modèle plus précis, il aurait fallu détailler le comportement du digicode pour chaque combinaison de chiffres possible (entrées $Un?, Deux?, ...Neuf?$). L'énumération de toutes ces valeurs peut créer une explosion combinatoire, bien que les algorithmes puissent être exécutés à la volée. De plus, le modèle des ioLTS ne permet pas de représenter des spécifications avec des variables de domaine infini. Il est donc intéressant de se tourner vers la génération de test symbolique, basée sur des modèles tels que les ioSTS (*Input/Output Symbolic Transition System*).

La méthode utilisée lors de la génération de tests symbolique est similaire à celle de la génération énumérative. En effet, la théorie du test de conformité est la même : la relation de conformité est toujours basée sur les traces de la spécification et de l'implémentation. De plus, nous avons choisi le modèle des ioSTS car c'est un modèle de plus haut niveau dont la sémantique est décrite en termes d'ioLTS, ce qui permet d'utiliser certaines propriétés des ioLTS pour prouver celles des ioSTS. Nous allons nous baser dans la partie qui suit sur le modèle symbolique des ioSTS, mais nous améliorerons l'expressivité des verdicts produits par la génération de test grâce à une interprétation différente de la sélection par objectif de test : la sélection par propriété de sûreté et/ou d'accessibilité.

Deuxième partie

Combinaison vérification et test de conformité

Introduction

La vérification formelle et le test de conformité sont deux approches bien établies pour la validation de systèmes réactifs. Dans ces deux approches, le principe général est le même puisqu'il consiste à vérifier une certaine cohérence entre deux représentations du même système :

- la vérification formelle consiste à comparer une implémentation ou une spécification formelle du système, à un ensemble de propriétés de plus haut niveau (décrites dans des logiques, par des langages ou par des automates) que le système devrait satisfaire. La vérification est exhaustive, si la propriété est satisfaite, cela signifie que toutes les exécutions du système la satisfont.
- le test de conformité compare le comportement observable d'une implémentation réelle "boîte noire" du système avec le comportement observable décrit par une spécification formelle. Le test n'est pas exhaustif, il s'effectue sur des exécutions de l'implémentation et ne peut pas en couvrir la totalité. Nous nous fondons dans cette partie sur la théorie du test de Jan Tretmans [Tre96] basée sur les systèmes de transitions à entrées/sorties (ioLTS) et sa relation de conformité ioco, présentés en partie I.

Ces deux techniques de validation sont encore souvent mises en opposition, bien qu'elles soient complémentaires. La vérification permet *a priori* de prouver exhaustivement une propriété sur le modèle de la spécification (ou un programme), ou de prouver sa violation en exhibant un contre-exemple. Cependant, pour des modèles expressifs (tels que les ioSTS décrits dans cette partie), certains problèmes de vérification sont indécidables, ou simplement trop complexes et la vérification peut ne pas être complètement effectuée. Nous obtenons alors une vérification partielle ou approximative. Par exemple, dans le cas d'une propriété de sûreté, la satisfaction sur une sur-approximation implique la satisfaction sur le modèle exact, mais son échec ne permet pas de conclure. Par ailleurs, la vérification ne porte que sur les modèles et ne permet pas de s'assurer que le système réel est correct. En revanche, le test s'effectue sur ce système réel par une interaction finie entre l'environnement (le testeur) et le système. Un ensemble fini de tests ne permet malheureusement pas de prouver que l'implémentation est conforme, mais permet au moins de détecter des non-conformités par rapport au modèle de la spécification.

Celle-ci représentant le comportement de référence du système pour la conformité, il est nécessaire, avant de générer des tests depuis cette spécification, de s'assurer

de la correction de celle-ci par rapport aux propriétés attendues du système. La complémentarité méthodologique est donc évidente, la vérification permettant d'abord de s'assurer de la correction de la spécification par rapport aux propriétés, et le test permettant ensuite de détecter des erreurs de l'implémentation par rapport à cette même spécification. Une complémentarité orthogonale réside aussi dans l'utilisation de techniques de vérification dans la génération de tests. Le principe est alors d'utiliser les algorithmes du *model checking* afin de générer des séquences de test (voir par exemple [GH99, JJ04, BHJP04, HLSU02]).

Cependant, la complémentarité méthodologique décrite plus haut n'est pas complètement satisfaisante. En effet, le but ultime est de s'assurer que l'implémentation satisfait les propriétés attendues. Or, en supposant établie la satisfaction des propriétés par la spécification, la conformité de l'implémentation ne permet pas, dans le cas général, de conclure à la satisfaction des propriétés par l'implémentation. Inversement, la non-conformité de l'implémentation, ne permet pas non plus d'établir la violation des propriétés par cette implémentation. Ceci est illustré à la figure 2.7. Il manque donc un lien formel entre vérification et test de conformité, problème que nous nous attachons à étudier dans cette partie du document.

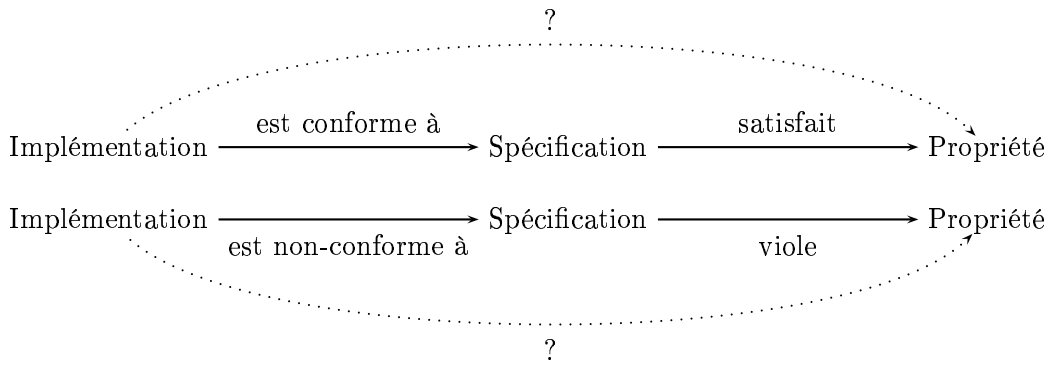


FIG. 2.7 – Manque d'un lien formel entre vérification et test de conformité

Le test peut aller plus loin que la seule détection de la non-conformité. En effet, les tests sont basés sur des comportements finis de la spécification expérimentés sur l'implémentation. Il permet donc, pour les propriétés de sûreté (rien de mauvais n'arrivera) ou d'accessibilité (quelque chose d'attendu arrivera), d'une part de détecter pendant l'exécution du test la violation de la propriété de sûreté (respectivement la satisfaction d'une propriété d'accessibilité) sur l'implémentation, mais aussi de compléter la vérification en détectant la violation (respectivement la satisfaction) sur la spécification, également pendant l'exécution du test.

Nous présentons de manière formelle une méthodologie pour la validation de systèmes réactifs basée sur ces constats, combinant de manière intégrée vérification formelle et test de conformité. Nous pouvons ainsi tester la non-conformité d'une implémenta-

tion par rapport à sa spécification tout en vérifiant des propriétés par rapport à la spécification mais aussi à l'implémentation. Nous nous basons sur des modèles de spécifications combinant des aspects contrôle et données : les ioSTS (pour *Input/Output Symbolic Transitions System*). La sémantique de ce modèle est un système de transitions à espaces d'états infinis, et pour lequel les problèmes d'accessibilité typiques de la vérification sont indécidables. La méthodologie proposée peut se décomposer en trois étapes :

1. les propriétés sont automatiquement vérifiées sur la spécification ; la complexité ou la non-décidabilité du problème de vérification peut résulter en une vérification qui peut ne pas aboutir ;
2. des cas de test sont automatiquement dérivés à partir de la spécification et des propriétés ;
3. les cas de test sont exécutés sur l'implémentation "boîte noire". Ils détectent :
   - la non-conformité de l'implémentation vis-à-vis de la spécification d'une part,
   - la violation des propriétés de sûreté sur la spécification et sur l'implémentation ainsi que la satisfaction des propriétés d'accessibilité sur la spécification et sur l'implémentation d'autre part.

Au-delà de cet aspect méthodologique, nous présentons dans cette partie la génération de tests à partir de propriétés de manière uniforme, en nous basant sur la notion d'observateur. Un observateur est un automate sur l'alphabet des actions observables ayant un ensemble d'états reconnaisseurs. Il permet de spécifier soit des propriétés d'accessibilité, soit la négation de propriétés de sûreté. Nous généralisons ainsi le principe des objectifs de test puisque ceux-ci sont des propriétés d'accessibilité. De plus, la spécification étant une propriété de sûreté, nous pouvons construire à partir de la celle-ci un testeur canonique correspondant à un observateur de non-conformité. Sa composition avec des observateurs d'autres propriétés servira de base à la génération de tests.

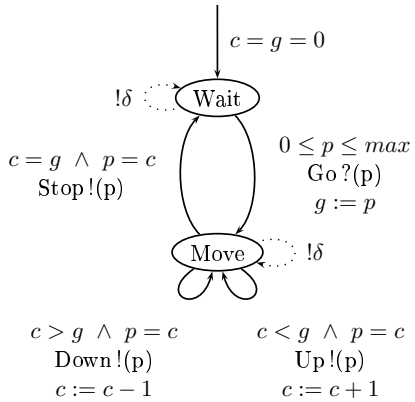
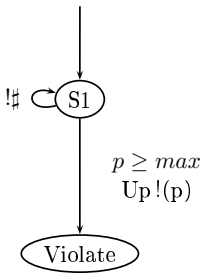


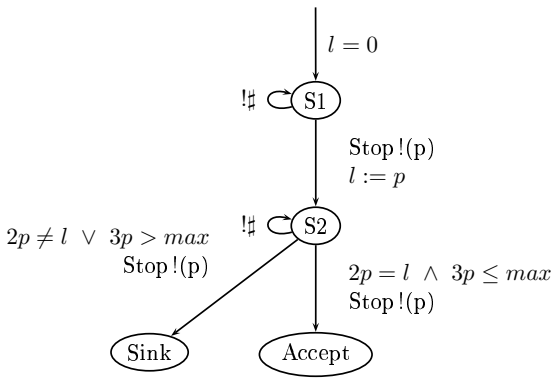
FIG. 2.8 – Spécification de l'ascenseur

Exemple. Voici un exemple de spécification avec variables : l'ascenseur. La représentation de cet exemple est donnée à la figure 2.8. Une entrée *Go*? portant le numéro de l'étage demandé est d'abord produite. La variable g (pour *goal*) est alors affectée de ce numéro. Si l'étage demandé est supérieur à l'étage courant (variable c pour *current* initialement affectée à 0), alors la sortie *Up*! est émise (l'ascenseur monte d'un étage, donc la variable c est incrémentée), sinon, la sortie *Down*! est produite (l'ascenseur descend, c est décrémentée). Si l'étage courant correspond à celui demandé ($c = g$) alors la sortie *Stop*! est émise. Sont également représentées sur la figure 2.8, les transitions dues à la suspension de la spécification (transitions portant la sortie δ).

Une propriété de sûreté pourrait par exemple être que l'ascenseur ne peut pas monter s'il a atteint le plus haut étage de l'immeuble (correspondant à la constante non affectée *max* de la figure 2.8). La négation de cette propriété est représentée par l'observateur de la figure 2.9(a). Le cas de test généré à partir de cette propriété et de la spécification est illustré à la figure 2.10(a). Cet exemple montre qu'un lien a été fait entre implémentation et propriété. En effet, si le verdict **FailViolate** est émis lors de l'exécution de ce cas de test sur l'implémentation, alors cela signifie que l'implémentation n'est pas seulement non-conforme à la spécification, elle viole en plus la propriété de sûreté. L'observateur négatif permet de guider le test afin de vérifier la propriété lors de l'exécution du cas de test sur l'implémentation. Cela peut être vu comme du *monitoring* : on vérifie en ligne, pendant l'exécution, si la propriété n'a pas été violée par la spécification et/ou l'implémentation.



(a) Observateur négatif représentant la négation de la propriété de sûreté

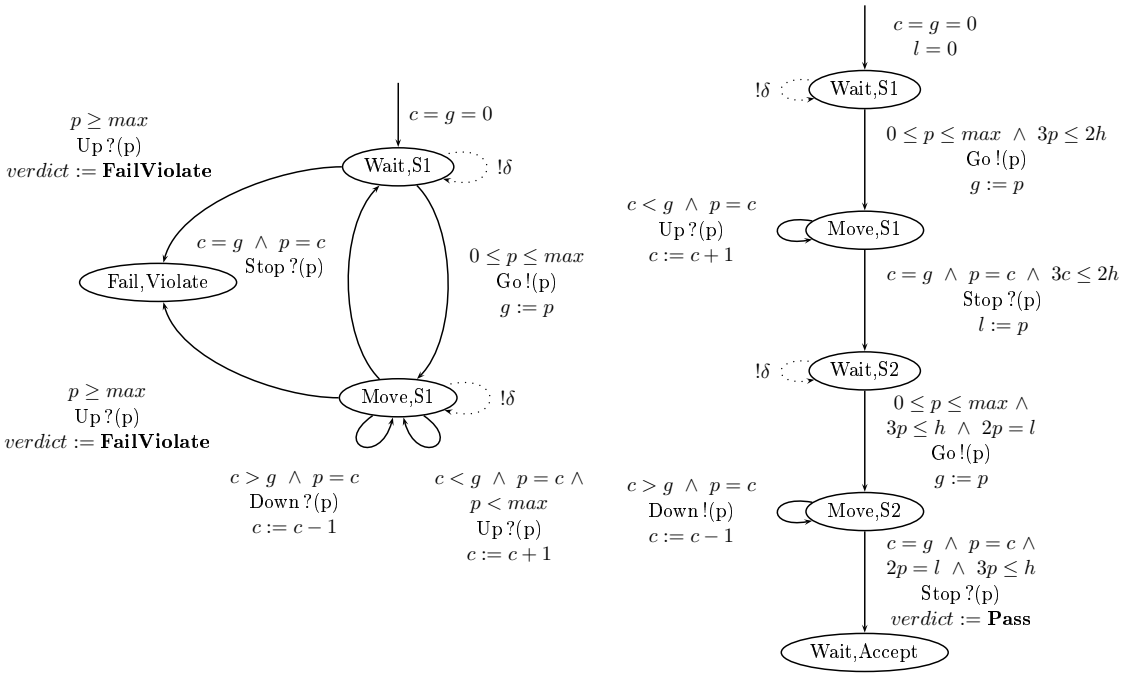


(b) Observateur positif représentant la propriété d'accessibilité.

FIG. 2.9 – Observateurs.

Une propriété d'accessibilité est représentée à la figure 2.9(b) : après un arrêt à un certain étage l (pour *level*), l'arrêt suivant devra être au numéro correspondant à la moitié de l'étage précédemment demandé ($2g = l$) tout en étant inférieur au tiers de la

hauteur de l'immeuble ($3g \leq max$). Celle-ci est un peu plus compliquée afin d'illustrer les approximations des calculs lors de la sélection dues au modèle des ioSTS. En effet, le but de cette propriété est de forcer les analyses de la phase de sélection à calculer que pour satisfaire cette propriété, le premier étage demandé doit être pair (l est pair) et inférieur ou égal aux deux tiers de la hauteur maximale ($3l \leq 2max$). Le cas de test généré à partir de la spécification et de cette propriété d'accessibilité est illustré à la figure 2.10(b). On y constate que les contraintes calculées sont sur-approximées : la contrainte $3l \leq 2max$ y est bien calculée puisque nous avons la contrainte $3p \leq 2max$ sur la première transition portant l'action $Target!$. La contrainte portant sur la parité de l (et donc de p sur cette même transition) est par contre perdue. Cet exemple illustre également le fait que si le verdict **Pass** est émis lors de l'exécution de ce cas de test sur l'implémentation, alors cela signifie que la spécification ainsi que l'implémentation vérifient cette propriété. Ainsi, même si la vérification n'a pas pu être concluante, nous pouvons prouver par le test que la propriété est satisfaite par la spécification.



(a) Cas de test généré à partir de la spécification et de l'observateur négatif.

(b) Cas de test généré à partir de la spécification et de l'observateur positif.

FIG. 2.10 – Cas de test.

Cette partie se base sur plusieurs publications de l'équipe VerTeCS sur les techniques de génération de tests énumérées et symboliques implémentées dans les outils développés par l'équipe, TGV (*Test Generation with Verification technology* [JJ04, Jér02]) et

STG (*Symbolic Test Generation* [CJRZ02, PJJ07]), en particulier [JJRZ05, RMT+04, RMJ05, CJMR06, CJMR07].

Celle-ci est structurée de la manière suivante. Dans le chapitre 3, sont présentés les modèles de systèmes infinis (les ioSTS et leur sémantique en termes d'ioLTS), les notions et les opérations de base sur ces modèles ainsi que la génération du testeur canonique. Le chapitre 4 introduit ensuite la notion d'observateur et les problèmes de vérification de propriétés. Il décrit ensuite la sélection de tests à partir de propriétés ainsi que l'interprétation des verdicts en fonction de celles-ci. Les cas de test produits ici contiennent des comportements ne permettant pas d'atteindre des verdicts ciblés par les propriétés. Ce chapitre s'attache à résoudre ce problème d'accessibilité en utilisant des techniques de vérification approchées basées sur une analyse par interprétation abstraite. Enfin, cette méthodologie est expérimentée au chapitre 5 sur l'outil STG.

Chapitre 3

Le modèle des ioSTS

Le modèle des *input/output Symbolic Transitions Systems* permet de représenter des systèmes infinis. Inspiré des *I/O automata* [LT99], le modèle des ioSTS étend les LTS par des données (pouvant être de domaine infini). Il permet de modéliser des programmes impératifs non récursifs, à mémoire bornée et communiquant avec leur environnement. La syntaxe de ce modèle est d'abord présentée, suivie par sa sémantique en termes de systèmes de transitions puis par certaines notations. Nous définirons ensuite trois opérations particulièrement importantes pour la génération de tests ce qui nous amènera pour finir à générer le testeur canonique.

3.1 Syntaxe et sémantique

3.1.1 Syntaxe des ioSTS

Le modèle des ioSTS est constitué par des variables (pouvant prendre leurs valeurs dans des domaines infinis), des gardes, des affectations et des actions d'entrées et sorties portant des paramètres de communication. Contrairement aux *I/O automata*, les ioSTS ne sont pas nécessairement complets en entrée. Comme nous le verrons par la suite, ce modèle servira pour les spécifications, les observateurs (ou objectifs de test) et les cas de test. Nous donnerons donc une définition des ioSTS assez générale pour modéliser ces trois objets.

Tout d'abord, nous devons clarifier certaines notations concernant les variables. Une variable v a un type, qui prend ses valeurs dans un ensemble noté $\mathcal{D}_v = \mathcal{D}(type(v))$. Pour un tuple $V = \langle v_1, \ldots, v_n \rangle$ de variables, on notera $\mathcal{D}(V)$ le produit $\mathcal{D}(type(v_1)) \times \ldots \times \mathcal{D}(type(v_n))$ des domaines des variables $v \in V$. Un élément de $\mathcal{D}(V)$ est une valuation des variables de V. On peut considérer un prédicat $P(V)$ sur un tuple de variables V soit comme un ensemble $P(V) \subseteq \mathcal{D}_V$, soit comme une formule logique dont la sémantique est une fonction : $\mathcal{D}_V \rightarrow \{true, false\}$. L'affectation à une variable v d'une expression dépendant d'un ensemble de variables V est une fonction du type $\mathcal{D}_V \rightarrow \mathcal{D}_v$. Une affectation d'un tuple X de variables est alors une fonction de type : $\mathcal{D}_V \rightarrow \mathcal{D}_X$. Les affectations (de la forme $x := A_x$) sont donc ici bien typées :

l'expression A_x est de même type que x.

Voici la définition du modèle des systèmes de transitions symboliques que nous utiliserons par la suite :

Définition 3.1 (ioSTS) *Un système de transitions symboliques à entrée/sortie (ioSTS) $\mathcal{M}$ est défini par un 4-uplet $\langle V, \Theta, \Sigma, \mathcal{T} \rangle$ où :*

- *V est un ensemble fini de variables ;*
- *Θ, appelé condition initiale, est un prédicat $\Theta \subseteq \mathcal{D}_V$ (nous supposons que Θ a une unique solution sur $\mathcal{D}_V$) portant sur les variables V ;*
- *$\Sigma = \Sigma_? \cup \Sigma_! \cup \Sigma_\tau$ est l'alphabet fini des actions. Il est constitué de trois ensembles disjoints d'entrées $\Sigma_?$, de sorties $\Sigma_!$ et d'actions internes Σ_τ. Chaque action $a \in \Sigma$ est caractérisée par sa signature $sig(a)$: un vecteur de paramètres $\langle p_1, \ldots, p_k \rangle$ spécifiant les types des paramètres de communication $P(a)$ portés par l'action a. La signature d'une action interne $a \in \Sigma_\tau$ est le tuple vide ;*
- *$\mathcal{T}$ est un ensemble fini de transitions symboliques. Chaque transition est un tuple (a, p, G, A) défini par :*
  - *une action $a \in \Sigma$, appelée l'action de la transition et un tuple de paramètres de communication $p = \langle p_1, \ldots, p_k \rangle$, locaux à la transition ; sans perte de généralité, nous supposons que chaque action a porte toujours le même vecteur de paramètres p ; celui-ci est supposé bien-typé par rapport à la signature de a : $sig(a) = \langle t_1, \ldots, t_k \rangle$ où les t_i désignent les types des paramètres ;*
  - *une expression booléenne G sur $V \cup P(a)$ portant sur les variables et les paramètres de communications de a, appelée la garde de la transition (la portée des paramètres est limitée à une transition) ;*
  - *une affectation $A : \mathcal{D}_V \times \mathcal{D}_{sig(a)} \rightarrow \mathcal{D}_V$ qui définit l'évolution des variables au cours de la transition. $A_v : \mathcal{D}_V \times \mathcal{D}_{sig(a)} \rightarrow \mathcal{D}_v$ représente la projection de A définissant l'évolution de la variable $v \in V$.*

Pour des raisons techniques, les gardes des ioSTS sont exprimées en une théorie fermée par élimination des quantificateurs : tout prédicat contenant des quantificateurs est équivalent à un prédicat n'en contenant pas. C'est par exemple le cas de l'arithmétique de Presburger. Cela permet à la satisfiabilité des gardes d'être décidable, ce qui est particulièrement important lors des phases de déterminisation (sous-section 3.2.2) et d'exécution (sous-section 4.2.2.4).

Contrairement à la définition du modèle des ioSTS que nous donnions dans [CJMR06, CJMR07], il n'y a pas ici de notion explicite de localité (ou point de contrôle) puisque les structures de contrôle d'un automate peuvent être encodées par une variable spécifique du programme servant de compteur. Nous utilisons cette variable dans les exemples illustrant les ioSTS afin de représenter les localités. Prenons la variable *pc* ayant comme domaine l'ensemble des localités possibles. Nous avons alors dans la condition initiale Θ la condition que *pc* a pour valeur la localité de départ. De plus, pour chaque transition, nous avons sur les gardes une condition portant sur

la valeur de pc et, si la transition ne boucle pas, nous avons dans les affectations le changement de valeur de cette variable. Ceci est illustré dans l'exemple suivant.

Exemple 3.1 *Un exemple d'ioSTS est donné figure 3.1. Initialement, ce système est en attente d'une entrée* Start *portant le paramètre entier p. Une fois cette entrée reçue, la valeur de p est sauvegardée dans la variable x. Puis, tant que la valeur de la variable x est strictement positive, sa valeur est émise via l'événement de sortie* Msg *portant le paramètre m (m prenant la valeur de x). Dans le même temps, la valeur de la variable x est décrémentée de 1. Lorsque celle-ci atteint 0, l'événement* Stop *est émis en direction de l'environnement. Le paramètre porté par l'entrée* Start *indique donc le nombre de messages (*Msg*) émis.*

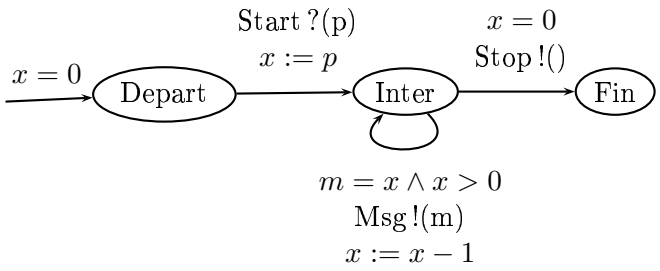


FIG. 3.1 – Exemple d'un ioSTS $\mathcal{S}$ avec localités.

Cet exemple est représenté à la figure 3.1 avec localités, ce qui permet une représentation graphique et par conséquence une lecture plus simple. Ce même exemple peut être donné sans notion de localité, mais avec une variable pc de domaine $\{Debut, Inter, Fin\}$ permettant de les encoder. La condition initiale Θ a alors pour valeur $pc = Debut \wedge x = 0$ et l'ensemble des transitions est le suivant :

- $(Start?,\ p,\ pc = Debut,\ \langle pc := Inter,\ x := p\rangle)$ *: une entrée Start est émise portant le paramètre p si la variable pc vaut $Debut$. pc prend alors la valeur $Inter$ et x vaut p.*
- $(Msg!,\ m,\ pc = Inter \wedge m = x \wedge x > 0,\ x := x - 1)$ *: une sortie Msg est émise portant le paramètre m si la variable pc vaut $Inter$, m vaut x et x est strictement supérieur à 0. x est alors décrémenté.*
- $(Stop!,\ null,\ pc = Inter \wedge x = 0,\ pc := Fin)$ *: une sortie $Stop$ est émise si la variable pc vaut $Inter$, et x vaut 0. pc prend alors pour valeur Fin. La valeur $null$ signifie qu'il n'y a pas de paramètre porté par l'action $Stop!$.*

Par la suite, le nom des localités n'ayant aucune importance, il ne sera plus représenté sur les exemples afin d'alléger la lecture du système.

3.1.2 Sémantique des ioSTS

Un ioSTS est en réalité une syntaxe permettant de définir de manière finie un système de transitions infini. La sémantique d'un ioSTS peut être définie en termes de systèmes de transitions à entrée/sortie (ioLTS pour *input/output Labelled Transitions Systems*).

Intuitivement, la sémantique ioLTS d'un ioSTS énumère tous les tuples de valeurs possibles (les valuations) des variables et des paramètres de communication. On notera $\nu \in \mathcal{D}_V$ une valuation possible de l'ensemble des variables V, et $\pi \in \mathcal{D}_{\text{sig(a)}}$ une valuation possible de l'ensemble des paramètres de communication de l'action a. La sémantique formelle d'un ioSTS est définie comme suit.

Définition 3.2 (Sémantique d'un ioSTS) *La sémantique d'un ioSTS* $\mathcal{M} = \langle V, \Theta, \Sigma, \mathcal{T} \rangle$ *est un ioLTS* $[\![\mathcal{M}]\!] = \langle Q, q_0, \Lambda, \rightarrow \rangle$*, défini par :*

- $Q = \mathcal{D}_V$ *: l'ensemble des états ;*
- $q_0 = \nu$ *tel que* $\nu \in \Theta$ *: l'état initial ;*
- $\Lambda = \{\langle a, \pi \rangle \mid a \in \Sigma \wedge \pi \in \mathcal{D}_{sig(a)}\}$ *: l'ensemble des actions valuées, partitionné en trois ensembles :* $\Lambda_?$ *les* entrées valuées, $\Lambda_!$ *les* sorties valuées *et* Λ_τ *les* actions internes*, tels que pour* $\# \in \{?, !, \tau\}$, $\Lambda_\# = \{\langle a, \pi \rangle \mid a \in \Sigma_\# \wedge \pi \in \mathcal{D}_{sig(a)}\}$ *;*
- $\rightarrow$ *est la plus petite relation dans* $Q \times \Lambda \times Q$ *définie par la règle suivante :*

$$\frac{(a, p, G, A) \in \mathcal{T} \quad \nu \in \mathcal{D}_V \quad \pi \in \mathcal{D}_{sig(a)} \quad \nu' \in \mathcal{D}_V \quad G(\nu, \pi) = true \quad \nu' = A(\nu, \pi)}{\nu \xrightarrow{\langle a, \pi \rangle} \nu'} \tag{3.1}$$

Un état de l'ioLTS est composé d'une valuation des variables de l'ioSTS. Dans l'état initial, la valeur des variables est définie de manière unique par la condition initiale de l'ioSTS Θ. Les transitions sont étiquetées par des actions valuées composées par le nom de l'action et la valuation des paramètres de communication. Les actions internes ne portent pas de paramètre de communication, les ensembles Λ_τ et Σ_τ peuvent donc être identifiés.

La règle 3.1 signifie qu'une transition (a, p, G, A) d'un ioSTS peut être tirée depuis un état ν s'il existe une valuation π des paramètres de communication p telle que $\langle \nu, \pi \rangle$ satisfait la garde G. Dans ce cas, l'action valuée $\langle a, \pi \rangle$ est validée et les variables sont affectées à leurs nouvelles valeurs, spécifiées par l'affectation A.

La sémantique d'un ioSTS peut être un ioLTS à états infinis à cause des domaines infinis que peuvent avoir les variables. Ces ioLTS peuvent également avoir un degré de branchement (nombre de transitions à partir d'un même état) infini puisque les domaines des paramètres de communication peuvent être infinis.

Certaines notions et propriétés des ioSTS sont définies en termes de leur sémantique ioLTS, nous allons donc pouvoir les réutiliser. Nous pouvons ainsi décrire le comportement d'un ioSTS grâce à sa sémantique (voir la définition 1.12 d'une exécution). Lors

de la modélisation pour la génération de tests, nous considérons que les variables (les états de la sémantique ioLTS) ne peuvent pas être observées par l'environnement. Nous ne considérons donc qu'une abstraction des exécutions : leur trace, définie en termes d'ioLTS à la définition 1.3.

Comme pour les ioLTS, nous écrivons $q \xrightarrow{\alpha} q'$ pour $(q, \alpha, q') \in \rightarrow$ et $q \xrightarrow{\alpha}$ pour $\exists q'$ tel que $q \xrightarrow{\alpha} q'$. Soit un sous-ensemble de l'alphabet $\Lambda' \subseteq \Lambda$, un état q de $\mathcal{M}$ est dit *complet* sur Λ' si $\forall \alpha \in \Lambda' \ : \ q \xrightarrow{\alpha}$. Un état est dit *complet* s'il est complet sur tout l'alphabet Λ. L'ioLTS $[\![\mathcal{M}]\!]$ est *complet* sur Λ' (respectivement Λ) si tous ses états sont complets sur Λ' (respectivement Λ). Nous pouvons également définir un ioSTS complet en termes d'ioSTS :

Définition 3.3 (ioSTS complet) *Un ioSTS $\mathcal{M}$ est* complet *sur Σ' (avec $\Sigma' \subseteq \Sigma$) si pour chaque action $a \in \Sigma'$, $\bigvee_{(a,p,G,A) \in \mathcal{T}} \ G = true$.*

3.2 Opérations sur les ioSTS

Cette section présente quelques opérations basiques sur les ioSTS qui seront utiles lors des phases de vérification et de génération de cas de test. Ces opérations sont le *produit synchrone*, la *déterminisation* et la *suspension* d'un ioSTS.

3.2.1 Composition de deux ioSTS : le produit synchrone

Le produit synchrone est utilisé, d'une part, en vérification pour définir l'ensemble des traces d'un ioSTS reconnues par un observateur et, d'autre part, en test de conformité afin de générer les tests et modéliser l'exécution synchrone d'un cas de test sur une implémentation.

Le produit synchrone de deux ioSTS $\mathcal{M}_1$ et $\mathcal{M}_2$ est un ioSTS pour lequel l'ensemble des traces (respectivement des traces reconnues) est donné par l'intersection des traces (respectivement des traces reconnues) de $\mathcal{M}_1$ et $\mathcal{M}_2$. Cette opération revient à faire sémantiquement le produit de deux ioLTS. Elle peut être définie à un niveau syntaxique sur les ioSTS en synchronisant les actions par la conjonction de leurs gardes. Cette opération impose donc que $\mathcal{M}_1$ et $\mathcal{M}_2$ partagent le même alphabet visible, les mêmes paramètres et n'aient aucune variable commune.

Nous allons d'abord définir la compatibilité de deux ioSTS, puis utiliser cette compatibilité pour définir le produit synchrone et enfin énoncer ce que cette opération produit sur les traces.

Définition 3.4 (Compatibilité entre ioSTS) *Pour $j = 1, 2$, les deux ioSTS $\mathcal{M}_j = \langle V^j, \Theta^j, \Sigma^j, \mathcal{T}^j \rangle$ tels que $\Sigma^j = \Sigma^j_? \cup \Sigma^j_! \cup \Sigma^j_\tau$ sont* compatibles *si $V^1 \cap V^2 = \emptyset$, $\Sigma^1_! = \Sigma^2_!$ et $\Sigma^1_? = \Sigma^2_?$ (l'ensemble des paramètres de communication est donc obligatoirement le même).*

En d'autres termes, deux ioSTS sont compatibles si, d'une part, leurs alphabets visibles (et donc leurs paramètres de communication) sont les mêmes et si, d'autre part, ils n'ont aucune variable en commun.

Définition 3.5 (produit synchrone) *Le produit synchrone $\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2$ entre deux ioSTS compatibles $\mathcal{M}_1$ et $\mathcal{M}_2$ est un ioSTS $\langle V, \Theta, \Sigma, \mathcal{T} \rangle$ défini par :*

- $V = V^1 \cup V^2$,
- $\Theta(\langle v^1, v^2 \rangle) = \Theta^1(v^1) \wedge \Theta^2(v^2)$,
- $\Sigma = \Sigma_? \cup \Sigma_! \cup \Sigma_\tau$ *avec* $\Sigma_? = \Sigma^1_? = \Sigma^2_?$, $\Sigma_! = \Sigma^1_! = \Sigma^2_!$ *et* $\Sigma_\tau = \Sigma^1_\tau \cup \Sigma^2_\tau$,
- $\mathcal{T}$ *: l'ensemble des transitions symboliques du système est défini par le plus petit ensemble qui satisfait les règles suivantes :*

$$(1) \quad \frac{(a, p, G_1, A_1) \in \mathcal{T}^1, \quad a \in \Sigma^1_\tau}{(a, p, G_1, A_1 \cup (x := x)_{x \in V^2}) \in \mathcal{T}} \text{ (et symétriquement pour } a \in \Sigma^2_\tau),$$

$$(2) \quad \frac{(a, p, G_1, A_1) \in \mathcal{T}^1 \quad (a, p, G_2, A_2) \in \mathcal{T}^2}{(a, p, G_1 \wedge G_2, A_1 \cup A_2) \in \mathcal{T}} \text{ (pour } a \in \Sigma_? \cup \Sigma_!).$$

Le produit synchrone fait évoluer indépendamment les actions internes, propres à chacun des ioSTS (règle (1)), mais synchronise leurs actions observables (règle (2)) puisqu'elles sont communes.

Les traces du produit synchrone de deux ioSTS compatibles correspondent donc à l'intersection des traces de chacun des ioSTS composés.

Lemme 3.1 *Traces*$(\mathcal{M}_1 \times \mathcal{M}_2) =$ *Traces*$(\mathcal{M}_1) \cap$ *Traces*$(\mathcal{M}_2)$,
Traces$(\mathcal{M}_1 \times \mathcal{M}_2, F_1 \times F_2) =$ *Traces*$(\mathcal{M}_1, F_1) \cap$ *Traces*$(\mathcal{M}_2, F_2)$, *F_1 et F_2 étant des ensembles d'états de $[\![\mathcal{M}]\!]$.*

3.2.2 ioSTS déterministe et déterminisation

Intuitivement, un ioSTS est *déterministe* si chacune de ses traces correspond exactement à une exécution. C'est par exemple le cas en test de conformité : le cas de test doit toujours donner le même verdict pour une même trace. On définit un ioSTS comme étant déterministe s'il n'a pas d'action interne et si les gardes des transitions étiquetées par la même action sont mutuellement exclusives.

Définition 3.6 (ioSTS déterministe) *Un ioSTS $\langle V, \Theta, \Sigma, \mathcal{T} \rangle$ est déterministe si $\Sigma_\tau = \emptyset$, et, si pour chaque action $a \in \Sigma$ et chaque paire de transitions distinctes $t_1 = (a, p, G_1, A_1)$ et $t_2 = (a, p, G_2, A_2)$ étiquetées par la même action, la conjonction des gardes $G_1 \wedge G_2$ est insatisfiable.*

Comme nous allons le voir, tous les ioSTS ne sont pas déterminisables. La déterminisation d'un ioSTS $\mathcal{M}$ signifie calculer un ioSTS déterministe ayant les mêmes traces que $\mathcal{M}$. Si un ioSTS $\mathcal{M}$ appartient à la classe des ioSTS déterminisables, la déterminisation de $\mathcal{M}$ se fait, comme celle d'un ioLTS, en deux étapes :

- l'élimination des actions internes, c'est-à-dire, pour un état q le calcul des états accessibles par des actions internes (q *after* ϵ),
- la résolution du non-déterminisme pour les actions observables, c'est-à-dire le calcul, pour un ensemble d'états Q et pour une action a, du sous-ensemble Q *after* a.

Chacune de ces étapes sera résumée dans les sous-sections 3.2.2.1 et 3.2.2.2. Une explication plus complète est donnée dans [JMR06].

3.2.2.1 Elimination des actions internes

L'idée, présentée dans [RdBJ00, Zin04], pour éliminer les actions internes d'un ioSTS est de calculer l'effet d'une séquence d'actions internes, finissant par une action observable, sur les variables et d'appliquer cet effet sur les gardes et les affectations de la dernière transition (celle de l'action observable). Si aucune action n'est observable après une séquence d'actions internes, les affectations de cette séquence n'ont aucune incidence sur les traces, nous ne tenons alors pas compte de cette séquence. L'opération d'élimination des actions internes résumée ici n'est possible que si l'ioSTS ne contient aucune boucle d'actions internes.

Cette opération consiste à composer les gardes et les affectations de deux transitions consécutives $t_1 = (\tau, p, G_1, A_1)$ étiquetée par une action interne τ et $t_2 = (a, p, G_2, A_2)$ étiquetée par une action observable (entrée ou sortie) a (voir figure 3.2). Nous notons alors $A_2 \circ A_1$ la fonction qui, pour chaque valuation ν des variables et chaque valuation π des paramètres dans $sig(a)$, associe la valuation $A_2(A_1(\nu), \pi)$ aux variables. Nous appliquons d'abord les affectations de la première transition t_1 puis celles de t_2 puisqu'elles peuvent tenir compte des valeurs données par A_1. De même, nous noterons $G_2 \circ A_1$ la fonction qui, pour chaque valuation ν des variables et chaque valuation π des paramètres dans $sig(a)$, associe la valeur booléenne $G_2(A_1(\nu), \pi)$. La garde G_2 de t_2 n'est vérifiée qu'après l'affectation A_1 de la première transition.

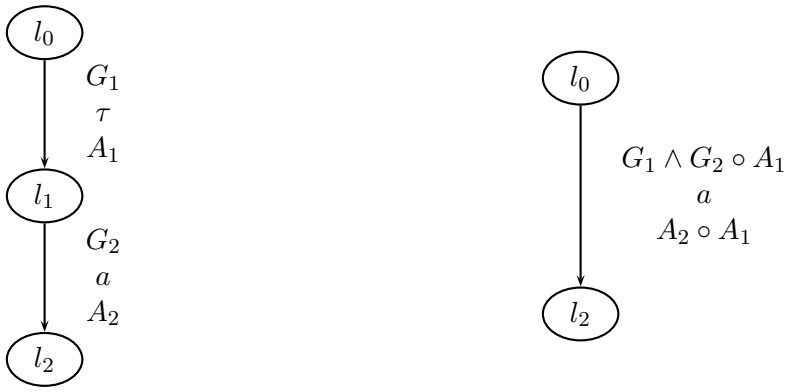


FIG. 3.2 – Elimination locale d'une action interne.

Notons que cette élimination locale d'une action interne est représentée à la fi-

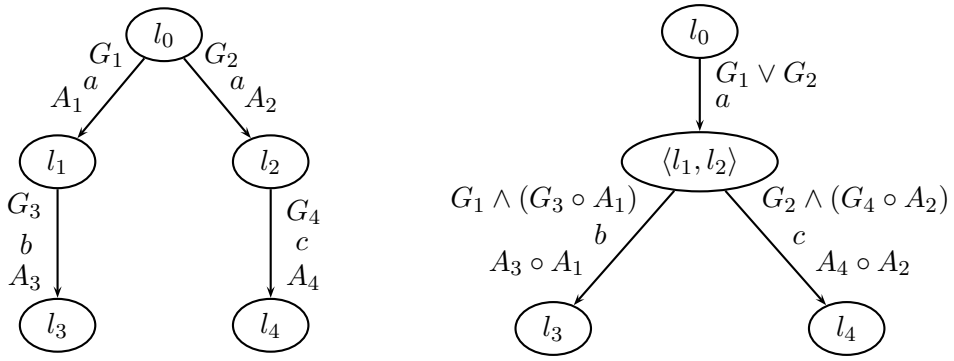
gure 3.2 avec des localités, ce qui donne une vision plus intuitive de cette opération. La notion de localités ne change rien à l'élimination des actions internes. Sans utiliser les localités, nous avons dans les gardes des conditions sur la variable servant de compteur pc et dans les affectations les changements de valeur de cette variable. Par rapport à la figure 3.2, cela signifie que nous avons dans G_1 $pc = l_0$, dans A_1 $pc := l_1$, dans G_2 $pc = l_1$ et dans A_2 $pc := l_2$. Par élimination de l'action interne, cela nous donne une transition portant la garde $G_1 \wedge G_2 \circ A_1$ et l'affectation $A_2 \circ A_1$. Cette garde contient les conditions sur pc $pc = l_0$ et $pc = l_1$ après l'affectation $pc := l_1$, ce qui équivaut alors à $true$. L'affectation $A_2 \circ A_1$ contient quant à elle les affectations $pc := l_1$ puis $pc := l_2$. L'opération d'élimination des actions internes est donc indépendante de la notion de localités.

L'élimination des actions internes ne peut terminer que sur les ioSTS ne contenant pas de boucles d'actions internes. Dans ce cas, cette procédure produit un ioSTS équivalent en termes de traces.

Lemme 3.2 $Traces(Elim(\mathcal{M})) = Traces(\mathcal{M})$

3.2.2.2 Résolution de choix non-déterministes sur les actions observables

La seconde étape de la déterminisation consiste à résoudre les choix non-déterministes sur les actions observables. Pour réaliser cette étape, une procédure de déterminisation symbolique est décrite dans [JMR06].



(a) ioSTS non déterministe

(b) ioSTS déterministe

FIG. 3.3 – Résolution locale d'un choix non-déterministe

Le principe de cette procédure est de découper la déterminisation en une séquence de déterminisations locales. Pour plus de simplicité, nous allons utiliser une variable contenant les localités de l'ioSTS. Cela permettra une meilleure intuition de la procédure utilisée par rapport à la figure 3.3. Prenons un ioSTS non-déterministe, représenté figure 3.3(a), qui, à partir d'une localité l_0, va avoir deux transitions étiquetées par la même

action observable a mais deux destinations différentes (l_1 et l_2) et, respectivement, deux affectations différentes (A_1 et A_2). L'idée du déterminisme local, représenté figure 3.3(b), est de créer une nouvelle localité (ici, $\langle l_1, l_2 \rangle$) ainsi que les trois nouvelles transitions qui en découlent.

La première transition va de l_0 à $\langle l_1, l_2 \rangle$. Sa garde est $G_1 \vee G_2$, son action est l'action observable a et son affectation est vide. Intuitivement, cette transition représente l'ensemble des transitions créant le non-déterminisme, excepté pour les affectations car on ne peut savoir laquelle effectuer de A_1 ou A_2. On va donc reporter ces affectations sur les prochaines transitions.

Une transition, étiquetée par l'action observable b va de $\langle l_1, l_2 \rangle$ à l_3. Grâce à l'action b de cette transition, nous savons que nous devons prendre en compte l'affectation A_1 reportée de la transition précédente (et non A_2). Nous incorporons donc A_1 à A_3, ce qui donne l'affectation $A_3 \circ A_1$. Nous devons faire de même pour la garde de la transition puisque la garde G_3 est évaluée après que l'affectation A_1 ait été effectuée. De plus, de manière similaire à l'affectation, nous devons prendre en compte la garde vérifiée à la transition précédente (en l'occurence G_1) et l'incorporer à la garde de la transition. Celle-ci est donc $G_1 \wedge (G_3 \circ A_1)$.

Une transition similaire à la précédente mais étiquetée par c va de $\langle l_1, l_2 \rangle$ à l_4.

La notion de localités est nécessaire lors de la déterminisation locale. Nous utilisons ici une abstraction des ioSTS sur le modèle équivalent contenant des localités. Sans cette notion, une solution serait de traiter différemment la variable pc encodant les localités.

Cette procédure de déterminisation ne termine pas dans le cas général, excepté pour des sous-classes particulières comme celle proposée dans [JMR06]. Les auteurs y proposent une procédure qui itère des déterminisations locales et qui termine pour une sous-classe des ioSTS appelée ioSTS à *lookahead* borné. Dans cette sous-classe, une trace bornée permet d'inférer quelle transition provoquant un choix non-déterministe a été prise.

Cette opération de déterminisation termine donc si l'ioSTS ne contient aucun cycle d'actions internes et s'il satisfait la condition d'un *lookahead* borné.

Par la suite, on notera $det(\mathcal{M})$ l'opération consistant à déterminiser l'ioSTS $\mathcal{M}$ (en supposant que celui-ci appartient à la classe des ioSTS déterminisables). Cette opération préservant les traces, on obtient le résultat suivant :

Lemme 3.3 $Traces(det(\mathcal{M})) = Traces(\mathcal{M})$.

3.2.3 Suspension d'un ioSTS

Comme pour les ioLTS (voir chapitre 2, sous-section 1.2.1), l'opération de suspension sur les ioSTS permet de mettre en évidence les blocages autorisés par la spécification. Cette opération transforme un ioSTS $\mathcal{M}$ en un ioSTS $\mathcal{M}^\delta$, appelé ioSTS suspendu de $\mathcal{M}$. Pour les ioLTS, à partir de chaque état de blocage est ajoutée une nouvelle transition qui

boucle sur celui-ci. Une nouvelle action δ de sortie est associée à cette transition. Cette action est admissible si et seulement si aucune action de sortie ou interne ne peut être tirée. Pour les ioSTS, seuls les blocages de sortie et les *deadlocks* sont détectés comme des blocages autorisés par la spécification. Le *livelock* ne nous intéresse pas puisque nous ne considérons que des ioSTS sans boucle d'actions internes. L'opération de suspension transforme les traces en traces de suspension [Tre99], c'est-à-dire des traces pouvant contenir des silences entre les actions. L'opération de suspension pour les ioSTS est définie comme suit, avec les effets escomptés sur la sémantique : $[\![\mathcal{M}^\delta]\!] = [\![\mathcal{M}]\!]^\delta$.

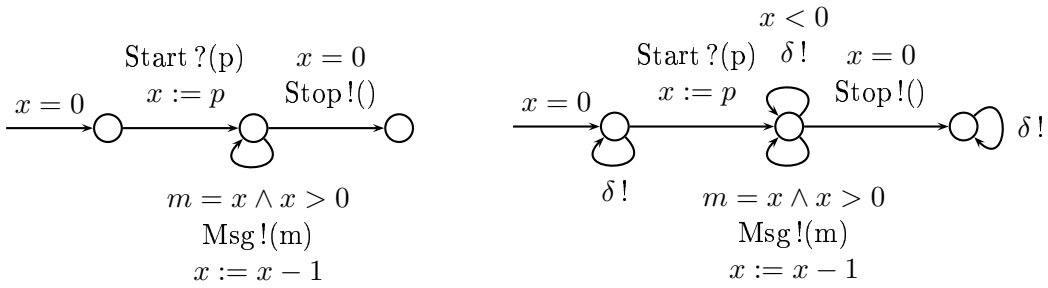
Définition 3.7 (Suspension) *Soit $\mathcal{M} = \langle V, \Theta, \Sigma, \mathcal{T} \rangle$ un ioSTS ayant pour alphabet $\Sigma = \Sigma_? \cup \Sigma_! \cup \Sigma_\tau$ avec $\delta \notin \Sigma$, l'automate suspendu $\mathcal{M}^\delta$ est le tuple $\langle V, \Theta, \Sigma^\delta, \mathcal{T}^\delta \rangle$, où :*

- *l'alphabet est augmenté d'une nouvelle sortie $\Sigma^\delta = \Sigma_? \cup \Sigma_{\delta!} \cup \Sigma_\tau$ avec $\Sigma_{\delta!} = \Sigma_! \cup \{\delta\}$,*
- *de nouvelles transitions sous forme de boucles étiquetées par δ sont ajoutées dans les états de blocage : $\mathcal{T}^\delta = \mathcal{T} \cup \{(\delta, p, G_\delta, (x := x)_{x \in V})\}$ avec*

$$G_\delta = \neg \left(\bigvee_{(a,p,G,A) \in \mathcal{T},\ a \in \Sigma_! \cup \Sigma_\tau} \exists \pi \in \mathcal{D}_{sig(a)} : G(\nu, \pi) \right) \tag{3.2}$$

G_δ est évaluée à vrai (true) lorsqu'aucune valeur ν des variables ni aucune valeur π des paramètres de communication ne peuvent être choisies de manière à ce qu'une sortie puisse être tirée. Une transition δ peut alors être tirée et boucler sur le même état. Notons que la satisfiabilité de G_δ est décidable puisque cette garde est la négation de la conjonction de gardes dont la satisfiabilité est décidable.

Exemple 3.2 *Considérons l'ioSTS $\mathcal{S}$ de la figure ?? (reproduit figure 3.4(a)). L'automate suspendu $\mathcal{S}^\delta$ correspondant est donné en figure 3.4(b). La garde $x < 0$ de la transition étiquetée par δ est obtenue en simplifiant l'expression $\neg(x = 0 \vee \exists m, m = x \wedge x > 0)$, qui correspond à la formule (3.2).*



(a) ioSTS $\mathcal{S}$ (b) ioSTS suspendu $\mathcal{S}^\delta$

FIG. 3.4 – Suspension d'un ioSTS

Si un ioSTS $\mathcal{M}$ modélise un système réactif, le comportement visible considéré lors du test est alors composé des traces de sa suspension $\mathcal{M}^\delta$. Nous noterons

$STraces(\mathcal{M}) = Traces(\mathcal{M}^\delta)$. Cet ensemble des traces suspendues sera considéré comme le comportement de référence pour tester la conformité par rapport à une spécification.

Comme présenté dans la partie I, nous générons des cas de test à partir d'une spécification $\mathcal{S}$. Pour cela, nous effectuons différentes opérations sur cette spécification afin de générer le testeur canonique à partir duquel seront sélectionnés les cas de test.

3.3 Génération du testeur canonique

Dans cette section, nous utilisons la méthode vue au chapitre 2 permettant de générer des tests à partir d'une spécification modélisée par un ioLTS. Pour cela, nous ferons un rappel sur la relation de conformité utilisée puis nous présenterons comment est obtenu le testeur canonique. Nous décrirons enfin les propriétés d'un cas de test généré par cette méthode.

3.3.1 Test de conformité à la ioco

La théorie du test à la ioco [Tre99], expliquée chapitre 1, peut être reformulée dans le contexte de spécifications décrites par des ioSTS. Ceci consiste essentiellement à préciser comment sont modélisés les différents acteurs : les spécifications, les implémentations et les cas de test, puis à définir formellement la conformité comme une relation entre spécifications et implémentations.

Les spécifications, implémentations et cas de test sont définis comme suit :

- la spécification est un ioSTS déterminisable $\mathcal{S} = \langle V^{\mathcal{S}}, \Theta^{\mathcal{S}}, \Sigma^{\mathcal{S}}, \mathcal{T}^{\mathcal{S}} \rangle$, avec $\Sigma^{\mathcal{S}} = \Sigma^{\mathcal{S}}_! \cup \Sigma^{\mathcal{S}}_?$: la spécification ne contient pas d'action interne. Sa sémantique est un ioLTS $[\![\mathcal{S}]\!] = S = \langle Q, q_0, \Lambda, \rightarrow \rangle$ avec $\Lambda = \Lambda_! \cup \Lambda_?$.
- l'implémentation est modélisée par un ioLTS (pas forcément déterministe) $I = (Q^I, Q^I_0, \Lambda_! \cup \Lambda_?, \rightarrow_I)$ ayant la même interface que $\mathcal{S}$. I est supposée être complète en entrée[1] et I^δ être son ioLTS suspendu.
- un cas de test pour une spécification $\mathcal{S}$ est un ioSTS déterministe $\mathcal{TC} = \langle V^{\mathcal{TC}}, \Theta^{\mathcal{TC}}, \Sigma^{\mathcal{TC}}, \mathcal{T}^{\mathcal{TC}} \rangle$, pour lequel $\Sigma^{\mathcal{TC}}_? = \Sigma^{\mathcal{S}}_!$ et $\Sigma^{\mathcal{TC}}_! = \Sigma^{\mathcal{S}}_?$ (les actions sont en miroir par rapport à la spécification). Le cas de test est équipé d'une variable $\mathsf{verdict} \in V^{\mathcal{TC}}$ de type énuméré : par exemple {none, fail, inconc}. Intuitivement, le verdict fail symbolise la non-conformité (l'implémentation est rejetée) et inconc signifie que les comportements désirés ne peuvent plus être atteints (voir section 4.2.2). D'autres verdicts peuvent être émis, comme nous le verrons également chapitre 4. Le cas de test est complet en entrée dans tous les états pour lesquels verdict = none ce qui lui permet d'être prêt à recevoir n'importe quelle sortie de l'implémentation, sauf quand un verdict est atteint et que l'exécution est arrêtée. Soit $TC = [\![\mathcal{TC}]\!] = \langle Q^{\mathrm{TC}}, q_0^{\mathrm{TC}}, \Lambda^{\mathrm{TC}}, \rightarrow_{\mathrm{TC}} \rangle$ la sémantique en termes

[1] Ceci permet à la composition entre l'implémentation et le cas de test de ne jamais se bloquer à cause d'entrées non implémentées.

d'ioLTS du cas de test. Nous notons par Fail (lorsque verdict = fail) et par Inconc (lorsque verdict = inconc) les sous-ensembles de Q^{TC} dans lesquels les verdicts sont émis.

Comme expliqué dans le chapitre 1 à la section 1.2, nous utilisons la relation de conformité ioco. Nous rappelons la caractérisation de cette relation : une implémentation I est ioco-conforme à une spécification $\mathcal{S}$, (I ioco $\mathcal{S}$), si

$$[STraces(\mathcal{S}) \cdot \Lambda^{\mathcal{S}}_{\delta!} \setminus STraces(\mathcal{S})] \cap STraces(I) = \emptyset$$

avec $\Lambda^{\mathcal{S}}_{\delta!} = \Lambda^{\mathcal{S}}_{!} \cup \{\delta\}$.

Exemple 3.3 *L'implémentation I_1 (figure 3.5) qui permet la trace $Start?(0) \cdot Msg!(0)$ n'est pas conforme à la spécification $\mathcal{S}$ décrite en figure ?? puisque cette trace n'est pas admissible par l'ioSTS $\mathcal{S}^{\delta}$ (figure 3.4). En effet, un message ne peut pas être émis si la variable x vaut 0. Le paramètre porté par l'action $Msg!$ ayant la même valeur que cette variable, $Msg!(0)$ ne peut être émis. La trace $Start?(1) \cdot \delta!$ (implémentation I_2) révèle également une non-conformité vis à vis de $\mathcal{S}$ puisque cette trace n'est pas admissible par la spécification. Par contre, la trace $Start?(1) \cdot Start?(1) \cdot Stop!$ (implémentation I_3) ne permet pas de conclure que l'implémentation est non-conforme à la spécification dans la mesure où $\mathcal{S}^{\delta}$ ne contraint pas les traces du système après l'occurrence du deuxième* Start ?.

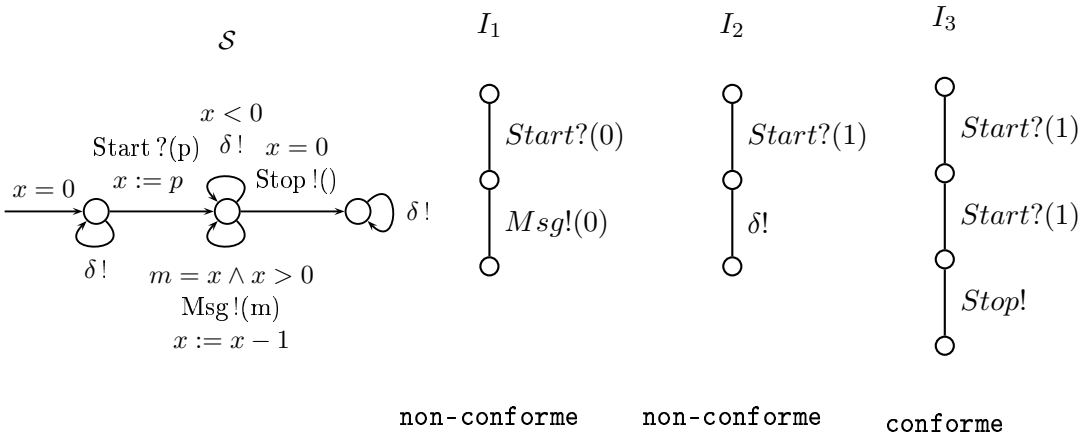


FIG. 3.5 – Conformité d'implémentations par rapport à une spécification $\mathcal{S}$

3.3.2 Testeur canonique

Un testeur canonique, pour une spécification et une relation donnée, permet de détecter toutes les implémentations qui ne sont pas conformes à la spécification pour

cette relation (voir chapitre 2). Afin d'obtenir ce testeur canonique, nous complétons en sortie la spécification suspendue déterminisée.

Définition 3.8 (Complétion en sortie) *Soit $\mathcal{M} = \langle V, \Theta, \Sigma, \mathcal{T} \rangle$ un ioSTS déterministe, le* complété en sortie *de $\mathcal{M}$ est l'ioSTS $\Sigma^!(\mathcal{M}) = \langle V', \Theta', \Sigma, \mathcal{T}' \rangle$ avec :*

- *$V' = V \cup \{\mathsf{verdict}\}$ avec* verdict *une variable de type énuméré* {none, fail}[2] *;*
- *$\Theta' = \Theta \wedge \mathsf{verdict} = \mathsf{none}$;*
- *$\mathcal{T}'$ est défini par les règles suivantes :*

$$\frac{t \in \mathcal{T}}{t \in \mathcal{T}'} \tag{3.3}$$

$$\frac{a \in \Sigma_! \quad G_a = \bigwedge_{(a,p,G,A) \in \mathcal{T}} \neg G}{[a(p) \ : \ G_a(v,p) \ ? \ \mathsf{verdict} := \mathsf{fail}] \ \in \ \mathcal{T}'} \tag{3.4}$$

L'ioSTS $\Sigma^!(\mathcal{M})$ est obtenu à partir de $\mathcal{M}$ en ajoutant une nouvelle variable verdict et, pour chaque transition de sortie n'existant pas dans $\mathcal{M}$, en ajoutant une transition dans $\Sigma^!(\mathcal{M})$ qui affecte la variable énumérée verdict à fail (équation 3.4). L'équation 3.3 indique que toutes les transitions de $\mathcal{M}$ sont conservées dans $\Sigma^!(\mathcal{M})$. Ainsi, toute sortie qui n'était pas admissible dans $[\![\mathcal{M}]\!]$ devient tirable dans $[\![\Sigma^!(\mathcal{M})]\!]$ et fait évoluer l'ioLTS dans un nouvel état pour lequel la variable verdict est égale à fail et à partir duquel aucune action n'est tirable (il s'agit d'un blocage). Le langage de ce nouvel ioSTS $\Sigma^!(\mathcal{M})$ satisfait le lemme suivant.

Lemme 3.4 *Soit $\mathcal{M}$ un ioSTS déterministe, alors*

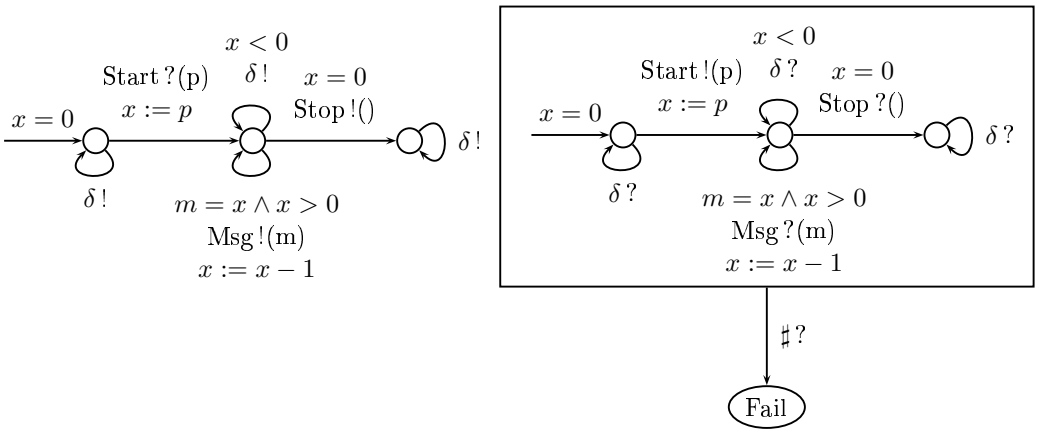
$$\mathit{Traces}(\Sigma^!(\mathcal{M}), \mathsf{Fail}_{\mathcal{M}}) = \mathit{Traces}(\mathcal{M}).\Lambda_!^{\mathcal{M}} \setminus \mathit{Traces}(\mathcal{M})$$

L'ensemble des traces reconnues par $\Sigma^!(\mathcal{M})$ menant à $\mathsf{Fail}_{\mathcal{M}}$ est exactement l'ensemble des traces non-conformes. Le complété en sortie est donc un testeur canonique pour la relation de conformité ioco. Il est le cas de test le plus général qu'on puisse obtenir puisqu'il reconnaît toutes les traces non-conformes. Nous noterons *Can($\mathcal{S}$)* le testeur canonique d'une spécification $\mathcal{S}$.

Exemple 3.4 *Nous reprenons la spécification suspendue vue précédemment à la figure 3.6(a). Le testeur canonique qui en résulte est représenté à la figure 3.6(b). Pour simplifier sa représentation, nous avons noté l'ensemble des transitions menant à* Fail *par le cadre fléché étiqueté par ♯? (l'action ♯? symbolisant les entrées possibles). Nous n'avons pas non plus représenté la négation des gardes.*

Le testeur canonique étant en soi un cas de test, il doit vérifier certaines propriétés.

[2]Nous ajouterons la valeur inconc à cette variable lors de la phase de sélection chapitre 4.



(a) ioSTS suspendu $\mathcal{S}^{\delta}$

(b) Testeur canonique *Can(S)*

FIG. 3.6 – Testeur canonique de $\mathcal{S}$

3.3.3 Propriétés du testeur canonique

Les propriétés vues au chapitre 1 à la sous-section 1.3.3 sont ici conservées puisque la théorie du test reste la même.

Proposition 3.1 *Soit TS, une suite de tests générée à partir de la spécification* $\mathcal{S}$*,*

- *TS est non-biaisée ssi* $\bigcup_{TC \in TS}$ *Traces(TC,* Fail*)* $\subseteq$ *Traces(Can(*$\mathcal{S}$*),* Fail*),*
- *TS est exhaustive ssi* $\bigcup_{TC \in TS}$ *Traces(TC,* Fail*)* $\supseteq$ *Traces(Can(*$\mathcal{S}$*),* Fail*).*

Cette proposition indique que des cas de test non-biaisés sont des observateurs extraits du testeur canonique et qu'une suite de tests exhaustive doit rejeter toutes les implémentations rejetées par *Can(*$\mathcal{S}$*)* et ainsi couvrir toutes les détections de non-conformité.

Comme expliqué dans le chapitre 2, nous allons maintenant sélectionner des cas de test sur ce testeur canonique et ainsi permettre de tester des comportements précis du système. Cependant, nous n'allons pas utiliser pour cela des objectifs de test mais des propriétés si possible préalablement vérifiées sur la spécification.

Chapitre 4

Sélection des tests pour des ioSTS

Dans la suite de cette partie du document, nous allons utiliser à la fois la notion de vérification, afin de déterminer si la spécification satisfait/viole la propriété, mais aussi de test afin de détecter s'il y a non-conformité et si l'implémentation ou la spécification viole/satisfait la propriété. Pour cela, nous allons d'abord définir comment nous représentons ces propriétés et comment nous les vérifions sur la spécification, puis nous décrirons comment un cas de test est sélectionné à partir du testeur canonique de la spécification et de ces propriétés.

4.1 Vérification : propriétés de sûreté et d'accessibilité

La vérification consiste à se poser la question : étant donné un système réactif modélisé par un ioSTS $\mathcal{M}$, et une propriété ψ définie sur ses traces, est-ce que $\mathcal{M}$ satisfait ψ ? La propriété ψ est une propriété que doivent satisfaire les traces de la spécification.

Nous pouvons travailler sur deux types de propriétés : les propriétés de sûreté et les propriétés d'accessibilité. Nous modélisons ces propriétés par le biais d'observateurs, qui sont des ioSTS déterministes ayant un prédicat particulier (que nous pouvons également voir comme un ensemble particulier d'états). La propriété de sûreté (respectivement d'accessibilité) est violée (respectivement satisfaite) quand un tel prédicat est satisfait (ou un tel état atteint). Il existe deux types d'observateurs, correspondant aux deux types des propriétés. Par convention, un observateur dit "négatif" représente la négation d'une propriété de sûreté tandis qu'un observateur dit "positif" représente une propriété d'accessibilité.

Définition 4.1 (Observateur) *Un observateur est un ioSTS déterministe complet* $\omega = \langle V^\omega, \Theta^\omega, \Sigma^\omega, \mathcal{T}^\omega \rangle$ *tel que :*

- Θ^ω *est la condition initiale ;*
- $\mathcal{T}^\omega$ *est l'ensemble fini des transitions symbolique ;*
- $\Sigma^\omega = \Sigma^\omega_! \cup \Sigma^\omega_?$ *: il n'y a pas d'action interne ;*
- V^ω *contient une variable* pc^ω *servant de compteur au programme (dans nos exemples, elle correspond aux localités). Cette variable a pour domaine* $\mathcal{D}_{pc^\omega}$ *avec*

$\mathsf{special} \in \mathcal{D}_{pc^\omega}$ *(*special *représente une localité distinguée). L'ensemble des états distingués pour lesquels* ($pc^\omega = \mathsf{special}$)*, est noté* Special*. Du point de vue de la sémantique* $\mathsf{Special} \in Q^\omega$.

Un observateur $(\omega, \mathsf{Special})$ *est compatible avec un ioSTS* $\mathcal{M}$ *si* ω *est compatible avec* $\mathcal{M}$ *(définition 3.4). L'ensemble des observateurs compatibles avec* $\mathcal{M}$ *est noté* $\Omega(\mathcal{M})$.

Le fait que l'observateur soit complet permet à celui-ci de ne pas restreindre les exécutions de la spécification (voir section 3.3).

Le langage[1] reconnu par la propriété représentée par l'observateur est l'ensemble des traces de l'observateur permettant d'atteindre la localité special. Ce langage est donné par le lemme 4.1.

Lemme 4.1

$$\mathit{Traces}(\omega, \mathsf{Special}_\omega) = \{\sigma \in \Lambda_\omega^* \mid \exists q_o \in Q_\omega^o,\ \exists s \in \mathsf{Special}_\omega \ .\ q_o \xrightarrow{\sigma}_\omega s\} \tag{4.1}$$

4.1.1 Vérification de propriétés de sûreté

Une propriété de sûreté permet de spécifier que quelque chose de mauvais n'arrivera jamais. Pour vérifier qu'une spécification satisfait une propriété de sûreté, il faut donc vérifier que celle-ci est satisfaite pour toutes les séquences de la spécification ou s'assurer qu'aucune séquence ne viole la propriété. Pour cela, la méthode classique en vérification est de prendre la négation de la propriété de sûreté et de vérifier si celle-ci peut être satisfaite par la spécification. Si la négation de la propriété est satisfaite, alors la propriété est violée :

$$\mathit{Traces}(\mathcal{S}) \subseteq \psi \ \Leftrightarrow\ \mathit{Traces}(\mathcal{S}) \cap \overline{\psi} = \emptyset.$$

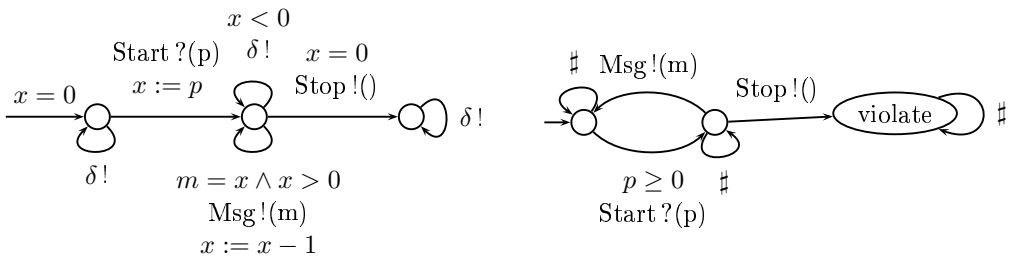
Remarque : Vérifier la violation d'une propriété de sûreté revient à vérifier la satisfaction d'une propriété d'accessibilité.

Une propriété de sûreté, dont la négation est modélisée par un observateur $(\omega, \mathsf{Violate}_\omega)$, est violée si $\mathsf{Violate}_\omega$ est atteint. Cet observateur ω (appartenant à $\Omega(\mathcal{M})$) définit une propriété de sûreté sur $(\Lambda^!_\mathcal{M} \cup \Lambda^?_\mathcal{M})^*$. Cette propriété est satisfaite par les séquences ne menant pas à Violate, c'est-à-dire les séquences appartenant à $(\Lambda^!_\mathcal{M} \cup \Lambda^?_\mathcal{M})^* \setminus \mathit{Traces}(\omega, \mathsf{Violate}_\omega)$ (et seulement ces séquences). En particulier, si $\mathcal{M}$ correspond à l'ioSTS suspendu $\mathcal{S}^\delta$ d'un ioSTS donné $\mathcal{S}$, alors la propriété est satisfaite par un sous-ensemble de $(\Lambda^!_\mathcal{S} \cup \{\delta\} \cup \Lambda^?_\mathcal{S})^*$.

Exemple 4.1 *L'observateur* ω_1 *de la figure 4.1(b) décrit une propriété de sûreté qui code le fait qu'entre une action d'entrée* Start *avec un paramètre* $p \geq 0$*, et une action de sortie* Stop*, le système doit émettre au moins une fois l'action de sortie* Msg*. L'ensemble des localités de violation est ici réduit au singleton* {violate}*. Les boucles* $\sharp$ *correspondent*

[1] Un observateur ne contenant pas d'action interne, le langage correspond à l'ensemble des traces.

à l'ensemble des actions (incluant l'action spécifique de blocage δ) qui ne sont pas portées par d'autres transitions symboliques. La boucle $\sharp$ sur l'état initial correspond également à la transition portant l'action Start *avec $p < 0$ comme paramètre.*



(a) ioSTS $\mathcal{S}^{\delta}$ (b) observateur négatif ω_1

FIG. 4.1 – Exemple d'observateur "négatif" ω_1.

Un ioSTS satisfait un observateur "négatif" si aucune trace de cet ioSTS n'est reconnue par l'observateur.

Définition 4.2 *Soient $\mathcal{M}$ un ioSTS et $(\omega, \mathsf{Violate}_\omega) \in \Omega(\mathcal{M})$ un observateur compatible avec $\mathcal{M}$, alors $\mathcal{M}$ satisfait $(\omega, \mathsf{Violate}_\omega)$, noté $\mathcal{M} \models^{-} (\omega, \mathsf{Violate}_\omega)$, si*

$$\mathit{Traces}(\mathcal{M}) \cap \mathit{Traces}(\omega, \mathsf{Violate}_\omega) = \emptyset.$$

Notons $\mathsf{Violate}_{\mathcal{M}||\,\omega}$ l'ensemble des états de $[\![\mathcal{M}||\omega]\!]$ pour lesquels $pc^{\omega} = \mathsf{violate}$.

D'après le lemme (3.1)

$$\mathit{Traces}(\mathcal{M}||\omega, \mathsf{Violate}_{\mathcal{M}||\,\omega}) \quad = \quad \mathit{Traces}(\mathcal{M}) \cap \mathit{Traces}(\omega, \mathsf{Violate}_\omega)$$

Ainsi, vérifier si $\mathcal{M} \models^{-} (\omega, \mathsf{Violate}_\omega)$ revient à vérifier que l'ensemble $\mathit{Traces}(\mathcal{M}||\omega, \mathsf{Violate}_{\mathcal{M}||\,\omega})$ est vide, ce qui peut être fait en vérifiant que $\mathsf{Violate}_{\mathcal{M}||\,\omega}$ n'est pas accessible depuis l'ensemble des états initiaux de $[\![\mathcal{M}||\omega]\!]$, ou inversement, en vérifiant que l'ensemble des états initiaux de $[\![\mathcal{M}||\omega]\!]$ n'est pas co-accessible depuis $\mathsf{Violate}_{\mathcal{M}||\,\omega}$.

Toutefois, le calcul des ensembles d'états accessibles et co-accessibles n'est en général pas calculable. Cet obstacle peut être contourné par l'utilisation d'approximations comme, par exemple, les approximations calculées par interprétation abstraite [CC77]. Ces techniques vont nous permettre d'effectuer une analyse approchée par sur-approximation des états accessibles et co-accessibles. Nous utilisons également ces techniques lors de la phase de sélection de test (chapitre 4). C'est dans ce but que l'outil STG (Symbolic Test Generation) [CJRZ02], développé par l'équipe VerTeCS, est interfacé avec un outil appelé NBac [Jea03]. Dans un premier temps, STG calcule automatiquement le produit $\omega||\mathcal{M}$; puis NBac réalise automatiquement un calcul approché des états co-accessibles (depuis les états de violation) sur le produit et

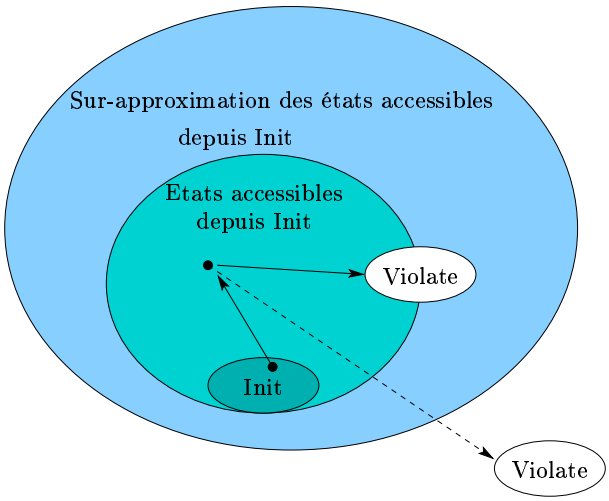


FIG. 4.2 – Illustration de la sur-approximation des états accessibles depuis les états initiaux Init lors de la vérification d'une propriété de sûreté

supprime les états non co-accessibles sur la sur-approximation. Il effectue ensuite une analyse approchée des états accessibles (depuis les états initiaux) afin de simplifier les gardes des transitions.

Ces outils peuvent être utilisés pour prouver, par exemple, que l'ioSTS $\mathcal{S}^\delta$ décrit en figure 4.1(a) ne satisfait pas la propriété dont la négation est représentée par l'observateur ω_1 (Figure 4.1(b)). En effet, l'entrée Start avec $p = 0$ comme paramètre permet que l'action de sortie Stop soit émise (sans que l'action msg ne le soit), ce qui viole la propriété. Si l'observateur décrivait presque la même propriété, mais avec le paramètre p porté par l'action Start strictement supérieur à zéro et non plus simplement supérieur ou égal à zéro, l'ioSTS $\mathcal{S}^\delta$ satisferait la propriété. Il est possible de montrer que les états de violation de la propriété ne sont pas accessibles. D'un autre coté, il nous est impossible avec ces outils d'affirmer qu'on pourra toujours prouver automatiquement qu'un ioSTS ne satisfait pas un observateur "négatif". En effet, si les états de violation de la propriété ne sont pas inclus dans la sur-approximation des états accessibles (et co-accessibles), alors les états de violation ne sont pas non plus inclus dans l'ensemble (exact) des états accessibles (et co-accessibles) (flèche à tirets menant à Violate figure 4.2). Nous sommes donc sûrs que la propriété de sûreté n'est pas violée. Par contre, si ces états sont inclus dans la sur-approximation (flèche à trait continu menant à Violate figure 4.2), alors on ne peut pas savoir s'ils appartiennent à l'ensemble exact des états accessibles (et co-accessibles). On ne peut donc pas savoir si la propriété est réellement violée ou si la violation apparente est due à la sur-approximation de l'analyse approchée.

Remarque : Notons qu'il est possible de voir le complété en sortie d'un ioSTS $\mathcal{M}$ comme un observateur en choisissant $\mathsf{Fail}_{\mathcal{M}}$ comme étant l'ensemble des états de violation. En se basant sur cette interprétation, la proposition suivante nous permet

de conclure que vérifier la conformité entre une implémentation et une spécification se réduit à la vérification d'une propriété de sûreté (la propriété dont la négation est représentée par l'observateur $(\Sigma^!(det(\mathcal{S}^\delta)), \mathsf{Fail}_{det(\mathcal{S}^\delta)})$.

Proposition 4.1 *I ioco* $\mathcal{S}$ *ssi* $I^\delta \models^- (\Sigma^!(det(\mathcal{S}^\delta)), \mathsf{Fail}_{det(\mathcal{S}^\delta)}))$.

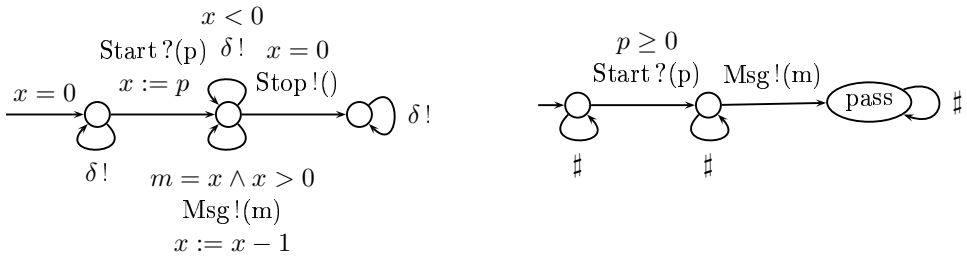
Cette proposition nous indique également que l'ioSTS $\Sigma^!(det(\mathcal{S}^\delta))$ est bien un testeur canonique pour la relation de conformité ioco.

4.1.2 Vérification de propriétés d'accessibilité

Au contraire des propriétés de sûreté, les propriétés d'accessibilité s'intéressent à montrer que quelque chose "de bon" se produit. Pour vérifier qu'une spécification vérifie une propriété d'accessibilité, il suffit de vérifier qu'au moins une séquence satisfait celle-ci.

Une propriété d'accessibilité, modélisée par un observateur $(\omega, \mathsf{Pass}_\omega)$, est donc satisfaite si l'état Pass_ω est atteignable par au moins une exécution.

Exemple 4.2 *L'observateur* ω_2 *(figure 4.3(b)) décrit une propriété d'accessibilité qui code le fait qu'après une action d'entrée* Start *avec un paramètre* $p \geq 0$*, le système doit émettre l'action de sortie* Msg*. L'ensemble des localités de satisfaction est ici réduit au singleton* {pass}.



(a) ioSTS $\mathcal{S}^\delta$ (b) observateur positif ω_2

FIG. 4.3 – Exemple d'observateur "positif" ω_2.

Un ioSTS satisfait un observateur "positif" si au moins une trace de cet ioSTS est reconnue par l'observateur.

Définition 4.3 *Soient* $\mathcal{M}$ *un ioSTS et* $(\omega, \mathsf{Pass}_\omega) \in \Omega(\mathcal{M})$ *un observateur compatible avec* $\mathcal{M}$*, alors* $\mathcal{M}$ *satisfait* $(\omega, \mathsf{Pass}_\omega)$*, noté* $\mathcal{M} \models^+ (\omega, \mathsf{Pass}_\omega)$*, si*

$$Traces(\mathcal{M}) \cap Traces(\omega, \mathsf{Pass}_\omega) \neq \emptyset.$$

Notons $\mathsf{Pass}_{\mathcal{M}||\,\omega}$ l'ensemble des états de $[\![\mathcal{M}||\omega]\!]$ pour lesquels $pc^{\omega} = \mathsf{pass}$.

D'après le lemme (3.1)

$$\mathit{Traces}(\mathcal{M}||\omega, \mathsf{Pass}_{\mathcal{M}||\,\omega}) \quad = \quad \mathit{Traces}(\mathcal{M}) \cap \mathit{Traces}(\omega, \mathsf{Pass}_{\omega})$$

Ainsi, vérifier si $\mathcal{M} \models^{+} (\omega, \mathsf{Pass}_{\omega})$ revient simplement à vérifier que l'ensemble $\mathit{Traces}(\mathcal{M}\|\omega, \mathsf{Pass}_{\mathcal{M}||\,\omega})$ est non vide.

Comme pour les propriétés de sûreté, nous pouvons nous servir des outils STG [CJRZ02] et NBac [Jea03] pour prouver, par exemple, que l'ioSTS $\mathcal{S}^{\delta}$ décrit en figure 4.3(a) ne satisfait pas l'observateur ω_2 (figure 4.3(b)). En effet, nous pouvons prouver que la propriété d'accessibilité n'est pas satisfaite. Il est ainsi possible de montrer que pour $p = 0$, les localités de satisfaction de la propriété ne sont pas accessibles. En effet, après réception de l'action Start avec $p = 0$ comme paramètre, l'action de sortie Msg ne peut plus être émise et donc la propriété ne peut plus être satisfaite. D'un autre côté, il nous est impossible avec ces outils d'affirmer que nous pourrons toujours prouver automatiquement qu'un ioSTS satisfait un observateur "positif". En effet, l'accessibilité des localités pass peut être due au fait que nous avons sur-approximé l'ensemble des états accessibles pour réaliser le calcul.

4.1.3 Combinaison d'observateurs

Le produit parallèle entre deux observateurs $(\omega, \mathsf{Special}_{\omega})$ et $(\varphi, \mathsf{Special}_{\varphi})$ peut également être vu comme une propriété (nous verrons l'utilité de ce type de combinaisons dans la section 4.2.1). Un choix naturel est de munir le produit $\omega||\varphi$ des états distingués $\mathsf{Special}_{\omega} \times \mathsf{Special}_{\varphi}$; d'après le lemme 3.1, les traces de ce nouvel ioSTS est alors donné par :

$$\mathit{Traces}(\omega||\varphi, \mathsf{Special}_{\omega} \times \mathsf{Special}_{\varphi}) = \mathit{Traces}(\omega, \mathsf{Special}_{\omega}) \cap \mathit{Traces}(\varphi, \mathsf{Special}_{\varphi})$$

On remarque que le produit parallèle de deux observateurs est lui-même un observateur ayant comme prédicat particulier ($pc^{\omega} = \mathsf{special}_{\omega}$, $pc^{\varphi} = \mathsf{special}_{\varphi}$). Si les deux observateurs sont tous deux des observateurs "négatifs" (respectivement "positifs"), alors leur produit peut être vu comme une propriété de sûreté (respectivement d'accessibilité).

Ainsi, l'observateur satisfaisant les traces $\mathit{Traces}(\omega||\varphi, \mathsf{Violate}_{\omega} \times \mathsf{Violate}_{\varphi})$ décrit le fait que les propriétés correspondant aux observateurs $(\omega, \mathsf{Violate}\omega)$ et $(\varphi, \mathsf{Violate}_{\varphi})$ sont violées toutes les deux. Il est bien évidemment possible de choisir d'autres ensembles d'états de violation. Ainsi, $\mathsf{Violate}_{\omega} \times (Q_{\varphi} \setminus \mathsf{Violate}_{\varphi})$ indiquera la violation de ω mais pas de φ, etc.

De même, l'observateur satisfaisant les traces $\mathit{Traces}(\omega||\varphi, \mathsf{Pass}_{\omega} \times \mathsf{Pass}_{\varphi})$ décrit le fait que les propriétés correspondant aux observateurs $(\omega, \mathsf{Pass}\omega)$ et $(\varphi, \mathsf{Pass}_{\varphi})$ sont satisfaites toutes les deux.

Il est également possible de faire le produit parallèle d'un observateur "négatif" avec un observateur "positif". Dans ce cas, nous gardons distinctes les significations des deux propriétés.

La vérification de la spécification peut ne pas aboutir. Cela n'a pas d'incidence sur la suite des évènements : si on n'a pas pu établir que la spécification satisfait ou viole la propriété, nous pourrons le vérifier lors de l'exécution du cas de test sur l'implémentation car nous utilisons ces propriétés comme guide lors de la sélection des tests.

4.2 Sélection des tests par des propriétés

Nous avons proposé dans le chapitre 3 une méthode permettant un testeur canonique à partir d'une spécification, nous allons maintenant généraliser la sélection vue au chapitre 2 (section 2.2) en utilisant une ou plusieurs propriété(s) (de sûreté ou d'accessibilité) comme guide. Dans ce cadre, un cas de test cherche à détecter des violations de la propriété de sûreté et/ou des satisfactions de la propriété d'accessibilité par l'implémentation et des violations de la conformité entre l'implémentation et la spécification. De plus, si la phase de vérification (section 4.1) n'a pu être formellement réalisée (la tentative de vérification de la propriété sur la spécification a échoué), le cas de test généré peut toujours servir, lorsqu'il est exécuté sur l'implémentation, à détecter des violations/satisfactions de la propriété par la spécification. Nous montrons également que les cas de test générés par notre méthode retournent toujours des verdicts corrects.

En principe, un testeur canonique est suffisant pour détecter toutes les implémentations qui ne seraient pas conformes à une spécification donnée. Toutefois, notre but est de détecter, en plus d'une éventuelle non-conformité, une violation/satisfaction potentielle d'autres propriétés de sûreté/accessibilité qui peuvent, par exemple, provenir d'exigences décrites dans un cahier des charges. Les observateurs (cf. définition 4.1) codant de telles propriétés peuvent également servir comme mécanisme de sélection pour les tests. En s'appuyant sur le lemme 3.1, le produit entre un observateur et le testeur canonique peut être utilisé pour définir un sous-ensemble des traces "intéressantes" parmi toutes les traces possibles que le testeur canonique peut générer.

4.2.1 Composition avec une propriété

Nous rappelons que pour un ioSTS $\mathcal{S}$ et un observateur compatible $\omega \in \Omega(\mathcal{S})$, l'ioSTS $\omega \times \mathit{Can}(\mathcal{S})$ peut également être interprété comme un observateur de $\mathcal{S}$ en choisissant convenablement les ensembles d'états de violation/satisfaction. Ainsi, suivant le choix de cet ensemble d'états, la signification de l'observateur et donc des tests qui en seront dérivés à partir d'une spécification sera différente.

Dans le reste de cette sous-section, toute composition entre le testeur canonique de la spécification $Can(\mathcal{S})$ et une propriété de sûreté ω^+ ou une propriété d'accessibilité ω^- formera un graphe de test, qui, après sélection, pourra être vu comme un cas de test raffinant le testeur canonique. Ce graphe de test explicitera les violations/satisfactions de la propriété de sûreté/accessibilité par des prédicats (les verdicts).

Dans la suite de cette section, nous allons présenter les compositions entre le testeur canonique et les propriétés (soit de sûreté, soit d'accessibilité), puis nous ferons une synthèse des verdicts pouvant être émis lors de l'exécution du cas de test sur l'implémentation.

4.2.1.1 Composition avec une propriété de sûreté

Notons $test_{\omega^-}(\mathcal{S}) = Can(\mathcal{S}) \times \omega^-$ la composition entre le testeur canonique et une propriété de sûreté représentée par ω^-. Cette composition permet d'obtenir trois verdicts différents, suivant les valeurs des variables pc de ω^- et $\mathsf{verdict}$ de $Can(\mathcal{S})$. Si une trace exécutée sur l'implémentation I permet d'atteindre un état pour lequel :

- $\mathsf{pc} = \mathsf{violate}$ et $\mathsf{verdict} \neq \mathsf{fail}$ alors une violation de la propriété tant par la spécification que par l'implémentation est détectée et le verdict **Violate** est émis. En effet, la variable pc a pour valeur $\mathsf{violate}$, ce qui signifie que la propriété a été violée par l'implémentation. Comme celle-ci est conforme à sa spécification, nous pouvons en déduire que la spécification viole aussi la propriété.
- $\mathsf{pc} \neq \mathsf{violate}$ et $\mathsf{verdict} = \mathsf{fail}$ alors il y a non-conformité entre implémentation et spécification et le verdict **Fail** est émis. Nous ne pouvons rien conclure de précis sur la propriété, nous savons juste que la trace menant à la non-conformité la satisfait.
- $\mathsf{pc} = \mathsf{violate}$ et $\mathsf{verdict} = \mathsf{fail}$ alors l'implémentation, d'une part ne satisfait pas la propriété définie par $(\omega^-, \mathsf{violate})$ et d'autre part n'est pas conforme à la spécification $\mathcal{S}$. Dans cette situation, le verdict **FailViolate** sera émis. La spécification satisfait alors la propriété pour les préfixes stricts de cette trace.
- $\mathsf{pc} \neq \mathsf{violate}$ et $\mathsf{verdict} \neq \mathsf{fail}$ alors rien de particulier (bon ou mauvais) ne s'est produit. La trace satisfait la propriété et est conforme. Nous ne pouvons rien en conclure de définitif, aucun verdict n'est émis et on continue les tests.

Un récapitulatif des verdicts possibles lors de la composition entre le testeur canonique et un observateur négatif est donné au tableau 4.1.

Un observateur est non-intrusif, on a donc :

$$Traces(test_{\omega^-}(\mathcal{S})) \subseteq Traces(Can(\mathcal{S}))$$

et

$$Traces(test_{\omega^-}(\mathcal{S}), \mathsf{Fail}) = Traces(test_{\omega^-}(\mathcal{S})) \cap Traces(Can(\mathcal{S}), \mathsf{Fail}),$$

ce qui signifie que $test_{\omega^-}(\mathcal{S})$ détecte chaque non-conformité à travers ses traces. $test_{\omega^-}(\mathcal{S})$ est donc un cas de test non-biaisé. Nous noterons aussi que

$$Traces(test_{\omega^-}(\mathcal{S}), \mathsf{Violate}) = STraces(\mathcal{S}) \cap Traces(\omega^-, \mathsf{Violate}).$$

pc_{ω^-}	$\mathsf{verdict}_{Can(S)}$	Conclusion immédiate	Conclusion induite	Verdict
violate		Violation par I, Trace conforme	Violation par S	**Violate**
	fail	Non-conformité		**Fail**
violate	fail	Violation par I, Non-conformité		**FailViolate**

TAB. 4.1 – Récapitulatif des verdicts de la composition avec une propriété de sûreté

Suivant les états considérés Fail ou Violate, l'ioSTS $test_{\omega^-}(\mathcal{S})$ peut être considéré soit comme un observateur des traces non-conformes soit comme un observateur des traces des exécutions violant la propriété.

Exemple 4.3 *Reprenons nos exemples des figures 3.6(b) représentant le testeur canonique de $\mathcal{S}$ et 4.1(b) représentant une propriété de sûreté dans les figures 4.4(a) et 4.4(b). La figure 4.4(c) représente le graphe de test calculé par la composition de Can($\mathcal{S}$) et ω_1.*

Nous constatons que sur cet exemple, la propriété est violée par l'implémentation si la sortie Stop *est émise sans qu'aucun message ne l'ait été (localités contenant* Violate*). De plus, si la sortie* Stop *est émise alors que la variable x de la spécification est différente de 0 (ce qui n'est pas prévu par la spécification), la non-conformité est détectée (localité* FailViolate*).*

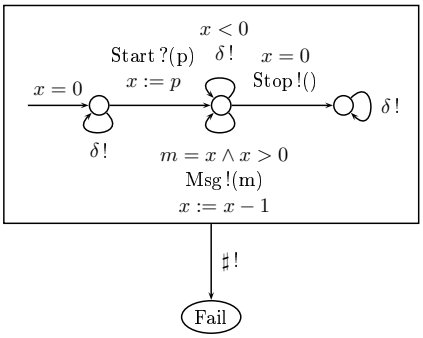
Si la propriété de sûreté avait eu $p > 0$ comme garde de la transition portant l'action Start*, alors cette propriété ne pourrait être violée par la spécification car aucune localité contenant* Violate *ne pourrait être atteinte sans qu'il y ait non-conformité. Dans notre exemple, la spécification viole la propriété seulement lorsque la sortie* Stop *est émise alors que x vaut 0 (en d'autres termes, alors qu'on a entré* Start?(0)*), ce qui est permis par la spécification.*

4.2.1.2 Composition avec une propriété d'accessibilité

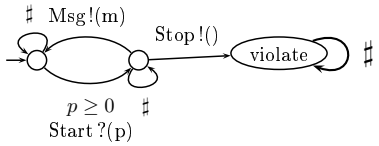
Notons $test_{\omega^+}(\mathcal{S}) = Can(\mathcal{S}) \times \omega^+$ la composition entre le testeur canonique et une propriété d'accessibilité représentée par ω^+. Cette composition permet d'obtenir trois verdicts différents, suivant les valeurs des variables pc de ω^+ et verdict de $Can(\mathcal{S})$. Si une trace exécutée sur l'implémentation I permet d'atteindre un état pour lequel :

- pc = pass et verdict $\neq$ fail alors l'implémentation et la spécification satisfont la propriété d'accessibilité et le verdict **Pass** est émis. A ce stade, aucune non-conformité n'a été détectée[2], la satisfaction de la propriété par l'implémentation implique donc la satisfaction de la propriété par la spécification.
- pc $\neq$ pass et verdict = fail alors il y a non-conformité entre implémentation et spécification et le verdict **Fail** est émis. Nous ne pouvons rien conclure par rapport

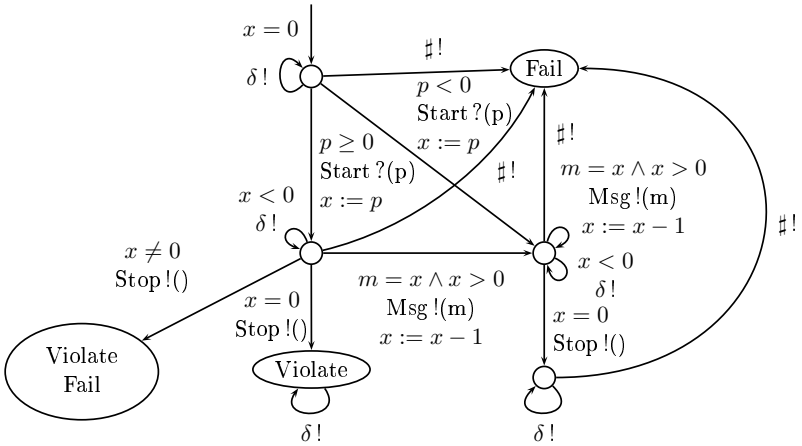
[2]Ce verdict **Pass** correspond au verdict habituel du même nom de la littérature sur le test de conformité.



(a) Testeur canonique *Can(S)*



(b) Observateur négatif ω_1



(c) Graphe de test de $Can(\mathcal{S}) \times \omega_1$

FIG. 4.4 – Composition de *Can(S)* avec ω_1

à la propriété, seulement que cette trace ne la satisfait pas.

- pc = pass et verdict = fail alors l'implémentation satisfait la propriété d'accessibilité mais il y a non-conformité entre implémentation et spécification. On en déduit que, pour cette trace, il y a incohérence entre la spécification et la propriété d'accessibilité vu que $\mathcal{S}$ ne contient pas cette trace. Le verdict **FailPass** est émis.
- pc $\neq$ pass et verdict $\neq$ fail alors rien de particulier (bon ou mauvais) ne s'est produit. La trace est conforme mais nous ne pouvons rien en conclure, aucun verdict n'est émis et on continue les tests.

Un récapitulatif des verdicts possibles lors de la composition entre le testeur canonique et un observateur positif est donné au tableau 4.2.

pc_{ω^+}	$\mathsf{verdict}_{Can(\mathcal{S})}$	Conclusion immédiate	Conclusion induite	Verdict
pass		Satisfaction par I, Trace conforme	Satisfaction par S	**Pass**
	fail	Non-conformité		**Fail**
pass	fail	Satisfaction par I, Non-conformité		**FailPass**

TAB. 4.2 – Récapitulatif des verdicts de la composition avec une propriété d'accessibilité

Comme pour un observateur négatif, $test_{\omega^+}(\mathcal{S})$ détecte chaque non-conformité à travers ses traces puisque l'observateur est non-intrusif :

$$Traces(test_{\omega^+}(\mathcal{S}), \mathsf{Fail}) = Traces(test_{\omega^+}(\mathcal{S})) \cap Traces(Can(\mathcal{S}), \mathsf{Fail}).$$

$test_{\omega^+}(\mathcal{S})$ est donc un cas de test non-biaisé. Nous noterons aussi que

$$Traces(test_{\omega^+}(\mathcal{S}), \mathsf{Pass}) = STraces(\mathcal{S}) \cap Traces(\omega^+, \mathsf{Pass}).$$

Suivant les états considérés Fail ou Pass, l'ioSTS $test_{\omega^+}(\mathcal{S})$ peut être considéré soit comme un observateur des traces non-conformes soit comme un observateur des traces des exécutions satisfaisant la propriété.

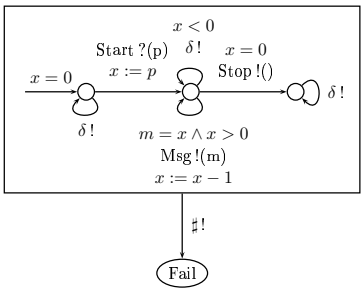
Exemple 4.4 *Reprenons nos exemples de testeur canonique de $\mathcal{S}$ et de la propriété d'accessibilité représentés figures 4.5(a) et 4.5(b). La figure 4.5(c) représente le graphe de test calculé par composition de $Can(\mathcal{S})$ et ω_2.*

Nous constatons que, sur cet exemple, la propriété est satisfaite par l'implémentation dès qu'un message est émis (états contenant Pass*). Comme il n'y a aucune contrainte particulière sur l'émission de* Msg *dans l'observateur représentant la propriété, on peut constater que la propriété peut être :*

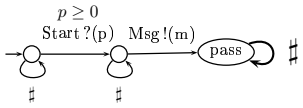
- *soit satisfaite à la fois par la spécification et par l'implémentation (état* Pass*),*
- *soit satisfaite seulement par l'implémentation alors que celle-ci n'est pas conforme à la spécification (état* FailPass*).*

4.2.1.3 Ensemble des verdicts

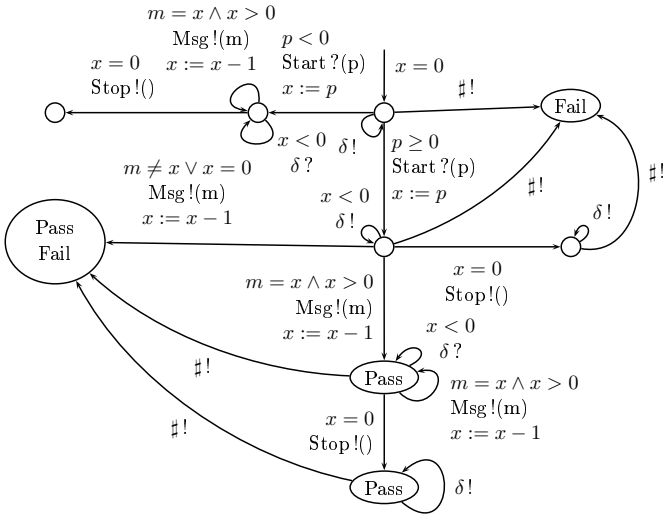
Afin de récapituler l'ensemble des verdicts, le tableau 4.3 résume les différentes localités possibles et les verdicts correspondants (la variable Verdict sera utilisée par la suite comme localité). Pour cela, nous allons considérer la composition du testeur canonique avec à la fois une propriété de sûreté et une propriété d'accessibilité. Nous noterons cette composition $test(\mathcal{S}) = Can(\mathcal{S}) \times \omega^+ \times \omega^-$. Les cases vides du tableau correspondent aux valeurs des variables autres que pass, violate et fail.



(a) Testeur canonique *Can(S)*



(b) Observateur positif ω_2



(c) Graphe de test de $Can(\mathcal{S}) \times \omega_2$

FIG. 4.5 – Composition de *Can(S)* avec ω_2

$test(\mathcal{S}) = Can(\mathcal{S}) \times \omega^+ \times \omega^-$ peut être vu comme un cas de test raffinant le testeur canonique puisque des violations/satisfactions de propriétés de sûreté/accessibilité par l'implémentation peuvent être détectées.

Définition 4.4 *Soient* $test(\mathcal{S}) = Can(\mathcal{S}) \times \omega^+ \times \omega^-$ *un cas de test équipé de localités dites "verdicts" et* I *une implémentation compatible avec* $\mathcal{S}$. I^δ *est alors compatible avec* $\mathcal{S}^\delta$, *et la composition parallèle* $I^\delta \times test(\mathcal{S})$ *est alors définie. Pour chaque trace* $\sigma \in Traces(I^\delta \times test(\mathcal{S}))$ *et chaque verdict*

$$W \in \{\mathbf{Pass}, \mathbf{Violate}, \mathbf{Fail}, \mathbf{FailPass}, \mathbf{FailViolate}, \mathbf{PassViolate}, \mathbf{FailPassViolate}\},$$

la trace σ *donne le verdict* W *pour l'implémentation* I *si* $\sigma \in Traces(test(\mathcal{S}), W)$.

La proposition suivante caractérise les verdicts donnés par l'exécution d'un cas de test

	$\mathsf{pc}_{\omega+}$	$\mathsf{verdict}_{Can(\mathcal{S})}$	$\mathsf{pc}_{\omega-}$	Verdict
(1)	pass			**Pass**
(2)			violate	**Violate**
(3)		fail		**Fail**
(4)	pass	fail		**FailPass**
(5)		fail	violate	**FailViolate**
(6)	pass		violate	**PassViolate**
(7)	pass	fail	violate	**FailPassViolate**
(8)				

TAB. 4.3 – Verdicts possibles

sur une implémentation par rapport à la spécification, à l'implémentation et aux propriétés.

Proposition 4.2 *Considérons une trace $\sigma \in Traces(I^\delta \times test(\mathcal{S}))$ donnant un verdict V pour une implémentation I. Pour chaque valeur possible de V, la signification du verdict est la suivante :*

(1) **Pass** *: l'implémentation et la spécification satisfont la propriété d'accessibilité (et aucune non-conformité n'a été détectée).*

(2) **Violate** *: l'implémentation et la spécification violent la propriété de sûreté (et aucune non-conformité n'a été détectée).*

(3) **Fail** *: l'implémentation n'est pas conforme à la spécification (et aucune satisfaction/violation de propriété n'a été détectée).*

(4) **FailPass** *: l'implémentation n'est pas conforme à la spécification et satisfait la propriété d'accessibilité.*

(5) **FailViolate** *: l'implémentation ne satisfait pas la propriété et n'est pas conforme à la spécification.*

(6) **PassViolate** *: l'implémentation et la spécification violent la propriété de sûreté et satisfont la propriété d'accessibilité.*

(7) **FailPassViolate** *: l'implémentation n'est pas conforme à la spécification, elle satisfait la propriété d'accessibilité et viole la propriété de sûreté.*

Pour construire un cas de test à partir de $test(\mathcal{S})$, toutes les entrées doivent être transformées en sorties et inversement (opération miroir). Lors de l'exécution du cas de test sur l'implémentation, les actions de l'implémentation et celles du cas de test doivent être complémentaires.

Exemple 4.5 *Le graphe de test $Can(\mathcal{S}) \times \omega_2 \times \omega_1$ des spécifications et propriétés vues figures 4.4(a), 4.4(b) et 4.5(b) est représenté figure 4.6. Pour plus de clarté, nous avons supprimé les transitions partant des localités* Pass *ou* Violate*, puisque, dans ces localités, la propriété est déjà violée/satisfaite par la spécification.*

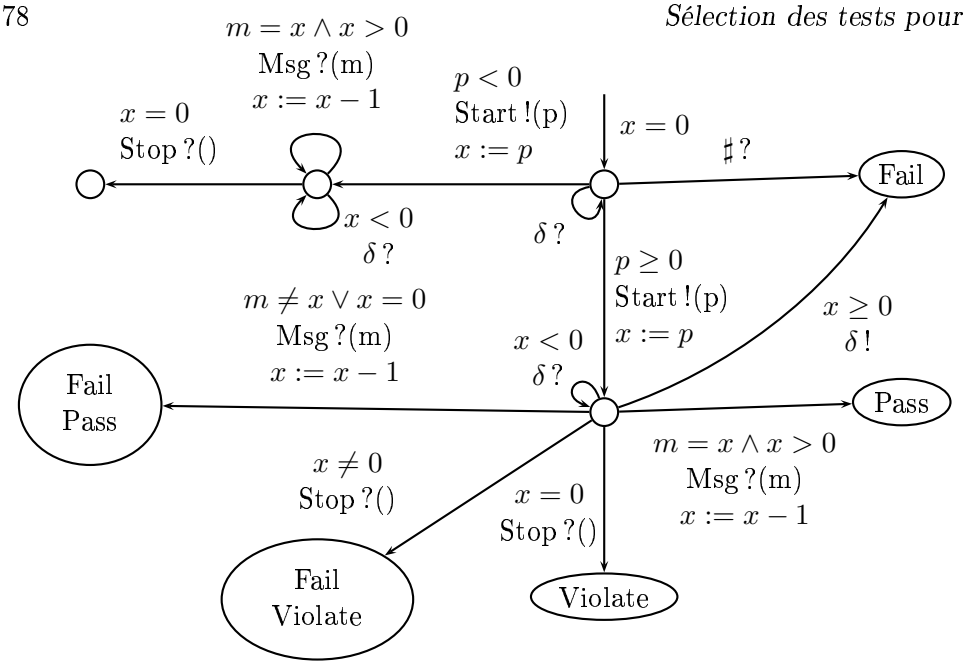


FIG. 4.6 – Graphe de test de $test(\mathcal{S}) = Can(\mathcal{S}) \times \omega_1 \times \omega_2$ après miroir.

Pour finir, le résultat est automatiquement analysé afin d'éliminer les transitions ne pouvant mener à la violation/satisfaction de la propriété.

4.2.2 Sélection des cas de test

Le but principal du processus de test décrit dans cette partie est :

1. de s'assurer de la conformité de l'implémentation par rapport à la spécification et
2. de détecter sur l'implémentation des violations/satisfactions des propriétés de sûreté/d'accessibilité requises par le système.

Générer des cas de test à partir du testeur canonique permet la détection de la non-conformité. Utiliser des observateurs permet :

- pour un observateur positif, de tester un comportement précis du système (principe d'un objectif de test),
- pour un observateur négatif, de pousser l'implémentation à la faute (on cherche une violation de la propriété).

Le graphe de test $test(\mathcal{S}) = Can(\mathcal{S}) \times \omega^+ \times \omega^-$ que nous avons décrit dans la sous-section précédente est un cas de test permettant tout cela. Cependant, celui-ci est de taille trop importante et ne permet aucun guidage du test de façon à (in)valider l'une ou l'autre des propriétés puisque les observateurs sont complets. Il apparaît donc nécessaire de focaliser le cas de test sur un sous-ensemble des propriétés, ce qui amène à sélectionner une partie de celui-ci en essayant de supprimer les comportements qui ne peuvent mener à aucun verdict vis-à-vis de ces propriétés.

Dans la suite de cette section, nous présentons une technique similaire à celle utilisée sur les ioLTS à la sous-section 2.2.3. Cette technique permet de détecter et d'éliminer des états et des transitions d'un graphe de test $test(\mathcal{S}, \omega)$[3] à partir desquels un ensemble Q_{Target} n'est plus atteignable. Le graphe de test est généré à partir d'une spécification $\mathcal{S}$ et d'un ensemble de propriétés (positives/négatives) ω. L'ensemble Q_{Target} correspond aux états produisant un verdict Target. Nous allons chercher à sélectionner sur le cas de test des comportements de telle manière que l'implémentation reste dans des états à partir desquels des verdicts "intéressants/importants" peuvent être émis. Le choix de ces verdicts est bien entendu subjectif et va fortement dépendre de l'application considérée. Ainsi, si on choisit de s'intéresser particulièrement aux propriétés d'accessibilité, on aura :

$$Q_{\mathsf{Target}} = \mathsf{Pass} \cup \mathsf{FailPass} \cup \mathsf{PassViolate} \cup \mathsf{FailPassViolate} \tag{4.2}$$

De manière générale, Q_{Target} sera une combinaison des sept ensembles de valeurs des variables (définies dans le tableau 4.3), où Target a pour valeur Pass ou Violate.

Le processus de sélection de test consisterait, idéalement, à sélectionner exactement les traces menant à Target : $Traces(test(\mathcal{S}, \omega), \mathsf{Target})$, plus les sorties non spécifiées prolongeant les préfixes de ces traces jusqu'à Fail, dénotant ainsi la non-conformité. Cependant, les implémentations ne sont pas contrôlables : leur comportement en sortie n'est pas complètement déterminé par les entrées. Le testeur doit donc considérer toutes les sorties possibles après une trace : les sorties à partir desquelles Target ou Fail sont atteignables mais aussi celles à partir desquelles Target n'est plus atteignable. Pour une plus grande efficacité, ce dernier cas doit être détecté le plus tôt possible et un verdict Inconc est alors émis.

Le problème de sélection est donc réduit à calculer, à partir d'un graphe de test, le sous-ensemble des états à partir desquels Target est atteignable. Ce sous-ensemble est noté $coreach(\mathsf{Target})$. Il peut être décrit par un plus petit point fixe :

$$coreach(\mathsf{Target}) = lfp(\lambda X.\mathsf{Target} \cup pre(X))$$

avec $pre(X) = \{q | \exists q' \in X, \exists \lambda \in \Lambda, q \xrightarrow{\lambda} q'\}$, l'ensemble des états à partir desquels X peut être atteint en une transition. Le calcul de l'ensemble $coreach(\mathsf{Target})$ est simple pour des systèmes dont l'ensemble d'états est fini et peut être résolu par des algorithmes de graphes. C'est par exemple le cas dans l'outil TGV [Jér02] lors de la sélection de test pour des ioLTS à états finis. Malheureusement, $coreach(\mathsf{Target})$ n'est pas calculable pour le modèle des ioSTS puisque ceux-ci ont pour sémantique des ioLTS dont l'ensemble d'états est infini. Toutefois, il existe des techniques qui permettent de calculer une sur-approximation de celui-ci, comme celles que nous allons utiliser, basées sur l'interprétation abstraite.

[3] Nous supposons que l'opération miroir a été effectuée.

4.2.2.1 Sélection de tests par analyse approchée

Face à ce problème de non-calculabilité, la solution proposée consiste donc à utiliser une analyse de co-accessibilité qui calcule une sur-approximation par interprétation abstraite [CC77] des états co-accessibles [Jea03]. Grâce à cette analyse approchée, le graphe de test modélisé par l'ioSTS $\mathit{test}(\mathcal{S},\omega)$ est transformé en un ioSTS représentant un cas de test $\mathcal{TC}$. Pour cela, l'analyse contraint les sorties et détecte les entrées inconclusives (les entrées ne pouvant pas conduire à Target) par transformations syntaxiques des gardes des transitions (voir définition 4.5).

Supposons qu'une sur-approximation $\mathit{coreach}^\alpha \supseteq \mathit{coreach}(\mathsf{Target})$ de l'ensemble exact des états co-accessibles depuis Target a été calculée, et que cette sur-approximation est représentée par une formule logique. Prenons un ensemble d'états $X \in \mathcal{D}_v$, représenté par une formule $X(v)$ et notons $\mathit{pre}(A)(X)(v,p)$ la pré-condition pour satisfaire X par l'affectation $A : \mathcal{D}_v \times \mathcal{D}_p \to \mathcal{D}_v$ telle que :

$$\mathit{pre}(A)(X)(v,p) \;=\; (\exists v' : X(v') \wedge (v' = A(v,p))) \;=\; X(A(v,p)).$$

En d'autres termes, $\mathit{pre}(A)(X)(v,p)$ représente l'ensemble des valeurs des variables v et des paramètres p à partir desquelles X est atteignable après l'affectation A. Notons que $\mathit{pre}(A)$ est monotone et que $\mathit{pre}^\alpha(A)(X)$ est une sur-approximation de $\mathit{pre}(A)(X)$. Dans ce contexte, $\mathit{pre}^\alpha(A)(\mathit{coreach}^\alpha)$ est une sur-approximation de l'ensemble des valeurs des variables et paramètres qui permettent de rester dans l'ensemble $\mathit{coreach}(\mathsf{Target})$ lorsqu'une transition est prise. En d'autres termes, $\mathit{pre}^\alpha(A)(\mathit{coreach}^\alpha)$ est une condition nécessaire pour rester dans $\mathit{coreach}(\mathsf{Target})$ et sa négation est une condition suffisante pour en sortir.

En utilisant ces analyses approchées, un cas de test peut être construit à partir de $\mathit{test}(\mathcal{S},\omega)$ comme décrit définition 4.5. L'opération de sélection consiste à éliminer les transitions dont le tirage mène dans des états à partir desquels l'ensemble des localités Target n'est plus accessible. Si une telle transition est étiquetée par une sortie, alors celle-ci peut être éliminée du cas de test. Un cas de test contrôlant ses sorties, il peut décider de ne pas réaliser cette action dans la mesure où l'accessibilité de Target n'est plus possible. Par contre, il n'est pas possible d'empêcher les entrées de se produire (celles-ci sont émises pas l'implémentation), ainsi les transitions étiquetées par des entrées à partir desquelles Target ne peut plus être atteint sont réorientées vers une localité appelée inconc.

Définition 4.5 *Un cas de test pour une spécification $\mathcal{S}$ et un observateur ω est un ioSTS déterministe $\mathcal{TC} = \langle V^{test}, \Theta^{test}, \Sigma^{\mathcal{TC}}, \mathcal{T}^{\mathcal{TC}}\rangle$ avec $\Sigma^{\mathcal{TC}}_! = \Sigma^{\mathcal{S}}_?$, et $\Sigma^{\mathcal{TC}}_? = \Sigma^{\mathcal{S}}_! \cup \{\delta\}$ (opération miroir). L'ensemble des transitions $\mathcal{T}^{\mathcal{TC}}$ est défini par :*

1. *une règle pour la sélection,*

$$\frac{(a,p,G,A) \in \mathcal{T}^{test} \quad a \in \Sigma^{\mathcal{TC}}_! \quad G' = \mathit{pre}^\alpha(A)(\mathit{coreach}^\alpha)}{(a,p,G \wedge G',A) \in \mathcal{T}^{\mathcal{TC}}}$$

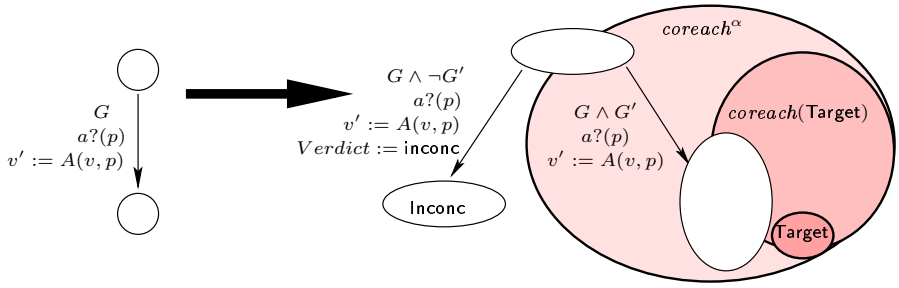


FIG. 4.7 – Illustration de la règle *split*

2. *une règle pour la non-conformité,*

$$\frac{(a,p,G,A) \in \mathcal{T}^{test} \quad a \in \Sigma_?^{\mathcal{TC}} \quad A_{\mathsf{Verdict}} = \mathsf{verdict} := \mathsf{fail}}{(a,p,G,A) \in \mathcal{T}^{\mathcal{TC}}}$$

3. *une règle split pour la détection des inconclusifs,*

$$\frac{(a,p,G,A) \in \mathcal{T}^{test} \quad a \in \Sigma_?^{\mathcal{TC}} \quad A_{\mathsf{Verdict}} \neq \mathsf{verdict} := \mathsf{fail} \quad G' = pre^{\alpha}(A)(coreach^{\alpha})}{(a,p,G \wedge G',A),(a,p,G \wedge \neg G',A') \in \mathcal{T}^{\mathcal{TC}}}$$

avec A' *défini par* $\left\{ \begin{array}{l} A'_{\mathsf{Verdict}} = \mathsf{verdict} := \mathsf{inconc}, \\ A'_v = A_v \ \textit{pour} \ v \neq \mathsf{Verdict} \end{array} \right\}$.

La règle de sélection contraint les gardes de toutes les transitions de sortie afin que leurs post-conditions puissent mener à $coreach^{\alpha}$ qui est une sur-approximation de l'ensemble des états menant à Target. En effet, le testeur contrôle ses transitions de sortie et peut donc les restreindre à $G' = pre^{\alpha}(A)(coreach^{\alpha})$ afin de pouvoir rester dans la sur-approximation des états co-accessibles $coreach^{\alpha}$ après que la transition ait été tirée. pre^{α} étant une sur-approximation, le but de cette règle est de supprimer les transitions qui mènent sûrement en dehors de $coreach^{\alpha}$. Les transitions restantes peuvent néanmoins ne pas mener à $coreach^{\alpha}$.

La règle pour la non-conformité permet de garder les transitions d'entrée menant au verdict Fail.

La règle *split*, illustrée figure 4.7, sépare en deux les transitions d'entrée ne menant pas à Fail grâce à la conjonction des gardes avec G' ou $\neg G'$: pour les valeurs des variables et paramètres qui ne vont certainement pas permettre d'atteindre $coreach^{\alpha}$ (quand $\neg G'$ est vraie), le verdict inconc est émis tandis que rien ne se passe (on garde la transition) si les valeurs des variables et paramètres permettent éventuellement d'atteindre $coreach^{\alpha}$ (quand G' est vraie). En effet, les transitions d'entrée ne peuvent être contrôlées, mais les situations à partir desquelles le verdict Target (Pass ou Violate) ne peut plus être atteint peuvent encore être détectées et marquées par le verdict Inconc.

Précisons que la sémantique ioLTS du cas de test $\mathcal{TC}$ est différente de celle de $test(\mathcal{S},\omega)$, en particulier pour les transitions de sortie qui ont été supprimées par la règle de sélection.

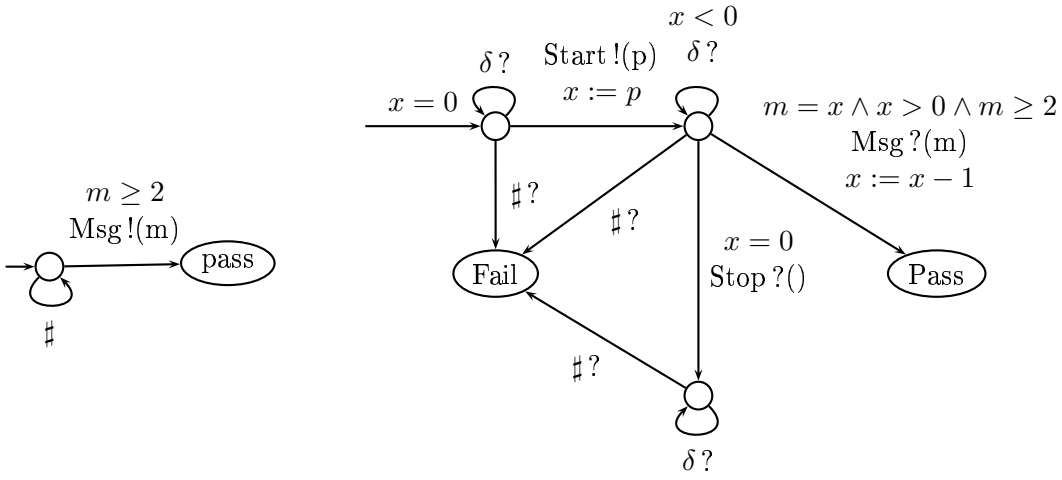
4.2.2.2 Simplification des tests

Il est encore possible de simplifier le cas de test, et cela, sans modifier sa sémantique, grâce à une sur-approximation $reach^{\alpha}(\Theta^{\mathcal{TC}})$ de ses états accessibles depuis sa condition initiale $reach(\Theta^{\mathcal{TC}})$. $reach^{\alpha}(\Theta^{\mathcal{TC}})$ est décrit comme suit :

$$reach(\Theta^{\mathcal{TC}}) = lfp(\lambda X.\Theta^{\mathcal{TC}} \cup post(X))$$

avec $post(X) = \{q' | \exists q \in X, \exists \lambda \in \Lambda, q \xrightarrow{\lambda} q'\}$, l'ensemble des états accessibles depuis X en une transition. Cette simplification consiste à enlever les transitions à partir desquelles les gardes ne sont pas satisfaites par la sur-approximation $reach^{\alpha}(\Theta^{\mathcal{TC}})$ de l'ensemble des états accessibles, c'est-à-dire les transitions (a, p, G, A) pour lesquelles $G \wedge reach^{\alpha}(\Theta^{\mathcal{TC}})$ vaut faux. Cette analyse d'accessibilité permet donc de supprimer des transitions dont les gardes ne peuvent être satisfaites. Elle permet également de simplifier les formules des gardes, ce qui rend la résolution des contraintes et donc l'exécution du cas de test sur l'implémentation plus rapide (voir sous-section 4.2.2.4).

Exemple 4.6 *Reprenons notre exemple de spécification dont le testeur canonique était représenté figure 3.6(b). Vérifions si l'implémentation permet d'émettre un message ayant pour paramètre un entier supérieur ou égal à 2. L'observateur ω représentant cette propriété est dessiné figure 4.8(a). Le graphe de test $test(\mathcal{S}, \omega)$ est représenté figure 4.8(b). Pour plus de clarté, les transitions partant de la localité* Pass *n'ont pas été représentées. Cela n'a aucune incidence puisqu'en pratique, l'exécution d'un cas de test sur une implémentation est arrêtée dès que la propriété est satisfaite.*



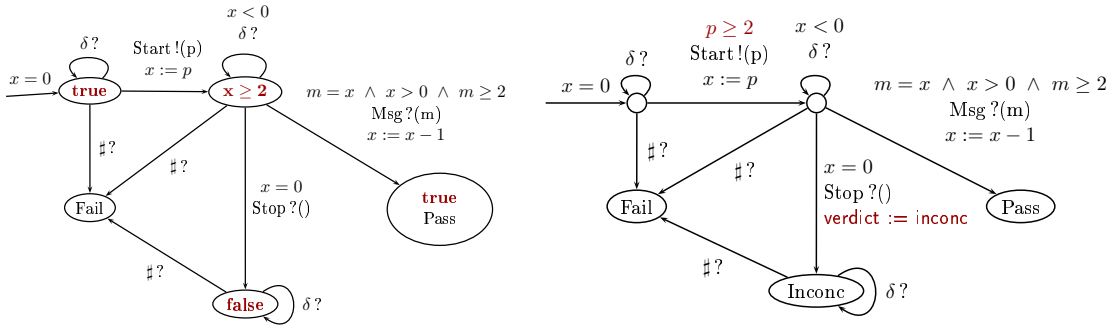
(a) Propriété ω

(b) Graphe de test $test(\mathcal{S}, \omega)$

FIG. 4.8 – Graphe de test obtenu à partir de ω et $\mathcal{S}$.

Nous nous intéressons ici aux états co-accessibles depuis $Q_{\mathsf{Target}} =$ Pass. Le calcul de $coreach^{\alpha}$(Pass) sur cet exemple est représenté figure 4.9(a) (les contraintes calculées sont

inscrites dans les localités en rouge et gras). Cette analyse a pour effet de contraindre la garde sur la première sortie avec $p \geq 2$ *car si* p *était inférieur à 2,* Pass *ne serait pas atteignable. La transition portant l'entrée* Stop *et la garde* $x = 0$ *nous amène dans un état à partir duquel* Pass *n'est plus atteignable. Le verdict* Inconc *est alors émis si la transition est tirée. Le cas de test résultant de cette analyse de co-accessibilité depuis* Pass *est représenté figure 4.9(b). L'analyse d'accessibilité va, quant à elle, supprimer la transition menant à* Inconc *(le calcul est en bleu et gras sur la figure 4.10(a)). En effet, nous avons la contrainte* $x \geq 2$ *sur la deuxième localité, ce qui fait qu'on ne peut tirer la transition portant l'entrée* Stop *vers* Inconc. *Ceci illustre bien le fait que contraindre les gardes des sorties en utilisant une sur-approximation de l'analyse de co-accessibilité peut permettre de supprimer les chemins ne menant pas à* Pass *(grâce à l'analyse de co-accessibilité, l'analyse d'accessibilité peut supprimer la transition menant à* Inconc*). Dans le cas de notre exemple où l'analyse est exacte, le cas de test résultant (voir figure 4.10(b)) est optimal par rapport à sa capacité à forcer l'atteignabilité de* Pass. *Néanmoins, des actions non contrôlables menant à* Inconc *peuvent persister.*



(a) Calcul de $coreach^{\alpha}$(Pass)

(b) Cas de test obtenu grâce au calcul de $coreach^{\alpha}$(Pass)

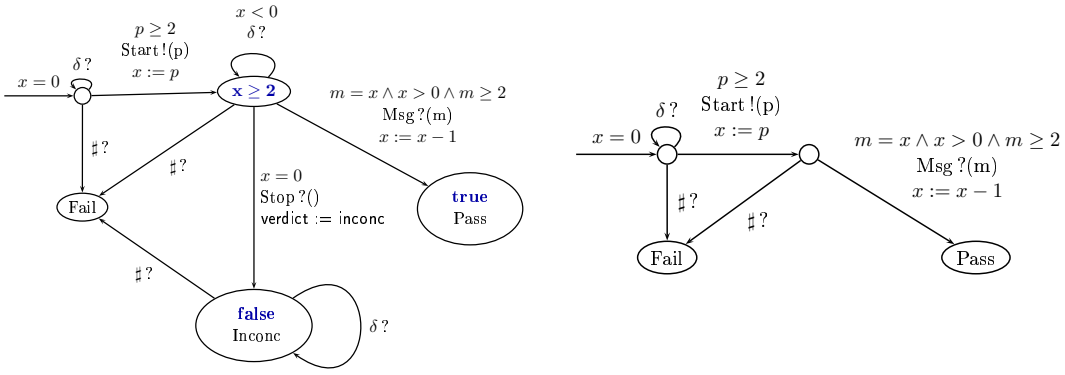
FIG. 4.9 – Cas de test obtenu après analyse de co-accessibilité.

En opérant une sélection sur le cas de test de cette manière, nous conservons toutes les traces du cas de test originel menant à Target, tout en préservant la correction des verdicts.

4.2.2.3 Propriétés des cas de test

Nous savons que le testeur canonique $Can(\mathcal{S})$ est non-biaisé (voir sous-section 3.3.3). Il est facile de constater que cette propriété est préservée lors du produit synchrone $test(\mathcal{S}, \omega) = Can(\mathcal{S}) \times \omega$ et lors de la sélection du cas de test $\mathcal{TC}$ puisque ces transformations ne peuvent pas ajouter de cas de rejet de l'implémentation. Tous les cas de test sont donc non-biaisés. Nous savons que

$$Traces(test(\mathcal{S}, \omega), \mathsf{Fail}) = Traces(test(\mathcal{S}, \omega)) \cap Traces(Can(\mathcal{S}), \mathsf{Fail}),$$



(a) Calcul de $reach^{\alpha}(\Theta^{test})$

(b) Cas de test obtenu grâce au calcul de $reach^{\alpha}(\Theta^{test})$

FIG. 4.10 – Cas de test obtenu après simplification.

on en déduit que

$$Traces(\mathcal{TC}, \mathsf{Fail}) = Traces(\mathcal{TC}) \cap Traces(Can(\mathcal{S}), \mathsf{Fail}).$$

Cette propriété indique que seules les implémentations non-conformes peuvent être rejetées (et que ce rejet arrive dès que possible).

En ce qui concerne l'exhaustivité à la limite, elle vient de la construction suivante. Par définition de la relation de conformité ioco, pour toute implémentation I non conforme, il existe une trace $\sigma.a$ appartenant à l'ensemble $STraces(\mathcal{S}) \cdot (\Lambda^{!}_{\mathcal{S}^{\delta}} \cup \{\delta\}) \setminus STraces(\mathcal{S})$. Le préfixe σ est donc une trace de $\mathcal{S}^{\delta}$, tandis que $\sigma.a$ appartient à $Traces(Can(\mathcal{S}), \mathsf{Fail})$. Comme, par définition, il n'y a pas de blocage de sortie (voir sous-section 3.2.3) dans $\mathcal{S}^{\delta}$, il existe une sortie b telle que $\sigma.b \in STraces(\mathcal{S})$. Cette trace appartient donc aux traces du testeur canonique $Traces(Can(\mathcal{S}))$ mais pas aux traces menant à la non-conformité $Traces(Can(\mathcal{S}), \mathsf{Fail})$. Prenons maintenant un observateur (négatif ou positif) de manière à ce que la trace $\sigma.b$ mène à Violate ou à Pass. Nous pouvons alors construire un cas de test $\mathcal{TC}$ à partir de $\mathcal{S}$ et ω. La trace $\sigma.b$ appartient alors à $Traces(\mathcal{TC})$ et $\sigma.a$ appartient plus particulièrement à $Traces(\mathcal{TC}, \mathsf{Fail})$. On obtient alors $\sigma.a \in STraces(I) \cap Traces(\mathcal{TC}, \mathsf{Fail})$, ce qui signifie que le cas de test $\mathcal{TC}$ peut rejeter l'implémentation. La proposition 3.1 est donc conservée après sélection.

4.2.2.4 Exécution

Considérons maintenant l'exécution du cas de test sur l'implémentation. Cette exécution est modélisée par une composition. Les silences de l'implémentation étant observés durant le test, nous composerons le cas de test avec la suspension de l'implémentation I^{δ} (les silences sont alors explicites). De manière formelle, l'exécution du cas de test $\mathcal{TC}$ sur une implémentation I est modélisée par la composition parallèle du cas de test

$TC = [\![\mathcal{TC}]\!]$ avec I^δ, synchronisés sur les actions communes. La composition parallèle entre un ioSTS et un ioLTS n'est pas définie, nous composons donc l'implémentation avec la sémantique du cas de test (pour plus de détails sur cette composition, se reporter à la sous-section 1.3.2). De plus, la sémantique est suffisante puisque nous ne travaillons que sur les traces.

Il est clair que $\mathit{Traces}(I^\delta||TC) = STraces(I) \cap \mathit{Traces}(TC) = STraces(I) \cap \mathit{Traces}(\mathcal{TC})$. Nous avons donc : $\mathit{Traces}(I^\delta||TC, Q_I \times \mathsf{Fail}) = STraces(I) \cap \mathit{Traces}(\mathrm{TC}, \mathsf{Fail})$. Un cas de test rejette une implémentation lorsque la variable verdict atteint la valeur Fail. Le cas de test peut donc rejeter l'implémentation si $I^\delta||TC$ peut mener à Fail dans TC : TC *mayfail* $I = (\mathit{Traces}(I^\delta||TC, Q_I \times \mathsf{Fail}) \neq \emptyset)$, ce qui est équivalent à $STraces(I) \cap \mathit{Traces}(\mathrm{TC}, \mathsf{Fail}) \neq \emptyset$.

D'un point de vue technique, les cas de test produits sont des ioSTS pour lesquels les valeurs des paramètres de communication ne sont pas encore instanciées.

Lorsque les actions sont des sorties, les valeurs de ces paramètres sont choisies, pendant l'exécution, parmi les valeurs satisfaisant les gardes (dans notre exemple figure 4.10(b), nous pouvons choisir $p = 3$ puisque p doit être supérieur ou égal à 2). Les valeurs des paramètres sont calculées par un solveur de contraintes durant l'exécution. L'utilisation d'une théorie décidable comme celle de l'arithmétique de Presburger est donc nécessaire pour exprimer les gardes afin que les techniques de résolution de contraintes puissent calculer ces valeurs.

Lorsque le cas de test reçoit des entrées de la part de l'implémentation (ou qu'un silence est observé), le cas de test étant complet en entrée et déterministe, nous devons vérifier quelle transition est tirable suivant la garde et la valeur du paramètre de communication reçue (dans notre exemple, nous allons dans l'état Pass si $m = 3$ ou n'importe quelle autre valeur supérieure à 2 et dans Fail si $m = 1$).

Chapitre 5

Expérimentations sur STG

Nous allons expérimenter dans ce chapitre la méthodologie de cette partie grâce à l'outil développé au sein de l'équipe VerTeCS, STG (Symbolic Test Generation) [CJRZ02, PJJ07]. STG est un outil pour la génération (sélection incluse) et l'exécution de cas de test symboliques afin de tester la conformité de systèmes réactifs par rapport à leur spécification formelle. Cet outil utilise le modèle des ioSTS et la relation de conformité ioco. Les phases de sélection et d'exécution se déroulent comme expliqué au chapitre 4, la sélection est basée sur une analyse approchée et l'exécution sur un solveur de contraintes. La sélection se fait par rapport à un objectif de test modélisé par un ioSTS. Cet objectif peut être interprété comme un observateur positif. Un objectif peut également représenter un observateur négatif mais il est nécessaire d'interpréter différemment les verdicts.

STG prend en entrée un fichier comprenant les canaux utilisés (les entrées/sorties) ainsi que leur type (le type des paramètres qu'ils portent) suivis des processus (généralement un seul représentant la spécification) et des objectifs de test sous forme d'ioSTS (avec localités). Cet outil peut manipuler des ioSTS avec variables booléennes et numériques et accepte des expressions bien typées composées d'expressions booléennes et linéaires. L'objectif de test peut contenir deux localités particulières :

- *Accept*, reconnue par STG comme étant la localité à atteindre pour satisfaire l'objectif;
- *Reject*, reconnue par STG comme étant la localité qu'on ne veut pas atteindre. Toutes les traces y menant seront supprimées lors de la sélection sur le graphe de test.

Nous pouvons ensuite préciser sur la ligne de commande quelles opérations doivent être effectuées pour sélectionner un cas de test. Les opérations disponibles sont le produit synchrone, la clôture (l'élimination des actions internes), le miroir, et la complétion en sortie (ou en entrée). Par défaut, l'ensemble de ces opérations est effectué afin de produire le graphe de test : clôture, complétion en sortie et miroir produisent le testeur canonique puis le produit entre ce testeur et l'objectif de test forme le graphe de test.

La sélection se fait ensuite sur ce graphe de test. Afin d'effectuer les analyses de la sélection, STG utilise l'outil NBac [Jea03]. Celui-ci effectue des analyses approchées d'accessibilité et de co-accessibilité sur les ioSTS grâce à des techniques d'interprétation abstraite. L'analyse d'accessibilité permet à STG de simplifier le graphe de test par élimination des localités non accessibles et simplification des gardes et des affectations. L'analyse de co-accessibilité (depuis les états accepteurs) sert quant à elle à la sélection en tant que telle puisqu'elle permet de renforcer les gardes des sorties afin de guider l'exécution du test vers la satisfaction de l'objectif. Elle permet également de détecter les entrées ne pouvant plus y mener et pour lesquelles un verdict **Inconc** devrait être émis.

Le cas de test obtenu après sélection est ensuite traduit en un programme java pour être exécuté sur l'implémentation. L'exécution est basée sur Synchronous Java [Pet02], un langage avec processus et communications sur rendez-vous ressemblant au Java. Le cas de test est un programme réactif qui peut contenir des choix entre plusieurs sorties et entre les valeurs possibles de leurs paramètres. Lorsqu'un choix doit être fait, l'exécution fait appel à un solveur de contraintes, Lucky [JR07], afin de calculer les valeurs de ces paramètres de sorties.

Nous allons présenter ici des expérimentations sur quelques exemples. Nous commencerons par un exemple simple (un jeu de Nim) puis nous illustrerons les problèmes de sur-approximation sur l'exemple de l'ascenseur vu en introduction de cette partie. Nous commenterons enfin l'étude de cas QuiDonc de France Telecom.

5.1 Un jeu de Nim

Un jeu de Nim est un jeu qui se jour à deux, à tour de rôle. Le principe est d'ôter un ou plusieurs objets (billes, bâtonnets, allumettes, etc.) d'un tas jusqu'à ce qu'il n'y en ait plus. Il existe différents jeux de Nim partant de ce principe. Nous allons ici prendre un jeu très simple : l'utilisateur décidera du nombre d'allumettes puis pourra ôter une à trois allumettes. Le système pourra alors à son tour en ôter une à trois et ainsi de suite. Le perdant sera celui qui devra ôter la dernière allumette.

La spécification de ce jeu est donnée à la figure 5.1 sous la forme d'un programme fourni en entrée à STG puis à la figure 5.3 sous la forme graphique générée par celui-ci. Tout d'abord, le nombre d'allumettes est initialisé à zéro ($nball := 0$) avant de prendre la valeur du paramètre porté par l'entrée $Nb\_allumettes$. Une entrée $Jouer$ portant le nombre d'allumettes p à ôter compris entre 1 et 3 est ensuite reçue. Le nombre d'allumettes est alors modifié en conséquence ($nball := nball - p$). Si le nombre d'allumettes est strictement supérieur à 1, le système émet alors une sortie $Enlever$ portant le nombre d'allumettes p à ôter compris entre 1 et 3. S'il ne reste qu'une

allumette, alors l'utilisateur a gagné et la sortie *Gagne* est émise, sinon il a perdu et la sortie *Perdu* est émise.

Prenons une propriété de sûreté à vérifier sur ce système, par exemple, le fait qu'on ne veuille pas que l'utilisateur puisse entrer un nombre d'allumettes lui permettant de gagner sans que le système ne puisse jouer. STG ne produisant des cas de test qu'en fonction d'objectifs de test et non pas de propriétés, il nous faut adapter cette propriété à un objectif de test. Afin de simuler un observateur négatif, représentation de la négation d'une propriété de sûreté (voir la sous-section 4.1.1), nous allons remplacer la localité *Violate* par *Accept* (figures 5.2 et 5.4). Nous allons également utiliser la localité *Reject*. Cette localité est ici utile car nous ne voulons pas que d'autres entrées *Jouer* soient possibles. Or, la complétion automatique sur l'alphabet de l'objectif (voir figure 5.5) aurait permis d'autres entrées *Jouer* après la première, et par conséquent, par produit avec la spécification, des sorties *Enlever*. Le but de l'objectif étant de vérifier que l'utilisateur peut gagner sans que le système ne puisse enlever d'allumettes, l'objectif n'aurait pas été correct.

Le testeur canonique et le produit entre celui-ci et l'objectif sont ensuite générés automatiquement. Le produit peut d'ailleurs être généré sous forme de graphe comme illustré à la figure 5.6. Les transitions menant à la localité *Fail* n'y sont pas représentées pour cause de lisibilité.

La sélection est ensuite effectuée sur ce produit. Une première analyse d'accessibilité est effectuée, suivie d'une analyse de co-accessibilité (dont on peut obtenir le graphe résultant) et d'une deuxième analyse d'accessibilité. Ces analyses génèrent alors le cas de test représenté à la figure 5.7. Les transitions menant à *Fail* ne sont pas représentées mais peuvent être obtenues par complétion des gardes. Nous pouvons constater que les analyses ont calculé que si l'utilisateur voulait gagner sans que le système ne puisse jouer, alors il ne devait entrer qu'entre 2 et 4 allumettes (le paramètre de l'action *Nb_allumettes* est donc compris entre 2 et 4). Nous pouvons en déduire que pour que la propriété de sûreté soit satisfaite, le nombre d'allumettes que peut entrer l'utilisateur ne doit pas être inférieur à 5. Nous pouvons donc ajouter cette contrainte sur la transition de la spécification portant l'entrée *Nb_allumettes* et continuer les tests suivant d'autres propriétés tant de sûreté que d'accessibilité (par exemple, vérifier que l'utilisateur peut gagner, perdre, etc.).

5.2 L'ascenseur

Prenons l'exemple de l'ascenseur expliqué dans l'introduction de cette partie. Après une entrée *Go* portant le numéro d'étage demandé, l'ascenseur monte (ou descend) jusqu'à atteindre cet étage. La spécification était présentée à la figure 2.8. Pour une lecture plus facile, elle est à nouveau représentée à la figure 5.8.

```
system nim;

gate
  Nb_allumettes(int);
  Jouer(int);
  Enlever(int);
  Perdu();
  Gagne();

process nim;
input
  Nb_allumettes, Jouer;

output
  Enlever, Perdu, Gagne;

variables
  nball     : int;

state
  init : Start;
  WaitNbAll;
  Joueur1;
  Joueur2;
  Perdre;
  Gagner;

transition
  from Start
    do {
        nball:=0
       }
  to WaitNbAll;

  from WaitNbAll
    sync Nb_allumettes?(p)
    do {
        nball := p
       }
  to Joueur1;

  from Joueur1
   if (p >= 1 and p <= 3 and p <= nball)
    sync Jouer?(p)
    do {
        nball := nball - p
       }
  to Joueur2;

  from Joueur2
   if (p >= 1 and p <= 3 and p < nball and nball > 1)
    sync Enlever!(p)
    do {
        nball := nball - p
       }
  to Joueur1;

  from Joueur2
   if (nball = 1)
    sync Gagne!()
  to Gagner;

  from Joueur2
   if (nball = 0)
    sync Perdu!()
  to Perdre;
```

FIG. 5.1 – Spécification du jeu de Nim (version programme).

```
process TPS;

input
  Nb_allumettes, Jouer;

output
  Gagne;

variables
  all : int;

state
  init : Start;
  All;
  Jeu1;
  Jeu2;
  Accept;
  Reject;

transition

  from Start
    do {
        all := 0
       }
  to All;

  from All
    sync Nb_allumettes?(p)
    do {
        all := p
       }
  to Jeu1;

  from Jeu1
    sync Jouer?(p)
  to Jeu2;

  from Jeu2
    sync Jouer?(p)
  to Reject;

  from Jeu2
    sync Gagne!()
  to Accept;
```

FIG. 5.2 – Objectif de test représentant la négation de la propriété de sûreté (version programme).

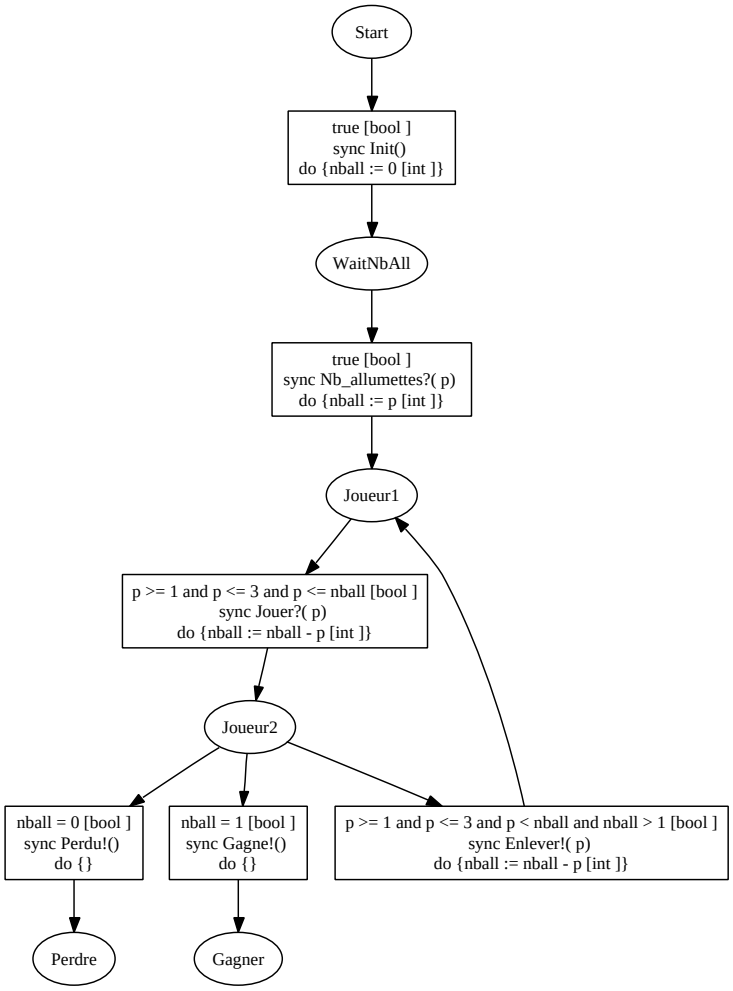


FIG. 5.3 – Spécification du jeu de Nim (version graphe).

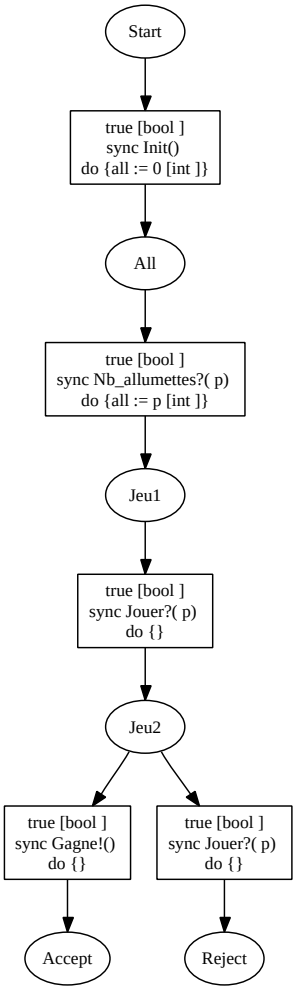


FIG. 5.4 – Objectif de test représentant la négation de la propriété de sûreté (version graphe).

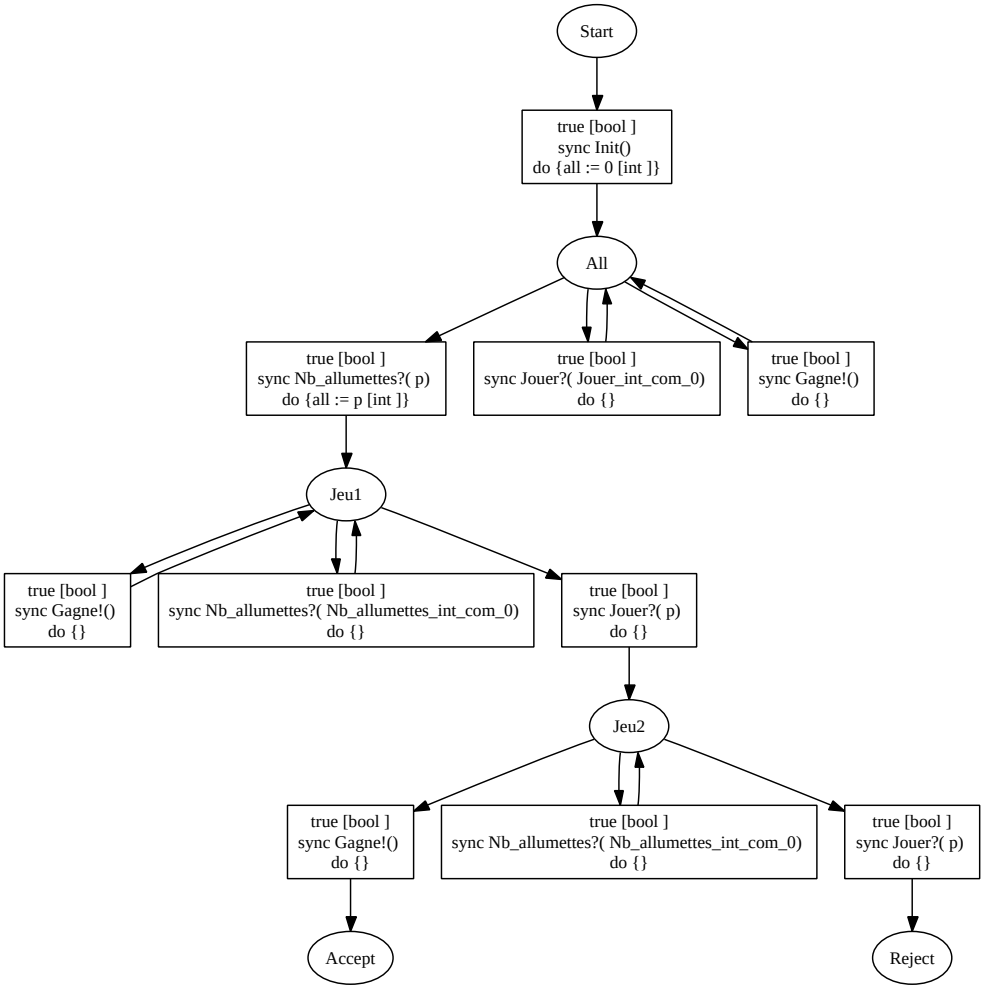


FIG. 5.5 – Objectif de test représentant la négation de la propriété de sûreté complété automatiquement.

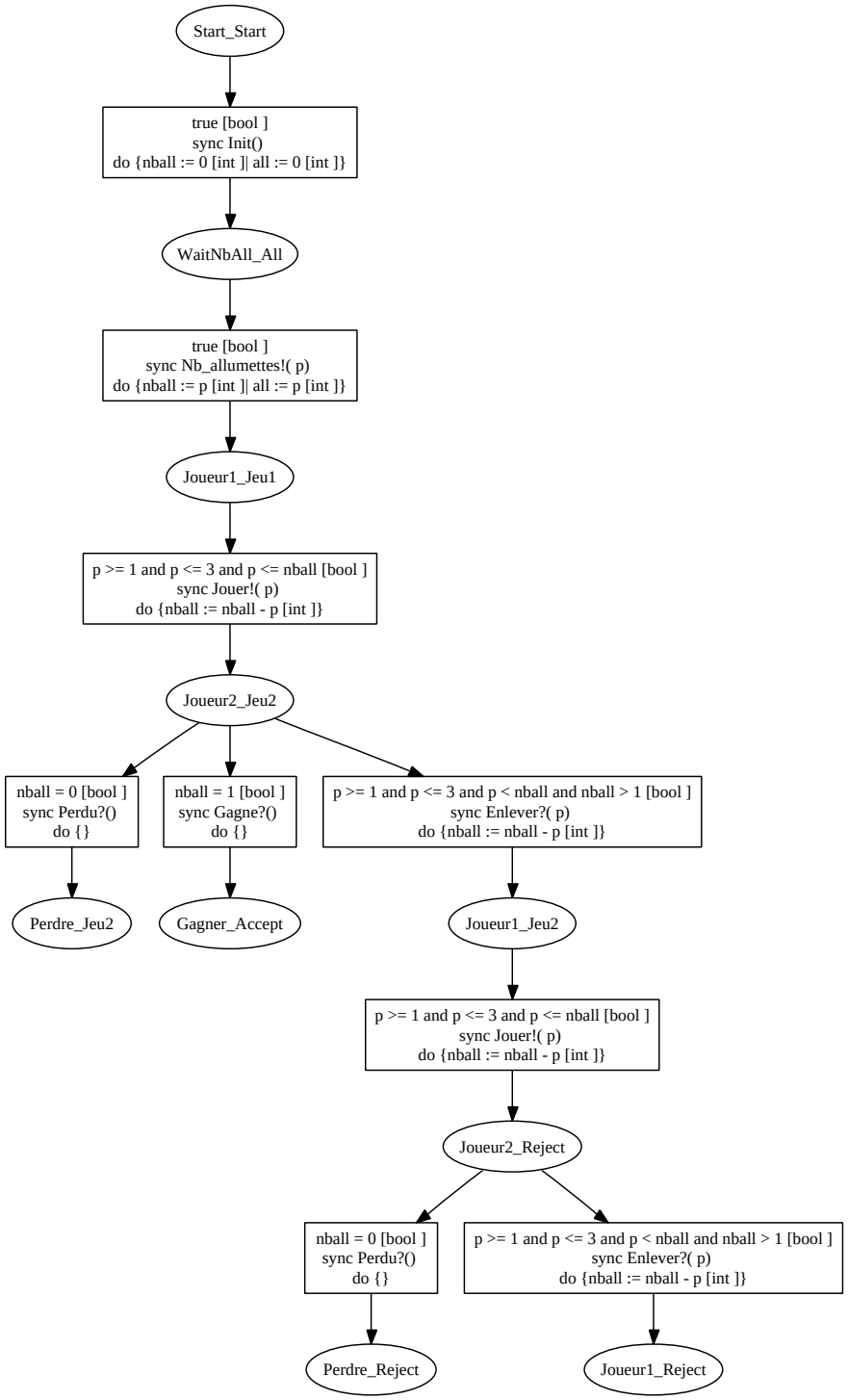


FIG. 5.6 – Produit entre le testeur canonique et l'objectif de test.

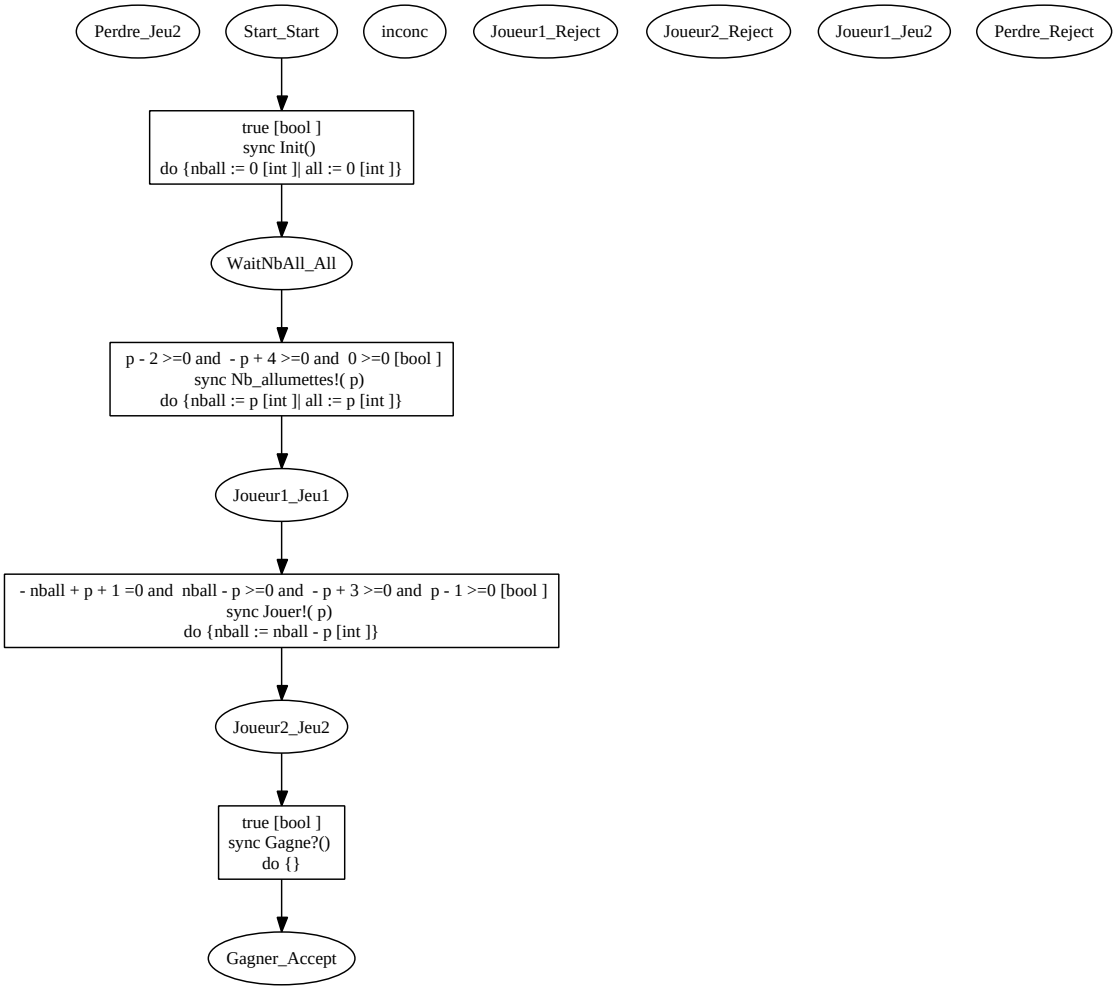


FIG. 5.7 – Cas de test généré à partir de la spécification du jeu de Nim et de l'objectif de test.

Propriété de sûreté. La négation de la propriété de sûreté consistant à vérifier que l'ascenseur ne peut pas monter plus haut que le nombre d'étages maximal est représentée par l'objectif de test de la figure 5.9. La spécification ayant pour garde $p \leq max$ sur la transition portant l'entrée *Go*, la localité *Accept* de l'objectif de test ne peut être atteinte. En effet, lors de la sélection, l'analyse de co-accessibilité de STG ne peut trouver aucun état accessible co-accessible depuis *Accept*. Il produit alors le message suivant :

```
Coreachability analysis: nbac command failed, no reachable states.
```

Nous pouvons en conclure que la spécification satisfait la propriété de sûreté. En revanche, nous ne pouvons rien dire quant à la satisfaction de la propriété par l'implémentation puisqu'aucun cas de test n'a été généré. En théorie, nous pourrions générer un cas de test à partir de la localité $Fail\_Violate$ (correspondant à $Fail\_Accept$ ici). Ainsi, si l'exécution du cas de test sur l'implémentation avait atteint cette localité, nous en aurions conclu que l'implémentation violait la propriété de sûreté en plus de ne pas être conforme à la spécification. Cependant, STG ayant été développé afin de générer des cas de test suivant des objectifs de test, le but n'est pas de vérifier cet objectif sur l'implémentation mais bien de vérifier si sur un comportement précis modélisé par un objectif de test, l'implémentation est conforme à la spécification. Nous ne pouvons donc simuler notre méthode combinant vérification et test sur STG que si la spécification ne vérifie pas la propriété de sûreté. Dans le cas contraire, nous pouvons au moins vérifier cette propriété sur la spécification.

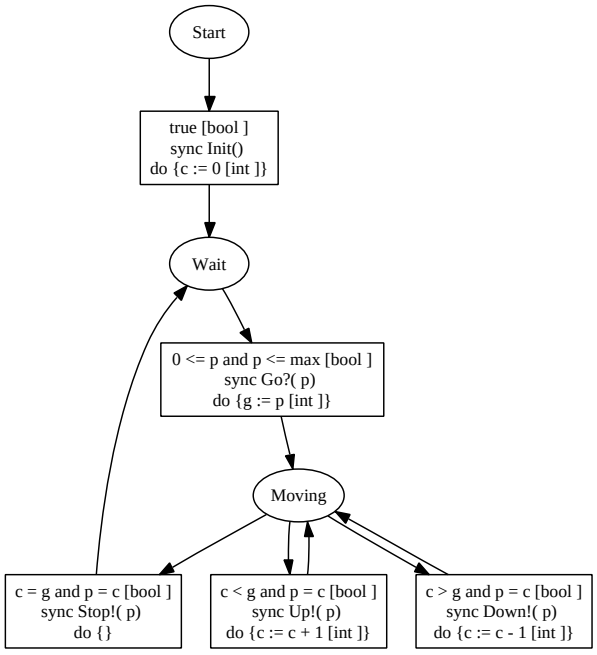


FIG. 5.8 – Spécification de l'ascenseur.

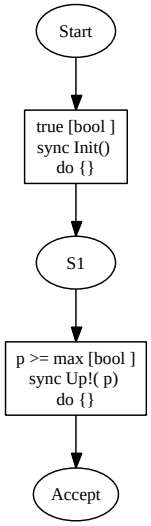


FIG. 5.9 – Objectif de test représentant la négation de la propriété de sûreté.

Propriété d'accessibilité. Reprenons la propriété d'accessibilité présentée dans l'introduction de cette partie, à la figure 2.9(b). Pour rappel, cette propriété un peu particulière consiste à vérifier qu'après un arrêt à un certain étage l, l'arrêt suivant doit être au numéro correspondant à la moitié de l'étage précédemment demandé l tout en étant inférieur au tiers du numéro de l'étage maximal (max). Cette propriété est représentée par l'objectif de test de la figure 5.10. Celui-ci est automatiquement complété par STG comme présenté à la figure 5.11. Nous constatons que cette complétion ajoute des transitions vers la localité *Reject* si la négation des gardes des transitions spécifiées ne vaut pas faux.

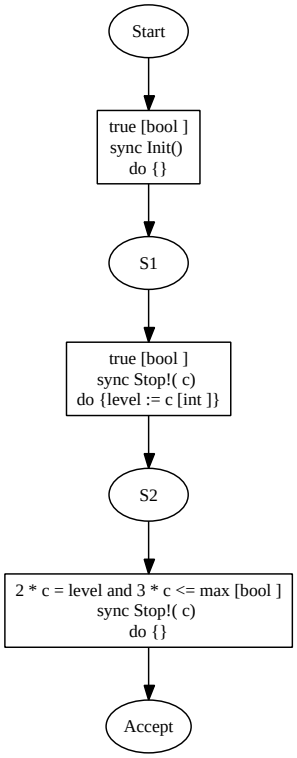


FIG. 5.10 – Objectif de test représentant la propriété d'accessibilité.

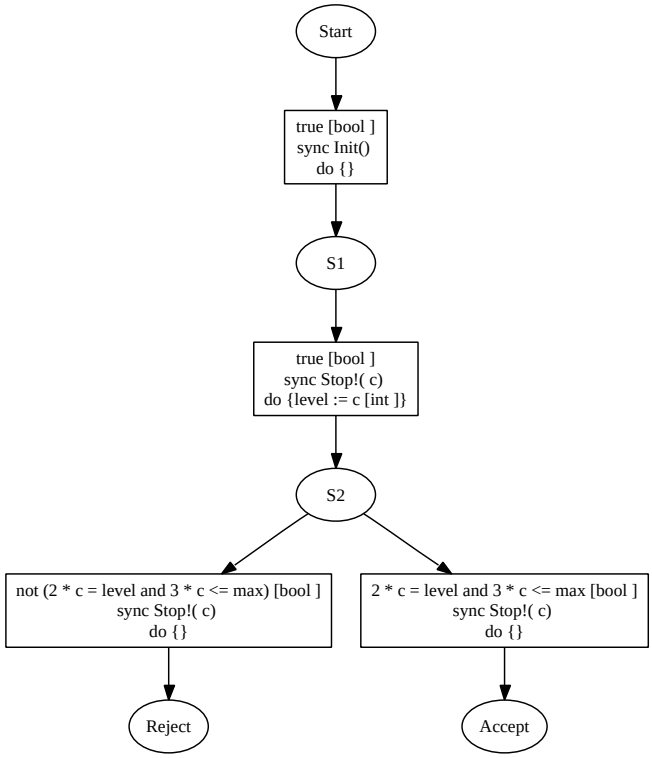


FIG. 5.11 – Objectif de test complété automatiquement.

Le produit entre le testeur canonique de la spécification et cet objectif de test complété est représenté à la figure 5.12.

La sélection effectuée sur ce produit génère le cas de test de la figure 5.13. Les transitions menant à *Fail* ne sont pas représentées mais peuvent être obtenues par complétion des gardes.

L'analyse de co-accessibilité permet de contraindre les gardes des transitions. Un extrait du graphe résultant de cette analyse est représentée à la figure 5.14. Nous pouvons

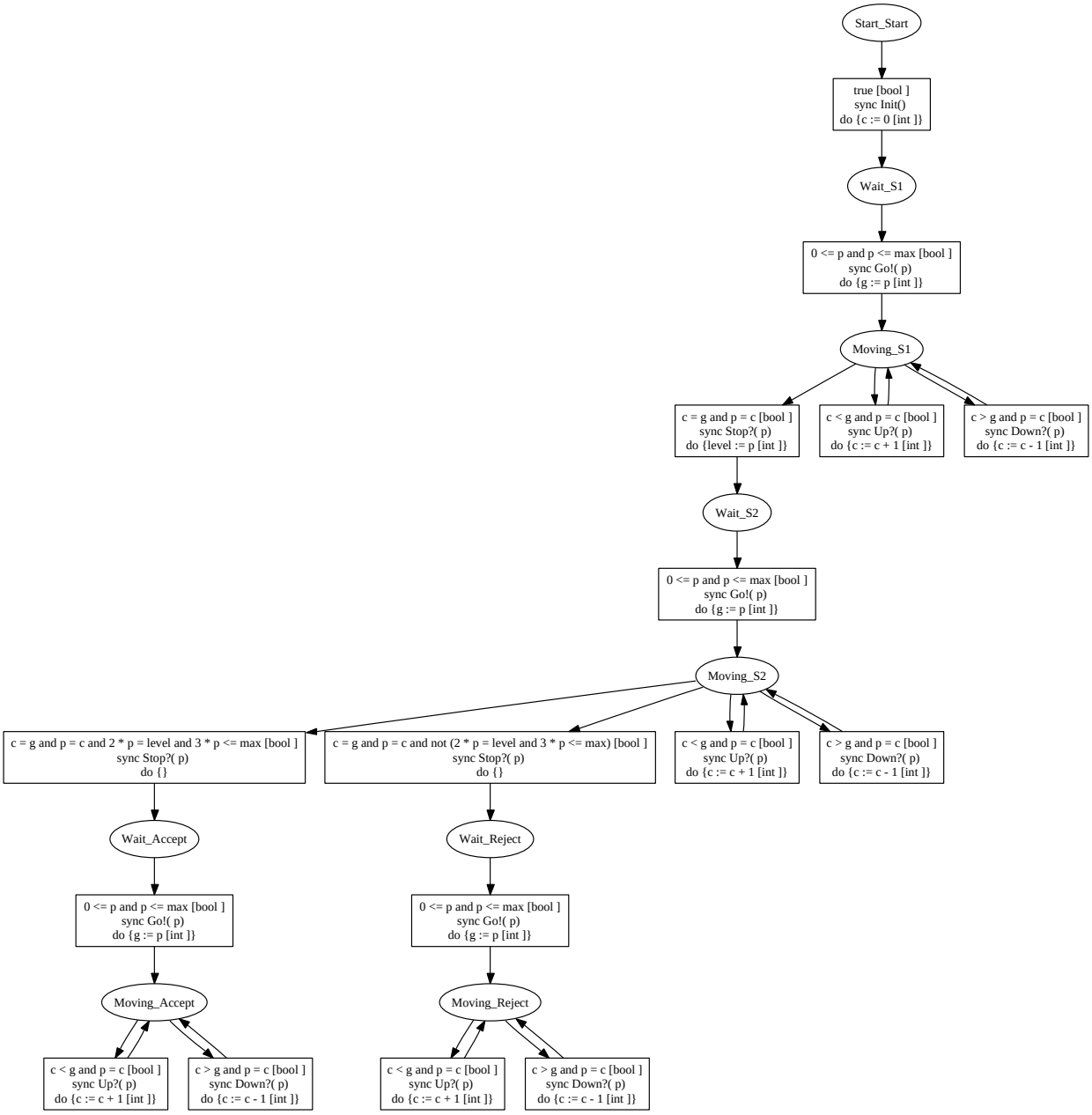


FIG. 5.12 – Produit entre le testeur canonique et l'objectif de test.

par exemple observer que pour atteindre l'objectif de test, l'analyse ajoute la contrainte $2max \geq 3p$ sur la première transition portant la sortie *Go*. Celle-ci a été calculée à partir de la garde de la deuxième transition *Go* : puisque $2p = l$ et $3p \leq max$, NBac en déduit que $3/2l \leq max$. La variable l correspondant au premier étage demandé, c'est-à-dire au paramètre porté par le premier *Go*, la contrainte s'applique à cette transition et se traduit par la garde $2max \geq 3p$.

Cette contrainte permet de satisfaire la garde de la deuxième transition portant la sortie *Go*, à condition que le premier étage demandé soit pair (puisque $2p = l$). Or, l'information portant sur la parité du paramètre porté par la première sortie *Go*, est perdue. En effet, NBac travaille avec des polyèdres rationnels et ne peut donc inférer de contraintes que sur des contraintes linéaires, ce qui n'est pas le cas ici. Nous obtenons donc une approximation du calcul des gardes et non une analyse exacte pour laquelle il aurait fallu effectuer une analyse prenant en compte les congruences. Nous ne pourrons donc pas obtenir une sélection optimale.

L'analyse d'accessibilité permet ensuite de simplifier le graphe en éliminant les transitions ne pouvant pas mener à la satisfaction de l'objectif. Par exemple, les transitions menant à la localité *Inconc* ou celles portant des gardes non satisfiables. Nous obtenons alors le cas de test de la figure 5.14.

Exécution. Nous pouvons maintenant, grâce à STG qui a converti le cas de test en programme SJava, exécuter ce cas de test sur une implémentation Java. Fixons par exemple le nombre d'étages à 10. Le testeur choisit alors la valeur du paramètre de la sortie *Go* de manière à ce qu'elle satisfasse les gardes $0 \leq p \leq 10$ et $3p \leq 20$. Supposons que l'étage choisi est le 4e ($g = 4$). Le testeur observe alors la trace $Up?(0).Up?(1).Up?(2).Up?(3).Stop?(4)$ si l'implémentation est conforme à la spécification. Le testeur n'aura alors pas d'autre choix, pour satisfaire les gardes de la seconde transition *Go* ($0 \leq p \leq 10 \ \wedge \ 2p = 4 \ \wedge \ 3p \leq 20$), que de choisir la valeur 2 (calculée par Lucky). L'ascenseur va alors descendre jusqu'à cet étage et le testeur devrait observer la trace $Down?(4).Down?(3).Stop?(2)$. Le verdict **Pass** sera alors émis : nous ne détectons pas de non-conformité sur cette trace et la propriété est satisfaite. Ce que nous pouvons observer lors de cette exécution est retranscrit à la figure 5.15.

Si le premier étage choisi est le 3 (solution également possible de $0 \leq p \leq 10$ et $3p \leq 20$), alors le testeur ne peut pas satisfaire les gardes de la deuxième transition *Go* : $0 \leq p \leq 10 \ \wedge \ 2p = 3 \ \wedge \ 3p \leq 20$ puisque 3 n'est pas un chiffre pair. Dans ce cas, le verdict Inconc est émis puisque nous ne pouvons plus satisfaire la propriété[1]. Ce que nous pouvons observer lors de cette exécution est retranscrit à la figure 5.16.

Discussion. Le solveur de contraintes Lucky n'a pas trouvé de solution pour tirer la deuxième transition de sortie *Go*. Un choix a été fait dans STG : lorsqu'il n'y a pas de solution à la résolution de contraintes d'une transition de sortie, alors le verdict **Inconc**

[1]Cette situation est due à l'approximation de l'analyse puisque nous ne pouvons contraindre le premier étage choisi à être pair.

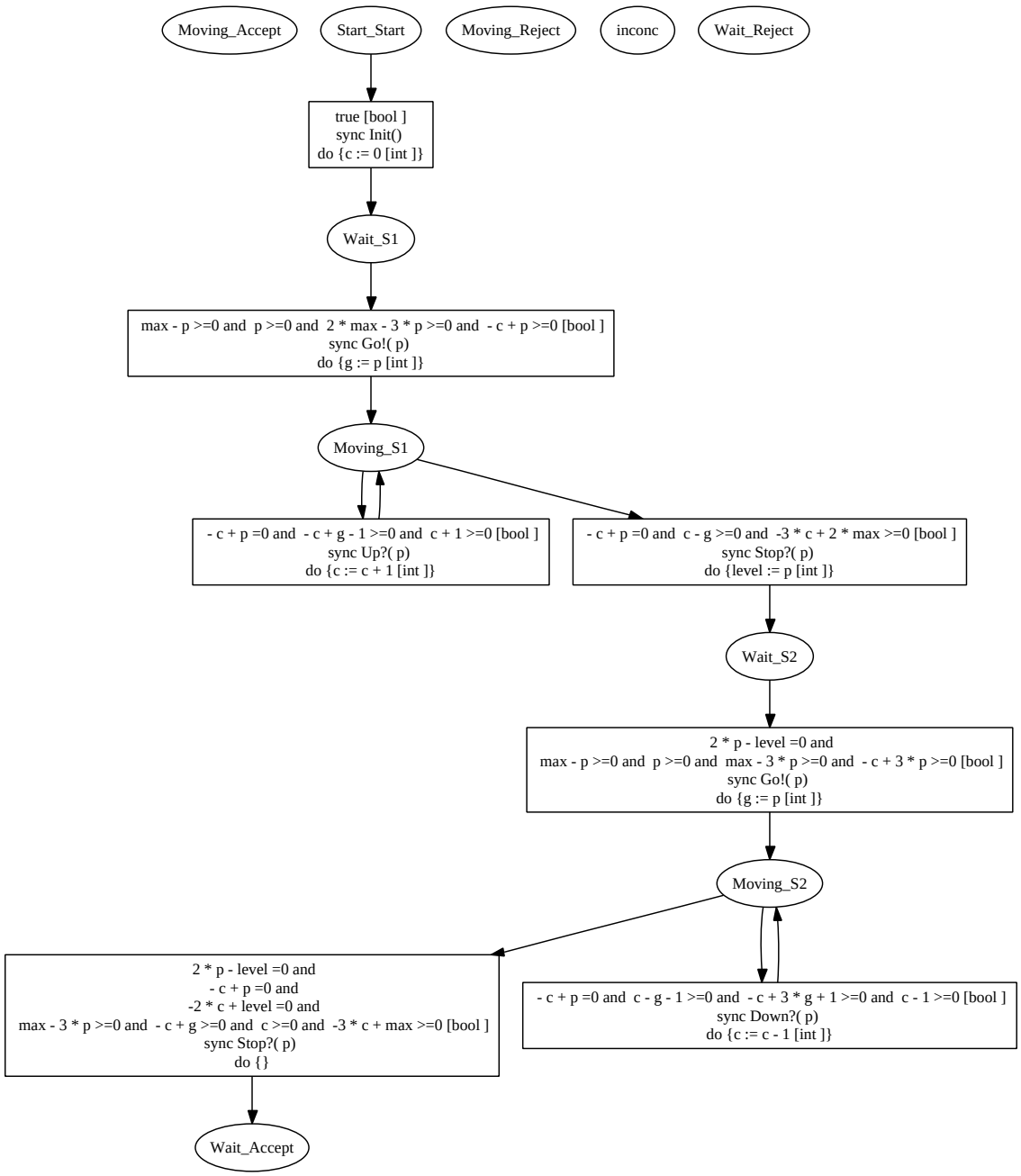


FIG. 5.13 – Cas de test.

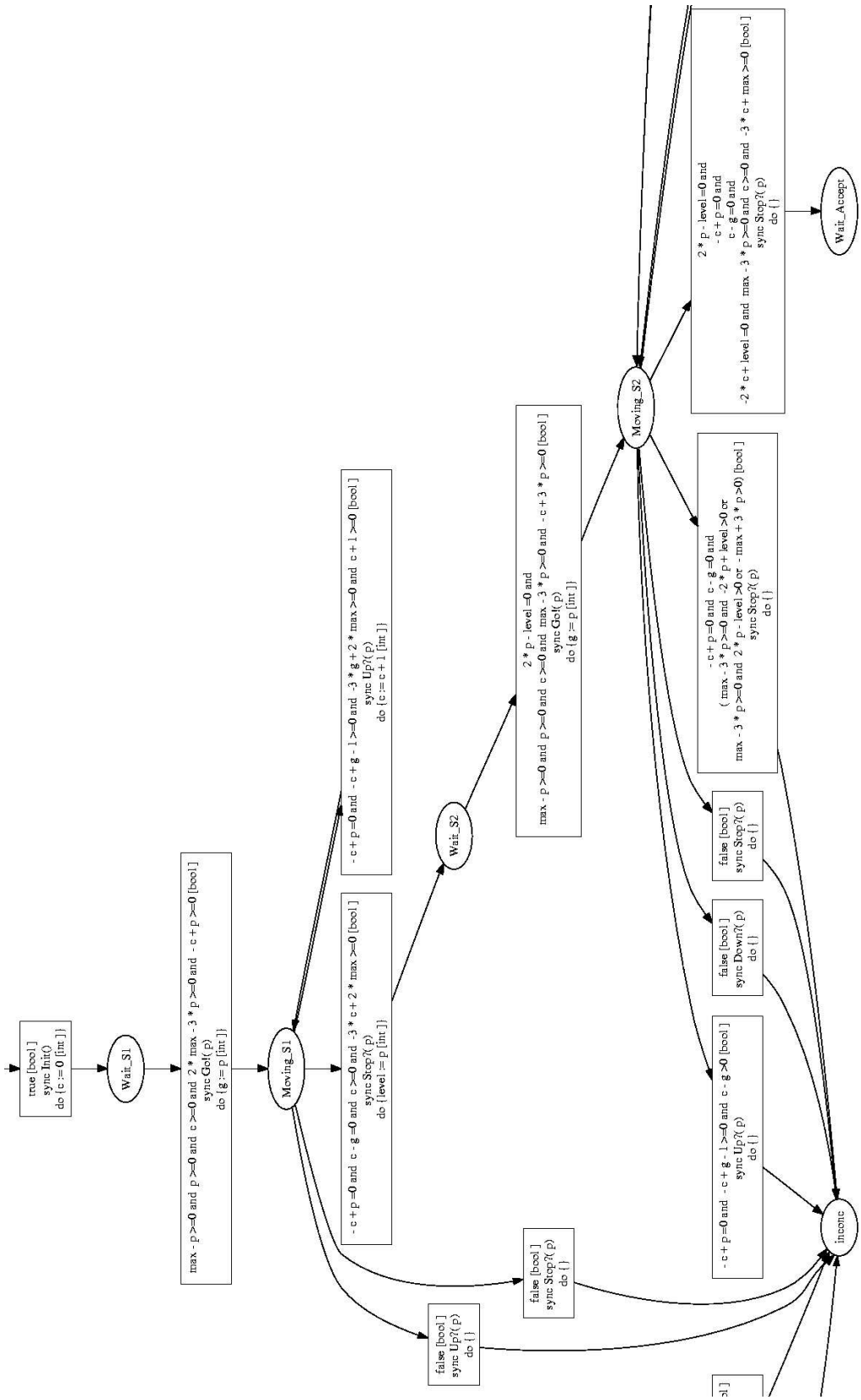


FIG. 5.14 – Extrait du graphe résultant de l'analyse de co-accessibilité.

```
System parameters:
max = 10
start implementation manager
start Tester
From State : Start_Start ->
-> To State : Wait_S1

start lift
                              curr = 0
From State : Wait_S1 ->
lucky_draw
-> To State : Moving_S1

message transmitted to implementation
                              lift::Go max = 10
                              goal = 4
                              curr = 0
                              --> Moving
                        <-- Up 0
                              curr = 1
                              --> Moving
                        <-- Up 1
                              curr = 2
                              --> Moving
                        <-- Up 2
                              curr = 3
                              --> Moving
                        <-- Up 3
                              curr = 4
                              --> Moving
                        <-- Stop 4
                              --> Wait
From State : Moving_S1 ->
Tester receiving message from Implementation
Up
-> To State : Moving_S1

From State : Moving_S1 ->
Tester receiving message from Implementation
Up
-> To State : Moving_S1

From State : Moving_S1 ->
Tester receiving message from Implementation
Up
-> To State : Moving_S1
```

```
From State : Moving_S1 ->
Tester receiving message from Implementation
Up
-> To State : Moving_S1

From State : Moving_S1 ->
Tester receiving message from Implementation
Stop
-> To State : Wait_S2

From State : Wait_S2 ->
lucky_draw
-> To State : Moving_S2

message transmitted to implementation
                              lift::Go max = 10
                              goal = 2
                              curr = 4
                              --> Moving
                        <-- Down 4
                              curr = 3
                              --> Moving
                        <-- Down 3
                              curr = 2
                              --> Moving
                        <-- Stop 2
                              --> Wait
From State : Moving_S2 ->
Tester receiving message from Implementation
Down
-> To State : Moving_S2

From State : Moving_S2 ->
Tester receiving message from Implementation
Down
-> To State : Moving_S2

From State : Moving_S2 ->
Tester receiving message from Implementation
Stop
-> To State : Wait_Accept

From State : Wait_Accept ->
final state reached: Wait_Accept
Verdict: Pass
```

FIG. 5.15 – Résultat de l'exécution : verdict **Pass**.

est émis. Cela se justifie par le fait que les gardes ont été renforcées pour satisfaire l'objectif de test. S'il n'y a pas de solution à ces contraintes, c'est que cet objectif ne peut plus être atteint.

Ce cas ne peut se présenter que pour les transitions de sortie. Pour les transitions d'entrée, Lucky trouvera toujours une solution puisque le cas de test est complet en entrée. Si l'implémentation a émis une sortie non spécifiée, le verdict **Fail** est émis, si l'implémentation a émis une sortie ne pouvant mener à la satisfaction de l'objectif, le verdict **Inconc** est émis.

5.3 QuiDonc

Nous avons, lors de nos expérimentations sur STG, modélisé par un ioSTS l'étude de cas (simplifiée) fournie par France Telecom, QuiDonc. Ce service web permet, à partir d'un numéro entré correct, d'obtenir le nom correspondant. Nous avons modélisé la partie consistant à vérifier le numéro entré, mais le graphe résultant est trop grand pour l'insérer dans ce document. En voici la spécification :

- si le numéro entré est un numéro à 10 chiffres commençant par 01, 02, 03, 04, 05 ou 08, le numéro est correct,
- si le numéro entré est un numéro à 4 chiffres commençant par 36, le numéro est correct,
- sinon, une erreur est émise et un nouveau numéro est entré (le nombre d'essais reste cependant limité).

Cet exemple est intéressant puisqu'il constitue l'application de notre méthodologie à un exemple réel mais les objectifs de test intéressants sont limités. En effet, les numéros de téléphone ne peuvent pas être utilisés comme des entiers puisqu'ils commencent généralement par un "0". Nous ne pouvons donc mettre aucune contrainte numérique portant sur ces numéros dans leur intégralité. Nous avons cependant effectué plusieurs tests, essentiellement avec des objectifs observant les variables de la spécification.

Cette étude de cas contient de la récursivité bornée (modélisée ici par une boucle) ce qui nous a donné l'idée de nous intéresser au test sur des systèmes récursifs, sujet de la partie III de ce document.

```
System parameters:
max = 10
start implementation manager
start Tester
 From State : Start_Start ->
Tester looking for states to fire ...
-> To State : Wait_S1

 From State : Wait_S1 ->
Tester looking for states to fire ...
start lift
                                curr = 0
lucky_draw
-> To State : Moving_S1

ImpManager: receiving ...
message transmitted to implementation
                                lift::Go max = 10
                                goal = 3
                                curr = 0
                                --> Moving
                            <-- Up 0
                                curr = 1
                                --> Moving
                            <-- Up 1
                                curr = 2
                                --> Moving
                            <-- Up 2
                                curr = 3
                                --> Moving
                            <-- Stop 3
                                --> Wait

 From State : Moving_S1 ->
Tester receiving message from Implementation
Up: 0
-> To State : Moving_S1

 From State : Moving_S1 ->
Tester receiving message from Implementation
Up: 1
-> To State : Moving_S1

 From State : Moving_S1 ->
Tester receiving message from Implementation
Up: 2
-> To State : Moving_S1

 From State : Moving_S1 ->
Tester receiving message from Implementation
Stop: 3
-> To State : Wait_S2

 From State : Wait_S2 ->
Tester looking for states to fire ...
lucky_draw
final state reached: Wait_S2
Verdict: Inconc
```

FIG. 5.16 – Résultat de l'exécution : verdict **Inconc**.

Conclusion et travaux connexes

Un système peut être regardé à différents niveaux d'abstraction : des propriétés de haut niveau, une spécification, une implémentation boîte noire, etc. Dans notre cadre, les propriétés et spécifications sont décrites par des ioSTS (*input/output Symbolic Transition Systems*), c'est-à-dire des automates étendus qui opèrent sur des variables et communiquent avec l'environnement par des entrées et sorties portant des paramètres de communication. La sémantique des ioSTS se définit en termes de systèmes de transitions infinis (les ioLTS) par énumération des comportements sur des actions valuées. On suppose que la sémantique de l'implémentation boîte noire peut être décrite par un ioLTS inconnu. Ceci permet de lier formellement l'implémentation à la spécification par une relation de conformité, la relation ioco définie par Tretmans [Tre96]. Par ailleurs, une relation de satisfaction lie ces deux représentations aux propriétés de haut niveau.

Dans cette partie nous proposons une méthodologie de validation afin de tester ces relations, c'est-à-dire permettant de détecter des incohérences entre les différentes vues du système. Tout d'abord, les propriétés d'accessibilité et/ou de sûreté sont vérifiées automatiquement sur la spécification, éventuellement de manière approchée, par exemple par interprétation abstraite. Ensuite, les cas de test sont automatiquement générés à partir de la spécification et des propriétés, puis exécutés sur l'implémentation. Lors de l'exécution, le cas de test permet à la fois de tester la conformité d'une implémentation (ce qui pourrait être vu comme du *monitoring*), mais également de vérifier une propriété sur cette exécution (ce qui est à proprement parler du *monitoring*).

Si la vérification sur la spécification a pu être menée à bien (par exemple elle a permis d'établir que la spécification satisfait une propriété de sûreté), l'exécution peut détecter la violation de la propriété par l'implémentation et/ou la non-conformité de l'implémentation par rapport à sa spécification. De plus, même si la vérification n'a pas permis de valider la propriété de sûreté, l'exécution du test peut également détecter la violation de la propriété par la spécification. Toute incohérence obtenue de cette façon est reportée à l'utilisateur sous la forme d'un verdict. L'approche est complètement formalisée et uniformisée par l'introduction de la notion d'observateur positif ou négatif (spécifiant respectivement les propriétés d'accessibilité et la négation de propriétés de sûreté).

Travaux connexes. [FMP03] décrit une approche pour la génération de tests depuis une spécification et des observateurs (automates de Rabin paramétrés) décrivant

des propriétés de logique temporelle linéaire. Dans cet article la spécification sert uniquement de guide à la génération, mais pas de référence à la conformité. Ainsi les tests générés détectent la violation de propriétés, mais pas la non-conformité.

L'approche décrite dans [ADX01] considère une spécification déterministe à ensemble d'états fini S et un invariant P supposé valide sur S. Des mutants S' de S sont construits en utilisant des opérateurs de mutation classiques. S et S' sont combinés pour produire une machine qui étend les séquences de S par les séquences de S'. Un *model-checker* est ensuite utilisé pour générer des séquences invalidant P, montrant que S' est un mutant de S violant P. Enfin, les séquences ainsi obtenues sont interprétées comme des tests et exécutées sur une implémentation. Contrairement à l'approche décrite dans cet article, notre méthode permet de considérer certaines spécifications non-déterministes à ensemble d'états infini. Nous pouvons de plus générer des tests sur une implémentation boîte noire tandis que [ADX01] requiert un mécanisme permettant d'observer des informations internes de l'implémentation (telles que les valeurs de variables).

Les auteurs de [GH99] utilisent une spécification S et une propriété de logique temporelle P supposée valide sur S. Ils utilisent alors la capacité d'un *model-checker* à construire des contre-exemples de $\neg P$ sur S. Ces contre-exemples sont alors interprétés comme des cas de test. L'idée est étendue dans [HLSU02] en formalisant des critères de couverture standard (`all-definitions, all-uses`, etc) en logique temporelle. La même approche est utilisée dans [BHJP04], mais en formalisant les critères de couverture par des observateurs. À nouveau les cas de test sont générés par *model-checking* des observateurs (ou de la propriété) sur la spécification.

Les approches décrites dans ces travaux reposent sur le *model-checking* et sont limitées à des spécifications d'états finis. De plus ils ne relient pas la satisfaction de propriétés à la conformité, et excepté [FMP03], ne définissent pas formellement de relation de conformité.

[FGLG07] décrit une approche de génération de tests pour composants. Le but est de tester l'intégration d'un composant au sein d'un système en extrayant de la spécification formelle de ce système les comportements précis faisant appel au composant. Les auteurs utilisent la projection de l'exécution symbolique de la spécification (modélisée par un ioSTS) sur le composant comme objectif à la génération de tests de celui-ci.

Dans [RMT+04], les auteurs présentent un premier pas de l'approche combinant vérification et conformité pour des systèmes à états finis. Dans le cas fini, la vérification est décidable, ce qui influence l'approche entière. En effet, la génération de tests ne nécessite alors pas de prendre en compte la possibilité pour la spécification de violer une propriété de sûreté.

Une approche orthogonale de combinaison du *model-checking* et du test de confor-

mité est abordée dans la vérification boîte noire (*Black box checking*) [PVY01]. Sous certaines hypothèses sur l'implémentation (déterminisme, borne sur le nombre d'états), le principe consiste en l'apprentissage du système par une suite de tests de taille exponentielle en la taille de l'implémentation, puis à vérifier des propriétés exprimées par des automates de Büchi.

Notre approche peut aussi être comparée à l'approche de [HR02] qui combine vérification, test et monitoring. Dans cette approche, le monitoring est passif (observation pure), alors que notre approche est réactive et adaptative, guidée par le choix d'entrées à offrir au système telles que calculées dans le cas de test.

Dans l'article [BFG+00], les auteurs utilisent une combinaison rudimentaire d'observateurs positifs et négatifs pour la génération de tests par l'outil TGV et ObjectGéode (Telelogic) à partir d'une spécification SDL du protocole SSCOP. Un observateur GOAL (langage d'observateurs d'ObjectGéode) décrivait une vue abstraite des actions du SSCOP, sur-approximant ses comportements et apparaissant dans la norme du protocole. Des objectifs de tests étaient aussi utilisés par TGV pour la génération de tests. La combinaison de la spécification SDL avec l'observateur permet de détecter la violation de l'observateur par les comportements de la spécification parcourus pendant la génération de tests, ceci afin de s'assurer de la non-violation de l'observateur au moins le long des tests générés.

Enfin, dans [JJRZ05, JJR07] un algorithme symbolique de sélection de cas de test à partir d'une spécification et d'*objectifs de tests* est proposé. La différence réside dans la méthodologie. Les objectifs de tests sont vus uniquement comme des moyens pratiques de sélection de tests fournis par l'utilisateur. Ce sont en fait aussi des propriétés d'accessibilité mais uniquement sur les exécutions et non sur les traces. Le cadre présenté ici pourrait donc s'appliquer à ces objectifs de tests si le modèle des ioSTS n'était pas différent de celui présenté dans cette partie.

Le modèle des ioSTS présenté dans ces papiers contient deux types de variables : les variables propres (ou internes) au système et les variables observées par celui-ci. Typiquement, une spécification ne contient que des variables propres tandis que l'objectif de test, en plus d'avoir ses propres variables, observe celles de la spécification (les contraintes des transitions peuvent porter sur ces variables externes). Ce modèle permet d'avoir des objectifs de test plus expressifs et donc de générer des cas de test plus précis et intéressants. Malheureusement, ce modèle ne peut être utilisé dans le cadre de la combinaison vérification et test telle que nous l'avons présentée. En effet, si nous utilisions des propriétés observant des variables de la spécification, nous ne pourrions rien conclure de la satisfaction/violation de la propriété par rapport à l'implémentation puisque nous ne connaissons de celle-ci que l'interface. Une solution que nous souhaitons développer est de faire du test avec observation de variables.

Troisième partie

Génération de tests pour des spécifications interprocédurales

Introduction

Nous avons vu la problématique de la sélection dans le cadre des ioLTS (lors de la partie I) : la spécification, les cas de tests et l'objectif de test sont alors modélisés par des ioLTS finis, ce qui induit des analyses exactes lors de la phase de sélection de test (voir chapitre 2). Nous avons ensuite illustré cette même phase de sélection dans le cas où ces mêmes objets étaient modélisés par des ioSTS (partie II). Les ioSTS étendent les ioLTS par des données de type infini et permettent de représenter des programmes impératifs non-récursifs. Nous obtenions alors des analyses approchées lors de la sélection de test (voir chapitre 4). Le but de cette partie est d'aborder le problème de sélection de test dans le cas où la spécification est modélisée par un ioPDS [CJJ07]. Le modèle des ioPDS étend celui des ioLTS par une pile portant sur un alphabet fini, ce qui permet de représenter des programmes récursifs manipulant des données de type fini. De telles spécifications peuvent être plus compactes que celles qui ne sont pas récursives et sont plus expressives que les programmes avec une seule procédure. La figure 5.17 résume les différents modèles.

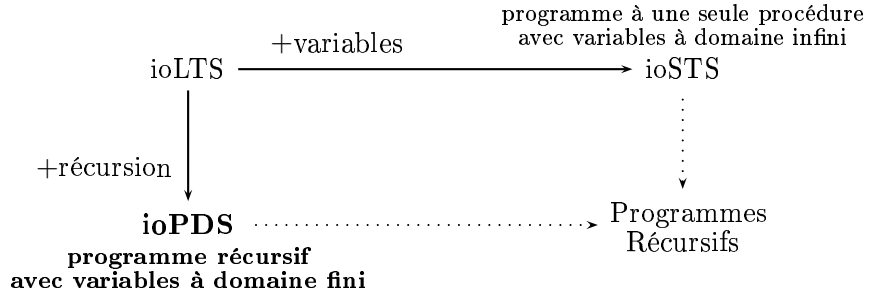
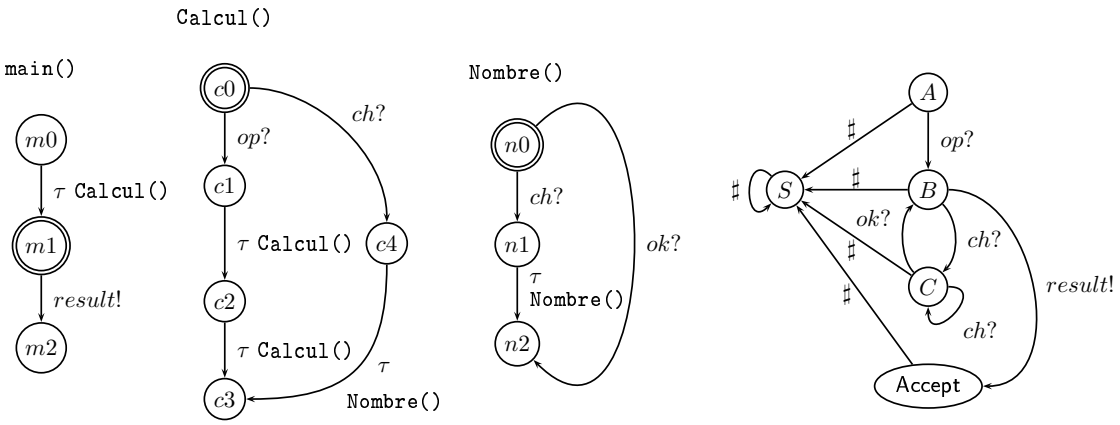


FIG. 5.17 – Modèles étendus à partir de celui des ioLTS

Exemple. Voici un exemple de spécification récursive à partir de laquelle un cas de test sera généré : une calculatrice avec écriture préfixe. Il n'est pas possible de représenter graphiquement un ioPDS, nous allons donc utiliser d'autres représentations pour nos exemples. Une représentation possible de la spécification de la calculatrice est donnée sous forme de graphe de flot de contrôle à la figure 5.18. Le graphe de flot de contrôle permet de représenter une spécification récursive de manière intuitive. Il permet de distinguer chacune des fonctions et, sur chacune d'elles, chaque appel de



(a) Spécification récursive (b) Objectif de test

FIG. 5.18 – Graphes de flot de contrôle de la spécification et de l'objectif de test

fonction, chaque émission ou réception de message et chaque affectation. Nous pouvons également modéliser une spécification récursive sous forme de programme. Un exemple de cette modélisation est présenté à la figure 5.19. La spécification représentée est celle d'une calculatrice avec écriture préfixe dont le comportement est le suivant. La fonction principale (`main()`) appelle la fonction `Calcul()` puis quand celle-ci a fini de s'exécuter, émet le résultat calculé. La fonction `Calcul()` a deux entrées possibles : si un chiffre (*ch*?) est reçu, alors la fonction `Nombre()` est appelée, si l'entrée est un opérateur (*op*?) alors la fonction `Calcul()` est rappelée deux fois récursivement. La fonction `Nombre()` a également deux comportements possibles suivant son entrée : si c'est un chiffre, elle est à nouveau appelée récursivement, si c'est le signal indiquant la fin du nombre entré (*ok*?), alors la fonction retourne à la fonction appelante. Les états avec doubles cercles désignent les points de contrôles stables, c'est-à-dire les points de contrôle à partir desquels une action observable est tirable (entrée ou sortie).

La méthode vue précédemment sur les ioLTS(chapitre 2) est ensuite appliquée : le testeur canonique est généré puis une sélection par objectif de test est effectuée dessus. L'objectif de test utilisé dans cet exemple est représenté sous forme d'ioLTS à la figure 5.18 et sous forme de programme à la figure 5.20. Nous rappelons que le symbole ♯ signifie "tous les autres éléments de l'alphabet" et l'état Accept dénote l'état final indiquant la satisfaction de l'objectif de test si celui-ci est atteint. Cet objectif est satisfait pour l'ensemble des traces suivantes :

$$Traces(TP, \mathsf{Accept}) = \{op \cdot (ch^+ \cdot ok)^* \cdot result\}.$$

La sélection va alors essayer de forcer le testeur canonique à produire la trace $op \cdot (ch^+ \cdot ok)^* \cdot result$ lors de son exécution sur l'IUT. Pour cela, des transformations vont être effectuées sur le programme représentant le produit entre ce testeur canonique et l'objectif de test (figure 5.21). Le produit consiste à ajouter aux points de contrôle

```
enum out_t { result };
enum inp_t { ch, op, ok };

void main(){
  m0: Calcul();
  m1: emit(p) when (p == result){};
  m2:
}

void calcul()
{
  c0: receive(p) when (p == op || p == ch){
        if (p == ch) goto c4;
      };
  c1: calcul();
  c2: calcul();
  c3: return;
  c4: nombre();
  c5:
}

void nombre()
{
  n0: receive(p) when (p == ch || p == ok){
        if (p == ok) goto n2;
      };
  n1: nombre();
  n2: return;
}
```

FIG. 5.19 – Spécification correspondant à la figure 5.18.

```
enum pc_t { A,B,C,D,S };
enum pc_t pc = A;

void TP(enum msg_t p)
{
  if    (pc == A && p == op ) pc = B;
  elsif (pc == B && p == ch)  pc = C;
  elsif (pc == C && p == ch)  pc = C;
  elsif (pc == C && p == ok)  pc = B;
  elsif (pc == B && p == result ){
      pc = D;
      verdict = pass;
      abort();
  }
  else pc = S;
}
```

FIG. 5.20 – Objectif de test correspondant à la figure 5.18.(b)

stables des appels à la fonction `TP()`, après avoir vérifié l'absence d'erreur de conformité à ces points. La fonction `TP()` définie à la figure 5.20 prend en entrée le dernier message échangé et implémente l'automate de la figure 5.18. Si l'état final est atteint, le verdict *Pass* est émis.

La sélection va renforcer les contraintes sur les émissions et réceptions grâce à une analyse de co-accessibilité et d'accessibilité. Le cas de test obtenu après sélection est décrit à la figure 5.22. Deux types de modifications ont été apportées par rapport au produit. Aux points de contrôles $c0r$ et $n0r$, lorsque des *delta* sont reçus, le verdict *Inconc* est émis. De plus, au point de contrôle $c0$, les conditions pour émettre un message ont été renforcées : un opérateur n'est émis que si la variable `pc` vaut A, en d'autres termes, seulement si la fonction `Calcul()` est appelée pour la première fois. Si `pc` vaut B, alors un chiffre est émis. Ainsi, nous constatons que la connaissance de la valeur de la variable `pc` permet de raffiner le cas de test afin de choisir les traces permettant la satisfaction de l'objectif de test. Le cas de test est donc optimal au point $c0$. En revanche, la sélection n'a pas permis d'être aussi précis au point de contrôle $n0$. Nous n'avons ici aucun moyen de savoir quand produire *ok* plutôt qu'un chiffre. La sélection ne permet donc pas de sélectionner le cas de test afin d'arriver à coup sûr à la satisfaction de l'objectif.

Nous présenterons notre contribution en trois chapitres : nous décrirons au chapitre 6 la méthode de génération de tests prenant en entrée une spécification récursive et un objectif de test non-récursif et retournant un cas de test récursif. Nous présenterons ensuite cette même méthode au chapitre 7 mais celle-ci sera cette fois basée sur des transformations de programmes. Les choix techniques seront guidés par la théorie du test et par les propriétés des modèles des ioPDS et des ioLTS, mais la génération en elle-même sera définie en termes de concepts de langages de programmation. Nous présenterons ensuite au chapitre 8 notre algorithme de sélection prenant en entrée le cas de test généré dans le chapitre précédent et le spécialisant. Nous formaliserons le problème de l'observation partielle dû à l'incapacité des cas de test à inspecter le contenu de leur propre pile. Nous comparerons les conséquences de ce problème sur les cas de test générés avec l'impact que peut avoir une analyse inexacte (approchée) comme celle faite lors de la sélection de test sur le modèle symbolique des ioSTS (chapitre 4). Nous proposerons une amélioration de cet algorithme de sélection permettant de minimiser l'impact négatif de l'observation partielle durant la sélection. Enfin, des expérimentations sur un analyseur interprocédural (InterprocStack) seront présentées au chapitre 9.

```
enum inp_t { result, delta };
enum out_t { ch, op, ok };
enum verdict_t { none, fail, pass, inconc };
enum verdict_t verdict = none;

void main(){
  m0: Calcul();
  m1: receive(p) when true{
  m1r:  if (p != result)
          { verdict = fail; abort(); }
        TP();
      }
  m2:
}

void calcul()
{
  c0: emit(p) when (p == op || p == ch){

  c0e:  TP();
        if (p == ch) goto c4;
      }
      []
      receive(p) when true{
  c0r:  if (p != delta)
          { verdict = fail; abort(); }
        TP();
        if(p == delta) goto c0;

      };
  c1: calcul();
  c2: calcul();
  c3: return;
  c4: nombre();
  c5:
}

void nombre()
{
  n0: emit(p) when (p == ch || p == ok){
  n0e:  TP();
        if (p == ok) goto n2;
      }
      []
      receive(p) when true{
  n0r:  if (p != delta)
          { verdict = fail; abort(); }
        TP();
        if(p == delta) goto n0;

      };
  n1: nombre();
  n2: return;
}
```

FIG. 5.21 – Produit

```
enum inp_t { result, delta };
enum out_t { ch, op, ok };
enum verdict_t { none, fail, pass, inconc };
enum verdict_t verdict = none;

void main(){
  m0: Calcul();
  m1: receive(p) when true{
  m1r:  if (p != result)
          { verdict = fail; abort(); }
        TP();
      }
  m2:
}

void calcul()
{
  c0: emit(p) when (p == op && pc == A
                || p == ch && pc == B){
  c0e:  TP();
        if (p == ch) goto c4;
      }
      []
      receive(p) when (p == delta){
  c0r:  if (p != delta)
          { verdict = fail; abort(); }
        TP();
        if(p == delta)
          { verdict = inconc; abort(); }
      };
  c1: calcul();
  c2: calcul();
  c3: return;
  c4: nombre();
  c5:
}

void nombre()
{
  n0: emit(p) when (p == ch || p == ok){
  n0e:  TP();
        if (p == ok) goto n2;
      }
      []
      receive(p) when (p == delta){
  n0r:  if (p != delta)
          { verdict = fail; abort(); }
        TP();
        if(p == delta)
          { verdict = inconc; abort(); }
      };
  n1: nombre();
  n2: return;
}
```

FIG. 5.22 – Cas de test obtenu après sélection

Chapitre 6

Modélisation et génération du testeur canonique à partir d'ioPDS

Nous avons récapitulé dans la partie I de ce document la théorie du test de conformité basée sur le modèle des ioLTS. Nous avons ensuite étendu ces principes aux systèmes de transitions symboliques dans la partie II. Nous allons appliquer ici les mêmes principes mais sur des spécifications récursives, représentées par des automates à pile à entrées/sorties, les ioPDS (*input/output PushDown Systems*).

6.1 Modélisation par des ioPDS

6.1.1 Syntaxe des ioPDS

De tels systèmes manipulent des données finies (voir définition 6.1) mais peuvent avoir un comportement infini dû à la récursivité. Ils sont donc plus expressifs que les ioLTS. Même si la récursivité était bornée, la spécification pourrait être représentée par un ioLTS grâce au dépliage (*inlining*) des appels récursifs mais serait alors nettement moins lisible qu'avec les ioPDS. La modélisation par un ioPDS est alors bien plus compacte que celle par ioLTS.

Définition 6.1 (ioPDS) *Un système à pile à entrées/sorties (ioPDS) est défini par un tuple* $\mathcal{P} = \langle G, \Gamma, \Lambda, c_0, \hookrightarrow \rangle$ *où :*

- G *est un ensemble fini d'états ;*
- Γ *est un alphabet fini de pile ;*
- $\Lambda = \Lambda_? \cup \Lambda_! \cup \Lambda_\tau$ *est un alphabet fini composé des actions visibles d'entrées* ($\Lambda_?$) *et de sorties* ($\Lambda_!$) *et des actions internes* (Λ_τ) *;*
- $c_0 \in G \times \Gamma$ *est la configuration initiale ;*
- $\hookrightarrow \subseteq (G \times \Gamma) \times \Lambda \times (G^* \times \Gamma^*)$ *est un ensemble fini de transitions étiquetées.*

Nous appellerons *configuration* une paire $\langle g, \omega \rangle$ composée d'un état $g \in G$ et du contenu de la pile $\omega \in \Gamma^*$. Nous rappelons que dans une pile, l'élément qui se trouve au sommet est le dernier à avoir été empilé et le premier que nous pouvons dépiler.

Nous représenterons un contenu de pile par un mot. Les symboles de la pile seront alors écrits de gauche à droite en partant du bas vers le haut de la pile. Une pile notée $\omega \cdot \gamma$ signifie alors que ω est le contenu de la pile ($\omega \in \Gamma^*$) et γ son sommet de pile ($\gamma \in \Gamma$).

Nous utiliserons par la suite les ioPDS pour modéliser les programmes récursifs. Nous distinguerons dans un système à pile trois types de transitions :

- les transitions portant sur le sommet de pile que nous appellerons les transitions simples ; ce sont les transitions étiquetées par des entrées ou des sorties et les transitions d'affectation de variables qui entraînent une modification du sommet de pile (dépilement puis empilement d'un nouveau sommet) sans modifier la taille de la pile ;
- les transitions d'empilement permettant de modéliser les appels de fonction (appelées transitions d'appel) ;
- les transitions de dépilement permettant de modéliser le retour de fonction (appelées transitions de retour).

Ces trois types de transitions seront illustrés dans l'exemple 6.1.

Une configuration sera composée de deux éléments : l'état comportant une éventuelle localité et des valeurs des variables globales (de type énuméré) et la pile. Cette pile contiendra des tuples composés des points de contrôle du programme modélisé et des valeurs des éventuelles variables locales.

Nous appellerons points de contrôle les points du programme avant toute instruction (entrées, sorties, affectations, appels et retours de fonctions). Nous distinguerons par la suite les points de contrôle stables (abrégés en PCS), qui seront les points de contrôle à partir desquels seront attendus une entrée et/ou une sortie, des points de contrôle instables (PCI) à partir desquels seule une action interne est tirable. La distinction entre PCS et PCI sera expliquée en sous-section 6.2.3.

Il est difficile de représentation graphiquement, de manière élégante, des ioPDS car il faut présenter à la fois les entrées/sorties mais également l'évolution de l'état de la pile. Nous modéliserons par la suite les exemples par un graphe de flot de contrôle, une modélisation assez intuitive. Les PCS y seront distingués par l'utilisation de doubles cercles.

Exemple 6.1 *Un exemple de spécification récursive est donné figure 6.1. Cette spécification est représentée par un graphe de flot de contrôle. Ce programme invite l'utilisateur à envoyer des messages. Lorsque celui-ci indique qu'il n'a plus de messages à envoyer, le programme lui renvoie alors le même nombre d'acquittements que de messages envoyés.*

Voici un descriptif du comportement plus détaillé. La fonction de départ `main()` *appelle la fonction* `F()`*. Celle-ci a deux comportements possibles : soit elle émet la sortie Pb! et retourne à sa fonction appelante (en l'occurrence* `main()`*), soit elle émet la sortie Invit!, appelle la fonction* `G()` *puis reçoit l'entrée Quit? et retourne à sa fonction*

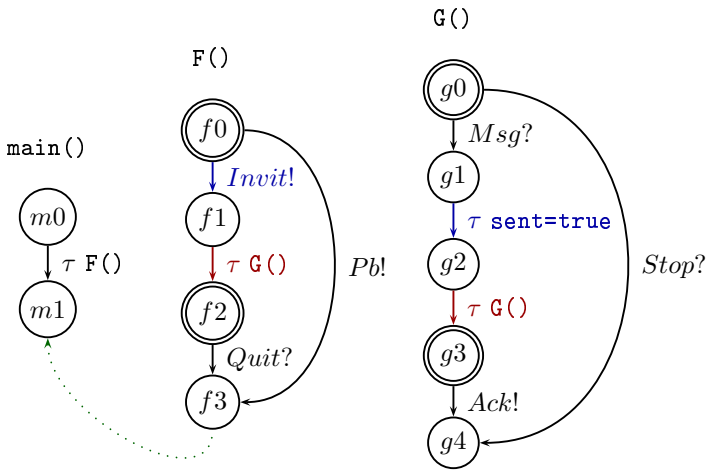


FIG. 6.1 – Spécification récursive reconnaissant les traces $Invit \cdot Msg^n \cdot Stop \cdot Ack^n \cdot Quit + Pb$ avec $n \geq 0$

appelante. Suivant la première entrée reçue dans la fonction `G()`*, celle-ci va avoir deux comportements possibles : si l'entrée est* $Stop?$*, elle retourne à sa fonction appelante tandis que si l'entrée est* $Msg?$*, la fonction* `G()` *est appelée à nouveau (récursivement) et après que cet appel soit terminé, la sortie* $Ack!$ *est émise.*

L'ensemble d'états $G = \{(p, sent) \mid sent \in \{true, false\}\}$ *est ici composé de la localité* p *et de la valeur de la variable globale* $sent$*. L'alphabet de pile correspond quant à lui à l'ensemble des points de contrôle* $(m_0, m_1, f_0, etc.)$*.*

Nous remarquons dans cet exemple les trois types de transitions (représentées sous forme de règles de transitions de l'automate à pile figure 6.2). La transition de f_0 *à* f_1 *portant la sortie* $Invit!$ *est par exemple une transition atomique, tout comme celle allant de* g_1 *à* g_2 *et portant l'affectation de la variable* `sent` *à true (transitions bleues sur la figure 6.1). Les transitions portant des appels de fonction telles que celle allant de* m_0 *à* m_1 *ou celle de* g_2 *à* g_3 *sont des transitions d'appel (transitions rouges). Enfin, les transitions de retour ne sont pas visibles sur ce graphe. Ce sont les transitions allant des états de fin de procédure comme* f_3 *aux états de fin d'appel comme* m_1 *(transition verte en pointillés sur la figure 6.1).*

Les traces reconnues par cette spécification sont $Traces(\mathcal{S}) = \{Pb!\} \cup \{Invit! \cdot (Msg?)^n \cdot Stop? \cdot (Ack!)^n \cdot Quit? \mid n \geq 0\}$*, qui est un langage algébrique (voir sous-section 6.1.2). Notons que la variable globale* `sent` *n'a aucune influence sur les traces de la spécification. Nous expliquerons dans le chapitre 8 l'utilité de cette variable.*

Transitions atomiques :	$\langle (p,-), \omega \cdot f_0 \rangle \overset{Invit!}{\hookrightarrow} \langle (p,-), \omega \cdot f_1 \rangle$
	$\langle (p,-), \omega \cdot g_1 \rangle \overset{\tau}{\hookrightarrow} \langle (p,true), \omega \cdot g_2 \rangle$
Transition d'appel :	$\langle (p,-), \omega \cdot f_1 \rangle \overset{\tau}{\hookrightarrow} \langle (p,-), \omega \cdot f_2 \cdot g_0 \rangle$
Transition de retour :	$\langle (p,-), \omega \cdot m_1 \cdot f_3 \rangle \overset{\tau}{\hookrightarrow} \langle (p,-), \omega \cdot m_1 \rangle$

FIG. 6.2 – Exemples de règles de transitions avec p une localité, ω le contenu de pile et $-$ représentant n'importe quelle valeur de la variable `sent`.

6.1.2 Sémantique des ioPDS

La sémantique d'un ioPDS peut être définie en termes de systèmes de transitions à entrées/sorties (ioLTS) infini. Intuitivement, la sémantique ioLTS d'un ioPDS énumère toutes les configurations possibles : chaque état de l'ioLTS correspond à une configuration de l'ioPDS. La sémantique formelle d'un ioPDS est définie comme suit.

Définition 6.2 (Sémantique d'un ioPDS) *Un ioPDS $\mathcal{P} = \langle G, \Gamma, \Lambda, c_0, \hookrightarrow \rangle$ génère un ioLTS infini $M = \langle Q^M, q_0^M, \Lambda, \rightarrow \rangle$ pour lequel :*

- $Q^M = G \times \Gamma^*$,
- $q_0^M = c_0$,
- $\rightarrow$ *est la relation de transition définie par la règle :*

$$\langle g, \gamma \rangle \overset{\lambda}{\hookrightarrow} \langle g', \gamma' \rangle \wedge \omega \in \Gamma^* \implies \langle g, \omega \cdot \gamma \rangle \overset{\lambda}{\rightarrow} \langle g', \omega \cdot \gamma' \rangle.$$

Exemple 6.2 *Une partie de la sémantique de la spécification de la figure 6.1 est représentée figure 6.3. Nous n'avons pas représenté l'intégralité de ce système puisque celui-ci est infini (la fonction `G()` a une récursivité non-bornée). Chaque état de l'ioLTS $[\![\mathcal{S}]\!]$ est une configuration de l'ioPDS $\mathcal{S}$, composée de l'état de l'ioPDS, c'est-à-dire un tuple avec une localité p et la valeur de la variable booléenne `sent`, et le contenu de la pile, en l'occurrence les points de contrôle du programme. L'état initial, correspondant à la configuration initiale de l'ioPDS, est composé du tuple $\langle p, ff \rangle$ puisque la variable globale `sent` a initialement la valeur false et du contenu de pile m_0 puisque le programme débute au point de contrôle initial m_0 de la fonction de départ `main()`.*

Les couleurs des différentes transitions vues dans l'exemple 6.1 ont été maintenues ici pour une meilleure lisibilité. Ainsi, nous pouvons observer un empilement lorsque la transition est un appel de fonction (transitions rouges) et un dépilement lors des retours (transition verte). Lors des transitions atomiques (couleur bleue), le sommet de pile est modifié, ce qui correspond à un dépilement suivi d'un empilement. La transition en pointillés indique que l'ioLTS continue par cette transition.

Langage des ioPDS. Comme pour les ioSTS, nous pouvons maintenant décrire le comportement d'un ioPDS grâce à sa sémantique (voir la définition 1.12 d'une exécution). Puisque les variables et le contenu de la pile (les états de la sémantique ioLTS)

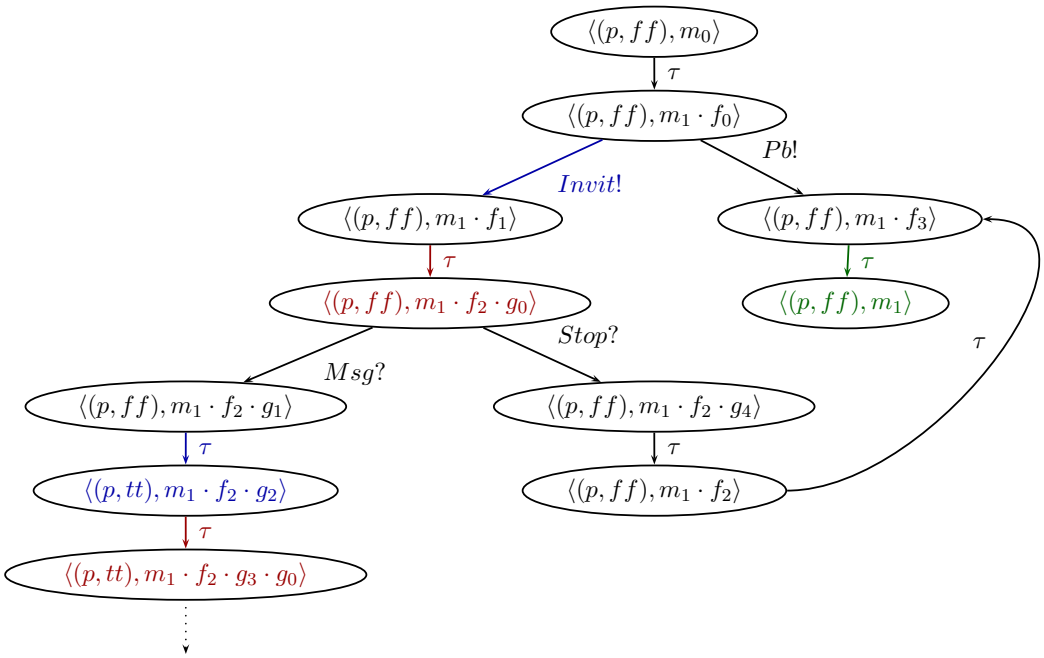


FIG. 6.3 – Sémantique $[\![\mathcal{S}]\!]$ de la spécification $\mathcal{S}$

ne peuvent pas être observés par l'environnement, nous ne considérons que les traces d'exécution (définition 1.3).

Le langage reconnu par un système à pile (PDS), et donc par un ioPDS, est un langage algébrique, dit également langage hors-contexte (*context-free* en anglais) et non un langage régulier (ou rationnel) comme pour les LTS. Les langages algébriques sont exactement les langages reconnus par un automate à pile ou par une grammaire algébrique [HMU01]. L'ensemble des traces d'un ioPDS est également algébrique puisque les traces sont un langage dans lequel les actions internes ne sont pas reconnues. Notons qu'en revanche le langage de la pile est rationnel.

Les langages algébriques ont des propriétés différentes des langages rationnels. En effet, ces langages sont clos par union, concaténation, étoile de Kleene, projection et projection inverse, mais contrairement aux langages réguliers, ils ne sont ni par intersection, ni par complémentation. Nous en verrons les conséquences à la sous-section 6.2.4.

6.2 Génération de tests : opérations sur les ioPDS

Nous utiliserons dans cette section la méthode, basée sur la relation de conformité ioco, vue au chapitre 2, permettant de générer des cas de test à partir d'une spécification. Cette méthode se divise en plusieurs transformations appliquées au modèle de la spécification : la déterminisation, la suspension et la complétion en sortie. Nous ver-

rons que la suspension et la complétion en sortie (sous-sections 6.2.2 et 6.2.3) sont des opérations applicables sur les ioPDS, mais que ce n'est pas le cas de la déterminisation (sous-section 6.2.1). Nous verrons également en sous-section 6.2.4 que les propriétés du produit avec un automate à pile, utile lors de la sélection par objectif de test (voir chapitre 2, section 2.2), contraint le modèle de l'objectif de test.

6.2.1 ioPDS déterministe et déterminisation

Intuitivement, un ioPDS est *déterministe* si chacune de ses traces correspond exactement à une exécution. On définit un ioPDS comme étant déterministe si, d'une part, il n'y a pas deux transitions de même étiquette tirables à partir d'une même configuration et d'autre part, si, lorsqu'une transition portant une action non-observable est possible, aucune autre transition ne l'est. Notons que le non-déterminisme observable (différentes sorties peuvent être émises à partir d'un même état) est permis dans un ioPDS déterministe.

Définition 6.3 (ioPDS déterministe) *Un ioPDS $\mathcal{P} = \langle G, \Gamma, \Lambda, c_0, \hookrightarrow \rangle$ est dit déterministe si et seulement si :*

- $\forall c, c_1, c_2 \in G \times \Gamma^*$ *et* $\lambda \in \Lambda$, $c \overset{\lambda}{\hookrightarrow} c_1$ *et* $c \overset{\lambda}{\hookrightarrow} c_2$ *implique* $c_1 = c_2$, *et*
- $\forall c \in G \times \Gamma^*$, *si* $c \overset{\tau}{\hookrightarrow}$, *alors* $\neg(\exists \lambda \in \Lambda : c \overset{\lambda}{\hookrightarrow})$.

La sémantique d'un ioPDS déterministe est un ioLTS déterministe dans le sens des ioPDS : soit $M = \langle Q^M, q_0^M, \Lambda^M, \rightarrow_M \rangle$ un ioLTS, celui-ci est déterministe si, pour $p, q, q' \in Q^M$, $\lambda \in \Lambda$, $p \overset{\lambda}{\rightarrow} q \wedge p \overset{\lambda}{\rightarrow} q' \Rightarrow q = q'$ et $p \overset{\tau}{\rightarrow} q \Rightarrow \neg(\exists \lambda \in \Lambda : p \overset{\lambda}{\rightarrow})$. Cette définition ne correspond pas à celle donnée à la définition 1.5 d'un ioLTS déterministe puisque les transitions d'actions internes sont ici possibles. Ceci entraîne le fait que, contrairement aux ioLTS et aux ioSTS déterministes, le langage d'un ioPDS $\mathcal{P}$ déterministe n'est pas égal aux traces de celui-ci : $L(\mathcal{P}) \neq \mathit{Traces}(\mathcal{P})$.

Le langage d'un automate à pile déterministe est exactement un langage déterministe algébrique. Celui-ci a les mêmes propriétés que les langages algébriques, mais il est en plus clos par complémentation. La classe des langages déterministes algébriques est donc strictement incluse dans celle des langages algébriques. En effet, il existe des langages algébriques non-déterministes. Un exemple classique est le langage des palindromes sur l'alphabet $\{a, b\}$. $\{w \in (a+b)^* \mid w \textit{ est un palindrome}\}$ est algébrique mais n'est pas déterministe. Intuitivement, on ne sait pas deviner où est le milieu du mot. En revanche, $\{w_1 c w_2 \mid w_1 w_2 \in (a+b)^* \textit{ est un palindrome}\}$ est un langage algébrique déterministe.

Comme expliqué au chapitre 2, dans la sous-section 2.1.1, il est nécessaire de générer un cas de test à partir d'une spécification déterministe. Pour cela, nous devons soit partir d'une spécification déterministe, soit déterminiser celle-ci. Malheureusement, il n'est pas possible de déterminiser un automate à pile dans le cas général, puisque tous les automates à pile ne sont donc pas déterminisables. De plus, le fait de savoir si un automate à pile est déterminisable est indécidable. Il n'existe donc aucun algorithme

de déterminisation sur les automates à pile.

Il existe par contre des classes d'automates à pile déterminisables telles que, par exemple, les *Visibly Pushdown Automata* (abrégés en VPA) présentés par Rajeev Alur et Parthasarathy Madhusudan dans [AM04]. Malheureusement, nous ne pouvons utiliser cette classe comme modèle de spécification. En effet, l'utilisation des VPA implique que les appels et retours de procédures soient visibles, ce que nous ne pouvons pas accepter dans le cadre du test boîte noire où seules les interactions du système avec son environnement sont observables. Il n'est alors pas naturel de d'observer les appels et retours de procédures lors de ce test. Ceux-ci sont internes au système, et seront donc considérés comme des actions internes.

Ne pouvant pas déterminiser un automate à pile, nous avons décidé de ne tenir compte que des spécifications déjà déterministes. Ceci restreint la classe de spécifications à partir desquelles nous pouvons générer un cas de test. Néanmoins, cette classe nous semble déjà suffisamment expressive.

6.2.2 Suspension d'un ioPDS

Comme pour les ioLTS (voir chapitre 2, sous-section 1.2.1), l'opération de suspension sur les ioPDS permet de mettre en évidence les blocages autorisés par la spécification en ajoutant des boucles étiquetées δ dans chaque état de blocage. A la différence avec les ioLTS nous ne pourrons signaler sur les ioPDS que les blocages de sortie, et par conséquent les blocages complets. Nous ne pourrons pas signaler les blocages vivants (*livelocks*). En effet, d'un point de vue technique, si nous avions voulu détecter les blocages vivants, l'ioPDS résultant de l'opération de suspension ne serait plus déterministe : nous aurions le choix à partir de l'état de blocage vivant entre une action interne et la boucle δ. Nous avons donc défini l'opération de suspension comme suit.

Définition 6.4 (Suspension d'un ioPDS) *L'automate de suspension d'un ioPDS* $\mathcal{P} = \langle G, \Gamma, \Lambda, c_0, \hookrightarrow \rangle$ *est un ioPDS* $\mathcal{P}^\delta = \langle G, \Gamma, \Lambda^\delta, c_0, \hookrightarrow_\delta \rangle$ *avec :*

- Λ^δ *: l'alphabet* Λ *augmenté d'une nouvelle sortie* δ *(*$\Lambda^\delta = \Lambda \cup \{\delta\}$ *avec* $\delta \in \Lambda^\delta_!$*),*
- $\hookrightarrow_\delta$ *: l'ensemble des transitions obtenu à partir de* $\hookrightarrow$ *par ajout de nouvelles transitions sous forme de boucles étiquetées par* δ *dans les configurations de blocage de sortie (et de deadlock) :*

$$\forall c \in G \times \Gamma,\ \forall \lambda \in \Lambda_! \cup \Lambda_\tau,\ c \overset{\lambda}{\not\hookrightarrow} \Rightarrow\ c \overset{\delta!}{\hookrightarrow} c.$$

La sémantique d'un ioPDS suspendu est un ioLTS suspendu mais seulement par rapport aux blocages de sortie : soit $M = \langle Q^M, q_0^M, \Lambda^M, \rightarrow_M \rangle$ un ioLTS, la suspension par rapport aux blocages de sortie de celui-ci est un ioLTS $M^\delta = \langle Q^M, q_0^M, \Lambda^M \cup \{\delta\}, \rightarrow_M \cup \rightarrow_\delta \rangle$ avec $\rightarrow_\delta$ définie par : $\forall \lambda \in \Lambda^M_! \cup \Lambda^M_?, p \in Q^M, p \overset{\lambda}{\not\rightarrow_M} \Rightarrow p \overset{\delta!}{\rightarrow_\delta} p$.

Nous noterons toujours $STraces(\mathcal{P}) = Traces(\mathcal{P}^\delta)$ l'ensemble des traces de l'ioPDS déterministe suspendu.

Exemple 6.3 *La suspension de la spécification de notre exemple est représentée en figure 6.4. Cette opération consiste à ajouter des transitions dans les configurations en blocage de sortie. Sur la représentation par graphe de flot de contrôle, cela consiste à ajouter des transitions aux états f_2 et g_0 à partir desquels seules des entrées sont possibles. Attention cependant à cette représentation : il pourrait sembler naturel d'ajouter également des transitions aux états en blocage complet comme f_3 alors que nous avons à partir de ce point de contrôle un retour de fonction, donc une action interne, non représentée sur le graphe.*

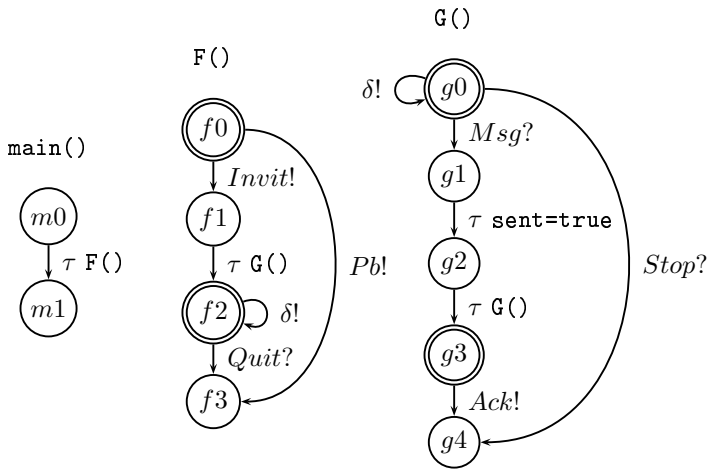


FIG. 6.4 – Spécification déterministe suspendue $\mathcal{S}^\delta$

.

6.2.3 Complétion en sortie d'un ioPDS

Comme expliqué au chapitre 2, la complétion en sortie permet de mettre en évidence sur la spécification les sorties non autorisées par celle-ci. Contrairement aux ioLTS, la complétion en sortie ne se fait pas à partir de toute configuration mais uniquement à partir de celles ayant pour sommet de pile un point de contrôle stable. La raison est la même que pour la suspension : si nous voulons garder un ioPDS déterministe, nous ne pouvons compléter en sortie une configuration à partir de laquelle une action interne est tirable.

Définition 6.5 (Complétion en sortie d'un ioPDS) *Soit un ioPDS déterministe suspendu $\mathcal{P}^\delta = \langle G, \Gamma, \Lambda^\delta, c_0, \hookrightarrow_\delta \rangle$, le complété en sortie de $\mathcal{P}^\delta$ est l'ioPDS $\Sigma^!(\mathcal{P}) = \langle G \cup \{\mathsf{Fail}\}, \Gamma, \Lambda^\delta, c_0, \hookrightarrow_{\Sigma^!(\mathcal{P}^\delta)} \rangle$ avec :*

$\hookrightarrow_{\Sigma^!(\mathcal{P}^\delta)} = \hookrightarrow_\delta \cup \{\langle g, \gamma \rangle \overset{\lambda}{\hookrightarrow}_{\Sigma^!(\mathcal{P}^\delta)} \langle \mathsf{Fail}, \gamma \rangle \mid \lambda \in \Lambda^\delta_!, \langle g, \gamma \rangle \overset{\lambda}{\not\rightarrow}_\delta \wedge \forall \tau \in \Lambda^\delta_\tau, \langle g, \gamma \rangle \overset{\tau}{\not\rightarrow}_\delta\}$.

Comme pour les ioLTS, le complété en sortie satisfait le lemme suivant :

Lemme 6.1

$$Traces(\Sigma^{!}(\mathcal{P}^{\delta}), \{Fail\}) = STraces(\mathcal{P}).\Lambda_{!}^{\delta} \setminus STraces(\mathcal{P}).$$

Exemple 6.4 *La complétion en sortie de la spécification est représentée en figure 6.5. Cette opération consiste à ajouter, à partir des PCS, des transitions de sorties non autorisées. Par exemple, depuis le PCS f_0, seules les sorties* `Invit !` *et* `Pb !` *sont autorisées par la spécification. Les autres sorties possibles (*`Ack !` *et δ!) mènent donc à la non-conformité, signalée par* Fail.

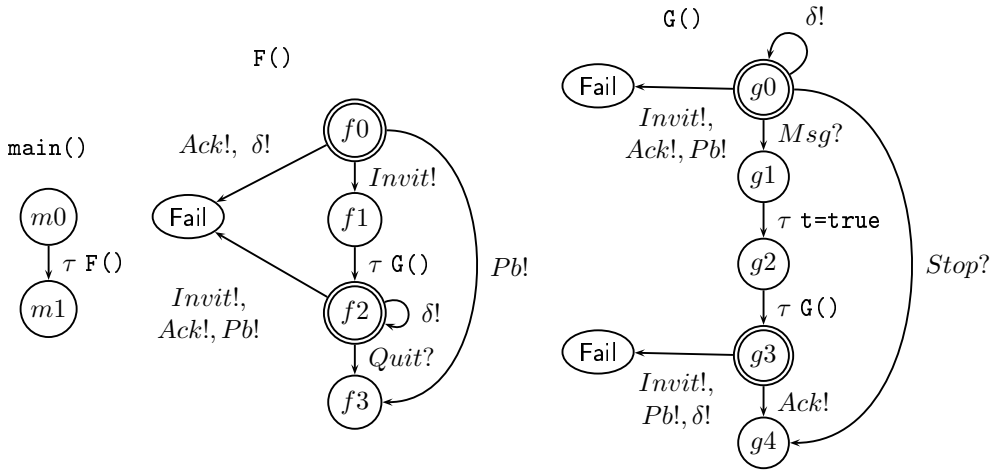


FIG. 6.5 – Spécification déterministe suspendue complétée en sortie $\Sigma^{!}(\mathcal{S}^{\delta})$

.

6.2.4 Produit avec un objectif de test

Le produit synchrone est une opération nécessaire dans notre méthode de génération de test lors de la sélection par objectif de test (voir le chapitre 2 à la section 2.2). Il est également nécessaire pour définir l'exécution du cas de test sur l'implémentation (voir le chapitre 1 à la sous-section 1.3.2).

Modélisation de l'objectif de test. Si nous choisissions de représenter un objectif de test par un ioPDS, alors le produit entre cet objectif et le testeur canonique ne donnerait pas un ioPDS puisque leurs langages ne sont pas clos par intersection. Intuitivement, il serait nécessaire que le produit résultant ait deux piles. Nous sortons alors de la classe des langages algébriques. L'objectif de test ne peut donc pas être modélisé par un ioPDS.

Si nous choisissions de représenter un objectif de test par un ioLTS, alors le produit entre cet objectif et le testeur canonique donnerait un ioPDS puisque les langages algébriques sont clos par intersection rationnelle. De plus, l'intersection d'un langage algébrique déterministe avec un langage régulier est un langage algébrique déterministe.

Nous allons donc modéliser l'objectif de test par un ioLTS. L'objectif de test aurait été plus expressif sous forme d'ioPDS, néanmoins, le test de conformité se basant sur l'ensemble des traces, le fait de modéliser l'objectif de test par un ioLTS semble raisonnable.

Définition 6.6 (Produit synchrone rationnel d'un ioPDS) *Soient un ioPDS déterministe suspendu* $\mathcal{P}^\delta = \langle G^\mathcal{P}, \Gamma, \Lambda^\delta, c_0^\mathcal{P}, \hookrightarrow_\mathcal{P} \rangle$ *et* $M = \langle Q, q_0, \Lambda_? \cup \Lambda_!^\delta, \rightarrow \rangle$ *un ioLTS déterministe. Le produit* $\mathcal{P}^\delta \times M$ *sera l'ioPDS* $\mathcal{PS} = \langle G^{\mathcal{PS}}, \Gamma, \Lambda^\delta, c_0^{\mathcal{PS}}, \hookrightarrow_{\mathcal{PS}} \rangle$ *avec :*

- $G^{\mathcal{PS}} = G^\mathcal{P} \times Q$,
- $c_0^{\mathcal{PS}} = \langle (g_0, q_0), \gamma_0 \rangle$ *avec* $c_0^\mathcal{P} = \langle g_0, \gamma_0 \rangle$,
- $\hookrightarrow_{\mathcal{PS}}$ *est le plus petit ensemble de transitions satisfaisant les règles :*

$$(1) \quad \frac{\langle g, \gamma \rangle \overset{\lambda}{\hookrightarrow}_\mathcal{P} \langle g', \gamma' \rangle \ \wedge \ q \overset{\lambda}{\rightarrow} q'}{\langle (g, q), \gamma \rangle \overset{\lambda}{\hookrightarrow}_{\mathcal{PS}} \langle (g', q'), \gamma' \rangle},$$

$$(2) \quad \frac{\langle g, \gamma \rangle \overset{\tau}{\hookrightarrow}_\mathcal{P} \langle g', \gamma' \rangle \ \wedge \ q \in Q \ \wedge \ \tau \in \Lambda_\tau}{\langle (g, q), \gamma \rangle \overset{\tau}{\hookrightarrow}_{\mathcal{PS}} \langle (g', q), \gamma' \rangle}.$$

Le principe du produit synchrone entre un ioPDS et un ioLTS est le même que pour le produit synchrone entre ioLTS ou ioSTS : il y a synchronisation sur les actions visibles tandis que les actions internes restent indépendantes. Ce produit sera illustré dans l'exemple 6.5 ci-dessous.

Le produit $\mathcal{PS}$ étant un ioPDS, sa sémantique est un ioLTS $[\![\mathcal{PS}]\!] = [\![\mathcal{P}]\!] \times M$ (produit de deux ioLTS). Nous obtenons donc la proposition suivante :

Proposition 6.1 $[\![\mathcal{P} \times M]\!] = [\![\mathcal{P}]\!] \times M$.

Preuve : Soient $\mathcal{P}^\delta = \langle G^\mathcal{P}, \Gamma, \Lambda^\delta, c_0^\mathcal{P}, \hookrightarrow_\mathcal{P} \rangle$ l'ioPDS suspendu de $\mathcal{P}$ et $M = \langle Q^\mathrm{M}, q_0^\mathrm{M}, \Lambda_? \cup \Lambda_!^\delta, \rightarrow_\mathrm{M} \rangle$ un ioLTS déterministe. Nous noterons $\mathcal{PS} = \langle G^{\mathcal{PS}}, \Gamma, \Lambda^\delta, c_0^{\mathcal{PS}}, \hookrightarrow_{\mathcal{PS}} \rangle$ le produit synchrone de $\mathcal{P}^\delta$ et M tel que défini à la définition 6.6. La sémantique de ce produit $[\![\mathcal{PS}]\!]$ est alors définie comme suit (voir définition 6.2) : $[\![\mathcal{PS}]\!] = [\![\mathcal{P}^\delta \times M]\!] = \langle Q^{\mathcal{PS}}, q_0^{\mathcal{PS}}, \Lambda^\delta, \rightarrow_{\mathcal{PS}} \rangle$ tel que

- $Q^{\mathcal{PS}} = G^{\mathcal{PS}} = G^\mathcal{P} \times Q^\mathrm{M}$;
- $q_0^{\mathcal{PS}} = c_0^{\mathcal{PS}} = \langle (g_0, q_0^\mathrm{M}), \gamma_0 \rangle$;
- $\rightarrow_{\mathcal{PS}}$ est la relation de transition définie par la règle :

  $\langle (g, q_\mathrm{M}), \gamma \rangle \overset{\lambda}{\hookrightarrow} \langle (g', q'_\mathrm{M}), \gamma' \rangle \wedge \omega \in \Gamma^* \implies \langle (g, q_\mathrm{M}), \omega \cdot \gamma \rangle \overset{\lambda}{\rightarrow} \langle (g', q'_\mathrm{M}), \omega \cdot \gamma' \rangle$, avec $q_\mathrm{M} = q'_\mathrm{M}$ si $\lambda \in \Lambda_\tau$ (par la définition du produit synchrone 6.6). Cela suppose, par la définition 6.6, que :
  - si $\lambda \in \Lambda_? \cup \Lambda_!^\delta$, alors $\langle g, \gamma \rangle \overset{\lambda}{\hookrightarrow}_\mathcal{P} \langle g', \gamma' \rangle$ et $q_\mathrm{M} \overset{\lambda}{\rightarrow}_\mathrm{M} q'_\mathrm{M}$;
  - si $\lambda \in \Lambda_\tau$, alors $\langle g, \gamma \rangle \overset{\lambda}{\hookrightarrow}_\mathcal{P} \langle g', \gamma' \rangle$ et $q_\mathrm{M} \in Q^\mathrm{M}$.

Soit $[\![\mathcal{P}^\delta]\!] = \langle Q^{\mathcal{P}}, q_0^{\mathcal{P}}, \Lambda^\delta, \rightarrow_{\mathcal{P}} \rangle$ la sémantique de $\mathcal{P}^\delta$. Le produit de cette sémantique avec l'ioLTS M est alors défini comme suit (voir définition 1.6) : $[\![\mathcal{P}^\delta]\!] \times M = \langle Q^\times, q_0^\times, \Lambda^\delta, \rightarrow_\times \rangle$ tel que

- $Q^\times = Q^{\mathcal{P}} \times Q^{\mathrm{M}} = G^{\mathcal{P}} \times Q^{\mathrm{M}}$;
- $q_0^\times = (q_0^{\mathcal{P}}, q_0^{\mathrm{M}}) = \langle (g_0, q_0^{\mathrm{M}}), \gamma_0 \rangle$ avec $q_0^{\mathcal{P}} = \langle g_0, \gamma_0 \rangle$;
- $\rightarrow_\times$ est la plus petite relation dans $Q^{\mathcal{P}} \times \Lambda^\delta \times Q^{\mathrm{M}}$ satisfaisant

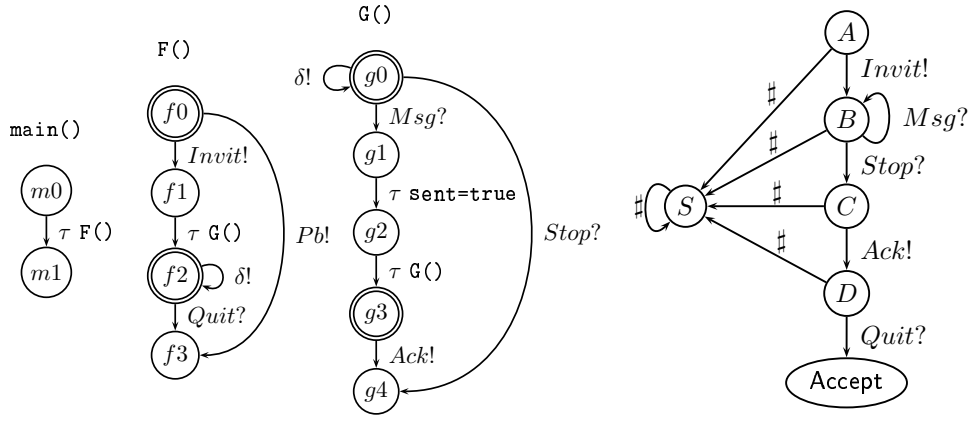
$$\langle (g, q_{\mathrm{M}}), \omega \cdot \gamma \rangle \xrightarrow{\lambda}_\times \begin{cases} \langle (g', q'_{\mathrm{M}}), \omega \cdot \gamma' \rangle \ si\ \lambda \in \Lambda_? \cup \Lambda_!^\delta \ \wedge\ \langle g, \omega \cdot \gamma \rangle \xrightarrow{\lambda}_{\mathcal{P}} \langle g', \omega \cdot \gamma' \rangle \ \wedge\ q_{\mathrm{M}} \xrightarrow{\lambda}_{\mathrm{M}} q'_{\mathrm{M}} \\ \langle (g', q_{\mathrm{M}}), \omega \cdot \gamma' \rangle \ si\ \lambda \in \Lambda_\tau \ \wedge\ \langle g, \omega \cdot \gamma \rangle \xrightarrow{\lambda}_{\mathcal{P}} \langle g', \omega \cdot \gamma' \rangle \end{cases}$$

avec $\omega \in \Gamma^*$.

Nous pouvons donc en conclure que $[\![\mathcal{PS}]\!] = [\![\mathcal{P} \times M]\!] = [\![\mathcal{P}]\!] \times M$.

Exemple 6.5 *L'objectif de test que nous voulons satisfaire sur cet exemple est représenté à la figure 6.6, à la droite de la spécification suspendue. Nous rappelons que cette spécification reconnaît les traces* $Invit \cdot \delta^* \cdot Msg^n \cdot \delta^* \cdot Stop \cdot Ack^n \cdot \delta^* \cdot Quit + Pb$ *avec* $n \geq 0$. *L'ensemble des traces reconnues par l'objectif est* $Invit \cdot Msg^* \cdot Stop \cdot Ack \cdot Quit$.

Il nous est difficile de représenter le produit entre la spécification et l'objectif de test par un graphe de flot de contrôle. Nous n'allons représenter à la figure 6.7 que certaines règles de transition de ce produit car l'ensemble des transitions est trop grand pour toutes les représenter. Nous avons décidé de ne représenter que les transitions reconnaissant l'ensemble de traces de la spécification satisfaisant l'objectif de test : $Invit \cdot Msg \cdot Stop \cdot Ack \cdot Quit$.



(a) Spécification récursive suspendue $\mathcal{S}^\delta$, (b) Objectif de test ω

FIG. 6.6 – Graphes de flot de contrôle de la spécification et de l'objectif de test.

La méthode décrite dans ce chapitre sur les ioPDS peut également être basée sur des transformations de programme, comme développé dans le chapitre 7.

$$
\begin{array}{lll}
\langle (p, ff, A), m_0 \rangle & \overset{\tau}{\hookrightarrow}_{\mathcal{PS}} & \langle (p, ff, A), m_1.f_0 \rangle \\
\langle (p, ff, A), m_1.f_0 \rangle & \overset{\text{Invit !}}{\hookrightarrow}_{\mathcal{PS}} & \langle (p, ff, B), m_1.f_1 \rangle \\
\langle (p, ff, B), m_1.f_1 \rangle & \overset{\tau}{\hookrightarrow}_{\mathcal{PS}} & \langle (p, ff, B), m_1.f_2.g_0 \rangle \\
\langle (p, ff, B), m_1.f_2.g_0 \rangle & \overset{\text{Msg ?}}{\hookrightarrow}_{\mathcal{PS}} & \langle (p, ff, B), m_1.f_2.g_1 \rangle \\
\langle (p, ff, B), m_1.f_2.g_1 \rangle & \overset{\tau}{\hookrightarrow}_{\mathcal{PS}} & \langle (p, tt, B), m_1.f_2.g_2 \rangle \\
\langle (p, tt, B), m_1.f_2.g_2 \rangle & \overset{\tau}{\hookrightarrow}_{\mathcal{PS}} & \langle (p, tt, B), m_1.f_2.g_3.g_0 \rangle \\
\langle (p, tt, B), m_1.f_2.g_3.g_0 \rangle & \overset{\text{Stop ?}}{\hookrightarrow}_{\mathcal{PS}} & \langle (p, tt, C), m_1.f_2.g_3.g_4 \rangle \\
\langle (p, tt, C), m_1.f_2.g_3.g_4 \rangle & \overset{\tau}{\hookrightarrow}_{\mathcal{PS}} & \langle (p, tt, C), m_1.f_2.g_3 \rangle \\
\langle (p, tt, C), m_1.f_2.g_3 \rangle & \overset{\text{Ack !}}{\hookrightarrow}_{\mathcal{PS}} & \langle (p, tt, D), m_1.f_2.g_4 \rangle \\
\langle (p, tt, D), m_1.f_2.g_4 \rangle & \overset{\tau}{\hookrightarrow}_{\mathcal{PS}} & \langle (p, tt, D), m_1.f_2 \rangle \\
\langle (p, tt, D), m_1.f_2 \rangle & \overset{\text{Quit ?}}{\hookrightarrow}_{\mathcal{PS}} & \langle (p, tt, \mathsf{Accept}), m_1.f_3 \rangle
\end{array}
$$

FIG. 6.7 – Transitions du produit $\mathcal{S}^{\delta} \times \omega$.

Chapitre 7

Modélisation et génération du testeur canonique par transformations de programme

Nous allons appliquer dans ce chapitre la même méthode que celle utilisée dans le chapitre précédent mais d'un point de vue transformations de programmes afin de simplifier la mise en oeuvre de la génération automatique de tests à partir de spécifications interprocédurales. Nous allons appliquer directement les transformations utiles à la génération de tests sur un programme impératif, ce qui va nous permettre de rester dans un raisonnement de programmeur. Nous pourrions également coder directement les ioPDS, mais il serait alors nécessaire, pour un outil de test, d'effectuer des traductions d'ioPDS à programme et inversement. Ici, tout est directement fait sur le programme lui-même.

Un petit langage de programmation impératif sera présenté en section 7.1 puis la génération de tests basée sur ce langage sera expliquée à la section 7.2.

7.1 Modélisation par un langage de programmation

La syntaxe et la sémantique du langage que nous avons déjà vu à l'exemple de l'introduction de cette partie est inspiré de BEBOP [BR00], un langage d'entrée de l'outil MOPED. MOPED est un *model-checker* pour logiques temporelles sur des systèmes à pile [ES01].

7.1.1 Petit langage de programmation

BEBOP utilise une syntaxe classique de langage impératif. Nous supposerons, pour plus de simplicité, que nous n'aurons comme structures de contrôle que les conditionnelles (*if*) avec saut (*goto*). De plus, nous supposerons que les paramètres des procédures seront remplacés par l'utilisation de variables globales (énumérées ou booléennes). Les caractéristiques principales de cette syntaxe sont données à la

figure 7.1.

Les principales particularités de cette syntaxe par rapport à BEBOP sont les instructions de communication et l'opérateur de choix non-déterministe entre ces communications (nous pouvons avoir le choix entre deux sorties par exemple). Ces instructions d'entrée et de sortie utilisent une variable globale particulière p contenant le message de cette entrée/sortie. Cette variable ne pourra être utilisée que dans la condition et le bloc associés à ces instructions. Nous supposerons que les émissions et réceptions ne pourront pas s'imbriquer les unes les autres.

L'opérateur de choix non-déterministe est l'opérateur []. Il ne peut être utilisé que pour les instructions de communication. Le non-déterminisme est permis dans ce cas car celui-ci est observable : à chaque trace de ce programme correspond une unique exécution.

Séquences	*block*	::=	$\epsilon \mid instr; block$
Instructions	*instr*	::=	$atom \mid callret \mid com$
Instructions atomiques	*atom*	::=	$var = expr \mid if\ (expr)$ goto *label*
Instructions interprocédurales	*callret*	::=	$proc()$ | return
Communications	*com*	::=	emit(p) when *expr* $\{block\}$
		|	receive(p) when *expr* $\{block\}$
		|	com [] com
Expressions	*expr*		

FIG. 7.1 – Syntaxe du langage.

Exemple 7.1 *L'exemple vu au chapitre précédent à la figure 6.1 est ici représenté à la figure 7.2 sous forme de programme.*

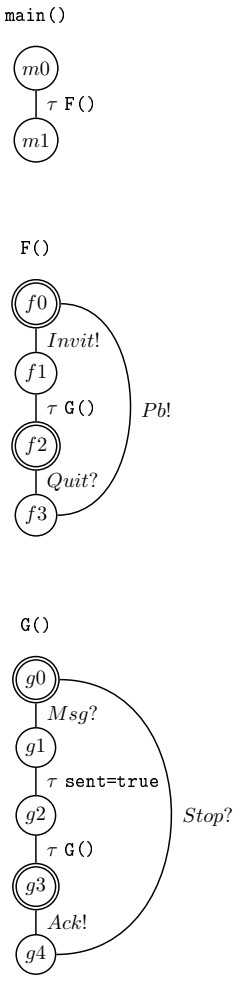
7.1.2 Sa sémantique en termes d'ioPDS

La sémantique de ce langage est définie sur les domaines suivants :

Point de contrôle	$k \in K$
Environnement global	$g \in GEnv = GVar \rightarrow Val$
Environnement local	$l \in LEnv = LVar \rightarrow Val$
Configuration	$(g, \sigma) \in C = GEnv \times (K \times LEnv)^+$

Nous supposerons que la variable spéciale p prend ses valeurs dans l'alphabet Λ. La sémantique est alors définie comme un ioPDS $\mathcal{P} = \langle G, \Gamma, \Lambda, c_0, \hookrightarrow \rangle$ avec :

- $G = GEnv$, les états de $\mathcal{P}$ sont constitués des valeurs des variables globales du programme,
- $\Gamma = K \times LEnv$, l'alphabet de pile de $\mathcal{P}$ est constitué des points de contrôle du programme et des valeurs des variables locales associées à chacun de ces points,




```
enum out_t { invit, ack, pb };
enum inp_t { stop, msg, quit };


bool t = false;

void main(){
  f();
}

void f()
{
  f0: emit(p) when (p == invit || p == pb){


        if (p==pb) goto f3;
      }
  f1: g();
  f2: receive(p) when (p == quit) {};



  f3: return;
}

void g()
{
  g0: receive(p) when (p == stop || p == msg){
        if (p == stop) goto g4;
      };



  g1: sent = true;
  g2: g();
  g3: emit(p) when (p == ack) {}


  g4: return;
}
```

FIG. 7.2 – Spécification correspondant à la figure 6.1.

- $c_0 = (g_0, (k_0, l_0))$ où $g_0 \in G$ et $(k_0, l_0) \in \Gamma$,
- $\hookrightarrow$ la relation de transition définie par les règles d'inférence qui suivent.

Nous utiliserons les transitions du graphe de flot de contrôle associé au programme pour représenter les instructions à chaque point de contrôle. Nous ne préciserons pas ici les règles d'inférence standard (pour cela, voir [ES01]), nous nous focaliserons sur la sémantique des instructions d'émission et de réception ainsi que sur les instructions atomiques, d'appel et de retour de fonction.

- Une instruction atomique génère une règle de la forme :

$$\frac{k \xrightarrow{atom} k'}{\langle g, (k, l)\rangle \overset{\tau}{\hookrightarrow} \langle g', (k', l')\rangle}$$

  avec une condition sur (g, l) dans le cas d'une instruction conditionnelle avec saut. g' et l' sont les valeurs respectives des variables globales et locales après une affectation.

- L'appel d'une procédure sans passage de paramètre explicite génère une règle de la forme :

$$\frac{k \xrightarrow{proc()} k'}{\langle g, (k, l)\rangle \overset{\tau}{\hookrightarrow} \langle g, (k', l) \cdot (s_{proc}, l'_0)\rangle}$$

  pour laquelle s_{proc} est le point de contrôle initial de la fonction appelée. Un nouveau point de départ d'une fonction est donc empilé avec l'environnement local initial de cette fonction l'_0. Les variables locales sont indéterminées au point d'entrée des fonctions appelées.

- La procédure de retour génère quant à elle une règle

$$\frac{k \xrightarrow{proc()} k' \qquad e_{proc} \xrightarrow{return} \ldots}{\langle g, (k', l') \cdot (e_{proc}, l)\rangle \overset{\tau}{\hookrightarrow} \langle g, (k', l')\rangle}$$

  où l'environnement de la fonction est dépilé. Le contrôle revient donc à la fonction appelante.

- Une instruction d'émission génère une règle de la forme :

$$\frac{k \xrightarrow{\mathsf{emit}(p)\ \mathsf{when}\ expr\ \{k':block\}} k''}{\langle g, (k, l)\rangle \overset{p}{\hookrightarrow} \langle g', (k', l)\rangle} \text{ if } [\![expr]\!](g', l) = \mathsf{true}$$

$$\frac{k \xrightarrow{\mathsf{emit}(p)\ \mathsf{when}\ expr\ \{k':block\}} k''}{\langle g, (k, l)\rangle \overset{\tau}{\hookrightarrow} \langle g', (k'', l)\rangle} \text{ if } [\![expr]\!](g', l) = \mathsf{false}$$

  avec $\forall v \neq p : g'(v) = g(v)$. La valeur précédente de la variable p est oubliée dès qu'on est dans l'environnement g' puisque sa portée est limitée à la condition et au

bloc associés à l'émission. Si l'environnement courant (g, l) satisfait la condition, p est émis et le contrôle passe donc au début du bloc, c'est-à-dire au point de contrôle k'. Si la condition n'est pas satisfaite, le contrôle passe directement à k''. Nous remarquons que nous avons ici un choix non-déterministe observable : l'instruction permet d'émettre n'importe quel message p qui satisfait la condition.

– La sémantique d'une instruction de réception est identique à celle de l'émission. Les émissions et les réceptions n'ont en effet besoin d'être distingués qu'au niveau de la relation de conformité.

Toutes les instructions, exceptées les émissions et réceptions génèrent des actions internes étiquetées par τ. Les points de contrôle stables du programme sont définis comme des points de contrôle au départ d'instructions de communication. Ils sont les seuls points à partir desquels des messages peuvent être échangés. De tels points de contrôle stables peuvent être séparés par un ou une séquence de points de contrôle ordinaires, reliés par des transitions d'actions internes.

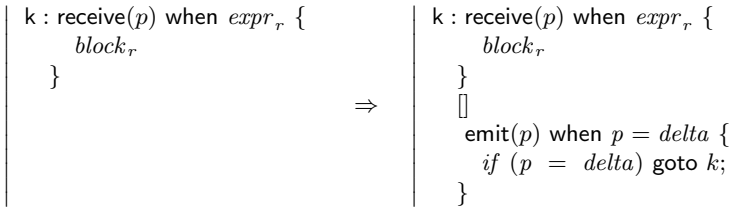
7.2 Génération de tests en termes de langage de programmation

Après avoir présenté les transformations faites sur le programme de la spécification pour obtenir le testeur canonique (sous-section 7.2.1), nous décrirons le produit de ce testeur avec l'objectif de test (sous-section 7.2.2).

7.2.1 Spécification interprocédurale et testeur canonique

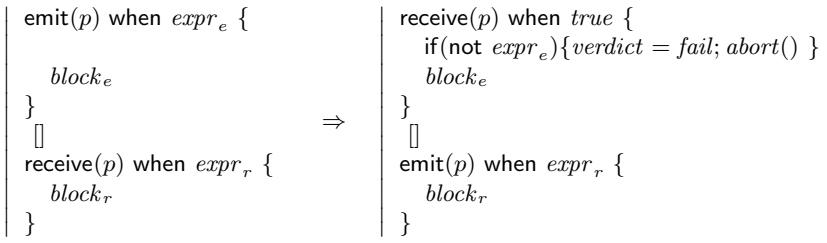
Une spécification interprocédurale $\mathcal{S}$ est un programme défini avec le langage vu précédemment dont la syntaxe est présentée à la figure 7.1. Cette syntaxe, et donc cette spécification, est déterministe : à une trace correspond une unique exécution finissant dans un point de contrôle stable. Un non-déterminisme observable est donc permis. On peut avoir le choix entre deux émissions et/ou réceptions, mais pas entre deux instructions internes (générant des transitions d'actions internes). Cette notion de déterminisme permet de construire facilement le testeur canonique de la spécification. Nous allons présenter les différentes transformations de langage dues à la génération de tests via un motif général au niveau des points de contrôle stables.

Afin de signaler les blocages de sortie autorisés par la spécification, nous allons émettre un δ dans les points de contrôle stables à partir desquels aucune émission n'est réalisée. Cela est produit par la transformation de programme suivante :



Cette opération mime la suspension définie sur les ioPDS au chapitre précédent à la section 6.2.2.

Le testeur canonique est ensuite obtenu par complétion en sortie aux PCS et l'opération miroir (inversion des entrées/sorties) dont la raison est expliquée dans la partie I (définition du cas de test 1.3.1). Ceci implique les transformations de programme suivantes :



Cette opération mime la complétion en sortie définie sur les ioPDS au chapitre précédent à la section 6.2.3 mais effectue également l'opération miroir.

Exemple 7.2 *Nous rappelons à la figure 7.3 le programme représentant la spécification $\mathcal{S}$. Son testeur canonique, obtenu après suspension, complétion en sortie puis miroir, est décrit à la figure 7.4. Un nouveau type et une variable globale* verdict *ont été introduits afin de stocker le verdict émis.*

7.2.2 Objectif de test

Comme expliqué au chapitre précédent à la sous-section 6.2.4, l'objectif de test est modélisé par un ioLTS. Nous allons implémenter cet ioLTS TP par une procédure TP(p) prenant en entrée le dernier message échangé. Un objectif de test est illustré à l'exemple 7.3.

Le produit consiste en une procédure appelant une autre procédure en chaque point de contrôle stable. Le produit entre le testeur canonique et cet objectif de test est obtenu par la transformation de programme suivante :

```
receive(p) when true {                    receive(p) when true{
  if(not expr_r)                            if(not expr_r)
    { verdict = fail; abort() }               { verdict = fail; abort() }
                                            TP(p);
  block_r                                   block_r
}                                 ⇒       }
 []                                         []
emit(p) when expr_e {                     emit(p) when expr_e {
                                            TP(p);
  block_e                                   block_e
}                                         }
```

L'appel à TP(p) est effectué après avoir vérifié la conformité puisque les traces acceptées doivent être conformes. L'émission du verdict Pass est réalisée par la procédure TP (la variable verdict prend alors la valeur pass), comme illustré dans l'exemple qui suit.

Exemple 7.3 *L'objectif de test décrit à la figure 6.6 est ici représenté à la figure 7.5 par un petit programme. La variable globale* pc *permet de se situer dans les états de l'objectif.*

Le produit de l'objectif avec le testeur canonique de la figure 7.4 est représenté à la figure 7.6. Les seules modifications apportées ici sont les appels à TP(p). *Des points de contrôle ont été ajoutés afin de simplifier les explications de la sélection dans l'exemple 8.1.*

```
enum out_t { invit, ack, pb };
enum inp_t { stop, msg, quit };


bool t = false;

void main(){
  f();
}

void f()
{
  f0: emit(p) when (p == invit || p == pb){


        if (p==pb) goto f3;
      }
  f1: g();
  f2: receive(p) when (p == quit) {};









  f3: return;
}

void g()
{
  g0: receive(p) when (p == stop || p == msg){
        if (p == stop) goto g4;
      };









  g1: sent = true;
  g2: g();
  g3: emit(p) when (p == ack) {}


  g4: return;
}
```

FIG. 7.3 – Spécification correspondant à la figure 6.1.

```
enum out_t { stop, msg, quit };
enum inp_t { invit, ack, pb, delta };
enum verdict_t { none, fail, pass, inconc };
enum verdict_t verdict = none;
bool t = false;

void main(){
  f();
}

void f()
{
  f0: receive (p) when true {
        if (p != invit && p != pb)
          { verdict = fail; abort(); }
        if (p == pb) goto f3;
      };
  f1: g();
  f2: emit(p) when (p == quit) {}
      []
      receive(p) when true {
        if (p != delta)
          { verdict = fail; abort() };
        if (p = delta) goto f2;
      };
  f3: return;
}

void g()
{
  g0: emit(p) when (p == stop || p == msg){
        if (p == stop) goto g4;
      }
      []
      receive(p) when true {
        if (p != delta)
          { verdict = fail; abort() };
        if (p = delta) goto g0;
      };
  g1: sent = true;
  g2: g();
  g3: receive(p) when true {
        if (p != ack){ verdict = fail; abort(); }
      }
  g4: return;
}
```

FIG. 7.4 – Testeur canonique associé à la spécification de la figure 7.3

```
enum pc_t { A,B,C,D,E,S };
enum pc_t pc = A;

void TP(enum msg_t p)
{
  if    ( pc == A && p == invit ) pc = B;
  elsif ( pc == B && p == msg )  pc = B;
  elsif ( pc == B && p == stop )  pc = C;
  elsif ( pc == C && p == ack )  pc = D;
  elsif ( pc == D && p == quit ){
      pc = E;
      verdict = pass;
      abort();
  }
  else pc = S;
}
```

FIG. 7.5 – Objectif de test correspondant à la figure 6.6.(b)

```
// Type and global variables Declarations
// ...

void main(){
  m0: f();
  m1:
}

void f()
{
  f0: receive (p) when true {
  f0r:  if (p != invit && p != pb)
          { verdict = fail; abort(); }
        TP(p);
        if (p == pb) goto f3;
      };
  f1: g();
  f2: emit(p) when (p == quit) {
  f2e:  TP(p)
      }
      []
      receive(p) when true {
  f2r:  if (p != delta)
          { verdict = fail; abort() };
        if (p = delta) goto f2;
      };
  f3: return;
}

void g()
{
  g0: emit(p) when (p == stop || p == msg){
  g0e:  TP(p);
        if (p == stop) goto g4;
      }
      []
      receive(p) when true  {
   g0r: if (p != delta)
          { verdict = fail; abort() };
        if (p = delta) goto g0;
      };
  g1: sent = true;
  g2: g();
  g3: receive(p) when true {
  g3r:  if (p != ack)
          { verdict = fail; abort(); }
        TP(p);
      }
  g4: return;
}
```

FIG. 7.6 – Produit

Chapitre 8

Sélection des tests sur le testeur canonique récursif

La sélection de test est ici basée sur les mêmes principes que pour les ioLTS (voir le chapitre 2 à la sous-section 2.2.3). Nous exploiterons en particulier la relation

$$pref_{\leq}(Traces(\mathcal{P}, \mathsf{Pass})) = Traces(\mathcal{P}, coreach(\mathsf{Pass}))$$

afin de reconnaître les traces (conformes) qui peuvent être acceptées par l'objectif de test dans le futur. Cependant, l'incapacité des cas de test à connaître l'intégralité du contenu de leur pile ne permet pas d'utiliser la totalité de l'information recueillie par l'analyse de co-accessibilité depuis Pass. Nous analyserons ce problème d'observation partielle dans la suite de ce chapitre.

8.1 Analyse de co-accessibilité

Nous rappelons que dans un ioPDS (en l'occurrence, celui généré par la sémantique de notre langage de programmation), une configuration est une paire $(g, \gamma) \in C$ composée d'un environnement global et d'une pile d'appels. L'ensemble des configurations correspondant au verdict *Pass* est l'ensemble $\mathsf{Pass} = \{(g, \gamma) \mid g(\mathsf{verdict}) = \mathsf{pass})\}$. Nous noterons par la suite l'ensemble des configurations co-accessibles depuis Pass : $coreach = \{c \in C \mid \exists c' \in \mathsf{Pass} : c \rightarrow^* c'\}$.

Nous allons utiliser certaines propriétés des ioPDS afin de calculer l'ensemble *coreach*. Ces propriétés justifient en partie le choix de ce modèle comme sémantique du langage utilisé et sa restriction à des variables de domaine fini.

Soient un ioPDS $\mathcal{P} = \langle G, \Gamma, \Lambda, c_0, \hookrightarrow \rangle$ et un ensemble de configurations $X \in \wp(G \times \Gamma^*) = G \rightarrow \wp(\Gamma^*)$. Cet ensemble X est régulier s'il associe un langage régulier à chaque état global. Il y a deux résultats importants à ceci. Le premier est qu'ainsi, l'ensemble des configurations co-accessibles (et respectivement accessibles) d'un ioPDS est régulier si l'ensemble final (respectivement initial) des configurations est régulier [Cau92]. Le second est que dans ce cas, l'ensemble des configurations

co-accessibles (respectivement accessibles) est calculable avec une complexité polynomiale [FWW97, BEM97]. L'outil MOPED implémente des algorithmes symboliques efficaces pour calculer ces ensembles, grâce à l'utilisation d'un modèle symbolique d'automate à pile où la relation de transition $\hookrightarrow$ est représentée par des BDDs [ES01]. L'ensemble Pass étant régulier, nous pouvons utiliser MOPED sur l'ioPDS généré par notre programme récursif afin d'obtenir l'ensemble régulier des configurations co-accessibles. Voici un exemple du résultat de ce calcul.

Exemple 8.1 *Reprenons le produit entre spécification et objectif de test de la figure 7.6. Nous avons indiqué dans le tableau de la figure 8.1(a), pour chaque point de contrôle, les configurations à partir desquelles nous pouvons atteindre la configuration finale* $\langle(-, \mathsf{Pass}, E), \omega\rangle$. *Les piles ne contiennent dans cet exemple que des points de contrôles puisqu'il n'y a aucune variable locale. Nous pouvons par exemple lire que* Pass *est accessible depuis* $g3_r$ *pour les configurations* $\langle(\mathit{ff}, -, C, ack), \omega.f_2 g_{3r}\rangle$ *et* $\langle(\mathit{tt}, -, C, ack), \omega.f_2 g_{3r}\rangle$ *: la valeur de la variable* `sent` *peut être à vrai ou à faux,* pc *vaut* C, p *vaut ack et le contenu de la pile reflète le fait que la fonction* `G()` *n'a été appelée qu'une seule fois. Si la fonction* `G()` *avait été appelée plusieurs fois, l'objectif ne serait pas atteint.*

8.2 Problème de l'observation partielle

La sélection consiste à contraindre les comportements du programme P en ajoutant des conditions au tir de transitions grâce aux informations calculées par l'analyse de co-accessibilité (voir la sélection sur les ioSTS à la section 4.2.2). Ces conditions portent sur les configurations, donc sur l'état (les valeurs des variables gobales) et le contenu de la pile. Elles vont permettre de sélectionner les sorties à émettre et à détecter les entrées qui feront quitter à P l'ensemble des traces acceptées par l'objectif de test. De plus, la sélection va permettre de détecter si l'objectif ne peut plus être satisfait (le verdict **Inconc** sera alors émis). Cependant, dans un langage impératif standard comme le nôtre, un programme ne peut observer que le sommet de la pile : il sait à quel point de contrôle il se trouve mais ignore par quels points il est passé pour y arriver. Or, il est nécessaire de connaître l'intégralité de la pile pour décider si la configuration courante est ou non co-accessible.

Définissons la fonction d'observation partielle α, qui à une configuration associe un tuple composé des valeurs des variables globales et du sommet de pile (un point de contrôle et les valeurs des variables locales à celui-ci) de cette configuration (le contenu de la pile, hormis le sommet est donc perdu) :

$$\begin{array}{rcl} C & \to & GEnv \times K \times LEnv \\ (g, \omega \cdot (k, l)) & \mapsto & (g, k, l) \end{array} .$$

Cette fonction est étendue aux ensembles de configurations. Définissons également $\gamma = \alpha^{-1}$ la fonction inverse correspondante. (α, γ) forme alors une connexion de Galois

Point de contrôle	Etats co-accessibles depuis $\langle(-, \mathsf{pass}, E), \omega\rangle$
m_0	$\langle(\mathit{ff}, -, A), \omega.m_0\rangle$ $\langle(\mathit{tt}, -, A), \omega.m_0\rangle$
m_1	$\langle(-, \mathsf{pass}, E), \omega.m_1\rangle$
f_0	$\langle(\mathit{ff}, -, A), \omega.f_0\rangle$ $\langle(\mathit{tt}, -, A), \omega.f_0\rangle$
f_{0r}	$\langle(\mathit{ff}, -, A, \mathit{invit}), \omega.f_{0r}\rangle$ $\langle(\mathit{tt}, -, A, \mathit{invit}), \omega.f_{0r}\rangle$
f_1	$\langle(\mathit{ff}, -, B), \omega.f_1\rangle$ $\langle(\mathit{tt}, -, B), \omega.f_1\rangle$
f_2	$\langle(\mathit{ff}, -, D), \omega.f_2\rangle$ $\langle(\mathit{tt}, -, D), \omega.f_2\rangle$
f_{2e}	$\langle(\mathit{ff}, -, D, \mathit{quit}), \omega.f_{2e}\rangle$ $\langle(\mathit{tt}, -, D, \mathit{quit}), \omega.f_{2e}\rangle$
f_{2r}	$\bot$
f_3	$\langle(-, \mathsf{pass}, E), \omega.f_3\rangle$
g_0	$\langle(\mathit{ff}, -, B), \omega.(f_2g_0 + f_2g_3g_0\rangle$ $\langle(\mathit{tt}, -, B), \omega.(f_2g_0 + f_2g_3g_0\rangle$
g_{0e}	$\langle(\mathit{ff}, -, B, \mathit{stop}), \omega.f_2g_3g_{0e}\rangle$ $\langle(\mathit{ff}, -, B, \mathit{msg}), \omega.f_2g_{0e}\rangle$ $\langle(\mathit{tt}, -, B, \mathit{stop}), \omega.f_2g_3g_{0e}\rangle$ $\langle(\mathit{tt}, -, B, \mathit{msg}), \omega.f_2g_{0e}\rangle$
g_{0r}	$\bot$
g_1	$\langle(\mathit{ff}, -, B), \omega.f_2g_1\rangle$ $\langle(\mathit{tt}, -, B), \omega.f_2g_1\rangle$
g_2	$\langle(\mathit{ff}, -, B), \omega.f_2g_2\rangle$ $\langle(\mathit{tt}, -, B), \omega.f_2g_2\rangle$
g_3	$\langle(\mathit{ff}, -, C), \omega.f_2g_3\rangle$ $\langle(\mathit{tt}, -, C), \omega.f_2g_3\rangle$
g_{3r}	$\langle(\mathit{ff}, -, C, \mathit{ack}), \omega.f_2g_{3r}\rangle$ $\langle(\mathit{tt}, -, C, \mathit{ack}), \omega.f_2g_{3r}\rangle$
g_4	$\langle(\mathit{ff}, -, C), \omega.f_2g_3g_4\rangle$ $\langle(\mathit{ff}, -, D), \omega.f_2g_4\rangle$ $\langle(\mathit{tt}, -, C), \omega.f_2g_3g_4\rangle$ $\langle(\mathit{tt}, -, D), \omega.f_2g_4\rangle$

(a) États co-accessibles

Point de contrôle	Etats accessibles depuis $\langle(\mathit{ff}, \mathsf{none}, A), m_0\rangle$
f_{0r}	$\langle(\mathit{ff}, \mathsf{none}, A, \mathit{invit}), m_1f_{0r}\rangle$ $\langle(\mathit{ff}, \mathsf{none}, A, \mathit{pb}), m_1f_{0r}\rangle$
f_{2e}	$\langle(\mathit{ff}, \mathsf{none}, C, \mathit{quit}), m_1f_{2e}\rangle$ $\langle(\mathit{tt}, \mathsf{none}, D, \mathit{quit}), m_1f_{2e}\rangle$ $\langle(\mathit{tt}, \mathsf{none}, S, \mathit{quit}), m_1f_{2e}\rangle$
f_{2r}	$\langle(\mathit{ff}, \mathsf{none}, C, \mathit{delta}), m_1f_{2r}\rangle$ $\langle(\mathit{tt}, \mathsf{none}, D, \mathit{delta}), m_1f_{2r}\rangle$ $\langle(\mathit{tt}, \mathsf{none}, S, \mathit{delta}), m_1f_{2r}\rangle$
g_{0e}	$\langle(\mathit{ff}, \mathsf{none}, B, \mathit{stop}), m_1f_2g_{0e}\rangle$ $\langle(\mathit{ff}, \mathsf{none}, B, \mathit{msg}), m_1f_2g_{0e}\rangle$ $\langle(\mathit{tt}, \mathsf{none}, B, \mathit{stop}), m_1f_2g_3^+g_{0e}\rangle$ $\langle(\mathit{tt}, \mathsf{none}, B, \mathit{msg}), m_1f_2g_3^+g_{0e}\rangle$
g_{0r}	$\langle(\mathit{ff}, \mathsf{none}, B, \mathit{delta}), m_1f_2g_{0r}\rangle$ $\langle(\mathit{tt}, \mathsf{none}, B, \mathit{delta}), m_1f_2g_3^+g_{0r}\rangle$
g_{3r}	$\langle(\mathit{tt}, \mathsf{none}, C, \mathit{ack}), m_1f_2g_{3r}^+\rangle$ $\langle(\mathit{tt}, \mathsf{none}, D, \mathit{ack}), m_1f_2g_{3r}^+\rangle$ $\langle(\mathit{tt}, \mathsf{none}, S, \mathit{ack}), m_1f_2g_{3r}^+\rangle$

(b) États accessibles depuis les PCS

Point de contrôle	Intersection des états co-accessibles et accessibles
f_{0r}	$\langle(\mathit{ff}, \mathsf{none}, A, \mathit{invit},), m_1f_{0r}\rangle$
f_{2e}	$\langle(\mathit{ff}, \mathsf{none}, C, \mathit{quit}), m_1f_{2e}\rangle$
f_{2r}	$\bot$
g_{0e}	$\langle(\mathit{tt}, \mathsf{none}, B, \mathit{stop}), m_1f_2g_3g_{0e}\rangle$ $\langle(\mathit{ff}, \mathsf{none}, B, \mathit{msg}), m_1f_2g_{0e}\rangle$
g_{0r}	$\bot$
g_{3r}	$\langle(\mathit{tt}, \mathsf{none}, C, \mathit{ack}), m_1f_2g_{3r}\rangle$

(c) Intersection entre les états co-accessibles et accessibles

FIG. 8.1 – Analyse du programme de la figure 7.6. *Les configurations sont composées des valeurs des variables globales* $(t, \mathsf{verdict}, \mathsf{pc}, p)$ *et du contenu de la pile (*$-$ *signifie "n'importe quelle valeur", et* $\omega = K^*$*). Comme la variable globale* p *n'est "active" qu'aux points de contrôles stables, sa valeur n'est précisée nulle part ailleurs.*

telle qu'utilisée dans l'interprétation abstraite [CC77] : la fonction α est une approximation et γ une concrétisation. A chaque point de contrôle k du programme, nous avons un ensemble de configurations X tel que $X(k) = \{c \in X \mid c = (g, \omega \cdot (k, l))\}$ soit sa projection sur le point de contrôle k. Le programme ne peut alors décider que de l'inclusion des valeurs des variables (g, l) dans l'approximation $\alpha(X(k)) = \{(g, k, l) \mid (g, \omega \cdot (k, l)) \in X(k)\}$. En d'autres termes, le programme peut connaître l'ensemble des valeurs que peuvent prendre les variables globales et locales à un point de contrôle k, quelque soit le chemin permettant d'atteindre k. $\alpha(X(k))$ est donc une sur-approximation des configurations possibles à un point de contrôle. En notant $\overline{coreach}$ le complémentaire de *coreach*, c'est-à-dire l'ensemble des traces ne pouvant atteindre Pass, nous pouvons être dans le cas où :

$$\gamma \circ \alpha(coreach(k)) \cap \gamma \circ \alpha(\overline{coreach}(k)) \neq \emptyset. \tag{8.1}$$

En d'autres termes, nous ne pouvons pas décider, en n'observant que le sommet de pile, si la configuration est co-accessible depuis Pass ou pas. Ceci est illustré par l'exemple 8.2.

Exemple 8.2 *Prenons par exemple l'ensemble des configurations co-accessibles au point de contrôle* $g0_e$ *:*

g_{0e}	$\langle(ff, -, B, stop), \omega.f_2 g_3 g_{0e}\rangle$
	$\langle(ff, -, B, msg), \omega.f_2 g_{0e}\rangle$
	$\langle(tt, -, B, stop), \omega.f_2 g_3 g_{0e}\rangle$
	$\langle(tt, -, B, msg), \omega.f_2 g_{0e}\rangle$

Nous constatons que nous pouvons alors distinguer si le programme doit émettre stop ou msg suivant la fonction appelante de `G()` *: d'après le contenu de la pile, msg sera émis si la fonction* `G()` *est appelée depuis* `F()` *tandis que stop sera émis s'il y a eu un appel récursif à* `G()`*. Or, à cause de l'observation partielle, le programme ne peut pas connaître l'intégralité du contenu de la pile, mais seulement son sommet (il sait où il se situe mais n'a pas de mémoire du chemin parcouru pour y arriver). En ne tenant compte que du sommet de pile, nous obtenons l'information suivante :*

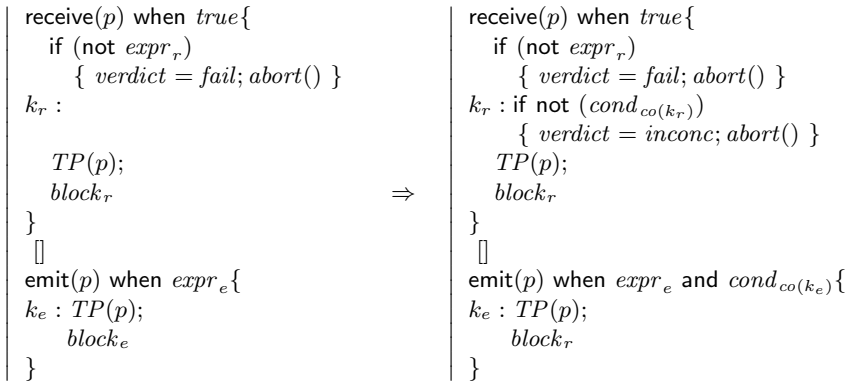
g_{0e}	$\langle(ff, -, B, stop), \omega.g_{0e}\rangle$
	$\langle(ff, -, B, msg), \omega.g_{0e}\rangle$
	$\langle(tt, -, B, stop), \omega.g_{0e}\rangle$
	$\langle(tt, -, B, msg), \omega.g_{0e}\rangle$

Nous ne pouvons alors pas anticiper quelle sortie émettre à ce point de contrôle pour satisfaire l'objectif de test.

8.3 Règles de sélection

Comme dans le cas des ioSTS, la sélection consiste d'une part à renforcer les gardes et d'autre part à ajouter des tests pour l'émission du verdict **Inconc**. Nous noterons $cond_{co(k)}(g, l)$ ces conditions induites de la sur-approximation de l'analyse

de co-accessibilité au point k $\alpha(coreach(k))$. Nous transformons alors le programme comme suit :



Lors de la réception de messages au point de contrôle k_r, après avoir vérifié la conformité, $\neg cond_{co(k_e)}$ est une condition suffisante pour sortir des préfixes des traces acceptées par Pass ($\gamma(\neg cond_{co(k)}) \subseteq \overline{coreach}(k)$). Si cette condition est satisfaite, alors le verdict *Inconc* est émis.

Lors de l'émission de messages, $cond_{co(k)}$ est une condition nécessaire pour rester dans les préfixes des traces acceptées ($\gamma(cond_{co(k)}) \supseteq coreach(k)$). Cette condition n'est pas suffisante puisque $\alpha(coreach(k))$ calcule une sur-approximation des configurations co-accessibles. Ainsi, une configuration appartenant à cette sur-approximation peut ne pas mener à Pass. Par contre, une configuration n'appartenant pas à cette sur-approximation ne peut absolument pas mener à Pass.

Le programme obtenu est un cas de test non-biaisé. En effet, le testeur canonique est non-biaisé (conséquence du lemme 6.1). Le produit et la sélection n'ajoutant aucun cas de détection de la non-conformité, ils conservent la propriété de non-biais du testeur canonique. Il existe une forte similarité entre cet algorithme de sélection et celui défini au chapitre 4 (sous-section 4.2.2) sur les systèmes symboliques avec variables à domaine infini. L'observation partielle ne nous permet pas ici d'avoir une sélection optimale, tandis que pour les ioSTS, il nous est impossible de calculer un ensemble exact d'états co-accessibles, ce qui ce traduit également en une sur-approximation.

8.4 Amélioration de la sélection grâce à l'analyse d'accessibilité

Nous pouvons améliorer l'algorithme de sélection grâce à l'analyse d'accessibilité. Notons par *reach* l'ensemble des configurations accessibles du programme P. Nous savons qu'au point de contrôle k, la configuration courante est forcément incluse dans $reach(k)$. Nous pouvons alors tester si la configuration fait partie de la sur-approximation

de l'ensemble des configurations co-accessibles et accessibles en testant son inclusion dans $\gamma \circ \alpha(\mathit{reach}(k) \cap \mathit{coreach}(k))$ plutôt que dans $\gamma \circ \alpha(\mathit{coreach}(k))$. Ce test est possible car puisque les ensembles des configurations accessibles et co-accessibles sont des ensembles réguliers, leur intersection l'est également. Nous avons toujours le cas problématique de la sur-approximation. L'équation (8.1) devient donc :

$$\gamma \circ \alpha(\mathit{reach}(k) \cap \mathit{coreach}(k)) \cap \gamma \circ \alpha(\mathit{reach}(k) \cap \overline{\mathit{coreach}}(k)) \neq \emptyset. \tag{8.2}$$

Il est clair que l'équation (8.2) implique l'équation (8.1) mais l'implication inverse est fausse. Désormais, $\mathit{cond}_{co(k_e)}$ désignera les conditions calculées par $\alpha(\mathit{reach}(k) \cap \mathit{coreach}(k))$. Comme illustré à l'exemple 8.3, l'utilisation de l'information d'accessibilité permet une sur-approximation plus précise, mais cela ne garantit pas une sélection optimale.

Exemple 8.3 *Le calcul des configurations accessibles depuis la configuration initiale pour les points de contrôle stables est représenté à la figure 8.1(b). Nous remarquons en* $g0_e$ *les configurations suivantes :*

g_{0e}	$\langle(\mathit{ff}, \mathsf{none}, B, \mathit{stop}), m_1 f_2 g_{0e}\rangle$
	$\langle(\mathit{ff}, \mathsf{none}, B, \mathit{msg}), m_1 f_2 g_{0e}\rangle$
	$\langle(\mathit{tt}, \mathsf{none}, B, \mathit{stop}), m_1 f_2 g_3^+ g_{0e}\rangle$
	$\langle(\mathit{tt}, \mathsf{none}, B, \mathit{msg}), m_1 f_2 g_3^+ g_{0e}\rangle$

Nous constatons, là encore, que nous ne pouvons pas anticiper quelle sortie émettre puisqu'à partir d'une même configuration, la variable globale p *peut prendre soit la valeur* stop*, soit la valeur* msg *(et ce, même en connaissant ici l'intégralité de la pile). Nous allons donc recouper les informations de l'analyse de co-accessibilité avec celles de l'analyse d'accessibilité.*

L'intersection entre l'ensemble des configurations accessibles et l'ensemble des configurations co-accessibles est représentée à la figure 8.1(c). Nous constatons que nous pouvons alors distinguer en $g0_e$ *quelle sortie émettre suivant la configuration :*

g_{0e}	$\langle(\mathit{tt}, \mathsf{none}, B, \mathit{stop}), m_1 f_2 g_3 g_{0e}\rangle$
	$\langle(\mathit{ff}, \mathsf{none}, B, \mathit{msg}), m_1 f_2 g_{0e}\rangle$

En effet, nous avons désormais $\mathit{cond}_{co(g0_e)} = (pc = B) \wedge (t \wedge p = \mathit{stop} \vee \neg t \wedge p = \mathit{msg})$ *plutôt que* $(pc = B)$*, condition calculée lors de l'analyse de co-accessibilité. En ne tenant compte que du sommet de pile, nous pouvons donc anticiper quelle sortie émettre suivant la valeur de la variable* `sent` *: si celle-ci est à vrai, un stop est émis, si elle est à faux, un* msg *est émis. Nous pouvons alors vérifier que l'équation 8.2 est fausse pour le point de contrôle* $k = g0_e$*, la sélection est alors optimale à ce point.*

Notons que la présence de la variable `sent` *aide à obtenir une sélection optimale au point de contrôle* g_0 *puisqu'elle permet de distinguer si la fonction* `G()` *a été appelée depuis* f_1 *ou depuis* g_2*. Cette variable n'a aucune influence sur la sémantique de*

la spécification par rapport à la relation de conformité ioco. Nous pouvons donc la supprimer. Si nous le faisions, nous ne pourrions pas sélectionner de manière optimale les sorties stop et msg à envoyer à l'IUT.

La figure 8.3 décrit le cas de test obtenu par sélection sur le produit représenté à la figure 8.2. Nous y remarquons l'ajout des conditions en g_0 sur la variable `sent` *ainsi que l'affectation* Inconc *à la variable* `verdict` *dès qu'une entrée fait sortir le programme des traces acceptées par l'objectif de test (PCS en $f0_r$, $f2_r$ et $g0_r$).*

```
// Type and global variables Declarations
// ...

void main(){
  m0: f();
  m1:
}

void f()
{
  f0: receive (p) when true {
  f0r:  if (p != invit && p != pb)
          { verdict = fail; abort(); }
        TP(p);
        if (p == pb) goto f3;

      };
  f1: g();
  f2: emit(p) when (p == quit) {
  f2e:  TP(p)
      }
      []
      receive(p) when true {
  f2r:  if (p != delta)
          { verdict = fail; abort() };
        if (p = delta) goto f2;

      };
  f3: return;
}

void g()
{
  g0: emit(p) when (p == stop || p == msg){

  g0e:  TP(p);
        if (p == stop) goto g4;
      }
      []
      receive(p) when true  {
   g0r: if (p != delta)
          { verdict = fail; abort() };
        if (p = delta) goto g0;

      };
  g1: sent = true;
  g2: g();
  g3: receive(p) when true {
  g3r:  if (p != ack)
          { verdict = fail; abort(); }
        TP(p);
      }
  g4: return;
}
```

FIG. 8.2 – Produit.

```
// Type and global variables Declarations
// ...

void main(){
  m0: f();
  m1:
}

void f()
{
  f0: receive (p) when true {
  f0r:  if (p != invit && p != pb)
          { verdict = fail; abort(); }
        TP(p);
        if (p == pb)
          { verdict = inconc; abort(); }
      };
  f1: g();
  f2: emit(p) when (p == quit) {
  f2e:  TP(p)
      }
      []
      receive(p) when true {
  f2r:  if (p != delta)
          { verdict = fail; abort() };
        if (p = delta)
          { verdict = inconc; abort(); }
      };
  f3: return;
}

void g()
{
  g0: emit(p) when (p == stop  && t == true)
                || (p == msg && t == false)){
  g0e:  TP(p);
        if (p == stop) goto g4;
      }
      []
      receive(p) when true  {
   g0r: if (p != delta)
          { verdict = fail; abort() };
        if (p = delta)
          { verdict = inconc; abort(); }
      };
  g1: sent = true;
  g2: g();
  g3: receive(p) when true {
  g3r:  if (p != ack)
          { verdict = fail; abort(); }
        TP(p);
      }
  g4: return;
}
```

FIG. 8.3 – Cas de test obtenu après sélection.

Chapitre 9

Expérimentations

Nous présenterons dans ce chapitre quelques expérimentations effectuées sur l'outil InterprocStack réalisé par Bertrand Jeannet. Cet outil est une extension d'Interproc : un analyseur interprocédural pour un petit langage de programmation impératif avec appels de procédure récursifs [int]. Cet analyseur calcule des invariants sur les variables numériques du programme par analyse avant. Il peut également calculer les conditions nécessaires à l'atteignabilité d'un point de contrôle donné par analyse arrière ou combiner alternativement chacune des deux analyses. Pour réaliser ces analyses, Interproc utilise la librairie APRON [Apr] afin d'abstraire les variables numériques et la librairie Fixpoint [Fix] afin de calculer les points-fixes. En plus des variables numériques (entiers, réels et flottants), Interproc accepte également les variables booléennes et énumérées. Contrairement aux variables numériques, les variables à domaine fini ne sont pas abstraites lors de l'analyse.

InterprocStack est une extension d'Interproc : là où Interproc utilise des analyses classiques, InterprocStack va abstraire les piles en utilisant une nouvelle forme de treillis abstrait : les automates de treillis [LGJ07, LG08]. Les automates de treillis permettent une abstraction plus précise du contenu de la pile que l'analyse relationnelle utilisée dans Interproc. Les analyses approchées seront donc effectuées dessus.

Quelques expérimentations vont être présentées ici à partir des exemples vus dans cette partie. Toutes ces expérimentations seront faites en utilisant les polyèdres convexes comme domaine abstrait pour des variables numériques (domaine utilisé par InterprocStack par défaut).

La syntaxe de l'outil InterprocStack étant différente de celle vue au chapitre 7, nous avons dû faire quelques adaptations.

Par exemple, InterprocStack ne gère pas de variables globales, nous avons donc utilisé les paramètres des fonctions pour les simuler.

De plus, nous ne distinguons pas les entrées des sorties, nous n'utilisons qu'un seul type énuméré pour simuler la communication (`com_t`). La fonction `random` utilisée régu-

lièrement permet de choisir une valeur du type énuméré aléatoirement. D'autres types énumérés sont utilisés dans ces exemples : l'un pour les verdicts, l'autre pour les états de l'objectif de test.

Les instructions `halt` et `fail` arrêtent toutes les deux le programme. La différence réside dans la marque faite par `fail` au dernier point de contrôle avant son appel. Cette marque servira de point de départ à l'analyse arrière.

L'instruction `assume expr` est équivalente à `if expr then skip ; else halt ;` et permet d'abstraire les affectations non-déterministes (l'instruction `skip` ne fait rien). Ainsi, si nous voulons qu'une variable `x` soit affectée à une valeur entre 0 et 2, nous pouvons l'écrire `x = random ; assume x>=0 and x <=2`.

9.1 Calculatrice avec écriture préfixe

Prenons l'exemple de la calculatrice avec écriture préfixe vu dans l'introduction de cette partie à la figure 5.18 et son objectif de test. La spécification de ce système comprend trois fonctions : la fonction principale `main`, la fonction récursive permettant d'effectuer les calculs `Calcul` et celle permettant d'entrer des nombres `Nombre`. Le produit du testeur canonique calculé à partir de cette spécification avec l'objectif de test est décrit à la figure 9.1 avec la syntaxe d'InterprocStack. Pour alléger le programme, nous n'avons pas tenu compte de l'opération de suspension, il n'y a donc aucun blocage modélisé ici.

La dernière fonction du programme ne portant pas de nom correspond à la fonction `main`. Celle-ci initialise d'abord les variables "globales" puis appelle la fonction `Calcul`. Au retour de cette fonction, si un message `Result` est produit, alors la fonction de l'objectif `TP` est appelée, sinon la variable `verdict` est affectée à `fail`. La fonction `Calcul()` a deux entrées possibles : si un chiffre (`Ch`) est reçu, alors la fonction `Nombre()` est appelée, si c'est un opérateur (`Op`) alors la fonction `Calcul()` est rappelée deux fois récursivement. Si la variable de communication `p` vaut `Result`, alors `verdict` prend la valeur `fail`. La fonction `Nombre()` a également deux comportements possibles suivant son entrée : si c'est un chiffre, elle est à nouveau appelée récursivement, si c'est le signal indiquant la fin du nombre entré (`ok`), alors la fonction retourne à la fonction appelante.

L'objectif de test modélisé ici satisfait l'ensemble des traces :

$$\mathit{Traces}(TP, D) = \{Op \cdot (Ch^{+} \cdot Ok)^{*} \cdot \mathit{Result}\}.$$

Si la variable simulant les états `pc` est affectée à `D`, alors `verdict` vaut `Pass`.

InterprocStack peut effectuer des analyses avant, arrière ou les deux. Cet outil va alors étiqueter les points de contrôle du résultat de l'analyse du programme. Nous ne pouvons pas mettre l'intégralité du résultat de l'analyse dans ce document, nous n'allons donc mettre que les résultats aux points de contrôle intéressants. Dans cet exemple, les points de contrôle intéressants sont les points $c0$ et $n0$ de la figure 5.21, c'est-à-dire les lignes 17 et 40 du programme (premiers `else` des fonctions `Calcul` et `Nombre`). Voici un extrait du résultat obtenu après une analyse arrière et avant :

```
typedef verdict_t = enum {Fail, Pass, Inconc, None};
com_t = enum {Result, Ch, Op, Ok};
pc_t = enum {A, B, C, D, S};

proc Calcul(v:verdict_t, c:pc_t)
     returns (verdict:verdict_t, pc:pc_t)
var p:com_t;
begin
  verdict = v;
  pc = c;

  p = random;
  assume p == Op or p == Ch or p == Result;
  if (p == Result) then
    verdict = Fail;
    halt;
  else
    (verdict, pc) = TP(verdict, pc, p);
    if (p==Ch) then
    (verdict, pc) = Nombre(verdict, pc);
    else
      (verdict, pc) = Calcul(verdict, pc);
      (verdict, pc) = Calcul(verdict, pc);
    endif;
  endif;
end

proc Nombre(v:verdict_t, c:pc_t)
     returns (verdict:verdict_t, pc:pc_t)
var p:com_t;
begin
  verdict = v;
  pc = c;

  p = random;
  assume p== Ok or p == Ch or p == Result;
  if (p == Result) then
    verdict = Fail;
    halt;
  else
     (verdict, pc) = TP(verdict, pc, p);
     if (p == Ch) then
       (verdict, pc) = Nombre(verdict, pc);
     endif;
  endif;
end
```

```
proc TP(v:verdict_t, c:pc_t, p:com_t)
     returns (verdict:verdict_t, pc:pc_t)
begin
  pc = c;
  verdict = v;

  if (pc == A and p == Op) then
    pc = B;
  else if (pc == B and p == Ch) then
    pc = C;
  else if (pc == C and p == Ch) then
    pc = C;
  else if (pc == C and p == Ok) then
    pc = B;
  else if (pc == B and p == Result) then
    pc = D;
    verdict = Pass;
    fail;
  else
    pc = S;
  endif;
  endif;
  endif;
  endif;
  endif;
end

var verdict:verdict_t, pc:pc_t, p:com_t;
begin
  verdict = None;
  pc = A;
  (verdict, pc) = Calcul(verdict, pc);
  p = random;
  if (p != Result) then
    verdict = Fail;
    halt;
  else
    (verdict, pc) = TP(verdict, pc, p);
  endif;
end
```

FIG. 9.1 – Produit entre le testeur canonique de la calculatrice et son objectif de test sous forme de programme.

```
Annotated program after backward and forward analysis
[...]
 else
    /* (L17 C6)
       { top IF c = A and p = Op and pc = A and v = None and verdict = None
                or c = B and p = Ch and pc = B and v = None and
                   verdict = None,
         bottom OTHERWISE } */
[...]
 else
   /* (L40 C6)
       { top IF c = C and p in {Ch,Ok} and pc = C and v = None and
                verdict = None,
         bottom OTHERWISE } */
[...]
```

Nous constatons que grâce à cette analyse, nous pouvons savoir quand émettre `Op` ou `Ch` à la ligne 17 : si `pc` vaut `A` alors on émet un `Op`, si `pc` vaut `B`, `Ch` est émis. Grâce à cette analyse, nous obtenons en ce point une sélection optimale. Ce n'est par contre pas le cas à la ligne 40 où nous ne pouvons pas savoir quand émettre `Ok` ou `Ch`.

9.2 Envoi et réception de messages

Reprenons l'exemple d'envoi et de réception de messages vu tout au long de la partie III. La spécification et l'objectif de test sont représentés à la figure 6.6. Nous avons vu au chapitre 8 qu'en théorie, sur cet exemple, nous arrivons à avoir une sélection optimale grâce à la variable `sent` en combinant les informations calculées par les analyses avant et arrière. Il est donc intéressant de voir ce qu'InterprocStack va calculer sur cet exemple.

Le programme représentant le produit entre le testeur canonique de cette spécification et l'objectif de test est donné à la figure 9.2 avec la syntaxe d'InterprocStack.

Comparons maintenant les résultats des analyses calculés à la figure 8.1 à ce que calcule InterprocStack au point de contrôle particulièrement intéressant de cet exemple : $g0$ (correspondant à la ligne 49 du programme, c'est-à-dire le premier `else` de la fonction `G`). L'analyse arrière d'InterprocStack rend le résultat suivant :

```
Annotated program after backward analysis
[...]
  else
    /* (L49 C6)
       { top IF p in {Stop,Msg} and pc = B
                or p in {Ack,Pb} and pc in {C,D}
```

```
typedef verdict_t = enum {Fail, Pass, Inconc, None};
com_t = enum {Stop, Msg, Quit, Invit, Ack, Pb};
pc_t = enum {A, B, C, D, E, S};

proc F(s:bool, v:verdict_t, c:pc_t)
     returns (sent:bool, verdict:verdict_t, pc:pc_t)
var p:com_t;
begin
  sent = s;
  verdict = v;
  pc = c;

  p = random;
  assume p==Invit or p==Ack or p==Pb;
  if (p != Invit and p != Pb) then
    verdict = Fail;
    halt;
  else
    (verdict, pc) = TP(verdict, pc, p);
    if (p == Invit) then
      (sent, verdict, pc) = G(sent, verdict, pc);

      p = random;
      assume p==Quit or p==Invit or p==Ack or p==Pb;
      if (p == Quit) then
        (verdict, pc) = TP(verdict, pc, p);
      else
        verdict = Fail;
        halt;
      endif;
    endif;
  endif;
end

proc G(s:bool, v:verdict_t, c:pc_t)
     returns (sent:bool, verdict:verdict_t, pc:pc_t)
var p:com_t;
begin
  sent = s;
  verdict = v;
  pc = c;

  p = random;
  assume p==Stop or p==Msg
         or p==Invit or p==Ack or p==Pb;
  if (p==Invit or p==Ack or p==Pb) then
    verdict = Fail;
    halt;
  else
    if (p==Stop or p==Msg) then
      (verdict, pc) = TP(verdict, pc, p);
      if (p==Msg) then
        sent = true;
        (sent, verdict, pc) = G(sent, verdict, pc);

        p = random;
        assume p==Invit or p==Ack or p==Pb;
        if (p!=Ack) then
          verdict = Fail;
          halt;
        else
          (verdict, pc) = TP(verdict, pc, p);
        endif;
      endif;
    endif;
  endif;
end
```

```
proc TP(v:verdict_t, c:pc_t, p:com_t)
     returns (verdict:verdict_t, pc:pc_t)
begin
  pc = c;
  verdict = v;

  if (pc == A and p == Invit) then
    pc = B;
  else if (pc == B and p == Msg) then
    pc = B;
  else if (pc == B and p == Stop) then
    pc = C;
  else if (pc == C and p == Ack) then
    pc = D;
  else if (pc == D and p == Quit) then
    pc = E;
    verdict = Pass;
    fail;
  else
    pc = S;
  endif;
  endif;
  endif;
  endif;
  endif;
end

var sent:bool, verdict:verdict_t, pc:pc_t;
begin
  sent = false;
  verdict = None;
  pc = A;
  (sent, verdict, pc) = F(sent, verdict, pc);
end
```

FIG. 9.2 – Produit entre le testeur canonique et son objectif de test sous forme de programme.

```
                or p in {Quit,Invit} and pc in {C,D},
          bottom OTHERWISE } */
[...]
```

Nous constatons que nous avons bien le même résultat : si la variable `pc` vaut `B`, nous ne savons pas si nous devons produire un `stop` ou un `msg`. Observons maintenant le résultat de l'analyse avant :

```
Annotated program after forward analysis
[...]
 else
    /* (L49 C6)
       { top IF not s and not sent and c = B and p in {Stop,Msg} and
                pc = B and v = None and verdict = None
                or s and sent and c = B and p in {Stop,Msg} and pc = B and
                   v = None and verdict = None,
         bottom OTHERWISE } */
[...]
```

Là encore, nous constatons avoir les mêmes résultats : nous ne pouvons pas non plus distinguer quelle sortie émettre, et ce, quelle que soit la valeur de la variable `sent`.

L'intersection entre les résultats de ces deux analyses devrait maintenant nous permettre d'avoir une sélection optimale. InterprocStack est jusqu'à présent lancé grâce à la ligne de commande suivante :

```
interprocstack -analysis bf -stack false false false false 0 1 exemple.spl
```

Cela signifie qu'InterprocStack va produire une analyse arrière (b) puis avant (f) avec une abstraction de la pile dont les bisimulations avant et arrière sont respectivement bornées à 0 et 1. Nous ne nous intéresserons pas aux autres paramètres ici, leurs valeurs étant, comme celles de la bisimulation, celles par défaut. Le résultat de cette ligne de commande est le suivant :

```
Annotated program after backward and forward analysis
[...]
  else
    /* (L49 C6)
       { top IF not s and not sent and c = B and p in {Stop,Msg} and
                pc = B and v = None and verdict = None
                or s and sent and c = B and p in {Stop,Msg} and pc = B and
                   v = None and verdict = None,
         bottom OTHERWISE } */
[...]
```

Nous constatons qu'InterprocStack ne produit alors pas une sélection optimale puisque nous avons toujours un choix entre les sorties `Stop` et `Msg`. Nous allons alors augmenter la borne de la bisimulation arrière :

```
interprocstack -analysis bf -stack false false false false 0 2 exemple.spl
```

ce qui produit le résultat suivant :

```
Annotated program after backward and forward analysis
[...]
 else
    /* (L49 C6)
       { top IF s and sent and c = B and p = Stop and pc = B and v = None and
                verdict = None
                or not s and not sent and c = B and p = Msg and pc = B and
                   v = None and verdict = None,
         bottom OTHERWISE } */
[...]
```

La sélection est ici optimale : si la variable `sent` vaut `vrai`, alors `Stop` sera émis, si elle vaut `faux` ce sera `Msg` qui le sera.

Nous arrivons donc à obtenir, sur ces deux exemples, les mêmes résultats avec InterprocStack que ceux calculés en théorie. Nous pourrions donc utiliser cet outil dans le cadre d'un logiciel de génération de tests sur des systèmes récursifs modélisés par des ioPDS. De plus, ce logiciel permet l'utilisation de variables numériques ce qui nous permettrait d'étendre notre théorie aux programmes récursifs en général.

Conclusion

Les principes et algorithmes de la génération automatique de tests dans le cadre du test de conformité à la ioco entre une implémentation réactive boîte noire et sa spécification sont ici étendus aux spécifications récursives, modélisables par des systèmes à pile. De telles spécifications peuvent être plus compactes et expressives que des spécifications non récursives.

Les cas de test générés sont sélectionnés à partir d'un objectif de test, un scénario (ou un ensemble de scénarios) que nous souhaitons observer pendant l'exécution du test. La méthode de génération de tests que nous proposons dans cette partie est basée sur la transformation de programmes et l'analyse de co-accessibilité. Cette dernière permet de décider si et comment l'objectif de test peut encore être satisfait. Cependant, malgré la possibilité d'avoir une analyse exacte, l'incapacité des cas de test à connaître le contenu de leur propre pile ne permet pas d'utiliser la totalité de l'information donnée par l'analyse.

Il est intéressant de noter la similarité des deux combinaisons : l'observation partielle et l'analyse exacte d'un côté (partie III sur les systèmes à pile avec variables à domaine fini), l'observation complète et l'analyse inexacte de l'autre (partie II sur les systèmes symboliques avec variables à domaine infini). Dans le cas de l'observation partielle, la fonction d'observation α que nous avons introduite agit exactement comme une fonction d'abstraction, et donc d'approximation. Ceci signifie que nous pouvons appliquer cette méthode aux programmes récursifs en général, sur lesquels l'analyse est approchée. La non-optimalité de la sélection est alors une conséquence de la combinaison entre observation partielle et analyse approchée, comme représenté sur le diagramme de la figure 9.3. Cette combinaison a été illustrée lors des expérimentations effectuées sur l'outil InterprocStack mais n'a pas encore été formalisée.

Méthodes alternatives Notre méthode de sélection expliquée au chapitre 8 est basée d'une part sur une analyse exacte calculant l'intégralité des configurations (au lieu de ne calculer que les parties visibles des configurations) et d'autre part sur les transformations de programme. Ces deux choix peuvent être discutés.

Concernant l'analyse, nous pourrions utiliser une méthode d'analyse classique interprocédurale moins précise, qui pourrait être exacte pour la partie observable de la pile. Cependant, cela pourrait mener à une sélection moins précise. En particulier, l'intersection entre l'ensemble des configurations co-accessibles et celui des configurations

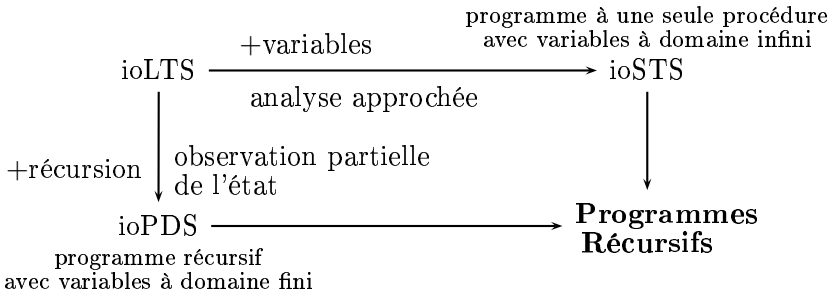


FIG. 9.3 – La sélection de test sur différents modèles

accessibles pourrait filtrer moins de valeurs.

Concernant les transformations de programme, nous pourrions ajouter des informations au programme afin d'avoir une plus grande connaissance sur la partie invisible des configurations. Nous pourrions par exemple ajouter une structure de données simulant une pile des points de retour de procédure, et l'utiliser pour tester si l'un de ces points fait partie d'une configuration co-accessible. Le cas de test résultant pourrait alors ne pas pouvoir être transformé en ioPDS, mais l'analyse pourrait être faite sur le programme intermédiaire comme dans le chapitre 8. L'exécution du cas de test pourrait être cependant plus lente puisque les tests de co-accessibilité impliqueraient des types de données plus complexes.

Conclusion générale

Le travail détaillé dans ce document avait pour objectif, d'une part, de combiner trois niveaux de description (propriétés, spécification et implémentation) en utilisant à la fois la vérification formelle et le test de conformité (partie II), d'autre part d'étendre la génération de tests de conformité à la ioco par l'expressivité du modèle de spécification (partie III).

Nous avons proposé dans la partie II de ce document une méthodologie permettant de détecter à la fois la violation/satisfaction des propriétés de sûreté/accessibilité mais également la non-conformité de l'implémentation par rapport à sa spécification. Les propriétés de sûreté et/ou d'accessibilité sont d'abord vérifiées sur la spécification. Le modèle utilisé étant celui des ioSTS, modèle complexe comportant des variables, la vérification peut alors être approximative et partielle. Les cas de test dérivés de la spécification et des propriétés sont ensuite exécutés sur l'implémentation. Si la vérification a pu établir la satisfaction de la propriété, l'exécution permet de détecter, en plus de la non-conformité, la satisfaction ou la violation de la propriété par l'implémentation. De plus, si la vérification n'a pas pu être concluante, l'exécution du test peut alors détecter la violation/satisfaction de la propriété de sûreté/accessibilité via un verdict. Les verdicts obtenus permettent donc d'exprimer la non-conformité entre implémentation et spécification mais également la satisfaction/violation de propriétés par l'une et/ou l'autre de ces deux entités. La combinaison entre vérification et test permet ainsi de relier trois niveaux de description différents : propriétés, spécification et implémentation.

Nous avons ensuite étendu la méthode de génération de tests vue en partie I à un modèle plus expressif que les ioLTS : les spécifications interprocédurales récursives. Le modèle utilisé dans la partie III est celui des ioPDS permettant les appels entre procédures et la récursivité mais ne portant, pour des raisons de simplicité, que sur des variables à domaine fini. Ce modèle permet une représentation plus expressive et plus compacte que des modèles non récursifs. Les cas de test récursifs sont alors dérivés à partir d'une spécification récursive et d'un objectif de test non-récursif. La sélection permettant de générer ce cas de test se fait par analyses de co-accessibilité et d'accessibilité. Ces analyses sont exactes, mais l'incapacité des cas de test à inspecter le contenu de leur propre pile induit une prise en compte partielle des résultats de l'analyse[1]. Nous ne pouvons alors pas assurer l'optimalité de la sélection mais grâce

[1]Nous considérons que les cas de test sont capables de savoir à quel point de contrôle ils sont, mais

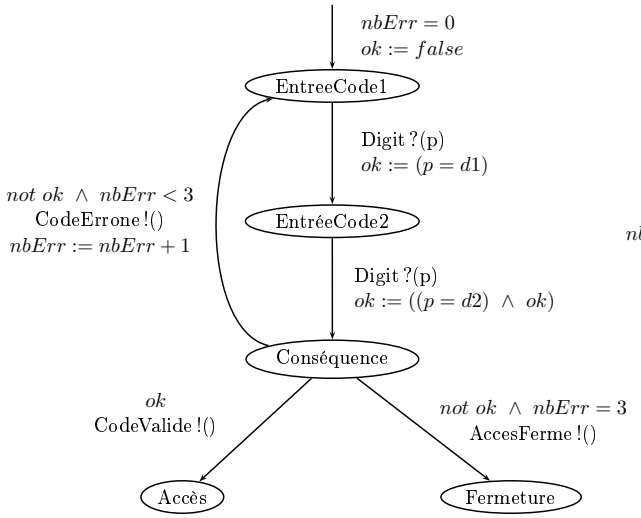
aux résultats combinés des analyses de co-accessibilité et d'accessibilité, nous pouvons minimiser l'impact négatif de l'observation partielle.

Ces deux parties sont traitées séparément dans ce document, mais peuvent néanmoins être combinées : la génération de tests sur les ioPDS se ferait alors grâce à une sélection par observateur et non par objectif de test.

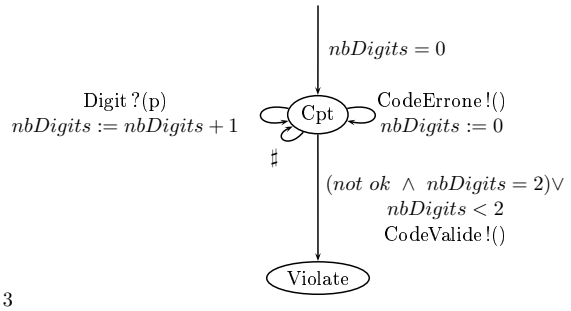
Perspectives. La combinaison entre vérification et test permet de relier trois niveaux de description, mais elle limite l'expressivité du modèle utilisé pour la spécification et les propriétés. En effet, dans [JJRZ05, JJR07] un algorithme symbolique de sélection de cas de test à partir d'une spécification et d'objectifs de test est proposé. Le modèle des ioSTS utilisé dans ces papiers est différent de celui que nous avons utilisé dans la partie II car il distingue deux types de variables : les variables propres (internes) au système et les variables observées (externes) par celui-ci. Cela permet à l'objectif de test d'observer des variables propres à la spécification (la spécification ne contenant que des variables internes). Ce modèle ne peut être utilisé dans le cadre de la combinaison vérification et test telle que nous l'avons présentée. En effet, si nous utilisions des propriétés observant des variables de la spécification, nous ne pourrions rien conclure de la satisfaction/violation de la propriété par rapport à l'implémentation puisque nous ne connaissons de celle-ci que l'interface. Prenons par exemple la spécification du digicode vue dans la partie I étendue par des variables (voir figure 9.4(a)). Une propriété de sûreté intéressante pourrait être qu'un utilisateur ne fournissant pas le code correct ne puisse pas accéder à la ressource (voir figure 9.4(b)). Il serait alors important de pouvoir vérifier cette propriété sur l'implémentation, malheureusement cela n'est pas possible. Si le verdict **Violate** (ou en l'occurrence **FailViolate**) était produit, nous ne pourrions rien conclure quant au lien entre la propriété et l'implémentation, la violation de la propriété dépendant de la variable observée *ok*. Une solution pourrait alors être de développer une méthode de génération de tests avec observation de variables mais cela impliquerait de faire des hypothèses sur le code de l'implémentation. Nous sortirions alors du cadre boîte noire.

Nous avons décidé dans la partie III au chapitre 6 que nous ne considérerions que les spécifications déterministes car dans le cas général, il est indécidable de savoir si les systèmes à pile sont déterminisables. Nous souhaiterions étendre les classes possibles de spécifications. Pour cela, il nous faudrait trouver une classe de systèmes à pile déterminisables intéressants pour la génération de tests. La classe déterminisable des *Visibly Pushdown Automata* (abrégés en VPA) présentés par Rajeev Alur et Parthasarathy Madhusudan dans [AM04] ne peut malheureusement pas être utilisée comme modèle de spécification car il implique que les appels et retours de procédures soient visibles. Une piste pourrait cependant être d'avoir des appels et retours de fonction indirectement observables. Il faudrait dans ce cas que chaque appel de procédure soit reconnaissable

ne peuvent pas savoir comment ils en sont arrivés là.



(a) Spécification du Digicode à 2 chiffres.



(b) Observateur représentant la négation d'une propriété de sûreté.

par une trace précise et qu'il en soit de même pour le retour (une sortie particulière juste avant le retour de la procédure par exemple). Dans l'exemple de la partie III, les fonctions `F()` et `G()` sont respectivement reconnaissables par la sortie $Invit!$ et par les entrées $Msg?$ et $Stop?$. De même, leurs retours sont observables par l'entrée $Quit?$ et par la sortie $Ack!$.

Nous utilisons dans la partie III de ce document le modèle des ioPDS, c'est-à-dire un système à pile à entrées/sorties avec variables à domaine fini. Nous souhaiterions étendre ce modèle à celui des systèmes récursifs en général, c'est-à-dire avec variables à domaine infini. Nous aurions alors une sélection non-optimale des cas de test due à la fois à l'observation partielle de la pile mais aussi aux analyses approchées, conséquence des variables à domaine infini. Afin d'effectuer des analyses approchées sur des configurations (et donc une pile) contenant des variables à domaine infini, nous pourrions modéliser le contenu de la pile par des automates de treillis, modèle développé par Tristan Le Gall et Bertrand Jeannet [LGJ07, LG08]. Cela a déjà été réalisé dans le cadre des expérimentations sur l'outil InterprocStack, comme expliqué au chapitre 9, mais n'a pas encore été formalisé. Il serait intéressant de développer notre méthode de génération de tests sur un modèle plus expressif que les ioPDS contenant des variables de ce type et d'implémenter ensuite cette génération de tests dans un outil.

Nous souhaiterions également étendre l'outil InterprocStack et notre théorie à des types de variables plus complexes. Nous ne travaillons pour l'instant que sur des variables simples dites scalaires. Il serait important d'étendre ce type de variables à des structures de données telles que les tableaux ou les listes, très souvent utilisés dans des

programmes impératifs. Pour les structures de données allouées dynamiquement et manipulant des pointeurs telles que les listes (piles, files, arbres, etc.), nous pourrions avoir recours à des techniques de *shape analysis* [JLRS04]. Ces techniques permettent d'analyser le programme en déterminant les informations dues aux structures de données manipulées.

Bibliographie

[ADX01] P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society, 2001.

[AM04] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, Chicago, Etats-Unis, Juin 2004. ACM.

[AN82] A. Arnold and M. Nivat. Comportements de processus. In *Colloque AFCET, Les mathématiques de l'informatique*, 1982.

[Apr] APRON : un ensemble de librairies sur différents domaines abstraits numériques partageant la même interface. http ://apron.cri.ensmp.fr/.

[BAL+90] E. Brinskma, A. Alderen, R. Langerak, J. van de Laagemat, and J. Tretmans. A formal approach to conformance testing. In *Protocol Secification, Testing and Verification (PSTV'90)*, pages 349–363, 1990.

[BBS+79] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E.F. Miller. Smotl : a system to construct samples for data processing program debugging. *IEEE Transactions on Software Engineering*, 5(1) :215–222, Janvier 1979.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata : application to model checking. In *Int. Conf. on Concurrency Theory, CONCUR'97*, volume 1243 of *LNCS*, 1997.

[BFG+00] M. Bozga, J.-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier. Verification and test generation for the SSCOP protocol. *Journal of Science of Computer Programming, special issue on Formal Methods in Industry*, 36(1) :27–52, Janvier 2000.

[BHJP04] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. Specifying and generating test cases using observer automata. In Jens Grabowski and Brian Nielsen, editors, *Proc. of Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 137–152. Springer, 2004.

[BR00] T. Ball and S. Rajamani. Bebop : a symbolic model checker for boolean programs. In *Workshop SPIN'00*, volume 1885 of *LNCS*, 2000.

[Bri88] E. Brinskma. A theory for the derivation of tests. In *Protocol Specification, Testing and Verification (PSTV'88)*, pages 63–74, 1988.

[Cau92] D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106, 1992.

[CC77] P. Cousot and R. Cousot. Abstract intrepretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CJJ07] C. Constant, B. Jeannet, and T. Jéron. Automatic test generation from interprocedural specifications. In *TestCom/Fates07*, number 4581 in LNCS, pages 41–57, Tallinn, Estonie, Juin 2007.

[CJMR06] C. Constant, T. Jéron, H. Marchand, and V. Rusu. Combinaison entre vérification et test pour la validation de systèmes réactifs. In *Traité I2C. Systèmes Temps Réel : Techniques de Description et de Vérification - Théorie et Outils*, volume 1, chapter 2, pages 59–88. Hermès Science, 2006.

[CJMR07] C. Constant, T. Jéron, H. Marchand, and V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33(8) :558–574, Août 2007.

[CJRZ02] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG : a symbolic test generation tool. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, number 2280 in LNCS, pages 470–475, 2002.

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

[DN84] J.W. Duran and S.C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4) :438–444, Juillet 1984.

[ES01] Javier Esparza and Stefan Schwoon. A BDD-based model checker for recursive programs. In *Computer Aided Verification, CAV'01*, volume 2102 of *LNCS*, 2001.

[FGLG07] A. Faivre, C. Gaston, and P. Le Gall. Symbolic model based testing for component oriented systems. In *TestCom/Fates07*, number 4581 in LNCS, pages 90–106, Tallinn, Estonie, Juin 2007.

[Fix] Fixpoint : une librairie OCaml implémentant un solveur de point-fixe. http ://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/fixpoint/.

[FMP03] J.C. Fernandez, L. Mounier, and C. Pachon. Property-oriented test generation. In *Formal Aspects of Software Testing Workshop*, number 2931 in LNCS, 2003.

[FWW97] Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes on Theoretical Computer Science*, 9, 1997.

[GH99] A. Gargantini and C.L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/SIGSOFT FSE*, pages 146–162, 1999.

[HLSU02] H.S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 327–341, Grenoble, France, Avril 2002. Springer.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1) :60–65, 2001.

[HR02] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Int. Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), Grenoble, France*, number 2280 in LNCS, pages 342–356, 2002.

[int] Interproc : analyseur interprocédural pour langage impératif avec appels de procédure (récursifs). http ://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html.

[ISO92] ISO/IEC 9646. Conformance testing methodology and framework, 1992.

[Jea03] B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1) :5–37, 2003.

[JJ04] 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. *Software Tools for Technology Transfer (STTT)*, 6, Octobre 2004.

[JJR07] B. Jeannet, T. Jéron, and V. Rusu. Model-based test selection for infinite state reactive systems. In F.S de Boer, M.M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods of Components and Objects - FMCO 2006, Amsterdam, Netherlands, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 47–69. Springer-Verlag, 2007.

[JJRZ05] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, Edinburgh, Ecosse, Avril 2005. Springer.

[JLRS04] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *The 11th International Static Analysis Symposium, SAS 2004*, LNCS, New York, Etats-Unis, 2004.

[JMR06] T. Jéron, H. Marchand, and V. Rusu. Symbolic determinisation of extended automata. In *4th IFIP International Conference on Theoretical Computer Science*, IFIP book series, Santiago, Chili, Août 2006. Springer Science and Business Media.

[JMRT04] T. Jéron, H. Marchand, V. Rusu, and V. Tschaen. Ensuring the conformance of reactive discrete-event systems by means of supervisory control. *International Journal of Production Research*, 42(14) :2809–2826, 2004.

[JR07] E. Jahier and P. Raymond. Generating random values using binary decision diagrams and convex polyhedra. In *Trends in Constraint Programming*, chapter 22, pages 349–356. ISTE, 2007.

[Jér02] T. Jéron. Tgv : théorie, principes et algorithmes. *Techniques et Sciences Informatiques, numéro spécial Test de Logiciels*, 21, 2002.

[Kin76] J.C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7) :385–394, Juillet 1976.

[Kor90] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8) :870–879, Août 1990.

[LG08] T. Le Gall. *Abstract lattices for the verification of systems with queues and stacks*. PhD thesis, Université de Rennes I, Juillet 2008.

[LGJ07] T. Le Gall and B. Jeannet. Lattice automata : a representation of languages over an infinite alphabet, and some applications to verification. In *The 14th International Static Analysis Symposium, SAS 2007*, number 4634 in LNCS, pages 52–68, Kongens Lyngby, Danemark, Août 2007.

[LT99] N. Lynch and M. Tuttle. Introduction to IO automata. *CWI Quarterly*, 3(2), 1999.

[Pet02] C. Petitpierre. Synchronous active objects introduce csp's primitives in java. In *Communicating Process Architectures (CPA'2002)*, pages 109–122. IOS-Press, septembre 2002.

[PJJ07] F. Ployette, B. Jeannet, and T. Jéron. Stg : a symbolic test generation tool for reactive systems. TESTCOM/FATES07 (Tool Paper), June 2007.

[PVY01] D. Peled, M. Vardi, and M. Yannakakis. Black-box checking. *Journal of Automata, Languages and Combinatorics*, 7(2) :225–246, 2001.

[RdBJ00] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *International Conference on Integrating Formal Methods (IFM'00)*, LNCS 1945, pages 338–357. Springer Verlag, Novembre 2000.

[RMJ05] Vlad Rusu, Hervé Marchand, and Thierry Jéron. Automatic verification and conformance testing for validating safety properties of reactive systems. In John Fitzgerald, Andrzej Tarlecki, and Ian Hayes, editors, *Formal Methods 2005 (FM05)*, volume 3582 of *LNCS*. Springer, Juillet 2005.

[RMT+04] V. Rusu, H. Marchand, V. Tschaen, T. Jéron, and B. Jeannet. From safety verifcation to safety testing. In *International Conference on Testing of Communicating Systems (TestCom04)*, volume 2978 of *LNCS*. Springer, 2004.

[TFW91] P. Thévenod-Fosse and H. Waeselynck. An investigation of statistical software testing. *Journal of Software Testing, Verification and Reliability*, 1(2) :5–25, Juillet 1991.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3) :103–120, 1996. Also : Technical Report No. 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[Tre99] J. Tretmans. Testing concurrent systems : a formal approach. In *CONCUR'99*, volume 1664 of *LNCS*, pages 46–65. Springer, 1999.

[ZHM97] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4) :366–427, Décembre 1997.

[Zin04] E. Zinovieva. *Méthodes symboliques pour la génération de tests de systèmes réactifs comportant des données.* PhD thesis, Université de Rennes 1, Novembre 2004.

Table des figures

Résumé

Nous nous intéressons dans ce document à la génération automatique de tests de conformité pour des implémentations réactives. Nous nous attachons dans un premier temps à étendre la méthode de génération de tests, basée sur la théorie du test de conformité à la ioco, en reliant trois niveaux de description (propriétés, spécification et implémentation). Nous combinons pour cela vérification formelle et test de conformité. Nous obtenons ainsi, lors de l'exécution du cas de test sur l'implémentation, des verdicts pouvant indiquer la non-conformité de l'implémentation, mais également la satisfaction/violation de la propriété par l'implémentation et/ou la spécification. Nous étendons dans un deuxième temps la génération de tests par l'expressivité du modèle de spécification en nous intéressant aux spécifications interprocédurales récursives. Notre méthode est basée sur une analyse exacte de co-accessibilité permettant de décider si et comment un objectif de test pourra être atteint. Cependant, l'incapacité des cas de test récursifs à connaître leur propre pile d'exécution ne permet pas d'utiliser la totalité des résultats de l'analyse. Nous discutons de ce problème d'observation partielle et de ses conséquences puis nous proposons un moyen de minimiser son impact. Enfin, nous expérimentons ces méthodes de génération de tests sur quelques exemples et une étude de cas.

Abstract

This thesis adresses the problem of automatic test case generation for testing the conformance of a reactive implementation. We first propose a methodology which extends the testing theory based on the ioco conformance relation by distinguishing three levels of description : properties, specification and implementation. The methogology integrates verification and conformance testing. The execution of the generated test cases on the implementation allows detecting conformance violations between implementation and specification, but also violation/satisfaction of the properties by the implementation or the specification. Secondly, we introduce a more expressive specification model : recursive interprocedural specifications. The test generation method we propose is based on coreachability analysis, which allows deciding whether and how the test purpose can still be satisfied. However, although it is possible to carry out an exact analysis, the inability of test cases to inspect their own stack prevents them from fully using the coreachability information. We discuss this partial observation problem, its consequences, and how to minimise its impact. Finally, we experiment these methods of test generation on several examples and a case study.