Symbolic Test Generation for Reactive Systems with Data.

Elena Leroux (zinovieva)

N° d'ordre: 3067

THÈSE

Présentée devant

devant l'Université de Rennes 1

pour obtenir

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

par

Eléna Zinovieva-Leroux

Équipe d'accueil : VerTeCs
École doctorale : MATISSE
Composante universitaire : IFSIC/IRISA

Titre de la thèse :

Méthodes symboliques pour la génération de tests de systèmes réactifs comportant des données

soutenue le 22 novembre 2004 devant la commission d'examen

M. :	Olivier	Ridoux	Président
MM. :	Jan	Tretmans	Rapporteurs
	Bruno	Legeard	
MM. :	Claude	Jard	Examinateurs
	Bruno	Marre	
	Vlad	Rusu	

To my parents Lidia and Alexandre,
my sister Irina and
my grandparents Maria and Nikolai.

Acknowledgments

First of all I would like to thank my scientific advisors Vlad Rusu, Thierry Jéron and Claude Jard for their help and encouragement throughout my studies. They have made sure that the work, started four years ago, gave the scientific results which published in the papers and written as a thesis; and more importantly I learned something from each step of the research process, starting from the exploration of a bibliography and a research field to the implementation of new ideas and publishing them in papers. I am grateful for that.

Jan Tretmans, Bruno Legeard, Bruno Marre and Olivier Ridoux are kindly thanked to their willing to judge this dissertation and to be members of the committee.

I also would like to thank all members of the research group VerTeCs for a good research environment. Special thanks are addressed to Duncan Clarke who gave a life to pure theoretical ideas of Vlad and Thierry in the world of techniques for software testing. The prototype developed by him and called later STG became a lego-like toy for me for the three years. I am very pleased to meet Duncan at the first year of my PhD study as he helped me to understand the rules of the "PhD game" and did not give me to sink in the huge volume of the new information. His enthusiasm and humor is missed. I am grateful to Bertrand Jeannet. In collaboration with him I developed semantic-based method for selection of test cases (*see* Chapter 7, Section 7.4) and implement it in STG. Discussions with Bertrand opened the interesting world of abstract interpretation for me. I am also thankful to François-Xavier Ponscarme who spend almost half of his time debugging STG.

I am most grateful to those who have expressed confidence to my ability. Yakov Zaidelman, Konstantin Shvachko, Sergey Duzhin and Walid Taha in particular were a great influence. Yakov sparked my interest in programming. Konstantin and Sergey helped me to make the first steps in computer science and applied mathematics. Konstantin and Walid encouraged me to pursue a PhD study. For them I reserve a mixture of emotions including gratitude.

Also I would like thank all my friends in France, Sweden, USA, Russia and elsewhere in the world who helped me to go through all difficulties I met along the way. Especially I would like to thank Mathieu Leroux who corrected some sections of the thesis and helped me to translate the extended abstract of this thesis in French. I am also thankful to Jullia Lawall for her careful reading of the introductory part of the thesis.

Finally, I reserve greatest thanks for my husband Aurélien who helped to clarify some hard parts of my dissertation by reading and discussing them with me, who always provided encouragement and moral support, and who helped me to remember the things that really matter in the life. I also would like to

express my gratitude to my parents Lidia and Alexandre, my sister Irina and my grandparents Maria and Nikolai, who have always been behind me. This thesis is dedicated to them.

Eléna Zinovieva-Leroux,
Rennes, November 2004.

Contents

Résumé

1.1 Introduction

Le test, dans toutes ses variantes, est l'une des techniques de validation de logiciels parmi les plus utilisées. Dans cette thèse, nous nous concentrons sur le *test de conformité* [ISO/IEC, 1992] appliqué aux systèmes réactifs [Harel and Pnueli, 1985]. Par *système réactif*, nous entendons un logiciel réagissant aux stimuli de son environnement. Le test de conformité consiste à vérifier si le comportement d'une implémentation réelle sous test *iut* d'un système réactif est correcte par rapport à une spécification formelle. Le code de l'*iut* est inconnu et son comportement est uniquement visible par interaction avec un testeur qui contrôle et observe l'*iut* à travers des interfaces spécifiques appelés *points de contrôle et d'observation* (PCO). Ceci explique la nature *boîte noire* du test de conformité.

Une approche théorique du test de conformité a été formulé dans les travaux de E. Brinsksma et J. Tretmans (*voir* par exemple, [Brinksma, 1988], [Tretmans, 1996b]). Cela permet de définir plusieurs notions: la *relation de conformité* qui fixe l'ensemble des implémentations sous test correctes par rapport à une spécification donnée, les *cas de test*, leur *exécution* sur une *iut* et les *verdicts de test* associés à l'exécution. Cependant, le processus d'écriture des cas de test manuels pour une spécification de grande taille est compliqué, cher et peut mener à des erreurs.

Ceci est l'une des raisons du développement intensif des techniques pour la génération automatique de tests fondées sur une base théorique solide. Certaines de ces techniques sont basées sur le modèle des systèmes de transitions à entrées/sorties (IOLTS[1]) et sur des algorithmes à la volée efficaces. Il existe déjà des outils universitaires (par exemple TorX [Belinfante et al., 1999], TGV [Fernandez et al., 1996]) et industriels (par exemple Autolink [Telelogic, 1998], TestComposer [Kerbrat and Ober, 1999]) qui implémentent ces algorithmes et produisent des cas de test corrects dans un cadre formel. Néanmoins, ces théories et outils ne prennent pas *explicitement* en compte les données du système car le modèle sous-jacent d'IOLTS ne permet pas de le faire. Ainsi, afin de modéliser une spé-

[1]Provient de la terminologie anglaise "Input-Output Labeled Transition System".

cification de systèmes réactifs par IOLTS, il est nécessaire d'énumérer les valeurs de chaque donnée utilisée par ce système. Ceci peut conduire au problème classique de l'explosion de l'espace d'états. De plus, cette énumération a aussi l'effet d'obtenir des cas de test où toutes les données sont instanciées. Cela contredit la pratique industrielle où les cas de test (écrits, par exemple, dans le langage TTCN [ISO/IEC/JTC1/SC21, 1992]) sont de vrais programmes avec des données (variables, constantes symboliques et paramètres de communication). La génération de tels cas de test exige de nouveaux modèles et techniques.

Dans cette thèse, nous avons atteint deux objectifs :

(1) Nous avons introduit un nouveau modèle appelé systèmes symboliques de transitions à entrées/sorties (IOSTS[2]) et qui inclut explicitement toutes les données d'un système réactif.

(2) Nous avons proposé et implémenté une nouvelle technique de génération de tests qui traite symboliquement toutes les données d'un système en combinant l'approche de génération de tests proposée dans les travaux précédents de notre équipe de recherche (*voir* par exemple, [Fernandez et al., 1996] et [Jéron, 2004]) avec l'interprétation abstraite (*voir* [Cousot and Cousot, 1976], [Cousot and Cousot, 1977]).

Ces travaux sont inspirés de l'article [Rusu et al., 2000].

1.2 État de l'art du test de conformité

La première partie de ce document consiste en deux chapitres décrivant l'état de l'art pour le test de conformité boîte noire.

Dans le premier chapitre de cette partie (le chapitre 2) nous introduisons les concepts formels utilisés dans la suite de cette thèse, c'est-à-dire que nous exposons les notions présentés dans "Formal Methods in Conformance Testing" [ISO/IEC, 1996], et nous évoquons brièvement les nouveaux développements de ces dernières années dans le domaine de la formalisation du test de conformité. Nous essayons en particulier de rendre le lecteur familier avec les principaux concepts utilisés dans le test de conformité boîte noire.

Dans le deuxième chapitre de cette partie (le chapitre 3) nous nous focalisons principalement sur le test des machines d'états finis et des systèmes de transitions à entrées/sorties. Nous présentons différents types de relations de conformité pour les IOLTS, ainsi que différentes approches pour la génération automatique de tests. Enfin, nous décrivons certains des outils existants utilisés

[2]Provient de la terminologie anglaise "Input-Output Symbolic Transition System".

pour le test de conformité de systèmes réactifs tels que TorX [Belinfante et al., 1999], TGV [Fernandez et al., 1996] et Agatha [Lugato et al., 2002].

1.3 Génération de tests symboliques

La seconde partie de ce document est le cœur de notre travail. Il comporte quatre chapitres dans lesquels nous introduisons un modèle symbolique utilisé afin de spécifier des systèmes réactifs, ainsi qu'une technique symbolique pour la génération de test basée sur ce modèle. Ici, nous résumons succinctement les points principaux dans cette partie de la thèse.

1.3.1 Modèle : système symbolique de transitions à entrées/sorties

Dans le chapitre 4 de cette thèse, nous définissons un modèle de systèmes réactifs que nous utilisons pour le test de conformité. Ce modèle est appelé système symbolique de transitions à entrées/sorties (IOSTS) et á été précédemment introduit dans [Rusu et al., 2000]. Le modèle des IOSTS est une version étendue du modèle des systèmes de transitions á entrées/sorties (IOLTS). Il inclut *explicitement* les données des systèmes réactifs et les manipule de façon *symbolique*.

Dans cette section, nous présentons une syntaxe des IOSTS au niveau intuitif (la syntaxe formelle de ce modèle est évoquée dans la Section 4.2, page 79). Puis, nous expliquons la sémantique des IOSTS et nous introduisons les notions de comportement et de trace, qui sont fréquemment employées tout au long de ce document. Enfin, nous présentons quelques sous-classes d'IOSTS utilisées plus tard en génération de tests symboliques.

1.3.1.1 Syntaxe des IOSTS

L'IOSTS $\mathcal{S}$ décrit dans la figure 1.1 est composé de *localités*, par exemple *Begin*, *Idle*, *Pay*, où *Begin* est la *localité initiale*, et de *transitions*. Les transitions sont étiquetées par des *actions*, des *gardes* et des *affectations*. Par exemple, la transition d'origine *Idle* et de destination *Pay* a la garde $(pCoinValue > 0)$, l'action d'entrée *Coin?* portant les données $pCoinValue$ depuis l'environnement ainsi que l'affectation $vPaid := vPaid + pCoinValue$. L'ensemble des actions est partitionné en trois sous-éléments disjoints d'actions d'*entrée*, de *sortie* et *internes*. Les actions d'entrée/sortie interagissent avec l'environnement et peuvent porter des données tandis que des actions internes sont utilisées pour des calculs internes. Par convention, les noms des actions d'entrée (*resp.* de sortie) s'achève par "?" (*resp.* "!"). L'IOSTS de la figure 1.1 a trois entrées : *Coin?*, *ChooseBeverage?*, *Cancel?*, deux sorties : *Deliver!*, *Return!*, et une action interne : *tau*. Il opère

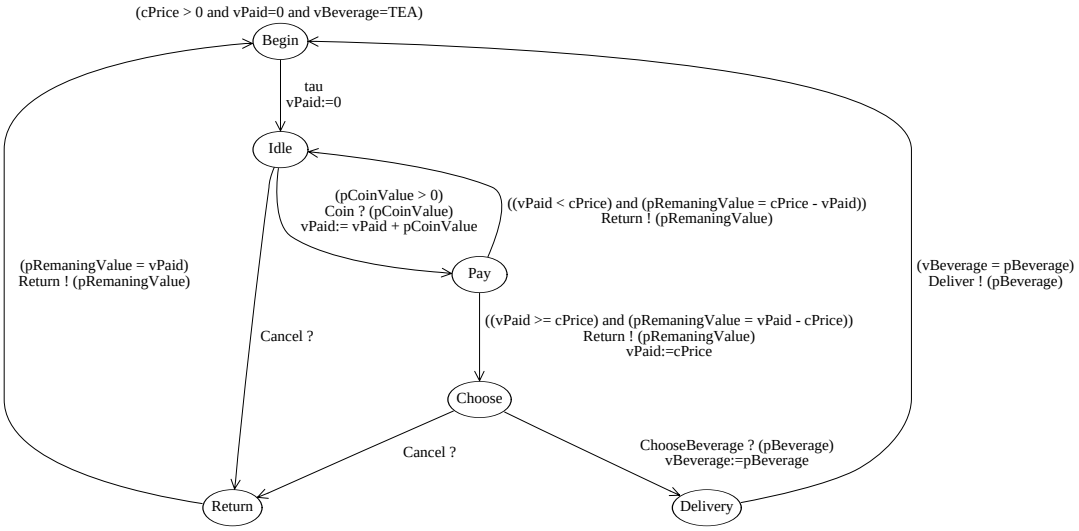


Figure 1.1: Un exemple d'IOSTS $\mathcal{S}$ (une machine à café).

avec des données symboliques qui sont les *variables* : $vPaid$, $vBeverage$, la *constante symbolique* : $cPrice$, et les *paramètres* : $pCoinValue$, $pRemaningValue$, $pBeverage$. Intuitivement, les *variables* sont des données avec lesquelles on calcule, les *constantes symboliques* sont des données qui ne peuvent pas être modifiées pendant la calcule et les *paramètres* sont des données pour communiquer avec l'environnement. La portée d'un paramètre est seulement la transition étiquetée par l'action qui porte ce paramètre. Ainsi, si la valeur du paramètre doit être utilisée dans des calculs ultérieurs, elle doit être mémorisée par son affectation à une variable. Par exemple, la valeur du paramètre $pBeverage$ est sauvée dans la variable $vBeverage$ en utilisant l'affectation $vBeverage := pBeverage$ de la transition conduisant à la localité *Delivery*. Puis, cette valeur, qui était mémorisée dans $vBeverage$, sera utilisée dans la garde $(vBeverage = pBeverage)$ de la transition sortant de *Delivery*.

1.3.1.2 Sémantique des IOSTS

Intuitivement la sémantique d'un IOSTS peut être représentée comme suit. Soit l'IOSTS (*voir* figure 1.1) représentant une machine à café. La machine part de la localité *Begin* avec une valeur du paramètre $cPrice$ satisfaisant la condition initiale $(cPrice > 0) \wedge (vPaid = 0) \wedge (vBeverage = \text{TEA})$, c'est-à-dire que le prix de n'importe quelle boisson distribuée dans la machine est positif, que la somme payée est égale à zéro, et que la boisson est du thé. Puis, l'IOSTS exécute

la transition étiquetée par l'action interne *tau*, il affecte la variable *vPaid*, qui mémorise la somme déjà réglée, à 0, et il atteint la localité *Idle*. Ensuite, la machine attend une pièce, dénotée par l'action d'entrée *Coin?* qui porte en *pCoinValue* la valeur de la pièce insérée. Dès que la machine reçoit une pièce, la variable *vPaid* est augmentée par *pCoinValue* et la machine va à la localité *Pay*. Si la somme n'est pas suffisante, c'est-à-dire $vPaid < cPrice$, la machine revient à la localité *Idle* et retourne (grâce à l'action de sortie *Return!*) la différence entre *vPaid* et *cPrice*. Dans la localité *Choose*, la machine attend le choix de la boisson (thé ou café), puis la distribue et revient à la localité *Begin*. Il convient de noter que dans les localités *Idle* et *Choose*, on peut appuyer sur le bouton *Cancel*, au quel cas la machine rend la somme déjà payée et revient à la localité initiale.

Plus formellement, la sémantique opérationnelle d'un IOSTS $\mathcal{M}$ est un IOLTS $[[\mathcal{M}]]$ (*voir* Définition 3.7 dans le Chapitre 3, page 41) dont les états, l'alphabet et la relation de transition sont définis comme suit. Un *état* s est une paire $\langle l, \vartheta \rangle$, où l est une localité et ϑ le vecteur de valeurs des variables et des constantes symboliques, par exemple, $s = \langle Delivery, \langle cPrice = 2, vPaid = 3, vBeverage = \text{TEA} \rangle \rangle$. Un *état initial* $s^0 = \langle l^0, \vartheta^0 \rangle$ est un état où l^0 est la localité initiale, et où ϑ^0 est un vecteur de valeurs des variables et des constantes symboliques qui satisfont à la condition initiale. Nous notons S (*resp.* S^0) l'ensemble de tous les états (*resp.* les états initiaux). Une *action valuée* α est une paire $\langle a, \omega \rangle$, où a est une action et où ω est un vecteur de valeurs des paramètres de a, par exemple, $\alpha = \langle Coin, \langle pCoinValue = 5 \rangle \rangle$ ou $\alpha = \langle tau, \langle \rangle \rangle$. Nous notons $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$ l'ensemble des actions valuées qui est partitionné en trois sous-ensembles d'entrée valuée, de sortie valuée et d'actions internes. Ensuite, nous définissons la *relation locale de transition* $\rightarrow_t \subseteq S \times \Lambda \times S$ comme l'ensemble de triplets $\langle s, \alpha, s' \rangle$, où $s = \langle l, \vartheta \rangle$, $s' = \langle l', \vartheta' \rangle$ sont des états et $\alpha = \langle a, \omega \rangle$ une action valuée telle que ϑ et ω sont des vecteurs de valeurs des variables, des constantes symboliques et des paramètres qui satisfont la garde d'une transition t d'origine l et de destination l' qui est étiqueté par l'action a, et telle que ϑ' est le nouveau vecteur de valeurs des variables et des constantes symboliques obtenues à partir de ϑ par les affectations de variables de t. La *relation globale de transition* $\rightarrow$ est $\bigcup_{t \in T}(\rightarrow_t)$, où T dénote l'ensemble des transitions symboliques de l'IOSTS. Nous écrivons $s \xrightarrow{\alpha} s'$ pour $\langle s, \alpha, s' \rangle \in \rightarrow$.

Comportements et traces. Dans cet paragraphe, nous introduisons les notions de comportements et de traces pour un IOSTS $\mathcal{M}$ arbitraire avec un ensemble S d'états, un ensemble $S^0 \subseteq S$ d'états initiaux, et un ensemble $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$ d'actions valuées. Ces notions nous permettent de raisonner formellement à propos de l'IOSTS $\mathcal{M}$.

Définition 1.1 (Comportement et ensemble des comportements) Un *comportement* β est une séquence d'états et d'actions valuées partant d'un état initial et suivant la relation de transition, c'est-à-dire :

$$\beta^{\rightarrow} \ : \quad s^0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots s_{n-1} \xrightarrow{\alpha_n} s_n$$

où $\rightarrow$ est la relation globale de transition, $s^0 \in S^0$, et pour tout $i \in [1, n]$: $s_i \in S$, $\alpha_i \in \Lambda$. On note $Behaviors(\mathcal{M})$ l'*ensemble* de tous les *comportements* de l'IOSTS $\mathcal{M}$. □

Afin de définir la notion de *trace* de l'IOSTS $\mathcal{M}$ nous introduisons d'abord la *relation de trace* $\Rightarrow \subseteq S \times (\Lambda^? \cup \Lambda^!)^* \times S$ comme suit :

- $s \overset{\varepsilon}{\Rightarrow} s' \quad \triangleq \quad (s = s') \vee (\exists s_1, \ldots, s_n \in S.\ [s = s_1 \xrightarrow{\tau_1} s_2 \ldots s_{n-1} \xrightarrow{\tau_n} s_n = s'])$, où pour tout $i = 1..n : \tau_i \in \Lambda^\tau$, et ε est la séquence vide.
- $s \overset{\alpha}{\Rightarrow} s' \quad \triangleq \quad \exists s_1, s_2 \in S.\ [s \overset{\varepsilon}{\Rightarrow} s_1 \xrightarrow{\alpha} s_2 \overset{\varepsilon}{\Rightarrow} s']$, où $\alpha \in (\Lambda^? \cup \Lambda^!)$.
- $s \overset{\sigma}{\Rightarrow} s' \quad \triangleq \quad \exists s_1, \ldots, s_n \in S.\ [s = s_1 \overset{\alpha_1}{\Rightarrow} s_2 \ldots s_{n-1} \overset{\alpha_{n-1}}{\Rightarrow} s_n = s']$, où pour tout $i = 1..n-1 : \alpha_i \in (\Lambda^? \cup \Lambda^!)$, et $\sigma \in (\Lambda^? \cup \Lambda^!)^*$.

Définition 1.2 (Ensemble de traces) L'*ensemble de traces* de $\mathcal{M}$ est défini comme suit :

$$Traces(\mathcal{M}) \quad \triangleq \quad \{\sigma \in (\Lambda^? \cup \Lambda^!)^* \mid \exists\ s^0 \in S^0, s \in S\ .\ [s^0 \overset{\sigma}{\Rightarrow} s]\}$$

□

1.3.1.3 Sous-classes d'IOSTS

Dans cette section, nous définissons des sous-classes d'IOSTS qui sont utilisées dans la méthode symbolique de génération de test présentée au Chapitre 7.

IOSTS instanciés et initialisés. Le fait de disposer de constantes symboliques dans le modèle IOSTS permet de décrire des spécifications génériques et, on le verra, de produire des tests eux-mêmes génériques. Cependant, au cours de l'exécution des tests, les valeurs de toutes les constantes symboliques et de toutes les variables doivent toujours être définies. Par conséquent, nous avons besoin d'introduire deux sous-classes d'IOSTS appelés IOSTS instanciés et initialisés.

Soit $\mathcal{M}$ un IOSTS avec un ensemble C de constantes symboliques, un ensemble V de variables et une condition initiale Θ. Soit aussi $\varsigma \in \mathsf{DOM}(C)$, où $\mathsf{DOM}(C)$ est un domaine de C, un vecteur de valeurs de constantes symboliques C. Alors :

(1) $\mathcal{M}$ est dit *instancié* si son ensemble C de constantes symboliques est vide.

(2) En remplaçant chaque constante symbolique $c \in C$ par une valeur $\varsigma(c)$, on obtient un IOSTS noté $\mathcal{M}(\varsigma)$ et appelé un *instance* de $\mathcal{M}$.

(3) $\mathcal{M}$ est dit *initialisé* si pour chaque instance $\mathcal{M}(\varsigma)$, il existe *au plus un* vecteur de valeurs de ses variables V qui satisfait à la condition initiale Θ.

Il est important de remarquer que chaque instance d'un IOSTS initialisé a un *état initial unique.*

IOSTS déterministe. Par la suite, nous interdisons les cas de test non-déterministes dans le test, car un verdict de test ne doit pas dépendre des choix internes du testeur. Par conséquent, dans ce paragraphe, nous introduisons une sous-classe d'IOSTS déterministes qui seront utilisés ultérieurement pour la représentation des cas de test.

Un IOSTS $\mathcal{M}$ est *déterministe* s'il ne contient pas d'actions internes, c'est-à-dire $\Lambda^\tau = \emptyset$, et si une transition au plus peut être exécutée à partir de chaque état, autrement dit, $\forall\ s \in S,\ \alpha \in (\Lambda^? \cup \Lambda^!)\ .\ [card(\{s' \in S \mid s \xrightarrow{\alpha} s'\}) \leq 1]$.

Il est important de remarquer que pour chaque trace σ de chaque instance de l'IOSTS initialisé et déterministe $\mathcal{M}$, l'ensemble d'états $\mathcal{M}$ after σ est un singleton. Ici, $\mathcal{M}$ after $\sigma \triangleq \{s \in S \mid \exists\ s^0 \in S^0\ .\ [s^0 \stackrel{\sigma}{\Rightarrow} s]\}$, où S (*resp.* S^0) est un ensemble d'états (*resp.* d'états initiaux) de $\mathcal{M}$. Cette proposition est formulée dans le théorème 4.1 au chapitre 4 (*voir* page 94).

IOSTS complets et complets en entrée. Dans ce paragraphe, nous introduisons deux sous-classes des IOSTS qui seront nécessaires durant la génération de test et le processus d'exécution de tests.

En effet, pour un IOSTS $\mathcal{M}$ représentant soit un cas de test soit une implémentation sous test, nous devons supposer qu'il accepte toujours (qu'il ne bloque pas) les entrées. Un tel IOSTS est appelé *complet en entrée* et défini comme suit : $\forall\ s \in S, \alpha \in \Lambda^?\ \exists\ s' \in S\ .\ [s \stackrel{\alpha}{\Rightarrow} s']$, où S est un ensemble d'états de $\mathcal{M}$ et $\Lambda^?$ est l'ensemble des actions d'entrée valuées de $\mathcal{M}$.

Par analogie avec l'approche de génération de test proposée par dans [Fernandez et al., 1996], notre méthode symbolique de génération décrite au chapitre 7 de ce document utilise un objectif de test comme mécanisme de sélection de test. En d'autres termes, le but des objectifs de test est de marquer les traces d'une spécification donnée qui devraient être testées, sans modifier l'ensemble de toutes les traces de cette spécification. Pour cela, chaque objectif de test TP doit être *complet* par rapport sa spécification *Spec*, c'est-à-dire que pour chaque état $s \in S_{TP}$ et pour chaque action valuée $\alpha \in (\Lambda_{TP} = \Lambda_{Spec})$, l'ensemble $\{s' \in S_{TP} \mid s \xrightarrow{\alpha} s'\}$ n'est pas vide, et la condition initiale Θ_{TP} de l'objectif de test TP ne contient

pas contraintes sur les variables et les constantes symboliques de la spécification *Spec*.

1.3.2 Opérations sur les IOSTS

Cette section résume les deux opérations principales sur les IOSTS définies au chapitre 5. L'*opération de composition parallèle* est utilisée dans l'exécution du test sur un système sous test car elle permet de modéliser une interaction entre deux processus représentés par des IOSTS. Cette opération est inspirée de la composition parallèle de deux processus modélisés soit par LTS, soit par IOLTS (*voir* par exemple, [Tretmans, 2002]). L'*opération produit* est l'opération principale de la génération de tests car elle permet une interaction entre les comportements d'une spécification et son objectif de test, et donc de sélectionner la partie de la spécification pour laquelle un cas de test doit être généré. Cette opération est inspirée de l'opération produit définie sur les IOLTS et utilisée dans la méthode de génération de tests décrite, par exemple, dans [Jéron, 2004]. La différence entre ces deux opérations réside dans le fait que le produit défini sur les IOSTS effectue une synchronisation non seulement sur les actions communes de deux systèmes donnés, mais également sur les données de ces systèmes (rappelons que le modèle IOSTS inclut explicitement les données du système). Ainsi, cette différence nous permet d'accomplir une sélection plus précise du cas de test.

Composition parallèle. L'opération de composition parallèle permet à chaque système d'exécuter indépendemment ses actions internes et impose une synchronisation sur les actions d'entrée et de sortie partagées. L'opération est définie pour des IOSTS compatibles. De manière informelle, deux IOSTS $\mathcal{M}_1$ et $\mathcal{M}_2$ sont *compatibles* si (1) ils n'ont pas de données communes, c'est-à-dire $D_1 \cap D_2 = \emptyset$, (2) l'alphabet des actions d'entrée (*resp.* de sortie) de $\mathcal{M}_1$ est égal à l'alphabet des actions de sortie (*resp.* d'entrée) de $\mathcal{M}_2$, et si les alphabets d'actions internes de $\mathcal{M}_1$ et $\mathcal{M}_2$ sont disjoints, (3) en outre, les actions communes doivent porter des paramètres du même type et en nombre égal dans les deux IOSTS. En outre, la définition formelle de compatibilité pour la composition parallèle est donnée dans la section 5.1 du chapitre 5.

Définition 1.3 (Composition parallèle) La *composition parallèle* entre deux IOSTS $\mathcal{M}_1 = \langle D_1, \Theta_1, L_1, l_1^0, \Sigma_1, T_1 \rangle$ et $\mathcal{M}_2 = \langle D_2, \Theta_2, L_2, l_2^0, \Sigma_2, T_2 \rangle$ compatibles pour la composition parallèle est un IOSTS $\mathcal{M}_1 \mid\mid \mathcal{M}_2 = \langle D, \Theta, L, l^0, \Sigma, T \rangle$, où :

(1) $D = V \cup C \cup P$, où $V = V_1 \cup V_2$, $C = C_1 \cup C_2$ et $P = P_1$.

(2) $\Theta = \Theta_1 \wedge \Theta_2$;

(3) $L = L_1 \times L_2$;

(4) $l^0 = \langle l_1^0, l_2^0 \rangle \in L$ est la localité initiale;

(5) $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ est l'alphabet des actions, où $\Sigma^? = \emptyset$, $\Sigma^! = \Sigma_1^! \cup \Sigma_2^!$, et $\Sigma^\tau = \Sigma_1^\tau \cup \Sigma_2^\tau$.

(6) l'ensemble des transitions symboliques T est construit à partir de T_1 et T_2 comme suit :

(a) pour chaque transition symbolique $t_1 = \langle l_1, a, \pi, G_1, A_1, l_1' \rangle \in T_1$ étiquetée par l'action interne $a \in \Sigma^\tau$, et pour chaque localité de la forme $\langle l_1, l_2 \rangle$ où $l_2 \in L_2$, l'IOSTS composé $\mathcal{M}_1 \,||\, \mathcal{M}_2$ a une transition symbolique de la forme $t = \langle \langle l_1, l_2 \rangle, a, \pi, G_1, A_1 \cup (\bigcup_{v \in V_2} (v := v)), \langle l_1', l_2 \rangle \rangle \in T$, où, afin d'obtenir une transition symbolique avec des ensembles d'affectations bien formés, nous affectons chaque variable de $\mathcal{M}_2$ à elle-même.

De la même manière, nous construisons une transition symbolique de $\mathcal{M}_1 \,||\, \mathcal{M}_2$ pour chaque transition symbolique $t_2 \in T_2$ sortant de $l_2 \in L_2$ et étiquetée par l'action interne $a \in \Sigma_2^\tau$, et pour chaque localité de la forme $\langle l_1, l_2 \rangle \in L$, où $l_1 \in L_1$.

(b) pour deux transitions symboliques $t_1 = \langle l_1, a, \pi, G_1, A_1, l_1' \rangle \in T_1$ et $t_2 = \langle l_2, a, \pi, G_2, A_2, l_2' \rangle \in T_2$ étiquetées par une action *commune* $a \in (\Sigma_1^? \cup \Sigma_1^!) = (\Sigma_2^? \cup \Sigma_2^!)$, une nouvelle transition symbolique $t = \langle \langle l_1, l_2 \rangle, a, \pi, G_1 \wedge G_2[\pi_2/\pi_1], A_1 \cup A_2[\pi_2/\pi_1], \langle l_1', l_2' \rangle \rangle \in T$ de $\mathcal{M}_1 \,||\, \mathcal{M}_2$ est construite, où $G_2[\pi_2/\pi_1]$ (*resp.* $A_2[\pi_2/\pi_1]$) est la garde (*resp.* l'ensemble des affectations) de la transition symbolique t_2 dans laquelle chaque paramètre $p_2^i \in \pi_2$ porté par l'action a de t_2 est remplacée par le paramètre correspondant $p_1^i \in \pi_1$ porté par l'action a de t_1.

□

Produit. L'opération produit est la principale opération dans notre méthode symbolique de génération de tests. Elle provoque "l'intersection" des comportements des deux IOSTS représentant, dans l'algorithme de génération de tests, une spécification et un objectif de test. Cela permet de sélectionner une partie de la spécification pour laquelle un cas de test doit être généré. L'opération produit est définie pour des IOSTS compatibles.

De manière informelle, deux IOSTS $\mathcal{M}_1$ et $\mathcal{M}_2$ sont *compatibles pour l'opération produit* si (1) ils ont les mêmes alphabets d'entrée, de sortie et d'actions internes et (2) $\mathcal{M}_1$ et $\mathcal{M}_2$ ne partagent aucune variable et aucun paramètre, mais

peuvent partager des constantes symboliques; de plus, les variables de $\mathcal{M}_1$ peuvent être des constantes symboliques de $\mathcal{M}_2$, et les variables de $\mathcal{M}_2$ peuvent être des constantes symboliques de $\mathcal{M}_1$. Intuitivement, le deuxième point signifie que nous interdisons que l'un des IOSTS change la valeur de l'une des variables de l'autre IOSTS, mais nous donnons la possibilité à un IOSTS d'observer les variables de l'autre IOSTS. On trouvera la définition formelle des IOSTS compatibles pour l'opération produit à la section 5.2 du chapitre 5.

L'IOSTS $\mathcal{M}_1 \times \mathcal{M}_2 = \langle D, \Theta, L, l^0, \Sigma, T\rangle$ qui est obtenu à partir de deux IOSTS compatibles $\mathcal{M}_1 = \langle D_2, \Theta_2, L_2, l_2^0, \Sigma_2, T_2\rangle$ et $\mathcal{M}_2 = \langle D_1, \Theta_1, L_1, l_1^0, \Sigma_1, T_1\rangle$ par l'opération de produit peut être défini par analogie avec $\mathcal{M}_1 \mid\mid \mathcal{M}_2$ (*voir* définition 1.3, page viii), où l'ensemble des données, l'alphabet des actions et l'ensemble des transitions est définit comme suit :

(1) $D = V \cup C \cup P$, où $V = V_1 \cup V_2$, $C = (C_1 \cup C_2) \setminus (V_1 \cup V_2)$ et $P = P_1$.

(2) $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$, où $\Sigma^? = \Sigma_1^? = \Sigma_2^?$, $\Sigma^! = \Sigma_1^! = \Sigma_2^!$ et $\Sigma^\tau = \Sigma_1^\tau = \Sigma_2^\tau$.

(3) L'ensemble de transitions symboliques T est obtenu à partir de T_1 et T_2 ainsi qu'il est expliqué ci-dessous.

Pour deux transitions symboliques $t_1 = \langle l_1, a, \pi, G_1, A_1, l_1'\rangle \in T_1$ et $t_2 = \langle l_2, a, \pi, G_2, A_2, l_2'\rangle \in T_2$ étiquetées par une action *commune* $a \in \Sigma_1 = \Sigma_2$, une nouvelle transition symbolique $t = \langle\langle l_1, l_2\rangle, a, \pi, G_1 \wedge G_2[\pi_2/\pi_1], A_1 \cup A_2[\pi_2/\pi_1], \langle l_1', l_2'\rangle\rangle \in T$ de $\mathcal{M}_1 \mid\mid \mathcal{M}_2$ est construite, où $G_2[\pi_2/\pi_1]$ (*resp.* $A_2[\pi_2/\pi_1]$) est la garde (*resp.* l'ensemble des affectations) de la transition symbolique t_2 dans laquelle chaque paramètre $p_2^i \in \pi_2$ porté par l'action a de t_2 est remplacé par le paramètre correspondant $p_1^i \in \pi_1$ porté par l'action a de t_1.

Traces de la composition parallèle et du produit. Nous nous intéressons aux relations suivantes entre les traces des systèmes obtenus par les opérations de composition parallèle et de produit et celles de leurs composants. Formellement :

Proposition 1.1 Pour deux IOSTS $\mathcal{M}_1$ et $\mathcal{M}_2$ qui sont compatibles pour l'opération de composition parallèle (*resp.* de produit) :

(1) $\mathit{Traces}(\mathcal{M}_1 \mid\mid \mathcal{M}_2) = \mathit{Traces}(\mathcal{M}_1) \cap \mathit{Traces}(\mathcal{M}_2)$, et

(2) si $\mathcal{M}_2$ est complet par rapport à $\mathcal{M}_1$, alors $\mathit{Traces}(\mathcal{M}_1 \times \mathcal{M}_2) = \mathit{Traces}(\mathcal{M}_1)$.

□

Le premier (*resp.* second) point de cette proposition est formulé comme le théorème 5.2, page 112 (*resp.* théorème 5.5, page 134) du chapitre 5.

1.3.3 Test de conformité avec les IOSTS

Dans cette section (qui est un bref résumé du Chapitre 6), nous décrivons la théorie du test de conformité qui sert de base à la méthode de génération de tests symboliques présentée dans la section suivante et implémentée comme l'outil de génération de tests symboliques (STG[3]). Le travail exposé dans cette section s'inspire principalement de la théorie de test de conformité développée par J. Tretmans (*voir* par exemple, [Tretmans, 1994], [Tretmans, 1996b]) et de la recherche effectuée au sein de l'équipe VerTeCs, IRISA (voir [Rusu et al., 2000], [Morel, 2000], [Jard and Jéron, 2002]). Dans notre théorie du test de conformité, les comportements des spécifications et des implémentations sous test sont modélisés par des systèmes symboliques de transitions à entrées/sorties, et les relations de conformité sont définies comme inclusion partielle de leurs traces. À la fin de cette section, nous explicitons la notion de cas de test corrects en ce qui concerne les spécifications et les objectifs de test.

1.3.3.1 Spécification

La spécification d'un système réactif est la description formelle des comportements du système qui sont généralement exprimés en utilisant des langages de description spécialisés, par exemple, SDL [ITU-T, 1994], LOTOS [ISO/IEC, 1988]. La sémantique opérationnelle de ces langages détaille tous les comportements possibles du langage.

Formellement, une *spécification Spec* est modélisée par un IOSTS $Spec = \langle D_{Spec}, \Theta_{Spec}, L_{Spec}, l^0_{Spec}, \Sigma_{Spec}, T_{Spec} \rangle$ initialisé (*voir* la définition donnée page vi) Dans cette thèse, nous considérons (pour le processus de génération de test) des spécifications sans cycles d'actions internes durant lesquelles le système exécute ses calculs internes et ne communique pas avec son environnement.

L'exemple d'un IOSTS représentant la spécification d'une machine à café est donné figure 1.1 (*voir* page iv).

1.3.3.2 Implémentation

L'implémentation sous test *iut*[4] est un système physique, par exemple des composants logiciels ou matériels. Comme dans le cas énuméré (*voir* par exemple les travaux de J. Tretmans [Tretmans, 1992], [Tretmans, 1996b] ou de T. Jéron [Jard and Jéron, 2002], [Jéron, 2004]), on suppose que l'*iut* peut être modélisée (c'est une "hypothèse de test" habituelle), afin de pouvoir raisonner sur la conformité de celle-ci avec une spécification. Dans notre recherche, on suppose qu'une implémentation sous test est modélisée par l'IOSTS $\mathcal{I}_{iut} = \langle D_{\mathcal{I}_{iut}}, \Theta_{\mathcal{I}_{iut}}, L_{\mathcal{I}_{iut}},$

[3]Provient de la terminologie anglaise "Symbolic Test Generator".
[4]Provient de la terminologie anglaise "Implementation Under Test".

$l^0_{\mathcal{I}_{iut}}, \Sigma_{\mathcal{I}_{iut}}, T_{\mathcal{I}_{iut}} \rangle$, dont on ne connaît que l'alphabet d'actions d'entrée et de sortie, et on suppose que (1) le type et le nombre de paramètres de ces actions sont les mêmes que celles d'une spécification *Spec* donnée, et (2) les paramètres de *iut* sont en bijection avec les paramètres de *Spec*. On peut trouver la définition formelle d'une implémentation sous test à la section 6.2 du chapitre 6.

1.3.3.3 Objectif de test

Par analogie avec l'approche de génération de tests proposée dans [Fernandez et al., 1996], notre méthode de génération de tests symboliques exposée au chapitre 7 de ce document, utilise le concept d'objectif de test. Un objectif de test décrit des comportements d'un système donné devant être testé, et sert à sélectionner une partie de la spécification du système pour laquelle un cas de test sera généré.

Formellement, soit un IOSTS *Spec* modélisant une spécification. Alors, un objectif de test de *Spec* est un IOSTS *TP*[5] avec la localité spéciale *Accept* tel que : *TP* est initialisé (*voir* page vi) et complet par rapport à *Spec* (*voir* page vii). En outre, nous devrions être capable d'exécuter l'opération produit entre *TP* et *Spec*, c'est-à-dire que *TP* devrait être compatible pour l'opération produit avec *Spec* (*voir* le paragraphe appelé **Produit** à la page page ix).

Exemple 1.1 (Objectif de test) L'IOSTS représenté figure 1.2 est un objectif de test pour la spécification d'une machine à café montrée à la figure 1.1 (*voir* page iv). Les transitions de l'objectif de test, représentées par une ligne discontinue, sont générées automatiquement (*voir* l'algorithme décrit à la section 7.1 du chapitre 7, page 169) afin d'obtenir un objectif de test complet quant à ses spécifications (*voir* la définition de la page vii).

L'objectif de test donné décrit des comportements où la machine distribue du café, où l'utilisateur introduit au moins la somme requise en une seule fois, et où il n'annule pas sa commande (*voir* les transitions continues du graphe montré à la figure 6.5). Un comportement accepté est indiqué par l'arrivée à la localité

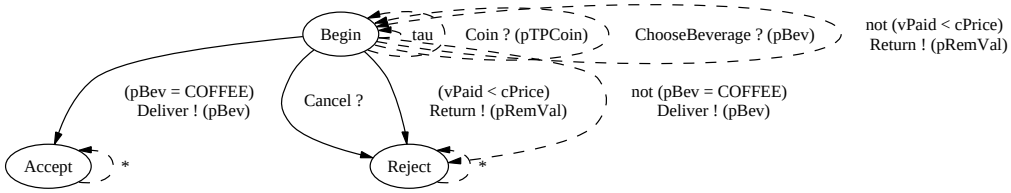


Figure 1.2: Objectif de test.

Accept. L'objectif de test rejette des comportements qui correspondent à l'action

[5]Provient de la terminologie anglaise "Test Purpose".

d'annulation (*Cancel*), ou à l'insertion de plus d'une pièce. Les comportements rejetés ne sont pas nécessairement faux, mais ils ne sont pas ciblés par l'objectif de test. Notons aussi que l'objectif de test *TP* observe, mais ne modifie pas la variable *vPaid* de la spécification *Spec*. □

1.3.3.4 Relation de conformité

Dans la sous-section précédente, nous avons défini les notions de spécification, d'implémentation sous test et d'objectif de test. Nous avons également supposé qu'ils pouvaient tous être exprimés dans le modèle des IOSTS. Ce postulat nous permet de raisonner formellement à propos des spécifications, des implémentations sous test et des objectifs de test.

Par conséquent, nous pouvons exprimer la conformité des implémentations par rapport aux spécifications (et possiblement "filtrée" avec un objectif de test) par des relations de conformité formelles entre leurs modèles. Une relation de conformité définit exactement l'ensemble des implémentations conformes à une spécification donnée.

Les relations de conformité introduites dans cette section et utilisées tout au long de cette thèse sont des versions plus faibles des relations de conformité ioconf et ioco exposées par J. Tretmans et al. pour des IOLTS (*voir* [Tretmans, 1995], [Tretmans, 1996b], [Tretmans, 2002]). Elles sont appelées ioc et ioc_{TP}. Nous définissons ces relations de conformité au lieu d'utiliser l'un de leurs prédécesseurs plus puissants (ioconf et ioco) car :

(1) ioc et ioc_{TP} ne prennent en compte aucun blocage syntaxique (c'est-à-dire blocages de sortie (ou outputlock), blocages complets (ou deadlocks) et blocages vivants (ou livelocks) présentés page 42) des IOSTS, et

(2) le problème de décider si un système représenté par un IOSTS est bloqué ou pas est *indécidable en général*. Cependant, dans le cas d'un IOSTS sans blocages vivants syntaxiques, il est possible de construire syntaxiquement l'IOSTS suspendu correspondant (un IOSTS suspendu est défini de manière semblable à un IOLTS suspendu, voir définition 3.9 page 44). Il est donc possible d'étendre la relation ioc à la relation ioco définie par J. Tretmans. L'idée de l'extension à déjà été proposée dans [Rusu et al., 2004].

Avant de donner la définition formelle des relations de conformité ioc et ioc_{TP} pour des spécifications, des objectifs de test et des implémentations *instanciés*, nous introduisons auparavant :

(1) l'*ensemble des actions de sortie valuées* pouvant être générées par un IOSTS $\mathcal{M}$ avec un ensemble d'états S, quand $\mathcal{M}$ est dans un état s' parmi un

ensemble d'états $S' \subseteq S$ est défini comme suit : $Out(S') \triangleq \{\alpha \in \Lambda^! \mid \exists\ s' \in S', s \in S\ .\ [s' \xrightarrow{\alpha} s]\}$.

(2) l'*ensemble des comportements acceptés* d'un IOSTS SP[6]$= (Spec \times TP)$ qui contient les comportements de la spécification *Spec* sélectionnés par l'objectif de test *TP* de *Spec* grâce à l'opération produit, défini comme suit :

$$\begin{aligned} ABehaviors(SP) \triangleq \{\ & \beta^{\rightarrow} : s^0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots s_{n-1} \xrightarrow{\alpha_n} s^{acc} \mid \\ & s^0 \in S^0_{SP} \ \wedge\ s^{acc} \in S^{acc}_{SP} \ \wedge\ \alpha_n \in \Lambda_{SP} \ \wedge \\ & \forall\ i \in [1, n-1]\ .\ [s_i \in S_{SP} \ \wedge\ \alpha_i \in \Lambda_{SP}]\ \} \end{aligned}$$

où Λ_{SP} est l'ensemble des actions valuées de SP, S_{SP} est l'ensemble des états de SP, S^0_{SP} est l'ensemble des états initiaux de SP, et S^{acc}_{SP} est l'ensemble des états accepteurs de SP défini comme suit : $\{s \in S_{SP} \mid \exists\ \langle l_{Spec}, Accept\rangle \in L_{SP},\ \vartheta \in \mathsf{DOM}(V_{SP} \cup C_{SP})\ .\ [s = \langle\langle l_{Spec}, Accept\rangle, \vartheta\rangle]\}$.

Il est important de remarquer que l'opération produit nous permet de sélectionner précisément les comportements de la spécification acceptée par l'objectif de test. En effet, d'après cette définition, on sait qu'un comportement accepté du produit synchrone SP est, à une projection près, un comportement accepté de l'objectif de test TP. De plus, d'après les théorèmes 5.3 et 5.4 (*voir* pages 126 et 132), sa projection est un comportement de la spécification *Spec*. Intuitivement, $ABehaviors(Spec \times TP)$ et $Behaviors(Spec) \cap ABehaviors(TP)$ sont égaux à une projection près.

(3) l'*ensemble des traces acceptées* du produit synchrone $SP = (Spec \times TP)$ qui contient les traces de la spécification *Spec* sélectionnées par l'objectif de test *TP* de *Spec* grâce à l'opération produit, noté $ATraces(SP)$, est obtenu, à partir de l'ensemble des comportements acceptés de SP par la *projection* sur les alphabets d'actions valuées d'entrée et de sortie de SP.

Proposition 1.2 (Ensemble de traces acceptées de *SP*) Pour une spécification *Spec* et un objectif de test *TP* de *Spec*, l'ensemble des traces acceptées de leur produit $Spec \times TP$ est inclus dans l'ensemble des traces de *Spec* et dans l'ensemble des traces acceptées de *TP*, c'est-à-dire :

$$ATraces(\underbrace{Spec \times TP}_{SP}) \subseteq Traces(Spec) \cap ATraces(TP)$$

Nous remarquons que, pour les traces acceptées du produit synchrone, il est impossible d'obtenir l'égalité comme dans le cas des comportements acceptés á cause de la projection. □

[6]Provient de la terminologie anglaise "Synchronous Product".

(4) l'*ensemble des préfixes* des traces acceptées $ATraces(SP)$ est défini comme suit :

$$Pref(ATraces(SP)) \triangleq \bigcup_{\sigma \in ATraces(SP)} \{\sigma' \in (\Lambda^?_{SP} \cup \Lambda^!_{SP})^* \mid \exists\, \sigma'' \in (\Lambda^?_{SP} \cup \Lambda^!_{SP})^* \,.\, [\sigma = \sigma' \cdot \sigma'']\}$$

où $\Lambda^?_{SP}$ et $\Lambda^!_{SP}$ sont les ensembles d'actions d'entrée et de sortie valuées de SP, et où $\sigma' \cdot \sigma''$ est l'opération de concaténation entre deux mots σ' et σ''.

(5) l'*ensemble des préfixes stricts* de $ATraces(SP)$ qui ne sont pas des traces acceptées de SP, est défini comme suit :

$$SPref(ATraces(SP)) = (Pref(ATraces(SP)) \setminus ATraces(SP))$$

Définition 1.4 (Relations de conformité ioc et ioc_{TP}) Soient *Spec* une spécification instanciée, TP un objectif de test instancié de *Spec*, et *iut* une implémentation modélisée par une IOSTS instanciée $\mathcal{I}_{iut}$. Alors :

(1) l'implémentation est *conforme* à la spécification, noté $(\mathcal{I}_{iut}\ \mathsf{ioc}\ Spec)$, si pour toutes les traces $\sigma \in Traces(Spec)$: $Out(\mathcal{I}_{iut}\ \mathsf{after}\ \sigma) \subseteq Out(Spec\ \mathsf{after}\ \sigma)$.

(2) l'implémentation est *conforme* à la spécification *relativement à* l'objectif de test, noté $(\mathcal{I}_{iut}\ \mathsf{ioc}_{TP}\ Spec)$, si pour toutes les traces $\sigma \in SPref(ATraces(Spec \times TP))$: $Out(\mathcal{I}_{iut}\ \mathsf{after}\ \sigma) \subseteq Out(Spec\ \mathsf{after}\ \sigma)$. □

Intuitivement, une implémentation est conforme à une spécification donnée si, après chaque trace de la spécification, les actions de sortie possibles de l'implémentation sont inclues dans celles de la spécification. Dans le second cas, l'inclusion doit être toujours vraie après chaque préfixe strict d'une trace de la spécification qui est sélectionné par un objectif de test donné grâce à l'opération produit.

La définition des relations de conformité ioc et ioc_{TP} est étendue au cas général (autrement dit aux spécifications, aux cas de test et aux implémentations non-instanciées) dans les sections 6.3 et 6.8 du chapitre 6. Dans ces mêmes sections, le lecteur peut également trouver les exemples illustrant ces relations de conformité.

1.3.3.5 Cas de test

Les cas de test introduits dans ce paragraphe jouent un rôle central dans le test. Au cours de l'exécution, les cas de test interagissent avec les implémentations sous test, observent leurs sorties et, en se basant sur ces observations, génèrent des

verdicts de test. La définition formelle d'un cas de test est donnée à la section 6.4 du chapitre 6 (*voir* page 143). Nous ne donnons ici qu'un bref résumé de cette définition. Ainsi, un cas de test est un IOSTS TC[7] contenant trois ensembles disjoints de localités Pass, Inconclusive et Fail telles que : TC est initialisé (*voir* page vi), déterministe (*voir* page vii) et complet en entrée (*voir* page vii).

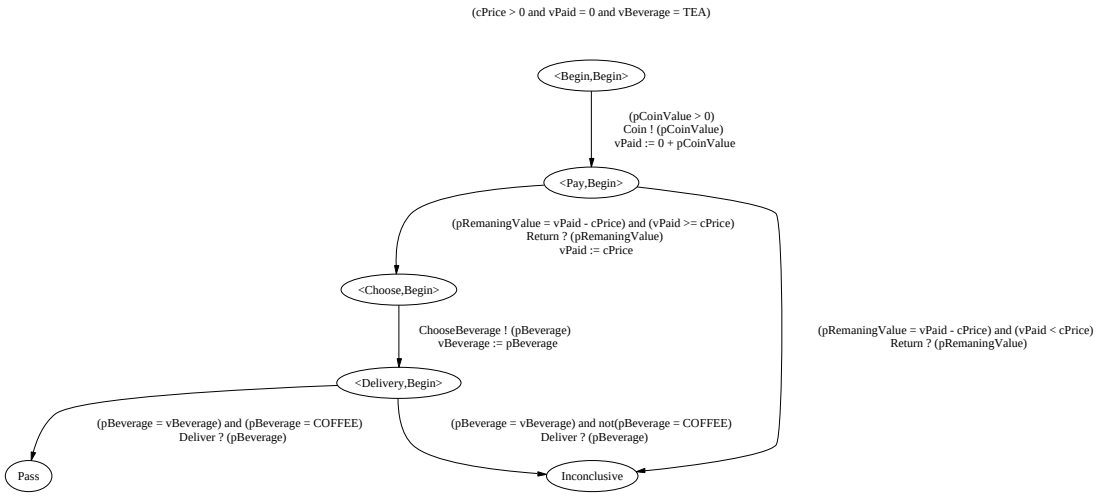


Figure 1.3: Cas de test.

Exemple 1.2 (Cas de test) La figure 1.3 présente un exemple de cas de test pour une machine à café obtenu à partir de la spécification montrée dans la figure 1.1 (*voir* page iv) et de l'objectif de test représenté sur la figure 1.2 (*voir* page xii). Il couvre tous les comportements de la spécification qui sont ciblés par l'objectif de test, c'est-à-dire qui acceptent un seul paiement et n'incluent pas la commande d'annulation. Notons que le cas de test montré à la figure 1.3 n'est pas complet en entrée car, à cause de l'espace limité de la figure 1.3, nous n'avons pas montré les localités étiquetées par le verdict *Fail*, dans lesquelles le cas de test arrive si une implémentation sous test produit une sortie non autorisée par la spécification. □

1.3.3.6 Exécution de test et verdict de test

L'*exécution de test* est le processus qui consiste à exécuter un cas de test TC sur une implémentation concrète sous test *iut*, d'observer des réponses de *iut*, et, basé sur ces réponses, de générer un *verdict de test*. Dans cette thèse, nous

[7]Provient de la terminologie anglaise "Test Case".

modélisons l'exécution de test par la *composition parallèle* entre TC et le modèle $\mathcal{I}_{iut}$ d'*iut*.

Avant de formaliser la notion de verdict de test, nous notons d'abord par PASS (*resp.* INCONCLUSIVE, FAIL) l'ensemble des états d'un cas de test donné dont les localités sont dans l'ensemble Pass (*resp.* Inconclusive, Fail).

Puis, nous considérons un cas de test TC et une implémentation sous test *iut* modélisée par $\mathcal{I}_{iut}$. Pour une trace $\sigma \in (\mathcal{I}_{iut} \parallel TC)$, le cas de test TC produit le *verdict Pass* (*resp. Inconclusive*, *Fail*) si l'état dans lequel TC arrive après σ appartient à PASS (*resp.* INCONCLUSIVE, FAIL). Formellement :

Définition 1.5 (Verdict de test) Soit σ une trace de $(\mathcal{I}_{iut} \parallel TC)$, alors :

$$\begin{array}{lll} (\mathsf{verdict}(\sigma) = \mathit{Pass}) & \textbf{if} & (TC \text{ after } \sigma \subseteq \mathsf{PASS}) \\ (\mathsf{verdict}(\sigma) = \mathit{Fail}) & \textbf{if} & (TC \text{ after } \sigma \subseteq \mathsf{FAIL}) \\ (\mathsf{verdict}(\sigma) = \mathit{Inconclusive}) & \textbf{if} & (TC \text{ after } \sigma \subseteq \mathsf{INCONCLUSIVE}) \end{array}$$

□

Nous remarquons que pour une implémentation donnée, un cas de test produit toujours le même verdict de test pour toute exécution d'une *même* trace sur cette implémentation. Ceci provient directement du fait que tout cas de test TC est un IOSTS initialisé et déterministe. Ainsi, il n'est pas difficile de montrer que pour chaque trace $\sigma \in \mathit{Traces}(\mathcal{I}_{iut} \parallel TC)$ (qui est aussi une trace de TC à cause de l'item (1) de la proposition 1.1, page x), tous les états de TC appartenant à $(TC \text{ after } \sigma)$ correspondent *exactement à une localité* du cas de test TC. Néanmoins, parce qu'une implémentation sous test peut avoir plusieurs réactions sur une entrée depuis le testeur, un cas de test peut produire plusieurs exécutions possibles donnant différents verdicts de test pour la même implémentation. Ainsi, une implémentation sous test peut être rejetée, acceptée, ou peut produire le verdict *Inconclusive* pour le même cas de test en suivant différentes traces possibles de la composition parallèle entre le cas de test et l'implémentation. De manière formelle : un cas de test TC *peut rejeter* une implémentation *iut* modélisée par $\mathcal{I}_{iut}$ s'il existe une trace σ dans la composition parallèle entre $\mathcal{I}_{iut}$ et TC après quoi TC produit le verdict *Fail*, c'est-à-dire :

$$(TC \text{ may\_fail } \mathcal{I}_{iut}) \quad \triangleq \quad \exists\, \sigma \in \mathit{Traces}(\mathcal{I}_{iut} \parallel TC)\ .\ [\mathsf{verdict}(\sigma) = \mathit{Fail}]$$

Les relations may_pass et may_inconc peuvent être définies de la même manière que la relation de rejet possible may_fail.

1.3.3.7 Propriétés attendue des cas de test

Dans les sous-sections précédentes nous avons introduit deux notions significatives utilisées en test de conformité : les cas de test (*voir* section 1.3.3.5, page xv) et

les relations de conformité (*voir* section 1.3.3.4, page xiii). L'objectif de cette sous-section est d'établir le lien entre ces deux notions définies indépendamment. Nous savons que ce lien existe. En effet, comme le but des cas de test est de nous donner des informations sur la conformité d'une implémentation par rapport à sa spécification, le cas de test doit conserver certaines propriétés de la relation de conformité. Par exemple, produire le verdict *Fail* doit impliquer la détection de non-conformité dans une implémentation sous test. On dit alors que le cas de test est *non biaisé*. En outre, nous définissons aussi plusieurs propriétés de cas de test qui nous permettent de relier les cas de test non seulement aux spécifications et aux implémentations, mais aussi aux objectifs de test.

La définition de telles propriétés est donnée plus bas. Il est essentiel de remarquer qu'elles sont ici formulées pour des spécifications, des objectifs de test, des cas de test et des implémentations *instanciées*. On trouvera les définitions générales de ces propriétés aux sections 6.6 et 6.10 du chapitre 6.

Définition 1.6 (Propriétés des cas de test) Soit *Spec* une spécification instanciée, *TP* un objectif de test instancié de *Spec*, *TC* un cas de test instancié généré automatiquement depuis *Spec* et *TP*, et *Iuts* un ensemble d'implémentations sous test instanciées. Alors :

(1) *TC* est *non biaisé* pour *Spec* et *Iuts*, si pour chaque implémentation $iut \in Iuts$ modélisée par $\mathcal{I}_{iut}$:

$$(\mathcal{I}_{iut} \ \mathsf{ioc} \ Spec) \implies \neg(TC \ \mathsf{may\_fail} \ \mathcal{I}_{iut})$$

(2) *TC* est *relativement complet* pour *Spec*, *TP* et *Iuts*, si pour chaque implémentation $iut \in Iuts$ modélisée par $\mathcal{I}_{iut}$:

$$\neg(\mathcal{I}_{iut} \ \mathsf{ioc}_{TP} \ Spec) \implies (TC \ \mathsf{may\_fail} \ \mathcal{I}_{iut})$$

(3) *TC* est *précis* pour *Spec*, *TP* et *Iuts*, si pour chaque implémentation $iut \in Iuts$ modélisée par $\mathcal{I}_{iut}$ et chaque trace $\sigma \in Traces(\mathcal{I}_{iut} \ || \ TC)$:

$$(\mathsf{verdict}(\sigma) = Pass) \implies (\sigma \in ATraces(Spec \times TP))$$

(3) *TC* est *concluant* pour *Spec*, *TP* et *Iuts*, si pour chaque implémentation $iut \in Iuts$ modélisée par $\mathcal{I}_{iut}$ et chaque trace $\sigma \in Traces(\mathcal{I}_{iut} \ || \ TC)$:

$$\begin{aligned}(\mathsf{verdict}(\sigma) = Inconclusive) \implies \ & \sigma \in (Traces(Spec) \cdot \Lambda^{!}_{Spec} \cap Traces(Spec)) \\ & \wedge \\ & \sigma \notin Pref(ATraces(Spec \times TP))\end{aligned}$$

□

Intuitivement, dire que le cas de test est *non biaisé* signifie qu'il ne rejette pas les implémentations conformes (dans un ensemble donné). Cette propriété peut être satisfaite en pratique, mais elle n'est pas suffisante en général. Par exemple, un cas de test acceptant toutes les implémentations possibles est non biaisé. L'*exhaustivité relative* signifie que le cas de test peut détecter, dans un ensemble donné, toutes les implémentations non conformes avec la spécification relativement à l'objectif de test. *Précision* signifie que le verdict *Pass* est donné lorsque la trace observée de l'implémentation est une trace de la spécification qui est sélectionnée par l'objectif de test. Cas de test *concluant* signifie que le verdict *Inconclusive* est donné quand la trace observée de l'implémentation est une trace de la spécification qui finit par une action de sortie mais aucun prolongement de cette trace ne peut produire le verdict *Pass*. Toutes les propriétés d'un cas de test sont illustrées par des exemples intuitifs aux sections 6.6 et 6.10 du chapitre 6.

Enfin, nous introduisons le concept de cas de test correct. Intuitivement, cela signifie que le cas de test donne toujours le bon verdict lorsqu'il est exécuté sur une implémentation sous test donnée. Formellement :

Définition 1.7 (Cas de test correct) Un cas de test *TC* est *correct* pour un ensemble *Iuts* d'implémentations s'il est non biaisé par rapport à *Spec* et *Iuts*; et relativement exhaustif, précis et concluant par rapport à *Spec*, *Iuts* et *TP* (*voir* définition 1.6, page xviii). □

1.3.4 Principes de la génération de tests symboliques

L'approche symbolique de génération de tests proposée dans cette thèse se décompose en plusieurs étapes résumées dans la figure 1.4 (*voir* page xx). Elle prend en entrée une spécification *Spec* et un objectif de test *TP* de *Spec*. Comme on accepte les objectifs de test incomplets, le premier pas de cette approche est de les rendre complets par rapport à leurs spécifications. Dans un second temps, on va calculer le produit entre la *Spec* donnée et le *TP* complété afin de marquer les comportements de *Spec* comme étant soit acceptés, soit rejetés par l'objectif de test, et on va obtenir le produit synchrone *SP* (*voir* figure 1.4, page xx). La troisième phase construit les comportements déterministes visibles de *SP*. En d'autres termes, elle va supprimer les actions internes et les choix non-déterministes possibles de *SP*. Le résultat de cette étape est noté SP_{vis} (*voir* figure 1.4, page xx). L'étape suivante, la sélection, consiste à extraire de SP_{vis} ses comportements permettant d'aller de la localité initiale aux localités *Pass* ou *Inconclusive*. Le résultat de la

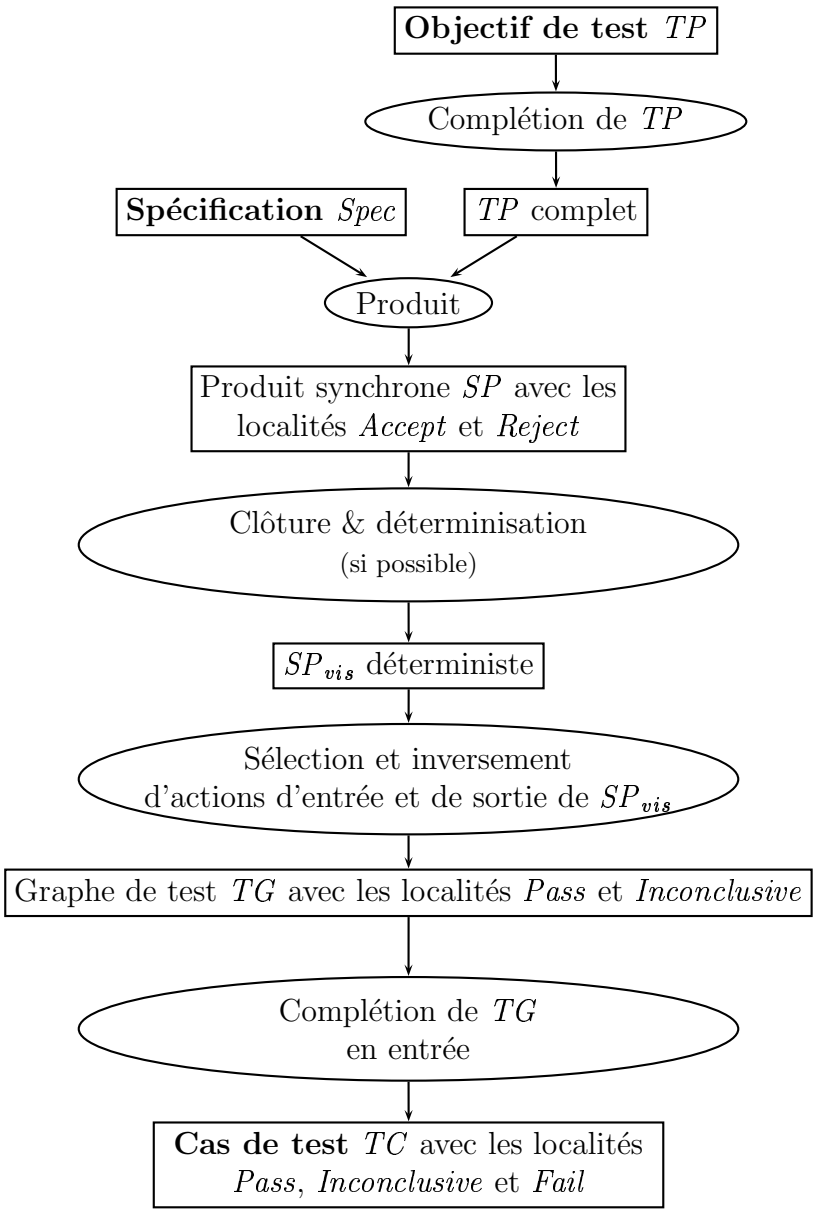


Figure 1.4: Génération de test symboliques.

sélection est appelé graphe de test et noté TG[8]. En outre, durant la sélection, nous inversons les alphabets des actions d'entrée et de sortie de TG rendant TG apte à communiquer avec les implémentations sous test au cours du processus d'exécution de test. Enfin, en rendant TG complet en entrée, nous obtenons un cas de test TC.

1.3.4.1 Rendre complet un cas de test

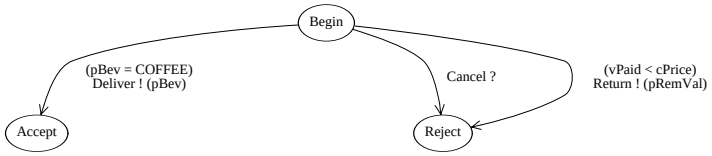
Dans la théorie du test présentée dans ce document, les cas de test, jouant le rôle de mécanisme de sélection, doivent être complets par rapport à leur spécification. En pratique, cette exigence complique souvent le processus d'écriture des objectifs de test. Ainsi, nous décidons d'accepter des objectifs de test incomplets de la part du programmeur et de les compléter automatiquement à la première étape du processus de génération de tests. Ceci nous permet de nous focaliser sur les comportements souhaités du système sous test en simplifiant sensiblement le processus d'écriture de l'objectif de test.

Néanmoins, il est facile d'assurer syntaxiquement la complétion de l'objectif de test par rapport à sa spécification. En effet, s'il existe une transition symbolique sortant d'une localité l, étiquetée par une action a et qui ne mène pas à la localité *Reject*, alors, nous ajoutons une transition symbolique de l à *Reject*, gardée par la négation de la disjonction de toutes les gardes des transitions étiquetées par l'action a et sortant de l. Sinon, nous ajoutons une boucle sur l étiquetée a. Le lecteur peut trouver l'algorithme qui rend complet un objectif de test donné par rapport à sa spécification à la section 7.1 du chapitre 7. Enfin, la figure 1.5(b) illustre un objectif de test complet obtenu à partir de la spécification montrée à la figure 1.1 et l'objectif de test dépeint à la figure 1.5(a) par l'algorithme expliqué intuitivement plus haut.

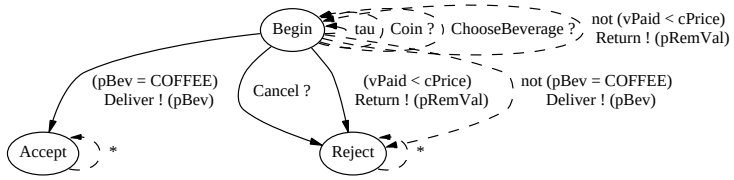
1.3.4.2 Produit synchrone

Pour l'étape suivante de notre méthode de génération de tests, nous calculons un produit synchrone SP entre une spécification *Spec* et un objectif de test TP complet relativement à *Spec* (*voir* section précédente). Le but de cette phase est d'identifier les comportements de *Spec* comme acceptés par TP. L'idée d'utiliser l'opération produit afin de marquer les comportements de *Spec* a déjà été employée en génération de tests, voir par exemple les articles suivants [Jéron and Morel, 1999], [Jard and Jéron, 2002]. Nous construisons le produit synchrone SP à partir des *Spec* et TP donnés en utilisant l'opération produit définie au paragraphe **Produit** page ix.

[8]Provient de la terminologie anglaise "Test Graph".



(a) Objectif de test TP.



(b) Objectif de test complet TP'.

Figure 1.5: Rendre complet un objectif de test donné TP par rapport à sa spécification $\mathcal{S}$ montrée à la figure 1.1 (*voir* page iv).

Il est important de souligner que comme *Spec* et TP sont compatibles pour l'opération produit, et que TP est complet par rapport à *Spec*, alors, à cause du second point de la Proposition 1.1 (*voir* page x), nous obtenons que l'opération produit conserve l'ensemble des traces de *Spec*, c'est-à-dire : $Traces(Spec) = Traces(\underbrace{Spec \times TP}_{SP})$.

Il est aussi important de signaler que les comportements acceptés du produit synchrone $Spec \times TP$ sont, à une projection près, les comportements de *Spec* acceptés par TP. Par contre, nous avons obtenu seulement l'inclusion des traces acceptées de $Spec \times TP$ dans l'intersection des traces de *Spec* et des traces acceptées de TP, c'est-à-dire : $ATraces(Spec \times TP) \subseteq Traces(Spec) \cap ATraces(TP)$. Cette inclusion provient de la projection (*voir* la proposition 1.2 page xiv).

Exemple 1.3 (Produit synchrone) La figure 1.6 (*voir* page xxiii) montre le résultat du calcul du produit pour la spécification d'une machine à café (*voir* figure 1.1, page iv) et pour l'objectif de test de cette spécification décrit à la figure 1.5(b) (*voir* page xxii). Le but de cet exemple est de souligner le fait que l'opération produit marque des localités de la spécification donnée avec *Accept*, rendant les comportements qui y mènent acceptés par l'objectif de test. Les comportements acceptés du produit calculé apparaîssent en vert sur

Figure 1.6: Produit synchrone $SP = Spec \times TP'$.

la figure 1.6 (*voir* page xxiii). Tous les autres comportements, imprimés en rouge, sont considérés comme rejetés. Les comportements rejetés indiquent les comportements de la spécification pour lesquels le cas de test ne sera pas généré. Ils seront éliminés au cours des étapes suivantes de la méthode de génération de tests. □

Les prochains pas de la méthode de génération de tests décrite dans le reste de ce chapitre consistent à transformer et à simplifier le produit $SP = (Spec \times TP)$ afin d'obtenir un cas de test correct dans le sens de la définition 1.7 (*voir* page xix).

1.3.4.3 Construction de comportements visibles

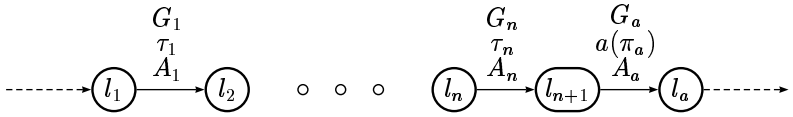
Il est important d'insister sur le fait que le test n'autorise pas le non-déterminisme car les verdicts de test ne doivent pas dépendre des choix internes du testeur. C'est pourquoi cette étape de la méthode de génération de tests est réservée à l'élimination des actions internes de l'IOSTS SP, c'est-à-dire à la construction de $\mathsf{closure}(SP)$, et à la résolution de choix non-déterministes restant pour les actions d'entrée/sortie de $\mathsf{det}(\mathsf{closure}(SP))$. Pour cela, nous proposons les opérations syntaxiques de clôture et de déterminisation telles que :

$$Traces(SP) = Traces(\mathsf{closure}(SP)) = Traces(\mathsf{det}(\mathsf{closure}(SP))) \quad (1.1)$$
$$ATraces(SP) = ATraces(\mathsf{closure}(SP)) = ATraces(\mathsf{det}(\mathsf{closure}(SP))) \quad (1.2)$$

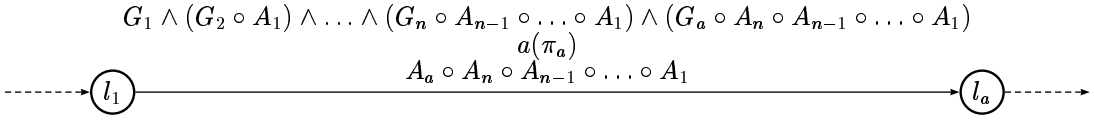
Les procédures de clôture et de déterminisation et leurs propriétés sont décrites aux sections A.1 et A.2 du chapitre 7, et brièvement résumées dans les deux paragraphes suivants.

Clôture : éliminer les actions internes. Pour éliminer des actions internes de SP, l'idée est de calculer l'effet de toute séquence d'actions internes qui mène à une transition symbolique étiquetée par une action d'entrée ou de sortie, et de coder cet effet dans la garde et les affectations de la dernière transition symbolique.

Ceci résulte en une procédure syntaxique simple qui termine si l'IOSTS SP n'a pas de blocages vivants syntaxiques (c'est-à-dire des cycles d'actions internes). Ceci est une hypothèse courante en test de conformité [Tretmans, 1999] posée pour l'ensemble de ce travail de recherche. Soit $l_1 \overset{\tau_1}{\hookrightarrow} l_2 \dots l_n \overset{\tau_n}{\hookrightarrow} l_{n+1}$ une séquence de transitions symboliques étiquetées par les actions internes $\tau_1, \dots \tau_n$ et menant à la transition symbolique $l_{n+1} \overset{a}{\hookrightarrow} l_{n+2}$ étiquetée par l'action a d'entrée ou de sortie (*voir* figure 1.7(a), page xxv). Supposons que les gardes et les affectations correspondant à τ_i $(i = 1..n)$ sont G_i et A_i; et que les gardes et les affectations correspondant à a sont G_a et A_a. Alors, cette séquence est remplacée par une



(a) Un fragment d'un IOSTS SP montrant une de ses séquences d'actions internes.



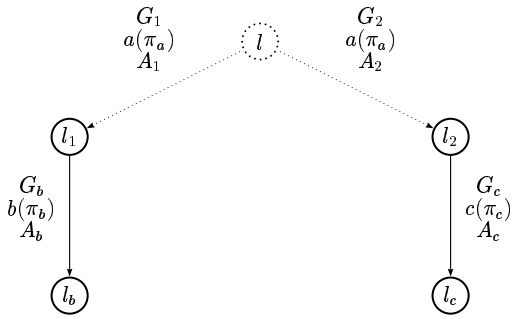
(b) Un fragment de l'IOSTS closure(SP) obtenu à partir de SP par la procédure de clôture.

Figure 1.7: Un exemple de l'IOSTS closure(SP).

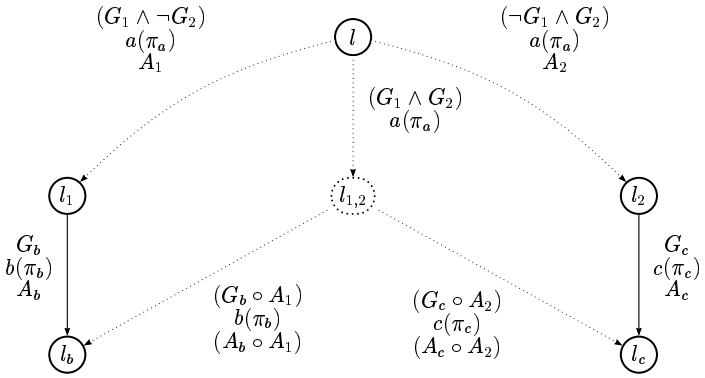
transition symbolique d'origine l_1, de destination l_{n+2}, d'action a, de garde $G_1 \wedge (G_2 \circ A_1) \wedge \ldots \wedge (G_n \circ A_{n-1} \circ \ldots A_1) \wedge (G_n \circ A_n \circ A_{n-1} \circ \ldots A_1)$, et d'affectations $A_a \circ A_n \circ A_{n-1} \circ \ldots A_1$, où $\circ$ est le symbole de la composition de fonctions (*voir* figure 1.7(b), page xxv).

Déterminisation. La déterminisation consiste à *retarder* l'effet d'un choix non-déterministe sur les actions observables qui le suivent. Par exemple, cela revient à transformer en trois transitions deux transitions symboliques (avec les gardes non-exclusives G_1 et G_2 et les affectations A_1 et A_2 *voir* figure 1.8(a), page xxvi). Une transition pour le cas où $(G_1 \wedge \neg G_2)$ est vrai, une autre pour le cas où $(\neg G_1 \wedge G_2)$ est vrai, et la dernière pour le cas où $(G_1 \wedge G_2)$ est vrai. Dans ce dernier cas, le choix d'affecter les variables selon A_1 ou A_2 est retardé jusqu'à l'action observable qui suit. Ainsi, si b est la prochaine action, alors l'affectation A_1 doit avoir été exécutée, donc l'affectation est composée de la garde et de l'affectation correspondant à b, ce qui produit la garde $(G_b \circ A_1)$ et les affectations $A_b \circ A_1$. De la même manière, si c est la prochaine action, alors A_2 doit être exécuté, ce qui produit la garde $(G_c \circ A_2)$ et les affectations $A_c \circ A_2$. Le résultat de l'IOSTS obtenu par la procédure expliquée plus haut est montré à la figure 1.8(b) (*voir* page xxvi).

Il est important de remarquer que cette procédure peut ne pas terminer (par exemple si $b = a$ et $l_b = l$, le retard continue à l'infini). Cependant, elle termine pour un IOSTS appartenant à la sous-classe non-triviale des IOSTS *déterministes à l'horizon* k, qui est définie à la section A.2.4 du chapitre 7.



(a) Fragment d'un IOSTS $\mathsf{closure}(SP)$, où les transitions t_1 et t_2 qui sont impliquées dans un choix non-déterministe, sont montrées en pointillé.



(b) Fragment de l'IOSTS $\mathsf{det}(\mathsf{closure}(SP))$ obtenu à partir de $\mathsf{closure}(SP)$ par la procédure de déterminisation (les localités et transitions symboliques nouvelles ou modifiées sont montrées en pointillé).

Figure 1.8: Déterminisation d'un IOSTS $\mathsf{closure}(SP)$.

Exemple. La figure 1.9 page xxvii illustre l'IOSTS $SP_{vis} = \mathsf{det}(\mathsf{closure}(SP))$ obtenu après extraction des comportements observables de l'IOSTS SP montré à la figure 1.6 page xxiii (c'est-à-dire après application des procédures de clôture et de déterminisation expliquées dans les deux derniers paragraphes). Les transitions en bleu sur cette figure indiquent deux nouvelles transitions symboliques qui remplacent les deux séquences d'actions internes de l'IOSTS SP.

1.3.4.4 Sélection d'un graphe de test

L'étape de sélection de notre méthode symbolique de génération de tests confère son originalité à la méthode. Elle consiste à transformer l'IOSTS $SP_{vis} = \mathsf{det}(\mathsf{closure}(Spec \times TP))$ obtenu à la suite de la phase précédente en un graphe de test TG qui :

(1) (a) peut conduire à satisfaire l'objectif de test TP à partir duquel SP_{vis} a été généré, et

(b) contient "moins" d'états inaccessibles que l' IOSTS SP_{vis}.

Afin de résoudre ces deux problèmes, l'algorithme va utiliser l'information donnée sur les états accessibles et co-accessibles de SP_{vis} qui sont définis comme suit :

- $Reach(S^0_{vis}) \triangleq \{s \in S_{vis} \mid \exists\, \sigma \in Traces(SP_{vis}),\ s^0 \in S^0_{vis}\ .\ [s^0 \stackrel{\sigma}{\Rightarrow} s]\}$ est l'ensemble de tous les états de SP_{vis} qui sont accessibles à partir d'un état initial $s^0 \in S^0_{vis}$.
- $CoReach(S^{acc}_{vis}) \triangleq \{s \in S_{vis} \mid \exists\, \sigma \in (\Lambda^?_{vis} \cup \Lambda^!_{vis})^*,\ s^{acc} \in S^{acc}_{vis}\ .\ [s \stackrel{\sigma}{\Rightarrow} s^{acc}]\}$ est l'ensemble des états de SP_{vis} qui *peuvent mener* à l'ensemble des états d'acceptation S^{acc}_{vis} de SP_{vis}.

(2) est capable de communiquer avec une implémentation sous test $iut \in Iuts$. Ceci signifie que les actions d'entrée de iut doivent être considérées comme des actions de sortie de TG, et vice versa. Notons que les actions de l'IOSTS SP_{vis} ont les mêmes directions (entrée/sortie) que les actions de iut (*voir* la construction de SP_{vis}). Ainsi, pour obtenir le graphe de test qui peut communiquer avec iut, les directions des actions de SP_{vis} doivent être inversées.

Il faudrait donc extraire de SP_{vis} la partie utile, c'est-à-dire un IOSTS dont l'ensemble des traces soit exactement l'ensemble des préfixes stricts des traces acceptées de SP_{vis}. Ceci repose sur le calcul de l'intersection de l'ensemble des états accessibles à partir des états initiaux de SP_{vis} et de l'ensemble des états co-accessibles à partir des états d'acceptation de SP_{vis}, c'est-à-dire $Reach(S^0_{vis}) \cap CoReach(S^{acc}_{vis})$. Or un calcul *exact* de cet ensemble d'états est en

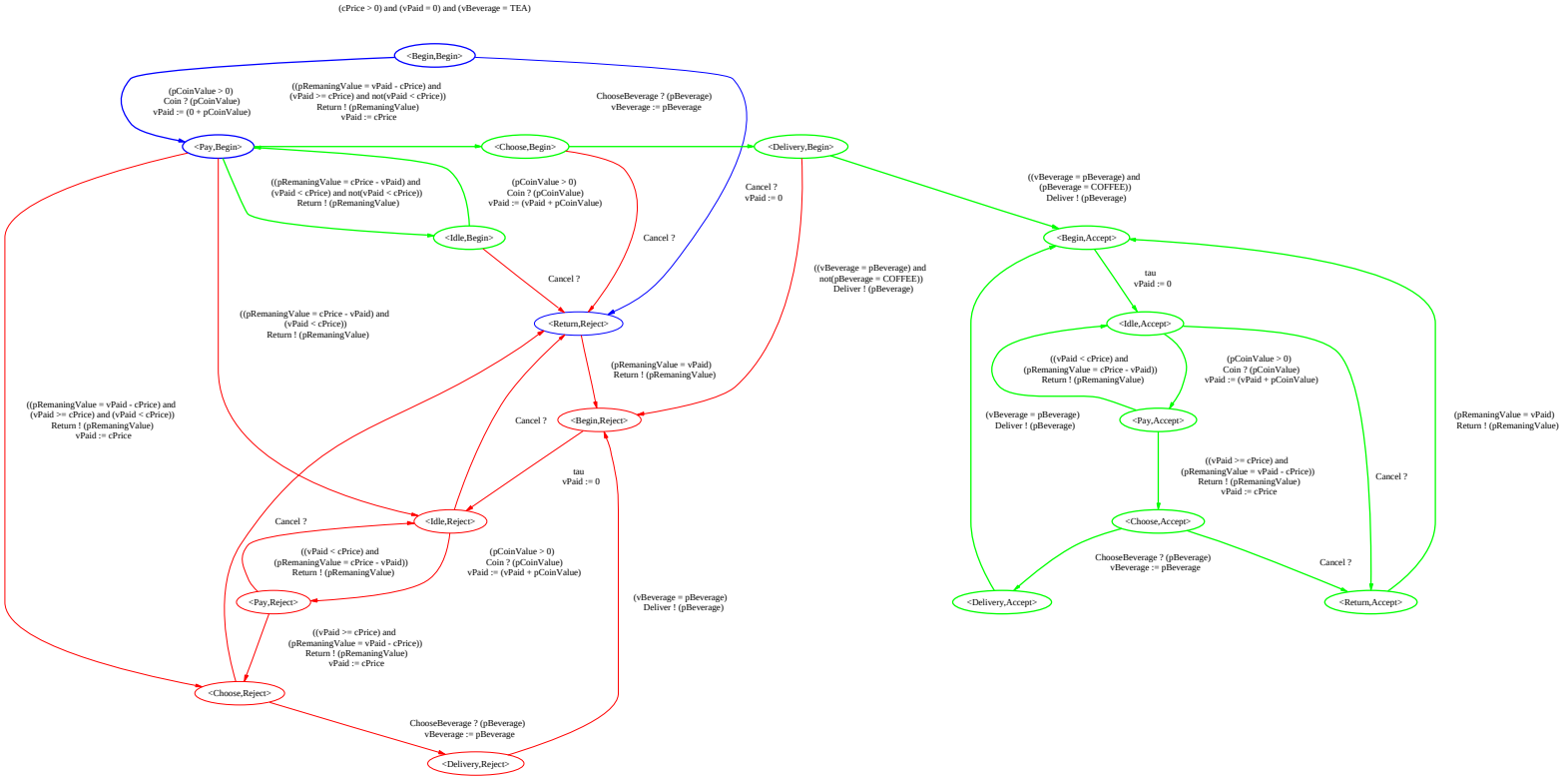


Figure 1.9: Un exemple d'IOSTS $SP_{vis} = \mathsf{det}(\mathsf{closure}(SP))$.

général impossible, les problèmes d'accessibilité et de co-accessibilité étant indécidables pour les IOSTS. Donc, nous sommes obligés de calculer seulement des *sur-approximations* de ces ensembles, que nous utilisons dans l'algorithme de sélection décrit plus bas.

Algorithme 1.1 (Sélection de graphe de test) Soient

- $SP_{vis} = \langle (V_{vis} \cup C_{vis} \cup P_{vis}), \Theta_{vis}, l^0_{vis}, L_{vis}, (\Sigma^?_{vis} \cup \Sigma^!_{vis}), T_{vis} \rangle$ un IOSTS obtenu à partir d'une spécification *Spec* et d'un objectif de test *TP* de *Spec* en appliquant les opérations de produit et de clôture, et en exécutant la procédure de déterminisation,

- $\mathsf{Reach}(SP_{vis})$ et $\mathsf{CoReach}(SP_{vis})$ deux prédicats caractérisant les sur-approximations des ensembles d'états accessibles et co-accessibles de l'IOSTS SP_{vis}, et

- pour chaque localité $l \in L_{vis}$ de SP_{vis}, $\mathsf{reach}(l)$ et $\mathsf{coreach}(l)$ deux prédicats qui caractérisent les sur-approximations des états accessibles et co-accessibles de la forme $\langle l, \vartheta \rangle$, où $\vartheta \in \mathsf{DOM}(V_{vis} \cup C_{vis})$.

  Dans la suite, nous utilisons également la notation $\mathsf{coreach}(l)[V/A]$, où A est un ensemble d'affectations pour les variables V et l est une localité. Cette notation identifie le prédicat caractérisant la sur-approximation des états co-accessibles de la forme $\langle l, \vartheta \rangle$, où chaque variable $v \in V$ est substituée au côté droit des affectations $A_v \in A$ correspondantes.

Alors le graphe de test TG est l'IOSTS

$$\langle\ (V_{TG} \cup C_{TG} \cup P_{TG}),\ \Theta_{TG},\ l^0_{TG},\ L_{TG},\ (\Sigma^?_{TG} \cup \Sigma^!_{TG}),\ T_{TG}\ \rangle$$

obtenu à partir de SP_{vis} comme suit :

(1) $V_{TG} = V_{vis}$, $C_{TG} = C_{vis}$ et $P_{TG} = P_{vis}$.

(2) $\Theta_{TG} = \Theta_{vis} \wedge \mathsf{CoReach}(SP_{vis})$.

(3) If $l^0_{vis} = Accept$, alors $l^0_{TG} = Pass$. Sinon, $l^0_{TG} = l^0_{vis}$.

(4)[9] $\Sigma^?_{TG} = \Sigma^!_{vis}$ et $\Sigma^!_{TG} = \Sigma^?_{vis}$.

[9] Afin d'obtenir un graphe de test TG apte à communiquer avec une implémentation sous test, nous devons inverser les alphabets des actions d'entrée/sortie de l'IOSTS SP_{vis}.

(5) L'ensemble des transitions symboliques de TG est : $T_{TG} \triangleq T_{TG}^{\mathsf{L2A}} \cup T_{TG}^{\mathsf{Inconc}}$, où :

(a) $T_{TG}^{\mathsf{L2A}} \triangleq \{ \langle l, a, \pi, G_{TG}, A, l_{TG} \rangle \mid$
$(\exists \langle l, a, \pi, G, A, l' \rangle \in T_{vis} \,.\, [l \neq \mathit{Accept} \ \wedge \ a \in (\Sigma_{vis}^{?} \cup \Sigma_{vis}^{!})]) \ \wedge$
$(G_{TG} = (G \wedge \mathsf{reach}(l) \wedge \mathsf{coreach}(l')[V_{TG}/A])$ est satisfiable$) \ \wedge$
$(l' = \mathit{Accept} \implies l_{TG} = \mathit{Pass}) \ \wedge \ (l' \neq \mathit{Accept} \implies l_{TG} = l') \}$
est l'ensemble des transitions menant soit à une localité $Pass$ soit à une localité à partir de laquelle il est possible d'atteindre une localité $Pass$, et

(b) $T_{TG}^{\mathsf{Inconc}} \triangleq \{ \langle l, a, \pi, G_{TG}, A, \mathit{Inconclusive} \rangle \mid$
$(\exists \langle l, a, \pi, G, A, l' \rangle \in T_{vis} \,.\, [l \neq \mathit{Accept} \ \wedge \ a \in \Sigma_{vis}^{!}]) \ \wedge$
$(G_{TG} = (G \wedge \mathsf{reach}(l) \wedge \neg\mathsf{coreach}(l')[V_{TG}/A])$ est satisfiable$) \}$
est l'ensemble des transitions menant à une localité $Inconclusive$ et étiquetée par une action *d'entrée* de TG.

(6) L'ensemble des localités L_{TG} se compose des origines et des destinations des transitions symboliques calculées plus haut T_{TG}, c'est-à-dire

$$L_{TG} \triangleq \bigcup_{\langle l,a,\pi,G,A,l' \rangle \in T_{TG}} \{l, l'\}$$

□

L'algorithme de sélection est illustrée par l'exemple suivant.

Exemple 1.4 Considérons l'IOSTS SP_{vis} décrit à la figure 1.9 (*voir* page xxvii). Alors, nous expliquons comment utiliser l'algorithme de sélection afin de produire des graphes de test différents.

Si nous calculons les sur-approximations de l'ensemble des états accessibles et co-accessibles basés uniquement sur la structure de contrôle de SP_{vis} (c'est-à-dire que pour chaque localité l de SP_{vis}, $\mathsf{reach}(l)$ et $\mathsf{coreach}(l)$ sont définis comme le prédicat $pc = l$, où pc est une variable de contrôle), alors nous obtenons le graphe de test TG_1 montré à la figure 1.10(a) (*voir* page xxxi). Dans ce cas, l'algorithme de sélection est très similaire à celui utilisé par la méthode de génération de tests pour IOLTS [Jéron, 2004]. Notons que cette approximation est imprécise car elle ne prend pas en compte les données de SP_{vis}.

Le second graphe de test TG_2 (*voir* figure 1.10(b), page xxxi) a été construit en utilisant les sur-approximations des ensembles des états accessibles et co-accessibles obtenus par l'analyse symbolique de SP_{vis} abstraite par polyèdres [Cousot and Halbwachs, 1978], [Jeannet, 2000b] (dans ce cas $\mathsf{reach}(l)$

(a) TG_1 (b) TG_2

Figure 1.10: Graphes de test obtenus à partir de l'IOSTS $SP_{\boldsymbol{vis}}$ montré à la figure 1.9 (*voir* page xxvii) par l'algorithme de sélection.

et $\mathsf{coreach}(l)$ sont plus précis que ceux calculés dans le cas précédent). Comme on peut le voir aisément sur la figure 1.10 (*voir* page xxxi), la sélection basée sur cette abstraction est plus précise que celle décrite plus haut. En effet, elle supprime avec succès deux localités *Inconclusive* qui apparaissaient dans TG_1. La première localité *Inconclusive* est éliminée en renforçant la garde de la première transition symbolique de TG_1 de telle façon qu'il est possible d'atteindre la localité *Pass*. Pour cela, nous avons utilisé l'information que la première somme payée doit être suffisante pour recevoir la boisson. La seconde localité *Inconclusive* est éliminée en utilisant le même principe que plus haut, c'est-à-dire en renforçant la garde de la transition symbolique où l'utilisateur choisit la boisson. En effet, nous obligeons l'utilisateur à choisir du café, ce qui correspond évidement à ce qu'il souhaite, voir l'objectif de test de la figure 1.2 page xii. □

Nous allons maintenant étudier la relation de trace entre l'IOSTS SP_{vis} et le graphe de test TG obtenu à partir de SP_{vis} par l'algorithme de sélection (*voir* page xxix). Remarquons tout d'abord que l'algorithme n'ajoute aucune transition symbolique à SP_{vis} mais restreint les gardes de certaines transitions et renomme certaines localités de SP_{vis} dans "pass" ou "inconclusive". Par conséquent, l'ensemble de traces de TG ne peut être plus grand que l'ensemble de traces de SP_{vis}. Cependant, nous ne pouvons assurément pas obtenir l'égalité entre $Traces(TG)$ et $Traces(SP_{vis})$. En effet, le but de l'algorithme de sélection est de choisir (1) les traces de SP_{vis} menant aux états à partir desquels il est possible d'aller vers les états d'acceptation, et (2) les traces de SP_{vis} se terminant par une action *d'entrée* valuée et menant aux états à partir desquels il est impossible d'aller vers un état d'acceptation, et dont les préfixes stricts sont des traces conduisant aux états à partir desquels il est possible d'aller vers des états d'acceptation. Dans le dernier cas, le graphe de test TG doit produire le verdict *Inconclusive*. (Étant donné que le problème de savoir si à partir d'un état on peut atteindre un état accepteur est indécidable, nous calculons une *sur-approximation* de l'ensemble des états possédants cette propriété.) Ainsi, il est clair que l'ensemble de traces de TG est inclus dans l'ensemble de traces de SP_{vis}. Dans la proposition ci-dessous, nous caractérisons l'ensemble des traces de SP_{vis} ainsi que ses sous-ensembles de traces menant aux états "pass" ou "inconclusive".

Proposition 1.3 (Traces de TG) Soit TG un graphe de test obtenu à partir de l'IOSTS $SP_{vis} = \mathsf{det}(\mathsf{closure}(Spec \times TP))$, où $Spec$ est une spécification avec l'alphabet des actions d'entrée valuées $A^!_{Spec}$, et TP est un objectif de test de $Spec$. Alors :

(1) $Traces(TG) \subseteq Traces(SP_{vis})$,

(2) $Traces_{pass}(TG) \subseteq ATraces(Spec \times TP)$, plus précisément,

(2.a) $Traces_{pass}(TG) = SPref(ATraces(Spec \times TP))$, et

(3) $Traces_{inconc}(TC) \subseteq (Traces(Spec) \cdot \Lambda^{!}_{Spec} \cap Traces(Spec)) \setminus Pref(ATraces(Spec \times TP))$.

Notons que si la calcul exact était possible, alors la dernière inclusion se transforme en une égalité.

où $Traces_{pass}(TG)$ et $Traces_{inconc}(TG)$ sont les ensembles de traces du graphe de test TG conduisant respectivement aux états "pass" et "inconclusive" de TG. □

Les propriétés (1), (2) et (3) de traces d'un graphe de test TG sont formellement prouvées dans la section 7.4 du chapitre 7. Il est aussi important de souligner que nous avons obtenu un cas de test TG qui est *déterministe* (*voir* lemme 7.1, page 195 au chapitre 7).

1.3.4.5 Rendre un graphe de test complet en entrée

La dernière étape de notre méthode symbolique de génération de tests consiste à transformer le graphe de test TG, obtenu à partir d'une spécification *Spec* et d'un objectif de test TP de *Spec* par les opérations et algorithmes décrits dans les sections précédentes, en un cas de test TC. À cause de la définition d'un cas de test donnée page xv, le cas de test TC doit toujours réagir sur n'importe quelle entrée venant d'une implémentation sous test *iut*. Ceci signifie que TC ne doit bloquer aucune entrée de *iut*, mais doit répondre négativement aux entrées incorrectes. Par conséquent, nous devons exiger que chaque localité du graphe de test TG, à l'exception des localités *Pass* et *Inconclusive*, (1) accepte n'importe quelle entrée de *iut*, et (2) redirige les entrées incorrectes vers la nouvelle localité *Fail*, c'est-à-dire rende TG *complet en entrée*. Pour accomplir ceci, nous proposons une procédure syntaxique simple.

Soit TG un graphe de test avec un ensemble des localités L, un ensemble des variables V, un ensemble des transitions symboliques T et un alphabet d'actions d'entrée/sortie $\Sigma = \Sigma^{?} \cup \Sigma^{!}$. Le cas de test TC est l'IOSTS obtenu à partir de TG en ajoutant une nouvelle localité $Fail \notin L$ et, pour chaque localité $l \in L$ et chaque action d'entrée $a \in \Sigma^{?}$, en ajoutant une transition symbolique $t \notin T$ d'origine l, de destination *Fail*, d'action a, d'affectations $A_t = \bigcup_{v \in V}(v := v)$ et de garde $G_t = \bigwedge_{t' = \langle l, a, \pi, G_{t'}, A_{t'}, l' \rangle}(\neg G_{t'})$. Autrement dit, toute entrée non exécutable dans TG devient exécutable dans TC et mène aux nouvelles localités (de blocage complet) *Fail*. L'algorithme qui transforme TG en un cas de test complet en entrée TC est donné dans la section 7.5 du chapitre 7.

Un cas de test TC obtenu par la procédure décrite plus haut contient trois types de verdicts : *Pass*, *Inconclusive* et *Fail*. Par conséquent, les ensembles de traces de TC qui contiennent respectivement les traces menant aux états "pass", "inconcusive" et "fail" sont notés respectivement $Traces_{pass}(TC)$, $Traces_{inconc}(TC)$ et $Traces_{fail}(TC)$. Dans la proposition ci-dessous, nous caractérisons ces ensembles de traces.

Proposition 1.4 (Traces de TC) Soit TC un cas de test obtenu à partir d'un graphe de test $TG = \mathsf{select}(\mathsf{det}(\mathsf{closure}(Spec \times TP)))$, où $Spec$ est une spécification avec un alphabet d'actions d'entrée valuées $\Lambda^{?}_{Spec}$, TP est un objectif de test de $Spec$ et select est l'algorithme 1.1 (*voir* page xxix). Alors :

(1) $Traces(TC) = Traces(TG) \cup Traces_{fail}(TC)$,

(2) $Traces_{pass}(TC) = Traces_{pass}(TG)$,

(3) $Traces_{inconc}(TC) = Traces_{inconc}(TG)$, et

(4) $Traces_{fail}(TC) \subseteq (Traces(Spec) \cdot \Lambda^{!}_{Spec} \setminus Traces(Spec))$, plus précisément

(4.a) $Traces_{fail}(TC) = (Traces(TG) \cdot \Lambda^{!}_{Spec} \setminus Traces(TG))$.

□

Les propriétés (1), (2), (3) et (4) de traces d'un cas de test TC sont formellement démontrées à la section 7.5 du chapitre 7, et la preuve de la propriété (4.a) est une conséquence directe de la procédure qui rend un graphe de test complet en entrée. Il est aussi important de souligner que le cas de test TC que nous avons obtenu est *déterministe* (pour les détails, le lecteur peut se référer au deuxième point du théorème 7.2, page 203 du chapitre 7).

Exemple 1.5 Considérons le graphe de test TG_2 montré à la figure 1.10(b) (*voir* page xxxi) obtenu par l'algorithme de sélection (*voir* page xxix). La figure 1.11 (*voir* page xxxv) illustre le cas de test complet en entrée TC obtenu, à partir du graphe de test TG donné, par la procédure expliquée dans cette section. □

1.3.5 Correction d'un cas de test

Dans la section 1.3.4, nous avons décrit comment générer un cas de test symbolique TC à partir d'une spécification $Spec$ et d'un objectif de test TP de $Spec$. Nous avons également montré que le cas de test résultant TC est *initialisé* (par la construction du cas de test) et *déterministe* (*voir* théorème 7.2, page 203). Cependant, nous n'avons pas montré que TC produit des résultats corrects lorsqu'il est

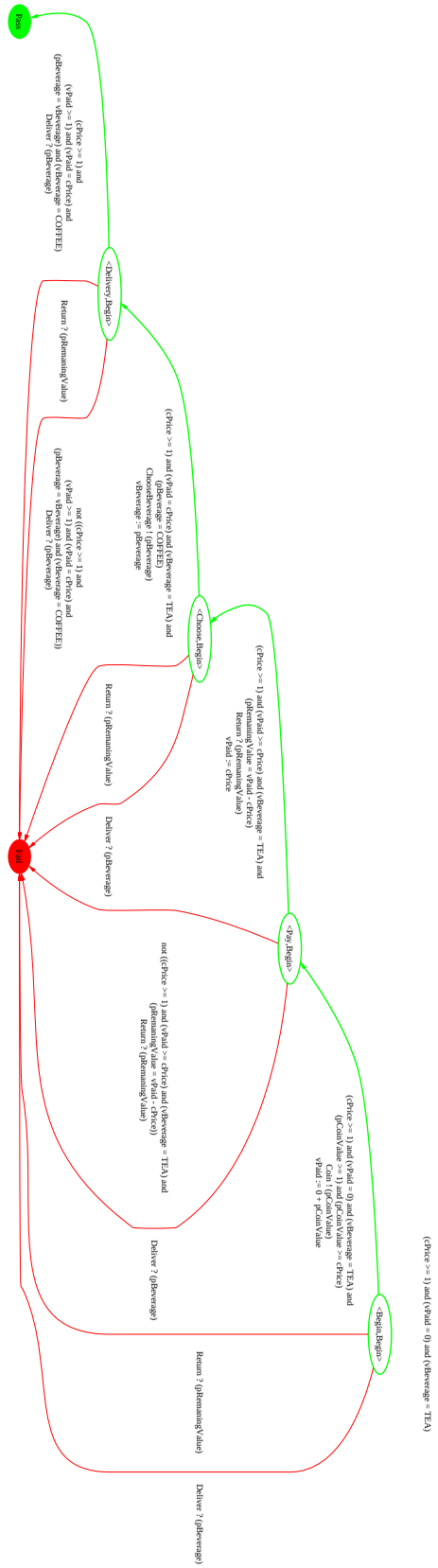


Figure 1.11: Cas de test complet en entrée *TC*.

exécuté sur une implémentation sous test. Ainsi, le but de cette thèse est de démontrer la correction du cas de test donné par rapport à la spécification *Spec*, l'objectif de test *TP* de *Spec*, et l'ensemble des implémentations *Iuts*, au sens de la définition 1.7 (*voir* page xix). Formellement :

Théorème 1.1 (Correction de *TC*) Le cas de test *TC* généré à partir de la spécification *Spec* et de l'objectif de test *TP* par notre méthode symbolique de génération de tests est *correcte* pour l'ensemble des implémentations sous test *Iuts*. □

Le lecteur trouvera la preuve formelle de la correction du cas de test à la section 7.6 du chapitre 7.

1.3.6 Conclusion

Nous avons tout d'abord proposé un modèle pour la représentation des systèmes réactifs appelé système symbolique des transitions à entrée/sortie (IOSTS). Ce modèle nous permet de décrire plus précisément les systèmes réactifs en prenant explicitement en compte leurs données. De plus, il nous autorise à générer des cas de test sous la forme de programmes avec des variables, des constantes symboliques et des paramètres (c'est-à-dire des cas de test génériques).

Ensuite, nous avons brièvement décrit les principes essentiels de la génération symbolique de tests basée sur les IOSTS. Cette méthode traite symboliquement les données d'un système en combinant l'approche de génération de tests proposée auparavant par notre groupe de recherche [Fernandez et al., 1996] avec l'interprétation abstraite [Cousot and Cousot, 1976], [Cousot and Cousot, 1977] que l'on utilise pour la sélection des cas de test. De plus, ce modèle nous permet d'éviter le problème de l'explosion de l'espace d'états et de dériver des cas de test génériques. Enfin, nous avons montré que notre méthode symbolique de génération de tests dérive des cas de test corrects.

L'ensemble de ces travaux constituent un extension de l'approche proposée initialement dans [Rusu et al., 2000].

1.4 Implémentation et résultats expérimentaux

La troisième partie de ce document est consacrée à l'implémentation de la méthode symbolique de génération de tests proposée dans la seconde partie de la thèse, ainsi qu'à l'étude de cas pour laquelle cette implémentation a été utilisée. Plus précisément, au chapitre 8, nous employons la théorie et les algorithmes présentés dans la deuxième partie de la thèse afin d'implémenter un prototype pour la génération automatique et symbolique de cas de test appelé STG[10] [Clarke

[10]Provient de la terminologie anglaise "Symbolic Test Generator".

et al., 2002], [Clarke et al., 2001b], [Clarke et al., 2001c]. Nous décrivons ensuite l'utilisation de STG dans le test d'un protocole de communication qui transfère un fichier de données présenté sous la forme d'une séquence de paquets. Le protocole est fondé sur le protocole classique du bit alterné, mais ne permet qu'un nombre fini de retransmissions de chaque paquet. Enfin, nous comparons STG à d'autres outils existants utilisés pour la génération de tests de conformité des systèmes réactifs.

1.5 Conclusion

Ce mémoire s'achève par un résumé des principaux résultats obtenus, et par une discussion sur les perspectives. Une brève description du dernier chapitre de ce document est donnée ci-après.

Résumé. Cette thèse se situe dans le domaine du test de conformité boîte noire pour les systèmes réactifs. De tels systèmes sont généralement complexes et de taille importante. Il n'est donc pas aisé de les implémenter sans erreur. Même une petite erreur peut mener à de sérieux dysfonctionnements du système (*voir* les exemples donnés page 2). Il est par conséquent essentiel de développer des techniques de détection d'erreur dans les systèmes réactifs qui les rendent plus fiables. Dans cette thèse, nous nous concentrons sur l'une de ces techniques appelée *le test*.

Au début de ce document, nous mentionnions qu'au cours des dernières décennies, les théories de test et les techniques pour la dérivation automatique de tests ont été développées. Certaines de ces techniques sont basées sur le modèle des machines d'états finis (FSM[11]), et d'autres sur le modèle des systèmes de transitions à entrée/sortie (IOLTS). Puisque, à notre avis, le modèle IOLTS est mieux adapté pour le test des systèmes réactifs que le modèle FSM, nous nous sommes focalisé sur l'étude des méthodes et algorithmes pour la dérivation automatique de tests fondés sur les IOLTS. Nous avons tout particulièrement pris en considération deux algorithmes à la volée efficaces proposés dans [Tretmans, 1992], [Jéron and Morel, 1999]. Nous avons également décrit des outils académiques (par exemple TorX [Belinfante et al., 1999], TGV [Fernandez et al., 1996]) et industriels (par exemple Autolink [Telelogic, 1998], TestComposer [Kerbrat and Ober, 1999]) déjà existants, qui implémentent ces algorithmes et produisent des cas de test corrects, ce qui signifie essentiellement qu'ils émettent toujours un verdict correct.

Néanmoins, les théories et les outils fondés sur des IOLTS sont relativement limités. Ils ne prennent pas *explicitement* en compte les données du système car

[11] Provient de la terminologie anglaise "Finite State Machine".

le modèle sous-jacent des IOLTS ne permet pas de le faire. Ainsi, pour modéliser une spécification du système réactif avec des IOLTS, il est nécessaire d'énumérer les valeurs de chaque donnée employée par le système. Ceci peut conduire au problème classique de l'explosion de l'espace d'états. De plus, cette énumération a aussi pour effet d'obtenir des cas de test où toutes les données sont instanciées. Ceci contredit la pratique industrielle où les cas de test (écrits, par exemple dans la langage TTCN [ISO/IEC/JTC1/SC21, 1992]) sont de vrais programmes avec des données (variables, constantes symboliques et paramètres de communication).

Afin de résoudre les problèmes mentionnés ci-devant, nous avons proposé une extension de l'approche [Rusu et al., 2000] pour la génération automatique de cas de test symboliques sous la forme de systèmes étendus de transitions avec des variables, des constantes symboliques et des paramètres de communication. Ces systèmes sont appelés systèmes symboliques de transitions à entrées/sorties (IOSTS).

Puis, nous avons présenté les concepts du test de conformité basé sur les IOSTS. En d'autres termes, nous avons défini formellement (1) la relation de conformité ioc entre une spécification et une implémentation sous test qui sont toutes deux modélisées avec des IOSTS, (2) la notion de cas de test et (3) les propriétés des cas de test, permettant d'établir un lien entre les cas de test et la relation de conformité.

Nous avons décrit ensuite l'approche de génération symbolique de tests fondée sur la théorie de test introduite plus haut et implémentée dans l'outil STG. La description de cet outil et une étude de cas du protocole de retransmission bornée sont données dans la troisième partie de notre travail. Enfin, nous avons comparé notre approche avec d'autres approches symboliques pour la génération automatique de tests. La majorité de celles-ci sont basées sur la propagation de contraintes et utilisent des techniques de résolution de contraintes. De surcroît, elles ne traitent pas du problème du non-déterminisme (rappelons que nous avons posé comme hypothèse que le non-déterminisme est formellement interdit dans le test). Les éléments originaux de notre approche sont : l'emploi d'une technique d'interprétation abstraite au moment de la sélection du cas de test et les solutions partielles au non-déterminisme. Notons que nous n'avons pas résolu tous les problèmes en rapport avec la génération symbolique de tests. En particulier, notre approche utilise aussi une technique de résolution de contrainte lorsque l'on instancie des cas de test symboliques pendant leur exécution.

Nous pensons que notre approche symbolique de génération de tests mérite d'être étudiée et que la recherche future peut améliorer son applicabilité aux systèmes réactifs industriels.

Recherche future. Les idées suivantes pour la recherche à venir sont inspirées du travail présenté dans cette thèse.

De ioc à ioco. Avant d'entrer en détails dans les perspectives décrites dans ce paragraphe, nous rappelons la différence entre les relations ioco [Tretmans, 1996a] et ioc (*voir* définition 1.4, page xv). Autrement dit, d'une part, la relation ioco crée un lien entre une spécification *suspendue* $\Delta(Spec)$ et le modèle *suspendu* d'une implémentation sous test $\Delta(\mathcal{I}_{iut})$. Elle vérifie si, après l'exécution de chaque trace de *suspension* (c'est-à-dire trace pouvant contenir l'action de sortie spéciale δ) de la spécification, l'implémentation produit ou pas uniquement des actions de sortie spécifiées. D'autre part, la relation ioc crée un lien entre une spécification *Spec* avec une implémentation sous test $\mathcal{I}_{iut}$, et effectue la même vérification que ioco mais pour chacune de ses traces *propres* (c'est-à-dire trace ne contenant aucune action δ) de la spécification.

Dans cette thèse, nous proposons une approche pour la génération symbolique de tests qui est basée sur le modèle IOSTS et sur la relation ioc. Cependant, cette approche a quelques faiblesses. L'une d'entre elles est le problème des blocages, qui n'ont pas été traités dans la partie de la thèse abordant la génération symbolique de tests. En revanche, nous avons décrit ceci dans l'introduction (*voir* page 42). Afin d'améliorer notre génération symbolique de tests dans ce domaine, il serait possible d'essayer l'approche suivante (publiée dans [Rusu et al., 2004]).

(1) Limiter le modèle des IOSTS à un modèle qui ne contient pas de blocages vivants syntaxiques (cycles d'actions internes), et qui utilise la relation ioco [Tretmans, 1996a] comme critère de correction au lieu de la relation ioc.

(2) Pour une spécification donnée, construire son IOSTS suspendu en codant tous les blocages potentiels (de sortie et complets) d'une spécification donnée avec l'action de sortie spéciale δ.

Remarquons que le problème de la détection des blocages est indécidable pour les IOSTS en général. Cependant, dans le cas de l'absence de blocages vivants syntaxiques sur un IOSTS $\mathcal{M}$, il est possible de construire syntaxiquement l'IOSTS suspendu de $\mathcal{M}$.

La solution classique pour détecter des blocages non-spécifiés dans une implémentation sous test durant une expérience de test est d'équiper chaque testeur avec une minuterie indiquant le temps que le testeur doit attendre avant qu'une sortie n'apparaisse. Si la sortie n'a pas lieu avant un certain temps, alors le testeur peut conclure que cette sortie n'aura plus lieu. Dans ce cas, il décide que l'implémentation est bloquée.

Nous pensons que les suggestions données dans ce paragraphe aideront à améliorer notre méthode symbolique de génération de tests.

Propriétés de sûreté. Le rapport [Rusu et al., 2004] propose une approche pour combiner la vérification et les techniques de test de conformité. Dans ce rapport, la spécification formelle *Spec* d'un système réactif donné est modélisé par IOSTS, et chaque propriété de sécurité de *Spec* est représentée de la même manière qu'un objectif de test employé dans la méthode symbolique de génération de tests. Alors,

Premièrement, chaque propriété est *vérifiée* sur *Spec* en utilisant des techniques automatiques (par exemple, l'interprétation abstraite) qui sont fiables mais pas nécessairement complètes pour la classe de propriétés de sécurité considérée ici.

Deuxièmement, pour chaque chaque propriété, un cas de test est *généré automatiquement* à partir de la spécification et de cette propriété et il est exécuté sur une implémentation boîte noire du système.

Si l'étape de vérification est réussie, c'est-à-dire si elle a établi que la spécification satisfait les propriétés, alors l'exécution peut détecter la violation de la propriété par l'implémentation et la relation de conformité standard ioco [Tretmans, 1996a] entre l'implémentation et la spécification. Si l'étape de vérification n'est pas réussie, c'est-à-dire si elle n'a pas permis de prouver ou de réfuter la propriété, alors l'exécution du test peut détecter en plus une violation de la propriété par la spécification.

Critères de couverture. L'outil STG (comme son prédécesseur TGV [Fernandez et al., 1996]) emploie les objectifs de test comme mécanisme de sélection de cas de test. Chaque objectif de test dans STG est donné sous la forme d'un graphe. Cette représentation offre à un développeur de logiciels une manière naturelle de décrire (partiellement) les comportements d'une spécification donnée devant être testée. Néanmoins, l'écriture manuelle d'objectifs de test exige une bonne connaissance de la spécification. Ceci signifie que le processus d'écriture des objectifs de test est encore difficile à accomplir pour les humains, surtout si une bonne couverture de la spécification doit être ciblée. Nous pensons que les critères classiques de couverture structurelle [Rapps and Weyuker, 1985] combinés avec l'analyse symbolique, peuvent apporter des objectifs de test intéressants.

Études de cas. Jusqu'à présent, nous avons seulement généré des cas de test à partir des spécifications académiques plus ou moins classiques d'une machine à café et d'un protocole de retransmission bornée. La prochaîne étape importante est d'appliquer notre approche afin de tester des systèmes industriels réalistes. Nous avons déjà fait les premiers pas dans cette

direction : nous avons généré des cas de test pour le porte-monnaie électronique (CEPS) [CEPSCO, 2000]. Cette étude de cas nous a donné les résultats prometteurs publiés dans [Clarke et al., 2001a]. Ainsi, il serait intéressant de poursuivre de telles expériences. Nous pensons également que de futures études de cas vont fournir des informations essentielles quant à l'applicabilité de notre approche symbolique de génération de tests et nous aidera à l'améliorer.

Chapter 1

Introduction

This chapter describes the context of the thesis and gives the motivations to the research presented in it. We start by introducing reactive systems, and emphasizing the importance of such systems by taking realistic situations from daily life. These situations motivate using validation techniques, such as testing and verification, in order to ensure correctness of reactive systems. Then, we focus on testing, describe the role of testing in the software development cycle, and discuss existing testing methods, with stress on conformance testing. Next, we compare testing with another validation technique (verification), and underline their complementarity by discussing advantages and drawbacks of these techniques. Finally, we present the main goals of the thesis and its outline.

1.1 Reactive Systems

The notion of reactive system was introduced by A. Pnueli and D. Harel in [Harel and Pnueli, 1985]. In this work, the authors make a distinction between *transformational* and *reactive systems* . The former accept inputs and, after a certain period of time, produce outputs, *i.e.* they operate in an autonomous way. For instance, a transformational system can be a program that calculates the square root of an input, or more a complex program such as a compiler. The latter interact permanently with its *environment* by continuously exchanging information. Many systems in the real world can be considered as reactive: vending machines, cash dispensers, or more complex software and hardware systems, such as silicon chips, communication protocols, smart cards, industrial plants, avionic systems and nuclear reactors. It is essential to realize how crucial reactive systems can be, and how important it is to perform testing or other validation techniques on them. For this purpose, we use a series of examples. All of them are available

from the "10 Great Bugs of History" collection [CNET, 2000] and from the collections of software bugs found at the "Software Quality Assurance and Testing Resource Center" [Hower, 2003].

NASA Mariner 1, Venus Probe (1962). "A probe launched from Cape Canaveral was set to go to Venus. After takeoff, the unmanned rocket carrying the probe went off course, and NASA had to blow up the rocket to avoid endangering lives on earth. NASA later attributed the error to a faulty line of Fortran code" (*see* [Myers, 1976]).

Therac-25 Radiation Machine (June 1985–January 1987). "The Therac-25 medical linear accelerator was responsible for six accidents involving massive overdoses of radiation, four of which lead to deaths. The proximate causes of the accidents were eventually attributed to two separate and hard-to-reproduce faults caused by race conditions in the data entry system." Leveson and Turner published a detailed investigation of the failures and the responses to them (*see* [Leveson and Turner, 1993]).

AT&T Long Distance Service (1990). "Switching errors in AT&T's call handling computers caused the company's long-distance network to go down for nine hours, the worst of several telephone outages in the history of the system. The meltdown affected thousands of services and was eventually traced to a single faulty line of code."

NASA Mars Climate Orbiter (October 1999). "In October of 1999 the $125 million NASA Mars Climate Orbiter spacecraft was believed to be lost in space due to a simple data conversion error. It was determined that spacecraft software used certain data in English units that should have been in metric units. Among other tasks, the orbiter was to serve as a communications relay for the Mars Polar Lander mission, which failed for unknown reasons in December 1999. Several investigating panels were convened to determine the process failures that allowed the error to go undetected." The full story and the report about that are available in [Isbell and Savage, 1999].

Britain's National Tax System (March 2002). "In March of 2002 it was reported that software bugs in Britain's national tax system resulted in more than 100,000 erroneous tax over-charges. The problem was partly attributed to the difficulty of testing the integration of multiple systems."

Of course, this thesis does not pretend to solve the problems presented in the examples above, it just shows some progress in formal test generation. The examples above are used to emphasize the importance of the research oriented to the improvement of existing validating techniques for reactive systems.

1.2 Testing

This section introduces testing and discusses its usability at the present time. It also briefly presents black-box conformance testing which is the main topic of all remaining chapters of this work.

1.2.1 General View

Software testing [Myers, 1979] is a process that consists of examining computer programs or systems with the intent of finding errors in them. The aim of testing is to make sure that systems will work correctly during their utilization. Testing can never be exhaustive for any realistic system because the time and effort that can be spent on it is always limited. Therefore, it cannot guarantee complete correctness of the system. That is to say, in E.W. Dijkstra's words:

> *Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.*

Testing is a time-consuming process, B. Boehm notices that:

> *"Checkout" and testing often consume 50% of software effort.*

Nevertheless, it should be done for each phase of the software development cycle. The main reason for this was mentioned by B. Boehm:

> *The cost of* [finding and] *fixing an error increases by an order of magnitude at each stage in development.*

The testing phases of this software development cycle are depicted in Figure 1.1 and explained below.

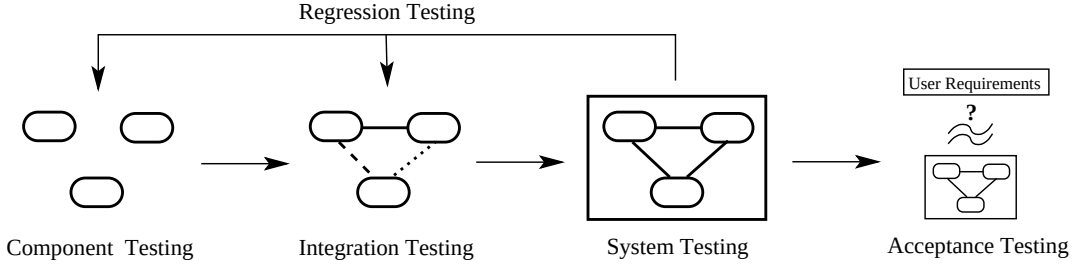


Figure 1.1: Phases of the testing process.

Component (Unit) Testing is the process of testing small components of a computer program, such as subprograms and subroutines. The purpose of this phase is to focus on the testing of "building blocks" of the program, instead of examining the entire program.

Integration Testing determines whether all the components, which will be linked in the given computer program, work correctly together, *i.e.* it tests "links" between components.

System Testing examines a whole program, *i.e.* all components of the program together. The goal of system testing is to reveal bugs that can be exposed while testing the entire integrated program and not only its components.

Regression Testing is the process of testing a program whose code has been modified. The modification of the code can occur while fixing errors or adding new functionalities, for instance. The aim of regression testing is to ensure that the modified program still satisfies its requirements and that its previous functionalities have not been affected.

Acceptance Testing is a process determining whether a program satisfies the requirements listed in the original development contract between customers and, for example, a software company.

Nowadays, as software systems become larger and much more complex, many software companies prefer to employ dedicated test engineers who focus only on writing and executing test cases, rather than requiring programmers to develop and test systems at the same time. The rest of this section discusses some of the existing methods for testing.

1.2.2 Testing Methods

Software testing (*e.g.* [Myers, 1979], [Beizer, 1990]) distinguishes structural and functional testing. As shown in Figure 1.2 , structural testing is based on the

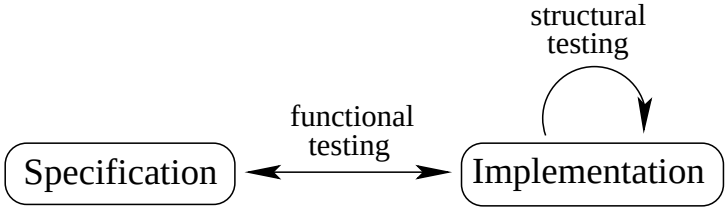


Figure 1.2: Structural and functional testing.

analysis of the internal structure of an *implementation* (a realistic, executable piece of software or hardware that should provide desired behaviors), while functional testing consists of checking whether an implementation of the software or hardware satisfies its *specification* (a description of the desired behaviors that define only what the system should do, not how it is done).

1.2.2.1 Structural Testing

Structural Testing, also referred to as *white-box testing* [Beizer, 1990], is a test suite design method that relies on the knowledge of the internal structure of the program, *i.e.* the test suite is derived from the program code. The aim of this method is to test the program code (1) by choosing test data for examining, for instance, each statement or every path in the program code; or (2) by checking the boundary conditions for selection and repetition of control structures of the program. Below we briefly describe some structural testing techniques.

Data-Flow Testing (*see* [Rapps and Weyuker, 1985]) is a technique for analyzing how variables of a program are bound to values and how these variables are used. More precisely, the purpose of data-flow testing is to construct a test suite that forces the execution of different interactions between the moment when variables are defined and when they are used in the program.

Mutation Testing (*see* [Budd et al., 1980], [Howden, 1982]) is a technique for testing faulty hypotheses. The goal of mutation testing is to construct a set of test cases that distinguish a given program from a mutated one, *i.e.* a program generated from the original one by applying mutational transformations (*e.g.* introducing errors to it).

Domain Testing (*see* [Beizer, 1990], [Jeng and Weyuker, 1994]) is a technique to check that the values taken by a variable, a condition, or an index are inside their specified or valid range. Domain testing also checks that the program accepts only valid input data, because it is unlikely to get reasonable results if incorrect input data have been entered.

1.2.2.2 Functional Testing

Functional Testing, also called *black-box testing* [Beizer, 1995], is a test suite design method that checks that the functionalities of a given program correspond to its specification without making any reference to its internal structures. The aim of this method is to derive test cases from a program specification, to execute them on the real program and to make sure that the latter behaves correctly by comparing the outputs produced by the program with these required in the specification. Bellow, we present examples of black-box testing.

Conformance Testing (*see* [Beizer, 1995], [Tretmans, 1992]) is a technique for checking whether functionalities mentioned in the specification of a given computer program have been implemented in this program. The aim of conformance testing is to generate a set of test cases that examines whether the program satisfies its specification. The question put here is: “Does the

computer program do what is should do?". Black-box conformance testing is discussed in further detail in Chapter 2 (*see* page 13).

Performance Testing (*see* [IEEE/ANSI, 1990]) is a technique for comparing the compliance of a given program with specified performance requirements. The purpose of performance testing is to measure the resources, such as execution time, response time, memory usage, *etc.* that are needed to carry out the program. The typical question in performance testing, is: "How fast does the program perform its task?".

Robustness Testing is a technique defined as: "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [IEEE/ANSI, 1990], *i.e.* the goal of robustness testing is to examine the behaviors of a given program in an erroneously behaving environment. The main question is here: "How does the program react if its environment does not behave as expected?".

Reliability Testing (*see* [Musa, 1975]) is a technique for testing whether a given program performs its tasks correctly during a specified period of time and in a specified environment. The question asked for reliability testing is: "How long can we rely on the correct functioning of the computer program?".

1.3 Testing and Verification

This section discusses two validation techniques that are used to increase the confidence in the correct functioning of systems as prescribed by their specification. These techniques are *testing* and *verification*. The aim of verification (*see* Figure 1.3) is to *prove the properties* of a computer system by *formal manipulations* on its mathematical *model*. Verification *gives a certainty* about satisfaction of these properties in the model. In comparison, testing (*see* Figure 1.3), as defined in Section 1.2, is a process of examining a *real*, *executable* implementation of the system with the *intention* of finding errors, and it can *never guarantee* that an

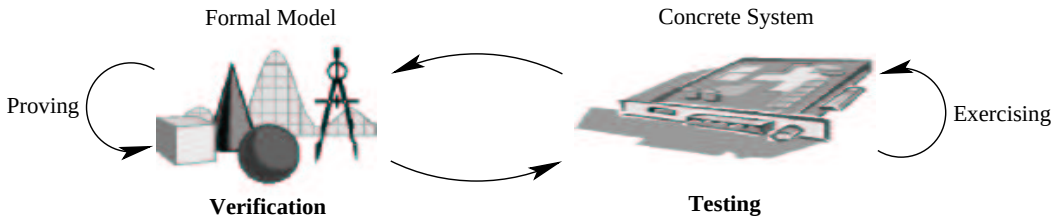


Figure 1.3: Testing and verification.

implementation after testing is error-free. Moreover, testing and verification are complementary techniques. In this thesis, we demonstrate that some verification techniques are applicable to testing.

1.4 About this Thesis

In the previous sections of the introductory part of the thesis we briefly described the testing area and its problems. The rest of this thesis is devoted to the black-box conformance testing for reactive systems. In this section we present the main contributions of our research and describe the structure of the thesis.

1.4.1 Motivation and Objectives

During the past years we have observed a continuous growth of software and hardware systems in size and complexity, which caused many erroneous systems to appear. As the majority of reactive systems are often crucial, the importance of error detection in these systems has increased. This has been one of the reasons for the current interest in testing reactive systems. During the last decades, testing theories and tools used for automatic test generation have been developed. In these theories and algorithms, the specifications of reactive systems are often modeled by variants of the labeled transition systems (LTS). However, these theories and tools do not explicitly take into account the systems data, since the underlying model of LTS does not allow to do that. This limitation of the model compels us to enumerate the values of the data before building the LTS model of a system. This may result in the classical state-space explosion problem. Moreover, this enumeration has also the effect of obtaining test cases, where all data are instantiated. This contradicts with industrial practice, where test cases (expressed, for instance, in TTCN [ISO/IEC/JTC1/SC21, 1992]) are real programs with variables and parameters. The generation of such test cases requires new models and techniques. In this thesis we have achieved two objectives.

First, we have introduced a new model called Input-Output Symbolic Transition Systems (IOSTS) which explicitly includes all the data of a reactive system.

Second, we have proposed and implemented a new test generation technique that symbolically treats all the data of a system by combining the test generation approach proposed by T. Jéron and his colleagues from IRISA/INRIA Rennes and the Verimag laboratory, Grenoble (*see* for instance, [Fernandez et al., 1996] and [Jéron, 2004]) with techniques of abstract interpretation (*see* [Cousot and Cousot, 1976], [Cousot and Cousot, 1977]).

1.4.2 Plan of the Thesis

The rest of the thesis is separated into three parts organized as follows.

Part I consists in two chapters describing the state of the art for black-box conformance testing. More precisely:

In **Chapter 2** we describe the formal background used in the remaining chapters of the thesis, *i.e.* we introduce the concepts presented in "Formal Methods in Conformance Testing" [ISO/IEC, 1996], and briefly report on new developments that have occurred in the field of formal conformance testing in the last few years. In particular, we attempt at making the reader familiar with the main concepts used in black-box conformance testing.

In **Chapter 3** we focus on testing finite state machines and (input-output) labeled transition systems. We present different kinds of conformance relations (*e.g.* conf, ioconf, ioco) for (IO)LTS, and different approaches for automatic test generations. Finally, we describe some of existing tools used in conformance testing of reactive systems.

Part II is the core of the thesis. It contains four chapters where we introduce a symbolic model used to express the operational semantics of reactive systems and a symbolic technique for test generation based on this model. The second part of our work begins with the motivations for symbolic test generation. The rest is organized as follows:

In **Chapter 4** we define an extension of the input-output labeled transition system model (called IOLTS) that explicitly includes data of a reactive system. Then, we describe the syntax and semantics of IOSTS, and introduce some subclasses of IOSTS used in symbolic test generation. The work contained in this chapter is an extended version of [Rusu et al., 2000].

In **Chapter 5** we introduce the two main operations on IOSTS which are used in our test generation technique and in the process of test execution. They are called product and parallel composition. In this chapter, we also present several relationships concerning the set of traces of the IOSTS obtained after the product or parallel composition operations. The operations defined in the chapter are inspired from those presented in [Rusu et al., 2000].

In **Chapter 6** we describe the theory of conformance testing that serves as a basis for the symbolic test generation method described in the next chapter and realized as the Symbolic Test Generator (STG) tool. The work presented in this chapter, is mainly inspired from the theory of conformance

testing developed by J. Tretmans (*see* for example, [Tretmans, 1994], [Tretmans, 1996b]) and from the research which has been done in the VerTeCs team, IRISA (see [Rusu et al., 2000], [Morel, 2000], [Jard and Jéron, 2002]). In our theory of conformance testing, behaviors of specifications and implementations under test are modeled with input-output symbolic transition systems, and the conformance relation is defined as a partial inclusion of their traces. At the end of the chapter, we define a notion of correct test cases with respect to specifications and test purposes.

In **Chapter 7** we discuss the symbolic test generation method implemented in the Symbolic Test Generator (STG) tool. The purpose of this method is to compute a test suite starting from a given specification and a test purpose. The three main steps of our symbolic test generation method are: (1) computing a synchronous product SP between the specification and the test purpose, (2) eliminating internal action and nondeterminism from SP, *i.e.* building the visible behaviors of SP denoted SP_{vis}, and (3) eventually, selecting the behaviors of SP_{vis} that are accepted by the test purpose. At the end of the chapter, we show that the generated test case covers all behaviors of the specification selected by the test purpose, and is correct.

Part III is devoted to the implementation of the symbolic test generation method proposed in the previous part of the thesis, and to a case study for which this implementation was used. More precisely:

In **Chapter 8** we use the theory and algorithms presented in **Part II**, in order to implement a prototype for automatic symbolic test case generation called STG (Symbolic Test Generator) [Clarke et al., 2002], [Clarke et al., 2001b], [Clarke et al., 2001c]. Then we describe the use of STG in the testing of a communication protocol that transfers a huge data file through lossy channels as a sequence of small packets. The protocol is based on the well-known alternating bit protocol, but it only allows a bounded number of retransmissions of each packet. Finally, we compare STG with other existing tools used for conformance test generation of reactive systems.

The thesis ends with a summary of the main results achieved during my PhD study at IRISA/INRIA Rennes, France, and a discussion on some perspectives (*see* Chapter 9, page 239).

Part I

State of the Art in Conformance Testing

Chapter 2

Formal Methods in Conformance Testing

In this chapter we introduce the background for formal testing, which serves as a fundamental basis for the work presented in this thesis that allows the automatic generation of symbolic tests for reactive systems. We first briefly discuss new developments in the field of testing and motivates the use of formal methods in automatic testing of reactive systems. We then summarize the main concepts of conformance testing which are introduced in the standards IS-9646 [ISO, 1991] *and FMCT* [ISO/IEC, 1996]*, and formalized in* [Tretmans, 1992]*. Finally, we present a test architecture in which a test suite is executed. It is important to notice that all concepts introduced in this chapter are given in generic level, i.e. they are independent of any particular formal method.*

Nowadays, software and hardware are getting more and more complex. They usually consists of a variety of components which can be produced by different manufacturers. This leads to compatibility problems between different products. In order to unify development processes and to define methods for testing conformance between a product and its specification, the International Organization for Standardization (ISO) provided the international standard IS-9646 [ISO, 1991] containing a methodology and a framework for testing of communication protocols, for example SSCOP [Kahlouche et al., 1999]. It turned out that this standard is also applicable to testing of other reactive systems, for example, smart cards [Clarke et al., 2001a], and hardware [Kahlouche et al., 1999]. The IS-9646 standard has been mainly oriented towards practical needs. It incorporates practical experience of test experts who have been involved in concrete testing. However, the formalism in IS-9646 is limited to definition of

the TTCN [ISO/IEC/JTC1/SC21, 1992] language which is now widely used in telecommunication area for the description of test suites.

The increasing usability of formal methods in the software/hardware development processes, as well as new developments in the theory of testing, has influenced the evolution of testing technology. This led to a joint project between the International Organization for Standardization (ISO) and the International Telecommunication Union (ITU) on "Formal Methods in Conformance Testing" (FMCT) [ISO/IEC, 1996]. The goal of this project was to "establish a theory and framework [...] which may be used to assess conformance of an implementation to behavior specified in a formal description". The FMCT standard defines the main concepts in conformance testing, which were formalized in [Tretmans, 1992], and which are summarized in this chapter.

2.1 Conformance

The notion of conformance, according to [ISO/IEC, 1996], is linked to implementations under test and specifications.

2.1.1 Specification

A *specification* of a reactive system is a formal description of behaviors that fixes the properties of the system. In general, the specification is developed from users' requirements and expressed using either natural language, such as English or French; or specialized description languages, such as SDL [ITU-T, 1994] or LOTOS [ISO/IEC, 1988]. The semantics of the specification can be represented, for example, by temporal logic [Pnueli, 1986], algebraic specifications [Dauchy et al., 1993], extended finite state machines [Petrenko et al., 1999], transition systems [Tretmans, 1992]. In this thesis, we use the following notations:

- SPECS for the universe of formal specifications, and
- *Spec* for a single specification that belongs to SPECS.

2.1.2 Implementation

An *implementation under test* is a concrete executable system, that is, one of the goals of a development process. The implementation can consist of, for instance, a single chip, hardware components or pieces of executable code. We denote by:

- IMPS – the universe of implementations, and
- $iut \in \mathsf{IMPS}$ – a concrete implementation under test.

In order to reason formally about an informal implementation $iut \in \mathsf{IMPS}$, it is assumed that each concrete implementation iut can be modeled by a formal object $\mathcal{I}_{iut}$ called a *model* of iut. The universe of the models of all implementations under test is denoted by MODS . In the literature on testing theory (*see* [Bernot, 1991]), this assumption is referred to as the *"testing hypothesis"*, and can be formulated in a semi-formal manner as follows:

Testing Hypothesis

$$\forall\ iut \in \mathsf{IMPS}\ \exists\ \mathcal{I}_{iut} \in \mathsf{MODS}.\ [iut \text{ is modeled by } \mathcal{I}_{iut}] \tag{2.1}$$

□

It is important to notice that the testing hypothesis *does not* imply that a model of the implementation under test iut is *known*, it only assumes that such a model *exists*.

2.1.3 Conformance as an Implementation Relation

To define *conformance* between an implementation under test $iut \in \mathsf{IMPS}$ and a specification $Spec \in \mathsf{SPECS}$, we use the notion of an implementation relation [Brinksma et al., 1990]. An *implementation relation* imp is a relation between the set of implementation models MODS and the set of specifications SPECS, *i.e.*

$$\mathsf{imp} \subseteq \mathsf{MODS} \times \mathsf{SPECS} \tag{2.2}$$

Definition 2.1 (Conformance) An implementation $iut \in \mathsf{IMPS}$ is *conformant to* a specification $Spec \in \mathsf{SPECS}$ if the existing model $\mathcal{I}_{iut} \in \mathsf{MODS}$ of iut is imp-related to $Spec$, *i.e.*

$$iut\ \mathsf{conformant\ to}\ Spec \quad \triangleq \quad \mathcal{I}_{iut}\ \mathsf{imp}\ Spec$$

□

The approach based on the implementation relation is shown on Figure 2.1. The implementation $iut \in \mathsf{IMPS}$ correctly implements the specification $Spec \in \mathsf{SPECS}$ if

(1) it is modeled by $\mathcal{I}_{iut}$ belonging to the set $\mathcal{M}_{iut} \subseteq \mathsf{MODS}$ of models, and

(2) the model $\mathcal{I}_{iut}$ implements $Spec$ according to the implementation relation imp.

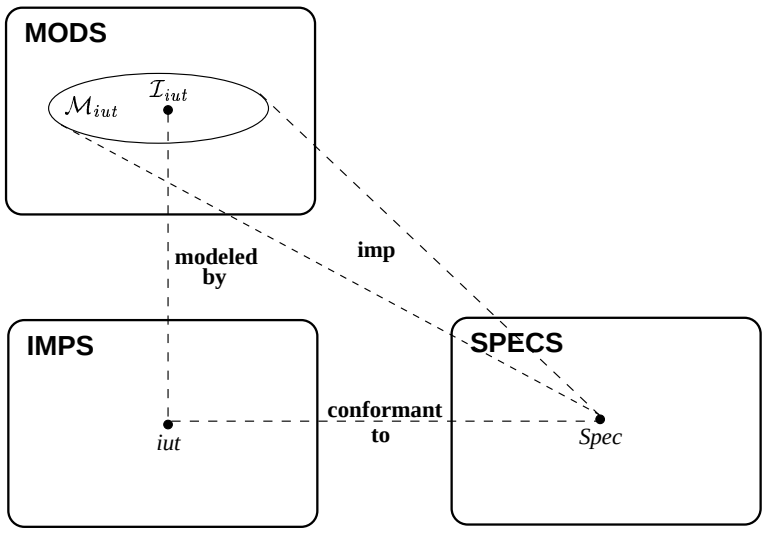


Figure 2.1: Conformance using an implementation relation.

It is important to notice that conformance between an implementation and a specification depends on the chosen implementation relation. Some relations are refusal preorder [Phillips, 1987], *conf* [Brinksma, 1988], and *ioco* [Tretmans, 1996a] which are discussed in Section 3.2.1 (*see* page 34).

2.2 Conformance Testing

Conformance testing assesses conformance of an unknown implementation under test ($iut \in \mathsf{IMPS}$) to its specification ($Spec \in \mathsf{SPECS}$) by means of test experiments. Experiments consist of stimulating *iut* in certain ways and observing its reactions (this process is called *test execution*). The decision made about conformance is based on these observations. Each such experiment is called a *test case*. We denote by:

- TESTS – the universe of test cases,
- $TC \in \mathsf{TESTS}$ – a single test case, and
- $TS \subseteq \mathsf{TESTS}$ – a set of test cases, which is called a *test suite*.

The purpose of the next two subsections is to describe the test execution process which is always performed in a concrete *test architecture*.

2.2.1 Test Architecture

A *test architecture* is an abstract description of the environment in which an implementation under test ($iut \in \mathsf{IMPS}$) is embedded, and where it communicates with a tester. The test architecture that is used throughout the thesis is depicted

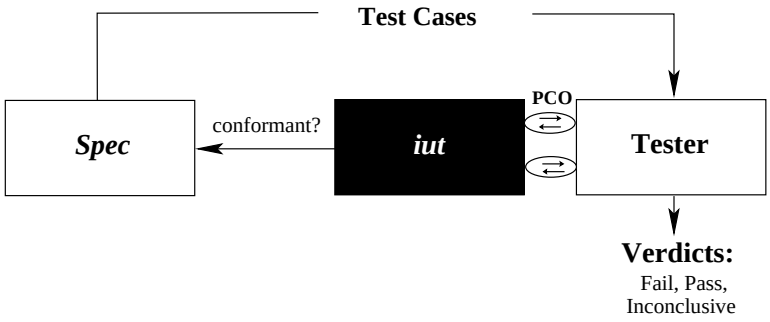


Figure 2.2: Test architecture and test execution.

in Figure 2.2. In order to perform testing of a reactive system, test cases have to be produced from a formal description of the system's behaviors, *i.e.* from a specification $Spec \in \mathsf{SPECS}$. Then, they have to be given to a *tester*, which is a program that executes test cases against a concrete implementation under test *iut*. As the thesis focuses on conformance testing (a kind of functional testing defined in Section 1.2.2.2, page 5), *iut* is considered as a black-box, which means that the code of *iut* is unknown and only the behavior of *iut* can be observed and analyzed. When *iut* interacts with the tester, its behavior becomes visible through interfaces called *points of control and observation* (*see* Figure 2.2), in short PCO.

The test architecture described above is ideal for real testing as the tester directly communicates with *iut*. However, due to practical limitations, very often the tester cannot access *iut* directly. Therefore, a *test context*, *i.e.* a system where *iut* is embedded when it is tested, has to be taken into account. The last situation is not the subject of this thesis, it is discussed in [Heerink, 1998].

2.2.2 Test Execution

A *test execution* is the process of stimulating a concrete implementation under test $iut \in \mathsf{IMPS}$ by: (1) performing various test events which are specified in a test case $TC \in \mathsf{TESTS}$, (2) observing the responses from the given *iut*, and (3) generating a test verdict based on these observations.

We denote by OBS the domain of all observations. Then, in order to reason formally about an execution process, we model it with the following function:

$$\mathsf{exec}: \quad \mathsf{TESTS} \times \mathsf{MODS} \longmapsto \mathcal{P}(\mathsf{OBS}) \tag{2.3}$$

This function computes, for each test case $TC \in \mathsf{TESTS}$ and each real implementation under test $iut \in \mathsf{IMPS}$ modeled by $\mathcal{I}_{iut} \in \mathsf{MODS}$, a subset of observations $obs \subseteq \mathcal{P}(\mathsf{OBS})$.

The purpose of testing is to indicate whether observations obtained after a test execution are correct or not. For that, we introduce a *verdict function* which assigns to each observation a pass or fail verdict.

$$\mathsf{verdict} : \quad \mathsf{OBS} \longmapsto \{Fail, Pass\} \tag{2.4}$$

A *Fail* verdict means that a divergence is detected between the implementation *iut* and its specification *Spec*. A *Pass* verdict [1] means that *iut* behaves in conformance to *Spec* for the given observation.

Finally, using Functions (2.3) and (2.4), we characterize situations where an implementation under test *iut* (which is modeled with $\mathcal{I}_{iut} \in \mathsf{MODS}$) *fails* or *passes* a test case TC:

$$\begin{array}{lcl} iut \text{ fails } TC & \triangleq & \exists\ \sigma \in \mathsf{exec}(TC, \mathcal{I}_{iut})\ .\ [\ \mathsf{verdict}(\sigma) = Fail\ ] \\ iut \text{ passes } TC & \triangleq & \neg(iut \text{ fails } TC) \end{array}$$

These notions can be generalized for a test suite $TS \subseteq \mathsf{TESTS}$ as follows:

$$iut \text{ fails } TS \quad \triangleq \quad \exists\ TC \in TS\ .\ [\ iut \text{ fails } TC\ ]$$
$$iut \text{ passes } TS \quad \triangleq \quad \neg(iut \text{ fails } TS) \quad \triangleq \quad \forall\ TC \in TS\ .\ [\ iut \text{ passes } TC\ ]$$

2.3 Test Suite Properties

This section studies the coherence between the notion of conformance (*see* Section 2.1.3, page 15) applied to concrete implementations of the reactive system and its formal specification, and the notion of successful test suite execution which was defined in Section 2.2. It formalizes and discusses the properties of test suites, which we want to obtain while generating the test suites automatically.

Soundness. A test suite $TS \subseteq \mathsf{TESTS}$ is *sound* if all implementations $iut \in \mathsf{IMPS}$ which are conformant to a specification $Spec \in \mathsf{SPECS}$ pass all tests belonging to this test suite TS, *i.e.*

$$\forall\ iut \in \mathsf{IMPS}\ .\ [\ (iut \text{ conformant to } Spec) \implies (iut \text{ passes } TS)\ ] \tag{2.5}$$

[1] Later in the thesis the notion of a *Pass* verdict is refined by the introduction of an *Inconclusive* verdict. This verdict indicates the violation of a scenario (called *test purpose*) for which a test case is generated.

The soundness property is achievable for practical testing, but it is not sufficient as the incorrect implementations, *i.e.* implementations that are non-conformant to a specification, can pass the test suite. Thus, a test suite accepting all implementations is also sound.

Exhaustiveness. A test suite $TS \subseteq \mathsf{TESTS}$ is *exhaustive* if all implementations $iut \in \mathsf{IMPS}$ leading to *Pass* during the test execution of TS, are conformant to a specification $Spec \in \mathsf{SPECS}$, *i.e.*

$$\forall\ iut \in \mathsf{IMPS}\ .\ [\ (iut\ \mathsf{passes}\ TS) \implies (iut\ \mathsf{conformant\ to}\ Spec)\ ] \tag{2.6}$$

The exhaustiveness property of a test suite is achievable only in theory, as it is often the case when the test suite should be infinite, for instance, when a specification of the system under test contains loops.

Completeness. A test suite $TS = \{TC_1, \ldots, TC_n\} \subseteq \mathsf{TESTS}$ is called *complete* if it is sound and exhaustive, *i.e.*

$$\forall\ iut \in \mathsf{IMPS}\ .\ [\ (iut\ \mathsf{conformant\ to}\ Spec) \iff (iut\ \mathsf{passes}\ TS)\ ] \tag{2.7}$$

It is important to note that a complete test suite distinguishes exactly between all conformant and non-conformant implementations.

Discussion. It is theoretically possible to construct complete test suites [Tretmans, 1992], *i.e.* a test suite which is sound and exhaustive, but for practical testing the exhaustiveness property becomes a very strong requirement as it is not possible to execute an infinite number of tests in a limited period of time. Thus, the standard IS-9646 [ISO, 1991] suggests to use a weaker requirement for real testing, namely soundness.

Chapter 3

Test Generation

In this chapter we present a brief overview of research and developments that have been done in conformance testing in last 35-40 years. We first describe several test generation methods based on the model of finite state machines. Then, we discuss some limitations of these methods that have activated the research on different kinds of transition systems (e.g. labeled transition systems, input-output labeled transition systems). Next, we present several testing relations defined on these transition systems, and used as correctness criteria in conformance testing. Finally, we describe some existing techniques for automatic test generation based on transition systems.

3.1 Test Generation based on Finite State Machines

A *Finite State Machine* (FSM) [Gill, 1962] is an abstract model which consists of a set of *states*, an *alphabet*, and a *transition function* that maps symbols of alphabet and current states to a next state. Computation begins in some state with an input string. It changes to new states depending on the transition function. There exist many variants of FSM, for instance, *Moore machines* that produce an output for each state after receiving an input, or *Mealy machines* that produce an output for each transition after receiving an input. The Mealy machines are widly used for the purposes of testing.

This section gives a formal definition of a Mealy machine and briefly describes several methods for test generation based on this model. The detailed description of all these methods is given in the survey [Lee and Yannakakis, 1996] written by D. Lee and M. Yanakakis.

3.1.1 Model: Mealy Machines

A Mealy machine [Gill, 1962] consists of states and transitions between states. Each transition is labeled by an input and output action. Formally:

Definition 3.1 (Mealy Machine) A *Mealy machine* is a tuple $\langle S, \Sigma, \delta, \omega \rangle$, where

- S is a nonempty, finite *set of states*,
- $\Sigma = \Sigma^? \cup \Sigma^!$ is a nonempty, finite *alphabet of actions* which consists of two disjoint alphabets of input $\Sigma^?$ and output $\Sigma^!$ actions,
- $\delta : S \times \Sigma^? \mapsto S$ is the partial *transition function*,
- $\omega : S \times \Sigma^? \mapsto \Sigma^!$ is the partial *output function*.

□

Before illustrating the definition of Mealy machines with an example, we remark that they are *deterministic* in the sense of the classical definition (*see* [Hopcroft and Ullman, 1979] or [Lewis and Papadimitriou, 1981]) which is $\forall\ s \in S,\ a \in \Sigma^?\ .\ [card(\delta(s,a)) \leq 1]$. This is due to the fact that both δ and ω are functions (*see* the definition above). Moreover, in order to be able to perform the execution of sequences of input actions on some Mealy machine, and observe resulting sequences of states or output actions, we extend the δ and ω functions as follows: $\delta^* : S \times (\Sigma^?)^* \mapsto S$ and $\omega^* : S \times (\Sigma^?)^* \mapsto (\Sigma^!)^*$.

Example 3.1 An example of Mealy machine $\mathcal{A}$ is presented as the transition diagram shown on Figure 3.1 (this example has been taken from the thesis of P. Morel [Morel, 2000]). This machine $\mathcal{A}$ has four states s_1, s_2, s_3 and s_4; two

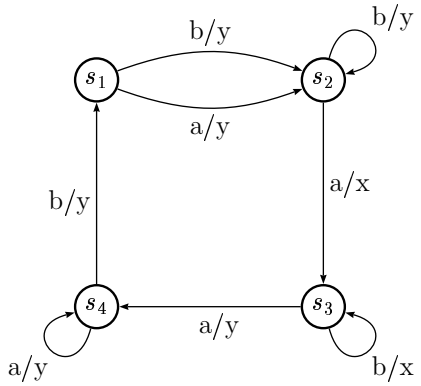


Figure 3.1: An example of Mealy machine $\mathcal{A}$.

input actions a and b; and two output actions x and y. Then, we explain how $\mathcal{A}$ behaves in different states, for instance:

(1) in the state s_1, the Mealy machine $\mathcal{A}$ produces the output action y by executing one of its input actions a or b, and moves to the state s_2,

(2) in the state s_3 the Mealy machine $\mathcal{A}$ can produce two different output actions and moves to two different states. More precisely, after receiving the input action a, $\mathcal{A}$ produces the output action y and moves to the state s_4, however after receiving the input action b, $\mathcal{A}$ produces the output action x and stays in the same state s_3.

□

Before stating the problem of conformance testing and explaining the use of Mealy machines in it, we present several useful properties of Mealy machines. Consider a Mealy machine $\mathcal{M} = \langle S, \Sigma, \delta, \omega \rangle$. We say that two states $s_i, s_j \in S$ of $\mathcal{M}$ are *equivalent* if and only if for each input sequence the given machine $\mathcal{M}$ produces the same output sequence, *i.e.* $\forall\ \sigma \in (\Sigma^?)^*\ .\ [\omega^*(s_i, \sigma) = \omega^*(s_j, \sigma)]$. Next, we formulate the following properties of Mealy machines. Similarly we can define the notion of equivalent states for two different Mealy machines which have a same alphabet of input and output actions.

Definition 3.2 (Properties of Mealy Machines)

(1) Two Mealy machines $\mathcal{M}$ and $\mathcal{M}'$ with same alphabet of actions are *equivalent* if and only if for every state of $\mathcal{M}$ there exists a corresponding equivalent state in $\mathcal{M}'$, and vice versa.

(2) A Mealy machine $\mathcal{M} = \langle S, \Sigma, \delta, \omega \rangle$ is *minimized* if and only if it does not have two equivalent states, *i.e.* $\forall\ s_i, s_j \in S\ .\ [\ (s_i \neq s_j) \implies (s_i \text{ is not equivalent to } s_j)\ ]$.

(3) A Mealy machine $\mathcal{M} = \langle S, \Sigma, \delta, \omega \rangle$ is *strongly connected* if and only if for every pair of states $s_i, s_j \in S$ there exists an input sequence $\sigma \in (\Sigma^?)^*$ after which $\mathcal{M}$ moves from s_i to s_j, *i.e.* $\delta^*(s_i, \sigma) = s_j$.

(4) A Mealy machine $\mathcal{M} = \langle S, \Sigma, \delta, \omega \rangle$ is *complete* if and only if all its states accept all input actions of $\mathcal{M}$, *i.e.* for each state $s \in S$ and each input action $a \in \Sigma^?$ the functions $\delta(s, a)$ and $\omega(s, a)$ are defined, *i.e.* δ and ω of $\mathcal{M}$ are total functions.

(5) A Mealy machine $\mathcal{M}$ is *symmetric* if and only if each vertex of its transition diagram has the same number of incoming and outgoing edges.

□

Example 3.2 Consider the Mealy machine $\mathcal{A}$ depicted in Figure 3.1 (*see* page 22). This machine:

(1) does not have any equivalent states. For instance, if we take the states s_1 and s_4, then there exists the sequence of input actions aa such that $\underbrace{\omega^*(s_1, aa)}_{yy} \neq \underbrace{\omega^*(s_4, aa)}_{yx}$.

(2) is minimized as it does not contains equivalent states (*see* the item (1)).

(3) is strongly connected, as each state of this machine is reachable from any other state by some sequence of input actions. For example, we can go from the state s_1 to state s_4 by the following sequence of input actions $abbaba$ (*see* Figure 3.1, page 22).

(4) is complete as for each state (s_1, s_2, s_3, s_4) and each output action (a, b), the machine $\mathcal{A}$ has exactly one transition (*see* Figure 3.1, page 22). For instance, from the state s_1 the machine $\mathcal{A}$ can move to the state s_2 by executing either a or b.

(5) is not symmetric, as the number of incoming and outgoing edges of s_1 and s_2 is different (*see* Figure 3.1, page 22).

□

3.1.2 The Problem of Conformance Testing

Consider a specification *Spec* $\in$ SPECS and an implementation under test *iut* $\in$ IMPS, where (1) *Spec* is modeled by a complete Mealy machine, *i.e.* its transition diagram and output function are known, and (2) *iut* is modeled by unknown Mealy machine ("black-box"), where we can observe only input and output behaviors of *iut*. Then, the *problem of conformance testing* (known also as "fault detection" problem) consists in determining whether the specification *Spec is equivalent* to the implementation *iut*.

This problem cannot be solved without any assumptions made on the given specification and implementation. Indeed, for any testing sequence we can construct a Mealy machine *iut* that is not equivalent to the given machine *Spec*, but produces the same output sequence as *Spec*. Thus, for the rest of the section we assume that:

(1) *Spec* and *iut* are modeled by strongly connected and minimized Mealy machines,

(2) *iut* has the same number of states as *Spec*, and

(3) sometimes we also suppose that *Spec* is symmetric.

Under these assumptions, in order to detect whether the given implementation *iut* is conformant to (*i.e.* equivalent to) the specification *Spec*, it is enough to verify that *iut* does not contain faults of two following types:

Output Faults: in a given state s after receiving an input action a, the implementation *iut* produces an output action x which is different from an output action specified by *Spec*,

Transfer Faults: from a given state after executing a transition labeled with a/x, the implementation *iut* moves to a state s'' which is different from a state specified by *Spec*.

Example 3.3 Consider a specification $\mathcal{A}$ represented by the Mealy machine shown on Figure 3.1 (*see* page 22); and an implementation under test $\mathcal{B}$ whose state diagram shown on Figure 3.2. The implementation $\mathcal{B}$ contains two following

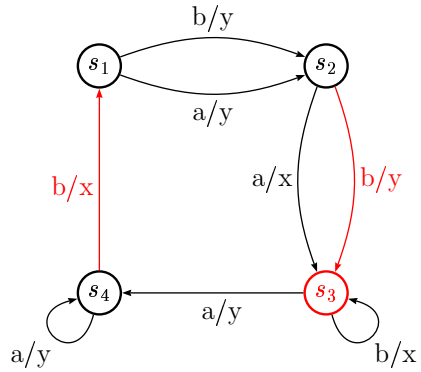


Figure 3.2: An example of Mealy machine $\mathcal{B}$.

faults:

(1) output fault, as it produces the output action x after applying the input action b in the state s_4 (*see* Figure 3.2) while the specification $\mathcal{A}$ produces the output action y in the same situation (*see* Figure 3.1, page 22), and

(2) transfer fault, as $\mathcal{B}$ moves to the state s_3 after executing the input action b in the state s_2 (*see* Figure 3.2) while in the same situation $\mathcal{A}$ rest in the state s_2 (*see* Figure 3.1, page 22).

□

Finally, we explain how to construct elementary tests that are able to detect output and transfer faults of the given implementation *iut*. Each *elementary test* consists in checking whether each transition of *iut* works as it is described in the specification *Spec*. Formally, for each transition $s \xrightarrow{a/x} s'$ of *Spec* it is necessary first to drive *iut* to the state s and to execute the required input action a in this state s; and then:

- to verify the produced by *iut* output (output faults testing), and
- to identify the target state of *iut* by a sequence of input actions (transfer faults testing).

The purpose of the test generation based on Mealy machines is to put all elementary tests together in order to obtain a test suite (*i.e.* a set of test cases) of minimal size. Moreover, this test suite must be sound and exhaustive, meaning that (1) it does not reject conformant implementation under test (*soundness*), and (2) non-conformant implementation is rejected by a test case belonging to this test suite (*exhaustiveness*).

3.1.3 Test Generation Methods

The following section is addressed to the different methods that allows to generate test cases starting from a specification modeled as a Mealy machines. Other detailed descriptions of these methods can be found in [Lee and Yannakakis, 1996], [Morel, 2000], or [Jéron, 2004].

3.1.3.1 Transition Tour (TT)

A *transition tour* is a sequence of input actions that executes each transition of a specification *Spec* that is modeled by a *strongly connected*, *complete* and *minimized* Mealy machine, at least once. It is important to notice that the transition tour does not allow to identify a state in which the specification *Spec* will move after executing this sequence. Thus, in general this sequence cannot be used for detecting transfer faults. However, this problem can be fixed by a modification of the Mealy machine, *i.e.* making it be able to report about its current state.

The problem of finding a minimal transition tour is well-known problem of graph theory. It consists in computating of an *Euler tour* which is a sequence of transitions that starts and ends at the same state and contains each transition of a given Mealy machine exactly once. However, the Euler tour exists only in *strongly connected* and *symmetric* Mealy machines (*see* the items (3) and (5) of Definition 3.2, page 23). Therefore, any specification represented by a non-symmetric Mealy machine must be transformed into symmetric one. This can be

done by duplication of some edges in the corresponding transition diagram (this operation in graph theory is as *augmentation* of the original graph). Each duplicated edge results in a transition that executes more than once in the resulting transition tour. The first concrete algorithm which computes an Euler tour was proposed by J. Edmonds and E.L. Johnson in [Edmonds and Johnson, 1973].

The problem of finding a transition tour in a strongly connected but non-symmetric Mealy machine is known as the *Chinese Postman Problem*.

Example 3.4 Consider the Mealy machine $\mathcal{A}$ whose transition diagram is depicted in Figure 3.1 (*see* page 22). It is strongly connected, complete and minimized (*see* Example 3.2, page 24). However, according to the same example the machine $\mathcal{A}$ it is not symmetric, thus, $\mathcal{A}$ does not contain an Euler tour.

In order to make $\mathcal{A}$ symmetric we duplicated the following transitions: $s_2 \xrightarrow{a/x} s_3$, $s_3 \xrightarrow{a/y} s_4$ and $s_4 \xrightarrow{b/y} s_1$. The result of this duplication is show on Figure 3.3. Then, for the obtained strongly connected Mealy machine $\mathcal{A}'$ we compute a tour

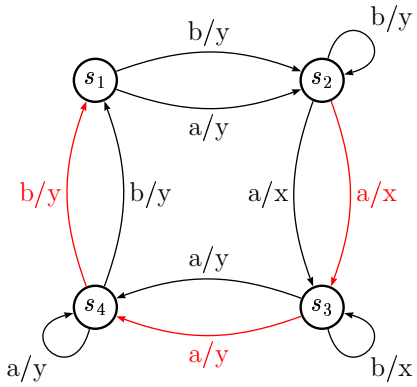


Figure 3.3: Symmetric Mealy machine $\mathcal{A}'$.

which (1) starts and ends in, for example, the state s_1, and (2) does not contain a same transition more then once. For the state s_1 this path can be following: $\rho_{s_1} : s_1 \xrightarrow{a/y} s_2 \xrightarrow{a/x} s_3 \xrightarrow{a/y} s_4 \xrightarrow{b/y} s_1 \xrightarrow{b/y} s_2 \xrightarrow{a/x} s_3 \xrightarrow{a/y} s_4 \xrightarrow{b/y} s_1$. Next, we remove all transitions used in ρ_{s_1} from $\mathcal{A}'$, and obtain four strongly connected and symmetric Mealy machines. For all of them we perform the same computation as for $\mathcal{A}'$, and obtain the three following tours:

$$\rho_{s_2} : \quad s_2 \xrightarrow{b/y} s_2$$
$$\rho_{s_3} : \quad s_3 \xrightarrow{b/x} s_3$$
$$\rho_{s_4} : \quad s_4 \xrightarrow{a/y} s_4$$

Finally, using the above computed tours we can construct transition tours for the Mealy machine $\mathcal{A}'$. One of these tours is:

$$s_1 \xrightarrow{a/y} s_2 \xrightarrow{b/y} s_2 \xrightarrow{a/x} s_3 \xrightarrow{a/y} s_4 \xrightarrow{a/y} s_4 \xrightarrow{b/y} s_1 \xrightarrow{b/y} s_2 \xrightarrow{a/x} s_3 \xrightarrow{b/x} s_3 \xrightarrow{a/y} s_4 \xrightarrow{b/y} s_1$$

□

The transition tour algorithm has first been applied for test generation by S. Naito and M. Tsunoyama in 1981 (*see* [Naito and Tsunoyama, 1981]). Then, in 1986 C.A. Uyar and A.T. Dahbura [Uyar and Dahbura, 1986] applied the algorithm of Euler tour to conformance testing. Finally, in collaboration with A.V. Aho and D. Lee, they have extended this algorithm and published it as [Aho et al., 1988].

3.1.3.2 Distinguishing Sequence (DS)

Consider a specification *Spec* that is modeled by a *strongly connected*, *complete* and *minimized* Mealy machine $\langle S, \Sigma, \delta, \omega \rangle$, where $\Sigma = \Sigma^? \cup \Sigma^!$. Then, a *distinguishing sequence* of *Spec* is a sequence of input actions that produces a different sequence of output actions for each state of *Spec*. Formally the existence of distinguishing sequence in *Spec* can be characterized by the following formula:

$$\exists\ DS \in (\Sigma^?)^*\ \forall\ s_i, s_j \in S\ .\ [\ (s_i \neq s_j)\ \Longrightarrow\ (\omega^*(s_i, DS) \neq \omega^*(s_j, DS))\ ]$$

It is important to note that the technique of distinguishing sequence detects both kinds of faults, *i.e.* output and transfer. However, a distinguishing sequence may only exist for *minimized* Mealy machines. Moreover, its existence is very rare even for minimized machines.

Example 3.5 Let us first consider the Mealy machine $\mathcal{A}$ shown on Figure 3.1 (*see* page 22). It is strongly connected, complete and minimized (*see* Example 3.2 on page 24). However, $\mathcal{A}$ does not contain any distinguishing sequence. Indeed, if this sequence starts with:

- the input action a, then it is not possible to distinguish the states s_3 and s_4 as after receiving a they produce the same output action y and move to the same state s_4,
- the input action b, then the state s_1 and s_2 cannot be distinguished for the similar reason.

Next, we consider the Mealy machine shown on Figure 3.4. It is easy to check that it is strongly connected, complete and minimized. Moreover, it contains distinguishing sequences. For instance, the sequence bb is the minimal distinguishing sequence of this machine. Indeed, if we apply the sequence bb to the different state of the machine, then we obtain different output sequences: $\omega^*(s_1, bb) = xy$, $\omega^*(s_2, bb) = yy$, $\omega^*(s_3, bb) = xx$ and $\omega^*(s_4, bb) = yx$. □

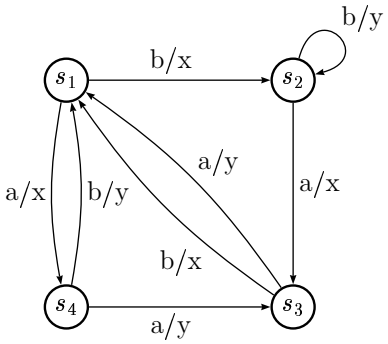


Figure 3.4: A Mealy machine containing distinguishing sequence.

The concepts of distinguishing sequence was defined as early as 1965 by E.F. Moore in his first papers on finite state machines. Then, there were developed by [Gill, 1962], [Hennie, 1964], [Gonenc, 1970], [Hsieh, 1971] and summarized in [Kohavi, 1978].

3.1.3.3 Unique Input-Output Sequence (UIO)

Consider a specification *Spec* that is modeled by a *strongly connected*, *complete* and *minimized* Mealy machine $\langle S, \Sigma, \delta, \omega \rangle$, where $\Sigma = \Sigma^? \cup \Sigma^!$. Then, a *unique input-output sequence* of a given state $s_i \in S$ of *Spec* is a sequence of input actions UIO_i that distinguishes s_i from all other states of the specification *Spec* (*i.e.* a sequence of output actions obtained after applying UIO_i in the state s_i is different from all sequences of output actions obtained after applying the same sequences UIO_i in all other states of *Spec*). Formally:

$$\forall\, s_i \in S\ \exists\ UIO_i \in (\Sigma^?)^*\ \forall\ s_j \in S\ .\ [\ (s_i \neq s_j)\ \Longrightarrow\ (\omega^*(s_i, UIO_i) \neq \omega^*(s_j, UIO_i))\ ]$$

Notice that the existence of UIO_i sequences for each state s_i of *Spec* is not guaranteed even if *Spec* is minimized.

Next, we illustrate the construction of unique input-output sequences on an example, and explain how to generate a test sequence using the information about them.

Example 3.6 (Generation of Test Sequences using UIO) Consider a Mealy machine $\mathcal{C}$ whose transition diagram is depicted in Figure 3.5. It is not hard to check that the given machine $\mathcal{C}$ is complete, strongly connected and minimized. Each state of this machine has its unique input-output sequence:

(1) For s_1, UIO_1 is equal to a. Indeed, the sequence of output actions obtained after a in the state s_1, *i.e.* $\omega^*(s_1, a) = y$, is different from $\omega^*(s_2, a) = \omega^*(s_3, a) = x$.

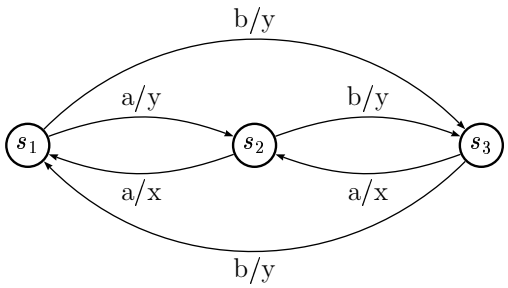


Figure 3.5: The Mealy machine $\mathcal{C}$ illustrating the UIO method.

(2) For s_2, UIO_2 is equal to aa. This is due to the fact that $\omega^*(s_2, aa) = xy$ is different from both sequences $\omega^*(s_1, aa) = yx$ and $\omega^*(s_3, aa) = xx$.

(3) For s_3, UIO_3 is equal to ba. This is because $\omega^*(s_3, ba) = yy$ is different from $\omega^*(s_1, ba) = \omega^*(s_2, ba) = yx$.

Using the information about the unique input-output sequences for each state of $\mathcal{C}$, we explain how to generate a test sequence. Intuitively, we have to compute a transition tour containing all the above computed unique input-output sequences. This problem is solved by introducing *pseudo transitions*. This means that for each unique input-output sequence UIO_i $(i = 1, 2, 3)$ we create a new pseudo transition that starts in the origin of UIO_i, ends in the target of UIO_i, and is labeled with input and output sequences corresponding to UIO_i. For instance,

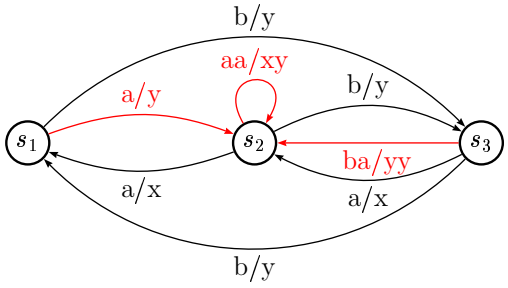


Figure 3.6: The Mealy machine $\mathcal{C}'$ obtained from the Mealy machine $\mathcal{C}$ (*see* Figure 3.5, page 30) by adding pseudo transitions shown in red.

for $UIO_2 = aa$ with origin and target in s_2, the Mealy machine $\mathcal{C}$ was augmented with the loop on s_2 that is labeled with aa/xy (*see* Figure 3.6).

Next, in order to generate a test sequence, we compute a transition tour (*see* Subsection 3.1.3.1, page 26) for a Mealy machine with pseudo transitions, which visits each pseudo transition once. For this purpose we can use either use the algorithm that solves the Chinese Postman Problem, or the algorithm that computes Euler tour. Remember in order to use the last algorithm a Mealy machine

must be symmetric, thus, we must transform the machine $\mathcal{C}'$ into symmetric one. The resulting Mealy machine $\mathcal{C}''$ of this transformation is depicted in Figure 3.7

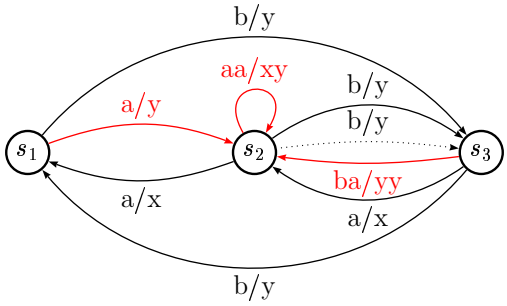


Figure 3.7: The symmetric Mealy machine $\mathcal{C}''$ obtained from $\mathcal{C}'$ shown on Figure 3.6 (*see* page 30).

(*see* page 31). This machine was obtained by duplication of the transition $s_2 \xrightarrow{b/y} s_3$ (*see* the dotted edge on Figure 3.7). One of testing sequences generated from $\mathcal{C}''$ by the transition tour method is:

$$s_1 \xrightarrow{a/y} s_2 \xrightarrow{aa/xy} s_2 \xrightarrow{b/y} s_3 \xrightarrow{ba/yy} s_2 \xrightarrow{b/y} s_3 \xrightarrow{b/y} s_1 \xrightarrow{b/y} s_3 \xrightarrow{a/x} s_2 \xrightarrow{a/x} s_1 \tag{3.1}$$

where the pseudo transitions $s_2 \xrightarrow{aa/xy} s_2$ and $s_3 \xrightarrow{ba/yy} s_2$ can be decomposed into two following sequences: $s_2 \xrightarrow{a/x} s_1 \xrightarrow{a/y} s_2$ and $s_3 \xrightarrow{b/y} s_1 \xrightarrow{a/y} s_2$ respectively.

The method based on UIO sequences produces conformance test of a good quality. However, it has a several problems. First, not all states of a Mealy machine have a UIO sequence, and even if they do, the UIO sequence may be too long to derive automatically. Second, in order to generate exhaustive test cases (*i.e.* they reject all non-conformant implementations under test) we must make supplementary assumption on implementations under test, *i.e.* we have to suppose that an implementation has the same UIO sequences as its specification. If we do not make this assumption then the generated test sequences may accept some non-conformant implementations. For instance, consider a specification

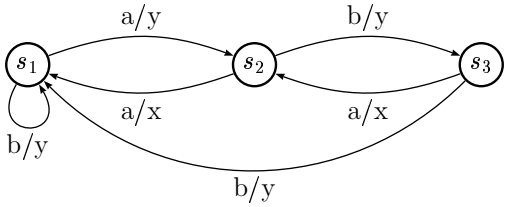


Figure 3.8: A Mealy machine representing an implementation under test.

that is modeled by the Mealy machine $\mathcal{C}$ (*see* Figure 3.6, page 30), and an

implementation under test depicted as the Mealy machine on Figure 3.8. It is clear that this implementation is not conformant to the specification $\mathcal{C}$, as, for instance, after applying the sequence ba in the state s_1, the specification $\mathcal{C}$ produces the sequence yx, but the implementation produces yy. However, the implementation is not rejected by the test sequence shown as Formula (3.1). This is due to the fact that the sequence of input actions ba is an UIO sequence for the state s_3 of the specification $\mathcal{C}$ and is not an UIO sequence for the same state of the implementation (we obtain the same output sequence yy after applying the sequence ba in the states s_3 and s_1). □

The concept of UIO sequences was first introduced by E.P. Hsieh in his paper [Hsieh, 1971]. Then, in 1985 K. Sabnani and A. Dahbura [Sabnani and Dahbura, 1985] applied this concept in conformance testing.

3.1.3.4 Characterizing Sequences (W-method)

Consider a specification *Spec* that is modeled by a *strongly connected*, *complete* and *minimized* Mealy machine $\langle S, \Sigma, \delta, \omega \rangle$, where $\Sigma = \Sigma^? \cup \Sigma^!$. Then, a *characterizing sequence* is a sequence of input actions that distinguishes two given different state of *Spec*. Formally,

$$\forall\ s_i, s_j \in S\ \exists\ w_{ij} \in (\Sigma^?)^*\ .\ [\ (s_i \neq s_j)\ \Longrightarrow\ (\omega^*(s_i, w_{ij}) \neq \omega^*(s_j, w_{ij}))\ ]$$

It is important to notice that a characterizing sequence *always exists* for each state of *minimized* Mealy machine.

The W-method constructs the set of characterizing sequences for all pairs of different states of the given Mealy machine, and then it constructs a test sequence in a similar way as the UIO method (*see* Subsection 3.1.3.3, page 29).

Example 3.7 Consider the Mealy machine $\mathcal{A}$ whose transition diagram is depicted in Figure 3.1 (*see* page 22). It is strongly connected, complete and minimized (*see* Example 3.2, page 24). Then, the set of characterizing sequences of this machine $\mathcal{A}$ consists of three input sequences a, b and aa. Indeed:

(1) aa is a characterizing sequence of s_1 and s_4 as $\underbrace{\omega^*(s_1, aa)}_{yx} \neq \underbrace{\omega^*(s_4, aa)}_{yy}$,

(2) a is a characterizing sequence of:

- s_1 and s_2 as $\underbrace{\omega^*(s_1, a)}_{y} \neq \underbrace{\omega^*(s_2, a)}_{x}$,
- s_2 and s_3 as $\underbrace{\omega^*(s_2, a)}_{x} \neq \underbrace{\omega^*(s_3, a)}_{y}$, and

– s_2 and s_4 as $\underbrace{\omega^*(s_2, a)}_{x} \neq \underbrace{\omega^*(s_4, a)}_{y}$,

(3) b is characterizing sequence of:

– s_1 and s_3 as $\underbrace{\omega^*(s_1, b)}_{y} \neq \underbrace{\omega^*(s_3, b)}_{x}$,

– s_2 and s_3 as $\underbrace{\omega^*(s_2, b)}_{y} \neq \underbrace{\omega^*(s_3, b)}_{x}$, and

– s_3 and s_4 as $\underbrace{\omega^*(s_3, b)}_{x} \neq \underbrace{\omega^*(s_4, b)}_{y}$.

Here we do not explain how to generate a test sequences using the computed above characterizing sequences as this procedure is the same as for the UIO method which was explained in Example 3.6 (*see* page 29). □

The W-method was first introduced by M.P. Vasilevskii in [Vasilevskii, 1973]. Then, it has been elaborated and adapted to conformance testing problem by T.S. Chow in [Chow, 1978].

3.1.4 Conclusion

The test generation methods that were described in this section are based on the well-established theory of finite state machines. Another advantage of these methods is that they generate sound and exhaustive test cases which are able to detect output and transfer faults in the system under test. However, these methods are not always applicable to real reactive systems (*e.g.* protocols or smart cards) due to (1) the size of these systems, and (2) the hypotheses that must be made on a specification and an implementation of this system in order to be able to apply one of these test generation methods. Moreover, most of them are very expensive on time and memory. These limitations have activated the research and development of methods for test generation based on an alternative model called transition systems.

3.2 Test Generation Based on Transition Systems

A *Transition System* (LTS) [Keller, 1976] is an abstract model based on two primitive notions of *state* and *transition*. Since their appearance, transition systems have been used as an underlying model for protocols in the fields of formal

verification, model checking, and testing. In this section we consider two extensions of transition systems called *Labeled Transition Systems* (LTS), where each transition is labeled with an observable or internal action, and *Input-Output Transition Systems* (IOLTS), where we make a distinction between observable (output), controllable (input) and internal actions. For both classes of transition systems, we give a brief overview of the testing relations. Finally, we describe several approaches for test generation based on these models, and the tools using there approaches.

In this section we have used the materials from the introduction parts of the theses [Heerink, 1998] and [Nielsen, 2000] written by A.W. Heerink and B. Nielsen respectively, the lecture notes [Tretmans, 2002] of J. Tretmans, and the habilitation document [Jéron, 2004] of T. Jéron.

3.2.1 Testing based on Labeled Transition Systems

In this section we introduce the formalism of Labeled Transition Systems (LTS), which is used to model the observable behaviors of reactive systems. We also describe the semantics of LTS, and discuses several testing relations defined on LTS.

3.2.1.1 Model: Labeled Transition Systems

Any labeled transition system ([Milner, 1980], [Brookes and Roscoe, 1985], [Brinksma, 1988]) consists of nodes and transitions between nodes that are labeled with observable or internal actions. Formally:

Definition 3.3 (LTS) A *LTS* is a tuple $\langle S, s^0, (\Sigma \cup \{\tau\}), \rightarrow \rangle$, where

- S is a countable, non-empty *set of states*,
- $s^0 \in S$ is the *initial state*,
- Σ is a countable *alphabet of observable actions*,
- τ is used to denote any *unobservable internal action*, and
- $\rightarrow \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ is the *transition relation*.

□

We illustrate the definition of LTS with an example.

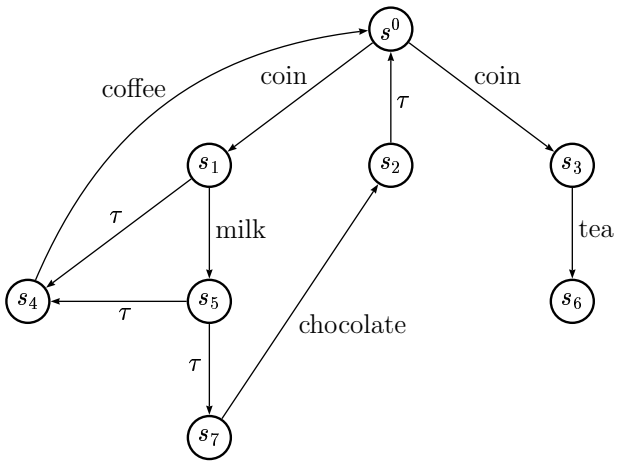


Figure 3.9: A coffee machine modeled by LTS $\mathcal{S}$.

Example 3.8 Consider the labeled transition system $\mathcal{S}$ shown on Figure 3.9. It consists of eight states $s^0, s_1, \ldots, s_7$, where s^0 is the initial state of $\mathcal{S}$, and ten transitions. Each transition of $\mathcal{S}$ is labeled with either an observable (*coin*, *milk*, *coffee*, *tea*, *chocolate*) or unobservable internal (τ) action.

This labeled transition system $\mathcal{S}$ represents a coffee machine delivering coffee, chocolate, or tea to the user. The behavior of this coffee machine can be described as follows: after receiving a coin from the user, the machine may move either to the state s_1 or to the state s_3. If the machine decided to go to the state s_3, then it delivers tea to the user, and ends its work in the state s_6. If the machine is in the state s_1 after receiving the coin, then there exist two possible scenarios: (1) the machine decides to move to the state s_4 by executing its internal action τ, to deliver coffee to the user, and to move to its initial state s^0; or (2) the user has a possibility to add milk to his/her beverage, after which the machine moves to the state s_5, where it executes one of its internal action, delivers coffee of chocolate to the user, and moves to the initial state s^0. □

Before giving an overview about testing relations defined on LTS, we introduce several notations for LTS. The same notations will be used later for Input-Output Labeled Transition Systems.

Definition 3.4 Let $\mathcal{M} = \langle S, s^0, (\Sigma \cup \{\tau\}), \rightarrow \rangle$ be an LTS. Let also:
- $S' \subseteq S$ be a subset of states $\mathcal{M}$,
- $\Sigma' \subseteq (\Sigma \cup \{\tau\})$ be a subset of actions of $\mathcal{M}$,
- $s, s' \in S$ be two states of $\mathcal{M}$,
- $a_{(i)} \in \Sigma$ be an observable action of $\mathcal{M}$,

- $\mu_{(i)} \in (\Sigma \cup \{\tau\})$ be either an observable or internal action of $\mathcal{M}$,
- $\sigma = a_1 \ldots a_n \in (\Sigma)^*$ be a sequence of observable actions,
- $\eta = \mu_1 \ldots \mu_k \in (\Sigma \cup \{\tau\})^*$ be a sequence of observable and internal actions, and
- $\varepsilon = \tau \ldots \tau \in (\{\tau\})^*$ be a sequence of internal actions.

Then,

(1) $s \xrightarrow{\mu} s' \quad \triangleq \quad \langle s, \mu, s' \rangle \in \rightarrow,$

(1.a) $s \xrightarrow{\mu} \quad \triangleq \quad \exists\, s' \in S \,.\, [\, s \xrightarrow{\mu} s' \,],$

(2) $s \xrightarrow{\eta} s' \quad \triangleq \quad s \xrightarrow{\mu_1 \ldots \mu_k} s'$
$\triangleq \quad \exists\, s^0, \ldots, s_k \in S \,.\, [\, s = s^0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_k} s_k = s' \,],$

(2.a) $s \xrightarrow{\eta} \quad \triangleq \quad \exists\, s' \in S \,.\, [\, s \xrightarrow{\eta} s' \,],$

(3) $s \stackrel{\varepsilon}{\Rightarrow} s' \quad \triangleq \quad [\, (s = s') \;\vee\; (s \xrightarrow{\tau \ldots \tau} s') \,],$

(4) $s \stackrel{a}{\Rightarrow} s' \quad \triangleq \quad \exists\, s_1, s_2 \in S \,.\, [\, s \stackrel{\varepsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \stackrel{\varepsilon}{\Rightarrow} s' \,],$

(4.a) $s \stackrel{a}{\not\Rightarrow} \quad \triangleq \quad \nexists\, s' \in S \,.\, [\, s \stackrel{a}{\Rightarrow} s' \,],$

(5) $s \stackrel{\sigma}{\Rightarrow} s' \quad \triangleq \quad s \stackrel{a_1 \ldots a_k}{\Longrightarrow} s'$
$\triangleq \quad \exists\, s^0, \ldots, s_n \in S \,.\, [\, s = s^0 \stackrel{a_1}{\Rightarrow} s_1 \stackrel{a_2}{\Rightarrow} \ldots \stackrel{a_n}{\Rightarrow} s_n = s' \,],$

(5.a) $s \stackrel{\sigma}{\Rightarrow} \quad \triangleq \quad \exists\, s' \in S \,.\, [\, s \stackrel{\sigma}{\Rightarrow} s' \,],$

(6) $\mathit{Traces}(\mathcal{M}) \quad \triangleq \quad \{\sigma \in (\Sigma)^* \mid s^0 \stackrel{\sigma}{\Rightarrow}\},$

(7) $(s \text{ after } \sigma) \quad \triangleq \quad \{s' \in S \mid s \stackrel{\sigma}{\Rightarrow} s'\},$

(7.a) $(\mathcal{M} \text{ after } \sigma) \quad \triangleq \quad (s^0 \text{ after } \sigma),$

(8) $\mathcal{M}$ is deterministic *iff* $\mathcal{M}$ does not have any internal actions and
$\forall\, s \in S,\ \sigma \in (\Sigma)^* \,.\, [\, \mathit{card}(s \text{ after } \sigma) \leq 1 \,],$

(9) $\mathit{DTraces}(\mathcal{M}) \quad \triangleq \quad \{\, \sigma \in \mathit{Traces}(\mathcal{M}) \mid \exists\, s \in (\mathcal{M} \text{ after } \sigma), \forall\, a \in \Sigma \,.$
$[\, s \stackrel{a}{\not\Rightarrow} \,] \,\},$

(10) $\mathsf{post}_{\Sigma'}(S') \quad \triangleq \quad \{\, s \in S \mid \exists\, a \in \Sigma',\ s' \in S' \,.\, [\, s' \xrightarrow{a} s \,] \,\},$

(11) $\mathsf{pre}_{\Sigma'}(S') \quad \triangleq \quad \{\, s \in S \mid \exists\, a \in \Sigma',\ s' \in S' \,.\, [\, s \xrightarrow{a} s' \,] \,\},$

(12) $\mathit{Reach}(S') \quad \triangleq \quad \{\, s \in S \mid \exists\, \sigma \in (\Sigma \cup \{\tau\})^*,\ s' \in S' \,.\, [\, s' \xrightarrow{\sigma} s \,] \,\},$

(13) $\mathit{CoReach}(S') \quad \triangleq \quad \{\, s \in S \mid \exists\, \sigma \in (\Sigma \cup \{\tau\})^*,\ s' \in S' \,.\, [\, s \xrightarrow{\sigma} s' \,] \,\}.$

□

The example below illustrates the notations given above.

Example 3.9 Let us consider a coffee machine modeled by the LTS depicted on Figure 3.9 (*see* page 35), whose semantics was informally explained in Example 3.8 (*see* page 34). Then, it is not hard to check that:

(1) $s^0 \xrightarrow{coin} s_1, \quad s_7 \xrightarrow{chocolate} s_2, \quad s_5 \xrightarrow{\tau} s_4,$

(2) $s_1 \xrightarrow{milk.\tau.coffee} s^0,$

(3) $s_1 \stackrel{\varepsilon}{\Rightarrow} s_4, \quad s_2 \stackrel{\varepsilon}{\Rightarrow} s^0,$

(4) $s_5 \stackrel{chocolate}{\Longrightarrow} s^0, \quad s_5 \stackrel{tea}{\not\Rightarrow},$

(5) $s^0 \stackrel{coin.milk.coffee}{\Longrightarrow} s^0,$

(6) The LTS $\mathcal{S}$ has an infinite number of traces, *i.e.* $Traces(\mathcal{S}) = \{\varepsilon;\ coin;\ coin.tea;\ coin.coffee;\ coin.milk.coffee; \ldots\}$

(7) $(s_5 \text{ after } \varepsilon) = \{s_4, s_5, s_7\}$, $(\mathcal{S} \text{ after } coin) = \{s_1, s_3, s_4\}$, $(\mathcal{S} \text{ after } coin.chocolate) = \emptyset$,

(8) The LTS $\mathcal{S}$ is non-deterministic as $card(s^0 \text{ after } coin) = 3$,

(9) The traces *coin.tea*, *coin.coffee.coin.tea* and *coin.milk.chocolate.coin.tea* lead to deadlocks, *i.e.* the LTS $\mathcal{S}$ cannot execute any observable action after them. Thus, all these traces belong to $DTraces(\mathcal{S})$.

(10) For the set of state $S' = \{s^0, s_4, s_7\}$ and set of actions $\Sigma' = \{coffee, tea, chocolate\}$, the set of successors of S' by the actions from Σ' is: $\mathsf{post}_{\Sigma'}(S') = \{s^0, s_2\}$. Indeed, the state s^0 is the successor of s_4 by *coffee*, and the state s_2 is the successor of s_7 by *chocolate*.

(11) For the set of state $S' = \{s_3, s_4, s_7\}$ and set of actions $\Sigma' = \{coin, milk, \tau\}$, the set of predecessors of S' by the actions from Σ' is: $\mathsf{pre}_{\Sigma'}(S') = \{s^0, s_1, s_5\}$. Indeed, the state s^0 is the predecessors of s_3 by *coin*, and the states s_1 and s_5 are the predecessors of s_4 and s_7 by τ.

(12) The set of states reachable from $S' = \{s_3, s_6\}$ is $Reach(S') = \{s_3, s_6\}$. Indeed, from the state s_6 we can reach only s_6 by the empty sequence of observable actions, and from s_3 we can reach s_6 by the sequence and s_3 by the empty sequence of observable actions.

Using Figure 3.9 (*see* page 35) the reader can easily check that the set of states reachable from $S' = \{s_5\}$ contains all states of the LTS $\mathcal{S}$, *i.e.* $Reach(S') = \{s^0, s_1, \ldots, s_7\}$.

(13) The set of states reachable from $S' = \{s_3, s_6\}$ is $CoReach(S') = \{s^0, s_1, \ldots, s_7\}$. From Figure 3.9 (*see* page 35) it is easy to see that from all state of the LTS $\mathcal{S}$ we can reach the states s_3 and s_6. For instance, from the state s_5 we may reach s_6 by, for example, the sequence *chocolate.coin.tea*. Thus, s_5 is coreachable from s_6.

The reader also can check that $CoReach(\{s_5\}) = \{s^0, s_1, s_2, s_4, s_5, s_7\}$.

□

Finally, we introduce the operation of parallel composition between two LTS which will be used in order to model the communication between two different systems.

Definition 3.5 (Parallel Composition ||) Let

(1) $\mathcal{M}_1 = \langle S_{\mathcal{M}_1}, s^0_{\mathcal{M}_1}, (\Sigma_{\mathcal{M}_1} \cup \{\tau\}), \rightarrow_{\mathcal{M}_1} \rangle$, $\mathcal{M}_2 = \langle S_{\mathcal{M}_2}, s^0_{\mathcal{M}_2}, (\Sigma_{\mathcal{M}_2} \cup \{\tau\}), \rightarrow_{\mathcal{M}_2} \rangle$ be two arbitrary LTS,

(2) $s_1, s'_1 \in S_{\mathcal{M}_1}$ be two states of $\mathcal{M}_1$,

(3) $s_2, s'_2 \in S_{\mathcal{M}_2}$ be two states of $\mathcal{M}_2$,

(4) $a \in (\Sigma_{\mathcal{M}_1} \cap \Sigma_{\mathcal{M}_2})$ be a *common* observable action of $\mathcal{M}_1$ and $\mathcal{M}_2$, and

(5) τ is the internal action of $\mathcal{M}_1$ and $\mathcal{M}_2$.

Then, the *parallel composition* between $\mathcal{M}_1$ and $\mathcal{M}_2$ with synchronization on *common observable actions* is the LTS:

$$\mathcal{M}_1 \,||\, \mathcal{M}_2 = \langle (S_{\mathcal{M}_1} \times S_{\mathcal{M}_2}), \langle s^0_{\mathcal{M}_1}, s^0_{\mathcal{M}_2} \rangle, (\Sigma_{\mathcal{M}_1} \cap \Sigma_{\mathcal{M}_2}) \cup \{\tau\}, \rightarrow_{(\mathcal{M}_1 \,||\, \mathcal{M}_2)} \rangle$$

where the transition relation $\rightarrow_{(\mathcal{M}_1 \,||\, \mathcal{M}_2)}$ is the smallest relation satisfying the three following inference rules:

$$\frac{s_1 \xrightarrow{\tau}_{\mathcal{M}_1} s'_1, \quad s_2 \in S_{\mathcal{M}_2}}{\langle s_1, s_2 \rangle \xrightarrow{\tau}_{(\mathcal{M}_1 \,||\, \mathcal{M}_2)} \langle s'_1, s_2 \rangle} \tag{3.2}$$

$$\frac{s_2 \xrightarrow{\tau}_{\mathcal{M}_2} s'_2, \quad s_1 \in S_{\mathcal{M}_1}}{\langle s_1, s_2 \rangle \xrightarrow{\tau}_{(\mathcal{M}_1 \,||\, \mathcal{M}_2)} \langle s_1, s'_2 \rangle} \tag{3.3}$$

$$\frac{s_1 \xrightarrow{a}_{\mathcal{M}_1} s'_1, \quad s_2 \xrightarrow{a}_{\mathcal{M}_2} s'_2}{\langle s_1, s_2 \rangle \xrightarrow{a}_{(\mathcal{M}_1 \,||\, \mathcal{M}_2)} \langle s'_1, s'_2 \rangle} \tag{3.4}$$

□

3.2.1.2 Testing Relations for LTS

The LTS formalism offers an easy way to describe the observable behaviors of reactive system. However, it also allows a possibility to design two different LTS describing exactly the same observable behaviors. This disadvantage stimulated a study of the equivalence and preorder relations based on the notion of observable behaviors. Many works have been done on establishing the relations between LTS. For instance, observation equivalence [Milner, 1980], failure equivalence [Hoare, 1985] and testing equivalence [de Nicola and Hennessy, 1984] were defined. An overview about these works can be found in [van Glabbeek, 2001]. Nevertheless, in this section we are more interested in *preorder* relations (*e.g.* testing preorder [de Nicola and Hennessy, 1984], refusal preorder [Phillips, 1987], conf relation [Brinksma, 1988]) which are used to express the notion of one system implements another. These relations can be used as the implementation relation defined in Section 2.1.3 of Chapter 2 (*see* page 15).

General Hypotheses. For the rest of this subsection we consider a class of LTS over an alphabet Σ, which we denoted as $\mathcal{LTS}(\Sigma)$. Moreover, by analogy with the work of J. Tretmans [Tretmans, 1996b], we restrict this class to *strongly convergent* LTS, *i.e.* LTS that do not contain infinite sequences of internal actions.

Testing Preorder [de Nicola and Hennessy, 1984], [de Nicola, 1987] was defined by R. de Nicola and M. Hennessy.

Intuitively, an implementation that can be modeled by $iut \in \mathcal{LTS}(\Sigma)$ is in the *testing preorder relation* with a specification $Spec \in \mathcal{LTS}(\Sigma)$ (*i.e.* $iut \leq_{te} Spec$) if for a test case $TC \in \mathcal{LTS}(\Sigma)$, the observations (*traces* and *deadlocks*) obtained during interaction of TC with the implementation iut are included into the possible observations obtained during the interaction of the same test case TC with the specification $Spec$.

The detailed description of testing preorder can be found in [Heerink, 1998], [Tretmans, 2002].

Refusal Preorder. I. Phillips in his thesis [Phillips, 1987] introduced the refusal testing theory that is based on a stronger preorder relation than testing preorder. He called this relation refusal preorder. The main difference between testing and refusal preorder is that the latter is able to detect the absence of all possible actions (*i.e.* deadlocks) and not to block on them. This is done by introducing a new *observable* action denoted δ. The δ-action occurs in the situations where an implementation under test in not able to interact with a test case.

Intuitively, an implementation that can be modeled by $iut \in \mathcal{LTS}(\Sigma)$ is in the *refusal preorder relation* with a specification $Spec \in \mathcal{LTS}(\Sigma)$ (*i.e.* $iut \leq_{rf} Spec$) if

for a test case $TC \in \mathcal{LTS}(\Sigma)$ (which may include the δ-*action*) the observations (*traces* and *deadlocks*) that can be obtained when TC communicates with the implementation *iut* can also be obtained during the communication between TC and the specification *Spec*.

The formal definition of refusal preorder can be found in the thesis of I. Phillips. The reader can also be addressed to [Heerink, 1998], [Nielsen, 2000] [Tretmans, 2002] that give good summaries of different works on testing and refusal preorders.

Conf Relation. In the works on the equivalence and preorder relations the researchers were interested in establishing correctness criteria between an implementation under test that can be modeled by $iut \in \mathcal{LTS}(\Sigma)$ and its formal specification $Spec \in \mathcal{LTS}(\Sigma)$ by (1) providing a set of experiments (or, in other words, test cases), and (2) analyzing the executions of these test cases on *iut* and *Spec*.

However, the problem of *test generation* is different from that of establishing correctness criteria. It can be formulated as follows: for a given correctness criterion and a given specification *Spec*, we have to generate a test suite $TS \subseteq \mathcal{LTS}(\Sigma)$ that is able to distinguish correct and incorrect implementations based on the observations.

The test generation problem was initially studied by E. Brinksma [Brinksma, 1988] at the end of 80s. In this work, he presented a method that, from a given specification derives a set of test cases which distinguish between correct and incorrect implementations with respect to the conf relation. The conf relation (which is neither equivalence nor preorder relation) is a modification of the previously described preorder relation, where (1) interactions between a test case and an implementation (or a specification) is modeled by the parallel composition (*see* Definition 3.5, page 38), and (2) only traces of the specification are considered.

Definition 3.6 (conf Relation) Let $Spec \in \mathcal{LTS}(\Sigma)$ be a specification and $iut \in \mathcal{LTS}(\Sigma)$ be a LTS which models a given implementation under test. Then, the conf *relation* is defined as follows:

$$(iut \ \mathsf{conf} \ Spec) \quad \triangleq \tag{3.5}$$
$$\forall\ TC \in \mathcal{LTS}(\Sigma)\ .\ [\ ((Traces(TC \parallel iut) \cap Traces(Spec)) \subseteq Traces(TC \parallel Spec))\ \wedge$$
$$((DTraces(TC \parallel iut) \cap Traces(Spec)) \subseteq DTraces(TC \parallel Spec))\ ]$$

□

The conf relation is well adapted for testing, as it restricts all possible observations to the traces of a specification. It tests only whether a given implementation does

what it should do. This simplifies the testing task, as we do not have to take care about unspecified behaviors.

In the paper [Brinksma, 1988], E. Brinksma also introduces a notion of *canonical tester* that can be automatically derived from a given specification. Intuitively, a canonical tester is an LTS that preserves the traces of a specification and that is able to decide whether an implementation is conf-related to the specification. There exist several works on automatic generation of canonical testers, for example, *see* [Brinksma, 1988], [Pitt and Freestone, 1990]. Detailed information about canonical testers and conf relation can be found in the thesis of J. Tretmans [Tretmans, 1992].

3.2.2 Testing based on Input-Output (Labeled) Transition Systems

The testing techniques based on LTS (*see* Section 3.2.1, page 34) are very often too theoretical to be used in practice. Indeed, it is assumed that a set of test cases exists. Moreover, the LTS model does not allow to describe asymmetric communication between tester and implementation under test, *i.e.* there is no difference between observable and controllable actions. However, this difference has a fundamental role in testing. Indeed, in the testing practice the tester *chooses* an input action a, transmits it to the implementation under test, and *observes* output actions produced by the implementation after a. Therefore, the model used in testing must be more detailed. At least, it has to make a difference between input and output actions. Many works had been done in this direction, for example, Input-Output Automata (IOA) [Lynch and Tuttle, 1989] proposed by N. Lynch and M. Tuttle, Input-Output State Machines (IOSM) [Phalippou, 1994] of M. Phalippou, Input-Output (Labeled) Transition Systems (IO(L)TS) [Tretmans, 1995], [Jéron, 2004].

In this section we introduce the IO(L)TS model. Then we describe two important testing relations, namely, ioconf and ioco, used in conformance testing. Finally, we describe several methodologies of test generation based on IO(L)TS and ioconf or ioco relations.

3.2.2.1 Model: Input-Output Labeled Transition Systems

The Input-Output Labeled Transition Systems (IOLTS) is a variant of LTS, where the alphabet of observable actions is separated into two disjoint alphabets of input and output action. Formally:

Definition 3.7 (IOLTS) An *IOLTS* is a tuple $\langle S, s^0, (\Sigma \cup \{\tau\}), \rightarrow \rangle$, where

– S is a countable, non-empty *set of states*,

- $s^0 \in S$ is the *initial state*,
- $\Sigma = \Sigma^? \cup \Sigma^!$ is a countable *alphabet of actions* which consists of two disjoint alphabets of *input* $\Sigma^?$ and *output* $\Sigma^!$,
- $\rightarrow \ \subseteq \ S \times (\Sigma \cup \{\tau\}) \times S$ is the *transition relation.*

□

Figure 3.9 (*see* page 35) can be used as an example of IOLTS, where *coin*, *milk* are input action, *coffee*, *chocolate*, *tea* are output actions, and τ is an internal action (in the rest of this section in all figures showing IOLTS, input actions will be marked with "?", and output actions with "!"). The description of this IOLTS was given in Example 3.8 (*see* page 34).

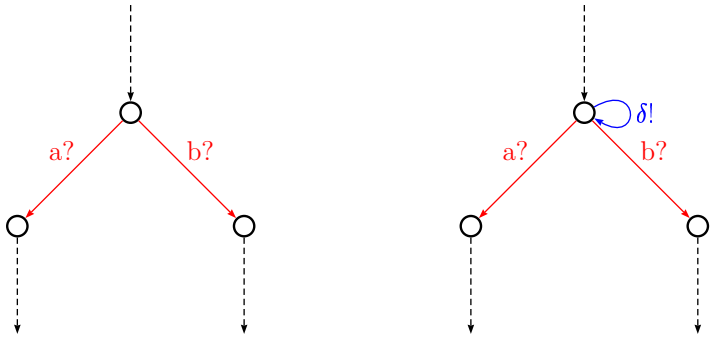
It is important emphasize that J. Tretmans in his works (*e.g.* [Tretmans, 1996b]) restricts the class of IOLTS to *weakly input-enabled* IOLTS, which are also called *input-compete* IOLTS and defined below. This restriction is omitted in this thesis. However, each time when we need to work with such IOLTS, we will explicitly mention about it.

Definition 3.8 (Input-Complete IOLTS) An IOLTS $\mathcal{M} = \langle S_\mathcal{M}, s^0_\mathcal{M}, (\Sigma_\mathcal{M} \cup \{\tau\}), \rightarrow_\mathcal{M} \rangle$ is *input-complete* if in each state of $\mathcal{M}$ all inputs actions of $\mathcal{M}$ are enabled (after execution of possible internal actions), *i.e.* $\forall\ s \in S_\mathcal{M},\ a \in \Sigma^?_\mathcal{M}\ .\ [s \stackrel{a}{\Rightarrow}]$. □

All definitions given in Section 3.2.1.1 (*see* Definition 3.4, page 35) can be applied to input-output labeled transition systems as IOLTS are LTS as well. Moreover, the operation of parallel composition defined on page 38 and used to model the interaction between an implementation under test and a test case, is exactly the same as for LTS. However, it is important to notice that output actions of the implementation must interact with input actions of the test case and vice versa.

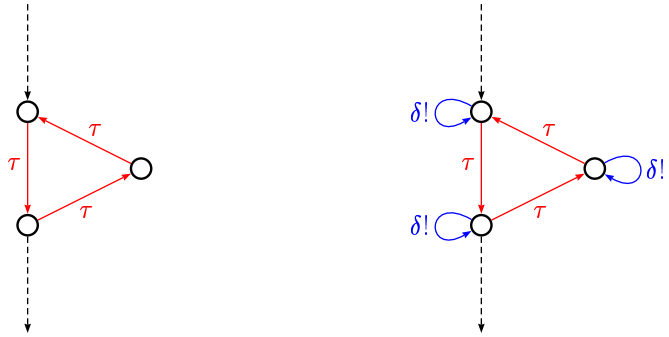
At the end of this subsection, we discuss the blocking problem as is appears in conformance testing. Let us consider an IOLTS $\mathcal{M} = \langle S_\mathcal{M}, s^0_\mathcal{M}, (\Sigma_\mathcal{M} \cup \{\tau\}), \rightarrow_\mathcal{M} \rangle$ interacting with its environment. During this interaction the system can either normally proceed its computations, or can be blocked. By analogy with the works of T. Jéron [Jard and Jéron, 2002], [Jéron, 2004], we distinguish three kinds of *blocking*:

Outputlock: the system is in a state (*see* the left-hand side of Figure 3.10(a), page 43), where it is only waiting for input actions from the environment. Formally, a state $s \in S_\mathcal{M}$ is an outputlock if $\{\mu \in (\Sigma_\mathcal{M} \cup \{\tau\}) \mid s \stackrel{\mu}{\rightarrow}\} \subseteq \Sigma^?_\mathcal{M}$.



(a) Outputlock

(b) Deadlock



(c) Livelock

Figure 3.10: Three possible blockings in IOLTS.

Deadlock: the system cannot perform any action (*see* the left-hand side of Figure 3.10(b), page 43). Formally, a state $s \in S_{\mathcal{M}}$ is a deadlock if $\{\mu \in (\Sigma_{\mathcal{M}} \cup \{\tau\}) \mid s \xrightarrow{\mu}\} = \emptyset$.

Livelock: the system is in a cycle of internal actions (*see* the left-hand side of Figure 3.10(c), page 43). Formally, a state $s \in S_{\mathcal{M}}$ is a livelock is $\exists\, \tau \ldots \tau \,.\, [s \xrightarrow{\tau \ldots \tau} s]$.

It is important to notice that a blocking of the given specification is not necessary an error. It is evident for outputlocks, but it is also true for livelocks (that can be obtained, for instance, as a consequence of the parallel composition between several systems) and, moreover, for deadlocks. Hence, a test case applied to an implementation under test (*iut*) must distinguish between specified blockings of *iut* (*i.e.* ones that exist in the specification), and non-specified blockings of *iut* (*i.e.* ones that produced *only* by the implementation). Therefore, all possible blockings of the specification have to be detected and marked with the special output action δ (*see* Figure 3.10, page 43).

Before giving the definition of an IOLTS in which all blockings are marked, we denote the set of all quiescent states (*i.e.* outputlocks, deadlocks and livelocks) of an IOLTS $\mathcal{M}$ as $Quiescent(\mathcal{M})$.

Definition 3.9 (Suspension IOLTS) For an IOLTS $\mathcal{M} = \langle S_{\mathcal{M}}, s^0_{\mathcal{M}}, (\Sigma_{\mathcal{M}} \cup \{\tau\}), \rightarrow_{\mathcal{M}} \rangle$, we define a suspension IOLTS $\Delta(\mathcal{M}) = \langle S_{\mathcal{M}}, s^0_{\mathcal{M}}, (\Sigma_{\Delta(\mathcal{M})} \cup \{\tau\}), \rightarrow_{\Delta(\mathcal{M})} \rangle$, where

(1) the alphabet of action is augmented with the special *output* action δ, *i.e.* $\Sigma_{\Delta(\mathcal{M})} = \Sigma_{\mathcal{M}} \cup \{\delta\}$, and

(2) the transition relation of $\Delta(\mathcal{M})$ is constructed as follows:

$$\rightarrow_{\Delta(\mathcal{M})} \;\triangleq\; \rightarrow_{\mathcal{M}} \;\cup\; \{s \xrightarrow{\delta} s \mid s \in Quiescent(\mathcal{M})\}$$

□

The *suspension traces* of an IOLTS $\mathcal{M}$ (denoted $STraces(\mathcal{M})$) are the traces of its suspension IOLTS $\Delta(\mathcal{M})$, *i.e.* $STraces(\mathcal{M}) = Traces(\Delta(\mathcal{M}))$.

3.2.2.2 Testing Relations for IOLTS

Jan Tretmans and his colleagues from the University of Twente, Netherlands have adapted the testing preorder $\leq_{te}$ and refusal preorder $\leq_{rf}$, which were informally introduced on page 39, for the IOLTS model, and obtained *input-output testing preorder* $\leq_{iot}$ and *input-output refusal preorder* $\leq_{ior}$ (*see* [Tretmans, 1996b],

[Tretmans, 2002]). Analogously to $\leq_{te}$ and $\leq_{rf}$ the new defined preorder relations (1) allow implementations under test to do what is specified, and (2) does not allow them to do more than what is specified. This requirement is too strong for conformance testing. Indeed, in conformance testing we are only interested in checking whether the implementation respects a specification, *i.e.* the question that is posed here is: does the implementation do what it should do. Using this argumentation Tretmans et al. introduced two new testing relations which they called ioconf [Tretmans, 1995] and ioco [Tretmans, 1996a] that are well adapted for conformance testing and automatic test derivation. The original versions of the ioconf and ioco relations treat the problem of specified and unspecified outputlocks and deadlocks (*see* page 42). T. Jéron and his colleagues have extended the conformance relations proposed by J. Tretmans (*i.e.* ioconf and ioco) for livelocks detection (*see* page 42).

In this subsection we give the formal definitions of the ioconf and ioco relations and illustrate them with examples.

General Hypotheses. For the rest of this subsection we consider:

(1) a specification whose behaviors are modeled by a *convergent* IOLTS $Spec = \langle S_{Spec}, s^0_{Spec}, (\Sigma_{Spec} \cup \{\tau\}), \rightarrow_{Spec} \rangle$, where $\Sigma_{Spec} = \Sigma^?_{Spec} \cup \Sigma^!_{Spec}$.

We remind that an IOLTS is *convergent* if it does not contain an infinite sequence of internal actions which goes through infinite number of different states.

(2) an implementation under test whose possible behaviors are modeled by an *input-complete* IOLTS $iut = \langle S_{iut}, s^0_{iut}, (\Sigma_{iut} \cup \{\tau\}), \rightarrow_{iut} \rangle$ with alphabets of input $\Sigma^?_{iut} \subseteq \Sigma^?_{Spec}$ and output $\Sigma^!_{iut} \subseteq \Sigma^!_{Spec}$ actions. The definition of an *input-complete* IOLTS can be found on page 42.

Before describing the ioconf and ioco relations, we give an intermediate definition of a set of output actions that can be generated by an IOLTS when it is in some state. Formally, for an IOLTS $\mathcal{M} = \langle S_{\mathcal{M}}, s^0_{\mathcal{M}}, (\Sigma_{\mathcal{M}} \cup \{\tau\}), \rightarrow_{\mathcal{M}} \rangle$ with alphabet $\Sigma_{\mathcal{M}} = \Sigma^?_{\mathcal{M}} \cup \Sigma^!_{\mathcal{M}}$, and a subset of states $S'_{\mathcal{M}} \subseteq S_{\mathcal{M}}$, we define:

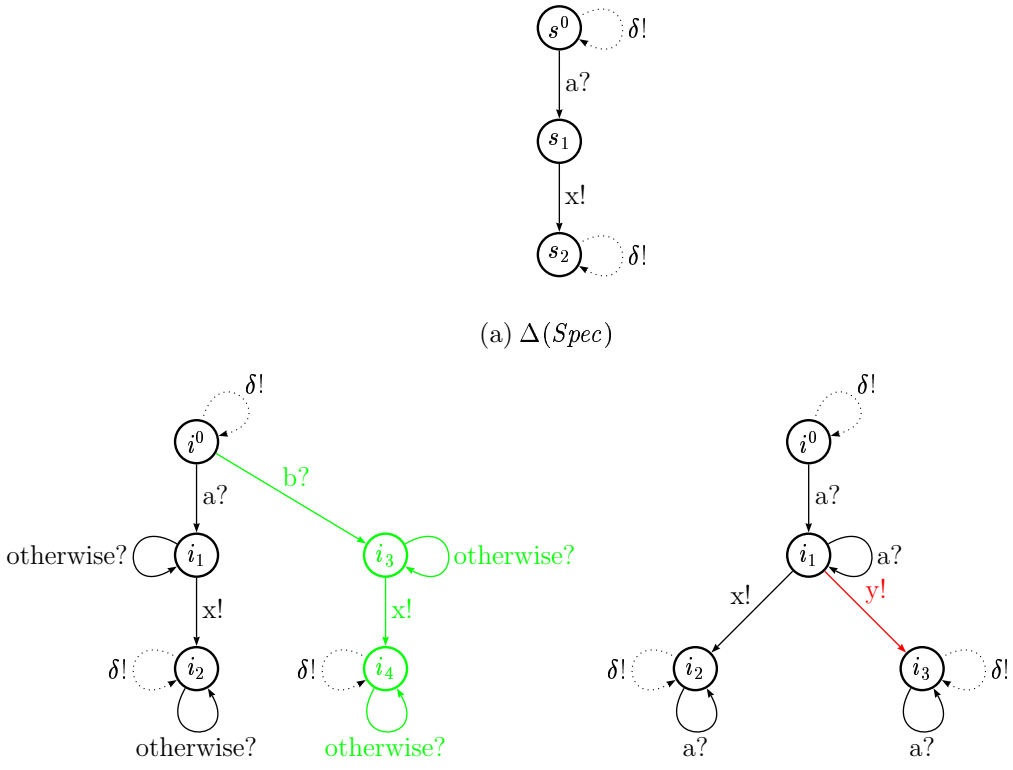
$$Out(S'_{\mathcal{M}}) \quad \triangleq \quad \{a \in \Sigma^!_{\mathcal{M}} \mid \exists\ s' \in S'_{\mathcal{M}}, s \in S_{\mathcal{M}}\ .\ [\ s' \xrightarrow{a} s\ ]\} \tag{3.6}$$

Ioconf Relation. Intuitively, the implementation under test *iut* is conformant to the specification *Spec* (with the special output action δ marking all possible blockings in *Spec*) according to ioconf-relation if for each trace of *Spec*, the implementation produces only outputs and blockings which are allowed by the specification. Formally:

Definition 3.10 (ioconf Relation) For a specification *Spec* and an implementation under test *iut* formally described in paragraph **General Hypotheses** (*see* page 45), the ioconf *relation* is defined as follows:

$$(iut \text{ ioconf } Spec) \quad \triangleq \quad \forall\ \sigma \in Traces(Spec)\ .\ [\ Out(\Delta(iut) \text{ after } \sigma) \subseteq Out(\Delta(Spec) \text{ after } \sigma)\ ] \tag{3.7}$$

□



(a) $\Delta(Spec)$

(b) $\Delta(iut_1)$

(c) $\Delta(iut_2)$

Figure 3.11: An example illustrating ioconf relation.

Example 3.10 To illustrate the definition of the ioconf relation, we consider a suspension specification $\Delta(Spec)$ shown on Figure 3.11(a) (*see* page 46) and two suspension implementations under test $\Delta(iut_1)$ and $\Delta(iut_2)$ depicted in Figures 3.11(b) and 3.11(c) respectively (*see* page 46). Notice that here and later,

we use the abbreviation **otherwise?** to indicate all possible inputs of an implementation under test.

It is not hard to see that the specification *Spec* (which can be obtained from $\Delta(Spec)$ by removing all loops on δ) only *partially* specifies the implementation iut_1 (which can be obtained from $\Delta(iut_1)$ by removing all dotted edges from Figure 3.11(b)), as iut_1 in its initial state can execute the unspecified input action b (*see* the green edges on Figure 3.11(b)). However, this implementation is *conformant* to the specification, *i.e.* (iut_1 ioconf *Spec*). Indeed, after each trace of *Spec* (which are ε, a and $a.x$), it produces the same outputs as the specification. For instance, $Out(\Delta(iut_1)\ \mathsf{after}\ a) = \{x\} \subseteq Out(\Delta(Spec)\ \mathsf{after}\ a) = \{x\}$ or $Out(\Delta(iut_1)\ \mathsf{after}\ a.x) = \{\delta\} \subseteq Out(\Delta(Spec)\ \mathsf{after}\ a.x) = \{\delta\}$.

Next, we consider the second implementation under test iut_2 obtained from $\Delta(iut_2)$ by removing all loops on output action δ. From Figure 3.11(c), page 46 it is easy to see that after input action a, the implementation iut_2 performs unspecified output action y (*see* the red edge on Figure 3.11(c)). Formally, $Out(\Delta(iut_2)\ \mathsf{after}\ a) = \{x, y\} \not\subseteq Out(\Delta(Spec)\ \mathsf{after}\ a) = \{x\}$. Therefore, this implementation is *not conformant* to its specification, *i.e.* $\neg(iut_2\ \mathsf{ioconf}\ Spec)$.

□

Ioco Relation. In this paragraph we introduce the ioco relation which treats specified and unspecified blockings. Intuitively, the implementation under test *iut* is ioco-conformant to the specification *Spec* if for each *suspension* trace σ of *Spec*, the outputs (including the δ-action) produced by suspension *iut* after this trace are the outputs that can be produced by suspension *Spec* after σ. Formally:

Definition 3.11 (ioco Relation) For a specification *Spec* and an implementation under test *iut* formally described in paragraph **General Hypotheses** (*see* page 45), the ioco *relation* is defined as follows:

$$(iut\ \mathsf{ioco}\ Spec) \quad \triangleq \quad \forall\ \sigma \in STraces(Spec)\ .\ [\ Out(\Delta(iut)\ \mathsf{after}\ \sigma) \subseteq Out(\Delta(Spec)\ \mathsf{after}\ \sigma)\ ] \tag{3.8}$$

□

Then, we illustrate the definition given above with an example. This example had been taken from the habilitation document [Jéron, 2004] of T. Jéron.

Example 3.11 Consider Figure 3.12 (*see* page 49) which presents the suspension IOLTS of a specification *Spec* and four suspension IOLTS of implementations under test iut_1–iut_4. Then,

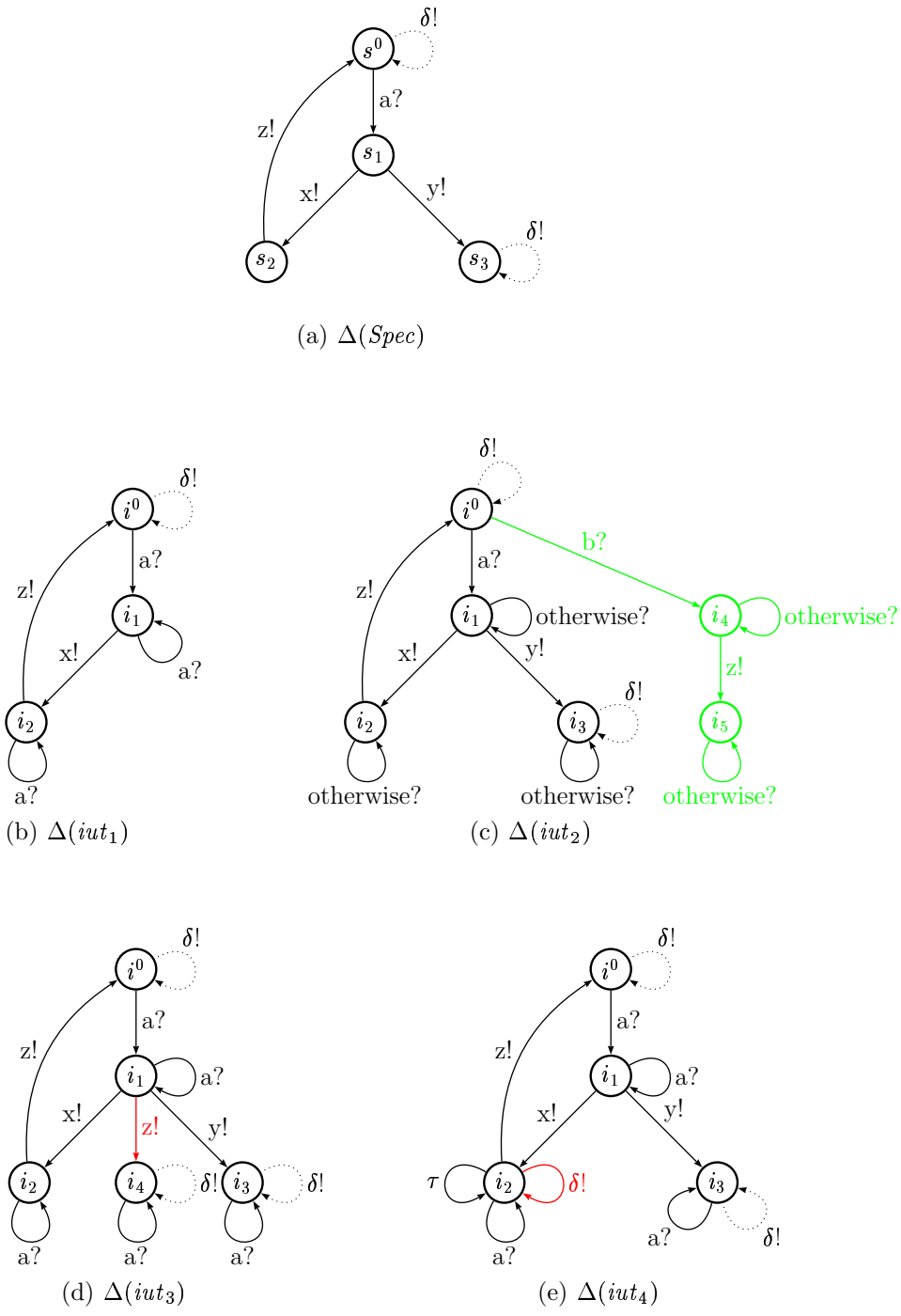


Figure 3.12: An example illustrating ioco relation.

(1) (iut_1 ioco $Spec$) as for all traces of the suspension specification $Spec$, the outputs of $\Delta(iut_1)$ are included into the outputs of $\Delta(Spec)$. For instance, $Out(\Delta(iut_1)\ \mathsf{after}\ \delta.a.x.z.\delta.\delta.a) = \{x\} \subseteq Out(\Delta(Spec)\ \mathsf{after}\ \delta.a.x.z.\delta.\delta.a) = \{x, y\}$, where $\delta.a.x.z.\delta.\delta.a \in STraces(Spec)$.

(2) (iut_2 ioco $Spec$) as all suspension traces of $\Delta(Spec)$ are traces of $\Delta(iut_2)$. Thus, it is not hard to check that after all suspension traces of $Spec$, $\Delta(iut_2)$ produces exactly same outputs as $\Delta(Spec)$.

Moreover, from the initial state i^0, $\Delta(iut_2)$ can execute the *input* action b (shown in green on Figure 3.12(c), page 49) which is not specified in $\Delta(Spec)$. However, the ioco relation allows this input action, because ioco checks only the inclusions of output actions after all possible traces of $\Delta(Spec)$. This observation gives us a possibility to describe partial specifications.

(3) $\neg(iut_3$ ioco $Spec)$ as the output action z of $\Delta(iut_3)$ after the input action a is not allowed by $\Delta(Spec)$ (*see* the red edge of Figure 3.12(d), page 49). Formally, $Out(\Delta(iut_3)\ \mathsf{after}\ a) = \{x, y, z\} \not\subseteq Out(\Delta(Spec)\ \mathsf{after}\ a) = \{x, y\}$.

(4) $\neg(iut_4$ ioco $Spec)$ as the blocking δ (here it is a livelock) of $\Delta(iut_4)$ after the suspension trace $a.x$ of $\Delta(Spec)$ is not allowed by the specification (*see* the loop shown in read in Figure 3.12(e), page 49). Indeed, $Out(\Delta(iut_4)\ \mathsf{after}\ a.x) = \{z, \delta\} \not\subseteq Out(\Delta(Spec)\ \mathsf{after}\ a) = \{z\}$.

□

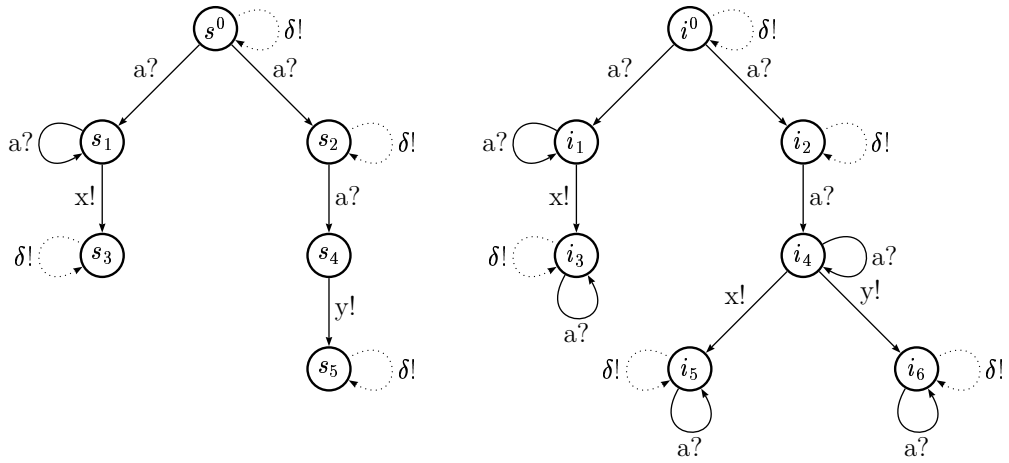
Ioconf vs. Ioco. The aim of this paragraph is to compare the two implementation relations ioconf and ioco (*see* Definitions 3.10 and 3.11, pages 45 and 47).

The ioconf and ioco conformance relations are quite similar. They both require an implementation under test to react correctly to the traces that are explicitly specified in the specification. In the same time, they allow the implementation to react in any possible way to traces that are not explicitly specified. This means that the ioconf and ioco relations permit the partial description of specifications, which significantly simplifies the testing tack.

Nevertheless, the ioconf and ioco relations are different. Indeed, the purpose of the ioco relation is to check whether after the execution of each *suspension* trace (*i.e.* trace that may contain the δ action marking all specified blockings) of a specification, an implementation under test produces only specified outputs or not. The ioconf relation checks the same things as ioco but for each proper trace (*i.e.* trace that does not contain any δ action) of a specification. This difference between the ioconf and ioco relations is illustrated on an example which is a

slightly modified version of the example used by J. Tretmans in his paper [Tretmans, 1996b].

Example 3.12 Consider, a specification *Spec* whose suspension version is shown on Figure 3.13(a) and an implementation under test *iut* whose suspension version



(a) $\Delta(Spec)$ (b) $\Delta(iut)$

Figure 3.13: An example illustrating difference between ioconf and ioco relations.

is depicted in Figure 3.13(b).

It is not hard to check that the implementation *iut* is not ioco-related to its specification *Spec*, *i.e.* $\neg(iut \text{ ioco } Spec)$. Consider the suspension trace $a.\delta.a$ of $\Delta(Spec)$. Then we obtain that: $Out(\Delta(iut) \text{ after } a.\delta.a) = \{x, y\}$ (*see* Figure 3.13(a)), where the output action x cannot be produced by $\Delta(Spec)$ after the trace $a.\delta.a$, indeed $Out(\Delta(Spec) \text{ after } a.\delta.a) = \{y\}$ (*see* Figure 3.13(a)).

However, *iut* is ioconf-related to *Spec*, *i.e.* $(iut \text{ ioconf } Spec)$. As for each trace σ of *Spec*, where σ is a word in the following language: $a + a.a^* + a.a^*.x + a.a.y$, the implementation *iut* can produce only specified output actions. For instance, consider the trace $a.a$ of *Spec*, then $Out(\Delta(iut) \text{ after } a.a) = \{x, y\} \subseteq Out(\Delta(Spec) \text{ after } a.a) = \{x, y\}$. Therefore, using the ioconf relation we cannot distinguish the left branch of *Spec* and *iut* from the right one. □

This section has intuitively shown that the ioco relation is stronger than ioconf. Moreover, it treats more carefully the problem of blockings than ioconf. Due to these reasons the ioco relation is used by many researchers as the basis for testing and test generation (*see* Subsections 3.2.3 and 3.2.4, page 51 and 64).

Nevertheless, in this thesis we build the testing theory and test generation method based on a weaker variant of the ioconf relation (and, therefore, ioco) which (1) does not treat the problem of blockings at all, and (2) is adapted for the *symbolic* variant of input-output labeled transition systems (presented in Chapter 4, page 77 and called IOSTS). The main reason for our choice is that the problem of detecting when a system modeled by IOSTS is blocked is *undecidable* in general. However, in the case of absence syntactical livelocks in a given IOSTS, it is possible to syntactically build its suspension IOSTS. Thus, in this case it is possible to extend our conformance relation to ioco defined by J. Tretmans. The idea of this extension has been proposed in [Rusu et al., 2004].

3.2.3 Ioco-Based Test Generation Algorithms

This subsection describes two test generation algorithms based on the IOLTS model and the ioco relation. Before describing these algorithms we consider (1) a specification a specification whose behaviors are modeled by a *convergent* IOLTS *Spec* with the alphabets of input $\Sigma^?_{Spec}$ and output $\Sigma^!_{Spec}$ actions, and (2) an implementation under test whose possible behaviors are modeled by an *input-complete* IOLTS $iut = \langle S_{iut}, s^0_{iut}, (\Sigma_{iut} \cup \{\tau\}), \rightarrow_{iut} \rangle$ with alphabets of input $\Sigma^?_{iut} \subseteq \Sigma^?_{Spec}$ and output $\Sigma^!_{iut} \subseteq \Sigma^!_{Spec}$ actions. Also we consider that the implementation relation (*see* Section 2.1.3, page 15) relating *Spec* and *iut* is instantiated with ioco (*see* Definition 3.11, page 47). Then, by analogy with the habilitation document dissertation [Jéron, 2004] of T. Jéron we formally define a *test case* as follows.

Definition 3.12 (Test Case) A *test case* is an IOLTS $TC = \langle S_{TC}, s^0_{TC}, \Sigma_{TC}, \rightarrow_{TC} \rangle$ such that:

(1) The set of states S_{TC} is equipped with three disjoint sets of states $\mathsf{Pass} \subseteq S_{TC}$, $\mathsf{Fail} \subseteq S_{TC}$ and $\mathsf{Inconclusive} \subseteq S_{TC}$ that do not have any successor.

(2) The alphabet of actions Σ_{TC} consists of only *input* and *output* actions, *i.e.* $\Sigma_{TC} = \Sigma^?_{TC} \cup \Sigma^!_{TC}$, where $\Sigma^?_{TC}$ is equipped with the special action δ indicating blockings detection in an implementation under test.

(3) TC is *controllable*, *i.e.* for any state $s \in S_{TC}$, TC does not have the choice neither between output actions, nor between input and output actions. Formally:

$$\forall\, s \in S_{TC} \,.\, [\, \exists\, a \in \Sigma^!_{TC} \,.\, [s \xrightarrow{a}_{TC}] \implies \forall\, b \in \Sigma_{TC} \,.\, [(b \neq a) \implies (s \not\xrightarrow{b}_{TC})]\,]$$

(4) All states of TC from which we can execute an *input* action are *input-complete*, *i.e.*

$$\forall\, s \in S_{TC} \,.\, [\, \exists\, a \in \Sigma^?_{TC} \,.\, [s \xrightarrow{a}_{TC}] \implies \forall\, b \in \Sigma^?_{TC} \,.\, [s \xrightarrow{b}_{TC}]\,]$$

(5) The states belonging to the Fail or Inconclusive set of states are reachable only by input actions, *i.e.*

$$\forall \langle s, a, s' \rangle \in \rightarrow_{TC} \ . \ [\ s' \in (\mathsf{Fail} \cup \mathsf{Inconclusive}) \implies a \in \Sigma^{?}_{TC}\ ]$$

(6) From each state of TC a verdict (*Fail*, *Pass*, or *Inconclusive*) is accessible:

$$\forall\ s \in S_{TC},\ \exists\ \sigma \in (\Sigma_{TC})^{*},\ s' \in (\mathsf{Pass} \cup \mathsf{Fail} \cup \mathsf{Inconclusive})\ .\ [\ s \stackrel{\sigma}{\rightarrow} s'\ ]$$

□

Some examples of test cases are shown on Figures 3.14 and 3.16 (*see* pages 54 and 62), where the abbreviation **otherwise?** is used to indicate all possible inputs of a tester that are sent by an implementation under test, and are not specified in a test case.

In order to model the execution of a test case TC on an implementation under test *iut* we use the parallel composition defined on page 38. The aim of the test execution is to produce a testing verdict which is either *Pass*, *Fail* or *Inconclusive*. Here, *Pass* means that no observable difference between a specification and an implementation is detected; *Fail* means that an implementation behaves differently from its specification; and *Inconclusive* (which is used only in testing based on a test purpose) means that no error is detected, but a test purpose is not satisfied. The formal definition of verdict can be found, for instance, in [Tretmans, 1996b], [Morel, 2000], [Jéron, 2004].

In the rest of this subsection we describe test generation algorithms whose main purpose is to derive sound and exhaustive test cases. These properties of test cases permit to relate derived test cases with the notion of conformance. Remind that a test case is (1) *sound* if it does not reject conformant implementations, and (2) *exhaustive* if implementations which do not conform to their specification may be rejected by some test case. In this section we do not give the formal definitions of soundness and exhaustiveness as they can be easily found in works of M.-C. Gaudel, T. Jéron, J. Tretmans and other researchers working in the testing area.

3.2.3.1 Blockings and Visible Behaviors of Specifications

Before describing the test generation algorithms, we remind that their aim is to derive a test case TC from a specification *Spec*, that detects unspecified blockings in an implementation under test *iut*. For this we have to find all possible blockings in *Spec* and mark them with the special output action δ. In other words, the first

preliminary step for the test generation algorithms is to build a suspension IOLTS $\Delta(Spec)$ for the given specification *Spec*.

Next, notice that during test execution, the tester first observes all possible reactions of the implementation *iut* on its stimuli (*i.e.* the sequences of input/output actions or traces), and, then, compare them with the specified ones. For this it is useful to have a direct access to the set of traces of the specification $Spec = \langle S_{Spec}, s^0_{Spec}, (\Sigma_{Spec} \cup \{\tau\}), \rightarrow_{Spec} \rangle$, which can be characterized by the deterministic IOLTS $\mathsf{det}(Spec) = \langle S_{det}, s^0_{det}, \Sigma_{det}, \rightarrow_{det} \rangle$, where

- $S_{det} = 2^{S_{Spec}}$ is the set of states,
- $s^0_{det} = (s^0_{Spec}\ \mathsf{after}\ \varepsilon)$ is the initial state,
- $\Sigma_{det} = \Sigma^?_{det} \cup \Sigma^!_{det}$ is the alphabet consisting of input $\Sigma^?_{det} = \Sigma^?_{Spec}$ and output $\Sigma^!_{det} = \Sigma^!_{Spec}$ actions,
- the set of transitions is constructed as follows:

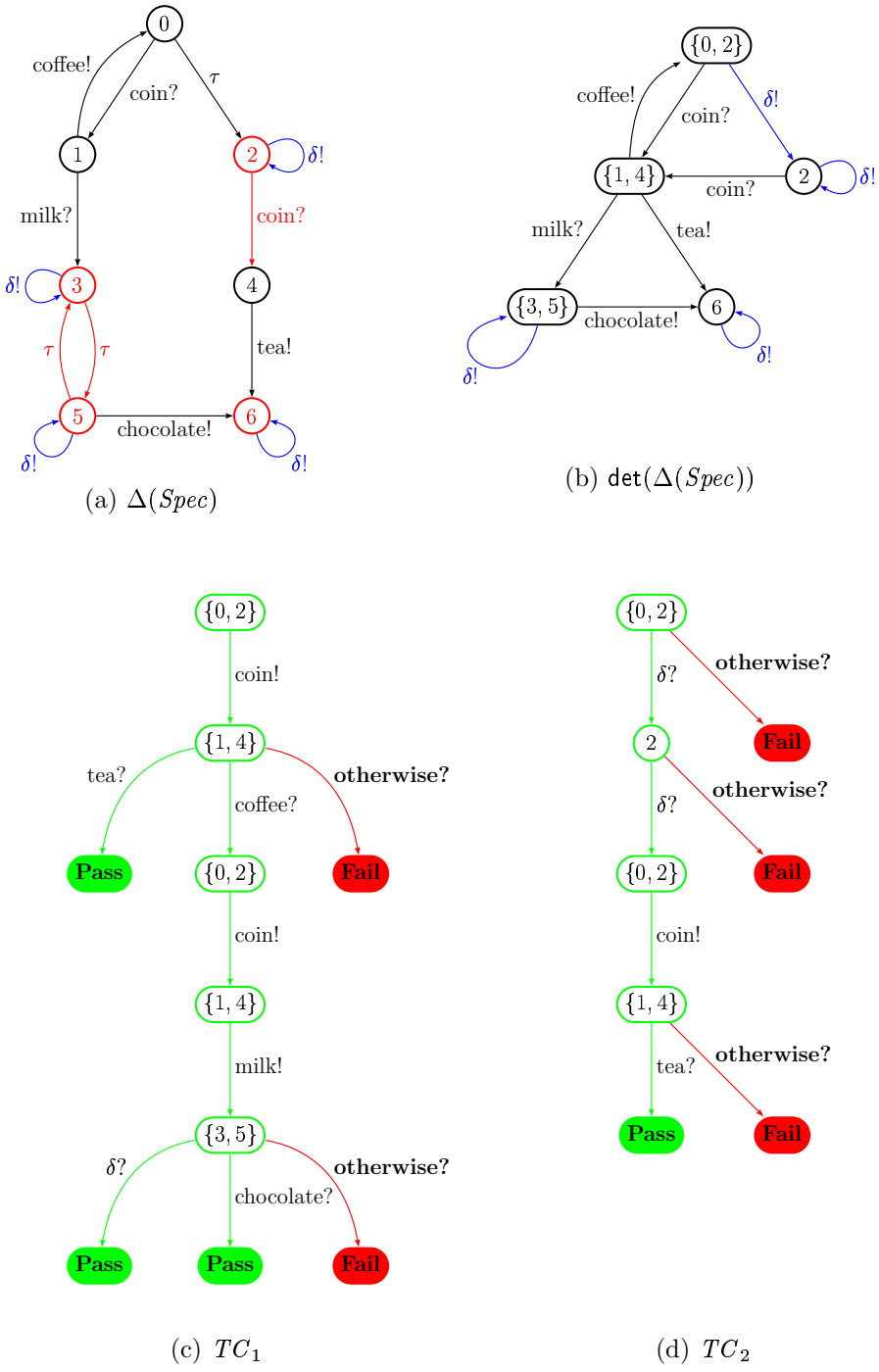
$$s \xrightarrow{a}_{det} s' \iff (s, s' \in S_{det}) \ \wedge \ (a \in \Sigma_{det}) \ \wedge \ (s' = (s\ \mathsf{after}\ a)).$$

This algorithm for transformation of a given IOLTS *Spec* into a deterministic one, *i.e.* $\mathsf{det}(Spec)$, where $Traces(Spec) = Traces(\mathsf{det}(Spec))$, is identical to the classical algorithm for determinization of finite state automata (*see* [Hopcroft and Ullman, 1979]), which consists in two steps: (1) τ-reduction, and (2) subset construction. It is illustrated on Figures 3.14(a) and 3.14(b) (*see* page 54).

Finally, it is important to emphasize that if we extract visible behaviors of a specification *Spec* before detecting blockings in it, then we risk to lose information about some blockings implicitly specified in *Spec*. For instance, if we determinize the specification *Spec* whose suspension IOLTS $\Delta(Spec)$ is shown on Figure 3.14(a) (*see* page 54), then in the resulting IOLTS $\mathsf{det}(Spec)$ we will not find neither outputlock nor livelock in *Spec*, as information about them was lost during determinization. Therefore, in the case when the problem of unspecified blockings must be treated, it is necessary first to construct $\Delta(Spec)$, and then to extract visible behaviors of $\Delta(Spec)$, *i.e.* to build $\mathsf{det}(\Delta(Spec))$.

3.2.3.2 The Approach of J. Tretmans

In this section we describe a slightly modified version of test generation algorithm proposed by J. Tretmans in his PhD thesis [Tretmans, 1992]. The difference between the original algorithm and the algorithm proposed in this section is that: the problem of detection of outputlocks, deadlocks (livelocks are not considered in the works of J. Tretmans), and visible behaviors of a specification *Spec* is incorporated into the original algorithm, but not in the proposed one. In the



(a) $\Delta(Spec)$

(b) $\mathsf{det}(\Delta(Spec))$

(c) TC_1

(d) TC_2

Figure 3.14: The approach of J. Tretmans.

proposed algorithm we suppose that $\mathsf{det}(\Delta(\mathit{Spec}))$ is already computed. In our opinion this trick simplifies understanding of the approach of J. Tretmans.

Algorithm 3.1 (Tretmans Approach) Let $\mathit{Spec}_{vis} = \mathsf{det}(\Delta(\mathit{Spec})) = \langle S_{vis}, s^0_{vis}, (\Sigma_{vis} \cup \{\tau\}), \rightarrow_{vis} \rangle$ be an IOLTS which was obtained from a given specification *Spec* by detection of all possible blockings in *Spec* (*i.e.* by building of $\Delta(\mathit{Spec})$), and by determinization of $\Delta(\mathit{Spec})$ (*see* the previous subsection, page 53). Let also $\Delta(\mathit{iut})$ be a suspension IOLTS of an implementation under test *iut*.

Then, from the specification *Spec* we to construct a test case $TC = \langle S_{TC}, s^0_{TC}, \Sigma_{TC}, \rightarrow_{TC} \rangle$, where

- $S_{TC} = S'_{TC} \cup \mathsf{Pass} \cup \mathsf{Fail}$, where $S'_{TC} \subseteq S_{vis}$,
- $s^0_{TC} = s^0_{vis}$,
- $\Sigma_{TC} = \Sigma^?_{TC} \cup \Sigma^?_{TC}$, where $\Sigma^?_{TC} \subseteq \Sigma^!_{vis}$ and $\Sigma^!_{TC} \subseteq \Sigma^?_{vis}$.

as follows:

(1) We create an IOLTS TC consisting of only one initial state, *i.e.* $S_{TC} = \{s^0_{TC}\}$ and $\rightarrow_{TC} = \emptyset$. This IOLTS will represent a test case at the end of the algorithm.

(2) **While** TC contains at least one state $s \in S_{TC}$ which (a) does not belong neither to the Pass nor Fail set of states, and (b) does not have any outgoing transitions belonging to $\rightarrow_{TC}$, then perform one of the following three steps:

Stopping Condition: $\mathsf{Pass} := \mathsf{Pass} \cup \{s\}$.

Notice that this step permits to stop the exploration of the given specification *Spec*.

Execution of an Input Action: If the set of input actions of Spec_{vis} that can be executed from the state s, is not empty, *i.e.*

$$in_{vis}(s) \quad \triangleq \quad \{ a \in \Sigma^?_{vis} \mid s \stackrel{a}{\rightarrow}_{vis} \} \neq \emptyset$$

then choose one of these actions, for instance, $a \in in_{Spec_{vis}}(s)$, and update the set of states S_{TC} and set of transitions $\rightarrow_{TC}$ of the test case TC as follows:

(a) $S_{TC} := S_{TC} \cup \{(s \text{ after } a)\}$,
(b) $\rightarrow_{TC} := \rightarrow_{TC} \cup \{\langle s, a, (s \text{ after } a)\rangle\}$.

Execution of All Output Actions: If the set of output actions of $Spec_{vis}$ that can be executed from the state s, is not empty, *i.e.*

$$out_{vis}(s) \quad \triangleq \quad \{a \in \Sigma^{?}_{vis} \mid s \xrightarrow{a}_{vis} \} \neq \emptyset$$

then update the set of states S_{TC} and set of transitions $\rightarrow_{TC}$ of the test case TC as follows:

$$\text{(a) } S_{TC} \quad := \quad S_{TC} \cup \left(\bigcup_{a \in out_{vis}(s)} \{(s \text{ after } a)\} \right) \cup \{Fail\},$$

$$\text{(b) } \rightarrow_{TC} \quad := \quad \rightarrow_{TC} \cup \left(\bigcup_{a \in out_{vis}(s)} \{\langle s, a, (s \text{ after } a)\rangle\} \right) \cup \left(\bigcup_{a \in (\Sigma^{!}_{\Delta(iut)}) \setminus (out_{vis}(s))} \{\langle s, a, Fail\rangle\} \right).$$

□

Finally, we illustrate the algorithm for test generation proposed by J. Tretmans with an example.

Example 3.13 Consider an IOLTS $\mathsf{det}(\Delta(Spec))$ shown on Figure 3.14(b) (*see* page 54). By applying the Tretmans approach (formalized as Algorithm 3.1) to this IOLTS, we can obtain the test cases depicted in Figures 3.14(c) and 3.14(d) (*see* page 54).

The first test case TC_1 (*see* Figure 3.14(c), page 54) stimulates an implementation representing a coffee machine with the **coin** action, and then waits for a response from the implementation *iut*. If TC_1 receives an action which is different from **coffee** or **tea**, then it stops with the *Fail* verdict. Otherwise it continues the computation as follows: (1) it can choose to stop by accepting the **tea** action and producing the *Pass* verdict, or (2) it can accept the **coffee** action and continue stimulate the implementation *iut* with the **coin** action until producing either *Pass* or *Fail* verdict.

The second test case TC_2 shown on Figure 3.14(d) (*see* page 54) chooses to be silent, *i.e.* it waits for some response from the implementation *iut*. If the implementation indicates that it is blocked (by sending δ action), then TC_2 decides to wait again for some response from *iut*. If it receives the δ action the second time, it stimulates *iut* with the **coin** action, waits for the **tea** action from *iut*, and then stops by producing the *Pass* verdict. In all other cases, the test case TC_2 stops with the *Fail* verdict. □

The set of test cases (possibly infinite) produced by Algorithm 3.1 (*see* page 55) is sound and exhaustive. This statement was formulated by J. Tretmans as Theorem 6.3 in his paper [Tretmans, 1996b].

3.2.3.3 Test Generation Guided by Test Purposes

In the previous subsection (*see* page 55) we have described the Tretmans approach, where the derivation of test cases strongly depends on non-deterministic choices of the algorithm (*see* page 55). Thus, it is not possible to generate test cases for exercising a specific part of a implementation under test. Nevertheless, this aspect is quite important for practical testing, as it is often necessary to test only a part of the implementation and not the entire implementation. The researchers working in the testing area have studied this aspect and proposed some solutions. One of these solutions is to guide the test derivation process with *test purposes* that formally describe a part of the specification for which test cases must be generated. It has been proposed by T. Jéron and his colleagues from IRISA/INRIA Rennes, France (*see* [Fernandez et al., 1996], [Morel, 2000], [Jard and Jéron, 2002], [Jéron, 2002], [Jéron, 2004]). In these works a test purpose is represented as an IOLTS consisting of incomplete sequences of specified input and output actions. Formally:

Definition 3.13 (Test Purpose) Let *Spec* be a specification with alphabets of input $\Sigma^{?}_{Spec}$ and output $\Sigma^{!}_{Spec}$ actions. Then, a *test purpose* of *Spec* is an IOLTS $TP = \langle S_{TP}, s^{0}_{TP}, \Sigma_{TP}, \rightarrow_{TP} \rangle$ such that:

(1) The set of states S_{TP} is equipped with two disjoint set of states $\mathsf{Accept} \subseteq S_{TP}$ and $\mathsf{Reject} \subseteq S_{TP}$.

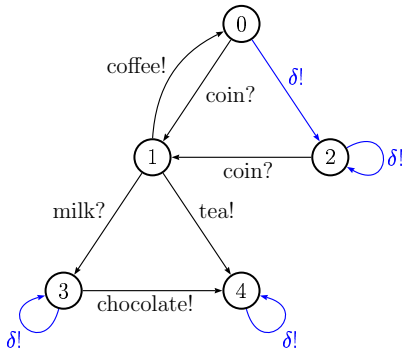
(2) $\Sigma_{TP} = \Sigma^{?}_{TP} \cup \Sigma^{!}_{TP}$, where $\Sigma^{?}_{TP} = \Sigma^{?}_{Spec}$ and $\Sigma^{!}_{TP} = \Sigma^{!}_{Spec} \cup \{\delta\}$.

(3) TP is deterministic (*see* the item (8) of Definition 3.4 on page 35).

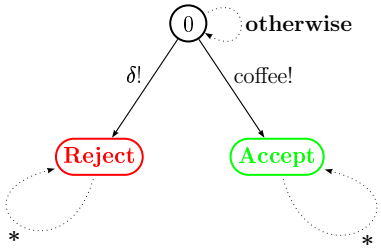
(4) TP is complete, *i.e.* $\forall\ s \in S_{TP},\ a \in \Sigma_{TP}\ .\ [\ s \xrightarrow{a}_{TP}\ ]$.

□

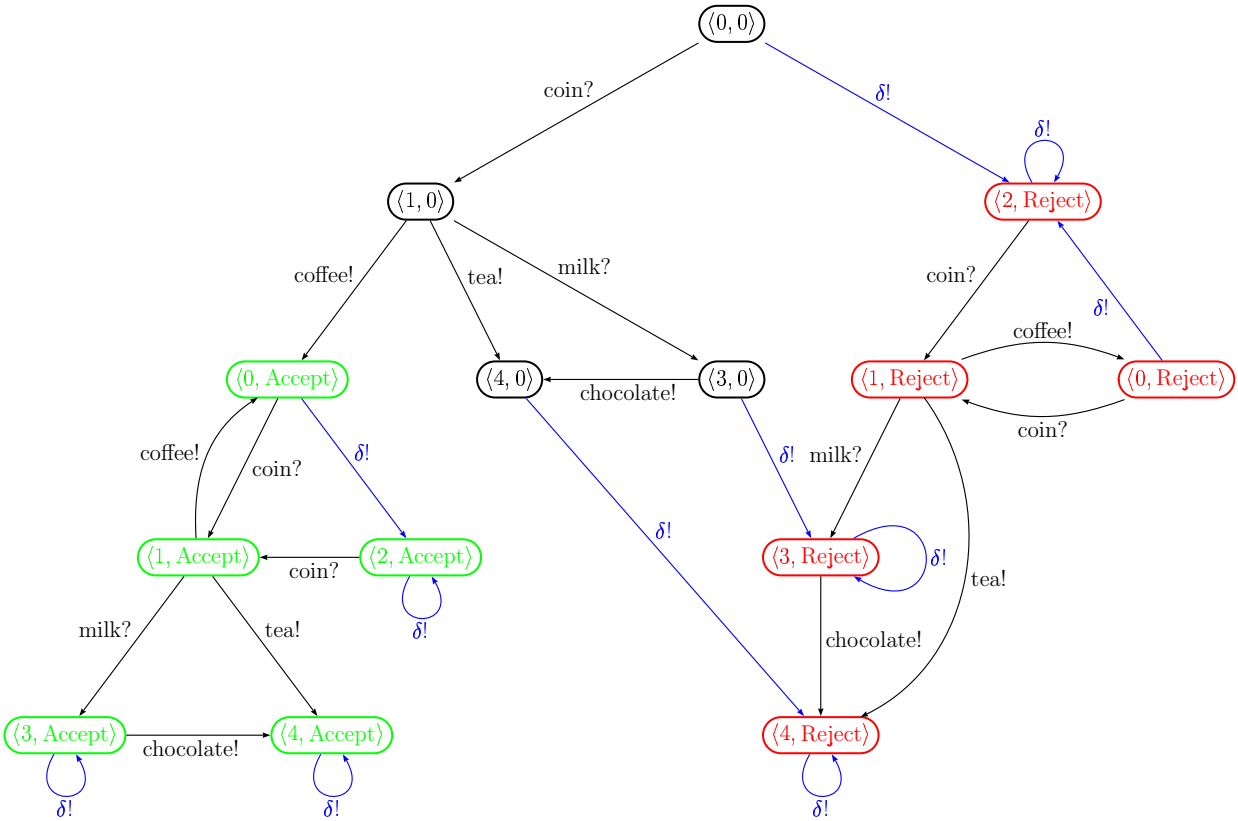
Figure 3.15(b) on page 58 gives an example of a possible test purpose for the specification *Spec* depicted in Figure 3.14(a) (*see* page 54) is shown on Figure 3.15(b) (*see* page 58). This test purpose TP describes behaviors of the coffee machine where the latter delivers coffee and does not block. An *accepted* (*resp. rejected*) *behavior* is indicated by the arrival into the *Accept* (*resp. Reject*) location. It is important to emphasize that the rejected behaviors of TP are not necessary erroneous. They are just the behaviors that are not targeted by TP. Notice also



(a) $\mathsf{det}(\Delta(Spec))$



(b) TP



(c) $SP = (\mathsf{det}(\Delta(Spec)) \times TP)$

Figure 3.15: Computation of the synchronous product.

that on Figure 3.15(b) (and later) we use the * abbreviation indicating a set of transitions labeled with all possible actions of TP except those specified.

Finally, we describe the ioco-based test generation algorithm which uses a test purpose in order to extract the part of a specification for which a test case should be derived.

Algorithm 3.2 (Test Generation Guided by Test Purposes) Consider a specification *Spec* and a test purpose TP of *Spec*. Then, a test case TC can be generated by the following steps:

Blockings and Visible Behaviors of *Spec*. The first step of the algorithm consists in the detection of all blockings in the given specification *Spec* (*i.e.* building of $\Delta(Spec)$) as well as extracting visible behaviors from $\Delta(Spec)$ (*i.e.* building of $\mathsf{det}(\Delta(Spec))$). This step was already explained in Subsection 3.2.3.1 (*see* page 53).

Synchronous Product. The purpose of the second step is to identify some visible behaviors of the specification as accepted using the given test purpose TP. In order to do this we compute a synchronous product between the IOLTS $\mathsf{det}(\Delta(Spec))$ obtained at the previous step of the algorithm, and the test purpose TP. Formally:

Definition 3.14 (Synchronous Product) Let $Spec_{vis} = \mathsf{det}(\Delta(Spec)) = \langle S_{vis}, s^0_{vis}, \Sigma_{vis}, \rightarrow_{vis} \rangle$ be the deterministic suspension IOLTS of a specification *Spec*, and $TP = \langle S_{TP}, s^0_{TP}, \Sigma_{TP}, \rightarrow_{TP} \rangle$ be a test purpose of *Spec*. Then, the IOLTS $SP = \langle S_{SP}, s^0_{SP}, \Sigma_{SP}, \rightarrow_{SP} \rangle$ is the *synchronous product* between $Spec_{vis}$ and TP, where:

(1) $S_{SP} = S_{vis} \times S_{TP}$ is the set of states equipped with two sets of accepting and rejecting states which are defined as follows: $\mathsf{Accept}_{SP} \triangleq S_{vis} \times \mathsf{Accept}_{TP}$ and $\mathsf{Reject}_{SP} \triangleq S_{vis} \times \mathsf{Reject}_{TP}$,

(2) $s^0_{SP} = \langle s^0_{vis}, s^0_{TP} \rangle$ is the initial state,

(3) $\Sigma_{SP} = \Sigma^?_{SP} \cup \Sigma^!_{SP}$ is the alphabet of input and output actions such that $\Sigma^?_{SP} = \Sigma^?_{vis} = \Sigma^?_{TP}$ and $\Sigma^!_{SP} = \Sigma^!_{vis} = \Sigma^!_{TP}$, and

(4) the transition relation $\rightarrow_{SP}$ is defined as follows:

$$\langle s_{vis}, s_{TP} \rangle \xrightarrow{a}_{SP} \langle s'_{vis}, s'_{TP} \rangle \iff (s_{vis} \xrightarrow{a}_{vis} s'_{vis}) \wedge (s_{TP} \xrightarrow{a}_{TP} s'_{TP})$$

□

An example of the synchronous product computed from the deterministic suspension IOLTS $\mathsf{det}(\Delta(Spec))$ (*see* Figure 3.15(a), page 58) and its test purpose TP (*see* Figure 3.15(b), page 58) is shown on Figure 3.15(c) (*see* page 58).

Selection or Complete Test Graph Construction. Remind that the aim of this test generation algorithm is to construct a test case that examines behaviors of the specification *Spec* that are selected by the test purpose TP. Therefore, it is not necessary to keep all traces of the synchronous product SP computed at the previous step of the algorithm. It is enough to choose behaviors of SP which (1) consist of only reachable states of SP, and (2) lead to accepting states of SP. (The notions of reachable and coreachable states which will be used later, is given as the items (12) and (13) on page 35.) Formally, we construct a complete test graph which is defined as follows:

Definition 3.15 (Complete Test Graph) Let $SP = \langle S_{SP}, s^0_{SP}, \Sigma_{SP}, \rightarrow_{SP} \rangle$ be the synchronous product computed from the deterministic suspension specification $Spec_{vis} = \mathsf{det}(\Delta(Spec))$ with alphabet of input actions $\Sigma^?_{vis}$ and a test purpose TP. Let also *iut* be an implementation under test with alphabet of output actions $\Sigma^!_{iut}$.

Then, a *complete test graph* is defined as an IOLTS $CTG = \langle S_{CTG}, s^0_{CTG}, \Sigma_{CTG}, \rightarrow_{CTG} \rangle$ selected from the synchronous product SP, where:

(1) $S_{CTG} = \mathsf{L2A} \cup \mathsf{Inconclusive} \cup \mathsf{Fail}$ is the set of states, where:

   (a) $\mathsf{L2A} = (Reach(\{s^0_{SP}\}) \cap CoReach(\mathsf{Accept}_{SP})) \setminus \mathsf{A2A}$, where $\mathsf{A2A} \triangleq \{s \in \mathsf{Accept}_{SP} \mid \mathsf{pre}_{\Sigma}(\{s\}) \subseteq (\mathsf{Accept}_{SP} \cup (S_{SP} \setminus Reach(\{s^0_{SP}\})))\}$, is the set of states that are reachable from the initial state of SP and coreachable from the accepting states of SP. Moreover, this set does not contain accepting states of SP that are direct successors of either other accepting states of SP, or unreachable from s^0_{SP} states of SP.
   Next, we denote by $\mathsf{Pass} = \mathsf{L2A} \cap \mathsf{Accept}_{SP}$ the set of *pass* states.

   (b) $\mathsf{Inconclusive} = \mathsf{post}_{\Sigma^!_{SP}}(\mathsf{L2A}) \setminus \mathsf{L2A}$ is the set of *inconclusive* states, where each state belonging to Inconclusive: (1) is a direct successor of a state belonging to L2A by an output action of SP, and (2) does not belong to L2A.

   (c) $\mathsf{Fail} = \{Fail\}$ is the set consisting on only one new state called *Fail*.

(2) $s^0_{CTG} = s^0_{SP}$ is the initial state in the case when $\mathsf{L2A} \neq \emptyset$. Notice that if $\mathsf{L2A} = \emptyset$, then CTG is the empty IOLTS.

(3) $\Sigma_{CTG} = \Sigma^{?}_{CTG} \cup \Sigma^{!}_{CTG}$ is the alphabet of input and output actions, where $\Sigma^{!}_{CTG} = \Sigma^{?}_{SP} = \Sigma^{?}_{\mathit{vis}}$ and $\Sigma^{?}_{CTG} = \Sigma^{!}_{\mathit{iut}} \cup \{\delta\}$.
It is important to notice that the alphabets of input and output actions are inverted due to the fact that: in order to be able to communicate with an implementation, (1) output actions and blockings of the implementation must be considered as input actions for a test case, and (2) input actions of the implementation must be interpreted as output actions of the test case.

(4) $\rightarrow_{CTG} \;=\; \rightarrow_{\mathsf{L2A}} \cup \rightarrow_{\mathsf{Inconclusive}} \cup \rightarrow_{\mathsf{Fail}}$ is the transition relation, where:

(a) $\rightarrow_{\mathsf{L2A}} \;=\; \{\langle s, a, s'\rangle \in \rightarrow_{SP} \mid s \in (\mathsf{L2A} \setminus \mathsf{Pass}) \wedge a \in \Sigma_{CTG} \wedge s' \in \mathsf{L2A}\}$ is the set of transitions leading to "lead to accept" states,

(b) $\rightarrow_{\mathsf{Inconclusive}} \;=\; \{\langle s, a, s'\rangle \in \rightarrow_{SP} \mid s \in (\mathsf{L2A} \setminus \mathsf{Pass}) \wedge a \in \Sigma^{?}_{CTG} \wedge s' \in \mathsf{Inconclusive}\}$ is the set of transitions leading to *inconclusive* states, and labeled with *input* actions of CTG,

(c) $\rightarrow_{\mathsf{Fail}} \;=\; \{\langle s, a, \mathit{Fail}\rangle \notin \rightarrow_{SP} \mid s \in (\mathsf{L2A} \setminus \mathsf{Pass}) \wedge a \in \Sigma^{?}_{CTG} \wedge s \overset{a}{\nrightarrow}_{SP}\}$ is the set of transitions leading to the *Fail* state, and labeled with *input* actions of CTG.
Notice that the introduction of the transitions leading to the *Fail* state makes CTG input-complete (with respect to input action of the given implementation *iut*). Therefore, CTG can properly react on an incorrect input action and blocking incoming from this implementation.

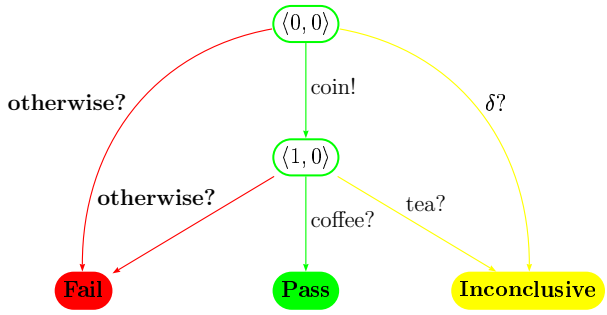
□

We illustrate the complete test graph construction with an example below.

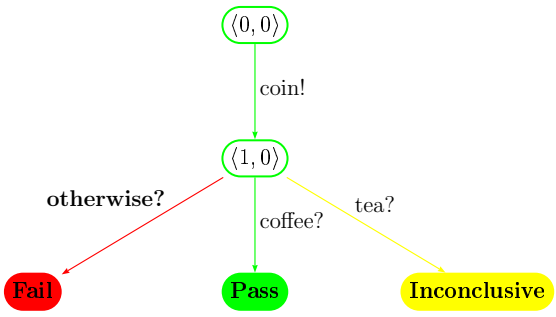
Example 3.14 The complete test graph obtained from the synchronous product shown on (*see* Figure 3.15(c), page 58) by the third step of the algorithm is shown on Figure 3.16(a) (*see* page 62). The two crucial points in this computation are the construction of the states and transitions of CTG.

First, we need to construct the set of "leads to accept" states. For this we need to compute:

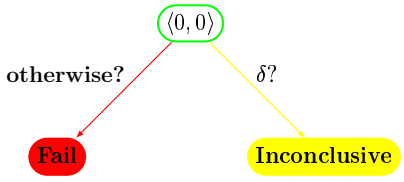
- the set of states reachable from the initial state $\langle 0, 0\rangle$ of SP, which is equal to the set of states of SP, *i.e.* $\mathit{Reach}(\{\langle 0, 0\rangle\}) = S_{SP}$,
- the set of states from which it is possible to go to pass, which is $\mathit{CoReach}(\mathsf{Accept}_{SP}) = \mathsf{Accept}_{SP} \cup \{\langle 0, 0\rangle, \langle 1, 0\rangle\}$, and



(a) CTG



(b) TC_1



(c) TC_2

Figure 3.16: The complete test graph and two possible test cases.

– the set of "accept to accept" states which is $\mathsf{A2A} = \{\langle 1, \mathit{Accept}\rangle, \{\langle 2, \mathit{Accept}\rangle, \{\langle 3, \mathit{Accept}\rangle, \{\langle 4, \mathit{Accept}\rangle\}$.

Using this information we construct the "leads to accept" states: $\mathsf{L2A} = (\mathit{Reach}(\{\langle 0,0\rangle\}) \cap \mathit{CoReach}(\mathsf{Accept}_{SP})) \setminus \mathsf{A2A} = \{\langle 1, \mathit{Accept}\rangle, \langle 0,0\rangle, \langle 1,0\rangle\}$, where the state $\langle 1, \mathit{Accept}\rangle$ is the pass state of CTG. The obtained states are shown in green on Figure 3.16(a) (*see* page 62).

Second, we compute the set of inconclusive states. For this we compute the direct successors of the "leads to accept" states previously calculated, by the output actions of SP: $\mathsf{post}_{\{\mathit{coffee}, \mathit{tea}, \mathit{chocolate}, \delta\}}(\mathsf{L2A}) = \{\langle 2, \mathit{Reject}\rangle, \langle 4, 0\rangle\}$. On Figure 3.16(a) (*see* page 62) these two states are merged into one state shown in yellow and called *Inconclusive*.

Third, we augment the set of state of CTG with one *Fail* state shown in red on Figure 3.16(a) (*see* page 62).

Next, we connect all green and yellow states with the same transitions with which they were connected in the IOLTS SP (*see* Figures 3.15(c) and 3.16(a), pages 58 and 3.16(a)). Finally, for each state different from *Pass*, *Fail* and *Inconclusive* we add a transition labeled with **otherwise?** (indicating all possible input actions which can be received by CTG) and leading to the *Fail* state. □

Controllability. The last step of the algorithm consists in solving the controllability conflicts in the complete test graph CTG computed at the previous step. There exist two kinds of controllability conflicts that may appear in a state s of CTG: the first one is the choice between *several output actions*, and the second one is the choice between *input and output actions* (*see* for example, the choice between *coin*! and δ? in the initial state $\langle 0,0\rangle$ of CTG shown on Figure 3.16(a), page 62).

These controllability conflicts can be solved by using a simple technique that for each state s of CTG either:

(1) keeps one transition labeled with an output action. In this case the test case takes the initiative, it chooses how to control an implementation under test (*see* the test case shown on Figure 3.16(b), page 62). Or,

(2) keeps all transitions labeled with input actions. In this case the test case gives an initiative to an implementation under test, and observes its responses (*see* the test case depicted in Figure 3.16(c), page 62).

Notice that after applying this technique to a complete test graph we may obtain several test cases. These test cases may contain states that are not

reachable (*resp.* not coreachable) from their initial (*resp.* pass) states. Therefore, it is recommended to perform reachability and coreachability analysis on the resulting test cases (*see* the selection step of the algorithm).

□

Each element of the set of test cases obtained from a specification *Spec* and a set of test purposes of *Spec* by Algorithm 3.2 (*see* page 59) satisfy all items of Definition 3.12 (*see* page 51). Moreover, this set of test cases is sound and exhaustive for the given set of test purposes. The last statement is proved by T. Jéron in [Jéron, 2004] (*see* Theorem 3.2.2).

3.2.4 Test Generation Tools

The aim of this section is to describe software tools used for automatic test generation based on IOLTS. In this thesis we cite only three of these works, namely, TorX, TGV and Autolink. However, there exist many other tools and techniques which can be used for testing reactive systems. For example, TVeda ([Phalippou and Groz, 1990], [Phalippou, 1994], [Clatin et al., 1995]) and TestComposer ([Kerbrat et al., 1999], [Kerbrat and Ober, 1999]).

3.2.4.1 TorX

TorX [Belinfante et al., 1999] is a tool combining ioco-based test generation (namely, the Tretmans approach explained in Subsection 3.2.3.2, page 55) together with test execution. It was developed in the University of Twente, the Netherlands. The TorX tool allows *on-the-fly testing* for LOTOS [ISO/IEC, 1988]

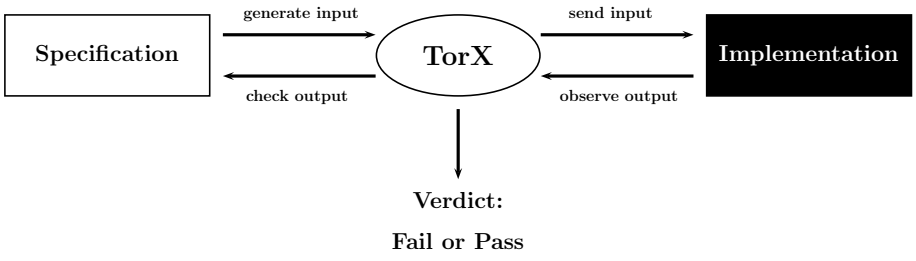


Figure 3.17: Architecture of TorX.

and PROMELA [Leue and Holzmann, 1999] specifications. This means that instead of deriving a complete test case, TorX generates (using a given specification)

an input action, and executes this action on an implementation under test (it is important to notice that before sending an input action to the implementation, TorX encode it into the format acceptable by the implementation). Then, it compares a decoded response (output or blocking) received from the implementation with specified ones, and depending on the result of this comparison, TorX either continues its computation, or stops by generating the *Pass* or *Fail* verdict. Figure 3.17 shows the architecture of the TorX tool that was explained above.

3.2.4.2 TGV

TGV (Test Generator with Verification technology) [Fernandez et al., 1996], [Jéron and Morel, 1999], [Morel, 2000], [Jard and Jéron, 2002], [Jéron, 2004] is a tool for automatic ioco-based test generation developed in collaboration by IRISA/INRIA Rennes, France and Verimag, Grenoble, France.

The architecture of TGV is shown on Figure 3.18 (*see* page 65). The TGV tool

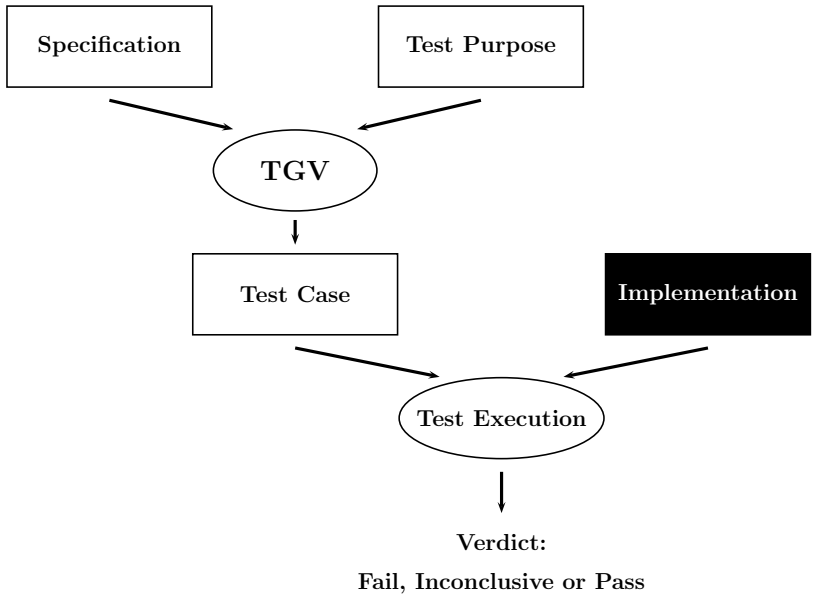


Figure 3.18: Architecture of TGV.

takes as its input a formal specification (expressed in LOTOS [ISO/IEC, 1988], SDL [ITU-T, 1994], UML [Fowler and Scott, 2000], or IF [Bozga et al., 2002] description languages) and a test purpose used as test selection criterion and formalized by an automata. Then, it automatically generates (*see* Algorithm 3.2 given in Subsection 3.2.3.3, page 59) TTCN [ISO/IEC/JTC1/SC21, 1992] test cases. The main characteristic of TGV is the use of *on-the-fly* test generation

technique [Fernandez et al., 1996] (*i.e.* the state space of the specification is not completely stored). This technique allows to avoid the combinatorial state explosion problem which is often the problem in generation of test cases for large-size reactive systems. Finally, test cases derived by TGV are executed on a real system under test, and a test verdict is obtained. Notice that in test generation where selection of test cases is based on test purposes, there are possibly three (and not two) kinds of verdict. The additional verdict is called *Inconclusive*. It means that during test execution no errors were detected but a given test purpose was not achieved.

The detailed information on the TGV tool and case studies where this tool was used, could be found in [Morel, 2000], [Jard and Jéron, 2002], [Jéron, 2004] and [Fernandez et al., 1997], [Jard et al., 2000], [du Bousquet et al., 2000] respectively.

3.2.4.3 Autolink

Autolink [Koch et al., 1998], [Schmitt et al., 1998], [Telelogic, 1998] (as its predecessor SaMsTaG [Grabowski, 1994], [Grabowski et al., 1993]) is a tool for automatic test derivation, where the selection mechanism is based on test purposes. This tool is developed in collaboration by the University of Lüebeck, Germany and the Swedish company Telelogic.

The Autolink tool takes as its inputs (1) a formal specification expressed in SDL, and (2) a test purpose formalized by a MSC [ITU-T, 1996] (Message Sequence Chart) that describes the *complete* sequence of specified input/output actions (and not incomplete or abstract one as it is done in TGV) to be tested. Then, it automatically generates a TTCN [ISO/IEC/JTC1/SC21, 1992] test case by state exploration simulating both SDL specification and MSC test purpose. If during the state exploration we received an event which

- violates the test purpose, but is valid according to the specification, then the Autolink tool creates the TTCN *Inconclusive* verdict,

- violates both test purpose and specification, then the Autolink tool creates the TTCN *Fail* verdict,

Otherwise, *i.e.* when the state exploration is not applicable, the Autolink stops by producing a resulting TTCN test case.

The two principal critics which can be given for Autolink are: (1) the test generation method used by Autolink is not based on well-studied conformance relations such as ioco or ioconf, and (2) state explosion problem which appears during the state exploration of large size reactive systems.

3.2.5 Conclusion

The test generation methods presented in this section are based on the hypothesis that the operational semantics of reactive systems, which are expressed in high-level description languages (*e.g.* LOTOS, SDL or UML), can be modeled by (input-output) labeled transition systems ((IO)LTS). The reactive systems often manipulate complex data structure. Moreover, they can exchange their data using input and output actions. However, the underlying model of (IO)LTS does not allow to explicitly describe the data of these systems. Therefore, in order to model a specification of such reactive system with (IO)LTS, it is necessary to enumerate values of each datum used by this system. This enumeration often leads to the combinatorial states explosion, moreover, in the case when a datum of the reactive system has an infinite domain, the enumeration is impossible. The possible solutions for this problem are briefly discussed in the next section.

3.3 Symbolic Test Generation Techniques

The purpose of this section is to present a brief description of the existing works in the test generation that use symbolic techniques. In this thesis we cite only some of these works. Nevertheless, there exist many tools and techniques that relate to the our work described in the two next parts of the thesis.

3.3.1 Symbolic Testing for LOTOS

M.-C. Gaudel, P.R. James and G. Lestiennes in their works (*see* [Gaudel and James, 1999], [Lestiennes and Gaudel, 2002]) proposed an approach that combines the test generation for algebraic specifications and the symbolic simulation for LOTOS [ISO/IEC, 1988].

The symbolic simulation for LOTOS was studied by E.H. Eertink in his thesis [Eertink, 1994], and implemented in the SMILE [Eertink, 1993] tool that can be used for the test generation. The algorithm proposed by E.H. Eertink first takes a LOTOS process and computes a set of symbolic transitions that can be executed initially. Then, this algorithm is applied recursively to the rest of the process that was obtained after execution one of the previously computed symbolic transitions. This technique is based on constraints propagation.

In 1999, M.-C. Gaudel and P.R. James published the paper [Gaudel and James, 1999], where they suggest to combine symbolic simulation (*see* the previous paragraph) with testing techniques that are based on abstract data types, where tests are selected according to one among several *selection hypotheses* which are chosen depending on (1) some knowledge about an implementation, (2) some coverage criteria of a specification, and (3) ultimately cost considerations. In

this paper the authors emphasized that the test generation must be based not only on the symbolic simulation (a kind of reachability analysis), but also on the backward propagation of constraints (coreachability analysis). This work was followed by another paper [Lestiennes and Gaudel, 2002], where G. Lestiennes and M.-C. Gaudel studies the same approach for the test derivation applied to Input-Output Transition Systems with data. In this work the authors use ioco as the conformance relation. Their work is currently not implemented. However, at the end of the paper [Lestiennes and Gaudel, 2002] the authors mentioned that they plan to automate the proposed method using existing test generation and constraint propagation tools.

3.3.2 Agatha

Agatha [Lugato et al., 2002] is the automatic test generator developed in CEA (Commissariat à l'Energie Atomique), France. This tool supports three phases described below.

The first phase consists in transforming an Estelle [ISO/TC97/SC21, 1997], SDL [ITU-T, 1994] or UML [Fowler and Scott, 2000] specification into a low-level model called EIOLTS (Extended Input-Output Labeled Transition System). EIOLTS are made up of locations and transitions between the locations. Each transition is labeled with: (1) an input action, (2) a guard which is a Boolean expression over the variables and parameters carried by the input action, (3) an output action, and (4) a set of variables' assignments.

The second phase of Agatha is the test generation which is based on the symbolic simulation of the states space using the exhaustive symbolic path coverage. The symbolic simulation computes an execution tree. Each node of this tree is a constraint on the variables characterizing a path of the specification that leads to this node. During the construction of an execution tree, Agatha simplifies constraints using the rewriting engine called Brute [Ishisone and Sawada, 2001]. Moreover, it uses the Omega [Kelly et al., 1995] tool for detecting the constraints inclusion, which is necessary to stop the state exploration.

The final phase of Agatha consists in searching a possible instantiation of the execution tree computed at the previous phase, which permits to choose the test case and to execute it on implementation under test. During this phase Agatha uses either the Omega, or Con'Flex [Rellier and Vardon, 1998] tools.

3.3.3 GATeL

GATeL [Marre and Arnould, 2000] is a tool used in order to derive test cases starting from a specification and a test purpose represented in the synchronous data flow language LUSTRE [Halbwachs et al., 1991]. A specification written in LUSTRE is a set of cyclic equations over variables, which allows to compute new values of the variables using their current values. A test purpose, which describes some important property to check, is an invariant or characterization of reachable states of some implementation under test. The aim of the test derivation is to search for a sequence that permits to satisfy the test purpose (the length of a test sequence generated by GATeL is bounded). The GATeL test derivation method is based on (1) computation of the product between the specification and the test purpose, and (2) the backward propagation of constraints (also known as coreachability analysis) from the goal state to an initial state of the computed product. GATeL is integrated with the ECLiPSe tool [Cheadle et al., 2003] in order to perform the constraint propagation.

It is important to notice that the GATeL tool generates test sequences form only *deterministic* specifications. Therefore, the conformance relation, that is used by GATeL, is to checks the equality between outputs obtained by the implementation during the test and outputs expected by the specification.

3.3.4 BZ-Testing-Tool

BZ-TT (BZ-Testing-Tool) [Ambert et al., 2002] is an environment for automatic boundary-value *conformance* and *robustness* testing from a formal specification given in either B [Abrial, 1996] or Z [Spivey, 1992] notation. Its commercial version LTG (Leirios Test Generator) is based on the same principles as BZ-TT, but, in addition to the B notation, it allows to describe specifications as Statecharts [Harel, 1987] or UML [Fowler and Scott, 2000] diagrams that are widely-used in industry. Notice that BZ-TT, as well as LTG, treats only deterministic specifications, thus, the generated by them test cases are represented as sequences.

The main principles of the BZ-Testing-Tool which takes as an input a system under the form of pre/post predicates over the system's variables with *finite* domains, are described below.

First, BZ-TT computes *boundary* states (*i.e.* states where at least one variable has a value at an extremum of its domain) for a given system. This computation is realized by the use of the set-constraint solver CLIS [Bouquet et al., 2002] that is implemented on the base of the SICStus Prolog [Intelligent Systems Laboratory, 2004].

Second, using a bounded symbolic simulation, BZ-TT takes the system under test from its initial state to some boundary state computed above. More precisely, this step consists in the exploration of the state-space of the given system starting from its initial state and using the *best-first search* algorithm.

Third, in the case when a bounded state is reached, BZ-TT tests all possible operations in this state. Notice that, if values of the state variables do not satisfy pre-conditions of some operation, they are used for the robustness testing.

Finally, we remark that the BZ-Testing-Tool allows to concatenate several test sequences together. Indeed, after the derivation of one test sequence for a boundary state, it is able to return to this boundary or the initial state of the system under test.

3.3.5 Test Generation Tools for Structural Testing

From the discussion above, the reader can notice that the symbolic test generation tools, used in functional (black-box) testing, are based on techniques of constraints propagation/solving. These techniques are also widely used in the structural (white-box) testing. For instance, they are used in the INKA prototype developed by A. Gotlieb et al. ([Gotlieb et al., 1998], [Gotlieb et al., 2000]) which automatically derives test cases for the programs written in a subset of the C language; and in the ATGen tool of C. Meudec [Meudec, 2001] that generates test cases for ADA programs [Barnes, 1984]. The main ideas of such tools are:

(1) to find a path in the control-flow graph of a given program that leads to a given state. Notice that the length of this path is bounded. This allows to unfold cycles of the program under test.

(2) to generate inputs by solving the constraints through the computed above path. This permit to obtain expected outputs.

It is important to remark that such tools are often limited to deterministic programs (here, we mean the *observable* determinism). This means that the outputs of a given program strictly depend on inputs, *i.e.* the tester has a full control on this program. It permits to automatically generate inputs for expected outputs.

3.3.6 Conclusion

In this section we have briefly described some existing methods and tools used for the symbolic derivation of test cases from a formal specification of a given system

under test. All of them use the constraint logic programming, moreover, they are limited to deterministic specifications. These tools allow to generate *precise* test cases under the form of test sequences (or trees) whose length is *bounded*.

The second part of the thesis is devoted to another symbolic test generation method with selection guided by test purposes. This method is quite different from ones mentioned in this section. It is based on a kind of transition system (called Input-Output Symbolic Transition System or IOSTS) that symbolically manipulates the data of a given system. It enables us (1) to avoid the state-space explosion problem, as instead of enumerating values of all variables, we manipulate symbolic states of the system that are characterized by formulas over the systems data, and (2) to generate symbolic test cases that have the form of programs and, therefore, are closer to the reality.

Moreover, our test generation method (more precisely, the mechanism of test case selection) is based on a technique of *abstract interpretation* ([Cousot and Cousot, 1976], [Cousot and Cousot, 1977]) instead of constraint solving. The test cases derived by our method have the form of a graph as we do not need to limit their length. They are less precise than the test cases generated by the tools described in this section. However, they permit to obtain a global view on the "past" and the "future" of a system under test. Finally remark that we are able to treat non-deterministic specifications belonging to the non-trivial subclass defined in Section A.2.4 (*see* page 304).

In the rest of this document we give the detailed description of the IOSTS model, the symbolic test generation method with selection guided by test purposes, and the tool implementing this method.

Part II

Symbolic Test Generation

Introduction

Motivation. In the last chapter of the introductory part of the thesis we saw that in the last decades, testing theory and techniques for automatic test generation of reactive systems have been developed. Some of these techniques are based on the (input-output) labeled transition systems model (*i.e.* (IO)LTS) and efficient on-the-fly algorithms [Tretmans, 1992], [Jéron and Morel, 1999]. There already exist academic (*e.g.* TorX [Belinfante et al., 1999], TGV [Fernandez et al., 1996]) and industrial (*e.g.* Autolink [Telelogic, 1998], TestComposer [Kerbrat and Ober, 1999]) tools that implement these algorithms and produce correct test cases in a formal framework. However, these theories and tools do not explicitly take into account the system data, since the underlying model of (IO)LTS does not allow to do that. Thus, in order to model a specification of reactive systems with (IO)LTS, it is necessary to enumerate the values of each data used by this system. This may result in the classical state-space explosion problem. Moreover, this enumeration also has the effect of obtaining test cases, where all data are instantiated. This contradicts industrial practice, where test cases (written, for instance, in the TTCN [ISO/IEC/JTC1/SC21, 1992] language) are real programs with data (variables, parameters). The generation of such test cases requires new models and techniques which we introduce in this part of the thesis.

Plan of Part II. The rest of this part is organized as follows.

In **Chapter 4** we first present a new model that allows to explicitly introduce the data of a system. This model is called input-output symbolic transition system (IOSTS). Then, we describe the syntax and semantics of IOSTS. Finally, we define some important subclasses of IOSTS, for instance, deterministic and complete IOSTS.

In **Chapter 5** we introduce the product operation and the operation of parallel composition for IOSTS, which are extensions of the analogous operation defined for IOLTS (*see* Definition 3.5 and 3.14, page 38 and 59). The product operation will be used in the symbolic test generation process in order to "intersect" behaviors of a specifications with behaviors of a test

purpose, and the parallel composition will be used during test execution in order to model the communication between an implementation under test and a derived test case.

In **Chapter 6** we build a formal background for conformance testing based on the IOSTS model and the ioc conformance relation that is a weaker variant of the standard ioconf and ioco conformance relations (it does not deal with the problem of blockings because the detection of livelocks in IOSTS is undecidable). Moreover, we define the notion of a correct test case.

In **Chapter 7** we describe our symbolic test generation method where the selection of test cases is guided by test purposes. This method is very close to the approach proposed by T. Jéron and his colleagues from IRISA/INRIA Rennes, France and described in Subsection 3.2.3.3 (*see* page 57). However, some steps of our symbolic test generation method are more complex than those proposed by T. Jéron et al. since they have to take into account the data of a given specification. For instance, the selection of a complete test graph must use a semantics-based analysis in order to select the set of states that are reachable from the initial state of a given IOSTS and from which it is possible to go towards some accepting state of this IOSTS. However, unlike the selection algorithm developed for IOLTS, the issue of computing the exact set of reachable and coreachable states is undecidable for IOSTS. Therefore, the selection algorithm for IOSTS is based on the *over-approximation* of reachable and coreachable sets of states. At the end of this chapter, we prove that our symbolic test generation method derives a set of correct test cases.

The symbolic test generation method was implemented in the STG tool. The description of this tool and a case study of the bounded retransmission protocol are given in **Part III** of the thesis.

The work described in the second and third parts of this thesis is the result of the collaboration with Vlad Rusu and Thierry Jéron, who have established the initial theoretical framework for symbolic test generation [Rusu et al., 2000], as well as Duncan Clarke, Bertrand Jannet and François-Xavier Ponscarme.

Chapter 4

Model: Input-Output Symbolic Transition Systems

In this chapter we introduce a model of reactive systems that we use for conformance testing. This model is called Input-Output Symbolic Transition System (IOSTS). IOSTS is an extended version of IOLTS defined in Subsection 3.2.2.1 of Chapter 3 (see page 41). It explicitly includes data of the reactive systems, and symbolically manipulates with them. At the beginning of the chapter we present an example of IOSTS which helps to understand the model and its semantics at an intuitive level. Then, we define the IOSTS formalism and discusses its syntax and semantics. Finally, we introduce some subclasses of IOSTS used in symbolic test generation. The work contained in this chapter is an extended version of one presented in [Rusu et al., 2000].

4.1 Running Example

This section gives an intuitive explanation of IOSTS using the example of a coffee machine depicted in Figure 4.1. This example is the running example for the current part of the thesis.

Syntax. The IOSTS $\mathcal{S}$ (*see* Figure 4.1) is made up of *locations*, *e.g. Begin*, *Idle*, *Pay*, where *Begin* is the *initial location*, and *transitions*. The transitions are labeled with *actions*, *guards* and *assignments*. For example, the transition with origin *Idle* and destination *Pay* has the guard $(pCoinValue > 0)$, the input action *Coin?* carrying the data *pCoinValue* from the environment, and the assignment $vPaid := vPaid + pCoinValue$. The set of actions is partitioned into three disjoint subsets of *input*, *output* and *internal* actions. The input/output actions interact

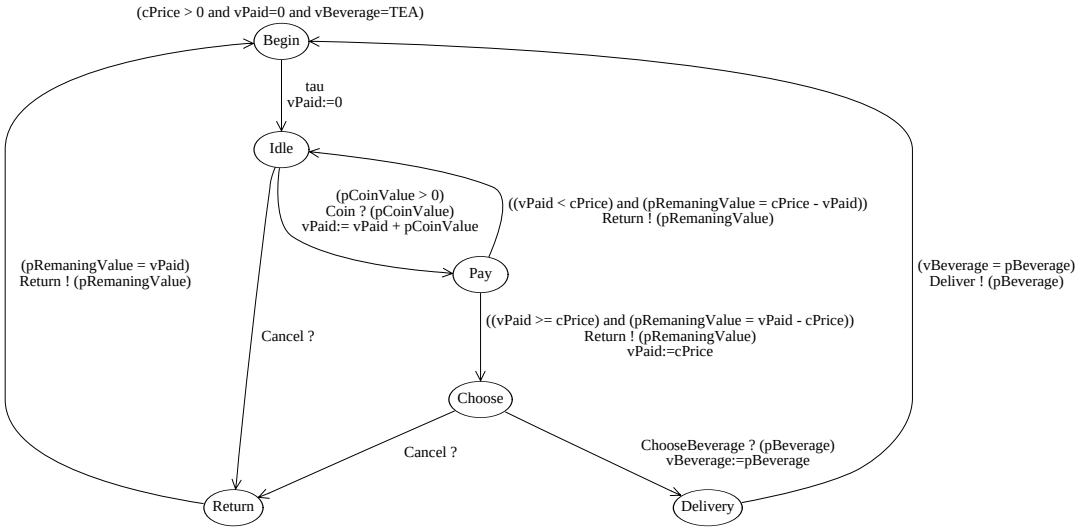


Figure 4.1: IOSTS $\mathcal{S}$ (a coffee machine).

with the environment and may carry data from/to it, while internal actions are used for internal computations. By convention the names of input (*resp.* output) actions end with "?" (*resp.* "!"). The IOSTS in Figure 4.1 has three inputs: *Coin?*, *ChooseBeverage?*, *Cancel?*, two outputs: *Deliver!*, *Return!*, and one internal action: *tau*. It operates with symbolic data consisting of the *variables*: *vPaid*, *vBeverage*, *symbolic constants*: *cPrice*, and *parameters*: *pCoinValue*, *pRemaningValue*, *pBeverage*. Intuitively, *variables* are data to compute with, *symbolic constants* are data which cannot be changed during the computation, and *parameters* are data to communicate with the environment. The scope of a parameter is only the transition labeled by action which carries that parameter. Thus, if the value of the parameter should be used in later computations, it should be memorized through an assignment to a variable. For instance, the value of the parameter *pBeverage* is saved in the variable *vBeverage* using the assignment $vBeverage := pBeverage$ of the transition leading to the *Delivery* location. Then this value, which was memorized in *vBeverage*, is used in the guard $(vBeverage = pBeverage)$ of the transition outgoing from *Delivery*.

Semantics. The coffee machine, represented as the IOSTS in Figure 4.1, starts in the location *Begin* with some values of the symbolic constant *cPrice* and the variables *vPaid*, *vBeverage* satisfying the initial condition $(cPrice > 0 \wedge vPaid = 0 \wedge vBeverage = \text{TEA})$, that is, the price of any beverage dispensed by the machine is strictly positive, and variables *vPaid* and *vBeverage* are respectively

equal to zero and to TEA. Then, it fires the transition labeled by the internal action *tau*, assigns the variable *vPaid* to 0, which memorizes the amount already paid, and reaches the location *Idle*. Next, the machine expects a coin, denoted by the *Coin?* input action that carries in *pCoinValue* the value of the inserted coin. When the coin is inserted the variable *vPaid* is increased by *pCoinValue*, and the machine moves to the *Pay* location. If payment is not enough *i.e.* $vPaid < cPrice$, the machine moves back to the *Idle* location and returns (through the *Return!(pRemaningValue)* output action) the difference between the paid amount and the cost of a beverage, *i.e.* $cPrice - vPaid$. Otherwise, the machine moves to the *Choose* location and returns in *pRemaningValue* the difference between *vPaid* and *cPrice*. In the *Choose* location, the machine waits for the choice of a beverage (tea or coffee), then delivers the beverage, and moves back to the *Begin* location. Note that in the locations *Idle* and *Choose*, the *Cancel* button can be pressed, in which case the machine returns the amount already paid and moves back to the initial location.

4.2 Syntax of IOSTS

At the beginning of this section we introduce types which are used to define a set of typed data. Then, we present the notion of data domain and defines well-typed expressions on the data. Finally, we give the formal definition of Input-Output Symbolic Transition Systems (IOSTS).

Types, Data, Domains and Well-Typed Expressions. Let $\mathfrak{T} = \{\mathsf{t}_1, \ldots, \mathsf{t}_{|\mathfrak{T}|}\}$ be a finite *set of data types* which consists of basic types, *e.g.* Boolean, natural, integer, enumerated types, and complex types, *e.g.* arrays, records, queues. Also let $D = \{d_1, \ldots, d_{|D|}\}$ be a finite *set of typed data*, where the type of each datum from D belongs to $\mathfrak{T}$. We denote by:

- $\mathsf{DOM}(\mathsf{t})$ – the *domain of the type* $\mathsf{t} \in \mathfrak{T}$ in which data of type t take their values. We assume that for each $\mathsf{t} \in \mathfrak{T}$, $\mathsf{DOM}(\mathsf{t})$ is not empty.
- $\mathsf{type}(d) \in \mathfrak{T}$ – the *type* of the element $d \in D$.
- $\mathsf{DOM}(d) = \mathsf{DOM}(\mathsf{type}(d))$ – the *domain of the datum* $d \in D$ in which d of type $\mathsf{type}(d)$ takes its values.
- $\mathsf{DOM}(D) = \prod_{d \in D} \mathsf{DOM}(d)$ – the *data domain* in which the data D take their values.

Example 4.1 Consider the IOSTS $\mathcal{S}$ depicted in Figure 4.1 (*see* page 78). It has the following set of typed data $D = \{cPrice, vPaid, vBeverage, pCoinValue,$

pRemaningValue, *pBeverage*$\}$, where the data *cPrice*, *vPaid*, *pCoinValue* and *pRemaningValue* have the natural number type (denoted by `nat`) and the data *vBeverage*, *pBeverage* have an enumerated type consisting of two elements TEA and COFFEE. Then, the domain of the set of data D is $\mathsf{DOM}(D) = \mathtt{nat} \times \mathtt{nat} \times \{\text{TEA}, \text{COFFEE}\} \times \mathtt{nat} \times \mathtt{nat} \times \{\text{TEA}, \text{COFFEE}\}$. □

Definition 4.1 (Well-Typed Expression) Let $\mathcal{F}_{\mathfrak{t}}$ be a set of functions such that each function $f_{\mathfrak{t}}$ belonging to $\mathcal{F}_{\mathfrak{t}}$ returns a value of type $\mathfrak{t} \in \mathfrak{T}$. Then, an expression *exp* over the typed data D is a *well-typed expression* of type $\mathfrak{t}$ if

(1) $exp \in \mathsf{DOM}(\mathfrak{t})$, which is a constant in $\mathsf{DOM}(\mathfrak{t})$, or

(2) $exp \in D$ and $\mathsf{type}(exp) = \mathfrak{t}$, or

(3) $exp = f_{\mathfrak{t}}(exp_1, \ldots, exp_n)$, where

  (a) $f_{\mathfrak{t}} \in \mathcal{F}_{\mathfrak{t}}$ is a function such that $f_{\mathfrak{t}} : \mathfrak{t}_1 \times \ldots \times \mathfrak{t}_n \mapsto \mathfrak{t}$, where $\mathfrak{t}_1, \ldots, \mathfrak{t}_n \in \mathfrak{T}$, and

  (b) each exp_i $(i = 1..n)$ is a well-typed expression of type $\mathfrak{t}_i$.

□

Among the types there are the standard types such as natural number (denoted by `nat`), Boolean (denoted by `bool`), *etc.* with their usual interpretation. Among the functions $\mathcal{F}_{\mathfrak{t}}$ $(\mathfrak{t} \in \mathfrak{T})$ there are standard functions like $+, <, \wedge$, *etc.* which are interpreted as usual.

Example 4.2 To illustrate the definition given above, we consider the set of data D from Example 4.1 (*see* page 79), and two sets of functions: $\mathcal{F}_{\mathtt{nat}} = \{+, -, *\}$, $\mathcal{F}_{\mathtt{bool}} = \{<, >, \leq, \geq, =, \neq, \wedge, \vee, \neg, \implies\}$, where each function has its *usual interpretation*. Then,

- the expression *true* is a well-typed expression of type `bool` as *true* belongs to the domain of the Boolean type, *i.e.* $true \in \mathsf{DOM}(\mathtt{bool})$.
- the expression $vPaid + pCoinValue$ is a well-typed expression of type `nat`. Indeed, it consists of the data *vPaid* and *pCoinValue* which have the type `nat` and the operation + returns a value of type `nat`, *i.e.* $+ \in \mathcal{F}_{\mathtt{nat}}$.

□

We denote by:

- $\mathcal{E}_{\mathfrak{t}}(D)$ – the *set of well-typed expressions* of type $\mathfrak{t} \in \mathfrak{T}$ over the set of typed data D.
- $\mathcal{B}(D) = \mathcal{E}_{\texttt{bool}}(D)$ – the *set of Boolean expressions* over the set of typed data D, *i.e.* the set of well-typed expressions of Boolean type (denoted by `bool`).

Definition 4.2 (IOSTS) An *IOSTS* is a tuple $\langle D, \Theta, L, l^0, \Sigma, T \rangle$ where

- $D = V \cup C \cup P$ is a finite set of *typed data* which consists of three mutually disjoint sets of *variables* V, *symbolic constants* C and *parameters* P, *i.e.* $(V \cap C = \emptyset) \wedge (V \cap P = \emptyset) \wedge (C \cap P = \emptyset)$.
- Θ is the *initial condition* belonging to the set of Boolean expression over the set of typed data D, *i.e.* $\Theta \in \mathcal{B}(D)$.
- L is a nonempty, finite *set of locations*.
- $l^0 \in L$ is the *initial location*.
- $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ is a nonempty, finite *alphabet of actions* which consists of three mutually disjoint alphabets of *input actions* $\Sigma^?$, *output actions* $\Sigma^!$ and *internal actions* Σ^τ, *i.e.* $(\Sigma^? \cap \Sigma^! = \emptyset) \wedge (\Sigma^? \cap \Sigma^\tau = \emptyset) \wedge (\Sigma^! \cap \Sigma^\tau = \emptyset)$.

  Each action $a \in \Sigma$ is characterized by its *signature* $\mathsf{sig}(a)$. The signature of a is a tuple of types $\mathsf{sig}(a) = \langle \mathfrak{t}_1, \ldots, \mathfrak{t}_k \rangle \in \mathfrak{T}^k$. The signature of an internal action $a \in \Sigma^\tau$ is the empty tuple $\langle\rangle$.
- T is a finite *set of symbolic transitions*. Each *symbolic transition* $t = \langle l, a, \pi, G, A, l' \rangle \in T$ consists of:
  - a location $l \in L$ called the *origin* of the symbolic transition.
  - an *action* $a \in \Sigma$.
  - a *tuple of parameters* $\pi = \langle p_1, \ldots, p_k \rangle \in P^k (k \in \mathbb{N})$ carried by the action a. This tuple of parameters π corresponds to the signature of the action a (*see* above) meaning that: (1) both tuples $\mathsf{sig}(a)$ and π have the same length, and (2) the type of each parameter p_i $(i = 1..k)$ from the tuple of parameters π is equal to the type $\mathfrak{t}_i$ $(i = 1..k)$ in $\mathsf{sig}(a)$.
  - a *guard* G which is a Boolean expression over the parameters carried by the action a, and variables and symbolic constants of the IOSTS, *i.e.* $G \in \mathcal{B}(V \cup C \cup \pi)$.

- a *set of assignments* A. Each assignment belonging to A has the form $v := A_v$, where $v \in V$ is a variable and $A_v \in \mathcal{E}_{\mathsf{type}(v)}(V \cup C \cup \pi)$ is a well-typed expression of type $\mathsf{type}(v)$ over variables, symbolic constants and the parameters carried by the action a. For *each* variable $v \in V$, the set of assignments A contains *exactly one* assignment.

  The graphical representation of IOSTS (*see* for instance Figure 4.1, page 78) does not show the assignments of the form $v := v$.
- a location $l' \in L$ called the *target* of the symbolic transition.

□

Example 4.3 For better understanding of the IOSTS formalism, we consider the IOSTS $\mathcal{S}$ shown on Figure 4.1 (*see* page 78), which consists of:

– the set of typed data: $D = \underbrace{\{cPrice\}}_{C} \cup \underbrace{\{vPaid, vBeverage\}}_{V} \cup \underbrace{\{pCoinValue, pRemaningValue, pBeverage\}}_{P}$. Note that the sets of symbolic constants C, variables V and parameters P are mutually disjoint;

– the initial condition $\Theta : (cPrice > 0 \wedge vPaid = 0 \wedge vBeverage = \text{TEA})$;

– the set of locations $L = \{Begin, Idle, Pay, Choose, Return, Delivery\}$;

– the initial location $l^0 = Begin \in L$;

– the alphabet of actions $\Sigma = \underbrace{\{Coin, Cancel\}}_{\Sigma^?} \cup \underbrace{\{Return, Deliver\}}_{\Sigma^!} \cup \underbrace{\{tau\}}_{\Sigma^\tau}$. The set of input $\Sigma^?$, output $\Sigma^!$ and internal Σ^τ actions are mutually disjoint.

  Each action belonging to Σ has its signature. For example, action *Coin* has signature $\langle \mathtt{nat} \rangle$, where by `nat` we denoted the natural number type.

– the set of transitions T. For instance, T contains the following transition:

$$t : \quad \langle \underbrace{Idle}_{l}, \underbrace{Coin}_{a}, \underbrace{pCoinValue}_{\pi}, \underbrace{(pCoinValue > 0)}_{G}, \underbrace{\{vPaid := vPaid + pCoinValue; vBeverage := vBeverage\}}_{A}, \underbrace{Pay}_{l'} \rangle$$

Notice that the set of assignments A contains the assignment $vBeverage := vBeverage$, which by convention, is not shown in Figure 4.1 (*see* page 78).

□

4.3 Semantics of IOSTS

This section is divided into two subsections: the first one describes the semantics of the IOSTS formalism, and the second one introduces the important notions of behaviors and traces that will be widely used through the thesis.

4.3.1 IOLTS as the semantics of IOSTS

In this section we formally define the semantics of IOSTS. More precisely, the semantics of an IOSTS $\mathcal{M}$ can be represented by an IOLTS $[[\mathcal{M}]]$ (*see* Definition 3.7 in Chapter 3, page 41), where:

(1) the set of states of $[[\mathcal{M}]]$ is the set of *valued states* of $\mathcal{M}$ (defined in Subsection 4.3.1.2, page 84, and called later "states"),

(2) the alphabet of actions of $[[\mathcal{M}]]$ is the alphabet of *valued actions* of $\mathcal{M}$ (introduced in Subsection 4.3.1.3, page 85), and

(3) the transition relation of $[[\mathcal{M}]]$ is extended to a *global transition relation* of $\mathcal{M}$, which will be defined in Subsection 4.3.1.4 (*see* page 86).

The rest of this section is organized as follows. First, we give several preliminary notations that are used in definitions of states and valued actions of IOSTS. Then, based on the notions of state and valued action we define two transition relations: $\rightarrow_t$ (called local transition relation) and $\rightarrow$ (called global transition relation). Finally, we give the formal definition of the IOSTS semantics.

4.3.1.1 Preliminary Notations and Definitions

Consider a set of typed data D and a datum d belonging to D, then:

(1) a vector of data values $\vartheta \in \mathsf{DOM}(D)$ is called *valuation of data* D, and

(2) a value $\vartheta_d \in \mathsf{DOM}(d)$ of the datum d is called *valuation of datum* d.

Next, we formally define a notion of satisfiability of a Boolean expression by a given valuation of data.

Definition 4.3 (Valuation Satisfies Expression) A valuation $\vartheta \in \mathsf{DOM}(D)$ *satisfies* a Boolean expression $exp \in \mathcal{B}(D)$ (denoted by $\vartheta \models exp$) if the expression $exp(d/\vartheta(d))$, where each datum $d \in D$ is replaced by its value $\vartheta(d)$ from ϑ, evaluates to $true$. □

Example 4.4 To illustrate Definition 4.3, we consider the following set of typed data: $D = \{vPaid, cPrice\}$, where the data $vPaid, cPrice$ are of the natural type (denoted by `nat`), and the following Boolean expression $exp : cPrice \leq (vPaid - 1)$. Then,

- the valuation $\vartheta = \langle vPaid = 5, cPrice = 3\rangle \in \mathsf{DOM}(D)$ satisfies exp as the expression $(3 \leq 5 - 1)$ evaluates to *true*, but
- the valuation $\vartheta = \langle vPaid = 3, cPrice = 5\rangle \in \mathsf{DOM}(D)$ does not satisfy exp as the expression $(5 \leq 3 - 1)$ does not evaluate to *true*.

□

In this thesis we assume that *satisfiability* of all guards decorating symbolic transitions of IOSTS is *decidable*. For example, these guards can be expressed in a *decidable* fragment of the theory of Presburger arithmetic [Presburger, 1929] with uninterpreted functions [Ackermann, 1954]. For more details see paper [Rusu and Zinovieva, 2001].

4.3.1.2 States

We give the definition of a state for an arbitrary IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T\rangle$, where the set of data D consists of three mutually disjoint sets of variables V, symbolic constants C and parameters P. Then, we illustrate this notion with simple examples. Finally, we introduce some new notations used later in this thesis.

Definition 4.4 (State) A *state* s of $\mathcal{M}$ is a pair $\langle l, \vartheta\rangle$, where $l \in L$ is a location and $\vartheta = \langle \vartheta_d\rangle_{d\in(V\cup C)} \in \mathsf{DOM}(V \cup C)$ is a *valuation of the variables and symbolic constants.* □

Example 4.5 The pair $\langle Delivery,\ \langle cPrice = 3, vPaid = 1, vBeverage = \text{TEA}\rangle\rangle$ is a state s of the IOSTS $\mathcal{S}$ depicted in Figure 4.1 (*see* page 78). This pair consists of the location $l = Delivery$ and the valuation $\vartheta = \langle 3, 1, TEA\rangle$ of the symbolic constant *cPrice* and the variables *vPaid* and *vBeverage*, where *cPrice* and *vPaid* are of type `nat` and *vBeverage* is of type $\texttt{enum} = \{\text{TEA}, \text{COFFEE}\}$.

□

Definition 4.5 (Initial State) An *initial state* $s^0 = \langle l^0, \vartheta^0\rangle$ of $\mathcal{M}$ is a state where l^0 is the initial location, and ϑ^0 is a valuation of the variables and symbolic constants that satisfies (*see* Definition 4.3, page 83) the initial condition Θ of $\mathcal{M}$.

□

Example 4.6 Consider the IOSTS $\mathcal{S}$ depicted in Figure 4.1 (*see* page 78). The state $s^0 = \langle Begin, \langle cPrice = 3, vPaid = 0, vBeverage = \text{TEA}\rangle\rangle$ is one of the possible initial states of $\mathcal{S}$, as *Begin* is its initial location, and the values of the symbolic constant *cPrice* and the variables *vPaid*, *vBeverage* satisfy the initial condition $(cPrice > 0 \wedge vPaid = 0 \wedge vBeverage = \text{TEA})$ of $\mathcal{S}$. □

Notations. For a given IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T\rangle$ with set of data $D = V \cup C \cup P$, we denote by:

- $S = \{\langle l, \vartheta\rangle \mid l \in L \ \wedge \ \vartheta \in \mathsf{DOM}(V \cup C)\}$ – the *set of states* of $\mathcal{M}$, and
- $S^0 = \{\langle l^0, \vartheta^0\rangle \mid l^0 \in L \ \wedge \ \vartheta^0 \in \mathsf{DOM}(V \cup C) \ \wedge \ \vartheta^0 \models \Theta\} \subseteq S$ – the *set of initial states* of $\mathcal{M}$.

4.3.1.3 Valued Actions

We introduce the notion of valued action for an IOSTS $\mathcal{M}$ with alphabet of actions Σ. Intuitively, a valued action is a pair consisting of an action $a \in \Sigma$ and a valuation of the parameters carried by a. Formally:

Definition 4.6 (Valued Action) Let a be an action of $\mathcal{M}$ and $\mathsf{sig}(a) = \langle \mathsf{t}_1, \ldots, \mathsf{t}_k\rangle$ be a signature of this action (*see* Definition 4.2, page 81). Then, a pair $\alpha = \langle a, \omega\rangle$, where ω is a vector of values $\langle \omega_1, \ldots, \omega_k\rangle$ such that for all $i = 1..k$, $\omega_i \in \mathsf{DOM}(\mathsf{t}_i)$, is called *valued action* of $\mathcal{M}$. □

Example 4.7 The pair $\alpha = \langle Coin, \langle 5\rangle\rangle$ is an example of valued action of the IOSTS $\mathcal{S}$ shown on Figure 4.1 (*see* page 78), where $a = Coin$ is an input action of $\mathcal{S}$ which has the signature $\mathsf{sig}(a) = \langle \mathtt{nat}\rangle$, and $\omega = \langle 5\rangle$, where $5 \in \mathsf{DOM}(\mathtt{nat})$, is one of the possible valuations of $\mathsf{sig}(a)$.

Another example of valued action of $\mathcal{S}$ is $\alpha = \langle tau, \langle\rangle\rangle$, where $a = tau$ is the internal action and ω is the empty tuple (remember that the signature of any internal action is the empty tuple, *see* Definition 4.2, page 81). □

Notations. We denote by Λ the *set of valued actions*. Λ is partitioned, according to the partitioning of the alphabet of actions Σ (*see* Definition 4.2, page 81), into three mutually disjoint subsets of *valued input actions* $\Lambda^?$, *valued output actions* $\Lambda^!$ and *internal actions* Λ^τ. Note that by abuse of notation, it is possible

to identify Σ^τ and Λ^τ as all internal actions carry an empty tuple of parameters (*see* Definition 4.2, page 81).

4.3.1.4 Local and Global Transition Relations

In this section we introduce the notions of local and global transition relations. They allows us to move from the syntactic level of the IOSTS formalism to the semantic one. The section contains an intuitive example which permits better understanding of these notions. At the end of the section, we give several notations which will be useful in the sequel.

We first introduce local and global transition relations intuitively. Let $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ be an IOSTS with set of states S and set of valued actions Λ, and $t = \langle l, a, \pi, G, A, l' \rangle \in T$ be a symbolic transition of $\mathcal{M}$. Then, the *local transition relation* $\rightarrow_t$ is the set of triples $\langle s, \alpha, s' \rangle$, where:

- $s = \langle l, \vartheta \rangle$ is a state, where $l \in L$ is the origin of the symbolic transition t, and ϑ is a valuation of the variables and symbolic constants of $\mathcal{M}$.
- $\alpha = \langle a, \omega \rangle$ is a valued action, where $a \in \Sigma$ is the action labeling the symbolic transition t, and ω is a *valuation of parameters* π carried by the action a.
- the pair of the valuations ϑ and ω satisfies (*see* Definition 4.3, page 83) the guard G of t, *i.e.* $\langle \vartheta, \omega \rangle \models G$.
- $s' = \langle l', \vartheta' \rangle$ is a state, where $l' \in L$ is the target of t, and ϑ' is the new valuation of the variables and symbolic constants of $\mathcal{M}$ obtained from ϑ and ω by execution of the parallel assignments belonging to A of t.

The union of all these sets of triples is called the *global transition relation* $\rightarrow$ of the IOSTS $\mathcal{M}$.

The formal definitions of the local and global transition relations, *i.e.* $\rightarrow_t$ and $\rightarrow$, are given below.

Definition 4.7 (Local Transition Relation $\rightarrow_t$) Let

- $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ be an IOSTS with set of states S and set of valued actions Λ,
- $t = \langle l, a, \pi, G, A, l' \rangle \in T$ be a symbolic transition of $\mathcal{M}$,
- $\vartheta, \vartheta' \in \mathsf{DOM}(V \cup C)$ be valuations of variables and symbolic constants of $\mathcal{M}$, and
- $\omega \in \mathsf{DOM}(\mathsf{sig}(a)) = \mathsf{DOM}(\pi)$ be a valuation of parameters π carried by action a of the given symbolic transition t

Then, the *local transition relation* $\rightarrow_t \subseteq S \times \Lambda \times S$ of $\mathcal{M}$ is the smallest relation satisfying the following inference rule:

$$\frac{\langle \vartheta, \omega \rangle \models G \qquad \forall\ v \in V.\ [\vartheta'_v = A_v(d/\vartheta_d,\ p/\omega_p)];\quad \forall\ c \in C.\ [\vartheta'_c = \vartheta_c]}{\langle l, \vartheta \rangle \xrightarrow{\langle a, \omega \rangle}_t \langle l', \vartheta' \rangle}$$

where $A_v(d/\vartheta_d,\ p/\omega_p)$ is the expression obtained by replacing in the expression A_v each variable or symbolic constant $d \in (V \cup C)$ by its value ϑ_d from the valuation ϑ, and each parameter $p \in \pi$ by its value ω_p from the valuation ω. □

Definition 4.8 (Global Transition Relation $\rightarrow$) The *global transition relation* $\rightarrow$ of $\mathcal{M}$ is the union of all sets $\rightarrow_t$ associated with each symbolic transitions $t \in T$, where T is the set of symbolic transitions of the IOSTS$\mathcal{M}$, *i.e.*

$$\rightarrow \ \triangleq \ \bigcup_{t \in T} \rightarrow_t \tag{4.1}$$

□

Example 4.8 To illustrate the notion of local transition relation $\rightarrow_t$ we consider:

(1) the IOSTS $\mathcal{S}$ shown on Figure 4.1 (*see* page 4.1) with the sets of variables $V = \{vPaid, vBeverage\}$, symbolic constants $C = \{cPrice\}$ and parameters $P = \{pCoinValue, pRemaningValue, pBeverage\}$, and

(2) the symbolic transition:

$$t = \langle \underbrace{Pay}_{l}, \underbrace{Return}_{a}, \underbrace{\langle pRemaningValue \rangle}_{\pi}, G, A, \underbrace{Choose}_{l'} \rangle$$

of $\mathcal{S}$, where $G : (vPaid \geq cPrice\ \wedge\ pRemaningValue = vPaid - cPrice)$ is the guard of t and $A : \{vPaid := cPrice, vBeverage := vBeverage\}$ is the set of assignments of t (note that the assignment $vBeverage := vBeverage$ is not shown in Figure 4.1 (*see* page 78) due to the agreement about the graphical representation of IOSTS made in Definition 4.2 (*see* page 81)).

Then, the local transition relation $\rightarrow_t$ corresponding to the symbolic transition t is a set of triples $\langle \underbrace{\langle Pay,\ \vartheta \rangle}_{s}, \underbrace{\langle Return,\ \omega \rangle}_{\alpha}, \underbrace{\langle Choose,\ \vartheta' \rangle}_{s'} \rangle$ such that the pair $\langle \vartheta, \omega \rangle$ satisfies the guard G of t, and ϑ' is a "new" valuation of variables and symbolic constants obtained from the assignments A of t.

An example of a triple belonging to $\rightarrow_t$ is shown below:

$$\langle\ \langle Pay,\ \langle 3, 5, \text{TEA}\rangle\rangle,\ \langle Return,\ \langle 2\rangle\rangle,\ \langle Choose,\ \langle 3, 3, \text{TEA}\rangle\ \rangle$$

where

- $\vartheta = \langle 3, 5, \text{TEA}\rangle$ is the "old" valuation of the variables and symbolic constants of $\mathcal{S}$, where $\vartheta(cPrice) = 3$, $\vartheta(vPaid) = 5$, and $\vartheta(vBeverage) = \text{TEA}$;
- $\omega = \langle 2\rangle$ is the valuation of the signature of the output action $a = Return$. Notice that the valuation ω gives a value to each parameter of the symbolic transition t, *i.e.* $\omega(pRemaningValue) = 2$;
- the pair of valuations $\langle \vartheta,\ \omega\rangle$ satisfies the guard G : $(vPaid \geq cPrice\ \wedge\ pRemaningValue = vPaid - cPrice)$ of t; and
- the "new" valuation $\vartheta' = \langle 3, 3, \text{TEA}\rangle$, where $\vartheta'(cPrice) = 3$, $\vartheta'(vPaid) = 3$, $\vartheta'(vBeverage) = \text{TEA}$, is obtained by executing the assignments of t which are: $vPaid := cPrice$ and $vBeverage := vBeverage$.

□

In the definition given below we recall (with minor modifications) some notations that was already introduced for (IO)LTS (*see* Definition 3.4 of the introductory part, page 35).

Definition 4.9 For an IOSTS $\mathcal{M}$ with set of states S and set of valued actions Λ, we introduce the following notations:

(1) $s \xrightarrow{\alpha} s'$ for the triple $\langle s, \alpha, s'\rangle$ belonging to the transition relation $\rightarrow$, where $s, s' \in S$ and $\alpha \in \Lambda$,

(2) $s \xrightarrow{\alpha}$ for $\exists\ s' \in S\ .\ [s \xrightarrow{\alpha} s']$, where $s \in S$, $\alpha \in \Lambda$,

(3) $s \overset{\alpha}{\not\rightarrow}$ for $\nexists\ s' \in S\ .\ [s \xrightarrow{\alpha} s']$, where $s \in S$, $\alpha \in \Lambda$, and

(4) $s \xrightarrow{\eta} s'$ for $\exists\ s_1, \ldots, s_n \in S\ .\ [s = s_1 \xrightarrow{\alpha_1} s_2 \ldots s_{n-1} \xrightarrow{\alpha_{n-1}} s = s_n]$, where for all $i = 1..n-1$: $\alpha_i \in \Lambda$, and $\eta \in (\Lambda)^*$.

□

4.3.1.5 Formal Definition of the IOSTS Semantics

In this section we give a definition of the semantics of an arbitrary IOSTS which can be expressed in terms of IOLTS as follows:

Definition 4.10 (Semantics of IOSTS) For an IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$, the semantics of $\mathcal{M}$ is defined by an IOLTS $[[\mathcal{M}]] = \langle S, S^0, \Lambda, \rightarrow \rangle$, where

- S is the *set of states* of $\mathcal{M}$,
- $S^0 \subseteq S$ is the *set of initial states* of $\mathcal{M}$,
- $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$ is the set of valued actions of $\mathcal{M}$, and
- $\rightarrow$ is the *global transition relation* of $\mathcal{M}$.

□

4.3.2 Behaviors, Sequences and Traces

In this section we introduce the notions of behaviors, sequences and traces for an arbitrary IOSTS $\mathcal{M}$ with a set of states S, a set of initial states $S^0 \subseteq S$, and set of valued actions $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$. These notions allow us to reason formally about the IOSTS $\mathcal{M}$.

Definition 4.11 (Behavior) A *behavior* $\beta^\rightarrow$ of $\mathcal{M}$ is a sequence of states and valued actions starting from an initial state s^0 and following the transition relation, *i.e.*

$$\beta^\rightarrow : \quad s^0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots s_{n-1} \xrightarrow{\alpha_n} s_n \tag{4.2}$$

where

- $\rightarrow$ is the global transition relation of $\mathcal{M}$ (*see* Definition 4.8, page 87),
- $n \in \mathbb{N}$ is the *length* of $\beta^\rightarrow$,
- $s^0 \in S^0$, and
- for all $i = 1..n$: $s_i \in S$, $\alpha_i \in \Lambda$.

□

Next, we define the notion of sequence. Informally, a sequence η of valued actions $\alpha_1 \dots \alpha_n$ of $\mathcal{M}$ is a sequence obtained from a behavior $\beta^{\rightarrow} : s^0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \dots s_{n-1} \xrightarrow{\alpha_n} s_n$ of $\mathcal{M}$ by dropping the states $s^0, \dots, s_n$. Formally:

Definition 4.12 (Set of Sequences) The *set of sequences* of $\mathcal{M}$ is the set:

$$Sequences(\mathcal{M}) \quad \triangleq \quad \{\eta \in (\Lambda)^* \mid \exists\ s^0 \in S^0, s \in S.\ [s^0 \xrightarrow{\eta} s]\} \tag{4.3}$$

□

Example 4.9 For the IOSTS $\mathcal{S}$ depicted in Figure 4.1 (*see* page 78) the following sequence:

$$\begin{aligned} \eta : \quad & \langle tau, \langle\rangle\rangle, \\ & \langle Coin, \langle pCoinValue = 2\rangle\rangle, \\ & \langle Return, \langle pRemaningValue = 1\rangle\rangle, \\ & \langle ChooseBeverage, \langle pBeverage = \text{COFFEE}\rangle\rangle, \\ & \langle Deliver, \langle pBeverage = \text{COFFEE}\rangle\rangle \end{aligned}$$

is a sequence of valued actions belonging to $Sequences(\mathcal{S})$. □

Next, we introduce a *trace relation* and based on this newly defined relation give a formal definition of a *trace* of the IOSTS $\mathcal{M}$.

Definition 4.13 (Trace Relation $\Rightarrow$) The *trace relation* $\Rightarrow \subseteq S \times (\Lambda^? \cup \Lambda^!)^* \times S$ of $\mathcal{M}$ is obtained from the global transition relation $\rightarrow$ defined above (*see* Definition 4.8, page 87) by dropping internal actions:

- $s \overset{\varepsilon}{\Rightarrow} s' \quad \triangleq \quad (s = s') \vee (\exists s_1, \dots, s_n \in S.\ [s = s_1 \xrightarrow{\tau_1} s_2 \dots s_{n-1} \xrightarrow{\tau_n} s_n = s'])$, where for all $i = 1..n$: $\tau_i \in \Lambda^\tau$, and ε is the empty sequence.
- $s \overset{\alpha}{\Rightarrow} s' \quad \triangleq \quad \exists s_1, s_2 \in S.\ [s \overset{\varepsilon}{\Rightarrow} s_1 \xrightarrow{\alpha} s_2 \overset{\varepsilon}{\Rightarrow} s']$, where $\alpha \in (\Lambda^? \cup \Lambda^!)$.
- $s \overset{\sigma}{\Rightarrow} s' \quad \triangleq \quad \exists s_1, \dots, s_n \in S.\ [s = s_1 \overset{\alpha_1}{\Rightarrow} s_2 \dots s_{n-1} \overset{\alpha_{n-1}}{\Rightarrow} s_n = s']$, where for all $i = 1..n-1$: $\alpha_i \in (\Lambda^? \cup \Lambda^!)$, and $\sigma \in (\Lambda^? \cup \Lambda^!)^*$.

□

Definition 4.14 (Set of Traces) The *set of traces* of $\mathcal{M}$ is defined as follows:

$$Traces(\mathcal{M}) \quad \triangleq \quad \{\sigma \in (\Lambda^? \cup \Lambda^!)^* \mid \exists\ s^0 \in S^0, s \in S\ .\ [s^0 \overset{\sigma}{\Rightarrow} s]\} \tag{4.4}$$

□

Example 4.10 The following sequence of valued input/output actions:

$$\begin{aligned}\sigma : \quad & \langle Coin, \langle pCoinValue = 2 \rangle \rangle, \\ & \langle Return, \langle pRemaningValue = 1 \rangle \rangle, \\ & \langle ChooseBeverage, \langle pBeverage = \text{COFFEE} \rangle \rangle, \\ & \langle Deliver, \langle pBeverage = \text{COFFEE} \rangle \rangle\end{aligned}$$

represents one of the possible traces of the coffee machine depicted in Figure 4.1 (*see* page 78).

□

Finally, we introduce the notion of reachable state, and define a set of states in which the IOSTS $\mathcal{M}$ can be after executing a given trace of $\mathcal{M}$.

Definition 4.15 (Reachable State) A state $s \in S$ is *reachable* in $\mathcal{M}$ if there exists a sequence $\eta \in Sequences(\mathcal{M})$ that ends in s, *i.e.* $\exists\, \eta \in Sequences(\mathcal{M}),\ s^0 \in S^0\ .\ [s^0 \xrightarrow{\sigma} s]$. □

Example 4.11 To illustrate the definition given above we consider the IOSTS $\mathcal{S}$ shown in Figure 4.1 (*see* page 78). Then, the state $s = \langle Begin, \langle vPaid = 1, vBeverage = \text{COFFEE} \rangle \rangle$ of $\mathcal{S}$ is reachable, as there exists the sequence η (*see* Example 4.9) which ends in this state s. Note that the state s is also reachable by the trace η shown in Example 4.10. □

Definition 4.16 ($\mathcal{M}$ after σ) The set of states in which an IOSTS $\mathcal{M}$ can be after the trace $\sigma \in Traces(\mathcal{M})$ is defined as:

$$(\mathcal{M} \text{ after } \sigma) \quad \triangleq \quad \{s \in S \mid \exists s^0 \in S^0.\ s^0 \stackrel{\sigma}{\Rightarrow} s\} \tag{4.5}$$

□

4.4 Subclasses of IOSTS

In the current section we define some subclasses of IOSTS which are used in the symbolic test generation method presented later in the thesis.

4.4.1 Instantiated, Initialized and Deterministic IOSTS

In this subsection we introduce the subclasses of instantiated, initialized and deterministic IOSTS, and illustrates them with simple examples. The main result of the subsection, which will be exploit lated in the thesis, is: if an IOSTS is initialized and deterministic, then each instance of this IOSTS moves to *exactly one state* after each trace.

Definition 4.17 (Instantiated IOSTS) An IOSTS $\mathcal{M}$ is *instantiated* if its set of symbolic constants C is empty. □

Definition 4.18 (Instance of IOSTS) Let $\mathcal{M}$ be an IOSTS with the set of symbolic constants C and $\varsigma \in \mathsf{DOM}(C)$ be a valuation of these symbolic constants. Then, the *instance* $\mathcal{M}(\varsigma)$ of $\mathcal{M}$ and ς is the IOSTS obtained by replacing each symbolic constant $c \in C$ by its value $\varsigma(c)$. Moreover, the set of symbolic constants of $\mathcal{M}(\varsigma)$ is empty, *i.e.* $C_{\mathcal{M}(\varsigma)} = \emptyset$. □

Observation 4.1 All instances of an IOSTS $\mathcal{M}$ are instantiated. □

Example 4.12 Consider the IOSTS $\mathcal{S}$ depicted in Figure 4.1 (*see* page 78), which has one symbolic constant *cPrice* of type `nat`, *i.e.* $C = \{cPrice\}$. Let us also consider one of the possible valuations of C, for example, $\varsigma = \langle 3 \rangle$, where $\varsigma(cPrice) = 3$. Then, by replacing in $\mathcal{S}$ each instance of the symbolic constant *cPrice* by its value 3, we obtain the instance $\mathcal{S}(\varsigma)$ of the IOSTS $\mathcal{S}$. □

Definition 4.19 (Initialized IOSTS) An IOSTS $\mathcal{M}$ with the initial condition Θ and the set of variables V is *initialized* if for every instance $\mathcal{M}(\varsigma)$, there exists *at most one* valuation of its variables which satisfies Θ (*see* Definition 4.3, page 83). □

Example 4.13 Let us consider an IOSTS $\mathcal{A}$ with the following initial condition $\Theta : (cPrice > 0) \wedge (vPaid = cPrice)$, where *cPrice* with $\mathsf{type}(cPrice) = \langle \mathtt{nat} \rangle$ is a symbolic constant of $\mathcal{A}$ and *vPaid* with $\mathsf{type}(vPaid) = \langle \mathtt{nat} \rangle$ is a variable of $\mathcal{A}$. This IOSTS is initialized, as

- for all instances $\mathcal{A}(\varsigma)$, where $\varsigma \in \mathsf{DOM}(\mathtt{nat}) \setminus \{0\}$, there exists exactly exactly one valuation of variable *vPaid* which satisfies Θ. For example, for $\mathcal{A}(\underbrace{\langle 3 \rangle}_{\varsigma})$, the only one valuation satisfying $(3 > 0) \wedge (vPaid = 3)$ is $\vartheta = \langle 3 \rangle$, where $\vartheta(vPaid) = 3$.

– for the instance $\mathcal{A}(\underbrace{\langle 0 \rangle}_{\varsigma})$, there are no valuations of the variable $vPaid$ which satisfies $(0 > 0) \wedge (vPaid = 0)$.

Next, if the initial condition of $\mathcal{A}$ is the following $\Theta : (cPrice > 0) \wedge (vPaid \geq cPrice)$, then $\mathcal{A}$ is not initialized, as, for example, for $\mathcal{A}(\underbrace{\langle 3 \rangle}_{\varsigma})$, there exist at least valuations of variable $vPaid$ satisfying $(3 > 0) \wedge (vPaid \geq 3)$, *e.g.* $\vartheta(vPaid) = 3$, $\vartheta(vPaid) = 42$. □

Next, we introduce the subclass of deterministic IOSTS. The definition of a deterministic IOSTS is inspired from that of a deterministic IOLTS (*see* the item (8) of Definition 3.4, page 35). Intuitively, an IOSTS is deterministic if it does not contain internal actions, and from each state at most one transition can be fired. Formally:

Definition 4.20 (Deterministic IOSTS) An IOSTS $\mathcal{M}$ with a set of states S and a set of valued actions $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$ is *deterministic* if:

(1) $\Lambda^\tau = \emptyset$, and

(2) for all states $s \in S$ and valued actions $\alpha \in \Lambda^? \cup \Lambda^!$, $card(\{s' \in S \mid s \xrightarrow{\alpha} s'\}) \leq 1$, where $card(K)$ denotes the *cardinality* of a set K.

□

We illustrate the definition of a deterministic IOSTS with an example.

Example 4.14 In Figure 4.2 (*see* page 94) we depict a simple coffee machine. This machine accepts a strictly positive amount of money from the user, and delivers either (1) only coffee if the user has paid less or equal to three units, or (2) coffee with milk in the case when the paid amount was three or more than three units. Note that if the user paid exactly three units, the outcome from the coffee machine is non-deterministic. Thus, due to the violation of the second item of Definition 4.20, the coffee machine shown on Figure 4.2 (*see* page 94) is non-deterministic. □

It is not hard to see that deterministic IOSTS are somewhat different from deterministic IOLTS as they can have several initial states (the definition above does not contain any restriction on the number of initial states of IOSTS). However, the testing theory prohibits to use a tester that can choose its initial state as the

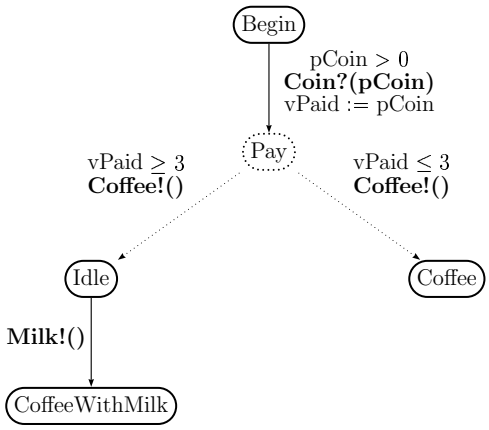


Figure 4.2: Non-deterministic coffee machine.

result of testing should not depend on internal choice of the tester. Therefore, the tester must be modeled with an IOSTS that is not only deterministic but also initialized. The next lemma shows that if an IOSTS is deterministic and initialized, then each instance of this IOSTS can move to *exactly one state* after each trace. Formally:

Theorem 4.1 ($(\mathcal{M}(\varsigma)$ after $\sigma)$ is a Singleton) Let $\mathcal{M}$ be an initialized and deterministic IOSTS with initial condition Θ, set of variables V and set of symbolic constants C. Then, for each trace σ of each instance $\mathcal{M}(\varsigma)$ of the IOSTS $\mathcal{M}$, where $\varsigma \in \mathsf{DOM}(C)$, the set of states $(\mathcal{M}(\varsigma)$ after $\sigma)$ is a singleton. □

Proof Consider an instance of the given IOSTS $\mathcal{M}$, *i.e.* $\mathcal{M}(\varsigma)$. Due to Definition 4.18 (*see* page 92) we know that $\mathcal{M}(\varsigma)$ does not have any symbolic constant, *i.e.* $C_{\mathcal{M}(\varsigma)} = \emptyset$. Also, as we know that $\mathcal{M}$ is initialized (*see* the hypotheses of the theorem), then there exists *at most one* valuation ϑ of variables V which satisfies the initial condition Θ of $\mathcal{M}(\varsigma)$ (*see* Definition 4.19, page 92). Next, we consider two cases:

(1) The valuation ϑ satisfying Θ does not exist. In this case, $\mathcal{M}(\varsigma)$ has an empty set of traces.

(2) The valuation ϑ satisfying Θ exists, and it is unique. Then, $\mathit{Traces}(\mathcal{M}(\varsigma)) \neq \emptyset$, and there exists *exactly one* initial state in $\mathcal{M}(\varsigma)$.

Consider an arbitrary trace $\sigma \in \mathit{Traces}(\mathcal{M}(\varsigma))$ starting in the unique initial state of $\mathcal{M}(\varsigma)$. By the hypothesis of the lemma, we know that $\mathcal{M}$ is deterministic. Thus, $\mathcal{M}(\varsigma)$ is deterministic as well. Hence, each time during the execution of σ on $\mathcal{M}(\varsigma)$, it is possible to fire *only one* symbolic transition

of $\mathcal{M}(\varsigma)$ (*see* Definition 4.20, page 93). Therefore, after the trace σ the IOSTS $\mathcal{M}(\varsigma)$ moves to exactly one state, *i.e.* $(\mathcal{M}(\varsigma)$ after $\sigma)$ is a singleton.

Q.E.D.

4.4.2 Complete and Input-Complete IOSTS

In this subsection we introduce the subclasses of complete and input-complete IOSTS. The hypotheses about completeness and input-completeness of IOSTS involved in testing will be needed in the symbolic test generation and test execution processes.

For an IOSTS representing either a test case or an implementation under test, we need to assume that is always accepts (does not block) all possible inputs. Such IOSTS is called *input-complete* and defined as follows:

Definition 4.21 (Input-Complete IOSTS) An IOSTS $\mathcal{M}$ is *input-complete* if for each state $s \in S$ and each valued input action $\alpha \in \Lambda^?$, $\mathcal{M}$ can move to some state $s' \in S$ following its trace relation $\Rightarrow$, *i.e.* possibly after a sequence of internal actions:

$$\forall\ s \in S, \alpha \in \Lambda^? \ \exists\ s' \in S\ .\ [s \stackrel{\alpha}{\Rightarrow} s']$$

□

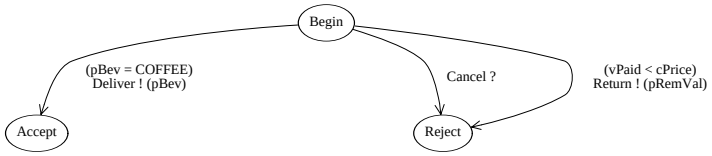
Notice that the definition given above is semantic. Nevertheless, it can be achieved using sufficient syntactical conditions. We illustrate the definition of an input-complete IOSTS with an example.

Example 4.15 Consider two IOSTS $\mathcal{S}_1$ and $\mathcal{S}_2$ depicted in Figure 4.3, where a transition $l \stackrel{*}{\rightarrow} l'$ is an abbreviation for the complement set of all transitions leaving l. Then,

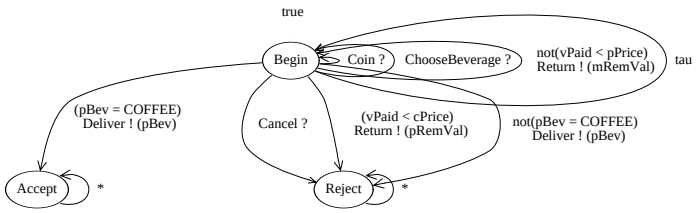
(1) $\mathcal{S}_1$ (*see* Figure 4.3(a)) is not input-complete, as for example, we cannot move from the state $s = \langle Begin, \vartheta \rangle$ to another state s' by taking the valued input action $\alpha = \langle Coin, \omega \rangle$,

(2) $\mathcal{S}_2$ (*see* Figure 4.3(b)) is input-complete due to the fact that from each state of $\mathcal{S}_2$ we can execute each valued input action of $\mathcal{S}_2$.

□



(a) $\mathcal{S}_1$



(b) $\mathcal{S}_2$

Figure 4.3: An example illustrating Definitions 4.21 and 4.22.

By analogy with the test generation approach proposed by T. Jéron (*see* Section 3.2.3.3, page 57), our symbolic test generation method that will be described in Chapter 7 of this document, uses a test purpose as the test selection mechanism. In other words, the aim of test purposes is to mark the traces of a given specification that should be tested without the global modification of all traces of this specification. For this, each IOSTS representing a test purpose must be *complete with respect to* another IOSTS representing a specification. Formally complete a IOSTS is defined as follows:

Definition 4.22 (Complete IOSTS with respect to another IOSTS) Let $\mathcal{M}_1 = \langle D_1, \Theta_1, L_1, l_1^0, \Sigma_1, T_1 \rangle$ and $\mathcal{M}_2 = \langle D_2, \Theta_2, L_2, l_2^0, \Sigma_2, T_2 \rangle$ be two IOSTS. Then, $\mathcal{M}_2$ is *complete with respect to* $\mathcal{M}_1$ if:

(1) the set of variables of $\mathcal{M}_2$ is disjoint with the set of symbolic constant of $\mathcal{M}_1$, *i.e.* $V_2 \cap C_1 = \emptyset$,

(2) $\mathcal{M}_1$ and $\mathcal{M}_2$ have the same alphabets of actions, *i.e.* $\Sigma_1 = \Sigma_2$,

(3) the initial condition Θ_2 does not contain constraints on the variables and symbolic constants of $\mathcal{M}_1$,

(4) for all locations $l \in L_2$ and all actions $a \in (\Sigma_1 = \Sigma_2)$, there exist transitions:

$$t^1 = \langle l, a, \pi, G^1, A^1, l^1 \rangle \in T_2, \ldots, t^k = \langle l, a, \pi, G^k, A^k, l^k \rangle \in T_2 \qquad (4.6)$$

such that $G^1 \vee \ldots \vee G^k$ evaluates to $true$ for all possible valuations of variables, symbolic constants and parameters. Remember that in this thesis we consider IOSTS whose guards are in a decidable theory for satisfiability.

□

Example 4.16 As in Example 4.15, we consider the two IOSTS $\mathcal{S}_1$ and $\mathcal{S}_2$ shown in Figure 4.3. Then,

(1) $\mathcal{S}_1$ (*see* Figure 4.3(a)) is not complete with respect to the IOSTS $\mathcal{S}$ shown in Figure 4.1 (*see* page 78) as, for example, there are no symbolic transitions labeled with action *Coin*, which leaves the location *Begin*.

(2) $\mathcal{S}_2$ (*see* Figure 4.3(b)) is complete with respect to the IOSTS $\mathcal{S}$ in Figure 4.1 (*see* page 78) due to the facts:

(a) $\mathcal{S}_2$ satisfies Item (1) of Definition 4.22, as $\mathcal{S}_2$ does not have any variables, thus the intersection of the set of variables of $\mathcal{S}_2$ with the set of symbolic constants of $\mathcal{S}$ is empty,

(b) $\mathcal{S}_2$ satisfies Item (2) of Definition 4.22, as $\Sigma_{\mathcal{S}_2} = \Sigma_{\mathcal{S}} = \{Coin, Beverage, Cancel, Return, Deliver, tau\}$,

(c) $\mathcal{S}_2$ satisfies Item (3) of Definition 4.22, as its initial condition $\Theta_2 = true$ does not contain the constraints on variables and symbolic constants of $\mathcal{S}$, and

(d) $\mathcal{S}_2$ satisfies Item (4) of Definition 4.22 (*see* Figure 4.3(b)). For example, there exists the symbolic transition leaving the location *Begin* labeled with action *Coin* and guarded with $true$. Another example is: there exist two symbolic transitions leaving the location *Begin* labeled with the action *Return* and guarded with $(vPaid < cPrice)$ and $\neg(vPaid < cPrice)$ respectively, where $(vPaid < cPrice) \vee \neg(vPaid < cPrice)$ evaluates to $true$.

□

Chapter 5

Operations with IOSTS

In this chapter we introduce several operations on IOSTS which are used in the test generation technique and in the process of the test execution. They are called product and parallel composition. The product operation is the main operation in the test generation as it allows to "intersect" the behaviors of a specification and its test purpose, and therefore, to select the part of the specification for which a test case has to be generated. The operation of parallel composition is used in the test execution on a system under test as this operation allows to model an interaction between two processes represented by IOSTS. Finally, we present several relationships concerning the sets of traces of IOSTS obtained after the product or parallel composition operations.

5.1 Parallel Composition

In this section we introduce the operation of parallel composition between two IOSTS with synchronization on their common input and output actions. This operation is inspired from the parallel composition between two processes modeled by either LTS or IOLTS (*see* Definition 3.5, page 38). As usual, the operation of parallel composition is used to model the execution of a test case on a black-box implementation (*see* Chapter 6, page 137).

Before giving the formal definition of the parallel composition, we describe the conditions under whose the parallel execution of two IOSTS is possible. Intuitively, two IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for the operation of parallel composition if and only if:

- $\mathcal{M}_1$ and $\mathcal{M}_2$ do not share variables, symbolic constants and parameters.
- The alphabet of input (*resp.* output) actions of $\mathcal{M}_1$ is equal to the alphabet of output (*resp.* input) actions of $\mathcal{M}_2$, and the alphabets of internal action

of $\mathcal{M}_1$ and $\mathcal{M}_2$ are disjoint. Moreover, the common actions must have the same signature in both IOSTS.

Definition 5.1 (Compatible For Parallel Composition) Two IOSTS $\mathcal{M}_1 = \langle D_1,\ \Theta_1, L_1, l_1^0, \Sigma_1, T_1\rangle$ and $\mathcal{M}_2 = \langle D_2, \Theta_2, L_2, l_2^0, \Sigma_2, T_2\rangle$ with actions alphabets $\Sigma_1 = \Sigma_1^? \cup \Sigma_1^! \cup \Sigma_1^\tau$ and $\Sigma_2 = \Sigma_2^? \cup \Sigma_2^! \cup \Sigma_2^\tau$ are *compatible for parallel composition* if and only if

(1) $\mathcal{M}_1$ and $\mathcal{M}_2$ have disjoint sets of data, *i.e.* $D_1 \cap D_2 = \emptyset$; and

(2) (a) the alphabet of input (*resp.* output) actions of $\mathcal{M}_1$ is the same as the alphabet of output (*resp.* input) actions of $\mathcal{M}_2$, *i.e.* $\Sigma_1^? = \Sigma_2^!$ and $\Sigma_1^! = \Sigma_2^?$,

(b) the alphabets of internal actions of $\mathcal{M}_1$ and $\mathcal{M}_2$ are disjoint, *i.e.* $\Sigma_1^\tau \cap \Sigma_2^\tau = \emptyset$, and

(c) each *common* action a, which is either input for $\mathcal{M}_1$ and output for $\mathcal{M}_2$, or output for $\mathcal{M}_1$ and input for $\mathcal{M}_2$, has the same signature in both IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$, *i.e.* $\mathsf{sig}_1(a) = \mathsf{sig}_2(a)$. More precisely, the length of the tuples of types $\mathsf{sig}_1(a) = \langle \mathfrak{t}_1^1, \ldots, \mathfrak{t}_1^k\rangle$ and $\mathsf{sig}_2(a) = \langle \mathfrak{t}_2^1, \ldots, \mathfrak{t}_2^k\rangle$ is the same, and each type $\mathfrak{t}_i^1$ of $\mathsf{sig}_1(a)$ corresponds to the type $\mathfrak{t}_i^2$ of $\mathsf{sig}_2(a)$, *i.e.* $\forall\ i = 1..k\ .\ [\mathfrak{t}_1^i = \mathfrak{t}_2^i]$.

□

Next, we formulate the operation of parallel composition and explain it on a simple example.

Definition 5.2 (Parallel Composition $||$) The *parallel composition* between two IOSTS $\mathcal{M}_1 = \langle D_1, \Theta_1,\ L_1, l_1^0, \Sigma_1, T_1\rangle$ and $\mathcal{M}_2 = \langle D_2, \Theta_2, L_2, l_2^0, \Sigma_2, T_2\rangle$ which are compatible for parallel composition is an IOSTS $\mathcal{M}_1||\mathcal{M}_2 = \langle D, \Theta, L, l^0, \Sigma, T\rangle$, where

– $D = V \cup C \cup P$, where $V = V_1 \cup V_2$, $C = C_1 \cup C_2$ and $P = P_1$.

The set of parameters P of $\mathcal{M}_1 \,||\, \mathcal{M}_2$ is equal to the set of parameters P_1 of $\mathcal{M}_1$ as during the synchronization of the symbolic transitions each parameter of $\mathcal{M}_2$ is replaced with the corresponding parameter of $\mathcal{M}_1$ (*see* Rule (5.3));

– $\Theta = \Theta_1 \wedge \Theta_2$;

– $L = L_1 \times L_2$;

– $l^0 = \langle l_1^0, l_2^0 \rangle \in L$ is the initial location;

– $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ is the alphabet of actions, where $\Sigma^? = \emptyset$, $\Sigma^! = \Sigma_1^! \cup \Sigma_2^!$, and $\Sigma^\tau = \Sigma_1^\tau \cup \Sigma_2^\tau$.

The alphabet of input actions $\Sigma^?$ is empty due to the following convention: the result of synchronization between two symbolic transitions labeled respectively with input and output actions is always a symbolic transition labeled with an *output* action (*see* below);

– the set of symbolic transitions T is constructed from T_1 and T_2 as follows:

- for each symbolic transition $t_1 \in T_1$ with origin $l_1 \in L_1$ and destination $l_1' \in L_1$, which is labeled with an internal action $a \in \Sigma_1^\tau$; and a location $l_2 \in L_2$, there exists a symbolic transition $t \in T$ with origin $\langle l_1, l_2 \rangle$ and destination $\langle l_1', l_2 \rangle$, which is labeled with the same action a and defined by the following inference rule:

$$\frac{\begin{array}{c}\langle l_1, a, \pi, G_1, A_1, l_1' \rangle \in T_1 \\ l_2 \in L_2 \\ a \in \Sigma_1^\tau \end{array}}{\langle \langle l_1, l_2 \rangle, a, \pi, G_1, A_1 \cup (\bigcup_{v \in V_2} (v := v)), \langle l_1', l_2 \rangle \rangle \in T} \tag{5.1}$$

where in order to obtain a symbolic transition with well-formed set of assignments (*i.e.* each variable of $\mathcal{M}_1 \parallel \mathcal{M}_2$ must be assigned), we assign each variable of $\mathcal{M}_2$ with itself (*i.e.* $\bigcup_{v \in V_2} (v := v)$). This solution is reasonable as variables of $\mathcal{M}_2$ should not change their values while $\mathcal{M}_1$ performs its internal action.

Similarly, a symbolic transition $t \in T$ is obtained from a symbolic transition $t_2 \in T_2$ labeled with $a \in \Sigma_2^\tau$ and a location $l_1 \in L_1$, by applying the following rule:

$$\frac{\begin{array}{c}\langle l_2, a, \pi, G_2, A_2, l_2' \rangle \in T_2 \\ l_1 \in L_1 \\ a \in \Sigma_2^\tau \end{array}}{\langle \langle l_1, l_2 \rangle, a, \pi, G_2, (\bigcup_{v \in V_1} (v := v)) \cup A_2, \langle l_1, l_2' \rangle \rangle \in T} \tag{5.2}$$

- for two symbolic transitions $t_1 \in T_1$ and $t_2 \in T_2$, which are labeled with a *common* action a that belongs to the set $(\Sigma_1^? \cup \Sigma_1^!) = (\Sigma_2^! \cup \Sigma_2^?)$, a new symbolic transition $t \in T$ is constructed using the inference rule:

$$\frac{\begin{array}{c}\langle l_1, a, \pi_1, G_1, A_1, l_1' \rangle \in T_1 \\ \langle l_2, a, \pi_2, G_2, A_2, l_2' \rangle \in T_2 \\ a \in (\Sigma_1^? \cup \Sigma_1^!) = (\Sigma_2^! \cup \Sigma_2^?); \end{array}}{\langle \langle l_1, l_2 \rangle, a, \pi_1, G_1 \wedge G_2[\pi_2/\pi_1], A_1 \cup A_2[\pi_2/\pi_1], \langle l_1', l_2' \rangle \rangle \in T} \tag{5.3}$$

where $G_2[\pi_2/\pi_1]$ (*resp.* $A_2[\pi_2/\pi_1]$) is the guard (*resp.* the set of assignments) of the symbolic transition t_2 in which each parameter $p_2^i \in \pi_2$ carried by the action a of t_2 is replaced by the corresponding parameter $p_1^i \in \pi_1$ carried by the action a of t_1.

□

Example 5.1 In order to explain the operation of parallel composition we consider two IOSTS $\mathcal{A}_1$ and $\mathcal{A}_2$ depicted in Figure 5.1 (*see* page 103).

The IOSTS $\mathcal{A}_1$ (*see* Figure 5.1(a)) either:

- receives the input action *EnterNumber* from its environment, increases the integer value entered to $\mathcal{A}_1$ through this action, and returns some integer value which is less or equal to the computed value through the output action *Result*, or
- returns some negative integer value to the environment through the output action *Result*.

The IOSTS $\mathcal{A}_2$ (*see* Figure 5.1(b)) returns to the environment an integer value which is negative or equal to zero, through the output action *EnterNumber* and memorizes this value into the variable y. Then, it decreases the value of the variable y by one while performing the internal action τ_2, and waits from the environment the input action *Result* carrying some integer value which is greater or equal to the value of y.

The IOSTS $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible for the operation of parallel composition (*see* Definition 5.1, page 100) as:

(1) $\mathcal{A}_1$ and $\mathcal{A}_2$ do not have common variables, symbolic constants and parameters: $\underbrace{\{x\}}_{V_1} \cap \underbrace{\{y\}}_{V_2} = \emptyset$, $\underbrace{\{c_1\}}_{C_1} \cap \underbrace{\{c_2\}}_{C_2} = \emptyset$ and $\underbrace{\{p_1\}}_{P_1} \cap \underbrace{\{p_2\}}_{P_2} = \emptyset$.

(2) (a) The alphabet of input (*resp.* output) actions of $\mathcal{A}_1$ is equal to the alphabet of output (*resp.* input) actions of $\mathcal{A}_2$, *i.e.* $\Sigma_1^? = \Sigma_2^! = \{EnterNumber\}$, $\Sigma_1^! = \Sigma_2^? = \{Result\}$; and the alphabets of internal actions of $\mathcal{A}_1$ and $\mathcal{A}_2$ are disjoint, *i.e.* $\Sigma_1^\tau \cap \Sigma_2^\tau = \{\tau_1\} \cap \{\tau_2\} = \emptyset$. And,

(b) The signatures of actions shared by $\mathcal{A}_1$ and $\mathcal{A}_2$ are the same. For instance, the input action *EnterNumber* of $\mathcal{A}_1$ and the output action *EnterNumber* of $\mathcal{A}_2$ have the same signature $\mathsf{sig}_1(EnterNumber) = \mathsf{sig}_2(EnterNumber) = \langle \mathtt{int} \rangle$, where `int` denotes the integer type.

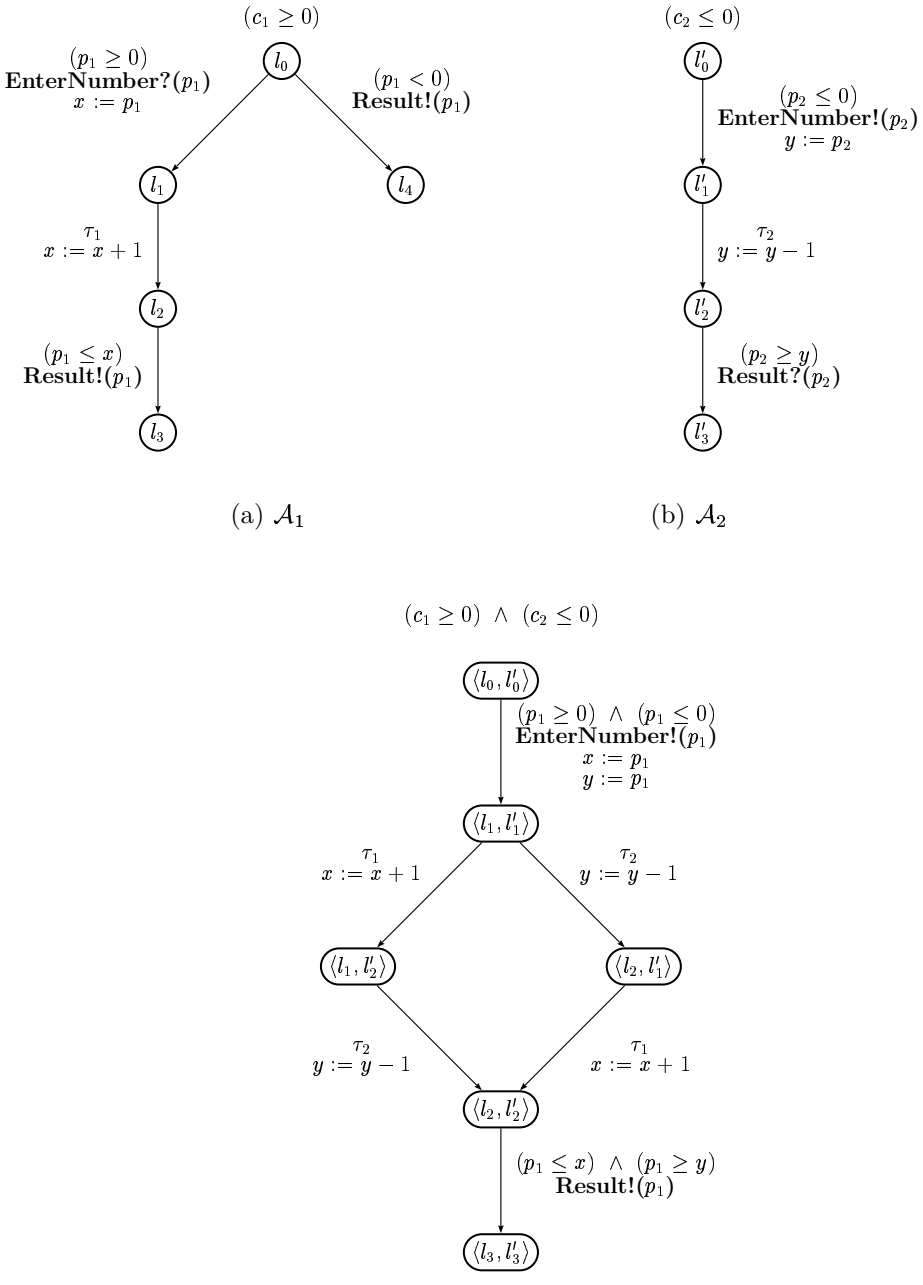


Figure 5.1: The parallel composition between two IOSTS $\mathcal{A}_1$ and $\mathcal{A}_2$.

On Figure 5.1(c) we depicted the IOSTS $\mathcal{A}_1 \parallel \mathcal{A}_2$ obtained from $\mathcal{A}_1$ and $\mathcal{A}_2$ by performing the parallel composition as follows:

(1) the set of data of $\mathcal{A}_1 \parallel \mathcal{A}_2$ consists of the following variables and symbolic constants: $\underbrace{\{x, y\}}_{V} \cup \underbrace{\{c_1, c_2\}}_{C} \cup \underbrace{\{p_1\}}_{P}$;

(2) the initial condition of $\mathcal{A}_1 \parallel \mathcal{A}_2$ is the conjunction of the initial conditions of $\mathcal{A}_1$ and $\mathcal{A}_2$, *i.e.* $\Theta : (c_1 \geq 0) \wedge (c_2 \leq 0)$ (*see* Figure 5.1(c));

(3) the initial location of $\mathcal{A}_1 \parallel \mathcal{A}_2$ is the pair $\langle l_0, l'_0 \rangle$ (*see* Figure 5.1(c)), where l_0 is the initial location of $\mathcal{A}_1$ and l'_0 is the initial location of $\mathcal{A}_2$;

(4) the alphabets of input, output and internal actions are constructed from the alphabets of input, output and internal actions of $\mathcal{A}_1$ or $\mathcal{A}_2$ as following: $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$, where

- $\Sigma^? = \emptyset$,
- $\Sigma^! = \underbrace{\{\mathit{EnterNumber}, \mathit{Result}\}}_{\Sigma^!_1 \cup \Sigma^!_2}$, and
- $\Sigma^\tau = \underbrace{\{\tau_1, \tau_2\}}_{\Sigma^\tau_1 \cup \Sigma^\tau_2}$

(5) finally, we explain how to construct the set of symbolic transitions of $\mathcal{A}_1 \parallel \mathcal{A}_2$.

(a) First, note that the IOSTS $\mathcal{A}_1$ staying in its initial location l_0 can perform either the action *EnterNumber* or the action *Result*, but $\mathcal{A}_2$ can execute only the action *EnterNumber* from its initial location l'_0. By Definition 5.2 (*see* page 100) we know that the symbolic transitions of $\mathcal{A}_1$ and $\mathcal{A}_2$ can be fired synchronously if they are labeled with the same actions. This means that at the first step we synchronize the symbolic transitions of $\mathcal{A}_1$ and $\mathcal{A}_2$ outgoing from their initial locations and labeled by the same action *EnterNumber*. After synchronization (*see* Rule (5.3) of Definition 5.2, page 100), we obtain the following symbolic transition of the IOSTS $\mathcal{A}_1 \parallel \mathcal{A}_2$: $t = \langle \langle l_0, l'_0 \rangle, \mathit{EnterNumber}, \pi, G, A, \langle l_1, l'_1 \rangle \rangle$, where

- *EnterNumber* is the input action carrying the tuple of parameters $\pi = \langle p_1 \rangle$;
- the guard G is the conjunction of the guard $(p_1 \geq 0)$ of $\mathcal{A}_1$ and the guard $(p_2 \leq 0)$ of $\mathcal{A}_2$, where each parameter p_2 is replaced by the parameter p_1, *i.e.* $G : (p_1 \geq 0) \wedge (p_1 \leq 0)$;

– the assignments A is the union of the set of assignments $\{x := p_1\}$ of $\mathcal{A}_1$ and the set of assignments $\{y := p_2\}$ of $\mathcal{A}_2$, where each parameter p_2 is replaced by the parameter p_1, *i.e.* $A : \{x := p_1, y := p_1\}$

The result of this synchronization is shown on Figure 5.1(c) (*see* the symbolic transition of $\mathcal{A}_1 \parallel \mathcal{A}_2$ outgoing from the initial location $\langle l_0, l'_0 \rangle$).

(b) After the synchronization explained above $\mathcal{A}_1$ and $\mathcal{A}_2$ move to the locations l_1 and l'_1 respectively. In these locations $\mathcal{A}_1$ and $\mathcal{A}_2$ can fire only their private internal actions τ_1 and τ_2 respectively. Thus, using Rules (5.1) and (5.2) of Definition 5.2 (*see* page 100) the composed IOSTS $\mathcal{A}_1 \parallel \mathcal{A}_2$ contains two following symbolic transitions:

– $\langle \langle l_1, l'_1 \rangle, \tau_1, \langle \rangle, true, \{x := x + 1, y := y\}, \langle l_2, l'_1 \rangle \rangle$
– $\langle \langle l_1, l'_1 \rangle, \tau_2, \langle \rangle, true, \{x := x, y := y - 1\}, \langle l_1, l'_2 \rangle \rangle$

(*see* Figure 5.1(c)).

(c) Continuing in the same way, we obtain the whole set of symbolic transitions of the IOSTS $\mathcal{A}_1 \parallel \mathcal{A}_2$ (*see* Figure 5.1(c)).

□

5.1.1 Traces of the Parallel Composition

In this section we consider two IOSTS $\mathcal{M}_1 = \langle D_1, \Theta_1, L_1, l_1^0, \Sigma_1, T_1 \rangle$ and $\mathcal{M}_2 = \langle D_2, \Theta_2, L_2, l_2^0, \Sigma_2, T_2 \rangle$ with sets of states S_1, S_2, and sets of valued actions Λ_1, Λ_2. We assume that $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for the parallel composition (*see* Definition 5.1, page 100). Then, we study the relationship between the set of traces of $\mathcal{M}_1 \parallel \mathcal{M}_2$ and the sets of traces of $\mathcal{M}_1$ and $\mathcal{M}_2$. The aim is to prove the equality between the traces of $\mathcal{M}_1 \parallel \mathcal{M}_2$ and the intersection of the traces of $\mathcal{M}_1$ and $\mathcal{M}_2$, *i.e.*

$$Traces(\mathcal{M}_1 \parallel \mathcal{M}_2) = Traces(\mathcal{M}_1) \cap Traces(\mathcal{M}_2) \tag{5.4}$$

This equality is given as the theorem at the end of this section (*see* page 112). It is important to emphasize that the result which we want to obtain (*i.e.* Equality (5.4)) is formulated at the semantics level of IOSTS while the definition of the parallel composition (*see* page 100) is given at the syntax level. Thus, in order to move from the syntax to the semantics level of IOSTS, we first study how the transition relations of the given IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ relate with the transition relations of IOSTS obtained by from $\mathcal{M}_1$ and $\mathcal{M}_2$ by the parallel composition (*see* Lemmas 5.1 and 5.2 on page 106 and 107). The obtained relations

allow us to show that for each sequence of valued input/output actions (*i.e.* for $\sigma \in (\Lambda_1 \setminus \Lambda_1^\tau)^* = (\Lambda_2 \setminus \Lambda_2^\tau)^*$), the IOSTS $\mathcal{M}_1 \parallel \mathcal{M}_2$ has a visible behavior corresponding to this sequence σ *if and only if* both IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ have visible behaviors corresponding to σ. This statement is formulated as Theorem 5.1 (*see* page 109) that naturally imply Equality (5.4) which we have to prove.

Before giving the detailed proof of Equality (5.4), we make an observation about the form of the states of the IOSTS obtained after the parallel composition between $\mathcal{M}_1$ and $\mathcal{M}_2$. Indeed, by definition of the parallel composition (*see* Definition 5.2, page 100) and the semantics of IOSTS (*see* Definition 4.10, page 89), all states of $\mathcal{M}_1 \parallel \mathcal{M}_2$ are of the form: $\langle\langle l_1, l_2\rangle, \langle \vartheta_1, \vartheta_2\rangle\rangle$, where $\langle l_1, \vartheta_1\rangle$ is a state of $\mathcal{M}_1$ and $\langle l_2, \vartheta_2\rangle$ is a state of $\mathcal{M}_2$. This observation is implicitly used in all lemmas and theorems of the current section.

Next, we prove that the IOSTS $\mathcal{M}_1 \parallel \mathcal{M}_2$ moves from the state $\langle\langle l_1, l_2\rangle, \langle \vartheta_1, \vartheta_2\rangle\rangle$ to the state $\langle\langle l_1', l_2\rangle, \langle \vartheta_1', \vartheta_2\rangle\rangle$ by executing an internal action τ belonging to Λ_1^τ *if and only if* the IOSTS $\mathcal{M}_1$ moves from $\langle l_1, \vartheta_1\rangle$ to $\langle l_1', \vartheta_1'\rangle$ by executing the same internal action and the IOSTS $\mathcal{M}_2$ does not change its state $\langle l_2, \vartheta_2\rangle$. Formally:

Lemma 5.1 Let

(1) $t_1 = \langle l_1, \tau, \langle\rangle, G_1, A_1, l_1'\rangle \in T_1$ be a symbolic transition of $\mathcal{M}_1$, which is labeled with an internal action $\tau \in \Sigma_1^\tau$,

(2) $l_2 \in L_2$ be a location of $\mathcal{M}_2$, and

(3) $t = \langle\langle l_1, l_2\rangle, \tau, \langle\rangle, G_1, A_1 \cup (\bigcup_{v \in V_2}(v := v)), \langle l_1', l_2\rangle\rangle$ be the symbolic transition of $\mathcal{M}_1 \parallel \mathcal{M}_2$ obtained from t_1 and l_2 by Rule (5.1) of Definition 5.2 (*see* page 101).

Then, for each internal action $\tau \in \Lambda_1^\tau$ (remember that Σ^τ and Λ^τ can be identified, see Section 4.3.1.3, page 85), the triple $\langle\ \langle\langle l_1, l_2\rangle, \langle \vartheta_1, \vartheta_2\rangle\rangle, \tau, \langle\langle l_1', l_2\rangle, \langle \vartheta_1', \vartheta_2\rangle\rangle\ \rangle$ belongs to $\rightarrow_t$ *if and only if* the triple $\langle\ \langle l_1, \vartheta_1\rangle, \tau, \langle l_1', \vartheta_1'\rangle\ \rangle$ belongs to $\rightarrow_{t_1}$. □

Proof

(1) As we know:

(a) $(V_1 \cup C_1) \cap (V_2 \cup C_2) = \emptyset$ (*see* Definition 5.1, page 100), and

(b) the guard G_1 of the symbolic transition t_1 of $\mathcal{M}_1$ is also the guard of the symbolic transition t of $\mathcal{M}_1 \parallel \mathcal{M}_2$ (*see* the formulation of the lemma),

then we have that G_1 does not contain constraints on the variables and symbolic constants of $\mathcal{M}_2$.

Thus, the valuation $\vartheta = \langle \vartheta_1, \vartheta_2 \rangle \in \mathsf{DOM}(V \cup C)$ of $\mathcal{M}_1 \parallel \mathcal{M}_2$ satisfies the guard G_1 of the symbolic transition t *if and only if* the valuation $\vartheta_1 \in \mathsf{DOM}(V_1 \cup C_1)$ of $\mathcal{M}_1$ satisfies the guard G_1 of the symbolic transition t_1.

Note that we did not mention anything about the valuation of parameters due to the fact that the symbolic transitions t_1 and t are labeled with the internal action τ which, by Definition 4.2 (*see* page 4.2), does not carry any parameters.

(2) From Rule (5.1)of Definition 5.2 (*see* page 101) we have that:

the "new" valuation $\vartheta' \in \mathsf{DOM}(V \cup C)$ of $\mathcal{M}_1 \parallel \mathcal{M}_2$ is obtained from the "old" one $\vartheta \in \mathsf{DOM}(V \cup C)$ by executing in parallel the assignments belonging to $A_1 \cup (\bigcup_{v \in V_2} (v := v))$ of the symbolic transition t *if and only if*

- the "new" valuation $\vartheta'_1 \in \mathsf{DOM}(V_1 \cup C_1)$ of $\mathcal{M}_1$ is obtained from the "old" one $\vartheta_1 \in \mathsf{DOM}(V_1 \cup C_1)$ by executing in parallel the assignments belonging to A_1 of the symbolic transition t_1, and
- the valuation $\vartheta_2 \in \mathsf{DOM}(V_2 \cup C_2)$ of $\mathcal{M}_2$ does not change.

The lemma follows directly from the items (1) and (2) above. **Q.E.D.**

Remark 5.1 By switching $\mathcal{M}_1$ and $\mathcal{M}_2$ in Lemma 5.1 we obtain a symmetrical result. □

By analogy with the previous lemma, we prove that the IOSTS $\mathcal{M}_1 \parallel \mathcal{M}_2$ moves from the state $\langle \langle l_1, l_2 \rangle, \langle \vartheta_1, \vartheta_2 \rangle \rangle$ to the state $\langle \langle l'_1, l'_2 \rangle, \langle \vartheta'_1, \vartheta'_2 \rangle \rangle$ by executing a valued input/output action α if and only if the IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ move from $\langle l_1, \vartheta_1 \rangle$ (*resp.* $\langle l_2, \vartheta_2 \rangle$) to $\langle l'_1, \vartheta'_1 \rangle$ (*resp.* $\langle l'_2, \vartheta'_2 \rangle$) by executing the same valued action α.

Lemma 5.2 Let

(1) $a \in (\Sigma^?_1 \cup \Sigma^!_1) = (\Sigma^?_2 \cup \Sigma^!_2) = \Sigma^!$ be a common action of $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_1 \parallel \mathcal{M}_2$,

(2) $t_1 = \langle l_1, a, \pi_1, G_1, A_1, l'_1 \rangle \in T_1$ be a symbolic transition of $\mathcal{M}_1$,

(3) $t_2 = \langle l_2, a, \pi_2, G_2, A_2, l'_2 \rangle \in T_2$ be a symbolic transition of $\mathcal{M}_2$, and

(4) $t = \langle \langle l_1, l_2 \rangle, a, \pi_1, G_1 \wedge G_2[\pi_2/\pi_1], A_1 \cup A_2[\pi_2/\pi_1], \langle l'_1, l_2 \rangle \rangle$ be the symbolic transition of $\mathcal{M}_1 \parallel \mathcal{M}_2$ obtained from t_1 and t_2 by Rule (5.3) of Definition 5.2 (*see* page 101).

Then, for each valued input or output action $\alpha = \langle a, \omega \rangle \in (\Lambda_1^? \cup \Lambda_1^!) = (\Lambda_2^! \cup \Lambda_2^?) = \Lambda^!$, the triple $\langle\ \langle\langle l_1, l_2\rangle, \langle \vartheta_1, \vartheta_2\rangle\rangle, \alpha, \langle\langle l_1', l_2'\rangle, \langle \vartheta_1', \vartheta_2'\rangle\rangle\ \rangle$ belongs to $\rightarrow_t$ *if and only if*

- the triple $\langle\langle l_1, \vartheta_1\rangle, \langle a, \omega\rangle, \langle l_1', \vartheta_1'\rangle\rangle$ belongs to $\rightarrow_{t_1}$, and
- the triple $\langle\langle l_2, \vartheta_2\rangle, \langle a, \omega\rangle, \langle l_2', \vartheta_2'\rangle\rangle$ belongs to $\rightarrow_{t_2}$.

□

Proof First, notice that as $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for composition, then the signature of each common action a of $\mathcal{M}_1$ and $\mathcal{M}_2$ is the same (*see* Definition 5.1, page 100). Thus, we can replace each parameter carried by a of t_2 with the corresponding parameter carried by a of t_1 in the guard G_2 and the assignments A_2 of t_2, *i.e.* $G_2[\pi_2/\pi_1]$ and $A_2[\pi_2/\pi_1]$. Using this remark we prove the lemma as follows:

(1) Due to Definition 5.1 (*see* page 100), we have that $(V_1 \cup C_1) \cap (V_2 \cup C_2) = \emptyset$. Thus, the guard G_1 (*resp.* G_2) of the symbolic transition t_1 (*resp.* t_2) does not contain constraints on the variables and symbolic constants of $\mathcal{M}_2$ (*resp.* $\mathcal{M}_1$).

Therefore, the pair of valuations $\langle\langle \vartheta_1, \vartheta_2\rangle, \omega\rangle$ satisfies the guard $G_1 \wedge G_2[\pi_2/\pi_1]$ of the symbolic transition t *if and only if*

- $\langle \vartheta_1, \omega\rangle$ satisfies the guard G_1 of the symbolic transition t_1, and
- $\langle \vartheta_2, \omega\rangle$ satisfies the guard G_2 of the symbolic transition t_3.

(2) From Rule (5.3) of Definition 5.2 (*see* page 101) we have that:

the "new" valuation $\vartheta' \in \mathsf{DOM}(V \cup C)$ of $\mathcal{M}_1 \,||\, \mathcal{M}_2$ is obtained from the "old" one $\vartheta \in \mathsf{DOM}(V \cup C)$ by executing in parallel the assignments belonging to $A_1 \cup A_2[\pi_2/\pi_1]$ of the symbolic transition t *if and only if*

- the "new" valuation $\vartheta_1' \in \mathsf{DOM}(V_1 \cup C_1)$ of $\mathcal{M}_1$ is obtained from the "old" one $\vartheta_1 \in \mathsf{DOM}(V_1 \cup C_1)$ by executing in parallel the assignments belonging to A_1 of the symbolic transition t_1, and
- the "new" valuation $\vartheta_2' \in \mathsf{DOM}(V_2 \cup C_2)$ of $\mathcal{M}_1$ is obtained from the "old" one $\vartheta_2 \in \mathsf{DOM}(V_2 \cup C_2)$ by executing in parallel the assignments belonging to A_2 of the symbolic transition t_2.

The lemma follows directly from the items (1) and (2) above. **Q.E.D.**

Before going forward we need to introduce the $\beta^{\Rightarrow}$-behaviors for a given IOSTS.

$\beta^{\Rightarrow}$-Behavior. Let $\mathcal{M}$ be an arbitrary IOSTS with alphabet of valued actions $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$, set of states S and set of initial states $S^0 \subseteq S$. Let also $\beta^{\rightarrow}$ be a behavior of $\mathcal{M}$ (*see* Definition 4.11, page 89). Then, the element obtained by the projection of $\beta^{\rightarrow}$ on the set of states S and on the alphabet of valued input/output actions $(\Lambda^! \cup \Lambda^?)$, is called $\beta^{\Rightarrow}$-behavior.

Formally, a *$\beta^{\Rightarrow}$-behavior* is a sequence of states and valued input/output actions $s^0 \overset{\alpha_1}{\Rightarrow} s_1 \overset{\alpha_2}{\Rightarrow} s_2 \ldots s_{n-1} \overset{\alpha_n}{\Rightarrow} s_n$, where (1) $\Rightarrow$ is the trace relation defined on page 90, (2) $n \in \mathbb{N}$ is the *length* of $\beta^{\Rightarrow}$, (3) $s^0 \in S^0$, and (4) for all $i = 1..n : s_i \in S,\ \alpha_i \in (\Lambda^? \cup \Lambda^!)$.

Remark that a $\beta^{\Rightarrow}$-behavior permits to have a direct connection with a trace of the IOSTS $\mathcal{M}$ (*see* Definition 4.14, page 90). Indeed, as the sequence of valued actions obtained from a given $\beta^{\Rightarrow}$-behavior by dropping all states is a trace of $\mathcal{M}$.

Next, we prove that for each sequence σ of valued input/output actions, an IOSTS $\mathcal{M}_1 \,||\, \mathcal{M}_2$ has a $\beta^{\Rightarrow}$-behavior corresponding to σ *if and only if* $\mathcal{M}_1$ and $\mathcal{M}_2$ have $\beta^{\Rightarrow}$-behaviors corresponding to the given sequence σ. Formally:

Theorem 5.1 For each sequence of valued actions $\sigma = \alpha_1 \ldots \alpha_n$ belonging to $(\Lambda_1^? \cup \Lambda_1^!)^* = (\Lambda_2^! \cup \Lambda_2^?)^*$, the IOSTS $\mathcal{M}_1 \,||\, \mathcal{M}_2$ has the following $\beta^{\Rightarrow}$-behavior:

$$\begin{aligned} \beta^{\Rightarrow} : \quad & \langle\langle l_1^0, l_2^0\rangle, \langle \vartheta_1^0, \vartheta_2^0\rangle\rangle \overset{\alpha_1}{\Rightarrow} \langle\langle l_1^1, l_2^1\rangle, \langle \vartheta_1^1, \vartheta_2^1\rangle\rangle \\ & \ldots \\ & \langle\langle l_1^{n-1}, l_2^{n-1}\rangle, \langle \vartheta_1^{n-1}, \vartheta_2^{n-1}\rangle\rangle \overset{\alpha_n}{\Rightarrow} \langle\langle l_1^n, l_2^n\rangle, \langle \vartheta_1^n, \vartheta_2^n\rangle\rangle \end{aligned}$$

if and only if

- $\mathcal{M}_1$ has $\beta_1^{\Rightarrow} : \langle l_1^0, \vartheta_1^0\rangle \overset{\alpha_1}{\Rightarrow} \langle l_1^1, \vartheta_1^1\rangle \ldots \langle l_1^{n-1}, \vartheta_1^{n-1}\rangle \overset{\alpha_n}{\Rightarrow} \langle l_1^n, \vartheta_1^n\rangle$, and
- $\mathcal{M}_2$ has $\beta_2^{\Rightarrow} : \langle l_2^0, \vartheta_2^0\rangle \overset{\alpha_1}{\Rightarrow} \langle l_2^1, \vartheta_2^1\rangle \ldots \langle l_2^{n-1}, \vartheta_2^{n-1}\rangle \overset{\alpha_n}{\Rightarrow} \langle l_2^n, \vartheta_2^n\rangle$.

□

Proof The proof is done by induction on the length of the sequence σ of valued actions belonging to $(\Lambda_1^? \cup \Lambda_1^!)^* = (\Lambda_2^! \cup \Lambda_2^?)^*$.

Induction Basis. Consider the empty sequence of valued actions ε. We prove that $\langle\langle l_1^0, l_2^0\rangle, \langle \vartheta_1^0, \vartheta_2^0\rangle\rangle$ is an initial state of $\mathcal{M}_1 \,||\, \mathcal{M}_2$ *if and only if* $\langle l_1^0, \vartheta_1^0\rangle$ and $\langle l_2^0, \vartheta_2^0\rangle$ are initial states of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively.

Due to Definition 5.1 (*see* page 100), we have that $(V_1 \cup C_1) \cap (V_2 \cup C_2) = \emptyset$. Thus, the initial condition Θ_1 (*resp.* Θ_2) of $\mathcal{M}_1$ (*resp.* $\mathcal{M}_2$) does not contain constraints on the variables and symbolic constants of $\mathcal{M}_2$ (*resp.* $\mathcal{M}_1$).

Thus, the valuation $\langle \vartheta_1^0, \vartheta_2^0\rangle \in \mathsf{DOM}(V \cup C)$ of $\mathcal{M}_1 \,||\, \mathcal{M}_2$, satisfies the initial condition $\Theta_1 \wedge \Theta_2$ of $\mathcal{M}_1 \,||\, \mathcal{M}_2$ *if and only if*

– $\vartheta_1^0 \in \mathsf{DOM}(V_1 \cup C_1)$ satisfies the initial condition Θ_1 of $\mathcal{M}_1$, and
– $\vartheta_2^0 \in \mathsf{DOM}(V_2 \cup C_2)$ satisfies the initial condition Θ_2 of $\mathcal{M}_2$.

Therefore, the induction basis is proved.

Induction Hypothesis. Assume that for a sequence of valued actions $\sigma' = \alpha_1 \dots \alpha_{n-1}$ the length of which is $n-1$, the IOSTS $\mathcal{M}_1 \,||\, \mathcal{M}_2$ has the $\beta^{\Rightarrow}$-behavior:

$$\begin{aligned}\beta^{\Rightarrow} : \quad & \langle\langle l_1^0, l_2^0\rangle, \langle \vartheta_1^0, \vartheta_2^0\rangle\rangle \overset{\alpha_1}{\Rightarrow} \langle\langle l_1^1, l_2^1\rangle, \langle \vartheta_1^1, \vartheta_2^1\rangle\rangle \\ & \dots \\ & \langle\langle l_1^{n-2}, l_2^{n-2}\rangle, \langle \vartheta_1^{n-2}, \vartheta_2^{n-2}\rangle\rangle \overset{\alpha_{n-1}}{\Rightarrow} \langle\langle l_1^{n-1}, l_2^{n-1}\rangle, \langle \vartheta_1^{n-1}, \vartheta_2^{n-1}\rangle\rangle\end{aligned}$$

if and only if $\mathcal{M}_1$ and $\mathcal{M}_2$ have the following $\beta^{\Rightarrow}$-behaviors:

$$\begin{aligned}\beta_1^{\Rightarrow} &: \langle l_1^0, \vartheta_1^0\rangle \overset{\alpha_1}{\Rightarrow} \langle l_1^1, \vartheta_1^1\rangle \dots \langle l_1^{n-2}, \vartheta_1^{n-2}\rangle \overset{\alpha_{n-1}}{\Rightarrow} \langle l_1^{n-1}, \vartheta_1^{n-1}\rangle \\ \beta_2^{\Rightarrow} &: \langle l_2^0, \vartheta_2^0\rangle \overset{\alpha_1}{\Rightarrow} \langle l_2^1, \vartheta_2^1\rangle \dots \langle l_2^{n-2}, \vartheta_2^{n-2}\rangle \overset{\alpha_{n-1}}{\Rightarrow} \langle l_2^{n-1}, \vartheta_2^{n-1}\rangle\end{aligned}$$

Induction Step. Consider a sequence of valued actions of length n, *i.e.* $\sigma = \underbrace{\alpha_1 \dots \alpha_{n-1}}_{\sigma'} \alpha_n$. We prove that:

$$\begin{aligned}\beta^{\Rightarrow} : \quad & \langle\langle l_1^0, l_2^0\rangle, \langle \vartheta_1^0, \vartheta_2^0\rangle\rangle \overset{\alpha_1}{\Rightarrow} \langle\langle l_1^1, l_2^1\rangle, \langle \vartheta_1^1, \vartheta_2^1\rangle\rangle \\ & \dots \\ & \langle\langle l_1^{n-1}, l_2^{n-1}\rangle, \langle \vartheta_1^{n-1}, \vartheta_2^{n-1}\rangle\rangle \overset{\alpha_n}{\Rightarrow} \langle\langle l_1^n, l_2^n\rangle, \langle \vartheta_1^n, \vartheta_2^n\rangle\rangle\end{aligned} \tag{5.5}$$

is a $\beta^{\Rightarrow}$-behavior of $\mathcal{M}_1 \,||\, \mathcal{M}_2$ *if and only if*

$$\beta_1^{\Rightarrow} : \langle l_1^0, \vartheta_1^0\rangle \overset{\alpha_1}{\Rightarrow} \langle l_1^1, \vartheta_1^1\rangle \dots \langle l_1^{n-1}, \vartheta_1^{n-1}\rangle \overset{\alpha_n}{\Rightarrow} \langle l_1^n, \vartheta_1^n\rangle \tag{5.6}$$

$$\beta_2^{\Rightarrow} : \langle l_2^0, \vartheta_2^0\rangle \overset{\alpha_1}{\Rightarrow} \langle l_2^1, \vartheta_2^1\rangle \dots \langle l_2^{n-1}, \vartheta_2^{n-1}\rangle \overset{\alpha_n}{\Rightarrow} \langle l_2^n, \vartheta_2^n\rangle \tag{5.7}$$

are $\beta^{\Rightarrow}$-behaviors of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively.

By using prefix closure and the induction hypothesis, we obtain that for the sequence σ', the prefix of $\beta^{\Rightarrow}$ (*see* Formula (5.5)) whose length is $n-1$, is a sequence of states and valued input/output actions of $\mathcal{M}_1 \,||\, \mathcal{M}_2$ *if and only if* the prefixes of $\beta_1^{\Rightarrow}$ and $\beta_2^{\Rightarrow}$ (*see* Formulas (5.6), (5.7)), whose lengths are also equal to $n-1$, are sequences of states and valued input/output actions of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively.

Next, we prove that the trace relation $\langle\langle l_1^{n-1}, l_2^{n-1}\rangle, \langle \vartheta_1^{n-1}, \vartheta_2^{n-1}\rangle\rangle \overset{\alpha_n}{\Rightarrow}$

$\langle\langle l_1^n, l_2^n\rangle, \langle \vartheta_1^n, \vartheta_2^n\rangle\rangle$ which is the last step of $\beta^{\Rightarrow}$ shown as Formula (5.5), holds in $\mathcal{M}_1 \mid\mid \mathcal{M}_2$ *if and only if* $\langle l_1^{n-1}, \vartheta_1^{n-1}\rangle \stackrel{\alpha_{\aleph}}{\Rightarrow} \langle l_1^n, \vartheta_1^n\rangle$ (the last step of $\beta_1^{\Rightarrow}$ shown as Formula (5.6)) holds in $\mathcal{M}_1$ and $\langle l_2^{n-1}, \vartheta_2^{n-1}\rangle \stackrel{\alpha_{\aleph}}{\Rightarrow} \langle l_2^n, \vartheta_2^n\rangle$ (the last step of $\beta_2^{\Rightarrow}$ shown as Formula (5.7)) holds in $\mathcal{M}_2$.

First, using Definition 4.13 (*see* page 90) we unfold the last steps of $\beta_1^{\Rightarrow}$, $\beta_2^{\Rightarrow}$ and $\beta^{\Rightarrow}$. Formulas (5.8), (5.9) and (5.10) respectively show results of this unfolding. Then, we prove:

$$\langle l_1^{n-1}, \vartheta_1^{n-1}\rangle \stackrel{\tau_1^1}{\rightarrow} \ldots \stackrel{\tau_1^i}{\rightarrow} \langle l_1, \vartheta_1\rangle \stackrel{\alpha_{\aleph}}{\rightarrow} \langle l_1', \vartheta_1'\rangle \stackrel{\tau_1^{i+1}}{\rightarrow} \ldots \stackrel{\tau_1^k}{\rightarrow} \langle l_1^n, \vartheta_1^n\rangle \tag{5.8}$$

$$\langle l_2^{n-1}, \vartheta_2^{n-1}\rangle \stackrel{\tau_2^1}{\rightarrow} \ldots \stackrel{\tau_2^j}{\rightarrow} \langle l_2, \vartheta_2\rangle \stackrel{\alpha_{\aleph}}{\rightarrow} \langle l_2', \vartheta_2'\rangle \stackrel{\tau_2^{j+1}}{\rightarrow} \ldots \stackrel{\tau_2^p}{\rightarrow} \langle l_2^n, \vartheta_2^n\rangle \tag{5.9}$$

where $\forall\ m \in 1..k\ .[\tau_1^m \in \Sigma_1^{\tau}]$, $\forall\ m \in 1..p\ .[\tau_2^m \in \Sigma_2^{\tau}]$, holds in $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively *if and only if*

$$\begin{aligned}&\langle\langle l_1^{n-1}, l_2^{n-1}\rangle, \langle \vartheta_1^{n-1}, \vartheta_2^{n-1}\rangle\rangle \stackrel{\tau_1^1}{\rightarrow} \ldots \stackrel{\tau^{i+j}}{\rightarrow} \langle\langle l_1, l_2\rangle, \langle \vartheta_1, \vartheta_2\rangle\rangle \stackrel{\alpha_{\aleph}}{\rightarrow} \\ &\langle\langle l_1', l_2'\rangle, \langle \vartheta_1', \vartheta_2'\rangle\rangle \stackrel{\tau^{i+j+1}}{\rightarrow} \ldots \stackrel{\tau^{k+p}}{\rightarrow} \langle\langle l_1^n, l_2^n\rangle, \langle \vartheta_1^n, \vartheta_2^n\rangle\rangle\end{aligned} \tag{5.10}$$

where $\forall\ m \in 1..(k+p)\ .[\tau^m \in (\Sigma_1^{\tau} \cup \Sigma_2^{\tau})]$, holds in $\mathcal{M}_1 \mid\mid \mathcal{M}_2$.

(1) Using Lemma 5.1 (*see* page 106) and Remark 5.1 (*see* page 107) iteratively we get:

the state $s = \langle\langle l_1, l_2\rangle, \langle \vartheta_1, \vartheta_2\rangle\rangle$ is reachable (*see* Definition 4.15, page 91) from $s^{n-1} = \langle\langle l_1^{n-1}, l_2^{n-1}\rangle, \langle \vartheta_1^{n-1}, \vartheta_2^{n-1}\rangle\rangle$ by executing the sequence of internal actions $\tau^1 \ldots \tau^{i+j}$ which was constructed by using each action belonging to the set: $\{\tau_1^1, \ldots, \tau_1^i, \tau_2^1, \ldots, \tau_2^j\}$ once

if and only if

the states $s_1 = \langle l_1, \vartheta_1\rangle$ and $s_2 = \langle l_2, \vartheta_2\rangle$ are reachable from the states $s_1^{n-1} = \langle l_1^{n-1}, \vartheta_1^{n-1}\rangle$ and $s_2^{n-1} = \langle l_2^{n-1}, \vartheta_2^{n-1}\rangle$ respectively by executing the following sequences of internal actions: $\tau_1^1 \ldots \tau_1^i$ and $\tau_2^1 \ldots \tau_2^j$, which are obtained from the sequence $\tau^1 \ldots \tau^{i+j}$ by projection on the set of internal actions of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively.

(2) From Lemma 5.2 (*see* page 107) we have:

the relation $\underbrace{\langle\langle l_1, l_2\rangle, \langle \vartheta_1, \vartheta_2\rangle\rangle}_{s} \stackrel{\alpha_{\aleph}}{\rightarrow} \underbrace{\langle\langle l_1', l_2'\rangle, \langle \vartheta_1', \vartheta_2'\rangle\rangle}_{s'}$ holds in $\mathcal{M}_1 \mid\mid \mathcal{M}_2$

if and only if

$\underbrace{\langle l_1, \vartheta_1\rangle}_{s_1} \stackrel{\alpha_{\aleph}}{\rightarrow} \underbrace{\langle l_1', \vartheta_1'\rangle}_{s_1'}$ holds in $\mathcal{M}_1$ and $\underbrace{\langle l_2, \vartheta_2\rangle}_{s_2} \stackrel{\alpha_{\aleph}}{\rightarrow} \underbrace{\langle l_2', \vartheta_2'\rangle}_{s_2'}$ holds in $\mathcal{M}_2$.

(3) Applying iteratively Lemma 5.1 (*see* page 106) and Remark 5.1 (*see* page 107) we obtain:
the state $s^n = \langle\langle l_1^n, l_2^n\rangle, \langle \vartheta_1^n, \vartheta_2^n\rangle\rangle$ is reachable from $s' = \langle\langle l_1', l_2'\rangle, \langle \vartheta_1', \vartheta_2'\rangle\rangle$ by executing the sequence of internal actions $\tau^{i+j+1} \dots \tau^{k+p}$ which was constructed by using each action belonging to the set: $\{\tau_1^{i+1}, \dots, \tau_1^k, \tau_2^{j+1}, \dots, \tau_2^p\}$ once
if and only if
the state $s_1^n = \langle l_1^n, \vartheta_1^n\rangle$ and $s_2^n = \langle l_2^n, \vartheta_2^n\rangle$ are reachable from $s_1' = \langle l_1', \vartheta_1'\rangle$ and $s_2' = \langle l_2', \vartheta_2'\rangle$ respectively by executing the following sequences of internal actions: $\tau_1^{i+1} \dots \tau_1^k$ and $\tau^{j+1} \dots \tau^p$, which are obtained from the sequence $\tau^{i+j+1} \dots \tau^{k+p}$ by projection on the set of internal actions of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively.

Therefore, we have proved that for any sequence σ of length n, the IOSTS $\mathcal{M}_1 \,||\, \mathcal{M}_2$ has the $\beta^{\rightarrow}$-behavior shown as Formula (5.5) *if and only if* $\mathcal{M}_1$ and $\mathcal{M}_2$ have $\beta^{\rightarrow}$-behaviors shown as Formulas (5.6) and (5.7) respectively.

Q.E.D.

Finally, we formally state and prove the theorem about the equality between the set of traces of $\mathcal{M}_1 \,||\, \mathcal{M}_2$ and the set of traces obtained as the intersection between the set of traces of $\mathcal{M}_1$ and set of traces of $\mathcal{M}_2$.

Theorem 5.2 (Traces of the Parallel Composition) For two IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ which are compatible for the parallel composition (Definition 5.1, page 100), the equality:

$$\mathit{Traces}(\mathcal{M}_1 \,||\, \mathcal{M}_2) = \mathit{Traces}(\mathcal{M}_1) \cap \mathit{Traces}(\mathcal{M}_2)$$

holds. □

Proof The proof of this theorem immediately follows from Definition 4.14 (*see* page 90) and Theorem 5.1 (*see* page 109). **Q.E.D.**

5.2 Product

This section introduces the product operation $\times_{IOSTS}$ that is the main operation in our symbolic test generation method (*see* Chapter 7, page 167). The product operation "intersects" the behavior of two IOSTS representing, in the test generation algorithm, a specification and a test purpose. This allows to select a part of the specification for which a test case has to be generated.

The operation $\times_{IOSTS}$ is inspired from the product operation $\times_{IOLTS}$ defined for IOLTS and used in the test generation method described in Subsection 3.2.3.3 (*see* page 57). The difference between $\times_{IOSTS}$ and $\times_{IOLTS}$ is that the former performs not only synchronization on common actions of two given systems, but also on data of these systems (remember that the IOSTS model explicitly includes systems data). Therefore, the product operation defined for IOSTS permits to perform more precise selection of test cases. In the rest of the thesis we only use the product operation defined for IOSTS which is simply denoted by $\times$ without index.

Before defining the product operation, we describe the conditions which permit to perform the product operation between two IOSTS. Intuitively, two IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for the product operation if and only if:

- $\mathcal{M}_1$ and $\mathcal{M}_2$ do not share either variables or parameters, but they may share symbolic constants. Moreover the variables of $\mathcal{M}_1$ can be symbolic constants of $\mathcal{M}_2$, and the variables of $\mathcal{M}_2$ can be symbolic constants of $\mathcal{M}_1$. Also, in the case of a common datum, this datum must have the same type in both IOSTS.

  This condition is needed to avoid the conflicts between common data of the IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$. Indeed, assume the situation when $\mathcal{M}_1$ and $\mathcal{M}_2$ have the common variable v, and they change differently the value of this variable on the symbolic transitions that will be synchronized. If we try to construct the product between these IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$, we will be faced with to the problem of choosing the right assignment of the variable v (*i.e.* we do not know if variable should be evaluated as in $\mathcal{M}_1$ or as in $\mathcal{M}_2$). To avoid this problem we forbid $\mathcal{M}_1$ and $\mathcal{M}_2$ to have the same variables.

  Nevertheless, a common datum v of $\mathcal{M}_1$ and $\mathcal{M}_2$ can be considered, for example, as a symbolic constant of $\mathcal{M}_1$ and a variable of $\mathcal{M}_2$. In this situation the conflict does not appear as, in the synchronous product $\mathcal{M}_1 \times \mathcal{M}_2$, we treat this datum v as a variable and, therefore, assign it as in $\mathcal{M}_1$.

- Their alphabets of input, output and internal actions are equal, and their common actions must have the same signature in both IOSTS.

Next we formally define the notion of compatibility for the product operation.

Definition 5.3 (Compatible For Product) Two IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ with sets of data: $D_1 = V_1 \cup C_1 \cup P_1, D_2 = V_2 \cup C_2 \cup P_2$ and sets of actions: $\Sigma_1 = \Sigma_1^? \cup \Sigma_1^! \cup \Sigma_1^\tau, \Sigma_2 = \Sigma_2^? \cup \Sigma_2^! \cup \Sigma_2^\tau$ are *compatible for the product operation* if

(1) – $\mathcal{M}_1$ and $\mathcal{M}_2$ do not have common variables and parameters, *i.e.* $V_1 \cap V_2 = \emptyset$ and $P_1 \cap P_2 = \emptyset$;

– each *common* datum $d \in (C_1 \cap C_2) \cup (V_1 \cap C_2) \cup (V_2 \cap C_1)$ of $\mathcal{M}_1$ and $\mathcal{M}_2$ has the same type in both IOSTS.

(2) – the alphabets of input, output and internal actions of $\mathcal{M}_1$ and $\mathcal{M}_2$ are exactly the same, *i.e.* $\Sigma_1^? = \Sigma_2^?, \Sigma_1^! = \Sigma_2^!, \Sigma_1^\tau = \Sigma_2^\tau$; and

– each *common* action a of $\mathcal{M}_1$ and $\mathcal{M}_2$ has the same signature in both IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$, *i.e.* $\mathsf{sig}_1(a) = \mathsf{sig}_2(a)$. This means that the length of the tuples of types $\mathsf{sig}_1(a) = \langle \mathfrak{t}_1^1, \ldots, \mathfrak{t}_1^k \rangle$ and $\mathsf{sig}_2(a) = \langle \mathfrak{t}_2^1, \ldots, \mathfrak{t}_2^k \rangle$ is the same, and each type $\mathfrak{t}_i^1$ of $\mathsf{sig}_1(a)$ corresponds to the type $\mathfrak{t}_i^2$ of $\mathsf{sig}_2(a)$, *i.e.* $\forall\, i = 1..k \;.\; [\mathfrak{t}_1^i = \mathfrak{t}_2^i]$.

□

Then, we formally define the product operation, make some observations about it, and illustrate this operation with a simple example.

Definition 5.4 (Product $\times$) The *product* between two IOSTS $\mathcal{M}_1 = \langle D_1, \Theta_1, L_1, l_1^0, \Sigma_1, T_1 \rangle$ and $\mathcal{M}_2 = \langle D_2, \Theta_2, L_2, l_2^0, \Sigma_2, T_2 \rangle$ which are compatible for the product operation is an IOSTS $\mathcal{M}_1 \times \mathcal{M}_2 = \langle D, \Theta, L, l^0, \Sigma, T \rangle$, where

– $D = V \cup C \cup P$, where $V = V_1 \cup V_2$, $C = (C_1 \cup C_2) \setminus (V_1 \cup V_2)$ and $P = P_1$.

The set of parameters P of $\mathcal{M}_1 \times \mathcal{M}_2$ is equal to the set of parameters P_1 of $\mathcal{M}_1$ as during the synchronization of the symbolic transitions each parameter of $\mathcal{M}_2$ is replaced with the corresponding parameter of $\mathcal{M}_1$ (*see* Rule (5.11));

– $\Theta = \Theta_1 \wedge \Theta_2$;

– $L = L_1 \times L_2$;

– $l^0 = \langle l_1^0, l_2^0 \rangle \in L$ is the initial location;

– $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ is the alphabet of actions, where $\Sigma^? = \Sigma_1^? (= \Sigma_2^?)$, $\Sigma^! = \Sigma_1^! (= \Sigma_2^!)$ and $\Sigma^\tau = \Sigma_1^\tau (= \Sigma_2^\tau)$; and

– the set of symbolic transitions T is constructed from T_1 and T_2 as follows:

For two symbolic transitions $t_1 \in T_1$ and $t_2 \in T_2$, which are labeled with an action a that belongs to the alphabet $\Sigma_1 = \Sigma_2$, a new symbolic transition $t \in T$ is constructed using the inference rule:

$$\frac{\begin{array}{c}\langle l_1, a, \pi_1, G_1, A_1, l_1' \rangle \in T_1 \\ \langle l_2, a, \pi_2, G_2, A_2, l_2' \rangle \in T_2 \\ a \in (\Sigma_1 = \Sigma_2)\end{array}}{\langle \langle l_1, l_2 \rangle, a, \pi_1, G_1 \wedge G_2[\pi_2/\pi_1], A_1 \cup A_2[\pi_2/\pi_1], \langle l_1', l_2' \rangle \rangle \in T} \tag{5.11}$$

where $G_2[\pi_2/\pi_1]$ (*resp.* $A_2[\pi_2/\pi_1]$) is the guard (*resp.* the set of assignments) of the symbolic transition t_2 in which each parameter $p_2^i \in \pi_2$ carried by the action a of t_2 is replaced by corresponding parameter $p_1^i \in \pi_1$ carried by the action a of t_1.

□

It is important to notice that the product operation is stable, *i.e.* the result of this operation is indeed an IOSTS. This statement follows directly from the construction of the product (*see* Definition 5.4). The only one less trivial point is to show that the assignment of the symbolic transitions of the IOSTS obtained after the product operation are well-formed. This point is formulated and proved below.

Observation 5.1 (Well-Formedness of Symbolic Transitions) Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two IOSTS compatible for the product operation (*see* Definition 5.3, page 113). Then, each symbolic transition of the IOSTS $\mathcal{M}_1 \times \mathcal{M}_2$ obtained by the product operation from $\mathcal{M}_1$ and $\mathcal{M}_2$ (*see* Definition 5.4, page 114) contains exactly one assignment for each variable of $\mathcal{M}_1 \times \mathcal{M}_2$. □

Proof Consider a variable v of $\mathcal{M}_1 \times \mathcal{M}_2$. Due to the fact that $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for the product operation (*see* Definition 5.3, page 113), we can consider the following cases:

(1) v is a variable of $\mathcal{M}_1$ and a symbolic constant of $\mathcal{M}_2$. In this case, v can be assigned with a new value in $\mathcal{M}_1$, but cannot be assigned in $\mathcal{M}_2$. Thus, if during the operation of parallel composition we synchronize a symbolic transition of $\mathcal{M}_1$, which contains an assignment to the variable v, with any other symbolic transition of $\mathcal{M}_2$, we will obtain a symbolic transition in $\mathcal{M}_1 \times \mathcal{M}_2$ containing only one assignment to v.

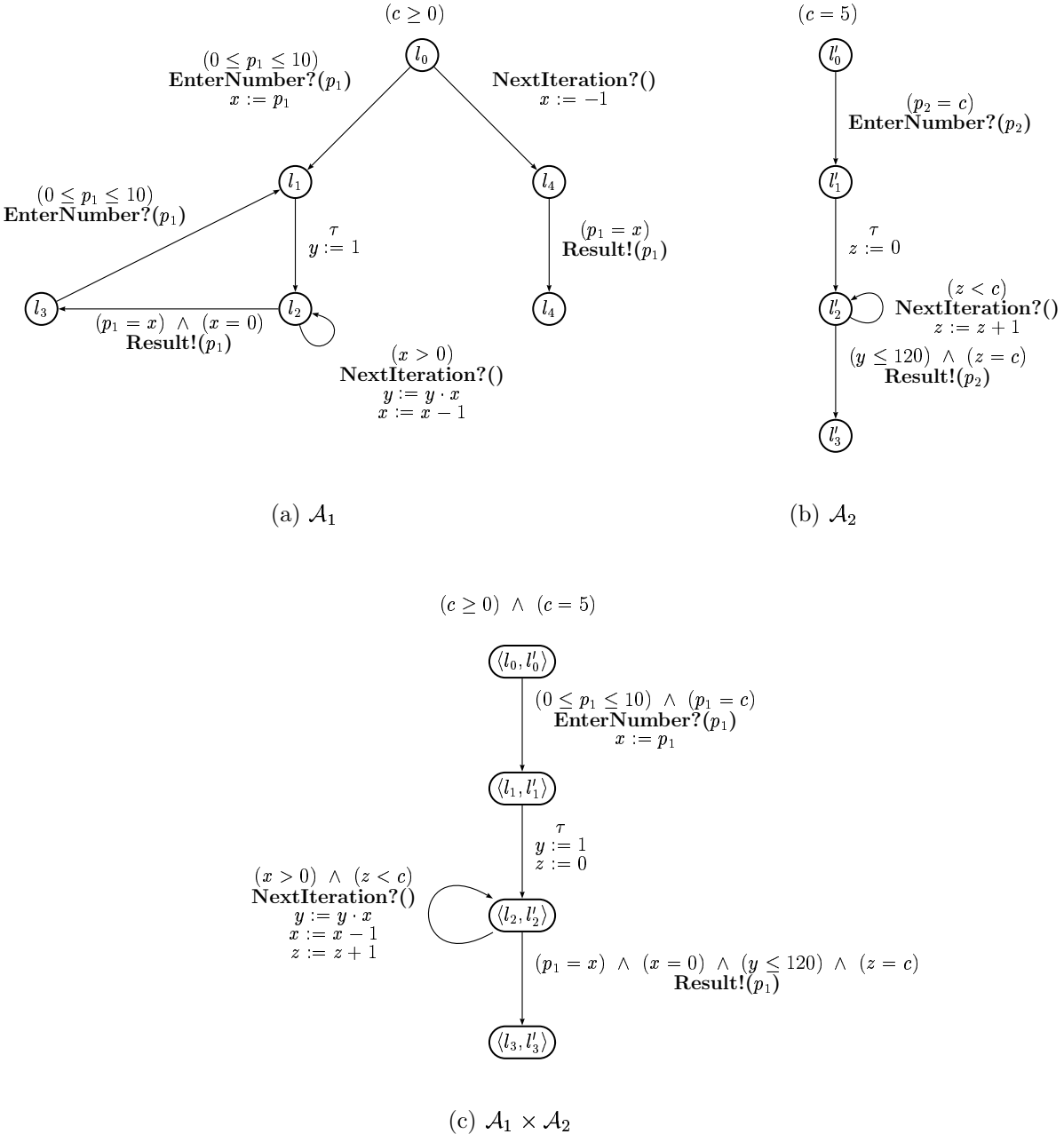
Similarly, we consider the case where v is a variable of $\mathcal{M}_2$ and symbolic constant of $\mathcal{M}_1$.

(2) v is a private variable of $\mathcal{M}_1$ or $\mathcal{M}_2$. In this case, the variable v can be assigned either in $\mathcal{M}_1$ or in $\mathcal{M}_2$, thus each symbolic transition of $\mathcal{M}_1 \times \mathcal{M}_2$ contains exactly one assignment to v.

Q.E.D.

Example 5.2 To illustrate the product operation we consider two IOSTS $\mathcal{A}_1$ and $\mathcal{A}_2$ shown on Figure 5.2 (*see* page 116).

The IOSTS $\mathcal{A}_1$ (*see* Figure 5.2(a)),



(a) $\mathcal{A}_1$ (b) $\mathcal{A}_2$

(c) $\mathcal{A}_1 \times \mathcal{A}_2$

Figure 5.2: The product operation between two IOSTS $\mathcal{A}_1$ and $\mathcal{A}_2$.

(1) in the case of receiving the input action *EnterNumber* from its environment, computes the factorial of the integer value entered to $\mathcal{A}_1$ through this action, returns the computed value through the output action *Result*, and repeats this operation again;

(2) in the case of receiving the input action *NextIteration*, $\mathcal{A}_1$ returns the value -1 using the output action *Result*.

The IOSTS $\mathcal{A}_2$ (*see* Figure 5.2(b)) receives some integer value, which is equal to the value of its symbolic constants c, from the environment using the input action *EnterNumber*; then it initializes the variable z to zero through its internal action τ; and receives at most five times the input action *NextIteration*, each time increasing the variable z by one; after that, $\mathcal{A}_2$ sends to the environment the output action *Result*.

These IOSTS $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible for the product operation (*see* Definition 5.3, page 113) as:

(1)
- $\mathcal{A}_1$ and $\mathcal{A}_2$ do not have neither common variables nor common parameters: $\underbrace{\{x, y\}}_{V_1} \cap \underbrace{\{z\}}_{V_2} = \emptyset$, and $\underbrace{\{p_1\}}_{P_1} \cap \underbrace{\{p_2\}}_{P_2} = \emptyset$;
- $\mathcal{A}_1$ and $\mathcal{A}_2$ have two common data: c and y, where c is a symbolic constant for both IOSTS, and y is a symbolic constant for $\mathcal{A}_2$ (as it never change its value, *see* Figure 5.2(b)) and a variables for $\mathcal{A}_1$. This data have the same types in $\mathcal{A}_1$ and $\mathcal{A}_2$, for example, the symbolic constant c has integer type in both IOSTS.

(2)
- The alphabets of input, output and internal actions of $\mathcal{A}_1$ and $\mathcal{A}_2$ are equal, *i.e.* $\Sigma_1^? = \Sigma_2^? = \{EnterNumber, NextIteration\}$, $\Sigma_1^! = \Sigma_2^! = \{Result\}$ and $\Sigma_1^\tau = \Sigma_2^\tau = \{tau\}$.
- The signatures of actions shared by $\mathcal{A}_1$ and $\mathcal{A}_2$ are the same. For instance, the output action *Return* of $\mathcal{A}_1$ and $\mathcal{A}_2$ has the same signature $\mathsf{sig}_1(Return) = \mathsf{sig}_2(Return) = \langle \mathtt{int} \rangle$ in both IOSTS, where `int` denotes the integer type.

On Figure 5.2(c) we depicted the IOSTS $\mathcal{A}_1 \times \mathcal{A}_2$ obtained from $\mathcal{A}_1$ and $\mathcal{A}_2$ by performing the product operation as follows:

(1) the set of data of $\mathcal{A}_1 \times \mathcal{A}_2$ consists of the following variables, symbolic constants and parameters: $\underbrace{\{x, y, z\}}_{V} \cup \underbrace{\{c\}}_{C} \cup \underbrace{\{p_1\}}_{P}$. Note that the symbolic constant y of $\mathcal{A}_2$ became the variable of $\mathcal{A}_1 \times \mathcal{A}_2$ as it was the variable in $\mathcal{A}_2$;

(2) the initial condition of $\mathcal{A}_1 \times \mathcal{A}_2$ is the conjunction of the initial conditions of $\mathcal{A}_1$ and $\mathcal{A}_2$, *i.e.* $\Theta : (c \geq 0) \wedge (c = 5)$ (*see* Figure 5.2(c));

(3) the initial location of $\mathcal{A}_1 \times \mathcal{A}_2$ is the pair $\langle l_0, l'_0 \rangle$ (*see* Figure 5.2(c)), where l_0 is the initial location of $\mathcal{A}_1$ and l'_0 is the initial location of $\mathcal{A}_2$;

(4) the alphabets of input, output and internal actions are the same as the alphabets of input, output and internal actions of $\mathcal{A}_1$ or $\mathcal{A}_2$;

(5) finally, we explain how to construct the set of symbolic transitions of $\mathcal{A}_1 \times \mathcal{A}_2$.

First, note that the IOSTS $\mathcal{A}_1$ staying in its initial location l_0 can perform either the action *EnterNumber* or the action *NextIteration*, but $\mathcal{A}_1$ can execute only the action *EnterNumber* from its initial location l'_0. By Definition 5.4 (*see* page 114) we know that the symbolic transitions of $\mathcal{A}_1$ and $\mathcal{A}_2$ can be fired synchronously if they are labeled with the same actions. This means that at the first step we synchronize the symbolic transitions of $\mathcal{A}_1$ and $\mathcal{A}_2$ outgoing from their initial locations and labeled by the same action *EnterNumber*. After synchronization, we obtain the following symbolic transition of the IOSTS $\mathcal{A}_1 \times \mathcal{A}_2$: $t = \langle \langle l_0, l'_0 \rangle, \mathit{EnterNumber}, \pi, G, A, \langle l_1, l'_1 \rangle \rangle$, where

- *EnterNumber* is the input action carrying the tuple of parameters $\pi = \langle p_1 \rangle$;
- the guard G is conjunction between the guard $(0 \leq p_1 \leq 10)$ of $\mathcal{A}_1$ and the guard $(p_2 = c)$ of $\mathcal{A}_2$, where each parameter p_2 is replaced by the parameter p_1, *i.e.* $G : (0 \leq p_1 \leq 10) \wedge (p_1 = c)$;
- the assignments A is the union between the set of assignments $\{x := p_1, y := y\}$ of $\mathcal{A}_1$ and the set of assignments $\{z := z\}$ of $\mathcal{A}_2$, where each parameter p_2 is replaced by the parameter p_1, *i.e.* $A : \{x := p_1, y := y, z := z\}$

The result of this synchronization is shown on Figure 5.2(c) (*see* the symbolic transition of $\mathcal{A}_1 \times \mathcal{A}_2$ outgoing from the initial location $\langle l_0, l'_0 \rangle$).

Continuing in the same way, we obtain the whole set of symbolic transitions of the IOSTS $\mathcal{A}_1 \times \mathcal{A}_2$ (*see* Figure 5.2(c)).

□

5.2.1 Preliminary Definitions and Notations for Product

This section presents preliminary definitions and notations used in order to prove several relationships about the sets of traces/sequences of the IOSTS obtained after the product operation. All new notions are illustrated with intuitive examples.

Definition 5.5 (Fusion) Let D_1 and D_2 be two sets of typed data, ϑ_1 be a valuation of D_1 and ϑ_2 be a valuation of D_2. Then the *fusion* of ϑ_1 and ϑ_2 is defined as follows:

$$\forall\ d \in D_1 \cup D_2\ .\ \mathsf{fusion}(\vartheta_1, \vartheta_2)(d) = \begin{cases} \vartheta_1(d), & \textbf{if} \quad d \in D_1 \\ \vartheta_2(d), & \textbf{otherwise} \end{cases}$$

□

Note that the operation of fusion of two valuations is asymmetric. It is useful for proving the traces property of the product (*see* Section 5.2.2, page 122), where some variables and symbolic constants can be shared.

Definition 5.6 (Projection) Let D be a set of typed data, ϑ be a valuation of D, and D_1 be a subset of D. Then the *projection* of ϑ onto D_1 is defined as follows:

$$\forall\ d \in D_1\ .\ \mathsf{projection}_{D_1}(\vartheta)(d) = \vartheta(d)$$

□

Example 5.3 This examples illustrates the fusion and projection operations defined above.

(1) Consider two sets of typed data: $D_1 = \{d_1, d_2, d_3\}$ and $D_2 = \{d_3, d_4\}$, where $\mathsf{DOM}(D_1) = \mathtt{nat} \times \mathtt{bool} \times \mathtt{nat}$ and $\mathsf{DOM}(D_2) = \mathtt{nat} \times \mathtt{bool}$. Consider also the valuation of D_1: $\vartheta_1 = \langle 3, true, 5\rangle$, and the valuation of D_2: $\vartheta_2 = \langle 7, false\rangle$. Then, by performing the operation of fusion between ϑ_1 and ϑ_2 we obtain the following valuation of D:

$$\mathsf{fusion}(\vartheta_1, \vartheta_2) = \langle \underbrace{3}_{d_1}, \underbrace{true}_{d_2}, \underbrace{5}_{d_3}, \underbrace{false}_{d_4} \rangle$$

(2) We consider the following set of typed data: $D = \{d_1, d_2, d_3, d_4\}$, where $\mathsf{DOM}(D) = \mathtt{nat} \times \mathtt{bool} \times \mathtt{nat} \times \mathtt{bool}$. Note that D is the union of two sets of typed data considered above, *i.e.* D_1 and D_2. Consider also the valuation of D: $\vartheta = \langle 3, true, 5, false\rangle$. Then, by performing the operation of projection onto D_1 and D_2, we obtain two valuations: $\vartheta_1 = \langle 3, true, 5\rangle$ and $\vartheta_2 = \langle 5, false\rangle$ of D_1 and D_2 respectively.

□

Observation 5.2 Let D_1 and D_2 be two sets of typed data, ϑ_1 be a valuation of D_1 such that ϑ_1 satisfies a Boolean expression $B_1 \in \mathcal{B}(D_1)$ and ϑ_2 be a valuation of D_2 such that ϑ_2 satisfies a Boolean expression $B_2 \in \mathcal{B}(D_2)$. If $\forall\ d \in D_1 \cap D_2\ .\ [\vartheta_1(d) = \vartheta_2(d)]$, then $\mathsf{fusion}(\vartheta_1, \vartheta_2) \models B_1 \wedge B_2$. □

Observation 5.3 Let $D = D_1 \cup D_2$ be a set of typed data, and ϑ be a valuation of D such that ϑ satisfies the conjunction of the following Boolean expressions $B_1 \in \mathcal{B}(D_1)$ and $B_2 \in \mathcal{B}(D_2)$, *i.e.* $\vartheta \models B_1 \wedge B_2$.

Then, the valuation $\vartheta_1 \in \mathsf{DOM}(D_1)$ (*resp.* $\vartheta_2 \in \mathsf{DOM}(D_2)$) obtained by the operation $\mathsf{projection}_{D_1}(\vartheta)$ (*resp.* $\mathsf{projection}_{D_2}(\vartheta)$) satisfies B_1 (*resp.* B_2), *i.e.* $\vartheta_1 \models B_1$ (*resp.* $\vartheta_2 \models B_2$). □

Lemma 5.3 Let D_1 and D_2 be two sets of typed data, ϑ_1 be a valuation of $D_1 \setminus D_2$, ϑ_2 be a valuation of $D_2 \setminus D_1$ and ϑ_3 be a valuation of $D_1 \cap D_2$. Then, $\mathsf{fusion}(\langle\vartheta_1, \vartheta_3\rangle, \langle\vartheta_2, \vartheta_3\rangle) = \langle\mathsf{fusion}(\vartheta_1, \vartheta_2), \vartheta_3\rangle$. □

Proof To prove the observation we consider three following cases:

(1) Consider a datum d belonging to $D_1 \setminus D_2$. Due to the fact that $\vartheta_1 \in \mathsf{DOM}(D_1 \setminus D_2)$ (*see* the formulation of the observation) we obtain:

- on one side, $\mathsf{fusion}(\langle\vartheta_1, \vartheta_3\rangle, \langle\vartheta_2, \vartheta_3\rangle)(d)$ is equal to $\langle\vartheta_1, \vartheta_3\rangle(d)$ (*see* Definition 5.5), which is equal to $\vartheta_1(d)$.
- on the other side, $\langle\mathsf{fusion}(\vartheta_1, \vartheta_2), \vartheta_3\rangle(d)$ is equal to $\mathsf{fusion}(\vartheta_1, \vartheta_2)(d)$, which by Definition 5.5 is equal to $\vartheta_1(d)$.

(2) Similarly, consider a datum d belonging to $D_2 \setminus D_1$. As we know that $\vartheta_2 \in \mathsf{DOM}(D_2 \setminus D_1)$ (*see* the formulation of the observation), then:

$$\mathsf{fusion}(\langle\vartheta_1, \vartheta_3\rangle, \langle\vartheta_2, \vartheta_3\rangle)(d) = \langle\vartheta_2, \vartheta_3\rangle(d) = \vartheta_2(d)$$
$$\langle\mathsf{fusion}(\vartheta_1, \vartheta_2), \vartheta_3\rangle(d) = \mathsf{fusion}(\vartheta_1, \vartheta_2)(d) = \vartheta_2(d)$$

(3) Finally, consider a datum d belonging to $D_1 \cap D_2$. By knowing that $\vartheta_3 \in \mathsf{DOM}(D_2 \cap D_1)$ (*see* the formulation of the observation), we have:

$$\mathsf{fusion}(\langle\vartheta_1, \vartheta_3\rangle, \langle\vartheta_2, \vartheta_3\rangle)(d) = \langle\vartheta_1, \vartheta_3\rangle(d) = \vartheta_3(d)$$
$$\langle\mathsf{fusion}(\vartheta_1, \vartheta_2), \vartheta_3\rangle(d) = \vartheta_3(d)$$

Q.E.D.

Finally, we show that an IOSTS obtained by the product operation from two initialized IOSTS is also initialized.

Lemma 5.4 (Initialized Product) Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two IOSTS with initial conditions $\Theta_1,\ \Theta_2$, sets of symbolic constant $C_1,\ C_2$ and sets of variables $V_1,\ V_2$, such that:

(1) $\mathcal{M}_1$ and $\mathcal{M}_2$ are initialized (*see* Definition 4.19, page 92), and compatible for the product operation (*see* Definition 5.3, page 113)

(2) $\mathcal{M}_2$ is complete with respect to $\mathcal{M}_1$ (*see* Definition 4.22, page 96).

Then, the IOSTS $\mathcal{M}_1 \times \mathcal{M}_2$ obtained from $\mathcal{M}_1$ and $\mathcal{M}_2$ by the product operation is also initialized. □

Proof Consider an instance $\mathcal{M}_1(\varsigma_1)$ (*resp.* $\mathcal{M}_2(\varsigma_2)$) of $\mathcal{M}_1$ (*resp.* $\mathcal{M}_2$), where $\varsigma_1 \in \mathsf{DOM}(C_1)$ (*resp.* $\varsigma_2 \in \mathsf{DOM}(C_2)$). By Definition 4.18 (*see* page 92) $\mathcal{M}_1(\varsigma_1)$ (*resp.* $\mathcal{M}_2(\varsigma_2)$) does not have any symbolic constant.

As $\mathcal{M}_1$ (*resp.* $\mathcal{M}_2$) is initialized, then due to Definition 4.19 (*see* page 92) there exists *at most one* valuation ϑ_1 (*resp.* ϑ_2) of variables V_1 (*resp.* V_2) which satisfies the initial condition Θ_1 (*resp.* Θ_2) of $\mathcal{M}_1(\varsigma_1)$ (*resp.* $\mathcal{M}_2(\varsigma_2)$). Next, consider two following cases:

(1) The valuation ϑ_1 satisfying the initial condition Θ_1 of $\mathcal{M}_1(\varsigma_1)$ does not exist. In this case, it is easy to check that $\mathsf{fusion}(\langle\vartheta_1, \varsigma_1\rangle, \langle\vartheta_2, \varsigma_2\rangle)$ that is a valuation of variables and symbolic constants of $\mathcal{M}_1 \times \mathcal{M}_2$, does not satisfy the initial condition $(\Theta_1 \wedge \Theta_2)$ of $\mathcal{M}_1 \times \mathcal{M}_2$.

The similar result can be obtained in the case, where the valuation ϑ_2 satisfying the initial condition Θ_2 of $\mathcal{M}_2(\varsigma_2)$ does not exist

(2) The valuations ϑ_1 and ϑ_2 satisfying Θ_1 and Θ_2 of $\mathcal{M}_1(\varsigma_1)$ and $\mathcal{M}_2(\varsigma_2)$ respectively exist, and each of these valuations is unique.

As we know that $\mathcal{M}_2$ is complete with respect to $\mathcal{M}_1$, then due to Definition 4.22 (*see* page 96) we obtain that the initial condition Θ_2 of $\mathcal{M}_2$ does not contain constraints over the data shared by $\mathcal{M}_1$ and $\mathcal{M}_2$. Thus, we can choose the values of these data to be equal to their values in ϑ_1 and ς_2. Finally, by applying Observation 5.2 (*see* page 120), we obtain that $\mathsf{fusion}(\langle\vartheta_1, \varsigma_1\rangle, \langle\vartheta_2, \varsigma_2\rangle) \models (\Theta_1 \wedge \Theta_2)$.

The items (1) and (2) imply that the IOSTS $\mathcal{M}_1 \times \mathcal{M}_2$ obtained from $\mathcal{M}_1$ and $\mathcal{M}_2$ is initialized. **Q.E.D.**

5.2.2 Traces of the Product

In this section we consider two IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ compatible for the product operation, and study several relationships between traces/sequences of $\mathcal{M}_1$, $\mathcal{M}_2$ and traces/sequences of the IOSTS $\mathcal{M}$, obtained as the result of the product operation between $\mathcal{M}_1$ and $\mathcal{M}_2$. The main purpose is to prove the equality between the set of traces of $\mathcal{M}_1 \times \mathcal{M}_2$ and the set of traces of $\mathcal{M}_1$, *i.e.*

$$\mathit{Traces}(\mathcal{M}_1 \times \mathcal{M}_2) = \mathit{Traces}(\mathcal{M}_1) \tag{5.12}$$

This equality will be formulated as Theorem 5.5 (*see* page 134). It is important to emphasize that Equality (5.12) holds under additional conditions made on $\mathcal{M}_1$ and $\mathcal{M}_2$ (unlike the similar equality for the operation of parallel composition, (*see* Equality (5.4), page 105), which holds without any additional conditions (*see* Theorem 5.2, page 112)).

The rest of this section consists of two parts, where the first one proves the inclusion of the set of sequences of $\mathcal{M}_1$ into the set of sequences of $\mathcal{M}_1 \times \mathcal{M}_2$, and the second one proves the opposite inclusion, *i.e.* the set of sequences of $\mathcal{M}_1 \times \mathcal{M}_2$ is included into the set of sequences of $\mathcal{M}_1$. We use these inclusions in order to prove the main theorem about trace-equivalence between $\mathcal{M}_1$ and $\mathcal{M}_1 \times \mathcal{M}_2$, which is formulated at the end of the section.

5.2.2.1 Sequences of the Product (First Inclusion).

The purpose of this section is to show that each sequence of $\mathcal{M}_1$ is also a sequence of $\mathcal{M}_1 \times \mathcal{M}_2$. To prove this statement, we first show that if $\mathcal{M}_1$ and $\mathcal{M}_2$ move from a state s_1 (*resp.* s_2) to another state s_1' (*resp.* s_2') by executing a valued action α, then $\mathcal{M}_1 \times \mathcal{M}_2$ moves also from a state s to another state s' by executing the same valued action α. Formally:

Lemma 5.5 Let $\mathcal{M}_1 = \langle D_1, \Theta_1, L_1, l_1^0, \Sigma_1, T_1 \rangle$ and $\mathcal{M}_2 = \langle D_2, \Theta_2, L_2, l_2^0, \Sigma_2, T_2 \rangle$ be IOSTS compatible for the product operation (*see* Definition 5.4, page 114) with sets of states S_1, S_2 and sets of valued actions Λ_1, Λ_2.

Let also $t_1 = \langle l_1, a, \pi_1, G_1, A_1, l_1' \rangle \in T_1$ be a symbolic transition of $\mathcal{M}_1$ and $t_2 = \langle l_2, a, \pi_2, G_2, A_2, l_2' \rangle \in T_2$ be a symbolic transition of $\mathcal{M}_2$.

Then, for all triples:

$$\hat{t}_1 = \langle \underbrace{\langle l_1, \vartheta_1 \rangle}_{s_1}, \underbrace{\langle a, \omega \rangle}_{\alpha}, \underbrace{\langle l_1', \vartheta_1' \rangle}_{s_1'} \rangle \in \rightarrow_{t_1}$$

and

$$\hat{t}_2 = \langle \underbrace{\langle l_2, \vartheta_2 \rangle}_{s_2}, \underbrace{\langle a, \omega \rangle}_{\alpha}, \underbrace{\langle l'_2, \vartheta'_2 \rangle}_{s'_2} \rangle \in \rightarrow_{t_2}$$

of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively, if:

(1) the values of the common variables and symbolic constants of $\mathcal{M}_1$ and $\mathcal{M}_2$ in $\vartheta_2, \vartheta'_2$ are equal to their values in $\vartheta_1, \vartheta'_1$, *i.e.* $\forall\, d \in (V_1 \cup C_1) \cap (V_2 \cup C_2)\ .\ [\vartheta_2(d) = \vartheta_1(d)\ \wedge\ \vartheta'_2(d) = \vartheta'_1]$, and

(2) $\alpha = \langle a, \omega \rangle$ is a valued action of $\mathcal{M}_1$ and $\mathcal{M}_2$, where $\omega \in (\mathsf{DOM}(\pi_1) = \mathsf{DOM}(\pi_2))$;

then, the triple:

$$\hat{t} = \langle \underbrace{\langle l, \mathsf{fusion}(\vartheta_1, \vartheta_2) \rangle}_{s}, \underbrace{\langle a, \omega \rangle}_{\alpha}, \underbrace{\langle l', \mathsf{fusion}(\vartheta'_1, \vartheta'_2) \rangle}_{s'} \tag{5.13}$$

belongs to $\rightarrow_t$ of $\mathcal{M}_1 \times \mathcal{M}_2$, where t is defined by Equation (5.11) of Definition 5.4 (*see* page 114). □

Proof From the hypotheses of the lemma and Definition 4.7 (*see* page 86) we obtain that:

(1) $\hat{t}_1 = \langle \langle l_1, \vartheta_1 \rangle, \langle a, \omega \rangle, \langle l'_1, \vartheta'_1 \rangle \rangle \in \rightarrow_{t_1}$ corresponds to the symbolic transition $t_1 = \langle l_1, a,\ \pi_1, G_1, A_1, l'_1 \rangle$; and the pair of valuations $\langle \vartheta_1, \omega \rangle$ satisfies the guard G_1 of t_1, and

(2) $\hat{t}_2 = \langle \langle l_2, \vartheta_2 \rangle, \langle a, \omega \rangle, \langle l'_2, \vartheta'_2 \rangle \rangle \in \rightarrow_{t_2}$ corresponds to the symbolic transition $t_2 = \langle l_2, a,\ \pi_2, G_2, A_2, l'_2 \rangle$; and the pair of valuations $\langle \vartheta_2, \omega \rangle$ satisfies the guard G_2 of t_2.

By performing the product operation (*see* Definition 5.4, page 114), we obtain the following symbolic transition in $\mathcal{M}_1 \times \mathcal{M}_2$:

$$t = \langle \langle l_1, l_2 \rangle, a, \pi_1, G_1 \wedge G_2[\pi_2/\pi_1], A_1 \cup A_2[\pi_2/\pi_1], \langle l'_1, l'_2 \rangle \rangle$$

Then, we prove that the triple $\hat{t}$ defined by Equation (5.13) belongs to $\rightarrow_t$.

First, as we know that:

(1) the valuations ϑ_1, ϑ'_1 over $(V_1 \cup C_1)$ of $\mathcal{M}_1$, and the valuations ϑ_2, ϑ'_2 over $(V_2 \cup C_2)$ of $\mathcal{M}_2$ exist, and

(2) $(V \cup C) = (V_1 \cup C_1) \cup (V_2 \cup C_2)$ which follows from Definition 5.4 (*see* page 114),

then there exist the valuations $\mathsf{fusion}(\vartheta_1, \vartheta_2)$ and $\mathsf{fusion}(\vartheta'_1, \vartheta'_2)$ over $(V \cup P)$ of $\mathcal{M}_1 \times \mathcal{M}_2$. Therefore, by the definition of a state (*see* Definition 4.4, page 84), $\mathcal{M}_1 \times \mathcal{M}_2$ has the following states: $s = \langle\langle l_1, l_2\rangle, \mathsf{fusion}(\vartheta_1, \vartheta_2)\rangle$ and $s' = \langle\langle l'_1, l'_2\rangle, \mathsf{fusion}(\vartheta'_1, \vartheta'_2)\rangle$.

Second, we prove that the symbolic transition t obtained from t_1 and t_2 by the product operation is executable in $\mathcal{M}_1 \times \mathcal{M}_2$, *i.e.* the pair of valuations $\langle \mathsf{fusion}(\vartheta_1, \vartheta_2), \omega\rangle$ satisfies the guard $(G_1 \wedge G_2[\pi_1/\pi_2])$ of t. Indeed, as we know that:

(1) $\langle \vartheta_1, \omega\rangle \models G_1$,

(2) $\langle \vartheta_2, \omega\rangle \models G_2[\pi_2/\pi_1]$ as:

   (a) $\langle \vartheta_2, \omega\rangle \models G_2$, and

   (b) $G_2[\pi_2/\pi_1]$ is the Boolean expression obtained from G_2 by substitution of all its parameters π_2 carried by the action a of t_2, with the parameters π_1 carried by the same action a of t_1. Remember that $\mathsf{sig}_1(a) = \mathsf{sig}_2(a)$.

(2) $\forall\ d \in (V_1 \cup C_1) \cap (V_2 \cup C_2)\ .\ [\vartheta_2(d) = \vartheta_1(d)]$ (*see* the hypotheses of the lemma),

then by Observation 5.2 (*see* page 120) the valuations $\mathsf{fusion}(\langle \vartheta_1, \omega\rangle, \langle \vartheta_2, \omega\rangle)$ satisfies the conjunction of the Boolean expressions G_1 and $G_2[\pi_2/\pi_1]$. Moreover, due to Lemma 5.3 (*see* page 120) we obtain that $\mathsf{fusion}(\langle \vartheta_1, \omega\rangle, \langle \vartheta_2, \omega\rangle) = \langle \mathsf{fusion}(\vartheta_1, \vartheta_2), \omega\rangle$. Therefore, $\langle \mathsf{fusion}(\vartheta_1, \vartheta_2), \omega\rangle \models (G_1 \wedge G_2[\pi_2/\pi_1])$, and the symbolic transition t of $\mathcal{M}_1 \times \mathcal{M}_2$ is executable.

Third, we prove that the "new" valuation ϑ' of the variables and symbolic constants of $\mathcal{M}_1 \times \mathcal{M}_2$ obtained by execution of the symbolic transition t, is equal to $\mathsf{fusion}(\vartheta'_1, \vartheta'_2)$.

Indeed, ϑ' is obtained from $\vartheta = \mathsf{fusion}(\vartheta_1, \vartheta_2)$ by parallel execution of the assignments belonging to $(A_1 \cup A_2[\pi_2/\pi_1])$. More precisely, ϑ' is constructed as follows: for each datum $d \in (V_1 \cup C_1) \cup (V_2 \cup C_2)$ of $\mathcal{M}_1 \times \mathcal{M}_2$,

$$\vartheta'(d) = \begin{cases} A_1(\langle \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta), \omega\rangle)(d) & \textbf{if} \quad d \in V_1 \\ A_2[\pi_2/\pi_1](\langle \mathsf{projection}_{(V_2 \cup C_2)}(\vartheta), \omega\rangle)(d) & \textbf{if} \quad d \in V_2 \\ \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta)(d) & \textbf{if} \quad d \in C_1 \setminus (V_2 \cup C_2) \\ \mathsf{projection}_{(V_2 \cup C_2)}(\vartheta)(d) & \textbf{if} \quad d \in C_2 \setminus (V_1 \cup C_1) \\ \vartheta(d) & \textbf{if} \quad d \in C_1 \cap C_2 \end{cases} \tag{5.14}$$

Remember that (1) $V_1 \cap C_1 = \emptyset$ and $V_2 \cap C_2 = \emptyset$ due to the definition of IOSTS (*see* page 81), and (2) $V_1 \cap V_2 = \emptyset$ as $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for the product operation.

Then, due to Definitions 5.5 and 5.6 (*see* pages 119) we have that:

(a) $\mathsf{projection}_{(V_1 \cup C_1)}(\underbrace{\mathsf{fusion}(\vartheta_1, \vartheta_2)}_{\vartheta}) = \vartheta_1$, and

(b) using the supplementary condition formulated as the lemma hypothesis, and saying that all common data of $\mathcal{M}_1$ and $\mathcal{M}_2$ have the same values, *i.e.* $\forall\, d \in (V_1 \cup C_1) \cap (V_2 \cup C_2)\, .\, [\vartheta_2(d) = \vartheta_1(d)]$, we have that $\mathsf{projection}_{(V_2 \cup C_2)}(\underbrace{\mathsf{fusion}(\vartheta_1, \vartheta_2)}_{\vartheta}) = \vartheta_2$.

Moreover, if a common datum d of $\mathcal{M}_1$, $\mathcal{M}_2$ is a constant in both IOSTS, then (1) d is a constant in $\mathcal{M}_1 \times \mathcal{M}_2$, and (2) its value cannot be change during execution of $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_1 \times \mathcal{M}_2$. Next, due to the lemma hypothesis saying that: $\forall\, d \in (V_1 \cup C_1) \cap (V_2 \cup C_2)\, .\, [\vartheta_2(d) = \vartheta_1(d)]$, we obtain:

$$\forall\, d \in (C_1 \cup C_2)\, .\, [\vartheta_1(d) = \vartheta_2(d) = \vartheta(d)]$$

Using this statement together with the items (a) and (b) above, we can rewrite Formula (5.14) as follows: for each datum d of $\mathcal{M}_1 \times \mathcal{M}_2$,

$$\vartheta'(d) = \begin{cases} A_1(\langle \vartheta_1, \omega \rangle)(d) & \textbf{if} \quad d \in V_1 \\ A_2[\pi_2/\pi_1](\langle \vartheta_2, \omega \rangle)(d) & \textbf{if} \quad d \in V_2 \\ \vartheta_1(d) & \textbf{if} \quad d \in C_1 \setminus (V_2 \cup C_2) \\ \vartheta_2(d) & \textbf{if} \quad d \in C_2 \setminus (V_1 \cup C_1) \\ \vartheta(d) = \vartheta_1(d) = \vartheta_2(d) & \textbf{if} \quad d \in C_1 \cap C_2 \end{cases} \tag{5.15}$$

We also know that: for each datum $d \in (V_1 \cup C_1)$ of $\mathcal{M}_1$,

$$\vartheta'_1(d) = \begin{cases} A_1(\langle \vartheta_1, \omega \rangle)(d) & \textbf{if} \quad d \in V_1 \\ \vartheta_1(d) & \textbf{if} \quad d \in C_1 \end{cases} \tag{5.16}$$

and for each datum $d \in (V_2 \cup C_2)$ of $\mathcal{M}_2$,

$$\vartheta'_2(d) = \begin{cases} A_2(\langle \vartheta_2, \omega \rangle)(d) & \textbf{if} \quad d \in V_2 \\ \vartheta_2(d) & \textbf{if} \quad d \in C_2 \end{cases} \tag{5.17}$$

Next, we remark that $A_2[\pi_2/\pi_1](\langle \vartheta_2, \omega \rangle) = A_2(\langle \vartheta_2, \omega \rangle)$. Indeed, as signature of the actions a labeling symbolic transitions t_1 and t_2, then we can substitute π_2 by π_1 in the expression A_2. This substitution does not change the semantics of the expression A_2.

Using this observation together with Formulas (5.15), (5.16) and (5.17), we obtain: for each datum d of $\mathcal{M}_1 \times \mathcal{M}_2$,

$$\vartheta'(d) = \begin{cases} \vartheta'_1(d) & \textbf{if} \quad d \in (V_1 \cup C_1) \\ \vartheta'_2(d) & \textbf{otherwise} \end{cases} \tag{5.18}$$

Finally, from Formula (5.18) and Definition 5.5 (*see* page 119) we get that $\vartheta' = \mathsf{fusion}(\vartheta'_1, \vartheta'_2)$.

Therefore, the triple $\hat{t} = \langle s, \alpha, s' \rangle$ of $\mathcal{M}_1 \times \mathcal{M}_2$, where $s = \langle \langle l_1, l_2 \rangle, \mathsf{fusion}(\vartheta_1, \vartheta_2) \rangle$ and $s' = \langle \langle l'_1, l_2 \rangle, \mathsf{fusion}(\vartheta'_1, \vartheta_2) \rangle$, belongs to $\rightarrow_t$ of $\mathcal{M}_1 \times \mathcal{M}_2$.

Q.E.D.

Remind that the aim of this subsection is to prove that if a given IOSTS $\mathcal{M}_1$ can fire a sequence η of valued actions, then the IOSTS $\mathcal{M}_1 \times \mathcal{M}_2$, where $\mathcal{M}_2$ is a complete IOSTS with respect to $\mathcal{M}_1$, can fire this sequence as well.

In order to prove this statement, we first show that if there exists some behavior of $\mathcal{M}_1$ corresponding to a sequence η belonging to $Sequences(\mathcal{M}_1)$ (*see* Definition 4.12, page 90), then there exists a behavior of $\mathcal{M}_1 \times \mathcal{M}_2$ corresponding to the same sequence η. It is important to notice that this statement holds under some additional condition made on $\mathcal{M}_2$.

Theorem 5.3 (Behaviors of the Product) Consider two IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ such that:

(1) $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for the product operation (*see* Definition 5.3, page 113), and

(2) $\mathcal{M}_2$ is complete with respect to $\mathcal{M}_1$ (*see* Definition 4.22, page 96).

If $\mathcal{M}_1$ has the following behavior $\beta_1^{\rightarrow} : \langle l_1^0, \vartheta_1^0 \rangle \xrightarrow{\alpha_1} \langle l_1^1, \vartheta_1^1 \rangle \ldots \langle l_1^{n-1}, \vartheta_1^{n-1} \rangle \xrightarrow{\alpha_n} \langle l_1^n, \vartheta_1^n \rangle$ which corresponds to a sequence $\eta = \alpha_1 \ldots \alpha_n$ belonging to $Sequences(\mathcal{M}_1)$. Then, there exists a behavior:

$$\beta^{\rightarrow} : \langle l^0, \vartheta^0 \rangle \xrightarrow{\alpha_1} \langle l^1, \vartheta^1 \rangle \ldots \langle l^{n-1}, \vartheta^{n-1} \rangle \xrightarrow{\alpha_n} \langle l^n, \vartheta^n \rangle$$

in $\mathcal{M}_1 \times \mathcal{M}_2$ corresponding to the same sequence η, where for all i from 0 to n, there exist a location $l_2^i \in L_2$ of $\mathcal{M}_2$ and a valuation $\vartheta_2^i \in \mathsf{DOM}(V_2 \cup C_2)$ of variables and symbolic constants of $\mathcal{M}_2$ such that:

– $l^i = \langle l_1^i, l_2^i \rangle$,

– $\vartheta^i = \mathsf{fusion}(\vartheta_1^i, \vartheta_2^i)$, and

– $\forall\ d \in (V_1 \cup C_1) \cap (V_2 \cup C_2)\ .\ [\vartheta_2^i(d) = \vartheta_1^i(d)]$.

□

Proof The proof of this theorem is done by induction on the length of the behavior $\beta_1^{\rightarrow}$ of $\mathcal{M}_1$.

Induction Basis. Consider a behavior $\beta_1^{\rightarrow} : \langle l_1^0, \vartheta_1^0 \rangle$ of length zero corresponding to the sequence $\eta \in \mathit{Sequences}(\mathcal{M}_1)$ of length zero and prove that $\mathcal{M}_1 \times \mathcal{M}_2$ has a behavior $\beta^{\rightarrow} : \langle l^0, \vartheta^0 \rangle$ corresponding η.

First, as $\beta_1^{\rightarrow} : \langle l_1^0, \vartheta_1^0 \rangle$ is a behavior of $\mathcal{M}_1$, then the valuation ϑ_1^0 of variables and symbolic constants of $\mathcal{M}_1$ satisfies the initial condition Θ_1 of $\mathcal{M}_1$, *i.e.* $\vartheta_1^0 \models \Theta_1$.

Second, consider some valuation ϑ_2^0 of the variables and symbolic constants of $\mathcal{M}_2$ satisfying the initial condition Θ_2, *i.e.* $\vartheta_2^0 \models \Theta_2$ and associate it with the initial location l_2^0 of $\mathcal{M}_2$. As the initial condition Θ_2 of $\mathcal{M}_2$ does not contain constraints over the data shared by $\mathcal{M}_1$ and $\mathcal{M}_2$ (*see* Definition 4.22, page 96), then we can choose the values of these data to be equal to their values in ϑ_1^0, *i.e.* $\forall\ d \in (V_1 \cup C_1) \cap (V_2 \cup C_2)\ .\ [\vartheta_2^0(d) = \vartheta_1^0(d)]$. Therefore, we constructed a behavior $\beta_2^{\rightarrow} : \langle l_2^0, \vartheta_2^0 \rangle$ of $\mathcal{M}_2$.

Finally, we prove that $\beta^{\rightarrow} : \langle l^0, \vartheta^0 \rangle$, where $l^0 = \langle l_1^0, l_2^0 \rangle$ and $\vartheta^0 = \mathsf{fusion}(\vartheta_1^0, \vartheta_2^0)$, is a behavior of $\mathcal{M}_1 \times \mathcal{M}_2$ corresponding to the sequence η.

(1) $l^0 = \langle l_1^0, l_2^0 \rangle$ is the initial location of $\mathcal{M}_1 \times \mathcal{M}_2$ (*see* Definition 5.4, page 114),

(2) $\vartheta^0 = \mathsf{fusion}(\vartheta_1^0, \vartheta_2^0)$ satisfies the initial condition $\Theta_1 \wedge \Theta_2$ of $\mathcal{M}_1 \times \mathcal{M}_2$. This statement is true, as by knowing that:

   – $(V \cup C) = (V_1 \cup C_1) \cup (V_2 \cup C_2)$ which follows from Definition 5.4 (*see* page 114),

   – the valuation ϑ_1^0 (*resp.* ϑ_2^0) of the variables and symbolic constants of $\mathcal{M}_1$ (*resp.* $\mathcal{M}_2$) satisfies the initial condition Θ_1 (*resp.* Θ_2) of $\mathcal{M}_1$ (*resp.* $\mathcal{M}_2$), and

   – $\forall\ d \in (V_1 \cup C_1) \cap (V_2 \cup C_2)\ .\ [\vartheta_2^0(d) = \vartheta_1^0(d)]$ (*see* the paragraph above).

   we can apply Observation 5.2 (*see* page 120), and obtain $\mathsf{fusion}(\vartheta_1^0, \vartheta_2^0) \models \Theta_1 \wedge \Theta_2$.

Therefore, we have proved that $\beta^{\rightarrow} : \langle l^0, \vartheta^0 \rangle$ is a behavior of $\mathcal{M}_1 \times \mathcal{M}_2$. By Definition 4.12 (*see* page 90) $\beta^{\rightarrow}$ corresponds to a sequence η of length zero.

Induction Hypothesis. Assume that $\mathcal{M}_1$ has the following behavior of length $n-1$:

$$\beta_1^{\rightarrow} : \langle l_1^0, \vartheta_1^0 \rangle \xrightarrow{\alpha_1} \langle l_1^1, \vartheta_1^1 \rangle \ldots \langle l_1^{n-2}, \vartheta_1^{n-2} \rangle \xrightarrow{\alpha_{n-1}} \langle l_1^{n-1}, \vartheta_1^{n-1} \rangle$$

corresponding to the sequence $\eta' = \alpha_1 \ldots \alpha_{n-1} \in \mathit{Sequences}(\mathcal{M}_1)$. Then, there exists a behavior of length $n-1$ in $\mathcal{M}_1 \times \mathcal{M}_2$:

$$\beta^{\rightarrow} : \langle l^0, \vartheta^0 \rangle \xrightarrow{\alpha_1} \langle l^1, \vartheta^1 \rangle \ldots \langle l^{n-2}, \vartheta^{n-2} \rangle \xrightarrow{\alpha_{n-1}} \langle l^{n-1}, \vartheta^{n-1} \rangle$$

corresponding to the same sequence η', where $\forall\ i = 0..n\ \exists\ l_2^i \in L_2, \vartheta_2^i \in \mathsf{DOM}(V_2 \cup C_2)\ .\ [l^i = \langle l_1^i, l_2^i \rangle \wedge \vartheta^i = \mathsf{fusion}(\vartheta_1^i, \vartheta_2^i) \wedge (\forall\ d \in (V_1 \cup C_1) \cap (V_2 \cup C_2)\ .\ [\vartheta_2^i(d) = \vartheta_1^i(d)])]$.

Induction Step. Consider a behavior of length n in $\mathcal{M}_1$ which corresponds to the sequence $\eta = \underbrace{\alpha_1 \ldots \alpha_{n-1}}_{\eta'} \alpha_n \in \mathit{Sequences}(\mathcal{M}_1)$, *i.e.*

$$\beta_1^{\rightarrow} : \langle l_1^0, \vartheta_1^0 \rangle \xrightarrow{\alpha_1} \langle l_1^1, \vartheta_1^1 \rangle \ldots \langle l_1^{n-1}, \vartheta_1^{n-1} \rangle \xrightarrow{\alpha_n} \langle l_1^n, \vartheta_1^n \rangle \tag{5.19}$$

First, by using prefix closure together with the induction hypothesis, we obtain that in $\mathcal{M}_1 \times \mathcal{M}_2$ there exists a behavior of length $n-1$:

$$\beta^{\rightarrow} : \langle l^0, \vartheta^0 \rangle \xrightarrow{\alpha_1} \langle l^1, \vartheta^1 \rangle \ldots \langle l^{n-2}, \vartheta^{n-2} \rangle \xrightarrow{\alpha_{n-1}} \langle l^{n-1}, \vartheta^{n-1} \rangle$$

corresponding to η', where $\forall\ i = 0..n-1\ \exists\ l_2^i \in L_2, \vartheta_2^i \in \mathsf{DOM}(V_2 \cup C_2)\ .\ [l^i = \langle l_1^i, l_2^i \rangle \wedge \vartheta^i = \mathsf{fusion}(\vartheta_1^i, \vartheta_2^i) \wedge (\forall\ d \in (V_1 \cup C_1) \cap (V_2 \cup C_2)\ .\ [\vartheta_2^i(d) = \vartheta_1^i(d)])]$.

Therefore, we have that:

(1) there exist
   - a location $l_2^{n-1} \in L_2$ in $\mathcal{M}_2$, and
   - a valuation $\vartheta_2^{n-1} \in \mathsf{DOM}(V_2 \cup C_2)$ in $\mathcal{M}_2$, such that the values of variables and symbolic constants shared by $\mathcal{M}_1$ and $\mathcal{M}_2$ are the same as in ϑ_1^{n-1},

(2) the state $s^{n-1} = \langle \langle l_1^{n-1}, l_2^{n-1} \rangle, \mathsf{fusion}(\vartheta_1^{n-1}, \vartheta_2^{n-1}) \rangle$ of $\mathcal{M}_1 \times \mathcal{M}_2$ is reachable by $\eta' = \alpha_1 \ldots \alpha_{n-1}$ (*see* Definition 4.15, page 91).

Second, we consider the last step of the behavior shown as Equation (5.19), *i.e.* the triple:

$$\langle l_1^{n-1}, \vartheta_1^{n-1} \rangle \xrightarrow{\alpha_n = \langle a, \omega \rangle} \langle l_1^n, \vartheta_1^n \rangle$$

This triple, due to Definitions 4.8 and 4.7 (*see* pages 87 and 86), corresponds to a symbolic transition $t_1^n = \langle l_1^{n-1}, a, \pi_1, G_1^n, A_1^n, l_1^n \rangle$; and the pair of valuations $\langle \vartheta_1^{n-1}, \omega \rangle$ satisfies the guard G_1^n of t_1^n.

Next, as $\mathcal{M}_2$ is complete with respect to $\mathcal{M}_1$, then using Item (4) of Definition 4.22 (*see* page 96), we get that $\mathcal{M}_2$ has the following symbolic transitions leaving the location $l_2^{n-1} \in L_2$ and labeled with the action a:

$$\begin{aligned}
& t_2^{n^1} = \langle l_2^{n-1}, a, \pi_2, G_2^{n^1}, A_2^{n^1}, l_2^{n^1} \rangle, \\
& \ldots \\
& t_2^{n^i} = \langle l_2^{n-1}, a, \pi_2, G_2^{n^i}, A_2^{n^i}, l_2^{n^i} \rangle, \\
& \ldots \\
& t_2^{n^k} = \langle l_2^{n-1}, a, \pi_2, G_2^{n^k}, A_2^{n^k}, l_2^{n^k} \rangle.
\end{aligned}$$

where $k \geq 1$. As we also know that $G_2^{n^1} \vee \ldots \vee G_2^{n^k}$ evaluates to $true$, then the pair of valuations $\langle \vartheta_2^{n-1}, \omega \rangle$ satisfies at least one of these guards. Suppose that $\langle \vartheta_2^{n-1}, \omega \rangle$ satisfies the guard of the i-th symbolic transition ($i = 1..n$), *i.e.* $\langle \vartheta_2^{n-1}, \omega \rangle \models G_2^{n^i}$. Then, there exists the valuation $\vartheta_2^{n^i} \in \mathsf{DOM}(V_2 \cup C_2)$ which is obtained from ϑ_2^{n-1} by executing in parallel the assignments $A_2^{n^i}$ of the symbolic transition $t_2^{n^i}$. Therefore, by Definition 4.7 (*see* page 86) the triple $\langle l_2^{n-1}, \vartheta_2^{n-1} \rangle \xrightarrow{\alpha_n = \langle a, \omega \rangle} \langle l_2^{n^i}, \vartheta_2^{n^i} \rangle$ belongs to $\rightarrow_{t_2^{n^i}}$.

Finally, as we know that:

(1) $\langle l_1^{n-1}, \vartheta_1^{n-1} \rangle \xrightarrow{\alpha_n = \langle a, \omega \rangle} \langle l_1^n, \vartheta_1^n \rangle \in \rightarrow_{t_1^n}$,

(2) $\langle l_2^{n-1}, \vartheta_2^{n-1} \rangle \xrightarrow{\alpha_n = \langle a, \omega \rangle} \langle l_2^{n^i}, \vartheta_2^{n^i} \rangle \in \rightarrow_{t_2^{n^i}}$, and

(3) $\forall\, d \in (V_1 \cup C_1) \cap (V_2 \cup C_2)\ .\ [\vartheta_2^{n-1}(d) = \vartheta_1^{n-1}(d)]$,

then using Lemma 5.5 (*see* page 122) we obtain that the triple:

$$\langle \langle l_1^{n-1}, l_2^{n-1} \rangle, \mathsf{fusion}(\vartheta_1^{n-1}, \vartheta_2^{n-1}) \rangle \xrightarrow{\alpha_n = \langle a, \omega \rangle} \langle \langle l_1^n, l_2^{n^i} \rangle, \mathsf{fusion}(\vartheta_1^n, \vartheta_2^{n^i}) \rangle$$

belongs to $\rightarrow_{t^n}$ of $\mathcal{M}_1 \times \mathcal{M}_2$, where t^n is the symbolic transition of $\mathcal{M}_1 \times \mathcal{M}_2$ obtained by Equation (5.11) of Definition 5.4 (*see* page 114) from the symbolic transitions t_1^n and $t_2^{n^i}$ of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively.

Therefore, we have proved there exists the behavior:

$$\beta^{\rightarrow} : \langle l^0, \vartheta^0 \rangle \stackrel{\alpha_1}{\rightarrow} \langle l^1, \vartheta^1 \rangle \ldots \langle l^{n-1}, \vartheta^{n-1} \rangle \stackrel{\alpha_n}{\rightarrow} \langle l^n, \vartheta^n \rangle$$

in $\mathcal{M}_1 \times \mathcal{M}_2$ corresponding to the same sequence η, where for all i from 0 to n, there exist a location $l_2^i \in L_2$ of $\mathcal{M}_2$ and a valuation $\vartheta_2^i \in \mathsf{DOM}(V_2 \cup C_2)$ of variables and symbolic constants of $\mathcal{M}_2$ such that:

- $l^i = \langle l_1^i, l_2^i \rangle$,
- $\vartheta^i = \mathsf{fusion}(\vartheta_1^i, \vartheta_2^i)$, and
- $\forall\ d \in (V_1 \cup C_1) \cap (V_2 \cup C_2)\ .\ [\vartheta_2^i(d) = \vartheta_1^i(d)]$.

Q.E.D.

The inclusion of the set of sequences of $\mathcal{M}_1$ into the set of sequences of $\mathcal{M}_1 \times \mathcal{M}_2$ follows directly from Theorem 5.3 and Definition 4.12 (*see* page 4.12). This statement is formulated as the corollary bellow.

Corollary 5.1 (Sequences of the Product) Consider two IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ such that:

(1) $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for the product operation (*see* Definition 5.3, page 113), and

(2) $\mathcal{M}_2$ is complete with respect to $\mathcal{M}_1$ (*see* Definition 4.22, page 96).

the following inclusion holds:

$$\mathit{Sequences}(\mathcal{M}_1) \subseteq \mathit{Sequences}(\mathcal{M}_1 \times \mathcal{M}_2) \tag{5.20}$$

□

5.2.2.2 Sequences of the Product (Second Inclusion).

The aim of the section is to prove that each sequence of $\mathcal{M}_1 \times \mathcal{M}_2$ is also a sequence of $\mathcal{M}_1$. At the beginning we show that if $\mathcal{M}_1 \times \mathcal{M}_2$, where $\mathcal{M}_2$ is complete with respect to $\mathcal{M}_1$, moves from a state s to a state s' by executing a valued action α, then $\mathcal{M}_1$ moves from a state s_1 to another state s_1' by executing the same valued action α. Formally:

Lemma 5.6 Let $\mathcal{M}_1 = \langle D_1, \Theta_1, L_1, l_1^0, \Sigma_1, T_1 \rangle$ and $\mathcal{M}_2 = \langle D_2, \Theta_2, L_2, l_2^0, \Sigma_2, T_2 \rangle$ be two IOSTS with set of states S_1, S_2 and set of valued actions Λ_1, Λ_2, such that:

(1) $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for the product operation (*see* Definition 5.3, page 113), and

(2) $\mathcal{M}_2$ is complete with respect to $\mathcal{M}_1$ (*see* Definition 4.22, page 96).

Let also $t = \langle\langle l_1, l_2\rangle, a, \pi_1, G_1 \wedge G_2[\pi_2/\pi_1], A_1 \cup A_2[\pi_2/\pi_1], \langle l'_1, l'_2\rangle\rangle$ be a symbolic transition of $\mathcal{M}_1 \times \mathcal{M}_2$, which is obtained from two symbolic transitions $t_1 = \langle l_1, a, \pi_1, G_1, A_1, l'_1\rangle$ and $t_2 = \langle l_2, a, \pi_2, G_2, A_2, l'_2\rangle$ of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively by Equation 5.11 of Definition 5.4 (*see* page 114).

Then, for each triple $\hat{t} = \langle \underbrace{\langle\langle l_1, l_2\rangle, \vartheta\rangle}_{s}, \underbrace{\langle a, \omega\rangle}_{\alpha}, \underbrace{\langle\langle l'_1, l'_2\rangle, \vartheta'\rangle}_{s'} \rangle$ belonging to $\rightarrow_t$, there exists a triple $\hat{t}_1 = \langle \underbrace{\langle l_1, \vartheta_1\rangle}_{s_1}, \underbrace{\langle a, \omega\rangle}_{\alpha}, \underbrace{\langle l'_1, \vartheta'_1\rangle}_{s'_1} \rangle$ belonging to $\rightarrow_{t_1}$, where ϑ_1 and ϑ'_1 are obtained from ϑ and ϑ' by projection onto $(V_1 \cup C_1)$, *i.e.* $\vartheta_1 = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta)$ and $\vartheta'_1 = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta')$. □

Proof

First, from the hypothesis of the lemma we have that the given triple $\hat{t}$ belonging to $\rightarrow_t$ corresponds to $t = \langle\langle l_1, l_2\rangle, a, \pi_1, G_1 \wedge G_2[\pi_2/\pi_1], A_1 \cup A_2[\pi_2/\pi_1], \langle l'_1, l'_2\rangle\rangle$. Then, from Definition 4.7 (*see* page 86) we obtain that the pair of valuation $\langle\vartheta, \omega\rangle$ satisfies the guard $G_1 \wedge G_2[\pi_2/\pi_1]$ of t.

Second, from the hypothesis of the lemma we also have that $\mathcal{M}_1$ has a transition $t_1 = \langle l_1, a, \pi_1, G_1, A_1, l'_1\rangle$. Then, we prove that $\mathcal{M}_1$ can move from the state $s_1 = \langle l_1, \vartheta_1\rangle$ to the state $s'_1 = \langle l'_1, \vartheta'_1\rangle$ by executing the valued action $\alpha = \langle a, \omega\rangle$.

Indeed, if we choose ϑ_1 to be equal to $\mathsf{projection}_{(V_1 \cup C_1)}(\vartheta)$ (*see* Definition 5.6, page 119) then as $\langle\vartheta, \omega\rangle \models G_1 \wedge G_2[\pi_2/\pi_1]$ we can use Observation 5.3 (*see* page 120) and obtain that the pair $\langle\vartheta_1, \omega\rangle$ satisfies G_1. Then, we compute the new valuation of variables and symbolic constants of $\mathcal{M}_1$, *i.e.* ϑ'_1, from ϑ_1 by executing in parallel the assignments A_1. Due to the fact that symbolic constants of $\mathcal{M}_1$ cannot be variables of $\mathcal{M}_2$, *i.e.* $C_1 \cap V_2 = \emptyset$ (*see* Item (1) of Definition 4.22, page 96), we obtain that the values of variables and symbolic constants in ϑ'_1 are exactly the same as their values in ϑ'. Thus, $\vartheta'_1 = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta')$.

Therefore, we have proved that there exists a triple $\hat{t}_1 = \langle \underbrace{\langle l_1, \vartheta_1\rangle}_{s_1}, \underbrace{\langle a, \omega\rangle}_{\alpha}, \underbrace{\langle l'_1, \vartheta'_1\rangle}_{s'_1} \rangle$ belonging to $\rightarrow_{t_1}$ of $\mathcal{M}_1$, where $\vartheta_1 = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta)$ and $\vartheta'_1 = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta')$.

Q.E.D.

Next, we prove that if there exists some behaviour of $\mathcal{M}_1 \times \mathcal{M}_2$ corresponding to a sequence η belonging to *Sequences*$(\mathcal{M}_1 \times \mathcal{M}_2)$ (*see* Definition 4.12, page 90), then there exists a behaviour of $\mathcal{M}_1$ corresponding to the same sequence η. This statement holds under some additional condition made on $\mathcal{M}_2$.

Theorem 5.4 (Behaviors of the Product) Consider two IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ such that:

(1) $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for the product operation (*see* Definition 5.3, page 113), and

(2) $\mathcal{M}_2$ is complete with respect to $\mathcal{M}_1$ (*see* Definition 4.22, page 96).

If $\mathcal{M}_1 \times \mathcal{M}_2$ has the behavior:

$$\beta^{\rightarrow} : \langle l^0, \vartheta^0 \rangle \xrightarrow{\alpha_1} \langle l^1, \vartheta^1 \rangle \ldots \langle l^{n-1}, \vartheta^{n-1} \rangle \xrightarrow{\alpha_n} \langle l^n, \vartheta^n \rangle$$

where for all i from 0 to n, there exist locations $l_1^i \in L_1$ of $\mathcal{M}_1$ and $l_2^i \in L_2$ of $\mathcal{M}_2$ such that $l^i = \langle l_1^i, l_2^i \rangle$. Moreover, this behavior corresponds to a sequence $\eta = \alpha_1 \ldots \alpha_n$ belonging to *Sequences*$(\mathcal{M}_1 \times \mathcal{M}_2)$.

Then, $\mathcal{M}_1$ has the behavior:

$$\beta_1^{\rightarrow} : \langle l_1^0, \vartheta_1^0 \rangle \xrightarrow{\alpha_1} \langle l_1^1, \vartheta_1^1 \rangle \ldots \langle l_1^{n-1}, \vartheta_1^{n-1} \rangle \xrightarrow{\alpha_n} \langle l_1^n, \vartheta_1^n \rangle$$

corresponding to the sequence η and to the locations l_2^i $(i = 1..n)$ of $\mathcal{M}_2$. Here, for all i from 0 to n, $\vartheta_1^i = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta^i)$ (*see* Definition 5.6, page 119). □

Proof This theorem is proved by induction on the length of the behavior $\beta^{\rightarrow}$ of $\mathcal{M}_1 \times \mathcal{M}_2$.

Induction Basis. Consider a behavior $\beta^{\rightarrow} : \langle l^0, \vartheta^0 \rangle$ of length zero, where $l^0 = \langle l_1^0, l_2^0 \rangle$, corresponding to the empty sequence $\eta \in$ *Sequences*$(\mathcal{M}_1 \times \mathcal{M}_2)$; and prove that $\mathcal{M}_1$ has a behavior $\beta_1^{\rightarrow} : \langle l_1^0, \vartheta_1^0 \rangle$ corresponding to η, where $\vartheta_1^0 = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta^0)$.

Indeed, as we know that $s^0 = \langle l^0, \vartheta^0 \rangle$ is the initial state of $\mathcal{M}_1 \times \mathcal{M}_2$, then by Definition 4.5 (*see* page 84) ϑ^0 must satisfy the initial condition of $\mathcal{M}_1 \times \mathcal{M}_2$, *i.e.* $\vartheta^0 \models \Theta_1 \wedge \Theta_2$. Then, as we know that:

- $(V \cup C) = (V_1 \cup C_1) \cup (V_2 \cup C_2)$ which follows from Definition 5.4 (*see* page 114), and
- $\vartheta_1^0 = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta^0)$,

then, from Observation 5.3 (*see* page 120) we obtain that ϑ_1^0 satisfies Θ_1. Therefore, we have constructed the behavior $\beta_1^{\rightarrow} : \langle l_1^0, \vartheta_1^0 \rangle$ of $\mathcal{M}_1$, which by Definition 4.12 correspond to the sequence η of length zero.

Induction Hypothesis. Assume that $\mathcal{M}_1 \times \mathcal{M}_2$ has the following behavior of length $n-1$:

$$\beta^{\rightarrow} : \langle l^0, \vartheta^0 \rangle \stackrel{\alpha_1}{\rightarrow} \langle l^1, \vartheta^1 \rangle \ldots \langle l^{n-2}, \vartheta^{n-2} \rangle \stackrel{\alpha_{n-1}}{\rightarrow} \langle l^{n-1}, \vartheta^{n-1} \rangle$$

corresponding to the sequence $\eta' = \alpha_1 \ldots \alpha_{n-1} \in \mathit{Sequences}(\mathcal{M}_1)$, where $\forall\ i = 0..n-1\ \exists\ l_1^i \in L_1, l_2^i \in L_2\ .\ [l^i = \langle l_1^i, l_2^i \rangle]$.

Then, $\mathcal{M}_1$ contains the behavior:

$$\beta_1^{\rightarrow} : \langle l_1^0, \vartheta_1^0 \rangle \stackrel{\alpha_1}{\rightarrow} \langle l_1^1, \vartheta_1^1 \rangle \ldots \langle l_1^{n-2}, \vartheta_1^{n-2} \rangle \stackrel{\alpha_{n-1}}{\rightarrow} \langle l_1^{n-1}, \vartheta_1^{n-1} \rangle$$

corresponding to the sequence η' and to the locations l_2^i $(i = 1..n-1)$ of $\mathcal{M}_2$. Here, $\forall\ i = 0..n-1\ .\ [\vartheta_1^i = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta^i)]$.

Induction Step. Consider a behavior of length n:

$$\beta^{\rightarrow} : \langle l^0, \vartheta^0 \rangle \stackrel{\alpha_1}{\rightarrow} \langle l^1, \vartheta^1 \rangle \ldots \langle l^{n-1}, \vartheta^{n-1} \rangle \stackrel{\alpha_n}{\rightarrow} \langle l^n, \vartheta^n \rangle \tag{5.21}$$

of $\mathcal{M}_1 \times \mathcal{M}_2$ which corresponds to a sequence $\eta = \underbrace{\alpha_1 \ldots \alpha_{n-1}}_{\eta'} \alpha_n \in \mathit{Sequences}(\mathcal{M}_1 \times \mathcal{M}_2)$, where $\forall\ i = 0..n\ \exists\ l_1^i \in L_1, l_2^i \in L_2\ .\ [l^i = \langle l_1^i, l_2^i \rangle]$.

First, by using prefix closure together with the induction hypothesis, we obtain that $\mathcal{M}_1$ has the behavior:

$$\beta_1^{\rightarrow} : \langle l_1^0, \vartheta_1^0 \rangle \stackrel{\alpha_1}{\rightarrow} \langle l_1^1, \vartheta_1^1 \rangle \ldots \langle l_1^{n-2}, \vartheta_1^{n-2} \rangle \stackrel{\alpha_{n-1}}{\rightarrow} \langle l_1^{n-1}, \vartheta_1^{n-1} \rangle$$

corresponding to the sequence η' and to the locations l_2^i $(i = 1..n-1)$ of $\mathcal{M}_2$. Here, $\forall\ i = 0..n-1\ .\ [\vartheta_1^i = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta^i)]$. Thus, the state $s_1^{n-1} = \langle l_1^{n-1}, \vartheta_1^{n-1} \rangle$ of $\mathcal{M}_1$ is reachable by $\eta' = \alpha_1 \ldots \alpha_{n-1}$ (*see* Definition 4.15, page 91).

Second, we consider the last step of the behavior shown as Equation (5.21), *i.e.* $\langle l^{n-1}, \vartheta^{n-1} \rangle \stackrel{\alpha_n}{\rightarrow} \langle l^n, \vartheta^n \rangle$, which, due to Definitions 4.8 and 4.7 (*see* pages 87 and 86), corresponds to a simbolic transition:

$$t^n = \langle \langle l_1^{n-1}, l_2^{n-1} \rangle, a, \pi_1, G_1^n \wedge G_2^n[\pi_2/\pi_1], A_1^n \cup A_2^n[\pi_2/\pi_1], \langle l_1^n, l_2^n \rangle \rangle$$

Then, from Lemma 5.6 (*see* page 130) we get that $\mathcal{M}_1$ can move from the state $s_1^{n-1} = \langle l_1^{n-1}, \vartheta_1^{n-1} \rangle$ to the state $s_1^n = \langle l_1^n, \vartheta_1^n \rangle$ by executing the

valued action $\alpha_n = \langle a, \omega \rangle$. Moreover, $\vartheta_1^{n-1} = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta^{n-1})$ and $\vartheta_1^n = \mathsf{projection}_{(V_1 \cup C_1)}(\vartheta^n)$.

Therefore, we have proved $\mathcal{M}_1$ has the behavior:

$$\beta_1^{\rightarrow} : \langle l_1^0, \vartheta_1^0 \rangle \xrightarrow{\alpha_1} \langle l_1^1, \vartheta_1^1 \rangle \ldots \langle l_1^{n-1}, \vartheta_1^{n-1} \rangle \xrightarrow{\alpha_n} \langle l_1^n, \vartheta_1^n \rangle$$

corresponding to the same sequence η and to the locations l_2^i $(i = 1..n)$ of $\mathcal{M}_2$. Here, where for all i from 1 to n, the valuation ϑ_1^i is obtained from ϑ^i by projection onto the set $(V_1 \cup C_1)$.

Q.E.D.

The inclusion of the set of sequences of $\mathcal{M}_1 \times \mathcal{M}_2$ into the set of sequences of $\mathcal{M}_1$ follows from Theorem 5.3 and Definition 4.12 (*see* page 4.12). This statement is formulated as the corollary above.

Corollary 5.2 (Sequences of the Product) For two IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ such that:

(1) $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for the product operation (*see* Definition 5.3, page 113), and

(2) $\mathcal{M}_2$ is complete with respect to $\mathcal{M}_1$ (*see* Definition 4.22, page 96),

the following inclusion holds:

$$\mathit{Sequences}(\mathcal{M}_1 \times \mathcal{M}_2) \subseteq \mathit{Sequences}(\mathcal{M}_1) \tag{5.22}$$

□

Finally, we formally state and prove the theorem about the equality between the set of traces of $\mathcal{M}_1 \times \mathcal{M}_2$ and the set of traces of $\mathcal{M}_1$ under some additional conditions made on $\mathcal{M}_1$ and $\mathcal{M}_2$.

Theorem 5.5 (Traces of the Product) For two IOSTS $\mathcal{M}_1$ and $\mathcal{M}_2$ such that:

(1) $\mathcal{M}_1$ and $\mathcal{M}_2$ are compatible for the product operation (*see* Definition 5.3, page 113), and

(2) $\mathcal{M}_2$ is complete with respect to $\mathcal{M}_1$ (*see* Definition 4.22, page 96),

the following equality:

$$\mathit{Traces}(\mathcal{M}_1 \times \mathcal{M}_2) = \mathit{Traces}(\mathcal{M}_1)$$

holds. □

Proof

($\subseteq$) The proof of the first inclusion which states that any trace of the IOSTS $\mathcal{M}_1 \times \mathcal{M}_2$ is also a trace of $\mathcal{M}_1$, follows from Definitions 4.12, 4.14 (*see* pages 90 and 90) and Corollary 5.1 (*see* page 130).

($\supseteq$) The second inclusion, *i.e.* each trace of $\mathcal{M}_1$ is also a trace of $\mathcal{M}_1 \times \mathcal{M}_2$, trivially follows from Definitions 4.12, 4.14 (*see* pages 90 and 90) and Corollary 5.2 (*see* page 134).

Q.E.D.

Chapter 6

Conformance Testing with IOSTS

In this chapter we describe the theory of conformance testing which serves as a basis for the symbolic test generation method described in the next chapter. The work presented in this chapter is mainly inspired from the theory of conformance testing developed by J. Tretmans (see [Tretmans, 1994], [Tretmans, 1996b]*, etc.) and the research done in the VerTeCs team at IRISA (see* [Rusu et al., 2000], [Morel, 2000], [Jard and Jéron, 2002]*). In our theory of conformance testing, behaviors of specifications and implementations under test are modeled with Input-Output Symbolic Transition Systems, and the conformance relation is defined as a partial inclusion of their traces.*

The Plan of the Chapter. At the beginning of the chapter, we give the formal definitions of a specification and an implementation under test. Then, we introduce the notion of conformance relation between them, and define test cases as well as their execution on implementations. Next, we link the notion of test case with the notion of conformance relation, and formulate two first properties (soundness and exhaustiveness) which have to be satisfied by test cases. Then, we give the formal definition of a test purpose which is used as a mechanism for test case selection. Finally, we redefine the conformance relation so that the test purposes can be taken into account, and formulate a few other properties which must be satisfied by test cases.

6.1 Specification

A *specification* ($Spec \in \mathsf{SPECS}$, where the universe SPECS of the specifications was introduced in Section 2.1.1, page 14 of Chapter 2) of a reactive system is a formal description of the system behaviors, which, in general, are expressed

using specialized description languages, for instance, SDL [ITU-T, 1994], LOTOS [ISO/IEC, 1988]. The operational semantics of these languages describes all possible behaviors of a system.

Formally, a *specification* is an initialized (*see* Definition 4.19, page 92) IOSTS $Spec = \langle D_{Spec}, \Theta_{Spec}, L_{Spec}, l^0_{Spec}, \Sigma_{Spec}, T_{Spec} \rangle$. In this work we consider (for the test generation process) specifications without cycles of internal actions during which the system performs its internal computations and does not communicate with its environment. These cycles are called *syntactic livelocks* and formally defined in Chapter 7 (*see* Definition A.4, page 257).

Example 6.1 (Specification) The IOSTS $\mathcal{S}$ depicted in Figure 6.1 is an example of specification for a coffee machine. The syntax and semantics of this

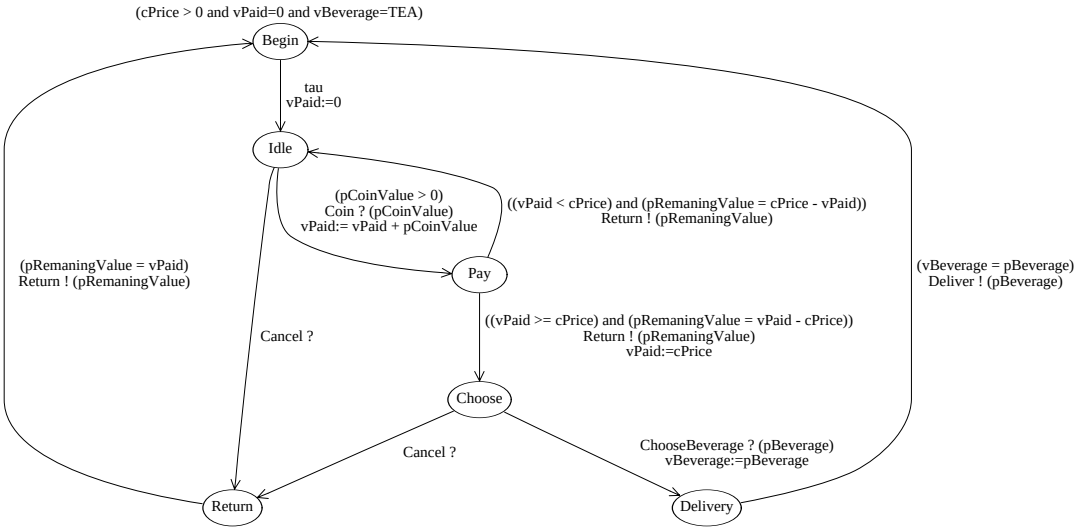


Figure 6.1: Specification.

system were explained in Section 4.1 of Chapter 4 (*see* page 77). □

6.2 Implementation

An *implementation under test* (*iut* $\in$ IMPS, where the universe IMPS of the implementations was introduced in Section 2.1.2, page 14 of Chapter 2) is a physical system, *e.g.* hardware or software components, which is represented in testing as a black box interacting with a tester (*see* Figure 2.2, page 17). It is important to note that *iut* is not a formal object, but due to the *"testing*

hypothesis" (*see* Chapter 2, Formula (2.1), page 15) it can be modeled by the formal object $\mathcal{I}_{iut} \in \mathsf{MODS}$ (MODS is the universe of implementation models introduced in Section 2.1.2, page 14 of Chapter 2).

Formally, let $Spec = \langle D_{Spec}, \Theta_{Spec}, L_{Spec}, l^0_{Spec}, \Sigma_{Spec}, T_{Spec} \rangle$ be a specification. Then, an *implementation under test* (*iut*) is modeled by an IOSTS $\mathcal{I}_{iut} = \langle D_{\mathcal{I}_{iut}}, \Theta_{\mathcal{I}_{iut}}, L_{\mathcal{I}_{iut}}, l^0_{\mathcal{I}_{iut}}, \Sigma_{\mathcal{I}_{iut}}, T_{\mathcal{I}_{iut}} \rangle$ such that:

(1) $\mathcal{I}_{iut}$ does not have any common data with the specification *Spec*, *i.e.* $D_{\mathcal{I}_{iut}} \cap D_{Spec} = \emptyset$.

(2) the set of symbolic constants $C_{\mathcal{I}_{iut}}$ is in bijection with the set of symbolic constants C_{Spec} of *Spec*, *i.e.* there exists a bijective function $f : C_{\mathcal{I}_{iut}} \rightarrowtail C_{Spec}$ such that $\forall\ c \in C_{\mathcal{I}_{iut}}\ .\ [\mathsf{type}(c) = \mathsf{type}(f(c))]$.

Indeed, according to the testing norms published in [ISO, 1991], it does not make any sense to check correctness of an implementation by generating test cases from a specification that does not agree on the values of the constants with the given implementation.

(3) the alphabets of input ($\Sigma^?_{\mathcal{I}_{iut}}$), output ($\Sigma^!_{\mathcal{I}_{iut}}$) actions are in the following relations with the corresponding alphabets of *Spec*: $\Sigma^?_{Spec} = \Sigma^?_{\mathcal{I}_{iut}}$, $\Sigma^!_{Spec} = \Sigma^!_{\mathcal{I}_{iut}}$. The signatures of their common actions must be the same.

(4) $\mathcal{I}_{iut}$ is input-complete (*see* Definition 4.21, page 95).

This assumption is needed as implementations under test must never refuse any input from the tester.

6.3 Conformance

In the first two sections of this chapter we have assumed that specifications for reactive systems and implementations under test can be modeled using the IOSTS formalism. This allows us to reason formally about specifications and implementations. Consequently, we can express the conformance of implementations with respect to specifications by a formal conformance relation between their models. A *conformance relation* defines exactly the set of implementations conformant to a given specification.

The conformance relation introduced in this section and used throughout the thesis is a weaker version of the conformance relations ioconf and ioco defined by J. Tretmans and al. for IOLTS (*see* [Tretmans, 1995], [Tretmans, 1996b], [Tretmans, 2002]). It is called ioc. The reasons of defining the ioc conformance relation instead of using one of its stronger predecessors (*i.e.* ioconf and ioco) are that:

(1) ioc does not take into account any quiescence (outputlocks, deadlocks, livelocks defined on page 42) of IOSTS, and

(2) the problem of deciding whether a system represented as IOSTS is quiescent or not is *undecidable in general* (due to existence of livelocks). Moreover, it is quite complex even in the case when the existence of livelocks in IOSTS is prohibited. We discuss this case in the last chapter of this thesis, where we propose an idea of extension of the ioc relation to the ioco relation defined by J. Tretmans.

In order to formally define the ioc conformance relation we first give an intermediate definition and observation. Consider an arbitrary IOSTS $\mathcal{M}$ with set of states S and set of valued actions $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$. Then:

Definition 6.1 ($Out(S')$ and $In(S')$) The sets of valued outputs and valued inputs, which can be generated by an IOSTS $\mathcal{M}$ when it is in some state s' among the set of states $S' \subseteq S$, are defined as follows:

$$Out(S') \triangleq \{\alpha \in \Lambda^! \mid \exists\, s' \in S', s \in S \,.\, [\, s' \xrightarrow{\alpha} s \,]\} \tag{6.1}$$

$$In(S') \triangleq \{\alpha \in \Lambda^? \mid \exists\, s' \in S', s \in S \,.\, [\, s' \xrightarrow{\alpha} s \,]\} \tag{6.2}$$

□

Next, we make an observation about the sets of valued input/output actions of $\mathcal{M}$ obtained after some trace which does not belong to the set of traces of $\mathcal{M}$.

Observation 6.1 For two arbitrary IOSTS $\mathcal{M}$, $\mathcal{M}'$ and a trace σ which is a trace of $\mathcal{M}'$ but not a trace of $\mathcal{M}$, *i.e.* $\sigma \in Traces(\mathcal{M}') \setminus Traces(\mathcal{M})$, $Out(\mathcal{M}\ \mathsf{after}\ \sigma)$ and $In(\mathcal{M}\ \mathsf{after}\ \sigma)$ are the empty sets. □

Indeed, as we know that $\sigma \notin Traces(\mathcal{M})$ then $(\mathcal{M}\ \mathsf{after}\ \sigma) = \emptyset$ (*see* Definition 4.16, page 91). Thus, using the Definition 6.1 we obtain $Out(\emptyset) = \emptyset$ and $In(\emptyset) = \emptyset$.

Finally, we can formally define the conformance relation ioc. Intuitively, an implementation *iut* is conformant to a specification *Spec* if for each trace of *Spec*, *iut* produces only outputs which are allowed by the specification. Formally:

Definition 6.2 (ioc for Instantiated IOSTS) Let $Spec \in \mathsf{SPECS}$ be an instantiated specification, and $iut \in \mathsf{IMPS}$ be an implementation which is modeled by the instantiated IOSTS $\mathcal{I}_{iut} \in \mathsf{MODS}$. Then, the *conformance relation* ioc between $\mathcal{I}_{iut}$ and *Spec*, is defined as follows:

$$(\mathcal{I}_{iut}\ \mathsf{ioc}\ Spec) \triangleq \tag{6.3}$$
$$\forall\, \sigma \in Traces(Spec).\ [\ Out(\mathcal{I}_{iut}\ \mathsf{after}\ \sigma) \subseteq Out(Spec\ \mathsf{after}\ \sigma)\ ]$$

where the relation after is defined on page 91. □

Notice that this definition takes into account only specifications and implementations that do not have any constants (*see* definition of instantiated IOSTS in Section 4.4.1, page 92). However, the conformance relation ioc can be naturally extended to IOSTS that are not instantiated, *see* Definition 6.3. This extension is possible due to the hypothesis (2) about implementations (*see* Section 6.2, page 138), saying that there exists a one-to-one correspondence between the symbolic constants of an implementation and the symbolic constants of a specification.

Definition 6.3 (ioc in General) An implementation $iut \in \mathsf{IMPS}$ is *conformant to* a specification $Spec \in \mathsf{SPECS}$ if for all possible valuations ς of their symbolic constants, the instance $\mathcal{I}_{iut}(\varsigma)$ of the model $\mathcal{I}_{iut}$ of iut is ioc-related to the instance $Spec(\varsigma)$ of the specification $Spec$, *i.e.*

$$(iut \ \mathsf{ioc} \ Spec) \quad \triangleq \quad \forall\, \varsigma \in \underbrace{\mathsf{DOM}(C_{\mathcal{I}_{iut}})}_{=\mathsf{DOM}(C_{Spec})} \,.\, [\ \mathcal{I}_{iut}(\varsigma) \ \mathsf{ioc} \ Spec(\varsigma)\ ]$$

□

The notion of conformance between implementations and specifications, which was introduced in this section, is illustrated with a simple example.

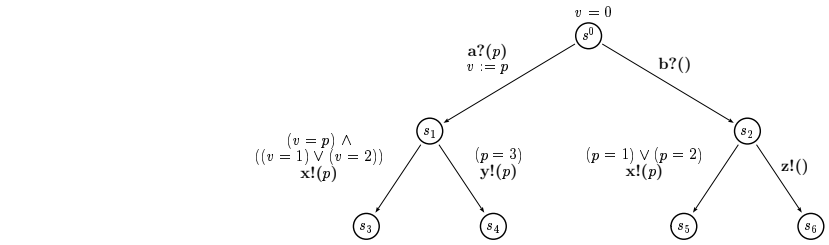
Example 6.2 Let us consider an example depicted in Figure 6.2. This figure consists of three subfigures: Figures 6.2(b) and 6.2(c) shows the models of the implementations iut_1 and iut_2 respectively, and Figure 6.2(a) represents the formal specification of these implementations.

As the specification *Spec* and the implementations iut_1 and iut_2 do not have any parameters, then they are already instantiated (*see* Definition 4.18, page 92). Thus, iut_1 conformant to *Spec* (*resp.* iut_2 conformant to *Spec*) if and only if $\mathcal{I}_{iut_1}$ ioc *Spec* (*resp.* $\mathcal{I}_{iut_2}$ ioc *Spec*).

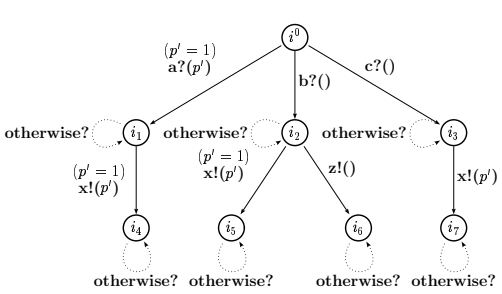
(1) ($\mathcal{I}_{iut_1}$ ioc *Spec*) as for each trace σ of *Spec*, the set of valued output actions of $\mathcal{I}_{iut_1}$ is included into the set of valued output actions obtained after σ by *Spec*. For example, after the trace $\langle \mathbf{b}, () \rangle \in Traces(Spec)$ we obtain that:

$$\underbrace{\{\langle \mathbf{x}, (1) \rangle, \langle \mathbf{z}, () \rangle\}}_{(\mathcal{I}_{iut_1} \ \mathsf{after} \ \langle \mathbf{b}, () \rangle)} \subseteq \underbrace{\{\langle \mathbf{x}, (1) \rangle, \langle \mathbf{x}, (2) \rangle, \langle \mathbf{z}, () \rangle\}}_{(Spec \ \mathsf{after} \ \langle \mathbf{b}, () \rangle)}$$

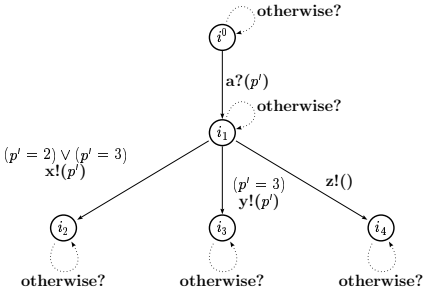
Notice also that $\mathcal{I}_{iut_1}$ can execute the input action $\mathbf{c}$ in its initial state, which is not specified by *Spec*. This additional input does not violate conformance



(a) *Spec*



(b) $\mathcal{I}_{iut_1}$



(c) $\mathcal{I}_{iut_2}$

Figure 6.2: An example illustrating the conformance relation ioc. Here, $(\mathcal{I}_{iut_1}$ ioc *Spec*) and $\neg(\mathcal{I}_{iut_2}$ ioc *Spec*).

between $\mathcal{I}_{iut_1}$ and *Spec*, as the conformance relation verifies the inclusion of valued outputs after the traces of *Spec*. Thus, ioc allows to model partial specifications.

(2) $\neg(\mathcal{I}_{iut_2}$ ioc *Spec*) as after the trace $\langle \mathbf{a}, (2) \rangle$ of *Spec* we obtain:

$$\underbrace{\{\langle \mathbf{x}, (2) \rangle, \langle \mathbf{x}, (3) \rangle, \langle \mathbf{y}, (3) \rangle, \langle \mathbf{z}, () \rangle\}}_{(\mathcal{I}_{iut_2} \text{ after } \langle \mathbf{a}, (2) \rangle)} \not\subseteq \underbrace{\{\langle \mathbf{x}, (2) \rangle, \langle \mathbf{y}, (3) \rangle\}}_{(\mathit{Spec} \text{ after } \langle \mathbf{a}, (2) \rangle)}$$

□

6.4 Test Case

This section introduces the notion of *test case* ($TC \in \mathsf{TESTS}$, where the universe TESTS of test cases was introduced in Section 2.2, page 16 of Chapter 2), which plays a central role in testing. During execution, the test cases interact with implementations under test, observe their outputs, and, based on these observations, they generate test verdicts (*see* Figure 2.2, page 17). Formally, a test case is defined as follows:

Definition 6.4 (Test Case) Let $\mathcal{I}_{iut} = \langle D_{\mathcal{I}_{iut}}, \Theta_{\mathcal{I}_{iut}}, L_{\mathcal{I}_{iut}}, l^0_{\mathcal{I}_{iut}}, \Sigma_{\mathcal{I}_{iut}}, T_{\mathcal{I}_{iut}} \rangle$ be the model of an implementation under test *iut*. Then, a *test case* is an IOSTS $TC = \langle D_{TC}, \Theta_{TC}, L_{TC}, l^0_{TC}, \Sigma_{TC}, T_{TC} \rangle$ such that:

(1) there exists an injective function $f : C_{\mathcal{I}_{iut}} \mapsto C_{TC}$ such that $\forall\, c \in C_{\mathcal{I}_{iut}} \,.\, [\mathsf{type}(c) = \mathsf{type}(f(c))]$.

This requirement is needed due to the fact that at the execution time the valuation of common symbolic constants of a test case and an implementation under test must be the same (*see* the testing standard published in [ISO, 1991]).

(2) the alphabet of actions Σ_{TC} consists of *only* input and output actions, *i.e.* $\Sigma_{TC} = \Sigma^?_{TC} \cup \Sigma^!_{TC}$ and $\Sigma^\tau_{TC} = \emptyset$.

(3) TC is compatible for the parallel composition with $\mathcal{I}_{iut}$ (*see* Definition 5.1, page 100).

(4) the set of locations L_{TC} is equipped with three mutually disjoint sets of locations $\mathsf{Pass} \subseteq L_{TC}$, $\mathsf{Fail} \subseteq L_{TC}$ and $\mathsf{Inconclusive} \subseteq L_{TC}$ labeled with verdicts *Pass*, *Fail* and *Inconclusive* respectively.

(5) TC is initialized (*see* Definition 4.19, page 92).

(6) TC is deterministic (*see* Definition 4.20, page 93).

(7) TC is input-complete (*see* Definition 4.21, page 95) except for locations belonging to the set $\mathsf{Pass} \cup \mathsf{Fail} \cup \mathsf{Inconclusive}$.

□

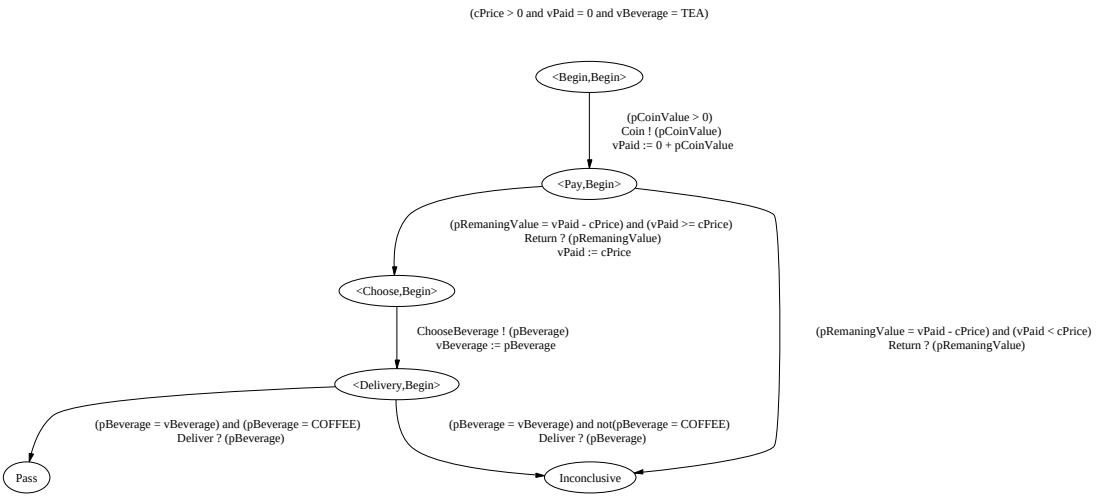


Figure 6.3: Test case.

Example 6.3 (Test Case) Figure 6.3 presents an example of test case for a coffee machine, which is obtained from the specification shown on Figure 6.1 (*see* page 138) and the test purpose depicted in Figure 6.5 (*see* page 149). It covers all the behaviors of the specification targeted by the test purpose, namely, it accepts only one payment and does not exercise pushing the cancel button. Note that the test case shown on Figure 6.3 in not input-complete, as, due to the limited space of Figure 6.3, we did not show locations labeled with the *Fail* verdict, in which the test case arrives if an implementation under test produces an output that is not allowed by the specification. □

6.5 Test Execution

Test execution, which was introduced in Section 2.2.2 of Chapter 2 (*see* page 17), is the process of running a test case ($TC \in \mathsf{TESTS}$) on a concrete black-box implementation under test ($iut \in \mathsf{IMPS}$), observing responses from iut, and based

on these responses, generating a *test verdict* (*see* Figure 2.2, page 17). In the thesis we model the test execution by the *parallel composition* (*see* Definition 5.2, page 100) between TC and the model $\mathcal{I}_{iut}$ of *iut*, which exists due to the *"testing hypothesis"* (*see* Formula 2.1 in Chapter 2, page 15).

Test Verdict. The this paragraph we formalize the notion of test verdict corresponding to each execution of a test case $TC = \langle D_{TC}, \Theta_{TC}, L_{TC}, l^0_{TC}, \Sigma_{TC}, T_{TC} \rangle$ on an implementation under test *iut*. Before giving the formal definition of the test verdict, we denote by:

- $\mathsf{PASS} = \{s = \langle l, \vartheta \rangle \in S_{TC} \mid l \in \mathsf{Pass}\}$,
- $\mathsf{FAIL} = \{s = \langle l, \vartheta \rangle \in S_{TC} \mid l \in \mathsf{Fail}\}$, and
- $\mathsf{INCONCLUSIVE} = \{s = \langle l, \vartheta \rangle \in S_{TC} \mid l \in \mathsf{Inconclusive}\}$

the sets of states of TC corresponding to the locations labeled with *Pass*, *Fail* and *Inconclusive*.

Intuitively, a *test verdict* is defined as follows: consider an observable trace σ of the parallel composition $(\mathcal{I}_{iut} \parallel TC)$, where $\mathcal{I}_{iut}$ is the model of an implementation under test *iut* and TC is a test case; then, TC produces the *Pass* (*resp. Fail*, *Inconclusive*) *verdict* if the state in which the test case TC arrives after the trace σ belongs to PASS (*resp.* FAIL, INCONCLUSIVE). Formally:

Definition 6.5 (Verdict) Let σ be a trace of $\mathcal{I}_{iut} \parallel TC$, then:

$$
\begin{array}{lll}
(\mathsf{verdict}(\sigma) = \mathit{Pass}) & \textbf{if} & (TC \ \mathsf{after}\ \sigma \subseteq \mathsf{PASS}) \\
(\mathsf{verdict}(\sigma) = \mathit{Fail}) & \textbf{if} & (TC \ \mathsf{after}\ \sigma \subseteq \mathsf{FAIL}) \\
(\mathsf{verdict}(\sigma) = \mathit{Inconclusive}) & \textbf{if} & (TC \ \mathsf{after}\ \sigma \subseteq \mathsf{INCONCLUSIVE})
\end{array}
$$

□

We remark that for a given implementation, a test case always produces a same test verdict after several executions of a *same* trace on this implementation. This follows directly from fact that any test case TC is an initialized and deterministic IOSTS (*see* the items (5) and (6) of Definition 6.4, page 143), thus, it is not hard to show that for each trace $\sigma \in \mathit{Traces}(\mathcal{I}_{iut} \parallel TC)$ (which is also a trace of TC due to Theorem 5.2, page 112), all states of TC belonging to $(TC \ \mathsf{after}\ \sigma)$ correspond to *exactly one location* of the test case TC.

Nevertheless, due to the fact that an implementation under test can have several reactions on an input from the tester, a test case can produce several possible executions giving possibly different test verdicts for the same implementation. Thus, an implementation under test may be rejected, accepted, or may produce

the *Inconclusive* verdict for the same test case while following different possible traces of the parallel composition between the test case and the implementation. Formally:

Definition 6.6 (may_fail) Let $TC \in \mathsf{TESTS}$ be a test case and $\mathcal{I}_{iut} \in \mathsf{MODS}$ be a model of an implementation $iut \in \mathsf{IMPS}$. Let also TC and $\mathcal{I}_{iut}$ be compatible for the parallel composition (*see* Definition 5.1, page 100).

Then, TC *may fail* $\mathcal{I}_{iut}$ if there exists a trace σ in the parallel composition between $\mathcal{I}_{iut}$ and TC after which TC produces the *Fail* verdict. Formally:

$$(TC \ \mathsf{may\_fail} \ \mathcal{I}_{iut}) \quad \triangleq \quad \exists\, \sigma \in \mathit{Traces}(\mathcal{I}_{iut} \ ||\ TC) \ .\ [\ \mathsf{verdict}(\sigma) = \mathit{Fail}\ ] \tag{6.4}$$

□

The relations may_pass and may_inconc can be defined in the same way as the relation of possible rejection may_fail.

6.6 Property of a Test Case: Soundness

In the previous sections of this chapter we have introduced two significant notions used in conformance testing, namely, test case (*see* Definition 6.4, page 143) and conformance relation (*see* Definition 6.3, page 141). The aim of this section is to establish the connection between these two independently defined notions. We know that this connection exists. Indeed, as the purpose of test cases is to inform us about conformance of an implementation with respect to its specification, then the test cases must hold some properties relative to a conformance relation. For instance, producing the *Fail* verdict must imply detection of non-conformance in an implementation under test. This property is called *soundness*. It is formally defined as follows:

Definition 6.7 (Soundness) A test case $TC \in \mathsf{TESTS}$ with a set of symbolic constants C is *sound* for a specification $\mathit{Spec} \in \mathsf{SPECS}$ and set of implementations under test $\mathit{Iuts} \subseteq \mathsf{IMPS}$ if:

$$\forall\, iut \in \mathit{Iuts}, \varsigma \in \mathsf{DOM}(C) \ .\ [\ (\mathcal{I}_{iut}(\varsigma) \ \mathsf{ioc} \ \mathit{Spec}(\varsigma)) \implies \neg(TC(\varsigma) \ \mathsf{may\_fail} \ \mathcal{I}_{iut}(\varsigma))\ ] \tag{6.5}$$

□

Intuitively, a test case is *sound* if it does not reject conformant implementations. This property can be achieved in practice, but for practical testing is not sufficient, as incorrect implementations can also pass the test case. For instance, a test case accepting all possible implementations is sound.

Example 6.4 Consider a test case TC and a specification *Spec* shown on Figures 6.4(a) and 6.4(b) (*see* page 148). Moreover, consider a set of implementations $Iuts = \{iut_1\}$, where the model of iut_1 is depicted in Figures 6.4(c) (*see* page 148). This test case TC is sound for *Spec* and $Iuts = \{iut_1\}$ as:

- the model of iut_1, *i.e.* $\mathcal{I}_{iut_1}$, is conformant to *Spec* (*see* Example 6.2, page 141);
- TC does not reject $\mathcal{I}_{iut_1}$ because for all maximal traces of $TC \,||\, \mathcal{I}_{iut_1}$ (*see* Figure 6.4(d), page 148) which are:

$$\underbrace{\langle \mathbf{a}, (1)\rangle\langle \mathbf{x}, (1)\rangle}_{\sigma_1}; \underbrace{\langle \mathbf{b}, ()\rangle\langle \mathbf{x}, (1)\rangle}_{\sigma_2}; \underbrace{\langle \mathbf{b}, ()\rangle\langle \mathbf{z}, ()\rangle}_{\sigma_3}$$

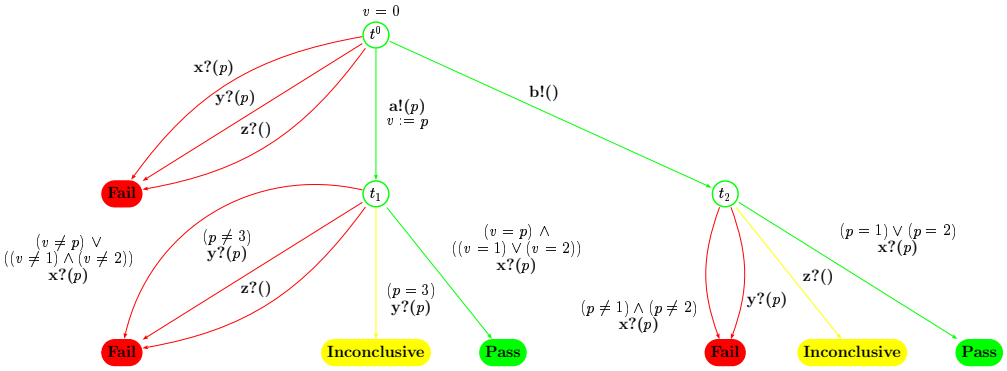
  the test case does not produce the *Fail* verdict, *i.e.* $\mathsf{verdict}(\sigma_1) = \mathsf{verdict}(\sigma_2) = Pass$, $\mathsf{verdict}(\sigma_3) = Inconclusive$.

Thus, TC does not reject implementations belonging to the set *Iuts*, which are conformant to *Spec*. Therefore TC is sound. □

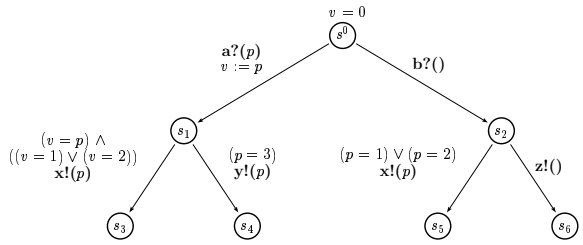
In testing the soundness of test cases is often defined together with *exhaustiveness*. A set of test cases is *exhaustive* if implementations which do not conform to a given specification may be rejected by a test case belonging to this set of test cases. However, in this part of the thesis the notion of exhaustiveness is not used, and therefore is not formally defined. It is replaced by another property called *relative exhaustiveness* which is defined in the sequel. This property permits to relate not only test cases, specifications and implementations, but also *test purposes* introduced in the next section and used as selection mechanisms in out test generation technique.

6.7 Test Purpose

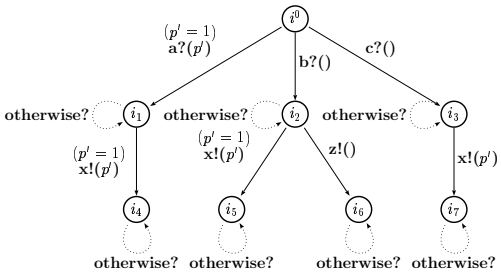
The test case generation technique, which is presented in the thesis, uses the concept of test purpose. A *test purpose* describes behaviors of the system to be tested, and it is used in order to select a part of the system's specification *Spec* (*see* Section 6.1, page 137) for which a test case TC (*see* Section 6.4, page 143) will be generated. Formally:



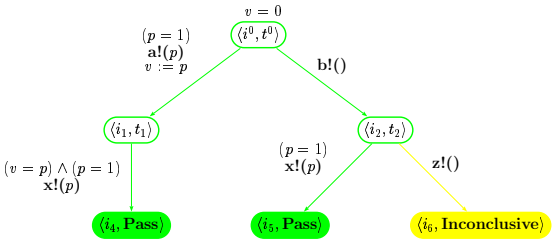
(a) *TC*



(b) *Spec*



(c) $\mathcal{I}_{iut_1}$



(d) $TC \parallel \mathcal{I}_{iut_1}$

Figure 6.4: Sound test case *TC* for *Spec* and $Iuts = \{iut_1\}$.

Definition 6.8 (Test Purpose) Let *Spec* be a specification. Then, a *test purpose* of *Spec* is an IOSTS $TP = \langle D_{TP}, \Theta_{TP}, L_{TP}, l^0_{TP}, \Sigma_{TP}, T_{TP} \rangle$, where $\Sigma = \Sigma^?_{TP} \cup \Sigma^!_{TP} \cup \Sigma^\tau_{TP}$, such that:

(1) TP is equipped with the special location $Accept \in L_{TP}$. This location plays the role of a selection mechanism, *i.e.* it is used to indicate behaviors of the specification *Spec* which have to be tested;

(2) Each symbolic transition $t \in T_{TP}$ of TP, whose target is the location $Accept$ and source is different from the target, is labeled with either input or output action, *i.e.* $\forall \langle l, a, \pi, G, A, l' \rangle \in T_{TP} \,.\, [(l \neq l' \wedge l' = Accept) \implies (a \in \Sigma^?_{TP} \cup \Sigma^!_{TP})]$;

(3) TP is initialized (*see* Definition 4.19, page 92);

(4) TP is complete with respect to *Spec* (*see* Definition 4.22, page 96); and

(5) TP is compatible for the product operation with *Spec* (*see* Definition 5.3, page 113).

□

To illustrate the definition above we consider the following example.

Example 6.5 (Test Purpose) Figure 6.5 (*see* page 149) presents one of the possible test purposes for the coffee machine whose specification is shown on Figure 6.1 (*see* page 138). The dashed edges of the given test purpose are generated automatically (*see* the algorithm described in Section 7.1 of Chapter 7, page 169) in order to obtain a complete test purpose with respect to its specification (*see* Definition 4.22, page 96).

The given test purpose describes behaviors where the machine delivers coffee and the user does not introduce coins more than once and does not cancel (see full edges of the graph shown on Figure 6.5). An accepted behavior is indicated by

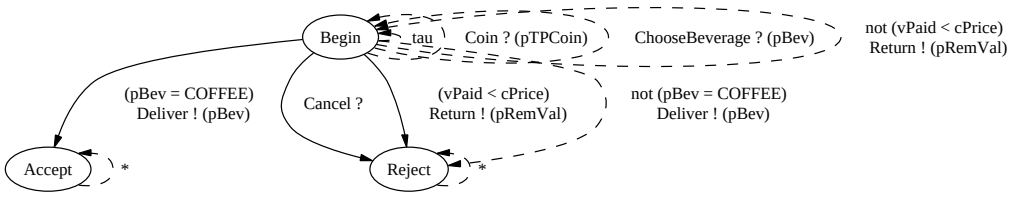


Figure 6.5: Test purpose.

arrival at the location *Accept*. The test purpose rejects behaviors that correspond

to pressing the *Cancel* button, or inserting more than one coin. Rejected behaviors are not necessarily erroneous, they are just behaviors that are not targeted by the test purpose.

The common datum *vPaid* of *TP* (*see* Figure 6.5) and *Spec* (*see* Figure 6.1, page 138) is a variable of *Spec* and a symbolic constant of *TP*. This does not contradict any test purpose hypothesis (*see* Definition 6.8). □

At the end of this section we consider a specification *Spec* and a test purpose *TP* of *Spec* with sets of variables V_{TP}, symbolic constants C_{TP}, states S_{TP} and initial states $S^0_{TP} \subseteq S_{TP}$; and introduce the notion of accepting trace for *TP*. Intuitively, an accepting trace of *TP* is a trace (*see* Definition 4.14, page 90) which leads to some state $s \in S_{TP}$ corresponding to the special location *Accept*. Formally:

Definition 6.9 (Accepting Traces of Test Purpose) The *set of accepting traces* of *TP* is defined as follows:

$$ATraces(TP) \triangleq \{\sigma \in Traces(TP) \mid \exists\, s^0 \in S^0_{TP}, \vartheta \in \mathsf{DOM}(V_{TP} \cup C_{TP}), s = \langle Accept, \vartheta \rangle \in S_{TP} \,.\, [s^0 \stackrel{\sigma}{\Rightarrow} s]\} \quad (6.6)$$

where $Accept \in L_{TP}$ is the special location of *TP* (*see* the hypothesis (1) of Definition 6.8, page 147). □

To illustrate the definition above we consider the example below.

Example 6.6 For instance, let us consider the test purpose *TP* depicted in Figure 6.5 and the following trace of this test purpose:

$$\sigma : \langle Coin, \langle 1 \rangle \rangle \langle Deliver, \langle \text{COFFEE} \rangle \rangle \langle Coin, \langle 3 \rangle \rangle \langle Deliver, \langle \text{TEA} \rangle \rangle$$

where $\langle 1 \rangle, \langle 3 \rangle$ (*resp.* $\langle \text{COFFEE} \rangle, \langle \text{TEA} \rangle$) are the valuations of the parameter *pTPCoin* (*resp. pBev*) carried by the input action *Coin* (*resp.* by the output action *Deliver*). This trace σ is the *accepting* trace of *TP* as it is possible to move from an initial state of *TP*, for example, $\langle Begin, \langle cPrice = 2, vPaid = 0 \rangle \rangle$, to an *accepting* state, for example, $\langle Accept, \langle cPrice = 2, vPaid = 0 \rangle \rangle$, by σ. □

6.8 Conformance Relative to Test Purpose

This section introduces a new conformance relation ioc_{TP}. The relation ioc_{TP} is slightly different from the relation ioc defined in Section 6.3 (*see* page 139). The reason of its introduction is to establish the connection between the three

components: implementation, specification and test purpose of the presented testing theory, where the selection mechanism is based on test purposes.

In order to define the conformance relation ioc_{TP}, we first give some preliminary definitions concerning the set of accepting traces and the sets of their different prefixes.

Intuitively, the set of accepting traces consists of the traces of a specification that are selected by a test purpose through the product operation (*see* Definition 5.4, page 114). Formally:

Definition 6.10 (Set of Accepting Traces of $Spec \times TP$) Let

(1) $Spec \in \mathsf{SPEC}$ be a specification with set of locations L_{Spec},

(2) TP be a test purpose of $Spec$ with set of locations L_{TP}, and

(3) $SP = Spec \times TP$ be an IOSTS obtained from $Spec$ and TP by the product operation (*see* Definition 5.4, page 114).

The IOSTS SP has the set of locations L_{SP}, the set of data $D_{SP} = V_{SP} \cup C_{SP} \cup P_{SP}$, the set of states S_{SP} and the set of initial states $S^0_{SP} \subseteq S_{SP}$.

Then, for the IOSTS $SP = Spec \times TP$ we define the *set of accepting traces* as:

$$
\begin{array}{rl}
ATraces(SP) & \triangleq \\
\{\ \sigma \in Traces(SP) \mid & \exists\ \langle l_{Spec}, Accept\rangle \in L_{SP}, \\
& \exists\ \vartheta \in \mathsf{DOM}(V_{SP} \cup C_{SP}), \\
& \exists\ s^0 \in S^0_{SP}, \\
& \exists\ s = \langle\langle l_{Spec}, Accept\rangle, \vartheta\rangle \in S_{SP}\ .\ [\ s^0 \stackrel{\sigma}{\Rightarrow} s\ ]\ \}
\end{array}
\tag{6.7}
$$

where $l_{Spec} \in L_{Spec}$ is a location of $Spec$ and $Accept \in L_{TP}$ is the special location of TP (*see* the hypothesis (1) of Definition 6.8, page 147).

□

Then, we study the relationship between sets of traces/accepting traces of a specification $Spec$, a test purpose TP of $Spec$, and the synchronous product $Spec \times TP$.

Theorem 6.1 (Set of Accepting Traces of $Spec \times TP$) For a specification $Spec \in \mathsf{SPECS}$ and a test purpose TP of $Spec$, the set of accepting traces of their product $Spec \times TP$ is included into the set of traces of $Spec$, *i.e.*

$$
ATraces(Spec \times TP) \subseteq Traces(Spec) \cap ATraces(TP) \tag{6.8}
$$

□

Proof Consider a trace $\sigma = \alpha_1 \dots \alpha_n$ belonging to $ATraces(Spec \times TP)$, and prove that this trace σ also belongs to $Traces(Spec)$ and $ATraces(TP)$.

(1) To prove the first membership, *i.e.* $\sigma \in Traces(Spec)$, notice that due to Definition 6.10 we have that $ATraces(Spec \times TP)$ is a subset of $Traces(Spec \times TP)$ which is equal to $Traces(Spec)$ (*see* Theorem 5.5, page 134). Therefore, each accepting trace $\sigma \in Spec \times TP$ is a trace of $Spec$, *i.e.* $\sigma \in Traces(Spec)$.

(2) To show the second membership, *i.e.* $\sigma \in ATraces(TP)$, we consider some sequence $\eta = \varepsilon_1 \alpha_1 \dots \varepsilon_n \alpha_n \varepsilon_{n+1}$ of $Spec \times TP$ corresponding to σ, where $\varepsilon_i \in (\Lambda^\tau)^*$ $(i = 1..n+1)$. It is important to notice that as $\sigma \in ATraces(Spec \times TP)$ then the sequence η leads to some accepting state $\langle \langle \tilde{l}^n_{Spec}, Accept \rangle, \tilde{\vartheta}^n \rangle \in (TP \text{ after } \sigma)$ (*see* Definition 4.16, page 91), where $l^n_{Spec} \in L_{Spec}$, $Accept \in L_{TP}$ and $\vartheta^n \in \mathsf{DOM}(V_{Spec \times TP} \cup C_{Spec \times TP})$.

Next, using the definition of a sequence (*see* page 90) we obtain that η corresponds to the following behavior of $Spec \times TP$:

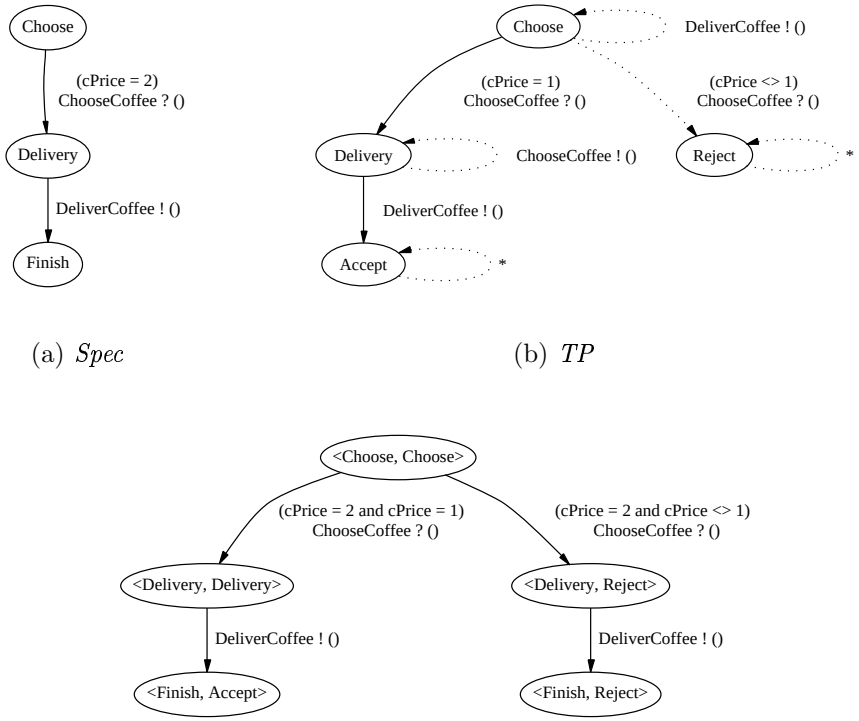
$$\begin{aligned} \beta^{\rightarrow}_{Spec \times TP} : \quad & \langle \langle l^0_{Spec}, l^0_{TP} \rangle, \vartheta^0 \rangle \overset{\varepsilon_1}{\Rightarrow} \langle \langle \tilde{l}^0_{Spec}, \tilde{l}^0_{TP} \rangle, \tilde{\vartheta}^0 \rangle \overset{\alpha_1}{\rightarrow} \langle \langle l^1_{Spec}, l^1_{TP} \rangle, \vartheta^1 \rangle \\ & \dots \\ & \langle \langle l^{n-1}_{Spec}, l^{n-1}_{TP} \rangle, \vartheta^{n-1} \rangle \overset{\varepsilon_n}{\Rightarrow} \langle \langle \tilde{l}^{n-1}_{Spec}, \tilde{l}^{n-1}_{TP} \rangle, \tilde{\vartheta}^{n-1} \rangle \overset{\alpha_n}{\rightarrow} \langle \langle l^n_{Spec}, l^n_{TP} \rangle, \vartheta^n \rangle \\ & \overset{\varepsilon_{n+1}}{\Rightarrow} \langle \langle \tilde{l}^n_{Spec}, Accept \rangle, \tilde{\vartheta}^n \rangle \end{aligned} \tag{6.9}$$

where for all i from 0 to n, there exist locations $l^i_{Spec}, \tilde{l}^i_{Spec} \in L_{Spec}$ of $Spec$; $l^i_{TP}, \tilde{l}^i_{TP} \in L_{TP}$ of TP; and valuations of variables and symbolic constants $\vartheta^i, \tilde{\vartheta}^i \in \mathsf{DOM}(V_{Spec \times TP} \cup C_{Spec \times TP})$ of $Spec \times TP$. Then, by analogy with the proof of Theorem 5.4 (*see* page 132), we can show that: if the synchronous product $Spec \times TP$ has the behavior shown as Formula (6.9), then the test purpose TP, which is complete with respect to $Spec$ (*see* Definition 6.8, page 147), has the following behavior:

$$\begin{aligned} \beta^{\rightarrow}_{TP} : \quad & \langle l^0_{TP}, \vartheta^0_{TP} \rangle \overset{\varepsilon_1}{\Rightarrow} \langle \tilde{l}^0_{TP}, \tilde{\vartheta}^1_{TP} \rangle \overset{\alpha_1}{\rightarrow} \langle l^1_{TP}, \vartheta^1_{TP} \rangle \\ & \dots \\ & \langle l^{n-1}_{TP}, \vartheta^{n-1}_{TP} \rangle \overset{\varepsilon_n}{\Rightarrow} \langle \tilde{l}^{n-1}_{TP}, \tilde{\vartheta}^{n-1}_{TP} \rangle \overset{\alpha_n}{\rightarrow} \langle l^n_{TP}, \vartheta^n_{TP} \rangle \\ & \overset{\varepsilon_{n+1}}{\Rightarrow} \langle Accept, \tilde{\vartheta}^n_{TP} \rangle \end{aligned}$$

corresponding to η, where for all $i = 1..n$, $\vartheta^i{}_{TP}$ and $\tilde{\vartheta}^i{}_{TP}$ belong to $\mathsf{DOM}(V_{TP} \cup C_{TP})$. Notice that this behavior ends in the accepting state $\langle Accept, \tilde{\vartheta}^n_{TP} \rangle$. Therefore, the trace σ corresponding to η belongs to the set of accepting traces of TP, *i.e.* $\sigma \in ATraces(TP)$.

Q.E.D.



(a) *Spec* (b) *TP*

(c) $Spec \times TP$

Figure 6.6: $ATraces(Spec \times TP) \subseteq Traces(Spec) \cap ATraces(TP)$ but $ATraces(Spec \times TP) \not\supseteq Traces(Spec) \cap ATraces(TP)$.

The example below illustrates Theorem 6.1. It is also used as a counterexample for the statement saying that the traces of the specification *Spec* are included into the set of accepting traces of $Spec \times TP$.

Example 6.7 Consider a coffee machine whose specification *Spec* is shown on Figure 6.6(a). This coffee machine delivers a coffee if the price is equal to two units. Consider also a test purpose TP of *Spec*, which is depicted in Figure 6.6(b). It describes behaviors where the coffee machine delivers coffee when the price is equal to one unit (see the full edges of TP). Then, we perform the product operation (*see* Definition 5.4, page 114) between *Spec* and TP, which result is shown in Figure 6.6(c). As our aim is to illustrate Inclusion (6.8) (*see* page 151), we compute the sets of traces of *Spec* and the sets of accepting traces of TP and

of $Spec \times TP$:

$$Traces(Spec) = \{\varepsilon; \langle ChooseCoffee, \langle\rangle\rangle; \langle ChooseCoffee, \langle\rangle\rangle\langle DeliverCoffee, \langle\rangle\rangle\}$$
$$ATraces(TP) = \{\langle ChooseCoffee, \langle\rangle\rangle^{+}\langle DeliverCoffee, \langle\rangle\rangle\}$$
$$ATraces(Spec \times TP) = \emptyset$$

From above it is easy to see that the set of accepting trace of $Spec \times TP$, which is the empty set in this example, is included into the intersection between $Traces(Spec)$ and $ATraces(TP)$, which is $\{\langle ChooseCoffee, \langle\rangle\rangle\langle DeliverCoffee, \langle\rangle\rangle\}$. Thus the inclusion $ATraces(Spec \times TP) \subseteq Traces(Spec) \cap ATraces(TP)$ holds.

However, the opposite inclusion, *i.e.* $ATraces(Spec \times TP) \supseteq Traces(Spec) \cap ATraces(TP)$, does not hold. This is because $Spec$ and TP do not synchronize on the values of common variables and symbolic constants. □

Next, we define two sets containing prefixes of accepting traces of the synchronous product between a specification $Spec$ and a test purpose TP of $Spec$. The first of these sets includes all possible prefixes of accepting traces of $(Spec \times TP)$. However, the second one consists of prefixes (1) which belong to the first set, and (2) which are not accepting traces of $(Spec \times TP)$. Formally:

Definition 6.11 (Set of Prefixes) For an IOSTS $SP = Spec \times TP$ with set of accepting traces $ATraces(SP)$ and sets of valued input $\Lambda^{?}$ and valued output $\Lambda^{!}$ actions, we define the *set of prefixes* of the accepting traces $ATraces(SP)$ as follows:

$$Pref(ATraces(SP)) \triangleq \bigcup_{\sigma \in ATraces(SP)} \{\sigma' \in (\Lambda^{?} \cup \Lambda^{!})^{*} \mid \exists\ \sigma'' \in (\Lambda^{?} \cup \Lambda^{!})^{*}.\ [\sigma = \sigma' \cdot \sigma'']\} \quad (6.10)$$

where $\sigma' \cdot \sigma''$ is the operation of concatenation between two strings σ' and σ''. □

Definition 6.12 (Set of Strict Prefixes) For an IOSTS $SP = Spec \times TP$ with set of accepting traces $ATraces(SP)$ and set of prefixes $Pref(ATraces(SP))$, we define the *set of strict prefixes* of $ATraces(SP)$ which are not accepting traces of SP, as follows:

$$SPref(ATraces(SP)) \triangleq (Pref(ATraces(SP)) \setminus ATraces(SP)) \quad (6.11)$$

□

Finally, we can formally introduce the conformance relation ioc_{TP}. This conformance relation defines the a set of implementations that are correct with respect to a specification *Spec* and relative to a test purpose *TP* of *Spec*. Intuitively, an implementation *iut* is conformant to *Spec* and relative to *TP* if for each strict prefix of an accepting trace of $(Spec \times TP)$ which is not an accepting trace of $(Spec \times TP)$, *iut* produces only outputs which are allowed by the specification. Formally:

Definition 6.13 (ioc_{TP} for Instantiated IOSTS) Let $Spec \in \mathsf{SPECS}$ be an instantiated specification, *TP* be an instantiated test purpose of *Spec*, and $iut \in \mathsf{IMPS}$ be an implementation which is modeled by instantiated IOSTS $\mathcal{I}_{iut} \in \mathsf{MODS}$. Then, the *conformance relation* ioc_{TP} between $\mathcal{I}_{iut}$, *Spec* and *TP* is defined as follows:

$$(\mathcal{I}_{iut}\ \mathsf{ioc}_{TP}\ Spec) \quad \triangleq \tag{6.12}$$
$$\forall\ \sigma \in SPref(ATraces(Spec \times TP))\ .\ [\ Out(\mathcal{I}_{iut}\ \mathsf{after}\ \sigma) \subseteq Out(Spec\ \mathsf{after}\ \sigma)\ ]$$

□

By analogy with the conformance relation ioc defined in Section 6.3 (*see* page 139), the conformance relation ioc_{TP} can be also extended to IOSTS that are not instantiated. Before giving a definition of ioc_{TP} in general we introduce some new notations.

Notations. Consider a set of symbolic constants C, a valuation of these symbolic constants $\varsigma \in \mathsf{DOM}(C)$ and an arbitrary IOSTS $\mathcal{M}$ with set of symbolic constants $C_{\mathcal{M}}$. Consider also that there exists an injective function $f : C \to C_{\mathcal{M}}$ such that $\forall\ c \in C\ .\ [\mathsf{type}(c) = \mathsf{type}(f(c))]$. Then, the projection of the valuation ς onto $C_{\mathcal{M}}$ is denoted as follows:

$$\varsigma \downarrow_{C_{\mathcal{M}}} \quad = \quad \mathsf{projection}_{C_{\mathcal{M}}}(\varsigma)$$

where $\mathsf{projection}$ is defined in Section 5.2.1 of Chapter 5 (*see* Definition 5.6, page 119).

Below we illustrate this new notation with an example.

Example 6.8 Consider:

(1) a set of symbolic constants $C = \{a, b, c\}$, where $\mathsf{DOM}(C_{\mathcal{A}}) = \mathtt{nat} \times \mathtt{bool} \times \mathtt{nat}$, and a valuation of these symbolic constants: $\varsigma = \langle 3, true, 5 \rangle$;

(2) an IOSTS $\mathcal{A}$ with set of symbolic constants $C_{\mathcal{A}} = \{d, e\}$, where $\mathsf{DOM}(C_{\mathcal{A}}) = \mathtt{bool} \times \mathtt{nat}$.

Moreover, the elements of $C_{\mathcal{A}}$ are in f-relation with elements of C, *i.e.* $b = f(d)$ and $a = f(e)$, such that $\mathsf{type}(b) = \mathsf{type}(f(d)) = \mathtt{bool}$ and $\mathsf{type}(a) = \mathsf{type}(f(e)) = \mathtt{nat}$.

Then, $\varsigma \downarrow_{C_{\mathcal{A}}} = \{true, 3\}$. □

Definition 6.14 (ioc_{TP} in General) An implementation $iut \in \mathsf{IMPS}$ is *conformant to* a specification $Spec \in \mathsf{SPECS}$ and relative to a test purpose TP of $Spec$ if for all possible valuations $\varsigma \in \mathsf{DOM}(C_{Spec} \cup C_{TP})$ of their symbolic constants, the instance $\mathcal{I}_{iut}(\varsigma \downarrow_{C_{\mathcal{I}_{iut}}})$ of the model $\mathcal{I}_{iut}$ of iut is $\mathsf{ioc}_{TP(\varsigma \downarrow_{C_{TP}})}$-related to the instance $Spec(\varsigma \downarrow_{C_{Spec}})$ of the specification $Spec$, *i.e.*

$$(iut\ \mathsf{ioc}_{TP}\ Spec) \quad \triangleq \quad \forall\, \varsigma \in \mathsf{DOM}(C_{Spec} \cup C_{TP})\ .\ [\ \mathcal{I}_{iut}(\varsigma \downarrow_{C_{\mathcal{I}_{iut}}})\ \mathsf{ioc}_{TP(\varsigma \downarrow_{C_{TP}})}\ Spec(\varsigma \downarrow_{C_{Spec}})\ ] \tag{6.13}$$

□

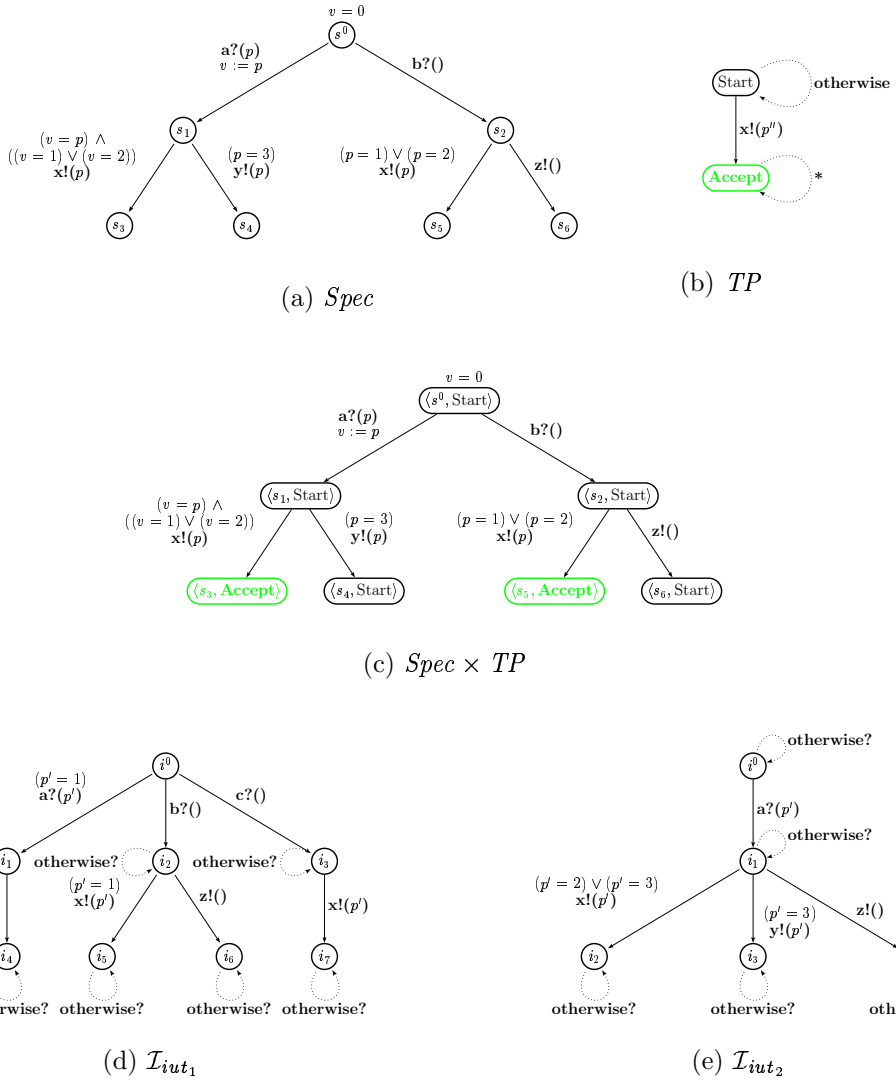
For better understanding of the definitions given above, we illustrate them with an example.

Example 6.9 Consider a specification *Spec* and a simple test purpose *TP* shown in Figures 6.7(a) and 6.7(b) (*see* page 157). We explain how to check if implementations iut_1 and iut_2 whose models are depicted in Figures 6.7(d) and 6.7(e) (*see* page 157) respectively, are conformant to *Spec* and relative to *TP*.

First notice that the specification *Spec*, the test purpose *TP* and the implementations iut_1 and iut_2 do not have any symbolic constants. Thus, they are already instantiated (*see* Definition 4.18, page 92). Therefore, $(iut_1\ \mathsf{ioc}_{TP}\ Spec)$ (*resp.* $(iut_2\ \mathsf{ioc}_{TP}\ Spec)$) if and only if $(\mathcal{I}_{iut_1}\ \mathsf{ioc}_{TP}\ Spec)$ (*resp.* $(\mathcal{I}_{iut_2}\ \mathsf{ioc}_{TP}\ Spec)$).

Then, in order to check whenever the relation ioc_{TP} holds between *Spec* and the models $\mathcal{I}_{iut_1}, \mathcal{I}_{iut_2}$ of the given implementations, we first compute the product between *Spec* and *TP*. The result of this computation is shown in Figure 6.7(c) (*see* page 157), where $ATraces(Spec \times TP) = \{\langle \mathbf{a}, (1)\rangle\langle \mathbf{x}, (1)\rangle; \langle \mathbf{a}, (2)\rangle\langle \mathbf{x}, (2)\rangle; \langle \mathbf{b}, ()\rangle\langle \mathbf{x}, (1)\rangle; \langle \mathbf{b}, ()\rangle\langle \mathbf{x}, (2)\rangle\}$. Finally:

(1) $(\mathcal{I}_{iut_1}\ \mathsf{ioc}_{TP}\ Spec)$ as for each strict prefix (*see* Definition 6.12, page 154) of each accepting trace $\sigma \in ATraces(Spec \times TP)$, the valued outputs of $\mathcal{I}_{iut_1}$ are included into the set of outputs obtained after this trace by *Spec*. For



(a) *Spec* (b) *TP*

(c) *Spec* × *TP*

(d) $\mathcal{I}_{iut_1}$ (e) $\mathcal{I}_{iut_2}$

Figure 6.7: An example illustrating the conformance relation ioc_{TP}. Here, $(\mathcal{I}_{iut_1}\ \mathsf{ioc}_{TP}\ Spec)$ and $\neg(\mathcal{I}_{iut_2}\ \mathsf{ioc}_{TP}\ Spec)$.

example, after the trace $\langle \mathbf{b}, () \rangle \in SPref(ATraces(Spec \times TP))$ we obtain that:

$$\underbrace{\{\langle \mathbf{x}, (1) \rangle, \langle \mathbf{z}, () \rangle\}}_{(\mathcal{I}_{iut_1} \text{ after } \langle \mathbf{b}, () \rangle)} \subseteq \underbrace{\{\langle \mathbf{x}, (1) \rangle, \langle \mathbf{x}, (2) \rangle, \langle \mathbf{z}, () \rangle\}}_{(Spec \text{ after } \langle \mathbf{b}, () \rangle)}$$

(2) $\neg(\mathcal{I}_{iut_2}\ \mathsf{ioc}_{TP}\ Spec)$ as after the trace $\langle \mathbf{a}, (1) \rangle$ belonging to $SPref(ATraces(Spec \times TP))$ we obtain:

$$\underbrace{\{\langle \mathbf{x}, (2) \rangle, \langle \mathbf{x}, (3) \rangle, \langle \mathbf{y}, (3) \rangle, \langle \mathbf{z}, () \rangle\}}_{(\mathcal{I}_{iut_2} \text{ after } \langle \mathbf{a}, (1) \rangle)} \not\subseteq \underbrace{\{\langle \mathbf{x}, (1) \rangle, \langle \mathbf{y}, (3) \rangle\}}_{(Spec \text{ after } \langle \mathbf{a}, (1) \rangle)}$$

□

6.9 Relationships Between ioc and ioc_{TP}

In this section we study relationships between the conformance defined in Section 6.3 (*see* page 139) and the conformance relative to a test purpose presented in Section 6.8 (*see* page 150).

6.9.1 Conformance Implies Relative Conformance

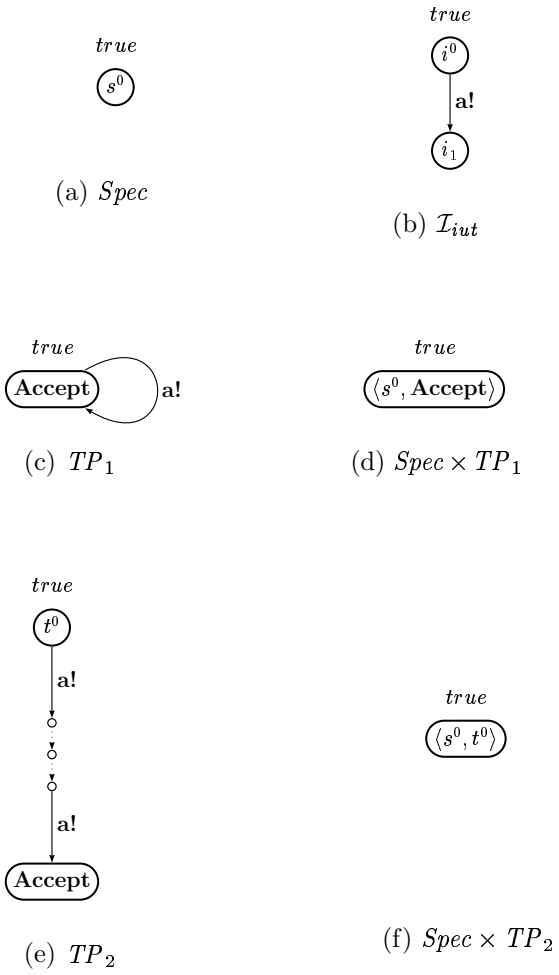
This section proves that if an implementation under test *iut* is conformant to a specification *Spec*, then for all test purposes *TP* of *Spec*, *iut* is conformant to *Spec* and relative to *TP*. Formally:

Theorem 6.2 ($\mathsf{ioc} \implies \mathsf{ioc}_{TP}$)

$$iut\ \mathsf{ioc}\ Spec \implies \forall\ TP\ .\ [\ iut\ \mathsf{ioc}_{TP}\ Spec\ ] \tag{6.14}$$

□

Proof The definition of conformance (*see* Definition 6.3, page 141) says that *iut* is conformant to *Spec* if for each trace σ of *Spec*, outputs of *iut* after σ are included into the outputs of *Spec* after σ. The definition of conformance relative to *TP* (*see* Definition 6.14, page 156) requires the same inclusion of outputs, but for each strict prefix of each accepting trace of $Spec \times TP$. As we know that the set of accepting traces of $Spec \times TP$ is a subset of traces of *Spec* (*see* Theorem 6.1, page 151), then by prefix closure each prefix σ' of each accepting trace of $(Spec \times TP)$ is a trace of *Spec*, and therefore Implication (6.14) holds.
Q.E.D.



(a) *Spec* (b) $\mathcal{I}_{iut}$ (c) TP_1 (d) $Spec \times TP_1$ (e) TP_2 (f) $Spec \times TP_2$

Figure 6.8: A counterexample showing that $\neg(\mathsf{ioc}_{TP} \implies \mathsf{ioc})$.

6.9.2 Relative Conformance Does Not Imply Conformance

In the previous section we have proved that conformance implies relative conformance. More precisely, if an implementation under test *iut* is conformant to a specification *Spec*, then for all test purposes *TP* of *Spec*, *iut* is not conformant with respect to *Spec* and *TP*. However, the backward implication does not hold. This fact is shown by the following counterexample.

Counterexample 6.1 ($\neg(\mathsf{ioc}_{TP} \implies \mathsf{ioc})$) Consider a specification *Spec* (*see* Figure 6.8(a)) such that its sets of data and symbolic transitions are empty, its initial condition is equal to true, its alphabet contains one output action a and its

set of locations contains only one location s^0 that is the initial location of *Spec*.

Consider also an implementation under test *iut* without data. Assume that *iut* is modeled by an IOSTS $\mathcal{I}_{iut}$ (*see* Figure 6.8(b)) with alphabet $\{a\}$, which consists of two locations i^0, i_1, where i^0 is the initial location of $\mathcal{I}_{iut}$, and one symbolic transition from i^0 to i_1 which is labeled by the output action a.

Remark that this implementation is not conformant to the given specification, *i.e.* $\neg(\mathcal{I}_{iut}\ \mathsf{ioc}\ Spec)$. Indeed, after the empty trace of *Spec* (which is the sole trace of the specification), *Spec* does not produce any output action while $\mathcal{I}_{iut}$ produces the output action a. *I.e.* we obtain that:

$$\forall\ \sigma \in \underbrace{Traces(Spec)}_{=\{\varepsilon\}}\ .\ [\ \underbrace{Out(\mathcal{I}_{iut}\ \mathsf{after}\ \sigma)}_{=\{a\}} \not\subseteq \underbrace{Out(Spec\ \mathsf{after}\ \sigma)}_{=\emptyset}\ ]$$

Next, if we show that for all test purposes TP of *Spec*, *iut* is conformant with respect to *Spec* and TP, then we have found an example showing that $\neg(\mathsf{ioc}_{TP} \implies \mathsf{ioc})$.

Consider the test purpose TP_1 shown on Figure 6.8(c). If we compute the product between *Spec* and TP_1, we obtain the IOSTS $(Spec \times TP_1)$ shown on Figure 6.8(d). This IOSTS has a sole accepting trace which is the empty trace, *i.e.* $ATraces(Spec \times TP_1) = \{\varepsilon\}$. Hence, the set of strict prefixes of $ATraces(Spec \times TP_1)$ is empty. Therefore, due to the Definition 6.13 (*see* page 155), we have: $(\mathcal{I}_{iut}\ \mathsf{ioc}_{TP_1}\ Spec)$.

Next, consider a test purpose TP_2 of the form of Figure 6.8(e). By using the product operation we obtain the IOSTS $(Spec \times TP_2)$ depicted in Figure 6.8(f). This IOSTS does not have any accepting trace, and therefore, it does not have any strict prefix of accepting traces. Thus, $(\mathcal{I}_{iut}\ \mathsf{ioc}_{TP_2}\ Spec)$ (*see* Definition 6.13, page 155).

For any other test purpose (with *true* as its initial condition) of *Spec*, the product operation gives either the IOSTS shown on Figure 6.8(d) or the IOSTS show on Figure 6.8(f). If we compute the product between *Spec* and any test purpose with *false* as its initial condition, we obtain an IOSTS that does not have any trace. Thus, $\mathcal{I}_{iut}$ is conformant with respect to *Spec* and such test purposes. From these facts, we can easily deduce that for each test purpose TP, $(\mathcal{I}_{iut}\ \mathsf{ioc}_{TP}\ Spec)$. Therefore, we have shown that $\neg(\mathsf{ioc}_{TP} \implies \mathsf{ioc})$. □

6.10 Properties of a Test Case: Relative Exhaustiveness, Accuracy and Conclusiveness

In addition to the (minimal) test case property called soundness (*see* Definition 6.7, page 146), we define another set of properties that one may expect from a test case, and which are guaranteed by our test generation method described in Chapter 7 (*see* page 167). Before giving the formal definitions of these properties, we consider:

(1) a specification $Spec \in \mathsf{SPECS}$ with set of variables V_{Spec} and set of symbolic constants C_{Spec},

(2) a test purpose TP of $Spec$ with set of variables V_{TP} and set of symbolic constants C_{TP},

(3) a test case $TC \in \mathsf{TESTS}$ derived by the symbolic test generation method described later in this thesis from $Spec$ and TP. This test case TC has the following set of symbolic constants $C_{TC} = (C_{Spec} \cup C_{TP}) \setminus (V_{Spec} \cup V_{TP})$, and

(4) a set of implementation under test $Iuts$ such that each implementation iut belonging to $Iuts$ is modeled by the IOSTS $\mathcal{I}_{iut} \in \mathsf{MODS}$ which:

   (a) is compatible for the parallel composition with TC (*see* Definition 5.1, page 100), and

   (b) has the set of symbolic constants $C_{\mathcal{I}_{iut}}$ such that there exists a bijective function $f : C_{\mathcal{I}_{iut}} \mapsto C_{Spec}$ such that $\forall\, c \in C_{\mathcal{I}_{iut}}\ .\ [\mathsf{type}(c) = \mathsf{type}(f(c))]$. Therefore, $\mathsf{DOM}(C_{\mathcal{I}_{iut}}) = \mathsf{DOM}(C_{Spec})$.

Next, we introduce the test case property which is called relative exhaustiveness. Intuitively, a test case TC is *relatively exhaustive* if implementations nonconformant with the specification $Spec$ and relative to the test purpose TP may be rejected by this test case. Formally:

Definition 6.15 (Relatively Exhaustiveness) A test case TC is *relatively exhaustive* for a specification $Spec$, a test purpose TP of $Spec$, and a set of implementations under test $Iuts$ if:

$$\begin{array}{l} \forall\ iut \in Iuts,\ \varsigma \in \mathsf{DOM}((C_{Spec} \cup C_{TP}) \setminus (V_{Spec} \cup V_{TP}))\ . \\ \quad [\ \neg(\mathcal{I}_{iut}(\varsigma \downarrow_{C_{\mathcal{I}_{iut}}})\ \ \mathsf{ioc}_{TP(\varsigma\downarrow_{C_{TP}})}\ \ Spec(\varsigma \downarrow_{C_{Spec}})) \implies \\ \qquad (TC(\varsigma)\ \ \mathsf{may\_fail}\ \ \mathcal{I}_{iut}(\varsigma \downarrow_{C_{\mathcal{I}_{iut}}}))\ ] \end{array} \tag{6.15}$$

□

Then, we present the test case property called accuracy. Intuitively, a test case TC is *accurate* if, when it produces the $Pass$ verdict, then the observed trace of the implementation under test is a trace of the specification $Spec$ that is selected by the test purpose TP. Formally:

Definition 6.16 (Accuracy) A test case TC is *accurate* for a specification $Spec$, a test purpose TP of $Spec$, and a set of implementations under test $Iuts$ if:

$$\begin{array}{l} \forall\ iut \in Iuts,\ \varsigma \in \mathsf{DOM}((C_{Spec} \cup C_{TP}) \setminus (V_{Spec} \cup V_{TP})), \\ \quad \sigma \in Traces(\mathcal{I}_{iut}(\varsigma \downarrow_{C_{\mathcal{I}_{iut}}}) \mid\mid TC(\varsigma))\ . \\ \quad [\, \mathsf{verdict}(\sigma) = Pass \implies \sigma \in ATraces(Spec \times TP)(\varsigma)\,] \end{array} \tag{6.16}$$

□

Next, we define the conclusiveness of a test case. Intuitively, a test case TC is *conclusive* if it produces the $Inconclusive$ verdict only when the observed trace of the implementation is a trace of the specification $Spec$ ending by an output action, but it cannot be extended into a trace producing the $Pass$ verdict. Formally:

Definition 6.17 (Conclusiveness) A test case TC is *conclusive* for a specification $Spec$, a test purpose TP of $Spec$, and a set of implementations under test $Iuts$ if

$$\begin{array}{l} \forall\ iut \in Iuts,\ \varsigma \in \mathsf{DOM}((C_{Spec} \cup C_{TP}) \setminus (V_{Spec} \cup V_{TP})), \\ \quad \sigma \in Traces(\mathcal{I}_{iut}(\varsigma \downarrow_{C_{\mathcal{I}_{iut}}}) \mid\mid TC(\varsigma))\ . \\ \quad [\, \mathsf{verdict}(\sigma) = Inconclusive \implies \\ \qquad \sigma \in (Traces(Spec(\varsigma \downarrow_{C_{Spec}})) \cdot \Lambda^{!}_{Spec(\varsigma \downarrow_{C_{Spec}})}) \cap Traces(Spec(\varsigma \downarrow_{C_{Spec}})) \ \wedge \\ \qquad \sigma \notin Pref(ATraces((Spec \times TP)(\varsigma)))\,] \end{array} \tag{6.17}$$

where:

- $Pref$ is the set of non-strict prefixes of all accepting traces of the synchronous product $(Spec \times TP)$ (*see* Definition 6.11, page 154), and
- $Traces(Spec(\varsigma \downarrow_{C_{Spec}})) \cdot \Lambda^{!}_{Spec(\varsigma \downarrow_{C_{Spec}})} \triangleq \{\sigma \cdot \alpha \mid \sigma \in Traces(Spec(\varsigma \downarrow_{C_{Spec}})) \wedge \alpha \in \Lambda^{!}_{Spec(\varsigma \downarrow_{C_{Spec}})}\}$

□

Finally, we illustrate all test case properties introduced above with an example.

Example 6.10 Consider a specification *Spec*, a test purpose *TP* of *Spec* and a test case *TC* shown on Figures 6.9(a), 6.9(b) and 6.9(d) (*see* page 164). Also consider a set of implementations $Iuts = \{iut_1\}$, where the model $\mathcal{I}_{iut_1}$ the implementation iut_1 is depicted in Figure 6.9(e) (*see* page 164).

In order to decide whether *TC* is relatively exhaustive, accurate and conclusive for *Spec*, *TP* and *Iuts* we perform the parallel composition between *TC* and $\mathcal{I}_{iut_1}$. The result of this operation is presented as Figure 6.9(f) (*see* page 164). Then,

(1) *TC* is relatively exhaustive for *Spec*, *TP* and *Iuts* because

- *TC* does not reject $\mathcal{I}_{iut_1}$ as it never produces the *Fail* verdict during its execution on the implementation iut_1 (*see* Figure 6.9(f), page 164);
- the model of iut_1, *i.e.* $\mathcal{I}_{iut_1}$, is conformant to *Spec* and relative to *TP* (*see* Example 6.9, page 156) .

(2) *TC* is accurate for *Spec*, *TP* and *Iuts* because the traces $\sigma_1 : \langle \mathbf{a}, (1)\rangle\langle \mathbf{x}, (1)\rangle$ and $\sigma_2 : \langle \mathbf{b}, ()\rangle\langle \mathbf{x}, (1)\rangle$ of $(\mathcal{I}_{iut_1} \parallel TC)$ producing the *Pass* verdict (*see* Figures 6.9(f), page 164) are the accepting traces of $(Spec \times TP)$ (*see* Figure 6.9(c), page 164).

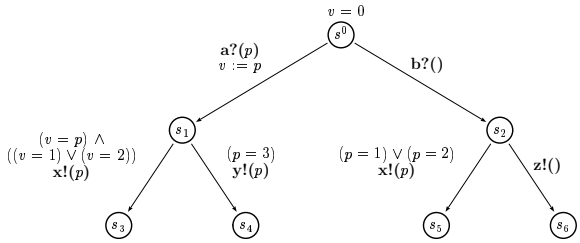
(3) *TC* is conclusive for *Spec*, *TP* and *Iuts* as the trace $\sigma_3 : \langle \mathbf{b}, ()\rangle\langle \mathbf{z}, ()\rangle$ of $(\mathcal{I}_{iut_1} \parallel TC)$ producing the *Inconclusive* verdict (*see* Figures 6.9(f), page 164) is a trace of *Spec* ending with the output action $\mathbf{z}$ of *Spec* (*see* Figure 6.9(a), page 164) and is not a prefix of accepting traces of $(Spec \times TP)$ (*see* Figure 6.9(c), page 164).

□

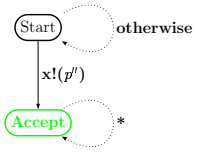
6.11 Correctness of Test Cases

This section introduces the notion of correct test case. The correctness of a test case intuitively means that the test case always gives the right verdict while executing it on a given implementation under test. The formal definition of a correct test case is given below.

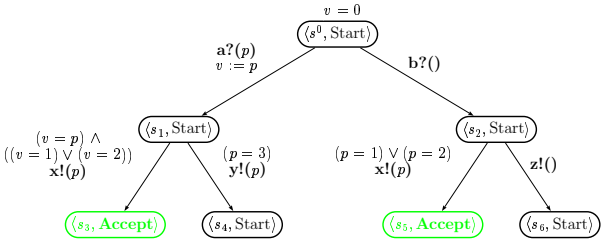
Definition 6.18 (Correctness) A test case $TC \in \mathsf{TESTS}$ which is generated from a specification $Spec \in \mathsf{SPECS}$ and a test purpose *TP* is *correct* for a set of implementations $Iuts \subseteq \mathsf{IMPS}$ if it is sound with respect to *Spec* and *Iuts*; and relatively exhaustive, accurate and conclusive with respect to *Spec*, *Iuts* and *TP* (*see* Definitions 6.7, 6.15, 6.16, 6.17, pages 146, 161, 162, 162). □



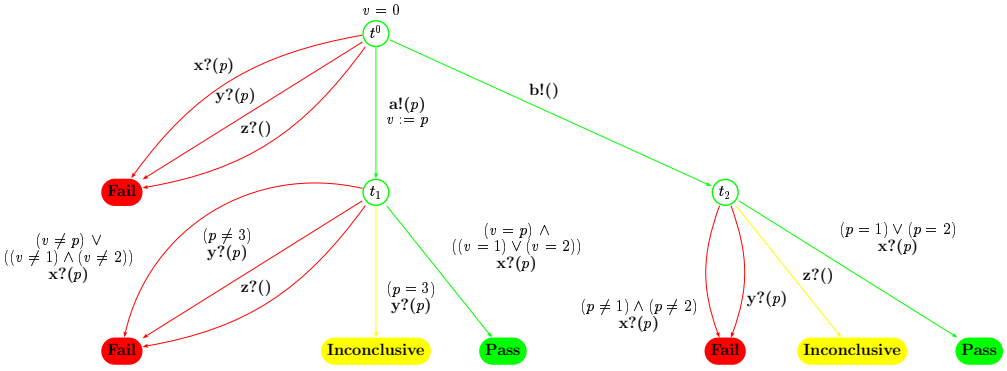
(a) *Spec*



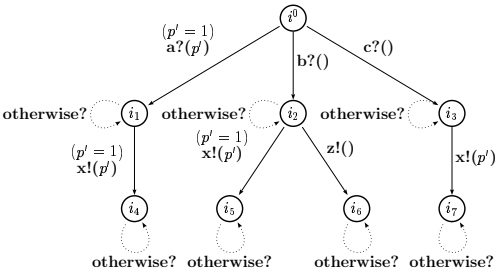
(b) *TP*



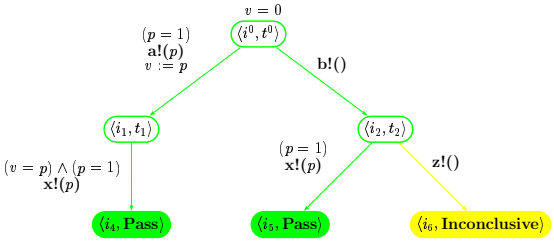
(c) *Spec* × *TP*



(d) *TC*



(e) $\mathcal{I}_{iut_1}$



(f) $TC \mid\mid \mathcal{I}_{iut_1}$

Figure 6.9: An example illustrating that a given test case *TC* is relatively exhaustive, accurate and conclusive for a specification *Spec*, test purpose *TP* of *Spec* and set of implementations $Iuts = \{\mathcal{I}_{...}\}$

Example 6.11 The test case TC shown on Figure 6.9(d) (*see* page 164) which is derived from a specification *Spec* and a test purpose TP of *Spec* (*see* Figures 6.9(a), 6.9(b) on page 164), is correct for a set of implementations $Iuts = \{iut_1\}$ (the model of iut_1 is shown on Figure 6.9(e), page 164) as it is sound (*see* Example 6.4, page 147), relatively exhaustive, accurate and conclusive (*see* Example 6.10, page 163).

□

Chapter 7

Symbolic Test Generation

The chapter describes our method for symbolic test generation implemented in the Symbolic Test Generator (STG) tool. The purpose of this method is to compute a test suite starting from a given specification and test purposes. The generated test cases must be correct in the sense of Definition 6.18 (see page 163), in particular, they are all sound, relative exhaustive, accurate and conclusive. At the beginning of the chapter we present the main steps of the method, then we describe them in more detail. Finally, we prove that the symbolic test generation method produces correct test cases.

Sketch of the Chapter. In this paragraph we briefly describe the main steps of a symbolic test generation method presented in this section. These steps are summarized in Figure 7.1 (*see* page 168) and explained below.

The symbolic test generation method takes as inputs a specification *Spec* and a test purpose *TP* of *Spec*. As incomplete test purposes are allowed (*see* explanations in Section 7.1), the first step of this method is to make them complete with respect to their specifications. The second step of the method consists in computing the product between the given *Spec* and completed *TP* in order to mark behaviors of *Spec* as accepted or rejected by the test purpose, and obtaining the synchronous product *SP* (*see* Figure 7.1, page 168). The third step builds visible deterministic behaviors of *SP*, in other words, it consists in removing internal actions and possible non-deterministic choices from *SP*. The result of this step is denoted by SP_{vis} (*see* Figure 7.1, page 168). The next step of the test generation, namely selection, extracts from SP_{vis} its behaviors allowing to go from the initial location to the *Pass* or *Inconclusive* locations. The result of selection is called a test graph and denoted by *TG*. Moreover, during the selection we invert the alphabets of input and output actions of *TG* making *TG* able to communicate

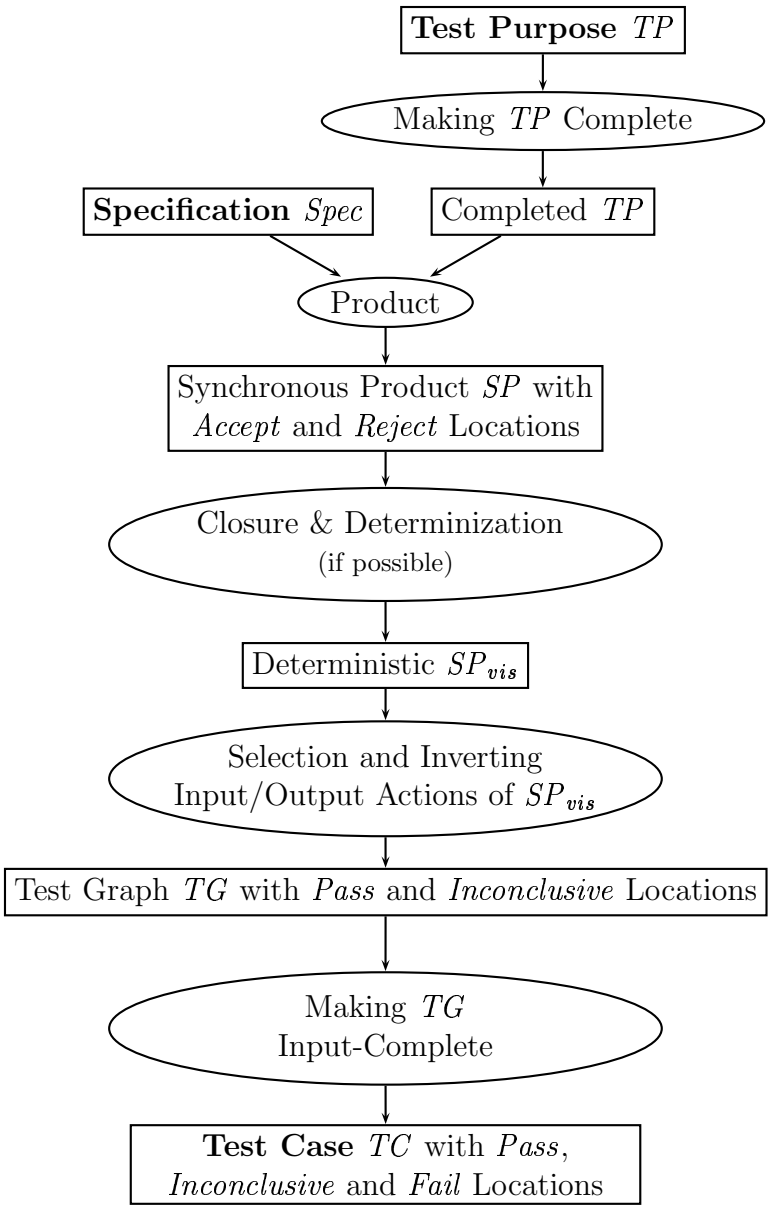


Figure 7.1: Symbolic test generation method.

with implementations under test during the test execution process. Finally, by making TG input-complete we obtain a test case TC.

7.1 Making a Test Purpose Complete

In the testing theory presented in this work, test purposes, playing the role of selection mechanism, must be complete with respect to their specification (*see* Definition 6.8, page 147). In practice this requirement often complicates the process of writing test purposes. Thus, we decided to accept incomplete test purposes from the system developer and complete them automatically at the first stage of the test generation process. This decision allows us to focus on the intended behaviors of the system under test, and significantly simplifies the process of writing the test purposes.

Plan of the Section. The rest of this section is organized as follows: first, we give an intuitive idea for an algorithm whose aim is to complete a given test purpose with respect to its specification. Then, we present the formal algorithm, and prove that this algorithm gives a correct result with respect to Definition 4.22 (*see* page 96). Finally, we illustrate the algorithm with an example.

Intuitive Idea of the Algorithm. Remember that in order to test a given specification, the system developer should provide a test purpose. In this thesis we require test purposes to be represented as IOSTS. This representation allows to (partially) describe behaviors of a given specification that should be tested (sequences of actions leading to the *Accept* location), or should not be tested (sequences of actions leading to the deadlock location called here *Reject*).

As we have remarked above, the developer can provide an incomplete test purpose which later will be completed with respect to its specification by Algorithm 7.1 (*see* page 170). The intuitive idea of this algorithm is that in each location of a given test purpose TP we must be able to execute any valued action of its specification *Spec*. Therefore, each location of TP must be completed with all actions of *Spec* as follows:

(1) Assume that an action a of *Spec* is not specified in a location l of TP. Hence, we suppose that in this location l, the developer does not care about presence or absence of the action a in an implementation under test iut. Thus, in the location l, we add a loop labeled by a.

(2) Assume that an action a of *Spec* is specified in a location l of TP. Moreover, the developer would like to test the presence of this action a in iut under some condition G. Hence, we assume that he/she does not want to test

occurrence of this action under another condition (different from G). In this case we add a new symbolic transition from l to *Reject*, which is labeled with a and guarded with the negation of G.

(3) Assume again that an action a of *Spec* is specified in a location l of *TP*, but the developer does not want to test occurrence of this action under a condition G. However, the developer wishes to test the presence of this action in *iut* under any other condition. Thus, in order to make l complete, we add a loop on a in l which is guarded with the negation of G.

Remark that we have considered all possible situations, where a location l of *TP* must be completed.

The ideas described above are realized by the following algorithm.

Algorithm 7.1 (Making a Test Purpose Complete with Respect to its Specification) Consider a specification *Spec* with alphabet of actions $\Sigma_{Spec} = \Sigma^{?}_{Spec} \cup \Sigma^{!}_{Spec} \cup \Sigma^{\tau}_{Spec}$, and an *incomplete* test purpose of this specification: $TP = \langle D_{TP}, \Theta_{TP}, L_{TP}, l^{0}_{TP}, \Sigma_{TP}, T_{TP} \rangle$, meaning that *TP* satisfies all hypotheses of Definition 6.8 (*see* page 147) except the fourth one.

The algorithm for making *TP* complete with respect to *Spec* consists of two steps described below.

First, it checks if the given *Spec* and *TP* respect the first and third items of Definition 4.22 (*see* page 96). Notice that the algorithm does not check the second item of this definition as *TP* is compatible for the product operation with *Spec* (*see* Definition 5.3, page 113), and therefore the equality of their alphabets of actions is achieved.

Second, (*i.e.* when the items above are successfully checked) the algorithm completes *TP* with respect to *Spec* as follows.

For a location $l \in L_{TP}$ and an action $a \in (\Sigma_{TP} = \Sigma_{Spec})$, we denote by:

$$T_{l,a} = \{t_1 = \langle l, a, \pi, G_1, A_1, l_1 \rangle, \ldots, t_n = \langle l, a, \pi, G_n, A_n, l_n \rangle\} \subseteq T_{TP}$$

the set of all symbolic transitions $t_i \in T_{l,a}$ $(i = 1..n)$ leaving the location l, labeled with the action a and guarded with a Boolean expression G_i that is not syntactically equal to *true*.

Then, for each location $l \in L_{TP}$ and each action $a \in (\Sigma_{TP} = \Sigma_{Spec})$:

(1) if $T_{l,a}$ is not empty, then:

(a) For the set of symbolic transitions $T^R_{l,a} = \{t_i \in T_{l,a} \mid l_i = \mathit{Reject}\}$, a self-loop

$$t' = \langle l, a, \pi, \neg(\bigvee_{t_i \in T^R_a} G_i), A', l\rangle$$

will be generated. Here and in the next items, $A' = \bigcup_{v \in V_{TP}} [v := v]$ is the set of identity assignments.

(b) For the symbolic transitions $t_i \in (T_{l,a} \setminus T^R_{l,a})$, a new transition

$$t' = \langle l, a, \pi, \neg(\bigvee_{t_i \in (T_{l,a} \setminus T^R_{l,a})} G_i), A', \mathit{Reject}\rangle$$

leading to the *Reject* location will be obtained.

(2) if $T_{l,a}$ is empty (*i.e.* there are no symbolic transitions outgoing from l and labeled with a), then a self-loop on the action a will be produced, *i.e.* a transition $t' = \langle l, a, \pi, \mathit{true}, A', l\rangle$ will be added to T_{TP}.

□

Next, we make an observation about the correctness of Algorithm 7.1 (*see* page 170).

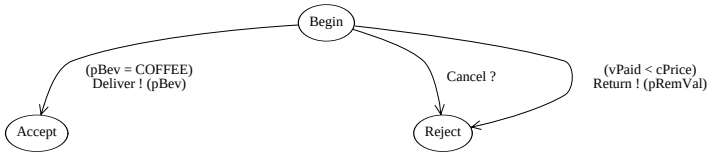
Observation 7.1 Let $\mathit{Spec} \in \mathsf{SPECS}$ be a specification, and TP be a test purpose of *Spec* which satisfies the first three hypotheses of Definition 4.22 (*see* page 96). Then, Algorithm 7.1 presented in this section generates a test purpose TP' that is complete with respect to the specification *Spec*. □

To prove this observation it is enough to show that TP' obtained from TP by Algorithm 7.1 preserves the hypothesis (4) of Definition 4.22 (*see* page 96):

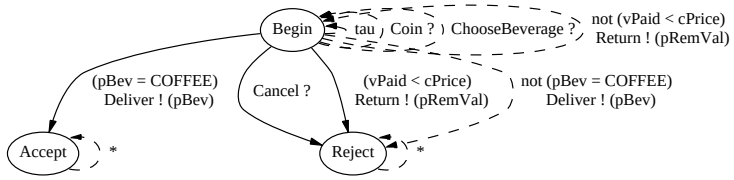
(1) Assume that TP has a symbolic transition from a location l to a location $l' \neq \mathit{Reject}$ that is labeled with an action a. Then, Algorithm 7.1 creates a new symbolic transition leading to the *Reject* location and guarded with the negation of the disjunction of all guards of symbolic transition outgoing from l and labeled with a.

(2) Otherwise, *i.e.* if TP does not have a symbolic transition outgoing from a location l and labeled with an action a, Algorithm 7.1 creates a loop on a in the location l.

Finally, we illustrate the algorithm making a test purpose complete with respect to its specification on an example given below.



(a) *TP*



(b) TP' obtained from TP and $\mathcal{S}$ by Algorithm 7.1

Figure 7.2: Making a given test purpose TP complete with respect to its specification $\mathcal{S}$ shown on Figure 6.1 (*see* page 138).

Example 7.1 Consider the specification of a coffee machine, which is shown on Figure 6.1 (*see* page 138) and the test purpose TP depicted in Figure 7.2(a) below.
Due to the facts that:

(1) TP does not have variables which at the same time are symbolic constants of *Spec*, as $\underbrace{\emptyset}_{V_{TP}} \cap \underbrace{\{cPrice\}}_{C_{Spec}} = \emptyset$;

(2) the initial condition Θ_{TP}, which is equal to *true*, does not contain any constraints on the variables (*i.e.* $vPaid, vBeverage$) and symbolic constants (*i.e.* $cPrice$) of *Spec*; and

(3) the alphabets of TP and *Spec* are equal, *i.e.* $\Sigma_{TP} = \Sigma_{Spec} = \{Coin, Cancel, ChooseBeverage\}^{?} \cup \{Return, Deliver\}^{!} \cup \{tau\}^{\tau}$.

we can perform the procedure explained in the second paragraph of the algorithm. Bellow we explain how the ***Begin*** location of TP (*see* Figure 6.5) must be completed.

– First, as TP has a symbolic transition outgoing from *Begin* and labelled with *Return*, which leads to *Reject* and has the guard $G : vPaid < cPrice$

different from *true*. Then, we add to *Begin* a self-loop labeled with the *Return* action and guarded with the negation of G, *i.e.* $\neg(vPaid < cPrice)$ (*see* Figure 7.2(b)).

– Second, as *TP* has a transition outgoing from *Begin* and labelled with *Deliver* which leads to *Accept* and has the guard $G : pBev = \text{COFFEE}$ that is different from *true*. Then, we add a new transition from *Begin* to *Reject*, which is labeled with the same action *Deliver* and guarded with the negation of G, *i.e.* $\neg(pBev = \text{COFFEE})$ (*see* Figure 7.2(b)).

– Finally, as *TP* does not have any symbolic transitions outgoing from *Begin* which are labeled with *Coin*, *ChooseBeverage*, and *tau*, then we add to *Begin* the self-loops on these actions (*see* Figure 7.2(b)).

The remaining locations of *TP*, namely *Accept* and *Reject* are completed with the self-loops on all actions of *TP* (*see* the symbolic transitions labeled with "*" on Figure 7.2(b)) as there are no symbolic transitions outgoing from these locations (*see* Figure 7.2(a)).

□

7.2 Product

At the next step of our test generation method we compute a synchronous product *SP* between a specification *Spec* and a completed (*see* previous section) test purpose *TP* of *Spec*. The aim of this step is to identify some behaviors of *Spec* as accepted by *TP*. The idea of using the product operation in order to mark the behaviors of *Spec* arises from model-checking [Clarke et al., 1999], and was already used in test generation, see for example, the following papers [Jéron and Morel, 1999], [Jard and Jéron, 2002]. We construct the synchronous product *SP* from the given *Spec* and *TP* using the product operation defined in Section 5.2 (*see* Definition 5.4). It is important to emphasize that as:

(1) *Spec* and *TP* are compatible for the product operation (*see* Definition 6.8, page 147), and

(2) *TP* is complete with respect to *Spec* (*see* also Definition 6.8, page 147),

then due to Theorem 5.5 (*see* page 134) we obtain that the product operation preserves the set of traces of *Spec*, *i.e.* $Traces(Spec) = Traces(\underbrace{Spec \times TP}_{SP})$. Moreover, from Theorem 6.1 (*see* page 151) we know that accepting traces of the synchronous product $Spec \times TP$ are a subset of the traces of *Spec* intersected with the accepting traces of *TP*, *i.e.* $ATraces(Spec \times TP) \subseteq Traces(Spec) \cap ATraces(TP)$.

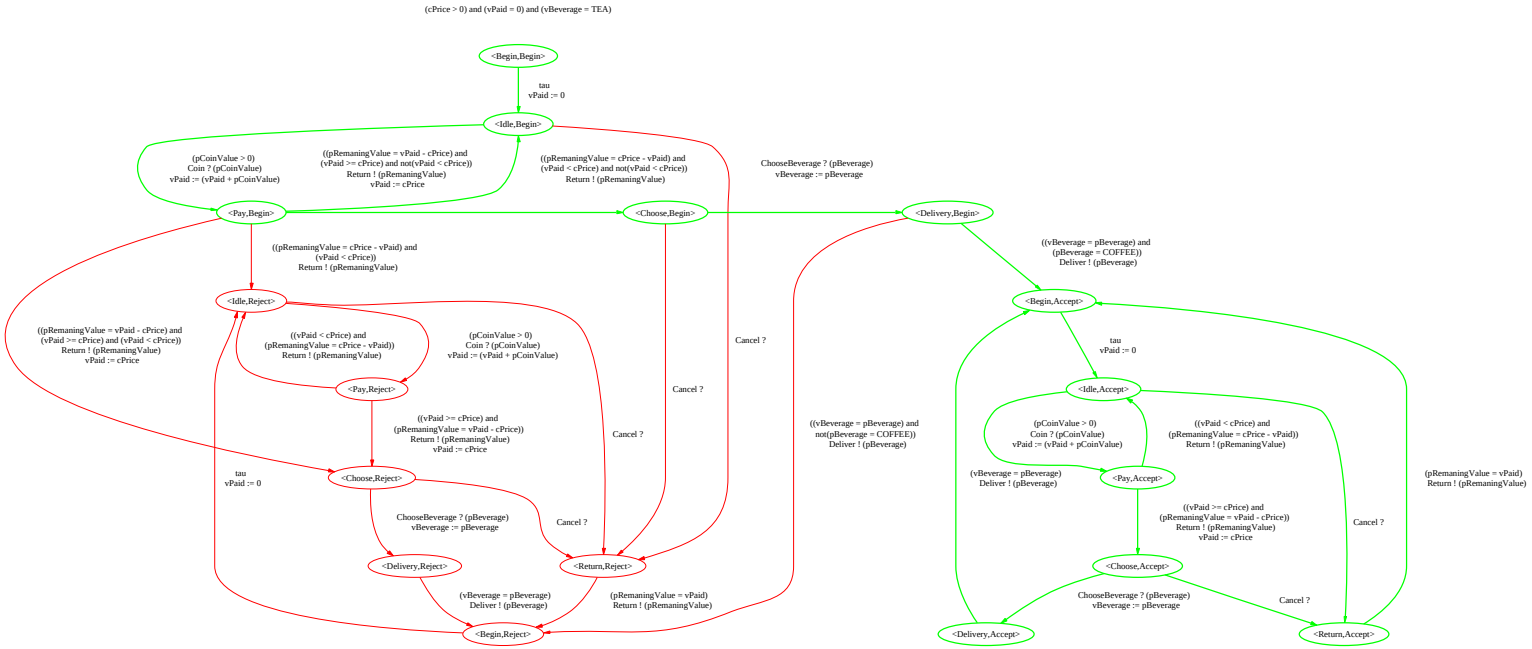


Figure 7.3: Synchronous product $SP = Spec \times TP'$.

Example 7.2 Figure 7.3 (*see* page 174) shows the result of the product computation for the specification of a coffee machine (*see* Figure 6.1, page 138) and the test purpose of this specification, which is depicted in Figure 7.2(b) (*see* page 172). This example does not describe how the product was computed, as this computation was explained in Section 5.2 (*see* Example 5.2, page 115). The purpose of the example is to emphasize that the product operation marks some locations of the given specification with *Accept*, making behaviors leading to them accepted by the test purpose. The accepted behaviors of the computed product are printed in green in Figure 7.3 (*see* page 174). All other behaviors, *i.e.* ones that are printed in red, are considered as rejected. The rejected behaviors indicate the behaviors of the specification for which the test case will not be generated. They will be eliminated on the next steps of the test generation method. □

The next steps of the test generation method described in the rest of this chapter consist in transforming and simplifying the product $SP = (Spec \times TP)$ in order to obtain a test case which is correct in the sense of Definition 6.18 (*see* page 163).

7.3 Construction of Visible Behaviors

It is important to emphasize that nondeterminism is prohibited in testing, as test verdicts should not depend on internal choices of the tester. That is why this step of the test generation method is reserved for elimination of internal actions from an IOSTS $SP = (Spec \times TP)$ obtained at the previous step of our symbolic test generation method (*i.e.* construction of $\mathsf{closure}(SP)$), and resolution of nondeterministic choices that remain for input/output actions of $\mathsf{closure}(SP)$ (*i.e.* construction of $\mathsf{det}(\mathsf{closure}(SP))$). For this we propose two syntactical operations closure and determinization such that:

$$Traces(SP) = Traces(\mathsf{closure}(SP)) = Traces(\mathsf{det}(\mathsf{closure}(SP))) \quad (7.1)$$

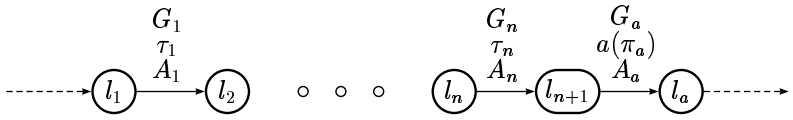
$$ATraces(SP) = ATraces(\mathsf{closure}(SP)) = ATraces(\mathsf{det}(\mathsf{closure}(SP))) \quad (7.2)$$

The syntactical procedures of closure and determinization and their properties are described in Appendixes A.1 and A.2 (*see* pages 246 and 274), and briefly summarized in the two following subsections.

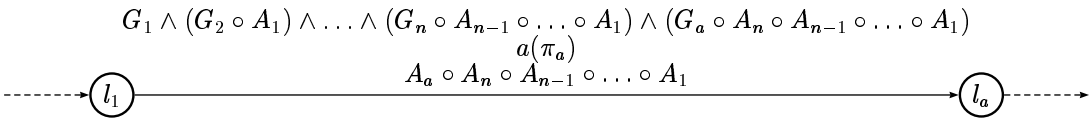
7.3.1 Closure: Eliminating Internal Actions

For eliminating internal actions from SP, the idea is to compute the effect of any sequence of internal actions that leads to an input- or output-labeled symbolic transition, and to encode this effect in the guard and assignments of the last symbolic transition.

This gives a simple syntactical procedure which terminates if the IOSTS SP does not have syntactic livelocks (*i.e.* cycles of internal actions). Notice that the condition about the absence of syntactic livelocks is a common hypothesis in conformance testing [Tretmans, 1999], which is made consistently through this document. Let $\rho : l_1 \overset{\tau_1}{\hookrightarrow} l_2 \ldots l_n \overset{\tau_n}{\hookrightarrow} l_{n+1}$ be a sequence of symbolic transi-



(a) A fragment of an IOSTS SP that illustrates one of its sequences of internal actions.



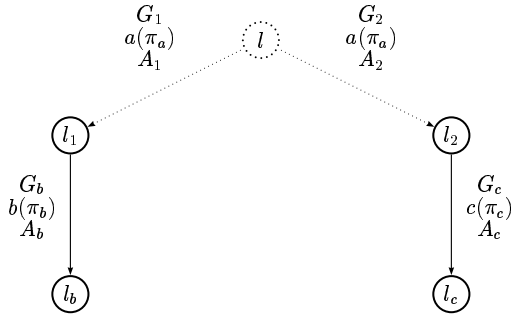
(b) A fragment of the IOSTS $\mathsf{closure}(SP)$ obtained from SP by closure.

Figure 7.4: An example of the IOSTS $\mathsf{closure}(SP)$.

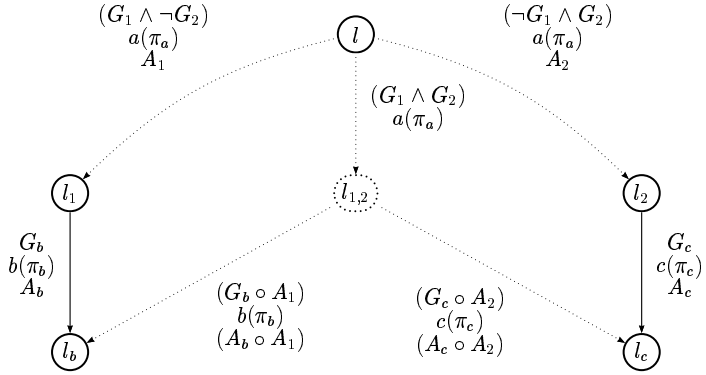
tion labeled with internal actions $\tau_1, \ldots, \tau_n$ that leads to the symbolic transition $l_{n+1} \overset{a}{\hookrightarrow} l_{n+2}$ labeled with either input or output action a (*see* Figure 7.4(a)). Assume that the guards and assignments corresponding to τ_i $(i = 1..n)$ are respectively G_i and A_i; and the guards and assignments corresponding to a are G_a and A_a. Then, in order to eliminate from the sequence ρ all internal actions, we replace ρ by one symbolic transition with origin l_1, target l_{n+2}, action a, guard $G_1 \wedge (G_2 \circ A_1) \wedge \ldots \wedge (G_n \circ A_{n-1} \circ \ldots \circ A_1) \wedge (G_n \circ A_n \circ A_{n-1} \circ \ldots \circ A_1)$, and assignments $A_a \circ A_n \circ A_{n-1} \circ \ldots \circ A_1$, where $\circ$ denotes function composition (*see* Figure 7.4(b)). Notice that such procedure conserves the effect of the sequence ρ.

7.3.2 Determinization

Determinization consists in *postponing* the effect if a non-deterministic choice on the observable actions that follow it. This leads to splitting, for instance, two symbolic transitions (with non-exclusive guards G_1 and G_2 and assignments A_1 and A_2 *see* Figure 7.5(a), page 177) into three: one for the case when $(G_1 \wedge \neg G_2)$ holds, another for the case when $(\neg G_1 \wedge G_2)$, and the last for the case when $(G_1 \wedge G_2)$ holds. In the letter case the choice whether to assign the variables according



(a) A fragment of an IOSTS $\mathsf{closure}(SP)$, where symbolic transitions t_1 et t_2 which are involved into a non-deterministic choice of $\mathsf{closure}(SP)$, are shown as dotted lines.



(b) A fragment of the IOSTS $\mathsf{det}(\mathsf{closure}(SP))$ obtained from SP by the procedure of determinization (the new/modified locations and symbolic transitions are shown as dotted lines).

Figure 7.5: Determinization of an IOSTS $\mathsf{closure}(SP)$.

to A_1 or A_2 is postponed until the observable action that follows. Thus, if b is the next action, then the assignments A_1 should have been executed. Hence the assignments A_1 is composed with the guard and the assignments corresponding to b. This results in the guard $(G_b \circ A_1)$ and the assignments $A_b \circ A_1$. Similarly, if c is the next action, then A_2 should be executed, which produces the guard $(G_c \circ A_2)$ and the assignments $A_c \circ A_2$. The result obtained from the IOSTS $\mathsf{closure}(SP)$ by the procedure explained above is shown on Figure 7.5(b) (*see* page 177)

It is important to notice that the procedure explained above may not terminate (*e.g.* if $a = b$ and $l_b = l$, the postponing goes forever). However, it does terminate for a non-trivial subclass IOSTS called IOSTS with lookahead k, which is defined in Appendix A.2.4 (*see* page 304).

7.3.3 Example Illustrating Closure and Determinization

Figure 7.6 illustrates the IOSTS $SP_{vis} = \mathsf{det}(\mathsf{closure}(SP))$ obtained after extraction of the observable behaviors from the IOSTS SP shown on Figure 7.3, page 174 (*i.e.* after application of the closure and determinization procedures explained in the previous two paragraphs). The blue edges on this figure indicate two new symbolic transitions replacing the two sequences of internal actions of the IOSTS SP. For more details the reader can check Example A.7 (*see* page 261).

7.4 Selection of a Test Graph

In this section we consider an IOSTS $SP_{vis} = \mathsf{det}(\mathsf{closure}(Spec \times TP)_{SP})$ obtained from a specification $Spec \in \mathsf{SPEC}$ and a test purpose TP of $Spec$ by the product operation (*see* Definition 5.4, page 114), the closure operation (*see* Definition A.7, page 260) and the procedure of global determinization (*see* page 304).

Due to the results given in Sections 7.2, A.1.5 and A.2.3 (*see* pages 173, 270 and 299), we have that:

(1) SP_{vis} has the same set of traces as $Spec$, *i.e.* $Traces(SP_{vis}) = Traces(Spec)$;

(2) moreover, the set $Traces(SP_{vis})$ contains accepting traces of SP_{vis} (*see* Definition A.12, page 300) which are the traces of $Spec$ selected by TP.

Next, we remind that the main purpose of the symbolic test generation method is to construct a test case that examines behaviors of the specification that are selected by the test purpose. Therefore, it is not necessary to keep all traces of SP_{vis} which would make a test case huge and unreadable. It is enough to choose traces of SP_{vis} that (1) lead to the accepting states of SP_{vis}, *i.e.* satisfaction of a test purpose TP, and (2) do not contain, or contain "fewer" (than SP_{vis}) unreachable states.

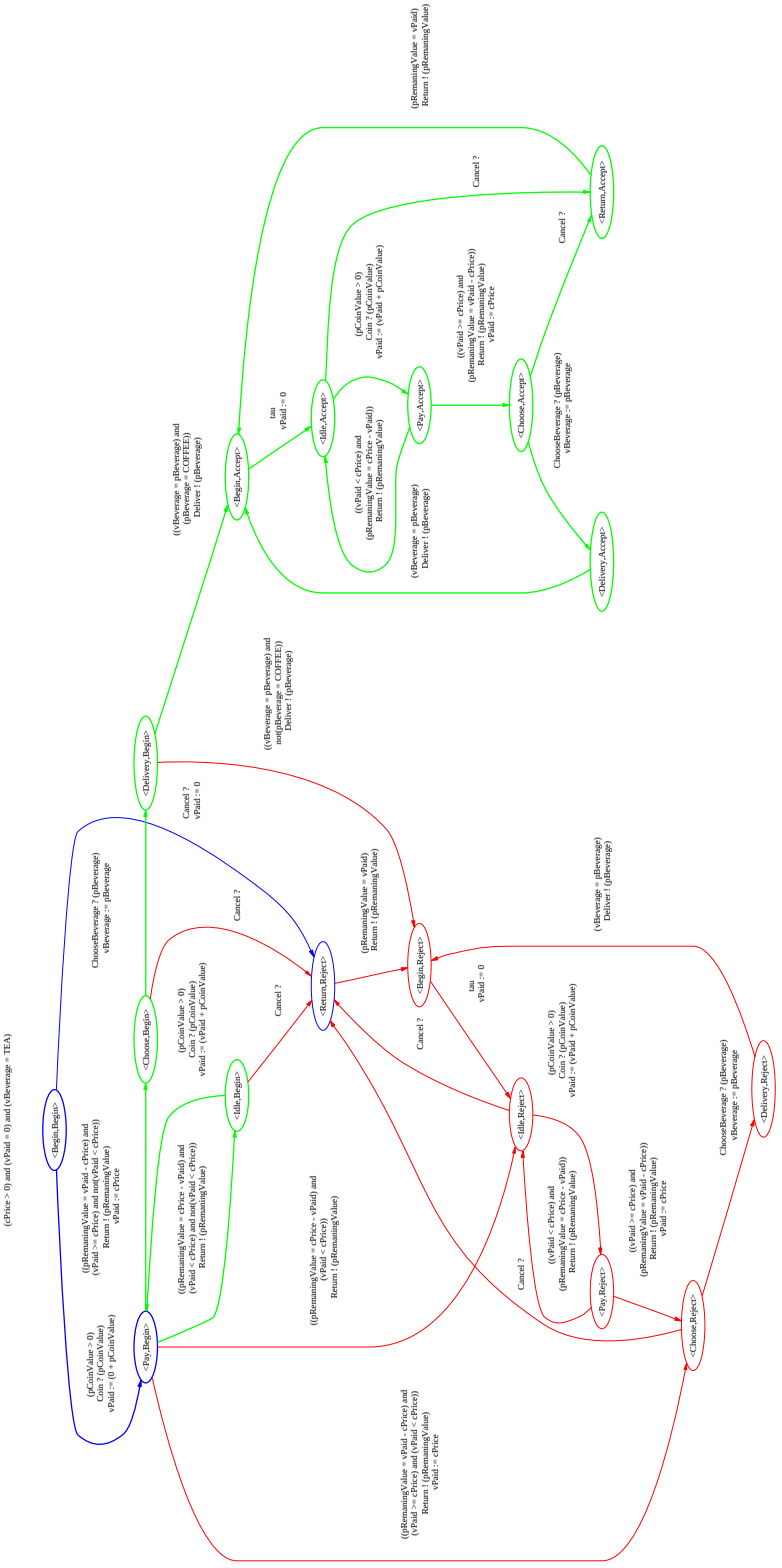


Figure 7.6: An example of the IOSTS $SP_{vis} = \mathsf{det}(\mathsf{cl}\mathsf{o}\mathsf{sure}(SP))$.

Plan of the Section. This section is organized as follows: at the beginning, we describe a methodology for symbolic analysis of IOSTS which is used in order to select a test graph TG from SP_{vis}. Then, we present the algorithm for the test graph selection. Finally, we formulate some properties preserved by this algorithm, which are needed for proving the correctness of the generated test cases.

7.4.1 Symbolic Analysis for IOSTS

The aim of this subsection is to present a methodology for analyzing input-output symbolic transition systems. The methodology consists of reachability and coreachability analysis (which are also known as forward and backward reachability analysis). It describes how to perform these analyses iteratively and emphasizes that the reachability and coreachability problems are undecidable in the general case, *i.e.* when the state space of the transition system is unbounded. Thus, in order to analyze IOSTS, we have to manipulate over-approximations of reachable and coreachable sets of states, rather than use the exact sets of reachable and coreachable states. It is important to emphasize that in this thesis we do not explain how to compute an over-approximation of the reachable/coreachable set of states. We make an assumption that this over-approximation can be computed by using, for instance, the control structure of an IOSTS, or intervals, or polyhedra (*see* [Cousot and Cousot, 1976], [Cousot and Cousot, 1977], [Cousot and Halbwachs, 1978], [Jeannet, 2000b]).

7.4.1.1 Reachability and Coreachability Analyses.

In this subsection we consider:

(1) a specification $Spec \in \mathsf{SPEC}$ with set of locations L_{Spec},

(2) a test purpose TP of $Spec$ with set of locations L_{TP}, and

(3) the IOSTS $SP_{vis} = \langle \underbrace{(V_{vis} \cup C_{vis} \cup P_{vis})}_{D_{vis}}, \Theta_{vis}, l^0_{vis}, L_{vis}, \underbrace{(\Sigma^?_{vis} \cup \Sigma^!_{vis})}_{\Sigma_{vis}}, T_{vis} \rangle$ without internal actions, which is obtained from $Spec$ and TP by applying the product and closure operations, and by performing the procedure of global determinization, *i.e.* $SP_{vis} = \mathsf{det}(\mathsf{closure}(Spec \times TP))$. The IOSTS SP_{vis} has the set of valued actions $\Lambda_{vis} = \Lambda^?_{vis} \cup \Lambda^!_{vis}$, set of states S_{vis}, set of initial states $S^0_{vis} \subseteq S_{vis}$. Moreover, for SP_{vis} we define a set of accepting states $S^{acc}_{vis} \subseteq S_{vis}$ as follows:

$$S^{acc}_{vis} \triangleq \tag{7.3}$$
$$\{\langle l_{vis}, \vartheta_{vis} \rangle \in S_{vis} \mid \vartheta_{vis} \in \mathsf{DOM}(V_{vis} \cup C_{vis}) \wedge$$
$$l_{vis} = \langle \langle l^1_{Spec}, l^1_{TP} \rangle, \ldots, \langle l^i_{Spec}, Accept \rangle, \ldots, \langle l^n_{Spec}, l^n_{TP} \rangle \rangle \in L_{vis}\}$$

where for all j from 1 to $n \geq 1$, $l^j_{Spec} \in L_{Spec}$ and $l^j_{TP} \in L_{TP}$, and $Accept$ is the special location of TP.

Then, in order to select a subgraph of SP_{vis} (called *test graph* and denoted by TG) leading to the satisfaction of the test purpose TP, and containing "fewer" unreachable states than SP_{vis}, we have to compute an intersection between the sets of reachable and coreachable states of given IOSTS SP_{vis}. We say that:

(1) a state $s_{vis} \in S_{vis}$ is *reachable* in SP_{vis} if there exists a trace $\sigma \in Traces(SP_{vis})$ that ends in s_{vis}, *i.e.* $\exists\, \sigma \in Traces(SP_{vis}),\ s^0_{vis} \in S^0_{vis} \,.\, [s^0_{vis} \stackrel{\sigma}{\Rightarrow} s_{vis}]$, and

(2) a state $s_{vis} \in S_{vis}$ is *coreachable* in SP_{vis} if from s_{vis} we *may go* to some accepting state $s^{acc}_{vis} \in S^{acc}_{vis}$, *i.e.* $\exists\, \sigma \in (\Lambda^? \cup \Lambda^!)^*, s^{acc}_{vis} \in S^{acc}_{vis} \,.\, [s_{vis} \stackrel{\sigma}{\Rightarrow} s^{acc}_{vis}]$.

This means that we have to perform reachability and coreachability analyses for the IOSTS SP_{vis}. These analyses are based on the predicate transformers post and pre which are used to compute successors and predecessors of some subset of states $S'_{vis} \subseteq S_{vis}$ of the given IOSTS SP_{vis}.

Predicate Transformers: $\mathsf{post}_t(\varphi)$ and $\mathsf{pre}_t(\varphi)$. Before giving the formal definitions of predicate transformers, we introduce two intermediate notion that are used later in this section.

First, let $l \in L_{vis}$ be a location of SP_{vis}, and $\varphi' \in \mathcal{B}(V_{vis} \cup C_{vis})$ be a predicate over the variables V_{vis} and symbolic constants C_{vis} of SP_{vis}. Then, the Boolean expression $\varphi_l : (pc = l) \wedge \varphi'$, where pc is a special variable that indicates the current location of the IOSTS SP_{vis}, is called the *projection* of φ' on l.

Second, we say that a state $s = \langle l, \vartheta \rangle \in S_{vis}$ of SP_{vis} *satisfies* the location predicate $\varphi \in \mathcal{B}(V_{vis} \cup C_{vis} \cup \{pc\})$ (denoted as $s \models \varphi$) *if and only if* (1) the variable pc is equal to the location l, and (2) the valuation ϑ of variables V_{vis} and symbolic constants C_{vis} satisfies φ'.

Definition 7.1 ($\mathsf{post}_t(\varphi)$) Let $t = \langle l, a, \pi, G, A, l' \rangle \in T_{vis}$ be a symbolic transition of the IOSTS SP_{vis}, and φ be a predicate over the variables and symbolic constants of SP_{vis}.

Then, the predicate transformer $\mathsf{post}_t(\varphi)$ of φ with respect to t is a predicate that characterizes the set of states reachable in one step from a state $s \in S_{vis}$ satisfying the predicate φ by executing the symbolic transition t, *i.e.*

$$\mathsf{post}_t(\varphi) \quad \triangleq \quad \exists\, s \in S_{vis},\ \alpha \in (\Lambda^?_{vis} \cup \Lambda^!_{vis}) \,.\, [s \xrightarrow{\alpha}_t s' \ \wedge \ s \models \varphi] \qquad (7.4)$$

□

Definition 7.2 ($\mathsf{pre}_t(\varphi)$) Let $t = \langle l, a, \pi, G, A, l' \rangle \in T_{vis}$ be a symbolic transition of the IOSTS SP_{vis}, and φ be a predicate over the variables and symbolic constants of SP_{vis}.

Then, the predicate transformer $\mathsf{pre}_t(\varphi)$ of φ with respect to t is a predicate that characterizes the set of states from which it is possible to reach the state $s' \in S_{vis}$ satisfying the predicate φ by executing the symbolic transition t, *i.e.*

$$\mathsf{pre}_t(\varphi) \quad \triangleq \quad \exists\ s' \in S_{vis}, \alpha \in (\Lambda^{!}_{vis} \cup \Lambda^{?}_{vis})\ .\ [s \xrightarrow{\alpha} s'\ \wedge\ s' \models \varphi] \tag{7.5}$$

□

Example 7.3 Consider the symbolic transition t of the IOSTS $\mathcal{S}$ depicted in Figure 6.1 (*see* page 138), that starts in the location *Idle* and leads to the location *Pay*. This symbolic transition is presented in Figure 7.7. Consider also a predicate

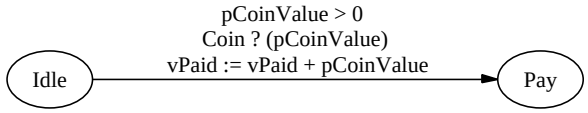


Figure 7.7: The symbolic transition of the IOSTS $\mathcal{S}$ (*see* Figure 6.1, page 138).

φ over the special variable pc, and the variables $V_{\mathcal{S}} = \{vPaid, vBeverage\}$ and symbolic constants $C_{\mathcal{S}} = \{cPrice\}$ of the IOSTS $\mathcal{S}$. For instance, φ is equal to $(pc = Idle) \wedge (cPrice > 0 \wedge vPaid = 0 \wedge vBeverage = \text{TEA})$. Then, the predicate transformer $\mathsf{post}_t(\varphi)$ of φ with respect to t is computed as follows:

$$\begin{aligned}
\mathsf{post}_t(\varphi) = \exists\ & pc, vPaid, vBeverage, cPrice, pCoinValue\ . \\
& [\ (pCoinValue > 0)\ \wedge \\
& \quad (pc' = Pay)\ \wedge \\
& \quad (vPaid' = vPaid + pCoinValue\ \wedge\ vBeverage' = vBeverage\ \wedge\ cPrice' = cPrice)\ \wedge \\
& \quad (pc = Idle\ \wedge\ cPrice > 0\ \wedge\ vPaid = 0\ \wedge\ vBeverage = \text{TEA})\ ] \\
& = \\
& \quad (pc' = Pay\ \wedge\ cPrice' > 0\ \wedge\ vPaid' > 0\ \wedge\ vBeverage' = \text{TEA})
\end{aligned}$$

If we compute the predicate transformer $\mathsf{pre}_t(\tilde{\varphi})$ of the predicate $\tilde{\varphi}$ which is the same as φ except of the variable pc is equal to Pay and not to $Idle$, with respect to the symbolic transition t shown in Figure 7.7, then we obtain:

$$\begin{aligned}
\mathsf{pre}_t(\tilde{\varphi}) = \exists\ & pc', vPaid', vBeverage', cPrice', pCoinValue\ . \\
& [\ (pCoinValue > 0)\ \wedge \\
& \quad (pc = Idle)\ \wedge \\
& \quad (vPaid' = vPaid + pCoinValue\ \wedge\ vBeverage' = vBeverage\ \wedge\ cPrice' = cPrice)\ \wedge \\
& \quad (pc' = Pay\ \wedge\ cPrice' > 0\ \wedge\ vPaid' = 0\ \wedge\ vBeverage' = \text{TEA})\ ] \\
& = \\
& \quad (pc = Idle\ \wedge\ cPrice > 0\ \wedge\ vPaid < 0\ \wedge\ vBeverage = \text{TEA})
\end{aligned}$$

□

Using the definitions given above we first describe the *exact* reachability and coreachability analyses for the given IOSTS SP_{vis} with set of states S_{vis}, set of initial states $S^0_{vis} \subseteq S_{vis}$, set of accepting states $S^{acc}_{vis} \subseteq S_{vis}$, and alphabet of valued actions $\Lambda = \Lambda^?_{vis} \cup \Lambda^!_{vis}$. It is important to emphasize that for IOSTS the exact computation of sets of reachable and coreachable states (proposed in the next two paragraphs) *does not terminate* in general. Thus in the sequel, instead of performing the exact computation, we calculate *over-approximations* of sets of reachable and coreachable states (*see* page 184).

Reachability Analysis. The purpose of reachability analysis is to compute a set of all states of the IOSTS SP_{vis} that are reachable from some state s^0 belonging to the set of initial states S^0_{vis} of SP_{vis}, *i.e.*

$$Reach(S^0_{vis}) \quad \triangleq \quad \{s \in S_{vis} \mid \exists\, \sigma \in Traces(SP_{vis}),\ s^0 \in S^0_{vis} \,.\, [s^0 \stackrel{\sigma}{\Rightarrow} s]\} \tag{7.6}$$

We can try to compute the exact set of reachable states $Reach(S^0_{vis})$ by (1) starting with the characteristic predicate S^0_{vis} that describes the set of initial states of the IOSTS SP_{vis}, and (2) iteratively increasing this set of states by taking its post-image until a fix point is reached, *i.e.* no new element is added into this set. More formally, we iteratively compute the set of reachable states of SP_{vis} as follows:

$$Reach(S^0_{vis}) \quad \triangleq \quad \bigvee_{n \in \mathbb{N}} \mathsf{post}^n(S^0_{vis}) \tag{7.7}$$

until the next computed element is already in $Reach(S^0_{vis})$. Here:

(1) for $n = 0$, $\mathsf{post}^0(S^0_{vis}) = S^0_{vis}$, and

(2) for $n > 0$, $\mathsf{post}^n(S^0_{vis}) = \bigvee_{t_1 \ldots t_n \in T^*_{vis}} [\mathsf{post}_{t_1 \ldots t_n}(S^0_{vis})]$, where

   (a) $t_1 \ldots t_n \in T^*_{vis}$ is a sequence of consecutive symbolic transitions that starts with the initial location of SP_{vis}, and

   (b) $\mathsf{post}_{t_1 \ldots t_n}(S^0_{vis}) \quad \triangleq \quad \mathsf{post}_{t_n}(\mathsf{post}_{t_{n-1}}(\ldots(\mathsf{post}_{t_1}(S^0_{vis}))\ldots))$.

However, this procedure does not always terminates for IOSTS. The example of a such IOSTS is shown on Figure 7.8 (*see* page 185).

Coreachability Analysis. The coreachability analysis in some sense is dual to the reachability one. Its aim is to compute the set of states of SP_{vis} which *may lead* to the set of accepting states S^{acc}_{vis} of SP_{vis}, *i.e.*

$$CoReach(S^{acc}_{vis}) \triangleq \{s \in S_{vis} \mid \exists\, \sigma \in (\Lambda^{?}_{vis} \cup \Lambda^{!}_{vis})^{*},\ s^{acc} \in S^{acc}_{vis} \,.\, [s \stackrel{\sigma}{\Rightarrow} s^{acc}]\} \qquad (7.8)$$

In order to compute the exact set of coreachable states we start from the characteristic predicate S^{acc}_{vis} representing the set of accepting states S^{acc}_{vis} of the IOSTS SP_{vis} and iteratively compute its pre-image until a fix point is reached. More formally, we iteratively compute the set of reachable states of SP_{vis} as follows:

$$CoReach(S^{acc}_{vis}) \triangleq \bigvee_{n \in \mathbb{N}} \mathsf{pre}^{n}(S^{acc}_{vis}) \qquad (7.9)$$

until the next computed element is already in $CoReach(S^{acc}_{vis})$. Here:

(1) for $n = 0$, $\mathsf{pre}^{0}(S^{acc}_{vis}) = S^{acc}_{vis}$, and

(2) for $n > 0$, $\mathsf{pre}^{n}(S^{acc}_{vis}) = \bigvee_{t_1 \ldots t_n \in T^{*}_{vis}} [\mathsf{pre}_{t_1 \ldots t_n}(S^{acc}_{vis})]$, where

   (a) $t_1 \ldots t_n \in T^{*}_{vis}$ is a sequence of consecutive symbolic transitions that ends in any location l such that there exists a state $s = \langle l, \vartheta \rangle \in S^{acc}_{vis}$,

   (b) $\mathsf{pre}_{t_1 \ldots t_n}(S^{acc}_{vis}) \triangleq \mathsf{pre}_{t_1}(\mathsf{pre}_{t_2}(\ldots(\mathsf{pre}_{t_n}(S^{acc}_{vis}))\ldots))$.

Notice that, this procedure (as the procedure for the computation of the exact set of reachable states described in the previous section) does not always terminates for IOSTS.

Over-Approximations of $Reach(S^{0}_{vis})$ and $CoReach(S^{acc}_{vis})$. The exact iterative computation of least fix points used in the reachability and coreachability analyses of IOSTS (*see* pages 183 and 184) is often very expensive for large input-output symbolic transition systems. Moreover, it is possible that this computation *does not terminate* since it might not converge to a fix point in a finite number of steps as it is shown below.

Indeed, if we consider the IOSTS depicted in Figure 7.8 (*see* page 185) that represents a counter which increases the value of the variable v by 2 each time when the input action A is executed. Intuitively, the least fix point of the considered IOSTS exists and it is the set of even natural numbers. However, the iterative computation of the least fix point used in reachability/coreachability analyses (*see* pages 183 and 184) does never converge. The reason is that at each step of the iterative computation we obtain a state which does not belong to the set of already computed states.

Therefore, it is not always possible to compute exact sets of reachable or coreachable states for an IOSTS SP_{vis}. However, there exist abstraction techniques

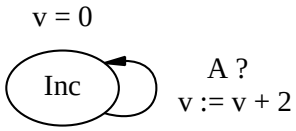


Figure 7.8: The IOSTS representing a counter.

which can be used for computation of over-approximations of the sought set of states (*see* [Cousot and Cousot, 1976], [Cousot and Cousot, 1977], [Cousot and Halbwachs, 1978], [Jeannet, 2000b]). In this work we do not give any details on how to compute the over-approximations of reachable and coreachable set of states of the IOSTS SP_{vis}, we just make the assumption that they exist.

7.4.2 Algorithm for the Test Graph Selection

At this point of the section we know what is the set of reachable and coreachable states of the IOSTS $SP_{vis} = \mathsf{det}(\mathsf{closure}(Spec \times TP))$, and we make an assumption about existence of an over-approximation of these sets of states. The purpose of this subsection is to propose an algorithm that transforms the given IOSTS SP_{vis} into a *test graph* TG which:

(1) – may lead to the satisfaction of the test purpose TP from which SP_{vis} was generated, and

– contains "fewer" unreachable states then the IOSTS SP_{vis}.

To solve these two problems the algorithm will use the given information about reachable and coreachable states of SP_{vis}.

(2) is able to communicate with an implementation under test $iut \in Iuts$. This means that input actions of iut should be considered as output actions of TG, and vice versa. Notice that, the actions of the IOSTS SP_{vis} have the same directions (input/output) as the actions of iut (*see* the construction of SP_{vis}). Thus, in order to obtain the test graph which can communicate with iut, the directions of the actions of SP_{vis} must be inverted.

The algorithm for the test graph selection is presented below.

Algorithm 7.2 (Selection of the Test Graph) Let

– $SP_{vis} = \langle (V_{vis} \cup C_{vis} \cup P_{vis}), \Theta_{vis}, l^0_{vis}, L_{vis}, (\Sigma^?_{vis} \cup \Sigma^!_{vis}), T_{vis} \rangle$ be an IOSTS which is obtained from a specification *Spec* and a test purpose TP of *Spec* by applying the product and closure operations, and performing the procedure of global determinization,

– $\mathsf{Reach}(SP_{vis})$ and $\mathsf{CoReach}(SP_{vis})$ be two predicates characterizing over-approximations of the sets of reachable and coreachable states of the IOSTS SP_{vis}, and

– for each location $l \in L_{vis}$ of SP_{vis}, $\mathsf{reach}(l)$ and $\mathsf{coreach}(l)$ be two predicates that characterize over-approximations of reachable and coreachable states of the form $\langle l, \vartheta \rangle$, where $\vartheta \in \mathsf{DOM}(V_{vis} \cup C_{vis})$.

  In the sequel, we also use the notation $\mathsf{coreach}(l)[V/A]$, where A is a set of assignments for variables V and l is a location. It denotes the predicate characterizing an over-approximation of coreachable states of the form $\langle l, \vartheta \rangle$, where each variable $v \in V$ is substituted with the right-hand side of the corresponding assignments $A_v \in A$.

Then, the test graph TG is the IOSTS

$$\langle\ (V_{TG} \cup C_{TG} \cup P_{TG}),\ \Theta_{TG},\ l^0_{TG},\ L_{TG},\ (\Sigma^?_{TG} \cup \Sigma^!_{TG}),\ T_{TG}\ \rangle$$

obtained from SP_{vis} as follows:

(1) $V_{TG} = V_{vis}$, $C_{TG} = C_{vis}$ and $P_{TG} = P_{vis}$.

(2) $\Theta_{TG} = \Theta_{vis} \wedge \mathsf{CoReach}(SP_{vis})$.

(3) If $l^0_{vis} = Accept$, then $l^0_{TG} = Pass$. Otherwise, $l^0_{TG} = l^0_{vis}$.

(4)[1] $\Sigma^?_{TG} = \Sigma^!_{vis}$ and $\Sigma^!_{TG} = \Sigma^?_{vis}$.

[1] In order to obtain a test graph TG that is able to communicate with an implementation under test, we must invert the alphabets of input/output actions of the IOSTS SP_{vis}.

(5) The set of symbolic transitions of TG is $T_{TG} \triangleq T_{TG}^{\mathsf{L2A}} \cup T_{TG}^{\mathsf{Inconc}}$, where:

(a) $T_{TG}^{\mathsf{L2A}} \triangleq \{ \langle l, a, \pi, G_{TG}, A, l_{TG} \rangle \mid$
$(\exists \langle l, a, \pi, G, A, l' \rangle \in T_{vis} \,.\, [l \neq Accept \wedge a \in (\Sigma_{vis}^{?} \cup \Sigma_{vis}^{!})]) \wedge$
$(G_{TG} = (G \wedge \mathsf{reach}(l) \wedge \mathsf{coreach}(l')[V_{TG}/A])$ is satisfiable$) \wedge$
$(l' = Accept \implies l_{TG} = Pass) \wedge (l' \neq Accept \implies l_{TG} = l') \}$
is the set of transitions leading either the a $Pass$ location, or to a location from which it is possible to reach some $Pass$ location, and

(b) $T_{TG}^{\mathsf{Inconc}} \triangleq \{ \langle l, a, \pi, G_{TG}, A, Inconclusive \rangle \mid$
$(\exists \langle l, a, \pi, G, A, l' \rangle \in T_{vis} \,.\, [l \neq Accept \wedge a \in \Sigma_{vis}^{!}]) \wedge$
$(G_{TG} = (G \wedge \mathsf{reach}(l) \wedge \neg\mathsf{coreach}(l')[V_{TG}/A])$ is satisfiable$) \}$
is the set of transitions leading to *inconclusive* locations and labeled with *input* actions of TG.

(6) The set of locations L_{TG} consists of origins and targets of the above computed symbolic transitions T_{TG}, *i.e.*

$$L_{TG} \triangleq \bigcup_{\langle l,a,\pi,G,A,l' \rangle \in T_{TG}} \{l, l'\}$$

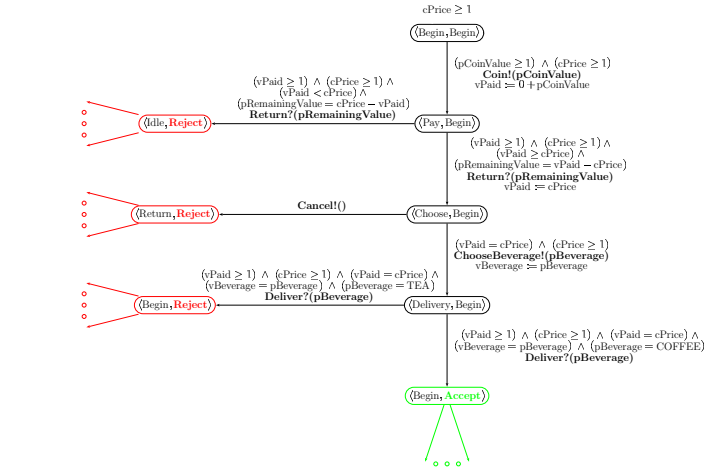
□

Bellow we present two examples that illustrate the selection algorithm. More precisely, we first explain how to use this algorithm in practice, and then we compare test graphs obtained by the same selection algorithm but with two different kinds of abstractions.

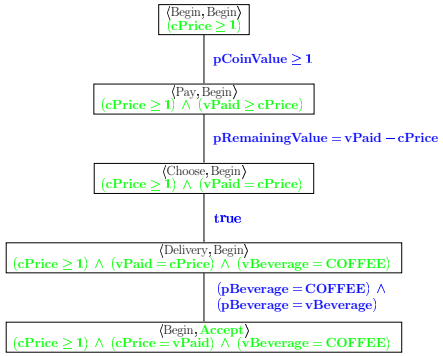
Example 7.4 The purpose of this example is to explain how the selection algorithm can be used in practice. One of the key points of this algorithm is the use of information about reachable and coreachable states of the analyzed IOSTS. This information can be obtained by any tool performing symbolic analyses. In the test generation tool, described in the third part of the thesis, we use NBac [Jeannet, 2000a] that approximates a given symbolic system with a polyhedron, and then performs reachability/coreachability analyses on it.

Consider the IOSTS SP'_{vis} shown on Figure 7.9(a) (*see* page 188). Due to the sake of simplicity, we assume that the alphabets of input/output actions of SP'_{vis} have been already inversed. In order to select a test graph TG from SP'_{vis}, we:

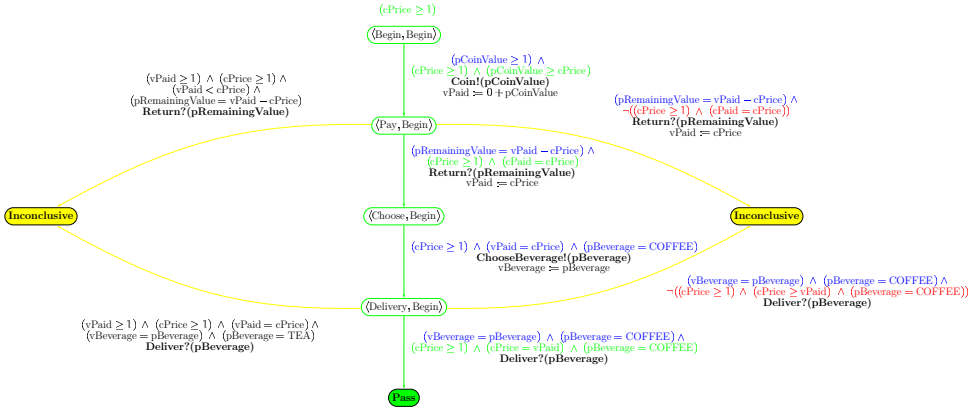
First, perform the approximate coreachability analysis starting from the accepting states (shown in green on Figure 7.9(a)) of SP'_{vis}. This analysis gives us an over-approximation of all states of SP'_{vis} from which we *may go* to accepting states. The result of the coreachability analysis is shown on Figure 7.9(b) (*see* page 188).



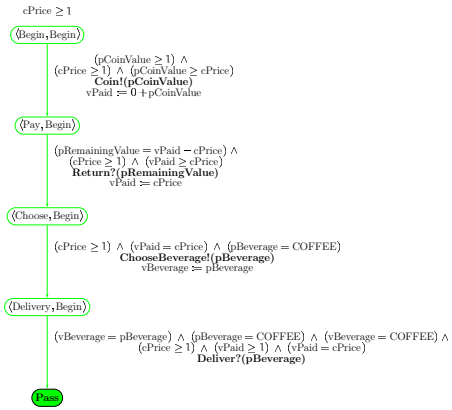
(a) $SP'_{vis} = \mathsf{det}(\mathsf{closure}(SP'))$.



(b) An over-aproximation of coreachable states of SP'_{vis}.



(c) SP''_{vis} obtained from SP'_{vis} after an approximated coreachability analysis.



(d) TG obtained from SP''_{vis} after an approximated reachability analysis.

Figure 7.9: An example illustrating the selection algorithm.

Second, using the information obtained at the first step, we modify SP'_{vis} as it is indicated in the selection algorithm. More precisely, we:

(1) remove the symbolic transitions (a) that are labeled with an *output action*, and (b) that leave the computed set of coreachable states of SP'_{vis}.

For instance, the symbolic transition labeled with the *Cancel* output action does not appear in SP''_{vis} because if we take this transition, it will not be possible to reach any accepting state of SP'_{vis}.

(2) redirect the symbolic transitions (a) that are labeled with an *input action*, and (b) that leave the computed set of coreachable states of SP'_{vis}, to an *Inconclusive* location.

For example, see the two symbolic transitions leading to the left *Inconclusive* location of SP''_{vis} depicted in Figure 7.9(c).

(3) modify the guards of the other symbolic transitions by taking into account the information about coreachable states.

For instance, consider the symbolic transition t leading to the location $\langle Pay, Begin\rangle$. We know that in order to reach some accepting state the price of the beverage should be strictly positive and the amount paid by the user should be greater or equal to the price (*i.e.* $(cPrice \geq 1) \wedge (vPaid \geq cPrice)$ which was computed during the approximate coreachability analysis, *see* the second node the graph shown on Figure 7.9(b)). This simply means that the user should introduce a coin whose value is greater or equal to *cPrice*. Notice that this consequence can be computed automatically, by taking the *pre*-image of coreachable states corresponding to the location $\langle Pay, Begin\rangle$. Therefore, in order to increase the probability to reach some accepting state, we should strengthen the guard of t with $(cPrice \geq 1) \wedge (pCoinValue \geq cPrice)$. The result of this operation is shown on Figure 7.9(c).

However, by strengthening guards of symbolic transitions labeled with an input action we risk to loose the soundness of the resulting test graph (and therefore, of the future test case). Therefore, in the case when we modify guards of such symbolic transitions, we also need to add other symbolic transitions that leave the set of coreachable states and lead to an *Inconclusive* location. The example of such symbolic transitions is shown on Figure 7.9(c) (*see* the symbolic transitions leading to the right *Inconclusive* location).

Finally, as the IOSTS SP''_{vis} can contain unreachable states, we try to detect and eliminate them by performing an approximated reachability analysis. It is

not hard to check that any state corresponding to an *Inconclusive* location is not reachable. Therefore, the symbolic transitions leading to *Inconclusive* will be removed. The test graph obtained after the reachability analysis is depicted in Figure 7.9(d) (*see* page 188).

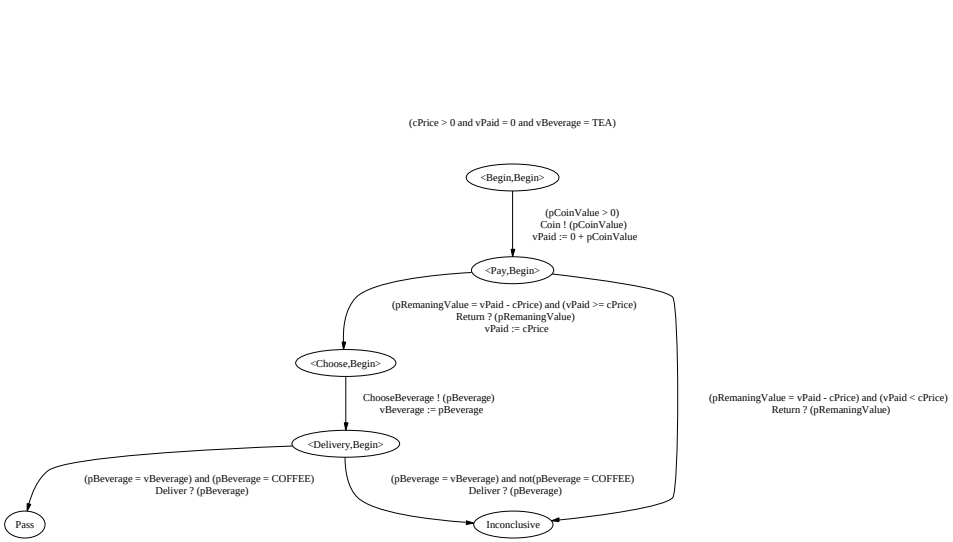
□

Example 7.5 Consider an IOSTS SP_{vis} depicted in Figure A.3 (*see* page 262). Then, we explain how to use the selection algorithm to produce different test graphs.

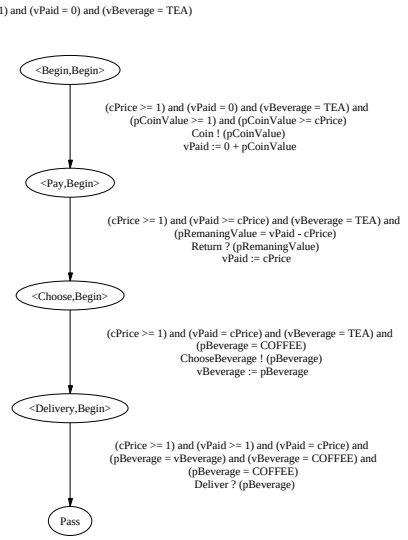
If we compute over-approximations of reachable and coreachable set of states based on only control structure of SP_{vis} (*i.e.* for each location l of SP_{vis}, $\mathsf{reach}(l)$ and $\mathsf{coreach}(l)$ are defined as the predicate $pc = l$, where pc is a control variable), then we obtain the test graph TG_1 shown on Figure 7.10(a) (*see* page 191). In this case the selection algorithm is very similar to one used in the test generation method for IOLTS (*see* Algorithm 3.2, page 59). Notice that this approximation is rough as it does not take into account data of SP_{vis}.

The second test graph TG_2 (*see* Figure 7.10(b), page 191) was constructed by using the over-approximations of sets of reachable and coreachable states obtained by the symbolic analysis of SP_{vis} abstracted with polyhedra [Cousot and Halbwachs, 1978], [Jeannet, 2000b] (in this case $\mathsf{reach}(l)$ and $\mathsf{coreach}(l)$ are definitely stronger than $pc = l$). As it is easy to see from Figure 7.10 (*see* page 191), the selection based on this abstraction is more precise then one described above. Indeed, it successfully eliminates two *Inconclusive* locations appeared in TG_1. The first *Inconclusive* locations is eliminated by strengthening the guard of the first symbolic transition of TG_1 in such way that it is possible to reach the *Pass* location. For this we have used the information that the first paid amount must be enough to receive the beverage. The second *Inconclusive* location is eliminated by using the same principle as above, *i.e.* by straightening the guard of the symbolic transition where the user chooses a beverage. Indeed, we force the user to choose a coffee in order to obtain this coffee. □

Next, we study the trace relation between IOSTS SP_{vis} and the test graph TG obtained from SP_{vis} by the selection algorithm (*see* page 185). First, we notice that the algorithm (1) does not augment SP_{vis} with any new symbolic transition and (2) does not weaken guards of any symbolic transition, therefore, the set of traces of TG cannot be bigger than the set of traces of SP_{vis}. However, we definitely cannot obtain the equality between $\mathit{Traces}(TG)$ and $\mathit{Traces}(SP_{vis})$. Indeed, the purpose of the selection algorithm is to choose (1) the traces of SP_{vis} leading to states from which it is possible to go to accepting states, and (2) the



(a) TG_1



(b) TG_2

Figure 7.10: Test graphs obtained from the IOSTS SP_{vis} shown on Figure A.3 (*see* page 262) by the selection algorithm.

traces of SP_{vis} ending with a valued *input* action and leading to states from which is not possible to go to any accepting state, and whose strict prefixes are traces leading to states from which it is possible to go to some accepting state. (In the last case the test graph TG should produce the *Inconclusive* verdict.) Thus, it is clear that the set of traces of TG is included into the set of traces of SP_{vis}. To prove this fact we first show the relation between behaviors of TG and SP_{vis} given below.

Theorem 7.1 (Behaviors of TG) Let $TG = \langle D, \Theta_{TG}, L_{TG}, l^0, \Sigma, T_{TG} \rangle$ be a test graph obtained from an IOSTS $SP_{vis} = \langle D, \Theta_{vis}, L_{vis}, l^0, \Sigma, T_{vis} \rangle$ by Algorithm 7.2 (*see* page 185). Then, for each sequence of valued actions $\sigma = \alpha_1 \ldots \alpha_n \in (\Lambda^?_{TG} \cup \Lambda^!_{TG})^* = (\Lambda^?_{vis} \cup \Lambda^!_{vis})^*$, if the test graph TG has the behavior:

$$\beta^{\rightarrow}_{TG} : \langle l^0_{TG}, \vartheta^0_{TG} \rangle \xrightarrow{\alpha_1} \langle l^1_{TG}, \vartheta^1_{TG} \rangle \ldots \langle l^{n-1}_{TG}, \vartheta^{n-1}_{TG} \rangle \xrightarrow{\alpha_n} \langle l^n_{TG}, \vartheta^n_{TG} \rangle$$

then there exists a location $l \in L_{vis}$ such that the IOSTS SP_{vis} has the behavior:

$$\beta^{\rightarrow}_{vis} : \langle l^0_{vis}, \vartheta^0_{vis} \rangle \xrightarrow{\alpha_1} \langle l^1_{vis}, \vartheta^1_{vis} \rangle \ldots \langle l^{n-1}_{vis}, \vartheta^{n-1}_{vis} \rangle \xrightarrow{\alpha_n} \langle l, \vartheta^n_{vis} \rangle$$

where $\forall\ i \in [0, n-1]\ .\ [l^i_{TG} = l^i_{vis}\ \wedge\ \vartheta^i_{TG} = \vartheta^i_{vis}]$ and $\vartheta^n_{TG} = \vartheta^n_{vis}$. □

Proof The proof is done by induction on the length of the sequence σ of valued actions belonging to $(\Lambda^?_{TG} \cup \Lambda^!_{TG})^* = (\Lambda^!_{vis} \cup \Lambda^?_{vis})^*$.

Induction Basis. Consider the empty sequence of valued actions ε, and a behavior of the test graph TG corresponding to ε, *i.e.* $\beta^{\rightarrow}_{TG} : \langle l^0_{TG}, \vartheta^0_{TG} \rangle$, where l^0_{TG} is the initial location of TG, and ϑ^0_{TG} is an initial valuation of the variables and symbolic constants of TG. Then, we prove that there exists an initial location l^0_{vis} such that $\beta^{\rightarrow}_{vis} : \langle l^0_{vis}, \vartheta^0_{vis} \rangle$ is a behavior of SP_{vis}, where $\vartheta^0_{vis} = \vartheta^0_{TG}$.

(1) As the test graph TG is obtained from the IOSTS SP_{vis} by Algorithm 7.2 (*see* page 185), then due to the item (3) of this algorithm we know that the initial location l^0_{vis} exists.

(2) Next, as $\langle l^0_{TG}, \vartheta^0_{TG} \rangle$ is an initial state of TG, then ϑ^0_{TG} satisfies the initial condition of TG, *i.e.* $\vartheta^0_{TG} \models \Theta_{TG}$ which by construction (*see* the item (2) of Algorithm 7.2, page 185) is equal to $(\Theta_{vis} \wedge \mathsf{CoReach}(SP_{vis}))$, where Θ_{vis} is the initial condition of the IOSTS SP_{vis}. This implies that the valuation ϑ^0_{TG} also satisfies Θ_{vis}, thus it can be used as an initial valuation of the variables and symbolic constants of SP_{vis} (remember that $V_{TG} = V_{vis}$ and $C_{TG} = C_{vis}$ due to the item (1) of the selection algorithm).

The items (1) and (2) imply that $\beta^{\rightarrow}_{vis} : \langle l^0_{vis}, \vartheta^0_{vis}\rangle$, where $\vartheta^0_{vis} = \vartheta^0_{TG}$, is a behavior of SP_{vis} corresponding to the empty trace ε.

Induction Hypothesis. Assume that for a sequence of valued actions $\sigma' = \alpha_1 \ldots \alpha_{n-1}$ of length $n-1$, if the test graph TG has the behavior:

$$\beta^{\rightarrow}_{TG} : \langle l^0_{TG}, \vartheta^0_{TG}\rangle \stackrel{\alpha_1}{\rightarrow} \langle l^1_{TG}, \vartheta^1_{TG}\rangle \ldots \langle l^{n-2}_{TG}, \vartheta^{n-2}_{TG}\rangle \stackrel{\alpha_{n-1}}{\rightarrow} \langle l^{n-1}_{TG}, \vartheta^{n-1}_{TG}\rangle \qquad (7.10)$$

then there exists a location $l \in L_{vis}$ such that the IOSTS SP_{vis} has the behavior:

$$\beta^{\rightarrow}_{vis} : \langle l^0_{vis}, \vartheta^0_{vis}\rangle \stackrel{\alpha_1}{\rightarrow} \langle l^1_{vis}, \vartheta^1_{vis}\rangle \ldots \langle l^{n-2}_{vis}, \vartheta^{n-2}_{vis}\rangle \stackrel{\alpha_{n-1}}{\rightarrow} \langle l, \vartheta^{n-1}_{vis}\rangle \qquad (7.11)$$

where $\forall\ i \in [0, n-2]\ .\ [l^i_{TG} = l^i_{vis}\ \wedge\ \vartheta^i_{TG} = \vartheta^i_{vis}]$ and $\vartheta^{n-1}_{TG} = \vartheta^{n-1}_{vis}$.

Induction Step. Consider a sequence of valued actions of length n, *i.e.* $\sigma = \underbrace{\alpha_1 \ldots \alpha_{n-1}}_{\sigma'}\alpha_n$. We prove that if:

$$\beta^{\rightarrow}_{TG} : \langle l^0_{TG}, \vartheta^0_{TG}\rangle \stackrel{\alpha_1}{\rightarrow} \langle l^1_{TG}, \vartheta^1_{TG}\rangle \ldots \langle l^{n-1}_{TG}, \vartheta^{n-1}_{TG}\rangle \stackrel{\alpha_n}{\rightarrow} \langle l^n_{TG}, \vartheta^n_{TG}\rangle \qquad (7.12)$$

is a behavior of TG, then there exists $l \in L_{vis}$ such that

$$\beta^{\rightarrow}_{vis} : \langle l^0_{vis}, \vartheta^0_{vis}\rangle \stackrel{\alpha_1}{\rightarrow} \langle l^1_{vis}, \vartheta^1_{vis}\rangle \ldots \langle l^{n-1}_{vis}, \vartheta^{n-1}_{vis}\rangle \stackrel{\alpha_n}{\rightarrow} \langle l, \vartheta^n_{vis}\rangle \qquad (7.13)$$

is a behavior of SP_{vis}, where $\forall\ i \in [0, n-1]\ .\ [l^i_{TG} = l^i_{vis}\ \wedge\ \vartheta^i_{TG} = \vartheta^i_{vis}]$ and $\vartheta^n_{TG} = \vartheta^n_{vis}$.

By using prefix closure and the induction hypothesis, we obtain that for the sequence σ', if the prefix of $\beta^{\rightarrow}_{TG}$ (*see* Formula (7.12)) whose length is $n-1$, is a behavior of TG, then there exists $l^{n-1}_{vis} \in L_{vis}$ such that the prefix of $\beta^{\rightarrow}_{vis}$ (*see* Formula (7.13)) whose length is also equal to $n-1$, is a behavior of SP_{vis}. Moreover, we know that $l^{n-1}_{vis} = l^{n-1}_{TG}$. Indeed, due to the selection algorithm (*see* page 185) l^{n-1}_{vis} can be different from l^{n-1}_{TG} *only* in the case where l^{n-1}_{TG} is equal either to *Pass*, or to *Inconclusive*. Remember that by construction neither *Pass* nor *Inconclusive* has a successor. However, l^{n-1}_{TG} has at least one successor which is l^n_{TG} (*see* the last transition relation of Formula (7.12)). Thus, l^{n-1}_{vis} is equal to l^{n-1}_{TG}.

Finally, we prove that as the last transition relation of the behavior $\beta^{\rightarrow}_{TG}$ (*see* Formula (7.12)), *i.e.*

$$\hat{t}^n_{TG} : \langle l^{n-1}_{TG}, \vartheta^{n-1}_{TG}\rangle \stackrel{\alpha_n = \langle a, \omega\rangle}{\longrightarrow} \langle l^n_{TG}, \vartheta^n_{TG}\rangle$$

holds in TG, then there exists a location $l \in L_{vis}$ such that the last transition relation of the behavior $\beta^{\rightarrow}_{vis}$ (*see* Formula (7.13), *i.e.*

$$\hat{t}^{n}_{vis} : \langle \underbrace{l^{n-1}_{vis}}_{= l^{n-1}_{TG}} , \underbrace{\vartheta^{n-1}_{vis}}_{= \vartheta^{n-1}_{TG}} \rangle \xrightarrow{\alpha_n = \langle a, \omega \rangle} \langle l, \vartheta^{n}_{vis} \rangle$$

holds in SP_{vis}, and $\vartheta^{n}_{vis} = \vartheta^{n}_{TG}$.

Due to the definition of a transition relation (*see* page 86), we know that TG has a symbolic transition:

$$t^{n}_{TG} = \langle l^{n-1}_{TG}, a, \pi, G_{TG}, A, l^{n}_{TG} \rangle$$

such that $\langle \vartheta^{n-1}_{TG}, \omega \rangle \models G_{TG}$ and $\vartheta^{n}_{TG} = A(\langle \vartheta^{n-1}_{TG}, \omega \rangle)$.

Next, as TG is obtained from the IOSTS SP_{vis} by Algorithm 7.2 (*see* page 185), then due to the item (5) of this algorithm, SP_{vis} has the following symbolic transition:

$$t^{n}_{vis} = \langle \underbrace{l^{n-1}_{vis}}_{l^{n-1}_{TG}}, a, \pi, G_{vis}, A, l \rangle$$

Moreover, due to the item (5) of Algorithm 7.2 (*see* page 185), we know that the guard G_{TG} is equal to either:

- $(G_{vis} \wedge \mathsf{reach}(l^{n-1}_{vis}) \wedge \mathsf{coreach}(l)[V_{TG}/A])$, or
- $(G_{vis} \wedge \mathsf{reach}(l^{n-1}_{vis}) \wedge \neg\mathsf{coreach}(l)[V_{TG}/A])$.

Thus, as $\langle \vartheta^{n-1}_{TG}, \omega \rangle \models G_{TG}$ and $\vartheta^{n-1}_{TG} = \vartheta^{n-1}_{vis}$, then $\langle \vartheta^{n-1}_{vis}, \omega \rangle$ satisfies G_{vis}. This means that the symbolic transition t^{n}_{vis} of SP_{vis} is executable. Also, as the symbolic transitions t^{n}_{TG} and t^{n}_{vis} have the same set of assignments A, then $\vartheta^{n}_{vis} = A(\langle \underbrace{\vartheta^{n-1}_{vis}}_{= \vartheta^{n-1}_{TG}}, \omega \rangle) = \vartheta^{n}_{TG}$.

Therefore, we obtain that the last transition relation $\hat{t}^{n}_{vis}$ of the behavior $\beta^{\rightarrow}_{vis}$ (*see* Formula (7.13)) holds in SP_{vis}. This proves the induction step of the theorem and the whole theorem.

Q.E.D.

Next, notice that the alphabets of the test graph TG and the IOSTS SP_{vis} are the same, and they consist of only input and output actions. Therefore, using the theorem above, we directly obtain the following corollary:

Corollary 7.1 (Traces of TG) Let TG be a test graph obtained from the IOSTS SP_{vis} by Algorithm 7.2 (*see* page 185). Then, all traces of TG are included in the set of traces of SP_{vis}, *i.e.*

$$Traces(TG) \subseteq Traces(SP_{vis}) \tag{7.14}$$

□

Finally, we prove that the selection algorithm (*see* page 185) produces a deterministic test graph. Formally:

Lemma 7.1 (TG is Deterministic) A test graph TG obtained from an IOSTS SP_{vis} by the selection algorithm presented above is deterministic in the sense of Definition 4.20 (*see* page 93). □

Proof

(1) We show that the test graph TG does not contain any internal actions.

Indeed, notice that the IOSTS SP_{vis} does not have any internal actions by construction (thanks to the closure operation defined on page 260). Thus, as TG is obtained from SP_{vis} by the selection algorithm (*see* page 185), then it also does not contain internal actions. Therefore, $\Lambda^{\tau} = \emptyset$.

(2) We prove that from each state of the test graph TG at most one symbolic transition can be fired, *i.e.* $\forall\ s \in S_{TG},\ \alpha \in (\Lambda^{?}_{TG} \cup \Lambda^{!}_{TG})\ .\ [card(\{s' \in S_{TG} \mid s \xrightarrow{\alpha} s'\}) \leq 1]$.

Indeed, we know that:

(a) the IOSTS SP_{vis} is deterministic by construction (thanks to the procedure of the global determinization given on page 304), and

(b) the selection algorithm (*see* page 185) transforms the IOSTS SP_{vis} to the test graph TG either (1) by removing non-executable symbolic transitions and symbolic transitions starting in and leading to *Accept* locations; or (2) by splitting each symbolic transitions of SP_{vis} labeled by action into two whose guards are disjoint.

Therefore, from the items (a) and (b) we can deduce the second statement of the lemma.

The items (1) and (2) imply that the test graph TG is deterministic (*see* Definition 4.20, page 93). **Q.E.D.**

7.4.3 Traces of TG Leading to Pass/Inconclusive States

Remember that in the previous subsection we proposed an algorithm which from a given IOSTS $SP_{vis} = \mathsf{det}(\mathsf{closure}(Spec \times TP))$, where $Spec$ is a specification and TP is a test purpose of $Spec$, generates a test graph TG which contains $Pass$ and $Inconclusive$ locations, and which can be executed on some implementation under test. Therefore, a set of traces of TG contains traces leading to $Pass$ and $Inconclusive$ states. In this section we are interested in studying some properties of such traces. In order to formally manipulate traces leading to $Pass$ or $Inconclusive$, we must give formal definitions for them.

Definition 7.3 (Traces of TG Leading to Pass/Inconclusive) Let TG be the test graph with set of states S_{TG}, set of initial states $S^0_{TG} \subseteq S_{TG}$, and set of pass and inconclusive states, *i.e.* $\mathsf{PASS} \subseteq S_{TG}$ and $\mathsf{INCONCLUSIVE} \subseteq S_{TG}$. Then, for TG we define two sets of traces leading to states belonging to either PASS or $\mathsf{INCONCLUSIVE}$, as follows:

$$Traces_{pass}(TG) \triangleq \{\sigma \in Traces(TG) \mid \exists\ s^0 \in S^0_{TG},\ s_{pass} \in \mathsf{PASS}\ .\ [s^0 \stackrel{\sigma}{\Rightarrow} s_{pass}]\} \tag{7.15}$$

$$Traces_{inconc}(TG) \triangleq \{\sigma \in Traces(TG) \mid \exists\ s^0 \in S^0_{TG},\ s_{inconc} \in \mathsf{INCONCLUSIVE}\ .\ [s^0 \stackrel{\sigma}{\Rightarrow} s_{inconc}]\} \tag{7.16}$$

Notice that in the rest of the thesis any trace belonging to $Traces_{pass}(TG)$ (*resp.* to $Traces_{inconc}(TG)$) will be called *pass trace* (*resp. inconclusive trace*). □

Now that the pass and inconclusive traces of the test graph TG are formally defined, we can study the relationship between them and and the sets of traces and accepting traces of the IOSTS SP_{vis} from which TG was obtained by the selection algorithm (*see* page 185).

Before going forward it is important to notice that the test graph TG is initialized (by construction) and deterministic (*see* Lemma 7.1, page 195) IOSTS. Thus, due to Theorem 4.1 (*see* page 94), TG cannot move to states corresponding to different locations after execution of a given trace, *e.g.* TG cannot be in a pass and inconclusive state after the same trace σ. This statement will be used implicitly in proofs of the lemmas presented below.

Lemma 7.2 (Traces of TG Leading to Pass) Let TG be the test graph generated from SP_{vis} by Algorithm 7.2 (*see* page 185), where SP_{vis} is the IOSTS obtained from the synchronous product between a specification $Spec$ and a test purpose TP by closure and determinization. Then:

$$Traces_{pass}(TG) \subseteq ATraces(Spec \times TP) \tag{7.17}$$

□

Proof Consider a *pass* trace σ of TG, and prove that this trace is an *accepting* trace of $Spec \times TP$.

At the beginning of the proof, notice that we have the two following properties:

(1) σ is a trace of SP_{vis}, *i.e.* $\sigma \in Traces(SP_{vis})$

Indeed, as σ is a pass trace of TG, then due to Definition 7.3 (*see* page 196) it is a trace of TG, *i.e.* $\sigma \in Traces(TG)$. Thus, using Corollary 7.1 (*see* page 195) we obtain that σ is also a trace of the IOSTS SP_{vis}.

(1) The set of initial states of TG is a subset of the set of initial states of SP_{vis}, *i.e.* $S^0_{TG} \subseteq S^0_{SP_{vis}}$.

Indeed, if $s^0 = \langle l^0, \vartheta^0 \rangle$ is an initial state of TG, then the valuation ϑ^0 satisfies the initial condition of TG that is $(\Theta_{vis} \wedge \mathsf{CoReach}(SP_{vis}))$, where Θ_{vis} is the initial condition of the IOSTS SP_{vis} and $\mathsf{CoReach}(SP_{vis})$ is the predicate characterizing an over-approximation of the set of all coreachable states of SP_{vis} defined as Formula (7.8) (*see* page 184). Thus, ϑ^0 satisfies the initial condition Θ_{vis} of SP_{vis}. Moreover, due to Algorithm 7.2 (*see* page 185) l^0 is the initial location of TG as well as SP_{vis}. Therefore, the state s^0 is also an initial state of SP_{vis}.

From the items (1) and (2) we deduce that the trace σ can be executed in both IOSTS TG and SP_{vis} from a same initial state s^0. It is important to notice that by choosing the initial state s^0, we fix values of all variables and symbolic constants of TG and SP_{vis}. Moreover, we know that TG and SP_{vis} are deterministic. Therefore, by analogy with the proof of the second item of Theorem 4.1 (*see* page 94), we can show that by executing the trace σ, TG (*resp.* SP_{vis}) moves from the state s^0 to *exactly one state* s (*resp.* s').

Next, as σ is a *pass* trace of TG, then due to Definition 7.3 (*see* page 196) it leads to the state s corresponding to a *Pass* location of TG. Thus, as all *Pass* locations of TG were obtained from the *accepting* locations of the IOSTS SP_{vis} (*see* Algorithm 7.2, page 185), then the trace σ of SP_{vis} leads to the state s' corresponding to an *Accept* location of SP_{vis}. Therefore, σ is an *accepting* trace of SP_{vis}.

Q.E.D.

It is important to emphasize that the opposite inclusion, *i.e.* $ATraces(Spec \times TP) \subseteq Traces_{pass}(TG)$, does not hold. Before showing this fact we notice that the set of accepting traces of $Spec \times TP$ is the same as the set of accepting traces of $ATraces(Spec \times TP) = ATraces(SP_{vis})$, where $SP_{vis} = \mathsf{det}(\mathsf{closure}(Spec \times TP))$

(*see* Theorems A.4 and A.9, pages 303 and 271). Therefore, it is enough to show that $ATraces(SP_{vis}) \not\subseteq Traces_{pass}(TG)$. This demonstration is done with the following example.

Example 7.6 Consider the IOSTS SP_{vis} shown on Figure 7.11(c) (*see* page 199) that was obtained form:

- the specification *Spec* of a coffee machine depicted on Figure 7.11(a) (*see* page 199), and
- the test purpose *TP* of *Spec* that is depicted on Figure 7.11(b) (*see* page 199),

by the product and closure operations, and the procedure of determinization. It is not hard to check that SP_{vis} has two accepting traces:

$$\begin{aligned} \sigma_1 : &\quad \langle Coin, \langle\rangle\rangle\langle Coffee, \langle\rangle\rangle, \text{ and} \\ \sigma_2 : &\quad \langle Coin, \langle\rangle\rangle\langle Coffee, \langle\rangle\rangle\langle Delivery, \langle\rangle\rangle \end{aligned}$$

Then, by apply the selection algorithm to SP_{vis}, we obtain the test graph *TG* shown on Figure 7.11(d) (*see* page 199). Indeed, the selection algorithm removed:

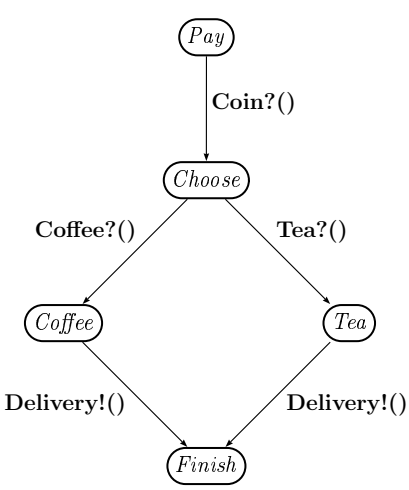
(1) the symbolic transition t_1 labeled with the *Tea* action, as (a) *Tea* is the *input* action of SP_{vis}, and (b) it does not lead to any state from which it is possible to go to some accepting state,

(2) the symbolic transition t_2 labeled with the ***Delivery*** action and outgoing from the $\langle Tea, Begin\rangle$ location, as it became in-executable after eliminating the symbolic transition t_2, and

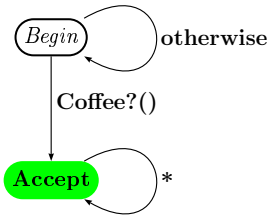
(3) the symbolic transition t_3 labeled with the ***Delivery*** action and outgoing from the $\langle Coffee, Accept\rangle$ location, as its origin is the accepting location of SP_{vis}.

From Figure 7.11(d) (*see* page 199) the reader can see that the resulting test graph *TG* has only one pass trace $\sigma_1 : \langle Coin, \langle\rangle\rangle\langle Coffee, \langle\rangle\rangle$, which is an accepting trace of SP_{vis}. Therefore, $Traces_{pass}(TG) \subseteq \underbrace{ATraces(SP_{vis})}_{=ATraces(Spec \times TP)}$, but $\underbrace{ATraces(SP_{vis})}_{=ATraces(Spec \times TP)} \not\subseteq Traces_{pass}(TG)$. □

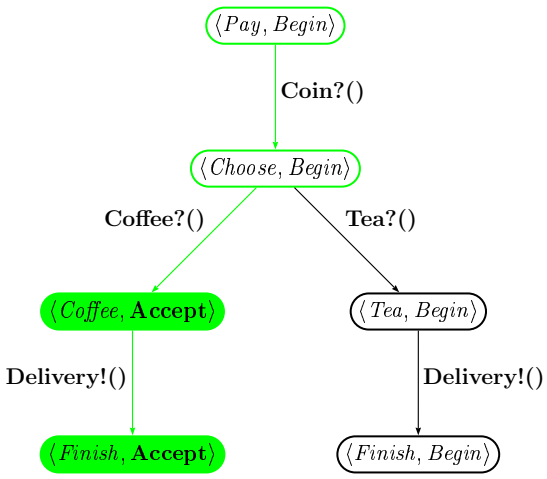
Before characterizing the set of inconclusive traces of the given test graph *TG*, we make an observation about the form of these traces.



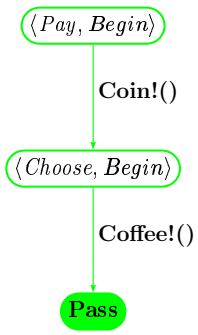
(a) *Spec*



(b) *TP*



(c) $Spec \times TP = SP_{vis}$



(d) *TG*

Figure 7.11: A counterexample showing that $ATraces(Spec \times TP) \not\subseteq Traces_{pass}(TG)$.

Observation 7.2 (The Form of an Inconclusive Trace of TG) Let TG be the test graph with set of valued input actions $\Lambda^?$. Then, any inconclusive trace σ of TG has the form: $\sigma' \cdot \alpha$, where $\sigma' \in \mathit{Traces}(TG)$ and $\alpha \in \Lambda^?$. □

This observation immediately follows from the construction of the test graph TG (*see* Algorithm 7.2, page 185). Indeed, as each *Inconclusive* location of TG is different from the initial location l^0_{TG} of TG, then the set of inconclusive states of TG is different from the set of initial states of TG. Therefore, σ leading to some inconclusive state cannot be an empty trace. Moreover, as we can move to an *Inconclusive* location only by taking a symbolic transition labeled with an *input* action (*see* the item (5.b) of Algorithm 7.2, page 185), then the last valued action α of σ is a valued *input* action of TG.

Lemma 7.3 (Traces of TG Leading to Inconclusive) Let TG be a test graph generated from a specification *Spec* and a test purpose TP of *Spec* as it was explained in Sections 7.1–7.4 (*see* pages 169–178). Then, a trace of TG leading to some inconclusive state is a trace of *Spec* ending with a valued output action. However, it is not a prefix (*see* Definition 6.11, page 154) of any accepting trace of $(\mathit{Spec} \times TP)$. Formally:

$$\mathit{Traces}_{\mathit{inconc}}(TG) \subseteq \qquad (7.18)$$
$$(\mathit{Traces}(\mathit{Spec}) \cdot \Lambda^!_{\mathit{Spec}} \ \cap \ \mathit{Traces}(\mathit{Spec})) \setminus \mathit{Pref}(\mathit{ATraces}(\mathit{Spec} \times TP))$$

where $\mathit{Traces}(\mathit{Spec}) \cdot \Lambda^!_{\mathit{Spec}} \triangleq \{\sigma \cdot \alpha \mid \sigma \in \mathit{Traces}(\mathit{Spec}) \wedge \alpha \in \Lambda^!_{\mathit{Spec}}\}$ □

Proof Consider a trace σ belonging to $\mathit{Traces}_{\mathit{inconc}}(TG)$. Due to Observation 7.2 (*see* page 200) σ has the form $\sigma' \cdot \alpha$, where $\sigma' \in \mathit{Traces}(TG)$ and $\alpha \in \Lambda^?_{TG}$.

(1) We prove that $\sigma = \sigma' \cdot \alpha \in (\mathit{Traces}(\mathit{Spec}) \cdot \Lambda^!_{\mathit{Spec}} \ \cap \ \mathit{Traces}(\mathit{Spec}))$.

   (a) First, we show that α is a valued *output* action of *Spec*.

   Indeed, we remind that during the selection of the test graph from the IOSTS $SP_{vis} = \mathsf{det}(\mathsf{closure}(\mathit{Spec} \times TP))$ we invert the directions (input/output) of all actions of SP_{vis}. Thus, as α is a valued *input* action of TG, then it is a valued *output* action of SP_{vis}. Therefore, α is a valued *output* action of *Spec*, *i.e.* $\alpha \in \Lambda^!_{\mathit{Spec}}$, due to the construction of SP_{vis} (*see* Sections 7.1–A.2, pages 169–274).

   (b) Finally, we show that $\sigma = \sigma' \cdot \alpha \in (\mathit{Traces}(\mathit{Spec}) \cdot \Lambda^!_{\mathit{Spec}} \cap \mathit{Traces}(\mathit{Spec}))$.
   Due to Definition 7.3 (*see* page 196) $\sigma = \sigma' \cdot \alpha$ is a trace of TG. Then,

we remind that:

$$
\begin{aligned}
& \mathit{Traces}(\underbrace{\mathsf{select}(\mathsf{det}(\mathsf{closure}(\mathit{Spec} \times \mathit{TP})))}_{TG}) \\
& \qquad \subseteq \text{ [\textit{see} Corollary 7.1, page 195]} \\
& \mathit{Traces}(\mathsf{det}(\mathsf{closure}(\mathit{Spec} \times \mathit{TP}))) \\
& \qquad = \text{ [\textit{see} Theorem A.9, page 303]} \\
& \mathit{Traces}(\mathsf{closure}(\mathit{Spec} \times \mathit{TP})) \\
& \qquad = \text{ [\textit{see} Theorem A.4, page 271]} \\
& \mathit{Traces}(\mathit{Spec} \times \mathit{TP}) \\
& \qquad = \text{ [\textit{see} Theorem 5.5, page 134]} \\
& \mathit{Traces}(\mathit{Spec})
\end{aligned}
\tag{7.19}
$$

Thus, $\sigma' \cdot \alpha$ belongs to $\mathit{Traces}(\mathit{Spec})$. Moreover, using prefix closure we get that $\sigma' \in \mathit{Traces}(\mathit{Spec})$. Therefore, as we know that $\alpha \in \Lambda^{!}_{\mathit{Spec}}$ (*see* the item (a) above), then $\sigma \in (\mathit{Traces}(\mathit{Spec}) \cdot \Lambda^{!}_{\mathit{Spec}} \cap \mathit{Traces}(\mathit{Spec}))$.

(2) We prove that the inconclusive trace σ of TG does not belong to the set of prefixes of accepting traces of $(\mathit{Spec} \times \mathit{TP})$.

This statement can be shown by contradiction. Assume that σ belongs to $\mathit{Pref}(\mathit{ATraces}(\mathit{Spec} \times \mathit{TP}))$, where $\mathit{Pref}(\mathit{ATraces}(\mathit{Spec} \times \mathit{TP}))$ is the set of all possible prefixes (strict and non-strict) of accepting traces of the IOSTS $\mathit{Spec} \times \mathit{TP}$ (*see* Definition 6.11, page 154). Then, there exists a sequence of valued input/output actions $\sigma'' \in (\Lambda^{?} \cup \Lambda^{!})^{*}$ such that $\sigma \cdot \sigma'' \in \mathit{ATraces}(\mathit{Spec} \times \mathit{TP})$.

(a) Assume that σ'' is the empty sequence of valued input/output actions. Then $\sigma \in \mathit{ATraces}(\mathit{Spec} \times \mathit{TP})$. Due to Definition 6.10 (*see* page 151) this means that σ leads to an accepting state of $\mathit{Spec} \times \mathit{TP}$. Thus, the trace σ must leads to a pass state in the test graph TG (*see* Algorithm 7.2, page 185). Therefore, as we know that $(\mathsf{PASS} \cap \mathsf{INCONCLUSIVE}) = \emptyset$, then the statement above contradicts with the assumption that σ is an inconclusive trace of TG, *i.e.* trace leading to some inconclusive state of TG.

(b) Assume that $\sigma'' \in (\Lambda^{?} \cup \Lambda^{!})^{+}$, then $\sigma \cdot \sigma'' \in \mathit{ATraces}(\mathit{Spec} \times \mathit{TP})$.

This means that the IOSTS $\mathit{Spec} \times \mathit{TP}$ can move from the state $s = \langle l, \vartheta \rangle$ obtained after executing σ, to another state $s' = \langle l', \vartheta' \rangle$ by the first valued action of σ'', say $\alpha' = \langle a, \omega \rangle$. Thus, we obtain that $\mathit{Spec} \times \mathit{TP}$ has at least one symbolic transition of the form $l \overset{a}{\hookrightarrow} l'$. $(\star)$

Next, due to our assumption that σ is an inconclusive trace of the test graph TG constructed from $Spec \times TP$, we know that the state s is an inconclusive state of TG, *i.e.* the location l is named *Inconclusive*. Moreover, we know that there are no symbolic transitions outgoing form the *Inconclusive* location. But due to the observation ($\star$) there is at least one symbolic transition outgoing from $l = Inconclusive$. Therefore, a contradiction is obtained.

From the item (a) and (b) we get that $\sigma \notin Pref(ATraces(Spec \times TP))$.

The items (1) and (2) prove the lemma. **Q.E.D.**

7.5 Making a Test Graph Input-Complete

In this section we consider a test graph TG produced from a specification $Spec \in \mathsf{SPECS}$ and a test purpose TP of $Spec$ by the operations and algorithms described in the previous sections of Chapter 7 (*see* Sections 7.1–7.4 page 169–178). The aim of this section is to transform the given test graph TG into a test case $TC \in \mathsf{TESTS}$ whose definition is given on page 143. Due to this definition the test case TC must always react on any input coming from an implementation under test $iut \in \mathsf{IMPS}$. This means that TC must not block any inputs from iut, but it must negatively respond on incorrect inputs. Therefore, we have to require that each location of the test graph TG, except the *Pass* and *Inconclusive* locations, (1) accepts any input from iut, and (2) redirects incorrect inputs to the new *Fail* location, *i.e.* make TG input-complete in the sense of Definition 4.21 (*see* page 95). Moreover, the test case TC must be deterministic, as it has to produce the same verdict all time while executing a same sequence of actions on a same implementation under test.

Plan of the Section. The rest of this section is organized as follows: first, we present an algorithm which makes a given test graph TG input-complete except in *Pass*, *Inconclusive* and *Fail* locations, *i.e.* which generates a test case TC. Then, we illustrate this algorithm with an example. Next, we prove that the result produced by the algorithm is input-complete in the sense of Definition 4.21 (*see* page 95), and deterministic (*see* Definition 4.20, page 93). Finally, we show some properties characterizing the sets of traces of the generated test case TC.

7.5.1 Algorithm Making TG Input-Complete

Algorithm 7.3 (Making a Test Graph Input-Complete) Consider a test graph TG with set of variables V_{TG}, set of actions $\Sigma_{TG} = \Sigma^{?}_{TG} \cup \Sigma^{!}_{TG}$, set of

locations L_{TG}, and set of symbolic transitions T_{TG}. Then, TG is completed with its input actions belonging to $\Sigma^{?}_{TG}$ as it is described below.

First, for a location $l \in L_{TP}$ and an action $a \in (\Sigma_{TP} = \Sigma_{Spec})$, we denote by:

$$T_{l,a} = \{t_1 = \langle l, a, \pi, G_1, A_1, l_1\rangle, \ldots, t_n = \langle l, a, \pi, G_n, A_n, l_n\rangle\} \subseteq T_{TG}$$

the set of all symbolic transitions of TG outgoing from l and labeled with action a.

Then, for each location $l \in (L_{TG} \setminus \{Pass, Inconclusive\})$ and each input action $a \in \Sigma^{?}_{TG}$:

(1) If (1) $T_{l,a}$ is not empty and (2) the disjunction of the gards of the symbolic transitions belonging to $T_{l,a}$ does not equal to *true*, *i.e.* $(G_1 \vee \ldots \vee G_{|T_{l,a}|}) \neq$ *true*, then we create a new symbolic transition:

$$t' = \langle l, a, \pi, \neg(G_1 \vee \ldots \vee G_n), A', Fail\rangle$$

leading to the *Fail* location. Here and in the next item, $A' = \bigcup_{v \in V_{TG}} [v := v]$ is the set of identity assignments.

(2) If $T_{l,a}$ is empty, *i.e.* in the case when there are no symbolic transitions outgoing from l and labeled with a, we create the following symbolic transition:

$$t' = \langle l, a, \pi, true, A', Fail\rangle$$

which also leads to the *Fail* location.

□

Then, we prove that Algorithm 7.3 is correct, namely, it produces an input-complete and deterministic IOSTS.

Theorem 7.2 (Correctness of Algorithm 7.3, page 202) An IOSTS $TG' = \langle D_{TG'}, \Theta_{TG'}, L_{TG'}, l^0_{TG'}, \Sigma_{TG'}, T_{TG'}\rangle$ with set of states $S_{TG'}$ and set of valued input actions $\Lambda^{?}_{TG'}$, which is produced form the test graph TG by Algorithm 7.3 (*see* page 202), is:

(1) input-complete (*see* Definition 4.21, page 95), except, in the locations *Pass*, *Fail* and *Inconclusive*, and

(2) deterministic (*see* Definition 4.20, page 93).

□

Proof

(1) We show that TG' is input-complete except in the locations *Pass*, *Fail* and *Inconclusive*.

Indeed, due to Algorithm 7.3 we obtain that for each location $l \in (L_{TG'} \setminus \{Pass, Fail, Inconclusive\})$ and each input action $a \in \Sigma^{?}_{TG'}$, the disjunction of the guards $G_1, \ldots, G_n$ of all symbolic transitions $t_1, \ldots, t_n \in T_{TG'}$ outgoing from l and labeled with a, is equal to *true*. Thus, for each state $s = \langle l, \vartheta \rangle \in S_{TG'}$ of TG', and each valued input action $\alpha = \langle a, \omega \rangle \in \Lambda^{?}_{TG'}$, there always exists a symbolic transition t_i $(i = 1..n)$ outgoing from l and labeled with a, whose guard is satisfied by the pair of valuations $\langle \vartheta, \omega \rangle$. Due to Definition 4.21 (*see* page 95) this means that TG' is input-complete except in the *Pass*, *Fail* and *Inconclusive* locations.

(2) We prove that TG' is deterministic in the sense of Definition 4.20 (*see* page 93).

First, we show that TG' does not contain any internal action.

Indeed, notice that the test graph TG does not have any internal action by construction (thanks to the closure operation defined on page 260). Thus, as TG' is obtained from TG by Algorithm 7.3 (*see* page 202), then it also does not contain internal action. Therefore, $\Lambda^{\tau} = \emptyset$.

Second, we show that from each state of TG' at most one symbolic transition can be fired, *i.e.* $\forall\ s \in S_{TG'},\ \alpha \in (\Lambda^{?}_{TG'} \cup \Lambda^{!}_{TG'})\ .\ [card(\{s' \in S_{TG'} \mid s \xrightarrow{\alpha} s'\}) \leq 1]$.

To prove this statement we need to consider the two following cases.

(a) Let s be a state of TG', and α be a valued *output* action of TG'. Then,

- as TG' is obtained from the test case TG by Algorithm 7.3 (*see* page 202) which *does not augment* TG with symbolic transitions labeled with *output* actions, and
- as TG is deterministic by construction (thanks to the procedure of the global determinization given on page 304),

thus, we immediately obtain that $card(\{s' \in S_{TG'} \mid s \xrightarrow{\alpha} s'\}) \leq 1$.

(b) Let $s = \langle l, \vartheta \rangle$ be a state of TG', and $\alpha = \langle a, \omega \rangle$ be a valued *input* action of TG'.

First, notice that if the location l is equal to either *Pass*, *Inconclusive* or *Fail*, then $card(\{s' \in S_{TG'} \mid s \xrightarrow{\alpha} s'\}) = 0$. This statement follows directly from the fact that the IOSTS TG'

does not contain any symbolic transition outgoing from *Pass*, *Inconclusive* or *Fail* locations (*see* the construction of TG', or more precisely, Algorithms 7.2 and 7.3 presented on pages 185 and 202).

Second, suppose that the IOSTS TG' obtained from the test graph TG by Algorithm 7.3 (*see* page 202), has $n \geq 0$ symbolic transitions $t_1, \ldots, t_n$ (a) outgoing from the location l which is different from *Pass*, *Inconclusive* and *Fail*, (b) labeled with the input action a, and (c) guarded with Boolean expressions $G_1, \ldots, G_n$. Then:

- If all symbolic transitions $t_1, \ldots, t_n$ are also symbolic transitions of TG, then due to Algorithm 7.3 (*see* page 202) we know that $(G_1 \vee \ldots \vee G_n) = true$. Next, as TG is deterministic, we get that $card(\{s' \in S_{TG'} \mid s \xrightarrow{\alpha} s'\}) = 1$.
- Otherwise, there exists *exactly one* symbolic transition, say t_i $(i = 1..n)$, which was added to TG' by Algorithm 7.3 (*see* page 202). The rest of the symbolic transitions, *i.e.* $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$ are symbolic transitions of TG such that:
  (i) the disjunction of their guards is not equal to true (*see* Algorithm 7.3, page 202), *i.e.* $(G_1 \vee \ldots \vee G_{i-1} \vee G_{i+1} \vee \ldots \vee G_n) \neq true$, and
  (ii) the guards of these symbolic transitions are mutually exclusive due to the fact that TG is deterministic.

  Next, as the symbolic transition t_i was added to TG' by Algorithm 7.3 (*see* page 202), then its guard G_i is equal to $\neg(G_1 \vee \ldots \vee G_{i-1} \vee G_{i+1} \vee \ldots \vee G_n)$. Thus:
  - the disjunction between $(G_1 \vee \ldots \vee G_{i-1} \vee G_{i+1} \vee \ldots \vee G_n)$ and the guard G_i of the symbolic transition t_i is trivially equal to $true$,
  - moreover, the guard G_i of t_i is disjoint with the guards of the symbolic transitions $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$, *i.e.* $\forall\, j \in [1, n]\, .\, [(j \neq i) \implies (G_i \wedge G_j) = false]$. Thus, using the item (ii) we obtain that the guards of all symbolic transitions $t_1, \ldots, t_i, \ldots, t_n$ are mutually exclusive.

  From the items above we deduce that $card(\{s' \in S_{TG'} \mid s \xrightarrow{\alpha} s'\}) = 1$.

Q.E.D.

At the end of the subsection we illustrate the algorithm producing an input-complete test graph with an example.

Example 7.7 Consider the test graph TG_2 shown on Figure 7.10(b) (*see* page 191) obtained after the selection algorithm (*see* page 185). The alphabet of input actions of the given TG_2 consists of the two following elements: *Return* and *Deliver*. Notice that TG_2 is not input-complete in the sense of Definition 4.21 (*see* page 95) as, for instance, in the location $\langle Begin, Begin\rangle$ it does not accept neither *Return* nor *Deliver* input actions. Thus, we complete TG_2 with the input actions as follows: we allow the test graph TG_2 to accept incorrect inputs in all locations except the location *Pass*, but we redirect them to the location *Fail*. For example,

(1) consider the location $\langle Delivery, Begin\rangle$ (*see* Figure 7.10(b), page 191). As TG_2 can receive the input action *Deliver* only when the guard

$$\begin{aligned} G: \quad & (cPrice \geq 1) \wedge \\ & (vPaid \geq 1) \wedge \\ & (vPaid = cPrice) \wedge \\ & (pBeverage = vBeverage) \wedge \\ & (vBeverage = \text{COFFEE}) \end{aligned}$$

is satisfied, then we create another symbolic transition accepting the input action *Deliver* in the case when the guard G is not satisfied, however this new symbolic transition must go to the *Fail* location (*see* Figure 7.12, page 207).

(2) consider the location $\langle Delivery, Begin\rangle$ of TG_2 (*see* Figure 7.10(b), page 191). This location must be completed with the input action *Return* by adding the new transition outgoing from $\langle Delivery, Begin\rangle$ leading to *Fail*, and labeled with the input action *Return* (*see* Figure 7.12, page 207).

The input-complete test case TC obtained from TG_2 is shown in Figure 7.12 (*see* page 207). Notice that, when this test case receives an incorrect input, *e.g.* *Return* in the location $\langle Delivery, Begin\rangle$, it does not block it, it produces the *Fail* verdict.

□

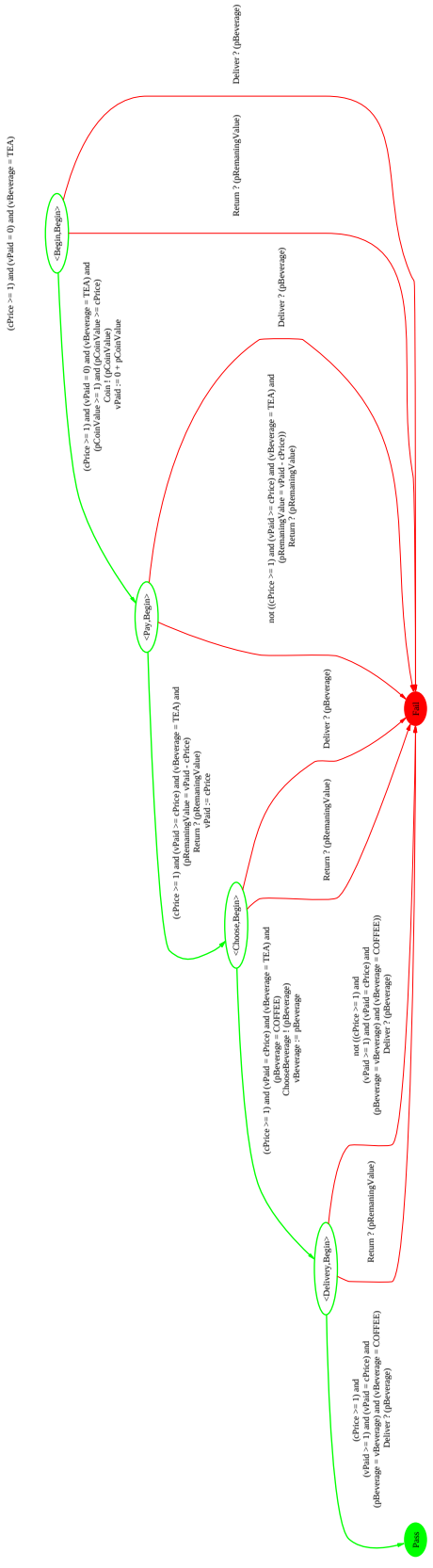


Figure 7.12: Input-complete test case TC.

7.5.2 Traces of a Test Case

In this subsection we consider a test case derived by the symbolic test generation method described in Chapter 7 (*see* page 167). The main purpose of this subsection is to characterize the set of traces of the given test case.

First, we notice that any test case $TC \in \mathsf{TESTS}$ has three kind of verdicts, namely, *Pass*, *Inconclusive* and *Fail*. Therefore, its set of traces contains traces leading to pass, inconclusive and fail states. We remind that the notions of traces leading to *Pass* and *Inconclusive* were already introduced for a test graph TG (*see* Definition 7.3, page 196), and they are the same for the test case TC. Therefore, we have to introduce only the notion of traces leading to fail states of TC.

Definition 7.4 (Traces of TC Leading to Fail) Let TC be a test case with set of states S_{TC}, set of initial states $S^0_{TC} \subseteq S_{TC}$, and set of fail states $\mathsf{FAIL} \subseteq S_{TC}$. Then, the set of traces leading to states belonging to FAIL is defined as follows:

$$Traces_{fail}(TC) \triangleq \{\sigma \in Traces(TC) \mid \exists\, s^0 \in S^0_{TC},\ s_{fail} \in \mathsf{FAIL}\ .\ [s^0 \stackrel{\sigma}{\Rightarrow} s_{fail}]\} \quad (7.20)$$

In the rest of the thesis any trace belonging to $Traces_{fail}(TC)$ is called a *fail trace*. □

Then, we consider the test case TC derived from some test graph TG by Algorithm 7.3 (*see* page 202), and make several observations/lemmas characterizing the sets of fail, pass and inconclusive traces of TC.

Lemma 7.4 (Pass Traces of TC) The set of pass traces of the test case TC is equal to the set of pass traces of the test graph TG, *i.e.*

$$Traces_{pass}(TC) = Traces_{pass}(TG)$$

□

Proof In order to prove the lemma we have to show the two following inclusions.

($\supseteq$) Consider a pass trace $\sigma = \alpha_1 \ldots \alpha_n$ of the test graph TG and prove that it is a pass trace of the test case TC.

By Definition 7.3 (*see* page 196) we know that the trace σ leads from an initial state $s^0 = \langle l^0, \vartheta^0 \rangle \in S^0_{TG}$ to some pass state $s_{pass} = \langle Pass, \vartheta_{pass} \rangle \in \mathsf{PASS}$ of TG. Moreover, we know that the trace σ corresponds to a sequence of consecutive symbolic transitions: $\rho : l^0 \stackrel{a_1}{\hookrightarrow} l_1 \ldots l_{n-1} \stackrel{a_n}{\hookrightarrow} Pass$ where $l_1, \ldots, l_{n-1}$ are locations of TG, and $a_1, \ldots, a_2$ are input/output actions of TG corresponding to $\alpha_1, \ldots, \alpha_n$ which constitute the trace σ.

Then, due to the last step of the test case construction (*see* Algorithm 7.3, page 202), we know that TG and TC have the same structure except of new symbolic transitions leading to the *Fail* location which were added to TC. Thus, TC contains the sequence of symbolic transitions ρ.

Finally, as TG and TC have the same initial condition (*see* Algorithm 7.3, page 202), and TC is deterministic (*see* Theorem 7.2, page 203), then the trace σ executing on TC from the same initial state s^0 as TG leads to the same pass state s_{pass} as TG.

Therefore, σ is a pass trace of the test case TC (*see* Definition 7.3, page 196 which is the same for TC).

(⊆) The proof of the second inclusion is similar to one of the first inclusion (⊇).

Q.E.D.

Lemma 7.5 (Inconclusive Traces of TC) The set of inconclusive traces of the test case TC is equal to the set of inconclusive traces of the test graph TG, *i.e.*

$$Traces_{inconc}(TC) = Traces_{inconc}(TG)$$

□

Proof This lemma can be proved similarly as Lemma 7.4 on page 208. **Q.E.D.**

Before characterizing the set of traces and the set of fail traces of the given test case TC, we make an observation about the form of a fail trace.

Observation 7.3 (The Form of a Fail Trace of TC) Let TC be a test case with set of valued input actions $\Lambda^?$. Then, any fail trace σ of TC has the form: $\sigma' \cdot \alpha$, where $\sigma' \in Traces(TC)$ and $\alpha \in \Lambda^?$. □

This observation follows directly from the construction of the test graph TC (*see* Algorithm 7.3, page 202). Indeed, as an *Fail* location of TC is different from the initial location l^0_{TC} of TC, then the set of fail states of TC is different from the set of initial states of TC. Therefore, σ leading to some fail state cannot be an empty trace. Moreover, as TC can move to an *Fail* location only by taking some symbolic transition labeled with an *input* action (*see* Algorithm 7.3, page 202), then the last valued action α of σ is a valued *input* action of TC.

Lemma 7.6 (Traces of TC) The set of traces of TC is the union between the set of traces of TG and the set of fail traces of TC, *i.e.*

$$Traces(TC) = Traces(TG) \cup Traces_{fail}(TC)$$

□

Proof The proof of this lemma follows from the two statements shown below.

(1) All traces of the test graph TG are included into the set of traces of the test case TC, *i.e.* $Traces(TG) \subseteq Traces(TC)$.

Indeed, as the test case TC is derived from TG by adding new symbolic transitions leading to the *Fail* location (*see* Algorithm 7.3, page 202), then it keeps the structure of TG. Therefore, any trace of TG will be a trace of TC.

(2) The trace σ is a trace of the test case TC and not a trace of the test graph TG *if and only if* it is a fail trace of TC, *i.e.* $(Traces(TC) \setminus Traces(TG)) = Traces_{fail}(TC)$.

(⊆) We show that $(Traces(TC) \setminus Traces(TG)) \subseteq Traces_{fail}(TC)$.

This inclusion can be proved by contradiction.

Consider a trace $\sigma = \alpha_1 \ldots \alpha_n$ belonging to $(Traces(TC) \setminus Traces(TG))$. Assume that this trace σ does not belong to $Traces_{fail}(TC)$. Thus, σ leads to a state corresponding to a location l different from *Fail*.

Next, we know that TC has a sequence of consecutive symbolic transitions: $\rho : l^0 \overset{a_1}{\hookrightarrow} l_1 \ldots l_{n-1} \overset{a_n}{\hookrightarrow} l$ corresponding to σ, where $l^0, l_1, \ldots, l_{n-1}$ are locations of TC and $a_1, \ldots, a_n$ are input/output actions of TC that correspond to $\alpha_1, \ldots, \alpha_n$. Moreover, $l^0 \neq Fail$ (*see* Algorithm 7.3, page 202), $l \neq Fail$ (due to the assumption that σ is not a fail trace of TC), and $l_1 \neq Fail, \ldots, l_{n-1} \neq Fail$ (due to the fact that there are no symbolic transitions outgoing from the *Fail* location, *see* Algorithm 7.3 on page 202).

Finally, as the sequence ρ of TC does not contain any symbolic transition leading to the *Fail* location, then due to Algorithm 7.3 (*see* page 202) ρ is also a sequence of consecutive symbolic transitions of TG. Therefore, the trace σ is the trace of TG. This contradicts with the assumption that $\sigma \in (Traces(TC) \setminus Traces(TG))$.

(⊇) We show that $(Traces(TC) \setminus Traces(TG)) \supseteq Traces_{fail}(TC)$.

Consider a trace $\sigma = \sigma' \cdot \alpha$ belonging to $Traces_{fail}(TC)$. Then, by definition of a fail trace (*see* page 208), we obtain that:

(a) σ is a trace of TC, and

(b) $(TC \text{ after } \sigma) \subseteq \mathsf{FAIL}$. Therefore, the last action of the trace σ, *i.e.* α corresponds to a symbolic transition of the form $l \overset{a}{\hookrightarrow} \mathit{Fail}$, where l is a location of TC and a is an input action of TC. Then, we know that the symbolic transitions of this form were added to TG by Algorithm 7.3 (*see* page 202), thus σ cannot be a trace of TG.

The items (a) and (b) imply ($\supseteq$).

Therefore, we have shown that $(\mathit{Traces}(TC) \setminus \mathit{Traces}(TG)) = \mathit{Traces}_{\mathit{fail}}(TC)$.

The proof of the lemma follows directly from the items (1) and (2). **Q.E.D.**

At the end of this subsection we study the relationship between the set of fail traces of the test case TC, and the set of traces of the specification *Spec* from which TC was generated.

Lemma 7.7 (Traces of TC Leading to Fail) Let TC be a test case generated from a specification $\mathit{Spec} \in \mathsf{SPECS}$ and a test purpose TP of *Spec* as it was explained in Sections 7.1–7.5 (*see* pages 169–202). Then, a trace of TC leading to some fail state is a trace of *Spec* concatenated with a valued output action, but it is not a trace of *Spec*.

$$\mathit{Traces}_{\mathit{fail}}(TC) \subseteq (\mathit{Traces}(\mathit{Spec}) \cdot \Lambda^{!}_{\mathit{Spec}}) \setminus \mathit{Traces}(\mathit{Spec}) \tag{7.21}$$

where $\mathit{Traces}(\mathit{Spec}) \cdot \Lambda^{!}_{\mathit{Spec}} \triangleq \{\sigma \cdot \alpha \mid \sigma \in \mathit{Traces}(\mathit{Spec}) \wedge \alpha \in \Lambda^{!}_{\mathit{Spec}}\}$ □

Proof Consider a trace σ belonging to $\mathit{Traces}_{\mathit{fail}}(TC)$. Due to Observation 7.3 (*see* page 209) σ has the form $\sigma' \cdot \alpha$, where $\sigma' \in \mathit{Traces}(TC)$ and $\alpha \in \Lambda^{?}_{TC}$.

(1) We prove that $\sigma = \sigma' \cdot \alpha$ belongs to $(\mathit{Traces}(\mathit{Spec}) \cdot \Lambda^{!}_{\mathit{Spec}})$.

(a) First, we show that the last valued action of σ, *i.e.* α is a valued *output* action of *Spec*.

Indeed, as:

– α is a valued *input* action of the test case TC derived from the specification *Spec* by the symbolic test generation method (*see* Sections 7.1–7.5, pages 169–202), and

– during selection (*see* Algorithm 7.2, page 185), the directions (input/output) of all actions were inverted,

then α is a valued *output* action of *Spec*, *i.e.* $\alpha \in \Lambda^{!}_{Spec}$.

(b) Second, we show that the strict maximal prefix of σ, namely σ', belongs to $Traces(Spec)$.

As $\sigma = \sigma' \cdot \alpha$ is a fail trace of TC, then due to the definition of a fail trace (*see* Definition 7.4, page 208) $\sigma = \sigma' \cdot \alpha$ is a trace of TC. Then, using prefix closure we obtain that σ' is also a trace of TC.

Next, remember that $Traces(TC) = Traces(TG) \cup Traces_{fail}(TC)$ (*see* Lemma 7.6, page 210), where $Traces(TG) \subseteq Traces(Spec)$ due to the following equalities and inclusion:

$$
\begin{aligned}
& Traces(\underbrace{\mathsf{select}(\mathsf{det}(\mathsf{closure}(Spec \times TP)))}_{TG}) \\
& \qquad \subseteq \text{ [\textit{see} Corollary 7.1, page 195]} \\
& Traces(\underbrace{\mathsf{det}(\mathsf{closure}(Spec \times TP))}_{SP_{vis}}) \\
& \qquad = \text{ [\textit{see} Theorem A.9, page 303]} \\
& Traces(\mathsf{closure}(Spec \times TP)) \\
& \qquad = \text{ [\textit{see} Theorem A.4, page 271]} \\
& Traces(Spec \times TP) \\
& \qquad = \text{ [\textit{see} Theorem 5.5, page 134]} \\
& Traces(Spec)
\end{aligned}
\tag{7.22}
$$

Thus, if we prove that σ' does not belong to the set of fail traces of TC, then we get that σ' is a trace of the test graph TG, and therefore, it is a trace of *Spec*.

We prove that $\sigma' \notin Traces_{fail}(TC)$ by contradiction. Assume that σ' belongs to $Traces_{fail}(TG)$. This means that it leads to some state corresponding to the *Fail* location of TC. Next, as we know that TC does not have any symbolic transition outgoing from the *Fail* location (*see* Algorithm 7.3, page 202), then TC does not have any valued actions $\alpha \in \Lambda_{TC}$ such that $\sigma \cdot \alpha$ is a trace of TC. This contradicts the fact that σ' is the strict maximal prefix of trace σ of TC.

Therefore, $\sigma' \notin Traces_{fail}(TC)$, hence, $\sigma' \in Traces(TG)$ (again, this is because $\sigma' \in Traces(TC)$ and $Traces(TC) = Traces(TG) \cup Traces_{fail}(TC)$). Finally, due to Formula 7.22, $\sigma' \in Traces(Spec)$.

The items (a) and (b) imply that $\sigma \in Traces(Spec) \cdot \Lambda^{!}_{Spec}$.

(2) We prove that a fail trace $\sigma = \sigma' \cdot \alpha$ of TC does not belong to $Traces(Spec)$.

This statement can be proved by contradiction.

Assume that $\sigma = \sigma' \cdot \alpha$ is a trace of the specification *Spec*. Then, due to Formula (7.22), this trace $\sigma = \sigma' \cdot \alpha$ belongs to $Traces(SP_{vis})$.

Next, as σ is a fail trace of TC, then due to the item (1) of the proof of this lemma, α is a valued *output* action of *Spec* and SP_{vis}. Then, as TG is obtained from SP_{vis} by the selection algorithm (*see* page 185) which does not prune any symbolic transitions of SP_{vis} that are labeled with *output* actions, and guarded with satisfiable expressions; then $\sigma = \sigma' \cdot \alpha$ is a trace of TG. This contradicts the fact that the fail trace $\sigma = \sigma' \cdot \alpha$ of TC does not belongs to the set of traces of TG (*see* the item (2) of the proof of Lemma 7.7, page 211).

Therefore, $\sigma \notin Traces(Spec)$, and the second part of the lemma is proved.

The items (1) and (2) imply the lemma. **Q.E.D.**

7.6 Correctness of a Test Case

At this point of the chapter we know how to generate a symbolic test case starting from a specification and a test purpose. However, we did not show that the generated test case produces correct results while executed on some implementation under test. Thus, the aim of this section is to prove the correctness of the given test case for a set of implementations in the sense of Definition 6.18 (*see* page 163).

General Hypotheses. In this section we consider:

(1) a specification $Spec \in \mathsf{SPECS}$ with set of variables V_{Spec} and set of symbolic constants C_{Spec}, which

   (a) does not contain any syntactic livelocks (*see* Definition A.4, page 257), and

   (b) is deterministic with lookahead $k \in \mathbb{N}$(*see* Definition A.15, page 307).

(2) a test purpose TP of *Spec* with set of variables V_{TP} and set of symbolic constants C_{TP},

(3) a test case $TC \in \mathsf{TESTS}$ derived by the symbolic test generation method (*see* Figure 7.1, page 168) from *Spec* and TP. This test case TC has the following set of symbolic constants $C_{TC} = (C_{Spec} \cup C_{TP}) \setminus (V_{Spec} \cup V_{TP})$, and

(4) the set of implementation under test $Iuts \subseteq \mathsf{IMPS}$ such that each implementation iut belonging to $Iuts$ is modeled by the IOSTS $\mathcal{I}_{iut} \in \mathsf{MODS}$ which:

   (a) is compatible for the parallel composition with TC (*see* Definition 5.1, page 100), and

   (b) has the set of symbolic constants $C_{\mathcal{I}_{iut}}$ such that there exists the bijective function $f : C_{\mathcal{I}_{iut}} \mapsto C_{Spec}$ such that $\forall\ c \in C_{\mathcal{I}_{iut}}\ .\ [\mathsf{type}(c) = \mathsf{type}(f(c))]$. Therefore, $\mathsf{DOM}(C_{\mathcal{I}_{iut}}) = \mathsf{DOM}(C_{Spec})$.

Then, we state and prove the following theorem about correctness of the given test case TC.

Theorem 7.3 (Correctness of TC) The test case TC which is generated from the specification *Spec* and a test purpose TP by the symbolic test generation method described in the first six sections of the current chapter is *correct* for the set of implementations under test $Iuts$. □

Proof Before proving correctness of the given test case we first introduce some notations. Let ς be a valuation of symbolic constants of TC, *i.e.* $\varsigma \in \mathsf{DOM}(C_{TC}) = \mathsf{DOM}((C_{Spec} \cup C_{TP}) \setminus (V_{Spec} \cup V_{TP}))$ (*see* the item (3) of the general hypotheses). Then, we denote by:

(a) $TC(\varsigma)$ an instance of the test case TC,

(b) $(Spec \times TP)(\varsigma)$ an instance of the synchronous product between *Spec* and TP.

   Notice that by test case construction, TC and $Spec \times TP$ have the same set of symbolic constants.

(c) $Spec(\varsigma_{Spec})$ an instance of the specification *Spec*, where $\varsigma_{Spec} = (\varsigma \downarrow_{C_{Spec}})$ and $\downarrow$ is the notation for the projection operation given on page 155.

(d) $TP(\varsigma_{TP})$ an instance of the test purpose TP, where $\varsigma_{TP} = (\varsigma \downarrow_{C_{TP}})$.

(e) $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$ an instance of the model $\mathcal{I}_{iut}$ of an implementation under test belonging to $Iuts$, where $\varsigma_{\mathcal{I}_{iut}} = (\varsigma_{Spec} \circ f)$ (this is possible to do due to the existence of the bijective function f between the sets of symbolic constants of $\mathcal{I}_{iut}$ and *Spec*, *see* item (4.b) of the general hypotheses).

Then, in order to prove the correctness of TC we have to prove its soundness, relative exhaustiveness, accuracy and conclusiveness. For this we consider an arbitrary instance $TC(\varsigma)$ of TC, and the corresponding instances $Spec(\varsigma_{Spec})$, $TP(\varsigma_{TP})$, $(Spec \times TP)(\varsigma)$ and $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$ of *Spec*, TP, $Spec \times TP$ and $\mathcal{I}_{iut}$.

Finally, before proving the above mentioned properties of the test case TC, it is important to emphasize that TC is:

(1) *initialized* (*see* Definition 4.19, page 92).

Indeed, we know that the given specification *Spec* and test purpose *TP* are initialized IOSTS (*see* Section 6.1 on page 137 and item (2) of Definition 6.8 on page 147). Moreover, we know that *TP* is complete with respect to *Spec* and compatible for the product operation with *Spec* (*see* the items (3) and (4) of Definition 6.8, 147). Therefore, we can use Lemma 5.4 (*see* page 121) saying that the synchronous product $(Spec \times TP)$ is an initialized IOSTS. Moreover, the test case TC generated from the initialized IOSTS $(Spec \times TP)$ is also initialized by construction.

(2) *deterministic* in the sense of Definition 4.20 (*see* page 93) by the test case construction. For more details see the second item of Theorem 7.2, page 203.

Therefore, for the instance $TC(\varsigma)$ of TC we implicitly make use of Theorem 4.1 (*see* page 94). *I.e.* when we say that a trace σ brings $TC(\varsigma)$ in a given state, we implicitly assume that $(TC(\varsigma)\ \mathsf{after}\ \sigma)$ is a singleton.

Soundness of TC. In this part of the proof we show that the test case TC rejects only non-conformant implementations under test. According to the definition of a sound test case (*see* page 146), we have to show the following implication:

$$(TC(\varsigma)\ \mathsf{may\_fail}\ \mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})) \implies \neg(\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})\ \mathsf{ioc}\ Spec(\varsigma_{Spec}))$$

We consider a trace $\sigma = \sigma' \cdot \alpha$ of $(\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}}) \parallel TC(\varsigma))$ producing the *Fail* verdict, and prove that $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$ is non-conformant with $Spec(\varsigma_{Spec})$ from which $TC(\varsigma)$ was generated.

Indeed, as $(\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}}) \parallel TC(\varsigma))$ generates the *Fail* verdict after executing the trace σ, *i.e.* $\mathsf{verdict}(\sigma) = Fail$, then we get that:

(1) $\sigma' \cdot \alpha \in Traces(TC(\varsigma))$ and $\sigma \in Traces(\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}}))$ (*see* Theorem 5.2, page 112), and

(2) $TC(\varsigma)\ \mathsf{after}\ \sigma \cdot \alpha \subseteq \mathsf{FAIL}$ (*see* Definition 6.5, page 145).

Thus,

(a) $\sigma' \cdot \alpha$ is a fail trace of $TC(\varsigma)$ (*see* items (1), (2) and Definition 7.4, page 208), and

(b) by the test case construction, α is a valued input action of $TC(\varsigma)$ (*see* Algorithm 7.3, page 202). Therefore, it is a valued output action of $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$.

Finally, as we know that $Traces_{fail}(TC) \subseteq (Traces(Spec) \cdot \Lambda^{!}_{Spec}) \setminus Traces(Spec)$ (*see* Lemma 7.7, page 211), then the trace $\sigma' \cdot \alpha$ of $TC(\varsigma)$ does not belong to $Traces(Spec)$, however, its maximal strict prefix does belong to $Traces(Spec)$, *i.e.* $\sigma' \cdot \alpha \notin Traces(Spec) \wedge \sigma' \in Traces(Spec)$.

Thus, we obtain that α is a valued output action which cannot be obtained in $Spec(\varsigma_{Spec})$ after the trace σ', *i.e.* $\alpha \notin Out(Spec(\varsigma_{Spec})\ \mathsf{after}\ \sigma')$.

Moreover, due to the items (1) and (b) given above, we know that α is a valued output action obtained in $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$ after σ', *i.e.* $\alpha \in Out(\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})\ \mathsf{after}\ \sigma')$.

Therefore, according to the definition of conformance given on page 140, we get that $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$ is non-conformant with $Spec(\varsigma_{Spec})$, and the soundness of the test case TC is proved.

Relative Exhaustiveness of TC. We prove that an implementation under test which is non-conformant to the specification *Spec* and relative to the test purpose TP may be rejected by the test case TC.

Formally, using Definition 6.15 (*see* page 161) we have to show that:

$$\neg(\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})\ \mathsf{ioc}_{TP(\varsigma_{TP})}\ Spec(\varsigma_{Spec})) \implies (TC(\varsigma)\ \mathsf{may\_fail}\ \mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}}))$$

Suppose that $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$ is not conformant to $Spec(\varsigma_{Spec})$ relative to $TP(\varsigma_{TP})$. Due to the definition of the conformance relative to the test purpose (*see* page 155) we know that:

$$\exists\ \sigma' \in SPref(ATraces((Spec \times TP)(\varsigma))),\ \alpha \in \Lambda^{!}\ . \quad [\ \alpha \in Out(\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})\ \mathsf{after}\ \sigma') \wedge \alpha \notin Out(Spec(\varsigma_{Spec})\ \mathsf{after}\ \sigma')\ ] \tag{7.23}$$

where $\Lambda^{!}$ is the set of valued output actions of the implementation $\mathcal{I}_{iut}$ and the specification *Spec*.

Let us choose:

(1) a trace σ' belonging to $SPref(ATraces((Spec \times TP)(\varsigma)))$, and

(2) a valued output action α of $Spec(\varsigma_{Spec})$ and $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$,

which satisfy Formula (7.23). Thus, we have that $\sigma = \sigma' \cdot \alpha$ is a trace of $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$, but not a trace of $Spec(\varsigma_{Spec})$, *i.e.* $\sigma \in Traces(\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}}))$ and $\sigma \notin Traces(Spec(\varsigma_{Spec}))$. $(\star)$

Then, we need to prove the two statements given below.

First, the trace σ' is a trace of $TC(\varsigma)$.

Indeed, as we know that σ' is a trace of $(Spec \times TP)(\varsigma)$ (*see* item (1) above and Definition 6.12 on page 154), then due to the following equalities:

$$
\begin{aligned}
& Traces((Spec \times TP)(\varsigma)) \\
& \qquad = \text{[see Theorem A.4, page 271]} \\
& Traces(\mathsf{closure}((Spec \times TP)(\varsigma))) \\
& \qquad = \text{[see Theorem A.9, page 303]} \\
& Traces(\underbrace{\mathsf{det}(\mathsf{closure}((Spec \times TP)(\varsigma))))}_{SP_{vis}(\varsigma)}
\end{aligned}
$$

we obtain that σ' is also a trace of $SP_{vis}(\varsigma)$.

Next, as we know that:

- σ' is a trace of $SP_{vis}(\varsigma)$ which is not an accepting trace, but the prefix of an accepting trace (*see* the item (1) at the beginning of the proof and Definition 6.12 on page 154), and
- the test graph TG is obtained from SP_{vis} by the selection algorithm (*see* page 185) whose purpose is to select all accepting traces of SP_{vis} such that their strict prefixes are not accepting traces of SP_{vis},

then σ' is a trace of the test graph $TG(\varsigma)$.

Therefore, as $Traces(TC(\varsigma)) = Traces(TG(\varsigma)) \cup Traces_{fail}(TC(\varsigma))$ (*see* Lemma 7.6 on page 210), then $\sigma' \in Traces(TC(\varsigma))$.

Second, $(TC(\varsigma)\ \mathsf{after}\ \sigma') \cap (\mathsf{PASS} \cup \mathsf{FAIL} \cup \mathsf{INCONCLUSIVE}) = \emptyset$.

Indeed, σ' is not an accepting trace of $(Spec \times TP)(\varsigma)$ (*see* the item (1) at the beginning of the proof and Definition 6.12 on page 154), then it cannot be a pass trace of $TC(\varsigma)$, as due to the selection algorithm (*see* page 185) only accepting traces may be transformed into pass traces of the test case. Therefore, $(TC(\varsigma)\ \mathsf{after}\ \sigma') \cap \mathsf{PASS} = \emptyset$.

Next, σ' is the prefix of an accepting trace of $(Spec \times TP)(\varsigma)$ (*see* the item (1) at the beginning of the proof and Definition 6.12 on

page 154), thus there exists at least one valued action α' executable after σ' in $Spec \times TP(\varsigma)$. Moreover, α' can also be executed in the test case $TC(\varsigma)$ after the trace σ' of $TC(\varsigma)$ (*see* Algorithm 7.2, page 185). Thus, σ' cannot be an inconclusive or fail trace of $TC(\varsigma)$ as by construction, the test case does not contain any symbolic transition outgoing from *Inconclusive* and *Fail* locations. Therefore, $(TC(\varsigma)\ \mathsf{after}\ \sigma') \cap (\mathsf{INCONCLUSIVE} \cup \mathsf{FAIL}) = \emptyset$, and the second statement has proved.

Due to the *First* and *Second* items above, we obtain that the trace σ' brings the test case $TC(\varsigma)$ to a state $s \in S_{TC}$ corresponding to a location $l \in L_{TC}$ different from *Pass*, *Inconclusive* and *Fail*. Moreover, as the test case is input-complete by construction (*see* the first item of Theorem 7.2, page 203), then by Definition 4.21 (*see* page 95) we know that any valued input action of $TC(\varsigma)$ can be executed from the state s.

Next, as α is an input valued action of $TC(\varsigma)$ (due to item (2) at the beginning of the proof, α is a valued output action of $Spec(\varsigma_{Spec})$; and due
($\star\star$) to the test case construction $\Lambda^{!}_{Spec(\varsigma_{Spec})} = \Lambda^{?}_{TC(\varsigma)}$), then $\sigma = \sigma' \cdot \alpha$ is a trace of $TC(\varsigma)$.

Finally:

(a) According to the observation ($\star$) we know that $\sigma \notin Traces(Spec(\varsigma_{Spec}))$. Thus, due to the following equalities and inclution:

$$
\begin{aligned}
& Traces(Spec(\varsigma_{Spec})) \\
& \qquad = \text{[}see\text{ Theorem 5.5, page 134]} \\
& Traces((Spec \times TP)(\varsigma)) \\
& \qquad = \text{[}see\text{ Theorem A.4, page 271]} \\
& Traces(\mathsf{closure}((Spec \times TP)(\varsigma))) \\
& \qquad = \text{[}see\text{ Theorem A.9, page 303]} \\
& Traces(\underbrace{\mathsf{det}(\mathsf{closure}((Spec \times TP)(\varsigma))))}_{SP_{vis}(\varsigma)} \\
& \qquad \supseteq \text{[}see\text{ Corollary 7.1, page 195]} \\
& Traces(\underbrace{\mathsf{select}(\mathsf{det}(\mathsf{closure}((Spec \times TP)(\varsigma)))))}_{TG(\varsigma)}
\end{aligned}
$$

we obtain that $\sigma \notin Traces(TG(\varsigma))$. Therefore, as we know that $\sigma \in Traces(TC(\varsigma))$ (*see* the observation ($\star\star$)), then due to Lemma 7.6 (*see* page 210), σ is a *fail* trace of $TC(\varsigma)$, *i.e.* $(TC(\varsigma)\ \mathsf{after}\ \sigma) \subseteq \mathsf{FAIL}$.

(b) Due to the observations ($\star$) and ($\star\star$) we know that $\sigma \in Traces(\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}}))$ and $\sigma \in Traces(TC(\varsigma))$. Then, by Theorem 5.2 (*see* page 112), $\sigma \in Traces(\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}}) \parallel TC(\varsigma))$.

The items (a) and (b) imply that the test case $TC(\varsigma)$ produces the *Fail* verdict after the execution of the trace σ on $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$. Thus, due to Definition 6.6 (*see* page 146) $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$ may be rejected by the test case $TC(\varsigma)$.

Therefore, the test case TC is relatively exhaustive.

Accuracy of TC. In this part of the proof we show that if a test case TC derived from a specification *Spec* and a test purpose TP of *Spec*, produces the *Pass* verdict, then the observed trace of the implementation under test *iut* is a trace of *Spec* that leads to acceptance of TP.

Due to Definition 6.17 (*see* page 162), we have to prove that:

$$\forall\, \sigma \in Traces(TC(\varsigma) \parallel \mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})) \;.\; [\; \mathsf{verdict}(\sigma) = Pass \implies \sigma \in ATraces((Spec \times TP)(\varsigma))\;]$$

Consider a trace σ belonging to the set of traces of the IOSTS $(TC(\varsigma) \parallel \mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}}))$. Then, as the test case $TC(\varsigma)$ produced the *Pass* verdict while executing σ on $\mathcal{I}_{iut}(\varsigma_{\mathcal{I}_{iut}})$, then due to Definition 6.5 (*see* page 145) we get that $(TC(\varsigma)\ \mathsf{after}\ \sigma) \subseteq \mathsf{PASS}$. Therefore, according to Definition 7.3 (*see* page 196), we obtain that σ is a *pass* trace of $TC(\varsigma)$.

Next, as we know that:

(1) $Traces_{pass}(TC(\varsigma)) = Traces_{pass}(TG(\varsigma))$ (*see* Lemma 7.4, page 208), where $TG(\varsigma)$ is an instance of the test graph TG from which the test case was obtained by the last step of the symbolic test generation method (*see* Figure 7.1, page 168), and

(2) $Traces_{pass}(TG(\varsigma)) \subseteq ATraces((Spec \times TP)(\varsigma))$ (*see* Lemma 7.2, page 196),

then σ is an accepting trace of $(Spec \times TP)(\varsigma)$. Therefore, the test case TC is accurate.

Conclusiveness of TC. We prove that the test case TC produces an *Inconclusive* verdict only when the observed trace of the implementation under test is a trace of the specification *Spec* ending with an output action, but it cannot be extended into a trace producing the *Pass* verdict.

According to Definition 6.17 (*see* page 162), we have to show that:

$$\begin{array}{l} \forall\ \sigma \in \mathit{Traces}(\mathit{TC}(\varsigma) \mid\mid \mathcal{I}_{\mathit{iut}}(\varsigma_{\mathcal{I}_{iut}}))\ . \\ \quad [\ \mathsf{verdict}(\sigma) = \mathit{Inconclusive} \implies \\ \quad\ \sigma \in (\mathit{Traces}(\mathit{Spec}(\varsigma_{\mathit{Spec}})) \cdot \Lambda^{!}_{\mathit{Spec}(\varsigma_{\mathit{Spec}})}) \cap \mathit{Traces}(\mathit{Spec}(\varsigma_{\mathit{Spec}})) \wedge \\ \quad\ \sigma \notin \mathit{Pref}(\mathit{ATraces}((\mathit{Spec} \times \mathit{TP})(\varsigma)))\ ] \end{array}$$

Consider a trace σ belonging to the set of traces of the IOSTS $(\mathit{TC}(\varsigma) \mid\mid \mathcal{I}_{\mathit{iut}}(\varsigma_{\mathcal{I}_{iut}}))$. Then, as the test case $\mathit{TC}(\varsigma)$ produced the *Inconclusive* verdict while executing σ on $\mathcal{I}_{\mathit{iut}}(\varsigma_{\mathcal{I}_{iut}})$, then due to Definition 6.5 (*see* page 145) we get that $(\mathit{TC}(\varsigma)\ \mathsf{after}\ \sigma) \subseteq \mathsf{INCONCLUSIVE}$. Therefore, according to Definition 7.3 (*see* page 196), we obtain that σ is an *inconclusive* trace of $\mathit{TC}(\varsigma)$.

Next, as we know that:

(1) $\mathit{Traces}_{\mathit{inconc}}(\mathit{TC}(\varsigma)) = \mathit{Traces}_{\mathit{inconc}}(\mathit{TG}(\varsigma))$ (*see* Lemma 7.5, page 209), where $\mathit{TG}(\varsigma)$ is an instance of the test graph from which the test case was obtained by the last step of the symbolic test generation method (*see* Figure 7.1, page 168), and

(2) $\mathit{Traces}_{\mathit{inconc}}(\mathit{TG}(\varsigma)) \subseteq (\mathit{Traces}(\mathit{Spec}(\varsigma_{\mathit{Spec}})) \cdot \Lambda^{!}_{\mathit{Spec}(\varsigma_{\mathit{Spec}})} \cap \mathit{Traces}(\mathit{Spec}(\varsigma_{\mathit{Spec}}))) \setminus \mathit{Pref}(\mathit{ATraces}((\mathit{Spec} \times \mathit{TP})(\varsigma)))$ (*see* Lemma 7.3, page 200),

then σ is a trace of the specification $\mathit{Spec}(\varsigma_{\mathit{Spec}})$ ending with an output action of $\mathit{Spec}(\varsigma_{\mathit{Spec}})$, but it not an accepting trace of $(\mathit{Spec} \times \mathit{TP})(\varsigma)$. Therefore, the test case TC is conclusive.

Q.E.D.

Conclusion

In the second part of the thesis we described the main principles of symbolic test generation. All along this part we were guided by the objectives listed below.

Choosing of a Model for Representation of Reactive Systems. In order to automatically generate test cases for reactive systems, these systems must be modelized. As we showed it in the introductory part of the thesis, there exist many models that can be used for modelizing such systems, for example, FSM or IOLTS. However, almost all of them are too limited: for instance, they do not allow the explicit representation of data. In this thesis we propose a model, called Input-Output Symbolic Transition System (IOSTS), that enables us to describe reactive systems more precisely by explicitly taking into account their data. This model allows us to generate test cases under the form of programs with variables, symbolic constants and parameters (*i.e.* generic test cases, which are closer to industrial practice, for instance, to TTCN [ISO/IEC/JTC1/SC21, 1992] test cases), and to avoid the classical state-space explosion problem.

Introducing a New Test Generation Method. In Chapter 7 of the thesis, we proposed a method for automatic test generation that treats the data of a given system symbolically by combining a test generation approach proposed earlier in our research group and described in Section 3.2.3.3 (*see* page 57) with *abstract interpretation* [Cousot and Cousot, 1976], [Cousot and Cousot, 1977]. This method enables us to avoid the problem of state-space explosion and to derive generic test cases.

One of the points that confers originality to our symbolic test generation method is the selection step. The purpose of this step is to compute a test graph from a given IOSTS SP_{vis} that consists in the states which are reachable from some initial state of SP_{vis}, and from which it is possible to go towards some accepting states of SP_{vis}. Remember that we cannot use the exact computation of reachable and coreachable states since, in general, the computation does not terminate. Therefore, we are compelled to compute only *over-approximations* of such states. For this, we use abstract interpretation, which is a very powerful technique enabling us to calculate different

properties of the system statically, for instance, the over-approximations of reachable and coreachable sets of states. However, one difficulty in the use of such a technique is the choice of the best abstraction, *i.e.* we try to make a compromise between a precise enough abstraction and the time-complexity of a technique using this abstraction. Moreover, we mention that in the STG tool that will be described in the next part of the thesis, we use an abstract interpretation technique as a black-box. This choice does not oblige us to know all theoretical details of abstract interpretation and, at the same time, it enables us to use and compare different existing kinds of abstractions. Up to now, we have connected STG with the NBac tool [Jeannet, 2000a] which, using the polyhedra abstraction, computes over-approximations of reachable and coreachable states. We believe that future research in this direction can improve our test case generation method.

At the end of the second part of the thesis proved that the symbolic test generation method derives abstract test cases that are correct in the sense of Definition 6.18, page 163 (*i.e.* they are sound, relatively exhaustive, accurate and conclusive). These test cases can be easily translated into some programming language (for example, C++ or Java) and executed on a real implementation under test (*see* [Clarke et al., 2001c] and [Ponscarme, 2002]).

In the rest of the thesis we will present the prototype called STG (Symbolic Test Generator) in which we have implemented the symbolic test generation algorithm proposed in the second part of the thesis. We will also describe the use of STG in the testing of the bounded retransmission protocol [Helmink et al., 1994]. Finally, we will compare STG with other existing tools that generate symbolic test cases for conformance testing of reactive systems.

Part III

Implementation and Experimental Results

Chapter 8

STG: Symbolic Test Generator

*STG is a tool that implements the symbolic test generation method presented in Chapter 6 (*see *page 137) in order to automatically derive symbolic test cases from a formal specification of a system under test and a test purpose which describes a set of the system's behaviors to be tested (see* [Clarke et al., 2002], [Clarke et al., 2001b], [Clarke et al., 2001c]). *We first present the architecture of the STG tool. Then, we describe the use of STG in testing of a communication protocol that transfers a data file though lossy channels as a sequence of packets. The protocol is based on the well-known alternating bit protocol, but it allows for only bounded number of retransmission of each packet. Finally, we compare STG with other tools used in the testing community.*

The first prototype of the STG tool has been developed by Duncan Clarke. In this prototype Duncan has implemented all steps of the symbolic test generation described in Chapter 6 (*see* page 137) except of the determinization, moreover, the selection of test cases (*i.e.* the computation of over-approximations of the reachable and coreachable sets of states) has been based on the control structure of IOSTS of (for more details see the paper [Rusu et al., 2000]). Recently, the author of this thesis in the collaboration with Bertrand Jeannet has implemented the semantic-based (or, data-based) selection of test cases (in order to compute over-approximations of the reachable and coreachsble sets of states of a given IOSTS, we approximate this IOSTS with polyhedra). This work was published in the paper [Zinovieva, 2002].

8.1 Architecture of STG

This section describes an architecture of the STG tool shown on Figure 8.1 (*see* page 227), and explains the connections of STG with other tools.

Currently the STG tool supports three phases that are briefly described in next paragraphs.

Parsing and Compiling. The phase of parsing and compiling takes as its input a file containing a formal specification and test purposes represented in a language similar to IF ([Bozga et al., 1999b], [Bozga et al., 2002]) which is rather intermediate representation between low-level models as FSM, LTS, or IOSTS, and popular high-level description languages as SDL [ITU-T, 1994], Statecharts [Harel, 1987] or UML [Fowler and Scott, 2000] (where contraints are represented in OCL [Warmer and Kleppe, 1998]). Then, this file is parsed, and the specification and test purposes are compiled into low-level IOSTS models which are used in the symbolic test generation phase.

Symbolic Test Generation. The phase of the symbolic test generation (1) takes as its inputs a specification *Spec* and a test purpose *TP* of *Spec* represented as IOSTS, and (2) derives from them a test case that covers all behaviors of the specification selected by the test purpose. The symbolic test generation phase has been presented in the details in Chapter 7 (*see* page 167). Below we remain the main steps of this phase, as implemented in the current version of the STG tool.

(a) *Product* allows to intersect the behaviors of the given specification *Spec* with the test purpose *TP* which must be complete with respect to *Spec*. The aim of this operation is to mark behaviors of the specification which we want to test as *accepting*. For details you can see Section 7.2 on page 173.

(b) *Closure and Determinization* takes the synchronous product *SP* from the previous step, and produces a trace-equivalent IOSTS SP_{vis} that (1) has no internal actions, and (2) is deterministic (*see* Sections A.1 and A.2, pages 246 and 274). Notice that in the current version of the STG tool the algorithm for determinization is in the phase of implementation.

(c) *Selection* allows to select a subgraph of SP_{vis} that leads to the satisfaction of the test purpose and contains fewer unreachable states as SP_{vis}. The detailed description of the selection step is given in Section 7.4 (*see* page 178).

(d) *Adding the Fail verdict.* This step consists in making the IOSTS computed at the previous step input-complete by adding symbolic transitions leading

Figure 8.1: Architecture of STG.

to the new *Fail* location. This step is needed as the generated test case must properly react on all inputs outgoing from an implementation under test. If some input is not specified in *Spec*, then the test case must produce the *Fail* verdict (*see* Section 7.5, page 202).

After all these steps we obtain a correct (*see* Definition 6.18, page 163) test case with three kinds of verdicts: *Pass*, *Inconclusive* and *Fail*. *Pass* means that no errors were detected and the test purpose was satisfied (*i.e.* there were no observable difference between implementation and specification). *Inconclusive* means that, although no errors were detected, the test purpose cannot be satisfied any more. *Fail* means that an error was detected.

Translation to C++/Java. The conversion phase translates the test case obtained after symbolic test generation, into a concrete test program capable of interacting with an implementation interface-compatible with original IOSTS specification. The test program is then ready to be compiled and linked with the black-box implementation under test for the test case execution. The details of the conversion phase into C++ and Java languages can be found in [Clarke et al., 2001c] and [Ponscarme, 2002] respectively.

8.1.1 Connection of STG with Other Tools

The STG tool is integrated with three tools described below.

Dotty [Gansner and North, 2000] is used to view IOSTS in graphical form, fully decorated with locations names, guards, input/output/internal actions and assignments. For instance, Figures 6.1, 6.5 and 7.12 (*see* page 138, 149 and page 207) which are the specification, the test purpose and the test case of a simple coffee machine, were produced by Dotty.

NBac [Jeannet, 2000a] is a tool for analyzing synchronous and deterministic reactive systems containing combination of Boolean and numerical variables. It is used for computing an over-approximation of reachable and coreachable sets of states for IOSTS. The information about the reachable and coreachable states is used by the selection algorithm (*see* page 185).

Omega Calculator [Kelly et al., 1995] is a tool for analyzing formulas in Presburger arithmetic. It is used during the test execution phase, in order to evaluate the constraints and select specific values for outputs of the tester.

8.2 Case Study: Bounded Retransmission Protocol

The Bounded Retransmission Protocol (BRP) is a simplified variant of a telecommunication protocol used by Philips. The purpose of this protocol is to transfer a *large* file across a lossy channel as a sequence of small packets (called *chunks*) within a *limited amount of time*. After the file transmission, the BRP indicates whether this file was successful transmitted or not. Moreover, there are possible situations in which the protocol does not know whether the retransmission of the file was a success. The BRP is parametrized by (1) the retransmission bound, and (2) the number of chunks.

The bounded retransmission protocol has attracted the interest of the research community due to the fact that it has interesting aspects to verify, to prove and to test. For instance, L. Helmink, M. Sellink and F. Vaandrager [Helmink et al., 1994] modeled the BRP as Input-Output Automata [Lynch and Tuttle, 1989] and proved its correctness using the COQ system [Dowek et al., 1993], [Bertot and Castéran, 2004]. R. Mateescu [Dowek et al., 1993] translated the μCRL representation of the BRP protocol given in [Groote and van de Pol, 1993] into the LOTOS [ISO/IEC, 1988] language and verified the correctness of this protocol using model-checking rather than theorem-proving. K. Havelund and N. Shankar [Havelund and Shankar, 1996] used an abstracted version of the BRP obtained by the PVS [Owre et al., 1992] theorem prover, and verified its correctness using model-checking with MURΦ [Melton et al., 1992] state exploration tool. V. Rusu, L. du Bousquet and T. Jéron [Rusu et al., 2000] proposed a theoretical approach to symbolic test generation, and illustrated it using the BRP protocol.

The aim of this section is to present the sender of the bounded retransmission protocol as an input-output symbolic transition system, and to generate a test case that examines some behaviors of this protocol. For this we will use the STG tool presented in the previous section.

Plan of the Section. In this section we first describe the architecture of the bounded retransmission protocol. We then explain how to modelize the sender part of this protocol as an IOSTS. We also present a particular test purpose for the BRP. Finally, we show the test case that was automatically derived (using the STG tool) from the BRP specification and the given test purpose.

8.2.1 Architecture of the BRP

The two main purposes of the bounded retransmission protocol presented in this section are (1) to receive a file from the user, and (2) to deliver this file

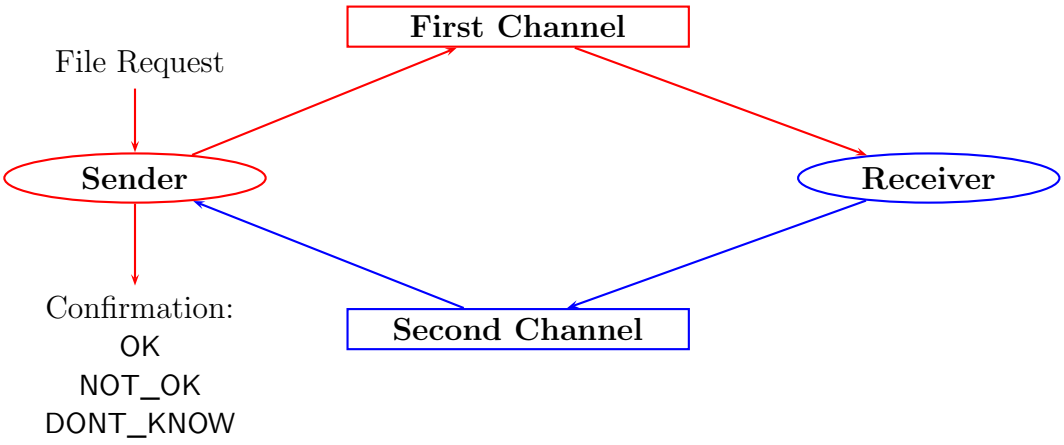


Figure 8.2: The architecture of the Bounder Retransmission Protocol.

as a sequence of small chunks accompanied with an alternating bit to the receiver. The alternating bit is used to detect the duplication of transmitted chunks. After each file transfer the user receives a confirmation: **OK** in the case of success, **NOT_OK** in the case of failure, *i.e.* some chunk was lost, and **DONT_KNOW** if neither success nor failure can be established (this case will be explained later).

More precisely, the BRP consists of a sender that transmits a chunk across a first lossy channel to a receiver that acknowledges each transmission along a second lossy channel (*see* Figure 8.2, page 230). These two channels can either lose a message (which is either a chunk of the file or acknowledgement) or deliver it correctly. After transmitting a chunk the sender waits for an acknowledgement. If no acknowledgement arrives then it perform the timeout and resends the chunk. The sender attempts to send each chunk at most *cChunkMax* times. If for some chunk (except of the last one) even the *cChunkMax* attempt failed, then the rest of the file is skipped and the user receives the **NOT_OK** confirmation. If the resent limit is reached for the last chunk of the file, then the file might have been transmitted successfully. In this case the sender is not sure, thus the user will receive the **DONT_KNOW** confirmation. Notice that the resend limit can be reached due to a lost acknowledgement as the acknowledgement channel is lossy as well.

In the rest of this section we explain a *unit* testing of the bounded retrans-

mission protocol. More precisely, we focus on the testing of the sender part of the BRP.

8.2.2 The Specification of the BRP sender

Figure 8.3 (*see* page 231) presents the IOSTS specification for the sender of the bound retransmission protocol whose architecture was described in the previous subsection. We remind that an IOSTS is made up of locations and symbolic transitions. Each symbolic transition is decorated with a guard, an input/output/internal action, and a set of assignments (we use the `:=` notation for a single assignment). A symbolic transition is firable when its guard is satisfied and a complementary action is offered by the environment for synchronization. Transitions labelled with internal actions (**Timeout1** and **Timeout2**) can be fired without synchronizing with the environment. The specification of the BRP sender has:

(1) two symbolic constants *cChunkMax* and *cRetMax*, which respectively stand for the maximal number of chunks in the file being sent, and for the maximal number of retransmissions of one chunk.

(2) four variables *vFile*, *vAltBit*, *vChunkNumber* and *vRetNumber*, where (a) *vFile* is used to memorize the file that the sender must transfer across a lossy channel; (b) *vAltBit* used in order to detect duplication of the chunks; and (3) *vChunkNumber* and *vRetNumber* are needed to count respectively the chunks and retransmissions.

(3) three communicating parameters *pFile*, *pChunk* and *pAltBit*, where (a) *pFile* is a file to be transmitted; and (b) *pChunk* and *pAltBit* are respectively a chunk and an alternating bit to be sent from the sender to the receiver thought the first lossy channel.

The BRP sender behaves as following. On reception of the **REQ** input action with a file *pFile* from the user, the sender first saves this file in a variable *vFile*, and then iteratively sends chunks *vFile[i]* (where $i = 0..cMaxChunk$) accompanied with an alternating bit *vAltBit* to the user thought the **SEND** output action. A chunk can be either acknowledged (*see* the **ACK** input action), or timed out (*see* the **Timeout1** internal action). In the case of time out, the sender resends the same chunk. The status of the whole transmission is described by either (a) **OK** meaning that all chunks were sent and acknowledged, (b) **DONT_KNOW** meaning that all chunks were sent and all except of last one were acknowledged, and (c) **NOT_OK** meaning that meaning that some intermediary chunk was not acknowledged.

Figure 8.3: The specification of the BRP sender.

8.2.3 The Test Purpose

In order to extract a test case from the given specification, we have to provide the STG tool with a supplementary information called test purpose. A test purpose describes the behaviors of the specification that have to be exercised.

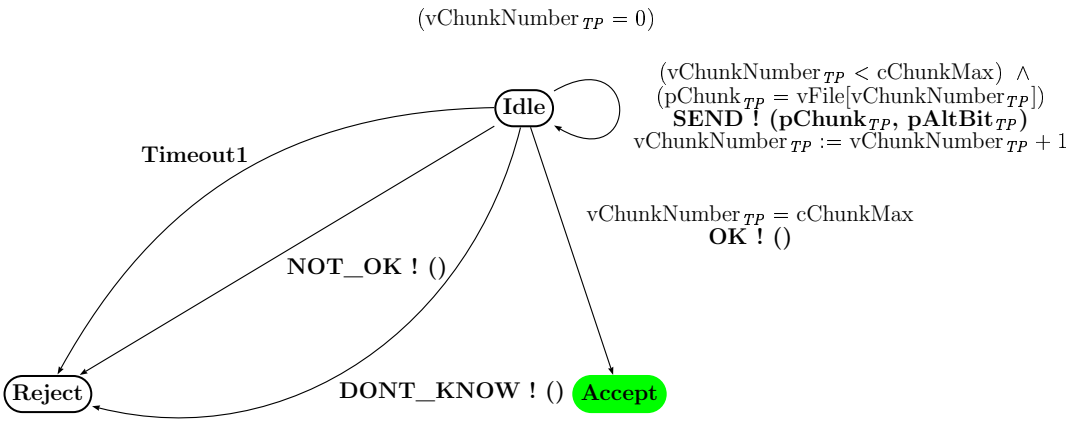


Figure 8.4: BRP test purpose.

Figure 8.4 (*see* page 233) illustrates a test purpose that selects from the BRP specification shown on Figure 8.3 (*see* page 231), the feature where the the BRP sender transmits all chunks exactly once (*i.e.* without retransmissions), and the user receives the **OK** confirmation indicating a successful outcome.

This test purpose has two symbolic constants $vFile$ and $cChunkMax$; one variable $vChunkNumber_{TP}$; and one parameter $pChunk_{TP}$. It is provided with the *Accept* location that indicates the accepting behaviors of the BRP specification. Also notice that the *Reject* location of the test purpose is used to discard executions in which the BRP sender does not behave as intended, *i.e.* it executes the internal action **Timeout1** known to be followed by a retransmission, or it send to the user the **DONT_KNOW** or **NOT_OK** confirmation.

8.2.4 The Test Case

Finally, Figure 8.5 (*see* page 234) shows the IOSTS representing a test case derived from the specification (*see* Figure 8.3, page 231) and test purpose (*see* Figure 8.4, page 233) by symbolic test generation process (for the details you can see either the Section 8.1, page 225 or Chapter 7, page 167). This test case is specific to this test purpose, as different test purposes generate different test cases.

It we take the close look at the test case generated by the STG tool and depicted in Figure 8.5 (*see* page 234), then we first can see that it examines

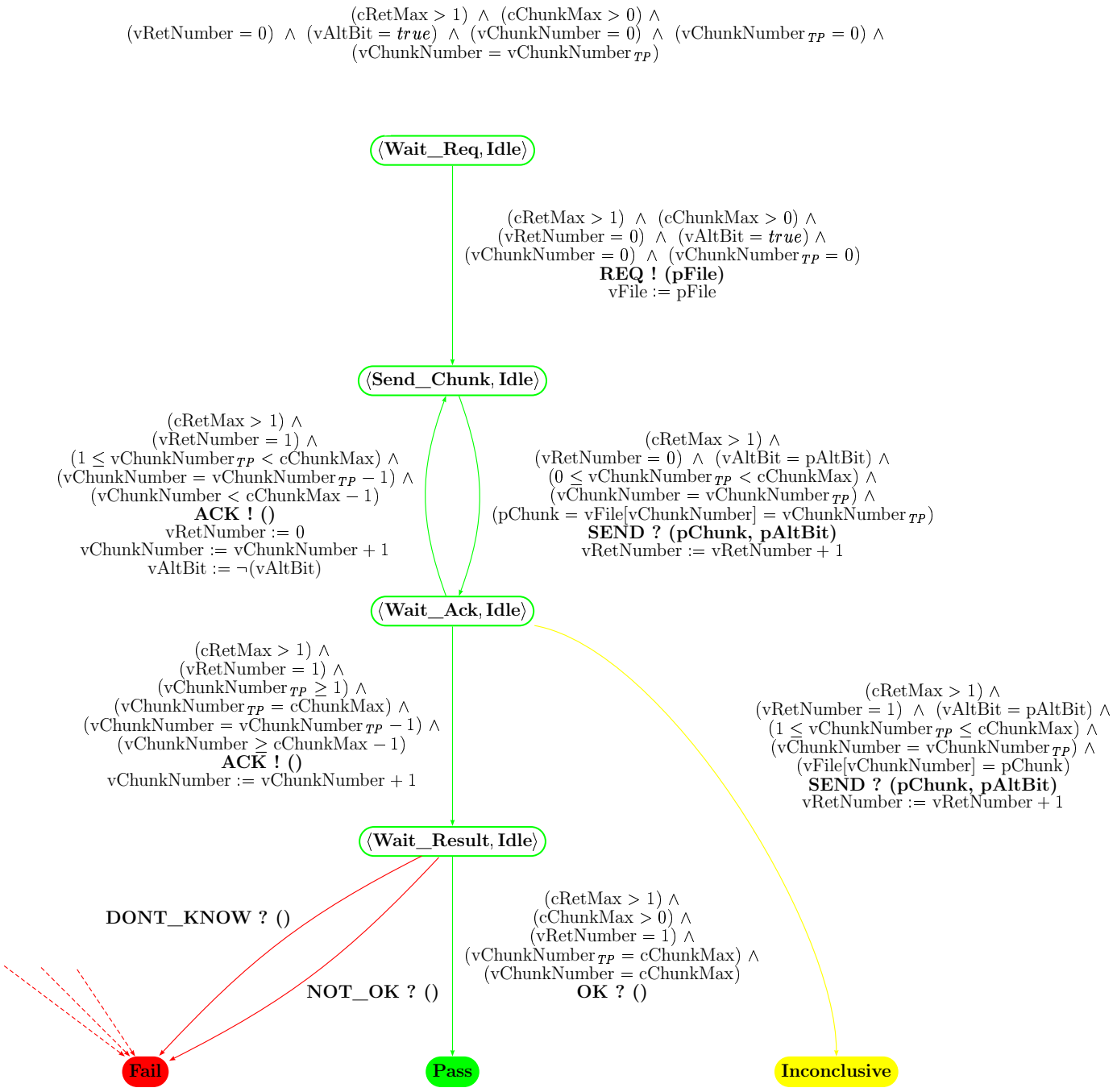


Figure 8.5: The test case for the BRP sender.

the behaviors of the specification that were targeted by the test purpose. More precisely, it starts by giving to the sender, a file ***vFile*** to transmit. Then it expects to receive the successive chunks of this file, and tries to acknowledge each chunk. If it succeeds and receives the **OK** confirmation, then the *Pass* verdict is generated (*i.e.* the test case moves to the *Pass* location shown in green on Figure 8.5, page 234). This means that the implementation behaved in conformance with its specification and the test purpose is satisfied, indeed, all chunks were sent without retransmission. If the test case gets another copy of the chunk that it has just received, then the sender performed a retransmission. In this case the *Inconclusive* verdict is generated (*i.e.* the test case moves to the *Inconclusive* location shown in yellow on Figure 8.5, page 234). This may happen if the test execution did not immediately acknowledge the sent chunk (*i.e.* the sender did not receive an acknowledgement, and, therefore, it proceeds to resending the same chunk.) Finally, if some other input is received, the verdict *Fail* is generated (*i.e.* the test case moves to the *Fail* location shown in red on Figure 8.5, page 234). Indeed, in this case the implementation emitted an output that is not allowed by specification. Due to the limited space Figure 8.5 (*see* page 234) does not show all symbolic transitions leading to the *Fail* location.

It is important to emphasize that the test case shown in Figure 8.5 (*see* page 234), like all other test cases generated by our method, incorporates its own oracle. All of the computation steps necessary to verify the correctness of numeric results are extracted from the specification and used by the tester to verify arguments as they are received.

8.2.5 Conclusion

This section has described an example of using STG for testing the sender of the bounded retransmission protocol. We have focused on the explanation of how to generate a test case starting from specification and test purpose represented as IOSTS models. We did not explain the first and last phases of the STG tool, namely:

- parsing and compiling of specifications and test purposes written in a dialect of IF-language [Bozga et al., 1999a], into low-level IOSTS models, and
- translation of the IOSTS test cases to C++/Java programs which are ready to be compiled, linked to an interface-compatible implementation, and executed.

as they are quite technical, and are described in [Clarke et al., 2001c] and [Ponscarme, 2002].

The STG tool is also used in the development of larger case studies based on the Common Electronic Purse System (CEPS) whose technical specification

is given in [CEPSCO, 2000]. In paper [Clarke et al., 2001a] we have already reported the first results obtained by testing the part of this specification which is called "CEP Inquiry – Slot Information" (*see* Section 8.7.1 of [CEPSCO, 2000]). Currently the STG tool is being used to incrementally test a prototype implementation of the full CEPS specification.

8.3 Related Works

In this section we present the main aspects that show the difference between our test generation method described in the second part of the thesis and implemented in the STG tool, and the other symbolic techniques for automatic test generation which were presented in the last section of the introductory part of the thesis.

Non-Deterministic Specifications. One of the aspects making a difference between STG and the tools mentioned in Section 3.3 (*see* page 67) is that: the STG tool is able to derive test cases from a subclass of *non-deterministic* specifications (the theoretical work about this aspect is done in Section A.2 of this thesis, *see* page 274), while other tools are often limited to *deterministic* specifications.

Abstract Interpretation. The second remarkable aspect concerns methods used for the test case selection and the form of generated test cases. It is not hard to notice that the techniques described in Section 3.3 (*see* page 67) are based on the *constraint technology* which has been popular during the last few years. It is also important to remark that these techniques are able to generate test cases in the form of test sequences (or trees) of a *bounded* length, while the STG tool derives test graphs.

Indeed, the GATeL [Marre and Arnould, 2000] tool derives test cases from a given LUSTRE formal specification by interpreting it as a set of constraints over boolean and integer variables, and then solving this set of constraints. The Agatha [Lugato et al., 2002] tool uses the symbolic simulation in order to generate a test execution tree, where each node is represented as a constraint over the variables characterizing the path of a given specification leading to this node, and then searchs for a possible instantiation of this tree using some constraint solver. The BZ-Testing-Tool [Ambert et al., 2002], developed by the researchers of LIFC, University of Franche-Comté, in order to derive test sequences, computes first all boundary states for a given system using the CLPS constraint solver, then takes the system from the initial state to some boundary state using the symbolic simulation, and finally tests all possible operations of the system in this boundary state. Moreover, the test generation tools used for structural (white-box) testing also use the constraint logic programming technology. For instance,

INKA [Gotlieb et al., 1998], based on the constraint solving, allows to generate a path in the control flow graph of a given C program leading to a specified point of this program. C. Meudec in his works on automatic test generation (*e.g.* [Meudec, 2001]) uses the CLP solver for symbolic execution of ADA programs in order to generate test cases.

The method proposed in this thesis is quite different from the methods and tools mentioned above. Indeed, instead of the use of the constraint technology in order to derive test cases, we choose to use a technique of abstract interpretation. This technique allows us to generate test cases under the form of test graphs instead of bounded test sequences as the most of the tools described above do. This representation of test cases gives us a possibility to have a global view on the "past" and the "future" of a system under test. Unfortunately, by obtaining the "generality" we loose the precision of derived test cases.

Another important aspect that distinguishes our test generation method from the other ones concerns the *labelling*. The labelling is the process of the instantiation of some system's variables with their values. This permits to activate some constraints. More precisely, the labelling consists in unfolding a symbolic test sequence according to the values of some variables. Therefore, the labelling can be considered as a kind of rough abstraction, where the system's control structure is refined by the possible values of some variables. The technique of abstract interpretation is more precise. Indeed, the control structure of a system under test is refined only when it is really needed, *i.e.* in the case when behaviors described the past and the future of the system are divergent.

The technique of abstract interpretation becomes more and more popular in the fields of verification and testing. For instance, commercial tools developed by the PolySpace Technology company (*see* `http://www.polyspace.com`), widely use abstract interpretation in order to automatically detect run-time errors at the compiling time of programs written in ADA, C and C++. The idea of such tools is following: they first compute some abstraction of a given program, and then solve the problem on this abstraction. If any error is not detected in the abstraction of the program, then it definitely does not exist in the program itself. However, if an error is detected in the abstraction, this means that the program *may contain* this error as well.

Chapter 9

Conclusion

The work presented in the preceding chapters has been focusing on the development of testing techniques for reactive systems, from theory up to implementation. In this chapter we reassess the main objectives that have been fixed at the beginning of the thesis, and we summarize the work that has been done in order to achieve these objectives. Finally, we give some ideas and hints for future research.

9.1 Summary

Motivation. This thesis has been placed in the area of black-box conformance testing for reactive systems. In the first chapter, we explained the importance of reactive systems in daily life. Such systems are usually large and complex, and it is thus not so evident to implement them without any error. Even a small error may lead to serious disfunctioning of the system (*see* the examples given on page 2). Therefore, it is important to develop techniques detecting errors in reactive systems and which make sure these systems are more error proof. In this thesis we focused on the study of one of those techniques called *testing*.

In the second and third chapters we mentioned that during the last decades testing theories and techniques for automatic test derivation have been developed. Some of these techniques have been based on the model of Finite State Machines (or FSM), and others on the model of (Input-Output) Labeled Transition Systems (or (IO)LTS). Since the model of IOLTS is more suitable for the testing of reactive systems than FSM, then we have focused on the study of the methods and algorithms for the automatic test derivation based on IOLTS. We have particularly reviewed two efficient on-the-fly algorithms proposed in [Tretmans, 1992], [Jéron and Morel, 1999]. We have also described already existing academic (*e.g.* TorX [Belinfante et al., 1999], TGV [Fernandez et al., 1996]) and industrial (*e.g.* Autolink [Telelogic, 1998], TestComposer [Kerbrat and Ober, 1999]) tools imple-

menting these algorithms and produce correct test cases, which means essentially that they always emit the correct verdict.

Nevertheless, the theories and tools based on IOLTS are somewhat limited. They do not *explicitly* take into account the system data because the underlying model of IOLTS does not allow to do it. Thus, in order to derive test cases from a specification of the reactive system modeled by an IOLTS, it is necessary to *enumerate* the values of each datum used by the system. This may result in the classical state-space explosion problem. Moreover, this enumeration also has the effect of obtaining test cases, where all data are instantiated. This contradicts the industrial practice, where test cases (written, for instance, in the TTCN [ISO/IEC/JTC1/SC21, 1992] language) are real programs with data (variables, symbolic constants and communication parameters).

Summary of the Thesis. In order to solve the issues mentioned above, we proposed a new approach for the automatic generation of *symbolic* test cases under the form of extended labeled transition systems with variables, symbolic constants and communication parameters. These systems are called Input-Output Symbolic Transition Systems (or IOSTS). In Chapter 4 we described the syntax and semantics of the IOSTS model. We also introduced some subclasses of IOSTS that are important for conformance testing (*e.g.* deterministic or input-complete IOSTS).

In Chapter 6 we presented the background for conformance testing based on IOSTS. Namely, we formally defined (1) the conformance relation ioc between a specification and an implementation under test that are both modeled with IOSTS, (2) the notion of a test case and (3) the properties of test cases that enable to establish the connection between test cases and the conformance relation. The work presented in this chapter was mainly inspired from the theory of conformance testing developed by J. Tretmans and his colleagues from the University of Twente, Netherlands (*see* [Tretmans, 1994], [Tretmans, 1996b]) and the research done in the VerTeCs team at IRISA/INRIA Rennes, France (*see* [Rusu et al., 2000], [Jard and Jéron, 2002]).

Then, in Chapter 7 we described the symbolic test generation approach based on the testing theory introduced above. This approach is a generalization of that proposed in earlier works of our group in collaboration with the Verimag laboratory in Grenoble [Fernandez et al., 1996] and referred to in the introductory part of the thesis. However, some steps of our symbolic test generation approach (especially concerning the selection of a test graph) are more complex than those proposed in [Fernandez et al., 1996], as they have to take into account the data of a given system.

The symbolic test generation approach was implemented in the STG tool. The description of this tool and a case study of the bounded retransmission protocol

were given in the third part of our work.

At the end of the thesis, we compared our approach with other symbolic approaches for automatic test generation. The majority of these are based on constraint propagation and use constraint solving techniques. Moreover, they do not deal with the issue of non-determinism (remember that non-determinism is highly prohibited in testing). The original points of our approach are: the use of a technique of abstract interpretation at the moment of test case selection, and the partial solution for non-determinism. Notice that we did not solve all problems with respect to symbolic test generation. In particular, our approach also uses a constraint solving technique while instantiating symbolic test cases during a test execution. The technical details of test execution were not explained in this document, but some of them can be found in [Clarke et al., 2001c].

We believe that our symbolic test generation approach deserves future attention, and that research to come can improve its applicability to industrial reactive systems.

9.2 Future Research

The following ideas for future research are inspired from the work presented in this thesis.

From ioc to ioco. Before going into details of the perspective described in this paragraph, we remind the difference between the ioco [Tretmans, 1996a] and ioc (*see* Definition 6.3, page 141) relations. That is, in the first hand, the ioco relation makes a connection between a *suspended* specification $\Delta(Spec)$ and the *suspended* model of an implementation under test $\Delta(\mathcal{I}_{iut})$. It checks whether after the execution of each *suspension* trace (*i.e.* trace that may contain the special output action δ) of the specification, the implementation produces only specified output actions or not. In the other hand, the ioc relation links a specification *Spec* with an implementation under test $\mathcal{I}_{iut}$, and is checks the same thing as ioco but for each *proper* trace (*i.e.* trace that does not contain any δ action) of the specification.

In this thesis, we have proposed an approach for symbolic test generation that is based on the IOSTS model and the ioc relation. However, this approach has some weaknesses. One of them is the problem of blockings that has not been addressed in the symbolic test generation part of the thesis. Nevertheless, this has been described in the introductory part (*see* page 42). In order to improve our symbolic test generation in this direction, one may attempt the following approach (published as the research report [Rusu et al., 2004]):

(1) Limit the IOSTS model to one that does not contain syntactic livelocks (cycles of internal actions) in it, and use the ioco relation [Tretmans, 1996a] as a correctness criterion instead of the ioc relation.

(2) For a given specification, construct its suspension IOSTS by encoding all potential blockings (outputlocks and deadlocks) of a given specification with the special output action δ.

Remark that the problem of blocking detection is undecidable for IOSTS in general. However, in the case of absentce of syntactic livelocks in an IOSTS $\mathcal{M}$, it is possible to construct syntactically the suspension IOSTS of $\mathcal{M}$.

The classical solution for the detection of unspecified blockings in an implementation under test during a test experiment, is to equip each tester with a timer indicating the amount of time that the tester has to wait for an output to occur. If the output does not occur before a certain amount of time, then the tester may conclude that this output will no longer occur. In this case, it decides (maybe wrongly) that the implementation is blocked.

We believe that the suggestions given in this paragraph will help to improve our symbolic test generation method.

Safety Properties. The report [Rusu et al., 2004] proposes an approach for combining verification and conformance testing techniques. In this report, the formal specification *Spec* of a given reactive system is modeled by IOSTS, and each safety property of *Spec* is represented in the same manner as a test purpose used in the symbolic test generation method. Then,

First, each property is *verified* on *Spec* using automatic techniques (*e.g.* abstract interpretation) that are sound but not necessary complete for the class of safety properties considered here.

Second, for each property, a test case is *automatically generated* from the specification and this property and is executed on a black-box implementation of the system.

If the verification step was successful, that is, it has established that the specification satisfies the properties, then the execution may detect the violation of the property by the implementation and the standard ioco conformance relation [Tretmans, 1996a] between implementation and specification. Otherwise, *i.e.* if the verification step did not conclude, that is, it did not allow to prove or to disprove the property, then the test execution may additionally detect a violation of the property by the specification.

Coverage Criteria. The STG tool (just as its predecessor TGV [Fernandez et al., 1996]) uses test purpose as the mechanism for test case selection. Each test purpose in STG is given under the form of a graph. This representation offers a software developer a natural way to (partially) describe the behaviors of a given specification to be tested. Nevertheless, the manual design of test purposes requires a good knowledge of the specification. This means that the process of the test purpose designing is still quite difficult for humans, in particular if a good coverage of the specification must be targeted. We believe that the classical structural coverage criteria [Rapps and Weyuker, 1985] combined with symbolic analysis may yield interesting test purposes.

Case Studies. So far, we have only generated test cases from more or less classical academic specifications of a coffee machine and the bounded retransmission protocol. The next important step is to apply our approach so as to test realistic industrial systems. We have already made the first steps in this direction: we generated some test cases for Common Electronic Purse System (CEPS) [CEPSCO, 2000]. This case study gave us the promising results published in [Clarke et al., 2001a]. Thus, it would be interesting to continue such experiments. We also believe that future case studies will contribute with essential information about the applicability of our symbolic test generation approach, and will help to improve our approach.

Appendix A

Appendix

A.1 Closure: Eliminating Internal Actions

Motivation. A test case should react promptly to all inputs from the implementation. A natural way to obtain this requirement is to make every location of a test case (except the verdict locations) input-complete. However, the possible inputs in some locations of the test case may be hidden by internal actions. For example, consider the location l_3 of the IOSTS $\mathcal{A}$ shown in Figure A.1 (*see* page 248). As you can see, $\mathcal{A}$ can execute from l_3 the internal actions τ_3 and τ_5, but the input action c can be executed only after τ_5. Then, if we make the location l_3 input-complete (*see* Definition 4.21, page 95), we obtain that the action c can also be executed before the internal action τ_5, which leads to modification of the system's behaviors. Thus, to make location l_3 input-complete we first have to eliminate τ_5. Therefore, in order to obtain the input-complete test case, all internal actions have to be eliminated from the synchronous product SP computed from a specification *Spec* and a test purpose TP of *Spec* on the previous step of the test generation method (*see* Section 7.2, page 173).

One more reason for elimination of internal actions from SP is that the resulting test case must be deterministic as a testing verdict must not depend on internal choices of the test case. Therefore, if the reader takes a close look at the definition of a deterministic IOSTS (*see* page 93), then he/she can see that the first requirement of being deterministic is to have the *empty* set of internal actions.

In the works of T. Jéron and his colleagues ([Jard and Jéron, 2002], [Jéron, 2004], *etc.*) on the test generation based on the IOLTS model, the operation of elimination of internal actions (called later *closure*) is incorporated directly into the determinization algorithm. However, in this thesis we prefer to separate the closure operation from the determinization algorithm as both of them are quite complex.

Sketch of the Section. The aim of the third step of the test generation method is to build visible behaviors of SP (*i.e.* to eliminate internal actions from SP) leaving the visible semantics of SP unchanged. However, the problem of the elimination of *all* internal actions from SP is *undecidable* in general due to the possible presents of syntactical livelocks in SP, (*see* explanations in Section A.1.2, page 257). This section proposes the closure operation that eliminates all internal actions from an IOSTS SP except of whose that are either directly leading to, or involved into, syntactic livelocks of SP. This operation is based on the collapsing of all sequences of internal actions described in the first subsection of the current section. Finally, we prove that the closure operation does not change the visible semantics of SP. In other words, we show that the IOSTS obtained after the closure algorithm is trace-equivalent to the given IOSTS SP.

A.1.1 The Collapsing Operation for a τ-Sequence

This section introduces the *collapsing operation* for a contiguous sequence of symbolic transitions, called τ-*sequence*, of some IOSTS. The aim of this operation is to encode the effect of such a sequence into one transition. It is important to have a clear understanding of the collapsing operation as it is the main operation used in Section A.1.3 in order to define the closure operation for IOSTS (*see* Definition A.7, page 260).

Before giving the formal definition of the collapsing operation we consider an IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$, where $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ is the alphabet of input, output and internal actions, and:

(1) call actions belonging to $(\Sigma \setminus \Sigma^\tau)$ *visible actions*, and

(2) denote a symbolic transition $t = \langle l, a, \pi, G, A, l' \rangle \in T$ by $l \overset{a}{\hookrightarrow} l'$.

Then, we introduce the notion of τ-sequence for the IOSTS $\mathcal{M}$. Intuitively, a τ-sequence is a *contiguous* sequence of symbolic transitions of the IOSTS $\mathcal{M}$ such that (1) all symbolic transitions involved into its maximal strict prefix are labeled with internal actions, and (2) the last transition of this sequence can be labeled with either a visible or internal action. Formally:

Definition A.1 (Set of τ-Sequences) Let $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ be an IOSTS. Then, the *set of* τ-*sequences*, denoted by $Sequences^\tau(\mathcal{M})$, is the set of all contiguous sequences of symbolic transitions:

$$\rho : \quad l \overset{\tau_1}{\hookrightarrow} l_2 \ldots l_{n-2} \overset{\tau_{n-1}}{\hookrightarrow} l_{n-1} \overset{a}{\hookrightarrow} l_n$$

where $n \geq 1$; $\quad l, l_1, \ldots, l_n \in L$; $\quad \tau_1, \ldots, \tau_{n-1} \in \Sigma^\tau$; $\quad$ and $a \in \Sigma$. □

Notice that τ-sequences are prefix closed. This observation follows directly from Definition A.1, and it is formally stated below.

Observation A.1 (Prefix Closure of τ-Sequence) Any prefix of a τ-sequence, whose length is greater then zero, is also a τ-sequence. □

For better understanding of the notion of τ-sequence we propose the following example.

Example A.1 Consider the IOSTS $\mathcal{A}$ shown in Figure A.1 (*see* page 248). To illustrate the notion of τ-sequences we consider the following sequences of symbolic

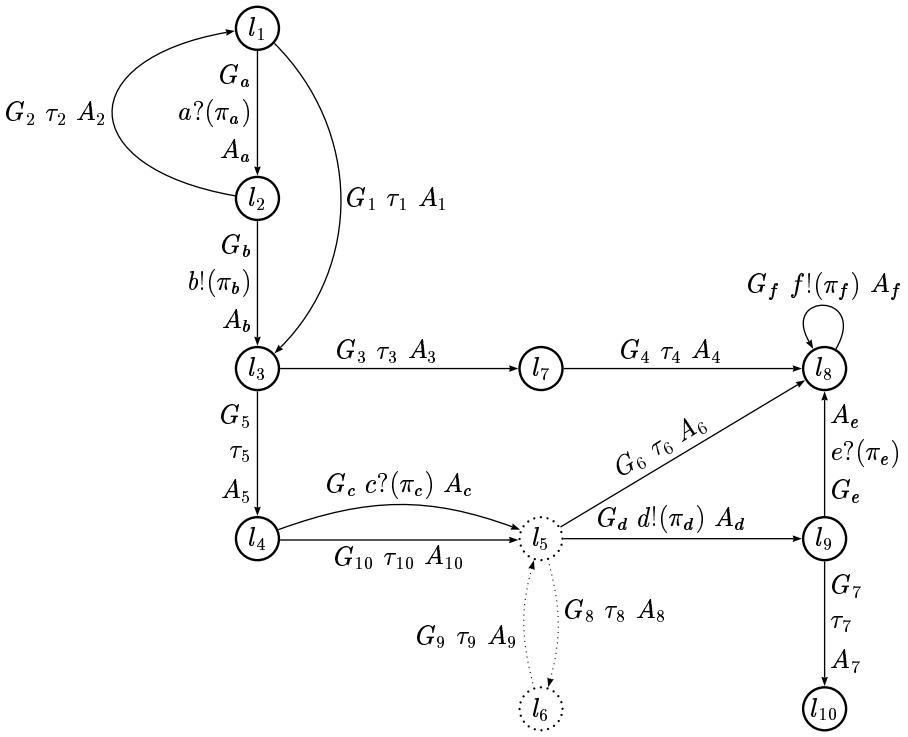


Figure A.1: The IOSTS $\mathcal{A}$ before the closure.

transition of $\mathcal{A}$:

$$\begin{aligned}
&l_1 \overset{a}{\hookrightarrow} l_2 && \in \mathit{Sequences}^{\tau}(\mathcal{A}),\\
&l_1 \overset{\tau_1}{\hookrightarrow} l_3 \overset{\tau_5}{\hookrightarrow} l_4 && \in \mathit{Sequences}^{\tau}(\mathcal{A}),\\
&l_5 \overset{\tau_8}{\hookrightarrow} l_6 \overset{\tau_9}{\hookrightarrow} l_5 \overset{d}{\hookrightarrow} l_9 && \in \mathit{Sequences}^{\tau}(\mathcal{A}),\\
&l_3 \overset{\tau_5}{\hookrightarrow} l_4 \overset{c}{\hookrightarrow} l_5 \overset{\tau_6}{\hookrightarrow} l_8 \overset{f}{\hookrightarrow} l_8 && \notin \mathit{Sequences}^{\tau}(\mathcal{A})
\end{aligned}$$

The first three sequences respect the Definition A.1 (*see* page 247), and therefore, belong to $\mathit{Sequences}^{\tau}(\mathcal{A})$. However, the last one does not belong to $\mathit{Sequences}^{\tau}(\mathcal{A})$, as its second symbolic transition, which belongs to the strict maximal prefix of this sequence, is labeled with visible action c, which contradicts the definition of τ-sequences.

The last important thing to notice is that the set of τ-sequences for $\mathcal{A}$ is infinite as $\mathcal{A}$ contains the cycle of internal actions $l_5 \overset{\tau_8}{\hookrightarrow} l_6 \overset{\tau_9}{\hookrightarrow} l_5$. (Such cycles of internal actions are called syntactic livelocks and are formally defined in the next section.) □

Next, we remind that the purpose of this section is to introduce the operation which transforms any τ-sequence into one single symbolic transition keeping its effect. This operation is called *collapsing operation* and defined as follows:

Definition A.2 (Collapsing Operation) Let

$$\rho: \quad \underbrace{l \overset{\tau_1}{\hookrightarrow} l_2}_{t_1} \ldots \underbrace{l_{n-2} \overset{\tau_{n-1}}{\hookrightarrow} l_{n-1}}_{t_{n-1}} \underbrace{\overset{a}{\hookrightarrow} l_n}_{t_n}$$

be a τ-sequence of an IOSTS $\mathcal{M}$, *i.e.* $\rho \in \mathit{Sequences}^{\tau}(\mathcal{M})$, such that each symbolic transition t_i $(i = 1..n)$ of ρ has a guard G_i and assignments A_i. Then, the *collapsing operation* transforms ρ into the single symbolic transition:

$$\mathsf{collapse}(\rho): \quad l \overset{a}{\hookrightarrow} l_n$$

with the guard $G = G_1 \wedge (G_2 \circ A_1) \wedge \ldots \wedge (G_n \circ A_{n-1} \circ \ldots \circ A_1)$ and the assignments $A = A_n \circ A_{n-1} \circ \ldots \circ A_2 \circ A_1$. □

It is important to notice that, as the definition of collapsing operation is given at the syntactic level, the guards and assignments are treated as expressions (not functions) on variables, symbolic constants and corresponding parameters. Therefore, the composition operation $\circ$ between them is implemented as a syntactic substitution.

For instance, consider two symbolic transition t' and t'' of an IOSTS $\mathcal{M}$ with set of variables V. Assume that t' is labeled with an *internal* action which by Definition 4.2 (*see* page 81) does not carry any parameter, and t'' is labeled with an arbitrary action carrying a tuple π (possibly empty) of parameters. Assume also that $A' = \{v := A'_v \mid v \in V\}$ and $A'' = \{v := A''_v \mid v \in V\}$ are the assignments of t' and t'' respectively, and G'' is the guard of t''. Then, $A'' \circ A'$ is the set of assignments $\{v := A''_v(\tilde{v}/A'_{\tilde{v}}) \mid v \in V \wedge \tilde{v} \in V\}$, where every occurrence of a variable $\tilde{v}$ in the right-hand side of the assignments A''_v has been replaced by the expression $A'_{\tilde{v}}$. Similarly, $G'' \circ A'$ is the guard obtained by substituting, for every $v \in V$, all the occurrences of v in G'' by the corresponding expression A'_v. Notice that, $A'' \circ A'$ and $G'' \circ A'$ depend on variables, symbolic constants of $\mathcal{M}$ and parameters π carried by the action labeling the symbolic transition t''.

To illustrate the collapsing operation defined above, we consider the following example.

Example A.2 As we know from Example A.1 (*see* page 247), the following contiguous sequence of symbolic transitions:

$$\rho_{\mathcal{A}}: \quad \underbrace{l_5 \overset{\tau_8}{\hookrightarrow} l_6}_{t_8} \overbrace{\overset{\tau_9}{\hookrightarrow} l_5}^{t_9} \underbrace{\overset{d}{\hookrightarrow} l_9}_{t_d}$$

is a τ-sequence of the IOSTS $\mathcal{A}$ shown in Figure A.1 (*see* page 248). Suppose that the symbolic transitions t_8, t_9 and t_d of $\rho_{\mathcal{A}}$ have the following guards and assignments:

Symbolic Transition	**Guard**	**Assignments**
t_8	$G_8 : v_1 \geq 0$	$A_8 : \{v_1 := v_1 - 2; \quad v_2 := 1\}$
t_9	$G_9 : v_1 + v_2 \geq 0$	$A_9 : \{v_1 := 0; \quad v_2 := v_1 + v_2\}$
t_d	$G_d : p \geq v_2$	$A_d : \{v_1 := v_2; \quad v_2 := p - v_2\}$

where v_1, v_2 are the variables of $\mathcal{A}$ and p is the parameter carried by the visible action d labeling the symbolic transition t_d.

Then, by applying the collapsing operation (*see* Definition A.2, page 249) to τ-sequence $\rho_{\mathcal{A}}$, we obtain the symbolic transition $\mathsf{collapse}(\rho_{\mathcal{A}}) : \quad l_5 \overset{d}{\hookrightarrow} l_9$ such that:

- the output action d carrying the same parameter p as the action d labeling the symbolic transition t_d,
- the guard G is equal to $\underbrace{(v_1 \geq 0)}_{G_8} \wedge \underbrace{((v_1 - 2) + 1 \geq 0)}_{G_9 \circ A_8} \wedge \underbrace{(p \geq (v_1 - 2) + 1)}_{G_d \circ A_9 \circ A_8}$, where $A_9 \circ A_8 : \{v_1 := 0; \quad v_2 := (v_1 - 2) + 1\}$, and

– the assignments $A = A_d \circ A_9 \circ A_8$ is $\{v_1 := (v_1 - 2) + 1; \quad v_2 = p - (v_1 - 2) + 1\}$.

□

Thus, at this point of the section we know how to syntactically collapse any τ-sequence ρ of an IOSTS $\mathcal{M}$ into a single symbolic transition $\mathsf{collapse}(\rho)$. The next step is to show that the effect of the symbolic transition $\mathsf{collapse}(\rho)$ is equivalent to that of ρ, *i.e.* we move from the syntactic level to the semantic one.

Notice that at the semantic level, guards and assignments of symbolic transitions of $\mathcal{M}$ are interpreted as functions. More precisely, let t be a symbolic transition with an action a carrying a tuple π of parameters, a guard G and assignments A. Then G can be identified with a function from valuations of variables V, symbolic constants C and parameters π to Booleans, *i.e.* $G : \mathsf{DOM}(V \cup C \cup \pi) \mapsto \{true, false\}$; and assignments can be identified with a function from valuations of V, C and π to valuations of V and C, *i.e.* $A : \mathsf{DOM}(V \cup C \cup \pi) \mapsto \mathsf{DOM}(V \cup C)$. Moreover, the composition $\circ$ between either two assignments, or a guard and assignments, at the semantic level is interpreted as the standard composition between functions.

For example, consider two symbolic transition t_1 and t_2 of an IOSTS $\mathcal{M}$ with set of variables V and set of constants C. Assume that t_1 is labeled with an *internal* action which by Definition 4.2 (*see* page 81) does not carry any parameters, and t_2 is labeled with an arbitrary action carrying a tuple π (possibly empty) of parameters. Then, for assignments A_1 and A_2 of t_1 and t_2 respectively, the assignments $A_2 \circ A_1$ denotes the function that associates, to valuations $\vartheta \in \mathsf{DOM}(V \cup C)$ and $\omega \in \mathsf{DOM}(\pi)$ the valuation of variables and symbolic constants $A_2(\langle A_1(\vartheta), \omega \rangle)$. Also, an assignment A_1 and a guard G_2 of a symbolic transitions t_1 and t_2 respectively, can be composed together, *i.e.* $G_2 \circ A_1$ denotes the function that associates, to valuations $\vartheta \in \mathsf{DOM}(V \cup C)$ and $\omega \in \mathsf{DOM}(\pi)$, the Boolean value $G_2(\langle A_1(\vartheta), \omega \rangle)$.

Next, we prove that for any τ-sequence ρ of an IOSTS $\mathcal{M}$, the effect of the symbolic transition $\mathsf{collapse}(\rho)$ is equivalent to that of ρ. Before showing this, we define the composition, denoted as $\circ$, between two local transition relations of $\mathcal{M}$.

Definition A.3 (Composition of Local Transition Relations) For two local transitions relations $\rightarrow_{t_1} \subseteq S \times \Lambda^\tau \times S$ and $\rightarrow_{t_2} \subseteq S \times \Lambda \times S$, where S is a set of states, Λ is a set of valued actions and $\Lambda^\tau \subseteq \Lambda$ is a set of internal actions, *the composition of* $\rightarrow_{t_1}$ and $\rightarrow_{t_2}$, denoted as $\rightarrow_{t_2} \circ \rightarrow_{t_1}$, is the set:

$$\{\langle s_1, \alpha, s_2 \rangle \in S \times \Lambda \times S \mid \exists\ s' \in S, \alpha' \in \Lambda^\tau\ .\ [\langle s_1, \alpha', s' \rangle \in \rightarrow_{t_1}\ \wedge\ \langle s', \alpha, s_2 \rangle \in \rightarrow_{t_2}]\}$$

□

Theorem A.1 (Preserving the Effect of τ-Sequences by the Collapsing Operation) Let

$$\rho: \quad \underbrace{l \overset{\tau_1}{\hookrightarrow} l_2}_{t_1} \ldots \underbrace{l_{n-2} \overset{\tau_{n-1}}{\hookrightarrow} l_{n-1}}_{t_{n-1}} \underbrace{\overset{a}{\hookrightarrow} l_n}_{t_n}$$

be a τ-sequence of an IOSTS $\mathcal{M}$, *i.e.* $\rho \in \mathit{Sequences}^{\tau}(\mathcal{M})$. Also, let $\rightarrow_{t_i}$ $(i = 1..n)$ and $\rightarrow_{\mathsf{collapse}(\rho)}$ be the local transition relations corresponding to the symbolic transition t_i of ρ and to the symbolic transition $\mathsf{collapse}(\rho)$. Then,

$$\rightarrow_{\mathsf{collapse}(\rho)} \quad = \quad (\rightarrow_{t_n} \circ (\rightarrow_{t_{n-1}} \circ \ldots \circ (\rightarrow_{t_2} \circ \rightarrow_{t_1}) \ldots))$$

□

Proof The proof is done by induction on the length of the τ-sequence $\rho \in \mathit{Sequences}^{\tau}(\mathcal{M})$.

Induction Basis. Consider a τ-sequence $\rho: \underbrace{l \overset{a}{\hookrightarrow} l_1}_{t_1} \in \mathit{Sequences}^{\tau}(\mathcal{M})$ of length one, where the symbolic transition t_1 has the guard G_1 and the assignments A_1. Then trivially $\rightarrow_{\mathsf{collapse}(\rho)} \quad = \quad \rightarrow_{t_1}$ (*see* Definition A.2, page 249).

Induction Hypothesis. Assume that for the τ-sequence

$$\rho': \quad \underbrace{l \overset{\tau_1}{\hookrightarrow} l_1}_{t_1} \ldots \underbrace{l_{n-2} \overset{a}{\hookrightarrow} l_{n-1}}_{t_{n-1}} \quad \in \mathit{Sequences}^{\tau}(\mathcal{M})$$

which has length $n-1$, the equality shown below holds.

$$\rightarrow_{\mathsf{collapse}(\rho')} \quad = \quad (\rightarrow_{t_{n-1}} \circ (\ldots \circ (\rightarrow_{t_2} \circ \rightarrow_{t_1}) \ldots))$$

Induction Step. Consider a τ-sequence of length n

$$\rho: \quad \underbrace{l \overset{\tau_1}{\hookrightarrow} l_1 \ldots l_{n-2} \overset{\tau_{n-1}}{\hookrightarrow} l_{n-1}}_{\rho'} \underbrace{\overset{a}{\hookrightarrow} l_n}_{t_n} \quad \in \mathit{Sequences}^{\tau}(\mathcal{M}) \tag{A.1}$$

and prove that the equality below holds.

$$\rightarrow_{\mathsf{collapse}(\rho)} \quad = \quad (\rightarrow_{t_n} \circ (\rightarrow_{t_{n-1}} \circ \ldots \circ (\rightarrow_{t_2} \circ \rightarrow_{t_1}) \ldots))$$

Due to Observation A.1 (*see* page 247), the maximal strict prefix ρ' of the given τ-sequence ρ (*see* Formula (A.1), page 252) is also a τ-sequence of $\mathcal{M}$. Thus, ρ' can be collapsed according to Definition A.2 (*see* page 249)

to the symbolic transition $\mathsf{collapse}(\rho') : l \overset{\tau_{n-1}}{\hookrightarrow} l_{n-1}$ with the guard $G' = G_1 \wedge (G_2 \circ A_1) \wedge \ldots \wedge (G_{n-1} \circ A_{n-2} \circ \ldots \circ A_1)$ and the assignments $A' = A_{n-1} \circ A_{n-2} \circ \ldots \circ A_1$. Consequently, the τ-sequence ρ (*see* Formula (A.1), page 252) can be equivalently represented as:

$$\rho : \quad \underbrace{l \overset{\tau_{n-1}}{\hookrightarrow} \overbrace{l_{n-1}}^{t_n} \overset{a}{\hookrightarrow} l_n}_{\mathsf{collapse}(\rho')} \tag{A.2}$$

Therefore, to prove the induction step it is enough to show the equality:

$$\rightarrow_{\mathsf{collapse}(\rho)} \quad = \quad \rightarrow_{t_n} \circ \rightarrow_{\mathsf{collapse}(\rho')} \tag{A.3}$$

Notice that $\mathsf{collapse}(\rho)$ is the syntactic transition $l \overset{a}{\hookrightarrow} l_n$ with the guard $G = G_1 \wedge (G_2 \circ A_1) \wedge \ldots \wedge (G_n \circ A_{n-1} \circ \ldots \circ A_1) = G' \wedge (G_n \circ A')$, and the assignments $A = A_n \circ A_{n-1} \circ \ldots \circ A_1 = A_n \circ A'$.

($\subseteq$) First, we prove that for all triples $\langle \underbrace{\langle l, \vartheta \rangle}_{s}, \underbrace{\langle a, \omega \rangle}_{\alpha_n}, \underbrace{\langle l_n, \vartheta_n \rangle}_{s_n} \rangle$ belonging to $\rightarrow_{\mathsf{collapse}(\rho)}$, there exist two triples $\langle \underbrace{\langle l, \vartheta \rangle}_{s}, \underbrace{\langle \tau_{n-1}, \langle\rangle \rangle}_{\alpha_{n-1}}, \underbrace{\langle l_{n-1}, \vartheta_{n-1} \rangle}_{s_{n-1}} \rangle$ and $\langle \underbrace{\langle l_{n-1}, \vartheta_{n-1} \rangle}_{s_{n-1}}, \underbrace{\langle a, \omega \rangle}_{\alpha_n}, \underbrace{\langle l_n, \vartheta_n \rangle}_{s_n} \rangle$ belonging respectively to $\rightarrow_{\mathsf{collapse}(\rho')}$ and $\rightarrow_{t_n}$. Here, ϑ, ϑ_{n-1} and ϑ_n are valuations of variables and symbolic constants of the IOSTS $\mathcal{M}$; and ω is a valuation of parameters π carried by the visible action a.

(1) As $\langle \underbrace{\langle l, \vartheta \rangle}_{s}, \underbrace{\langle a, \omega \rangle}_{\alpha_n}, \underbrace{\langle l_n, \vartheta_n \rangle}_{s_n} \rangle$ belongs to the local transition relation $\rightarrow_{\mathsf{collapse}(\rho)}$, then, due to Definition 4.7 (*see* page 86), we obtain that:
   (a) the pair of valuations $\langle \vartheta, \omega \rangle$ satisfies the guard $G = G' \wedge (G_n \circ A')$ of $\mathsf{collapse}(\rho)$, and
   (b) the valuation ϑ_n is obtained from the pair of valuations $\langle \vartheta, \omega \rangle$ by the assignments A, *i.e.* $\vartheta_n = A(\langle \vartheta, \omega \rangle)$.

(2) Consider the state $s_{n-1} = \langle l_{n-1}, \vartheta_{n-1} \rangle$, where l_{n-1} is the target of the symbolic transition $\mathsf{collapse}(\rho')$, and ϑ_{n-1} is the valuation of variables and symbolic constants of $\mathcal{M}$ obtained from ϑ using the assignment A' of $\mathsf{collapse}(\rho')$. (Notice that we did not mention the valuation of parameters, as the symbolic transition $\mathsf{collapse}(\rho')$ is labeled with the internal action τ_{n-1} which, due to Definition 4.2 (*see* page 81), does not carry any parameters.) Then,

(a) $\langle \underbrace{\langle l, \vartheta \rangle}_{s}, \underbrace{\langle \tau_{n-1}, \langle\rangle \rangle}_{\alpha_{n-1}}, \underbrace{\langle l_{n-1}, \vartheta_{n-1} \rangle}_{s_{n-1}} \rangle \quad \in \quad \rightarrow_{\mathsf{collapse}(\rho')}$ due to the following facts:

- l and l_{n-1} are respectively the origin and the target of $\mathsf{collapse}(\rho')$,
- τ_{n-1} is the internal action labeling $\mathsf{collapse}(\rho')$ and carrying the empty tuple of parameters,
- ϑ satisfies the guard G' of $\mathsf{collapse}(\rho')$ because:
  - G' does not depend on any parameters as $\mathsf{collapse}(\rho')$ is labeled with the internal action τ_{n-1} that does not carry any messages (*see* Definition 4.2, page 81), and
  - $\langle \vartheta, \omega \rangle \models G' \wedge (G_n \circ A')$ (*see* the item (1.a) above).
- $\vartheta_{n-1} = A'(\vartheta)$ (*see* the beginning of the item (2)).

(b) $\langle \underbrace{\langle l_{n-1}, \vartheta_{n-1} \rangle}_{s_{n-1}}, \underbrace{\langle a, \omega \rangle}_{\alpha_n}, \underbrace{\langle l_n, \vartheta_n \rangle}_{s_n} \rangle \quad \in \quad \rightarrow_{t_n}$ due to the facts:

- l_{n-1} and l_n are respectively the origin and the target of t_n (*see* Formula (A.2), page 253),
- a is the visible action labeling the symbolic transition t_1 and carrying the tuple π of parameters,
- $\langle \vartheta_{n-1}, \omega \rangle$ satisfies the guard G_n of t_n. Indeed, from the item (1.a) above we know that $\langle \vartheta, \omega \rangle \models G' \wedge (G_n \circ A')$. Therefore:

$$
\begin{array}{l}
[\langle \vartheta, \omega \rangle \models G_n \circ A'] \\
\iff \\
[G_n(A'(\langle \vartheta, \omega \rangle)) \text{ evaluates to } \mathit{true}] \\
\iff \quad \text{[} A' \textit{ does not depend on the parameters as the first symbolic transition } \mathsf{collapse}(\rho') \textit{ of } \rho \textit{ (see Formula (A.2), page 253) is labeled with internal action } \tau_{n-1} \textit{ which does not carry any parameters due to Definition 4.2 (see page 81).} \text{]} \\
[G_n(\langle A'(\vartheta), \omega \rangle) \text{ evaluates to } \mathit{true}] \\
\iff \\
[\langle A'(\vartheta), \omega \rangle \models G_n] \\
\iff \quad \text{[} A'(\vartheta) = \vartheta_{n-1} \textit{ due to Definition 4.7 (see page 86).} \text{]} \\
[\langle \vartheta_{n-1}, \omega \rangle \models G_n]
\end{array}
$$

- ϑ_n is obtained from the pair of valuations $\langle \vartheta_{n-1}, \omega \rangle$ by the assignments A_n. Indeed, from the item (1.b) we know that $\vartheta_n = A(\langle \vartheta, \omega \rangle)$. As the set of assignments A is equal to $A_n \circ A'$, then $A(\langle \vartheta, \omega \rangle) = (A_n \circ A')(\langle \vartheta, \omega \rangle)$. The latter can

be equally represented, using the standard definition of the functions composition, as $A_n(A'(\langle \vartheta, \omega \rangle))$. Next, notice that A' does not depend on any parameters as it is the assignments of the symbolic transition $\mathsf{collapse}(\rho')$ that is labeled with internal action τ_{n-1} which due to Definition 4.2 (*see* page 81) does not carry parameters. Then, $A_n(A'(\langle \vartheta, \omega \rangle))$ can be rewritten as $A_n(\langle A'(\vartheta), \omega \rangle)$. Finally, as $\vartheta_{n-1} = A'(\vartheta)$ (*see* the item (2.a)), then $A_n(\langle A'(\vartheta), \omega \rangle) = A_n(\langle \vartheta_{n-1}, \omega \rangle)$. Therefore, $\vartheta_n = A_n(\langle \vartheta_{n-1}, \omega \rangle)$.

Finally, the items (2.a) and (2.b) together with Definition A.2 (*see* page 249) imply that the triple $\langle s, \alpha_n, s_n \rangle$ belongs to $\rightarrow_{t_n} \circ \rightarrow_{\mathsf{collapse}(\rho')}$. Therefore, the inclusion $\rightarrow_{\mathsf{collapse}(\rho)} \subseteq \rightarrow_{t_n} \circ \rightarrow_{\mathsf{collapse}(\rho')}$ is proved.

($\supseteq$) We prove that all triples $\langle \underbrace{\langle l, \vartheta \rangle}_{s}, \underbrace{\langle a, \omega \rangle}_{\alpha_n}, \underbrace{\langle l_n, \vartheta_n \rangle}_{s_n} \rangle$ belonging to $\rightarrow_{t_n} \circ \rightarrow_{\mathsf{collapse}(\rho')}$, also belong to $\rightarrow_{\mathsf{collapse}(\rho)}$. Here, ϑ and ϑ_n are valuations of variables and symbolic constants of the IOSTS $\mathcal{M}$; and ω is a valuation of parameters π carried by the visible action a.

First, as we know that $\langle \underbrace{\langle l, \vartheta \rangle}_{s}, \underbrace{\langle a, \omega \rangle}_{\alpha_n}, \underbrace{\langle l_n, \vartheta_n \rangle}_{s_n} \rangle$ belongs to $\rightarrow_{t_n} \circ \rightarrow_{\mathsf{collapse}(\rho')}$, then by Definition A.3 (*see* page 251) there exist the state $s_{n-1} = \langle l_{n-1}, \vartheta_{n-1} \rangle \in S$ and the valued action $\alpha_{n-1} = \langle \tau_{n-1}, \langle\rangle \rangle \in \Lambda^\tau$ such that $\langle s, \alpha_{n-1}, s_{n-1} \rangle \in \rightarrow_{\mathsf{collapse}(\rho')}$ and $\langle s_{n-1}, \alpha_n, s_n \rangle \in \rightarrow_{t_n}$. From

$$\langle \underbrace{\langle l, \vartheta \rangle}_{s}, \underbrace{\langle \tau_{n-1}, \langle\rangle \rangle}_{\alpha_{n-1}}, \underbrace{\langle l_{n-1}, \vartheta_{n-1} \rangle}_{s_{n-1}} \rangle \in \rightarrow_{\mathsf{collapse}(\rho')},$$
$$\langle \underbrace{\langle l_{n-1}, \vartheta_{n-1} \rangle}_{s_{n-1}}, \underbrace{\langle a, \omega \rangle}_{\alpha_n}, \underbrace{\langle l_n, \vartheta_n \rangle}_{s_n} \rangle \in \rightarrow_{t_n}$$

and Definition 4.7 (*see* page 86) we obtain:

(a) ϑ satisfies the guard G' of $\mathsf{collapse}(\rho')$,

(b) $\vartheta_{n-1} = A'(\vartheta)$, where A' are the assignments of $\mathsf{collapse}(\rho')$,

(c) $\langle \vartheta_{n-1}, \omega \rangle$ satisfies the guard G_n of t_n. This means:

$$
\begin{array}{l}
[\langle \vartheta_{n-1}, \omega \rangle \models G_n] \\
\Longleftrightarrow \\
[\langle A'(\vartheta), \omega \rangle \models G_n] \\
\Longleftrightarrow \\
[A'(\langle \vartheta, \omega \rangle) \models G_n] \\
\Longleftrightarrow \\
[G_n(A'(\langle \vartheta, \omega \rangle)) \text{ evaluates to } \mathit{true}] \\
\Longleftrightarrow \\
[\langle \vartheta, \omega \rangle \models G_n \circ A']
\end{array}
$$

Moreover, as we know that $\vartheta \models G'$ (*see* the item (a) above), we obtain $\langle \vartheta, \omega \rangle \models G' \wedge (G_n \circ A')$.

(d) $\vartheta_n = A_n(\langle \vartheta_{n-1}, \omega \rangle) = A_n(\langle A'(\vartheta), \omega \rangle) = A_n(A'(\langle \vartheta, \omega \rangle)) = (A_n \circ A')(\langle \vartheta, \omega \rangle)$.

Therefore, the triple $\langle \underbrace{\langle l, \vartheta \rangle}_{s}, \underbrace{\langle a, \omega \rangle}_{\alpha_n}, \underbrace{\langle l_n, \vartheta_n \rangle}_{s_n} \rangle$ has the following properties:

- the locations l and l_n are respectively the origin and the target of the symbolic transition $\mathsf{collapse}(\rho)$ (*see* Formula (A.2), page 253),
- a is the action labeling $\mathsf{collapse}(\rho)$ and carrying the tuple π of parameters,
- the pair of valuations $\langle \vartheta, \omega \rangle$ satisfies the guard $G = G' \wedge (G_n \circ A')$ of $\mathsf{collapse}(\rho)$ (*see* the item (c) above), and
- the valuation ϑ_n is obtained from $\langle \vartheta, \omega \rangle$ by applying the assignments $A = (A_n \circ A')$ of $\mathsf{collapse}(\rho)$ (*see* the item (d) above).

Thus, due to the definition about the local transition relation of IOSTS (*see* page 86) we conclude that the triple $\langle s, \alpha_n, s_n \rangle$ belongs to $\rightarrow_{\mathsf{collapse}(\rho)}$. Therefore, the inclusion $\rightarrow_{\mathsf{collapse}(\rho)} \supseteq \rightarrow_{t_n} \circ \rightarrow_{\mathsf{collapse}(\rho')}$ is proved.

The items ($\subseteq$) and ($\supseteq$) prove Equality (A.3) (*see* page 253). Finally, using Equality (A.3) (*see* page 253) together with the induction hypothesis, we prove the induction step and the whole theorem.

Q.E.D.

A.1.2 Syntactic Livelocks in IOSTS

Before giving a short introduction for this section, we remind that the purpose of Section A.1 is to define the syntactical operation for an IOSTS $\mathcal{M}$ that eliminates *all* internal actions from $\mathcal{M}$ leaving visible semantics of $\mathcal{M}$ unchanged. Such an operation can be defined only under condition of absence of cycles of internal actions during which the system performs internal computations without communicating with its environment. To illustrate this statement we give an example.

Example A.3 Consider the IOSTS $\mathcal{A}$ modeling the system which generates a multiple of three. It is easy to see that this IOSTS has an infinite number of

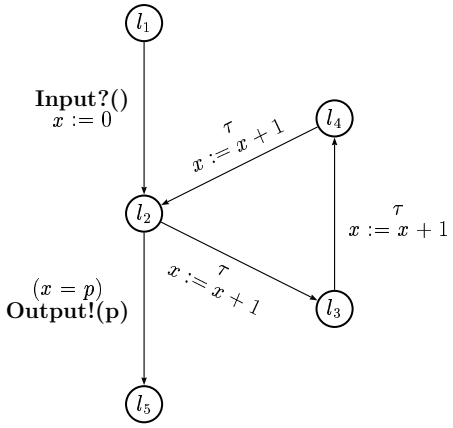


Figure A.2: The IOSTS $\mathcal{A}$ which is modeling mutiplication by three.

τ-sequences, *i.e.* $Sequences^{\tau}(\mathcal{A}) = \{Input; (\tau)^*; (\tau\tau\tau)^* Output\}$. Therefore, in order to eliminate all internal actions from $\mathcal{A}$, the collapsing operation has to be applied infinitely many times. Due to these arguments, we limit our research to IOSTS which do not contain any syntactic livelocks, and consequently which have a finite number of τ-sequences (*see* Theorem A.2, page 261). □

This section presents the formal definition of a cycle of internal actions which is called later as *syntactic livelock*.

Definition A.4 (Syntactic Livelock) A *syntactic livelock* is a sequence of symbolic transitions (1) which begins and ends in the same location, (2) which does not go through the same location twice (except the location with which this sequence begins and ends), and (3) each symbolic transition of this sequence is labeled with internal action, *i.e.*

$$l_1 \overset{\tau_1}{\hookrightarrow} l_2 \ldots l_{n-1} \overset{\tau_{n-1}}{\hookrightarrow} l_1$$

where (1) $l_1, l_2, \ldots, l_{n-1} \in L$, (2) for all i and j from 1 to $n-1$, if $i \neq j$ then $l_i \neq l_j$, and (3) $\tau_1, \ldots, \tau_{n-1} \in \Sigma^\tau$. □

We illustrate the notion of syntactic livelock with an example below.

Example A.4 Let us consider the IOSTS $\mathcal{A}$ depicted in Figure A.1. Consider also the sequence of symbolic transition $l_5 \overset{\tau_8}{\hookrightarrow} l_6 \overset{\tau_9}{\hookrightarrow} l_5$. This sequence is the syntactic livelock of $\mathcal{A}$ as (1) it begins and ends in the same location l_5, (2) the intermediate location l_6 does not appear in this sequence more then ones, and (3) all symbolic transitions involved into the sequence are labeled with internal actions τ_8 and τ_9. □

A.1.3 Closure for IOSTS without Syntactic Livelocks

The section introduces the closure operation for IOSTS that do not contain any syntactic livelocks defined in Section A.1.2 (*see* page 257). The purpose of this operation is to eliminate *all* internal actions from an IOSTS by computing the effect of any τ-sequence that ends with a symbolic transition t labeled with a *visible* action, and by encoding it in t. The section ends with a proof that an IOSTS obtained after the closure operation is well-formed.

Before defining the closure operation, it is necessary to introduce two new notations concerning an IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$. The first one is the set $\mathsf{post}_{\Sigma'}(L')$ of locations that can be directly reached from a location belonging to some subset L' of L by an action from some subset Σ' of Σ. This notion is used in the closure operation to define the set of locations for the resulting IOSTS $\mathsf{closure}(\mathcal{M})$. The second new notion is the set of $\tau.a$-sequences for $\mathcal{M}$. This set contains all τ-sequences (*see* Definition A.1, page 247) of $\mathcal{M}$ ending with a symbolic transition labeled with a visible action of $\mathcal{M}$. It is used to identify all τ-sequences of $\mathcal{M}$ that must be collapsed during the closure operation. The formal definitions of $\mathsf{post}_{\Sigma'}(L')$ and the set of $\tau.a$-sequences are given below. For better understanding these definitions are supported with examples.

Definition A.5 ($\mathsf{post}_{\Sigma'}(L')$) Let $\mathcal{M}$ be an IOSTS with set of locations L and alphabet of actions Σ. Let also L' be a subset of L and Σ' be a subset of Σ. Then, the set:

$$\mathsf{post}_{\Sigma'}(L') = \{\ l \in L \quad | \quad \exists\ a \in \Sigma', l' \in L'\ .\ [l' \overset{a}{\hookrightarrow} l]\ \}$$

is the set of locations that are the successors of locations belonging to L' by an action $a \in \Sigma'$. □

Example A.5 Consider the IOSTS $\mathcal{A}$ with set of locations $L_{\mathcal{A}} = \{l_1, l_2, l_3, l_4, l_5, l_7, l_8, l_9\}$ depicted in Figure A.1 (*see* page 248). Notice that for this example we do not take into account the symbolic transitions of $\mathcal{A}$ which are shown as dotted lines in Figure A.1 (*see* page 248) as they correspond to a syntactic livelock.

Then, we compute the set of successors of the locations $L_{\mathcal{A}}$ by some visible action belonging to $(\Sigma_{\mathcal{A}} \setminus \Sigma^{\tau}_{\mathcal{A}}) = \{a, b, c, d, e, f\}$ as follows:

Location	**Action**	**Successor**
l_1	a	l_2
l_2	b	l_3
l_3	—	—
l_4	c	l_5
l_5	d	l_9
l_7	—	—
l_8	f	l_8
l_9	e	l_8

Thus, we obtain $\mathsf{post}_{(\Sigma_{\mathcal{A}} \setminus \Sigma^{\tau}_{\mathcal{A}})}(L_{\mathcal{A}}) = \{l_2, l_3, l_5, l_8, l_9\}$. □

Next, as it has been announced at the beginning of the section, we formally define the notion of $\tau.a$-sequences.

Definition A.6 (Set of $\tau.a$-Sequences) For an IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ and a location $l \in L$, the *set of $\tau.a$-sequences* for l, denoted by $\mathit{Sequences}^{\tau.a}(l)$, is the set of all sequences of symbolic transitions starting in the location l

$$\rho: \quad l \overset{\tau_1}{\hookrightarrow} l_1 \ldots l_{n-2} \overset{\tau_{n-1}}{\hookrightarrow} l_{n-1} \overset{a}{\hookrightarrow} l_n$$

where $n \geq 2$, $\quad l, l_1, \ldots, l_n \in L$, $\quad \tau_1, \ldots, \tau_{n-1} \in \Sigma^{\tau}$, $\quad$ and $a \in (\Sigma \setminus \Sigma^{\tau})$. □

It is not hard to see that the definitions of τ-sequences and $\tau.a$-sequences (*see* Definitions A.1 and A.6, pages 247 and 259) lead to the following observation:

Observation A.2 Let $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ be an IOSTS. Then, for each location $l \in L$, all $\tau.a$-sequences starting in l are τ-sequences of $\mathcal{M}$, *i.e.*

$$\forall\, l \in L \;.\; [\mathit{Sequences}^{\tau.a}(l) \subseteq \mathit{Sequences}^{\tau}(\mathcal{M})]$$

□

We illustrate the definition about $\tau.a$-sequences with an example.

Example A.6 Consider the IOSTS $\mathcal{A}$ shown in Figure A.1 (*see* page 248) without syntactic livelocks, *i.e.* without transitions shown as the dotted lines. Then we compute $\tau.a$-sequences that start, for instance, in the locations l_1 and l_9 of $\mathcal{A}$.

(1) For the location l_1 the set $\mathit{Sequences}^{\tau.a}(l_1)$ consists of the four following $\tau.a$-sequences:

$$\left\{ \begin{array}{l} l_1 \overset{\tau_1}{\hookrightarrow} l_3 \overset{\tau_5}{\hookrightarrow} l_4 \overset{c}{\hookrightarrow} l_5, \\ l_1 \overset{\tau_1}{\hookrightarrow} l_3 \overset{\tau_5}{\hookrightarrow} l_4 \overset{\tau_{10}}{\hookrightarrow} l_5 \overset{d}{\hookrightarrow} l_9, \\ l_1 \overset{\tau_1}{\hookrightarrow} l_3 \overset{\tau_5}{\hookrightarrow} l_4 \overset{\tau_{10}}{\hookrightarrow} l_5 \overset{\tau_6}{\hookrightarrow} l_8 \overset{f}{\hookrightarrow} l_8, \\ l_1 \overset{\tau_1}{\hookrightarrow} l_3 \overset{\tau_3}{\hookrightarrow} l_7 \overset{\tau_4}{\hookrightarrow} l_8 \overset{f}{\hookrightarrow} l_8 \end{array} \right\}$$

For example, the first sequence, *i.e.* $l_1 \overset{\tau_1}{\hookrightarrow} l_3 \overset{\tau_5}{\hookrightarrow} l_4 \overset{c}{\hookrightarrow} l_5$, belongs to $\mathit{Sequences}^{\tau.a}(l_1)$, as we can syntactically move from the location l_1 to the location l_4 following the symbolic transitions labeled with internal actions τ_1 and τ_5, and then from the location l_4 we can execute the visible action c (*see* Figure A.1, page 248).

(2) For the locations l_9 the sets of $\tau.a$-sequences, *i.e.* $\mathit{Sequences}^{\tau.a}(l_9)$, is empty. Indeed, the single symbolic transitions outgoing from this location is labeled only with visible actions e.

Notice also that the sets $\mathit{Sequences}^{\tau.a}(l_1)$ and $\mathit{Sequences}^{\tau.a}(l_9)$ are the subsets of $\mathit{Sequences}^{\tau}(\mathcal{A})$. (We explained how to compute $\mathit{Sequences}^{\tau}(\mathcal{A})$ in Example A.1, page 247.) □

Now, using the notions of $\tau.a$-sequences and $\mathsf{post}_{\Sigma'}(L')$ together with the collapsing operation (*see* Definition A.2, page 249), we can give the *constructive definition* of the closure operation that eliminates *all* internal actions from an IOSTS that does not contain any syntactic livelocks.

Definition A.7 (Closure) Let $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ be an IOSTS without any syntactic livelocks (*see* Definition A.4, page 257). Then, the *closure operation* transforms the given IOSTS $\mathcal{M}$ into the IOSTS $\mathsf{closure}(\mathcal{M}) = \langle D, \Theta, L', l^0, \Sigma', T' \rangle$, where:

(1) $\Sigma' = \Sigma \setminus \Sigma^{\tau}$.

Thus, the alphabet of internal actions of $\mathsf{closure}(\mathcal{M})$ is empty.

(2) $L' = \{l^0\} \cup \mathsf{post}_{\Sigma \setminus \Sigma^{\tau}}(L) \subseteq L$.

Thus, the set of locations of $\mathsf{closure}(\mathcal{M})$ consists of the initial location l^0 of $\mathcal{M}$ and all the locations of $\mathcal{M}$ which are directly reachable from the locations of $\mathcal{M}$ by visible actions.

(3) $T' = (T \setminus T^\tau) \cup T^{\mathsf{collapse}(\rho)}$, where

- $T^\tau \triangleq \{l \overset{\tau}{\hookrightarrow} l' \mid l \in L \wedge l' \in L \wedge \tau \in \Sigma^\tau\} \subseteq T$, *i.e.* it is the set of all symbolic transitions of $\mathcal{M}$ labeled with internal actions of $\mathcal{M}$, and
- $T^{\mathsf{collapse}(\rho)} \triangleq \{\mathsf{collapse}(\rho) \mid \rho \in \bigcup_{l \in L'}(\mathit{Sequences}^{\tau.a}(l))\}$, *i.e.* it is the set of new symbolic transitions obtained from the set of all $\tau.a$-sequences starting in the locations L' of $\mathsf{closure}(\mathcal{M})$, *i.e.* $\bigcup_{l \in L'}(\mathit{Sequences}^{\tau.a}(l))$, by applying to each τ-sequence $\rho \in \bigcup_{l \in L'}(\mathit{Sequences}^{\tau.a}(l))$ the collapsing operation (*see* Definition A.2, page 249).

□

It is important to notice that the closure operation is stable, *i.e.* the result of this operation is indeed an IOSTS. This statement follows directly from the construction of the closure (*see* Definition A.7). The only non-trivial point is to show that the set of transitions of the IOSTS obtained after the closure operation is finite. This point is formulated as the theorem below.

Theorem A.2 (Well-Formedness of $\mathsf{closure}(\mathcal{M})$) Let $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T\rangle$ be an IOSTS without syntactic livelocks (*see* Definition A.4, page 257). Then, the set of transition of the IOSTS $\mathsf{closure}(\mathcal{M})$ obtained from $\mathcal{M}$ by the closure operation (*see* Definition A.7, page 260) is finite. □

Proof The proof of the theorem follows directly from the statement: for all visible actions $a \in (\Sigma \setminus \Sigma^\tau)$ and all locations $l \in L'$, the set of $\tau.a$-sequences starting in l is finite, which is proved by contradiction below.

Assume that the set $\mathit{Sequences}^{\tau.a}(l)$, where l is a location belonging to L', is infinite. This means that there exists a $\tau.a$-sequence ρ belonging to $\mathit{Sequences}^{\tau.a}(l)$ whose length is greater or equal to $|T^\tau| + 2$, where $|T^\tau|$ is the number of symbolic transitions (in $\mathcal{M}$) labeled with internal actions. Next, from the definition of $\tau.a$-sequences (*see* page 259), we know that the maximal strict prefix of ρ consists of only internal transitions. Thus, as the length of this prefix is greater or equal to $|T^\tau| + 1$, then it goes through the same internal transition at least twice. Therefore, $\mathcal{M}$ contains a cycle of internal actions, which contradicts the assumption that $\mathcal{M}$ does not contain syntactic livelocks (*see* Definition A.4, page 257). **Q.E.D.**

The example presented below illustrates the closure operation defined on the page 260.

Example A.7 (Closure of IOSTS without Syntactic Livelocks) Consider

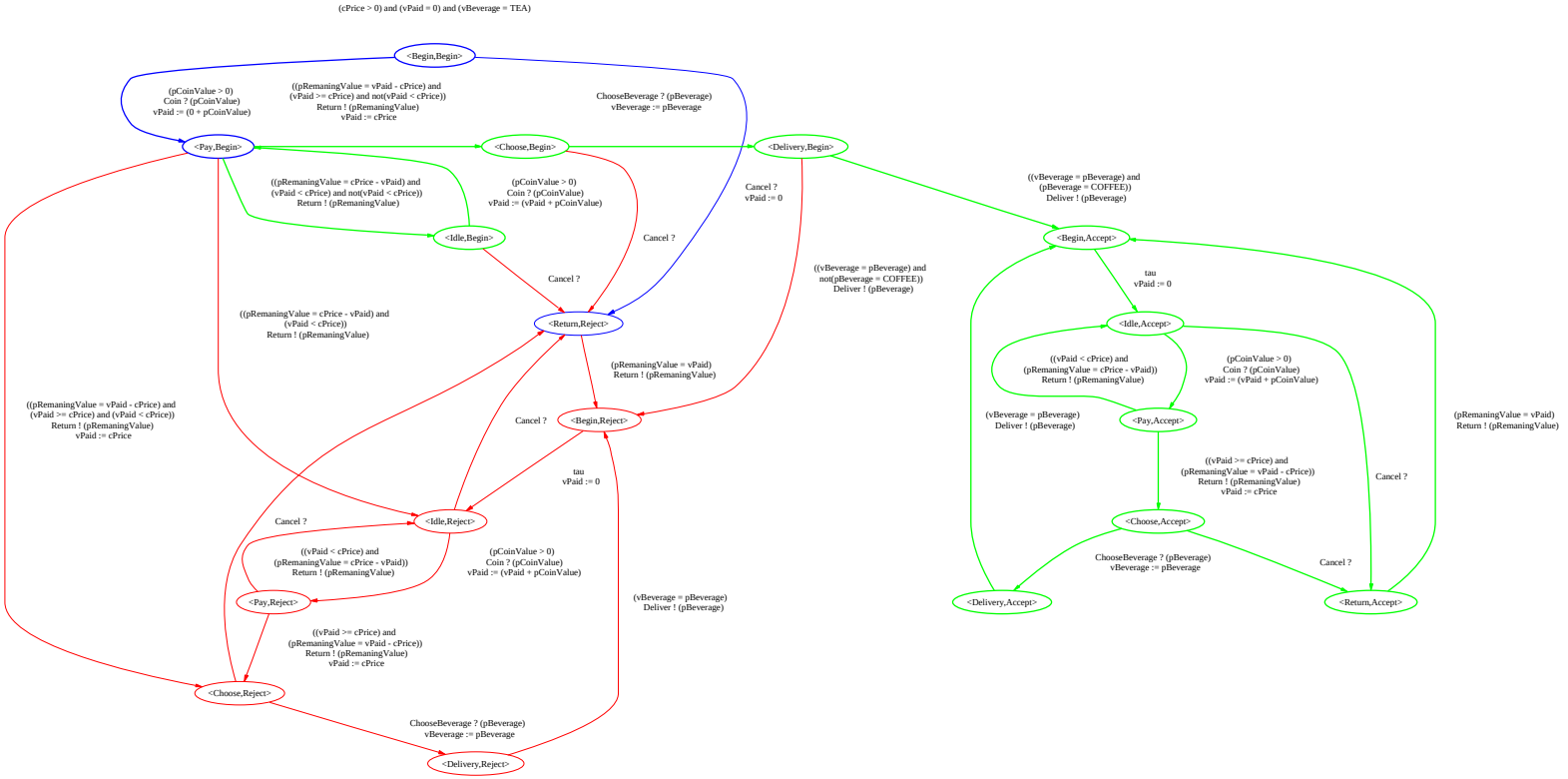


Figure A.3: The IOSTS *SP* (*see* Figure 7.3, page 174) after the closure operation.

the IOSTS SP shown on Figure 7.3 (*see* page 174). Notice that SP does not contain any syntactic livelock. We explain how to compute the IOSTS $\mathsf{closure}(SP)$.

(1) First, by removing the single internal action τ from the alphabet of SP, we obtain the alphabet of $\mathsf{closure}(SP)$, *i.e.*

$$\Sigma_{\mathsf{closure}(SP)} = \underbrace{\{Cancel, Coin, ChooseBeverage\}}_{\Sigma^{?}_{\mathsf{closure}(SP)}} \cup \underbrace{\{DeliverReturn\}}_{\Sigma^{!}_{\mathsf{closure}(SP)}}$$

where $\Sigma^{\tau}_{\mathsf{closure}(SP)} = \emptyset$.

(2) Next, using Figure 7.3 (*see* page 174) it is not hard to check that the set of location of $\mathsf{closure}(SP)$ is:

$$L_{\mathsf{closure}(SP)} = \underbrace{\{\langle Begin, Begin\rangle\}}_{l^0_{SP} \,=\, l^0_{\mathsf{closure}(SP)}} \cup \underbrace{L_{SP} \setminus \{\langle Begin, Begin\rangle\}}_{\mathsf{post}_{\Sigma_{SP} \setminus \Sigma^{\tau}_{SP}}(L_{SP})} = L_{SP}$$

(3) Finally, we compute the set of τ-sequences, which consists of two following elements:

$$\rho_1 : \quad \langle Begin, Begin\rangle \overset{\tau}{\hookrightarrow} \langle Idle, Begin\rangle \overset{Coin}{\hookrightarrow} \langle Pay, Begin\rangle$$
$$\rho_2 : \quad \langle Begin, Begin\rangle \overset{\tau}{\hookrightarrow} \langle Idle, Begin\rangle \overset{Cancel}{\hookrightarrow} \langle Return, Reject\rangle$$

By applying the collapsing operation to each of these τ-sequences, we obtain the set of syntactic transitions $T^{\mathsf{collapse}(\rho)}$ containing two symbolic transitions shown in blue on Figure A.3 (*see* page 262). Then, by replacing ρ_1 and ρ_2 with these symbolic transitions, we obtain the resulting IOSTS $\mathsf{closure}(SP)$ depicted in Figure A.3 (*see* page 262).

□

A.1.4 Traces of the Closure

In this section we consider an IOSTS $\mathcal{M}$ that does not contain syntactic livelocks (*see* Definition A.4, page 257), and study the relationship between traces of $\mathcal{M}$ and traces of the IOSTS $\mathsf{closure}(\mathcal{M})$ obtained from $\mathcal{M}$ by the closure operation (*see* Definitions A.7, page 260). The purpose of this section is to prove the equality between the set of traces of $\mathcal{M}$ and the set of traces of $\mathsf{closure}(\mathcal{M})$, *i.e.*

$$\mathit{Traces}(\mathsf{closure}(\mathcal{M})) = \mathit{Traces}(\mathcal{M}) \tag{A.4}$$

General Hypotheses. For the rest of this section we consider an IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ with set of states S and set of valued actions $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$. Suppose also that $\mathcal{M}$ does not contain any syntactic livelock (*see* Definition A.4, page 257). Finally, we consider the IOSTS $\mathsf{closure}(\mathcal{M})$ obtained from $\mathcal{M}$ by the closure operation (*see* Definition A.7, page 260).

Lemma A.1 (Behaviors of the Closure) The IOSTS $\mathcal{M}$ has a behavior:

$$\beta^{\rightarrow}_{\mathcal{M}} : \quad s_0 \overset{\varepsilon_1}{\Rightarrow} \tilde{s}_0 \overset{\alpha_1}{\rightarrow} s_1 \ldots s_{n-1} \overset{\varepsilon_n}{\Rightarrow} \tilde{s}_{n-1} \overset{\alpha_n}{\rightarrow} s_n$$

where for all i from 0 to n, $s_i, \tilde{s}_i \in S$, $\alpha_i \in (\Lambda \setminus \Lambda^\tau)$ and $\varepsilon_i \in (\Lambda^\tau)^*$ *if and only if* the IOSTS $\mathsf{closure}(\mathcal{M})$ has the behavior:

$$\beta^{\rightarrow}_{\mathsf{closure}(\mathcal{M})} : \quad s_0 \overset{\alpha_1}{\rightarrow} s_1 \ldots s_{n-1} \overset{\alpha_n}{\rightarrow} s_n$$

□

Proof

(⇒) First, we prove that for all behaviors $\beta^{\rightarrow}_{\mathcal{M}} : s_0 \overset{\varepsilon_1}{\Rightarrow} \tilde{s}_0 \overset{\alpha_1}{\rightarrow} s_1 \ldots s_{n-1} \overset{\varepsilon_n}{\Rightarrow} \tilde{s}_{n-1} \overset{\alpha_n}{\rightarrow} s_n$ of $\mathcal{M}$, where for all $i = 1..n$, $s_i, \tilde{s}_i \in S$, $\alpha_i \in (\Lambda \setminus \Lambda^\tau)$ and $\varepsilon_i \in (\Lambda^\tau)^*$, there exists a behavior $\beta^{\rightarrow}_{\mathsf{closure}(\mathcal{M})} : s_0 \overset{\alpha_1}{\rightarrow} s_1 \ldots s_{n-1} \overset{\alpha_n}{\rightarrow} s_n$ in $\mathsf{closure}(\mathcal{M})$. The proof of this implication is done by induction on $n \in \mathbb{N}$.

Induction Basis. The case when $n = 0$, *i.e.* where we have to show that if the state s_0 is an initial state of $\mathcal{M}$, then this state is also an initial state of $\mathsf{closure}(\mathcal{M})$, follows directly from the facts that IOSTS $\mathcal{M}$ and $\mathsf{closure}(\mathcal{M})$ have the same sets of data and the same initial conditions (*see* Definition A.7, page 260).

Induction Hypothesis. For $n-1$, assume that for a behavior $\beta'^{\rightarrow}_{\mathcal{M}} : s_0 \overset{\varepsilon_1}{\Rightarrow} \tilde{s}_0 \overset{\alpha_1}{\rightarrow} s_1 \ldots s_{n-2} \overset{\varepsilon_{n-1}}{\Rightarrow} \tilde{s}_{n-2} \overset{\alpha_{n-1}}{\rightarrow} s_{n-1}$ of $\mathcal{M}$, where for all $i = 1..n-1$, $s_i, \tilde{s}_i \in S$, $\alpha_i \in (\Lambda \setminus \Lambda^\tau)$ and $\varepsilon_i \in (\Lambda^\tau)^*$, there exists the behavior $\beta'^{\rightarrow}_{\mathsf{closure}(\mathcal{M})} : s_0 \overset{\alpha_1}{\rightarrow} s_1 \ldots s_{n-2} \overset{\alpha_{n-1}}{\rightarrow} s_{n-1}$ in $\mathsf{closure}(\mathcal{M})$.

Induction Step. For n, consider a behavior:

$$\beta^{\rightarrow}_{\mathcal{M}} : \quad \underbrace{\underbrace{s_0 \overset{\varepsilon_1}{\Rightarrow} \tilde{s}_0 \overset{\alpha_1}{\rightarrow} s_1 \ldots s_{n-1} \overset{\varepsilon_n}{\Rightarrow} \tilde{s}_{n-1}}_{\beta'^{\rightarrow}_{\mathcal{M}}} \overset{\alpha_n}{\rightarrow} s_n}_{\text{last step}} \tag{A.5}$$

of $\mathcal{M}$, where for all $i = 1..n$, $s_i, \tilde{s}_i \in S$, $\alpha_i \in (\Lambda \setminus \Lambda^\tau)$ and $\varepsilon_i \in (\Lambda^\tau)^*$. Then, we prove that exists the behavior:

$$\beta^{\rightarrow}_{\mathsf{closure}(\mathcal{M})} : \quad \underbrace{\underbrace{s_0 \overset{\alpha_1}{\rightarrow} s_1 \ldots s_{n-1}}_{\beta'^{\rightarrow}_{\mathsf{closure}(\mathcal{M})}} \overset{\alpha_n}{\rightarrow} s_n}_{\text{last step}} \tag{A.6}$$

of $\mathsf{closure}(\mathcal{M})$.

First, as behaviors of IOSTS are prefix closed, then $\beta'^{\rightarrow}_{\mathcal{M}}$, which is a prefix of behavior $\beta^{\rightarrow}_{\mathcal{M}}$ of $\mathcal{M}$ (*see* Formula (A.5)), is also a behavior of $\mathcal{M}$. Thus, using the induction hypothesis we obtain that $\mathsf{closure}(\mathcal{M})$ has the behavior $\beta'^{\rightarrow}_{\mathsf{closure}(\mathcal{M})}$, which is a prefix of $\beta^{\rightarrow}_{\mathsf{closure}(\mathcal{M})}$ shown as Formula (A.6).
This means that as in $\mathcal{M}$ the state s_{n-1} is reachable from the initial state s_0 by the sequence $\varepsilon_1\alpha_1 \ldots \varepsilon_{n-1}\alpha_{n-1}$, then it is also reachable from the same state s_0 in $\mathsf{closure}(\mathcal{M})$ by the sequence $\alpha_1 \ldots \alpha_{n-1}$.

Second, we have to show that as in $\mathcal{M}$ the state $s_n = \langle l_n, \vartheta_n\rangle$ is reachable from the state $s_{n-1} = \langle l_{n-1}, \vartheta_{n-1}\rangle$ by a sequence of internal actions $\varepsilon_n = \langle \tau_n^1, \langle\rangle\rangle \ldots \langle \tau_n^k, \langle\rangle\rangle$ (possibly empty) followed by the visible valued action $\alpha_n = \langle a, \omega\rangle$, then it is also reachable in $\mathsf{closure}(\mathcal{M})$ from the same state $s_{n-1} = \langle l_{n-1}, \vartheta_{n-1}\rangle$ by the only visible valued action $\alpha_n = \langle a, \omega\rangle$. In other words, we have to prove that as

$$\underbrace{\langle l_{n-1}, \vartheta_{n-1}\rangle}_{s_{n-1}} \underbrace{\xrightarrow{\langle \tau_n^1, \langle\rangle\rangle} \ldots \xrightarrow{\langle \tau_n^k, \langle\rangle\rangle}}_{\stackrel{\varepsilon_n}{\Rightarrow}} \underbrace{\langle \tilde{l}_{n-1}, \tilde{\vartheta}_{n-1}\rangle}_{\tilde{s}_{n-1}} \xrightarrow{\alpha_n = \langle a, \omega\rangle} \underbrace{\langle l_n, \vartheta_n\rangle}_{s_n} \tag{A.7}$$

which is the last step of $\beta^{\rightarrow}_{\mathcal{M}}$ shown as Formula (A.5), that was unfolded using Definition 4.13 (*see* page 90), holds in $\mathcal{M}$, then the relation

$$\underbrace{\langle l_{n-1}, \vartheta_{n-1}\rangle}_{s_{n-1}} \xrightarrow{\alpha_n = \langle a, \omega\rangle} \underbrace{\langle l_n, \vartheta_n\rangle}_{s_n} \tag{A.8}$$

which is the last step of $\beta^{\rightarrow}_{\mathsf{closure}(\mathcal{M})}$ shown as Formula (A.6), holds in $\mathsf{closure}(\mathcal{M})$.

(1) As we know that Formula (A.7) holds in the IOSTS $\mathcal{M}$, then this IOSTS has the $\tau.a$-sequence $\rho : \underbrace{l_{n-1} \xhookrightarrow{\tau_n^1}}_{t_1} \ldots \underbrace{\xhookrightarrow{\tau_n^k} \tilde{l}_{n-1}}_{t_k} \underbrace{\xhookrightarrow{a} l_n}_{t_{k+1}}$.
Therefore, as the IOSTS $\mathsf{closure}(\mathcal{M})$ is obtained from $\mathcal{M}$ by the closure operation (*see* Definition A.7, page 260), then it contains the symbolic transition $t_{\mathsf{closure}(\mathcal{M})} : l_{n-1} \xhookrightarrow{a} l_n$ such that $t_{\mathsf{closure}(\mathcal{M})} = \mathsf{collapse}(\rho)$.

(2) Using Definition A.2 (*see* page 249) and the fact that Formula (A.7) holds in $\mathcal{M}$, we obtain that the triple $\langle s_{n-1}, \alpha_n, s_n\rangle$ belongs to $(\rightarrow_{t_1} \circ (\ldots \circ (\rightarrow_{t_k} \circ \rightarrow_{t_{k+1}})))$.

Next, due to Observation A.2 (*see* page 259) we know that the $\tau.a$ sequence ρ is also a τ-sequence of $\mathcal{M}$, thus we can use Theorem A.1 (*see* page 252). From this theorem we obtain that as the triple $\langle s_{n-1}, \alpha_n, s_n \rangle$ belongs to $(\rightarrow_{t_1} \circ (\ldots \circ (\rightarrow_{t_k} \circ \rightarrow_{t_{k+1}})))$ (*see* the first sentence of the item (2)), then it also belongs to $\rightarrow_{t_{\mathsf{closure}(\mathcal{M})}}$ of $\mathsf{closure}(\mathcal{M})$. This means that the relation presented as Formula (A.8) holds in $\mathsf{closure}(\mathcal{M})$.

Therefore, we have proved that if in $\mathcal{M}$ the state s_n is reachable from the initial state s_0 by the sequence $\varepsilon_1\alpha_1 \ldots \varepsilon_n\alpha_n$, then it is also reachable in $\mathsf{closure}(\mathcal{M})$ from the same state s_0 by the sequence $\alpha_1 \ldots \alpha_n$. This statement implies the induction step. Thus, the first implication of the lemma is proved.

(⇐) Second, we prove that for all behaviors $\beta^{\rightarrow}_{\mathsf{closure}(\mathcal{M})} : s_0 \stackrel{\alpha_1}{\rightarrow} s_1 \ldots s_{n-1} \stackrel{\alpha_n}{\rightarrow} s_n$ of $\mathsf{closure}(\mathcal{M})$, where $s_1, \ldots, s_n \in S$ and $\alpha_1, \ldots, \alpha_n \in (\Lambda \setminus \Lambda^\tau)$, there exist $\varepsilon_1, \ldots, \varepsilon_n \in (\Lambda^\tau)^*$ and $\tilde{s}^1, \ldots, \tilde{s}^{n-1} \in S$ such that $\beta^{\rightarrow}_{\mathcal{M}} : s_0 \stackrel{\varepsilon_1}{\Rightarrow} \tilde{s}_0 \stackrel{\alpha_1}{\rightarrow} s_1 \ldots s_{n-1} \stackrel{\varepsilon_n}{\Rightarrow} \tilde{s}_{n-1} \stackrel{\alpha_n}{\rightarrow} s_n$ is a behavior of $\mathsf{closure}(\mathcal{M})$.

The proof of this implication is done by induction on $n \in \mathbb{N}$.

Induction Basis. The case when $n = 0$, *i.e.* where we have to show that if the state s_0 is an initial state of $\mathsf{closure}(\mathcal{M})$, then this state is also an initial state of $\mathcal{M}$, can be proved exactly as the basis step of the first implication **(⇒)** (*see* page 264).

Induction Hypothesis. For $n-1$, assume that for a behavior $\beta'^{\rightarrow}_{\mathsf{closure}(\mathcal{M})} : s_0 \stackrel{\alpha_1}{\rightarrow} s_1 \ldots s_{n-2} \stackrel{\alpha_{n-1}}{\rightarrow} s_{n-1}$ of $\mathsf{closure}(\mathcal{M})$, where $s_1, \ldots, s_{n-1} \in S$, $\alpha_1, \ldots, \alpha_{n-1} \in (\Lambda \setminus \Lambda^\tau)$, there exist $\varepsilon_1, \ldots, \varepsilon_{n-1} \in (\Lambda^\tau)^*$ and $\tilde{s}^1, \ldots, \tilde{s}^{n-2} \in S$ such that $\beta'^{\rightarrow}_{\mathcal{M}} : s_0 \stackrel{\varepsilon_1}{\Rightarrow} \tilde{s}_0 \stackrel{\alpha_1}{\rightarrow} s_1 \ldots s_{n-2} \stackrel{\varepsilon_{n-1}}{\Rightarrow} \tilde{s}_{n-2} \stackrel{\alpha_{n-1}}{\rightarrow} s_{n-1}$ is a behavior of $\mathcal{M}$.

Induction Step. For n, consider a behavior:

$$\beta^{\rightarrow}_{\mathsf{closure}(\mathcal{M})} : \quad \underbrace{s_0 \stackrel{\alpha_1}{\rightarrow} s_1 \ldots s_{n-1}}_{\beta'^{\rightarrow}_{\mathsf{closure}(\mathcal{M})}} \underbrace{\stackrel{\alpha_n}{\rightarrow} s_n}_{\text{last step}} \tag{A.9}$$

of $\mathsf{closure}(\mathcal{M})$, where $s_1, \ldots, s_n \in S$, $\alpha_1, \ldots, \alpha_n \in (\Lambda \setminus \Lambda^\tau)$. Then, we prove that $\mathcal{M}$ has the following behavior:

$$\beta^{\rightarrow}_{\mathcal{M}} : \quad \underbrace{s_0 \stackrel{\varepsilon_1}{\Rightarrow} \tilde{s}_0 \stackrel{\alpha_1}{\rightarrow} s_1 \ldots s_{n-1} \stackrel{\varepsilon_n}{\Rightarrow} \tilde{s}_{n-1}}_{\beta'^{\rightarrow}_{\mathcal{M}}} \underbrace{\stackrel{\alpha_n}{\rightarrow} s_n}_{\text{last step}} \tag{A.10}$$

for $\varepsilon_1, \ldots, \varepsilon_n \in (\Lambda^\tau)^*$.

First, as behaviors of IOSTS are prefix closed, then $\beta'^{\rightarrow}_{\mathsf{closure}(\mathcal{M})}$, which is a prefix of behavior $\beta^{\rightarrow}_{\mathsf{closure}(\mathcal{M})}$ of $\mathsf{closure}(\mathcal{M})$ (*see* Formula (A.9)), is also a behavior of $\mathsf{closure}(\mathcal{M})$. Thus, using the induction hypothesis we obtain that $\mathcal{M}$ has the behavior $\beta'^{\rightarrow}_{\mathcal{M}}$, which is a prefix of $\beta^{\rightarrow}_{\mathcal{M}}$ shown as Formula A.10.

This means that as in $\mathsf{closure}(\mathcal{M})$ the state s_{n-1} is reachable from the initial state s_0 by the sequence $\alpha_1 \ldots \alpha_{n-1}$, then it is also reachable from the same state s_0 in $\mathcal{M}$ by the sequence $\varepsilon_1\alpha_1 \ldots \varepsilon_{n-1}\alpha_{n-1}$.

Second, we have to show that as in $\mathsf{closure}(\mathcal{M})$ the state $s_n = \langle l_n, \vartheta_n \rangle$ is reachable from the state $s_{n-1} = \langle l_{n-1}, \vartheta_{n-1} \rangle$ by the visible valued action $\alpha_n = \langle a, \omega \rangle$ then it is also reachable in $\mathcal{M}$ from the same state $s_{n-1} = \langle l_{n-1}, \vartheta_{n-1} \rangle$ by a sequence of internal actions $\varepsilon_n = \langle \tau_n^1, \langle\rangle \rangle \ldots \langle \tau_n^k, \langle\rangle \rangle$ (possibly empty) followed by the visible valued action $\alpha_n = \langle a, \omega \rangle$. In other words, we have to prove that if the relation

$$\underbrace{\langle l_{n-1}, \vartheta_{n-1} \rangle}_{s_{n-1}} \overset{\alpha_n = \langle a, \omega \rangle}{\longrightarrow} \underbrace{\langle l_n, \vartheta_n \rangle}_{s_n} \tag{A.11}$$

which is the last step of $\beta^{\rightarrow}_{\mathsf{closure}(\mathcal{M})}$ shown as Formula (A.9), holds in $\mathsf{closure}(\mathcal{M})$, then there exist k intermediate states in $\mathsf{closure}(\mathcal{M})$ and k valued internal actions $\langle \tau_n^1, \langle\rangle \rangle, ..., \langle \tau_n^k, \langle\rangle \rangle \in \Lambda^\tau$, where $k \in \mathbb{N}$, such that

$$\underbrace{\langle l_{n-1}, \vartheta_{n-1} \rangle}_{s_{n-1}} \underbrace{\overset{\langle \tau_n^1, \langle\rangle \rangle}{\longrightarrow} \ldots \overset{\langle \tau_n^k, \langle\rangle \rangle}{\longrightarrow}}_{\overset{\varepsilon_n}{\Rightarrow}} \underbrace{\langle \tilde{l}_{n-1}, \tilde{\vartheta}_{n-1} \rangle}_{\tilde{s}_{n-1}} \overset{\alpha_n = \langle a, \omega \rangle}{\longrightarrow} \underbrace{\langle l_n, \vartheta_n \rangle}_{s_n} \tag{A.12}$$

which is the last step of $\beta^{\rightarrow}_{\mathcal{M}}$ shown as Formula (A.10), that was unfolded using Definition 4.13 (*see* page 90), holds in $\mathcal{M}$.

To prove this statement first notice that as the relation shown as Formula (A.11) holds in the IOSTS $\mathsf{closure}(\mathcal{M})$, then:

(1) $\mathsf{closure}(\mathcal{M})$ has the symbolic transition $t_{\mathsf{closure}(\mathcal{M})} : l_{n-1} \overset{a}{\hookrightarrow} l_n$, and

(2) the triple $\langle s_{n-1}, \alpha_n, s_n \rangle$ belongs to $\rightarrow_{t_{\mathsf{closure}(\mathcal{M})}}$.

Next, we remind that $\mathcal{M}$ is the IOSTS from which $\mathsf{closure}(\mathcal{M})$ was constructed using the closure operation (*see* Definition A.7, page 260). Therefore, as $\mathsf{closure}(\mathcal{M})$ contains the symbolic transition $t_{\mathsf{closure}(\mathcal{M})}$, then either:

(a) $\mathcal{M}$ has exactly the same symbolic transition $t_{\mathcal{M}} : l_{n-1} \overset{a}{\hookrightarrow} l_n$. Thus, as we know that the triple $\langle s_{n-1}, \alpha_n, s_n \rangle$ belongs to $\rightarrow_{t_{\mathsf{closure}(\mathcal{M})}}$ of $\mathsf{closure}(\mathcal{M})$ (*see* the item (2) above), then this triple also belongs to $\rightarrow_{t_{\mathcal{M}}}$ of $\mathcal{M}$. Or,

(b) $\mathcal{M}$ has a $\tau.a$-sequence ρ : $\underbrace{l_{n-1} \overset{\tau_n^1}{\hookrightarrow}}_{t_1} \ldots \underbrace{\overset{\tau_n^k}{\hookrightarrow} \tilde{l}_{n-1}}_{t_k} \underbrace{\overset{a}{\hookrightarrow} l_n}_{t_{k+1}}$ such that $t_{\mathsf{closure}(\mathcal{M})} = \mathsf{collapse}(\rho)$. Due to Observation A.2 (*see* page 259) we know that the $\tau.a$ sequence ρ is also a τ-sequence of $\mathcal{M}$. Thus, we can use Theorem A.1 (*see* page 252) from which we obtain that: as the triple $\langle s_{n-1}, \alpha_n, s_n \rangle$ belongs to $\rightarrow_{t_{\mathsf{closure}(\mathcal{M})}}$ of $\mathsf{closure}(\mathcal{M})$ (*see* the item (1) above), then this triple also belongs to $(\rightarrow_{t_1} \circ (\ldots \circ (\rightarrow_{t_k} \circ \rightarrow_{t_{k+1}})))$ of $\mathcal{M}$. Due to Definition A.2 (*see* page 249) this means that there exist intermediate states between s_{n-1} and s_n and valued internal actions $\langle \tau_n^1, \langle\rangle \rangle, ..., \langle \tau_n^k, \langle\rangle \rangle$ such that sequence of relations shown as Formula (A.12) holds in $\mathcal{M}$.

Therefore, we have proved that if in $\mathsf{closure}(\mathcal{M})$ the state s_n is reachable from the initial state s_0 by the sequence $\alpha_1 \ldots \alpha_n$, then it is also reachable in $\mathcal{M}$ from the same state s_0 by the sequence $\varepsilon_1 \alpha_1 \ldots \varepsilon_n \alpha_n$. This statement implies the induction step. Thus, the second implication of the lemma is proved.

Q.E.D.

Next, just before proving the trace-equivalence between $\mathcal{M}$ and $\mathsf{closure}(\mathcal{M})$ (*see* Equality (A.4)), we show that:

(1) for each sequence $\eta_{\mathsf{closure}(\mathcal{M})}$ of the IOSTS $\mathsf{closure}(\mathcal{M})$, $\mathcal{M}$ has a sequence $\eta_{\mathcal{M}}$ obtained from $\eta_{\mathsf{closure}(\mathcal{M})}$ by inserting some sequences of internal actions (possibly empty) before each *visible* valued action of $\eta_{\mathsf{closure}(\mathcal{M})}$; and

(2) for each sequence $\eta_{\mathcal{M}}$ of $\mathcal{M}$, $\mathsf{closure}(\mathcal{M})$ has the sequence $\eta_{\mathsf{closure}(\mathcal{M})}$ obtained from $\eta_{\mathcal{M}}$ by dropping all *internal* actions.

These statements are formalized as the following observation:

Observation A.3 (Sequences of the Closure) The sequence $\eta_{\mathsf{closure}(\mathcal{M})} = \alpha_1 \ldots \alpha_n$, where $\alpha_i \in (\Lambda^? \cup \Lambda^!)$ $(i = 1..n)$, belongs to *Sequences*$(\mathsf{closure}(\mathcal{M}))$ *if and only if* $\exists\ \varepsilon_1, \ldots, \varepsilon_n \in (\Lambda^\tau)^*\ .\ [\eta_{\mathcal{M}} = \varepsilon_1 \alpha_1 \ldots \varepsilon_n \alpha_n \in \mathit{Sequences}(\mathcal{M})]$. □

This observation follows directly from the definition of a sequence (*see* page 90) and Lemma A.1 (*see* page 264).

Finally, we formally state and prove the theorem about the trace-equivalence between an IOSTS $\mathcal{M}$ and the IOSTS $\mathsf{closure}(\mathcal{M})$ obtained from $\mathcal{M}$ by the closure operation.

Theorem A.3 (Traces of the Closure) The result of the closure operation (*see* Definition A.7, page 260) applied to an IOSTS $\mathcal{M}$ without syntactic livelocks has the same set of traces as $\mathcal{M}$, *i.e.*

$$\mathit{Traces}(\mathsf{closure}(\mathcal{M})) = \mathit{Traces}(\mathcal{M})$$

□

Proof

(⊆) First, we show that each trace of $\mathsf{closure}(\mathcal{M})$ is also a trace of $\mathcal{M}$.

Consider an arbitrary trace $\sigma = \alpha_1 \dots \alpha_n$ belonging to $\mathit{Traces}(\mathsf{closure}(\mathcal{M}))$. As the IOSTS $\mathsf{closure}(\mathcal{M})$ does not have any internal action (*see* Definition A.7, page 260), then the set of traces of $\mathsf{closure}(\mathcal{M})$ is equal to its set of sequences (*see* Definition 4.12, page 90) *i.e.* $\mathit{Traces}(\mathsf{closure}(\mathcal{M})) = \mathit{Sequences}(\mathsf{closure}(\mathcal{M}))$. Thus, the trace σ belongs to $\mathit{Sequences}(\mathsf{closure}(\mathcal{M}))$. Next, due to the first implication **(⇒)** of Observation A.3 (*see* page 268) we obtain that there exist the sequence $\eta = \varepsilon_1\alpha_1 \dots \varepsilon_n\alpha_n$ of $\mathcal{M}$ for some $\varepsilon_1, \dots, \varepsilon_n \in (\Lambda^\tau)^*$, where Λ^τ is the alphabet of internal actions of $\mathcal{M}$. Then, from the definitions of sequences and traces (*see* pages 90 and 90 respectively), we know that if we drop all internal action from the sequence $\eta = \varepsilon_1\alpha_1 \dots \varepsilon_n\alpha_n$ of $\mathcal{M}$, then we obtain the trace $\alpha_1 \dots \alpha_n$ of $\mathcal{M}$. Therefore, $\sigma \in \mathit{Traces}(\mathcal{M})$.

(⊇) Second, we prove that each trace of $\mathcal{M}$ is also a trace of $\mathsf{closure}(\mathcal{M})$.

Consider a trace $\sigma = \alpha_1 \dots \alpha_n$ belonging to $\mathit{Traces}(\mathcal{M})$. Due to Definitions 4.12 and 4.14 (*see* pages 90 and 90), this trace σ corresponds to some sequence $\eta = \varepsilon_1\alpha_1 \dots \varepsilon_n\alpha_n$ of $\mathcal{M}$, where $\varepsilon_i \in (\Lambda^\tau)^*$ $(i = 1..n)$. Next, using the second implication **(⇐)** of Observation A.3 (*see* page 268) we get that there exist the sequence $\eta' = \alpha_1 \dots \alpha_n$ of $\mathsf{closure}(\mathcal{M})$. Finally, as $\alpha_1, \dots, \alpha_n$ are *visible* valued actions, then the sequence η', which is exactly the same as σ, is the trace of $\mathsf{closure}(\mathcal{M})$. Therefore $\sigma \in \mathit{Traces}(\mathsf{closure}(\mathcal{M}))$.

Q.E.D.

A.1.5 Traces and Accepting Traces of closure($Spec \times TP$)

In this section we consider two IOSTS $Spec \times TP$ and closure($Spec \times TP$). The first one is obtained by the product operation (*see* Definition 5.4, page 114) from a specification *Spec* and a test purpose *TP* of *Spec*. The second one is obtained from $Spec \times TP$ by the closure operation (*see* Definition A.7, page 260). The purpose of this section is to study the relationships between traces (*see* Definition 4.14, page 90) and accepting traces (*see* Definition 6.10, page 151) of $Spec \times TP$ and closure($Spec \times TP$).

Before going into details we make an observation about absence of syntactic livelocks (defined in Section A.1.2, page 257) in $Spec \times TP$ with alphabet of actions $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ and set of locations $L = L_{Spec} \times L_{TP}$. Formally:

Lemma A.2 Let *Spec* be a specification without any syntactic livelock, and *TP* be a test purpose of *Spec*. Then, the result of the product operation between *Spec* and *TP*, *i.e.* $Spec \times TP$, is free of syntactic livelocks. □

Proof The proof of the observation is done by contradiction. Assume that the synchronous product $Spec \times TP$ has a syntactic livelock (*see* Definition A.4, page 257), for instance:

$$\rho_{Spec \times TP} : \quad \langle l^1_{Spec}, l^1_{TP} \rangle \overset{\tau_1}{\hookrightarrow} \langle l^2_{Spec}, l^2_{TP} \rangle \ldots \langle l^{n-1}_{Spec}, l^{n-1}_{TP} \rangle \overset{\tau_{n-1}}{\hookrightarrow} \langle l^1_{Spec}, l^1_{TP} \rangle \quad \text{(A.13)}$$

where $\forall\ i = 1..n-1\ .\ [\tau_i \in \Sigma^\tau \wedge \langle l^i_{Spec}, l^i_{TP} \rangle \in L]$.

Then, due to Definition 5.4 (*see* page 114), we know that the product $Spec \times TP$ is obtained from *Spec* and *TP* by synchronization on *all* their actions, which are the same as the actions of $Spec \times TP$. Thus, as $Spec \times TP$ has the sequence of symbolic transitions $\rho_{Spec \times TP}$ shown as Formula (A.13), then *Spec* and *TP* have the following sequences of symbolic transitions:

$$\begin{aligned} \rho_{Spec} : &\quad l^1_{Spec} \overset{\tau_1}{\hookrightarrow} l^2_{Spec} \ldots l^{n-1}_{Spec} \overset{\tau_{n-1}}{\hookrightarrow} l^1_{Spec} \\ \rho_{TP} : &\quad l^1_{TP} \overset{\tau_1}{\hookrightarrow} l^2_{TP} \ldots l^{n-1}_{TP} \overset{\tau_{n-1}}{\hookrightarrow} l^1_{TP} \end{aligned} \quad \text{(A.14)}$$

Now, from Formula (A.14) we see that *Spec* contains a syntactic livelock (*see* Definition A.4, page 257). This contradicts with the hypothesis made in Section 6.1 (*see* page 137), which says that *Spec* does not contain syntactic livelocks. **Q.E.D.**

Then, using Lemma A.2 and Theorem A.3 (*see* page 269) we obtain that the closure operation preserves the set of traces of $Spec \times TP$, *i.e.*

$$Traces(\text{closure}(Spec \times TP)) = Traces(Spec \times TP) \quad \text{(A.15)}$$

Finally, we show equality between sets of accepting traces of $\mathsf{closure}(Spec \times TP)$ and $Spec \times TP$.

Theorem A.4 (Accepting Traces of $\mathsf{closure}(Spec \times TP)$) The result of the closure operation (*see* Definition A.7, page 260) applied to a synchronous product $Spec \times TP$ (where $Spec$ does not contain any syntactic livelock) has the same set of accepting traces as $Spec \times TP$, *i.e.*

$$ATraces(\mathsf{closure}(Spec \times TP)) = ATraces(Spec \times TP)$$

□

Proof

(⊆) First, we show that each accepting trace of $\mathsf{closure}(Spec \times TP)$ is also an accepting trace of $Spec \times TP$.

Consider an arbitrary accepting trace $\sigma = \alpha_1 \dots \alpha_n$ belonging to $ATraces(\mathsf{closure}(Spec \times TP))$ and leading to an accepting state $s_{Accept} = \langle\langle l, Accept\rangle, \vartheta\rangle$, where $\langle l, Accept\rangle \in L_{\mathsf{closure}(Spec \times TP)}$ and $\vartheta \in \mathsf{DOM}(V_{\mathsf{closure}(Spec \times TP)} \cup C_{\mathsf{closure}(Spec \times TP)})$.

Due to Definition A.7 (*see* page 260), the IOSTS $\mathsf{closure}(Spec \times TP)$ does not have any internal actions. Thus, the given accepting trace σ is also a sequence (*see* Definition 4.12, page 90) of $\mathsf{closure}(Spec \times TP)$ leading to the same accepting state s_{Accept}.

Next, due to Lemma A.2 (*see* page 270) about the absence of syntactic livelocks in $Spec \times TP$, we can use Observation A.3 (*see* page 268). From the first implication (⇒) of this observation we obtain that there exist the sequence $\eta = \varepsilon_1\alpha_1 \dots \varepsilon_n\alpha_n$ of $Spec \times TP$ for some $\varepsilon_1, \dots, \varepsilon_n \in (\Lambda^{\tau}_{Spec \times TP})^*$, where $\Lambda^{\tau}_{Spec \times TP}$ is the alphabet of internal actions of $Spec \times TP$. Moreover, notice that the first implication of the Observation A.3 (*see* page 268) follows from the second implication (⇐) of Lemma A.1 (*see* page 264). If we make attention to the proof of the latter implication (*see* page 266), we obtain that the accepting state s_{Accept} is reachable in $Spec \times TP$ by η.

Then, from the definitions of sequences, traces and accepting traces (*see* pages 90, 90 and 151 respectively), we know that if we drop all internal action from the sequence $\eta = \varepsilon_1\alpha_1 \dots \varepsilon_n\alpha_n$ of $Spec \times TP$, then we obtain the accepting trace $\alpha_1 \dots \alpha_n$ of $Spec \times TP$. Therefore, $\sigma \in ATraces(Spec \times TP)$.

(⊇) Second, we prove that each accepting trace of $Spec \times TP$ is also an accepting trace of $\mathsf{closure}(Spec \times TP)$.

Consider an accepting trace $\sigma = \alpha_1 \dots \alpha_n$ belonging to $ATraces(Spec \times TP)$, and leading to some accepting state s_{Accept}.

($\star$) First, we make an observation which says that $Spec \times TP$ can move from a non-accepting location to an accepting location only by taking a symbolic transition labeled with an input/output action, *i.e.* formally: $\forall\, \langle\langle l_{Spec}, l_{TP}\rangle, a, \pi, G, A, \langle l'_{Spec}, l'_{TP}\rangle\rangle \in T_{Spec\times TP}\,.\,[(l_{TP} \neq l'_{TP} \wedge l'_{TP} = Accept) \implies (a \in \Sigma^{?}_{Spec\times TP} \cup \Sigma^{!}_{Spec\times TP})]$, where l_{TP}, l'_{TP} are locations of TP and l_{Spec}, l'_{Spec} are locations of $Spec$. This observation follows from the hypotheses (2) and (4) made on TP (*see* Definition 6.8, page 147), and the product construction between $Spec$ and TP (*see* Definition 5.4, page 114).

Second, using Definitions 4.12 and 4.14 (*see* pages 90 and 90) we have that there exists a sequence $\eta = \varepsilon_1\alpha_1 \dots \varepsilon_n\alpha_n\varepsilon_{n+1}$ of $Spec \times TP$ corresponding to the trace σ and leading to the same accepting state s_{Accept}. Here, $\varepsilon_1, \dots, \varepsilon_{n+1} \in (\Lambda^{\tau}_{Spec\times TP})^{*}$. Next, due to definition of a sequence (*see* page 90) we obtain that η corresponds to the following behavior $\beta^{\rightarrow}$:

$$\begin{array}{l} \langle\langle l^0_{Spec}, l^0_{TP}\rangle, \vartheta^0\rangle \stackrel{\varepsilon_1}{\Rightarrow} \langle\langle \tilde{l}^0_{Spec}, \tilde{l}^0_{TP}\rangle, \tilde{\vartheta}^0\rangle \stackrel{\alpha_1}{\rightarrow} \langle\langle l^1_{Spec}, l^1_{TP}\rangle, \vartheta^1\rangle \\ \dots \\ \langle\langle \tilde{l}^{n-1}_{Spec}, \tilde{l}^{n-1}_{TP}\rangle, \tilde{\vartheta}^{n-1}\rangle \stackrel{\alpha_n}{\rightarrow} \langle\langle l^n_{Spec}, l^n_{TP}\rangle, \vartheta^n\rangle \stackrel{\varepsilon_{n+1}}{\Rightarrow} \underbrace{\langle\langle \tilde{l}^n_{Spec}, \underline{Accept}\rangle, \tilde{\vartheta}^n\rangle}_{s_{Accept}} \end{array} \tag{A.16}$$

where for all i from 0 to n, $\langle l^i_{TP}, l^i_{Spec}\rangle, \langle \tilde{l}^i_{Spec}, \tilde{l}^i_{TP}\rangle \in L_{Spec\times TP}$ and $\vartheta^i, \tilde{\vartheta}^i \in \mathsf{DOM}(V_{Spec\times TP} \cup C_{Spec\times TP})$.

Then, we show that the state $\langle\langle l^n_{Spec}, l^n_{TP}\rangle, \vartheta^n\rangle$ of $\beta^{\rightarrow}$ (*see* Formula (A.16)) is an accepting state of $Spec \times TP$, *i.e.* $l^n_{TP} = Accept$. This statement can be proved by contradiction. First, we unfold the last step of $\beta^{\rightarrow}$ (*see* Formula A.16) according to Definition 4.13 (*see* page 90) as follows:

$$\langle\langle l^n_{Spec}, l^n_{TP}\rangle, \vartheta^n\rangle \underbrace{\stackrel{\langle\tau^1_n,\langle\rangle\rangle}{\longrightarrow} \dots \stackrel{\langle\tau^k_n,\langle\rangle\rangle}{\longrightarrow}}_{\stackrel{\varepsilon_{n+1}}{\Rightarrow}} \underbrace{\langle\langle \tilde{l}^n_{Spec}, Accept\rangle, \tilde{\vartheta}^n\rangle}_{s_{Accept}}$$

Then, we assume that $\langle\langle l^n_{Spec}, l^n_{TP}\rangle, \vartheta^n\rangle$ is an non-accepting state of $Spec \times TP$, *i.e.* $l^n_{TP} \neq Accept$. Next, using Observation ($\star$) the state reachable from the non-accepting state $\langle\langle l^n_{Spec}, l^n_{TP}\rangle, \vartheta^n\rangle$ by τ^1_n (*i.e.* the first internal action of ε_{n+1}) is also non-accepting. Repeating this argument $k-1$ times (*i.e.* for all remaining internal actions of ε_{n+1}) we obtain that the state s_{Accept} is non-accepting. This contradicts with the fact that η corresponding to $\beta^{\rightarrow}$ is an accepting sequence of $Spec \times TP$. Therefore, $\langle\langle l^n_{Spec}, l^n_{TP}\rangle, \vartheta^n\rangle$ is the accepting state, where $l^n_{TP} = Accept$.

Next, consider the sequence $\eta' = \varepsilon_1\alpha_1 \ldots \varepsilon_n\alpha_n$ of $Spec \times TP$ obtained from the sequence η by eliminating its last sequence of internal actions, *i.e.* ε_{n+1}. From above we know that η' leads to the accepting state $s'_{Accept} = \langle\langle l^n_{Spec}, \underbrace{l^n_{TP}}_{Accept}\rangle, \vartheta^n\rangle$.

Then, as we know that $Spec \times TP$ does not have any syntactic livelocks (*see* Lemma A.2), and $\eta' \in Sequences(Spec \times TP)$, we can use the second implication $(\Leftarrow)$ of Observation A.3 (*see* page 268). From this implication we obtain that there exists the sequence $\eta'' = \alpha_1 \ldots \alpha_n$ of $\mathsf{closure}(Spec \times TP)$. Moreover, notice that the second implication of the Observation A.3 (*see* page 268) follows from the first implication $(\Rightarrow)$ of Lemma A.1 (*see* page 264). If we make attention to the proof of the latter implication (*see* page 264), then we obtain that the accepting state s'_{Accept} is reachable in $\mathsf{closure}(Spec \times TP)$ by η''.

Finally, (1) as $\alpha_1, \ldots, \alpha_n$ are *visible* valued actions, then the sequence η'', which is exactly the same as σ, is a trace of $\mathsf{closure}(Spec \times TP)$, and (2) as η'' leads to the accepting state s'_{Accept}, then the trace σ of $\mathsf{closure}(Spec \times TP)$ is an accepting trace (*see* Definition 6.10, page 151), *i.e.* $\sigma \in ATraces(\mathsf{closure}(Spec \times TP))$.

Q.E.D.

A.2 Determinization

This section is devoted to the determinization of an IOSTS $\mathsf{closure}(SP)$ obtained from the previous step of our test generation algorithm (*see* Section A.1, page 246). This IOSTS may be non-deterministic due to several cause, for instance, we have considered a sub specification of the initially deterministic specification, or a specification obtained from the parallel composition of two deterministic systems executed in parallel where some communications are hidden. It is important to emphasize that nondeterminism is prohibited in testing, as test verdicts should not depend on internal choices of the tester. That is why this step of the test generation method is reserved for elimination of the nondeterminism from the IOSTS $\mathsf{closure}(SP)$. This means building another IOSTS, denoted $\mathsf{det}(\mathsf{closure}(SP))$, which has the same traces as $\mathsf{closure}(SP)$, and therefore the same traces as the product $SP = Spec \times TP$ (*see* Theorem A.3, page A.3), but without non-deterministic choices. Notice that elimination of nondeterminism from symbolic input/output transition systems is a difficult problem in general. In this section we propose a procedure that treats some typical situations of nondeterminism like one presented in Figures A.4(a) and A.7(a) (*see* pages 276 and 288).

Plan of the Section. The section is separated into three parts.

For the first two parts we consider an IOSTS $\mathcal{M}$ without internal actions which has $n \geq 2$ symbolic transitions $t_1, \ldots, t_n$ outgoing from the same location l, labeled with the same action a and guarded with Boolean expressions that are not mutually exclusive. This means that $\mathcal{M}$ has a *non-deterministic choice* in the location l.

In the first part of this section, we study the particular case of IOSTS with a sole non-deterministic choice, such that the symbolic transitions $t_1, \ldots, t_n$ involved into the non-deterministic choice of $\mathcal{M}$ have the *same set of assignments*. In order to solve non-determinism in $\mathcal{M}$, we introduce the operation of local determinization, which syntactically transforms the non-deterministic IOSTS $\mathcal{M}$ into a deterministic one leaving the trace semantics of $\mathcal{M}$ unchanged.

In the second part of the section, we generalize the particular case mentioned above, *i.e.* we consider the situation, where the symbolic transitions $t_1, \ldots, t_n$ involved into the non-deterministic choice of $\mathcal{M}$ have *different sets of assignments*. In this section we first show that the general case of local determinization of IOSTS can be reduced to the particular one, and then we propose an algorithm which (1) syntactically solves the local non-determinism in $\mathcal{M}$ and (2) preserves the semantics of $\mathcal{M}$.

In the third part, we propose a procedure that transforms general non-deterministic IOSTS $\mathcal{M}$ without internal action into a deterministic one $\mathsf{det}(\mathcal{M})$

using the local determinization algorithm given in the second part of this section. The two main purposes of this part are to study (1) the trace relation between $\mathcal{M}$ and $\mathsf{det}(\mathcal{M})$, and (2) conditions under which the procedure of global determinization terminates.

A.2.1 Local Determinization : Particular Case

This subsection is devoted to the problem of local determinization of IOSTS without internal actions. More precisely, consider an IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, (\Sigma^? \cup \Sigma^!), T\rangle$ with $n \geq 2$ symbolic transitions $t_1, \ldots, t_n$ starting in a location $l \in L$ and labeled with an action $a \in \Sigma$. Assume that the guards $G_1, \ldots, G_n$ of $t_1, \ldots, t_n$ respectively are not mutually exclusive. Thus in the location l the IOSTS $\mathcal{M}$ has a *non-deterministic choice* between its symbolic transitions $t_1, \ldots, t_n$ (the formal definition of a non-deterministic choice is given on page 277). We also assume that the sets of assignments of the symbolic transitions $t_1, \ldots, t_n$ involved into the non-deterministic choice of $\mathcal{M}$, are the *same*. The generalization of this particular situation is given in Section A.2.2 (*see* page 287).

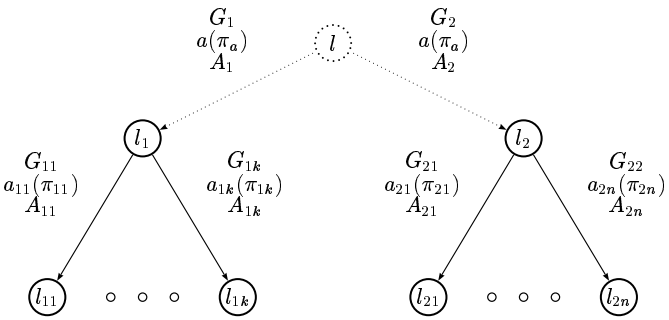
The aim of this subsection is to introduce a new operation on IOSTS $\mathcal{M}$ which solves the local non-determinism between $n \geq 2$ symbolic transitions $t_1, \ldots, t_n$ (in the case when the sets of assignments of $t_1, \ldots, t_n$ are *equal*), leaving the trace semantics of the given IOSTS $\mathcal{M}$ unchanged.

Example A.8 (Local Determinization of $\mathcal{A}$) This example explains at the intuitive level the general idea of the operation that transforms an IOSTS which has a non-deterministic choice in a location l, into a deterministic IOSTS. The formal definition of this operation is given on page 277.

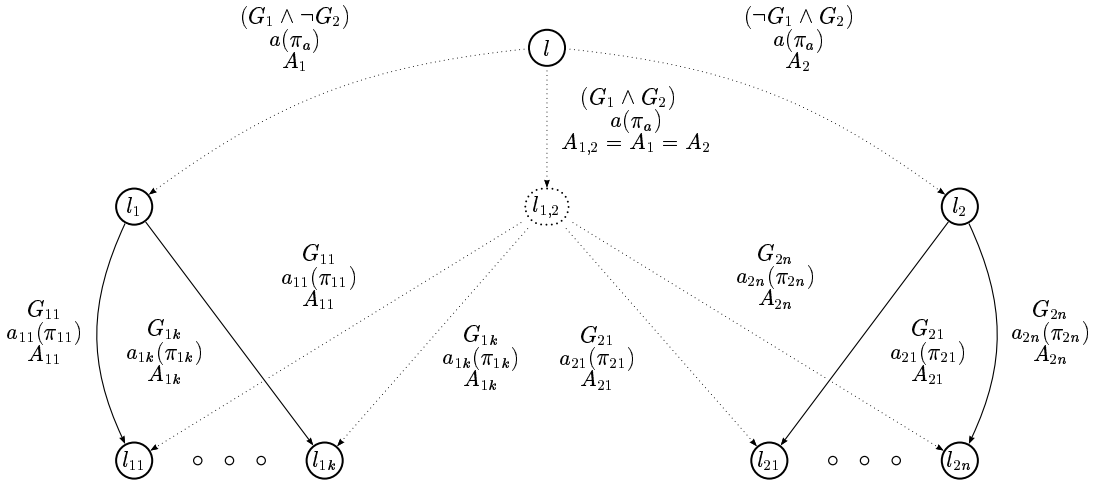
Consider an IOSTS $\mathcal{A}$, a fragment of which is shown on Figure A.4(a) (*see* page 276). We suppose that $\mathcal{A}$ has a non-deterministic choice between two symbolic transitions $t_1 = \langle l, a, \pi_a, G_1, A_1, l_1\rangle$ and $t_2 = \langle l, a, \pi_a, G_2, A_2, l_2\rangle$ which have the same sets of assignments, *i.e.* $A_1 = A_2$. These transitions are depicted as the bold edges in Figure A.4(a).

The local determinization of the given above IOSTS $\mathcal{A}$ is illustrated with Figure A.4 (*see* page 276). It consists in:

(1) splitting the symbolic transitions t_1 and t_2 (with guards G_1 and G_2 and assignments A_1 and A_2) into three: one leading to the location l_1 for the case when $(G_1 \wedge \neg G_2)$ holds, another leading to the location l_2 for the case when $(\neg G_1 \wedge G_2)$ holds, and the last leading to the new location $l_{1,2}$ for the case when $(G_1 \wedge G_2)$ holds. Due to the fact that the sets of assignments of t_1 and t_2 are the same, *i.e.* $A_1 = A_2$, we can easily determine that in the case when $G_1 \wedge G_2$ holds, *i.e.* both symbolic transitions t_1 and t_2 can be executed, the variables must be assigned in the same way. This means that



(a) A fragment of an IOSTS $\mathcal{A}$, where the transitions t_1 and t_2 which are involved into non-deterministic choice, are shown as dotted lines, and they have the *same set of assignments*, *i.e.* $A_1 = A_2$.



(b) A fragment of the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{A})$ obtained from $\mathcal{A}$ by the operation of local determinization (the new/modified locations and symbolic transitions are shown as dotted lines).

Figure A.4: Local Determinization of an IOSTS $\mathcal{A}$.

the set of assignments A' of the symbolic transition $t_{1,2}$ guarded with the Boolean expression $(G_1 \wedge G_2)$ is equal to $A_{1,2} = A_1 = A_2$.

(2) duplicating each symbolic transition outgoing from targets l_1 and l_2 of the symbolic transitions t_1 and t_2, and replacing the source of this new transition with the location $l_{1,2}$ which is the target of the symbolic transition $t_{1,2}$.

The result of the local determinization of $\mathcal{A}$ is shown in Figure A.4(b). □

Next, just before giving the formal definition of the operation of local determinization, we first introduce the notion of non-deterministic choice in an IOSTS.

Definition A.8 (Non-Deterministic Choice) Let $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ be an IOSTS, where $D = V \cup C \cup P$ and $\Sigma = \Sigma^? \cup \Sigma^!$. Then, we say that $\mathcal{M}$ has a *non-deterministic choice* in the location $l \in L$ if there exist $n \geq 2$ symbolic transitions $t_1, \ldots, t_n \in T$ with:

(1) same origin $l \in L$,

(2) same action $a \in \Sigma^! \cup \Sigma^?$ carrying a tuple of parameters π_a, and

(3) guards $G_1, \ldots, G_n$ such that there exists a pair of valuations $\langle \vartheta, \omega \rangle$, where $\vartheta \in \mathsf{DOM}(V \cup C)$ and $\omega_a \in \mathsf{DOM}(\pi_a)$, satisfying their conjunction, *i.e.* $\langle \vartheta, \omega_a \rangle \models (G_1 \wedge \ldots \wedge G_n)$.

Notice that it is possible to decide whether a given guard is satisfiable or not, as we made an assumption that all guards decorating symbolic transitions of an IOSTS are expressions in a decidable theory (*see* page 84).

□

Then, we present the operation which solves a non-determinism between $n \geq 2$ symbolic transitions of some IOSTS which are involved into a non-deterministic choice of this IOSTS.

Definition A.9 (Local Determinization of $\mathcal{M}$: Particular Case) Let

(1) $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ be an IOSTS with empty set of internal actions, *i.e.* $\Sigma^\tau = \emptyset$.

(2) $T_{l,a} \triangleq \{t_i = \langle l, a, \pi_a, G_i, A_i, l_i \rangle \mid t_i \in T\}$ be the set of symbolic transitions involved into a non-deterministic choice of $\mathcal{M}$ in a location l for an action a (*see* Definition A.8, page 277). Moreover, we assume that the sets of assignments of the symbolic transitions belonging to $T_{l,a}$ are the *same*, *i.e.* $\forall\, i, j \in [1, |T_{l,a}|]\ .\ [A_i = A_j]$.

(3) $L_{l,a} \triangleq \{l_i \in L \mid \exists\ t_i = \langle l, a, \pi_a, G_i, A_i, l_i\rangle \in T_{l,a}\}$ be the set of targets of the symbolic transitions belonging to $T_{l,a}$.

(4)[1] $2^{\{1,\ldots,|T_{l,a}|\}}$ be the set of all possible subsets constructed from the the set of the indexes $\{1,\ldots,|T_{l,a}|\}$ of the symbolic transitions $T_{l,a}$; and P be an element of this set $2^{\{1,\ldots,|T_{l,a}|\}}$.

The set $2^{\{1,\ldots,|T_{l,a}|\}}$ allows to obtain all possible mutually exclusive combinations between the guards of the symbolic transitions $T_{l,a}$ involved into the non-deterministic choice of $\mathcal{M}$.

Then, the *operation of local determinization* illustrated with Figure A.5 (*see* page 279) transforms the given IOSTS $\mathcal{M}$ into the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{M}) = \langle D, \Theta, L', \Sigma, T'\rangle$, where:

(1) $L' = (L \setminus L_{l,a}) \cup L'_{l,a}$, where $L'_{l,a} \triangleq \{l_P \mid P \in 2^{\{1,\ldots,|T_{l,a}|\}}\}$.

(2) $T' = (T \setminus T_{l,a}) \cup T'_{l,a} \cup (\ \bigcup_{P\in 2^{\{1,\ldots,|T_{l,a}|\}}} T_P\ )$, where

(a) $T'_{l,a}$ is a set of new symbolic transitions $t_P = \langle l, a, \pi_a, G_P, A_P, l_P\rangle$, where $P \in 2^{\{1,\ldots,|T_{l,a}|\}}$, with:

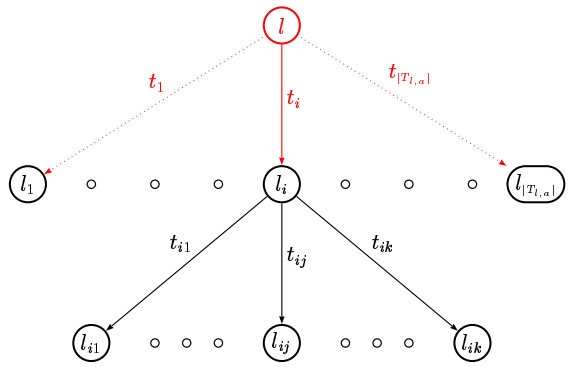
- $l_P \in L'_{l,a}$ is the target of t_P,
- $G_P = (\ \bigwedge_{i\in P} G_i\ ) \wedge (\ \bigwedge_{i\in(\{1,\ldots,|T_{l,a}|\}\setminus P)} \neg G_i\ )$ is the guard of t_P, and
- A_P is the set of assignments of t_P, which is the same as the set of assignments of all symbolic transitions belonging to $T_{l,a}$, *i.e.* $\forall\ i \in [1, |T_{l,a}|]\ .\ [A_P = A_i]$.

Notice that the guards of symbolic transitions belonging to $T'_{l,a}$ are mutually exclusive. Thus, by replacing $T_{l,a}$ by $T'_{l,a}$ in the set of all symbolic transitions T of $\mathcal{M}$ we solve the non-determinism between the symbolic transitions $T_{l,a}$ in $\mathcal{M}$.

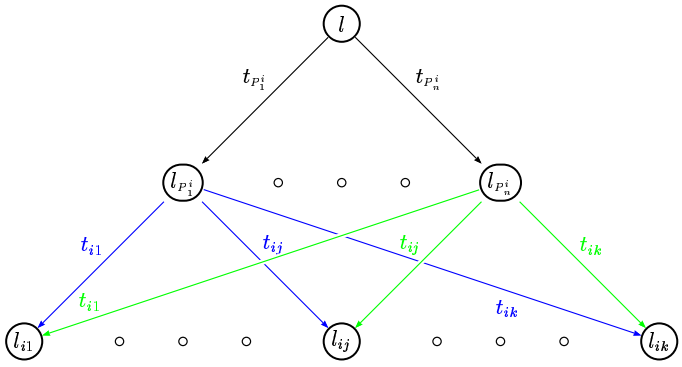
(b) T_P (*see* Figure A.5, page 279) is a set of symbolic transitions $\langle l_P, b_i, \pi_{b_i}, G_{ij}, A_{ij}, l_{ij}\rangle$ outgoing from the location $l_P \in L'_{l,a}$, where $P \in 2^{\{1,\ldots,|T_{l,a}|\}}$, such that there exists a symbolic transitions $t_i = \langle l, a, \pi_a, G_i, A_i, l'\rangle$, where $i \in P$, involved into a non-deterministic choice and t_i is followed by symbolic transitions $t_{ij} = \langle l', b_i, \pi_{b_i}, G_{ij}, A_{ij}, l_{ij}\rangle$ belonging to T of $\mathcal{M}$. Formally:

$$T_P = \{\langle l_P, b_i, \pi_{b_i}, G_{ij}, A_{ij}, l_{ij}\rangle \mid P \in 2^{\{1,\ldots,|T_{l,a}|\}} \wedge\ l_P \in L'_{l,a}\ \wedge \exists\ t_i = \langle l, a, \pi_a, G_i, A_i, l'\rangle \in T_{l,a},\ t_{ij} = \langle l', b_i, \pi_{b_i}, G_{ij}, A_{ij}, l_{ij}\rangle \in T\ .\ [\ (i \in P)\ ]\} \quad (A.17)$$

[1] Strictly speaking, the set $2^{\{1,\ldots,|T_{l,a}|\}}$ contains the empty set. However, in this thesis we assume that $\emptyset \notin 2^{\{1,\ldots,|T_{l,a}|\}}$.



(a) A fragment of an IOSTS $\mathcal{M}$, where the symbolic transitions $t_1, \ldots, t_{|T_{l,a}|}$ which are involved into non-deterministic choice, are shown in red color.



(b) A fragment of the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ obtained from $\mathcal{M}$ by the operation of local determinization. Here, for all n from 1 to $2^{|T_{l,a}|-1}$, P_n^i is a subset belonging to $2^{\{1,\ldots,|T_{l,a}|\}}$ such that $i \in P_n^i$.

Figure A.5: The example illustrating the operation of local determinization of an IOSTS $\mathcal{M}$.

□

Next, we illustrate the operation of local determinization defined above with a concrete example.

Example A.9 Consider a coffee machine depicted as the IOSTS $\mathcal{S}$ in Figure A.6(a) (*see* page 281). This coffee machine delivers either a coffee with milk or a coffee with sugar when the paid amount is strictly positive. Notice that in the location $l = Pay$ (shown in red) the IOSTS $\mathcal{S}$ has a non-deterministic choice between two symbolic transitions (shown in red) labeled with the same output action *Coffee*. It is important to notice that these symbolic transitions have the same set of assignments.

Then, in order to make $\mathcal{S}$ deterministic we *first* enumerate all symbolic transitions outgoing from the location $l = Pay$ and labeled with the action Coffee, *i.e.* we obtain the set:

$$T_{Pay,Coffee} = \{\ t_1 = \langle Pay,\, Coffee, \langle\rangle, \underbrace{vPaid \geq 3}_{G_1}, \{vPaid := vPaid\}, WaitMilk\rangle,$$
$$t_2 = \langle Pay,\, Coffee, \langle\rangle, \underbrace{vPaid \leq 3}_{G_2}, \{vPaid := vPaid\}, WaitSugar\rangle\ \}$$

This enumeration also gives the numbers to the targets of the symbolic transitions belonging to $T_{Pay,Coffee}$, *i.e.*

$$L_{Pay,Coffee} = \{l_1 = WaitMilk, l_2 = WaitSugar\}$$

Next, we the compute the set of possible subsets from the set of indexes $\{1, 2\}$ used to enumerate the symbolic transitions involved into non-deterministic choice of the IOSTS $\mathcal{S}$:

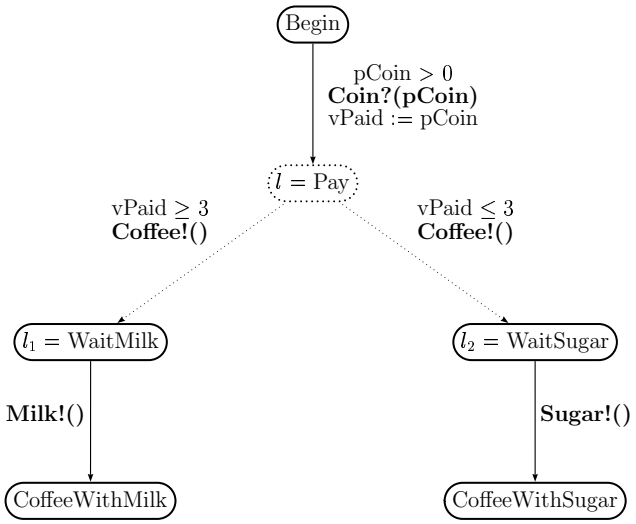
$$2^{\{1,2\}} = \{\{1\}, \{2\}, \{1, 2\}\}$$

Second, we modify the IOSTS $\mathcal{S}$ as follows:

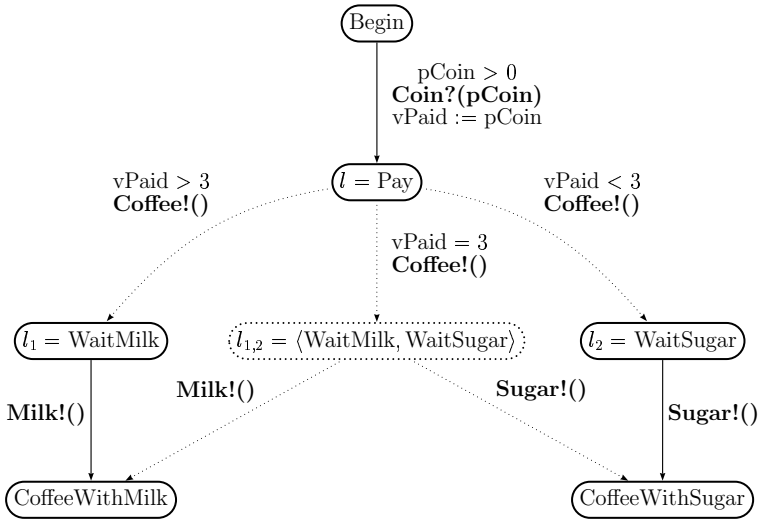
(1) From the set of locations of $\mathcal{S}$ (*see* Figure A.6(a), page 281) we remove all locations belonging to $L_{Pay,Coffee}$ and insert the following locations:

$$L'_{Pay,Coffee} = \{l_1 = WaitMilk, l_2 = WaitSugar, l_{1,2} = \langle WaitMilk, WaitSugar\rangle\}$$

which is computed from the sets $2^{\{1,2\}}$ and $L_{Pay,Coffee}$ (*see* the item (1) of Definition A.9 on page 277 for the detailed explanation). After this procedure we obtain that $\mathsf{det}_{\mathsf{loc}}(\mathcal{S})$ has the same set of locations as $\mathcal{S}$ except of the location $l_{1,2} = \langle WaitMilk, WaitSugar\rangle$ (*see* the location shown in blue on Figure A.6(b), page 281).



(a) The coffee machine $\mathcal{S}$ with the nondeterministic choice between two symbolic transitions shown as dotted lines.



(b) The coffee machine $\mathsf{det}_{\mathsf{loc}}(\mathcal{S})$ in which nondeterminism is solved by the operation of local determinization.

Figure A.6: An example illustrating the operation of local determinization.

(2) From the set of symbolic transitions of $\mathcal{S}$ (*see* Figure A.6(a), page 281) we remove all symbolic transitions belonging to $T_{Pay,Coffee}$ and add the following new symbolic transitions:

$$T'_{Pay,Coffee} =$$
$$\{\ t_1 = \langle Pay, Coffee, \langle\rangle, \underbrace{vPaid > 3}_{(G_1 \wedge \neg G_2)}, \{vPaid := vPaid\}, \underbrace{WaitMilk}_{l_1}\rangle,$$
$$t_2 = \langle Pay, Coffee, \langle\rangle, \underbrace{vPaid < 3}_{(\neg G_1 \wedge G_2)}, \{vPaid := vPaid\}, \underbrace{WaitSugar}_{l_2}\rangle,$$
$$t_{1,2} = \langle Pay, Coffee, \langle\rangle, \underbrace{vPaid = 3}_{(G_1 \wedge G_2)}, \{vPaid := vPaid\}, \underbrace{\langle WaitMilk, WaitSugar\rangle}_{l_{1,2}}\rangle\ \}$$

which is computed from the sets $2^{\{1,2\}}$, $L_{Pay,Coffee}$ and $T_{Pay,Coffee}$ (*see* the item (2.a) of Definition A.9 on page 277 for the detailed explanation). The result of this procedure you can see on Figure A.6(b), page 281 (*see* the blue edges outgoing from the location $l = Pay$).

(3) Finally, we duplicate two symbolic transitions outgoing from the locations $l_1 = WaitMilk$ and $l_2 = WaitSugar$ and change their origins with the location $l_{1,2} = \langle WaitMilk, WaitSugar\rangle$ (*see* Figure A.6(b), page 281).

The resulting IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{S})$ is deterministic in the sense of Definition 4.20 (*see* page 93). □

A.2.1.1 Traces of $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$: Particular Case

In this subsection we consider an IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T\rangle$ with set of states S and set of valued actions $\Lambda = \Lambda^? \cup \Lambda^!$. We suppose that $\mathcal{M}$ has a non-deterministic choice between $n \geq 2$ of its symbolic transitions $t_1 = \langle l, a, \pi_a, G_1, A_1, l_1\rangle, \ldots, t_n = \langle l, a, \pi_a, G_n, A_n, l_n\rangle$ (*see* Definition A.8, page 277) which have the *same set of assignments* (*i.e.* $\forall\ i, j \in [1, n]\ .\ [A_i = A_j]$). Finally, we also consider the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ obtained from $\mathcal{M}$ by the operation of local determinization (*see* Definition A.9, page 277).

The main purpose of this subsection is to prove that the operation of local determinization of the IOSTS $\mathcal{M}$ defined in the previous subsection (*see* page 277) preserves the semantics of the given IOSTS $\mathcal{M}$.

In order to prove the statement above we first show that for all valued actions α_a corresponding to an action a of $\mathcal{M}$, the IOSTS $\mathcal{M}$ moves from a state s to a state s_a by taking one of the symbolic transitions $t_1, \ldots, t_n$ labeled with the action a and involved into the non-deterministic choice of $\mathcal{M}$ *if and only if* the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ (*see* Definition A.9, page 277) moves from the same state s to a state s'_a by executing the same valued action α_a. Formally:

Lemma A.3 Let

(1) $\mathcal{M} = \langle D_{\mathcal{M}}, \Theta_{\mathcal{M}}, L_{\mathcal{M}}, l^0_{\mathcal{M}}, \Sigma_{\mathcal{M}}, T_{\mathcal{M}} \rangle$ be an IOSTS with set of states $S_{\mathcal{M}}$ and set of valued actions $\Lambda_{\mathcal{M}}$,

(2) $t_a = \langle l, a, \pi_a, G_a, A_a, l_a \rangle$ be a symbolic transition of $\mathcal{M}$ belonging to a set $T_{l,a} = \{t_1, \ldots, t_n\} \subseteq T_{\mathcal{M}}$ of $n \geq 2$ symbolic transitions involved into the non-deterministic choice of the IOSTS $\mathcal{M}$ in the location l for the action a (*see* Definition A.8, page 277), and

(3) $\mathsf{det}_{\mathsf{loc}}(\mathcal{M}) = \langle D_{\mathsf{det}_{\mathsf{loc}}(\mathcal{M})}, \Theta_{\mathsf{det}_{\mathsf{loc}}(\mathcal{M})}, L_{\mathsf{det}_{\mathsf{loc}}(\mathcal{M})}, l^0_{\mathsf{det}_{\mathsf{loc}}(\mathcal{M})}, \Sigma_{\mathsf{det}_{\mathsf{loc}}(\mathcal{M})}, T_{\mathsf{det}_{\mathsf{loc}}(\mathcal{M})} \rangle$ be the IOSTS with set of states $S_{\mathsf{det}_{\mathsf{loc}}(\mathcal{M})}$ and set of valued actions $\Lambda_{\mathsf{det}_{\mathsf{loc}}(\mathcal{M})}$, which is obtained from $\mathcal{M}$ by the operation of local determinization in the location l for the action a.

Then, for all states $s = \langle l, \vartheta \rangle$ and $s_a = \langle l_a, \vartheta_a \rangle$ belonging to $S_{\mathcal{M}}$ and all valued actions α_a belonging to $\Lambda_{\mathcal{M}}$ of the form $\langle a, \omega_a \rangle$, we obtain that: the relation $s \xrightarrow{\alpha_a} s_a$ holds in the IOSTS $\mathcal{M}$ *if and only if* there exists a state $s'_a = \langle l'_a, \vartheta_a \rangle \in S_{\mathsf{det}_{\mathsf{loc}}(\mathcal{M})}$, where $l'_a \in L_{\mathsf{det}_{\mathsf{loc}}(\mathcal{M})}$, such that the relation $s \xrightarrow{\alpha_a} s'_a$ holds in the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$. □

Proof First, notice that the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ has a set $T'_{l,a} = \{t'_1, \ldots, t'_{(2^n-1)}\}$ of $(2^n - 1)$ symbolic transitions which were obtained from the set $T_{l,a}$ of $\mathcal{M}$ by the item (2.a) of Definition A.9 (*see* page 277). It is important to emphasize that:

(1) the guards $G'_1, \ldots, G'_{(2^n-1)}$ of the symbolic transitions $t'_1, \ldots, t'_{(2^n-1)}$ belonging to $T'_{l,a}$ are mutually exclusive, and

(2) the disjunction of these guards are equal to the disjunction of the guards of the transitions $t_1, \ldots, t_n$, belonging to $T_{l,a}$ *i.e.* $\bigvee_{i \in [1,(2^n-1)]}(G'_i) = \bigvee_{i \in [1,n]}(G_i)$.

The items (1) and (2) imply the following fact: the pair of valuations $\langle \vartheta, \omega_a \rangle$ satisfies the guard G_a of the symbolic transition t_a of $\mathcal{M}$ *if and only if* this pair satisfies the guard G'_a of exactly one symbolic transition $t'_a \in T'_{l,a}$ of $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$.

Therefore, the IOSTS $\mathcal{M}$ may move from the state $s = \langle l, \vartheta \rangle$ to the state $s_a = \langle l_a, \vartheta_a \rangle$, where $\vartheta_a = A_a(\langle \vartheta, \omega_a \rangle)$, by executing the symbolic transition t_a. In the same time the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ may move from the same state s to the state $s'_a = \langle l'_a, \vartheta'_a \rangle$ by executing the symbolic transition t'_a.

Finally, as the symbolic transitions belonging to $T_{l,a}$ and $T'_{l,a}$ have the same set of assignments (*see* the hypothesis about $\mathcal{M}$ made on the page 282 and the item

(2.a) of Definition A.9, page 277), then the set of assignments A_a of $t_a \in T_{l,a}$ is also the set of assignments of $t'_a \in T'_{l,a}$. Thus, the valuation of variables and symbolic constants of $\mathcal{M}$ and $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ obtained after the execution of t_a and t'_a from the same state s are equal, *i.e.* $\vartheta_a = \vartheta'_a$. **Q.E.D.**

Second, we show that the operation of local determinization preserves the effects of sequences consisting of two symbolic transitions $t_a t_b$, where the first symbolic transition of this sequence is involved into a non-deterministic choice of $\mathcal{M}$ (*i.e.* t_a is equal to one of the symbolic transitions $t_1, \ldots, t_n$). Formally:

Lemma A.4 (Preserving Effects by the Operation of Local Determinization) Let $t_a = \langle l, a, \pi_a, G_a, A_a, l_a \rangle$ be one of the $n \geq 2$ symbolic transitions $t_1, \ldots, t_n$ which are involved into a non-deterministic choice of the IOSTS $\mathcal{M}$ in the location l for the action a (*see* Definition A.8, page 277). Assume that the symbolic transition t_a is followed by at least one symbolic transition $t_b = \langle l_a, b, \pi_b, G_b, A_b, l_b \rangle$ of the IOSTS $\mathcal{M}$.

Then, for all states $s = \langle l, \vartheta \rangle$ and $s_b = \langle l_b, \vartheta_b \rangle$ of the IOSTS $\mathcal{M}$ and all sequences $\alpha_a \alpha_b$ of valued actions, which has the form $\langle a, \omega_a \rangle \langle b, \omega_b \rangle$, we obtain that: the relation $s \xrightarrow{\alpha_a \alpha_b} s_b$ holds in the IOSTS $\mathcal{M}$ *if and only if* this relation holds in $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ obtained from $\mathcal{M}$ by the operation of local determinization (*see* page 277). □

Proof According to the item (4) of Definition 4.9 (*see* page 88), the statement of the lemma can be reformulated as follows: there exists a state $s_a = \langle l_a, \vartheta_a \rangle$ of $\mathcal{M}$ such that

$$\underbrace{\langle l, \vartheta \rangle}_{s} \xrightarrow{\langle a, \omega_a \rangle}_{t_a} \underbrace{\langle l_a, \vartheta_a \rangle}_{s_a} \xrightarrow{\langle b, \omega_b \rangle}_{t_b} \underbrace{\langle l_b, \vartheta_b \rangle}_{s_b}$$

holds in the IOSTS $\mathcal{M}$ *if and only if* there exist two symbolic transitions t'_a and t'_b of $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ and a state $s'_a = \langle l'_a, \vartheta'_a \rangle$ of $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$, where $\vartheta'_a = \vartheta_a$, such that

$$\underbrace{\langle l, \vartheta \rangle}_{s} \xrightarrow{\langle a, \omega_a \rangle}_{t'_a} \underbrace{\langle l'_a, \vartheta'_a \rangle}_{s'_a} \xrightarrow{\langle b, \omega_b \rangle}_{t'_b} \underbrace{\langle l_b, \vartheta_b \rangle}_{s_b}$$

holds in the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$.

(1) The proof of the statement:

> the relation $s \xrightarrow{\alpha_a}_{t_a} s_a$ holds in $\mathcal{M}$ *if and only if* the relation $s \xrightarrow{\alpha_a}_{t'_a} s'_a$ holds in $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$, where (a) t'_a is chosen exactly as in Lemma A.3 (*see* page 283), and (b) $s'_a = \langle l'_a, \vartheta'_a \rangle$ is a state of $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ such that $\vartheta'_a = \vartheta_a$

is similar to the proof of Lemma A.3 (*see* page 283).

(2) Next, we know that the symbolic transition t_a involved into the non-deterministic choice of $\mathcal{M}$ is followed by the symbolic transition t_b. Then, due to the item (2.b) of Definition A.9 (*see* page 277) the symbolic transition t'_a of $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ is also followed by the symbolic transition t'_b which was obtained from t_b by replacing its source l_a with the target of t'_a, *i.e.* by l'_a. This means that t'_a is equal to $\langle l'_a, b, \pi_b, G_b, A_b, l_b\rangle$.

Thus, the pair of valuations $\langle \vartheta_a, \omega_b\rangle$ satisfies the guard G_b of the symbolic transition t_b of the IOSTS $\mathcal{M}$ *if and only if* the pair of valuations $\langle \vartheta'_a, \omega_b\rangle$, where $\vartheta'_a = \vartheta_a$ (*see* the item (1)), satisfies the guard of the symbolic transition t'_b of the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$, which is exactly the same as the guard of t_b, *i.e.* G_b.

Finally, as t_b and t'_b have the same set of assignments A_b and the same target l_b, then both IOSTS $\mathcal{M}$ and $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ will be found in the same state $s_b = \langle l_b, \vartheta_b\rangle$ after the execution of the symbolic transitions t_b and t'_b from the states s_a and s'_a respectively.

The items (1) and (2) imply the statement formulated at the beginning of the proof. Therefore, the lemma is proved. **Q.E.D.**

Finally, we consider an IOSTS $\mathcal{M}$ with a non-deterministic choice between $n \geq 2$ symbolic transitions which have the *same* set of assignments; and show that the operation of local determinization presented on page 277 preserves the semantics of $\mathcal{M}$. Formally:

Theorem A.5 (Traces of $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$: Particular Case) Let $\mathcal{M} = \langle D, \Theta, L, l^0, (\Sigma^? \cup \Sigma^!), T\rangle$ be an IOSTS with a non-deterministic choice (*see* Definition A.8, page 277) between $n \geq 2$ symbolic transitions $T_{l,a} = \{t_1, \ldots, t_n\}$ which have a same set of assignments. Let also $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ be the IOSTS obtained from $\mathcal{M}$ by the operation of the local determinization (*see* Definition A.9, page 277). Then, these IOSTS $\mathcal{M}$ and $\mathsf{det}_{\mathsf{loc}}(\mathcal{M})$ have the same sets of traces, *i.e.*

$$\mathit{Traces}(\mathsf{det}_{\mathsf{loc}}(\mathcal{M})) = \mathit{Traces}(\mathcal{M}) \tag{A.18}$$

□

Proof At the beginning of the proof we formulate and show the following three statements.

(1) For all sequences of valued actions $\eta = \alpha_1 \ldots \alpha_k$:

$$\beta^{\rightarrow}: \quad s^0 \xrightarrow{\eta} s_k \quad = \quad s^0 \xrightarrow{\alpha_1}_{\tilde{t}_1} s_1 \ldots s_{k-1} \xrightarrow{\alpha_k}_{\tilde{t}_k} s_k \tag{A.19}$$

where

(a) s^0 is an initial state of $\mathcal{M}$, and $s_1, \ldots, s_k$ are states of $\mathcal{M}$, and

(b) for all i from 1 to k, $\tilde{t}_i$ is a symbolic transition of $\mathcal{M}$ which *does not belong* to $T_{l,a}$,

is a behavior of the IOSTS $\mathcal{M}$ *if and only if* Formula (A.20) is also a behavior of the IOSTS $\mathsf{det_{loc}}(\mathcal{M})$.

This statement follows directly from the fact that the operation of local determinization defined on page 277 does not modify the symbolic transitions of $\mathcal{M}$ that are not involved into a non-deterministic choice of $\mathcal{M}$.

(2) For every sequence of valued actions $\eta'\alpha_a$, where α_a is a valued action corresponding to a symbolic transition t_a *involved into the non-deterministic choice* of $\mathcal{M}$ (*i.e.* $t_a \in T_{l,a}$), the IOSTS $\mathcal{M}$ has the behavior:

$$\beta^{\rightarrow}_{\mathcal{M}} : \quad s^0 \xrightarrow{\eta'} s_{k-1} \xrightarrow{\alpha_a}_{t_a} s_k$$

where s^0 is an initial state of $\mathcal{M}$, and $s_1, \ldots, s_k$ are states of $\mathcal{M}$, *if and only if* the IOSTS $\mathsf{det_{loc}}(\mathcal{M})$ has the following behavior:

$$\beta^{\rightarrow}_{\mathsf{det_{loc}}(\mathcal{M})} : \quad s^0 \xrightarrow{\eta'} s_{k-1} \xrightarrow{\alpha_a}_{t'_a} s'_k$$

where s'_k is a state of $\mathsf{det_{loc}}(\mathcal{M})$ which has the same valuation of variables and symbolic constants as the state s_k of $\mathcal{M}$.

The statement follows from the item (1) above and Lemma A.3 (*see* page 283).

(3) For every sequence of valued actions $\eta'\alpha_a\alpha_b\eta''$ such that $\alpha_a\alpha_b$ is a sequence of two valued actions corresponding to a sequence of consecutive symbolic transitions t_at_b, where t_a is *involved into the non-deterministic choice* of $\mathcal{M}$ (*i.e.* $t_a \in T_{l,a}$),

$$\beta^{\rightarrow} : \quad s^0 \xrightarrow{\eta'} s_i \xrightarrow{\alpha_a\alpha_b} s_{i+2} \xrightarrow{\eta''} s_k \tag{A.20}$$

where s^0 is an initial state of $\mathcal{M}$, and $s_1, \ldots, s_k$ are states of $\mathcal{M}$, is a behavior of the IOSTS $\mathcal{M}$ *if and only if* Formula (A.20) is the behavior of the IOSTS $\mathsf{det_{loc}}(\mathcal{M})$.

This statement follows from the item (1) above and Lemma A.4 (*see* page 284).

Finally, from the statements proved in the items (1), (2) and (3) of the theorem, and the hypothesis saying that $\mathcal{M}$ does not have internal actions; we obtain Equality (A.18) using the definition of traces given on page 90. Therefore, the theorem is proved. **Q.E.D.**

A.2.2 Local Determinization : General Case

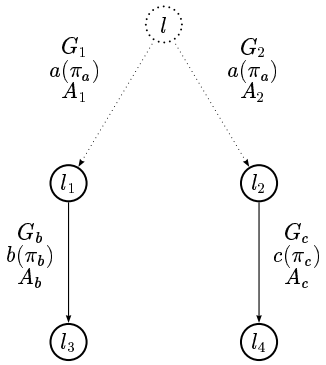
In this subsection we consider an IOSTS without internal actions that has a non-deterministic choice between $n \geq 2$ symbolic transitions (*see* Definition A.8, page 277) decorated with *different* sets of assignments. The purpose of this section is to attack the problem of the local determinization of this IOSTS, and to propose an algorithm which solves it. In this section we show that the problem of local determinization can be reduced to the one that was studied in the previous section (*see* page 275).

Example A.10 (Local Determinization of $\mathcal{B}$) Consider the IOSTS $\mathcal{B}$ depicted in Figure A.7(a) (*see* page 288). Assume that this IOSTS has a non-deterministic choice in the location l between two symbolic transitions t_1 and t_2 shown as the red edges on the figure. Notice that in this section we do not make any assumption about equality of the assignments of t_1 and t_2, *i.e.* we assume that $A_1 \neq A_2$. The aim of this example is to explain how to algorithmically solve the the non-determinism in the given IOSTS $\mathcal{B}$. Notice that our solution will be based on the operation of local determinization proposed in the previous section (*see* Definition A.9, page 277).

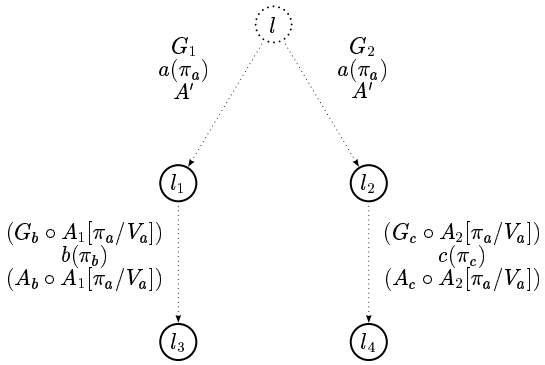
The general idea of the algorithm is the same as the one used to deteterminize the IOSTS $\mathcal{A}$ with two symbolic transitions involved into non-deterministic choice of $\mathcal{A}$, which have a same set of assignments (*see* Example A.8, page 275). We remind that this idea consists in splitting the symbolic transitions t_1 and t_2 (with guards G_1 and G_2 and assignments A_1 and A_2) involved into the non-deterministic choice of $\mathcal{B}$ into three: t'_1 with the guard $(G_1 \wedge \neg G_2)$, t'_2 with the guard $(\neg G_1 \wedge G_2)$, and $t_{1,2}$ with the guard $(G_1 \wedge G_2)$.

The main problem of splitting these symbolic transitions t_1 and t_2 is concerned with their sets of assignments A_1 and A_2 respectively. Indeed, in the case when $(G_1 \wedge G_2)$ holds, *i.e.* both symbolic transitions t_1 and t_2 can be executed, it is not possible to determine which set of assignments A_1 or A_2 should be performed (notice that A_1 is different from A_2). However, we can postpone the procedure of the variables assignment to the symbolic transitions that follow t_1 and t_2. We do it in three steps:

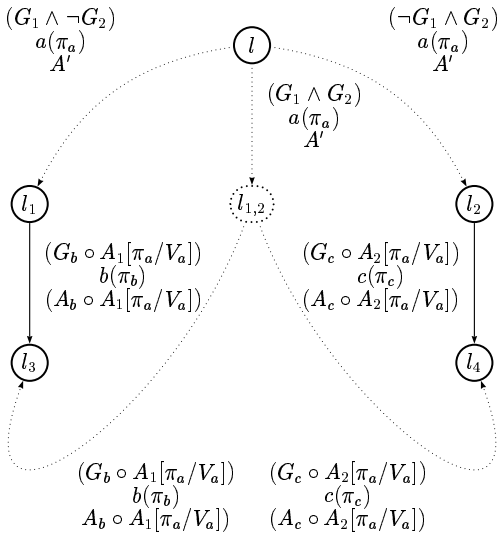
(1) Propagate the assignments of the symbolic transitions t_1 and t_2 onto the guards and assignments of the symbolic transitions following them using



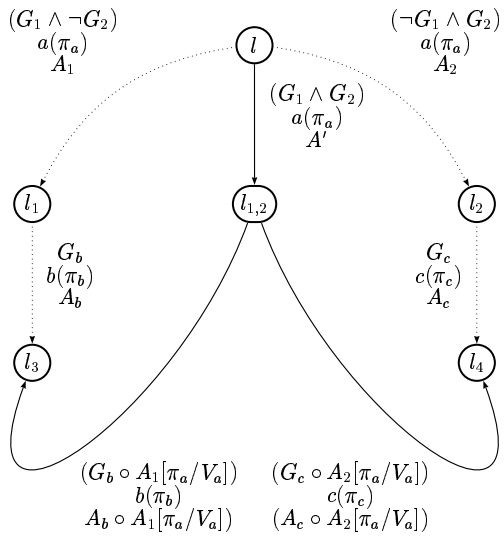
(a) A fragment of an IOSTS $\mathcal{B}$, where the transitions t_1 and t_2 which are involved into a non-deterministic choice, are shown as dotted lines, and they have *different sets of assignments*, *i.e.* $A_1 \neq A_2$.



(b) A fragment of the IOSTS $\mathcal{B}'$ obtained from $\mathcal{B}$ after the propagation of assignments A_1 and A_2 onto symbolic transitions t_b and t_c.



(c) A fragment of the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{B}')$ obtained from $\mathcal{B}'$ by the operation of local determinization (*see* Definition A.9, page 277).



(d) A fragment of the resulting IOSTS $\mathsf{det}(\mathcal{B})$ obtained from $\mathsf{det}_{\mathsf{loc}}(\mathcal{B}')$ by canceling the propagation of assignments (*see* Figure A.7(b)).

Figure A.7: Local Determinization of an IOSTS $\mathcal{B}$.

the operation of assignments propagation (*see* page 291). After this procedure, we obtain an IOSTS $\mathcal{B}'$ (*see* Figure A.7(b), page 288) in which the symbolic transitions corresponding to t_1 and t_2 are still involved into the non-deterministic choice (*see* the red edges of the figure), but they have the same set of assignments A'. This set of assignments A' is used to save the values of the parameters π_a carried by the action a into some new "observation variables", and keep the rest of the variables unchanged.

(2) Apply the operation of local determinization (*see* Definition A.9, page 277) to the IOSTS $\mathcal{B}'$, and obtain the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{B}')$ (*see* Figure A.7(c), page 288).

(3) In the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{B}')$, cancel the propagation of the assignments which was done at the first step, and obtain the resulting IOSTS $\mathsf{det}(\mathcal{B})$ shown on Figure A.7(d) (*see* page 288).

Notice that in practice the decomposition in the three steps explained above is not relevant as it is possible to transform directly the IOSTS $\mathcal{B}$ into the resulting IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{B}')$. Thus, in practice the local determinization (general case) should be done in one step. However, in theory we prefer to keep all these steps as they simplify very much the proof of the fact that the local determinization preserves the trace semantics of the given IOSTS $\mathcal{B}$ (*see* Theorem A.7, page 298). □

Plan of the Subsection. The aim of this subsection is to introduce an algorithm which allows to transform an IOSTS $\mathcal{M}$ with a non-deterministic choice between $n \geq 2$ symbolic transitions $t_1, \ldots, t_n$ with *different sets of assignments*, into the IOSTS $\mathsf{det}(\mathcal{M})$ in which this non-determinism between $t_1, \ldots, t_n$ is solved.

For this purpose we first define the operation transforming a given IOSTS into the IOSTS that contains variables used to memorize values of parameters carried by the actions of the given IOSTS (*see* Subsection A.2.2.1, page 290). The reason of introducing this operation is that, while performing the determinization of a given IOSTS, we need to conserve values of parameters in order to be able to postpone the effects of symbolic transitions involved into non-deterministic choices of IOSTS to symbolic transitions that follow them (*see* Subsection A.2.2.2, page 291). Next, we present the formal algorithm used for determinization of the IOSTS memorizing parameters that has a non-deterministic choice between n symbolic transitions with different sets of assignments (*see* Subsection A.2.2.3, page 295). Finally, we show that this algorithm preserves the trace semantics of the given IOSTS (*see* Theorem A.7, page 298).

A.2.2.1 IOSTS Memorizing Parameters

This subsection introduces the operation that allows to transform a given IOSTS $\widetilde{\mathcal{M}}$ into the IOSTS $\mathcal{M}$ conserving the values of all parameters carried by the actions of $\widetilde{\mathcal{M}}$. Formally:

Definition A.10 (IOSTS Memorizing Parameters) For an IOSTS $\widetilde{\mathcal{M}}$ with set of data $D = V \cup P \cup C$ and set of symbolic transitions T, we define the IOSTS $\mathcal{M}$ that *memorize parameters* P of $\widetilde{\mathcal{M}}$, as follows:

(1) For each parameter $p \in P$ we create a new variable $v_p \notin V$ such that $\mathsf{type}(p) = \mathsf{type}(v_p)$.

(2) For each symbolic transition $t = \langle l, a, \pi_a, G, A, l' \rangle \in T$ and each parameter $p \in P$, if p is a parameter carried by the action a, *i.e.* $p \in \pi_a$, then the set of assignments A is augmented with the new assignment $v_p := p$, where v_p is the variable corresponding to the parameter p.

This means that the value of each parameter from the tuple π_a is memorized in the variable corresponding to this parameter.

□

Observation A.4 (Sequences and Traces of the IOSTS $\mathcal{M}$ Memorizing Parameters of the IOSTS $\widetilde{\mathcal{M}}$) Two IOSTS $\widetilde{\mathcal{M}}$ and $\mathcal{M}$, where $\mathcal{M}$ is the IOSTS memorizing parameters of $\widetilde{\mathcal{M}}$ (*see* Definition A.10), have the same sets of sequences and traces, *i.e.*

$$Sequences(\widetilde{\mathcal{M}}) = Sequences(\mathcal{M}) \tag{A.21}$$
$$Traces(\widetilde{\mathcal{M}}) = Traces(\mathcal{M}) \tag{A.22}$$

□

Indeed, the IOSTS $\mathcal{M}$ is obtained from the given IOSTS $\widetilde{\mathcal{M}}$ by introducing new variables that are *observation variables*, as they are only defined in $\mathcal{M}$ and never used. *I.e.* they do not occur either (1) in the initial condition of $\mathcal{M}$ and in the guards of the symbolic transitions of $\mathcal{M}$ (notice that the initial condition as well as guards of $\mathcal{M}$ are the same as the initial condition and guards of $\widetilde{\mathcal{M}}$), or (2) on the right-hand sides of the assignments of the symbolic transitions of $\mathcal{M}$. Hence, these observation variables cannot influence the sets of sequences and traces of $\widetilde{\mathcal{M}}$.

A.2.2.2 Propagation of Assignments

In this subsection we consider a sequence δ consisting of two consecutive symbolic transitions of the IOSTS $\mathcal{M}$. The main purposes of the subsection are (1) to introduce the operation which postpones the effect of the first symbolic transition of δ to the symbolic transition that follows it, and (2) to show that this operation does not change the semantics of the sequence δ.

Definition A.11 (Propagation of Assignments) Let

$$\delta : \quad \underbrace{\langle l_1, a, \pi_a, G_a, A_a, l_2 \rangle}_{t_a} \underbrace{\langle l_2, b, \pi_b, G_b, A_b, l_3 \rangle}_{t_b}$$

be a sequence of two consecutive symbolic transitions of the IOSTS $\mathcal{M}$. Then, the *operation of propagation of assignments* transforms δ into the following sequence of symbolic transitions:

$$\mathsf{propagation}(\delta) : \underbrace{\langle l_1, a, \pi_a, G_a, A'_a, l_2 \rangle}_{t'_a} \underbrace{\langle l_2, b, \pi_b, (G_b \circ A_a[\pi_a/V_a]), (A_b \circ A_a[\pi_a/V_a]), l_3 \rangle}_{t'_b}$$

such that:

- $A'_a = \{v := v \mid v \in (V \setminus V_a)\} \cup \{v_i^a := \pi_i^a \mid v_i^a \in V_a \wedge \pi_i^a \in \pi_a\}$, where V_a is the set of variables used to memorize values of parameters π_a carried by the action a (*see* Definition A.10, page 290);
- $(G_b \circ A_a[\pi_a/V_a])$ is the composition of the guard G_b of t_b and the assignments A_a of t_a in which each occurrence of any parameter from π_a is replaced with its corresponding variable from the set V_a;
- $(A_b \circ A_a[\pi_a/V_a])$ is the composition between the assignments A_b of t_b and the assignments A_a of t_a, where each occurrence of any parameter from π_a in the right-hand side of A_a is replaced with its corresponding variable from the set V_a.

□

Next, we denote by $\mathsf{propagation}_\delta(\mathcal{M})$ the IOSTS obtained from the IOSTS $\mathcal{M}$ by replacing the sequence δ of two consecutive transitions with the sequence $\mathsf{propagation}(\delta)$.

Finally, we show that the operation of propagation of assignments preserves the effect of the sequence δ. Formally:

Theorem A.6 (Preserving Effects by Propagation of Assignments) Let

$$\delta : \quad \underbrace{\langle l_1, a, \pi_a, G_a, A_a, l_2 \rangle}_{t_a} \underbrace{\langle l_2, b, \pi_b, G_b, A_b, l_3 \rangle}_{t_b} \tag{A.23}$$

where $l_1 \neq l_2 \neq l_3$, be a sequence of two consecutive symbolic transitions of the IOSTS $\mathcal{M}$ that was obtained from the IOSTS $\widetilde{\mathcal{M}}$ by memorizing its parameters (*see* Definition A.10, page 290). Let also

$$\begin{array}{c} \mathsf{propagation}(\delta) : \\ \underbrace{\langle l_1, a, \pi_a, G_a, A'_a, l_2 \rangle}_{t'_a} \underbrace{\langle l_2, b, \pi_b, (G_b \circ A_a[\pi_a/V_a]), (A_b \circ A_a[\pi_a/V_a]), l_3 \rangle}_{t'_b} \end{array} \tag{A.24}$$

where $A'_a = \{v := v \mid v \in (V \setminus V_a)\} \cup \{v_i^a := \pi_i^a \mid v_i^a \in V_a \wedge \pi_i^a \in \pi_a\}$, be the sequence of the IOSTS $\mathsf{propagation}_\delta(\mathcal{M})$, *i.e.* $\mathsf{propagation}(\delta)$ is the result of propagation of assignments in δ (*see* Definition A.11, page 291).

Then, for all states s_1 and s_3 of the IOSTS $\mathcal{M}$, and for all sequences of valued actions of the form $\eta : \underbrace{\langle a, \omega_a \rangle}_{\alpha_a} \underbrace{\langle b, \omega_b \rangle}_{\alpha_b}$ we have that:
the relation $s_1 \xrightarrow{\eta} s_3$ (*see* the item (4) of Definition 4.9, page 88) holds in $\mathcal{M}$ *if and only if* the same relation holds in $\mathsf{propagation}_\delta(\mathcal{M})$. □

Proof In order to prove the theorem we have to show that there exists a state s'_2 of $\mathcal{M}$ such that $s_1 \xrightarrow{\alpha_a}_{t_a} s_2 \xrightarrow{\alpha_b}_{t_b} s_3$ holds in the IOSTS $\mathcal{M}$ *if and only if* there exists a state s'_2 of $\mathsf{propagation}_\delta(\mathcal{M})$ such that $s_1 \xrightarrow{\alpha_a}_{t'_a} s'_2 \xrightarrow{\alpha_b}_{t'_b} s_3$ holds in the IOSTS $\mathsf{propagation}_\delta(\mathcal{M})$. For more details see the item (4) of Definition 4.9 on page 88.

To show this statement ($\star$) we create an intermediate IOSTS $\mathcal{M}'$ which is the same as the IOSTS $\mathcal{M}$ except that:

(a) the set of locations L of $\mathcal{M}$ is augmented with a new location $l'_2 \notin L$,

(b) the alphabet of actions Σ of $\mathcal{M}$ is augmented with the internal action $\tau \notin \Sigma$, and

(c) instead of the sequence δ of $\mathcal{M}$ the intermediate IOSTS $\mathcal{M}'$ has the following sequence of three consecutive symbolic transitions:

$$\delta' : \underbrace{\langle l_1, a, \pi_a, G_a, A'_a, l_2 \rangle}_{t''_a} \underbrace{\langle l_2, \tau, \langle\rangle, true, A_a[\pi_a/V_a], l'_2 \rangle}_{t_\tau} \underbrace{\langle l'_2, b, \pi_b, G_b, A_b, l_3 \rangle}_{t''_b} \tag{A.25}$$

($\star\star$) It is not hard to check that the closure operation (*see* Definition A.7, page 260) applied to the IOSTS $\mathcal{M}'$ produces the IOSTS $\mathsf{propagation}_\delta(\mathcal{M})$, *i.e.* $\mathsf{closure}(\mathcal{M}') = \mathsf{propagation}_\delta(\mathcal{M})$.

Then, in order to show the statement ($\star$) we formulate and prove two equivalences (*see* the next two paragraphs) that imply ($\star$), and therefore, the whole theorem.

First, we prove that there exists a state $s_2 = \langle l_2, \vartheta_2 \rangle$ of $\mathcal{M}$ such that

$$\underbrace{\langle l_1, \vartheta_1 \rangle}_{s_1} \xrightarrow{\langle a, \omega_a \rangle} t_a \; \underbrace{\langle l_2, \vartheta_2 \rangle}_{s_2} \xrightarrow{\langle b, \omega_b \rangle} t_b \; \underbrace{\langle l_3, \vartheta_3 \rangle}_{s_3}$$

holds in $\mathcal{M}$ *if and only if* there exist states $s'_2 = \langle l_2, \vartheta'_2 \rangle$ and $s''_2 = \langle l'_2, \vartheta_2 \rangle$, (†) where $\vartheta'_2 \in \mathsf{DOM}(V \cup C)$, such that:

$$\underbrace{\langle l_1, \vartheta_1 \rangle}_{s_1} \xrightarrow{\langle a, \omega_a \rangle} t''_a \; \underbrace{\langle l_2, \vartheta'_2 \rangle}_{s'_2} \xrightarrow{\langle \tau, \langle\rangle \rangle} t_\tau \; \underbrace{\langle l'_2, \vartheta_2 \rangle}_{s''_2} \xrightarrow{\langle b, \omega_b \rangle} t''_b \; \underbrace{\langle l_3, \vartheta_3 \rangle}_{s_3}$$

holds in $\mathcal{M}'$. This equivalence (†) is proved in two steps:

(1) We first show that there exists the state s_2 of $\mathcal{M}$ such that $s_1 \xrightarrow{\alpha_a}_{t_a} s_2$ holds in $\mathcal{M}$ *if and only if* there exist two states s'_2 and s''_2 of $\mathcal{M}'$ such that $s_1 \xrightarrow{\alpha_a}_{t_a} s'_2 \xrightarrow{\alpha_\tau}_{t_\tau} s''_2$ holds in $\mathcal{M}'$.
Indeed,

   (a) the pair of valuations $\langle \vartheta_1, \omega_a \rangle$ satisfies the guard G_a of the symbolic transition t_a of the IOSTS $\mathcal{M}$ *if and only if* it satisfies the guard of the symbolic transition t''_a of the IOSTS $\mathcal{M}'$, which is exactly the same as the guard of t_a, *i.e.* G_a;

   (b) notice that as the symbolic transition t_τ is guarded with the Boolean expression *true*, then any valuations ϑ'_2 of variables and symbolic constants trivially satisfy the guard of t_τ.

   The items (a) and (b) imply that the symbolic transition t_a of $\mathcal{M}$ is executable from the state $s_1 = \langle l_1, \vartheta_1 \rangle$ *if and only if* the sequence $t''_a t_\tau$ of $\mathcal{M}'$ is executable from the same state s_1.

   Next, by executing the symbolic transition t_a from the state s_1 the IOSTS $\mathcal{M}$ moves to the state $s_2 = \langle l_2, \vartheta_2 \rangle$; and by executing the sequence $t''_a t_\tau$ from the state s_1 the IOSTS $\mathcal{M}'$ moves to the state $s''_2 = \langle l'_2, \vartheta_2 \rangle$. It is important to notice that the valuations of variables and symbolic constants of the states s_2 and s''_2 are identical. This is because: by the construction of the sequence δ' the assignments of the

sequence $t''_a t_\tau$ have the same effect as the assignments of the symbolic transition t_a.

The argumentation given above proves the item (1).

(2) Next we show that the relation $s_2 \xrightarrow{\alpha_b}_{t_b} s_3$ holds in $\mathcal{M}$ *if and only if* the relation $s''_2 \xrightarrow{\alpha_b}_{t_b} s_3$ holds in $\mathcal{M}'$.

Indeed, the pair of valuations $\langle \vartheta_2, \omega_b \rangle$ satisfies the guard G_b of the symbolic transition t_b of the IOSTS $\mathcal{M}$ *if and only if* it satisfies the guard of the symbolic transition t''_b of the IOSTS $\mathcal{M}'$, which is exactly the same as the guard of t_b, *i.e.* G_b. Then, as t_b and t''_b have the same set of assignments A_b and the same target l_3, then both IOSTS $\mathcal{M}$ and $\mathcal{M}'$ will be found in the same state $s_3 = \langle l_3, \vartheta_3 \rangle$ after executing the symbolic transition t_b (*resp.* t''_b) from the state s_2 (*resp.* s''_2). Therefore, the item (2) is proved.

The items (1) and (2) imply the equivalence (†).

Second, we prove that there exist states $s'_2 = \langle l_2, \vartheta'_2 \rangle$ and $s''_2 = \langle l'_2, \vartheta_2 \rangle$, where $\vartheta'_2 \in \mathsf{DOM}(V \cup C)$, such that:

$$\underbrace{\langle l_1, \vartheta_1 \rangle}_{s_1} \xrightarrow{\langle a, \omega_a \rangle}_{t''_a} \underbrace{\langle l_2, \vartheta'_2 \rangle}_{s'_2} \xrightarrow{\langle \tau, \langle\rangle \rangle}_{t_\tau} \underbrace{\langle l'_2, \vartheta_2 \rangle}_{s''_2} \xrightarrow{\langle b, \omega_b \rangle}_{t''_b} \underbrace{\langle l_3, \vartheta_3 \rangle}_{s_3} \tag{A.26}$$

(††) holds in $\mathcal{M}'$ *if and only if* the sequence of the following relations:

$$\underbrace{\langle l_1, \vartheta_1 \rangle}_{s_1} \xrightarrow{\langle a, \omega_a \rangle}_{t'_a} \underbrace{\langle l_2, \vartheta'_2 \rangle}_{s'_2} \xrightarrow{\langle b, \omega_b \rangle}_{t'_b} \underbrace{\langle l_3, \vartheta_3 \rangle}_{s_3} \tag{A.27}$$

holds in $\mathsf{propagation}_\delta(\mathcal{M})$.

First, by looking at Formula (A.25) we obtain that $t_\tau t''_b$ is a $\tau.a$-sequence of the IOSTS $\mathcal{M}'$ (*see* Definition A.6 page 259). Then, we apply the closure operation (*see* Definition A.7, page 260) to $\mathcal{M}'$, and, due to observation ($\star\star$), obtain the IOSTS $\mathsf{propagation}_\delta(\mathcal{M})$. It is important to notice that during this operation the $\tau.a$-sequence $t_\tau t''_b$ of $\mathcal{M}'$ was transformed into the symbolic transition t_b of $\mathsf{propagation}_\delta(\mathcal{M})$ by the collapsing operation defined on page 249.

Second, as $\mathsf{collapse}(t_\tau t''_b) = t_b$, then from Theorem A.1 (*see* page 252) we can deduce that $s'_2 \xrightarrow{\alpha_\tau}_{t_\tau} s''_2 \xrightarrow{\alpha_b}_{t''_b} s_3$ holds in $\mathcal{M}'$ *if and only if* $s'_2 \xrightarrow{\alpha_b}_{t_b} s_3$ holds in $\mathsf{propagation}_\delta(\mathcal{M})$.

Finally, the equivalence (††) is proved by noticing that the relation $s_1 \xrightarrow{\alpha_a}_{t''_a} s'_2$ which is the first step of Formula (A.26), holds in $\mathcal{M}$ *if and only if*

the relation $s_1 \xrightarrow{\alpha_a}_{t'_a} s'_2$ which is the first step of Formula (A.27), holds in $\mathsf{propagation}_\delta(\mathcal{M})$. This follows from the fact that the symbolic transition t''_a of δ' (*see* Formula (A.25), page 292) is the same as the symbolic transition t'_a of $\mathsf{propagation}(\delta)$ (*see* Formula (A.24), page 292).

The equivalences (†) and (††) imply the equivalence ($\star$). Therefore, the theorem is proved. **Q.E.D.**

A.2.2.3 Algorithm for Local Determinization (General Case) and Traces of $\mathsf{det}(\mathcal{M})$

In this section we consider an IOSTS $\widetilde{\mathcal{M}}$ without internal actions, and the IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T\rangle$ obtained from $\widetilde{\mathcal{M}}$ by Definition A.10 (*see* page 290). We also consider that the latter IOSTS has the set of states S and the set of valued actions $\Lambda = \Lambda^? \cup \Lambda^!$. Finally, we suppose that $\mathcal{M}$ has a non-deterministic choice between $n \geq 2$ symbolic transitions $t_1 = \langle l, a, \pi_a, G_1, A_1, l_1\rangle, \ldots, t_n = \langle l, a, \pi_a, G_n, A_n, l_n\rangle$ (*see* Definition A.8, page 277) such that (1) $t_1, \ldots, t_n$ can have *different sets of assignments*, and (2) for all i from 1 to n, l_i is different from l, *i.e.* the self-loops on the action a is forbidden in the location l. The last restriction is needed in order to be able to correctly perform the operation of assignments propagation on the first step of the algorithm of local determinization.

The two main purposes of this subsection are: (1) to propose an algorithm for solving non-determinism in the given IOSTS $\mathcal{M}$, and (2) to prove that this algorithm preserves the semantics of $\mathcal{M}$.

Algorithm A.1 (Local Determinization of $\mathcal{M}$: General Case) The algorithm for the local determinization of the IOSTS $\mathcal{M}$ consists of the three following steps:

Step I : Propagation of Assignments in the IOSTS $\mathcal{M}$. The purpose of this step it to modify the given IOSTS $\mathcal{M}$ such that the operation of local determinization (*see* Definition A.9, page 277) can be applied to it.

To reach this purpose we postpone the effects of the symbolic transitions $t_1, \ldots, t_n$ involved into the non-deterministic choice of $\mathcal{M}$, onto the symbolic transitions that follow them. This is done by replacing each sequence $t_i t_{ij}$ of $\mathcal{M}$, where $i = 1..n$ and $j = 1..k$, with the sequence $t'_{ij}t''_{ij} = \mathsf{propagation}(t_i t_{ij})$ (*see* Definition A.11, page 291).

It is important to emphasize that after the modification of the IOSTS $\mathcal{M}$ described in the item (1) above the semantics of $\mathcal{M}$ may change. This effect happens *only* in the case when there exists at least one symbolic transition which has the same target as one of the symbolic transitions $t_1, \ldots, t_n$, but (‡)

which is different from this symbolic transition (*see* Figure A.8, page 297). However, we fix this problem during the third step of the algorithm.

Step II : Local Determinization of the IOSTS $\mathcal{M}$. After the first step of the algorithm we obtain an IOSTS $\mathcal{M}'$ with the non-deterministic choice between the n symbolic transitions $t'_1, \ldots, t'_n$ corresponding to $t_1, \ldots, t_n$ of $\mathcal{M}$. Notice that all symbolic transitions $t'_1, \ldots, t'_n$ have the *same set of assignments*. Thus, at second step of this algorithm we can apply the operation of local determinization (*see* Definition A.9, page 277) to the IOSTS $\mathcal{M}'$. This operation returns the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{M}')$ where the non-deterministic choice between $t'_1, \ldots, t'_n$ is solved. Figures A.7(b) and A.7(c) (*see* pages 288 and 288 respectively) illustrate the second step of the algorithm.

Step III : Canceling the Propagation of Assignments made on Step I. The third step of the algorithm solves the problem mentioned as the remark (‡) in the first step of this algorithm. We remind that this problem is due to the modification of the semantics of the given IOSTS $\mathcal{M}$ by the assignments propagation performed for each symbolic transition involved into a non-deterministic choice of $\mathcal{M}$.

The solution of this problem is depicted in Figure A.7 (*see* page 288) and formalized below.

(1) We select all sequences $t''_i t'_j$ of $\mathsf{det}_{\mathsf{loc}}(\mathcal{M}')$, where $i = 1..n$ and $j = 1..k$, such that:

   (a) t''_i is the symbolic transition of $\mathsf{det}_{\mathsf{loc}}(\mathcal{M}')$ with the guard $G''_i = G'_i \wedge \neg(\ \bigvee_{(j \in [1,n]\ \wedge\ j \neq i)} G'_j\ )$, where G'_i is the guard of the symbolic transition t'_i involved into the non-deterministic choice of $\mathcal{M}'$, and

   (b) t'_{ij} is the symbolic transition following t'_i in $\mathcal{M}'$.

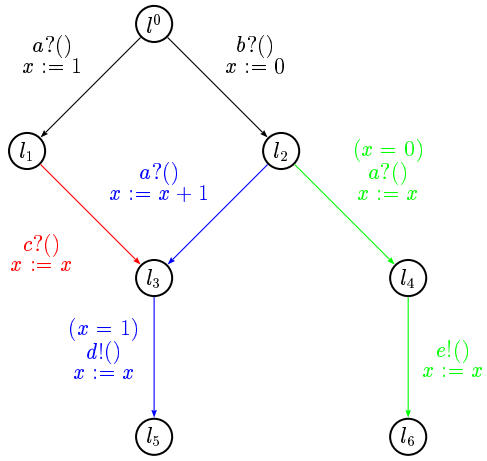
(2) Each sequence $t''_i t'_j$ of $\mathsf{det}_{\mathsf{loc}}(\mathcal{M}')$ selected by the item (1) above is replaced with the sequence $t'''_i t_{ij}$ such that $t''_i t'_j = \mathsf{propagation}(t'''_i t_{ij})$, where

   (a) t'''_i is the symbolic transition of $\mathsf{det}(\mathcal{M})$ which same as the symbolic transition t_i of $\mathcal{M}$ except of the guard which stays the same as the guard of t''_i of $\mathsf{det}_{\mathsf{loc}}(\mathcal{M}')$,

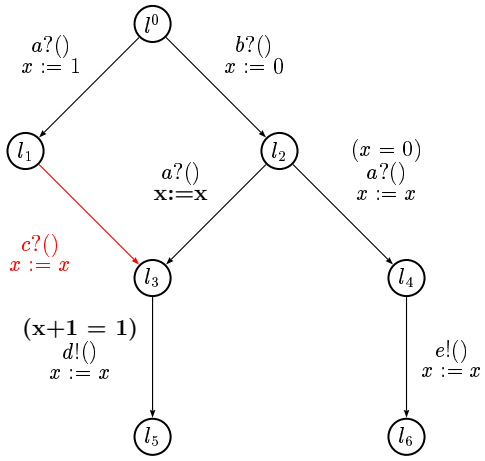
   (b) t_{ij} is a symbolic transition following t_i in $\mathcal{M}$.

At the end of this step we obtain the deterministic IOSTS $\mathsf{det}(\mathcal{M})$ which has the same semantics as the given IOSTS $\mathcal{M}$ (*see* Theorem A.7, page 298).

□



(a) The IOSTS $\mathcal{M}$ with the following set of traces: $\mathit{Traces}(\mathcal{M}) = \{a; b; ac; ba; \boldsymbol{acd}; bad; bae\}$.



(b) The IOSTS $\mathsf{propagation}(\mathcal{M})$ which is obtained from $\mathcal{M}$ by the assignments propagation (*see* Definition A.11, page 291) applied to the sequences of $\mathcal{M}$ shown in blue and green (*see* Figure A.8(a)). The set of traces of this IOSTS is following: $\mathit{Traces}(\mathsf{propagation}(\mathcal{M})) = \{a; b; ac; ba; bad; bae\}$. This set does not contain the trace $\boldsymbol{acd}$ of $\mathcal{M}$, thus the semantics of $\mathcal{M}$ and $\mathsf{propagation}(\mathcal{M})$ are different.

Figure A.8: The IOSTS $\mathcal{M}$ in which the operation of assignments propagation leads to a modification of the semantics of $\mathcal{M}$.

Finally, we state and prove the theorem about trace-equivalence between the given IOSTS $\mathcal{M}$ and the IOSTS $\mathsf{det}(\mathcal{M})$ obtained from $\mathcal{M}$ by the algorithm for the local determinization described above.

Theorem A.7 (Traces of $\mathsf{det}(\mathcal{M})$: General Case) Let $\mathcal{M}$ be the IOSTS without internal actions, wchich is obtained from an IOSTS $\widetilde{\mathcal{M}}$ by memorizing its parameters (*see* Definition A.10, page 290). Suppose that $\mathcal{M}$ has a non-deterministic choice between $n \geq 2$ symbolic transitions $t_1 = \langle l, a, \pi_a, G_1, A_1, l_1\rangle, \ldots, t_n = \langle l, a, \pi_a, G_n, A_n, l_n\rangle$ (*see* Definition A.8, page 277) such that:

(1) $t_1, \ldots, t_n$ can have *different sets of assignments*, and

(2) for all i from 1 to n, l_i is different from l, *i.e.* the self-loops on the action a is forbidden in the location l. This restriction is needed in order to be able to correctly perform the operation of assignments propagation on the first step of Algorithm A.1 (*see* page 295) .

Then, the resulting IOSTS $\mathsf{det}(\mathcal{M})$ obtained from the IOSTS $\mathcal{M}$ by the algorithm of local determinization given on page 295 has the same set of traces as $\mathcal{M}$, *i.e.*

$$\mathit{Traces}(\mathsf{det}(\mathcal{M})) = \mathit{Traces}(\mathcal{M})$$

□

Proof We assume that each symbolic transition t_i ($i = 1..n$) involved into the non-deterministic choice of $\mathcal{M}$ is followed by symbolic transitions $t_{i1}, \ldots, t_{ik}$. Then,

(1) During **Step I** of the algorithm for local determinization of $\mathcal{M}$ we replace each sequence $t_i t_{ij}$ ($i = 1..n$ and $j = 1..k$) with the sequence $t'_i t'_{ij} = \mathsf{propagation}(t_i t_{ij})$ and obtain the new IOSTS $\mathcal{M}'$. Then, due to Theorem A.6 (*see* page 292), we obtain that these replacements preserve the effects of the sequences $t_i t_{ij}$. Therefore, the semantics of these sequences is remining unchanged, but the semantics of the whole IOSTS $\mathcal{M}$ may change, *i.e.* it is possible that $\mathit{Traces}(\mathcal{M}) \neq \mathit{Traces}(\mathcal{M}')$

(2) During **Step II** of the algorithm we apply the operation of local determinization defined on page 277 to $\mathcal{M}'$ and obtain the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathcal{M}')$. Then, due to Lemma A.3 (*see* page 283), we have that this operation preserves the effects of each sequences of symbolic transitions $t'_i t'_{ij}$ of the IOSTS $\mathcal{M}'$, where $i = 1..n$ and $j = 1..k$. By Theorem A.5 (*see* page 285) we also get that $\mathit{Traces}(\mathsf{det}_{\mathsf{loc}}(\mathcal{M}')) = \mathit{Traces}(\mathcal{M}')$.

(3) During **Step III** of the algorithm we cancel the propagation of assignments made on **Step I** and obtain the IOSTS $\mathsf{det}(\mathcal{M})$. Then, due to Theorem A.6 (*see* page 292) we have that the effect of any sequences of two consecutive symbolic transitions is preserved by the operation of propagation of assignments. Thus, if in $\mathsf{det}_{\mathsf{loc}}(\mathcal{M}')$ we replace each sequence of symbolic transitions $t_i'''t'_{ij} = \mathsf{propagation}(t_i'''t_{ij})$ with the sequences $t_i'''t_{ij}$ (*see* **Step III** of Algorithm A.1), then the semantics of $\mathcal{M}$ will be preserved, *i.e.* $\mathit{Traces}(\mathsf{det}(\mathcal{M})) = \mathit{Traces}(\mathcal{M})$.

Q.E.D.

A.2.3 Traces and Accepting Traces of $\mathsf{det}(\mathsf{closure}(\mathit{Spec} \times \mathit{TP}))$

In this section we consider an IOSTS $\mathsf{closure}(SP)'$ with set of data $D = V \cup C \cup P$ and set of actions $\Sigma = \Sigma^? \cup \Sigma^!$, which was computed from the synchronous product $SP = \mathit{Spec} \times \mathit{TP}$ by the closure operation as it is explained in Section A.1 (*see* page 246).

Then, we augment the set of variables V of $\mathsf{closure}(SP)'$ with some observation variables used to memorize values of parameters P carried by the actions Σ of $\mathsf{closure}(SP)'$. *I.e.* we create an IOSTS $\mathsf{closure}(SP)$ memorizing parameters P of the IOSTS $\mathsf{closure}(SP)'$ (*see* Definition A.10, page 290). According to Observation A.4 (*see* page 290) the semantics of the IOSTS $\mathsf{closure}(SP)'$ and $\mathsf{closure}(SP)$ is the same, *i.e.* $\mathit{Traces}(\mathsf{closure}(SP)') = \mathit{Traces}(\mathsf{closure}(SP))$. Moreover, it is not hard to check that the IOSTS $\mathsf{closure}(SP)'$ and $\mathsf{closure}(SP)$ have the same set of accepting traces. (Notice that the accepting traces of $\mathsf{closure}(SP)'$ and $\mathsf{closure}(SP)$ are defined similarly as the accepting traces of the synchronous product SP, *see* Definition 6.10 on page 151).

Finally, we consider an IOSTS $\mathsf{det}(\mathsf{closure}(SP))$ computed from $\mathsf{closure}(SP)$ by the algorithm of local determinization presented in the previous subsection (*see* page 295). The aim of this section is to study the relationships between traces and accepting traces of $\mathsf{closure}(SP)$ and $\mathsf{det}(\mathsf{closure}(SP))$.

Traces of $\mathsf{det}(\mathsf{closure}(SP))$. Using Theorem A.7 (*see* page 298) we obtain that the algorithm of local determinization preserves the set of traces of the IOSTS $\mathsf{closure}(SP)$, *i.e.*

$$\mathit{Traces}(\mathsf{det}(\mathsf{closure}(SP))) = \mathit{Traces}(\mathsf{closure}(SP)) \tag{A.28}$$

Accepting Traces of $\mathsf{det}(\mathsf{closure}(SP))$. In this paragraph we first make an observation about the form of the locations of the IOSTS $\mathsf{det}(\mathsf{closure}(SP))$. Then

we use this observation in order to define the notion of accepting trace for $\mathsf{det}(\mathsf{closure}(SP))$. Finally, we prove that the IOSTS $\mathsf{det}(\mathsf{closure}(SP))$ has the same set of accepting traces as the IOSTS $\mathsf{closure}(SP)$.

Observation A.5 (The Form of Locations of $\mathsf{det}(\mathsf{closure}(Spec \times TP))$) All location of an IOSTS $\mathsf{det}(\mathsf{closure}(Spec \times TP))$ are of the form:

$$\langle\langle l^1_{Spec}, l^1_{TP}\rangle, \ldots, \langle l^n_{Spec}, l^n_{TP}\rangle\rangle$$

where for all i from 1 to $n \geq 1$, l^i_{Spec} and l^i_{TP} are locations of a specification *Spec* and a test purpose *TP* of *Spec* respectively. □

Intuitively, the set of accepting traces of the IOSTS $\mathsf{det}(\mathsf{closure}(SP))$ consists of the traces leading to states that correspond to the locations containing the word *Accept* in their names. Formally:

Definition A.12 (Set of Accepting Traces of $\mathsf{det}(\mathsf{closure}(Spec \times TP))$) Let

(1) $Spec \in \mathsf{SPEC}$ be a specification with set of locations L_{Spec},

(2) TP be a test purpose of *Spec* with set of locations L_{TP}, and

(3) $\mathsf{det}(\mathsf{closure}(SP))$ be an IOSTS obtained from the synchronous product $SP = Spec \times TP$ by the closure operation (*see* Definition A.7, page 260) and the algorithm of local determinization (*see* Algorithm A.1, page 295).

The IOSTS $\mathsf{det}(\mathsf{closure}(SP))$ has the set of locations L, the set of data $D = V \cup C \cup P$, the set of states S and the set of initial states $S^0 \subseteq S$.

Then, for the IOSTS $\mathsf{det}(\mathsf{closure}(SP))$ we define the *set of accepting traces* as follows:

$$ATraces(\mathsf{det}(\mathsf{closure}(SP))) \triangleq \tag{A.29}$$
$$\{\ \sigma \in Traces(\mathsf{det}(\mathsf{closure}(SP))) \mid \exists\ s^0 \in S^0,$$
$$\langle l_1, \ldots, \underbrace{\langle l_{Spec}, Accept\rangle}_{l_i}, \ldots, l_n\rangle \in L,$$
$$\vartheta \in \mathsf{DOM}(V \cup C),$$
$$s = \langle\langle l_1, \ldots, \underbrace{\langle l_{Spec}, Accept\rangle}_{l_i}, \ldots, l_n\rangle, \vartheta\rangle \in S\ .$$
$$[\ s^0 \stackrel{\sigma}{\Rightarrow} s\ ]\ \}$$

where $l_{Spec} \in L_{Spec}$ is a location of *Spec* and $Accept \in L_{TP}$ is the special location of *TP* (*see* the hypothesis (1) of Definition 6.8, page 147). □

Next, in order to prove the equality between sets of accepting traces of $\mathsf{closure}(SP)$ and $\mathsf{det}(\mathsf{closure}(SP))$, we first show the set of accepting traces of $\mathsf{closure}(SP)$ is equal to the set of accepting traces of the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathsf{closure}(SP))$ obtained from $\mathsf{closure}(SP)$ by the operation of local determinization. Formally:

Theorem A.8 (Accepting Traces of $\mathsf{det}_{\mathsf{loc}}(\mathsf{closure}(Spec \times TP))$) Let $\mathsf{closure}(Spec \times TP)$ be an IOSTS with a unique non-deterministic choice (*see* Definition A.8, page 277) between n symbolic transitions which have a *same set of assignments*.

Then, the resulting IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathsf{closure}(Spec \times TP))$ obtained from $\mathsf{closure}(Spec \times TP)$ by the operation of local determinization (*see* Definition A.9, page 277) has the same set of of accepting traces traces as $\mathsf{closure}(Spec \times TP)$, *i.e.*

$$ATraces(\mathsf{closure}(Spec \times TP)) = ATraces(\mathsf{det}_{\mathsf{loc}}(\mathsf{closure}(Spec \times TP)))$$

□

Proof

(⊆) First, we show that each accepting trace of $\mathsf{closure}(Spec \times TP)$ is also an accepting trace of $\mathsf{det}_{\mathsf{loc}}(\mathsf{closure}(Spec \times TP))$.

Consider an arbitrary accepting trace $\sigma = \alpha_1 \dots \alpha_k$ belonging to $ATraces(\mathsf{closure}(Spec \times TP))$ and leading to an accepting state $s_{Accept} = \langle\langle l, Accept\rangle, \vartheta\rangle$, where $\langle l, Accept\rangle \in L_{\mathsf{closure}(Spec \times TP)}$ and $\vartheta \in \mathsf{DOM}(V_{\mathsf{closure}(Spec \times TP)} \cup C_{\mathsf{closure}(Spec \times TP)})$.

Due to Definition A.7 (*see* page 260) the IOSTS $\mathsf{closure}(Spec \times TP)$ does not have any internal actions. Thus, the given accepting trace σ is also a sequence (*see* Definition 4.12, page 90) of $\mathsf{closure}(Spec \times TP)$ leading to the same accepting state s_{Accept}.

Next, we consider two cases below.

(1) The sequence σ corresponds to one of the following behaviors of $\mathsf{closure}(Spec \times TP)$:

(a) $\beta_1^{\rightarrow}$: $s^0 \xrightarrow{\sigma} s_k = s^0 \xrightarrow{\alpha_1}_{t_1} s_1 \dots s_{k-1} \xrightarrow{\alpha_k}_{t_k} s_{Accept}$, where

- s^0 is an initial state of $\mathsf{closure}(Spec \times TP)$,
- $s_1, \dots, s_{k-1}$ are states of $\mathsf{closure}(Spec \times TP)$, and
- $t_1, \dots, t_k$ are symbolic transition of $\mathsf{closure}(Spec \times TP)$ which *are not involved* into the (unique) non-deterministic choice of $\mathsf{closure}(Spec \times TP)$.

(b) $\beta_2^{\rightarrow}$: $s^0 \xrightarrow{\sigma'} s_i \xrightarrow{\alpha_{i+1}\alpha_{i+2}} s_{i+2} \xrightarrow{\sigma''} s_{Accept}$, where

- s^0 is an initial state of $\mathsf{closure}(Spec \times TP)$,
- $s_1, \ldots, s_{k-1}$ are states of $\mathsf{closure}(Spec \times TP)$, and
- $\sigma = \sigma'\alpha_{i+1}\alpha_{i+2}\sigma''$, where $\alpha_{i+1}\alpha_{i+2}$ is a sequence of two valued actions corresponding to a sequence of the consecutive symbolic transitions $t_{i+1}t_{i+2}$, where t_{i+1} is *involved* into the non-deterministic choice of $\mathsf{closure}(Spec \times TP)$.

As all symbolic transitions involved into the unique non-deterministic choice of $\mathsf{closure}(Spec \times TP)$ have a same set of assignments (*see* the hypothesis of the theorem), then we can use Theorem A.5 (*see* page 285). Due to the statements formulated as the items (1) and (3) in the proof of this theorem, we obtain that the behaviors $\beta_1^{\rightarrow}$ and $\beta_2^{\rightarrow}$ shown above are also behaviors of the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathsf{closure}(Spec \times TP))$. Thus, the state s_{Accept} is reachable in $\mathsf{det}_{\mathsf{loc}}(\mathsf{closure}(Spec \times TP))$ by the sequence σ. Finally, according to Definitions 4.12, 4.14 and A.12 (*see* pages 90, 90 and 300) the sequence σ is also an accepting trace of $\mathsf{det}_{\mathsf{loc}}(\mathsf{closure}(Spec \times TP))$.

(2) The sequence σ corresponds to the following behaviors of $\mathsf{closure}(Spec \times TP)$:

$$\beta^{\rightarrow}_{\mathsf{closure}(Spec \times TP)} : \quad s^0 \xrightarrow{\sigma'} \underbrace{\langle l_{k-1}, \vartheta_{k-1}\rangle}_{s_{k-1}} \xrightarrow{\alpha_k = \langle a, \omega\rangle}_{t_k} \underbrace{\langle\langle l, Accept\rangle, \vartheta\rangle}_{s_{Accept}} \quad \text{(A.30)}$$

where

- s^0 is an initial state of $\mathsf{closure}(Spec \times TP)$,
- $s_1, \ldots, s_{k-1}$ are states of $\mathsf{closure}(Spec \times TP)$, and
- $\sigma = \sigma'\alpha_k$, where α_k is a valued action corresponding to a symbolic transition t_k *involved* into the non-deterministic choice between $p \geq 2$ symbolic transitions $t_1, \ldots, t_k, \ldots, t_n$ of $\mathsf{closure}(Spec \times TP)$.

In this case, due to the statement formulated as the item (2) in the proof of Theorem A.5, page 285 (which we can use due to the hypothesis about common set of assignments for all symbolic transitions involved into the unique non-deterministic choice of $\mathsf{closure}(Spec \times TP)$) we get that the IOSTS $\mathsf{det}_{\mathsf{loc}}(\mathsf{closure}(Spec \times TP))$ has the behavior:

$$\beta^{\rightarrow}_{\mathsf{det}_{\mathsf{loc}}(\mathsf{closure}(Spec \times TP))} : \quad s^0 \xrightarrow{\sigma'} \underbrace{\langle l_{k-1}, \vartheta_{k-1}\rangle}_{s_{k-1}} \xrightarrow{\alpha_k = \langle a, \omega\rangle}_{t'_k} \underbrace{\langle\overbrace{\langle l_1, \ldots, l_i, \ldots, l_n\rangle}^{l'}, \vartheta\rangle}_{s'_{Accept}}$$

Next, we show that there exists i-th member ($i = 1..n$) of the location l' of $\mathsf{det_{loc}}(\mathsf{closure}(Spec \times TP))$ such that l_i is the accepting location $\langle l, Accept\rangle$ of $\mathsf{closure}(Spec \times TP)$.

Indeed, first notice that t'_k is one of the symbolic transitions of $\mathsf{det_{loc}}(\mathsf{closure}(Spec \times TP))$ obtained from the symbolic transitions $t_1, \ldots, t_k, \ldots, t_n$ involved into the non-deterministic choice of $\mathsf{closure}(Spec \times TP)$ by the operation of local determinization (*see* the item (2.a) of Definition A.9, page 277). Moreover, t'_k leads to a location of the form $l' = \langle l_1, \ldots, l_i, \ldots, l_n\rangle$ of $\mathsf{det_{loc}}(\mathsf{closure}(Spec \times TP))$ that is the tuple of targets of some symbolic transitions involved into the non-deterministic choice of $\mathsf{closure}(Spec \times TP)$, whose guards are satisfied by the pair of valuations $\langle \vartheta_{k-1}, \omega\rangle$. As we know that $\langle \vartheta_{k-1}, \omega\rangle$ satisfies the guard of the symbolic transition t_k of $\mathsf{closure}(Spec \times TP)$ (*see* Formula (A.30) and Definition 4.7 on page 86) which leads to the accepting location $\langle l, Accept\rangle$, and involved into the non-deterministic choice of $\mathsf{closure}(Spec \times TP)$, then $\langle l, Accept\rangle$ is the i-th member (for some $i = 1..n$) of the tuple l'.

Then, as the location $l' = \langle l_1, \ldots, \underbrace{\langle l, Accept\rangle}_{l_i}, \ldots, l_n\rangle$ contains the word $Accept$, then any state s'_{Accept} corresponding to this location is an accepting state of $\mathsf{det_{loc}}(\mathsf{closure}(Spec \times TP))$. Therefore, due to Definitions 4.12, 4.14 and A.12 (*see* pages 90, 90 and 300) the sequence σ corresponding to the behavior $\beta^{\rightarrow}_{\mathsf{det_{loc}}(\mathsf{closure}(Spec \times TP))}$ of $\mathsf{det_{loc}}(\mathsf{closure}(Spec \times TP))$ and leading to the accepting state s'_{Accept} is the accepting trace of $\mathsf{det_{loc}}(\mathsf{closure}(Spec \times TP))$.

The items (1) and (2) imply that $\sigma \in ATraces(\mathsf{det_{loc}}(\mathsf{closure}(Spec \times TP)))$.

($\supseteq$) The proof of the second inclusion, *i.e.* the proof of the fact that each accepting trace of $\mathsf{det_{loc}}(\mathsf{closure}(Spec \times TP))$ is also an accepting trace of $\mathsf{closure}(Spec \times TP)$, can be done by analogy with the proof of the first inclusion by using items (1), (2) and (3) of the demonstration of Theorem A.5 (*see* page 285).

Q.E.D.

Finally, we state and prove the theorem about equality between sets of accepting traces of $\mathsf{closure}(SP)$ and $\mathsf{det}(\mathsf{closure}(SP))$.

Theorem A.9 (Accepting Traces of $\mathsf{det}(\mathsf{closure}(Spec \times TP))$) The result of Algorithm A.1 (*see* page 295) applied to an IOSTS $\mathsf{closure}(Spec \times TP)$ with

unique non-deterministic choice (*see* Definition A.8, page 277), has the same set of accepting traces traces as $\mathsf{closure}(Spec \times TP)$, *i.e.*

$$ATraces(\mathsf{closure}(Spec \times TP)) = ATraces(\mathsf{det}(\mathsf{closure}(Spec \times TP)))$$

□

Proof The proof of the theorem is similar to that of Theorem A.7 (*see* page 298) except of (a) reasoning about accepting traces and not traces; and (b) using Theorem A.8 instead of Theorem A.5 in the item (2) of the proof (*see* page 298). **Q.E.D.**

A.2.4 Global Determinization

This section describes a simple procedure for the global determinization of an IOSTS without internal actions. The general idea is to iterate through all locations of the IOSTS, and each time when a non-deterministic choice (*see* Definition A.8, page 277) is detected, to apply the algorithm of local determinization defined in previous section (*see* Algorithm A.1, page 295). Formally:

Procedure A.1 (Global Determinization of $\mathcal{M}$) Let $\widetilde{\mathcal{M}}$ be an IOSTS without internal actions, and $\mathcal{M} = \langle D, \Theta, L, l^0, (\Sigma^? \cup \Sigma^!), T \rangle$ be the IOSTS memorizing parameters of $\widetilde{\mathcal{M}}$ (*see* Definition A.10, page 290). Then, in order to make the IOSTS $\mathcal{M}$ deterministic in the sense of Definition 4.20 (*see* page 93), we perform the following steps:

(1) From the set of locations L of $\mathcal{M}$ construct the subset $L' \subseteq L$ of all locations $l \in L$ which are origins of symbolic transitions involved into non-deterministic choices (*see* Definition A.8, page 277).

(2) While the set L' is not empty, do:

   (a) For each location l belonging to L', perform the algorithm of local determinization presented on page 295.

   (b) At the end of the loop described in the item (a), we obtain a new (possibly non-deterministic) IOSTS $\mathcal{M}'$ for which we recompute L'.

□

It is important to notice that Procedure A.1 presented above does not always terminate. However, if this procedure does terminate, it clearly produces an IOSTS without non-deterministic choices, *i.e.* a deterministic IOSTS in sense of

Definition 4.20 given on page 93. The termination problem is strongly connected with the existence of cycles in IOSTS.

Indeed, let us consider the subclass of acyclic IOSTS (an example of these IOSTS is shown in Figure A.6(a), page 281). Then, for any IOSTS $\mathcal{M}$ belonging to this subclass, we can guarantee that the procedure of global determinization terminates. Indeed, first we remind that the algorithm of local determinization (*see* Algorithm A.1, page 295) propagates a non-determinism of $\mathcal{M}$ only forward. Therefore, by iterating through all locations involved into non-deterministic choices of $\mathcal{M}$ and applying Algorithm A.1, we obtain a deterministic IOSTS $\mathcal{M}$.

However, if the IOSTS $\mathcal{M}$ contains cycles, it is possible that Procedure A.1 does not terminates. An example of such IOSTS is shown on Figure A.9 (*see* page 305) below and explained in the item (1) of Example A.12 (*see* page 307).

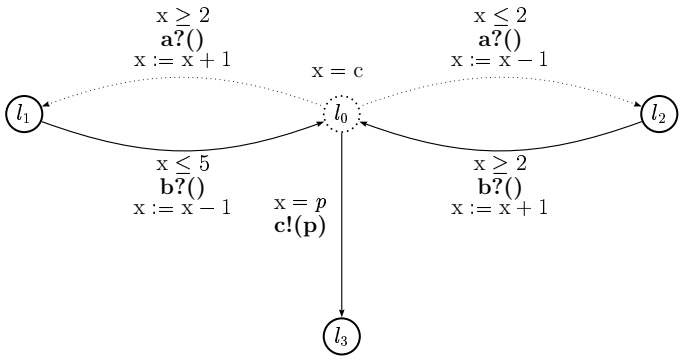


Figure A.9: An IOSTS $\mathcal{A}$ which does not belong to the class of deterministic IOSTS with lookahead k for any $k \in \mathbb{N}$.

Due to the fact that the subclass of acyclic IOSTS is very limited, our purpose is to find another subclass (1) which is wider that the subclass of acyclic IOSTS, and (2) for which Procedure A.1 terminates. The idea about this subclass was inspired from the paper [Angluin, 1982]. By analogy with this paper the subclass studied in the rest of this section is called *deterministic IOSTS with lookahead* k.

Plan of the Section. At the beginning of this section we introduce the subclass of IOSTS with lookahead k. Then, we show that the problem of checking whether an IOSTS without internal actions belongs to this class or not is decidable. Finally, we prove that the procedure for global determinization terminates for all IOSTS belonging to the subclass of IOSTS with lookahead k.

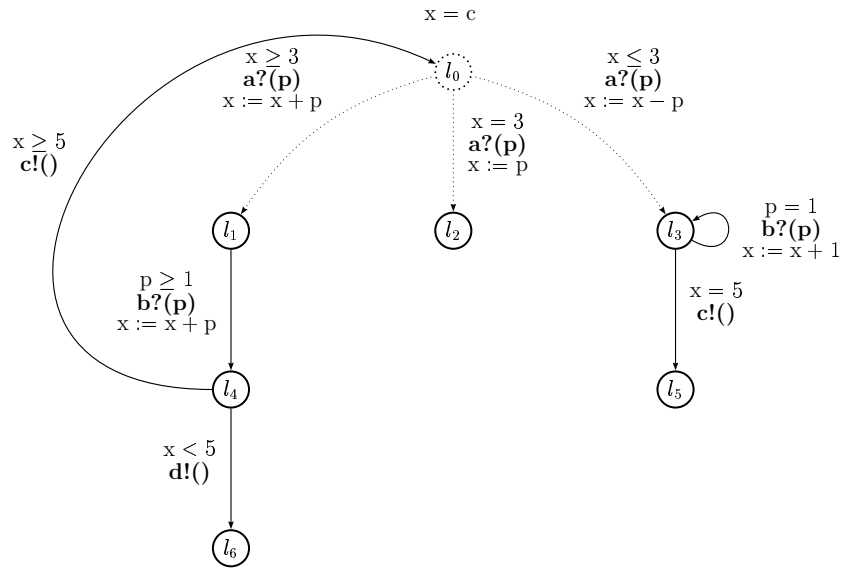


Figure A.10: An example of deterministic IOSTS $\mathcal{B}$ with lookahead 3.

A.2.4.1 Deterministic IOSTS with Lookahead k

Before giving the formal definition of a deterministic IOSTS with lookahead k, we introduce two intermediate notions defined on the syntax level of IOSTS. Let us consider an IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, (\Sigma^? \cup \Sigma^!), T \rangle$ without internal actions, *i.e.* $\Sigma^\tau = \emptyset$, then:

Definition A.13 (l after ϱ) The set of locations in which the IOSTS $\mathcal{M}$ can be after executing a sequence of input/output actions $\varrho = a_1 \dots a_k \in (\Sigma^? \cup \Sigma^!)^*$ from a location $l \in L$, is defined as:

$$(l \text{ after } \varrho) \triangleq \{\, l' \in L \mid \varrho = a_1 \dots a_k \wedge \exists\, l_1, \dots, l_{k-1} \,.\, [l \overset{a_1}{\hookrightarrow} l_1 \dots l_{k-1} \overset{a_k}{\hookrightarrow} l'] \,\} \quad \text{(A.31)}$$

where $l \overset{a}{\hookrightarrow} l'$ denotes a symbolic transition of $\mathcal{M}$. □

Definition A.14 (k**-Follower)** Let k be a fixed nonnegative integer. Then, a sequence of input/output actions $\varrho \in (\Sigma^? \cup \Sigma^!)^k$ is said to be a *k-follower* of the location $l \in L$ in the IOSTS $\mathcal{M}$ if the set of locations after executing the sequence ϱ from the location l of $\mathcal{M}$ is not empty, *i.e.* $(l \text{ after } \varrho) \neq \emptyset$. □

Example A.11 To illustrate the definition of a k-follower given above, we use the IOSTS $\mathcal{B}$ depicted in Figure A.10. The IOSTS $\mathcal{B}$ has the following alphabet of input/output actions: $(\Sigma^? \cup \Sigma^!) = (\{a, b\} \cup \{c, d\})$. Then, the sequence:

- $abcabb$ is a 6-follower of the location l_0,
- bc is a 2-follower of locations l_1 and l_3,
- d is a 1-follower of location l_4.

At the end, we notice that each location of the IOSTS $\mathcal{B}$ has exactly one 0-follower, namely, the empty string ε. This observation is valid for any arbitrary IOSTS. □

Next, we formally define a deterministic IOSTS with lookahead k and illustrate this definition with an example.

Definition A.15 (Deterministic IOSTS with Lookahead k) An IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, (\Sigma^? \cup \Sigma^!), T \rangle$ without internal actions is defined to be:

(1) *deterministic with lookahead* 0 if $\mathcal{M}$ does not contain any non-deterministic choice in the sense of Definition A.8 (*see* page 277), and

(2) *deterministic with lookahead* $k \geq 1$ if:

for any pair of distinct symbolic transitions $t_1 = \langle l, a, \pi_a, G_1, A_1, l_1 \rangle$ and $t_2 = \langle l, a, \pi_a, G_2, A_2, l_2 \rangle$ involved into a non-deterministic choice of $\mathcal{M}$ (*see* Definition A.8, page 277), there is no sequence of input/output actions $\varrho \in (\Sigma^? \cup \Sigma^!)^*$ that is a common k-follower of l_1 and l_2.

□

Example A.12 (Deterministic IOSTS with Lookahead k)

First, consider the IOSTS $\mathcal{A}$ depicted in Figure A.9 (*see* page 305). Notice that this IOSTS $\mathcal{A}$ has the unique non-deterministic choice in the location l_0 between two syntactic transitions shown in red and labeled with the input action a. However, $\mathcal{A}$ is *not* deterministic with lookahead k for any $k \in \mathbb{N}$. Indeed,

(a) As $\mathcal{A}$ contains a non-deterministic choice in the location l_0, then it is not deterministic with lookahead 0.

(b) It is not hard to check (*see* Figure A.9, page 305) that for any $k \geq 1$, and any pair of distinct syntactic transitions involved into non-deterministic choice of $\mathcal{A}$, there always exists at least one *common* k-follower for each target of these syntactic transitions. For instance, the sequence ba and bc of length 2 are 2-followers of l_1 and l_2; or the sequence $bababab$ of length 7 is 7-follower of the same locations. Therefore, $\mathcal{A}$ is not deterministic with lookahead $k \geq 1$.

Second, consider the IOSTS $\mathcal{B}$ shown on Figure A.10 (*see* page 306). This IOSTS is deterministic with lookahead 3.

Indeed, for each pair of distinct syntactic transitions t_1, t_2 and t_3 (with targets respectively l_1, l_2 and l_3) shown in red and involved into the unique non-deterministic choice of $\mathcal{B}$, the IOSTS $\mathcal{B}$ does not have any sequence that is a *common* 3-follower of the targets of these syntactic transitions. More precisely:

(a) The locations l_1 and l_2 (which are the targets of t_1 and t_2 respectively) as well as the locations l_2 and l_3 (which are the targets of t_1 and t_3 respectively) do not have a *common* follower other than the empty sequence (*see* Figure A.10).

(b) However, the locations l_1 and l_3 (which are the targets of t_1 and t_3 respectively) have:

- a common 0-follower which is the empty sequence ε,
- a common 1-follower which is the string b, and
- a common 2-follower which is the string bc.

But, they do not have any common 3-follower (*see* Figure A.10).

Therefore, according to Definition A.15 the IOSTS $\mathcal{B}$ is deterministic with lookahead 3.

It is interesting to notice that the IOSTS $\mathcal{B}$ is deterministic not only for $k = 3$, but for all $k \geq 3$. This observation, which is generalized below, leads to the definition of the IOSTS with *smallest* lookahead k.

□

Observation A.6 If the IOSTS $\mathcal{M}$ is deterministic with lookahead k then it is deterministic with lookahead $(k + p)$, where $p \in \mathbb{N}$. □

Definition A.16 (Deterministic IOSTS with Smallest Lookahead k) An IOSTS $\mathcal{M}$ is *deterministic with smallest lookahead* k if:

(1) it is deterministic with lookahead k (*see* Definition A.15), but

(2) it is *not* deterministic with lookahead $(k - 1)$.

□

For instance, the IOSTS $\mathcal{B}$ shown in Figure A.10 (*see* page 306) is deterministic with lookahead 3, and not deterministic with lookahead 2 (*see* the second part of Example A.12, page 307). Therefore, 3 is the smallest lookahead for $\mathcal{B}$.

The next question posed in this section is: whether exists $k \in \mathbb{N}$ such that a given IOSTS is deterministic with lookahead k. The answer to this question is given in the next paragraph.

A.2.4.2 Is an IOSTS Deterministic with Lookahead k for some $k \in \mathbb{N}$?

The problem of checking whether a given IOSTS $\mathcal{M}$ with set of locations L is deterministic with lookahead k for some $k \in \mathbb{N}$, is decidable under the assumption that we can decide the satisfiability of the guards (this assumption was made consistently throughout the thesis). In order to describe a solution for this problem, we need to introduce the notion of a *maximal common* k follower for two locations of IOSTS $\mathcal{M}$.

Definition A.17 (Maximal Common k-Follower of Two Locations) Let l_1 and l_2 be two distinct locations of the IOSTS $\mathcal{M}$. Then, a sequence of input/output actions ϱ is defined to be a *maximal common k-follower* of l_1 and l_2 if:

(1) ϱ is a common k-follower of l_1 and l_2, and

(2) l_1 and l_2 does not have another common k'-follower ϱ', where $k' > k$, such that ϱ is a strict prefix of ϱ'.

□

Next, to check whether a given IOSTS is deterministic with lookahead k for some $k \in \mathbb{N}$, we perform the following algorithm.

Algorithm A.2 (Detecting whether an IOSTS is Deterministic with Lookahead k for some $k \in \mathbb{N}$) Let $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ be an IOSTS without internal actions, *i.e.* $\Sigma = \Sigma^? \cup \Sigma^!$. Then, we detect whether the given IOSTS $\mathcal{M}$ belongs to the subclass of deterministic IOSTS with lookahead $k \in \mathbb{N}$ for some $k \in \mathbb{N}$, in the three following steps:

(1) If the IOSTS $\mathcal{M}$ does not have any location $l \in L$ involved into some non-deterministic choice of $\mathcal{M}$ (*see* Definition A.8, page 277), then it is deterministic with lookahead 0.

(2) Otherwise, *for each pair* of distinct locations $l_i, l_j \in L$ belonging to $(l \text{ after } a)$, where the location l and the action a are involved into a non-deterministic choice of $\mathcal{M}$, we try to detect whether there exists an integer k such that l_i and l_j have a *maximal common* k-follower for some $k \in \mathbb{N}$. In the case of existence of k we add it into an initially empty set of integers K, otherwise the algorithm stops. Formally, we perform the following steps:

(2.1) We create the two following sets:

- $P_{treated} := \emptyset$,
- $P_{new} := \{\langle l_i, l_j \rangle\}$ (*i.e.* at the beginning of each iteration the set P_{new} must contain exactly one pair for which we try to compute k).

We also initialize the integer k to zero.

(2.2) While $(P_{new} \neq \emptyset)$ do:

If $(P_{new} \cap P_{treated}) = \emptyset$, then

- $P_{treated} := P_{treated} \cup P_{new}$,
- $P'_{new} := P_{new}$,
- $P_{new} := \{\langle l'_i, l'_j \rangle \in (L \times L) \mid \exists\, \langle l_i, l_j \rangle \in P'_{new},\ a \in (\Sigma^? \cup \Sigma^!)\ .\ [l_i \overset{a}{\hookrightarrow} l'_i \in T \wedge l_j \overset{a}{\hookrightarrow} l'_j \in T]\}$, and
- $k := k + 1$.

Else, there exists an identical infinite sequence starting in the locations l_i and l_j. Thus, the locations l_i and l_j do not have a maximal common successor. Therefore, the algorithm **stops** with the result that the IOSTS does not belong to the subclass of deterministic IOSTS with lookahead k.

(2.3) Add computed k to the set K.

(3) Finally, we choose the maximal integer k among all integers belonging to the set K. This integer corresponds to a longest maximal common k-follower. Thus, we conclude that $\mathcal{M}$ is deterministic with lookahead $(k+1)$.

□

Theorem A.10 (Termination of Algorithm A.2, page 309) The algorithm detecting whether an IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ without internal actions is deterministic with lookahead k for some $k \in \mathbb{N}$, terminates. □

Proof

First, we show that the inner loop represented as the item (2.2) of Algorithm A.2, terminates.

The proof is done by the contradiction. Suppose that this inner loop does not terminate. This means that the set P_{new} is never empty. Thus, at each iteration of this loop the size of the set $P_{treated}$ is increased at least by one. ($\star$) Indeed, $P_{treated}$ is augmented only in the case when the intersection between $P_{treated}$ and P_{new} is empty. Hence, the size of $P_{treated}$ grows beyond any bound (*see* the observation ($\star$)), but in the same time $P_{treated} \subseteq (L \times L)$, where L is the *finite* set of location of $\mathcal{M}$ (*see* Definition 4.2, page 81). We obtain contradiction, therefore, the inner loop of Algorithm A.2 terminates.

Second, we notice that the outer loop of Algorithm A.2 (*see* the item (2), page 309) also terminates.

The proof of this statement follows directly from the fact that the set of locations L of the IOSTS $\mathcal{M}$ is finite; and therefore the set of the pairs of distinct locations $l_i, l_j \in L$ belonging to (l after a), where the location l and the action a are involved into a non-deterministic choice of $\mathcal{M}$, is also finite.

Finally, the two items above imply the termination of Algorithm A.2 (*see* page 309). **Q.E.D.**

A.2.4.3 Termination of the Procedure of Global Determinization

The purpose of this paragraph and the whole section is to show that for any $k \in \mathbb{N}$, the procedure of global determinization (*see* page 304) terminates for all IOSTS belonging to the class of deterministic IOSTS with lookahead k introduced in Subsection A.2.4.1 (*see* page 306). Formally:

Theorem A.11 (Termination of Procedure A.1, page 304) If there exists $k \in \mathbb{N}$ for which an IOSTS $\mathcal{M} = \langle D, \Theta, L, l^0, (\Sigma^? \cup \Sigma^!), T \rangle$ is deterministic with smallest lookahead k then Procedure A.1 (*see* page 304) applied to $\mathcal{M}$ terminates. □

Proof The proof of this theorem is done by induction on $k \in \mathbb{N}$.

Induction Basis. Consider a deterministic IOSTS $\mathcal{M}$ with lookahead 0. Due to Definition A.15 (*see* page 307) this means the set of locations L' involved into non-deterministic choices of $\mathcal{M}$, which is computed at the first step of Procedure A.1 (*see* page 304), is empty. Thus, the procedure of global determinization trivially terminates.

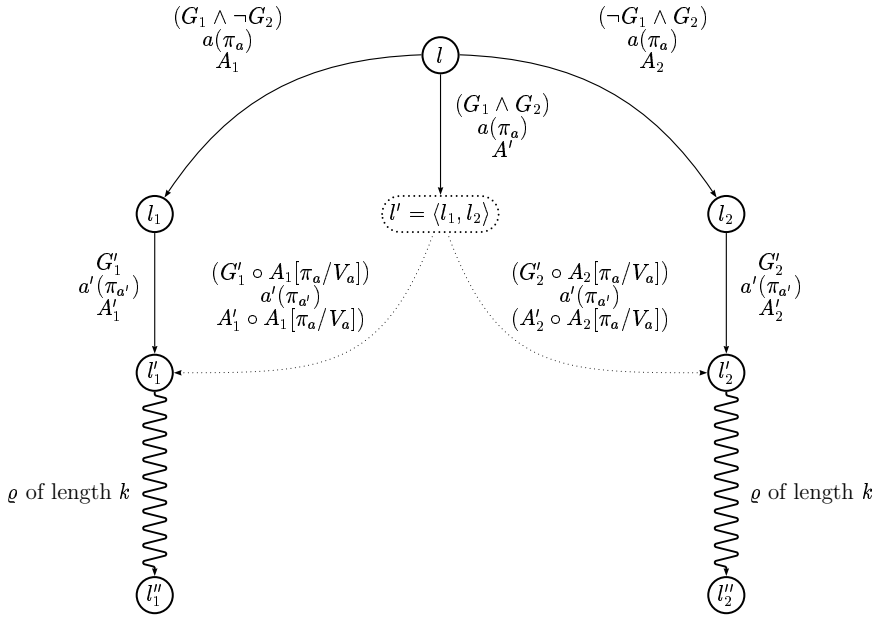


Figure A.11: A fragment of the deterministic IOSTS $\mathcal{M}'$ with smallest lookahead $(k+1)$ which was obtained from an IOSTS $\mathcal{M}$ by the first iteration of the inner loop of Procedure A.1 (*see* the item (2.a) on page 304). Here, the locations l'_1 and l'_2 belonging to $(\langle l_1, l_2 \rangle \text{ after } a')$, where $\langle l_1, l_2 \rangle$ and a' are involved into some nondeterministic choice of $\mathcal{M}'$, have the common k-follower ϱ shown as the zigzag sequences.

Induction Hypothesis. Assume that for a deterministic IOSTS $\mathcal{M}$ with smallest lookahead k, Procedure A.1 (*see* page 304) terminates.

Induction Step. Consider a deterministic IOSTS $\mathcal{M}$ with smallest lookahead $(k+1)$, and show that the procedure of global determinization applied to $\mathcal{M}$ terminates.

(1) Let $\mathcal{M}'$ be an IOSTS obtained from $\mathcal{M}$ after performing the algorithm of local determinization for all locations involved into nondeterministic choices of $\mathcal{M}$ (*i.e.* after the first iteration of the inner loop of Procedure A.1, *see* the item (2.a) on page 304). Then, we show that the IOSTS $\mathcal{M}'$ is deterministic with smallest lookahead which is less or equal to k.

The proof of this statement is done by contradiction. Assume that $\mathcal{M}'$ (*see* Figure A.11, page 312) is deterministic with smallest lookahead $(k+1)$. Then, due to Definition A.16 (*see* page 308) we can deduce

that there exist locations $l'_1, l'_2 \in L_{\mathcal{M}'}$ such that:

(a) l'_1 and l'_2 belong to $(l' \text{ after } a')$ for some location $l' \in L_{\mathcal{M}'}$ and some action $a' \in (\Sigma^? \cup \Sigma^!)$, which are involved into a non-deterministic choice of $\mathcal{M}'$, and

(b) l'_1 and l'_2 have a common k-follower, but they do not have any common $(k+1)$-follower.

Next, as l' is involved into a non-deterministic choice of $\mathcal{M}'$, and as $\mathcal{M}'$ has been obtained from the IOSTS $\mathcal{M}$ by solving non-determinism in *all* its locations involved into non-deterministic choices; then l' is a newly created location, *i.e.* $l' \in L_{\mathcal{M}'}$ but $l' \notin L$.

Finally, as l' is a new location obtained by the algorithm of local determinization (*see* page 295), then there exists at least two locations l_1 and l_2 in $\mathcal{M}$ such that:

(c) $l' = \langle \ldots, l_1, \ldots, l_2, \ldots \rangle$,

(d) l_1 and l_2 belongs to $(l \text{ after } a)$ for some location $l \in L$ and some action $a \in (\Sigma^? \cup \Sigma^!)$, which are involved into a non-deterministic choice of $\mathcal{M}$, and

(e) l_1 (*resp.* l_2) is the direct predecessor of l'_1 (*resp.* l'_2) by the action a'. For better understanding look at Figure A.11 (*see* page 312).

Thus, as we know that l'_1 and l'_2 of $\mathcal{M}'$ have a common k-follower, but do not have any common $(k+1)$-follower (*see* item (a)), then due to the item (e) we obtain that the locations l_1 and l_2 of $\mathcal{M}$ have the common $(k+1)$-follower, but do not have any common $(k+2)$-follower. This statement together with the item (d) and Definition A.16 (*see* page 308), imply that the IOSTS $\mathcal{M}$ is deterministic with smallest lookahead $(k+2)$, which contradicts to the assumption that $\mathcal{M}$ is the deterministic IOSTS with smallest lookahead $(k+1)$.

Therefore, at the end of the first iteration of the inner loop of Procedure A.1 (*see* page 304), we obtain a deterministic IOSTS $\mathcal{M}'$ with smallest lookahead which is less or equal to k.

(2) Using the induction hypothesis, we conclude that Procedure A.1 (*see* page 304) terminates.

Q.E.D.

Hence, in this section we have located the class of IOSTS with lookahead k for which the procedure of global determinization (*see* page 304) terminates, and produces a deterministic IOSTS in sense of Definition 4.20 given on page 93.

List of Figures

Bibliography

Abrial, J.-R. (1996). *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

Ackermann, W. (1954). *Solvable Cases of the Decision Problem*. North-Holland Publishing Company.

Aho, A., Dahbura, A., Lee, D., and Uyar, M. (1988). An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. In Aggrawal, S. and Sabnani, K., editors, *Protocol Specification, Verification, and Testing*.

Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Vacelet, N., and Utting, M. (2002). Bz-tt: A tool-set for test generation from Z and B using contraint logic programming. In Hierons, R. and Jéron, T., editors, *Proceedings of the Formal Approaches to Testing of Software, FATES'02 workshop of CONCOUR'02*, pages 105–120.

Angluin, D. (1982). Inference of reversible languages. *Journal of the Association for Computing Machinery*, 29(3):741–765.

Barnes, J. (1984). *Programming in Ada*. Addison-Wesley, Second Edition.

Beizer, B. (1990). *Software Testing Techniques*. New York: Van Nostrand Reinhold.

Beizer, B. (1995). *Black-Box Testing*. John Wiley and Sons.

Belinfante, A., Feenstra, J., de Vries, R., Tretmans, J., Goga, N., Feijs, L., and Mauw, S. (1999). Formal test automation: a simple experiment. In Csopaki, G., Dibuz, S., and Tarnay, K., editors, *International Workshop on the Testing of Communication Systems (IWTCS'99)*, pages 179–196. Kluwer Academic Publishers.

Bernot, G. (1991). Testing against formal specifications: A theoretical view. In Abramsky, S. and Maibaum, T., editors, *TAPSOFT'91*, volume 2-494 of *Lecture Notes in Computer Science*, pages 99–119. Springer-Verlag.

Bertot, Y. and Castéran, P. (2004). *Coq'Art: The Calculus of Inductive Constructions*. EATCS Series.

Bouquet, F., Legeard, B., and Peureux, F. (2002). Clps-b: A constraint solver for B. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systemsw (TACAS'02)*, volume 2280 of *LNSC*, pages 188–204, Grenoble, France.

Bozga, M., Fernandez, J.-C., Ghirvu, L., Graf, S., Krimm, J.-P., and Mounier, L. (1999a). IF: An intermediate representation and validation environment for timed asynchronous systems. In Wing, J., Woodcock, J., and Davies, J., editors, *Proceedings of World Conference on Formal Methods, FM'99*, volume 1708 of *LNCS*, pages 307–327, Toulouse, France. Springer-Verlag.

Bozga, M., Fernandez, J.-C., Ghirvu, L., Graf, S., Krimm, J.-P., Mounier, L., and Sifakis, J. (1999b). IF: An intermediate representation for SDL and its applications. In Dssouli, R., Bochmann, G., and Lahav, Y., editors, *SDL FORUM'99*, Montreal, Canada. Elsevier Science Publishers B.V.(North-Holland).

Bozga, M., Graf, S., and Mounier, L. (2002). IF-2.0: A validation environment for component-based real-time systems. In Brinksma, E. and Larsen, K., editors, *CAV'02*, volume 2404 of *LNCS*, Copenhagen, Denmark. Springer-Verlag.

Brinksma, E. (1988). A theory for the derivation of tests. In Aggarwal, S. and Sabnani, K., editors, *Protocol Specification, Testing, and Verification (PSTV VIII)*, pages 63–74. Elsevier Science Publishers B.V.(North-Holland).

Brinksma, E., Alderden, R., Langerak, J., de Lagemaat, R. V., and Tretmans, J. (1990). A formal approach to conformance testing. In Meer, J. D., Mackert, L., and Effelsberg, W., editors, *Second International Workshop on Protocol Test Systems*, pages 349–363. North Holland.

Brookes, S. and Roscoe, A. (1985). An improved failures model for communicating sequential processes. In *Proceedings of NSF-SERC Seminar on Concurrency*, volume 197 of *LNCS*. Springer-Verlag.

Budd, T., DeMillo, R., Lipton, R., and Sayward, F. (1980). Theoretical and empirical studies on using program mutation to test the functional correctness of program. *7th ACM Symp. on Princ. of Programming Languages (POPL'80)*, pages 220–233.

CEPSCO (2000). Common Electronic Purse Specifications - Technical Specification (`http://www.cepsco.org`).

Cheadle, A., Harvey, W., Sadler, A., Schimpf, J., Shen, K., and Wallace, M. (2003). ECLiPSe: An introduction. Technical report, IC-Parc, Imperial College London.

Chow, T. (1978). Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187.

Clarke, D., Jéron, T., Rusu, V., and Zinovieva, E. (2001a). Automated test and oracle generation for smart-card applications. In *Proceedings of the International Conference on Research in Smart Cards*, volume 2140 of *LNCS*, pages 58–70, Cannes, France. Springer-Verlag.

Clarke, D., Jéron, T., Rusu, V., and Zinovieva, E. (2001b). STG: A tool for generating symbolic test programs and oracles from operational specifications. In Gruhn, V., editor, *Proceedings of the joint 8th ESEC and 9th FSE*, ACM Press, pages 301–302, Vienna, Austria.

Clarke, D., Jéron, T., Rusu, V., and Zinovieva, E. (2001c). STG: A tool for generating symbolic test programs and oracles from operational specifications (long version). Available at `http://www.irisa.fr/prive/lenaz/research/papers/esec-full.ps`.

Clarke, D., Jéron, T., Rusu, V., and Zinovieva, E. (2002). STG: A Symbolic Test Generation tool. In *The 8th International Conference on Tools and Algorithms for the Construction and Analysis of System (TACAS'02)*, volume 2280 of *LNCS*, pages 470–475, Grenoble, France. Springer-Verlag.

Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

Clatin, M., Groz, R., Phalippou, M., and Thummel, R. (1995). Two approches linking a test generation tool with verification techniques. In *International Workshop on Protocol Test Systems (IWPTS'95)*, Ervy, France.

CNET (2000). 10 great bugs of history. Available at `http://www.bus.tu.ac.th/usr/angsana/IS301-1-42/Outline/greatbug.htm`.

Cousot, P. and Cousot, R. (1976). Static determination of dynamic properties of programs. In *The 2nd International Symposium on Programming*. Dunod, Paris.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252.

Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In *The 5th ACM Symposium on Principles of Programming Languages (POPL'78)*, Tucson, Arizona, USA. Dunod, Paris.

Dauchy, P., Gaudel, M.-C., and Marre, B. (1993). Using algebraic specifications in software testing: a case study on the software of an automatic subway. *Journal of Systems and Software*, 21(3):229–244.

de Nicola, R. (1987). Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237.

de Nicola, R. and Hennessy, M. (1984). Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133.

Dowek, G., Felty, A., Herbelin, H., Huet, G., Murthy, C., Parent, C., Paulin-Mohring, C., and Werner, B. (1993). The COQ proof assistant: User's guide. Technical report, INRIA – Rocquencourt.

du Bousquet, L., Ramangalahy, S., Simon, S., Viho, C., Belinfante, A., and de Vries, R. G. (2000). Formal test automation: The conference protocol with TGV/TorX. In Ural, H., Probert, R., and Bochmann, G., editors, *IFIP 13th International Conference on Testing of Communication Systems (TestCom2000)*. Kluwer Academic Publishers.

Edmonds, J. and Johnson, E. (1973). Matching Euler tours and the chinese postman. *Mathematical Programming*, 5:88–124.

Eertink (1993). SMILE: User manual, relise 4.0. University of Twente, The Netherlands.

Eertink, E. (1994). *Simulation Thechniques For the Validation of LOTOS Specification*. PhD thesis, University of Twente, the Netherlands.

Fernandez, J.-C., Jard, C., Jéron, T., and Viho, C. (1996). Using on-the-fly verification techniques for the generation of test suites. In Alur, T. and Henzinger, A., editors, *Comuter-Aided Verification (CAV'96)*, volume 1102 of *LNCS*, New Brunswick, New Jersey, USA. Springer-Verlag.

Fernandez, J.-C., Jard, C., Jéron, T., and Viho, C. (1997). An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2):123–146.

Fowler, M. and Scott, K. (2000). *UML Distilled: a Brief Guide to the Standard Object Modeling Language*. 2nd Edition.

Gansner, E. R. and North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233.

Gaudel, M.-C. and James, P. (1999). Testing algebraic data types and processes: a unifyin theory. *Formal Acpects of Computing*, 10(5-6):436–451.

Gill, A. (1962). *Introduction to the Theory of Finite State Machine*. New York, McGraw-Hill.

Gonenc, G. (1970). A method for the design of fault detection experiments. *IEEE Transactions on Computing*, C-19.

Gotlieb, A., Botella, B., and Rueher, M. (1998). Automatic test data generation using constraint solving techniques. *ACM SIGSOFT Software Engineering Notes*, 23(2):53–62.

Gotlieb, A., Botella, B., and Rueher, M. (2000). A CLP framework for computing structural test data. In *Proceedings of the Conference on Computational Logic (CL'00)*, pages 399–413, Imperial College, London, UK.

Grabowski, J. (1994). SDL and MSC based test case generation: An overall view of the SaMsTaG method. Technical Report IAM-94-005, University of Berne, Institute for Informatics, Berne, Switzerland.

Grabowski, J., Hogrefe, D., and Nahm, R. (1993). A method for the generation of test cases based on SDL and MSCs. Technical Report IAM-93-010, University of Berne, Institute for Informatics, Berne, Switzerland.

Groote, J.-F. and van de Pol, J. (1993). A bounded retransmission protocol for large data packets. Technical report, Logic Group, Utrecht University.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous data flow programming language LUSTRE. In *Proceeding of the IEEE*, volume 79-9, pages 1305–1320.

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

Harel, D. and Pnueli, A. (1985). On the development of reactive systems. In Apt, K., editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO ASI Series*, pages 477–498, New York. Springer-Verlag.

Havelund, K. and Shankar, N. (1996). Experiments in theorem proving and model checking for protocol verification. In *FME'02*.

Heerink, A. W. (1998). *Ins and Outs in Refusal Testing*. PhD thesis, University of Twente, the Netherlands.

Helmink, L., Sellink, M., and Vaandrager, F. (1994). Proof-checking a data link protocol. In Barendregt, H. and Nipkow, T., editors, *Proceedings of the 1st Internatioanl Workshop on Types for Proofs and Programs*, volume 806 of *LNCS*, pages 127–165, Berlin. Springer-Verlag.

Hennie, F. (1964). Fault detecting experiments for sequential circuits. In *The 5th Ann. Symp. Switching Circuit Theory and Logical Design*, pages 95–110.

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice Hall International.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Inc.

Howden, W. (1982). Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379.

Hower, R. (1996-2003). Software quality assurance and testing resource center. Available at `http://www.softwareqatest.com/`.

Hsieh, E. (1971). Checking experiments for sequential machines. *IEEE Transactions on Computers*, C-20(10):1156–1166.

IEEE/ANSI (1990). *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12. IEEE Press, New York, NY, USA.

Intelligent Systems Laboratory (2004). *SICStus Prolog User's Manual*. Intelligent Systems Laboratory, Swedish Institute of Computer Science, Kista, Sweden.

Isbell, D. and Savage, D. (1999). Mars climate orbiter failure board releases report, numerous nasa actiona underway in rerponse. Available at `http://mars.jpl.nasa.gov/msp98/news/mco991110.html`.

Ishisone, M. and Sawada, T. (2001). Brute: brute force rewriting engine. Available at `http://www.theta.ro/cafeobj`.

ISO (1991). Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework. International Standard IS-9646. Geneve.

ISO/IEC (1988). LOTOS - a formal description technique based on the temporal ordering of observational behaviour. Technical Report 8807, International Organization for Standards - Information Processing Systems - Open Systems Interconnection.

ISO/IEC (1992). International Standard 9646-1/2/3, OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework.

ISO/IEC (1996). ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. Proposed ITU-T Z.500 and Committee Draft on "Formal Methods in Conformance Testing". CD 13245-1. ISO – ITU-T. Geneve.

ISO/IEC/JTC1/SC21 (1992). Information technology – open systems interconnection – conformance testing methodology and framework – part 3: The Tree and Tabular Combined Notation.

ISO/TC97/SC21 (1997). Estelle – a formal description thechnique based on an extended state transition model.

ITU-T (1994). Recommendation Z-100. Specification and Description Language (SDL).

ITU-T (1996). ITU-T recommendation Z-120: Message Sequence Chart (MSC).

Jard, C. and Jéron, T. (2002). TGV: Theory, principles and algorithms. *The Sixth World Conference on Integrated Design and Process Technology (IDPT'02).*

Jard, C., Jéron, T., and Morel, P. (2000). Verification of test suites. In *TestCom 2000, IFIP TC 6 / WC 6.1, the IFIP 13th International Conference on Testing of Communicating Systems*, volume 2477 of *LNCS*, Ottawa, Ontario, Canada. Kluwer Academic Publishers.

Jeannet, B. (2000a). Dynamic partitioning in linear relation analysis. Technical Report RS-00-38, BRICS.

Jeannet, B. (2000b). Dynamic partitioning in linear relation analysis. Application to verification of reactive systems. *Formal Methods in System Design.*

Jeng, B. and Weyuker, E. J. (1994). A simplified domain-testing strategy. *ACM Transactions on Software Engineering and Methodology*, 3(3):254–270.

Jéron, T. and Morel, P. (1999). Test generation derived from model-checking. In *Computer Aided Verification (CAV '99)*, volume 1633 of *LNCS*, pages 108–122.

Jéron, T. (2002). TGV: Théorie, principles et algorithmes. *Techniques et Sciences Informatiques, numéro spécial Test de Logiciels.*

Jéron, T. (2004). Contribution à la génération automatique de tests pour les systèmes réactifs. *Habilitation à Diriger des Recherches, Université de Rennes I, France.*

Kahlouche, H., Viho, C., and Zendri, M. (1999). Hardware testing using a communication protocol conformance testing tool. In *The International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99).*

Keller, R. (1976). Formal verification of parallel programs. *Commun. ACM*, 19:561–572.

Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpiesman, T., and Wonnacott, D. (1995). The omega library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park.

Kerbrat, A., Jéron, T., and Groz, R. (1999). Automated test genration from SDL specifications. In *SDL'99 The Next Millenium, 9th SDL Forum*, pages 135–152, Montréal, Canada. Elsevier Science Publishers B.V.(North-Holland).

Kerbrat, A. and Ober, I. (1999). Automated test generation from SDL/UML specifications. In *12th International Software Quality Week*, San Jose CA, USA.

Koch, B., Grabowski, J., Hogrefe, D., and Schmitt, M. (1998). Autolink - a tool for automatic test generation from SDL specifications. In *IEEE International Workshop on Industrial Strength Formal Specification Techniques (WIFT'98)*, Boca Raton FL, USA.

Kohavi, Z. (1978). *Switching and Finite Automata Theory.* 2nd Edition McGraw-Hill.

Lee, D. and Yannakakis, M. (1996). Principles and methods of testing finite state machines - a survey. *IEEE*, 84(8):1090–1123.

Lestiennes, G. and Gaudel, M.-C. (2002). Testing processes from formal specifications with inputs, outputs and data types. In *The 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, Annapolis, Maryland, USA. IEEE Computer Society Press.

Leue, S. and Holzmann, G. (1999). V-promela: A visual, object-oriented language for spin. In *ObjectOriented Real-Time Distributed Computing (ISORC'99)*, pages 14–23, Houston, Texas, USA. IEEE Computer Society Press.

Leveson, N. and Turner, C. (1993). An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41.

Lewis, H. R. and Papadimitriou, C. H. (1981). *Elements of the Theory of Computation*. Prentice-Hall Software. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Lugato, D., Bigot, C., and Valot, Y. (2002). Validation and automatic test generation on UML models: Agata approch. *Electronics Notes in Theoretical Computer Science*, 66(2).

Lynch, N. and Tuttle, M. (1989). An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246.

Marre, B. and Arnould, A. (2000). Test sequence generation from LUSTRE description: GATeL. In *Proceedings of the Automated Software Engineering Conference*, pages 229–237. IEEE Computer Society.

Melton, R., Dill, D., and Ip, C. N. (1992). Murphi annotated reference manual. Technical report, Stanford University.

Meudec, C. (2001). ATGen: Automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability Journal*.

Milner, R. (1980). A calculus of communicating systems. In *Lecture Notes in Computer Science*, volume 92 of *LNCS*. Springer-Verlag.

Morel, P. (2000). *Une Algorithmique Efficace pour la Génération Automatique de Tests de Conformité*. PhD thesis, Université de Rennes I, France.

Musa, J. (1975). A theory of software reliability and its application. *IEEE Transactions on Software Engineering*, SE-1(3):312–327.

Myers, G. (1976). *Software Reliability: Principles and Practice*. McGraw-Hill, New York.

Myers, G. (1979). *The Art of Software Testing*. John Wiley & Sons.

Naito, S. and Tsunoyama, M. (1981). Fault detection for sequential machines by transition tours. In *The 11th IEEE Fault Tolerant Computing Conference*, pages 238–243.

Nielsen, B. (2000). *Specification and Test of Real-Time System.* PhD thesis, Aalborg University, Denmark.

Owre, S., Rashby, J., and Shankar, N. (1992). PVS: A prototype verification system. In *Proceedings of the 11st Internatioanl Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga - NY.

Petrenko, A., Boroday, S., and Groz, R. (1999). Confirming configuration in efsm. In *Formal Methods for Protocol Engeneering and Distributed Systems*, pages 5–24, Beijing, China. Kluwer Academic Publisher.

Phalippou, M. (1994). *Relations d'Implantations et Hypothèses de Test sur les Automates à Entreées et Sorties.* PhD thesis, Université de Bordeaux, France.

Phalippou, M. and Groz, R. (1990). Evaluation of an empirical approach for computer-aided test case generation. In Davidson, I., editor, *3rd International Workshop on Protocol Test Systems (IWPTS'90)*, McLean VA, USA.

Phillips, I. (1987). Refusal testing. *Theoretical Computer Science*, 50(2):241–284.

Pitt, H. and Freestone, D. (1990). The deriviation of conformance tests from LOTOS specifications. In *IEEE Transactions on Software Engineering*, volume 16-12, pages 1337–1343.

Pnueli, A. (1986). Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency. Overviews and Tutorials*, pages 510–584, New York. Springer-Verlag.

Ponscarme, F.-X. (2002). Contributions au développement et à l'utilisation de l'outil STG. Technical report, Université de Rennes 1.

Presburger, M. (1929). Uber die vollstandigkeit eines gewissen systems der arithmetik ganzer zahlen in welchem die addition als einzige operation hervortritt. In *Sprawozdanie z I Kongresu Matematykow Krajow Slowcanskich Warszawa*, pages 92–101, Poland.

Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375.

Rellier, J.-P. and Vardon, F. (1998). Con'Flex version 1.2: Manuel de l'utilisateur. Available at `http://www-bia.inra.fr`.

Rusu, V., du Bousquet, L., and Jéron, T. (2000). An approach to symbolic test generation. In *International Conference on Integrating Formal Methods*, volume 1945 of *LNCS*, pages 338–357, Dagstuhl, Germany. Springer-Verlag.

Rusu, V., Marchand, H., and Jéron, T. (2004). Verification and symbolic test generation for safety properties. Technical Report Internal Publication 5285, INRIA – Rennes.

Rusu, V. and Zinovieva, E. (2001). Analyzing automata with presburger arithmetic and uninterpreted function symbols. *Electronic Notes in Theoretical Computer Science*, 50(4):329–343.

Sabnani, K. and Dahbura, A. (1985). A new technique for generating protocol tests. *ACM Computer Communications*, 15(4).

Schmitt, M., Ek, A., Grabowski, J., Hogrefe, D., and Koch, B. (1998). Autolink - putting SDL-based test generation into practice. In Petrenko, A. and Yevtuschenko, N., editors, *Testing of Communicating Systems*, volume 11. Kluwer Academic Publishers.

Spivey, J. (1992). *The Z Notation: A Reference Manual*. Prentice-Hall, Second Edition.

Telelogic (1998). Autolink tutorial. Technical report, Telelogic AB Malmö, Sweden.

Tretmans, G. J. (1992). *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, the Netherlands.

Tretmans, J. (1994). A formal approach to conformance testing. In *The 6th International Workshop on Protocol Test Systems*, number C-19 in IFIP Transactions, pages 257–276.

Tretmans, J. (1995). Testing labeled transition systems with inputs and outputs. In Cavalli, A. and Budkowski, S., editors, *The 8th International Workshop on Protocol Test Systems*, pages 461–476, Ervy, France.

Tretmans, J. (1996a). Test generation with inputs, outputs, and quiescence. In Margaria, T. and Steffen, B., editors, *The 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*. Springer-Verlag.

Tretmans, J. (1996b). Test generation with inputs, outputs and repetative quiescence. *Software – Concepts and Tools*, 17(3):103–120.

Tretmans, J. (1999). Testing concurrent systems: a formal approach. In *CONCUR'99*, volume 1664 of *LNCS*. Springer-Verlag.

Tretmans, J. (2002). Testing techniques. Lecture Notes.

Uyar, C. and Dahbura, A. (1986). Optimal test sequence generation for protocols: The chinese postman algorith applied to Q.931. In *IEEE Global Telecommunication Conference*.

van Glabbeek, R. (2001). *The Linear Time - Branching Time Spectrum I*, chapter 1 of Handbook of Process Algebra, pages 3–99. Elsevier Science Publishers B.V.(North-Holland).

Vasilevskii, M. (1973). Failure diagnosis of automata. *Kibernetika*, 4:98–108.

Warmer, J. and Kleppe, A. (1998). *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley.

Zinovieva, E. (2002). Symbolic test generation for reactive systems. In *The Summer School on Modelling and Verifying Parallel Processes (MOVEP'02)*, pages 429–434, Nantes, France.

Index

Symbols

A

B

C

D

E

F

Résumé

La complexité croissante des systèmes réactifs fait que le test devient une technique de plus en plus importante dans le développement de tels systèmes. Un grand intérêt est notamment accordé au test de conformité qui consiste à vérifier si les comportements d'un système sous test sont corrects par rapport à sa spécification. Au cours des dernières années, les théories et outils de test de conformité pour la génération automatique de test se sont développés. Dans ces théories et algorithmes, les spécifications des systèmes réactifs sont souvent modélisées par différentes variantes des systèmes de transitions. Cependant, ces théories et outils ne prennent pas explicitement en compte les données du système puisque le modèle sous-jacent de système de transitions ne permet pas de le faire. Ceci oblige à énumérer les valeurs des données avant de construire le modèle de système de transitions d'un système, ce qui peut provoquer le problème de l'explosion de l'espace d'états. Cette énumération a également pour effet d'obtenir des cas de test où toutes les données sont instanciées. Or, cela contredit la pratique industrielle où les cas de test sont de vrais programmes avec des variables et des paramètres. La génération de tels cas de test exige de nouveaux modèles et techniques. Dans cette thèse, nous atteignons deux objectifs. D'une part, nous introduisons un modèle appelé système symbolique de transitions à entrée/sortie qui inclut explicitement toutes les données d'un système réactif. D'autre part, nous proposons et implémentons une nouvelle technique de génération de test qui traite symboliquement les données d'un système en combinant l'approche de génération de test proposée auparavant par notre groupe de recherche avec des techniques d'interprétation abstraite. Les cas de test générés automatiquement par notre technique satisfont des propriétés de correction: ils émettent toujours un verdict correct.

Mot-clés : test de conformité, génération symbolique de test, analyse symbolique.

Abstract

Due to the increasing complexity of reactive systems, testing has become an important technique in the process of the development of such systems. In particular, a great deal of effort has been devoted to conformance testing, which consists in checking whether the behaviors of a system under test are correct with respect to its specification. During the last decades, conformance testing theories and tools for automatic test generation have been developed. In these theories and algorithms, the specifications of reactive systems are often modeled by different variants of Labeled Transition Systems (LTS). However, these theories and tools do not explicitly take into account the system's data, since the underlying model of LTS are not able to do that. This limitation of the model compels to enumerate the values of the data before building the LTS model of a system. This may result in the state-space explosion problem. Moreover, this enumeration also has the effect of obtaining test cases where all the data are instantiated. This contradicts with industrial practice, where test cases are real programs with variables and parameters. The generation of such test cases requires new models and techniques. In this thesis we have achieved two objectives. First, we have introduced a model called Input-Output Symbolic Transition Systems (IOSTS) which explicitly includes all the data of a reactive system. Secondly, we have proposed and implemented a new test generation technique that symbolically treats all the data of a system by combining the test generation approach proposed earlier in our research group with techniques of abstract interpretation. The test cases automatically derived by our technique satisfy some correction properties. This essentially means that they always emit a correct verdict.

Keywords: conformance testing, symbolic test generation, symbolic analysis.