Contributions à la génération de tests à partir d'automates à pile temporisés

Hana M'Hemdi

UNIVERSITE DE TUNIS EL MANAR
FACULTE DES SCIENCES DE TUNIS **ECOLE DOCTORALE SPIM**

THESE

présentée en vue de l'obtention du
Doctorat en Informatique

par

Hana M'HEMDI
(Master en Informatique, Université de Bourgogne Franche-Comté)

Contributions à la Génération de tests à partir d'automates à pile temporisés

Soutenue le 23/09/2016 devant le Jury :

Hacène FOUCHAL	**Professeur (Univ. de Reims), Rapporteur**
Moncef TAJINA	**Professeur (ENSI-Manouba, Tunisie), Rapporteur**
Pierre-Cyrille HEAM	**Professeur (Univ. de Bourgogne Franche-Comté), Examinateur**
Ouajdi KORBAA	**Professeur (ISTICom-Sousse, Tunisie), Examinateur**
Nicolas STOULS	**MCF (INSA Lyon), Examinateur**
Jacques JULLIAND	**Professeur (Univ. de Bourgogne Franche-Comté), Directeur de thèse**
Riadh ROBBANA	**Professeur (INSAT-Tunis, Tunisie), Directeur de thèse**
Pierre-Alain MASSON	**MCF (Univ. de Bourgogne Franche-Comté), Co-directeur de thèse**

Thèse préparée sous convention de cotutelle
FST-Tunis, Tunisie (Laboratoire LIP2) et Université de Bourgogne Franche Comté, France (Laboratoire femto-st)

UNIVERSITE DE TUNIS EL MANAR
FACULTE DES SCIENCES DE TUNIS **UNIVERSITE DE FRANCHE COMTE**

Contributions à la Génération de tests à partir d'automates à pile temporisés

HANA M'HEMDI

REMERCIEMENTS

J'aimerais d'abord remercier mes encadrants de thèse :

Monsieur **Jacques JULLIAND**, professeur à l'université de Bourgogne Franche-Comté, pour m'avoir supporté pendant toutes ces années de thèse. Grâce à ses conseils judicieux, ses relectures, sa patience et son aide précieuse, j'ai pu mener à bien mes travaux de thèse et aboutir à mes résultats et publications. J'ai eu beaucoup de plaisir à travailler avec lui.

Monsieur **Riadh ROBBANA**, professeur à l'Institut national des sciences appliquées et de technologie de Carthage, Tunisie, pour ses conseils, sa présence continuelle et son aide précieuse, autant du côté scientifique que du côté humain.

Monsieur **Pierre-Alain MASSON**, maître de conférences à l'université de Bourgogne Franche-Comté, pour m'avoir fait bénéficier tout au long de ce travail de sa grande compétence et de son efficacité certaine que je n'oublierai jamais. J'aimerais également lui dire à quel point j'ai apprécié sa grande disponibilité.

Je tiens à remercier monsieur **Hacène FOUCHAL**, professeur à l'université de Reims et monsieur **Moncef TAJINA**, professeur à l'école nationale des sciences de l'informatique de Manouba, Tunisie, pour avoir accepté de rapporter cette thèse et pour leurs remarques pertinentes et précieuses qui ouvrent de nouveaux horizons pour ce travail. Merci à monsieur **Pierre-Cyrille HEAM**, professeur à l'université de Bourgogne Franche-Comté, monsieur **Ouajdi KORBAA**, professeur à l'institut supérieur d'informatique et des techniques de communication de Hammam Sousse, Tunisie et monsieur **Nicolas STOULS**, maître de conférences à l'institut national des sciences appliquées de Lyon pour avoir accepté d'examiner ce travail.

Je profite aussi ici d'exprimer mes remerciements à toutes les personnes que j'ai côtoyées pendant ces années de thèse, notamment les membres respectifs du laboratoire FEMTO-ST et du laboratoire LIP2.

Je souhaite remercier ma mère Moufida et mon père Belhassen, ainsi que mes frères Abdelkader et Hani et mes sœurs Hanene et Manel pour leurs encouragements, leur confiance et leur soutien tout au long de mes études. Je souhaite aussi remercier tous mes chers amis pour leur soutien et leur disponibilité. Merci particulièrement à Huda, Hayfa, Tarek, Bassem, Ibrahim, Rania, Lemia, Hamida, Anis, Mouhebeddine, Lucas et Karla.

SOMMAIRE

TABLE DES FIGURES

LISTE DES TABLES

LISTE DES ALGORITHMES

LISTE DES DÉFINITIONS

I

CONTEXTE ET PROBLÉMATIQUE

1

INTRODUCTION

Sommaire

1.1/ CONTEXTE

Le travail présenté dans cette thèse s'inscrit dans le cadre du test à partir de modèles, plus particulièrement à partir d'automates à pile temporisés qui sont des extensions des automates temporisés et des automates à pile.

Dans cette partie, nous allons tout d'abord introduire les systèmes temps-réel. Ensuite, nous présentons les automates à pile et les automates à pile temporisés. Finalement, nous présentons le concept de test des systèmes informatiques.

1.1.1/ LES SYSTÈMES TEMPS-RÉEL

Les systèmes temps-réels sont souvent des systèmes critiques, dans le sens où d'une part, ils peuvent mettre en cause des vies humaines et d'autre part, leur destruction induit des pertes d'argent considérables. Aujourd'hui, la plupart des systèmes intelligents sont construits par composants dont certains sont logiciel. De ce faît, la vérification et la validation des composants logiciels des systèmes temps-réel est un des enjeux majeurs pour le développement de systèmes automatisés. C'est notamment le cas des systèmes de transports, des systèmes médicaux, des systèmes militaires, etc. Parmi les exemples connus de défection de systèmes dues à des défauts logiciel, on peut citer le crash d'Ariane 5 lors de son premier vol, des erreurs de cibles de lanceurs de missiles Patriot et des surexpositions aux radiations par des appareils de radiothérapie. Développer des systèmes fiables mais aussi avoir les moyens de garantir leur bon fonctionnement sont des enjeux importants. Les modèles de tels systèmes doivent être vérifiés et la conformité de leurs implémentations par rapport à leurs modèles doit être validée.

Les systèmes temps-réel peuvent être modélisés par des automates temporisés [AD94] dont la sémantique est donnée par des systèmes de transition ayant une infinité d'états.

Des techniques d'abstraction en systèmes de transition de taille finie ont été mises au point pour définir des méthodes de vérification algorithmiques de ces systèmes [ACD93].

1.1.2/ LES AUTOMATES À PILE

Les automates à pile ont été largement étudiés [MS85][ABB97][BEM97a][Wal00]. Ce sont des systèmes de transitions munis d'une pile de taille non bornée. Par conséquent, ils sont plus expressifs que les systèmes à états finis. Un état dans un système à pile correspond à un point de contrôle dans l'automate plus une configuration de la pile. Un système à pile peut donc avoir un nombre infini d'états étant donné que la pile est non bornée. L'une des utilisations de ces automates est de modéliser des programmes avec appels de procédures incluant des appels récursifs.

1.1.3/ LES AUTOMATES À PILE TEMPORISÉS

Un automate à pile temporisé combine le modèle des automates à pile et des automates temporisés. Il existe deux sortes d'automates à pile temporisés : (1). automate à pile temporisé [BER95][Dan03][Dan01] [EM06] équipé avec des horloges globales et tel que les symboles de la pile ne sont pas datés, (2). automate à pile temporisé [Dan01] où les éléments de la pile sont datés par des horloges qui s'incrémentent comme les horloges globales. De plus, les opérations sur la pile ont un intervalle de temps comme argument en plus du symbole empilé ou dépilé. Les auteurs de [CL15] prouvent que le modèle de Abdulla et al. [Dan01] est expressivement équivalent à un automate à pile temporisé sans pile datée.

1.1.4/ TEST DE SYSTÈMES INFORMATIQUES

Les activités de validation et de vérification sont des activités visant à assurer qu'un système informatique développé satisfait bien les besoins exprimés et les exigences requises. La vérification consiste à s'assurer qu'un modèle du système à développer satisfait les exigences exprimées dans le cahier des charges. Pour ce qui est des exigences fonctionnelles, la démarche de vérification consiste à garantir qu'un modèle comportemental du système satisfait les exigences formalisées par des propriétés invariantes, de sûreté, de vivacité, d'équité et de non blocage. Les méthodes formelles de vérification basées sur la preuve ou sur les techniques de model checking adressent ce problème. Une fois le modèle vérifié, celui-ci peut être utilisé pour engendrer des suites de tests. Puis, ces tests sont exécutés sur les implémentations pour valider que celles-ci sont conformes au modèle vérifié. Cette démarche est appelée *MBT* (Model Based Testing) ou test à partir de modèles.

Le test est une activité très importante dans le processus de développement des systèmes informatiques. Le recours aux méthodes formelles constitue une voie appropriée pour promouvoir le test et automatiser sa génération à partir du modèle formel de la spécification. L'utilisation d'outils pour générer automatiquement des cas de test réduit considérablement le coût du processus de test et permet d'évaluer sa qualité par des mesure de couverture. Dans notre travail, nous nous intéressons au test de conformité

considéré ici comme un test fonctionnel permettant de vérifier si une implantation d'un système est conforme à sa spécification. Ce concept de conformité est exprimé à l'aide d'une relation de conformité.

1.2/ MOTIVATIONS ET CONTRIBUTIONS

L'objectif de la thèse est de contribuer à la validation de la catégorie des systèmes récursifs temps-réels modélisables par des automates à pile temporisés. Plusieurs approches de vérification ont été élaborées pour les automates temporisés [ACD93][AAS12][BPDG98] et pour les automates à pile [ABB97][BEM97b]. Il existe également de nombreux algorithmes de calcul des ensembles des états accessibles. Par contre, peu de travaux ont été effectués pour des automates à pile temporisés. Les techniques de vérification étant très coûteuses, nous orientons nos contributions sur le test de conformité des implémentations vis à vis de leur modèle. Or, à notre connaissance, il n'y a pas eu de travaux sur la génération de tests pour des systèmes décrits à base d'automates à piles temporisés avec deadlines. C'est l'objet de cette thèse. Nous allons donc nous intéresser à cette problématique de génération de tests pour ce modèle. Cette génération doit faire face à plusieurs problèmes dont :

- discrétiser une infinité d'états dans le cadre continu : ce problème est dû à la continuité du temps. La réduction de l'espace d'états d'un domaine continu est une phase très importante. Il s'agit d'un passage d'un domaine continu vers un domaine discret.
- limiter l'explosion combinatoire du nombre d'états dans le cadre discret : la génération des tests à partir d'un système fini avec un nombre important d'états peut être un problème difficile.

Les problèmes sur lesquels nous contribuons sont :

- la définition d'une relation de conformité pour les *TPAIO*.
- la définition de méthodes de génération de tests limitant l'explosion combinatoire.
- la définition de mesures de couverture de tests.

Vu qu'on s'intéresse au test de conformité à partir d'un automate à pile temporisé avec entrées/sorties et deadlines appelé *TPAIO*, notre première contribution est la définition d'une nouvelle relation de conformité appelée *tpioco* pour les *TPAIO*. C'est une adaptation de la relation *tioco* définie pour les *TAIO*. Les deadlines imposent des conditions de progression du temps dans des localités du *TPAIO*. Une deadline peut être soit $lazy$ (pas d'urgence), soit $delayable$ (avant qu'il ne soit trop tard), soit $eager$ (dès que possible).

La deuxième contribution est une solution algorithmique de complexité polynomiale de génération de tests à partir d'un automate à pile temporisé déterministe avec entrées/sorties et deadline $lazy$ seulement. Cette méthode adapte aux automates à pile temporisés une méthode de calcul des états accessibles définie pour des automates à pile [FWW97]. L'adaptation consiste à prendre en compte les contraintes de temps en plus des contraintes de pile. La méthode consiste à définir un algorithme de calcul d'un automate temporisé d'accessibilité supprimant les contraintes de pile en les respectant. Cet algorithme délivre également une table des chemins d'exécution permettant d'atteindre chaque localité à partir de la localité initiale. Nous définissons également un testeur à

partir de l'automate à pile temporisé de départ. Enfin, à partir de ce testeur et de la table des chemins d'exécution, pour les chemins exécutables en respectant les contraintes temporelles, nous produisons une suite de tests couvrant l'ensemble d'états. Pour limiter la complexité à une complexité polynomiale, nous avons défini volontairement un algorithme de calcul des automates temporisés d'accessibilité incomplet. L'activité de test étant par essence incomplète, cette limite n'est pas gravissime, mais a pour conséquence que certaines localités atteignables peuvent ne pas être prises en compte dans les tests produits. Pour limiter cet inconvénient, nous avons défini des heuristiques améliorant le taux de couverture en pratique sur les exemples. Ce travail a donné lieu à la publication [MJMR15b].

La troisième contribution est une méthode de génération de tests à partir d'un *TPAIO* déterministe avec sorties seulement et deadline $delayable$ seulement. Cette méthode est appliquée pour tester des programmes temporisés récursifs. Notre méthode adapte la méthode de génération de tests à partir d'automates temporisés [KT09] aux automates à pile temporisés. Elle définit la génération d'un testeur à pile temporisé déterministe. Puis nous définissons une méthode pour engendrer un automate d'accessibilité. Enfin, nous engendrons une suite de tests couvrant les états et les transitions atteignables. Comme les *TPAIO* sont des abstractions des programmes récursifs temporisés, les tests ne sont pas, dans le cas général, assurés d'être des exécutions concrètes. C'est le cas de notre exemple qui est une abstraction d'un programme récursif auquel on a ajouté des contraintes temporelles. Pour sélectionner les cas de test concrets et leurs données, on propose d'utiliser une exécution symbolique qui est une interprétation abstraite qui simule l'exécution du programme pour des données en entrée. Ce travail a donné lieu à la publication [MJMR15a].

La quatrième contribution est une généralisation du processus de génération de tests à partir d'un *TPAIO* déterministe avec sorties seulement et deadline $delayable$ au cas des *TPAIO* non déterministes avec entrées/sorties et trois types de deadlines. La première étape est la construction d'un testeur à partir d'un *TPAIO* donné pour résoudre les contraintes d'horloges, déterminiser le *TPAIO* et produire des verdicts d'échec $fail$ modélisant des cas de non conformité. La deuxième étape est le calcul d'un automate d'atteignabilité à partir du testeur obtenu afin de calculer un ou plusieurs chemins entre deux localités symboliques en respectant les contraintes de pile. La troisième étape consiste à générer des cas de tests en utilisant chaque transition de l'automate d'atteignabilité allant d'une localité symbolique initiale vers une localité symbolique finale. Les cas de tests sont individualisés pour vérifier si l'une des exécutions d'un modèle d'implémentation conduit à $fail$. Dans le cadre d'une exécution des tests sur une véritable implémentation, il est préférable de considérer un arbre de tests afin de poursuivre l'observation le plus loin possible pour limiter les verdicts inconclusifs. Nous évaluons la capacité de notre méthode à détecter des implémentations non-conformes par une technique de mutation qui permet de modifier automatiquement un *TPAIO* donné par l'application d'un opérateur de mutation. Nos résultats expérimentaux montrent que la grande majorité des mutants non-conformes sont détectés. Ce travail a donné lieu aux publications [MJMR15d][MJMR15c].

1.3/ ORGANISATION DU DOCUMENT

Ce mémoire est formé de trois parties.

La première partie contient les trois chapitres suivants. Le présent chapitre présente le contexte, les motivations et les objectifs de la thèse. Nous introduisons des définitions, notations, propositions et théorèmes indispensables à la lecture de cette thèse dans le chapitre 2. Nous présentons tout d'abord les notions relatives aux systèmes de transition étiquetés, puis les notions relatives aux automates temporisés et enfin les notions relatives aux automates à pile. Nous présentons d'abord dans le chapitre 3 quelques notions générales pour le domaine du test, en particulier sur les méthodes de génération de tests à partir de modèles et le test de conformité. Nous présentons aussi des méthodes de génération de tests à partir d'un système de transition étiquetés, un automate temporisé avec entrées/sorties et un automate à pile.

La deuxième partie contient les quatre chapitres suivants. Elle présente nos contributions. Nous présentons dans le chapitre 4 le modèle sur lequel portent nos travaux et notre relation de conformité pour ce modèle qui est notre première contribution. Le chapitre 5 présente notre solution algorithmique de complexité polynomiale de génération de tests à partir d'un automate à pile temporisé avec entrées/sorties où les deadlines de toutes les transitions sont $lazy$. Ce chapitre présente notre deuxième contribution. Le chapitre 6 présente notre troisième contribution qui est le test de conformité à partir d'un programme récursif temporisé modélisé par des automates à pile temporisés déterministes où toutes les transitions ont pour deadline $delayable$. Notre quatrième contribution est présentée dans le chapitre 7. C'est une méthode de génération de tests à partir d'un *TPAIO* non déterministe prenant en compte les trois types de deadline.

La troisième partie contient le chapitre 8 qui clôt ce mémoire de thèse en présentant les conclusions et en exposant les perspectives de ces travaux.

2

PRÉLIMINAIRES

Sommaire

Dans ce premier chapitre, nous introduisons quelques notions et nous donnons des définitions des modèles qui sont largement utilisés pour spécifier des systèmes comportementaux et des systèmes temps-réel dans le but de les tester et de les vérifier. Nous commençons par introduire le modèle des systèmes de transitions avec actions d'entrées/sorties (section 2.1) qui permettent une observation externe des systèmes. Nous définissons dans la section 2.2 le modèle classique des automates temporisés, tel qu'il a été introduit par Alur et Dill dans [AD94] pour les vérifier par model-checking. Nous définissons aussi dans la section 2.2 le modèle des automates temporisés avec deadlines (*TAD*) [BST98], le modèle de automates temporisés avec entrées/sorties (*TAIO*) et la composition parallèle de deux *TAIO*. Le temps étant interprété dans le domaine des nombres réels, la sémantique d'un automate temporisé est un système de transition infini. La réduction de l'espace d'états d'un domaine continu vers un domaine discret est une phase importante pour la vérification et le test formel des systèmes temps-réel. Nous présentons dans la section 2.3 des méthodes de discrétisation des automates temporisés comme la réduction par partitionnement en régions, la réduction par partitionnement en zones et la réduction par discrétisation du temps. Nous donnons dans la section 2.4 des règles pour la simplification d'un automate temporisé. Étant donné que le déterminisme est nécessaire pour la génération des cas de tests, nous présentons des méthodes de déterminisation des automates temporisés dans la section 2.5. Enfin, nous présentons dans la section section 2.6 les automates à pile.

2.1/ SYSTÈMES DE TRANSITIONS ÉTIQUETÉS (*LTS*) ET *IOLTS*

Un système de transitions étiquetées *LTS* (Labeled Transition System) [Arn94][BT00] est un modèle abstrait composé d'un ensemble de transitions entre des états. Ce modèle est utilisé pour donner une sémantique formelle à des nombreuses descriptions de

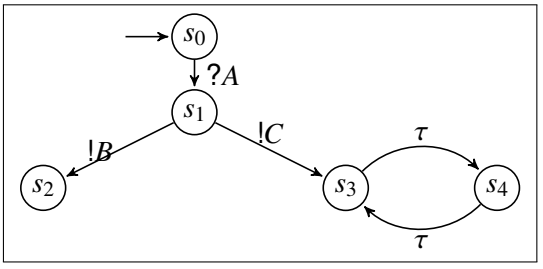


FIGURE 2.1 – Exemple d'*IOLTS*

systèmes comportementaux comme les systèmes d'événements [Abr10], les automates, les automates à pile, les automates temporisés, etc. Les systèmes de transitions à entrées/sorties *IOLTS* (Input Output Labeled Transtion System) [Tre96] de Tretmans sont une extension des *LTS* qui font la distinction entre les actions contrôlables (les actions d'entrées), les actions observables (les actions de sorties) et les actions internes notées τ ni contrôlables, ni observables. Ces distinctions de catégories d'actions permettent de définir des relations de conformité entre une implémentation et ce modèle.

Formalisons maintenant la notion d'*IOLTS* dans la Def. 1.

Définition 1 : *IOLTS* [Tre96] - Système de Transitions avec Entrées/Sorties

Un système de transitions avec entrées/sorties (*IOLTS*) est un quadruplet $T = \langle S, s_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Delta \rangle$ où :
- S est un ensemble d'états,
- s_0 est l'état initial,
- Σ_{in} est un ensemble fini de symboles d'actions d'entrée,
- Σ_{out} est un ensemble fini de symboles d'actions de sortie,
- τ est une action interne,
- $\Delta \subseteq S \times \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\} \times S$ est un ensemble fini de transitions étiquetées par un symbole d'action.

Chaque transition est un triplet $tr = (s, a, s')$ notée $s \xrightarrow{a} s'$ où :
- s est l'état source de la transition,
- a est une action qui peut être soit $?A$, soit $!B$, soit τ pour respectivement modéliser une action de réception d'un symbole $A \in \Sigma_{in}$, d'envoi d'un symbole $B \in \Sigma_{out}$ et de réalisation d'une action interne τ,
- s' est l'état cible de la transition.

Un chemin π d'un *IOLTS* est une séquence finie de transitions : $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots s_{n-1} \xrightarrow{a_{n-1}} s_n$. Dans le cas des *IOLTS*, un chemin représente une exécution du système modélisé. Sa trace est la séquence finie des symboles d'actions qui l'étiquettent $\rho = a_0 a_1 ... a_n$. Ainsi, ρ appartient à $(\Sigma_{in} \cup \Sigma_{out} \cup \{\tau\})^*$.
Notons qu'un *LTS* est classiquement un quadruplet (S, s_0, Σ, Δ) où Σ ne représente qu'une seule catégorie d'action où on ne distingue pas les entrées, des sorties et des actions internes.

Exemple 2.1.1. La Fig 2.1 présente un exemple d'*IOLTS* où $S = \{s_0, s_1, s_2, s_3, s_4\}$, τ est une action interne et inobservable, $\Sigma_{in} = \{A\}$ et $\Sigma_{out} = \{B, C\}$.

Un blocage peut être observé au niveau d'un état. Cette notion est liée à l'absence d'observation. Nous distinguons les trois types de blocages [Mor00] suivants :

— *deadlock* : aucune évolution possible, c'est à dire qu'aucune transition n'est tirable à partir d'un état. C'est le cas de l'état s_2 du *IOLTS* présenté dans la Fig. 2.1.

— *outputlock* : un état est en blocage de sortie si le système est en état d'attente d'une action provenant de l'environnement. C'est le cas de l'état s_0 de l'*IOLTS* présenté dans la Fig. 2.1.

— *livelock* : Un état est dans un livelock s'il existe un cycle d'actions internes. C'est le cas des états s_3 et s_4 de l'*IOLTS* présenté dans Fig. 2.1.

2.2/ AUTOMATES TEMPORISÉS

Dans cette partie, nous allons définir une syntaxe et une sémantique des automates temporisés, des automates temporisés avec deadlines et des automates temporisés avec entrées/sorties. Nous définissons aussi la composition parallèle de deux automates temporisés avec entrées/sorties.

2.2.1/ AUTOMATE TEMPORISÉ

Les automates temporisés (*TA* pour Timed Automata) modélisent des systèmes temps réel. Ils ont été introduits par Alur et Dill dans [AD94]. Ce sont des automates munis d'horloges modélisées par des variables réelles positives continues. Ils sont composés d'un ensemble fini de localités et d'une relation de transition entre ces localités. Le temps s'écoule dans les localités et les valeurs des horloges augmentent toutes à la même vitesse.

Soit X un ensemble d'horloges, $Grd(X)$ est un langage de gardes d'horloge défini par la conjonction d'expressions $x \sharp n$ où x est une horloge de X, n est une constante de type entier et $\sharp \in \{<, \leq, >, \geq, =\}$. $CC(X)$ est le langage des contraintes d'horloge défini comme des conjonctions d'expressions $e \sharp n$ où e est soit x, $x - x'$.

Formalisons maintenant la notion de *TA* dans la Def. 2.

Définition 2 : *TA* [AD94] - Automate Temporisé

Un automate temporisé (*TA*) est 6-uplet $T = (L, l_0, \Sigma, X, \Delta, F)$ où :
— L est un ensemble fini de localités,
— l_0 est la localité initiale,
— Σ est un ensemble fini de symboles d'actions,
— X est un ensemble fini d'horloges,
— $F \subseteq L$ est un ensemble de localités finales,
— $\Delta \subseteq L \times \Sigma \times Grd(X) \times 2^X \times L$ est un ensemble fini de transitions.

Chaque transition est un quintuplet $tr = (l, a, g, X', l')$ noté $l \xrightarrow{a,g,X'} l'$ où :
— l est la localité source de la transition,
— a est l'action exécutée quand la transition est franchie, $a \in \Sigma$

— g est la garde qui doit être satisfaite par les horloges de X,
— X' est l'ensemble des horloges remises à zéro lors du franchissement de la transition,
— l' est la localité cible de la transition.

La sémantique d'un *TA* est un *LTS* $< S^T, s_0^T, \Sigma \cup \mathbb{R}^+, \Delta^T >$ où s_0^T est l'état initial, S^T est un ensemble d'états et Δ^T est un ensemble de transitions. Un état de S^T est un couple $(l, v) \in L \times (X \to \mathbb{R}^+)$ où l représente la localité de contrôle courante et v est une valuation des horloges de l'ensemble X. La sémantique est un système de transition infini car chaque horloge peut prendre une infinité de valeurs lors de l'écoulement du temps. Dans cette sémantique, il existe deux types de transitions qui sont des triplets dans l'ensemble $S^T \times \Sigma \cup \mathbb{R}^+ \times S^T$: (1) des transitions d'écoulement du temps : pour $t \in \mathbb{R}^+$, on note $(l, v) \to^t (l, v')$ si $v' = v + t$ et (2) des transitions discrètes : pour $A \in \Sigma$, on note $(l, v) \to^A (l', v')$ s'il existe une transition $(l, A, g, X', l') \in \Delta$ telle que v satisfasse g et $v' = v[X' := 0]$ où $v[X' := 0]$ représente la remise à zéro des horloges de X' dans la valuation d'horloges v.

Dans la section 2.3, nous présentons des méthodes pour discrétiser ces *LTS* et donner à un *TA* une sémantique par un *LTS* de taille finie permettant de définir des méthodes algorithmiques de vérification.

2.2.2/ AUTOMATE TEMPORISÉ AVEC DEADLINES

Un automate temporisé avec deadlines appelé *TAD* (Timed Automata with Deadlines) est une extension du modèle des *TA* en ajoutant aux transitions du *TA* des deadlines [BST98][SY96]. Une transition avec deadline est un 6-uplet (l, A, g, d, X', l') noté $l \xrightarrow{a,g,d,X'} l'$ où d est une deadline, soit **lazy**, soit **delayable**, soit **eager**. Cette deadline détermine la deadline du déclenchement de l'action A. La deadline *lazy* n'exprime aucune deadline particulière et permet à l'action de A de se déclencher quand sa garde g est satisfaite, mais, aussi au temps de s'écouler indéfiniment sans déclencher A. La deadline *delayable* oblige, quand sa garde est satisfaite, l'action A à se déclencher avant que sa garde ne soit plus satisfaite : elle interdit au temps de s'écouler au delà, sauf si l'instant initial d'arrivée dans la localité était déjà au delà de l'instant maximum où la garde peut être vraie. La deadline *eager* oblige l'action A à se déclencher aussitôt que sa garde est satisfaite. Au delà le temps ne peut pas s'écouler, sauf si l'instant initial d'arrivée dans la localité est au delà de l'instant maximum décrivant comment le temps est autorisé à s'écouler par rapport au déclenchement d'une transition. Ces conditions d'écoulement de temps sont appelées *TPC* (Time Progress Condition). La Fig. 2.2 illustre les différents cas de deadline pour la garde g d'une transition. Le temps peut s'écouler indéfiniment avec la deadline *lazy*. Il ne peut pas s'écouler avec la deadline *eager* quand g est évalué à vrai. Il peut s'écouler avec la deadline *delayable* mais pas à l'instant où le temps a atteint l'instant maximal où la garde g est vraie.

Dans la sémantique du *TAD*, il existe deux types de transitions satisfaisant des conditions supplémentaires liées aux deadlines :

- **Transition d'écoulement du temps** : pour $t \in \mathbb{R}^+$, il y a une transition $(l, v) \to^t (l, v')$ où $v' = v + t$, si et seulement s'il n'existe pas de transition $(l, a, g, d, X', l') \in \Delta$ telle que : (1). soit $d = delayable$ et il existe t_1 et t_2 tels que $0 \leqslant t_1 < t_2 \leqslant t$, $v + t_1 \vDash g$

et $v + t_2 \not\models g$, (2). soit $d = eager$ et il existe t_1 tel que $0 \leqslant t_1 < t$ et $v + t_1 \models g$.

- **Transition discrète** : pour $A \in \Sigma$, il y'a une transition $(l, v) \rightarrow^A (l', v')$ s'il existe une transition $(l, A, g, d, X', l') \in \Delta$ et que v satisfait la garde g et que $v' = v[X' := 0]$, c'est à dire v' est la valuation de v dans laquelle les horloges de X' ont été remises à zéro.

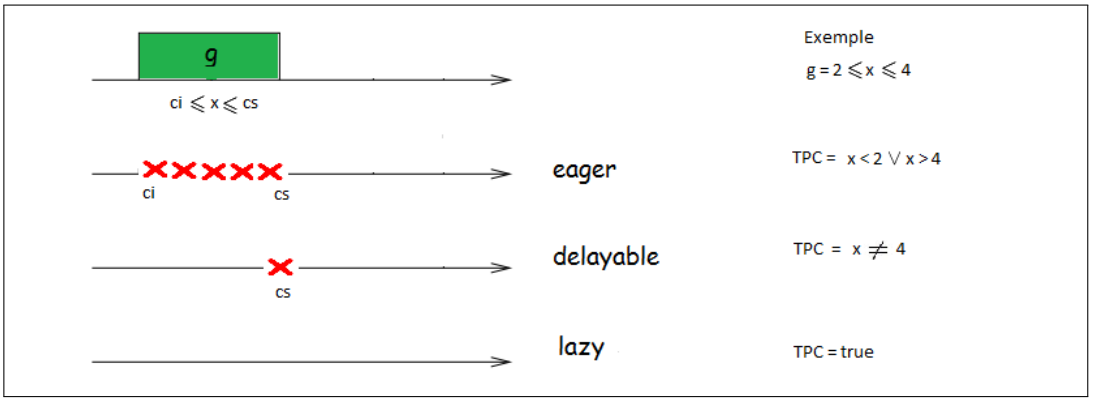


FIGURE 2.2 – Réprésentation des *TPC* pour une transition de garde g, selon la deadline- Les 'X' représentent les instants où le temps n'est pas autorisé à s'écouler dans la localité source de la transition de garde g et de deadline indiqué

Les auteurs de [BST98] ne permettent pas que les gardes des transitions avec deadline $delayable$ soient sous la forme $x < c$ (inférieur strict) car il n'existe pas de "dernier instant" avant c où la garde est encore vraie. Ils ne permettent pas non plus que les gardes des transitions avec deadline $eager$ soient sous la forme $x > c$ (supérieur strict) car il n'y a pas de "premier instant" après c où la garde n'est plus vraie.

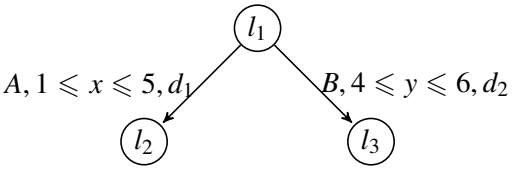


FIGURE 2.3 – Partie d'un exemple de *TAD*

On présente dans la table 2.1 la *TPC* de la localité l_1 du *TAD* présenté dans la Fig. 2.3 pour les différents types de deadline pour les deux transitions $(l_1, A, 1 \leqslant x \leqslant 5, d_1, \emptyset, l_2)$ et $(l_1, B, 4 \leqslant y \leqslant 6, d_2, \emptyset, l_3)$. La valuation d'horloge dans la localité l_1 est une valuation quelconque. Le temps peut s'écouler indéfiniment dans la localité l_1 si $d_1 = lazy$ et $d_2 = lazy$. Pour $d_1 = delayable$ et $d_2 = lazy$, le temps ne peut s'écouler que si $x \neq 5$. Par contre, le temps ne peut s'écouler dans l_1 que si $x < 1 \vee x > 5$ pour $d_1 = eager$ et $d_2 = lazy$ car si la valeur courante de l'horloge x à l_1 est inférieure à 1, alors, le temps peut s'écouler avant que la valeur d'horloge x atteigne la valeur 1. Mais, si la valeur courante de l'horloge x à l_1 est strictement supérieure à 5, alors, le temps peut s'écouler indéfiniment et il n'est pas permis de franchir la transition étiquetée avec A. Si $d_1 = delayable$ et $d_2 = eager$, alors, le temps peut s'écouler avant que l'horloge y n'ait atteint la valeur 4 et que la valeur de

l'horloge x soit différente de 5 ou que la valeur d'horloge y dépasse 6 et que la valeur de l'horloge x soit différente de 5.

	$d_1 = lazy$	$d_1 = delayable$	$d_1 = eager$
$d_2 = lazy$	$true$	$x \neq 5$	$x < 1 \vee x > 5$
$d_2 = delayable$	$y \neq 6$	$y \neq 6 \wedge x \neq 5$	$y \neq 6 \wedge (x < 1 \vee x > 5)$
$d_2 = eager$	$y < 4 \vee y > 6$	$(y < 4 \vee y > 6) \wedge x \neq 5$	$(y < 4 \vee y > 6) \wedge (x < 1 \vee x > 5)$

TABLE 2.1 – *TPC* de la localité l_1 du *TAD* de la fig. 2.3 en fonction des deadlines d_1 et d_2

2.2.3/ AUTOMATE TEMPORISÉ AVEC ENTRÉES/SORTIES (*TAIO*)

Un automate temporisé avec entrées/sorties (*TAIO* pour Timed Automata with Inputs and Outputs) [KT09] est un automate temporisé avec deadlines en considérant que $\Sigma = \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}$ où Σ_{in} est un ensemble fini de symboles d'entrées, Σ_{out} est un ensemble fini de symboles de sorties et τ est une action interne et inobservable. L'environnement, ainsi qu'un testeur, envoie des commandes de Σ_{in} et observe des sorties de Σ_{out}. L'implémentation sous test (IUT pour Implementation Under Test), envoie des actions observable de Σ_{out} et accepte des commandes de Σ_{in}. Pour un *TAIO*, il est nécessaire d'introduire :

- Son chemin π est une séquence finie de transitions : $l_0 \xrightarrow{a_0,g_0,d_0,X_0} l_1 \xrightarrow{a_1,g_1,d_1,X_1} l_2 \cdots l_{n-1} \xrightarrow{a_{n-1},g_{n-1},d_{n-1},X_{n-1}} l_n$.
- Son exécution est un chemin de l'*IOLTS* qui définit sa sémantique. Il alterne des transitions d'écoulement de temps et des actions discrètes. $\sigma = (l_0, v_0) \rightarrow^{t_0} (l_0, v_0 + t_0) \rightarrow^{a_0} (l_1, v_1) \rightarrow^{t_1} (l_1, v_1 + t_1) \rightarrow^{a_1} (l_2, v_2) \rightarrow^{t_2} ... \rightarrow^{a_{n-1}} (l_n, v_n)$ où $t_i \in \mathbb{R}^+$ et $a_i \in \Sigma$ pour chaque $0 \leqslant i \leqslant n-1$ est une exécution de π si v_i satisfait g_i pour $0 \leqslant i < n$ et si les t_i satisfont les *TPC*.
- Sa trace est une séquence finie $\rho = t_0 a_0 t_1 a_1 ... t_n a_n$ de $(\Sigma \cup \mathbb{R}^+)^*$.

$s_0^T \rightarrow^\rho s$ signifie que l'état s est atteignable à partir de l'état initial s_0^T, i.e. il existe une exécution σ à partir de s_0^T vers s dont la trace est ρ. $s_0^T \rightarrow^\rho$ signifie qu'il existe s' tel que $s_0^T \rightarrow^\rho s'$. On note $RT(\Sigma)$ l'ensemble des traces finies $(\Sigma \cup \mathbb{R})^*$. Pour une trace ρ, on définit :

- $P_{\Sigma_1}(\rho)$ est la projection de la trace ρ sur $\Sigma_1 \subseteq \Sigma$ en conservant les délais (et en cumulant ceux qui se succèdent directement). Par exemple, si $\rho = 5a4b2$, alors, $P_{\{a\}}(\rho) = 5a42 = 5a6$.
- $Time(\rho)$ est la somme de tous les délais de ρ. Par example, $Time(5a42) = 11$.

Soit un *TAIO* T, on dit que :

- T est déterministe si pour toute localité l dans L, pour chaque action a dans Σ_τ et pour tous les couples de transitions distinctes $t_1 = (l, a, g_1, X_1, l_1)$ et $t_2 = (l, a, g_2, X_2, l_2)$ dans Δ alors $g_1 \wedge g_2$ n'est pas satisfiable.
- T est observable s'il n'existe aucune transition étiquetée par τ.
- $Reach(T) = \{s^T \in S^T \mid \exists \rho.(\rho \in RT(\Sigma) \wedge s_0^T \rightarrow^\rho s^T\}$ est l'ensemble des états atteignables du *TAIO* T.
- T est non bloquant si $\forall (s,t).(s \in Reach(T) \wedge t \in \mathbb{R}^+ \Rightarrow \exists \rho.(\rho \in RT(\Sigma_{out} \cup \{\tau\}) \wedge Time(\rho) = t \wedge s \rightarrow^\rho))$.

- T est complet en entrée s'il accepte n'importe quelle entrée dans chaque localité.

Deux *TAIO* peuvent synchroniser leurs actions de sorties et d'entrées. Soit $T_1 = (L^{T_1}, l_0^{T_1}, \Sigma_{in}^{T_1} \cup \Sigma_{out}^{T_1} \cup \{\tau^{T_1}\}, X^{T_1}, \Delta^{T_1}, F^{T_1})$ et $T_2 = (L^{T_2}, l_0^{T_2}, \Sigma_{in}^{T_2} \cup \Sigma_{out}^{T_2} \cup \{\tau^{T_2}\}, X^{T_2}, \Delta^{T_2}, F^{T_2})$ deux *TAIO*. L'expression $A \in \Sigma_{T_1 \to T_2}$ signifie que $A \in \Sigma_{out}^{T_1}$ et $A \in \Sigma_{in}^{T_2}$. $A \in \Sigma_{T_2 \to T_1}$ signifie que $A \in \Sigma_{in}^{T_1}$ and $A \in \Sigma_{out}^{T_2}$. Deux *TAIO* T_1 et T_2 où $\Sigma_{T_2 \to T_1} \subseteq \Sigma_{in}^{T_1} \cap \Sigma_{out}^{T_2}$ et $\Sigma_{T_1 \to T_2} \subseteq \Sigma_{out}^{T_1} \cap \Sigma_{in}^{T_2}$ sont compatibles si $X^{T_1} \cap X^{T_2} = \emptyset$, les ensembles $\Sigma_{T_1 \to T_2}$, $\Sigma_{out}^{T_1} \setminus \Sigma_{T_1 \to T_2}$, $\Sigma_{T_2 \to T_1}$, $\Sigma_{in}^{T_1} \setminus \Sigma_{T_2 \to T_1}$, $\Sigma_{in}^{T_2} \setminus \Sigma_{T_1 \to T_2}$ et $\Sigma_{out}^{T_2} \setminus \Sigma_{T_2 \to T_1}$ sont disjoints deux à deux, T_1 et T_2 sont complets en entrées et toutes les transitions avec entrées sont *lazy*.

Exemple 2.2.1. La Fig. 2.4 montre un exemple de *TAIO*. Ce *TAIO* est présenté dans [KT05]. Il modélise un système de détection d'action de simple ou double clic d'une souris. Dans la localité l_1, la transition $l_1 \xrightarrow{?clic, x<1, lazy} l_2$ est déclenchable tant que $x < 1$, mais si le temps s'écoule jusqu'à 1 sans arrivée de clic, alors la transition $l_1 \xrightarrow{!simple, x=1, eager, \{x\}} l_0$ est franchie en deadline et le système reconnait donc un simple clic. La transition $l_1 \xrightarrow{?clic, x<1, lazy} l_2$ n'est alors plus franchissable. Si le deuxième clic arrive tant que $x < 1$ est vraie, il est pris en compte par la séquence l_1, l_2, l_0. Les autres clics qui arriveraient après $x = 1$ sont ignorés par la transition réflexive $l_2 \to l_2$. Un n-clic est un double clic avec $n \geqslant 2$.

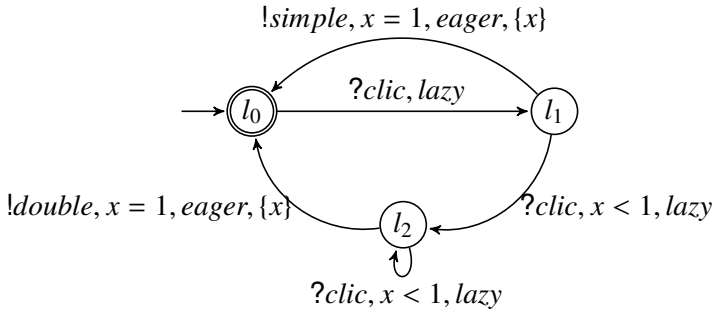


FIGURE 2.4 – *TAIO* de reconnaissance de clics simple ou double

2.2.4/ COMPOSITION PARALLÈLE DE DEUX *TAIO*

La composition parallèle de systèmes temporisés est considérée comme une extension de la composition parallèle de systèmes non temporisés. Soit $T_1 = \langle L^1, l_0^1, \Sigma_{in}^1 \cup \Sigma_{out}^1 \cup \{\tau^1\}, X^1, \Delta^1, F^1 \rangle$ et $T_2 = \langle L^2, l_0^2, \Sigma_{in}^2 \cup \Sigma_{out}^2 \cup \{\tau^2\}, X^2, \Delta^2, F^2 \rangle$ deux *TAIO* compatibles. La composition parallèle de T_1 et T_2 est un *TAIO* dont les transitions sont définies selon les règles suivantes :

- T_1 évolue indépendamment sur une action qui est une de ses actions d'entrée ou de sortie ou interne et qui n'est ni une action de sortie ni une action d'entrée dans T_2.
- T_2 évolue indépendamment sur une action qui est une de ses actions d'entrée ou de sortie ou interne et qui n'est ni une action de sortie ni une action d'entrée dans T_1.
- les transitions de T_1 et T_2 se synchronisent sur les actions communes qui sont des actions de sortie de T_1 et d'entrée de T_2.

- les transitions de T_1 et T_2 se synchronisent sur les actions communes qui sont des actions de sortie de T_2 et d'entrée de T_1.

Formalisons maintenant la notion de composition parallèle de deux *TAIO* dans la Def. 3.

Définition 3 : Composition parallèle de deux *TAIO* [KT09]

Soit $T_1 = \langle L^{T_1}, l_0^{T_1}, \Sigma_{in}^{T_1} \cup \Sigma_{out}^{T_1} \cup \{\tau^{T_1}\}, X^{T_1}, \Delta^{T_1}, F^{T_1} \rangle$ et $T_2 = \langle L^{T_2}, l_0^{T_2}, \Sigma_{in}^{T_2} \cup \Sigma_{out}^{T_2} \cup \{\tau^{T_2}\}, X^{T_2}, \Delta^{T_2}, F^{T_2} \rangle$ deux *TAIO* compatibles, la composition parallèle $PC = T_1 \parallel T_2$ est le *TAIO* $\langle L^{T_1} \times L^{T_2}, (l_0^{T_1}, l_0^{T_2}), \Sigma_{in}^{T_1} \cup \Sigma_{out}^{T_1}, X^{T_1} \cup X^{T_2}, \Delta, F^{T_1} \times F^{T_2} \rangle$ où Δ est une relation de transition telle que :

(i) $((l^{T_1}, l^{T_2}), a, g^{T_1}, d^{T_1}, X^{T_1\prime}, (l^{T_1\prime}, l^{T_2})) \in \Delta$ si $(l^{T_1}, a, g^{T_1}, d^{T_1}, X^{T_1}, l^{T_1\prime}) \in \Delta^{T_1}$ et $a \in (\Sigma_{out}^{T_1} \setminus \Sigma_{in}^{T_2}) \cup \Sigma_{in}^{T_1} \setminus \Sigma_{out}^{T_2}) \cup \{\tau^{T_1}\}$.

(ii) $((l^{T_1}, l^{T_2}), a, g^{T_2}, d^{T_2}, X^{T_2\prime}, (l^{T_1}, l^{T_2\prime})) \in \Delta$ si $(l^{T_2}, a, g^{T_2}, d^{T_2}, X^{T_1}, l^{T_2\prime}) \in \Delta^{T_2}$ et $a \in (\Sigma_{out}^{T_2} \setminus \Sigma_{in}^{T_1}) \cup (\Sigma_{in}^{T_2} \setminus \Sigma_{out}^{T_1}) \cup \{\tau^{T_2}\}$.

(iii) $((l^{T_1}, l^{T_2}), a, g^{T_1} \wedge g^{T_2}, d^{T_1}, X^{T_1} \cup X^{T_2}, (l^{T_1\prime}, l^{T_2\prime})) \in \Delta$ si $(l^{T_1}, a, g^{T_1}, d^{T_1}, X^{T_1}, l^{T_1\prime}) \in \Delta^{T_1}$ et $(l^{T_2}, a, g^{T_2}, X^{T_2}, lazy, l^{T_2\prime}) \in \Delta^{T_2}$ et $a \in \Sigma_{T_1 \longrightarrow T_2}$.

(iv) $((l^{T_1}, l^{T_2}), a, g^{T_1} \wedge g^{T_2}, d^{T_2}, X^{T_1} \cup X^{T_2}, (l^{T_1\prime}, l^{T_2\prime})) \in \Delta$ si $(l^{T_1}, a, g^{T_1}, lazy, X^{T_1}, l^{T_1\prime}) \in \Delta^{T_1}$ et $(l^{T_2}, a, g^{T_2}, X^{T_2}, d^{T_2}, l^{T_2\prime}) \in \Delta^{T_2}$ et $a \in \Sigma_{T_2 \longrightarrow T_1}$.

2.3/ DISCRÉTISATION DES AUTOMATES TEMPORISÉS

La sémantique d'un automate temporisé est un système de transition infini. Des techniques d'abstraction vers des systèmes de transition de taille finie ont été proposées dans la littérature pour définir des méthodes de vérification algorithmiques. Néanmoins la complexité de ces méthodes est telle qu'elles sont difficilement applicables sur des systèmes réels. La problématique est donc de définir des méthodes de vérification et/ou de validation dont la complexité les rend utilisables en pratique sur des modèles de grande taille. Nous présentons dans la partie 2.3.1 une technique de réduction de l'espace d'états par partitionnement en régions. Une technique de réduction par partitionnement en zones de l'espace d'états est présenté dans la partie 2.3.2. Nous évoquons une méthode de réduction par discrétisation du temps dans la partie 2.3.3.

2.3.1/ RÉDUCTION PAR PARTITIONNEMENT EN RÉGIONS

Dans cette partie, on s'intéresse à une technique de réduction de l'espace d'états. Cette technique est une technique de partitionnement en régions. Elle permet de passer d'un domaine dense à un domaine discret. L'ensemble infini des états est partitionné selon une relation d'équivalence en des groupes qui sont des classes d'équivalence d'états appelées des régions. En appliquant cette technique, on obtient un automate appelé **Automates de régions**.

Avant de définir la relation d'équivalence (voir Def. 4), les notations utilisées sont :

- fract(t) : la partie décimale de t, $\lfloor t \rfloor$: la partie entière de t.
- pour tout x de X, on note par c_x la constante la plus grande apparaissant dans les contraintes sur l'horloge x dans le *TA* considéré.

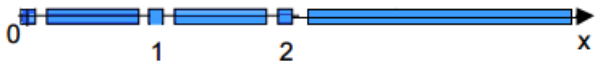
Définition 4 : Équivalence de deux valuations d'horloges [AD94]

Deux valuations v_1 et v_2 sont équivalentes, ce que l'on note $v_1 \cong v_2$ si :

1. Pour toute horloge $x \in X$, soit $\lfloor v_1(x) \rfloor = \lfloor v_2(x) \rfloor$, soit $v_1(x) > c_x$ et $v_2(x) > c_x$.
2. Pour toutes horloges $x_1, x_2 \in X$ telle que $v_1(x_1) \leq c_{x_1}$ et $v_2(x_2) \leq c_{x_2}$, fract$(v_1(x_1)) \leq$fract$(v_1(x_2))$ si et seulement si fract$(v_2(x_1)) \leq$fract$(v_2(x_2))$.
3. Pour toute horloge $x \in X$ telle que $v_1(x) \leqslant c_x$, fract$(v_1(x)) = 0$ si et seulement si fract$(v_2(x)) = 0$.

Le nombre de régions dépend à la fois de la valeur maximale qui peut être atteinte par une horloge de X et du nombre d'horloges.

Exemple 2.3.1. L'ensemble des régions dans le cas $c_x = 2$ avec une seule horloge x est défini ainsi : $R = \{0,]0, 1[, 1,]1, 2[, 2,]2, \infty[\}$ et représenté ainsi :



La Fig. 2.5 présente 28 régions dans le cas $c_x = 2$ et $c_y = 1$ avec deux horloges x et y.

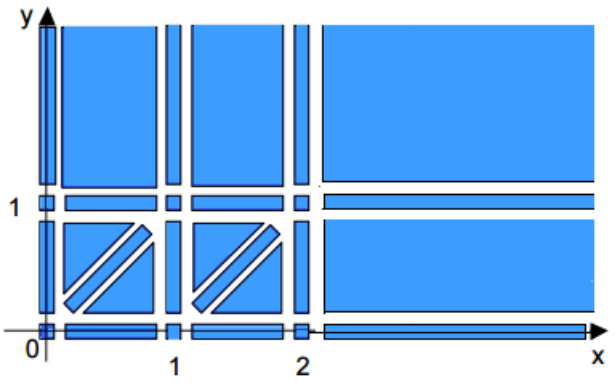


FIGURE 2.5 – Ensemble des régions dans le cas $c_x = 2$ et $c_y = 1$

Définition 5 : Successeur d'une région [AD94]

La région r' est la région suivante de r, notée $succ(r) = r'$ si et seulement si :

$$\forall v.(v \in r \Rightarrow \exists t.(t \in R^+ \wedge v + t \in r' \wedge \forall t'.(t' < t \Rightarrow v + t' \in (r \cup r')))$$

Un automate des régions est une abstraction finie d'un automate temporisé, tout en préservant des propriétés intéressantes. Il est formé d'un ensemble fini d'états qui sont des couples (l, α) où $l \in L$ et α est une région d'horloge. Il permet d'étudier les propriétés d'un automate temporisé car il y a une bisimulation temporelle entre un automate temporisé et son automate des régions. Il est formellement défini dans la Def. 6.

Définition 6 : Automate des régions [AD94]

Soit $T = (L, l_0, \Sigma, X, T, F)$ un *TA*, son automate des régions $R(T) = \langle S^T, s_0^T, \Sigma, \Delta^T \rangle$ est un *LTS* défini ainsi :

- Les états de S^T sont des couples (l, α) où $l \in L$ et α est une région d'horloge.
- L'état initial est $s_0^T = (l_0, [v_0])$ où $v_0(x) = 0$ pour toute horloge de X.
- Il y a un arc $((l, \alpha), a, (l', \alpha'))$ dans Δ^T si et seulement s'il existe une transition (l, a, g, X', l') dans T et une région d'horloge α" telles que :
  — α" est le successeur de α (voir Def. 5),
  — α" satisfait g
  — $\alpha' = \alpha$"$[X' := 0]$

Exemple 2.3.2. La Fig. 2.6 présente un exemple d'un automate temporisé muni de deux horloges x et y et qui est présenté dans [AD94].

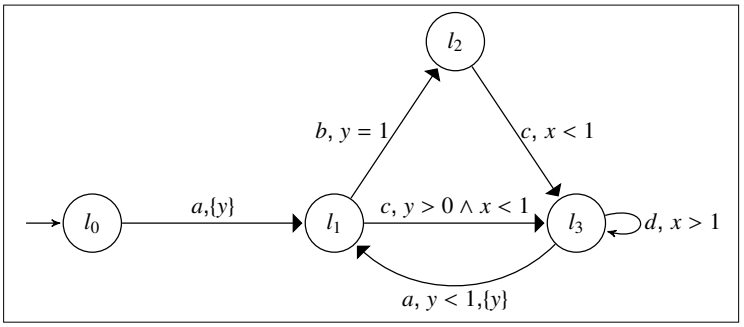


FIGURE 2.6 – *TA* à deux horloges

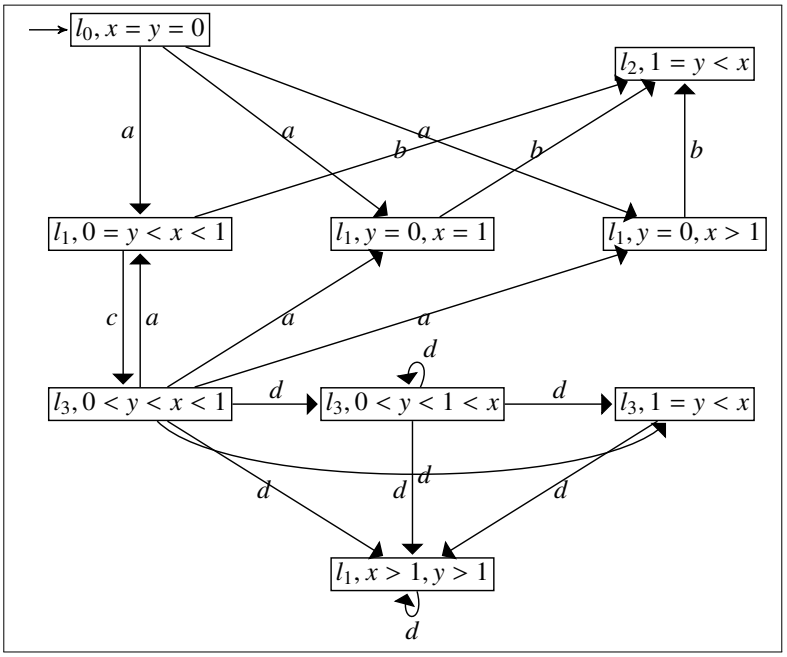


FIGURE 2.7 – Automate des régions du *TA* de la Fig. 2.6

La Fig. 2.7 présente une partie de l'automate des régions de l'automate temporisé présenté dans la Fig. 2.6. Seules les régions accessibles à partir de la région initiale $(l_0, x = y = 0)$ sont présentées. Nous avons $C_x = 1$ et $C_y = 1$. L'état initial est $(l_0, x = y = 0)$. On a trois régions à partir de l'état initial $x = y \wedge 0 < x < 1$, 1 , $x = y \wedge x > 1$. Donc, on peut atteindre trois nouveaux états à partir de l'état initial par l'action a mais dans trois régions distinctes. Il y a par exemple l'arc $((l_0, x = y = 0), a, (l_1, y = 0, 0 < x < 1))$ car la région $y = x \wedge 0 < x < 1$ est le successeur de $x = y = 0$. La région $y = 1 \wedge x = 1$ satisfait la garde *true* et la région $y = 0, x = 1$ est égale à $(y = 1, x = 1)[y := 0]$. On trouve aussi les arcs $((l_0, x = y = 0), a, (l_1, 0 = y < x < 1))$ et $((l_0, x = y = 0), a, (l_1, y = 0, x > 1))$. La transition $(l_2, c, x < 1, \emptyset, l_3)$ n'est pas déclenchable car pour quand y est égal à 1 dans la localité l_2, alors, la valeur de l'horloge x est supérieure à 1.

A la fin de cette phase, on obtient un graphe des régions de taille exponentielle. Le graphe des régions est donc exponentiel selon le nombre d'horloges et selon le codage binaire de la constante maximale qui peut être atteinte par les horloges. Cette méthode de partitionnement de l'espace infini d'états en un ensemble fini de régions a été largement utilisée pour la vérification d'accessibilité sur les systèmes temps-réel. Elle permet de montrer la décidabilité du problème du vide pour la classe des automates temporisés. Le problème du vide est de tester si le langage reconnu par l'automate est vide ou non.

Théorème 2.3.1 *(*Problème de vide [AD94]). Le problème du vide est décidable pour les automates temporisés. C'est un problème PSPACE-complet.

L'automate des régions est un automate d'états fini reconnaissant exactement le même langage non temporisé que son automate temporisé. Le problème d'accessibilité consiste à décider si un état de contrôle dans un automate temporisé donné est atteignable. La décidabilité de l'accessibilité dans un automate temporisé est une propriété primordiale dans l'utilisation de ce modèle pour la vérification et pour le test. Les automates temporisés possèdent une infinité d'états, ce qui rend délicate la génération de tests à partir d'un tel modèle.

Théorème 2.3.2. L'accessibilité d'un état de contrôle dans un automate temporisé est un problème PSPACE-complet [AD94] [AD90a].

La preuve de décidabilité pour ce problème est fondée sur le graphe des régions.

Théorème 2.3.3. Une localité l est accessible dans un automate temporisé T si et seulement si un état de la forme (l, r) est accessible dans son graphe des régions.

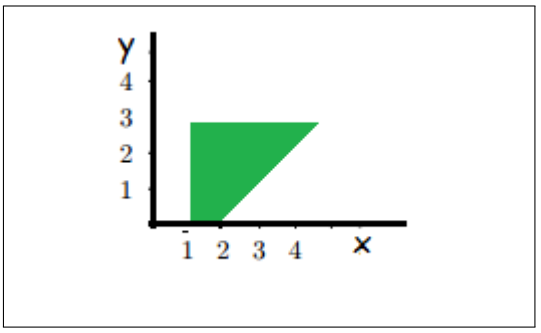
2.3.2/ RÉDUCTION PAR PARTITIONNEMENT EN ZONES [DT98]

Les régions, introduites dans la sous-section 2.3.1, constituent une représentation symbolique. Cependant, le nombre de régions est exponentiel dans la taille de l'automate. Pour vérifier ce modèle par model-checking, il est nécessaire de maitriser sa taille même s'il est un modèle fini. La plus communément utilisée est la représentation symbolique appelée zone qui est utilisée dans la majorité des algorithmes développés pour les systèmes temporisés et qui est plus compacte que les régions.

Une zone d'horloges [BCBB09] est un espace convexe défini par une conjonction d'inégalités qui comparent soit une valeur d'horloge à un nombre entier ou la différence

entre deux valeurs d'horloge à un nombre entier. Elle est sous la forme : $\bigwedge_{0 \leqslant i \neq j \leqslant nb}$ $x_i - x_j \prec n_{ij}$ où $\prec \in \{<, \leqslant\}$, nb est le nombre d'horloges, $n_{ij} \in \mathbb{N}$ et x_i sont des horloges. On introduit une horloge spéciale x_0 qui est toujours égale à zéro.

Exemple 2.3.3. La figure ci-dessous donne un exemple d'une représentation graphique de la zone $x - x_0 > 1 \wedge y \leqslant 3 \wedge x - y < 2$ à deux horloges x et y avec x_0 qui est une horloge constante à valeur 0.



2.3.2.1/ GRAPHE DES ZONES

L'abstraction par graphe des zones propose une solution pour réduire l'espace d'états tout en préservant des propriétés d'accessibilité équivalente au graphe des régions. Chaque état du graphe est un couple localité et zone. Il est formellement défini dans la Def. 7.

Définition 7 : Graphe des zones [DT98]

Soit $T = (L, l_0, \Sigma, X, T, F)$ un *TA*, son graphe des zones $Z(T)$ $\langle S^T, s_0^T, \Sigma, \Delta^T \rangle$ est un *LTS* défini ainsi :

- Les états de S^T sont des couples (l, z) où $l \in L$ et z est une zone.
- L'état initial est $s_0^T = (l_0, z_0)$ où l_0 est la localité initiale de T et z_0 est la zone initiale telle que toutes les horloges de X sont initialisées à zéro.
- Il y a un arc $((l, z), a, (l', z'))$ dans Δ^T si et seulement s'il existe une transition (l, a, g, X', l') dans T et une zone z" telles que :
  — z'' = est le successeur de z,
  — z'' satisfait g,
  — $z' = z''[X' := 0]$,
  — $z' = K - approximation(z')$,
  — z' est une zone non vide.

L'opération K-approximation est bien détaillée dans la sous section Sec. 2.3.2.3. C'est une opération d'extrapolation. Elle permet d'assurer la terminaison de la construction du graphe des zones et que le nombre des états du graphe des zones soit fini. On a besoin d'une structure de données pour représenter les zones. Une DBM est une structure de données qui permet de tester l'inclusion de deux zones et de calculer aisément les différentes opérations sur les zones comme l'intersection de deux zones, le successeur d'une zone, l'image d'une zone par une remise à zéro et la k-approximation d'une zone.

2.3.2.2/ LES DBMS

Une DBM (pour Difference Bounded Matrix) [Dil89] est une structure de données qui permet de représenter des systèmes de contraintes de différences [CLRS09], mais elle a un intérêt particulier pour la vérification des systèmes temporisés car elle permet de représenter les zones. Chaque zone d'horloges peut être représentée par une DBM. Les DBM peuvent être utilisées pour évaluer la satisfiabilité d'une zone donnée.

Une DBM d'une zone d'horloges z est une structure de données permettant de représenter une zone par une matrice carrée M de dimension $nb \times nb$ où nb est le nombre d'horloges de z, incluant une horloge spéciale x_0 qui est toujours égale à zéro. Un élément $M_{i,j}$ est sous la forme $(n_{ij}, \prec)$ où $n_{ij} \in \mathbb{N}$ et $\prec \in \{<, \leqslant\}$.

La Fig. 2.8 montre une DBM M qui représente la zone définie ainsi : $1 \leqslant x_1 \leqslant 4 \wedge 1 \leqslant x_2 \leqslant 3 \wedge x_1 - x_2 \leqslant 1$. Par exemple $M_{2,1} = x_2 - x_1 = (2, \leqslant)$. La valeur de 2 est obtenue : on a $1 \leqslant x_1 \leqslant 4$ et $1 \leqslant x_2 \leqslant 3$ donc $1 - 4 \leqslant x_2 - x_1 \leqslant 3 - 1$.

$x_i - x_j$	x_0	x_1	x_2
x_0	$(0, \leqslant)$	$(-1, \leqslant)$	$(-1, \leqslant)$
x_1	$(4, \leqslant)$	$(0, \leqslant)$	$(1, \leqslant)$
x_2	$(3, \leqslant)$	$(2, \leqslant)$	$(0, \leqslant)$

FIGURE 2.8 – DBM de la contrainte $1 \leqslant x_1 \leqslant 4 \wedge 1 \leqslant x_2 \leqslant 3 \wedge x_1 - x_2 \leqslant 1$

Les DBM peuvent être utilisées aussi pour évaluer la satisfiabilité d'une zone. Une DBM est non satisfiable si et seulement si la condition suivante est vérifiée :

$$\sum_{i=0}^{nb-1} (M_{i,i+1}) + M_{nb,1} < (0, \leqslant) \tag{2.1}$$

L'addition de deux éléments $M_{i,j} = (n, e)$ et $M_{i_1,j_1} = (n_1, e_1)$ est égale à $(n + n_1, min(e, e_1))$ où $<$ est représenté par 0 et $\leqslant$ par 1.

Les opérations d'intersection de deux zones, de remise à zéro d'une zone, de calcul du successeur d'une zone sont définies sur la structure de données DBM.

Intersection de deux zones $z \cap z'$: Soit M et M' deux DBM qui représentent les zones z et z' et $M'' = M \cap M'$, l'intersection de M et M' est une DBM M'' telle que $M''_{i,j} = min(M_{i,j}, M'_{i,j})$.

Remise à zéro d'une zone : La remise à zero d'un ensemble d'horloges X' dans une zone z est égale $z' = z[X' := 0]$. La zone z est représentée par la DBM M. La zone z' est représentée par la DBM M' telle que :

- $M'_{i,0} = (0, \leqslant)$ pour tout $x_i \in X'$

- $M'_{0,i} = (0, \leqslant)$ pour tout $x_i \in X'$
- $M'_{i,j} = M'_{0,j}$ pour tout $x_i \in X'$ et $x_j \notin X'$
- $M'_{j,i} = M'_{j,0}$ pour tout $x_i \in X'$ et $x_j \notin X'$

Successeur d'une zone $z^{\uparrow}$: la zone $z' = (z \cap g)[X' := 0]$ est la zone suivante de la zone z en franchissant une transition (l, a, g, X', l').

Exemple 2.3.4. On prend comme exemple la zone z définie ainsi : $1 \leqslant x_1 \leqslant 4 \wedge 1 \leqslant x_2 \leqslant 3 \wedge x_1 - x_2 \leqslant 1$ qui est représentée par une DBM M_g dans la Fig. 2.8. La zone z' est le successeur de la zone z en franchissant la transition $(l, a, x_1 \leqslant 4 \wedge x_2 \leqslant 2, \{x_2\}, l')$. On obtient la zone $z' = (z \cap g)[X' := 0]$ telle que $g = x_1 \leqslant 4 \wedge x_2 \leqslant 2$. La zone $0 \leqslant x_1 \leqslant 4 \wedge 0 \leqslant x_2 \leqslant 2$ est représentée par la DBM de la Fig. 2.9.(a). L'intersection entre z et g donne une DBM M' qui est présentée dans la Fig. 2.9.(b). Ainsi, $z'[\{x_2\} := 0]$ est représenté par la DBM M'' qui est présentée dans la Fig. 2.9.(c). Pour appliquer la remise à zéro de l'horloge x_2 sur $z \cap g$, on applique les règles suivantes :

- $M''_{2,0} = (0, \leqslant)$ // $i = 2$
- $M''_{0,2} = (0, \leqslant)$ // $i = 2$
- $M''_{2,0} = M''_{0,0}$ et $M''_{0,2} = M''_{0,0}$ // si $i = 2$ et $j = 0$
- $M''_{2,1} = M''_{0,1}$ et $M''_{1,2} = M''_{1,0}$ // si $i = 2$ et $j = 1$

On obtient finalement la zone $z'' = 1 \leqslant x_1 \leqslant 4 \wedge 1 \leqslant x_1 - x_2 \leqslant 4$.

$x_i - x_j$	x_0	x_1	x_2
x_0	$(0, \leqslant)$	$(0, \leqslant)$	$(0, \leqslant)$
x_1	$(4, \leqslant)$	$(0, \leqslant)$	$(4, \leqslant)$
x_2	$(2, \leqslant)$	$(2, \leqslant)$	$(0, \leqslant)$

(a).

$x_i - x_j$	x_0	x_1	x_2
x_0	$(0, \leqslant)$	$(-1, \leqslant)$	$(-1, \leqslant)$
x_1	$(4, \leqslant)$	$(0, \leqslant)$	$(1, \leqslant)$
x_2	$(2, \leqslant)$	$(2, \leqslant)$	$(0, \leqslant)$

(b).

$x_i - x_j$	x_0	x_1	x_2
x_0	$(0, \leqslant)$	$(-1, \leqslant)$	$(0, \leqslant)$
x_1	$(4, \leqslant)$	$(0, \leqslant)$	$(4, \leqslant)$
x_2	$(0, \leqslant)$	$(-1, \leqslant)$	$(0, \leqslant)$

(c).

FIGURE 2.9 – (a). DBM de la garde g, (b). DBM de $z \cap g$, (c). DBM de $z \cap g[\{x_2\} := 0]$

2.3.2.3/ K-APPROXIMATION OU EXTRAPOLATION [DT98]

L'utilisation des zones permet une représentation symbolique d'ensembles de valuations. L'application de la méthode de construction du graphe des zones sans appliquer l'opération K-approximation peut donner un nombre des zones qui reste infini et un algorithme de construction qui ne termine pas. La Fig. 2.10.(b) présente un graphe des zones infini pour le *TA* présenté dans la Fig. 2.10.(b) car la construction qui ne se termine pas si on n'applique pas l'opération $K - approximation$: la valeur de x croît d'une unité dans la localité l_1 et cela ne va jamais terminer car la garde est toujours satisfiable. Mais, le graphe de zones est fini si on applique par exemple la 3-approximation pour obtenir le graphe de zones qui est présenté dans la Fig. 2.10.(c).

La technique d'extrapolation permet de réduire le nombre de transitions concurrentes, le nombre de localités et par là même la complexité du calcul. Pour chaque automate temporisé, il existe une constante maximale $K \in \mathbb{N}$ qui apparaît dans ses gardes. L'extrapolation d'une zone z par rapport à K est la plus petite zone contenant z définie par des contraintes utilisant des constantes entre -K et K. L'intuition derrière cet opérateur d'extrapolation est

la suivante : le *TA* ne permet de tester que des contraintes d'horloges bornées par K ; si la valeur d'une d'horloge dépasse K, sa valeur précise n'a que peu d'importance, il est uniquement utile de savoir qu'elle est plus grande que K. Il faut d'abord noter qu'appliquer cet opérateur d'extrapolation à chaque étape du calcul itératif assure sa terminaison car il y a un nombre fini de zones. Pour chaque expression $x_i - x_j \prec n_{ij}$ d'une zone z, on applique les règles suivantes :

- les bornes supérieures qui sont supérieures à K sont éliminées,
- les bornes inférieures qui sont supérieures à K sont remplacées par K,
- toutes les autres bornes sont préservées.

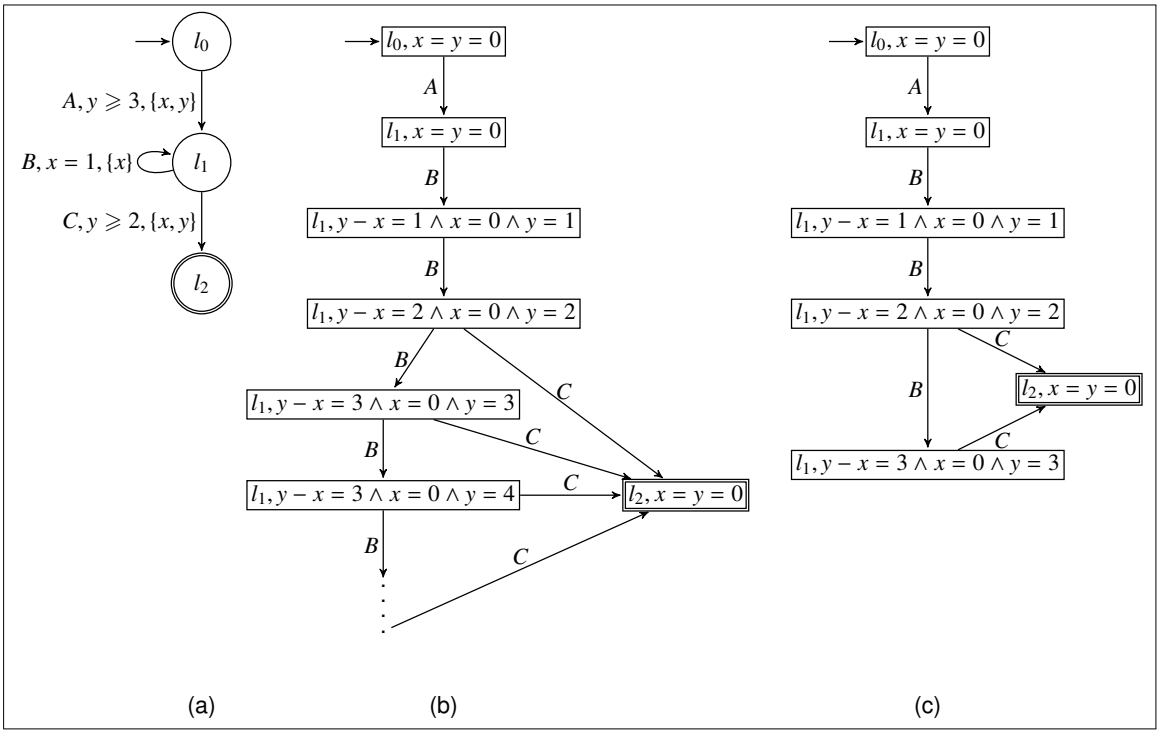


FIGURE 2.10 – (a). *TA*, (b). son graphe de zones infini et (c). son graphe de zones fini avec 3-approximation

Exemple 2.3.5 *(*extrapolation d'une zone).

On considère une zone $Z = 1 < x < 4 \wedge y > 4 \wedge x - y < 1$ et une valeur maximale $K = 2$, alors l'extrapolation de Z produit la zone $Z' = 1 < x \wedge y > 2 \wedge x - y < 1$.

2.3.3/ RÉDUCTION PAR DISCRÉTISATION DU TEMPS [HMP92] [ROB95]

Plusieurs méthodes de test sont basées sur une modélisation discrète du temps. La réduction par discrétisation consiste à ne considérer que des états entiers, c'est-à-dire des états dont les valeurs prises par les horloges sont des entiers positifs. Les états réels sont en général arrondis à des états entiers.

2.4/ RÈGLES DE SIMPLIFICATION D'UN AUTOMATE TEMPORISÉ AVEC ϵ–TRANSITIONS

Les automates temporisés contiennent des horloges qui évoluent de manière continue avec le temps et qui mesurent ainsi les délais séparant les différentes actions du système modélisé. Chaque transition contient une garde (sur la valeur des horloges) décrivant quand la transition peut être exécutée et un ensemble d'horloges qui doivent être remises à zéro lors du franchissement de la transition. Pour introduire des règles de simplification combinant plusieurs transitions successives, on a besoin de cette définition :

Définition 8 : Fermeture en arrière d'une contrainte [BPDG98]

Soit g une contrainte d'horloge et X est un ensemble d'horloges. La fermeture en arrière de g notée $\overleftarrow{g}^X$ est une formule qui est satisfaite par une valuation d'horloge v si g est satisfaite par la valuation d'horloge $v[X := 0] + t$ pour au moins une valeur $t \geq 0$

$$v \vDash \overleftarrow{g}^X \text{ si } \exists t.(t \geq 0 \wedge v[X := 0] + t \vDash g)$$

A l'instant où a lieu la remise à zéro des horloges de X, la fermeture en arrière d'une contrainte est satisfaite si cette contrainte pourra être satisfaite dans le futur, à un instant t, après la remise à zéro des horloges de X.

Remarque (sur la fermeture en arrière d'une contrainte)

Il n'est pas vrai que $\overleftarrow{g_1 \wedge g_2}^X = \overleftarrow{g_1}^X \wedge \overleftarrow{g_2}^X$.

Démonstration.

On montre que $\overleftarrow{x > 3 \wedge x < 2}^{\{x\}}$ est un contre exemple. On montre qu'il n'est pas vrai que $\overleftarrow{x > 3 \wedge x < 2}^{\{x\}} = \overleftarrow{x > 3}^{\{x\}} \wedge \overleftarrow{x < 2}^{\{x\}}$ pour une valuation v quelconque :

- pour toute valuation v, $v \nvDash \overleftarrow{x > 3 \wedge x < 2}^{\{x\}}$ car il n'existe pas t tel que $t \geqslant 0 \wedge x = 0 + t \vDash x > 3 \wedge x < 2$.
- par contre, il existe une valuation v telle que $v \vDash \overleftarrow{x > 3}^{\{x\}} \wedge \overleftarrow{x < 2}^{\{x\}}$ car il existe t_1 et t_2 tels que $t_1 \geqslant 0 \wedge x = 0 + t_1 \vDash x > 3 \wedge t_2 \geqslant 0 \wedge x = 0 + t_2 \vDash x < 2$, par exemple $t_1 = 4$ et $t_2 = 1$.

□

Une transition (l, a, g, X', l') est appelée une ϵ-transition [AD90b] ou une transition silencieuse si $a = \epsilon$.

Théorème 2.4.1 *(*[BPDG98]*).* Pour chaque automate temporisé avec ϵ-transitions, il existe un automate temporisé équivalent sans ϵ-transitions.

Ce théorème est prouvé en construisant à partir de n'importe quel automate temporisé avec ϵ-transitions, un automate temporisé qui accepte le même langage temporisé.
En utilisant la fermeture en arrière d'une contrainte, on distingue deux types de transitions dans un automate temporisé : des transitions élémentaires et des transitions fusionnées. Une transition élémentaire est une transition dont la garde est une contrainte d'horloge sans fermeture en arrière. Une transition fusionnée est une transition dont la garde est

une contrainte d'horloge avec fermeture en arrière. On définit la règle de fusion de deux ϵ-transitions élémentaires successives dans la Def. 9.

Définition 9 : Règle de fusion de deux ϵ-transitions élémentaires successives

R_1 : s'il existe les deux ϵ-transitions successives $l_1 \xrightarrow{\epsilon,g_1,X_1} l_2 \xrightarrow{\epsilon,g_2,X_2} l_3$, on les fusionne en l'$\epsilon$-transition suivante : $l_1 \xrightarrow{\epsilon,g_1\wedge\overleftarrow{g_2}^{X_1},X_2} l_3$.

On définit la règle de fusion de $n-1$ $\epsilon-$transitions élémentaires successives dans la Def. 10.

Définition 10 : Règle de fusion de $n-1$ $\epsilon-$transitions élémentaires successives

R_2 : la succession de $n-1$ $\epsilon-$transitions $l_1 \xrightarrow{\epsilon,g_1,X_1} l_2 \xrightarrow{\epsilon,g_2,X_2} l_3 ... l_{n-1} \xrightarrow{\epsilon,g_{n-1},X_{n-1}} l_n$

est fusionnée en l'ϵ-transition suivante : $l_1 \xrightarrow{\epsilon,g_1\wedge g_2\wedge...\wedge\overleftarrow{\overleftarrow{g_{n-1}}^{...}}^{X_{n-2}},X_{n-1}} l_n$ avec $n \geqslant 3$.

Proposition 2.4.1. Pour toute valuation v_1 telle que $v_1 \vDash g_1$ et pour toute valuation v_n, il existe un chemin qui part de l'état (l_1, v_1) et atteint l'état(l_n, v_n) et qui exécute $l_1 \xrightarrow{\epsilon,g_1,X_1} l_2 \xrightarrow{\epsilon,g_2,X_2} l_3 ... l_{n-1} \xrightarrow{\epsilon,g_{n-1},X_{n-1}} l_n$, si et seulement s'il existe une transition $(l_1, v_1) \xrightarrow{\epsilon} (l_n, v_n)$ qui exécute l'ϵ-transition suivante : $l_1 \xrightarrow{\epsilon,g_1\wedge g_2\wedge...\wedge\overleftarrow{\overleftarrow{g_{n-1}}^{...}}^{X_{n-2}},X_{n-1}} l_n$.

Démonstration.

On montre que si on exécute la séquence de transitions successives élémentaire $l_1 \xrightarrow{\epsilon,g_1,X_1} l_2 \xrightarrow{\epsilon,g_2,X_2} l_3 ... l_{n-1} \xrightarrow{\epsilon,g_{n-1},X_{n-1}} l_n$ ou la transition fusionnée $l_1 \xrightarrow{\epsilon,g_1\wedge g_2\wedge...\wedge\overleftarrow{\overleftarrow{g_{n-1}}^{...}}^{X_{n-2}},X_{n-1}} l_n$, on atteint la localité l_n à partir d'une valuation d'horloge v depuis la localité l_1 avec la même valuation d'horloges.

Pour qu'une séquence des transitions élémentaires successives $l_1 \xrightarrow{\epsilon,g_1,X_1} l_2 \xrightarrow{\epsilon,g_2,X_2} l_3 ... l_{n-1} \xrightarrow{\epsilon,g_{n-1},X_{n-1}} l_n$ soit exécutée, les contraintes suivantes doivent être satisfaites pour atteindre la localité l_i depuis la localité l_1 à partir de la valuation d'horloge v :

$l_2 : v \vDash g_1$, /* on atteint la localité l_2 si la valuation d'horloge v satisfait la garde g_1*/
$l_3 : \exists t_1.(t_1 \geqslant 0 \wedge v[X_1 := 0] + t_1 \vDash g_2)$,
$l_4 : \exists t_1, t_2.((t_1 \geqslant 0 \wedge t_2 \geqslant 0 \wedge (v[X_1 := 0] + t_1)[X_2 := 0] + t_2 \vDash g_3)$,
...
$l_n : \exists t_1, ..., t_{n-2}.(t_1 \geqslant 0 \wedge ... \wedge t_{n-2} \geqslant 0 \wedge (..v[X_1 := 0] + t_1)[X_2 := 0] + t_2 \vDash g_3 ... + t_{n-3})[X_{n-2:=0}] + t_{n-2} \vDash g_{n-1})$.

Pour qu'une transition fusionnée soit exécutée, les contraintes suivantes doivent être satisfaites. On pose que v est une valuation d'horloges qui satisfait la contrainte g_1.

- $g_1 \wedge \overleftarrow{\overleftarrow{g_2}^{X_1}}^{X_1}$: cette contrainte est vraie si $\exists t_1.(v \vDash g_1 \wedge t_1 \geqslant 0 \wedge v[X_1 := 0] + t_1 \vDash g_2)$,
- $g_1 \wedge g_2 \wedge \overleftarrow{\overleftarrow{g_3}^{X_2}}^{X_1}$: cette contrainte est vraie si $\exists t_1, t_2.(v \vDash g_1 \wedge (t_1 \geqslant 0 \wedge v[X_1 := 0] + t_1 \vDash g_2) \wedge (t_2 \geqslant 0 \wedge (v[X_1 := 0] + t_1)[X_2 := 0] + t_2 \vDash g_3)$,
- ...

- $l_1 \xrightarrow{\epsilon, g_1 \wedge g_2 \wedge ... \wedge \overleftarrow{g_{n-1}}^{X_{n-2}}, X_{n-1}} l_n$: cette contrainte est vraie $\exists t_1, ..., t_{n-2}.(t_1 \geqslant 0 \wedge ... \wedge t_{n-2} \geqslant 0 \wedge (..v[X_1 := 0] + t_1)[X_2 := 0] + t_2 \vDash g_3 ... + t_{n-3})[X_{n-2} := 0] + t_{n-2} \vDash g_{n-1})$.

Dans les deux cas, on arrive à la localité l_n avec la même valuation. Donc, on peut fusionner des transitions successives en une seule transition qui accepte le même mot temporisé. □

Exemple 2.4.1. La Fig. 2.11 montre la fusion de trois ϵ-transitions successives en une seule ϵ-transition.

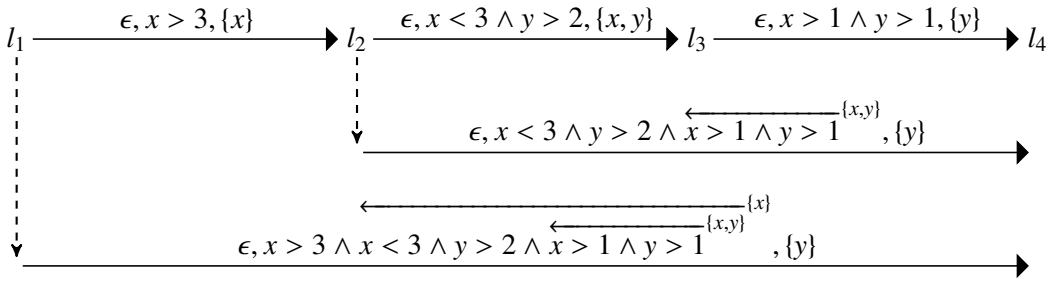


FIGURE 2.11 – Fusion de trois ϵ-transitions successives en une seule transition déchlenchable

Une exécution possible de la séquence de 3 ϵ-transitions successives de la Fig. 2.11 est : $(l_1, (0, 0)) \rightarrow^4 (l_1, (4, 4)) \rightarrow^\epsilon (l_2, (0, 4)) \rightarrow^1 (l_2, (1, 5)) \rightarrow^\epsilon (l_3, (0, 0)) \rightarrow^2 (l_3, (2, 2)) \rightarrow^\epsilon (l_4, (2, 0))$.

La contrainte $x > 3 \wedge x < 3 \wedge y > 2 \wedge \overleftarrow{x > 1 \wedge \overleftarrow{y > 1}^{\{x,y\}}}^{\{x\}}$ est satisfaite sur la valuation $(4, 4)$ si $\exists t_1, t_2.(v \vDash x > 3 \wedge (t_1 \geqslant 0 \wedge v[x := 0] + t_1 \vDash (x < 3 \wedge y > 2)) \wedge (t_2 \geqslant 0 \wedge (v[x := 0] + t_1)[x := 0, y := 0] + t_2 \vDash (x > 1 \wedge y > 1))$. On prend par exemple $t_1 = 1$ et $t_2 = 2$.

Exemple 2.4.2.

La Fig. 2.12 montre la fusion de trois ϵ-transitions successives en une seule ϵ-transition. Ces ϵ-transitions successives appartiennent à un *TA* avec deux horloges x et y. Une configuration de ce *TA* est sous la forme $(s, (x, y))$ où $s \in S$.

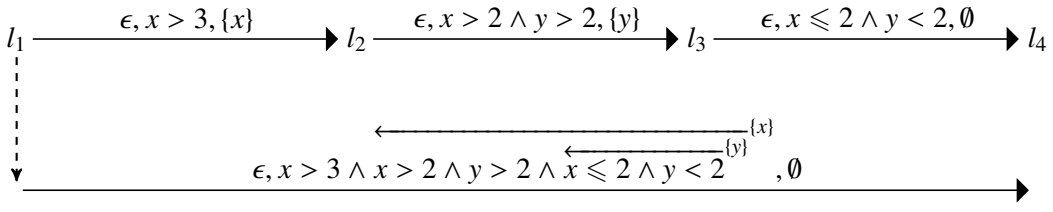


FIGURE 2.12 – Fusion de trois ϵ-transitions successives en une seule transition non déclenchable

On remarque que la localité l_4 n'est pas atteignable à partir de la localité l_1. Vu que le temps s'écoule et qu'il n'y a pas de remise à zéro de l'horloge x entre l_2 et l_3, alors, la condition $x \leqslant 2 \wedge y < 2$ ne peut pas être satisfaite en l_3 car la contrainte $x > 2$ doit être satisfaite pour atteindre l_3. En appliquant la règle de fusion, on obtient la con-

trainte suivante $x > 3 \wedge x > 2 \wedge y > 2 \wedge \overleftarrow{\overleftarrow{x \leqslant 2 \wedge y < 2}^{\{y\}}}^{\{x\}}$. La satisfiabilité de cette contrainte s'exprime par $\exists t_1, t_2.(v \vDash x > 3 \wedge (t_1 \geqslant 0 \wedge v[x := 0] + t_1 \vDash (x > 2 \wedge y > 2)) \wedge (t_2 \geqslant 0 \wedge (v[x := 0] + t_1)[y := 0] + t_2 \vDash (x \leqslant 2 \wedge y < 2))$. Elle n'est pas satisfiable car il n'existe pas de valeurs de t_1 et t_2 qui satisfassent cette contrainte.

2.5/ DÉTERMINISATION DES AUTOMATES TEMPORISÉS

Les automates temporisés ne sont pas tous déterminisables en général. La déterminisabilité d'un *TA* est indécidable. Pour la vérification, il est essentiel pour l'expressivité d'autoriser la modélisation des spécifications par des automates non-déterministes. Mais, pour la génération des cas de tests, le déterminisme est nécessaire pour effectuer l'observation des tests sur une implémentation. La déterminisation est décidable pour quelques classes d'automates temporisés déterminisables qui appartiennent à une sous-classe stricte des automates temporisés [BBBB09]. On trouve par exemple dans cette sous classes les automates :

- *TA* $Event - clock$ [AFH99] qui sont des automates temporisés qui possèdent seulement une horloge pour chaque action et qui sont réinitialisées à chaque occurrence de l'action.
- *TA* à réinitialisations entières qui sont des automates temporisés où toutes les contraintes sont sous la forme $(x = c)$ [SPKM08].
- *TA* fortement non-Zeno [Tri05] : un automate est fortement non-Zeno si dans chaque cycle $l_1 \rightarrow l_2 ... \rightarrow l_1$, il existe une horloge qui est remise à zéro et minorée par 1. Une unité de temps s'est au plus écoulée dans chaque cycle.

En littérature, il existe deux approches de déterminisation. On trouve par exemple [BBBB09] qui propose une méthode de déterminisation exacte mais sans assurer la terminaison. Krichen et Tripakis[KT09] proposent une méthode de sur-approximation de déterminisation des automates temporisés avec une terminaison garantie. La méthode de determinisation présentée dans [BSJK15] produit un automate déterminisé si possible et avec une sur-approximation déterministe sinon. Elle est à la fois plus générale que la procédure de [BBBB09] et plus précise que la méthode de sur-approximation présentée dans [KT09].
Nous présentons dans la partie 2.5.1 une méthode de déterminisation exacte. Nous présentons dans la partie 2.5.2 une méthode de déterminisation avec sur-approximation. Une méthode de déterminisation avec sur-approximation plus précise est présentée dans la partie 2.5.3.

2.5.1/ DÉTERMINISATION EXACTE [BBBB09]

[BBBB09] propose une méthode de déterminisation exacte. La déterminisation d'un automate temporisé T est appliquée en quatre étapes :

1. Dépliage de T sous la forme d'un arbre temporisé infini : cette étape permet de définir un arbre infini qui reconnaît les mêmes traces que T. Elle consiste à déplier T en utilisant une nouvelle horloge à chaque niveau de l'arbre. L'arbre obtenu conserve les traces de T.

2. Abstraction des régions : cette étape consiste à transformer une transition en plusieurs régions si plusieurs régions. Elle préserve aussi les traces de T.
3. Déterminisation symbolique : cette étape consiste à déterminiser l'arbre infini. On obtient un arbre reconnaissant les mêmes traces.
4. Répliage de l'arbre : cette étape consiste à réduire le nombre d'horloges en oubliant les horloges inutiles à la volée.

La terminaison ce ces étapes de déterminisation n'est pas garantie.

2.5.2/ DÉTERMINISATION AVEC SUR-APPROXIMATION [KT09]

Krichen et Tripakis ont proposé une approche dans [KT09] qui permet de construire un automate temporisé déterministe acceptant au moins les mots acceptés par l'automate temporisé donné. Ils ont proposé un algorithme qui permet de construire un automate temporisé déterministe *DTA* (Deterministic Timed Automata) avec une seule horloge à partir d'un automate temporisé à plusieurs horloges. Cette horloge unique est remise à zéro à chaque transition. Les localités des *DTA* sont des localités symboliques. Elles sont étiquetées par un couple composé d'une localité du *TA* et d'une zone de valuation des horloges. Cette zone est exprimée en fonction des horloges du *TA* et de la nouvelle horloge y.

Avant de présenter la définition et les principes de calcul des transitions d'un *DTA* à partir d'un *TA* donné, on a besoin de définir les notations suivantes :

- $usucc(l^D) = \{l^{D'} \mid \exists(l,v).((l,v) \in l^D \wedge \exists\rho.(\rho \in RT(\{\tau\}) \wedge (l,v) \rightarrow^{\rho} (l',v') \wedge (l',v') \in l^{D'}))\}$. C'est l'ensemble des localités symboliques atteignables à partir de la localité symbolique l^D par une séquence d'actions non observables.
- $dsucc(l^D, a) = \{l^{D'} \mid \exists(l,v).((l,v) \in l^D \wedge (l,v) \rightarrow^{a} (l',v') \wedge (l',v') \in l^{T'}\}$. C'est l'ensemble de localités symboliques atteignables à partir de l^D par l'action a.
- $\Delta_a = \{tr \mid tr \in \Delta \wedge \exists(l,g,d,X',l').(tr = (l,a,g,d,X',l'))\}$ est l'ensemble de transitions de Δ étiquetées par a.
- $\Delta_a(l^D, u) = \{(l,a,g,d,X',l') \in \Delta_a \mid \exists(l,v).((l,v) \in l^D$ et $v \wedge u \wedge g$ est satisfiable$\}$ est l'ensemble des transitions étiquetées par a dont les gardes sont satisfaites par une localité symbolique (l,v) de l^D où $v \in CC(X \cup \{y\})$ et où l'horloge y est égale à u.

Deux valuations d'horloges u_1 et u_2 sont équivalentes pour les transitions étiquetées par l'étiquette a et pour la localité symbolique l^D si et seulement si l'ensemble de transitions étiquetées par l'étiquette a déclenchables aux instants u_1 et u_2 sur cet ensemble d'états sont identiques. Une relation d'équivalence entre deux partitions u_1 et $u_2 \in Grd(\{y\})$ est définie ainsi :

$$u_1 \backsim^{a}_{l^D} u_2 \text{ si et seulement si } \Delta_a(l^D, u_1) = \Delta_a(l^D, u_2). \tag{2.2}$$

L'exemple 2.5.1 illustre un exemple de calcul des classes d'équivalences à partir d'une localité symbolique donnée et d'une étiquette donnée.

Exemple 2.5.1. On suppose qu'on a une localité $l^D_1 = \{(l_1, -2 \leqslant x - y \leqslant 1)\}$ qui appartient à un *DTA* et deux transitions de son *TA* étiquetées par l'étiquette A : $tr_1 = (l_1, A, g_1, \emptyset, l_2)$ où $g_1 = x \leqslant 3$ et $tr_2 = (l_1, A, g_2, \emptyset, l_3)$ où $g_2 = x \geqslant 2$. Il existe trois classes d'équivalences :

- $y < 1$: l'intersection entre $(-2 \leqslant x - y \leqslant 1)$, $(x \geqslant 2)$ et $(y < 1)$ est vide comme on peut le voir dans la Fig. 2.13. Donc la transition tr_2 n'est pas déclenchable dans cet espace. L'intersection entre $(-2 \leqslant x - y \leqslant 1)$, $(x \leqslant 3)$ et $(y < 1)$ n'est pas vide et donc la transition tr_1 est déclenchable dans cet espace. Alors, $\Delta_A(\{(l_1, -2 \leqslant x - y \leqslant 1)\}, y < 1) = \{tr_1\}$.
- $1 \leqslant y \leqslant 5$: l'intersection entre $(-2 \leqslant x - y \leqslant 1)$, $(x \geqslant 2)$ et $(1 \leqslant y \leqslant 5)$ n'est pas vide comme on peut le voir dans la Fig. 2.13. De même, l'intersection entre $(-2 \leqslant x - y \leqslant 1)$, $(x \leqslant 3)$ et $(1 \leqslant y \leqslant 5)$ n'est pas vide. Donc, les transitions tr_1 et tr_2 sont déclenchables et $\Delta_A(\{(l_1, -2 \leqslant x - y \leqslant 1)\}, 1 \leqslant y \leqslant 5) = \{tr_1, tr_2\}$. On a considéré la partition $1 \leqslant y \leqslant 5$ comme une seule classe d'équivalence car les 9 partitions suivantes sur y sont équivalentes d'après la condition 2.2.

$$\left.\begin{array}{l}
\Delta_A((l_1, -2 \leqslant x - y \leqslant 1), 1 \leqslant y \leqslant 1) \\
\Delta_A((l_1, -2 \leqslant x - y \leqslant 1), 1 < y < 2) \\
\Delta_A((l_1, -2 \leqslant x - y \leqslant 1), 2 \leqslant y \leqslant 2) \\
\Delta_A((l_1, -2 \leqslant x - y \leqslant 1), 2 < y < 3) \\
\Delta_A((l_1, -2 \leqslant x - y \leqslant 1), 3 \leqslant y \leqslant 3) \\
\Delta_A((l_1, -2 \leqslant x - y \leqslant 1), 3 < y < 4) \\
\Delta_A((l_1, -2 \leqslant x - y \leqslant 1), 4 \leqslant y \leqslant 4) \\
\Delta_A((l_1, -2 \leqslant x - y \leqslant 1), 4 < y < 5) \\
\Delta_A((l_1, -2 \leqslant x - y \leqslant 1), 5 \leqslant y \leqslant 5)
\end{array}\right\} = \{tr_1, tr_2\}$$

- $y > 5$: l'intersection entre $(-2 \leqslant x - y \leqslant 1)$, $(x \geqslant 2)$ et $(y > 5)$ n'est pas vide comme on peut le voir dans la Fig. 2.13. Donc la transition tr_2 est déclenchable dans cet espace. L'intersection entre $(-2 \leqslant x - y \leqslant 1)$, $(x \leqslant 3)$ et $(y > 5)$ est vide et donc la transition tr_1 n'est pas déclenchable dans cet espace. Alors, $\Delta_A(\{(l_1, -2 \leqslant x - y \leqslant 1)\}, y > 5) = \{tr_2\}$.

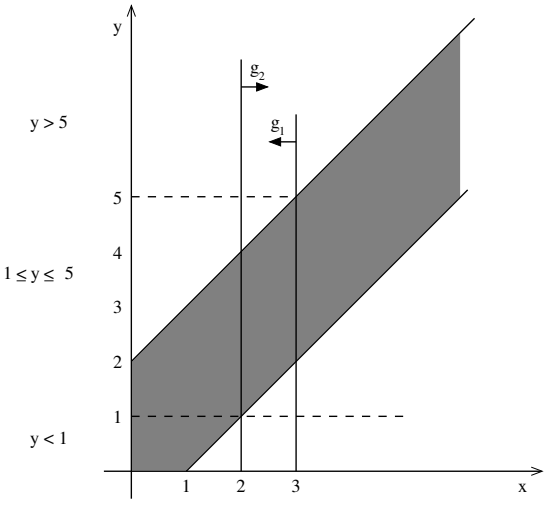


FIGURE 2.13 – Illustration dans [KT09] de $\curvearrowright^{a}_{(l_1, -2 \leqslant x-y \leqslant 1)}$. La zone $-2 \leqslant x - y \leqslant 1$ est grisée

L'algorithme 2.5.1 est un algorithme de déterminisation avec sur-approximation [KT09] d'un *TA* donnée. La construction du *DTA* se fait à partir de la localité initiale qui est étiquetée par la localité initiale du *TA* et par une valuation d'horloges initiale où toutes

les horloges et l'horloge y sont initialisées à zéro. Elle s'effectue ensuite par calculs successifs des meilleures sur-approximations des gardes des transitions ajoutées.

Soit $l_0^{X\cup\{y\}}$ une localité symbolique initiale formée par la localité initiale l_0 et la valuation d'horloges initiale où les horloges de $X \cup \{y\}$ sont initialisées à zéro. La localité symbolique initiale l_0^D est égale à $usucc(l_0^{X\cup\{y\}})$ où toutes les horloges de T et l'horloge y sont initialisées à zéro. Le principe de calcul des localités symboliques et des transitions du *DTA* est de répéter les étapes suivantes : la sélection d'une localité symbolique l^D qui n'est pas encore ajoutée dans L^D et puis l'ajout des nouvelles transitions à Δ^D en appliquant : pour chaque action $act \in \Sigma$, les partitions associées à l^D sont $u \in \{[0,0],]0,1[, [1,1], ..., [K,K],]K,\infty)\}$ où K est la constante maximale qui apparait dans une zone de la localité symbolique l^D ou dans la garde de l'une des transitions de Δ. Pour réduire le nombre des partitions, on fusionne les intervalles successifs et qui ont le même ensemble $\Delta_{act}(l^D, u)$. Pour chaque grande partition u : si $\Delta_{act}(l^D, u) \neq \emptyset$ alors, la transition $(l^D, act, u, \{y\}, usucc(dsucc(l^D \cap u, act)))$ est ajoutée dans Δ^D et la localité symbolique $usucc(dsucc(l^D \cap u, act))$ est ajoutée à L^D si elle n'existe pas déjà. L'opérateur **usucc** peut conduire à un nombre infini de localités [DT98]. Pour rendre le nombre d'état fini, on passe par une technique d'abstraction. Cette technique vise à assurer que le nombre de localités possibles du *DTA* reste fini. Elle s'appelle technique d'extrapolation [DT98][BY03][Bou04] ou de normalisation. Elle est bien expliquée dans la section 2.3.2.3.

Algorithme 2.5.1 Calcul du *DTA* à partir d'un *TPAIO* [KT09]

INPUTS : un *TA* $T = (L, l_0, \Sigma, X, \Delta, F)$,
OUTPUTS : un *DTA* $D = (L^D, l_0^D, \Sigma, \{y\}, \Delta^D, F^D)$
VARIABLES : $act \in \Sigma$, $New\_SL \subseteq (L \times CC(X \cup \{y\}))$, $u \in Grd(\{y\})$, U ensemble de $Grd(\{y\}$, $l_0^D, l^D, l^{D'} \in L \times CC(X \cup \{y\})$

BEGIN
1: $l_0^D \leftarrow usucc(l_0^{X\cup\{y\}})$
2: $New\_SL \leftarrow \{l_0^D\}$ // localités symboliques non traitées
3: $L^D \leftarrow \emptyset$, $\Delta^D \leftarrow \emptyset$
4: **tant que** $New\_SL \neq \emptyset$ **faire**
5: $l^D \leftarrow choisirElement(New\_SL)$
6: **pour** chaque $act \in \Sigma$ **faire**
7: $U \leftarrow$ les classes d'équivalence induites par $\sim_{l^D}^{act}$
8: **pour** $u \in U$ **faire**
9: $l^{D'} \leftarrow usucc(dsucc(l^D \cap u, act))$
10: **si** $l^{D'} \neq \emptyset$ **alors**
11: $\Delta^D \leftarrow \Delta^D \cup \{(l^D, act, u, \{y\}, l^{D'})\}$
12: **fin si**
13: **fin pour**
14: **fin pour**
15: $New\_SL \leftarrow New\_SL \setminus \{l^D\}$
16: $L^D \leftarrow L^D \cup \{l^D\}$
17: **fin tant que**
FIN.

L'algorithme se termine quand $New\_SL = \emptyset$. En appliquant cet algorithme on obtient un *DTA* qui est un *TA* dont le nombre de localités symboliques peut être double-

ment exponentiel par rapport au nombre des localités du *TA* de départ. La fonction $choisirElement(New\_SL)$ retourne un élément quelconque de l'ensemble New_SL. L'avantage de cette méthode est qu'on a toujours un nombre de localités symboliques inférieur au nombre de régions du graphe [SVD01] des régions d'un *TA* donné.

Exemple 2.5.2. La Fig. 2.14.(b) présente le *DTA* du *TA* présenté dans la Fig. 2.14.(a). Le *DTA* a une seule horloge nommée y. La localité symbolique initiale est $\{(l_0, x = y)\}$ où toutes les horloges sont égales. On trouve par exemple que la trace $0.5\ A\ 0.5\ C$ est acceptée par le *DTA* et le *TA*. Par contre, on trouve par exemple que la trace $0.9\ A\ 0.8\ C$ est acceptée par le *DTA*, alors qu'elle n'est pas acceptée par le *TA*. Ce problème illustre que le *DTA* est une sur-approximation du *TA*.

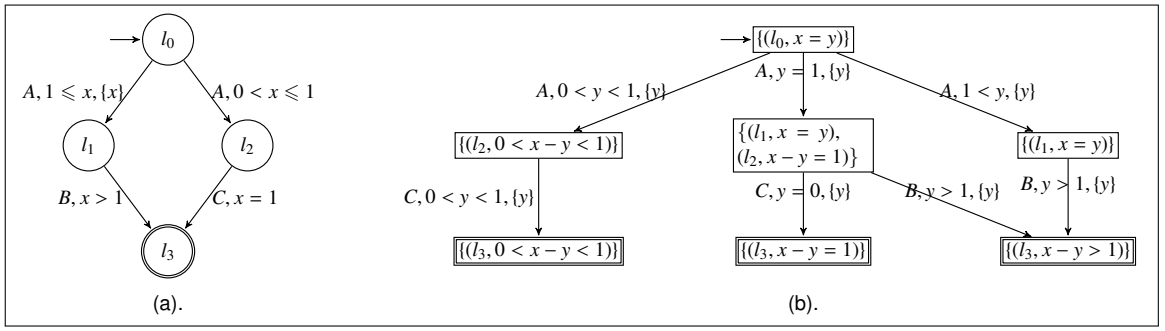


FIGURE 2.14 – (a) *TA* et (b) son *DTA*

2.5.3/ DÉTERMINISATION AVEC SUR-APPROXIMATION PLUS PRÉCISE

Les auteurs de [BBBB09] proposent de combiner la méthode de déterminisation exacte et la méthode de déterminisation avec sur-approximation en une seule méthode. Cette méthode combine les avantages des deux approches en une seule méthode. Elle est basée sur les mêmes calculs de déterminisation que [KT09] en cherchant les meilleurs réinitialisations possibles à chaque transition. Un jeu fini $turn - based$ de sûreté [BCD05] est construit. Il a deux types de joueurs : "Determinisator" et "Spoiler". Determinisator choisit quand réinitialiser la nouvelle horloge y et les choix de Spoiler sont des couples (a, r) où a est une action et r est une région sur y.

Dans chaque état, l'ensemble des configurations dans lesquelles on peut résider est retenu, c'est-à-dire dans quelles localités de *TA* T, les estimations des horloges de T en fonction de l'horloge y. De plus, les états que Determinisator veut éviter de manière à ce que les stratégies gagnantes de Determinisator correspondent à un déterminisé exact de T sont grisés. Dans un état, les configurations obtenues par une sur-approximation sont marquées. L'état est considéré comme sûr tant qu'il reste une configuration non marquée dans un état. Sinon, il est grisé.

2.6/ AUTOMATES À PILE

Les systèmes à pile sont d'anciens modèles. Ils ont été largement étudiés [MS85][ABB97][BEM97a][Wal00]. Ils sont utilisés aussi pour abstraire des

programme récursifs [DHKM14b] et des programmes booléens.

2.6.1/ DÉFINITION

Un automate à pile [ABB97] est une machine abstraite utilisée en théorie des langages, généralisant les automates finis. Il est composé d'un ensemble fini de localités et d'une relation de transition entre ces localités. Il dispose en plus d'une mémoire non bornée organisée en pile (dernier entré / premier sorti). Il peut stocker des éléments et en retirer de la pile, mais, uniquement au sommet.
Formalisons la notion de *PA* (Pushdown Automata) dans la Def. 11. Un *PA* est normalisé quand il sépare ses actions en 3 catégories : empiler, dépiler et faire une autre action.

Définition 11 : *PA* [ABB97] - Automate à Pile

Un *PA* est un 6-uplet $T = (L, s_0, \Sigma, \Gamma, \Delta, F)$ où :
- L est un ensemble fini de localités,
- $l_0 \in L$ est la localité initiale,
- Σ est un ensemble fini de symboles d'entrée,
- Γ est ensemble fini de symboles spécifiques à la pile (on a $\Sigma \cap \Gamma = \emptyset$),
- $F \subseteq L$ est un ensemble de localités finales,
- $\Delta \subseteq L \times \Sigma \cup \Gamma^{+-} \times L$ où $\Gamma^{+-} = \{a^+ \mid a \in \Gamma\} \cup \{a^- \mid a \in \Gamma\}$ est l'ensemble des transitions.

Une transition est un triplet (l, a, l') tel que :
- l est une localité source,
- l' est une localité cible,
- a est l'action qui va être exécutée quand la transition sera franchie. L'action peut être :
  — a^- : dépiler le symbole a au sommet de la pile avec $a \in \Gamma$,
  — a^+ : empiler le symbole a dans la pile avec $a \in \Gamma$,
  — A : lire le symbole A sans changer l'état de la pile avec $A \in \Sigma$.

Le nombre de symboles utilisés dans la pile est fini. La sémantique d'un automate à pile est un *LTS* $< S^T, s_0^T, \Sigma \cup \Gamma^{+-}, \Delta^T >$ où S^T est un ensemble d'états qui sont des couples de $L \times \Gamma^*$, $s_0^T = (l_0, \perp)$ où $\perp$ représente la pile vide et Δ^T est un ensemble de transitions qui sont des triplets $((l, p), a, (l', p'))$ où (l, p) et (l', p') sont des états. Les transitions de Δ^T sont définies ainsi :
- $((l, p), a^+, (l', p.a)) \in \Delta^T$ si $(l, a^+, l') \in \Delta$ et $(l, p) \in S^T$
- $((l, p.a), a^-, (l', p)) \in \Delta^T$ si $(l, a^-, l') \in \Delta$ et $(l, p.a) \in S^T$
- $((l, p), A, (l', p)) \in \Delta^T$ si $(l, A, l') \in \Delta$ et $(l, p) \in S^T$.

Il existe trois modes de reconnaissance d'un langage par un *PA*.
- Reconnaissance par localité finale : un mot est accepté si, à partir de la localité initiale, il peut être entièrement lu en arrivant à une localité de F, ceci quel que soit le contenu de la pile à ce moment-là. La pile est supposée vide au départ.
- Reconnaissance par pile vide : ce mode de reconnaissance autorise un *PA* à accepter un mot si, à partir de la localité initiale, il peut être entièrement lu en vidant la pile.
- Reconnaissance par localité finale et pile vide : ce mode de reconnaissance autorise un *PA* à accepter un mot si, à partir de la localité initiale, il peut être entièrement lu en vidant la pile et en arrivant à une localité de F.

2.6.2/ ACCESSIBILITÉ

[BEM97a] et [FWW97] montrent que pour un automate à pile T, l'ensemble des configurations accessibles depuis la localité initiale de T peut être reconnu par un automate fini qui est construit de façon effective en temps polynomial.

Les auteurs de [FWW97] proposent une méthode de calcul de l'ensemble des mots de pile accessibles dans un automate à pile sans symboles d'entrée $T = (L, s_0, \Sigma, \Gamma, \Delta, F)$ avec $\Sigma = \emptyset$. Ils proposent de calculer une représentation de toutes les localités atteignables de T et toutes les configurations possibles dans T. Cette représentation est un automate d'atteignabilité $T^R = \langle L, l_0, \{\epsilon\} \cup \Gamma, \Delta^R, F\rangle$. L'automate d'atteignabilité est un automate fini avec $\Sigma = \{\epsilon\} \cup \Gamma$. Une ϵ–transition (l, ϵ, l') est une transition qui atteint la localité l' avec le même contenu de pile qu'en l. Une transition (l, a, l') est une transition qui atteint la localité l' en ajoutant a au sommet de la pile. Ils proposent un ensemble de règles qui sont présentées dans la Def. 12 qui permettent de calculer de nouvelles ϵ-transitions à partir de transitions existantes. La règle R_1 ajoute des ϵ-transitions réflexives dans T^R pour chaque localité de T. La règle R_2 ajoute des transitions étiquetées par a dans T^R pour chaque transition étiquetées par a^+ dans T. La règle R_3 fusionne en une seulement ϵ-transition une transition d'empilement, suivie d'une ϵ-transition ; puis d'une transition de dépilement de symbole précédemment empilé. La règle R_4 fusionne deux ϵ-transitions consécutives en une seule ϵ-transition.

Définition 12 : *RA* d'un *PA*- Automate d'Accessibilité d'un Automate à Pile

Le *RA* d'un *PA* $(L, l_0, \emptyset, \Gamma, \{y\}, \Delta, F)$ est un automate fini $(L, l_0, \{\epsilon\} \cup \Gamma, \Delta^R, F)$ où $\Delta^R \subseteq L \times \{\epsilon\} \cup \Gamma^{+-} \times L$ est la relation satisfaisant les conditions suivantes :

- $\mathbf{R}_1$: $(l, \epsilon, l) \in \Delta^R$ si $l \in L$,
- $\mathbf{R}_2$: $(l_1, a, l_2) \in \Delta^R$ si $(l_1, a^+, l_2) \in \Delta$,
- $\mathbf{R}_3$: $(l_1, \epsilon, l_4) \in \Delta^R$ si $(l_1, a^+, l_2) \in \Delta$, $(l_2, \epsilon, l_3) \in \Delta^R$ et $(l_3, a^-, l_4) \in \Delta$,
- $\mathbf{R}_4$: $(l_1, \epsilon, l_3) \in \Delta^R$ si $(l_1, \epsilon, l_2) \in \Delta^R$ et $(l_2, \epsilon, l_3) \in \Delta^R$

Le principe de l'algorithme 2.6.1 de calcul de *RA* de [FWW97] est de fusionner en premier en une seule ϵ-transition une séquence d'une transition d'empilement suivie d'une transition de dépilement du même symbole, et ensuite, de fusionner ces ϵ-transitions entre elles ou avec des transitions d'empilement et de dépilement pour construire des nouvelles ϵ-transitions. Son entrée est un *PA* avec des actions de pile seulement. Il retourne un ensemble d'ϵ-transitions. Il calcule les ϵ-transitions en enregistrant l'information dans les structures de données C_Direct et C_Trans. Il énumère toutes les paires possibles de localités. Il cherche ensuite si elles peuvent être exploitées afin de construire une nouvelle ϵ-transition.

L'algorithme de calcul de *RA* est divisé en deux étapes : une étape d'initialisation de la ligne 1 à 16, et une étape de calcul de la ligne 17 à 34. Une variable stack est utilisée pour enregistrer les ϵ-transitions qui ont été calculées, mais qui ne sont pas encore exploitées en conjonction avec des autres transitions pour construire d'autres ϵ-transitions. Elle est initialisée de la ligne 2 à la ligne 4 en empilant (l, l) où $l \in L$. Les structures de C_Direct et C_Trans sont initialement vides (lignes 5-7). La structure C_Direct est utilisée pour appliquer les règles R_3. Elle est initialisée de la ligne 8 à 10. La structure C_Trans est utilisée pour appliquer la règle R_4. Elle est initialisée de la ligne 11

à 15.

Dans la deuxième étape (lignes 16-31), l'algorithme traite toutes les transitions de $stack$ et les exploite à l'aide de C_Direct et C_Trans. Pour chaque nouvelle ϵ-transition entre deux localités l et l', cet algorithme vérifie les deux possibilités suivantes pour calculer d'autres ϵ-transitions :

- en utilisant $C\_Direct((l,l'))$ (ligne 22-24) : pour chaque élément (l_1,l_2) dans $C\_Direct((l,l'))$, notre algorithme enregistre le couple (l_1,l_2) dans $stack$ pour ajouter après une nouvelle ϵ-transition entre l_1 et l_2 c'est à dire si on ajoute une nouvelle ϵ-transition entre l et l' et qu'il existe une transition d'empilement entre l_1 et l dans T et une transition de dépilement de même symbole entre l' et l_2, alors une ϵ-transition entre l_1 et l_2 est ajoutée dans Δ^T si elle n'existe pas déjà.
- en utilisant $C\_Trans((l,l'))$ (ligne 23-29) : pour chaque élément (l_1,l_2,l_3,l_4) dans $C\_Trans((l,l'))$: il existe deux cas pour l'ajout de nouvelles ϵ–transitions :

  — $l_1 = l'$: pour chaque ϵ–transition entre l' et l_2, une ϵ-transition est ajoutée entre l et l_2 comme illustré dans la Fig. 5.6(a), par prolongement à gauche.

  — $l_2 = l$: pour chaque ϵ–transition entre l_1 et l, une ϵ-transition est ajoutée entre l_1 et l' comme illustré dans Fig. 5.6(b), par prolongement à droite.

La règle R_2 est appliquée en utilisant les transitions d'empilement de T (ligne 32-34).

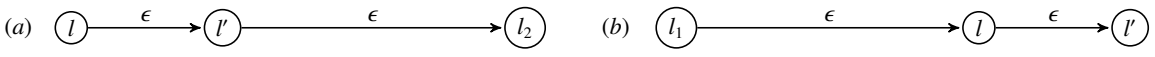
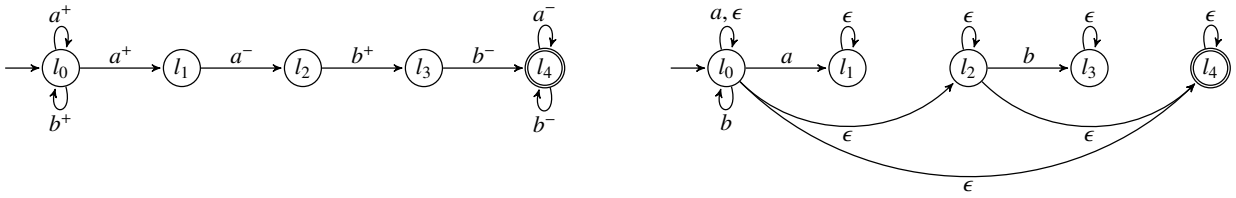


FIGURE 2.15 – Possibilités de calcul d'une nouvelle ϵ-transition

Exemple 2.6.1. La Fig. 2.16.(b) présente le *RA* du *PA* $T = \langle L, l_0, \emptyset, \Gamma, \Delta, F\rangle$ qui est présenté dans la Fig. 2.16.(a) où $\Gamma = \{a,b\}$. Le table 2.2 présente toutes les transitions du *RA* où la première colonne présente la transition et la deuxième colonne indique la règle appliquée pour obtenir cette transition.



(a) (b)

FIGURE 2.16 – (a) un *PA* et (b) son *RA*

Algorithme 2.6.1 Calcul du *RA* [FWW97] d'un *PA* donné

ENTRÉE : Un *PA* $\langle L, l_0, \emptyset, \Gamma, \Delta, F\rangle$
SORTIE : un *RA* $\langle L, l_0, \{\epsilon\} \cup \Gamma, \Delta^R, F\rangle$
VARIABLES : $stack \in$ pile de $L \times L$; $C\_Direct \in L \times L \rightarrowtail$ pile de $L \times L$; $C\_Trans \in L \times L \rightarrowtail$ pile de $L \times L \times L \times L$; $l, l', l_1, l_2, l_3, l_4 \in L$;
DÉBUT
1: $stack \leftarrow \emptyset$, $\Delta^R \leftarrow \emptyset$
2: **pour** chaque localité $l \in L$ **faire**
3: empiler (l, l) dans $stack$ // R_1
4: **fin pour**
5: **pour** chaque couple $(l, l') \in (L \times L)$ **faire**
6: $C\_Direct(l, l') \leftarrow \emptyset$, $C\_Trans(l, l') \leftarrow \emptyset$
7: **fin pour**
8: **pour** chaque couple $(l_1 \xrightarrow{a^+} l_2, l_3 \xrightarrow{a^-} l_4) \in (\Delta \times \Delta)$ où $a \in \Gamma$ **faire**
9: $C\_Direct(l_2, l_3) \leftarrow C\_Direct(l_2, l_3) \cup \{(l_1, l_4)\}$ // R_3
10: **fin pour**
11: **pour** chaque triplet $(l, l', l'') \in (L \times L \times L)$ **faire**
12: **si** $l \neq l'$ **alors**
13: $C\_Trans(l, l') \leftarrow C\_Trans(l, l') \cup \{(l', l'', l, l''))\} \cup \{(l'', l, l'', l'))\}$ // R_4
14: **fin si**
15: **fin pour**
16: **tant que** $stack \neq \emptyset$ **faire**
17: $(l, l') \leftarrow$ dépiler$(stack)$
18: **si** $(l, \epsilon, l') \notin \Delta^R$ **alors**
19: $\Delta^R \leftarrow \Delta^R \cup \{(l, \epsilon, l')\}$
20: **pour** (l_1, l_2) in $C\_Direct((l, l'))$ **faire**
21: empiler (l_1, l_2) dans $stack$
22: **fin pour**
23: **pour** (l_1, l_2, l_3, l_4) in $C\_Trans((l, l'))$ **faire**
24: **si** $(l_1, \epsilon, l_2) \in \Delta^R$ **alors**
25: empiler (l_3, l_4) dans $stack$
26: **sinon**
27: empiler (l_3, l_4) dans $C\_Direct((l_1, l_2))$
28: **fin si**
29: **fin pour**
30: **fin si**
31: **fin tant que**
32: **pour** chaque transition $l_1 \xrightarrow{a^+} l_2$ où $a \in \Gamma$ **faire**
33: $\Delta^R \leftarrow \Delta^R \cup \{(l_1, a, l_2)\}$ // R_2
34: **fin pour**
FIN.

Finkel et al. proposent le théorème 2.6.1 qui montre le lien d'atteignabilité des états entre un *PA* et son *RA*. Ils proposent aussi le théorème 2.6.2 à propos de la complexité de calcul du *RA* d'un *PA* donné.

Théorème 2.6.1. Un état (l, p) est atteignable dans un *PA* si et seulement si la localité l est atteignable dans son *RA* avec le mot p [FWW97].

Théorème 2.6.2. Le *RA* d'un *PA* donné est calculé avec une complexité $O(n^3)$ où n est le nombre de localités du *PA* [FWW97].

Transition de Δ^R	Règle
(l_0, ϵ, l_0)	R_1 car $l_0 \in L$
(l_1, ϵ, l_1)	R_1 car $l_1 \in L$
(l_0, ϵ, l_2)	R_3 car $(l_0, a^+, l_1) \in \Delta$, $(l_1, \epsilon, l_1) \in \Delta^R$ et $(l_1, a^-, l_2) \in \Delta$
(l_2, ϵ, l_2)	R_1 car $l_2 \in L$
(l_3, ϵ, l_3)	R_1 car $l_3 \in L$
(l_2, ϵ, l_4)	R_3 car $(l_2, b^+, l_3) \in \Delta$, $(l_3, \epsilon, l_3) \in \Delta^R$ et $(l_3, b^-, l_4) \in \Delta$
(l_0, ϵ, l_4)	R_4 car $(l_0, \epsilon, l_2) \in \Delta^R$ et $(l_2, \epsilon, l_4) \in \Delta^R$
(l_4, ϵ, l_4)	R_1 car $l_4 \in L$
(l_0, a, l_0)	R_2 car $(l_0, a^+, l_0) \in \Delta$
(l_0, b, l_0)	R_2 car $(l_0, b^+, l_0) \in \Delta$
(l_0, a, l_1)	R_2 car $(l_0, a^+, l_1) \in \Delta$
(l_2, b, l_3)	R_2 car $(l_2, b^+, l_3) \in \Delta$

TABLE 2.2 – Transitions du *RA* presenté dans la Fig. 2.16.(b)

2.7/ CONCLUSION

Nous avons présenté dans ce chapitre quelques notions et définitions des modèles qui sont largement utilisées pour spécifier des systèmes comportementaux et des systèmes temps-réel dans le but de les tester et de les vérifier. Dans le chapitre suivant, nous présentons des méthodes existantes de génération de tests à partir de ces modèles.

3

ÉTAT DE L'ART

Sommaire

Tester est l'une des principales méthodes qui peuvent être utilisées pour s'assurer de la correction d'un logiciel. De ce fait, il y a donc une activité de recherche très importante pour automatiser la génération et l'exécution des tests. L'approche de génération de tests à partir de modèles a pour objectif de vérifier si une implémentation est conforme à une spécification. Dans ce chapitre, nous allons nous intéresser aux méthodes de tests à partir d'*IOLTS*, de *TAIO* et de *PA*. Les *IOLTS* et les *TAIO* sont des systèmes qui interagissent avec leur environnement par des opérateurs d'entrée et sortie, ce qui est souvent le cas par exemple pour des applications critiques. Notons que les automates à pile peuvent être utilisés pour modéliser des programmes récursifs.

Dans la Sec. 3.1, nous présentons quelques notions habituelles dans le domaine du test, test structurel et test fonctionnel, génération de tests à partir de modèles et test de conformité. Nous présentons dans la Sec. 3.2 des méthodes de génération de tests à partir d'un *IOLTS*. Nous montrons quelques méthodes de génération de tests à partir d'un *PA* dans la Sec. 3.3. Enfin, nous présentons dans la Sec. 3.4 des méthodes de génération de tests à partir d'un *TAIO*.

3.1/ TEST STRUCTUREL ET FONCTIONNEL

Les tests peuvent aussi être catégorisés en fonction des différents aspects du système qu'ils visent à exercer. Il existe dans la littérature plusieurs méthodes de test. Parmi celles-ci, on peut distinguer deux grandes familles de test de programme : les méthodes de test structurel qui sont basées sur l'analyse du code source et les méthodes de test fonctionnel qui s'appuient sur la spécification des programmes. Le test de conformité appartient à la famille du test fonctionnel dont l'objectif est de répondre à la question suivante : "Est ce que l'implémentation sous test est conforme à sa spécification ?". Notons qu'une spécification d'un système transformationnel peut-être de type pre-post. Par contre la

spécification d'un système réactif est de type comportemental sous forme d'un système d'actions ou d'un système état-transition.

3.1.1/ NOTION DE TEST

Dans la littérature, on trouve de nombreuses définitions de la notion de test, plus ou moins différentes. Ci-dessous, nous en présentons quelques unes qui sont prises de la thèse [SK06].

- *"Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats obtenus"* - IEEE (Standard Glossary of software Engineering Terminology) [IEE90].
- *"Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts"* [Mye79].
- *"Le processus de test consiste en une collection d'activités qui visent à démontrer la conformité d'un programme par rapport à sa spécification. Ces activités se basent sur une sélection systématique des cas de test et l'exécution des segments et des chemins du programme"* [Was77].
- *"L'objectif du processus de test est limité à la détection d'éventuelles erreurs d'un programme. Tous les efforts de localisation et de correction d'erreurs sont classés comme des tâches de débogage. Ces tâches dépassent l'objectif des tests. Le processus de test est donc une activité de détection d'erreurs, tandis que le débogage est une activité plus difficile consistant en la localisation et la correction des erreurs détectées* [IW78].

Présentons maintenant les deux grandes classes de test [Bei90][Bei95a] : le test structurel et le test fonctionnel.

3.1.2/ TEST STRUCTUREL PAR EXÉCUTION SYMBOLIQUE ET MUTATION

Le test structurel, ou test boite blanche [Bei90], est basé sur une analyse du programme. Il nécessite le code source du programme (ou d'un modèle). Les tests structurels s'intéressent principalement aux structures de contrôle et aux détails procéduraux. Ils s'intéressent également aux dépendances de données représentées par un graphe de flot de données. Ils permettent de vérifier si les aspects intérieurs de l'implantation ne contiennent pas d'erreurs de logique. Ils s'appuient sur l'analyse du code source de l'application pour établir les tests en fonction de critères de couverture du graphe de flot de contrôle (instructions, point de contrôle, chemins) ou du graphe de flot de données ou d'un modèle de fautes.

3.1.2.1/ GÉNÉRATION DE TESTS COUVRANT LE GRAPHE DE FLOT DE CONTRÔLE

Elle se base sur une représentation du programme par un graphe de flot de contrôle. Les nœuds du graphe représentent des blocs d'instructions, les arcs sont orientés et représentent la possibilité de transfert de l'exécution d'un nœud à un autre. La génération de tests se base sur des critères de couverture du graphe. Il existe plusieurs critères de

couverture [ZHM97] comme la couverture de tous-les-nœuds (bloc d'instructions), la couverture de tous-les-arcs, la couverture de tous-les-chemins-indépendants, la couverture de toutes-les-PLCS (Portions Linéaires de Code suivies d'un Saut), la couverture des chemins limites et intérieurs, la couverture de tous-les-k-chemins et la couverture de tous-les-chemins quand c'est possible. Le critère "tous les nœuds" consiste à générer des cas de test qui passent au moins par chaque nœud (bloc d"instructions). Le critère "tous les arcs" vise à sélectionner les cas de test qui couvrent tous les arcs d'enchainement de blocs d'instructions. Le critère "tous les chemins" consiste à générer les cas de test qui couvrent tous les chemins du graphe du flot de contrôle.

3.1.2.2/ GÉNÉRATION DE TESTS COUVRANT LE FLOT DE DONNÉES

La couverture de flot de données [FW88][Her76] se base sur une représentation du programme par un graphe de dépendances de données. Le flot de données est représenté en annotant le graphe de contrôle par des informations sur les utilisations de variables par le programme :

- Def(x) : instruction pour définir x comme par exemple : affectation de x, instruction de lecture de x, etc.
- use(x) : instruction d'utilisation de x comme par exemple : x dans le membre droit d'une affectation, utilisation de x dans une condition, etc.
- paire Def-Use pour x : il existe au moins une trace d'exécution passant par Def puis par Use, tel que x n'est pas redéfini entre Def et Use.
- chemin Def-Use pour x : un chemin passant par une paire Def-Use de x.

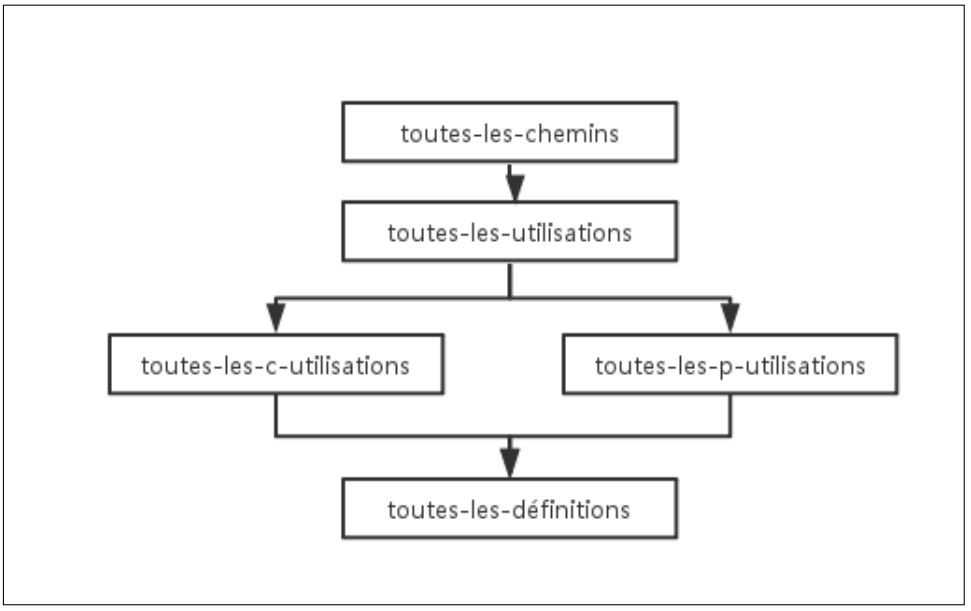


FIGURE 3.1 – Hiérarchie des critères de couverture basés sur le flot de données

La génération de test se base sur des critères de couverture comme "toutes les définitions de variables" et "toutes les utilisations". Elle se base sur des critères de couverture comme :

- toutes-les-définitions : il existe au moins un chemin qui couvre chaque définition dans un test.

- toutes-les-utilisations : les tests obtenus couvrent toutes les utilisations pour chaque définition, et pour chaque utilisation accessible depuis cette définition. Il existe deux types d'utilisation : p-utilisation si une variable est utilisée dans un prédicat d'une instruction de décision (if, while, etc.) et c-utilisation dans les autres cas (par exemple un calcul).
- tous-les-chemins : il consiste à couvrir tous les chemins possibles entre la définition et la référence, en se limitant aux chemins sans cycle.

La Fig. 3.1 présente la hiérarchie des critères basés sur le flot de données. $\boxed{a} \rightarrow \boxed{b}$ signifie que b est inclus dans a.

3.1.2.3/ EXÉCUTION SYMBOLIQUE

L'exécution symbolique est une technique introduite par King [Kin76]. Elle est utilisée pour la génération de tests particuliers pour définir les données engendrant les tests structurels couvrant certains chemins d'exécution. Elle consiste à associer avec un prédicat appelé prédicat de chemin, que doivent respecter des données de test pour suivre un chemin sélectionné. Des valeurs symboliques sont utilisées comme entrées du programme à la place des données concrètes, et des expressions symboliques représentent les valeurs des variables du programme. A chaque chemin exécuté symboliquement est associée une conjonction de contraintes (sans quantificateurs) portant sur les entrées symboliques : cette formule est appelée condition de chemin. La condition de chemin accumule les contraintes du flot de contrôle. Soit π un chemin qui part d'une localité initiale à destination d'une localité finale. L'exécution symbolique consiste à appliquer les étapes suivantes :

1. donner des valeurs symboliques aux variables qui sont des données ;
2. initialiser la condition de chemin à true ;
3. suivre un chemin π nœud par nœud depuis la localité initiale en appliquant ces deux règles selon la nature de nœud :
   - si le nœud est un bloc d'instructions, exécuter les instructions sur les valeurs symboliques ;
   - si le nœud est un bloc de décision avec une condition C, appliquer un des deux cas suivants :
     — ajouter la condition C exprimée sur les valeurs symboliques à la condition de chemin pour générer un chemin où la condition C est satisfaite ;
     — ajouter la négation de la condition C exprimée sur les valeurs symboliques à la condition de chemin pour générer un chemin où la condition C n'est pas satisfaite.

Exemple 3.1.1. L'algorithme 3.1.1 présente un algorithme de calcul du maximum de trois entiers donnés. Les instructions de cet algorithme sont étiquetées de l_0 à l_5.

Algorithme 3.1.1 Algorithme calculant le maximum de trois entiers donnés

int maximum(données : int a, int b, int c)
résultat : max //maximum de a, b, c
l_0 : $max \leftarrow a$
si $l_1 : b > max$ **alors**
$l_2 : max \leftarrow b$
fin si
si $l_3 : c > max$ **alors**
$l_4 : max \leftarrow c$
fin si
l_5 : retourner max
end.

Le tableau suivant présente les quatre chemins d'exécution et le prédicat de chacun de ces chemins où a_0, b_0 et c_0 sont des valeurs symboliques de a, b et c. Ces chemins sont obtenus en construisant l'arbre présenté dans la Fig. 3.2 en faisant des branchements à chaque point de choix, où l_i est le numéro de l'instruction de l'algorithme 3.1.1 et PC est le prédicat.

	Chemins d'exécution	Prédicat de chemins	cas de test
1	l_0,l_1,l_3,l_5	$b_0 \leqslant a_0 \wedge c_0 \leqslant a_0$	$a_0 = 2, b_0 = 0, c_0 = 0$
2	l_0,l_1,l_3,l_4,l_5	$b_0 \leqslant a_0 \wedge c_0 > a_0$	$a_0 = 1, b_0 = 0, c_0 = 2$
3	l_0,l_1,l_2,l_3,l_5	$b_0 > a_0 \wedge c_0 \leqslant b_0$	$a_0 = 1, b_0 = 2, c_0 = 1$
4	l_0,l_1,l_2,l_3,l_4,l_5	$b_0 > a_0 \wedge c_0 > b_0$	$a_0 = 1, b_0 = 2, c_0 = 3$

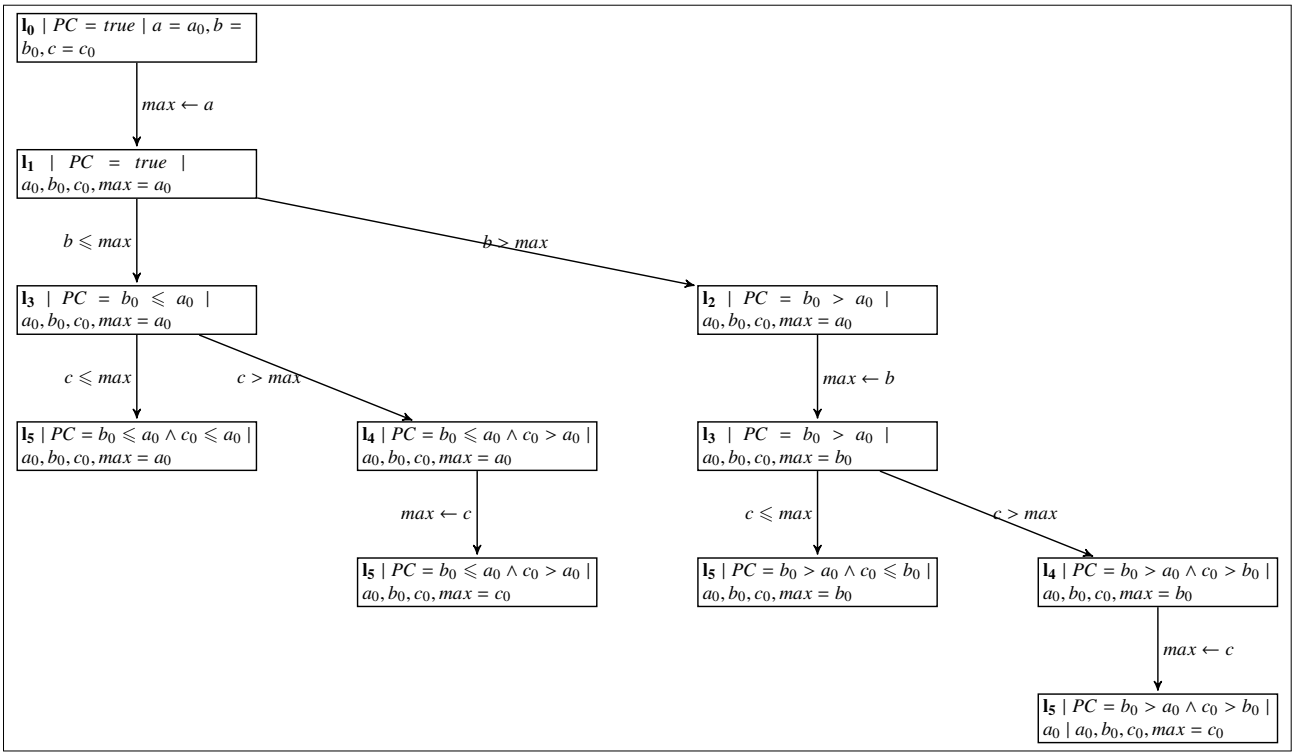


FIGURE 3.2 – Arbre des chemins d'exécution possibles de l'algorithme 3.1.1

Il existe des méthodes de test qui combinent exécution concrète et symbolique. Elles sont appelées méthodes concoliques. Elles ont permis le développement de plusieurs outils de génération de tests pour les programmes écrits en langage C tels que DART [GKS05] PathCrawler [WMMR05], Cute [SMA05].

3.1.2.4/ GÉNÉRATION ET ÉVALUATION PAR MUTATION

Le test par mutation est une technique de test structurel proposé par Millo [DLS78a]. Elle consiste à générer des versions erronées du programme sous test appelé mutant qui est une copie exacte du programme sous test après l'injection d'une erreur. Des opérateurs de mutation décrivent les fautes susceptibles d'être présentes dans une transformation de modèles. Le testeur crée chaque mutant en appliquant ces opérateurs. Le test de mutation est une technique pour l'évaluation et l'amélioration d'une suite de tests [Ham77][DLS78b]. La mutation peut être utilisée dans les cas suivants :

- Génération de tests par mutation : elle consiste à générer des cas de tests permettent de détecter des erreurs injectées. Les cas de tests engendrés à partir d'un mutant représentent des exécutions qui détectent une erreur pour une implémentation qui engendrerait cette exécution. Les auteurs de [AJT15] présentent un processus de génération de tests par mutation à partir d'un modèle donné. Il existe plusieurs méthodes comme celle présentée dans [ALN13] pour tester des automates temporisés, [ABJK11][SHJ11] pour tester des modèles UML et [BHM+10][HRK11] dans le cadre de modèles Simulink. Une publication récente rédigée par Jia and Harman [JH11] montre l'intérêt pour les tests de mutation et souligne les problèmes ouverts de générer des cas de test au moyen d'une analyse de mutation.

- Évaluation de tests par mutation : c'est une technique qui vise à évaluer la qualité d'une suite de tests par rapport aux fautes les plus probables qui ont été envisagées. La qualité d'une suite de tests est évaluée quantitativement par la proportion de mutants qu'il est capable de détecter.

3.1.3/ TEST FONCTIONNEL DE CONFORMITÉ ENGENDRÉ À PARTIR DE MODÈLES

Le test fonctionnel est basé sur la relation entre la spécification et l'interface du programme. Il n'examine pas le programme, mais il examine son comportement observable. On présente dans cette partie le test fonctionnel, le processus de la génération de tests à partir de modèles et le test de conformité.

3.1.3.1/ TEST FONCTIONNEL

Le test fonctionnel (appelé aussi test de boîte noire) [Bei95a] vise à examiner le comportement fonctionnel d'un logiciel et sa conformité avec la spécification du logiciel. Il s'intéresse aux besoins fonctionnels du logiciel. Il comporte deux étapes importantes : l'identification des fonctions que le logiciel est supposé offrir et la création de données de test qui vont servir à vérifier si ces fonctions sont réalisées par le logiciel ou pas.

3.1.3.2/ GÉNÉRATION DE TESTS À PARTIR DE MODÈLES

Le Model Based Testing (*MBT*) [LY96][Bei95b][UL07] ou "Test à partir de modèles" est une approche de génération de tests fonctionnels à partir de modèles qui est une description du système appelée spécification formelle. Elle utilise un solveur de contraintes pour générer des cas de test fonctionnels couvrant les comportements modélisés du système sous test. Le processus de MBT est présenté dans la Fig. 3.3. Ses principales phases sont les suivantes :

1. développement pour produire une implémentation appelée système sous test *SUT* (pour System Under Test) à partir d'un cahier des charges ;
2. conception d'un modèle abstrait dédié au test d'un système donné : concevoir un modèle à partir duquel seront produits les cas de test à exécuter sur le système sous test ;
3. génération de tests pilotée par les modèles et une stratégie de test ;
4. concrétisation pour établir un lien entre les éléments du modèle et les éléments concrets du système. C'est une étape encore largement manuelle et qui nécessite l'expertise de l'ingénieur pour passer d'une suite de tests abstraits en suite de tests exécutables ;
5. exécution des tests sur le système sous test et constitution de leur verdict en comparant les résultats obtenus sur le *SUT* aux résultats attendus par les tests. L'oracle est défini par le modèle ;

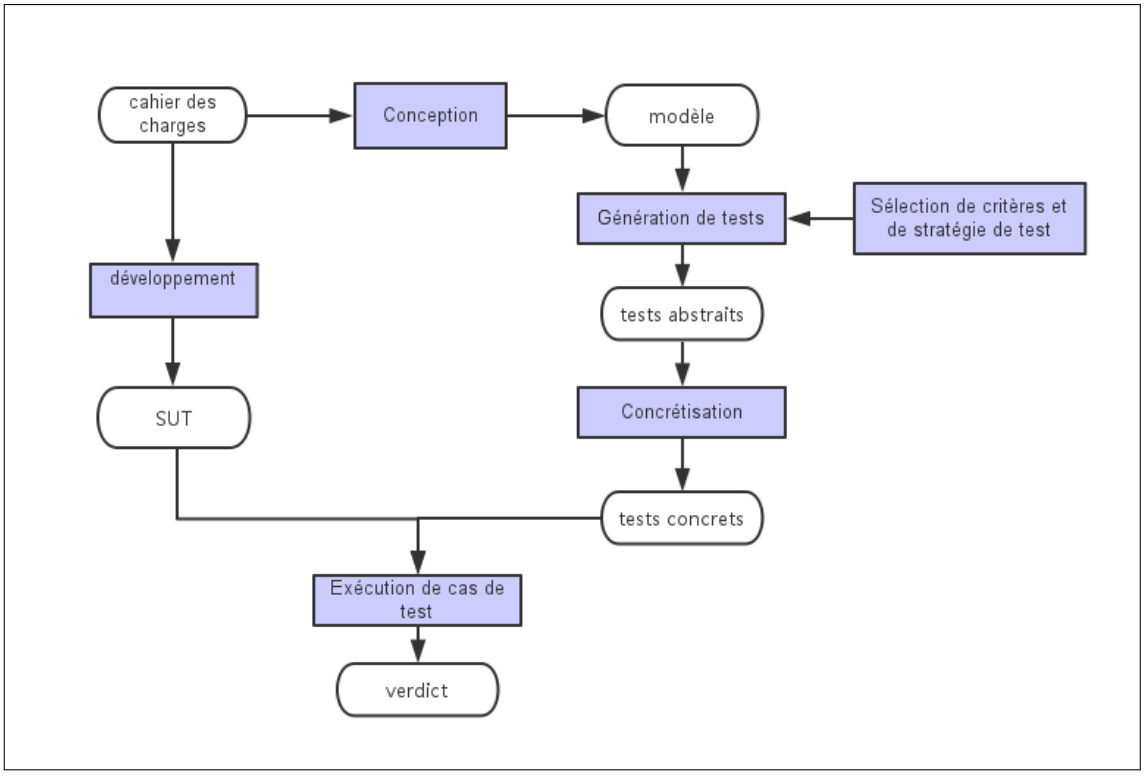


FIGURE 3.3 – Processus du model-based testing

Il existe des outils *MBT* comme CertifyIt[1], Matelo[2], etc.

3.1.3.3/ TEST DE CONFORMITÉ

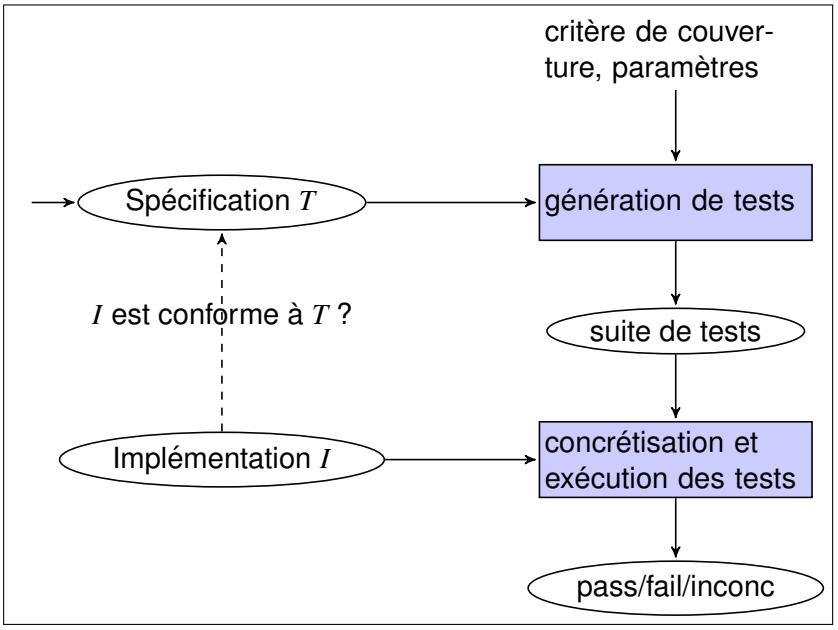


FIGURE 3.4 – Processus de test de conformité

Le test de conformité [Tre93][RAY87] est un test fonctionnel qui consiste à vérifier qu'une implémentation est conforme à sa spécification. La spécification est une description formelle des comportements attendus du système. L'implémentation à tester n'est connue que par ce qui est observable à partir de son interface et la conformité est définie entre une spécification et ce qui est observable sur le système sous test. Le test de conformité permet donc de déterminer la conformité d'un composant ou d'un système à vis à vis de ses exigences qui ont été spécifiées dans le modèle. Il vise à vérifier que l'implémentation remplit bien les fonctions attendues, et c'est pourquoi on le dit test fonctionnel. Dans cette section, nous présentons le processus de test de conformité qui est présenté dans la Fig. 3.4. Les étapes sont les suivantes :

- Génération de tests à partir d'une spécification : c'est une activité pilotée par le testeur dont le but est de produire des cas de test à partir de la spécification selon des critères de couverture sélectionnés. Ces cas de test observent une implémentation relativement à une relation de conformité donnée.
- Concrétisation puis exécution des tests : c'est un processus qui stimule l'implémentation sous test par un cas de test. L'implémentation réagit à ces stimulus et délivre des réponses qui vont être observées et utilisées pour annoncer le verdict de test selon une relation de conformité. Il existe trois types de verdict :
  - **pass** si l'application des séquences de test permet d'observer sur l'implémentation des réactions conformes à celles de la spécification. Donc, il n'y a pas d'erreur détectée et l'on dit que le test a réussi.

1. http://www.smartesting.com/fr/certifyit/
2. http://www.all4tec.net/MaTeLo/homematelo.html

- **inconc** si l'application des séquences de test ne permet pas de conclure sur la conformité de l'implémentation et qu'il n'y a pas d'erreur détectée, mais que l'objectif n'est plus atteignable.
- **fail** si l'exécution montre la présence d'erreurs dans l'implémentation car on observe une réponse qui n'est pas la réponse attendue. On dit qu'une erreur de non conformité est détectée et l'implémentation n'est pas conforme à sa spécification.

Nous avons introduit dans ce chapitre des notions liées au test d'une façon générale. Et nous avons détaillé les notions de test structurels et de test fonctionnels. Dans les sections suivantes, on présente des méthodes de génération de tests existantes à partir des modèles comme les *IOLTS*, les *PA* et les *TAIO*.

3.2/ GÉNÉRATION DE TESTS À PARTIR D'*IOLTS*

Dans cette partie, nous présentons une méthode de génération de tests basée sur les systèmes de transitions à entrées/sorties.

3.2.1/ RELATION DE CONFORMITÉ *ioco*

Dans cette partie, on définit la relation de conformité *ioco* de Tretmans [Tre99] pour les automates non temporisés avec entrées/sorties. Pour présenter la relation de conformité *ioco* pour un *IOLTS* $T = \langle S, s_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Delta \rangle$, on a besoin de définir les notations suivantes :

- s after $\rho = \{s' \mid s \rightarrow^{\rho}\}$ est l'ensemble d'états de T atteignable par la trace ρ.
- $STraces(T) = \{\rho \in (\Sigma_{in} \cup \Sigma_{out} \cup \{\tau\})^* \mid s_0 \rightarrow^{\rho}\}$ est l'ensemble de traces observables pour T.
- $out(s) = \{a \in \Sigma_{out} \mid s \rightarrow^{a}\}$ est l'ensemble des actions de sortie qui peuvent être observées à partir de l'état s.
- $out(S) = \bigcup_{s \in S} out(s)$ est l'ensemble des actions de sortie qui peuvent être observées à partir d'un ensemble d'états S.

Définition 13 : *ioco* [Tre99] - Relation de conformité *ioco*

Soit $T = \langle S, s_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Delta \rangle$ une spécification et $I = \langle S^I, s_0^I, \Sigma_{in}^I \cup \Sigma_{out}^I \cup \{\tau^I\}, \Delta^I \rangle$ une implémentation de T. Formellement, I est conforme à T, noté I *ioco* T, ssi $\forall \rho.(\rho \in STraces(T) \implies out(I \textit{ after } \rho) \subseteq out(T \textit{ after } \rho))$.

Elle signifie que l'implémentation I est conforme à la spécification T si et seulement si après toute trace de T, chaque action de sortie de I est spécifiée par T.

Exemple 3.2.1. La Fig. 3.5.(a) présente un exemple d'une spécification T où $\Sigma_{in} = \{a\}$ et $\Sigma_{out} = \{b, c\}$. Les figures 3.5.(b), 3.5.(c) et 3.5.(d) présentent des exemples d'implémentations appelées I_1, I_2 et I_3. L'implémentation I_1 est conforme à T. L'implémentation I_2 n'est pas conforme à T car $out(I_2 \; after \; \epsilon) = \{b\} \not\subseteq out(T \text{ after } \epsilon) = \{\}$. L'implémentation I_3 n'est pas conforme à T car $out(I_3 \; after \; ?a) = \{d\} \not\subseteq out(T \; after \; ?a) = \{b, c\}$.

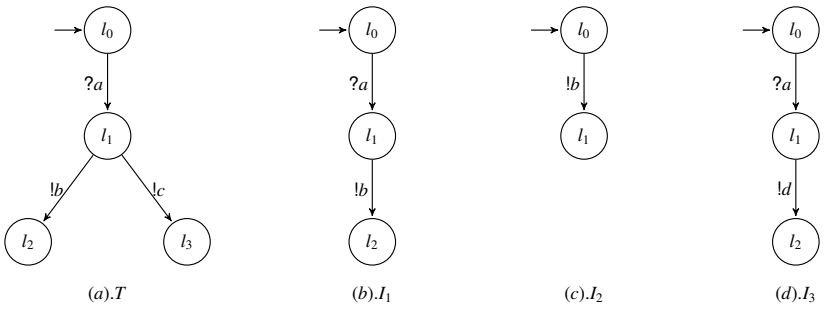


FIGURE 3.5 – Exemple d'une spécification et de trois implémentations

3.2.2/ GÉNÉRATION DE TEST À PARTIR D'UN *IOLTS*

Algorithme 3.2.1 Génération de cas de tests à partir d'un *IOLTS* donné [Tre99]

ENTRÉES : $T = \langle S, s_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Delta \rangle$
SORTIES : $TC = \langle S^{TC}, s_0, \Sigma_{in}^{TC} \cup \Sigma_{out}^{TC}, \Delta^{TC} \rangle$// cas de test
VARIABLES : $s, s' \in S$, $a \in \Sigma_{in} \cup \Sigma_{out}$, i : entier
DÉBUT

$s \leftarrow s_0$
tant que true **faire**
  pour chaque état puits s de TC différent de $pass$ et $fail$ **faire**
    $i \leftarrow pick(\{0,1,2\})$ //$pick(X)$ choisit aléatoirement un élément de X.
    si $i = 0$ **alors**
      $a \leftarrow pick(\Sigma_{in})$
      $s' \leftarrow succ(s, a)$
      ajouter $s \xrightarrow{a} s'$ dans TC
    sinon
      si $i = 1$ **alors**
        pour chaque $a \in \Sigma_{out}$ **faire**
          $s' \leftarrow s\ after\ a$
          si $s' \neq \emptyset$ **alors**
            ajouter $s \xrightarrow{a} s'$ dans TC
          sinon
            ajouter $s \xrightarrow{a} fail$ dans TC
          fin si
        fin pour
      sinon
        changer l'état s avec $pass$ dans TC
      fin si
    fin si
  fin pour
fin tant que

FIN.

Nous présentons dans cette partie un algorithme de génération de tests pour les systèmes de transitions à entrées/sorties basés sur la relation de conformité *ioco*. Un

cas de test $TC = \langle S^{TC}, s_0, \Sigma_{in}^{TC} \cup \Sigma_{out}^{TC} \cup \{\tau^{TC}\}, \Delta^{TC} \rangle$ d'un *IOLTS* $T = \langle S, s_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Delta \rangle$ est un *IOLTS* tel que $S^{TC} \subseteq S \cup \{fail, pass\}$ et $\Sigma_{out}^{TC} = \Sigma_{in}$ et $\Sigma_{in}^{TC} = \Sigma_{out}$. Ainsi, le test émet des entrées à destination de la spécification et reçoit des sorties de celle-ci. Un algorithme de génération de tests aléatoire à partir d'un *IOLTS* est proposé dans [Tre99]. Cet algorithme est présenté dans l'algorithme 3.2.1. A chaque itération de l'algorithme, le cas de test est augmenté par l'ajout des nouvelles transitions dans TC. À chaque étape, l'algorithme propose de choisir aléatoirement entre les trois possibilités suivantes :

- arrêter en émettant le verdict $pass$ (ligne 20),
- émettre une entrée a si c'est possible (ligne 5-8),
- recevoir les sorties et ajouter une transition qui mène à un verdict $fail$ en cas de sortie non autorisée (ligne 10-18).

Une transition reliant deux nœuds est étiquetée par une action (d'entrée ou de sortie). L'algorithme peut décider de stopper un chemin dans l'arbre en ajoutant un verdict $pass$.

3.3/ GÉNÉRATION DE TESTS À PARTIR DE *PA*

Un programme peut être modélisé par un automate fini. La génération de tests à partir de cette abstraction a un risque d'avoir de nombreux tests abstraits qui ne sont pas concrétisables c'est à dire qu'il n'existe pas les éléments de l'abstraction et le programme. Les automates à pile peuvent être des abstractions de programmes récursifs. C'est une abstraction qui peut être précise qu'une abstraction d'un programme par un automate fini. Il est possible d'obtenir moins de tests non concrétisables à partir de cette abstraction car un cas de test d'un *PA* dépend des appels de fonctions récursives.

On présente dans cette partie des méthodes différentes de génération de tests à partir d'un *PA*. Les critères de couvertures d'un *PA* sont toutes les localités, toutes les transitions, etc. Les auteurs de [HM15] proposent une méthode de génération de tests à partir d'un automate à pile selon un nouveau critère qui est basé à la fois sur les localités et sur les configurations de la pile.

3.3.1/ MODÉLISATION D'UN PROGRAMME RÉCURSIF PAR UN *PA*

Un programme récursif peut être modélisé par un automate à pile. L'abstraction d'un programme récursif par un automate à pile est expliquée dans [DHKM14b]. Elle consiste à modéliser en premier le programme récursif par un graphe de flot de contrôle qui ne tient pas compte des appels récursifs. Puis, on calcule un *PA* à partir du graphe de flot de contrôle. Le *PA* a les mêmes transitions que le graphe de flot de contrôle sauf les transitions qui font des appels récursifs. Les transformations sont les suivants : soit tr une transition d'appel récursif, on ajoute au *PA* une transition d'empilement de la localité source de tr vers la localité initiale du *PA*, puis une transition de dépilement du même symbole de chaque localité finale du *PA* vers la localité cible de tr. Ensuite, on retire tr du *PA*.

Exemple 3.3.1. La suite de Fibonacci est la suite d'entier $(U_n)_{n \geqslant 0}$ définie récursivement

par :

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \text{ pour tout } n \geqslant 2 \end{cases} \tag{3.1}$$

On peut exprimer directement cette définition par l'algorithme 3.3.1 de calcul récursif. Les instructions de cet algorithme sont étiquetées de l_0 à l_6. La Fig. 3.6 décrit le graphe de flot de contrôle de ce programme. La Fig. 3.7 présente un *PA* qui est une abstraction du programme 3.3.1 en utilisant le graphe de flot de contrôle de la Fig. 3.6 et en y effectuant les transformations décrites ci-dessus. Le *PA* contient les mêmes transitions que celles du graphe de flot de contrôle sauf les transitions $(l_3, res_1 = Fib(n-1), l_4)$ et $(l_4, res_2 = Fib(n-2), l_4)$. Pour la transition $(l_3, res_1 = Fib(n-1), l_4)$, deux transitions ont été ajoutées au *PA* car il existe une seule localité finale dans le graphe de flot de contrôle. Soit F^+ et F^- respectivement les actions d'empilement et de dépilement du symbole F. Les deux transitions sont (l_3, Fib_1^+, l_0) qui est l'appel récursif de la fonction $Fib(n-1)$ et la transition (l_6, Fib_1^-, l_4) qui est le retour d'appel $Fib(n-1)$. Pour la transition $(l_4, res_2 = Fib(n-2), l_4)$, les transitions (l_4, Fib_2^+, l_0) qui est l'appel récursif de la fonction $Fib(n-2)$ et la transition (l_6, Fib_2^-, l_5) qui est le retour d'appel $Fib(n-2)$ sont ajoutées au *PA*. Ainsi, $\Gamma = \{Fib_1, Fib_2\}$ et $\Sigma = \{int\ res_1, n \leqslant 1, n > 1\ return\ n, return\ res_1 + res_2\ \}$.

Algorithme 3.3.1 Algorithme récursif calculant le n^{eme} nombre de la suite de Fibonacci $Fib(n)$

```
int  Fib(int n)
    l0 : int res1, res2 ;
    si  l1 : n ⩽ 1 alors
      l2 : return n ;
    sinon
      l3 : res1 = Fib(n − 1) ; //Fib1+
      l4 : res2 = Fib(n − 2) ; //Fib2+
      l5 : return res1 + res2 ;
    fi
l6 : end.
```

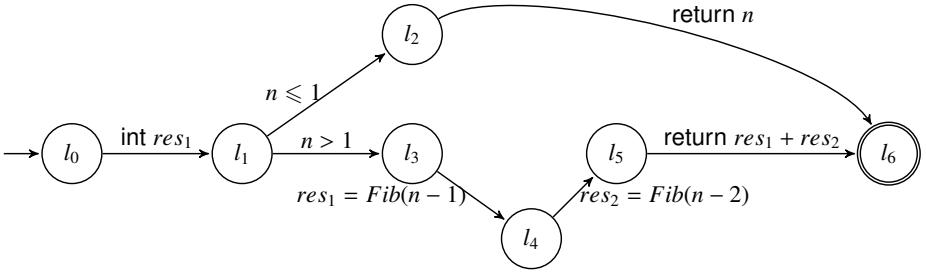


FIGURE 3.6 – Graphe de flot de contrôle modélisant le programme 3.3.1

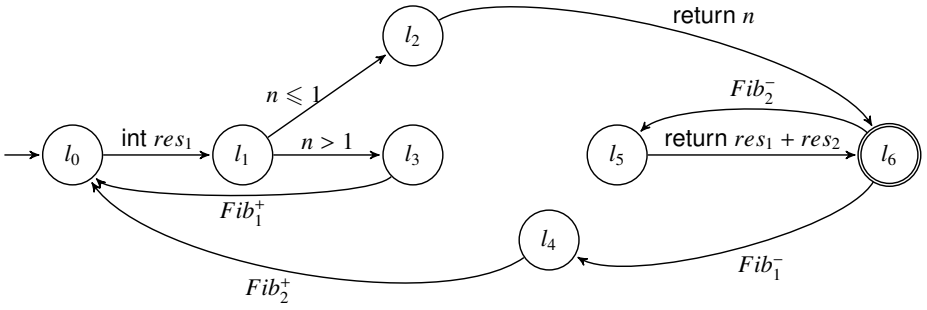


FIGURE 3.7 – *PA* modélisant le programme récursif 3.3.1

3.3.2/ GÉNÉRATION DE TESTS À PARTIR D'UN AUTOMATE À PILE

Il existe dans la littérature plusieurs méthodes de génération de tests à partir d'un *PA*. Nous en présentons quelques unes ici.

3.3.2.1/ GÉNÉRATION DE TESTS À PARTIR DE GRAMMAIRES

Les grammaires algébriques sont des structures fondamentales en informatique. Elles sont un modèle très utilisé, notamment dans les compilateurs. Il existe plusieurs algorithmes classiques qui permettent la transformation d'un automate à pile en une grammaire algébrique (par exemple [Sip96]). Produire des mots à partir d'une grammaire, selon différents critères de couverture ou d'autres contraintes, est nécessaire dans de nombreux domaines tels que l'analyse/compilation, la génération des cas de test, la bioinformatique, etc. Les auteurs de [XZC11] proposent un outil de génération de tests à partir d'une grammaire donnée. La génération de tests est basée sur des algorithmes et des techniques de couverture de règles, telle que celle qui est définie dans [Pur72][ZW09][AV08].

3.3.2.2/ GÉNÉRATION ALÉATOIRE-UNIFORME D'ARBRES DE DÉRIVATION

Les auteurs de [FZC94][FS09] proposent des techniques combinatoires pour générer aléatoirement et uniformément des arbres d'exécution à partir de grammaires algébriques. Plusieurs outils existants comme GenRgenS [PTD06] ou le module CS de Mupad [DDZ98] peuvent être utilisés à cet effet.

3.3.2.3/ MÉTHODE DE TEST ALÉATOIRE D'UN PROGRAMME RÉCURSIF

Une technique qui permet de combiner la génération aléatoire avec un critère de couverture, à partir d'une grammaire algébrique ou d'un automate à pile est définie dans [DHK13][DHKM14a]. Cette approche de génération des tests abstraits consiste à :

1. générer le graphe de flot de contrôle du programme récursif,
2. transformer le graphe de flot de contrôle en un automate à pile normalisé,

3. générer une grammaire algébrique à partir de l'automate à pile,
4. générer aléatoirement et uniformément des arbres de dérivation de la grammaire algébrique,
5. traduire ces arbres de dérivation en chemins dans le programme,
6. calculer des valeurs d'entrée du programme récursif, afin que les exécutions correspondent aux chemins générés.

3.4/ GÉNÉRATION DE TESTS À PARTIR DE *TAIO*

Il existe de nombreux travaux sur la vérification des automates temporisés [ACH94][DOY94][DY95][BER94]. Certains outils [DOTY96][BLL+98] ont été développés à cet effet. Mais d'autres différentes méthodes de génération de tests à partir des systèmes temporisés ont été proposées. Nous présentons dans cette partie des méthodes formelles de test des systèmes temps-réel basées sur des *TAIO*.

3.4.1/ TEST DE CONFORMITÉ À PARTIR D'UN *TAIO*

Plusieurs relations de conformité ont été introduites dans la littérature pour les systèmes temporisés. On trouve par exemple comme bisimulation temporisée [SVD01], relation d'inclusion de traces [HLM+08a][KJM04], relation de conformité relativisée *rtioco* [LMN05], relation de conformité $\sqsubseteq_{tioco}$ [BB04], relation de conformité *tioco* [KT09], etc. Plusieurs méthodes de test de conformité à partir d'un *TAIO* sont proposées [KT09][Fou02].

Dans cette partie, nous allons décrire les éléments permettant de définir la relation *tioco* pour des *TAIO*. Le principe est similaire à *ioco* en considérant l'écoulement du temps comme une sortie. Il existe deux types de tests considérés pour le test de conformité qui sont : les tests analogiques pour lequel le temps est dense et est mesuré par une horloge exacte, et les tests digitaux [HMP92] où le temps est mesuré par une horloge périodique.

3.4.1.1/ RELATION DE CONFORMITÉ *tioco*

Dans cette partie, on définit la relation de conformité *tioco* [KT09] pour les *TAIO*. Cette relation est une extension de la relation de conformité *ioco* de Tretmans [Tre99] pour les automates non temporisés avec entrées/sorties. *tioco* est une adaptation pour les systèmes temporisés de *ioco* qui considère l'écoulement du temps comme une sortie observable. La principale différence est que *ioco* prend en compte la notion de quiescence, contrairement à *tioco* [KT09] où les délais sont explicitement spécifiés. Les hypothèses sont que la spécification est un *TAIO* non bloquant et l'implémentation est un *TAIO* non bloquant et complet en entrée. Pour présenter la relation de conformité *tioco* pour un *TAIO* $T = \langle L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, X, \Delta, F \rangle$, on a besoin de définir les notations suivantes : soit $\rho \in RT(\Sigma_{in} \cup \Sigma_{out})$ une trace d'exécution :

- T *after* $\rho = \{s \in S^T \mid \exists \rho'.(\rho' \in RT(\Sigma_\tau) \wedge s_0^T \rightarrow^{\rho'} s \wedge P_\Sigma(\rho') = \rho)\}$ est l'ensemble d'états de T atteignables par une trace ρ' dont la projection $P_\Sigma(\rho')$ sur les actions observables et contrôlables est ρ.

- *ObsTTraces*$(T) = \{P_\Sigma(\rho) \mid \rho \in RT(\Sigma_\tau) \wedge s_0^T \rightarrow^\rho\}$ est l'ensemble des traces temporisées observables pour un *TAIO* T.
- *elapse*$(s) = \{t \mid t > 0 \wedge \exists \rho.(\rho \in RT(\{\tau\}) \wedge Time(\rho) = t \wedge s \rightarrow^\rho)\}$ est l'ensemble des délais qui peuvent s'écouler à partir de l'état s sans qu'une action ne soit observée.
- *out*$(s) = \{a \in \Sigma_{out} \mid s \rightarrow^a\} \cup elapse(s)$ est l'ensemble des actions de sortie ou délais qui peuvent être observés à partir de l'état s.

Définition 14 : *tioco* [KT09] - Relation de conformité *tioco*

Soit $T = \langle L, l_0, \Sigma_\tau, X, \Delta, F\rangle$ une spécification et $I = \langle L^I, l_0^I, \Sigma_\tau^I, X^I, \Delta^I, F^I\rangle$ une implémentation de T. Formellement, I est conforme à T noté I *tioco* T ssi $\forall \rho.(\rho \in ObsTTraces(T) \implies out(I \textit{ after } \rho) \subseteq out(T \textit{ after } \rho))$.

Elle signifie que l'implémentation I est conforme à la spécification T si et seulement si après toute trace de T, chaque action de sortie ou délai de I est spécifié par T.

Exemple 3.4.1. La Fig. 3.8.(a) présente un exemple d'une spécification T où $\Sigma_{in} = \{A\}$ et $\Sigma_{out} = \{B\}$. La spécification T est non bloquante. Les Figures 3.8.(b), 3.8.(c), 3.8.(d) et 3.8.(e) présentent des exemples d'implémentations appelées I_1, I_2, I_3 et I_4. Ces implémentations sont non-bloquantes. On suppose que I_1, I_2, I_3 et I_4 sont complètes en entrées. La spécification T doit envoyer B au plus tard 6 unités de temps et au plus tôt 2 unités de temps après la réception de A. L'implémentation I_1 est conforme à T parce qu'elle produit B entre 3 et 5 unités de temps après avoir reçu A. L'implémentation I_2 n'est pas conforme à T parce que, par exemple, $out(I_2\ after\ ?A\ ) = \mathbb{R}^+ \not\subseteq out(T\ after\ ?A\ ) = \{B\} \cup [0,6]$. L'implémentation I_3 n'est pas conforme à T parce que, par exemple, $out(I_3\ after\ ?A\ ) = \{C\} \cup [0,5] \not\subseteq out(T\ after\ ?A) = \{B\} \cup [0,6]$. L'implémentation I_4 est conforme à T bien que l'action $?D$ ne soit pas spécifiée dans T : la relation ne porte que sur les sorties et les délais, c'est à dire ce qui est observable de l'extérieur. Comme la trace $?D$ n'appartient pas à $obsTTraces(T)$, I_4 est conforme.

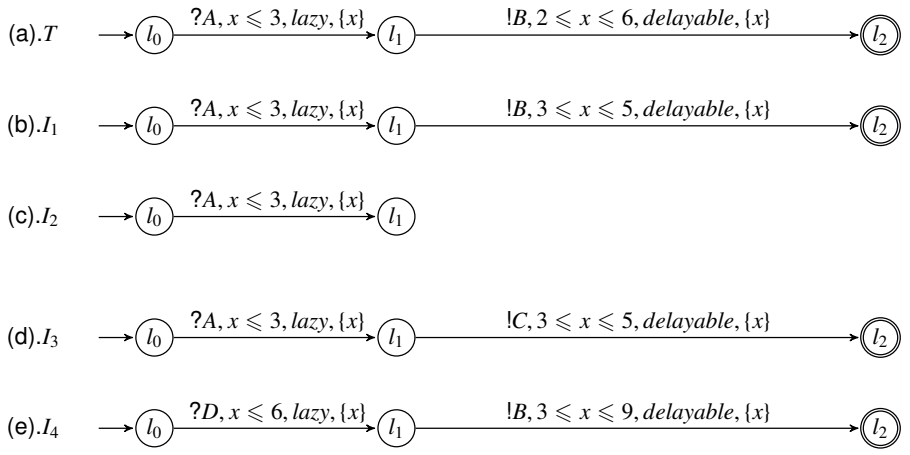


FIGURE 3.8 – Exemple d'une spécification et de quatre implementations

3.4.1.2/ GÉNÉRATION DE TESTS ANALOGIQUES

Un test à horloge analogique mesure exactement le délai entre deux actions et émet une action à n'importe quel point dans le temps. On considère que le testeur est muni d'une

horloge, et qu'il peut mesurer les instants de façon exacte. Avant de présenter l'algorithme de génération de tests analogiques qui proposé dans [KT09], on a besoin de définir les notations suivante : $tsucc(S,t) = \{(l,v+t) \mid \exists(l,v).((l,v) \in S \wedge \exists\rho.(\rho \in RT(\{\tau\}) \wedge Time(\rho) = t \wedge (l,v) \rightarrow^t (l,v+t)\}$ qui est l'ensemble des états atteignables à partir d'un ensemble de l'un des états de S par l'écoulement de temps d'un délai t. Cet algorithme est inspiré de l'algorithme de [Tre96] pour la génération de tests à partir d'un *IOLTS* donné. Il est présenté dans l'algorithme 3.4.1. En partant de l'ensemble d'états $S = tsucc(\{(l_0,0)\},0)$, le testeur applique l'une des possibilités suivantes :

- attendre un délai t : s'il reçoit une action b (ligne 4), alors, $S \leftarrow dsucc(tsucc(S,t),b)$ (ligne 6). Sinon, $S \leftarrow tsucc(S,t)$ si aucune action n'est reçue après un délai t (ligne 8).
- émettre une action d'entrée : la fonction $valid\_entrees(S)$ choisit aléatoirement une entrée possible de l'état S (ligne 17-20).
- produire le verdict $fail$ et arrêter le test.

Le testeur peut arrêter le test à tout moment et déclarer le verdict $pass$.

Algorithme 3.4.1 Génération à la volée de tests analogiques à partir d'un *TAIO* [KT09]

ENTRÉE : un *TA* $T = \langle L, l_0, \Sigma, X, \Delta, F\rangle$
VARIABLES : $a,b \in \Sigma$, S : un ensemble de $L \times (X \rightarrow \mathbb{R}^+)$, i : entier, t : réel ,
DÉBUT

1: $S \leftarrow tsucc(\{(l_0,0)\},0)$
2: **tant que** *vrai* **faire**
3: $t \leftarrow 0$ // t est une horloge pour mesurer un délai
4: attendre la réception d'une action de sortie b
5: **si** b est reçue à l'instant t **alors**
6: $S \leftarrow dsucc(tsucc(S,t),b)$
7: **sinon**
8: $S \leftarrow tsucc(S,t)$ // il n'y a pas d'action reçue à l'instant t
9: **fin si**
10: **si** $S = \emptyset$ **alors**
11: annoncer $fail$
12: arrêter
13: **fin si**
14: **si** $valid\_entrees(S) \neq \emptyset$ **alors**
15: $i \leftarrow pick(\{0,1\})$// 0 pour envoyer une entrée et 1 pour continuer l'observation
16: **fin si**
17: **si** $i = 0$ **alors**
18: $a \leftarrow valid\_entrees(S)$// choisir aléatoirement une entrée possible de S
19: $S \leftarrow dsucc(S,a)$
20: **fin si**
21: **fin tant que**
FIN.

Exemple 3.4.2. La Fig. 3.9.(b) présente un exemple de test analogique à partir du *TAIO* présenté dans la Fig. 3.9.(a). Le verdict $fail$ est annoncé si le testeur observe b avant deux ou après cinq unités de temps après l'émission de a. Il est annoncé aussi en cas d'observation d'une action différente de b. L'action $?othw$ exprime l'observation de n'importe quelle action différente de b ou l'observation de l'action b dans un délai qui ne

satisfait pas la garde $2 \leq x \leq 5$.

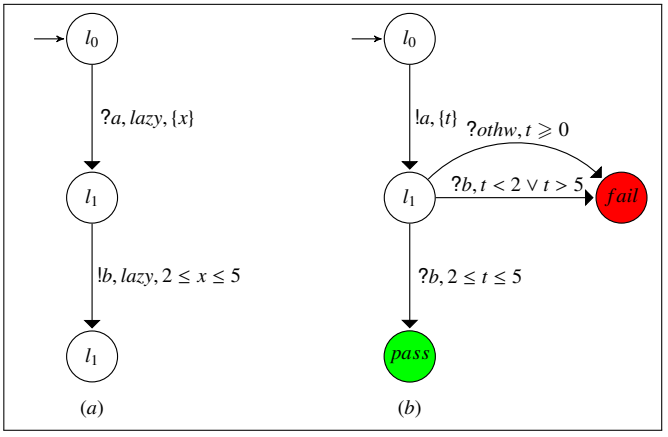


FIGURE 3.9 – Exemple de test analogique et de son *TAIO*

3.4.1.3/ TEST DIGITAL

Un test à horloge digitale mesure seulement le nombre de ”tics” d'une horloge périodique enregistré entre deux actions. Il émet une action immédiatement après avoir observé une action ou un ”tic”. le testeur utilise une horloge périodique sous forme d'automate pour mesurer le temps.

Exemple 3.4.3. La Fig. 3.10 représente un exemple de cas de test digital pour la spécification qui est présentée dans la Fig. 3.9.(a). Le verdict $fail$ est annoncé si le testeur observe b avant deux ou après cinq tics après l'émission de a.

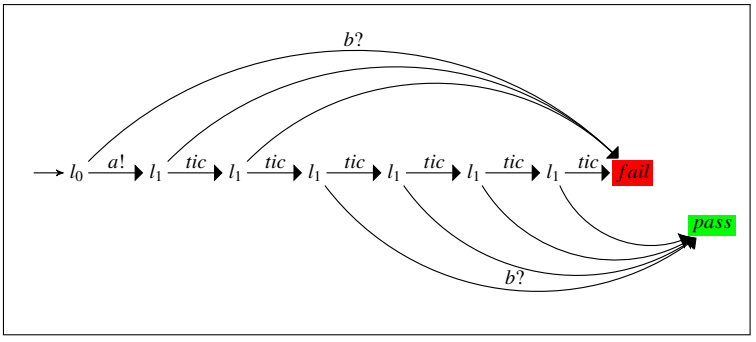


FIGURE 3.10 – Exemple de test digital du *TAIO* presenté dans la Fig. 3.9.(a)

3.4.2/ TEST PAR MODEL CHECKING

Un model checker est un outil automatique pour la vérification formelle de propriétés des systèmes. On trouve par exemple KRONOS [DOTY96] et UPPAAL [LPY97] pour les systèmes temps-réel. KRONOS vérifie si un automate temporisé satisfait ou non une formule décrite contenant des propriétés comportementales et temporelles. L'algorithme de model checking est basé sur une représentation symbolique de l'espace infini d'états

par des ensembles linéaires de contraintes. Un guide complet de son utilisation est décrit dans UPPAAL [LPY97] qui implémente un model checking symbolique qui permet de construire le graphe des zones qui est une représentation abstraite du graphe des régions (voir la Sec. 2.3.2).

Il existe une méthode formelle pour la génération des cas de test en utilisant la technologie du model checking UPPAAL [HLN+03][HLM+08b]. Le model checking prend en entrée le modèle de la spécification du système temps-réel et une propriété d'atteignabilité. Il produit en sortie un une séquence d'exécution si cette propriété n'est pas vérifiée. Il est facile par la suite de transformer cette séquence d'exécution en des cas de test.

3.5/ CONCLUSION

Nous avons introduit dans ce chapitre la notion de test ainsi que différents types de test. Nous avons présentés différentes méthodes de génération de tests à partir d'*IOLTS*, de *PA* et de *TAIO*. Nous présentons dans le chapitre suivant notre modèle qui est un automate à pile temporisé avec entrées/sorties. Puis, nous présentons les contributions de cette thèse qui sont des méthodes de génération de tests à partir de ce modèle.

II

CONTRIBUTIONS

4

AUTOMATE À PILE TEMPORISÉ AVEC ENTRÉES/SORTIES

Sommaire

Un automate temporisé permet de modéliser un système temps réel. Il est obtenu en munissant un automate fini d'une ou plusieurs horloges continues avançant toutes à la même vitesse. Un automate à pile est une généralisation d'un automate fini. Il dispose en plus d'une mémoire infinie organisée en pile. Il peut stocker des éléments et y accéder uniquement au sommet de la pile. On s'intéresse dans nos travaux à un modèle qui combine automate temporisé et automate à pile. Un automate à pile peut modéliser un programme récursif. Les travaux décrits dans [BER95][Dan03][Dan01] [EM06] proposent un modèle d'automate à pile temporisé avec seulement des horloges globales et tel que les symboles de la pile ne sont pas datés par des horloges. Plus récemment, Abdulla et al. [AAS12] proposent une extension des automates à pile temporisés où les éléments de la pile sont datés par des horloges qui s'incrémentent comme les autres horloges. De plus, les opérations sur la pile ont un intervalle de temps comme argument en plus du symbole empilé ou dépilé. Les auteurs de [CL15] prouvent que le modèle de Abdulla et al. [AAS12] est expressivement équivalent à un automate à pile temporisé sans pile datée. Les auteurs de [TW10] introduisent des automates temporisés récursifs qui considèrent que les horloges sont des variables. Un automate récursif temporisé permet de passer les valeurs d'horloges en utilisant soit le passage par valeur, soit le passage par variable. Mais, cette fonctionnalité n'est pas prise en charge dans le modèle des automates à pile temporisés.

On considère dans nos travaux des systèmes modélisés par des automates à pile temporisés avec entrées/sorties (*TPAIO*). Ce modèle est muni d'une pile non datée et non bornée et de variables à valeurs réelles appelées horloges. Il peut modéliser des systèmes temps-réel récursifs. La génération de tests à partir d'un *TPAIO* pourrait s'appliquer à des cas industriels tels que celui décrit décrit dans [CJL+09] qui définit une synthèse automatique de contrôleurs robustes et optimaux. Ces types de contrôleurs opèrent sur des variables continues qui varient dans le temps. Par exemple, l'augmentation de la pression varie dans le temps. Comme présenté dans [TW10], ces systèmes

peuvent être modélisés par des automates récursifs temporisés. [BMP10] affirme qu'un automate récursif temporisé peut être modélisés par un *TPAIO*. Donc, le système de [CJL+09] peut être modélisé par un *TPAIO*.

Ce chapitre est organisé de la manière suivante : on définit dans la Sec. 4.1 le modèle sur lequel portent nos travaux qui est celui des automates à pile temporisés avec entrées/sorties, ainsi que leur sémantique. On définit aussi dans la Sec. 4.2 la relation de conformité *tpioco* que nous considérons pour les *TPAIO*.

4.1/ *TPAIO* : DÉFINITION, SÉMANTIQUE ET EXEMPLES

Un automate à pile temporisé avec entrées/sorties appelé *TPAIO* (pour Timed Pushdown Automata with Inputs and Outputs) est un automate temporisé avec entrées/sorties et avec une pile non bornée. Il peut déposer dans ou extraire des éléments de la pile, mais, uniquement au sommet. Il est composé d'un ensemble fini de localités dénoté L et d'une relation de transitions dénotée Δ entre ces localités. Il est muni d'un ensemble fini d'horloges qui avancent toutes à la même vitesse. Chaque horloge de l'ensemble X a une valeur qui croît comme le temps, et qui peut être réinitialisée lors du franchissement d'une transition. Le temps s'écoule dans les localités et les actions (transitions) sont instantanées. Chaque transition est étiquetée par une garde qui indique si elle est franchissable ou pas en fonction de la valeur des horloges. Elle est aussi étiquetée par une deadline qui indique son niveau d'urgence. Elle est également étiquetée par un symbole de l'un des trois ensembles disjoints, celui des symboles d'entrée dénoté Σ_{in}, celui des symboles de sortie dénoté Σ_{out} et celui des symboles de pile dénoté Γ. Les actions d'empilement et de dépilement appartiennent à l'ensemble dénoté par Γ^{+-}. Les actions ayant pour exposant $+$, comme a^+ sont des empilements, celles ayant pour exposant $-$ sont des dépilements du symbole qui précède. Il existe aussi un symbole d'action non observable dénoté τ.
Formalisons maintenant la notion de *TPAIO* avec deadlines dans la Def. 15.

Définition 15 : *TPAIO*- Automates Temporisé à Pile avec Entrées/Sorties

Un *TPAIO* est un 7-uplet $= \langle L, l_0, \Sigma, \Gamma, X, \Delta, F \rangle$ où :

- L est un ensemble fini de localités,
- l_0 est la localité initiale,
- $\Sigma = \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}$ est un ensemble fini de symboles d'action : Σ_{in} est l'ensemble de symboles d'entrées, Σ_{out} est l'ensemble de symboles de sorties et τ est une action interne inobservable. Ces ensembles sont disjoints deux à deux,
- Γ est un ensemble fini de symboles de pile (on a $\Gamma \cap \Sigma = \emptyset$),
- X est un ensemble fini d'horloges,
- $F \subseteq L$ est un ensemble de localités finales,
- $\Delta \subseteq (L \times \Sigma \cup \Gamma^{+-} \times Grd(X) \times \{lazy, delayable, eager\} \times 2^X \times L)$ est un ensemble fini de transitions.

Chaque transition dans un *TPAIO* est de la forme $tr = (l, a, g, d, X', l')$ que l'on note aussi $l \xrightarrow{a,g,d,X'} l'$ où :

- l est la localité source et l' est la localité cible de la transition,
- a est le symbole de l'action qui est exécutée quand la transition est franchie. L'action peut être :
  — a^- : pour dépiler le symbole $a \in \Gamma$ du sommet de la pile,
  — a^+ : pour empiler le symbole $a \in \Gamma$ dans la pile,
  — $?A$ ou $!B$: respectivement pour recevoir le symbole $A \in \Sigma_{in}$ ou envoyer le symbole $B \in \Sigma_{out}$ sans changer l'état de la pile,
  — τ pour une action interne.
- g est la garde qui doit être satisfaite par les horloges pour que la transition soit franchissable,
- d est une deadline qui peut être soit *lazy*, soit *delayable* soit *eager*,
- X' est l'ensemble des horloges remises à zéro lors du franchissement de la transition.

La deadline d d'une transition peut être (voir la section 2.2.2) :

- $lazy$ (paresseuse) : si la garde de la transition est évaluée à vrai, la transition peut être franchie, mais le temps est aussi autorisé à s'écouler indéfiniment sans que la transition soit franchie. Il se peut donc qu'elle ne soit jamais franchie,
- $delayable$ (diffèrable) : si la garde de la transition est évaluée à vrai, le temps peut passer tant que cette garde reste évaluée à vrai et la transition sera franchie avant que la garde ne passe à faux,
- $eager$ (immédiate) : si la garde de la transition est évaluée à vrai, la transition est franchie immédiatement.

Définissons maintenant la sémantique d'un *TPAIO* T comme un *LTS* $\langle S^T, s_0^T, \Sigma \cup \Gamma \cup \{\tau\}, \Delta^T \rangle$ à nombre d'états infini (car le temps est dense) où s_0^T est l'état initial, S^T est un ensemble d'états et Δ^T est un ensemble de transitions. Un état d'un *TPAIO* est un triplet $(l, v, p) \in L \times (X \to \mathbb{R}^+) \times \Gamma^*$ où l représente la localité de contrôle courante, v est une valuation pour les horloges et p est le contenu de la pile. $\perp$ représente la pile vide et pour une pile non vide $p = p'.a$ où $p' \in \Gamma^*$ et $a \in \Gamma$, on note $Top(p) = a$ le sommet de la pile. On note $Top(p) = \perp$ si la pile p est vide. Dans la sémantique d'un *TPAIO*, il existe deux types de transitions :

- **transitions d'écoulement du temps** : pour $t \in \mathbb{R}^+$, on les note par $(l, v, p) \to^t (l, v', p)$ où $v' = v + t$. Une telle transition existe dans la sémantique si et seulement s'il n'existe aucune transition $(l, act, g, d, X', l') \in \Delta$ où soit $act \in \Gamma^+ \cup \Sigma$, soit $act = a^- \in \Gamma^-$ et $Top(p) = a$ telle que : (1). soit $d = delayable$ et il existe t_1 et t_2 tel que $0 \leqslant t_1 < t_2 \leqslant t$ sachant que $v + t_1 \vDash g$ et $v + t_2 \nvDash g$, (2). soit $d = eager$ et il existe t_1 tels que $0 \leqslant t_1 < t$ et $v + t_1 \vDash g$.

- **transitions d'action** : Il existe trois types de transitions d'action :
  — des transitions d'empilement : pour $a \in \Gamma$, on les note par $(l, v, p) \to^{a^+} (l', v', p.a)$ si la transition $(l, a^+, g, X', l') \in \Delta$.
  — des transitions de dépilement : pour $a \in \Gamma$, on les note par $(l, v, p.a) \to^{a^-} (l', v', p)$ si la transition $(l, a^-, g, X', l') \in \Delta$.
  — des transitions d'entrées/sorties ou internes et inobservables : pour $A \in \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}$, on les note par $(l, v, p) \to^A (l', v', p)$ si la transition $(l, A, g, X', l') \in \Delta$.

  Ces trois types de transition d'action existent dans la sémantique si les conditions suivantes sont satisfaites :

  1. v satisfait la garde g,

2. $v' = v[X' := 0]$, c'est à dire v' est la valuation de v dans laquelle les horloges de X' ont été remises à zéro.

Les deadlines des transitions imposent des conditions de progression du temps (*TPC*) aux localités dont elles sont issues. Les *TPC* nous permettent de proposer le concept de valeur d'horloges autorisées (*ACV* : authorized clock values) dans une localité, en fonction des valeurs d'arrivée dans la localité. Les *ACV* sont définies au moyen des définitions 16 et 17. Elles vont nous permettre de simplifier l'expression de la sémantique des transitions d'écoulement du temps dans un *TPAIO* avec deadlines.

Supposons que chaque garde de chaque transition est exprimée par une contrainte sur une seule horloge x sous la forme suivante : $ci \leqslant x \wedge x \leqslant cs$ où ci et cs sont respectivement des bornes inférieure et supérieure exprimées par des constantes entières. Ceci permet aussi de définir les deux formes de contraintes suivantes :

- $x \leqslant cs \equiv 0 \leqslant x \wedge x \leqslant cs$
- $x = cs \equiv cs \leqslant x \wedge x \leqslant cs$

Pour généraliser à tous les cas de gardes on considère qu'une garde qui porte sur plusieurs horloges est de la forme : $\bigwedge_{x \in X}(ci_x \leqslant x \wedge x \leqslant cs_x)$.

Pour définir les valeurs d'horloges autorisées *ACV* (Authorized Clock Values) dans une localité munie de la transition sortante $l \xrightarrow{act,g,d,X'} l'$, on a besoin de définir la notation suivante : $ACV_l(v_0, p, v)$ qui est le prédicat qui est vrai lorsque la valuation v est autorisée dans la localité l alors que le système y est arrivé avec une valuation v_0 et un contenu de pile p. Le prédicat $((act \in \Gamma^+ \cup \Sigma) \vee (act \in \Gamma^- \wedge act = a^- \wedge Top(p) = a))$ est vrai quand le contenu de la pile p autorise l'action act et faux sinon, quand le symbole à dépiler n'est pas au sommet de la pile. Notons qu'on a obligatoirement $v \geqslant v_0$. La Def. 16 définit l'*ACV* d'une transition $l \xrightarrow{act,g,d,X'} l'$, notée $ACV_{l \xrightarrow{act,g,d,X'} l'}(v_0, p, v)$.

Définition 16 : $ACV_{l \xrightarrow{act,g,d,X'} l'}(v_0, p, v)$ - Valeur d'horloges autorisées pour une transition

Étant donné v_0, l'instant d'arrivée dans la localité l et p le contenu de la pile dans la localité l, la valuation d'horloge v est autorisée dans cette localité l pour la transition $l \xrightarrow{act,g,d,X'} l'$ par la définition suivante : $ACV_{l \xrightarrow{act,g,d,X'} l'}(v_0, p, v)$=

- cas $d = lazy$: $true$
- cas $d = delayable$: $((act \in (\Gamma^+ \cup \Sigma) \vee (act \in \Gamma^- \wedge act = a^- \wedge Top(p) = a)) \Rightarrow (v(x) \geqslant v_0(x)) \wedge (v_0(x) \leqslant cs \Rightarrow v(x) \leqslant cs)$.
- cas $d = eager$: $((act \in \Gamma^+ \cup \Sigma) \vee (act \in \Gamma^- \wedge act = a^- \wedge Top(p) = a)) \Rightarrow (v(x) \geqslant v_0(x)) \wedge (v_0(x) < ci \Rightarrow v(x) \leqslant ci) \wedge (ci \leqslant v_0(x) \leqslant cs \Rightarrow (v(x) = v_0(x)))$.

La Fig. 4.1 illustre $ACV_{l \xrightarrow{a^-, ci \leqslant x \wedge x \leqslant cs, d, X'} l'}(v_0, p.a, v)$ en indiquant en gras quand la valuation d'horloges v est autorisée, avec le symbole a est au sommet de la pile. Par exemple, si la deadline d est $delayable$ et la localité l est atteinte avec $v_0 \leqslant c_s$, les valeurs autorisées de v sont les valeurs entre v_0 et c_s. Par contre, si la deadline est $eager$, seule la valeur v_0

est autorisée quand v_0 appartient à l'intervalle $[c_i, c_s]$. Autrement dit, la transition doit être franchise immédiatement.

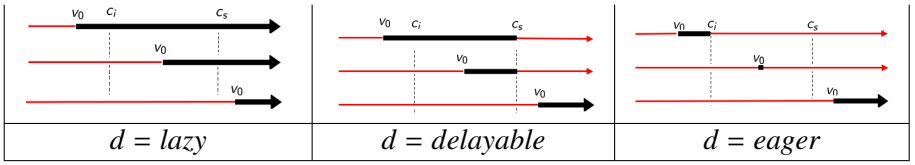


FIGURE 4.1 – Valeurs d'horloges autorisées représentées en gras pour chaque deadline d'une transition déclenchable et pour trois classes d'instants d'arrivée.

Figure 4.2 illustre $ACV_{l \xrightarrow{b^-, ci \leqslant x \wedge x \leqslant cs, d, X'} l'}(v_0, p.a, v)$ en indiquant en gras quand la valuation v est autorisée dans le cas où le symbole $b \neq a$ à dépiler n'est pas au sommer de la pile $p.a$. Donc, pour chaque deadline d, le temps est autorisé à s'écouler indéfiniment.

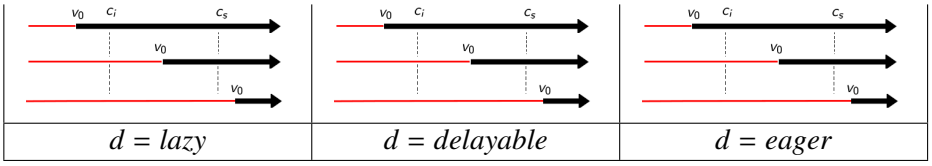


FIGURE 4.2 – Valeurs d'horloges autorisées représentées en gras pour chaque deadline d'une transition de dépilement non déclenchable et pour trois classes d'instants d'arrivée.

Pour une localité l ayant n transitions en sortie, l'*ACV* de l est la conjonction des *ACV* de ses transitions sortantes. L'*ACV* est définie dans la Def. 17.

Définition 17 : $ACV_l(v_0, p, v)$- Valeur d'horloges autorisées pour une localité

Étant donné v_0, l'instant d'arrivée dans la localité l et p le contenu de la pile dans la localité l, la valuation d'horloge v est autorisée dans la localité l ayant n transitions de sortie si elle satisfait la définition suivante : $ACV_l(v_0, p, v) = \bigwedge_{i=1}^{n} ACV_{l \xrightarrow{act_i, g_i, d_i, X_i} l_i}(v_0, p, v)$.

Les *ACV* nous permettent de simplifier l'expression des transitions d'écoulement du temps dans la sémantique des *TPAIO* (voir page 59) de la manière suivante :

Une transition d'écoulement du temps $(l, v, p) \rightarrow^t (l, v + t, p)$ avec $t \in \mathbb{R}^+$ existe dans la sémantique d'un *TPAIO* si et seulement si $ACV_l(v, p, v + t) = true$.

Exemple 4.1.1. Les tables 4.1, 4.2 et 4.3 présentent les valeurs de vérité de l'*ACV* de la localité l_1 du *TPAIO* présenté dans la Fig. 4.3 en le considérant pour une pile vide et différentes valeurs de v_0 et v et les différents types de deadline d_1 pour la transition $(l_1, a^+, 1 \leqslant x \leqslant 5, d_1, l_2)$. L'*ACV* de cette localité est donnée par :

- cas $d_1 = lazy$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} true$
- cas $d_1 = delayable$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} (v(x) \geqslant v_0(x)) \wedge (v_0(x) \leqslant 5 \Rightarrow v(x) \leqslant 5)$
- cas $d_1 = eager$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} (v(x) \geqslant v_0(x)) \wedge (v_0(x) < 1 \Rightarrow v(x) \leqslant 1) \wedge (1 \leqslant v_0(x) \leqslant 5 \Rightarrow (v(x) = v_0(x)))$

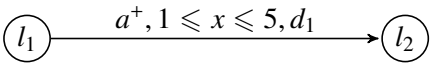


FIGURE 4.3 – Transition avec deadline d'un *TPAIO*

$d_1 = lazy$		
v_0	v	$ACV_{l_1}(v_0, \perp, v)$
$x \geqslant 0$	$x \geqslant 0$	true

TABLE 4.1 – Valeurs de l'*ACV* de la localité l_1 pour une pile vide et différentes valeurs v_0 et v et la deadline *lazy*

$d_1 = delayable$		
v_0	v	$ACV_{l_1}(v_0, \perp, v)$
$x \leqslant 5$	$v_0(x) \leqslant x \leqslant 5$	$true$
	$x > 5$	$false$
$x > 5$	$x > 5$	$true$

TABLE 4.2 – Valeurs de l'*ACV* de la localité l_1 pour une pile vide et différentes valeurs v_0 et v et la deadline *delayable*

$d_1 = eager$		
v_0	v	$ACV_{l_1}(v_0, \perp, v)$
$x \leqslant 1$	$v_0(x) \leqslant x \leqslant 1$	$true$
	$x > 1$	$false$
$1 < x \leqslant 5$	$x = v_0(x)$	$true$
	$x \neq v_0(x)$	$false$
$x > 5$	$x > 5$	$true$

TABLE 4.3 – Valeurs de l'*ACV* de la localité l_1 pour une pile vide et différentes valeurs v_0 et v et la deadline *eager*

Exemple 4.1.2. Le *TAD* présenté dans la Fig. 2.3 (page 13) est un cas particulier d'un *TPAIO* où $\Sigma = \{A, B\}$, $\Gamma = \emptyset$ où le contenu de la pile est toujours vide. L'*ACV* pour les différents types de deadline pour les deux transitions $(l_1, A, 1 \leqslant x \leqslant 5, d_1, \emptyset, l_2)$ et $(l_1, B, 4 \leqslant y \leqslant 6, d_2, \emptyset, l_3)$ est :

- cas $d_1 = lazy$:
  - cas $d_2 = lazy$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} true$
  - cas $d_2 = delayable$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} true \wedge (v(y) \geqslant v_0(y)) \wedge (v_0(y) \leqslant 6 \Rightarrow v(y) \leqslant 6)$
  - cas $d_2 = eager$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} true \wedge (v(y) \geqslant v_0(y)) \wedge (v_0(y) < 4 \Rightarrow v(y) \leqslant 4) \wedge (4 \leqslant v_0(y) \leqslant 6 \Rightarrow (v(y) = v_0(y)))$
- cas $d_1 = delayable$:
  - cas $d_2 = lazy$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} (v(x) \geqslant v_0(x)) \wedge (v_0(x) \leqslant 5 \Rightarrow v(x) \leqslant 5)$

- cas $d_2 = delayable$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} (v(x) \geqslant v_0(x)) \wedge (v_0(x) \leqslant 5 \Rightarrow v(x) \leqslant 5) \wedge (v(y) \geqslant v_0(y)) \wedge (v_0(y) \leqslant 6 \Rightarrow v(y) \leqslant 6)$
- cas $d_2 = eager$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} (v(x) \geqslant v_0(x)) \wedge (v_0(x) \leqslant 5 \Rightarrow v(x) \leqslant 5) \wedge (v(y) \geqslant v_0(y)) \wedge (v_0(y) < 4 \Rightarrow v(y) \leqslant 4) \wedge (4 \leqslant v_0(y) \leqslant 6 \Rightarrow (v(y) = v_0(y)))$

- cas $d_1 = eager$:
  - cas $d_2 = lazy$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} true \wedge (v(x) \geqslant v_0(x)) \wedge (v_0(x) < 1 \Rightarrow v(x) \leqslant 1) \wedge (1 \leqslant v_0(x) \leqslant 5 \Rightarrow (v(x) = v_0(x)))$
  - cas $d_2 = delayable$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} (v(y) \geqslant v_0(y)) \wedge (v_0(y) \leqslant 6 \Rightarrow v(y) \leqslant 6) \wedge (v(y) \geqslant v_0(y)) \wedge (v_0(y) \leqslant 6 \Rightarrow v(y) \leqslant 6)$
  - cas $d_2 = eager$: $ACV_{l_1}(v_0, \perp, v) \stackrel{def}{=} (v(x) \geqslant v_0(x)) \wedge (v_0(x) < 1 \Rightarrow v(x) \leqslant 1) \wedge (1 \leqslant v_0(x) \leqslant 5 \Rightarrow (v(x) = v_0(x))) \wedge (v(y) \geqslant v_0(y)) \wedge (v_0(y) < 4 \Rightarrow v(y) \leqslant 4) \wedge (4 \leqslant v_0(y) \leqslant 6 \Rightarrow (v(y) = v_0(y)))$

Les transitions d'écoulement du temps ne changent pas de localité. Le temps s'écoule dans les localités, mais, soit pas au delà de l'instant de début de la déclenchabilité des transitions sortantes de deadline *eager*, soit pas au delà de l'instant de fin de déclenchabilité des transitions de deadline *delayable*. Notons cependant que si l'instant d'arrivée dans la localité dépasse l'instant de fin de déclenchement des transitions $eager$ et $delayable$, alors, le temps continue à s'écouler indéfiniment. Notons que les transitions *lazy* peuvent laisser le temps s'écouler à l'infini. Notons enfin que les transitions de dépilement impossibles à cause d'un mauvais sommet de pile laissent également le temps s'écouler à l'infini. Les transitions d'action permettent de changer de localité par empilement, dépilement, réception ou émission d'un symbole ou par une action interne inobservable.

Un *TPAIO* est normalisé s'il exécute séparément les empilements et les dépilements. Il existe trois modes de reconnaissance d'un langage par un *TPAIO*.

- Reconnaissance par localité finale : un mot est accepté si, à partir de la localité initiale, il peut être entièrement lu en arrivant à une localité de F, ceci quel que soit le contenu de la pile à ce moment-là. La pile est supposée vide au départ.
- Reconnaissance par pile vide : un mot est accepté si, à partir de la localité initiale, il peut être entièrement lu en vidant la pile.
- Reconnaissance par localité finale et pile vide : un mot est accepté si, à partir de la localité initiale, il peut être entièrement lu en vidant la pile et en arrivant à une localité de F.

Exemple 4.1.3. La Fig. 4.4 montre un exemple de *TPAIO* avec deadlines. Ce *TPAIO* modélise un système de détection de $n-$clics de souris ($n \geqslant 1$). Il compte le nombre de clics qui ont eu lieu dans une unité de temps après le premier clic en utilisant une pile. L'horloge x est mise à zéro après la réception du premier clic ($?clic$), puis le nombre de clics est immédiatement incrémenté ($clic_1^+$) et la localité l_2 est atteinte. Si aucun clic n'a lieu pendant une unité de temps, alors la pile est dépilée ($clic_1^-$), l'automate revient à la localité l_0 et donc un simple clic est détecté. Sinon, l'automate passe à la localité l_3 et commence à calculer le nombre de clics ($clic_n^+$) pendant une unité de temps. Il atteint la localité l_6 après l'écoulement d'une unité de temps, puis, la pile est dépilée autant de

fois que le nombre de clics et l'automate atteint la localité l_0. Les transitions *eager* se déclenchent dés que $x = 1$ est satisfait. Les transitions *delayable* se déclenchent dans l'intervalle $[1, 2]$ en pouvant laisser passer le temps sans dépasser $x = 2$. Les transitions *lazy* peuvent laisser le temps s'écouler sans aucune restriction. Par exemple, dans la localité l_2, la transition $l_2 \xrightarrow{?clic, x<1, lazy} l_3$ est déclenchable tant que $x < 1$ est satisfait, mais, si le temps s'écoule jusqu'à 1 sans arrivée de clic, alors, la transition $l_2 \xrightarrow{clic_1^-, x=1, eager, \{x\}} l_0$ est franchie en urgence et la transition $l_2 \xrightarrow{?click, x<1, lazy} l_3$ n'est plus franchissable.

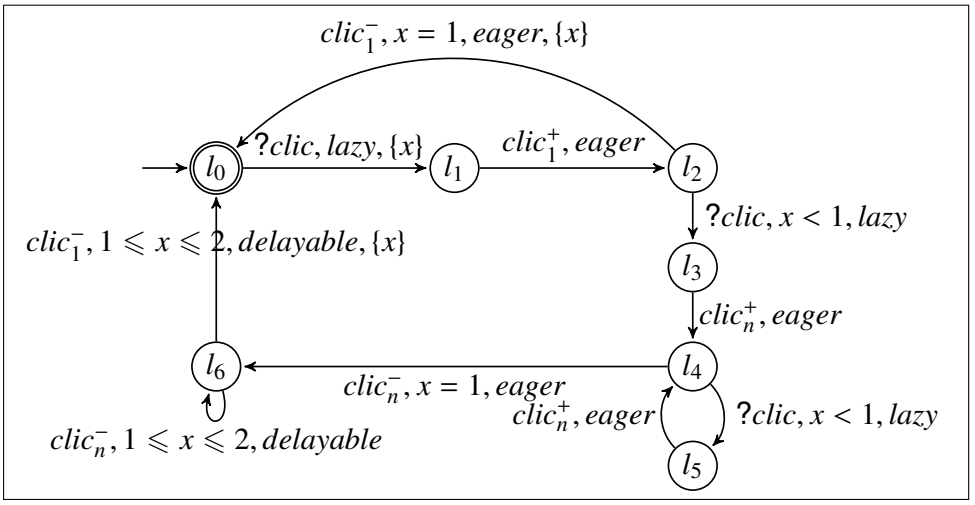


FIGURE 4.4 – Exemple d'un *TPAIO* décrivant la reconnaissance d'un simple ou de plusieurs clics

4.2/ RELATION DE CONFORMITÉ *tpioco*

On introduit, dans cette sous section, une relation de conformité pour les automates à pile temporisés avec entrées/sorties qu'on appelle *tpioco* (Timed Pushdown Input/Output Conformance relation). Cette relation est une extension de la relation de conformité des systèmes temps-réel *tioco* [KT09] qui est elle-même inspirée de la relation de conformité non temporisée *ioco*. C'est la même relation que *tioco* pour les *TAIO* mais avec un ensemble d'actions égal à $\Sigma \cup \Gamma^{+-}$ au lieu de Σ. On suppose aussi qu'on peut observer en plus des actions de sortie, les actions de pile afin de détecter la non-conformité aussi par rapport aux actions de la pile. Les actions observables sont donc $\Sigma_{out} \cup \Gamma^{+-}$. Les actions d'entrées restent Σ_{in}.

Exemple 4.2.1. La Fig. 4.5.(a) présente un exemple de spécification T où $\Sigma_{in} = \{A\}$, $\Sigma_{out} = \{B\}$ et $\Gamma = \{a\}$. La spécification T est non bloquante. Les figures 4.5.(b), 4.5.(c), 4.5.(d) et 4.5.(e) présentent des exemples d'implémentations appelées I_1, I_2, I_3 et I_4. On suppose que I_1, I_2, I_3 et I_4 sont complètes en entrée ce qui n'apparaît pas dans la figure pour ne pas la surcharger. La spécification T doit envoyer B avant 4 unités de temps et au plus tôt 2 unités de temps après la réception de A. Elle peut empiler a à la 4^{eme} unité de temps après la réception de A si elle n'a pas déjà envoyé B. L'implémentation I_1 est conforme à T parce que par exemple $out(I_1\ after\ ?A\ ) = \{B, a^+\} \cup [0, 4] \subseteq out(T\ after\ ?A\ ) = \{B, a^+\} \cup [0, 4]$. L'implémentation I_2 n'est pas conforme à T parce que $out(I_2\ after\ ?A\ ) = \mathbb{R}^+ \nsubseteq out(T\ after\ ?A\ ) = \{B, a^+\} \cup [0, 4]$. L'implémentation I_3 n'est pas conforme à T parce

que $out(I_3\ after\ ?A\ a^+\ ) = \{a^+\} \cup \mathbb{R}^+ \nsubseteq out(T\ after\ ?A\ a^+\ ) = \{a^-\} \cup \mathbb{R}^+$. L'implémentation I_4 est conforme à T parce que l'action $?C$ n'est pas spécifiée dans T. Elle est conforme car $out(I_4\ after\ ?A\ ) = \emptyset \subseteq out(T\ after\ ?A\ ) = \{B, a^+\} \cup [0,4]$. L'implémentation I_5 est conforme à T parce que par exemple $out(I_1\ after\ ?A\ ) = \{B\} \cup [0,3] \nsubseteq out(T\ after\ ?A\ ) = \{B, a^+\} \cup [0,4]$. $out(I_1\ after\ ?A\ ) \neq \{a^-, B\} \cup [0,3]$ est vrai parce que le symbole a n'existe pas au sommet de la pile après l'observation de la trace $?A$.

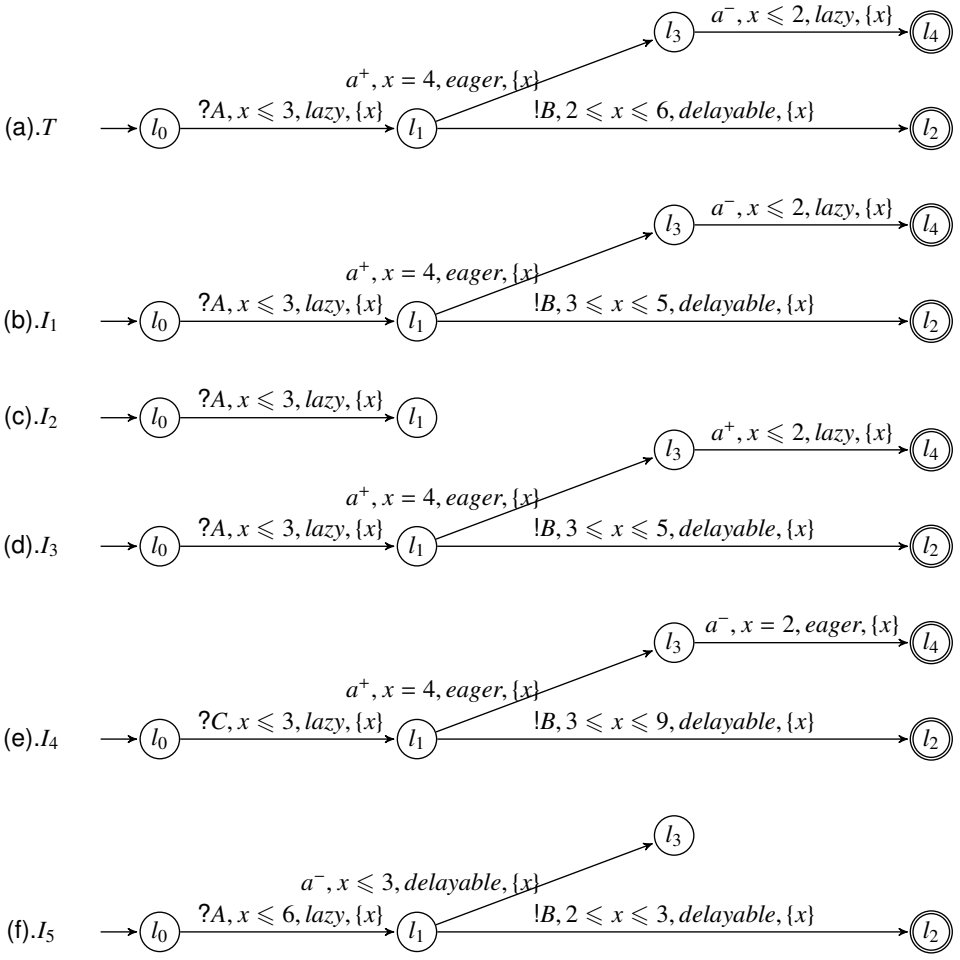


FIGURE 4.5 – Exemple de spécification et de cinq implémentations

4.3/ ATTEIGNABILITÉ DANS UN *TPAIO*

Le problème d'accessibilité dans un *TPAIO* consiste à décider si un état d'un automate à pile temporisé est atteignable. La décidabilité de l'accessibilité dans un automate à pile temporisé est une propriété importante dans l'utilisation de ce modèle pour la vérification et pour le test.

Théorème 4.3.1 *(*[Dan01]*)*. L'accessibilité d'un état dans un automate à pile temporisé est un problème EXPTIME-complete.

4.4/ CONCLUSION

Dans ce chapitre on a présenté le modèle des automates à pile temporisés avec entrées/sorties et notre première contribution qui est une relation de conformité pour ce modèle avec la définition des *ACV* (valeurs d'horloge autorisées) des localités. Dans le chapitre suivant, on présente une méthode polynomiale et incomplète de génération de tests à partir d'un *TPAIO* déterministe et avec des deadlines $lazy$. La méthode est incomplète au sens où certains des états accessibles ne sont pas détectés comme tels étant donné l'approximation effectuée pour obtenir une méthode polynomiale.

5

GÉNÉRATION DE TESTS À PARTIR D'UN *TPAIO* DÉTERMINISTE

Sommaire

On propose dans ce chapitre une solution algorithmique de complexité polynomiale de génération de tests à partir d'un automate à pile temporisé déterministe avec entrées/sorties. On considère pour cette étude que les deadlines de toutes les transitions du modèle sont *lazy*. L'atteignabilité des localités dans un *TPAIO* peut se vérifier en temps polynomial, dans un *PA* aussi. Mais, quand on associe des contraintes de pile et d'horloge comme dans notre modèle, alors l'atteignabilité devient exponentielle [CLPV10]. Notre première contribution est une méthode qui adapte aux automates à pile temporisés une méthode de calcul des états accessibles définie pour des automates à pile. L'adaptation consiste à prendre en compte les contraintes de temps en plus des contraintes de pile. Un testeur est également défini à partir de l'automate à pile temporisé de départ. Notre seconde contribution est de proposer une méthode incomplète mais polynomiale de génération de tests à partir d'un *TPAIO* déterministe et sans spécification de deadline.

Pour limiter la complexité à une complexité polynomiale, on a défini volontairement un algorithme de calcul des automates temporisés d'accessibilité incomplet. On calcule, en utilisant un algorithme polynomial, un automate temporisé d'atteignabilité *RTA* (Reachability Timed Automaton) à partir d'un *TPAIO* donné. Pour vérifier si une succession de transitions temporisées peut être exécutée ou non, on calcule les fermetures arrières des contraintes d'horloge, et on les évalue en utilisant des procédures de décision de satisfiabilité. On calcule aussi une table des chemins qui associe à chaque transition du *RTA* un ou plusieurs chemins du *TPAIO*. Le prix à payer pour cette polynomialité est l'incomplétude de la méthode, qui peut ne pas détecter certains états atteignables. L'activité de test étant par nature incomplète, cette limite est acceptable dans ce cadre et a pour conséquence que certains états atteignables peuvent ne pas être pris en compte dans les tests produits. Les cas de tests obtenus sont enrichis par des transitions qui atteignent le verdict $fail$ et $inconc$. Ces transitions sont obtenues

à partir du testeur *TPTIO* (Timed Pushodwn Tester with Inputs and Outputs) qui est calculé à partir du *TPAIO* en ajoutant des transitions qui mènent à $fail$ et $inconc$ à partir de chaque localité du *TPAIO*. Cette contribution a donné lieu à la publication [MJMR15b].

Ce chapitre est organisé de la manière suivante : la Sec. 5.1 décrit le processus d'application des trois étapes majeures de la génération de tests à partir d'un *TPAIO* déterministe. Elle définit informellement chaque étape. La première étape est la construction d'un testeur à partir d'un *TPAIO*. Elle est présentée dans la Sec. 5.2. La seconde étape décrite dans la Sec. 5.3 présente la méthode de calcul d'un *RTA* à partir d'un *TPAIO*. La Sec. 5.4 présente la troisième étape qui est la génération d'un arbre de tests. La Sec. 5.5 discute de la correction, l'incomplétude et la couverture de notre méthode.

5.1/ PROCESSUS DE GÉNÉRATION DE TESTS

Le diagramme de la Fig. 5.1 présente les trois étapes majeures pour la génération de tests à partir d'un *TPAIO* déterministe. Pour les travaux présentés dans ce chapitre, les deadlines de toutes les transitions de notre modèle sont *lazy*. Ces étapes sont les suivantes.

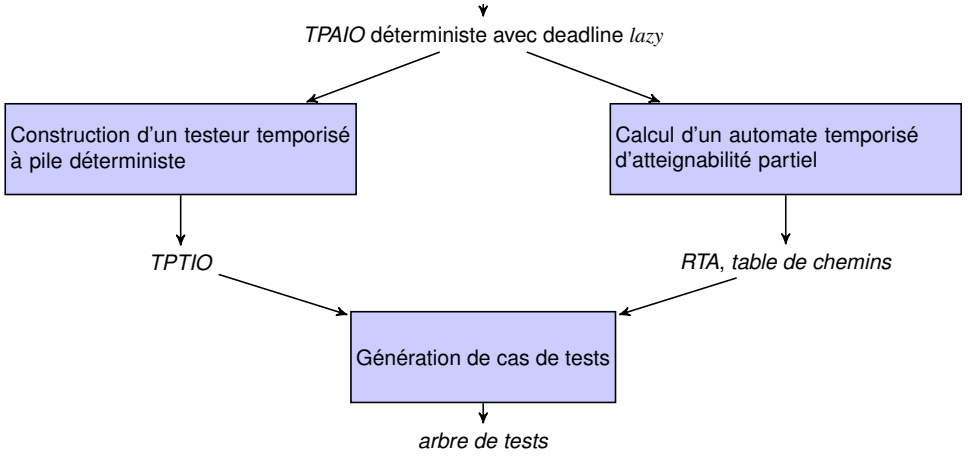


FIGURE 5.1 – Processus de génération de tests à partir d'un *TPAIO*

1. **Construction d'un testeur temporisé à pile déterministe** : afin de détecter la non conformité, on propose d'ajouter des transitions qui mènent à un verdict $fail$ à partir d'une localité courante. Vu qu'on considère que toutes les deadlines sont à $lazy$, il est possible d'avoir des exécutions qui se bloquent dans un état en laissant le temps s'écouler à l'infini. Comme il est impossible d'attendre indéfiniment pour savoir si l'implémentation est conforme à la spécification ou non, dans ce cas, on propose d'ajouter un verdict inconclusif nommé $inconc$ qui indique que l'implémentation n'a pas produit. L'action lui permettant de quitter la localité, mais n'a pas produit non plus d'erreur en produisant une action au delà du temps imparti. Donc, on propose dans cette étape de calculer un testeur temporisé à pile avec entrées/sorties déterministe appelé *TPTIO* (Timed Pushodwn Tester with Inputs and Outputs). Ce

testeur est le *TPAIO* d'origine enrichi avec deux nouvelles localités $fail$ et $inconc$ et des nouvelles transitions qui y mènent. Cette étape est présentée dans la Sec. 5.2

2. **Calcul d'un automate temporisé d'atteignabilité** partielle avec sa table de chemins. Un cas de test tient compte des contraintes de pile et d'horloges. Les actions de dépilement dépendent du contenu de la pile car, par exemple, il est impossible de dépiler un symbole qui n'existe pas au sommet de pile. Cette étape calcule un ou plusieurs chemins entre deux localités en respectant les contraintes de la pile. Un *RTA* est un automate temporisé fini. Une table de chemins associe un ou plusieurs chemins du *TPAIO* à chaque transition du *RTA*. Cette étape est présentée dans la Sec. 5.3.
3. **Génération d'un arbre de tests** à partir d'un *TPAIO* en utilisant le *TPTIO*, le *RTA* et sa table de chemins. Cette étape est divisée en deux phases : (a). générer un arbre de tests qui est un arbre de chemins associés aux transitions du *RTA* qui partent d'une localité initiale vers une localité finale du *RTA* ; (b). enrichir l'arbre de tests avec les transitions qui atteignent $fail$ et $inconc$ en utilisant son *TPTIO*. Cette étape est présentée dans la Sec. 5.4.

5.2/ CONSTRUCTION D'UN TESTEUR À PARTIR D'UN *TPAIO*

De la même façon que dans [Tre96] pour les *TAIO*, on calcule un testeur à partir d'un *TPAIO* afin de détecter la non-conformité entre une implémentation et un *TPAIO*. Le testeur calculé appelé *TPTIO* est un *TPAIO* dont les actions de sorties sont les actions d'entrées (Σ_{in}) du *TPAIO*, les actions d'entrées sont celles de sorties (Σ_{out}) du *TPAIO* et les actions de la pile sont celles de la pile (Γ) du *TPAIO* comme l'indique la Def. 18. C'est un *TPAIO* qui est enrichi par une nouvelle localité puits $fail$ et des transitions qui l'atteignent à partir d'autres localités. Il est obtenu à partir d'un *TPAIO* déterministe en inversant les entrées et les sorties pour interagir avec l'implémentation à tester et en complétant avec des transitions menant à $fail$ indiquant l'échec du test appelée $fail$. Vu qu'on considère des *TPAIO* où les deadlines sont toutes à $lazy$, les exécutions qui se bloquent dans un état quelconque en laissant le temps s'écouler sans faire d'actions discrètes sont conformes au modèle comme nous n'avons pas fait l'hypothèse de systèmes non bloquants. Les tests qu'on engendrent ne sont jamais des exécutions bloquantes car on ne saurait pas les détecter en pratique. En effet, il est impossible d'attendre indéfiniment pour savoir si l'implémentation est réellement bloquante. Par conséquent, du point de vue pratique, pour exécuter les tests, on peut admettre que le banc de test déclenche un dépassement de temps au bout d'un temps fixé arbitrairement et suffisamment long. Dans ce cas, le verdict est $inconc$ car on ne sait pas ce que fait réellement le système et laisse passer le temps (conforme avec la sémantique de la deadline $lazy$) ou si le système aurait engendré un $fail$ plus tard. En fait, on ne sait pas conclure le système se bloque. En réalité on observe que le système ne fait plus rien vis à vis de l'extérieur, mais, il peut faire des actions internes. C'est une sorte de quiescence. Pour chaque localité l, un ensemble de transitions qui atteint $fail$ et $inconc$ à partir de l est défini. Dans la Def. 18, $\Delta_{\rightarrow fail}$ définit que la localité $fail$ est atteinte dans chacun des cas suivants :

- (i) Observation d'une action de la pile ou de sortie non spécifiée : autrement dit $fail$ est atteint si le système observe à la localité l une action $act \in \Gamma^{+-} \cup \Sigma_{out}$, mais, qu'il n'existe pas de transition (l, act, g, X', l') dans Δ.

- (ii) Observation d'une action de pile ou de sortie plus tôt ou plus tard que spécifiée.

- (iii) Observation d'une action non autorisée dans T, mais qui peut-être exécutée par l'implémentation.

Dans la Def. 18, $\Delta_{\rightarrow inconc}$ définit que la localité $inconc$ est atteinte dans le cas suivant :

- (iv) Pas d'observation d'action de la pile ou de sortie avant le dépassement du délai arbitrairement fixé pour l'observation des actions de la pile et de sortie dans la localité courante.

Lorsque le testeur atteint $fail$, cela signifie que l'exécution observée sur l'implémentation est erronée, c'est-à-dire non conforme à la spécification. Lorsque le testeur atteint $inconc$, cela signifie qu'on ne sait pas si l'implémentation est conforme ou non. On ne sait pas si l'implémentation est conforme ou si le système aurait engendré un $fail$ plus tard.

Définition 18 : Testeur à pile temporisé déterministe

Un *TPTIO* $T^T = \langle L \cup \{fail\} \cup \{inconc\}, l_0, \Sigma^T, \Gamma, X, \overline{\Delta} \cup \Delta_{\rightarrow fail} \cup \Delta_{\rightarrow inconc}, F \rangle$ d'un *TPAIO* $T = \langle L, l_0, \Sigma, \Gamma, X, \Delta, F \rangle$ est défini ainsi où $\Sigma^T_{in} = \Sigma_{out} \cup \{othw\} \cup \{time\}$ et $\Sigma^T_{out} = \Sigma_{in}$. Les transitions $\overline{\Delta}$ sont les mêmes que celles de Δ dans lesquelles on a inversé les émissions et les réceptions. Les transitions de $\Delta_{\rightarrow fail}$ et $\Delta_{\rightarrow inconc}$ sont définies comme suit pour tout $l \in L$ et pour tout $act \in \Gamma^{+-} \cup \Sigma_{out}$:

- (i) $(l, act, true, \emptyset, fail) \in \Delta_{\rightarrow fail}$ si $\forall .(act', g, X', l').(l, act', g, X', l') \in \Delta \implies act' \neq act)$.
- (ii) $(l, act, \neg g, \emptyset, fail) \in \Delta_{\rightarrow fail}$ si $(l, act, g, X', l') \in \Delta$.
- (iii) $(l, ?othw, true, \emptyset, fail) \in \Delta_{\rightarrow fail}$.
- (iv) $(l, ?time, g_{inconc}, \emptyset, inconc) \in \Delta_{\rightarrow inconc}$ où $g_{inconc} = \bigwedge_{(l,act,g,X',l') \in \Delta} \neg g$

L'algorithme 5.2.1 permet de calculer un testeur T^T à partir d'un *TPAIO* donné. Il calcule les transitions qui atteignent $fail$. Il traite les quatre cas de la Def. 18. La ligne 5 de l'algorithme correspond au cas (iii) de la Def. 18. La ligne 8 de l'algorithme correspond au cas (i) de la Def. 18 où on ajoute la transition $(l, act, true, \emptyset, fail)$ s'il n'existe aucune transition (l, act, g, X', l') dans Δ où $act \in \Sigma_{out} \cup \Gamma^{+-}$. La ligne 10 de l'algorithme correspond au cas (ii) de la Def. 18 où on ajoute la transition $(l, act, \neg g, \emptyset, fail)$ où $act \in \Sigma_{out} \cup \Gamma^{+-}$ et g est la garde d'une transition de Δ qui part de l avec le label act. La ligne 15 de l'algorithme correspond au cas (iv) de la Def. 18 où on ajoute la transition $(l, ?time, g_{inconc}, \emptyset, inconc)$ où g_{inconc} est la conjonction des négations des gardes des transitions étiquetées par une action de la pile et de sortie et qui quittent la localité l.
L'exemple 5.2.1 présente un *TPAIO* qui sert à illustrer les concepts de ce chapitre, en particulier la notion de testeur et de *RTA*.

Exemple 5.2.1. La Fig. 5.2 présente exemple de *TPAIO* où $\Sigma_{in} = \{A, D, F, H\}$, $\Sigma_{out} = \{B, C, E, G\}$ et $\Gamma = \{pow\}$.

Algorithme 5.2.1 Calcul du testeur à pile temporisé déterministe à partir d'un *TPAIO*

ENTRÉE : Un *TPAIO* $T = \langle L, l_0, \Sigma, \Gamma, X, \Delta, F\rangle$

SORTIE : un *TPAIO* $T^T = \langle L \cup \{fail\} \cup \{inconc\}, l_0, \Sigma^T, \Gamma, X, \overline{\Delta} \cup \Delta_{\rightarrow fail} \cup \Delta_{\rightarrow inconc}, F\rangle$ qu'est un testeur

VARIABLES : $l \in L$, $\Delta_{\rightarrow fail} \subseteq L \times \Sigma \times Grd(X) \times 2^X \times fail$, $\Delta_{\rightarrow inconc} \subseteq L \times \Sigma \times Grd(X) \times 2^X \times inconc$, $act \in \Sigma$, $g, g_{inconc} \in Grd(X)$, $X' \subseteq X$

DÉBUT

1: $\Delta_{\rightarrow fail} \leftarrow \emptyset$
2: $\Delta_{\rightarrow inconc} \leftarrow \emptyset$
3: **pour** chaque $l \in L$ **faire**
4: $g_{inconc} \leftarrow true$
5: $\Delta_{\rightarrow fail} \leftarrow \Delta_{\rightarrow fail} \cup \{(l, ?othw, true, \emptyset, fail)\}$ // cas(iii)
6: **pour** $act \in \Sigma_{out} \cup \Gamma^{+-}$ **faire**
7: **si** $\neg\exists.(l, act, g, X', l') \in \Delta$ **alors**
8: $\Delta_{\rightarrow fail} \leftarrow \Delta_{\rightarrow fail} \cup \{(l, act, true, \emptyset, fail)\}$ // cas(i)
9: **sinon**
10: $\Delta_{\rightarrow fail} \leftarrow \Delta_{\rightarrow fail} \cup \{(l, act, \neg g, \emptyset, fail)\}$// cas(ii)
11: $g_{inconc} = g_{inconc} \wedge \neg g$
12: **fin si**
13: **fin pour**
14: **si** $g_{inconc} \neq true$ **alors**
15: $\Delta_{\rightarrow inconc} \leftarrow \Delta_{\rightarrow inconc} \cup \{(l, ?time, \neg g_{inconc}, \emptyset, inconc)\}$// cas(iv)
16: **fin si**
17: **fin pour**

FIN.

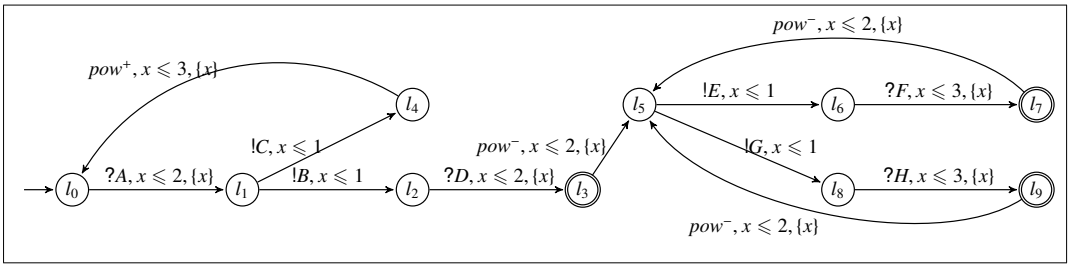


FIGURE 5.2 – Exemple d'un *TPAIO*

La Fig. 5.3 présente le testeur associée au *TPAIO* de la Fig 5.2. Le label $a_1|a_2|...|a_n$ dénote un ensemble de labels $\{a_1, a_2, ..., a_n\}$. Les notations de F_0 à F_4 sont les abréviations suivantes : $F_0 \stackrel{def}{=} ?B|?C|?E|?G|pow^+$, $F_1 \stackrel{def}{=} F_0|pow^-$, $F_2 \stackrel{def}{=} ?E|?G|pow^+|pow^-$, $F_3 \stackrel{def}{=} ?B|?C|?E|?G|pow^-$ et $F_4 \stackrel{def}{=} ?B|?C|pow^+|pow^-$. Le testeur de la Fig. 5.3.(a) possède par exemple la transition $(l_1, F_2, true, \emptyset, fail)$ car $\{E, G, pow^+, pow^-\} \subseteq \Sigma_{out} \cup \Gamma^{+-}$ et il n'existe aucune transition qui quitte l_1 avec les labels E, G, pow^+ ou pow^-. Cette transition est ajoutée pour satisfaire le cas (i) dans la Def. 18. Pour satisfaire le cas (ii), il contient aussi la transition $(l_1, ?B|?C, x > 1, \emptyset, fail)$ car $\{B, C\} \subseteq \Sigma_{out} \cup \Gamma^{+-}$. Son *TPAIO* (Fig 5.2) possède les transitions $(l_1, ?B, x \leqslant 1, \{x\}, l_2)$ et $(l_1, ?C, x \leqslant 1, \{x\}, l_4)$ dont la négation de la garde est égale à $x > 1$. Cette transition est ajoutée selon le cas (ii) dans la Def. 18. Pour satisfaire le cas (iii), il contient par exemple la transition $(l_1, ?othw, true, \emptyset, fail)$. Pour satisfaire le cas (iv), il contient par exemple la transition $(l_1, ?time, x > 1, \emptyset, inconc)$.

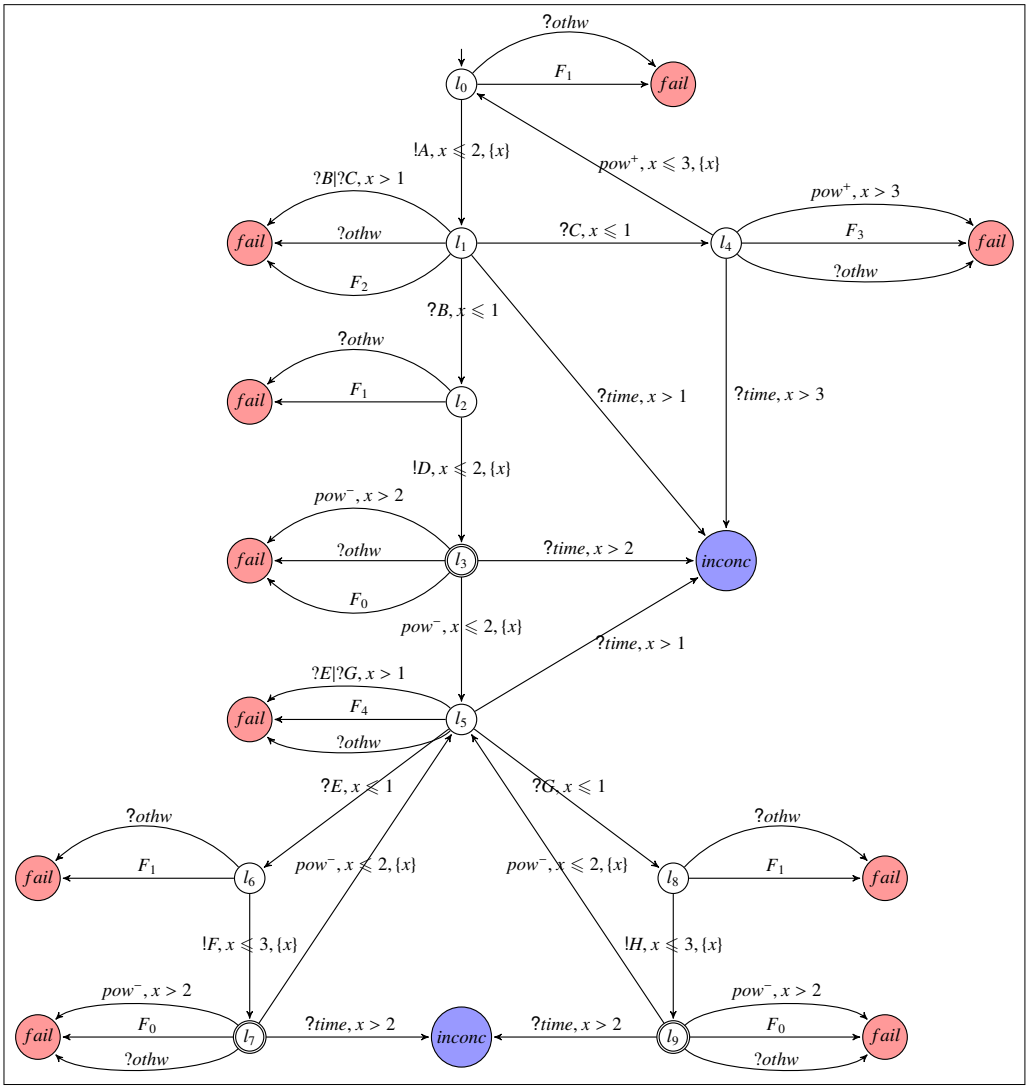


FIGURE 5.3 – Testeur associé au *TPAIO* de la Fig 5.2

5.3/ CALCUL D'UN *RTA* À PARTIR D'UN *TPAIO*

On définit à partir d'un *TPAIO* un automate temporisé d'accessibilité (*RTA*) qui est un *TAIO* dont l'ensemble des transitions peut être infini et où chacune d'elles est étiquetée par ϵ. Pour cela, on complète l'ensemble de règles définies pour les automates à pile issues de [FWW97], afin de traiter les *TPAIO* en prenant en considération les contraintes d'horloges. Le calcul consiste principalement à fusionner une séquence de deux transitions consécutives d'empilement et de dépilement en une ϵ-Transition qui correspond alors à un chemin qui respecte les contraintes de la pile. Ensuite, selon la Def. 19, les ϵ-transitions consécutives sont fusionnées, etc. Les contraintes d'horloges sont accumulées au cours de ce processus par le calcul de leur fermeture arrière. Cela préserve l'accessibilité comme le prouve le Lemme 5.3.1. Soit $GrdB(X)$ un langage des gardes d'horloges avec les fermetures en arrière définies par cette grammaire $gb ::= g \mid g \wedge b$ où $g \in Grd(X)$, $b ::= \overleftarrow{g}^{X'} \mid \overleftarrow{gb}^{X'}$ où X' est un sous ensemble des horloges de X.

Définition 19 : Fusion de deux transitions successives

Les ϵ-transitions successives : $t_{1k} = l_1 \xrightarrow{\epsilon, g_1 \wedge g_2 \wedge ... \wedge \overleftarrow{g_{k-1}}^{\overleftarrow{...}^{X_{k-2}}}, X_{k-1}} l_k$ et $t_{kn} = l_k \xrightarrow{\epsilon, g_k \wedge g_{k+1} \wedge ... \wedge \overleftarrow{g_{n-1}}^{\overleftarrow{...}^{X_{n-2}}}, X_{n-1}} l_n$ sont fusionnées en une ϵ-transition $t_{1n} = l_1 \xrightarrow{\epsilon, g_1 \wedge g_2 \wedge ... \wedge g_{k-1} \wedge g_k \wedge g_{k+1} \wedge ... \wedge \overleftarrow{g_{n-1}}^{\overleftarrow{...}^{X_{n-2}}}, X_{n-1}} l_n$ où $k \geqslant 2$ et $n \geqslant k+1$.

Par exemple, pour $k = 2$ et $n = 3$, la fusion des deux ϵ-transitions $l_1 \xrightarrow{\epsilon, g_1, X_1} l_2 \xrightarrow{\epsilon, g_2, X_2} l_3$ donne l'ϵ-transition $l_1 \xrightarrow{\epsilon, g_1 \wedge \overleftarrow{g_2}^{X_1}, X_2} l_3$.

Lemme 5.3.1. Soit t_{1k}, t_{kn} et t_{1n} les transitions définies dans la Def. 19. La localité l_n est atteignable depuis la localité l_1 en appliquant successivement la transition t_{1k} et la transition t_{kn} si et seulement si l_n est atteignable à partir de l_1 en appliquant la transition t_{1n}.

Démonstration. On prouve d'abord l'implication de gauche à droite (le si). On suppose que la localité l_n est atteignable à partir de la localité l_1 par l'application successive de t_{1k} et t_{kn}. Cela signifie qu'il existe une valuation d'horloges v_1 et une succession de délais $t_1, t_2, ..., t_{k-1}, ..., t_{n-2}$ tels que $(l_1, v_1) \rightarrow^{\epsilon} (l_2, v'_1) \rightarrow^{t_1} (l_2, v_2) ... \rightarrow (l_{k-1}, v_{k-1}) \rightarrow^{\epsilon} (l_{k-1}, v'_{k-1}) \rightarrow^{t_{k-1}} (l_k, v_k) ... \rightarrow (l_{n-2}, v_{n-2}) \rightarrow^{\epsilon} (l_{n-1}, v'_{n-2}) \rightarrow^{t_{n-2}} (l_{n-1}, v_{n-1}) \rightarrow^{\epsilon} (l_n, v'_{n-1})$ soit une exécution où

- $v'_1 = v_1[X_1 := 0]$ et $v_2 = v'_1 + t_1$
- ...
- $v'_{k-1} = v_{k-1}[X_{k-1} := 0]$ et $v_k = v'_{k-1} + t_{k-1}$
- ...
- $v'_{n-2} = v_{n-2}[X_{n-2} := 0]$ et $v_{n-1} = v'_{n-2} + t_{n-2}$
- $v'_{n-1} = v_{n-1}[X_{n-1} := 0]$

Avec une telle supposition, la transition t_{1n} est franchissable car les valeurs $v_1, t_1, t_2, ..., t_{n-2}$ rendent la garde de la transition t_{1n} satisfiable. Alors, la localité l_n est atteignable par la même exécution.

Puis, on prouve maintenant l'implication de droite à gauche (le seulement si). On suppose que la localité l_n est atteignable à partir de la localité l_1 par l'application de t_{1n}. Cela signifie qu'il existe une valuation d'horloges v_1 et une succession de délais $t_1, t_2, ..., t_{k-1}, ..., t_{n-2}$ tels que $(l_1, v_1) \rightarrow^{\epsilon} (l_2, v'_1) \rightarrow^{t_1} (l_2, v_2) ... \rightarrow (l_{k-1}, v_{k-1}) \rightarrow^{\epsilon} (l_{k-1}, v'_{k-1}) \rightarrow^{t_{k-1}} (l_k, v_k) ... \rightarrow (l_{n-2}, v_{n-2}) \rightarrow^{\epsilon} (l_{n-1}, v'_{n-2}) \rightarrow^{t_{n-2}} (l_{n-1}, v_{n-1}) \rightarrow^{\epsilon} (l_n, v'_{n-1})$ soit une exécution où

- $v'_1 = v_1[X_1 := 0]$ et $v_2 = v'_1 + t_1$
- ...
- $v'_{k-1} = v_{k-1}[X_{k-1} := 0]$ et $v_k = v'_{k-1} + t_{k-1}$
- ...
- $v'_{n-2} = v_{n-2}[X_{n-2} := 0]$ et $v_{n-1} = v'_{n-2} + t_{n-2}$
- $v'_{n-1} = v_{n-1}[X_{n-1} := 0]$

Avec une telle supposition, la localité l_n est atteignable à partir de la localité l_1 par l'application successive de t_{1k} et t_{kn}. Les transitions t_{1k} et t_{kn} sont franchissables car les valeurs $v_1, t_1, t_2, ..., t_{k-1}$ rendent la garde de la transition t_{1k} satisfiable et les valeurs $v_k, t_{k+1}, ..., t_{n-2}$ rendent la garde de la transition t_{kn} satisfiable. Alors, la localité l_n est atteignable par la même exécution. □

5.3.1/ RÈGLES DE CALCUL D'UN AUTOMATE TEMPORISÉ D'ACCESSIBILITÉ COMPLET

Soit $\langle L, l_0, \Sigma, \Gamma, X, \Delta, F\rangle$ un *TPAIO*. On propose de calculer une représentation de la partie de L qui est son sous-ensemble de localités accessibles. Cette représentation est appelée Automate temporisé d'atteignabilité (*RTA*) d'un *TPAIO*. Un *RTA* est un *TA* avec $\Sigma = \{\epsilon\}$. Une ϵ-transition $l \xrightarrow{\epsilon,g,X'} l'$ est une transition qui atteint la localité l' à partir de la localité l sans modifier le contenu de la pile. On propose dans la Def. 20 la définition de ce *RTA* par application par saturation des règles RA_1 à RA_4. Ces règles sont inspirées des règles de definition d'automate d'accessibilité pour un automate à pile présentées dans l'article [FWW97]. Elles consistent à fusionner en une seule ϵ-transition une séquence de transitions qui ne changent pas le contenu de la pile. Cette ϵ-transition accumule les contraintes d'horloges. La règle RA_1 prend en compte les transitions de lecture d'un symbole. La règle RA_2 fusionne une transition d'empilement suivie d'une transition de dépilement. La règle RA_3 fusionne une transition d'empilement suivie d'une ϵ-transition puis d'une transition de dépilement. La règle RA_4 fusionne deux ϵ-transitions consécutives.

Définition 20 : *RTA* d'un *TPAIO*

Un *RTA* d'un *TPAIO* $\langle L, l_0, \Sigma, \Gamma, X, \Delta, F\rangle$ est un *TA* $\langle L, l_0, \{\epsilon\}, X, \Delta^R, F\rangle$ où $\Delta^R \subseteq L \times \{\epsilon\} \times GrdB(X) \times 2^X \times L$ est une relation satisfaisant les conditions suivantes RA_1 à RA_4. Soit $A \in \Sigma$ et $a \in \Gamma$:

- **RA**$_1$: $l_1 \xrightarrow{\epsilon,g_1,X_1} l_2 \in \Delta^R$ si $l_1 \xrightarrow{A,g_1,X_1} l_2 \in \Delta$
- **RA**$_2$: $l_1 \xrightarrow{\epsilon,g_1 \wedge \overleftarrow{g_2}^{X_1},X_2} l_3 \in \Delta^R$ si $l_1 \xrightarrow{a^+,g_1,X_1} l_2 \in \Delta$ et $l_2 \xrightarrow{a^-,g_2,X_2} l_3 \in \Delta$
- **RA**$_3$: $l_1 \xrightarrow{\epsilon,g_1 \wedge g_2 \wedge g_3 \wedge \ldots \wedge g_{k-1} \wedge \overleftarrow{g_k}^{X_{k-1}},X_k} l_{k+1} \in \Delta^R$ si $l_1 \xrightarrow{a^+,g_1,X_1} l_2 \in \Delta, l_2 \xrightarrow{\epsilon,g_2 \wedge g_3 \wedge \ldots \wedge \overleftarrow{g_{k-1}}^{X_{k-2}},X_{k-1}} l_k \in \Delta^R$ et $l_k \xrightarrow{a^-,g_k,X_k} l_{k+1} \in \Delta$
- **RA**$_4$: $l_1 \xrightarrow{\epsilon,g_1 \wedge g_2 \wedge \ldots \wedge g_{k-1} \wedge g_k \wedge g_{k+1} \wedge \ldots \wedge \overleftarrow{g_{n-1}}^{X_{n-2}},X_{n-1}} l_n \in \Delta^R$ si $l_1 \xrightarrow{\epsilon,g_1 \wedge g_2 \wedge \ldots \wedge \overleftarrow{g_{k-1}}^{X_{k-2}},X_{k-1}} l_k \in \Delta^R$ et $l_k \xrightarrow{\epsilon,g_k \wedge g_{k+1} \wedge \ldots \wedge \overleftarrow{g_{n-1}}^{X_{n-2}},X_{n-1}} l_n \in \Delta^R$

Lemme 5.3.2. Les transitions d'un *RTA* qui sont obtenues par l'application des règles RA_1 à RA_4 sur un *TPAIO* sont franchissables si et seulement si les transitions du *TPAIO* fusionnées sont franchissables.

Démonstration. La preuve est faite par récurrence et pour chaque règle. Notre hypothèse de récurrence est que les transitions d'un *RTA* sont correctes (c'est à dire franchissables si et seulement si les transitions du *TPAIO* qu'elles fusionnent sont franchissables) avant une fusion dans une nouvelle transition. On prouve que les règles RA_1 et RA_2 qui construisent les transitions d'un *RTA* à partir de transitions du *TPAIO* définissent des transitions correctes. Puis, on prouve que les règles RA_3 et RA_4 conservent cette correction sous l'hypothèse que les transitions du *RTA* qu'elles fusionnent sont correctes.

- Cas de la règle RA_1 : $l_1 \xrightarrow{A,g_1,X_1} l_2 \in \Delta$ est franchissable s'il existe une valuation d'horloges v_1 où $v_1 \models g_1$. Donc, la transition $l_1 \xrightarrow{\epsilon,g_1,X_1} 1_2 \in \Delta^R$ est aussi franchissable à partir de v_1.

- Cas de la règle RA_2 : en ce qui concerne les contraintes de pile, les transitions sont successivement franchissables car il est toujours possible de dépiler un symbole a juste après son empilement. En ce qui concerne les contraintes d'horloges, les transitions $l_1 \xrightarrow{a^+,g_1,X_1} l_2 \in \Delta$ et $l_2 \xrightarrow{a^-,g_2,X_2} l_3 \in \Delta$ sont franchissables s'il existe une valuation d'horloges v_1 et un délai t_2 tels que $v_1 \models g_1$ et $v_1[X_1 := 0] + t_2 \models g_2$. C'est exactement les mêmes conditions dans lesquelles la transition $l_1 \xrightarrow{\epsilon, g_1 \wedge \overleftarrow{g_2}^{X_1}, X_2} l_3 \in \Delta^R$ est franchissable, i.e. $v_1 \models g_1 \wedge \overleftarrow{g_2}^{X_1}$. Cette condition est équivalente à $v_1 \models g_1$ et $v_1 \models \overleftarrow{g_2}^{X_1}$. D'après la Def. 8, $v_1 \models \overleftarrow{g_2}^{X_1}$ est satisfiable si $\exists t_2.(t_2 \geqslant 0 \wedge v_1[X_1 := 0] + t_2 \models g_2)$ est satisfiable.
- Cas de la règle RA_3 : les contraintes de la pile sont satisfiables pour les mêmes raisons que dans le cas précédent car d'après la Def. 20 du calcul des ϵ-transitions une ϵ-transition ne change pas l'état de la pile. En ce qui concerne les contraintes d'horloges elles sont aussi satisfiables pour les raisons suivantes : la séquence de trois transitions $l_1 \xrightarrow{a^+,g_1,X_1} l_2 \in \Delta$, $(l_2, \epsilon, g_2 \wedge g_3 \wedge ... \wedge \overleftarrow{g_{k-1}}^{\overleftarrow{...}^{X_{k-2}}}, X_{k-1}, l_k) \in \Delta^R$ et $(l_k, a^-, g_k, X_k, l_{k+1}) \in \Delta$ est franchissable s'il existe $v_1, t_2, ..., t_k$ que :
$v_1 \models g_1$ et $v_2 = v_1[X_1 := 0]$,
$v_2 + t_2 \models g_2$ et $v_3 = (v_2 + t_2)[X_2 := 0]$,
...
$v_{k-1} + t_{k-1} \models g_{k-1}$ et $v_k = (v_{k-1} + t_{k-1})[X_{k-1} := 0]$
$v_k + t_k \models g_k$
La transition $(l_1, \epsilon, g_1 \wedge g_2 \wedge g_3 \wedge ... \wedge g_{k-1} \wedge \overleftarrow{g_k}^{\overleftarrow{...}^{X_{k-1}}}, X_k, l_{k+1}) \in \Delta^R$ est franchissable car sa garde est satisfiable d'après la Def. 8. En effet, si on prend par exemple $k = 3$, la garde $g_1 \wedge g_2 \wedge \overleftarrow{g_3}^{\overleftarrow{X_2}^{X_1}}$ est satisfiable pour une valuation d'horloges v_1 si la condition suivante est satisfiable $v_1 \models g_1 \wedge g_2 \wedge \overleftarrow{g_3}^{\overleftarrow{X_2}^{X_1}}$. En utilisant la Def. 8, on obtient la condition suivante : $v_1 \models g_1 \wedge g_2 \wedge \overleftarrow{g_3}^{\overleftarrow{X_2}^{X_1}} \equiv v_1 \models g_1 \wedge v_1 \models g_2 \wedge \overleftarrow{g_3}^{\overleftarrow{X_2}^{X_1}} \equiv v_1 \models g_1 \wedge \exists t_2.(t_2 \geqslant 0 \wedge v_1[X_1 := 0] + t_2 \models g_2 \wedge \overleftarrow{g_3}^{\overleftarrow{X_2}^{X_1}}) \equiv v_1 \models g_1 \wedge \exists t_2.(t_2 \geqslant 0 \wedge v_1[X_1 := 0] + t_2 \models g_2 \wedge v_1[X_1 := 0] + t_2 \models \overleftarrow{g_3}^{X_2}) \equiv v_1 \models g_1 \wedge \exists t_2.(t_2 \geqslant 0 \wedge v_1[X_1 := 0] + t_2 \models g_2 \wedge \exists t_3.(t_3 \geqslant 0 \wedge (v_1[X_1 := 0] + t_2)[X_2 := 0] + t_3 \models g_3))$. Notre hypothèse est : il existe v_1, t_2, t_3 où $v_1 \models g_1$ et $v_2 = v_1[X_1 := 0]$ et $v_2 + t_2 \models g_2$ et $v_3 = (v_2 + t_2)[X_2 := 0]$ et $v_3 + t_3 \models g_3$. Ce raisonnement se généralise au cas k quelconque.
- Cas de la règle RA_4 : il s'agit d'une conséquence directe du lemme 5.3.1 car la règle RA_4 est la règle de la Def. 19.

□

Remarque 5.3.1. En appliquant ces règles, on obtient un automate temporisé avec un nombre fini de localités, mais avec un nombre potentiellement infini de transitions. Les règles RA_1 et RA_2 ne produisent qu'un nombre fini de transitions pour n'importe quel *RTA* calculé. Par contre, on peut calculer une infinité de transitions en appliquant les règles RA_3 et RA_4 comme illustré dans l'exemple 5.3.1 et l'exemple 5.3.2.

Exemple 5.3.1. Dans la figure 5.4, on présente un exemple de *TPAIO* dont le *RTA* a une infinité de transitions. Pour la règle RA_3, il existe une situation qui amène à une infinité de

transitions dans le *RTA*. Cette situation est $l_1 = l_2$ et $l_k = l_{k+1}$. La figure 5.4.(a) présente un exemple d'un *TPAIO* où cette situation existe. La figure 5.4.(b) présente une partie du *RTA* qui correspond au *TPAIO* présenté dans la figure 5.4.(a). Chaque nouvelle ϵ-transition calculée qui passe de la localité l_0 à la localité l_1 est satisfiable à partir d'une valuation d'horloges quelconque. On peut en construire une infinité en recomposant la dernière ϵ-transition construite avec les transitions réflexives a^+ et a^- respectivement sur les localités l_0 et l_1. On notera qu'elle couvre toujours les mêmes transitions du *TPAIO*.

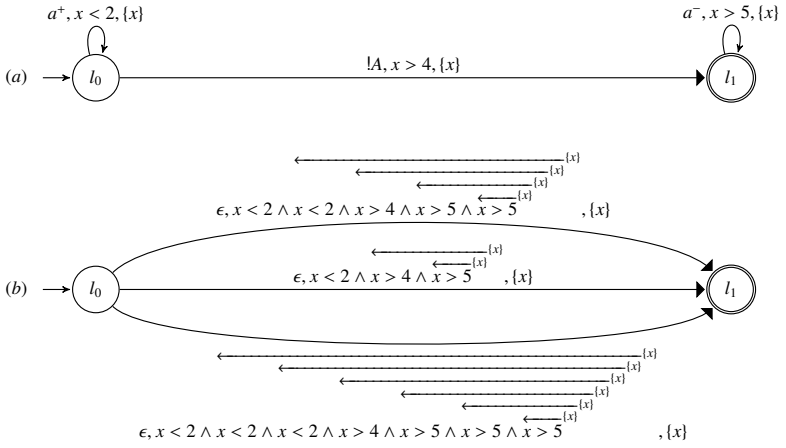


FIGURE 5.4 – Un exemple d'applicabilité infinie de la règle RA_3

Exemple 5.3.2. Dans, la Fig. 5.5, on présente un deuxième exemple de *TPAIO* dont le *RTA* a une infinité de transitions par application de la règle RA_4. Pour cette règle, il existe deux situations qui amènent à une infinité de transitions dans le *RTA*. Les deux possibilités sont $l_1 = l_k$ ou $l_k = l_n$. La Fig. 5.5.(a) présente un exemple d'un tel *TPAIO*. La figure 5.5.(b) présente une partie du *RTA* du*TPAIO* présenté dans la Fig. 5.5.(a) en appliquant deux fois la règle RA_4 dans le cas de $l_k = l_n$. On peut construire une infinité d'ϵ-transitions en recomposant la dernière ϵ-transition construite avec l'ϵ-transition réflexive sur la localité l_1. On notera qu'elle couvre toujours les mêmes transitions de son *TPAIO*.

Dans ce mémoire, nous proposons l'algorithme 5.3.1 qui applique un nombre fini de fois les règles RA_1 à RA_4 et produit un *RTA* fini mais partiel.

5.3.2/ ALGORITHME DE CALCUL D'UN *RTA* FINI ET INCOMPLET.

Dans cette section, on présente une méthode polynomiale qui nous permet de construire un *RTA* fini. On propose un algorithme qui applique un nombre fini de fois les règles RA_1 à RA_4 pour le calcul d'un *RTA* fini à partir d'un *TPAIO*. Cet algorithme est inspiré de l'algorithme de Finkel *et al* pour les *PA* présenté dans [FWW97] et qui calcule les ϵ-transitions d'un automate à pile donné. Les modifications que nous avons apportées à l'algorithme de [FWW97] sont les suivantes :

1. On ne calcule pas seulement les ϵ-transitions, mais on mémorise aussi les chemins qu'elles fusionnent. Toute ϵ-transition du *RTA* correspond à un ou plusieurs chemins du *TPAIO*.

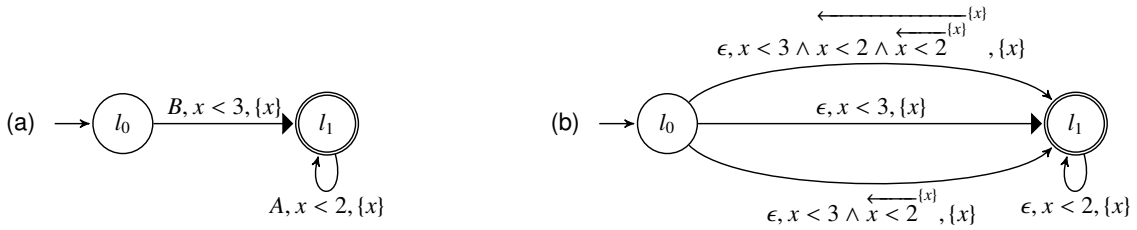


FIGURE 5.5 – Un exemple d'applicabilité infinie de la règle RA_4

2. Étant donné que le problème abordé dans [FWW97] est de calculer l'ensemble des localités atteignables dans un automate à pile, leur algorithme ne calcule qu'une seule ϵ-transition entre deux localités l et l', car l'ajout d'une autre ne changerait pas l'atteignabilité. Dans notre cas, on ajoute une nouvelle ϵ-transition entre deux localités déjà reliées par une ϵ-transition à chaque fois que le chemin associé couvre une nouvelle transition du *TPAIO*.

3. On ajoute la règle RA_1 pour prendre en compte les actions de Σ, en plus de Finkel *et al* qui ne traitent que le cas d'automates avec seulement des actions de pile.

4. Les ϵ-transitions réflexives ne sont pas utilisées dans [FWW97] pour prolonger (sur leur droite ou sur leur gauche) les ϵ-transitions existantes, parce qu'elles ne changent rien à l'atteignabilité des états. Par exemple, si on a deux transitions (l, ϵ, l) et (l, ϵ, l'), alors la localité l' est atteignable à partir de l et on n'a pas besoin de fusionner ces deux transitions pour conclure que l' est atteignable à partir de la localité l. Dans notre cas, on ne peut pas les ignorer en raison de leurs contraintes d'horloge. On verra que, par exemple un chemin $l \xrightarrow{\epsilon} l'$ peut ne pas être exécutable, alors qu'un chemin $l \xrightarrow{\epsilon} l \xrightarrow{\epsilon} l'$ l'est car l'exécutabilité dépend des contraintes d'horloges.

Le principe de l'algorithme est de fusionner en premier en une seule ϵ-transition une séquence d'une transition d'empilement suivie d'une transition de dépilement du même symbole, et ensuite, de fusionner ces ϵ-transitions entre elles ou avec des transitions d'empilement et de dépilement pour construire des nouvelles ϵ-transitions. On adapte l'algorithme de Finkel *et al* au cas d'un *TPAIO*. Les ϵ-transitions sont fusionnées de la même façon que dans [FWW97] pour les contraintes de pile, mais on a adapté les règles pour tenir compte des contraintes d'horloge et on calcule les gardes des transitions fusionnées. L'évaluation de satisfiabilité de ces gardes est reportée à une deuxième phase, lorsque toutes les fusions auront été réalisées. En outre, notre algorithme calcule une table des chemins, qui associe chaque transition du *RTA* avec un ou plusieurs chemins du *TPAIO*. Cette table des chemins servira à engendrer des tests couvrant toutes les transitions dans le meilleur des cas.

Notre algorithme 5.3.1 calcule les transitions de Δ^R dans la table *paths*. Son entrée est un *TPAIO*. Il retourne une table de chemins qui associe à chaque transition du *RTA* un ensemble de chemins du *TPAIO*. Pour présenter cet algorithme, on a défini le type **PATHS** = Seq(Δ) qui est une séquence de transitions du *TPAIO*, et un type **PATH_TABLES** =$\Delta^R \rightarrowtail$ ensemble de $PATHS$ qui est une fonction totale bijective qui associe un ensemble de chemins à chaque transition du *RTA*. Cet algorithme calcule les fermetures transitives des ϵ-transitions seulement en enregistrant l'information dans

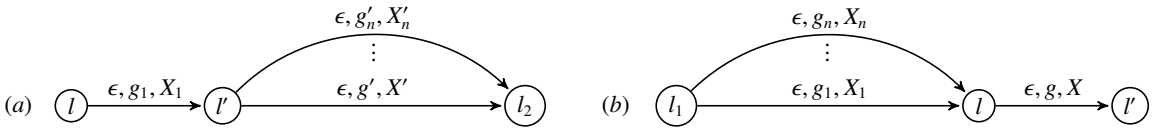


FIGURE 5.6 – Possibilités de calcul d'une nouvelle ϵ-transition

les structures de données C_Direct et C_Trans. Il énumère toutes les paires possibles de localités. Il cherche ensuite si elles peuvent être exploitées afin de construire une nouvelle ϵ-transition dans le *RTA*.

L'algorithme est divisé en deux étapes : une étape d'initialisation de la ligne 1 à 16, et une étape de calcul de la ligne 17 à 46. Une variable stack est utilisée pour enregistrer les ϵ-transitions qui ont été calculées, mais qui ne sont pas encore exploitées en conjonction avec des autres transitions pour construire d'autres ϵ-transitions. Elle est initialisée de la ligne 1 à la ligne 3 en empilant $(l, l', [l, A, g, X', l'])$ où $A \in \Sigma_{in} \cup \Sigma_{out}$, et de la ligne 4 à 6 en empilant $(l, l', \emptyset)$. Les structures de C_Direct et C_Trans sont initialement vides (lignes 7-9). La structure C_Direct est utilisée pour appliquer les règles RA_2 et RA_3. Pour chaque paire (l, l'), l'ensemble $C\_Direct(l, l')$ est initialisé avec une séquence de deux transitions : une transition d'empilement a^+ et une transition de dépilement a^- (lignes 10-12). La structure C_Trans est utilisée pour appliquer la règle RA_4. Elle est initialisée de la ligne 13 à 16.

Dans la deuxième étape (lignes 17-46), l'algorithme traite toutes les transitions de $stack$ et les exploite à l'aide de C_Direct et C_Trans. Pour chaque nouvelle ϵ-transition et son chemin π entre deux localités l et l', cet algorithme vérifie les deux possibilités suivantes pour calculer d'autres ϵ-transitions :

- en utilisant $C\_Direct((l, l'))$ (ligne 33-35) : pour chaque élément $(l_1, l_2, [(l_1, a^+, g_1, X_1, l), (l', a^-, g', X', l_2)])$ dans $C\_Direct((l, l'))$, notre algorithme ajoute une nouvelle ϵ-transition entre l_1 et l_2 dont le chemin est $[(l_1, a^+, g_1, X_1, l)\hat{}\pi\hat{}(l', a^-, g', X', l_2)]$ où $t\hat{}\pi$ dénote la concaténation d'une transition t avec le chemin π qui est un chemin entre l et l'.
- en utilisant $C\_Trans((l, l'))$ (ligne 36-44) : pour chaque élément (l_1, l_2, l_3, l_4) dans $C\_Trans((l, l'))$: il existe deux cas pour l'ajout de nouvelles ϵ–transitions :
  - — $l_1 = l'$: pour chaque ϵ–transition entre l' et l_2, une ϵ-transition est ajoutée entre l et l_2 comme illustré dans la Fig. 5.6(a), par prolongement à gauche.
  - — $l_2 = l$: pour chaque ϵ–transition entre l_1 et l, une ϵ-transition est ajoutée entre l_1 et l' comme illustré dans Fig. 5.6(b), par prolongement à droite.

Notre algorithme 5.3.1 utilise les fonctions suivantes : (i) $MergeTransitions(\pi)$ qui retourne une ϵ-transition et fusionne une séquence de transitions consécutives π en une seule ϵ-transition, (ii) $isNewTransitions(\pi, \Delta)$ qui vérifie si la séquence de transitions π contient une transition qui n'est pas dans Δ et (iii) $getPaths(l, l', paths)$ qui retourne un ensemble de chemins dans *paths* qui va de l à l'. Pour assurer la terminaison de notre algorithme, l'ajout d'une nouvelle ϵ-transition temporisée (l, ϵ, g, X, l') n'est possible que si son chemin couvre une nouvelle transition dans le *TPAIO*. Notre algorithme utilise la structure de données $coveredTransitions$ qui associe un ensemble de transitions à une paire de localités. Pour chaque nouvelle ϵ-transition entre deux localités l et l', $coveredTransitions$ enregistre les nouvelles transitions couvertes entre l et l' (lines 29-32).

Algorithme 5.3.1 Algorithme de calcul d'un *RTA* fini et partiel

ENTRÉE : Un *TPAIO* $\langle L, l_0, \Sigma, \Gamma, X, \Delta, F\rangle$
SORTIE : $paths \in PATH\_TABLE$
VARIABLES : $stack \in$ pile de $L\times L\times PATHS$; $C\_Direct \in L\times L \rightarrowtail$ ensemble d'élément de $L\times L\times PATHS$; $C\_Trans \in L\times L \rightarrowtail$ ensemble d'élément de $L\times L\times L\times L$; $l, l', l_1, l_2, l_3, l_4 \in L$; $\pi, \pi_1 \in PATHS$; $tr \in \Delta^R$; $t \in \Delta$; $coveredTransitions \in L\times L \rightarrowtail$ ensemble de transitions de Δ
DÉBUT
1: **pour** chaque transition $(l, A, g, X, l') \in \Delta$ où $A \in \Sigma$ **faire**
2: empiler $(l, l', [(l, A, g, X, l')])$ dans $stack$ // implémente la règle RA_1
3: **fin pour**
4: **pour** chaque localité $l \in L$ **faire**
5: empiler $(l, l, \emptyset)$ dans $stack$
6: **fin pour**
7: **pour** chaque couple $(l, l') \in (L \times L)$ **faire**
8: $C\_Direct(l, l') \leftarrow \emptyset$; $C\_Trans(l, l') \leftarrow \emptyset$; $coveredTransitions(l, l') \leftarrow \emptyset$
9: **fin pour**
10: **pour** chaque couple $(l_1 \xrightarrow{a^+, g_1, X_1} l_2, l_3 \xrightarrow{a^-, g_3, X_3} l_4) \in (\Delta \times \Delta)$ où $a \in \Gamma$ **faire**
11: $C\_Direct(l_2, l_3) \leftarrow C\_Direct(l_2, l_3) \cup \{((l_1, l_4), [(l_1, a^+, g_1, X_1, l_2), (l_3, a^-, g_3, X_3, l_4)])\}$
12: **fin pour**
13: **pour** chaque triplet $(l, l', l'') \in (L \times L \times L)$ **faire**
14: $C\_Trans(l, l') \leftarrow C\_Trans(l, l') \cup \{(l', l'', l, l''))\}$
15: $C\_Trans(l, l') \leftarrow C\_Trans(l, l') \cup \{(l'', l, l'', l'))\}$
16: **fin pour**
17: **tant que** $stack \neq emptyStack$ **faire**
18: $(l, l', \pi) \leftarrow$ dépiler$(stack)$
19: $tr \leftarrow MergeTransitions(\pi)$
20: **si** $isNewTransitions(\pi, coveredTransitions((l, l')))$ **alors**
21: **si** $\pi \neq []$ **alors**
22: **si** $tr \notin dom(paths)$ **alors**
23: $paths(tr) \leftarrow \{\pi\}$
24: **sinon**
25: $paths(tr) \leftarrow paths(tr) \cup \{\pi\}$
26: **fin si**
27: **fin si**
28: **pour** t dans π **faire**
29: **si** $t \notin coveredTransitions((l, l'))$ **alors**
30: $coveredTransitions((l, l')) \leftarrow coveredTransitions((l, l')) \cup \{t\}$
31: **fin si**
32: **fin pour**
33: **pour** $(l_1, l_2, [(l_1, a^+, g_1, X_1, l), (l', a^-, g', X', l_2)])$ in $C\_Direct((l, l'))$ **faire**
34: empiler $(l_1, l_2, (l_1, a^+, g_1, X_1, l)\hat{}\pi\hat{}(l', a^-, g', X', l_2)$ dans $stack$ // $t\hat{}\pi$ note la concaténation de t et π
35: **fin pour**
36: **pour** (l_1, l_2, l_3, l_4) in $C\_Trans((l, l'))$ **faire**
37: **pour** chaque π_1 dans $getPaths(l_1, l_2, paths)$ **faire**
38: **si** $l' = l_1$ **alors**
39: empiler $(l, l_2, \pi\hat{}\pi_1)$ dans $stack$
40: **sinon**
41: empiler $(l_1, l', \pi_1\hat{}\pi)$ dans $stack$
42: **fin si**
43: **fin pour**
44: **fin pour**
45: **fin si**
46: **fin tant que**
FIN.

Notre algorithme est polynomial. L'algorithme de [FWW97] a une complexité $O(n^3)$ où n est le nombre de localités du *PA*. Un *TPAIO* contient des transitions d'entrées et de sorties que notre algorithme traite différemment que dans [FWW97]. La terminaison de notre algorithme dépend du nombre de transitions entre deux localités.

L'exemple 5.3.3 illustre l'application de cet algorithme sur le *TPAIO* de la Fig. 5.2.

Exemple 5.3.3. La table 5.1 montre toutes les transitions et les chemins associés partant

de la localité initiale l_0 et arrivant dans une localité finale l_3, l_7 ou l_9 du *RTA* du *TPAIO* de la Fig. 5.2. Une ϵ-transition peut être associée à plusieurs chemins. C'est le cas de l'ϵ-transition qui va de l_0 à l_7 et qui est associée à deux chemins : le deuxième chemin couvre les transitions $(l_5, E, x \leqslant 1, \emptyset, l_6)$ et $(l_6, F, x \leqslant 3, \{x\}, l_7)$ indiquées en gras, qui ne sont pas couvertes par le premier chemin du *TPAIO*. Le *RTA* a seulement des ϵ-transitions, donc, chaque chemin fait autant de dépilement que d'empilement pour passer d'une localité à une autre sans changer l'état de la pile. Les chemins font le nombre de cycle minimal pour satisfaire les contraintes de pile et la couverture de toutes les transitions. Par exemple, les deux chemins entre l_0 et l_7 couvrent toutes les transitions du *TPAIO*. Donc, on ne mémorise pas un troisième chemin entre l_0 et l_7. Les gardes nommées g_0, g_1, et g_2 dans la table 5.1 sont exprimées en fonction des gardes nommées g'_1 et g'_2. Elles sont les suivantes :

$$g_0 \overset{def}{=} x \leqslant 2 \wedge x \leqslant 1 \wedge \overleftarrow{x \leqslant 2}^{\emptyset}{}^{\{x\}}, \quad g'_1 \overset{def}{=} x \leqslant 2 \wedge x \leqslant 1 \wedge \overleftarrow{x \leqslant 3}^{\emptyset}{}^{\{x\}},$$

$$g_1 \overset{def}{=} x \leqslant 2 \wedge x \leqslant 1 \wedge x \leqslant 3 \wedge x \leqslant 2 \wedge x \leqslant 1 \wedge x \leqslant 2 \wedge \overleftarrow{g'_1}^{\emptyset}{}^{\{x\}}{}^{\emptyset}{}^{\{x\}}{}^{\emptyset}{}^{\{x\}},$$

$$g'_2 \overset{def}{=} x \leqslant 2 \wedge x \leqslant 1 \wedge x \leqslant 2 \wedge x \leqslant 2 \wedge x \leqslant 1 \wedge x \leqslant 3 \wedge x \leqslant 2 \wedge \overleftarrow{g'_1}^{\emptyset}{}^{\{x\}}{}^{\emptyset}{}^{\{x\}}{}^{\emptyset}{}^{\{x\}}{}^{\emptyset}{}^{\{x\}},$$

$$g_2 \overset{def}{=} x \leqslant 2 \wedge x \leqslant 1 \wedge x \leqslant 3 \wedge x \leqslant 2 \wedge x \leqslant 1 \wedge x \leqslant 3 \wedge \overleftarrow{g'_2}^{\emptyset}{}^{\{x\}}{}^{\emptyset}{}^{\{x\}}{}^{\emptyset}{}^{\{x\}}{}^{\emptyset}{}^{\{x\}}$$

Transitions	Chemins associés
$(l_0, \epsilon, g_0, \{x\}, l_3)$	$l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{B,x\leqslant 1} l_2 \xrightarrow{D,x\leqslant 2,\{x\}} l_3$
$(l_0, \epsilon, g_1, \{x\}, l_7)$	$l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{C,x\leqslant 1} l_4 \xrightarrow{pow^+,x\leqslant 3,\{x\}} l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{B,x\leqslant 1} l_2 \xrightarrow{D,x\leqslant 2,\{x\}} l_3 \xrightarrow{pow^-,x\leqslant 2,\{x\}} l_5 \xrightarrow{E,x\leqslant 1} l_6 \xrightarrow{F,x\leqslant 3,\{x\}} l_7$
$(l_0, \epsilon, g_1, \{x\}, l_9)$	$l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{C,x\leqslant 1} l_4 \xrightarrow{pow^+,x\leqslant 3,\{x\}} l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{B,x\leqslant 1} l_2 \xrightarrow{D,x\leqslant 2,\{x\}} l_3 \xrightarrow{pow^-,x\leqslant 2,\{x\}} l_5 \xrightarrow{G,x\leqslant 1} l_8 \xrightarrow{H,x\leqslant 3,\{x\}} l_9$
$(l_0, \epsilon, g_2, \{x\}, l_7)$	$l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{C,x\leqslant 1} l_4 \xrightarrow{pow^+,x\leqslant 3,\{x\}} l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{C,x\leqslant 1} l_4 \xrightarrow{pow^+,x\leqslant 3,\{x\}} l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{B,x\leqslant 1} l_2 \xrightarrow{D,x\leqslant 2,\{x\}} l_3 \xrightarrow{pow^-,x\leqslant 2,\{x\}} l_5 \xrightarrow{G,x\leqslant 1} l_8 \xrightarrow{H,x\leqslant 3,\{x\}} l_9 \xrightarrow{pow^-,x\leqslant 2,\{x\}} l_5 \xrightarrow{E,x\leqslant 1} l_6 \xrightarrow{F,x\leqslant 3,\{x\}} l_7$
	$l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{C,x\leqslant 1} l_4 \xrightarrow{pow^+,x\leqslant 3,\{x\}} l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{C,x\leqslant 1} l_4 \xrightarrow{pow^+,x\leqslant 3,\{x\}} l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{B,x\leqslant 1} l_2 \xrightarrow{D,x\leqslant 2,\{x\}} l_3 \xrightarrow{pow^-,x\leqslant 2,\{x\}} l_5 \xrightarrow{\mathbf{E,x\leqslant 1}} l_6 \xrightarrow{\mathbf{F,x\leqslant 3,\{x\}}} l_7 \xrightarrow{pow^-,x\leqslant 2,\{x\}} l_5 \xrightarrow{E,x\leqslant 1} l_6 \xrightarrow{F,x\leqslant 3,\{x\}} l_7$
$(l_0, \epsilon, g_2, \{x\}, l_9)$	$l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{C,x\leqslant 1} l_4 \xrightarrow{pow^+,x\leqslant 3,\{x\}} l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{C,x\leqslant 1} l_4 \xrightarrow{pow^+,x\leqslant 3,\{x\}} l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{B,x\leqslant 1} l_2 \xrightarrow{D,x\leqslant 2,\{x\}} l_3 \xrightarrow{pow^-,x\leqslant 2,\{x\}} l_5 \xrightarrow{G,x\leqslant 1} l_8 \xrightarrow{H,x\leqslant 3,\{x\}} l_9 \xrightarrow{pow^-,x\leqslant 2,\{x\}} l_5 \xrightarrow{G,x\leqslant 1} l_8 \xrightarrow{H,x\leqslant 3,\{x\}} l_9$
	$l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{C,x\leqslant 1} l_4 \xrightarrow{pow^+,x\leqslant 3,\{x\}} l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{C,x\leqslant 1} l_4 \xrightarrow{pow^+,x\leqslant 3,\{x\}} l_0 \xrightarrow{A,x\leqslant 2,\{x\}} l_1 \xrightarrow{B,x\leqslant 1} l_2 \xrightarrow{D,x\leqslant 2,\{x\}} l_3 \xrightarrow{pow^-,x\leqslant 2,\{x\}} l_5 \xrightarrow{\mathbf{E,x\leqslant 1}} l_6 \xrightarrow{\mathbf{F,x\leqslant 3,\{x\}}} l_7 \xrightarrow{pow^-,x\leqslant 2,\{x\}} l_5 \xrightarrow{G,x\leqslant 1} l_8 \xrightarrow{H,x\leqslant 3,\{x\}} l_9$

TABLE 5.1 – Table de chemins, pour les transitions partant de l_0 vers une localité finale du *RTA* du *TPAIO* de la Fig. 5.2, avec leurs chemins dans le *TPAIO*

5.4/ GÉNÉRATION D'UN ARBRE DE TESTS

On propose de sélectionner les exécutions qui atteignent une localité finale avec une pile vide, pour produire un ensemble de cas de tests correspondant aux exécutions nominales de la spécification. Pour cela, on sélectionne les ϵ-transitions qui vont d'une localité initiale à une localité finale. La garde d'une ϵ-transition avec fermeture en arrière est exprimée par un système d'inégalités linéaires sur des nombres réels. Par exemple, dans la table 5.1, la garde de la transition $l_0 \xrightarrow{\epsilon,g_0,\{x\}} l_3$ est exprimée par la formule suivante $\exists(t_1,t_2,t_3) \in \mathbb{R}^3.\ t_1 \leqslant 2 \wedge t_2 \leqslant 1 \wedge t_2 + t_3 \leqslant 2$. Pour vérifier la satisfiabilité d'une garde d'une transition dans un *RTA*, on a besoin d'un outil d'évaluation de satisfiabilité de formules logiques sur les réels qui nous permet de vérifier si cette garde est satisfiable ou non, comme par exemple l'outil Z3 [dMB08]. En utilisant les chemins de toutes les ϵ-transitions avec garde satisfiable qui va d'une localité initiale à une localité finale, on construit un ensemble de chemins de test. Cet ensemble est obtenu en ajoutant à chaque localité un contenu de la pile tel qu'il est défini dans la Def. 21.

Définition 21 : Ensemble de chemins de tests d'un *TPAIO* déterministe avec deadline *lazy*

Soit $T = \langle L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F\rangle$ un *TPAIO* qui est une spécification, soit $R = \langle L, l_0, \{\epsilon\}, X, \Delta^R, F\rangle$ le *RTA* de T et $paths$ la table des chemins qui associe chaque transition de R avec un ou plusieurs chemins de T. Un ensemble de chemins de test ST est défini par : $(l_0,\epsilon) \xrightarrow{act_0,g_0,X_0} (l_1,p_1) \xrightarrow{act_1,g_1,\overleftarrow{X_0}} \ldots \xrightarrow{act_n,g_n,\overleftarrow{\ldots}\overleftarrow{X_0}} (l_{n+1},\epsilon)$ est un chemin de test si la transition $l_0 \xrightarrow{\epsilon,g_0\wedge g_1\wedge\ldots\wedge\overleftarrow{\ldots}\overleftarrow{g_n}\overleftarrow{X_{n-1}},X_n} l_{n+1}$ existe dans Δ^R, $l_0 \xrightarrow{act_0,g_0,X_0} l_1 \xrightarrow{act_1,g_1,\overleftarrow{X_0}} \ldots \xrightarrow{act_n,g_n,\overleftarrow{\ldots}\overleftarrow{X_0}} l_{n+1}$ est un chemin associé à cette transition dans la table $paths$, sa garde $g_0 \wedge g_1 \wedge \ldots \wedge \overleftarrow{\ldots}\overleftarrow{g_n}\overleftarrow{X_{n-1}}$ est satisfiable, $l_{n+1} \in F$ et pour i de 1 à n, on applique une des règles suivantes pour définir p_i :

- $p_i = p_{i+1}$ si et $act_i \in \Sigma$,
- $p_{i+1} = p_i.act_i$ si $act_i \in \Gamma^+$,
- $p_i = p_{i+1}.act_i$ si $act_i \in \Gamma^-$.

Les modèles communiquent avec leur environnement via leurs actions d'entrées et de sorties. Comme l'environnement envoie des entrées à l'implémentation, chaque cas de test est intégré dans une structure d'arbre pour être en mesure de réagir à toute action de sortie qu'il reçoit. L'arbre de cas de tests TC est défini par une structure de données de type $trie$ appelée TC contenant un ensemble de chemins de tests ST. Nous détaillons dans les algorithmes 5.4.1 et 5.4.2 comment le trie est calculé à partir de ST. Ses nœuds sont des couples formés d'une localité et d'un contenu de pile. Les verdicts sont ensuite insérés dans TC en remplaçant ses feuilles par des feuilles $pass$, et en ajoutant les transitions qui atteignent $fail$ et $inconc$ à partir de n'importe quel nœud différent de $pass$ de TC. L'algorithme 5.4.3 détaille l'insertion de ces verdicts dans TC. Les cas de tests obtenus visent à couvrir toutes les localités et toutes les transitions du *TPAIO*, et il n'est pas toujours possible d'énumérer tous les chemins de la spécification. En conséquence, il est possible d'observer un comportement qui est correct dans la spécification mais qui n'existe par dans l'arbre de tests (il n'est pas couvert). Pour cette raison, on propose d'ajouter des transitions qui mènent au verdict $inconc$. Le critère d'arrêt pour la construction du *RA* est la couverture de toutes les transitions entre les localités, car notre méthode vise à

couvrir toutes les transitions et toutes les localités du *TPAIO*. Il n'est pas toujours possible d'énumérer tous les chemins possibles de la spécification en appliquant ce critère de couverture. Pour tenir compte de l'observation d'une action autorisée dans la spécification et pas dans le cas de test, on propose d'arrêter l'exécution du cas de test en atteignant le verdict $inconc$. Donc, on propose d'ajouter des transitions dans l'arbre de tests qui mènent à $inconc$. Ce sont des transitions spécifiées et qui n'existent pas dans le cas de tests calculé. Ces transitions prennent en compte les contraintes de pile. On ne peut pas avoir de transition de dépilement qui mène à $inconc$ et qui dépileraient un symbole qui n'existe pas au sommet de la pile.

Définition 22 : Arbre de cas de tests

Soit ST un ensemble de chemins de tests d'une spécification T et $T^T = \langle L \cup \{fail\} \cup \{inconc\}, l_0, \Sigma^T, \Gamma, X, \overline{\Delta} \cup \Delta_{\rightarrow fail} \cup \Delta_{\rightarrow inconc}, F \rangle$ le *TPTIO* dont il est issu. L'arbre de cas de tests de ST est un trie $TC = \langle N, E, w \rangle$ où N est un ensemble de nœuds, $E \subseteq N \times N$ est un ensemble d'arcs entre les nœuds et $w : V \times V \rightarrow (\Sigma^T \cup \Gamma^{+-}) \times Grd(X) \times X$ est un étiquetage des transitions. On note $Feuille(N)$ l'ensemble des feuilles dans N. TC est défini à partir du trie $TC' = \langle N', E', w' \rangle$ qui est l'ensemble de chemins de tests ST où les nœuds sont les source et cible des transitions du ST , i.e. des couples formés d'une localité de L et d'un contenu de pile (mot de Γ^*), les transitions sont les transitions de ST et les labels sont les étiquettes des transitions de ST. TC est défini à partir de TC' comme suit :

- $E = E' \setminus \{n \xrightarrow{act,g,d,X'} n' \mid n \xrightarrow{act,g,d,X'} n' \in E' \text{ et } n' \in Feuille(N')\} \cup \{n \xrightarrow{act,g,d,X'} pass \mid \exists n' \cdot (n \xrightarrow{act,g,d,X'} n' \in E') \text{ et } n' \in Feuille(N')\} \cup E_{fail} \cup E_{inconc}$ où $E_{fail} = \{(l,p) \xrightarrow{act,g,d,X'} fail \mid l \xrightarrow{act,g,d,X'} fail \in \Delta^T\}$ et $E_{inconc} = \{(l,p) \xrightarrow{act,g,d,X'} inconc \mid l \xrightarrow{act,g,d,X'} inconc \in \Delta^T\} \cup \{(l,p) \xrightarrow{act,g,d,X'} inconc \mid \exists l'.(l' \in L \wedge l \xrightarrow{act,g,d,X'} l' \in \Delta^T \wedge act \in \Sigma^T \cup \Gamma^+ \wedge \neg\exists(l',p').((l,p) \xrightarrow{act,g,d,X'} (l',p') \in E)\} \cup \{(l,p) \xrightarrow{act,g,d,X'} inconc \mid \exists l'.(l' \in L \wedge l \xrightarrow{a^-,g,d,X'} l' \in \Delta^T \wedge a \in \Gamma \wedge \neg\exists(l',p').((l,p) \xrightarrow{a^-,g,d,X'} (l',p') \in E) \wedge \exists p'.(p' \in \Gamma^* \wedge p = p'.a)\}$

- $N = N' \setminus Feuille(N') \cup \{pass_1, \ldots, pass_n \mid n = card(Feuille(N'))\} \cup \{fail_1, \ldots, fail_m \mid m = card(E_{fail})\} \cup \{inconc_1, \ldots, inconc_k \mid k = card(E_{inconc})\}$

- $w = w' \setminus \{n, n', a, g, X') \mid n \rightarrow n' \in E' \wedge n' \in Feuille(N')\} \cup \{(n, pass, a, g, X') \mid \exists n'.(n \rightarrow n' \wedge w(n,n' = a, g, X')\} \cup \{((l,p), fail, a, g, X') \mid \exists d.(l \xrightarrow{act,g,d,X'} fail \in \Delta^T)\} \cup \{((l,p), inconc, a, g, X') \mid \exists d.(l \xrightarrow{act,g,d,X'} inconc \in \Delta^T)\} \cup \{((l,p), inconc, act, g, d, X') \mid \exists l'.(l' \in L \wedge l \xrightarrow{act,g,d,X'} l' \in \Delta^T \wedge act \in \Sigma^T \cup \Gamma^+ \wedge \neg\exists(l',p').((l,p) \xrightarrow{act,g,d,X'} (l',p') \in E)\} \cup \{(l,p), inconc, a^-, g, d, X') \mid \exists l'.(l' \in L \wedge l \xrightarrow{a^-,g,d,X'} l' \in \Delta^T \wedge a \in \Gamma \wedge \neg\exists(l',p').((l,p) \xrightarrow{a^-,g,d,X'} (l',p') \in E) \wedge \exists p'.(p' \in \Gamma^* \wedge p = p'.a)\}$

Le résultat est un arbre de tests, dans lequel les actions sont soit observables (les actions de la pile Γ^{+-} et les actions de sortie de Σ_{out}), soit contrôlables (les actions d'entrée de Σ_{in}). Les feuilles de cet arbre différentes de fail et de inconc sont remplacées par des feuilles de verdict pass. La figure 5.7 montre un arbre de tests qui présente les sept chemins correspondant aux cinq ϵ-transitions suivantes : $(l_0, \epsilon, g_0, \emptyset, l_3)$, $(l_0, \epsilon, g_1, \emptyset, l_7)$, $(l_0, \epsilon, g_1, \emptyset, l_9)$,

$(l_0, \epsilon, g_2, \emptyset, l_7)$ et $(l_0, \epsilon, g_2, \emptyset, l_9)$ dont les chemins sont présentés dans la table 5.1. On trouve par exemple la transition $(l_1, ?C, x \leqslant 1, \emptyset, inconc)$ car l'observation de C est spécifiée mais l'action C n'existe pas dans la suite de tests à cause du critère de couverture.

Algorithme 5.4.1 Trie creerTrie(ST)- Transformation d'un ensemble de chemins de test en Trie

ENTRÉES: ST un ensemble de chemins de tests
SORTIES: $TC = \langle N, E, w \rangle$ le trie de ST

```
Trie TC ← (creerNœud(l_0^R, ϵ), ∅, ∅)
pour chaque π ∈ ST faire
    TC ← insererChemin(π, TC)
fin pour
// ajout des transitions qui mènent à inconc
pour chaque n_s = (l, p) ∈ N s.t. n_s ≠ n_pass et n_s ≠ n_fail faire
    pour chaque (l, act, g, X', l') ∈ Δ faire
        si ¬∃n_t.(n_t ∈ N s.t. (n_s, n_t) ∈ E et w(n_s, n_t) = act, g, X') et ¬(act = a⁻ et ∀p'.(p' ∈ Γ* ⇒
        p ≠ p'.a)) ) alors
            Nœud n_inconc ← creerNœud(inconc)
            N ← N ∪ {n_inconc}
            E ← E ∪ {(n_s, n_inconc)}
            w(n_s, n_inconc) ← act, g, X'
        fin si
    fin pour
fin pour
retourner TC
```

Algorithme 5.4.2 Trie insererChemin(p, TC)-Ajout d'un chemin dans un Trie

ENTRÉES: $\pi = (l_0, \epsilon) \xrightarrow{act_0, g_0, X_0} (l_1, p_1) \xrightarrow{act_1, g_1, X_1} (l_2, p_2) \ldots \xrightarrow{act_n, g_n, X_n} (l_{n+1}, \epsilon)$ un chemin de ST ; $TC = \langle N, E, w \rangle$ un trie
SORTIES: le trie TC enrichi avec le chemin π

```
// parcours du préfixe de π commun à un chemin de TC
Nœud n_s ← racine(TC)
int i ← 0
tant que i < n + 1 et ∃n_t ∈ N s.t. (n_s, n_t) ∈ E et w(n_s, n_t) = act_i, g_i, X' faire
    n_s ← n_t
    i ← i + 1
fin tant que
// ajout de suffixe de π qui n'est pas commun
pour int k = i à n pas de 1 faire
    Nœud n_t ← creerNœud(l_{k+1}, p_{k+1})
    N ← N ∪ {n_t}
    E ← E ∪ {(n_s, n_t)}
    w(n_s, n_t) ← act_k, g_k, {y}
    n_s ← n_t
fin pour
retourner TC
```

Le trie d'un ensemble de chemins de tests est l'arbre préfixe d'un ensemble de chemins

de tests. Il est défini dans la forme d'un triplet $\langle N, E, w\rangle$ (voir la Def. 22). Le nœud $n \in N$ est un couple (l, p) où $l \in L$ et $p \in \Gamma^*$. Un arc de E est un couple $(n_s, n_t) \in N \times N$. La fonction $w : V \times V \to (\Sigma^T \cup \Gamma^{+-}) \times Grd(y) \times \{y\}$ étiquette les arcs.

L'algorithme 5.4.2 est utilisé pour insérer un chemin de test $\pi \in ST$ dans le trie $TC = \langle N, E, w\rangle$. L'algorithme 5.4.1 crée un trie d'un ensemble de chemins de tests ST. L'algorithme 5.4.3 ajoute les verdicts pass, fail et inconc au trie d'un ensemble de chemins de tests.

Algorithme 5.4.3 Trie ajouterVerdicts(TC, T^T)-Ajout des verdicts dans un Trie

---

ENTRÉES: $T = \langle L, l_0, \Sigma, \Gamma, X, \Delta, F\rangle$: un *TPAIO*, $T^T = \langle L \cup \{fail\} \cup \{inconc\}, l_0, \Sigma^T, \Gamma, X, \overline{\Delta} \cup \Delta_{\to fail} \cup \Delta_{\to inconc}, F\rangle$: le *TPTIO* de T ;
$TC = \langle N, E, w\rangle$ le trie d'un ensemble de chemins de T
SORTIES: le trie TC enrichie avec les verdicts pass, inconc et fail
// Remplacer les feuilles par le verdict pass
pour chaque $(n_s, n_t) \in E$ s.t. $n_t \in$ *Feuille*(N) **faire**
Nœud $n_{pass} \leftarrow$ creerNœud$(pass)$
$N \leftarrow N \cup \{n_{pass}\}$
$w(n_s, n_{pass}) \leftarrow w(n_s, n_t)$
$E \leftarrow E \setminus \{(n_s, n_t)\} \cup \{(n_s, n_{pass})\}$
fin pour
// ajouter les verdicts fail et inconc du *TPTIO* pour les états différents de $pass$
pour chaque $n_s = (l, p) \in N$ s.t. $n_s \neq n_{pass}$ **faire**
pour chaque $(l, act, g, X', fail) \in \Delta_{\to fail}$ **faire**
Nœud $n_{fail} \leftarrow$ creerNœud$(fail)$
$N \leftarrow N \cup \{n_{fail}\}$
$E \leftarrow E \cup \{(n_s, n_{fail})\}$
$w(n_s, n_{fail}) \leftarrow act, g, X'$
fin pour
si $\exists (l, act, g, X', inconc) \in \Delta_{\to inconc}$ **alors**
Nœud $n_{inconc} \leftarrow$ creerNœud$(inconc)$
$N \leftarrow N \cup \{n_{inconc}\}$
$E \leftarrow E \cup \{(n_s, n_{inconc})\}$
$w(n_s, n_{inconc}) \leftarrow act, g, X'$
fin si
fin pour
retourner TC

La proposition 5.4.1 montre qu'un arbre de tests est un *TPAIO* acyclique.

Proposition 5.4.1. Un arbre de tests TC est un *TPAIO* acyclique T.

Démonstration. Soit $TC = \langle N, E, w\rangle$ un arbre de tests qui peut être vu comme un *TPAIO* $T = \langle L, l_0, \Sigma, \Gamma, X, \Delta, F\rangle$ où :

- $L = N$ et $l_0 = racine(TC)$,
- $\Sigma = \{act \mid \exists (n, n', g, y).((n, n') \in E$ et $w(n, n') = (a, g, \{y\})$, et $(a =!act$ ou $a =?act))\}$,
- $\Gamma = \{act \mid \exists (n, n', g, y).((n, n') \in E$ et $w(n, n') = (a, g, \{y\})$ et $(a = act^-$ ou $a = act^+))\}$,
- $X = \{y\}$ si $\exists (n, n', act, g).((n, n') \in E$ et $w(n, n') = (act, g, \{y\}))$,

- $\Delta = \{(n, act, g, \{y\}, n') \mid (n, n') \in E$ et $w(n, n') = (act, g, \{y\})\}$,
- $F = \{n' \mid \exists (n, act, g, y).((n, n') \in E$ et $n' = pass$ et $w(n, n') = (act, g, \{y\})\}$.

□

5.5/ CORRECTION, INCOMPLÉTUDE ET COUVERTURE DE TEST DE LA MÉTHODE

Cette section discute de la correction, de l'incomplétude et de la couverture de test de notre méthode de génération des tests à partir d'un *TPAIO* déterministe donné.

5.5.1/ CORRECTION

La définition 23 définit l'atteignabilité d'une localité dans un *RTA* donné.

Définition 23 : Atteignabilité dans un *RTA*

Une localité l_i est atteignable dans un *RTA* si et seulement s'il existe une ϵ-transition $l_0 \xrightarrow{a,g,X'} l_i$ telle que g est satisfiable.

Pour prouver l'atteignabilité d'une localité l_i dans un *RTA*, on prouve qu'il existe une séquence d'ϵ-transitions qui part de l_0 pour aller à l_i où toutes les contraintes d'horloges sont satisfiables. Si la séquence est formée par une seule ϵ-transition, il est suffisant d'évaluer la satisfiabilité de sa garde. Sinon, les contraintes d'horloge sont constituées de fermetures arrières qui n'ont pas été vérifiées, alors que les contraintes de la pile ont déjà été vérifiées par la construction du *RTA* avec les règles RA_1 à RA_4. Par conséquent, on peut oublier les contraintes de la pile. Par la fusion des ϵ-transitions consécutives en une seule ϵ-transition en utilisant la règle RA_4, on peut finalement obtenir une seule ϵ−transition qui passe de l_0 à l_i. Décider de l'atteignabilité d'une localité l_i se ramène donc à vérifier la satisfiabilité de la garde de cette ϵ−transition.

Nous nous intéressons aussi au cas du test de conformité. On souhaite vérifier qu'une implémentation dont on ne peut observer que les entrées et sorties, est conforme à une spécification donnée ou non. Le théorème 5.5.1 et le lemme 5.5.1 spécifient la correction de notre méthode de test en prouvant que si une exécution de l'implémentation I aboutit à une localité $fail$, parce que c'est une exécution interdite par sa spécification T, l'implémentation I n'est pas conforme à la spécification T selon la relation de conformité *tpioco*.

Théorème 5.5.1. Une localité l est atteignable dans un *TPAIO* si et seulement si elle est atteignable dans son *RTA*.

Démonstration. C'est une conséquence directe du Lemme 5.3.2. □

Lemme 5.5.1. Soit $\pi = l_0 \xrightarrow{a_0,g_0,X_0} l_1 \xrightarrow{a_1,g_1,X_1} l_2 \xrightarrow{a_2,g_2,X_2} ...l_{n-1} \xrightarrow{a_{n-1},g_{n-1},X_{n-1}} l_n \xrightarrow{a_n,g_n,X_n} fail$ un chemin de l'arbre de tests d'une spécification $T = \langle L, l_0, \Sigma_{in} \cup \Sigma_{out}, \Gamma, X, \Delta, F\rangle$ où $l_i \in L$, $g_i \in Grd(X)$ et $a_i \in \Sigma_{in} \cup \Sigma_{out} \cup \Gamma^{+-} \cup \{othw\}$ pour $0 \leqslant i \leqslant n$. Si le verdict $fail$ est observé

en exécutant π sur l'implémentation I, alors l'implémentation I n'est pas conforme à la spécification T d'après la definition de *tpioco*.

Démonstration. Soit $\rho = t_0 a_0 t_1 a_1 t_2 ... t_{n-1} a_{n-1} t_n a_n \in RT(\Sigma_{in} \cup \Sigma_{out} \cup \Gamma^{+-})$ une trace d'une exécution du chemin π. L'état $(l_n, v_n + t_n, p_n)$ est l'état courant après l'exécution du chemin de trace $t_0 a_0 t_1 a_1 t_2 ... t_{n-1} a_{n-1} t_n$. Il existe ces trois cas pour atteindre $fail$:

- $fail$ est atteint après l'observation de a_n dans le cas d'une action de sortie ou une action de la pile non spécifiée dans la spécification selon le cas (i) de la Def. 18 du testeur. S'il n'existe aucune transition qui appartient à Δ et qui quitte l_n avec le label a_n où $a_n \in \Gamma^{+-} \cup \Sigma_{out}$, alors, cette transition $(l_n, a_n, true, \emptyset, fail)$ est une transition de ce testeur. Donc, $a_n \notin out(T\ after\ t_0 a_0 t_1 a_1 t_2 ... t_{n-1} a_{n-1} t_n)$ et I n'est pas conforme à T.
- $fail$ est atteint après l'observation d'une action de pile ou de sortie plus tôt ou plus tard que spécifié selon, le cas (ii) de la Def. 18 de ce testeur. Il existe une transition qui appartient à Δ et qui quitte l_n avec le label a_n où $a_n \in \Gamma^{+-} \cup \Sigma_{out}$. Mais, La valuation des horloges courantes ne satisfait pas la garde g_n. Donc, $a_n \notin out(T\ after\ t_0 a_0 t_1 a_1 t_2 ... t_{n-1} a_{n-1} t_n)$ et I n'est pas conforme à T.
- $fail$ est atteint après l'observation de a_n dans la cas d'une action non autorisée dans la spécification, mais qui est exécutée par l'implémentation selon le cas (iii) de la Def. 18 du testeur. S'il n'existe aucune transition qui appartient à Δ et qui quitte l_n avec le label a_n où $a_n \notin \Gamma^{+-} \cup \Sigma_{out}$, alors cette transition $(l_n, ?othw, true, \emptyset, fail)$ est une transition de ce testeur. Donc, $a_n \notin out(T\ after\ t_0 a_0 t_1 a_1 t_2 ... t_{n-1} a_{n-1} t_n)$ et I n'est pas conforme à T.

□

Pour chaque non-conformité détectée par un chemin d'un arbre de tests, il y a une non-conformité entre l'implémentation et la spécification (*TPAIO*).

5.5.2/ INCOMPLÉTUDE

La complexité de l'algorithme de la Fig. 5.3.1 est polynomiale grâce à la limitation de la construction des ϵ-transitions par le critère de couverture consistant à n'ajouter que les ϵ-transitions qui permettent de couvrir une nouvelle transition du *TPAIO*. Dans le cas d'un *PA*, il existe une ϵ-transition dans son automate d'accessibilité entre deux localités l et l' si et seulement si l' est atteignable à partir de l dans le *PA* [FWW97]. Ceci prend en compte les contraintes de pile. Mais dans le cas d'un *TPAIO*, l'atteignabilité dépend aussi des contraintes d'horloge : il est suffisant que la garde d'une ϵ-transition de l à l' soit satisfiable pour dire que l' est atteignable à partir de l dans ce *TPAIO*. Mais, si la garde n'est pas satisfiable, l' pourrait quand même être atteignable à partir de l, mais à travers un autre chemin. Comme on a abandonné le calcul de certains des chemins possibles, on ne peut plus conclure que l' n'est pas atteignable. Par contre, si on conclut qu'elle est atteignable, alors, elle l'est par correction de la méthode (voir la partie 5.5.1). La Fig. 5.8 illustre l'incomplétude de notre algorithme. La Fig. 5.8(a) présente un *TPAIO*. La Fig. 5.8(b) montre le *RTA* obtenu en appliquant notre algorithme. Le *RTA* contient la transition horizontale $l_0 \rightarrow l_1$ qui fusionne $l_0 \rightarrow l_0$ suivi de $l_0 \rightarrow l_1$ et la transition courbe qui fusionne deux cycles $l_0 \rightarrow l_0$ suivi de $l_0 \rightarrow l_1$ et d'un cycle $l_1 \rightarrow l_1$. La localité l_1 n'est

pas atteignable dans ce *RTA*, parce qu'aucune des deux gardes des transitions $l_0 \xrightarrow{\epsilon,\dots} l_1$ n'est satisfiable. Pourtant, l_1 est atteignable dans le *TPAIO*. Cela aurait pu être détecté si la transition $l_0 \xrightarrow{\epsilon,x\leqslant 1\wedge x\leqslant 1\wedge x\leqslant 1\wedge x=0\wedge y\geqslant 3\wedge x\leqslant 2\wedge x\leqslant 2,\{x\}} l_1$ qui effectue 3 cycles $l_0 \to l_0$, la transition $l_0 \to l_1$ puis 2 cycles $l_1 \to l_1$ avait été ajoutée en appliquant la règle RA_3.

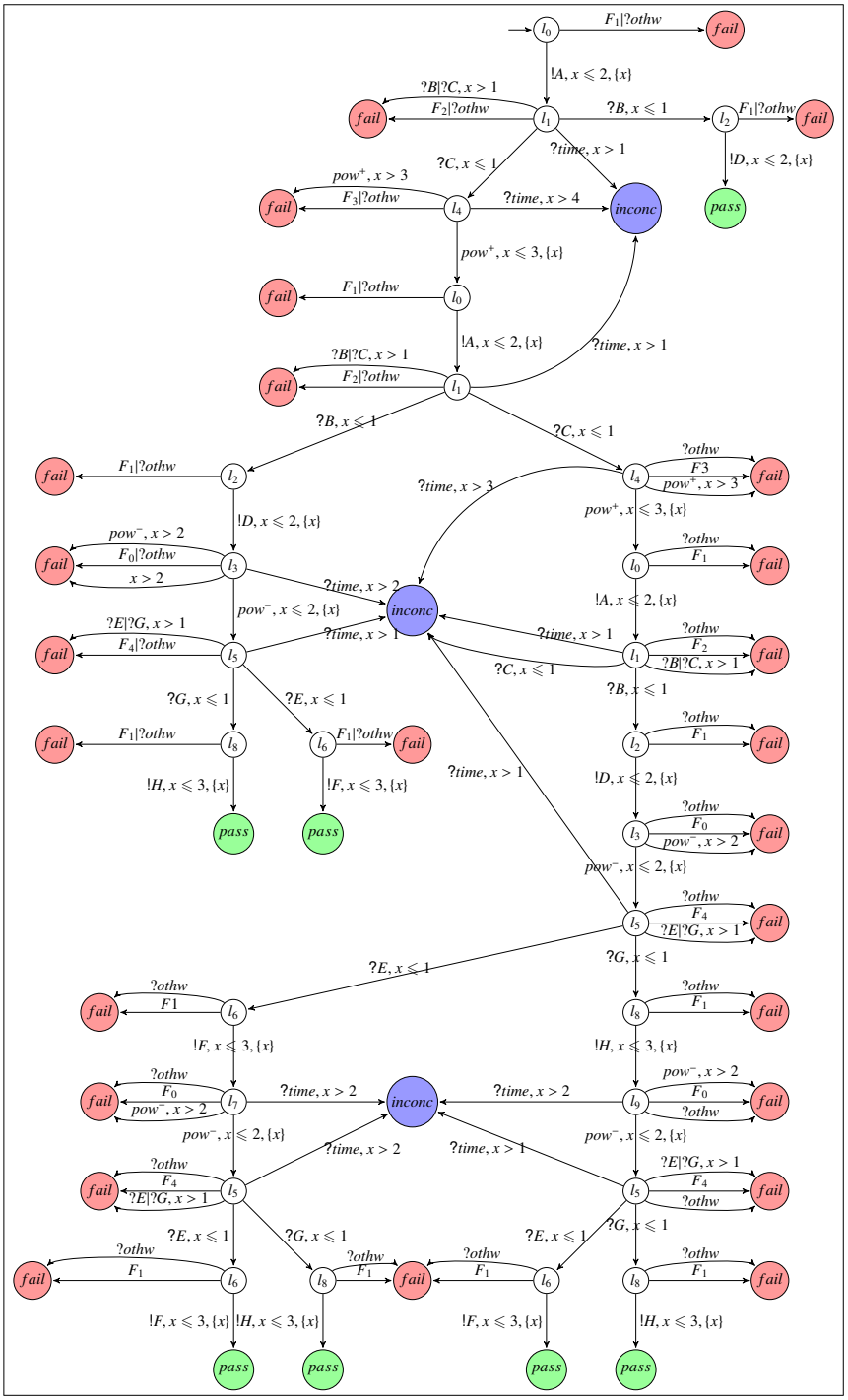


FIGURE 5.7 – Arbre de tests du *TPAIO* de la Fig 5.2

Notons que 3 cycles qui laissent écouler chacun au maximum une unité de temps sont nécessaires pour pouvoir satisfaire la condition $y \geqslant 3$ sur la transition $l_0 \longrightarrow l_1$. Cette transition n'a pas été ajoutée par notre algorithme parce qu'elle ne couvre aucune nouvelle transition entre l_0 et l_1. Notre algorithme est donc incomplet, dans le but de préserver une complexité polynomiale.

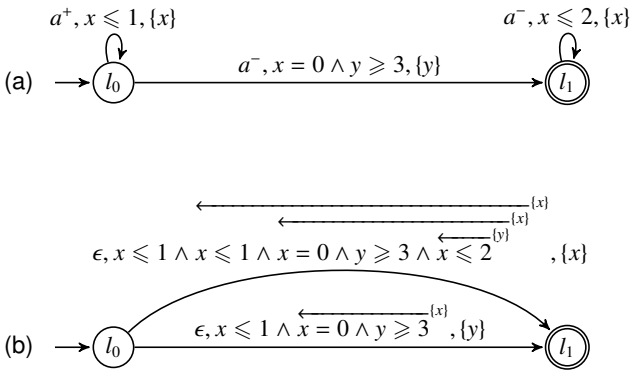


FIGURE 5.8 – *RTA* (b) représentant l'atteignabilité incomplète du *TPAIO* (a)

5.5.3/ COUVERTURE DES TESTS

Dans la section 5.3.2, on a présenté une méthode de calcul d'un *RTA* à partir d'un *TPAIO*. L'algorithme qui calcule ce *RTA* a pour critère d'arrêt la couverture de toutes les transitions du *TPAIO*. Il ajoute une nouvelle ϵ-transition (l, ϵ, g, X, l') seulement si son chemin couvre une nouvelle transition du *TPAIO* entre les localités l et l'. Les chemins de toutes les ϵ-transitions qui partent d'une localité initiale à une localité finale du *RTA* couvrent toutes les transitions du *TPAIO*. Mais, on ne peut pas conclure que les cas de tests calculés à partir de ces chemins couvrent toutes les transitions du *TPAIO*. En effet, la garde d'une ϵ-transition temporisée peut ne pas être satisfiable. Mais, si les gardes de toutes les ϵ-transitions qui partent d'une localité initiale et terminent dans une localité finale sont satisfiables, alors toutes les localités et toutes les transitions atteignables du *TPAIO* sont couvertes. C'est le cas dans notre exemple de la Fig. 5.2. En synthèse, dans le meilleur des cas notre méthode couvre toutes les localités et toutes les transitions atteignables du *TPAIO*.

5.6/ CONCLUSION

On a présenté dans ce chapitre une méthode polynomiale et incomplète de génération de tests à partir d'un *TPAIO*. A notre connaissance, ce problème n'était pas résolu pour les *TPAIO* dans la littérature. La méthode consiste à définir un testeur d'un *TPAIO* pour établir le verdict de tests. Puis, elle calcule un automate temporisé d'atteignabilité qui permet de prendre en compte les contraintes de pile. Les contraintes d'horloges sont accumulées en utilisant des fermetures en arrières lors du calcul du *RTA*. Elles sont évaluées à la fin de la construction du *RTA* en utilisant un SMT-solveur. On calcule aussi une table de chemins qui associe à chaque transition du *RTA* un ou plusieurs chemins du *TPAIO*. Enfin, la méthode génère un arbre de tests à partir de cette table de chemins

du *RTA* et du testeur, couvrant les localités et les transitions atteignables du *TPAIO*.

Cette méthode n'est pas complète. Cela a pour conséquence que certaines localités atteignables peuvent ne pas être prises en compte dans les tests produits. Dans le chapitre suivant, on propose une méthode de génération de tests à partir d'un *TPAIO* qui améliore la complétude et traite des *TPAIO* déterministes.

6

TEST DE CONFORMITÉ À PARTIR D'UN PROGRAMME RÉCURSIF TEMPORISÉ

Sommaire

Dans le chapitre précédent, on a présenté une méthode incomplète de génération de tests à partir d'un *TPAIO* déterministe avec deadline $lazy$ seulement. On propose dans ce chapitre de compléter la méthode de génération de tests présentée dans le chapitre précèdent. Notre deuxième contribution est une méthode de génération de tests à partir d'un *TPAIO* déterministe avec sorties seulement et deadline $delayable$ seulement. Ce modèle est une abstraction d'un programme récursif temporisé. Notre méthode de génération de tests adapte la méthode de génération de tests à partir d'automates temporisés [KT09] aux automates à pile temporisés. Elle définit une méthode de génération d'un testeur à pile temporisé déterministe avec sorties seulement. Enfin, une suite de tests couvrant les états et les transitions atteignables est engendrée.

Pour générer des cas des tests permettant d'assurer la couverture de toutes les localités et de toutes les transitions d'un *TPAIO* donné, on propose de calculer un *STPTIO* (pour Symbolic Timed Pusdhown Tester with Inputs and Outputs) à partir du *TPAIO* en adaptant la méthode de déterminisation de Krichen et Tripakis [KT09] (pour les automates temporisés avec entrées/sorties et sans pile) au cas des *TPAIO*. Même dans le cadre de programmes déterministes, cette étape est utile car elle produit un modèle avec une seule horloge, qui se remet à zero à chaque transition. Ce modèle est enrichi avec des verdicts $fail$ qui peuvent être observés en cas d'action non spécifiée. Nous proposons ensuite une méthode qui adapte aux *STPTIO* la méthode de calcul des états accessibles de Finkel et al. [FWW97] définie pour des automates à pile pour calculer un *RA* (Reachability Automata). L'adaptation consiste essentiellement à calculer les transitions du *RA* comme des π-transitions où π est un chemin du *STPTIO*. Les chemins qui partent d'une localité symbolique à une localité finale sont utilisés pour générer des cas de tests sous forme d'arbres. Cette contribution a donné lieu à la

publication [MJMR15a].

Ce chapitre est organisé de la manière suivante : la Sec. 6.1 décrit les trois étapes majeures de la génération de tests à partir d'un *TPAIO* déterministe avec sorties seulement. La première étape est la construction d'un *STPTIO* à partir d'un *TPAIO*. Elle est présentée dans la Sec. 6.2. La seconde étape, décrite dans la Sec. 6.3, présente la méthode de calcul d'un *RA* à partir d'un *STPTIO*. La Sec. 6.4 présente la troisième étape qui est la génération des arbres de tests. La Sec. 6.6 discute la correction, l'incomplétude et la couverture de test de notre méthode. La Sec. 6.5 présente l'approche adoptée pour évaluer la qualité des suites de tests par mutation. Elle consiste à exécuter les cas de test sur des implémentations obtenues par mutation de la spécification et annoncer le verdict relativement à la relation de conformité.

6.1/ PROCESSUS DE GÉNÉRATION DE TESTS

Le diagramme de la Fig. 6.1 présente les trois étapes majeures pour la génération de tests à partir d'un *TPAIO* déterministe avec sorties seulement. Les deadlines de toutes les transitions sont *delayable*. Ces étapes sont les suivantes.

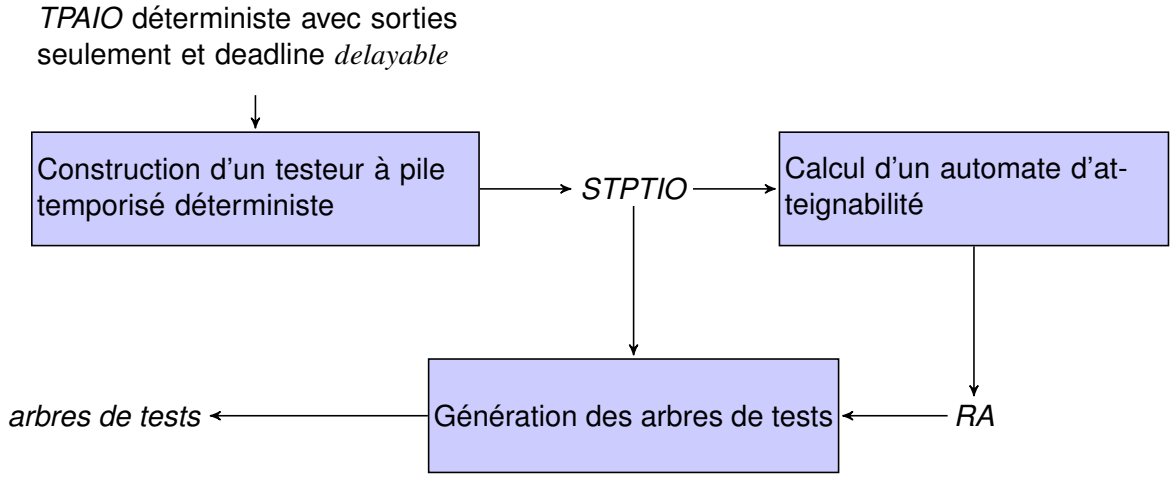


FIGURE 6.1 – Processus de génération de tests à partir d'un *TPAIO*

1. **Construction d'un testeur (*STPTIO*) à partir d'un *TPAIO*** : un *TPAIO* spécifie des contraintes d'horloges et des contraintes de pile. On propose de calculer un testeur temporisé à pile avec entrées/sorties appelé *STPTIO* (Symbolic Timed Pushdown Tester with Inputs and Outputs) qui résout les contraintes d'horloges et les contraintes de pile. Ce testeur est un *TPAIO* à localités symboliques avec une seule horloge et enrichi avec une nouvelle localité $fail$ représentant un verdict de non conformité à la spécification. Il est enrichi aussi avec une nouvelle localité $inconc$ pour inconclusif et les transitions qui y mènent pour modéliser que le testeur limite le temps d'observation des implémentations. Cette étape est présentée dans la Sec. 6.2.

2. **Calcul d'un automate d'atteignabilité (*RA*) à partir du testeur** : Pour générer des cas des tests permettant d'assurer la couverture de toutes les localités et toutes les transitions du *STPTIO*, on propose de calculer un ou plusieurs chemins entre deux localités symboliques en respectant les contraintes d'horloges et les contraintes de pile. Le *RA* est un automate fini avec des π-transitions étiquetées par des chemins du *TPAIO* satisfaisant les contraintes d'horloges et les contraintes de pile. Cette étape est présentée dans la Sec. 6.3.

3. **Génération d'arbres de test** à partir d'un *TPAIO* en utilisant le *STPTIO* et son *RA*. Cette étape est divisée en deux phases : (a). générer des chemins de tests pour chaque cas de test. Un cas de test est associé à chaque π-transition du *RA* qui part d'une localité symbolique initiale vers une localité symbolique finale ; (b). enrichir chaque chemin de test avec les transitions qui atteignent $fail$ et $inconc$ en utilisant son *STPTIO*. On obtient un arbre de tests par cas de test. Les cas de tests sont individualisés car il est possible de calculer des valeurs d'entrée associées à un chemin d'exécution de la fonction sous test étant donné que notre modèle est une abstraction d'une fonction récursive temporisée. Cette étape est présentée dans la Sec. 6.4.

6.2/ CONSTRUCTION D'UN TESTEUR À PARTIR D'UN *TPAIO*

Krichen et Tripakis proposent une méthode de test de conformité à partir d'un *TAIO* non déterministe dans [KT09]. Ils proposent un algorithme de génération de cas de tests en calculant un testeur avec une seule horloge et une remise à zéro de l'horloge à chaque observation d'une action. Ils proposent une méthode de déterminisation d'un *TAIO* donné avec une estimation sur-approximée de l'ensemble des localités atteignables (voir la Sec. 2.5.2 de ce mémoire). Cette méthode fait des calculs successifs des meilleures sur-approximations des gardes de transitions ajoutées.

Définition 24 : Testeur à pile temporisé déterministe *STPTIO*

Un *STPTIO* $T^T = (L^T, l_0^T, \Sigma_{out}^T, \Gamma, \{y\}, \Delta^T, F^T)$ d'un *TPAIO* $T = (L, l_0, \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F\rangle$ est un *TPAIO* avec une seule horloge y qui est différente des horloges de X où :

- $l_0^T \in L \times CC(X \cup \{y\}) \times \Gamma^*$ est la localité symbolique initiale,
- $L^T \subseteq (L \times CC(X \cup \{y\}) \times \Gamma^*) \cup \{fail, inconc\})$ est un ensemble de localités symboliques,
- $\Sigma_{out}^T = \emptyset$ et $\Sigma_{in}^T = \Sigma_{out} \cup \{othw, time\}$,
- $\Delta^T \subseteq L^T \times \Sigma_{in}^T \cup \Gamma^{+-} \times Grd(\{y\}) \times \{y\} \times L^T$ est un ensemble fini de transitions,
- $F^T = \{(l, v, p) \mid (l, v, p) \in L^T \text{ et } l \in F\}$ est un ensemble des localités symboliques finales.

On propose d'adapter la méthode de [KT09] définie pour les *TAIO* non déterministes pour calculer un testeur à partir d'un *TPAIO* déterministe avec sorties seulement. Le testeur calculé est appelé *STPTIO*. Il est constitué de localités symboliques qui sont des triplets formés non seulement d'une localité du *TPAIO* et d'une contrainte d'horloge comme dans [KT09], mais également d'une valeur de pile. On introduit une nouvelle localité $fail$ et des transitions qui l'atteignent à partir d'autres localités symboliques. C'est

un *TPAIO* avec une seule horloge qui se remet à zéro à chaque franchissement d'une action et dont toutes les transitions ont la deadline *lazy*. Donc, toutes les gardes de ce testeur sont satisfiables. Afin de détecter la non-conformité entre une implémentation et un *TPAIO*, les actions d'entrée du *STPTIO* sont les actions de sortie (Σ_{out}) du *TPAIO*, et les actions de pile les mêmes (Γ) que celles du *TPAIO*. Nous notons $othw$ un nom quelconque d'action différent de chaque nom de Σ_{out}, de Γ^{+-} et de τ.

Dans ce chapitre, nous avons choisi pour deadline par défaut $delayable$ ce qui fait que le temps ne peut pas s'écouler indéfiniment dans les localités. Mais, on peut accepter aussi des exécutions dans lesquelles le temps peut s'écouler à l'infini même avec une deadline $delayable$. C'est le cas où l'instant d'arrivée dans la localité est plus grand que l'instant maximal où une transition est déclenchable. Pour ne pas avoir à attendre indéfiniment avant d'émettre un verdict, on propose de fixer un temps d'attente choisi arbitrairement et suffisamment long max. Pour ce cas, on ajoute une nouvelle localité $inconc$ correspondant à un verdict inconclusif, et les transitions qui y mènent correspondent au cas où cette attente maximum est écoulée. Définissons maintenant la notion de *STPTIO* dans la Def. 24. Les localités symboliques d'un *STPTIO* sont un couple (localité du *TPAIO*, contraintes). Les contraintes portent sur les horloges du *TPAIO* et sur la nouvelle horloge y du *STPTIO*. Les localités sont dites symboliques car elles représentent un ensemble de localités pour l'ensemble des valeurs d'horloges qui satisfont la contrainte. Par exemple, la localité symbolique $(l_0, 0 \leqslant x - y \leqslant 0, \perp)$ dans la Fig. 6.4 représente toutes les localités pour les valeurs d'horloges x, y telles que $x = y$.

Avant de présenter la définition et les principes de calcul des transitions d'un *STPTIO* à partir d'un *TPAIO* donné, on a besoin d'adapter les notations suivantes déjà introduites en Sec. 2.5.2 :

- $usucc(l^T) = l^{T'}$ sachant que $\exists \rho.(\rho \in RT(\{\tau\}) \wedge l^T \rightarrow^{\rho} l^{T'})$ est une localité symbolique successeur de la localité symbolique l^T par une séquence d'actions non observables. (le u est pour unobservable)
- $dsucc(l^T, a) = l^{T'}$ sachant que $l^T \rightarrow^{a} l^{T'}$ est la localité symbolique atteignable à partir de la localité symbolique l^T par l'action a. (le d est pour discrète)
- $\Delta_{act}((l, v, p), u) = \{(l, act, g, X', l') \in \Delta_{act} \mid ((act \in (\Gamma^+ \cup \Sigma) \vee (act \in \Gamma^- \wedge act = a^- \wedge Top(p) = a)) \wedge v \wedge u \wedge g$ satisfiable $\}$ est l'ensemble de transitions étiquetées par act dont les gardes sont satisfaites par la localité symbolique (l, v, p), où l'action act est déclenchable, et où l'horloge y est égale à u. Vu que notre modèle est un *TPAIO* déterministe, Δ_{act} et $\Delta_{act}(l^T, u)$ contiennent au maximum une transition.

L'adaptation est de redéfinir $usucc$, $dsucc$ et $\Delta_a((l, v, p), u)$ pour une seule localité symbolique et pas pour un ensemble de localités symboliques. On a besoin de définir en plus la notation suivante :

- $(l, v, p) \rightarrow^{u} (l, v + u, p)$ qui est une transition temporisée qui dénote l'écoulement du temps dans une localité l pour une valuation d'horloge v avec un intervalle de temps u. Cette transition existe si et seulement si $ACV_l(v, p, v + u) = true$.

La localité symbolique initiale l_0^T est égale à $usucc(l_0^{X \cup \{y\}})$. Le principe de calcul des localités symboliques et des transitions du *STPTIO* est inspiré de la méthode de déterminisation avec sur-approximation présentée dans la Sec. 2.5.2. Il consiste à répéter les étapes suivantes : la sélection d'une localité symbolique l^T qui n'est pas encore ajoutée dans L^T et puis l'application de l'une de ces possibilités pour ajouter des nou-

velles transitions à Δ^T (voir Def.25) :

- La transition $(l^T, ?time, u, \{y\}, fail)$ est ajoutée dans Δ^T (cas (I) Def. 25) dans le cas de dépassement de temps dans la localité l^T dû à la violation de l'*ACV*. Le temps ne peut pas s'écouler avec un intervalle de temps u pour l^T.
- Pour chaque action $act \in \Sigma_{out} \cup \Gamma^{+-}$, les partitions associées à (l, v, p) sont $u \in \{[0,0],]0,1[, [1,1], ..., [K,K],]K, \infty[\}$. Pour chaque grande partition u, il existe deux possibilités : si $\Delta_{act}(l^T, u) = \emptyset$ alors, la transition $(l^T, act, u, \{y\}, fail)$ est ajoutée dans Δ^T (cas (III) de la Def.25). Elle représente le cas de l'observation d'une action de pile ou de sortie non spécifiée. Sinon, la transition $(l^T, act, u, \{y\}, usucc(dsucc(l^T \cap u, act)))$ est ajouté dans Δ^T (cas (II) de la Def.25) et la localité symbolique $usucc(dsucc(l^T \cap u, act))$ est ajoutée à L^T si elle n'existe pas déjà.
- Une transition $(l^T, ?othw, y \geqslant 0, \{y\}, fail)$ est ajoutée dans Δ^T (cas (IV) de la Def. 25). Elle représente le cas d'observation d'une action non autorisée dans T, mais qui peut être exécutée par l'implémentation.
- Une transition $(l^T, ?time, y > max, \{y\}, inconc)$ est ajoutée dans Δ^T (cas (V)de la Def. 25) dans le cas où le temps est autorisé à s'écouler indéfiniment dans la localité l^T. Elle n'autorise pas l'écoulement de temps au delà de max unités de temps. La valeur de max est la durée maximale d'observation qui est choisie arbitrairement par l'ingénieur de test. Elle doit être supérieure à la plus grande constante K apparaissant dans les constantes d'horloge.

Formalisons maintenant la notion de transitions du *STPTIO* dans la Def. 25.

Définition 25 : Transitions du Testeur à pile temporisé déterministe *STPTIO*

Un *STPTIO* $T^T = (L^T, l_0^T, \Sigma_{in}^T, \Gamma, \{y\}, \Delta^T, F^T)$ d'un *TPAIO* $T = (L, l_0, \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F\rangle$ est un *TPAIO*. Soit act une action de Σ_{out} ou de Γ^{+-}, les transitions de Δ^T sont définies comme suit :

(I) $(l^T, ?time, u, \{y\}, fail) \in \Delta^T$ si $l^T \nrightarrow^u$.

(II) $(l^T, act, u, \{y\}, l^{T'}) \in \Delta^T$ si $l^{T'} = usucc(dsucc(l^T \cap u, act))$.

(III) $(l^T, act, u, \{y\}, fail) \in \Delta^T$ si $\Delta_{act}(l^T, u) = \emptyset$.

(IV) $(l^T, ?othw, y \geqslant 0, \{y\}, fail) \in \Delta^T$.

(V) $(l^T, ?time, y > max, \{y\}, inconc) \in \Delta^T$ si $l^T \rightarrow^{\mathbb{R}^+}$.

Remarque 6.2.1. L'application répétitive du cas (II) de la Def.25 peut produire un nombre infini de transitions et de localités comme illustré dans l'exemple 6.2.1.

Exemple 6.2.1.

Dans la figure 6.2, on présente un exemple de *TPAIO* dont le *STPTIO* a une infinité de transitions. Pour le cas (II) de la Def.25, il existe une situation qui amène une infinité de transitions dans le *STPTIO*. Cette situation est une transition d'empilement réflexive franchissable provoquant un cycle qui fait un empilement d'un symbole sans le dépiler. L'empilement du symbole produit une nouvelle localité symbolique car l change le contenu de la pile, et la pile d'un *TPAIO* est non bornée. La figure 6.2.(b) présente une partie du

STPTIO du *TPAIO* présenté dans la figure figure 6.2 (a). Le *TPAIO* a la transition réflexive $(l_0, a^+, x \leqslant 2, lazy, \{x\}, l_0)$. On peut construire une infinité de transitions en calculant le successeur de l_0 par l'action a^+ car la pile est non bornée. On obtient une nouvelle localité symbolique à chaque empilement.

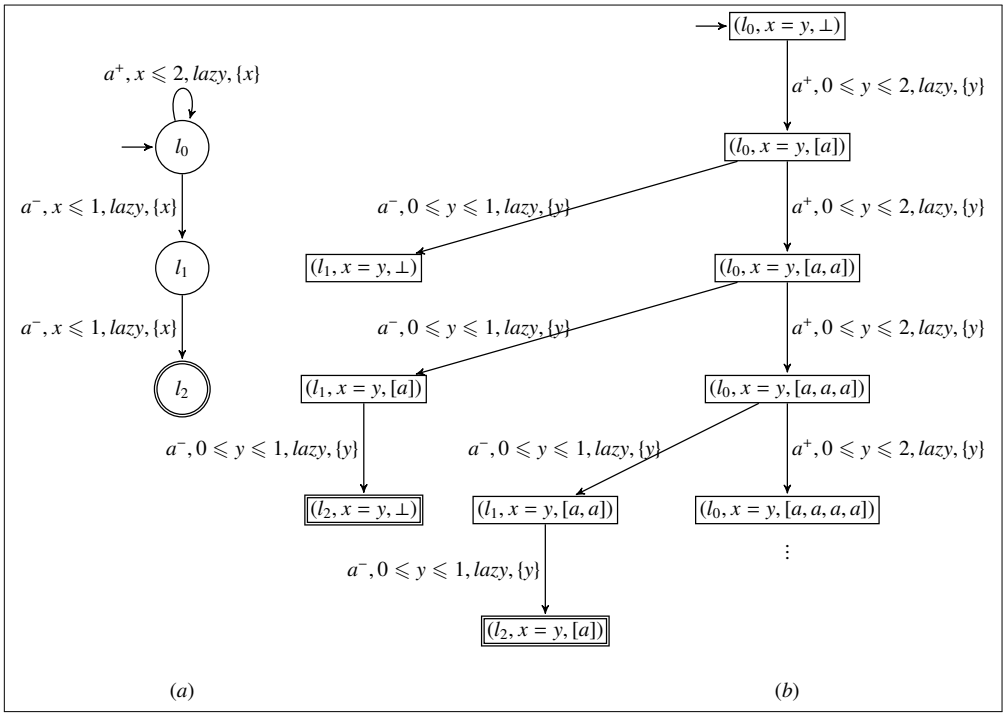


FIGURE 6.2 – (b) partie du *STPTIO* du *TPAIO* (a) sans les transitions vers $fail$ et $inconc$

Dans ce mémoire, nous proposons l'algorithme 6.2.1 qui permet de calculer un testeur T^T avec un nombre fini de transitions et de localités à partir d'un *TPAIO* donné. Pour présenter cet algorithme, on définit une variable $max\_Empilement \in L \times \Gamma^+ \rightarrowtail$ entier qui est une fonction totale bijective qui associe un entier à un couple formé d'une localité et d'un symbole de la pile. Cette variable est utilisée pour limiter le nombre d'empilement pour chaque symbole. L'algorithme est divisé en deux étapes : une étape d'initialisation de la ligne 1 à la ligne 9 et une étape de calcul de la ligne 10 à la ligne 39. La localité symbolique initiale l_0^T est égale à $usucc(l_0^{X \cup \{y\}})$. L'ensemble des localités symboliques L^T est initialisé avec $fail$ et $inconc$. L'ensemble des transitions Δ^T est initialement vide. Un ensemble New_SL est utilisé pour enregistrer les localités symboliques qui ont été calculées, mais qui ne sont pas encore exploitées pour calculer leurs successeurs. Il est initialisé avec la localité symbolique initiale l_0^T. La fonction $nb\_transitions(act, \Delta)$ calcule le nombre de transitions dans Δ avec l'action act. Pour chaque couple (l, a), $max\_Empilement(l, a)$ est initialisé avec le maximum entre 1 et le nombre de transitions de dépilement du symbole a dans le *TPAIO* de départ. Le calcul de T^T est un calcul itératif : à chaque fois, on choisit une localité l^T de *New_SL* et on fait les étapes suivantes : (1). la transition $(l^T, ?othw, y \geqslant 0, \{y\}, fail)$ est ajoutée dans Δ^T qui correspond au cas d'une action non autorisée dans T, mais qui peut-être exécutée par l'implémentation selon le cas (IV) de la Def. 25 du testeur (ligne 12) ; (2). la transition $(l^T, ?time, u, \{y\}, fail)$ est ajoutée dans Δ^T qui évite le dépassement de temps selon le cas (I) de la Def. 25 du testeur (ligne 14) ; (3). la transition $(l^T, ?time, y > max, \{y\}, inconc)$ est ajoutée dans Δ^T pour éviter l'écoulement de temps

Algorithme 6.2.1 Calcul du testeur à pile temporisé avec une seule horloge à partir d'un *TPAIO*

ENTRÉES : *TPAIO* $T = (L, l_0, \Sigma_{out}, \Gamma, X, \Delta, F)$ et $max : entier$
SORTIES : *STPTIO* $T^T = (L^T, l_0^T, \Sigma_{in}^T, \Gamma, \{y\}, \Delta^T, F^T)$
VARIABLES : $act \in \Sigma_{out} \cup \Sigma^{+-}$, $New\_SL \subseteq (L \times CC(X \cup \{y\}) \times \Gamma^*)$, $u \in Grd(\{y\})$, U ensemble de $Grd(\{y\})$, $l_0^T, l^T, l^{T'} \in L \times CC(X \cup \{y\}) \times \Gamma^*)$, $a \in \Gamma$, $l \in L$, $v \in CC(X \cup \{y\}$, $v \in \Gamma^*$, $max\_Empilement \in L \times \Gamma \rightarrowtail$ entier

BEGIN
$l_0^T \leftarrow usucc(l_0^{X \cup \{y\}})$
$New\_SL \leftarrow \{l_0^T\}$ // localités symboliques non traitées
$L^T \leftarrow \{fail, inconc\}$
$\Delta^T \leftarrow \emptyset$
pour chaque $l \in L$ **faire**
  pour chaque $a \in \Gamma$ **faire**
    $max\_Empilement(l, a^+) \leftarrow Max(1, nb\_transitions(a^-, \Delta))$
  fin pour
fin pour
tant que $New\_SL \neq \emptyset$ **faire**
  $l^T \leftarrow choisirElement(New\_SL)$
  $\Delta^T \leftarrow \Delta^T \cup \{(l^T, ?othw, y \geqslant 0, lazy, \{y\}, fail\}$ // cas (IV)
  si $\exists u.(l^T \nrightarrow^u)$ **alors**
    $\Delta^T \leftarrow \Delta^T \cup \{(l^T, ?time, u, lazy, \{y\}, fail\}$ // cas (I)
  sinon
    $\Delta^T \leftarrow \Delta^T \cup \{(l^T, ?time, y \geqslant max, lazy, \{y\}, inconc\}$ // cas (V)
  fin si
  pour chaque $act \in \Sigma_{out} \cup \Gamma^{+-}$ **faire**
    $U \leftarrow$ les classes d'équivalence induites par $\sim_{l^T}^{act}$
    pour $u \in U$ **faire**
      $l^{T'} \leftarrow usucc(dsucc(l^T \cap u, act))$
      si $l^{T'} \neq null$ **alors**
        si $act \in \Sigma_{out} \cup \Gamma^-$ ou ($act \in \Gamma^+$ et $l^T = (l, v, p)$ et $max\_Empilement(l, act) > 0$) **alors**
          $\Delta^T \leftarrow \Delta^T \cup \{(l^T, ?act, u, lazy, \{y\}, l'^T)\}$ // cas (II)
          si $act \in \Gamma^+$ **alors**
            $max\_Empilement(l, act) \leftarrow max\_Empilement(l, act) - 1$
          fin si
          si $l^{T'} \notin L^T$ **alors**
            $New\_SL \leftarrow New\_SL \cup \{l^{T'}\}$
          fin si
        fin si
      sinon
        $\Delta^T \leftarrow \Delta^T \cup \{(l^T, ?act, u, lazy, \{y\}, fail)\}$ // cas (III)
      fin si
    fin pour
  fin pour
  $New\_SL \leftarrow New\_SL \setminus \{l^T\}$
  $L^T \leftarrow L^T \cup \{l^T\}$
fin tant que
FIN.

au delà de max unités de temps selon le cas (V) de la Def. 25 du testeur (ligne 16) pour éviter l'écoulement de temps indéfiniment ; (4). pour chaque action $act \in \Gamma^{+-} \cup \Sigma_{out}$ de pile et de sortie : l'algorithme calcule les partitions U induites par $\sim_{l^T}^{act}$ (voir la Sec. 2.5.2). Puis, il calcule pour chaque partition u de U le successeur $l^{T'} = usucc(dsucc(l^T \cap u, a))$ de la localité symbolique l^T et alors il existe deux possibilités :

- Cas $l^{T'}$ n'est pas vide : la transition $(l^T, act, u, \{y\}, l^{T'})$ est ajoutée à Δ^T selon le cas (II) de la Def. 25 du testeur (ligne 24), et la localité symbolique $l^{T'}$ est ajoutée dans New_SL si elle n'existe pas dans L^T pour ne traiter chaque localité symbolique qu'une seule fois.
- Cas $l^{T'}$ est vide : pour assurer la terminaison de l'algorithme et la couverture de toutes les transitions, la transition $(l^T, act, u, \{y\}, fail)$ est ajoutée à Δ^T selon le cas (III) de la Def. 25 du testeur (ligne 33) si et seulement si act est une action de dépilement ou de sortie ou d'empilement autorisé. Soit $l^T = (l, v, p)$, un empilement par l'action act est autorisé si et seulement si $max\_Empilement(l, act) > 0$. Ensuite, $max\_Empilement(l, act)$ est décrémenté si act est une action d'empilement (ligne 26) pour indiquer qu'après un empilement on a droit à un de moins.

L'algorithme se termine quand $New\_SL = \emptyset$. En appliquant cet algorithme on obtient un *STPTIO* qui est un *TPAIO* dont le nombre de localités symboliques peut être doublement exponentiel par rapport au nombre des localités du *TPAIO* de départ. La fonction $choisirElement(New\_SL)$ retourne un élément quelconque de l'ensemble New_SL. L'avantage de cette méthode est qu'on a toujours un nombre de localités symboliques inférieur au nombre de régions du graphe des régions [SVD01] d'un *TPAIO* donné. $null$ représente une localité symbolique vide. L'exemple 6.2.2 illustre le notion de testeur de ce chapitre.

Exemple 6.2.2. La Fig. 6.3 présente un *TPAIO* qui est constitué de l'automate à pile de la Fig. 3.7 auquel on a ajouté arbitrairement des contraintes d'horloges. Les labels sont les instructions étiquetées dans le programme de la Fig. 3.3.1. Les actions de Σ_{out} sont étiquetées par les labels A à E défini ainsi : $A \stackrel{def}{=}$ int res_1, res_2, $B \stackrel{def}{=} n \leqslant 1$, $C \stackrel{def}{=} n > 1$, $D \stackrel{def}{=} return\ n$ et $E \stackrel{def}{=} return\ res_1 + res_2$. Il n'y a pas d'actions d'entrées. La deadline de chaque transition est fixée à *delayable* pour indiquer une limite du temps d'exécution de chaque action. Cet automate à pile temporisé permet d'observer les actions élémentaires de calcul des nombres de Fibonacci tout en observant les délais fixés pour l'exécution de chaque action.

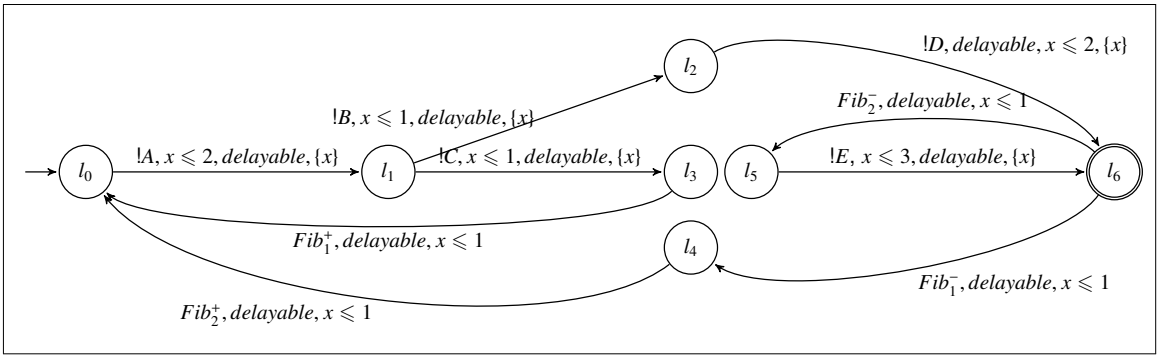


FIGURE 6.3 – *TPAIO* modélisant le programme de la Fig. 3.3.1

La Fig. 6.4 présente le testeur (*STPTIO*) associé au *TPAIO* de la Fig. 6.3 où $v_i \stackrel{def}{=} 0 \leqslant x - y \leq i$. Le label décrit par $a_1|a_2|...|a_n$ dénote l'ensemble des labels $\{a_1,a_2,...,a_n\}$. La

notation F est celle du label suivant $Fib_1^+|Fib_2^+|?A|?B|?C|?D|?E$ et les notations $F0$ à F_5 sont les abréviations suivantes : $F0 \stackrel{def}{=} F \setminus \{?A\}$, $F1 \stackrel{def}{=} F \setminus \{?B, ?C\}$, $F2 \stackrel{def}{=} F \setminus \{?D\}$, $F3 \stackrel{def}{=} F \setminus \{Fib_1^+\}$, $F4 \stackrel{def}{=} F \setminus \{Fib_2^+\}$ et $F5 \stackrel{def}{=} F \setminus \{?E\}$. Pour chaque l de L, les variables $max\_Empilement(l, Fib_1^+)$ et $max\_Empilement(l, Fib_2^+)$ sont initialisées avec 1 car le nombre des transitions avec Fib_1^- ou Fib_2^- est égale à 1. Le testeur de la Fig. 6.4 possède par exemple la transition $((l_0, v_0, \perp), ?A, y > 2, \{y\}, fail)$ qui illustre le cas (III) de la Def. 25 car $A \in \Sigma_{in}^T$ et la contrainte $0 \leqslant x - y \leq 0 \wedge x \leqslant 2 \wedge y > 2$ où $x \leqslant 2$ est la garde de la transition $(l_0, A, x \leqslant 2, \{x\}, l_1)$ n'est pas satisfaite. Il possède aussi par exemple la transition $((l_0, v_0, \perp), F_0, y \geqslant 0, \{y\}, fail)$ (cas (III) Def.25) car il n'existe aucune transition qui quitte l_0 avec l'un des labels C, B, D, E, F, G, H, pow^+ ou pow^-. Il a aussi la transition $((l_1, v_0, \perp), ?C, y \leqslant 1, \{y\}, (l_3, v_0, \perp))$ où $(l_3, v_0, \perp) = usucc(dsucc((l_1, v_0, \perp) \cap [0, 1], C))$ qui illustre le cas (II) de la Def. 25. Pour satisfaire le cas (IV) de la Def. 25, il a par exemple la transition $((l_1, v_0, \perp), ?othw, y \geqslant 0, \{y\}, fail)$. Pour satisfaire le cas (I) de la Def. 25, il a par exemple la transition $((l_0, v_0, \perp), ?time, y > 2, \{y\}, fail)$ car le temps ne peut pas s'écouler au delà de deux unités de temps dans la localité l_0. On ne trouve aucune transition aboutissant à un verdict inconclusif (cas (V) de la Def. 25) car aucune localité du *TPAIO* ne laisse pas le temps s'écouler à l'infini. En effet, chaque transition du *TPAIO* a le deadline $delayable$ et remet l'horloge à zero.

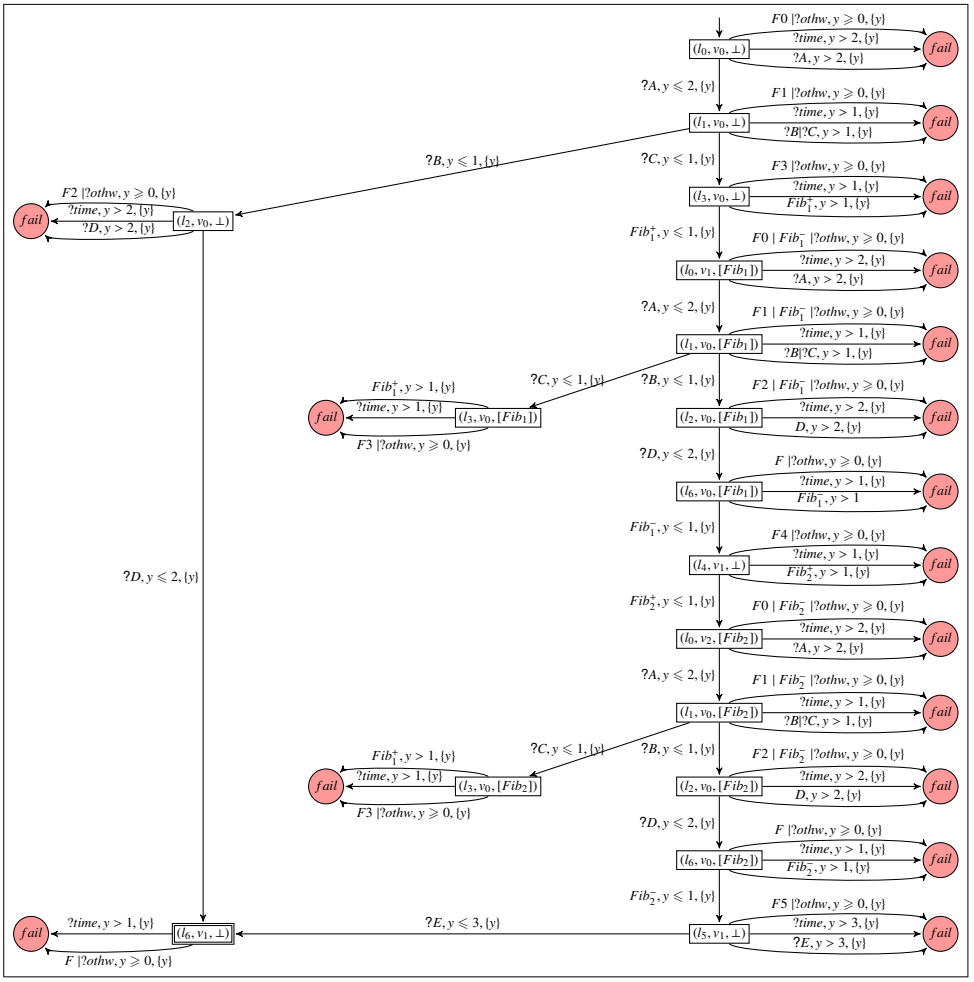


FIGURE 6.4 – Testeur (*STPTIO*) associé au *TPAIO* de la Fig. 6.3

6.3/ CALCUL D'UN *RA* À PARTIR D'UN *STPTIO*

Soit $(L^T, l_0^T, \Sigma_{in}^T, \Gamma, \{y\}, \Delta^T, F^T)$ un *STPTIO*. On propose de calculer autant de chemins nécessaire entre deux localités symboliques pour couvrir toutes les transitions qui les relient. Nous la représentons par un automate appelé automate d'atteignabilité (*RA*) d'un *STPTIO*.

Un *RA* est un automate fini avec des transitions étiquetées par des séquences de transitions du *STPTIO* ($\in (\Delta^T)^*$). Une transition étiquetée appelée $\pi-$transition et dénotée $(l^T, \pi, l^{T'})$ est une transition qui atteint la localité symbolique $l^{T'}$ à partir de la localité symbolique l^T en passant par le chemin π et qui retrouve en $l^{T'}$ le même état de pile qu'en l^T. On propose dans la Def. 26 la définition de ce *RA* par application par saturation des règles R_1 à R_4.

Définition 26 : *RA* d'un *STPTIO*

Le *RA* d'un *STPTIO* $(L^T, l_0^T, \Sigma_{in}^T, \Gamma, \{y\}, \Delta^T, F^T)$ est un automate $(L^R, l_0^T, (\Delta^T)^*, \Delta^R, F^T)$ où $L^R = L^T \setminus \{fail, inconc\}$ et $\Delta^R \subseteq L^R \times (\Delta^T)^* \times L^R$ est la plus petite relation satisfaisant les conditions suivantes. Soit $t_1 \stackrel{def}{=} (l_1^T, a^+, g_1, \{y\}, l_2^T)$, $t_2 \stackrel{def}{=} (l_2^T, a^-, g_2, \{y\}, l_3^T)$ et $t_3 \stackrel{def}{=} (l_3^T, a^-, g_3, \{y\}, l_4^T)$ qui sont trois transitions de Δ^T où l_2^T, l_3^T, l_4^T sont différents de la localité $fail$ et $inconc$.

- **R**$_1$: $(l^T, t, l^{T'}) \in \Delta^R$ si $t \stackrel{def}{=} (l^T, A, g, \{y\}, l^{T'})$ et $t \in \Delta^T$,
- **R**$_2$: $(l_1^T, t_1.t_2, l_3^T) \in \Delta^R$,
- **R**$_3$: $(l_1^T, t_1.\pi.t_3, l_4^T) \in \Delta^R$ si $(l_2^T, \pi, l_3^T) \in \Delta^R$,
- **R**$_4$: $(l_1^T, \pi_1.\pi_2, l_3^T) \in \Delta^R$ si $(l_1^T, \pi_1, l_2^T) \in \Delta^R$, $(l_2^T, \pi_2, l_3^T) \in \Delta^R$.

On a adapté l'algorithme proposé par Finkel et al. [FWW97] pour calculer un automate d'atteignabilité à partir d'un automate à pile afin de calculer un *RA* à partir d'un *STPTIO*. Nos modifications sont les suivantes :

1. On ne calcule pas seulement les transitions du *RA*, mais dans les $\pi-$transitions on mémorise aussi les chemins de son *STPTIO* auxquels elles correspondent. Chaque $\pi-$transition du *RA* correspond à un chemin π dans son *STPTIO*.

2. Étant donné que le problème abordé dans [FWW97] est de calculer l'ensemble des localités atteignables dans un automate à pile, l'algorithme ne calcule qu'une seule ϵ-transition entre deux localités prouvant cette atteignabilité. Dans notre cas, l'objectif est de générer des cas de tests qui couvrent toutes les transitions atteignables. Donc, on va calculer plusieurs transitions ayant des chemins différents entre deux localités symboliques. On ajoute une nouvelle transition entre deux localités à chaque fois que son chemin couvre une nouvelle transition du *STPTIO*.

3. On ajoute la règle R_1 pour prendre en compte des actions de Σ car dans Finkel et al. les automates considérés ont seulement des actions de pile.

4. Les $\epsilon-$transitions réflexives ne sont pas utilisées dans [FWW97] parce qu'elles ne changent rien à l'atteignabilité des localités. Dans notre cas, on ne peut pas les ignorer en raison de la possibilité de la couverture de nouvelles transitions. On verra

que, par exemple le chemin π de la transition $l^T \xrightarrow{\pi} l^T$ peut couvrir des nouvelles transitions par rapport π' qui est le chemin de la transition $l^T \xrightarrow{\pi'} l'^T$.

Théorème 6.3.1. Soit p un contenu de la pile d'un *TPAIO*. Une localité symbolique l_i^T est atteignable avec p pour contenu de la pile dans le *STPTIO* du *TPAIO* s'il existe une π-transition (l_0^T, π, l_i^T) dans son *RA*.

Démonstration. La preuve est faite par récurrence et pour chaque règle. Notre hypothèse de récurrence est que les transitions d'un *RA* satisfont le théorème 6.3.1 avant une fusion dans une nouvelle transition. Premièrement, on prouve que les règles R_1 et R_2 qui définissent les transitions d'un *RA* à partir des transitions du *TPAIO* sont des transitions satisfaisant le théorème 6.3.1. Puis, on prouve après que les règles R_3 et R_4 conservent cette propriété sous l'hypothèse que les transitions du *RA* qu'elles fusionnent la satisfont.

- Cas de la règle R_1 : s'il existe une transition $(l_1, v_1, p_1) \xrightarrow{(l_1,v_1,p_1)\xrightarrow{A,g_1,\{y\}}(l_2,v_2,p_2)} (l_2, v_2, p_2) \in \Delta^R$, la localité symbolique (l_2, v_2, p_2) est atteignable à partir de la localité symbolique (l_1, v_1, p_1) dans le *STPTIO* parce que $v_1 \wedge g_1$ est satisfiable par definition du *STPTIO* et l'action A ne modifie pas le contenu de la pile.
- Cas de la règle R_2 : s'il existe une transition $(l_1, v_1, p_1) \xrightarrow{(l_1,v_1,p_1)\xrightarrow{a^+,g_1,\{y\}}(l_2,v_2,p_2)\xrightarrow{a^-,g_2,\{y\}}(l_3,v_3,p_3)} (l_3, v_3, p_3) \in \Delta^R$, la localité symbolique (l_3, v_3, p_3) est atteignable à partir de la localité symbolique (l_1, v_1, p_1) dans le *STPTIO* car il est toujours possible de dépiler un symbole a juste après de son empilement et $v_1 \wedge g_1$ et $v_2 \wedge g_2$ sont satisfiables par definition du *STPTIO*. Un empilement suivi par un dépilement du même symbole ne change pas le contenu de la pile.
- Cas de la règle R_3 : s'il existe une transition $(l_1, v_1, p_1) \xrightarrow{(l_1,v_1,p_1)\xrightarrow{a^+,g_1,\{y\}}(l_2,v_2,p_2)\xrightarrow{\pi}(l_3,v_3,p_3)\xrightarrow{a^-,g_3,\{y\}}(l_4,v_4,p_4)} (l_4, v_4, p_4) \in \Delta^R$, la localité symbolique (l_4, v_4) est atteignable à partir de (l_1, v_1) dans le *STPTIO* car la localité symbolique (l_3, v_3, p_3) est atteignable à partir de la localité symbolique (l_2, v_2, p_2) selon l'hypothèse de récurrence et il est toujours possible de dépiler a après son empilement et $v_1 \wedge g_1$ et $v_3 \wedge g_3$ sont satisfiables par définition du *STPTIO*.
- Cas de la règle R_4 : s'il existe une transition $(l_1, v_1, p_1) \xrightarrow{(l_1,v_1,p_1)\xrightarrow{\pi_1}(l_2,v_2,p_2)\xrightarrow{\pi_2}(l_3,v_3,p_3)} (l_3, v_3, p_3) \in \Delta^R$, la localité symbolique (l_3, v_3, p_3) est atteignable à partir de (l_1, v_1) dans le *STPTIO* car la localité symbolique (l_2, v_2, p_2) est atteignable à partir de (l_1, v_1, p_1) et la localité symbolique (l_3, v_3, p_3) est atteignable à partir de (l_2, v_2, p_2) selon l'hypothèse de récurrence.

□

En appliquant ces règles, on obtient un automate avec un nombre fini de localités symboliques, mais avec un nombre potentiellement infini de transitions. Les règles R_1 et R_2 ne produisent qu'un nombre fini de transitions pour n'importe quel *RA* calculé. Par contre, on peut calculer une infinité de transitions en appliquant les règles R_3 et R_4.

Dans ce mémoire, nous proposons une méthode de calcul qui applique un nombre fini de fois les règles R_1 à R_4 et produit un *RA* fini mais partiel. Le principe de cette méthode est le suivant : l'ajout d'une nouvelle π-transition $(l^T, \pi, l^{T'})$ est possible si et seulement si

le chemin π couvre au moins une nouvelle transition entre les localités symboliques l^T et $l^{T'}$. L'exemple 6.3.1 illustre cette méthode sur le testeur de la Fig. 6.4.

Exemple 6.3.1. Dans le *RA* du *STPTIO* de la fig. 6.4, Il existe deux transitions $((l_0, v_0, \perp), \pi, (l_6, v_0, \perp))$ qui partent de la localité symbolique $(l_0, v_0, \perp)$ et arrivent à la localité symbolique $(l_6, v_0, \perp)$ où π prend les valeurs suivantes :

1. $\pi \overset{def}{=} (l_0, v_0, \perp) \xrightarrow{?A, y\leqslant 2, \{y\}} (l_1, v_0, \perp) \xrightarrow{?B, y\leqslant 1, \{y\}} (l_2, v_0, \perp) \xrightarrow{?D, y\leqslant 2, \{y\}} (l_6, v_0, \perp)$ et
2. $\pi \overset{def}{=} (l_0, v_0, \perp) \xrightarrow{?A, y\leqslant 2, \{y\}} (l_1, v_0, \perp) \xrightarrow{?C, y\leqslant 1, \{y\}} (l_3, v_0, \perp) \xrightarrow{Fib_1^+, y\leqslant 1, \{y\}}$
$(l_0, v_1, [Fib_1]) \xrightarrow{?A, y\leqslant 2, \{y\}} (l_1, v_0, [Fib_1]) \xrightarrow{?B, y\leqslant 1, \{y\}} (l_2, v_0, [Fib_1]) \xrightarrow{?D, y\leqslant 2, \{y\}}$
$(l_6, v_0, [Fib_1]) \xrightarrow{Fib_1^-, y\leqslant 1, \{y\}} (l_4, v_1, \perp) \xrightarrow{Fib_2^+, y\leqslant 1, \{y\}} (l_0, v_2, [Fib_2]) \xrightarrow{?A, y\leqslant 2, \{y\}}$
$(l_1, v_0, [Fib_2) \xrightarrow{?B, y\leqslant 1, \{y\}} (l_2, v_0, [Fib_2]) \xrightarrow{?D, y\leqslant 2, \{y\}} (l_6, v_0, [Fib_2]) \xrightarrow{Fib_2^-, y\leqslant 1, \{y\}}$
$(l_5, v_1, \perp) \xrightarrow{?E, y\leqslant 3, \{y\}} (l_6, v_0, \perp).$

Ces deux transitions couvrent toutes les transitions du *STPTIO* qui n'atteignent pas $fail$.

6.4/ GÉNÉRATION DES CAS DE TESTS

Pour générer des tests couvrant toutes les transitions du *STPTIO*, donc du *TPAIO*, nous utilisons tous les chemins qui sont associés à toutes les π−transitions du *RA* qui partent d'une localité symbolique initiale et se terminent dans l'une des localités symboliques finales. Dans la Def. 27, nous définissons cet ensemble de chemins en y ajoutant la définition de la pile.

Définition 27 : Ensemble de chemins de test d'une *TPAIO* déterministe avec deadline *delayable*

Soit $T = (L, l_0, \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F)$ un *TPAIO* qui est une spécification, $T^T = (L^T, l_0^T, \Sigma_{in}^T, \Gamma, \{y\}, \Delta^T, F^T)$ le *STPTIO* de T et $RA = (L^R, l_0^T, (\Delta^T)^*, \Delta^R, F^T)$ le *RA* de T^T. ST est un ensemble de chemins de test défini comme suit : $\pi = l_0^T \xrightarrow{act_0, g_0, \{y\}} l_1^T \xrightarrow{act_1, g_1, \{y\}} l_2^T ... \xrightarrow{act_n, g_n, \{y\}} l_{n+1}^T$ est un chemin de test de ST si et seulement si π est le chemin qui étiquette une π-transition $l_0^T \xrightarrow{\pi} l_{n+1}^T$ de Δ^R où $l_{n+1}^T \in F^T$.

Vu que notre modèle est une abstraction d'une fonction récursive temporisée, on peut calculer les données d'entrées de cette fonction pour chaque cas de test. Alors, les cas de tests obtenus peuvent être individualisés puisqu'il y a une relation bi-univoque entre le chemin de test et les données d'entrée. Ainsi, pour chaque chemin de test, on calcule un cas de test défini par une valeur des données de la fonction récursive. Notre ensemble de cas de tests TC est défini à partir de l'ensemble de chemins de test ST de l'automate d'atteignabilité. On calcule un cas de test tc pour chaque chemin de test dans ST qui est un *TPAIO*. Ses localités sont des localités symboliques. Les verdicts sont ensuite ajoutés par remplacement de la localité cible de la dernière transition du chemin de test par $pass$ et l'ajout des transitions qui atteignent $fail$ à partir de toutes les localités différentes de $pass$. Les transitions qui atteignent $fail$ ou $inconc$ sont obtenues à partir du *STPTIO*. Définissons d'abord dans la Def. 28 un cas de test à partir d'un chemin de test donné.

Définition 28 : Cas de test d'un *TPAIO* déterministe avec deadline *delayable*

Soit $T = \langle L, l_0, \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F\rangle$ un *TPAIO* qui est une spécification, $T^T = \langle L^T, l_0^T, \Sigma_{in}^T, \Gamma, \{y\}, \Delta^T, F^T\rangle$ le *STPTIO* de T et $RA = (L^R, l_0^T, (\Delta^T)^*, \Delta^R, F^T)$ le *RA* de T^T. Un cas de test, associé à un chemin de test $\pi = (l_0, v_0, \bot) \xrightarrow{act_0,g_0,\{y\}} (l_1, v_1, p_1) \xrightarrow{act_1,g_1,\{y\}} (l_2, v_2, p_2)...\xrightarrow{act_n,g_n,\{y\}} (l_{n+1}, v_{n+1}, \bot)$, est un *TPAIO* acyclique et déterministe $tc = \langle L^{tc}, (l_0^R, \bot), \Sigma_{in}^T, \Gamma, \{y\}, \Delta^{tc}, \{pass, fail, inconc\}\rangle$ où :

- $\Delta^{tc} = \{(l_i, v_i, p_i) \xrightarrow{act_i,g_i,\{y\}} (l_{i+1}, v_{i+1}, p_{i+1})\}$ dans π pour i de 0 à $n-1\}$ $\cup\{(l_n, v_n, p_n) \xrightarrow{act_n,g_n,\{y\}} pass\} \cup \Delta_{fail}^{tc} \cup \Delta_{inconc}^{tc}$ où :
  - $\Delta_{fail}^{tc} = \{(l, v, p) \xrightarrow{act,g,\{y\}} fail \mid l^T \xrightarrow{act,g,\{y\}} fail \in \Delta^T\}$
  - $\Delta_{inconc}^{tc} = \{(l, v, p) \xrightarrow{?time,g,\{y\}} inconc \mid l^T \xrightarrow{?time,g,\{y\}} inconc \in \Delta^T\}$
- $L^{tc} = \{(l_i, v_i, p_i) \mid (l_i, v_i, p_i)$ est dans π pour $i \in 1..n\} \cup \{pass, fail, inconc\}$

Les tests générés doivent être exécutés. Comme les *TPAIO* sont des abstractions des programmes récursifs temporisés, les tests ne sont pas, dans le cas général, assuré d'être des exécutions concrètes. C'est le cas de notre exemple qui est une abstraction d'un programme récursif auquel ou a ajouté des contraintes temporelles. Pour sélectionner les cas de test concrets et ses données, on propose d'utiliser une exécution symbolique [God12] qui est une interprétation abstraite qui simule l'exécution du programme pour des données en entrée. L'évaluation de satisfiabilité d'une contrainte peut être faite par l'outil Z3 [dMB08]. Pour les contraintes qui sont satisfiables, le cas de test est concretisable et le solveur fournit une valeur des données qui satisfont les contraintes. Elles représentent les données en entrée concrètes qui engendrent l'exécution du cas de test considéré. Par exemple, le chemin de test de la Fig. 6.5(b) qui conduit à $pass$ représente la trace $ACFib_1^+ABDFib_1^-Fib_2^+ABDFib_2^-E$ qui correspond aux instructions successives suivantes : int res_1, res_2 ; $n > 1$; $res_1 = Fib(n-1)$; int res_1, res_2 ; $n - 1 \leqslant 1$; return $n-1$; $res_1 = Fib(n-2)$; int res_1, res_2 ; $n - 2 \leqslant 1$; return $n-2$ et return $res_1 + res_2$. Il correspond au prédicat de chemin suivant : $n > 1 \wedge n - 1 \leqslant 1 \wedge n - 2 \leqslant 1$. Cette contrainte est satisfiable et la solution est $n = 2$. Donc, ce cas de test correspond à l'appel $Fib(2)$. Dans notre cas, on obtient deux cas de tests qui sont concrets. Le deuxième cas de test est présenté dans la Fig. 6.5(a). Son prédicat de chemin est $n \leqslant 1$. Il correspond à l'appel $Fib(0)$ ou à l'appel $Fib(1)$. Vu que les données de test définissent un chemin d'exécution précis, l'exécution d'un cas de test est pilotée par ces données. Le chemin d'exécution est connu à l'avance et il n'est pas possible d'avoir des branchements différents avec observation d'action non prévue dans le cas de test. C'est pourquoi on n'a pas ajouté de transitions qui mèneraient à $inconc$ dans le cas d'un branchement non prévu.

6.5/ EXPÉRIMENTATION

Notre modèle est une abstraction d'un programme récursif temporisé. Pour évaluer notre méthode, on propose de muter le programme ayant servi à extraire la spécification afin d'obtenir plusieurs implémentations. Les programmes servant d'implémentations sont obtenus en introduisant des modifications (mutation) du code dans le programme ayant servi pour définir la spécification. Par exemple, on change aléatoirement un opérateur "<"

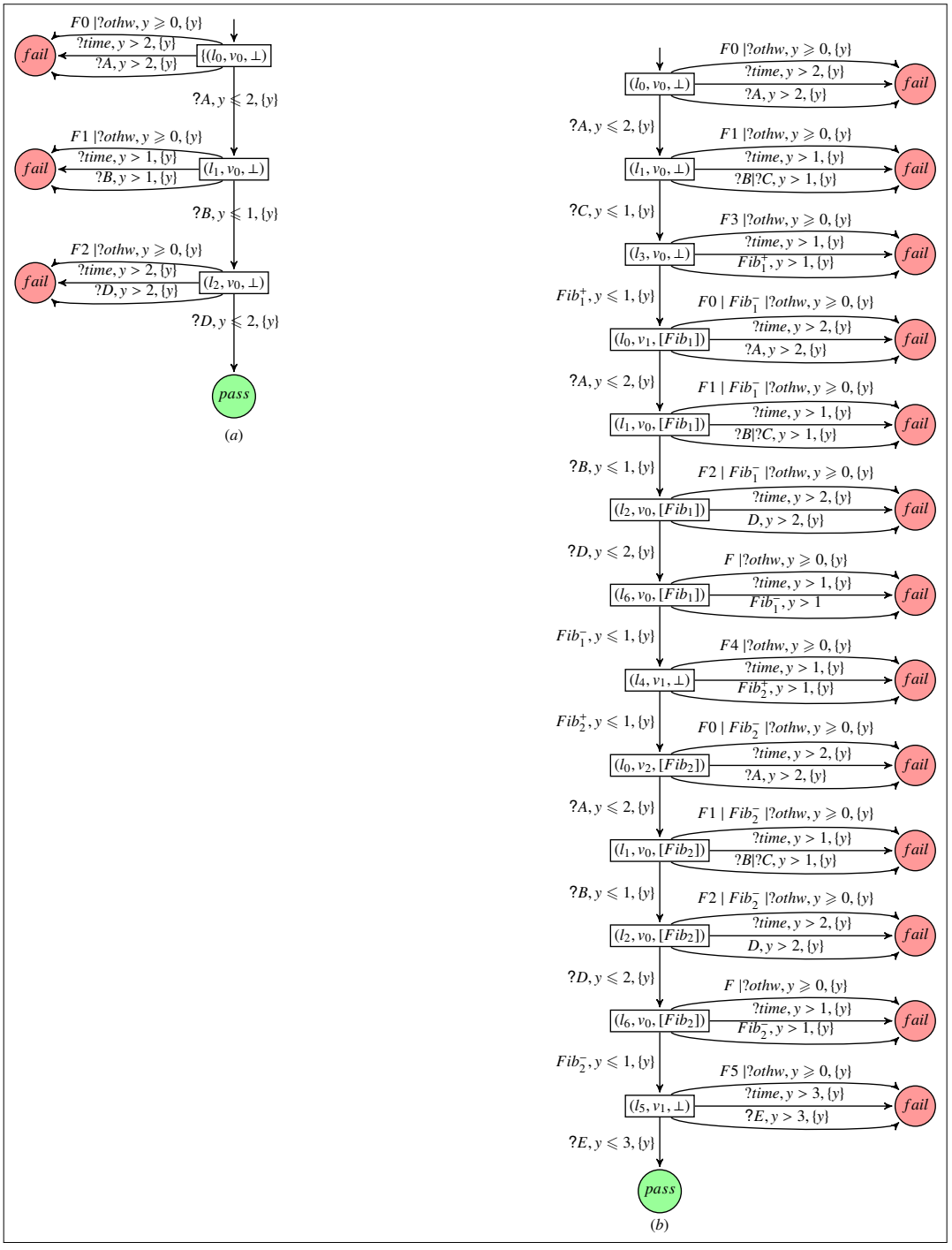


FIGURE 6.5 – Deux cas de tests du *TPAIO* de la Fig. 6.3

par " ! =" ou "> " ou " ⩽" ou "⩾ " ou "==". Il existe des outils pour muter un programme. On trouve par exemple l'outil appelé Mutate[1] qui est disponible gratuitement développé par Arun Babu pour muter des programmes. Nous avons fait des changements pour que ce programme mute des programmes écrits en java. Pour chaque donnée de test, on calcule

1. http://members.femto-st.fr/sites/femto-st.fr.pierre-cyrille-heam/files/content/VESONTIO%20-%202014/mutate.py

sa trace dans le programme muté et puis on l'exécute sur son cas de test. L'exécution d'une trace sur un cas de test donné est présentée dans l'algorithme 6.5.1 qui permet de comparer une trace d'implémentation ρ avec un cas de test tc afin d'atteindre un verdict qui peut être $pass$, $fail$ ou $inconc$. L'algorithme consiste à parcourir la trace ρ et à traiter à chaque fois ces deux situations :

- un écoulement de temps δ_i (lignes 3-8) : l'algorithme vérifie s'il existe dans tc une transition tr étiquetée par $time$ de tc qui mène à $fail$ ou $inconc$ et que la valuation d'horloges courante satisfait la garde de la transition tr. Si la condition est vérifiée, l'algorithme s'arrête et retourne $inconc$ ou $fail$. Sinon, il passe à l'état suivant par un écoulement de temps de durée δ_i.
- l'exécution d'une action a_i (lignes 9-14) : l'algorithme vérifie s'il existe une transition tr étiquetée par a_i de tc qui mène à un verdict qui peut être $pass$, $fail$ ou $inconc$ et que la valuation d'horloges courante satisfait la garde de la transition tr. Si la condition est vérifiée, l'algorithme s'arrête et retourne le verdict. Sinon, il passe à l'état suivant par l'exécution de l'action a_i.

Algorithme 6.5.1 Comparaison d'une trace de l'implémentation avec un cas de test

ENTRÉES : $T = (L, l_0, \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F\rangle$ // un *TPAIO*
$tc = \langle L^{tc}, (l_0^R, \bot), \Sigma_{in}^T, \Gamma, \{y\}, \Delta^{tc}, \{pass, fail, inconc\}\rangle$ // cas de test
$\rho = \delta_0 a_0 \delta_1 a_1 \delta_2 ... \delta_{n-1} a_{n-1} \delta_n a_n$ // une trace d'une exécution d'un chemin π de l'implémentation
SORTIES : $verdict \in \{fail, pass, inconc\}$
VARIABLES : $current \in L \times (\{y\} \to \mathbb{R}^+) \times \Gamma^*$// état courant.
$a_i \in \Sigma_{in}^T$; $p \in \Gamma^*$; $\delta \in \mathbb{R}^+$; $g \in Grd(\{y\})$; $v' \in CC(X \cup \{y\})$; $v \in (\{y\} \to \mathbb{R}^+)$; i : entier
BEGIN

1: $current \leftarrow (l_0, 0, \bot)$
2: **pour** i de 0 à n **faire**
3: // traiter δ_i
4: Soit $current = (l, v, p)$
5: **si** $\exists(v', g).((l, v', p), ?time, g, \{y\}, s') \in \Delta^{tc} \wedge v \vDash v' \wedge v + \delta_i \vDash g) \wedge s' \in \{inconc, fail\}$ **alors**
6: retourner s' // le verdict peut être $fail$ ou $inconc$
7: **fin si**
8: $current \leftarrow current \rightarrow^{\delta_i}$ // passage à l'état suivant par une transition d'écoulement du temps d'une durée δ_i
9: // traiter a_i
10: Soit $current = (l, v, p)$
11: **si** $\exists(v', g).((l, v', p), a_i, g, \{y\}, s') \in \Delta^{tc} \wedge v \vDash v' \wedge v \vDash g) \wedge s' \in \{inconc, fail, pass\}$ **alors**
12: retourner s' // le verdict peut être $fail$, $pass$ ou $inconc$
13: **fin si**
14: $current \leftarrow current \rightarrow^{a_i}$// passage à l'état suivant par une transition d'action a_i
15: **fin pour**
16: retourner $pass$

FIN.

Pour vérifier la satisfiabilité des formules des lignes 5 et 11 de l'algorithme, on utilise un outil d'évaluation de satisfiabilité de formules logiques. L'expression $v \vDash v'$ est vraie si v est une valuation d'horloge qui satisfait une contrainte v' Plusieurs outils ont été développés pendant des dernières années notamment, des solveurs SMT pour évaluer

efficacement de telles satisfiabilités. Nous avons choisi le solveur Z3 [dMB08] pour évaluer la satisfiabilité d'une formule donnée.

Instruction mutée	nouvelle instruction	n	Trace
return n	return -1	0	$0.2\ (int\ res1, res2)\ 0.5\ (n \leqslant 1)\ 0.2\ (return - 1)$
si $n \leqslant 1$	si $n \leqslant 2$	2	$0.2\ (int\ res1, res2)\ 0.5\ (n \leqslant 2)\ 0.2\ (return\ n)$

TABLE 6.1 – Résultats expérimentaux sur le *TPAIO* de la Fig. 6.3

Nous proposons dans un but d'expérimentation de muter l'algorithme 3.3.1 dans la Sec. 3.3.1 (suite de Fibonacci) afin d'avoir une implémentation non conforme à la spécification présentée dans la Fig. 6.3. Notons que le programme muté peut donner parfois une implémentation conforme. Ensuite, nous enregistrons la trace obtenue pour chaque donnée de test et nous la comparons avec le cas de test qui a la même donnée de test en utilisant l'algorithme 6.5.1. Ce travail est actuellement réalisé à la main. Ce travail est actuellement à la main. Son automatisation reste à implanter. Deux exemples d'implémentations non conformes à la spécification et détectées par notre méthode sont présentés dans le tableau 6.1. La première colonne est l'instruction mutée de l'algorithme 3.3.1. La deuxième colonne indique l'instruction après mutation. La troisième colonne est la donnée de test, c'est à dire la valeur du paramètre n définissant le cas de test. La quatrième colonne est la trace obtenue.

6.6/ CORRECTION, INCOMPLÉTUDE ET COUVERTURE DE LA MÉTHODE

Cette section discute de la correction, de l'incomplétude et de la couverture de test de notre méthode de génération de tests à partir d'un *TPAIO* déterministe avec sorties seulement.

6.6.1/ CORRECTION

Nous nous intéressons au cas du test de conformité où l'on souhaite vérifier si une implémentation est conforme à une spécification donnée. Le théorème 6.6.1 spécifie la correction de notre méthode de test en prouvant que si une exécution de l'implémentation I aboutit à une localité $fail$, c'est une exécution interdite par sa spécification T. Donc, l'implémentation I n'est pas conforme à la spécification T selon la relation de conformité $tpioco$.

Théorème 6.6.1. Soit $\pi = (l_0, v_0, \perp) \xrightarrow{a_0,g_0,\{y\}} (l_1, v_1, p_1) \xrightarrow{a_1,g_1,\{y\}} ...(l_{n-1}, v_{n-1}, p_{n-1}) \xrightarrow{a_{n-1},g_{n-1},\{y\}} (l_n, v_n, p_n) \xrightarrow{a_n,g_n,\{y\}} fail$ un chemin définissant un cas de test pour les données de test d d'une spécification $T = \langle L, l_0, \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F\rangle$ où $l_i \in L$, v_i est une contrainte d'horloge dans $CC(X \cup \{y\})$, $g_i \in Grd(\{y\})$, $p_i \in \Gamma^*$, $a_i \in \Sigma_{out} \cup \Gamma^{+-} \cup \{othw\}$ pour chaque i tel que $0 \leqslant i \leqslant n$ et $l_{n+1} = fail$. Si le verdict $fail$ est observé en exécutant l'implémentation I pour la donnée de test d, alors I n'est pas conforme à la spécification T.

Démonstration.

Atteindre $fail$ est dû à un des deux cas suivants :

1. cas d'une action de sortie non autorisée : soit $\rho = \delta_0 a_0 \delta_1 a_1 \delta_2 ... \delta_{n-1} a_{n-1} \delta_n a_n \in RT(\Sigma \cup \Gamma^{+-} \cup \{othw\})$ une trace d'une exécution d'un chemin π pour des de test d. l_n^T est la localité symbolique courante après l'exécution de la trace $\delta_0 a_0 \delta_1 a_1 \delta_2 ... \delta_{n-1} a_{n-1} \delta_n$ et g_n est la partition calculée. Atteindre $fail$ est dû à un des deux cas suivants :

   - cas d'une action de sortie existante dans la spécification mais non autorisée sur la localité courante ou à l'instant courant : $fail$ est atteint après l'observation de a_n où $a_n \in \Gamma^{+-} \cup \Sigma_{out}$ dans le cas d'une action de la pile ou d'une action de sortie non spécifiée dans la spécification ou l'observation de a_n dans le cas d'un délai non spécifié selon le cas (III) de la Def. 25 du testeur. Si $\Delta_{a_n}(l_n^T, v_n) = \emptyset$, alors, la transition $(l_n^T, a_n, v_n, \{y\}, fail)$ est une transition du *STPTIO*. Donc, $a_n \notin out(T \ after \ \delta_0 a_0 \delta_1 a_1 \delta_2 ... \delta_{n-1} a_{n-1} \delta_n)$, et I n'est pas conforme à T.

   - cas d'une action de sortie non existante dans la spécification : $fail$ est atteint après l'observation de a_n où $a_n \notin \Gamma^{+-} \cup \Sigma_{out} \cup \mathbb{R}^+$ dans le cas d'une action non autorisée dans la spécification, mais qui est exécutée par l'implémentation selon le cas (IV) de la Def. 25 du testeur. Si $\Delta_{a_n}(l_n^T, v_n) = \emptyset$, alors, la transition $(l_n^T, ?othw, y \geqslant 0, \{y\}, fail)$ est une transition du *STPTIO*. Donc, par définition de $othw \ a_n \notin out(T \ after \ \delta_0 a_0 \delta_1 a_1 \delta_2 ... \delta_{n-1} a_{n-1} \delta_n)$, et I n'est pas conforme à T.

2. cas d'un dépassement de temps : soit $\rho = \delta_0 a_0 \delta_1 a_1 \delta_2 ... \delta_{n-1} a_{n-1} \delta_n \in RT(\Sigma \cup \Gamma^{+-} \cup \{othw\})$ une trace d'une exécution d'un chemin π. (l_n, v_n, p_n) est la localité symbolique courante après l'exécution de la trace $\delta_0 a_0 \delta_1 a_1 \delta_2 ... \delta_{n-1} a_{n-1}$ et g_n est la grande partition calculée . Atteindre $fail$ est dû à un dépassement de temps : $fail$ est atteint après l'observation de $\delta_n + a_n$ où $a_n \in \mathbb{R}^+$ dans le cas où $ACV_{l_n}(v_n, p_n, v_n + \delta_n) = false$ selon le cas (I) de la Def. 25 du testeur.

□

Donc, pour chaque erreur (état $fail$) engendrée par un cas de test, il y a une non-conformité entre l'implémentation et la spécification.

6.6.2/ INCOMPLÉTUDE

La relation d'équivalence est utilisée pour calculer un *STPTIO* à partir d'un *TPAIO* donné. Mais, elle peut conduire à une perte de précision pour réduire le nombre des localités symboliques du *STPTIO*. Il devrait être possible de construire des cas de tests plus précis que ceux calculés par notre méthode. Considérons par exemple le *TPAIO* de la Fig. 6.6(a). Le *STPTIO* de la Fig. 6.6(c) est plus précis que celui de la Fig. 6.6(b). L'exécution de la trace $0a^+2a^-$ atteint la localité symbolique $(l_2, 0 \leqslant x - y < 4, \perp)$ du *STPTIO* de la Fig. 6.6(b). Il atteint aussi la localité symbolique $(l_2, 0 \leqslant x - y \leqslant 3, \perp)$ du *STPTIO* de la Fig. 6.6(c), mais pas la localité symbolique $(l_2, 0 < x - y < 4, \perp)$. On remarque que la localité symbolique $(l_2, 0 \leqslant x - y \leqslant 3)$ est plus précise que la localité symbolique $(l_2, 0 \leqslant x - y < 4, \perp)$.

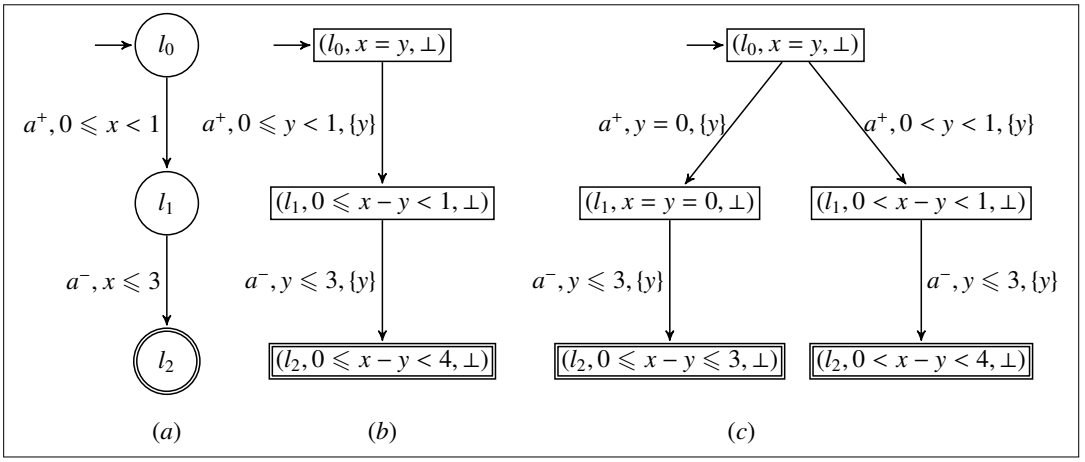


FIGURE 6.6 – Deux *STPTIO* (b) et (c) de *TPAIO* (a) (sans les localités vers $fail$). (c) est plus précis que (b)

6.6.3/ COUVERTURE DES TESTS

Dans la Sec. 6.3, on a présenté une méthode de calcul d'un *RA* à partir d'un *STPTIO* donné. Notre algorithme qui calcule ce *RA* a pour critère d'arrêt la couverture de toutes les transitions du *STPTIO*. Il ajoute une nouvelle π−transition $(l^T, \pi, l^{T'})$ si et seulement si son chemin π couvre au moins une nouvelle transition entre les localités symboliques calculées l^T et $l^{T'}$ par rapport aux autres transitions déjà calculées. Les chemins de toutes les π−transitions qui partent d'une localité initiale symbolique pour atteindre une localité finale du *STPTIO* sont pris comme cas de test. Mais, on ne peut pas affirmer que tous les cas de tests générés sont concrétisables. Donc, il n'est pas garanti que tous les cas de tests générés concrets couvrent toutes les transitions du *STPTIO*. Par contre, si tous les cas de tests générés sont concrétisables, alors, toutes les localités et toutes les transitions du *TPAIO* sont couvertes. C'est le cas de notre exemple de la Fig. 6.3.

6.7/ CONCLUSION

On a présenté dans ce chapitre une méthode de génération de tests à partir d'un programme récursif temporisé qui est modélisé par un automate à pile temporisé déterministe avec sorties seulement. La méthode consiste à définir un testeur déterministe *STPTIO* d'un *TPAIO* donné en adaptant la méthode de déterminisation de Krichen et Tripakis au cas des *TPAIO* pour ajouter le verdict $fail$ et enlever les localités symboliques non atteignables à cause de contraintes temporelles. Puis, elle calcule un automate d'atteignabilité qui est un automate fini avec des π-transitions prenant en compte les contraintes de la pile où π est un chemin du *STPTIO*. Enfin, la méthode génère des cas de tests couvrant les localités et les transitions atteignables du *TPAIO*.

Le modèle utilisé dans ce chapitre est un *TPAIO* déterministe dont toutes les deadlines sont $delayable$. On propose une généralisation de cette méthode à des *TPAIO* non déterministes avec un choix libre des deadlines dans le chapitre suivant.

7

TEST DE CONFORMITÉ À PARTIR D'UN *TPAIO* AVEC DEADLINES

Sommaire

Dans le chapitre précédent, nous avons proposé une méthode de génération de tests à partir d'un automate à pile temporisé déterministe avec sorties seulement et avec des deadlines $delayable$ seulement. On propose dans ce chapitre une méthode pour un modèle qui est un *TPAIO* non déterministe et avec deadlines quelconques. Il présente trois caractéristiques différentes par rapport au modèle utilisé dans le chapitre précédent : *TPAIO* non déterministe, prise en compte des deadlines $lazy$, $delayable$ et $eager$ et prise en compte des actions d'entrée. Les deadlines imposent des conditions de progression au temps dans des localités du *TPAIO*. On propose une méthode de génération de tests qui tient compte non seulement d'une observation d'une action non spécifiée ou non autorisée, mais qui détecte aussi les violations des *ACV* avec n'importe quelle deadline. Cette contribution a donné lieu aux publications [MJMR15d][MJMR15c].

Nous généralisons le processus de génération de tests à partir d'un *TPAIO* déterministe avec sorties seulement et deadline $delayable$ au cas des *TPAIO* non déterministes et avec entrées/sorties et trois types de deadlines. La première étape est la construction d'un *STPTIO* à partir d'un *TPAIO* donné pour résoudre les contraintes d'horloges et les contraintes de pile, déterminiser le *TPAIO* et aussi ajouter des transitions qui mènent à de nouvelles localités $fail$ et $inconc$ afin de détecter la non conformité. La deuxième étape est le calcul d'un automate d'atteignabilité *RA* à partir du *STPTIO* obtenu afin de calculer un ou plusieurs chemins entre deux localités symboliques. Cette étape a déjà été présentée dans la Sec. 6.3 du chapitre précédent. La troisième étape consiste à générer des cas de tests en utilisant chaque π-transition du *RA* qui part d'une localité symbolique initiale vers une localité symbolique finale. Les cas de tests sont individualisables et individualisés pour vérifier si l'une des exécutions d'un modèle d'implémentation conduit à $fail$. Dans le cadre d'une exécution des tests sur une véritable implémentation, il est préférable de considérer un arbre de tests afin de poursuivre l'observation le plus loin

possible, selon les sorties de l'implémentation, pour limiter les verdicts inconclusifs. Nous évaluons la capacité de notre méthode à détecter des implémentations non-conformes obtenues par une technique de mutation qui permet de modifier automatiquement un *TPAIO* donné par l'application d'un opérateur de mutation. Nos résultats expérimentaux montrent que la grande majorité des mutants non-conformes sont détectés.

On applique le même processus de génération de tests que celui présenté dans le chapitre précédent. On présente seulement les modifications liées aux deadlines sur les deux étapes modifiées : construction du *STPTIO* et génération de tests. La première étape est la construction d'un *STPTIO* à partir d'un *TPAIO*. Elle est présentée dans la Sec. 7.1. La Sec. 7.2 présente la dernière étape qui est la génération de tests. La Sec. 7.3 discute de la correction de notre méthode. Nous définissons des opérateurs de mutation et le produit synchronisé qui modélise l'exécution d'un test sur un mutant utilisé comme modèle d'implémentation dans la Sec. 7.4. Cette section présente aussi nos résultats expérimentaux. Enfin, nous concluons en Sec. 7.6.

7.1/ CONSTRUCTION D'UN *STPTIO* À PARTIR D'UN *TPAIO*

Définition 29 : Testeur symbolique à pile temporisé déterministe *STPTIO*

Un *STPTIO* $T^T = \langle L^T, l_0^T, \Sigma_{out}^T \cup \Sigma_{in}^T, \Gamma, \{y\}, \Delta^T, F^T \rangle$ d'un *TPAIO* $T = \langle L, l_0, \Sigma_{out} \cup \Sigma_{in} \cup \{\tau\}, \Gamma, X, \Delta, F \rangle$ est un *TPAIO* à horloge unique y, qui est différente des horloges de X, où :

- l_0^T est la localité symbolique initiale,
- $L^T \subseteq 2^{(L \times CC(X \cup \{y\}) \times \Gamma^*)} \cup \{fail, inconc\}$ est un ensemble de localités symboliques,
- $\Sigma_{in}^T = \Sigma_{out} \cup \{othw\} \cup \{time\}$,
- $\Sigma_{out}^T = \Sigma_{in}$,
- $F^T = \{l^T \mid \exists (l, v, p).((l, v, p) \in l^T \wedge l \in F)\}$ est un ensemble de localités symboliques finales,
- $\Delta^T \subseteq L^T \times \Sigma_{out}^T \cup \Sigma_{in} \cup \Gamma^{+-} \times Grd(\{y\}) \times \{y\} \times L^T$ est un ensemble fini de transitions.

On propose d'adapter la méthode de [KT09] définie pour les *TAIO* non déterministes avec deadlines pour calculer un testeur à partir d'un *TPAIO* non déterministe avec deadlines. Le testeur calculé est appelé *STPTIO* (pour Symbolic Timed Pushdown Tester with Inputs and Outputs) qui est un observateur des implémentations testées. Il intègre des localités symboliques, des nouvelles localités $fail$ et $inconc$ et des transitions qui les atteignent à partir d'une autre localité symbolique. C'est un *TPAIO* avec une seule horloge qui se remet à zéro à chaque franchissement d'une action, et dont toutes les transitions ont la deadline *lazy*. N'ayant qu'une seule horloge, toutes les gardes de ce testeur sont satisfiables. Afin de détecter la non-conformité entre une implémentation et un *TPAIO*, les actions d'entrées du *STPTIO* sont les actions de sorties (Σ_{out}) du *TPAIO*, les actions de sorties du *STPTIO* sont les actions d'entrées (Σ_{in}) du *TPAIO*, et les actions de la pile sont celles de la pile (Γ) du *TPAIO*. Nous notons $othw$ un nom d'action quelconque différent de chaque nom de Σ_{out}, de Γ^{+-} et de τ et $time$ une action pour observer l'écoulement du temps. Les localités symboliques d'un *STPTIO* sont des

couples (localité du *TPAIO*, contraintes). Les contraintes portent sur les horloges du *TPAIO* et sur la nouvelle horloge y du *STPTIO*.
Un *STPTIO* est formellement défini dans la Def. 29.

Les localités sont dites symboliques car elles représentent un ensemble de localités pour l'ensemble des valeurs d'horloges qui satisfont la contrainte. La localité $fail$ est atteignable selon l'une de ces trois possibilités :

- observation d'une action de la pile (Γ^{+-}) ou de sortie (Σ_{out}) non spécifiée ou arrivant plus tôt ou plus tard que spécifiée,
- observation d'un instant ne satisfaisant pas l'*ACV* dans l'état courant,
- observation d'une action non autorisée dans T, mais qui peut être exécutée par l'implémentation.

Le système est parfois autorisé à laisser le temps s'écouler indéfiniment, par exemple dans les localités où les deadlines de toutes les transitions sortantes sont $lazy$. On propose dans ce cas d'ajouter des transitions qui mènent à $inconc$ afin d'éviter l'attente infinie et le blocage de l'observation dans une localité. La localité $inconc$ est atteignable en cas de non observation d'une action de sortie ou de la pile au cours d'un délai arbitrairement fixé pour l'arrivée des actions de la pile et de sortie dans la localité courante. Ce cas de figure est inconclusif car étant dans une localité où le temps peut s'écouler indéfiniment, passé le délai où des actions peuvent se produire, s'il n'est pas arrivé d'action, on ne sait pas s'il n'en arrivera jamais (cas $pass$) où s'il y en arrivera une plus tard (cas $fail$). Donc, un délai maximum d'observation de l'implémentation est choisi arbitrairement, mais, il doit être supérieur au plus grand délai d'arrivée d'une action autorisée.

Avant de présenter les définitions des transitions d'un *STPTIO* à partir d'un *TPAIO*, on a besoin de redéfinir les notations suivantes pour le cas des *STPTIO* :

- $usucc(l^T) = \{l^{T'} \mid \exists(l,v,p).((l,v,p) \in l^T \wedge \exists\rho.(\rho \in RT(\{\tau\}) \wedge (l,v,p) \rightarrow^{\rho} (l',v',p) \wedge (l',v',p) \in l^{T'}))\}$. C'est l'ensemble des localités symboliques atteignables à partir de la localité symbolique l^T par une séquence d'actions non observables.
- $dsucc(l^T,a) = \{l^{T'} \mid \exists(l,v,p).((l,v,p) \in l^T \wedge (l,v,p) \rightarrow^{a} (l',v',p') \wedge (l',v',p') \in l^{T'})\}$. C'est l'ensemble de localités symboliques atteignables à partir de l^T par l'action a.

On a besoin aussi de définir la notation suivante :

- $tsucc(l^T,u) = \{l^{T'} \mid \exists(l,v,p).((l,v,p) \in l^T \wedge (l,v,p) \rightarrow^{u} (l,v+u,p) \wedge (l,v+u,p) \in l^{T'})\}$ est l'ensemble de localités symboliques atteignables à partir de l^T par l'écoulement de temps avec un intervalle u.

Formalisons maintenant la notion de transition du *STPTIO* dans la Def. 30.
La localité symbolique initiale $l_0^{X\cup\{y\}}$ est égale à $usucc(l_0^T)$ où toutes les horloges de T et l'horloge y sont initialisées à zéro. Le principe de calcul des localités symboliques, des transitions entre localités symboliques et des transitions du *STPTIO* et de répéter les étapes suivantes : sélection d'une localité qui n'est pas encore ajoutée dans L^T et puis application de l'une de ces possibilités pour ajouter des nouvelles transitions à Δ^T :

1. La transition $(l^T, ?othw, y \geqslant 0, \{y\}, fail)$ est ajoutée à Δ^T.

2. La transition $(l^T, ?time, y > K, \{y\}, inconc)$ est ajoutée à Δ^T dans le cas où le temps peut s'écouler indéfiniment dans la localité symbolique l^T et où K est la durée maximale d'observation qui est choisie arbitrairement par l'ingénieur de test comme

étant supérieure à toutes les valeurs qui apparaissent dans les gardes de son *TPAIO*.

3. La transition $(l^T, ?time, u, \{y\}, fail)$ est ajoutée dans Δ^T dans le cas d'un dépassement de temps dans la localité de l^T dû à la violation d'*ACV* si $\forall(l, v, p) \in l^T.(ACV_l(v,p,v + u) = false)$, c'est à dire que le temps ne peut pas s'écouler au delà de l'intervalle de temps u dans chaque $(l, v, p) \in l^T$

4. Pour chaque grande partition u calculée, on applique les pas suivants :
- actions d'entrées : pour chaque action d'entrée $A \in \Sigma_{in}$ si $usucc(dsucc(l^T \cap u, A) \neq \emptyset$, alors la transition $(l^T, A, u, \{y\}, usucc(dsucc(l^T \cap u, A)))$ est ajoutée à Δ^T ;
- actions de sorties ou de pile : pour chaque action $a \in \Sigma_{out} \cup \Gamma^{+-}$, si $usucc(dsucc(l^T \cap u, a)) = \emptyset$, alors la transition $(l^T, a, u, \{y\}, fail)$ est ajoutée à Δ^T. Sinon, la transition $(l^T, a, u, \{y\}, usucc(dsucc(l^T \cap u, a)))$ est ajoutée à Δ^T.

Définition 30 : Transitions du Testeur à pile temporisé *STPTIO*

Soit K un délai d'observation des implémentations qui soit supérieur au plus grand délai de déclenchement d'une action quelconque quand le temps peut s'écouler indéfiniment. Un *STPTIO* $T^T = \langle L^T, l_0^T, \Sigma_{out}^T \cup \Sigma_{in}^T, \Gamma, \{y\}, \Delta^T, F^T \rangle$ d'un *TPAIO* $T = \langle L, l_0, \Sigma_{out} \cup \Sigma_{in} \cup \{\tau\}, \Gamma, X, \Delta, F \rangle$ non déterministe est un *TPAIO*. Les transitions de Δ^T sont définies comme suit :

(I) $(l^T, ?othw, y \geqslant 0, \{y\}, fail) \in \Delta^T$.

(II) $(l^T, ?time, y > K, lazy, \{y\}, inconc) \in \Delta^T$ si $tsucc(l^T, y \geqslant 0) \neq \emptyset$.

(III) $(l^T, ?time, u, \{y\}, fail) \in \Delta^T$ si $tsucc(l^T, u) = \emptyset$.

(IV) $(l^T, act, u, \{y\}, usucc(dsucc(l^T \cap u, act))) \in \Delta^T$ si $usucc(dsucc(l^T \cap u, act)) \neq \emptyset$ et $act \in \Sigma_{in} \cup \Sigma_{out} \cup \Gamma^{+-}$,

(V) $(l^T, act, u, \{y\}, fail) \in \Delta^T$ si $usucc(dsucc(l^T \cap u, act)) = \emptyset$ et $act \in \Sigma_{out} \cup \Gamma^{+-}$.

L'application répétitive du cas (IV) de la Def. 30 peut produire un nombre infini de transitions et de localités. On propose d'utiliser comme dans le chapitre précédent une variable afin de limiter le nombre d'empilement possible. Les différences entre la méthode de calcul du testeur dans le chapitre précédent et dans ce chapitre sont :
- une localité symbolique est un ensemble de triplets (l, v, p) au lieu d'un seul triplet au chapitre 6 où l est une localité, v est une valuation d'horloges p le contenu de la pile.
- le testeur de ce chapitre peut avoir des transitions étiquetées par une action de sortie, ce qui n'est pas le cas dans le chapitre 6 qui calcule un testeur avec des transitions étiquetées par une action d'entrée seulement ;
- le calcul de successeur de chaque localité symbolique tient compte non seulement de la deadline *delayable* mais également des deadlines *lazy* et *eager*.

À partir du *STPTIO* engendré par cette étape, un *RA* est calculé comme au chapitre précédent (voir Sec. 6.3)

Exemple 7.1.1. La Fig. 7.1 présente un exemple d'un *TPAIO* non déterministe T où $\Sigma_{in} = \{A\}$, $\Sigma_{out} = \emptyset$ et $\Gamma = \{a\}$.

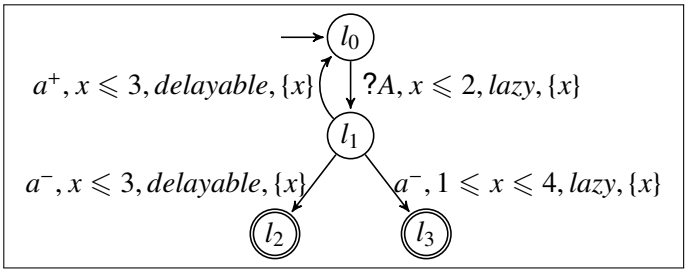


FIGURE 7.1 – Exemple d'un *TPAIO* non déterministe

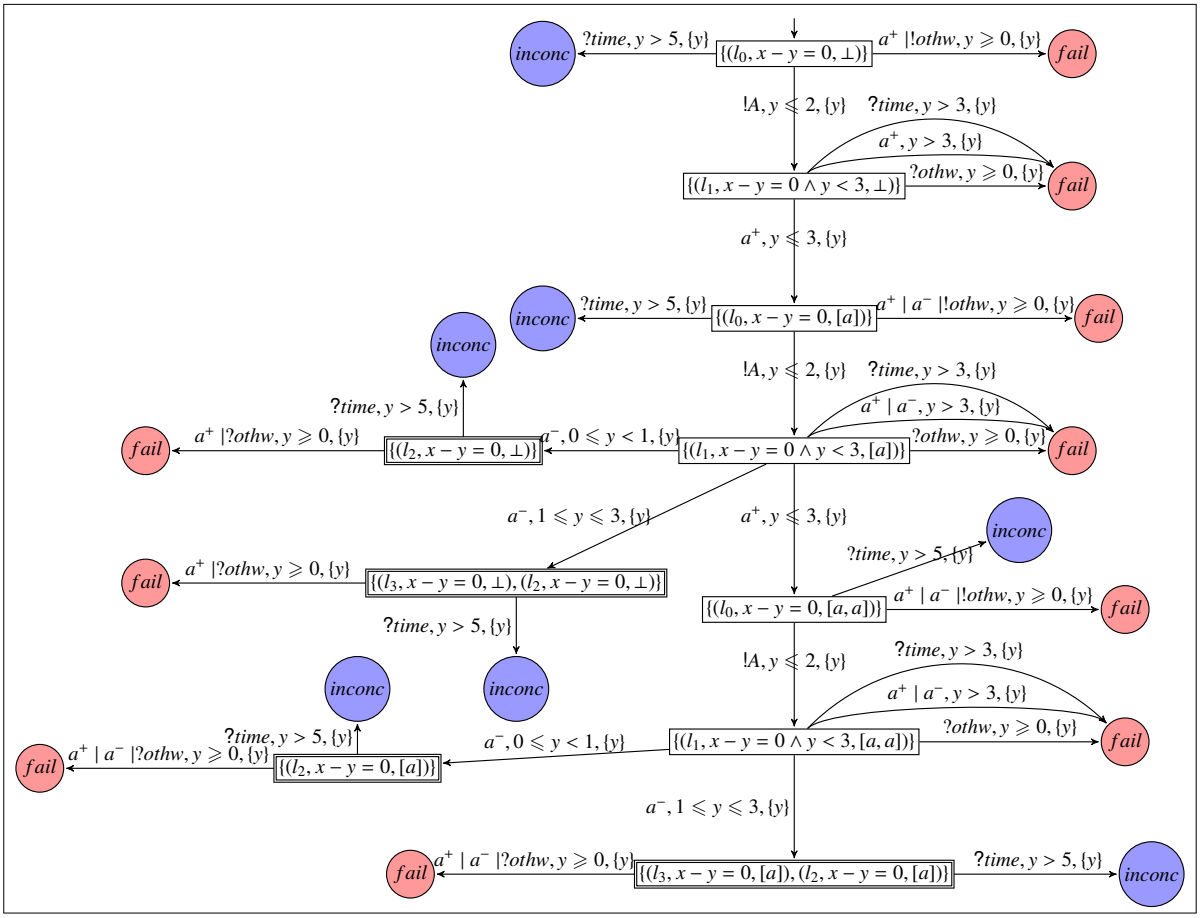


FIGURE 7.2 – Testeur (*STPTIO*) associé au *TPAIO* de la Fig. 7.1

La Fig. 7.2 présente le *STPTIO* T^T du *TPAIO* présenté dans la Fig. 7.1. La localité symbolique initiale est $l_0^T = \{(l_0, x - y = 0, \bot)\}$. Il n'existe aucune transition qui quitte la localité symbolique l_0^T avec l'étiquette *time* car toutes transitions qui quittent l_0 de T sont avec la deadline *lazy* et l'*ACV* de la localité l_0 est *true*. Alors, il n'y a pas de dépassement de temps dans l_0^T et le temps peut s'écouler indéfiniment dans la localité l_0. La localité symbolique $usucc(dsucc(\{l_0^T\}, A))$ est égale à $l_1^T = \{(l_1, x - y = 0 \wedge y \leqslant 3, \bot)\}$ parce que T n'accepte pas la transition d'écoulement de temps $(l_1, x = 0, \bot) \rightarrow^t$ où $t > 3$ vu qu'il existe les transitions $(l_1, a^-, x \leqslant 3, delayable, \emptyset, l_2)$ et $(l_1, a^+, x \leqslant 3, delayable, \{x\}, l_0)$. Il existe trois transitions avec les gardes $0 \leqslant y < 1$, $1 \leqslant y \leqslant 3$ et $y \leqslant 3$ à partir de la localité symbolique

l_1^T. Puisque l'observation de a^- ou de a^+ n'est plus autorisé après trois unités de temps, alors, la transition $(\{(l_1, x-y=0 \wedge y \leqslant 3, \perp)\}, ?time, y>3, lazy, \{y\}, fail)$ est ajoutée à Δ^T, selon le cas (III) de la Def.30 des transitions du *STPTIO*. Il n'y a pas de transition qui quitte la localité symbolique l_1^T avec l'étiquette a^- et la garde $3 < y \leqslant 4$, parce que la transition $(l_1^T, ?time, y>3, lazy, \{y\}, fail)$ est dans Δ^T. Comme il n'existe aucune transition étiquetée par a^+ qui quitte l_2, alors la transition $(\{(l_2, x-y=0, [a])\}, a^+, y \geqslant 0, lazy, \{y\}, fail)$ est ajoutée à Δ^T selon le cas (V) de la Def.30 des transitions du *STPTIO*. La transition $(\{(l_2, x-y=0, [a])\}, ?othw, y \geqslant 0, lazy, \{y\}, fail)$ est dans Δ^T selon le cas (I) de la Def.30 des transitions du *STPTIO*. Le délai d'attente K est fixé arbitrairement à 5 unités de temps, ce qui est supérieur à 4 qui est le délai maximum de déclenchement d'une action dans les localités l_0, l_2 et l_3 où le temps peut s'écouler indéfiniment. Pour satisfaire le cas (II) de la Def.30, le *STPTIO* contient par exemple la transition $(\{(l_0, x-y=0, [a])\}, ?time, y>5, lazy, \{y\}, inconc)$ car le temps peut s'écouler indéfiniment dans la localité l_4.

7.2/ GÉNÉRATION D'ARBRES DE TEST

Définition 31 : Cas de test d'un *TPAIO* non déterministe avec deadline

Soit $T = \langle L, l_0, \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F\rangle$ un *TPAIO* qui est une spécification, $T^T = \langle L^T, l_0^T, \Sigma_{in}^T, \Gamma, \{y\}, \Delta^T, F^T\rangle$ le *STPTIO* de T et $RA = \langle L^R, l_0^T, (\Delta^T)^*, \Delta^R, F^T\rangle$ le *RA* de T^T. Un cas de test d'un chemin de test $\pi = l_0^T \xrightarrow{act_0,g_0,\{y\}} l_1^T \xrightarrow{act_1,g_1,\{y\}} l_2^T ... \xrightarrow{act_n,g_n,\{y\}} l_{n+1}^T$ est un *TPAIO* acyclique et déterministe $tc = \langle L^{tc}, l_0^R, \Sigma^T, \Gamma, \{y\}, \Delta^{tc}, \{pass, fail, inconc\}\rangle$ où :

- $\Delta^{tc} = \{l_i^T \xrightarrow{act_i,g_i,\{y\}} l_{i+1}^T\}$ dans π pour $i \in 0$ à $n-1 \cup \{l_n^T \xrightarrow{act_n,g_n,\{y\}} pass\}$ $\cup \Delta_{fail}^{tc} \cup \Delta_{inconc}^{tc}$ où :
  - $\Delta_{fail}^{tc} = \{l^T \xrightarrow{act,g,\{y\}} fail \mid l^T \xrightarrow{act,g,\{y\}} fail \in \Delta^T\}$,
  - $\Delta_{inconc}^{tc} = \{l^T \xrightarrow{?time,g,\{y\}} inconc \mid l^T \xrightarrow{?time,g,\{y\}} inconc \in \Delta^T\} \cup \{l^T \xrightarrow{act,g,d,\{y\}} inconc \mid \exists l^{T\prime}.(l^{T\prime} \in L^R \wedge l^T \xrightarrow{act,g,lazy,\{y\}} l^{T\prime} \in \Delta^T \wedge act \in \Sigma \cup \Gamma^{+-} \wedge \neg\exists l^{T\prime}.(l^T \xrightarrow{act,g,d,X'} l^{T\prime} \in \Delta^{tc})\}$
- $L^{tc} = \{l_i^T \mid l_i^T$ est dans π pour $i \in 1..n\} \cup \{pass, fail, inconc\}$

Pour générer des tests couvrant toutes les transitions du *STPTIO*, donc du *TPAIO*, nous utilisons tous les chemins qui sont associés à toutes les π–transitions du *RA* qui partent d'une localité symbolique initiale et se terminent dans l'une des localités symboliques finales. L'ensemble de chemins de test est défini de manière identique à celle du chapitre précédent par la Def. 27.

Pour évaluer l'efficacité de la méthode par mutation en utilisant le produit synchronisé d'un test et d'un modèle d'implémentation, on peut considérer les cas de tests individuellement pour vérifier si l'une des exécutions du modèle de l'implémentation conduit à $fail$. C'est pour cela que l'on définit pour chaque chemin de test un cas de test. Notre ensemble de cas de tests TC est défini à partir de l'ensemble des chemins de test ST

de l'automate d'atteignabilité. On calcule un cas de test tc pour chaque chemin de test dans ST qui est un *TPAIO*. Ses localités sont des localités symboliques. Les verdicts sont ensuite ajoutés par remplacement de la localité cible de la dernière transition du chemin de test par $pass$ et l'ajout des transitions qui atteignent $fail$ et $inconc$ à partir de toutes les localités différentes de $pass$. Les transitions qui atteignent $fail$ et $inconc$ sont obtenues à partir du *STPTIO*.

La différence entre un arbre de test calculé dans le chapitre 6 et dans ce chapitre est qu'on prend en compte l'observation d'une action autorisée dans la spécification et pas dans le cas de test. Donc, on propose d'arrêter l'exécution du cas de test en atteignant le verdict $inconc$. On propose d'ajouter des transitions dans l'arbre de tests qui mènent à $inconc$ qui ajoutent des transitions décrites dans la spécification et qui n'existent pas dans le cas de test calculé. Ses transitions tiennent compte des contraintes de la pile. On ne peut pas effectuer de transition de dépilement qui mène à $inconc$ et qui dépile un symbole qui n'existe pas au sommet de la pile.
Définissons d'abord dans la Def. 31 un cas de test à partir d'un chemin de test donné.

7.3/ CORRECTION DE LA MÉTHODE

Nous nous intéressons au cas du test de conformité pour lequel on souhaite vérifier si une implémentation est conforme à une spécification donnée. Le théorème 7.3.1 formule la correction de notre méthode de test en prouvant que si une exécution de l'implémentation I aboutit à une localité $fail$, alors c'est une exécution interdite par sa spécification T. Donc, l'implémentation I n'est pas conforme à la spécification T selon la relation de conformité $tpioco$.

Théorème 7.3.1. $\pi = l_0^T \xrightarrow{a_0,g_0,\{y\}} l_1^T \xrightarrow{a_1,g_1,\{y\}} ...l_{n-1}^T \xrightarrow{a_{n-1},g_{n-1},\{y\}} l_n^T \xrightarrow{a_n,g_n,\{y\}} fail$ un chemin d'un cas de test d'une spécification $T = \langle L, l_0, \Sigma \cup \{\tau\}, \Gamma, X, \Delta, F\rangle$ où $l_i^T \in 2^{(L \times CC(X \cup \{y\}) \times \Gamma^*)}$, $g_i \in Grd(\{y\})$, $a_i \in \Sigma \cup \Gamma^{+-} \cup \{othw\}$ pour chaque $0 \leqslant i \leqslant n$. Si le verdict $fail$ est observé en exécutant l'implémentation I, alors I n'est pas conforme à la spécification T.

Démonstration. Soit $\rho = \delta_0 a_0 \delta_1 a_1 \delta_2 ... \delta_{n-1} a_{n-1} \delta_n a_n \in RT(\Sigma \cup \Gamma^{+-} \cup \{othw\})$ une trace d'une exécution du chemin π. l_n^T est la localité symbolique courante après l'exécution de la trace $\delta_0 a_0 \delta_1 a_1 \delta_2 ... \delta_{n-1} a_{n-1} \delta_n$ et g_n est la grande partition calculée. Atteindre $fail$ est dû à un des trois cas qui sont expliqués dans la Sec. 6.6 sauf que les notations $usucc$, $dsucc$ et $\Delta_a(l^T, u)$ sont définies dans ce chapitre pour un ensemble des localités symboliques □

Donc, pour chaque non conformité détectée par un cas de test, il y a une non-conformité entre l'implémentation et la spécification.

7.4/ EXPÉRIMENTATION

Nous présentons dans cette section une méthode pour évaluer les capacités de notre méthode à détecter des implémentations non-conformes. Pour cela, nous appliquons une approche de mutation qui procède par l'obtention d'un modèle d'une implémentation I en appliquant un opérateur de mutation à une spécification donnée T. Ensuite, nous

essayons de détecter si l'implémentation I est conforme ou non à T en calculant le produit synchronisé de I et des cas de tests engendrés à partir de T. Nous évaluons le nombre de mutants non-conformes détectés par notre méthode. Le rapport entre le nombre de mutants non-conformes détectés et le nombre de mutants non conformes évalue l'efficacité de notre méthode de génération de tests.

Dans la partie 7.4.1, nous définissons des opérateurs de mutation permettant de générer des mutants à partir d'une spécification donnée. Nous définissons le produit synchronisé d'un cas de test et d'un mutant dans la partie 7.4.2. Ensuite, nous évaluons les capacités de notre méthode à détecter la non conformité dans la partie 7.4.3.

7.4.1/ OPÉRATEURS DE MUTATION D'UN *TPAIO*

Soit $T = \langle L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F\rangle$ un *TPAIO*, les mutants sont définis en appliquant un des opérateurs suivants :

- **Changer action** : cet opérateur change une seule transition de Δ par remplacement de son label par un autre. Le changement est opéré soit sur une action de sortie, soit sur une action de la pile. Le mutant obtenu est $M = (L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta \setminus \{t\} \cup \{t_M\}, F)$, sachant que $t = (l, a, g, d, X', l') \in \Delta$, $t_M = (l, b, g, d, X', l')$, $b \in \Sigma_{out} \cup \Gamma^{+-}$ et $a \neq b$.

- **Changer garde** : cet opérateur change une seule transition de T par remplacement d'un opérateur relationnel dans sa garde. Le mutant obtenu est $M = (L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta \setminus \{t\} \cup \{t_M\}, F)$, sachant que $t = (l, a, g, d, X', l') \in \Delta$, $t_M = (l, a, g_m, d, X', l')$ où $a \in \Sigma_{out} \cup \Gamma^{+-}$, $g = \bigwedge_i x_i \sharp_i n_i \wedge x \sharp n$, $g_m = \bigwedge_i x_i \sharp_i n_i \wedge x \sharp_m n$, $\sharp, \sharp_i, \sharp_m \in \{<, \leq, >, \geq, =\}$ et $\sharp_m \neq \sharp$.

- **Nier garde** : cet opérateur change une seule transition de T par remplacement de sa garde par sa négation. Le mutant obtenu est $M = (L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta \setminus \{t\} \cup \{t_M\}, F)$, où $t = (l, a, g, d, X', l') \in \Delta$, $t_M = (l, a, \neg g, d, X', l')$ et $a \in \Sigma_{out} \cup \Gamma^{+-}$.

- **Changer source** : cet opérateur change seulement une transition avec une action de sortie ou de la pile de T par remplacement de sa localité source par une autre. Le mutant obtenu est $M = (L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta \setminus \{t\} \cup \{t_M\}, F)$, où $t = (l, a, g, d, X', l') \in \Delta$, $t_M = (l_m, a, g, d, X', l')$, $l \neq l_m$ et $a \in \Sigma_{out} \cup \Gamma^{+-}$.

- **Changer cible** : cet opérateur change une seule transition de Δ par remplacement de sa localité cible par une autre. Le mutant obtenu est $M = (L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta \setminus \{t\} \cup \{t_M\}, F)$, sachant que $t = (l, a, g, d, X', l') \in \Delta$, $t_M = (l, a, g, d, X', l'_m)$ et $l' \neq l'_m$.

- **Changer deadline** : cet opérateur change une seule transition avec une action de sortie ou de pile de Δ par remplacement de sa deadline par une autre. Le mutant obtenu est $M = (L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta \setminus \{t\} \cup \{t_M\}, F)$, sachant que $t = (l, a, g, d, X', l') \in \Delta$, $t_M = (l, a, g, d', X', l')$, $d' \in \{lazy, delayable, eager\}$, $d \neq d'$ et $act \in \Sigma_{out} \cup \Gamma^{+-}$.

7.4.2/ PRODUIT SYNCHRONISÉ D'UNE IMPLÉMENTATION I ET D'UN CAS DE TEST

Une implémentation *I* n'est pas conforme à une spécification si elle exécute un chemin d'un cas de test qui atteint $fail$. Pour chaque cas de test tc de TC, la non conformité d'une implémentation *I* avec une spécification T peut être détectée par le calcul du produit synchronisé de *I* et du cas de test tc. Le produit synchronisé définit l'intersection des langages de tc et de I. Soit $\mathcal{L}(tc(T))$ l'ensemble des exécutions de $tc(T)$ atteignant le verdict $pass$ ou $fail$, $\mathcal{L}(I)$ l'ensemble des exécutions de I. Alors, l'intersection $\mathcal{L}(tc(T)) \bigcap \mathcal{L}(I)$ définit l'ensemble des exécutions communes entre tc et I. Si l'une des exécutions atteint le verdict $fail$, alors l'implémentation I n'est pas conforme à sa spécification T. Le modèle d'implémentation I est défini dans la Def. 32.

Définition 32 : Modèle d'implémentation

Soit $T^M = \langle L^M, l_0^M, \Sigma_{out}^M \cup \Sigma_{in}^M, \Gamma, \{z\}, \Delta^M, F^M \rangle$ un *STPTIO* d'un *TPAIO* $M = \langle L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F \rangle$. Une implémentation modélisée par M est le *TPAIO* $I = \langle L^M \setminus \{fail\}, l_0^M, \Sigma_{in} \cup \Sigma_{out} \cup \{time\}, \Gamma, \{z\}, \Delta^I, F^M \rangle$ sachant que $\Delta^I = (\Delta^M \setminus \Delta^M_{\rightarrow fail}) \cup \Delta^I_{time}$ où $\Delta^M_{\rightarrow fail} = \{(l^M, act, g, lazy, \{z\}, fail) \in \Delta^M\}$ et $\Delta^I_{time} = \{(l^M, !time, \neg g, lazy, \{z\}, l^M) \mid (l^M, ?time, g, lazy, \{z\}, fail) \in \Delta^M\} \cup \{(l^M, !time, true, lazy, \{z\}, l^M) \mid \exists g.(l^M, ?time, g, lazy, \{z\}, fail) \in \Delta^M\}$.

Une transition $(l^{tc}, ?time, g, \{y\}, \textit{fail})$ dans tc modélise une violation de l'*ACV* lorsque la valuation d'horloges satisfait la garde g. Mais, si on suppose qu'un *TPAIO* $M = \langle L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F \rangle$ est un mutant d'une spécification, il ne possède pas de transition $(l^{tc}, !time, g, \{y\}, \textit{fail})$ étiquetée par $!time$ pouvant se synchroniser avec un cas de test. Autrement dit, le temps pourrait s'écouler indéfiniment dans une localité l de M sans être détecté, bien que, ce ne soit pas autorisé dans tc à cause des transitions avec deadlines $eager$ ou $delayable$ qui quittent l. Pour remédier à cette situation, nous proposons dans la Def. 32 de calculer une implémentation à partir d'un mutant dans le but de modéliser une *ACV* spécifiant comment le temps peut s'écouler dans n'importe quelle localité. Soit $M = \langle L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma, X, \Delta, F \rangle$ un mutant et $T^M = \langle L^M, l_0^M, \Sigma_{out}^M \cup \Sigma_{in}^M, \Gamma, \{z\}, \Delta^M, F^M \rangle$ son *STPTIO*. L'implémentation I de M est un *TPAIO* où $L^M \setminus \{fail\}$ est son ensemble de localités, l_0^M est sa localité initiale, Σ_{out} est ses actions de sorties, Σ_{in} est ses actions d'entrées , Γ est son alphabet de pile, z est une nouvelle horloge ($z \notin X$) et ses transitions Δ^M sont les transitions de Δ^M l'exclusion des transitions de Δ^M qui atteignent la localité $fail$ et l'inclusion de transitions réflexives étiquetées par $!time$ qui modélisent la condition d'écoulement du temps dans chaque localité. Une transition $(l^M, ?time, g, lazy, \{y\}, \textit{fail})$ dans Δ^M signifie que le temps ne peut pas s'écouler dans l^M si la garde g est satisfiable. Le temps ne peut s'écouler dans l^M que si $\neg g$ la négation de la garde est satisfiable. La condition $\neg g$ représente la condition d'écoulement de temps dans la localité symbolique l^M. Donc, si la transition $(l^M, ?time, g, lazy, \{z\}, \textit{fail})$ est ajoutée à Δ^M, alors, la transition $(l^M, !time, \neg g, lazy, \{z\}, l^M)$ est ajoutée à Δ^I. Mais, s'il n'existe aucune transition étiquetée avec $time$ qui part de la localité initiale l^M dans Δ^M, cela signifie que le temps peut s'écouler indéfiniment dans l^M, alors, la transition $(l^M, !time, true, lazy, \{z\}, l^M)$ est dans Δ^I.

Exemple 7.4.1. La Fig. 7.3.(b) présente le *STPTIO* d'un exemple d'un mutant présenté dans la Fig. 7.3.(a) où z est une nouvelle horloge et $v_0, .., v_n$ sont des valuations d'horloges. La Fig. 7.3.(c) présente l'implémentation du mutant présenté dans la Fig. 7.3.(a). Par exemple, le temps ne peut pas s'écouler au delà de 3 unités de temps dans la localité l_1

du mutant. C'est pourquoi nous trouvons la transition $(\{(l_1, v_1, \perp)\}, ?time, z > 3, lazy, \{z\}, fail)$ dans le *STPTIO* et $(\{(l_1, v_1, \perp)\}, !time, z \leqslant 3, lazy, \{z\}, \{(l_1, v_1, \perp)\})$ dans l'implémentation.

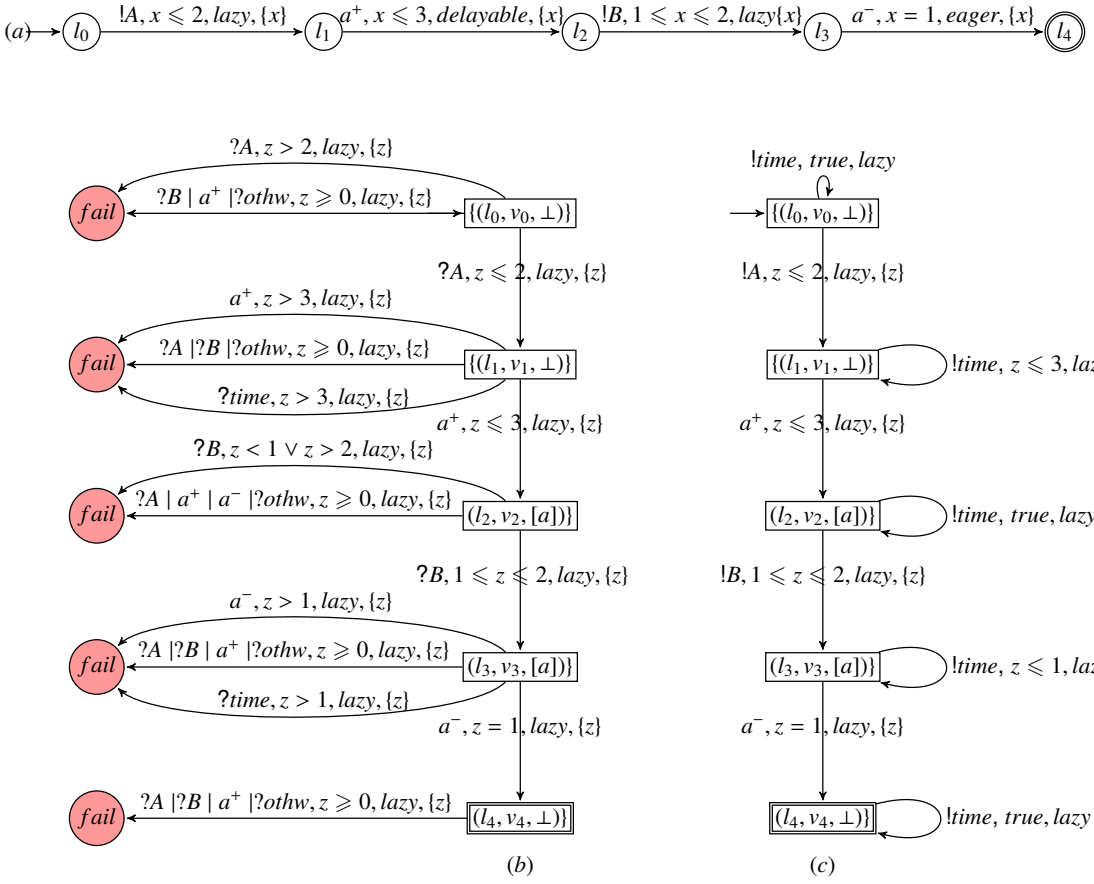


FIGURE 7.3 – ((b). le *STPTIO* et (c). l'implémentation du *TPAIO* presenté dans la Fig. 7.3.(a)

Soit $T = \langle L, l_0, \Sigma_{in} \cup \Sigma_{out} \cup \{\tau\}, \Gamma^I, X^I, \Delta^I, F^I \rangle$ une spécification, $I = \langle L^I, l_0^I, \Sigma_{in}^I \cup \Sigma_{out}^I \cup \{\tau\}, \Gamma^I, X^I, \Delta^I, F^I \rangle$ une implémentation et $tc = \langle L^{tc}, l_0^{tc}, \Sigma_{in}^{tc} \cup \Sigma_{out}^{tc}, \Gamma^{tc}, X^{tc}, \Delta^{tc}, F^{tc} \rangle$ un cas de test de T compatible avec I. On note $SP = I \parallel tc$ le produit synchronisé de I et tc. Le produit synchronisé synchronise à la fois le temps et les actions communes de pile, d'entrée et de sortie. Ainsi, il synchronise les actions communes de $\Sigma_{I \to tc} \cup \Sigma_{tc \to I} \cup (\Gamma^{tc+-} \cap \Gamma^{I+-})$. Il est inspiré de la composition parallèle de deux *TAIO* qui est définie dans la Def. 3. Il tient compte seulement des règles qui synchronisent les actions communes pour calculer les chemins en commun entre l'implémentation et le cas de test. On n'a pas besoin des transitions où l'implémentation ou le cas de test évoluent indépendamment. Donc, une transition est ajoutée dans le produit synchronisé pour chacune des deux possibilités suivantes :

(i). les transitions étiquetées avec *othw* dans tc se synchronisent avec les transitions d'une implémentation étiquetées avec une action qui est une action de sortie ou une action de la pile dans I et qui n'est pas une action d'entrée ou une action de la pile dans tc.

(ii). les transitions de tc et I se synchronisent quand elles sont étiquetées par une

action commune d'entrée ou de sortie ($\Sigma_{I\to tc} \cup \Sigma_{tc\to I}$) ou une action commune de la pile ($\Gamma^{tc+-} \cap \Gamma^{I+-}$).

Nous définissons le produit synchronisé d'une implémentation *I* et d'un cas de test tc dans la Def. 33.

Définition 33 : Produit synchronisé d'une implémentation *I* et d'un cas de test tc

Soit $I = \langle L^I, l_0^I, \Sigma_{in}^I \cup \Sigma_{out}^I \cup \{\tau\}, \Gamma^I, X^I, \Delta^I, F^I \rangle$ une implémentation où $\Sigma_{tc\to I} \subseteq \Sigma_{in}^I$ et $\Sigma_{I\to tc} \subseteq \Sigma_{out}^I$, et $tc = \langle L^{tc}, l_0^{tc}, \Sigma_{in}^{tc} \cup \Sigma_{out}^{tc}, \Gamma^{tc}, X^{tc}, \Delta^{tc}, F^{tc} \rangle$ un cas de test compatible avec *I* où $\Sigma_{I\to tc} \subseteq \Sigma_{in}^{tc}$ et $\Sigma_{tc\to I} \subseteq \Sigma_{out}^{tc}$, le produit synchronisé $I \parallel tc$ est un *TPAIO* $\langle L^I \times L^{tc}, (l_0^I, l_0^{tc}), \Sigma_{in}^I \cup \Sigma_{out}^I, \Gamma^I \cup \Gamma^{tc}, X^I \cup X^{tc}, \Delta, F^I \times F^{tc} \rangle$ où Δ est la relation définie ainsi :

(i). $((l^I, l^{tc}), act, g^I \wedge g^{tc}, lazy, X^I \cup \{y\}, (l^{I'}, fail)) \in \Delta$ si $(l^I, act, g^I, lazy, X^I, l^{I'}) \in \Delta^I$ et $(l^{tc}, othw, g^{tc}, \{y\}, lazy, fail) \in \Delta^{tc}$ et $act \in (\Sigma_{out}^I \setminus \Sigma_{in}^{tc})$ ou $act \in (\Gamma^{I+-} \setminus \Gamma^{tc+-})$,

(ii). $((l^I, l^{tc}), act, g^I \wedge g^{tc}, lazy, X^I \cup \{y\}, (l^I, l^{tc'})) \in \Delta$ si $(l^I, act, g^I, lazy, X^I, l^{I'}) \in \Delta^I$ et $(l^{tc}, act, g^{tc}, \{y\}, lazy, l^{tc'}) \in \Delta^{tc}$ et $act \in \Sigma_{I\longrightarrow tc} \cup \Sigma_{tc\longrightarrow I} \cup (\Gamma^{tc+-} \cap \Gamma^{I+-})$.

L'implémentation *I* n'est pas conforme à la spécification T si une localité $(l^I, fail)$ est atteignable dans l'un des produits synchronisés $SP \stackrel{def}{=} I \parallel tc$ où $l^I \in L^I$ pour chaque cas de test tc de l'ensemble de cas de test TC.

7.4.3/ RÉSULTATS EXPÉRIMENTAUX

Nous avons développé un prototype pour générer des cas de tests à partir d'un *TPAIO* non déterministe. Il est développé en java. Nous avons aussi implémenté les opérateurs de mutation de *TPAIO* qui sont présentés dans la Sec. 7.4.1 afin de générer des implémentations par mutation et détecter les mutants non conformes.

L'expérimentation a été réalisée sur l'exemple du *TPAIO* de la Fig. 7.4 qui modélise un système de détection d'action de n clics ($n \geqslant 1$). Ce *TPAIO* compte le nombre de clics qui ont eu lieu dans une unité de temps après le premier clic en utilisant une pile. Il est décrit dans la section 4.1.

Nous avons obtenu 183 mutants en appliquant les sept opérateurs de mutation présentés dans la Sec. 7.4.1. Les résultats de l'exécution des tests générés à partir du *TPAIO* sont présentés dans le tableau 7.1. La première colonne indique l'opérateur de mutation appliqué. La deuxième colonne indique le nombre de mutants obtenus par l'application de l'opérateur de mutation. La troisième colonne indique le nombre de mutants non conformes à la spécification pour chaque opérateur de mutation. La quatrième colonne donne le nombre de mutants non-conformes et non détectés par notre méthode, alors que la cinquième colonne indique le nombre de mutants tués, dont une non-conformité est prononcée en appliquant notre méthode.

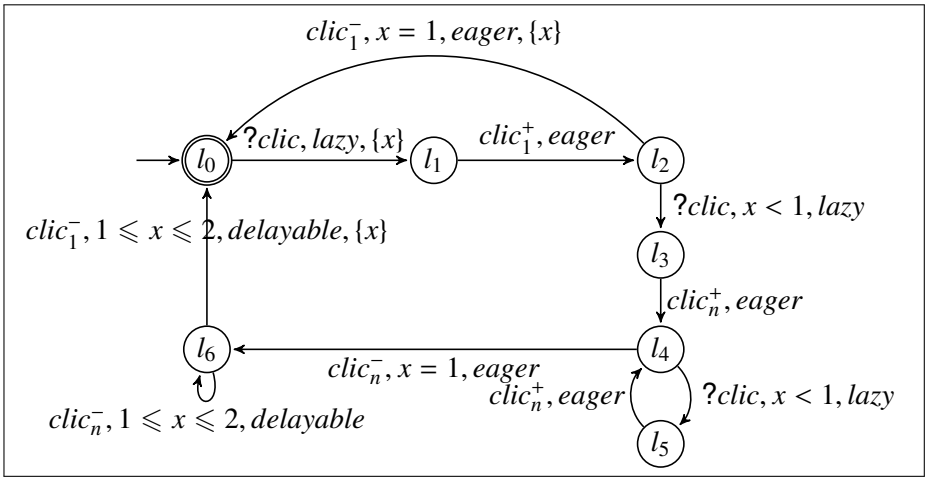


FIGURE 7.4 – *TPAIO* décrivant la détection d'une action n–clics ($n \geqslant 1$)

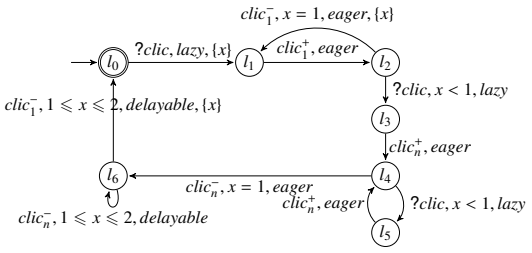
Opérateur de mutation	Mutants	Non conformes	non détectés	détectés
Changer action	33	21	0	21
Changer garde	7	3	0	3
Nier garde	7	3	0	3
Changer avec une nouvelle action de sortie	10	10	0	10
Change avec une nouvelle action d'empilement	10	10	0	10
Changer source	42	31	0	31
Changer cible	60	43	2	41
Changer deadline	14	6	0	6
Total	183	127	2	125

TABLE 7.1 – Résultats expérimentaux sur l'exemple des n-clics de la Fig. 7.4

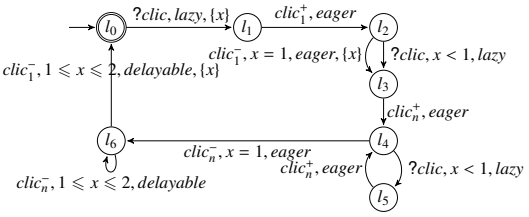
Ce tableau montre que 125 mutants non-conformes sur 127 ont été détectés par notre méthode, c'est à dire 98% des mutants non-conformes. 56 mutants sont conformes à la spécification. En effet, l'application de l'un des opérateurs de mutation ne contredit pas nécessairement la spécification. La Fig. 7.6 montre quelques exemples de mutants conformes obtenus par l'application d'un opérateur de mutation sur l'automate de la Fig. 7.4. Le mutant de la Fig. 7.6.(a) est obtenu par l'application de l'opérateur de mutation "changer action" qui consiste à changer la transition $(l_0, ?clic, true, lazy, \{x\}, l_1)$ par $(l_0, clic_1^-, true, lazy, \{x\}, l_1)$. Le mutant de la Fig. 7.6.(b) est obtenu par l'application de l'opérateur de mutation "changer deadline" qui consiste à changer la transition $(l_2, clic_1^-, x = 1, eager, \{x\}, l_0)$ par $(l_2, clic_1^-, x = 1, delayable, \{x\}, l_0)$. Le mutant de la Fig. 7.6.(c) est obtenu par l'application de l'opérateur de mutation "changer source" qui consiste à changer la transition $(l_6, clic_n^-, 1 \leqslant x \leqslant 2, delayable, \emptyset, l_6)$ par $(l_4, clic_n^-, 1 \leqslant x \leqslant 2, delayable, \emptyset, l_6)$. Le mutant de la Fig. 7.6.(d) est obtenu par l'application de l'opérateur de mutation "changer cible" qui consiste à changer la transition $(l_2, clic_1^-, x = 1, eager, \{x\}, l_0)$ par $(l_2, clic_1^-, x = 1, eager, \{x\}, l_4)$. Le mutant de la Fig. 7.6.(e) est obtenu par l'application de l'opérateur de mutation "changer garde" qui consiste à changer la transition $(l_6, clic_n^-, 1 \leqslant x \leqslant 2, delayable, \emptyset, l_6)$ par

$(l_6, clic_n^-, 1 \leqslant x < 2, delayable, \emptyset, l_6)$. Le mutant de la Fig. 7.6.(f) est obtenu par l'application de l'opérateur de mutation "nier garde" qui consiste à changer la transition $(l_6, clic_n^-, 1 \leqslant x \leqslant 2, delayable, \{x\}, l_0)$ par $(l_6, clic_n^-, 1 > x \vee x > 2, delayable, \{x\}, l_0)$.

Notre méthode ne détecte pas deux mutants qui ne sont pas conformes à la spécification. La Fig. 7.5 montre deux exemples de mutants non-conformes obtenus par l'application d'un opérateur de mutation et dont notre méthode ne détecte pas la non conformité. Le mutant de la Fig. 7.5.(a) est obtenu par l'application de l'opérateur de mutation "changer cible" qui consiste à changer la transition $(l_2, clic_1^-, x = 1, eager, \{x\}, l_0)$ par $(l_2, clic_1^-, x = 1, eager, \{x\}, l_1)$. Le mutant de la Fig. 7.5.(b) est obtenu par l'application de l'opérateur de mutation "changer cible" qui consiste à changer la transition $(l_2, clic_1^-, x = 1, eager, \{x\}, l_0)$ par $(l_2, clic_1^-, x = 1, eager, \{x\}, l_3)$. Notre méthode ne détecte pas la non conformité de ces deux cas à cause du critère de couverture qui est la couverture de toutes les transitions. Un critère de couverture comme les k-chemins pourrait atteindre cet objectif. La Sec. 7.5 explique pourquoi le critère de couverture des transitions est faible pour la détection des mutants non-conformes et pourquoi celui des k-chemins pourrait résoudre ce problème. Le critère "tous les k-chemins" est une variante du critère "tous les chemins" (voir la section 3.1.2.2). Il limite la recherche de tous les chemins du graphe de flot de contrôle du programme aux chemins qui ont au plus k itérations consécutives des boucles.



(a). changer cible



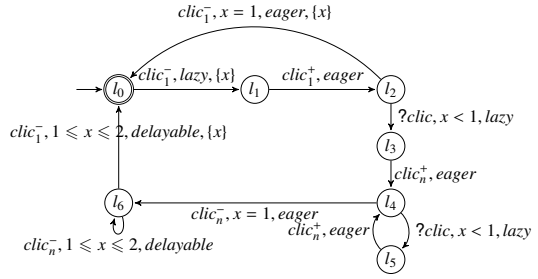
(b).changer cible

FIGURE 7.5 – Mutants non conformes et non détectés du *TPAIO* de la Fig. 7.4

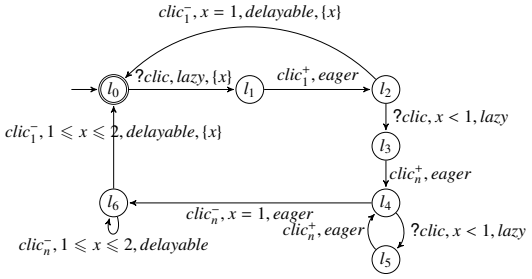
7.5/ INCOMPLÉTUDE

Nos résultats expérimentaux montrent que la grande majorité des mutants non-conformes sont détectés. Notre méthode ne détecte pas la non-conformité pour quelques mutants non-conformes à cause du critère de couverture qui est la couverture de toutes les transitions. Nous montrons à travers l'exemple 7.5.1 que l'incomplétude de notre méthode est due à ce critère de couverture des transitions, et que le critère "tous les k-chemins" serait suffisant pour détecter la non-conformité.

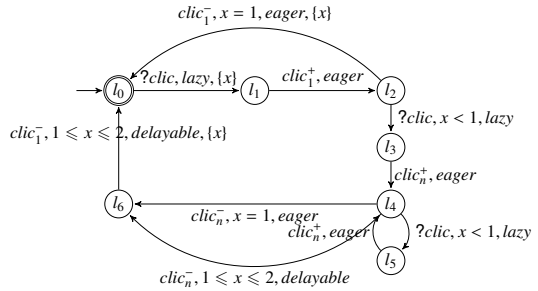
Exemple 7.5.1. La Fig. 7.7.(b) présente un mutant M d'un *TPAIO* T présenté dans la Fig. 7.7.(a). Il est obtenu par l'application de l'opérateur de mutation "changer cible" qui consiste à changer la transition $(l_1, !B, x \leqslant 1, delayable, \{x\}, l_0)$ par $(l_1, !B, x \leqslant 1, delayable, \{x\}, l_2)$. Le mutant M n'est pas conforme à la spécification T car par exemple $out(M\ after\ 0.5!A0.2!B) = [0,1] \cup \{D\} \nsubseteq out(T\ after\ 0.5!A0.2!B) = [0,2] \cup \{A\}$.



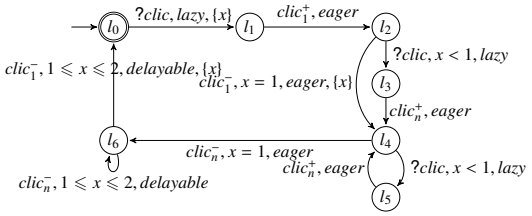
(a). changer action



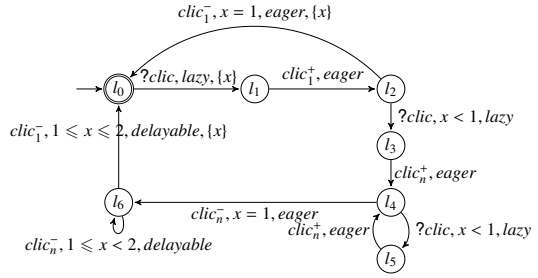
(b). changer deadline



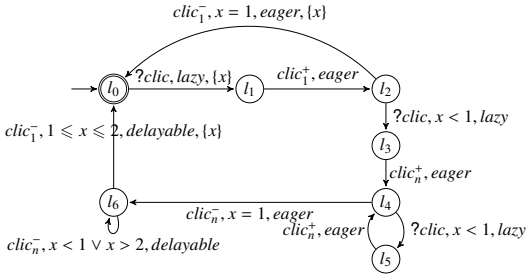
(c). changer source



(d). changer cible



(e). changer garde



(f). nier garde

FIGURE 7.6 – Mutants non-conformes et détectés du *TPAIO* de la Fig. 7.4

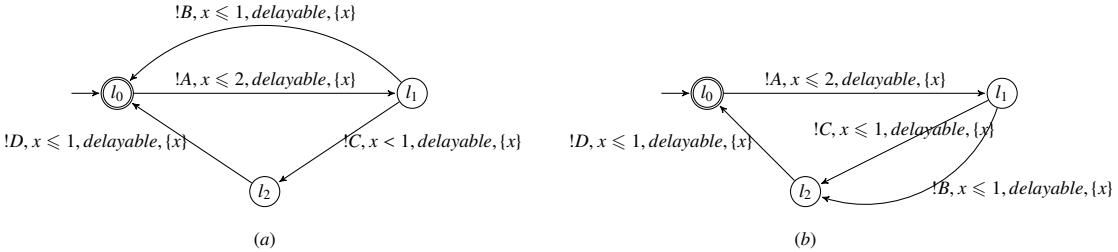


FIGURE 7.7 – (a). Exemple d'un *TPAIO* et (b). de l'un de ses mutants

Les figures 7.8.(a) et 7.8.(b) sont deux cas tests obtenus couvrant toutes les transitions du *TPAIO* présenté dans la Fig. 7.7.(a) en appliquant notre méthode de génération de tests. Notre méthode ne détecte pas la non-conformité en utilisant ces deux cas de tests. Notre méthode d'évaluation de détection consiste à vérifier si une localité $fail$ est atteignable dans l'un des deux produits synchronisés de l'implémentation et d'un cas de test. Le

produit synchronisé est l'intersection des langages d'un cas de test de la spécification et de l'implémentation. Pour notre cas, la non conformité est détectée avec l'exécution d'une trace du chemin $\pi = l_0 \xrightarrow{!A,x\leqslant 2,delayable,\{x\}} l_1 \xrightarrow{!B,x\leqslant 1,delayable,\{x\}} l_0 \xrightarrow{!D,x\leqslant 1,delayable,\{x\}} l_0$ de l'implémentation. Si ce chemin était synchronisé avec l'un des cas de tests, on pourrait détecter la non conformité. Mais, c'est seulement la partie $l_0 \xrightarrow{!A,x\leqslant 2,delayable,\{x\}} l_1 \xrightarrow{!B,x\leqslant 1,delayable,\{x\}} l_0$ qui est synchronisée avec le cas de test présenté dans le Fig. 7.8.(a). La dernière transition $l_0 \xrightarrow{!D,x\leqslant 1,delayable,\{x\}} l_0$ ne peut plus être synchronisée car il n'existe aucune transition qui quitte $pass$ dans le cas de test. Si le cas de test effectuait une étape de plus, on pourrait détecter la non-conformité. Par exemple, si on utilise le cas de test présenté dans la Fig. 7.9 qui assure la couverture des 2-chemins, le chemin π pourrait être synchronisé avec le chemin $\{(l_0, v_0, \bot)\} \xrightarrow{!A,y\leqslant 2,lazy,\{y\}} \{(l_1, v_1, \bot)\} \xrightarrow{!B,y\leqslant 1,lazy,\{y\}} \{(l_0, v_0, \bot)\} \xrightarrow{!D,y\geqslant 0,lazy,\{y\}} fail$ du cas de test présenté dans la Fig. 7.9 et la non conformité serait détectée.

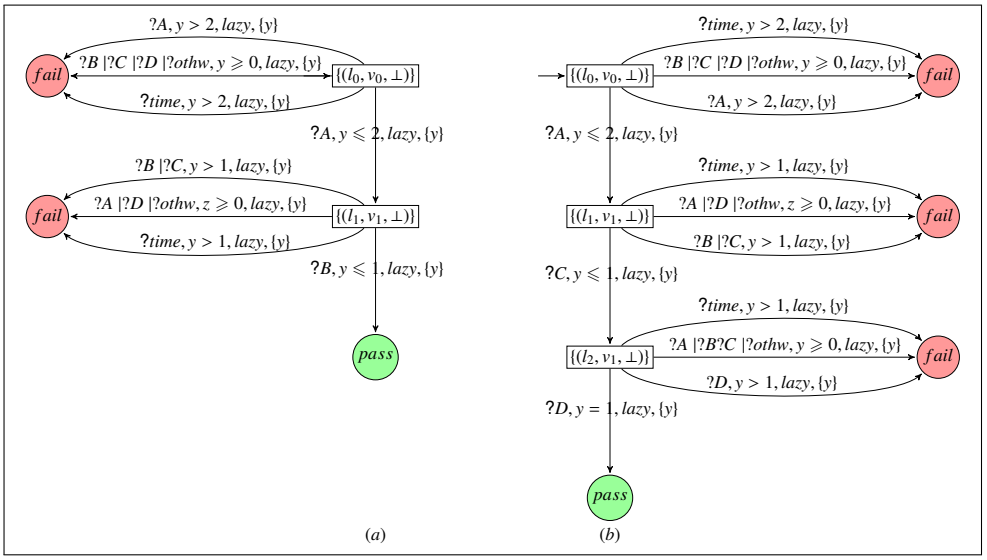


FIGURE 7.8 – Deux cas de tests couvrant toutes les transitions pour le *TPAIO* présenté dans la Fig. 7.7.(a)

7.6/ CONCLUSION

Nous avons présenté dans ce chapitre une méthode de test de conformité à partir d'un *TPAIO* non déterministe avec n'importe quelle deadline. Nous avons proposé une méthode pour calculer des cas de test qui atteignent une localité finale avec une pile vide avec le critère de couverture "toutes les transitions". Nous avons calculé en premier lieu le *STPTIO* qui est un testeur du *TPAIO*. Le testeur est un *TPAIO* avec une seule horloge. Il déterminise le *TPAIO* donné. Il ajoute des possibilités qui mènent à $fail$ pour détecter la non conformité. Les contraintes de la pile sont résolues par le calcul d'un *RA* qui calcule des chemins de test à partir du testeur. Puis, on génère des cas de tests couvrant les transitions atteignables du *TPAIO*. Notre méthode est évaluée en utilisant des opérateurs de mutation qui permettent de générer des mutants à partir d'un *TPAIO*

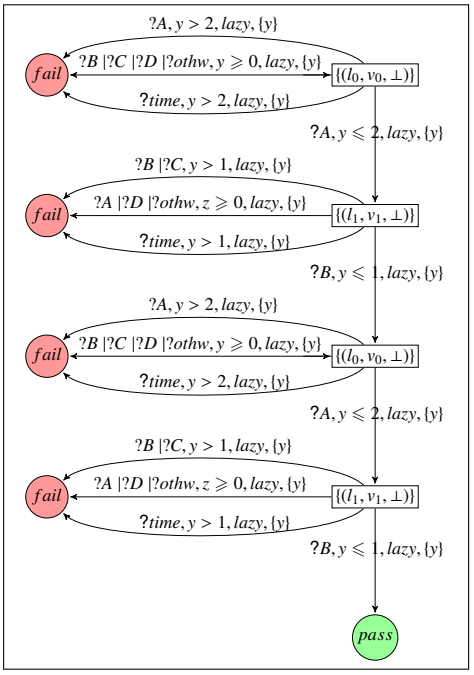


FIGURE 7.9 – Cas de test couvrant un 2-chemin pour le *TPAIO* présenté dans la Fig. 7.7.(a)

donné. Nos résultats expérimentaux montrent que la grande majorité des mutants non-conformes sont détectés. Notre méthode ne détecte pas la non conformité pour quelques mutants non-conformes à cause du critère de couverture d'assez faible puissance qui est la couverture de toutes les transitions. Un critère de couverture comme les k-chemins pourrait améliorer les capacités de détection de non conformité de la méthode.

III

CONCLUSION ET PERSPECTIVES

8

CONCLUSION ET PERSPECTIVES

Nous présentons dans ce chapitre nos conclusions sur le travail effectué au cours de cette thèse, ainsi que les perspectives envisagées.

8.1/ BILAN

Ce travail a été motivé par l'intérêt qu'il y a à proposer des méthodes de génération de tests pour des systèmes décrits à base d'automates à pile temporisés avec deadlines. Les deadlines modélisent des conditions de progression du temps dans les localités du *TPAIO*. A notre connaissance, c'est la première fois que ce problème est abordé. Les cas de tests générés établissent un verdict relativement à une relation de conformité. Pour ce faire, nous avons défini une relation de conformité pour les *TPAIO*, ce qui est notre première contribution.

Notre deuxième contribution est une méthode polynomiale permettant la génération de cas de tests pour des *TPAIO* avec deadline *lazy* seulement. Cette solution ne passe pas par la discrétisation comme les autres solutions afin d'éviter le problème d'explosion combinatoire et de réduire la complexité de calcul. C'est une adaptation aux automates à pile temporisés d'une méthode de calcul des états accessibles définie pour des automates à pile seulement [FWW97]. Elle consiste à fusionner des transitions de l'automate à pile temporisé allant d'une localité initiale à une localité finale en respectant les contraintes de la pile afin d'obtenir une seule transition. La garde de cette transition est vérifiée en utilisant un outil d'évaluation de satisfiabilité de formules logiques sur les réels comme les solveurs SMT. Par contre, cette solution est incomplète. L'incomplétude est un compromis acceptable dans le cadre de la génération de tests qui est par nature incomplète permettant ainsi une solution moins coûteuse.

Notre troisième contribution est une méthode de génération de tests à partir d'un *TPAIO* déterministe avec sorties seulement et deadline *delayable* seulement. Cette méthode est appliquée pour tester des programmes temporisés récursifs. Nous avons calculé un testeur à pile temporisé déterministe en adaptant la méthode d'approximation de [KT09]. Puis nous avons défini une méthode pour engendrer un automate d'accessibilité. Enfin, nous avons engendré une suite de tests couvrant les états et les transitions atteignables. Comme les *TPAIO* sont des abstractions des programmes récursifs temporisés, les tests ne sont pas, dans le cas général, assurés d'être des exécutions concrètes. Pour

sélectionner les cas de test concrets et leurs données, nous avons utilisé de l'exécution symbolique. Nous avons adopté une approche pour évaluer la qualité des suites de tests générées. L'approche consiste à comparer des traces obtenues lors de l'exécution d'un programme muté avec des données de test avec un cas de test afin d'annoncer le verdict conformément à la relation de conformité.

Notre quatrième contribution est la généralisation du processus de génération de tests à partir d'un *TPAIO* déterministe avec sorties seulement et deadline $delayable$ au cas des *TPAIO* non déterministes avec entrées/sorties et les trois types usuels de deadlines : $lazy$, $delayble$, $eager$. Nous avons défini la construction d'un testeur à partir d'un *TPAIO* donné pour résoudre les contraintes d'horloges, déterminiser le *TPAIO* et produire des verdicts d'échec $fail$ modélisant des cas de non-conformité. Puis, nous avons calculé un automate d'atteignabilité à partir du testeur obtenu afin de calculer un ou plusieurs chemins entre deux localités symboliques en respectant les contraintes de pile. Enfin, nous avons généré des cas de tests en utilisant chaque transition de l'automate d'atteignabilité allant d'une localité symbolique initiale vers une localité symbolique finale. Nous avons évalué la capacité de notre méthode à détecter des implémentations non-conformes par une technique de mutation qui permet de modifier automatiquement un *TPAIO* donné par l'application d'un opérateur de mutation. Nos résultats expérimentaux montrent que la grande majorité des mutants non-conformes sont détectés. Notre méthode ne détecte pas la non-conformité pour quelques mutants non-conformes à cause du critère de couverture d'assez faible puissance qui est la couverture de toutes les transitions. Un critère de couverture comme les k-chemins pourrait améliorer les capacités de détection de non-conformité de la méthode.

Notre cinquième contribution est l'implémentation de la méthode de génération de tests à partir d'un *TPAIO* général avec entrées/sorties et avec deadlines quelconques. Nous avons aussi implémenté en java une méthode d'évaluation expérimentale par une technique de mutation. Ainsi, une méthode de génération d'un mutant à partir d'un *TPAIO* donné en appliquant une opérateur de mutation est implémentée. Nos résultats expérimentaux montrent que la grande majorité des mutants non-conformes sont détectés.

8.2/ PERSPECTIVES

Nous allons mentionner dans cette partie les perspectives de nos travaux de recherche.

Nos résultats expérimentaux montrent que la grande majorité des mutants non-conformes sont détectés. Notre méthode ne détecte pas la non-conformité pour quelques mutants non-conformes à cause du critère de couverture qui est la couverture de toutes les transitions. Nous devons étendre nos méthodes de génération de tests pour prendre en compte d'autres critères de couverture de tests afin de détecter plus de mutants non-conformes. Les mutants non-conformes qui n'ont pas été détectés auraient pu l'être si nos tests avaient couvert plus de chemins que ceux nécessaires à simplement couvrir toutes les transitions. Un critère de couverture améliorant pourrait donc être de couvrir tous les k-chemins d'un *TPAIO*.

Nous proposons d'améliorer notre méthode d'évaluation expérimentale avec d'autres exemples concrets comme des programmes récursifs avec contraintes temps réel au lieu d'ajouter arbitrairement des contraintes temporelles à un programme récursif.

Les cas de test peuvent être sélectionnés selon des critères à définir. La couverture de toutes les transitions est le critère pour générer des cas de tests. Un autre critère pour sélectionner les cas de test consiste à formuler un objectif de test et à générer les cas de test qui satisfont cet objectif. Donc, nous proposons d'étendre nos méthodes de génération de tests à partir d'un *TPAIO* pour prendre en compte des objectifs de test ciblant les comportements à tester. Ces objectifs pourraient être modélisés par des automates de Büchi temporisés.

Une méthode de génération de tests par mutation à partir d'un *TAIO* est proposée dans [ALN13]. Cette méthode permet de générer des cas de tests qui permettent de détecter des erreurs injectées. Nous proposons d'étendre nos méthodes de génération de tests par une méthode de génération de tests par mutation à partir d'un *TPAIO* en s'inspirant des travaux de [ALN13].

LISTE DES SYMBOLES

ACV Authorized-Clock Values

DBM Difference Bounded Matrix

DTA Deterministic Timed Automata

STPTIO Symbolic Timed Pushdown Tester with Inputs and Outputs

ioco Input-Output Conformance Relation

IOLTS Input Output Labeled Transtion System

LTS Labeled Transition System

MBT Model Based Testing

PA Pushdown Automata

RA Reachability Automata

RTA Reachability Timed Automata

rtioco Relativized Timed Conformance Relation

SUT System Under Test

TA Timed Automata

TAD Timed Automata with Deadlines

TAIO Timed Automata with Inputs and Outputs

tioco Timed Input-Output Conformance Relation

TPA Timed Pushdown Automata

TPAIO Timed Pushdown Automata with Inputs and Outputs

TPC Time Progress Condition

tpioco Timed Pushdown-Input-Output Conformance Relation

TPTIO Timed Pushdown Tester with Inputs and Outputs

BIBLIOGRAPHIE

[AAS12] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Jari Stenman. Dense-timed pushdown automata. In *LICS*, pages 35–44, 2012.

[ABB97] Jean-Michel Autebert, Jean Berstel, and Luc Boasson. Context-free languages and push-down automata. In *Handbook of Formal Languages*, pages 111–174. Springer, 1997.

[ABJK11] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Uml in action : A two-layered interpretation for testing. *SIGSOFT Softw. Eng. Notes*, 36(1) :1–8, 2011.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B : System and Software Engineering*. 2010.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Inf. Comput.*, (1) :2–34, 1993.

[ACH94] Rajeev Alur, Costas Courcoubetis, and Thomas A. Henzinger. *CONCUR '94 : Concurrency Theory : 5th International Conference, Uppsala, Sweden, August 22–25, 1994, Proceedings*, chapter The Observational Power of Clocks, pages 162–177. Springer, 1994.

[AD90a] Rajeev Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, pages 322–335, 1990.

[AD90b] Rajeev Alur and David Dill. *Automata, Languages and Programming : 17th International Colloquium Warwick University, England, July 16–20, 1990 Proceedings*, pages 322–335. Springer Berlin Heidelberg, 1990.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *TCS*, (2) :183 – 235, 1994.

[AFH99] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata : a determinizable class of timed automata. *Theoretical Computer Science*, 211(1–2) :253 – 273, 1999.

[AJT15] Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. *Science of Computer Programming*, 97, Part 4 :383 – 404, 2015.

[ALN13] BernhardK. Aichernig, Florian Lorber, and Dejan Nickovic. Time for mutants-model-based mutation testing with timed automata. In *Tests and Proofs*, volume 7942 of *LNCS*, pages 20–38. Springer, 2013.

[Arn94] André Arnold. *Finite Transition Systems : Semantics of Communicating Systems*. Prentice Hall International (UK) Ltd., 1994.

[AV08] Tiago L. Alves and Joost Visser. A case study in grammar engineering. In *Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, pages 285–304, 2008.

[BB04] Laura Brandán Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In *IN FATES'04*, pages 64–78. Springer, 2004.

[BBBB09] Christel Baier, Nathalie Bertrand, Patricia Bouyer, and Thomas Brihaye. When are timed automata determinizable ? In *Automata, Languages and Programming, 36th Internatilonal Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part II*, pages 43–54, 2009.

[BCBB09] Patricia Bouyer, Copyright C, Patricia Bouyer, and Patricia Bouyer. This document in subdirectoryrs/02/35/ timed automata may cause some troubles, 909.

[BCD05] Patricia Bouyer, Fabrice Chevalier, and Deepak D'Souza. *Foundations of Software Science and Computational Structures : 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings*, chapter Fault Diagnosis Using Timed Automata, pages 219–233. Springer Berlin Heidelberg, 2005.

[Bei90] Boris Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[Bei95a] Boris Beizer. *Black-box Testing : Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Bei95b] Boris Beizer. *Black-box Testing : Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[BEM97a] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata : Application to model-checking. In *CONCUR '97 : Concurrency Theory*, LNCS, pages 135–150. Springer Berlin Heidelberg, 1997.

[BEM97b] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata : Application to model-checking. CONCUR '97, pages 135–150, London, 1997. Springer.

[BER94] Ahmed Bouajjani, Rachid Echahed, and Riadh Robbana. *Computer Aided Verification : 6th International Conference, CAV '94 Stanford, California, USA, June 21–23, 1994 Proceedings*, chapter Verification of context-free timed systems using linear hybrid observers, pages 118–131. Springer, Berlin, Heidelberg, 1994.

[BER95] Ahmed Bouajjani, Rachid Echahed, and Riadh Robbana. On the automatic verification of systems with continuous variables and unbounded discrete data structures. In *Hybrid Systems II*, volume 999 of *LNCS*, pages 64–85, 1995.

[BHM+10] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. Mutation-based test case generation for simulink models. In *Proceedings of the 8th International Conference on Formal Methods for Components and Objects*, pages 208–227, 2010.

[BLL+98] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Yi Wang, and Carsten Weise. New Generation of UPPAAL. In *Int. Workshop on Software Tools for Technology Transfer*, June 1998.

[BMP10] Massimo Benerecetti, Stefano Minopoli, and Adriano Peron. Analysis of timed recursive state machines. In *TIME 2010*, pages 61–68. IEEE, 2010.

[Bou04] Patricia Bouyer. Forward analysis of updatable timed automata. *Form. Methods Syst. Des.*, 24(3) :281–320, 2004.

[BPDG98] Béatrice Bérard, Antoine Petit, Volker Diekert, and Paul Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36(2-3) :145–182, 1998.

[BSJK15] Nathalie Bertrand, Amélie Stainer, Thierry Jéron, and Moez Krichen. A game approach to determinize timed automata. *Formal Methods in System Design*, 46(1) :42–80, 2015.

[BST98] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. volume 1536 of *LNCS*, pages 103–129. Springer Berlin Heidelberg, 1998.

[BT00] Ed Brinksma and Jan Tretmans. Testing transition systems : An annotated bibliography. In *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000*, pages 187–195, 2000.

[BY03] Johan Bengtsson and Wang Yi. *Formal Methods and Software Engineering : 5th International Conference on Formal Engineering Methods, ICFEM 2003, Singapore, November 5-7, 2003. Proceedings*, chapter On Clock Difference Constraints and Termination in Reachability Analysis of Timed Automata, pages 491–503. 2003.

[CJL+09] Franck Cassez, Jan J. Jessen, Kim G. Larsen, Jean-François Raskin, and Pierre-Alain Reynier. Automatic synthesis of robust and optimal controllers — an industrial case study. In *HSCC '09*, pages 90–104. Springer, 2009.

[CL15] Lorenzo Clemente and Slawomir Lasota. Timed pushdown automata revisited. *CoRR*, abs/1503.02422, 2015.

[CLPV10] Rohit Chadha, Axel Legay, Pavithra Prabhakar, and Mahesh Viswanathan. Complexity bounds for the verification of real-time software. In *VMCAI'10*, volume 5944 of *LNCS*, pages 95–111. Springer, 2010.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[Dan01] Zhe Dang. Binary reachability analysis of pushdown timed automata with dense clocks. In *In CAV*, pages 506–517. Springer, 2001.

[Dan03] Zhe Dang. Pushdown timed automata : a binary reachability characterization and safety verification. *Theoretical Computer Science*, 302(1–3) :93 – 121, 2003.

[DDZ98] Alain Denise, Isabelle Dutour, and Paul Zimmermann. CS : a MuPAD package for counting and randomly generating combinatorial structures. In *Proc. 10th Conference on Formal Power series and Algebraic Combinatorics (FPSAC)*, 1998.

[DHK13] Aloïs Dreyfus, Pierre-Cyrille Héam, and Olga Kouchnarenko. Random grammar-based testing for covering all non-terminals. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*, pages 210–215, 2013.

[DHKM14a] Aloïs Dreyfus, Pierre-Cyrille Héam, Olga Kouchnarenko, and Catherine Masson. A random testing approach using pushdown automata. *Journal of Software Testing, Verification, and Reliability*, 24 :656 – 683, 2014.

[DHKM14b] Alois Dreyfus, Pierre-Cyrille Héam, Olga Kouchnarenko, and Catherine Masson. A random testing approach using pushdown automata. *STVR, Software Testing, Verification and Reliability*, 24(8) :656–683, 2014.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989.

[DLS78a] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection : Help for the practicing programmer. *Computer*, 11(4) :34–41, April 1978.

[DLS78b] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection : Help for the practicing programmer. *Computer*, 11(4) :34–41, 1978.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3 : An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In *In Proc. of Hybrid Systems III, LNCS 1066*, pages 208–219. Springer Verlag, 1996.

[DOY94] C. Daws, A. Olivero, and S. Yovine. Verifying et-lotos programs with kronos. In Dieter Hogrefe and Stefan Leue, editors, *Proceedings of the 7th. International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE'94*, pages 227–242, Bern, Switzerland, 1994. Formal Description Techniques VII, Chapman & Hall.

[DT98] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In *TACAS*, volume 1384 of *LNCS*, pages 313–329, 1998.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, pages 66–. IEEE Computer Society, 1995.

[EM06] Michael Emmi and Rupak Majumdar. Decision problems for the verification of real-time software. In *Hybrid Systems : Computation and Control*, volume 3927 of *LNCS*, pages 200–211. 2006.

[Fou02] Hacène Fouchal. Conformance testing techniques for timed systems. In *SOFSEM 2002 : Theory and Practice of Informatics, 29th Conference on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 22-29, 2002, Proceedings*, pages 1–19, 2002.

[FS09] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[FW88] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10) :1483–1498, 1988.

[FWW97] Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems (ext. abs.). In *Infinity*, volume 9 of *ENTCS*, pages 27–37, 1997.

[FZC94] Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132(1–2) :1 – 35, 1994.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart : Directed automated random testing. *SIGPLAN Not.*, 40(6) :213–223, 2005.

[God12] Patrice Godefroid. Test generation using symbolic execution. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 24–33, 2012.

[Ham77] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4) :279–290, 1977.

[Her76] P. M. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3) :92–96, 1976.

[HLM+08a] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Formal methods and testing. chapter Testing Real-time Systems Using UPPAAL, pages 77–117. Springer, 2008.

[HLM+08b] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Formal methods and testing. chapter Testing Real-time Systems Using UPPAAL, pages 77–117. 2008.

[HLN+03] Anders Hessel, Kim Guldstrand Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using uppaal. In *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003*, pages 114–130, 2003.

[HM15] Pierre-Cyrille Héam and Hana M'Hemdi. Covering both stack and states while testing push-down systems. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–7, 2015.

[HMP92] Thomas Henzinger, Zohar Manna, and Amir Pnueli. What good are digital clocks ? pages 545–558. Springer-Verlag, 1992.

[HRK11] Nannan He, Philipp Rümmer, and Daniel Kroening. Test-case generation for embedded simulink via formal concept analysis. In *DAC*, pages 224–229, 2011.

[IEE90] IEEE. Ieee standard glossary of software engineering terminology, 1990.

[IW78] Infotech and C.H. White. *System Reliability and Integrity*. Number v. 1 in Infotech Maidenhead : Infotech state of the art report. 1978.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5) :649–678, 2011.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7) :385–394, July 1976.

[KJM04] Ahmed Khoumsi, Thierry Jéron, and Hervé Marchand. *Formal Approaches to Software Testing : Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003. Revised Papers*, chapter Test Cases Generation for Nondeterministic Real-Time Systems, pages 131–146. Springer, 2004.

[KT05] Moez Krichen and Stavros Tripakis. State identification problems for timed automata. In *Testing of Communicating Systems, 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005, Montreal, Canada, May 31 - June 2, 2005, Proceedings*, pages 175–191, 2005.

[KT09] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Form. Methods Syst. Des.*, (3) :238–304, June 2009.

[LMN05] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. *Formal Approaches to Software Testing : 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, chapter Online Testing of Real-time Systems Using Uppaal, pages 79–94. Springer, 2005.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1 :134–152, 1997.

[LY96] D. Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8) :1090–1123, 1996.

[MJMR15a] H. M'Hemdi, J. Julliand, P.-A. Masson, and R. Robbana. Conformance testing for timed recursive programs. In *ICIS 2015, 14th IEEE/ACIS Int. Conf. on Computer and Information Science*, volume 614 of *SCI, Studies in Computational Intelligence*, pages 203–219, Las Vegas, United States, June 2015. Springer.

[MJMR15b] H. M'Hemdi, J. Julliand, P.-A. Masson, and R. Robbana. Test generation from timed pushdown automata with inputs and outputs. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–10, 2015.

[MJMR15c] Hana M'Hemdi, Jacques Julliand, Pierre-Alain Masson, and Riadh Robbana. Conformance testing for non deterministic timed pushdown automata with deadlines. In *Enabling Technologies : Infrastructure for Collaborative Enterprises (WETICE), 2016 IEEE 25th International Conference on*, pages ***–***, 2015.

[MJMR15d] Hana M'Hemdi, Jacques Julliand, Pierre-Alain Masson, and Riadh Robbana. Non-deterministic timed pushdown automata-based testing evaluated by mutation. In *Enabling Technologies : Infrastructure for Collaborative Enterprises (WETICE), 2015 IEEE 24th International Conference on*, pages 198–203, 2015.

[Mor00] P. Morel. *Une algorithmique efficace pour la génération automatique de tests de conformité*. PhD thesis, Université de Rennes I, France, February 2000.

[MS85] David E. Muller and Paul E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37 :51 – 75, 1985.

[Mye79] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[PTD06] Yann Ponty, Michel Termier, and Alain Denise. GenRGenS : Software for Generating Random Genomic Sequences and Structures. *Bioinformatics*, 22(12) :1534–1535, 2006.

[Pur72] Paul Purdom. A sentence generator for testing parsers. *Bit Numerical Mathematics*, 12 :366–375, 1972.

[RAY87] Osi conformance testing. *Computer Networks and ISDN Systems*, 14(1) :79 – 98, 1987.

[Rob95] Riadh Robbana. *Spécification et vérification de systèmes hybrides*. PhD thesis, Grenoble 1, 1995. Th. : informatique.

[SHJ11] Rupert Schlick, Wolfgang Herzner, and Elisabeth Jöbstl. Fault-based generation of test cases from uml-models - approach and some experiences. In *SAFECOMP*, volume 6894 of *Lecture Notes in Computer Science*, pages 270–283. Springer, 2011.

[Sip96] Michael Sipser. Introduction to the theory of computation. *SIGACT News*, 27(1) :27–29, 1996.

[SK06] Fares Saad Khorchef. *A Framework for Robustness Testing of Communicating Protocols*. PhD thesis, Université Sciences et Technologies - Bordeaux I, December 2006.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute : A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5) :263–272, 2005.

[SPKM08] P. Vijay Suman, Paritosh K. Pandya, Shankara Narayanan Krishna, and Lakshmi Manasa. Timed automata with integer resets : Language inclusion and expressiveness. In *Formal Modeling and Analysis of Timed Systems, 6th International Conference, FORMATS 2008, Saint Malo, France, September 15-17, 2008. Proceedings*, pages 78–92, 2008.

[SVD01] Jan Springintveld, Frits Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1–2) :225 – 257, 2001.

[SY96] J. Sifakis and S. Yovine. Compositional specification of timed systems. In *STACS 96. 13th Annual Symposium on Theoretical Aspects of Computer Science. Proceedings*, pages 347 – 59, 1996.

[Tre93] Jan Tretmans. A formal approach to conformance testing. In *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems, Pau, France, 28-30 September, 1993*, pages 257–276, 1993.

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

[Tre99] Jan Tretmans. Testing concurrent systems : A formal approach. In *Concurrency Theory*, volume 1664 of *LNCS*, pages 46–65. Springer, 1999.

[Tri05] Stavros Tripakis. Checking timed büchi automata emptiness efficiently. In *Formal Methods in System Design*, pages 267–292, 2005.

[TW10] Ashutosh Trivedi and Dominik Wojtczak. Recursive timed automata. In *ATVA'10*, volume 6252 of *LNCS*, pages 306–324. Springer, 2010.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing : A Tools Approach*. Morgan Kaufmann Publishers Inc., 2007.

[Wal00] Igor Walukiewicz. Model checking CTL properties of pushdown systems, December 2000.

[Was77] A. Wasserman. *On the Meaning of Discipline in Software Design and Developmen*. Software Engineering Techniques, Infotech State of the Art Reportt. 1977.

[WMMR05] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. *Dependable Computing - EDCC 5 : 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005. Proceedings*, chapter PathCrawler : Automatic Generation of Path Tests by Combining Static and Dynamic Analysis, pages 281–292. Springer Berlin Heidelberg, 2005.

[XZC11] Zhiwu Xu, Lixiao Zheng, and Haiming Chen. A toolkit for generating sentences from context-free grammars. *Int. J. Software and Informatics*, 5(4) :659–676, 2011.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4) :366–427, 1997.

[ZW09] L. Zheng and D. Wu. A sentence generation algorithm for testing grammars. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, pages 130–135, 2009.

Résumé :

La vérification et la validation des composants logiciels des systèmes temps réel est un des enjeux majeurs pour le développement de systèmes automatisés. Les modèles de tels systèmes doivent être vérifiés, et la conformité de leur implémentation par rapport à leur modèle doit être validée. Nous nous plaçons dans le cadre des systèmes récursifs temps réels modélisables par des automates à pile temporisés avec deadlines (*TPAIO*). Les deadlines imposent des conditions de progression du temps. L'objectif de cette thèse est de proposer des méthodes de génération de tests pour les *TPAIO*.

Nos contributions sont les suivantes. Premièrement, une relation de conformité pour les *TPAIO* est introduite. Deuxièmement, une méthode polynomiale de génération de tests à partir d'un *TPAIO* déterministe avec deadline *lazy* est définie. Elle consiste à définir un algorithme de calcul d'un automate temporisé d'accessibilité incomplet en respectant les contraintes de pile. Cette méthode est incomplète. L'incomplétude n'étant pas un problème car l'activité de test est par essence incomplète. Troisièmement, nous définissons une méthode générant des cas de tests à partir d'un *TPAIO* déterministe avec sorties seulement et deadline *delayable* seulement. Elle d'applique aux abstractions de programmes récursifs temporisés. Elle consiste à générer des cas de tests en calculant un testeur sur-approximé. Finalement, nous avons proposé une généralisation du processus de génération de tests à partir d'un *TPAIO* général avec entrées/sorties et avec deadlines quelconques. La capacité de cette dernière méthode à détecter des implémentations non conformes est évaluée par une technique de mutation.

Mots-clés : systèmes temps-réel, automate temporisé, automate à pile temporisé, deadline, test, vérification, mutation.

Abstract:

The verification and validation of software components for real-time systems is a major challenge for the development of automated systems. The models of such systems must be verified and the conformance of their implementation w.r.t their model must be validated. Our framework is that of real-time recursive systems modelled by timed pushdown automata with deadlines (*TPAIO*). The deadlines impose time progress conditions. The objective of this thesis is to propose test generation methods from *TPAIO*.

Our contributions are as follows. Firstly, a conformance relation for *TPAIO* is introduced. Secondly, a polynomial method of test generation from a deterministic *TPAIO* with only *lazy* deadlines is defined. It consists of defining a polynomial algorithm that computes a partial reachability timed automaton by removing the stack constraints. This method is incomplete. The incompleteness is not a problem because software testing is an incomplete activity by nature. Thirdly, we define a method for generating test cases from a deterministic *TPAIO* with only outputs and *delayable* deadlines. It applies to the abstractions of timed recursive programs. It consists of generating test cases by computing an over-approximated tester. Finally, we propose a generalization of the test generation process from a non deterministic *TPAIO* with any deadlines. Its ability to detect non conform implementation is assessed by a mutation technique.

Keywords: real-time systems, timed automata, timed pushown automata, deadline, test, verification, mutation.