Model Based Testing : principes et applications dans le cadre temporisé

Antoine Rollet
Université de Bordeaux (LaBRI - CNRS UMR 5800)
Talence - France
http://www.labri.fr/~rollet
Email: rollet@labri.fr

***Abstract*—Le but de ce document est de présenter les principes du test de conformité à base de modèles dans le cadre de systèmes réactifs temporisés. Il s'agit d'un test en boîte noire où la spécification est décrite par un modèle formel qui sert de base à la génération de cas de test. Ces derniers sont appliqués sur l'implémentation afin d'obtenir un verdict. Après avoir rappelé quelques principes sur le test formel à base de modèles et les différentes approches, nous nous focalisons dans un premier temps sur les systèmes à base de transitions (IOLTS) et les méthodes de génération associées, de façon non déterministe ou par objectif de test. Ensuite, nous voyons comment étendre ces travaux dans le cadre temporisé en utilisant notamment une extension des automates temporisés d'Alur et Dill avec entrées/sorties. Enfin, nous évoquons une extension plus récente de ces travaux qui s'applique sur des systèmes temporisés à flot de données.**

I. Introduction

Compte tenu de la complexité croissante des applications informatiques, il est de plus en plus nécessaire de mettre en place des techniques de validation appropriées. Le test est une de ces techniques très répandue qui consiste à exécuter une implémentation réelle à l'aide d'un testeur (on parle d'IUT pour *Implementation Under Test*). En général[1], on distingue deux grandes catégories dans les méthodes de test :

- Le test en boîte blanche : on considère que l'on connaît la plupart des éléments du système, notamment le code de l'application (on parle de test structurel).
- Le test en boîte noire : l'implémentation est vue comme une boîte noire inconnue, dont on ne connaît que l'interface, et on se base sur la spécification de référence pour générer des cas de test (c'est donc du test fonctionnel).

Durant cette dernière décennie, les méthodes formelles ont permis de mettre au point des méthodologies de développement et de validation permettant d'augmenter la confiance envers certaines applications informatiques, notamment dans des domaines critiques. Dans cet état d'esprit, le test à *base de modèles*, ou MBT pour *Model Based Testing* consiste à utiliser une spécification décrite par un modèle formel comme référence pour générer des tests. Cette approche permet ainsi de fournir un cadre mathématique rigoureux dans un domaine où les aspects empiriques ont tendance à perdurer, et ainsi diminuer le coût des tests.

[1]Notons que ces définitions peuvent varier selon les communautés

Dans ce document, nous allons nous intéresser au test de conformité de systèmes réactifs. Un système réactif est un système qui interagit avec son environnement, ce qui est souvent le cas par exemple pour des applications critiques. Nous cherchons donc à nous assurer, en utilisant des cas de test, que l'IUT est bien conforme à sa spécification (ici formelle).

Nous allons montrer quelques approches possibles pour tester formellement des systèmes réactifs, mais il en existe bien d'autres (voir [1] pour une description plus large). L'objectif ici n'est donc pas de faire un état des lieux des méthodes de test, mais plutôt de se concentrer sur quelques unes afin d'identifier les problématiques sous-jacentes. Dans le domaine du test à base de modèles, différentes tendances se sont développées (provenant de communautés différentes) comme par exemple le test à base d'automates qui utilise essentiellement les machines à états finis - FSM pour *Finite State Machine* - (une synthèse est disponible dans [2] et [3]) ou encore le test fondé sur les systèmes de transitions et qui dérive essentiellement des algèbres de processus, que nous allons détailler dans la suite. Notons que ces tendances ont de nombreux points communs. Des travaux de test se sont aussi focalisés sur des systèmes décrits par des langages synchrones comme Lustre ([4], [5]). L'intérêt de cette approche est que le langage de programmation utilisé possède une sémantique, ce qui permet de le vérifier formellement, et de l'utiliser comme base pour des tests sur des critères aussi bien structurels que fonctionnels.

Nous allons donc nous focaliser sur le test à partir de Systèmes à Transitions Etiquetées (LTS pour *Labelled Transitions Systems*) et quelques extensions temporisées qui s'inspirent des automates temporisés d'Alur et Dill ([6]). Nous allons ainsi voir comment il est possible de générer des cas de test à partir d'un spécification, à la volée ou hors ligne à partir d'un objectif de test, et comment définir la conformité d'une IUT par rapport à sa spécification. Une étude ayant servi pour rédiger ce papier peut être trouvée dans [7].

Ce document est organisé de la manière suivante : la section II, présente une approche de test à base de LTS à entrées sorties (IOLTS), ensuite la section III montre une extension temporisée des IOLTS et propose un cadre de test adapté à ce modèle. La section IV décrit une approche de test spécifique dans le cas des systèmes temporisés à flot de données, et enfin

un bilan succinct est proposé en section V.

II. Test à base d'IOLTS

Dans cette section, nous allons énoncer les principes du test pour des systèmes modélisés par des machines à transitions étiquetées. Nous nous focaliserons sur les modèles à entrées sorties, les IOLTS (Input Output Labelled Transition System) ce qui est plus réaliste dans le cadre du test. En effet, l'implémentation et le testeur vont interagir : le fait d'utiliser des entrées et sorties permet de distinguer les éléments *contrôlables* par le testeur (les entrées de l'implémentation) des éléments *observables* par le testeur (les sorties de l'implémentation) qu'il ne peut qu'observer de façon passive. Les IOLTS permettent ainsi de modéliser les interactions entre testeur et implémentation en utilisant la composition d'automates. Notons que les premiers travaux sur ce genre de modèles proposaient aussi une approche théorique sans entrées sorties ([8]). Cette section reprend essentiellement les résultats de [8], [9] et [10].

A. Définitions et notations

Definition 1 (IOLTS)**.** *Un IOLTS (Input Output Labelled Transition System) est un quadruplet $M = (Q^M, A^M, \longrightarrow_M, q_0^M)$ tel que :*

- *Q^M est un ensemble fini non vide d'états*
- *$q_0^M \in Q^M$ est l'état initial*
- *A^M est un alphabet d'actions partitionné en trois ensembles disjoints tel que $A^M = A_I^M \cup A_O^M \cup I^M$ avec :*
  - *A_I^M l'alphabet d'entrées (notées avec un ?)*
  - *A_O^M l'alphabet de sorties (notées avec un !)*
  - *I^M l'alphabet des actions internes (notées τ_k)*
- *$\longrightarrow_M \subseteq Q^M \times A^M \times Q^M$ la relation de transition.*

On définit $A_{VIS}^M = A_I^M \cup A_O^M$ l'ensemble des actions visibles.

Voici quelques notations usuelles sur les IOLTS.
Soit $M = (Q^M, A^M, \longrightarrow_M, q_0^M)$ un IOLTS[2], $\mu_k \in A^M$ des actions quelconques, $a_k \in A_{VIS}^M$ des actions visibles et $\tau_k \in I^M$ des actions internes et $\sigma \in {A_{VIS}^M}^*$ une séquence d'actions visibles, et q, q' et $q_k \in Q^M$ des états. On note

- $q \xrightarrow{\mu}_M q'$ pour $(q, \mu, q') \in \longrightarrow_M$,
- $q \xrightarrow{\mu}_M$ pour $\exists q'$ tel que $q \xrightarrow{\mu}_M q'$
- $q \xrightarrow{\mu_1 \dots \mu_n} q'$ pour $\exists q_0, \dots q_n : q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_n} q_n = q'$,
- $\Gamma_M(q) \triangleq \{\mu \in A^M | q \xrightarrow{\mu}_M\}$ l'ensemble des actions tirables en q, et notamment
  - $Out_M(q) \triangleq \Gamma_M(q) \cap A_O^M$ les sorties tirables.
  - $In_M(q) \triangleq \Gamma_M(q) \cap A_I^M$ les entrées tirables.

Comme nous sommes dans un cadre de test, les raisonnements se font essentiellement sur les traces d'exécution visibles. Pour cela il est nécessaire d'introduire :

- $q \overset{\epsilon}{\Rightarrow} q' \triangleq q = q' \vee q \xrightarrow{\tau_1 \tau_2 \dots \tau_n} q'$
- $q \overset{a}{\Rightarrow} q' \triangleq \exists q_1, q_2 : q \overset{\epsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Rightarrow} q'$ (ϵ étant la séquence vide).

[2]en cas de non ambiguïté, l'exposant pourra être omis

Ensuite, on étend ces notations pour les séquences d'actions visibles :

- $q \overset{a_1 \dots a_n}{\Rightarrow} q' \triangleq \exists q_0, \dots, q_n : q = q_0 \overset{a_1}{\Rightarrow} q_1 \dots \overset{a_n}{\Rightarrow} q_n = q'$
- $q \overset{\sigma}{\Rightarrow} \triangleq \exists q' : q \overset{\sigma}{\Rightarrow} q'$.

Ainsi il est possible de définir l'ensemble des traces possibles à partir d'un état :

$Traces(q) \triangleq \{\sigma \in {A_{VIS}^M}^* | \, q \overset{\sigma}{\Rightarrow}\}$, et par extension $Traces(M) \triangleq Traces(q_0^M)$,

et l'ensemble des états accessibles à partir d'un état :

q *after* $\sigma \triangleq \{q' \in Q | \, q \overset{\sigma}{\Rightarrow} q'\}$.

On généralise de façon habituelle à l'IOLTS en considérant M *after* σ pour q_0^M *after* σ et pour un ensemble d'états $P \subseteq Q^M$, P *after* $\sigma \triangleq \bigcup_{q \in P} q$ *after* σ.

La figure 1 donne un exemple d'IOLTS. Ce dernier nous servira de spécification, notée S par la suite.

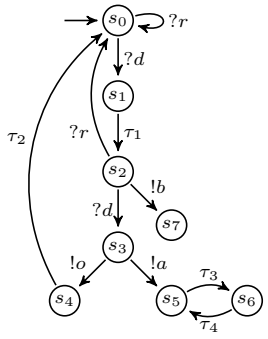


Figure 1. Une spécification S sous forme d'IOLTS

Sur cet IOLTS, on a par exemple :

- $\Gamma(s_0) = \{?d, ?r\}$,
- $Out(s_0) = \emptyset$,
- $In(s_0) = \{?d, ?r\}$,
- $s_1 \overset{\epsilon}{\Rightarrow} s_2$,
- $s_1 \overset{?d}{\Rightarrow} s_3$,
- $s_1 \overset{?d.!o}{\Rightarrow} s_0$,
- s_2 *after* $?d.!o = \{s_0, s_4\}$,
- s_0 *after* $?d.!a = \emptyset$,
- $\{s_0, s_2\}$ *after* $?d = \{s_1, s_2, s_3\}$,
- $Traces(S) = Traces(s_0) = \{\epsilon, ?d, ?r, ?d.?r, ?r.?d, ?d.!b, ...\}$.

B. Propriétés et hypothèses pour le test

Nous cherchons à vérifier qu'une implémentation sous test (IUT pour Implementation Under Test) est conforme à sa spécification S. Nous supposons que S est un IOLTS $(Q^S, A^S, \longrightarrow_S, s_0^S)$, sans restriction particulière, si ce n'est qu'il doit être non divergent (i.e. sans séquences infinies d'actions internes passant par une infinité d'états différents).

La notion de déterminisme est assez similaire à celle pour les automates d'états finis. Elle sera utilisée par la suite. Intuitivement, un IOLTS est déterministe s'il n'a aucune action interne, et si pour tout état q, il n'y a pas plusieurs transitions sortantes ayant la même étiquette. Formellement :

Definition 2 (IOLTS déterministe)**.** *Un IOLTS* $M = (Q^M, A^M, \longrightarrow_M, q_0^M)$ *est dit déterministe s'il n'a aucune action interne (i.e.* $I^M = \emptyset$*), et* $\forall q, q', q'' \in Q^M, \forall a \in A^M_{VIS}, (q \xrightarrow{a} q' \wedge q \xrightarrow{a} q'') \Rightarrow q' = q''$

L'IUT est une boîte noire, donc inconnue, mais elle est supposée modélisable par un IOLTS afin de pouvoir formaliser les interactions et définir la conformité avec la spécification de façon rigoureuse. On nomme cet IOLTS $IUT = (Q^{IUT}, A^{IUT}, \longrightarrow_{IUT}, s_0^{IUT})$, avec $A^{IUT} = A_I^{IUT} \cup A_O^{IUT} \cup I^{IUT}$. L'alphabet de IUT est supposé compatible avec la spécification S, i.e. $A_I^S \subseteq A_I^{IUT}$ et $A_O^S \subseteq A_O^{IUT}$. Afin d'éviter les blocages, on considère que l'IUT accepte n'importe quelle entrée de l'alphabet à tout moment, i.e. elle est *complète en entrée* :

Definition 3 (Complétude en entrée)**.** $\forall q \in Q, \forall a \in A_I, q \overset{a}{\Rightarrow}$.

Comme la plupart des raisonnements se font sur les traces visibles, il est parfois nécessaire de déterminer l'ensemble des traces d'un IOLTS, ce qui revient à le déterminiser :

Definition 4 (Déterminisation d'un IOLTS (det))**.** *Soit l'IOLTS* $M = (Q^M, A^M, \longrightarrow_M, q_0^M)$*, on définit l'IOLTS* $det(M)$ *par :* $det(M) = (2^{Q^M}, A^M_{VIS}, \longrightarrow_{det}, q_0^M \text{ after } \epsilon)$ *avec* $P \xrightarrow{a}_{det} P' \Leftrightarrow P, P' \in 2^{Q^M}, a \in A^M_{VIS}$ *et* $P' = P$ *after* a*.*

On remarque que $Traces(M) = Traces(det(M))$.

C. Blocages

Parfois il est possible que l'IUT se bloque. Dans ce cas, il ne s'agit pas forcément d'une erreur car il est possible que ce blocage soit spécifié. La théorie de test sur les IOLTS prend en compte ces aspects. En réalité il existe plusieurs sortes de blocages (pour $q \in Q$) :

- *deadlock* : aucune évolution possible : $\Gamma(q) = \emptyset$. C'est le cas de l'état s_7 figure 1.
- *outputlock* : le système attend une action de l'environnement : $\Gamma(q) \subseteq A_I^M$. C'est le cas de l'état s_0 figure 1.
- *livelock* : boucle d'actions internes : $\exists \tau_1, ...\tau_n : q \xrightarrow{\tau_1 ... \tau_n} q$. C'est le cas des états s_5 et s_6 figure 1.

De façon générale, un état ayant une de ces caractéristiques est appelé *quiescent*. On note l'ensemble de ces états $quiescent(M)$. Souvent en pratique, on les repère par des *timeout*. D'un point de vue formel, on considérera les blocages comme une sortie particulière, notée δ, qui sera rajoutée à la spécification. Le fait de les exprimer explicitement permet de les conserver en cas de déterminisation. L'IOLTS obtenu après ajout des états quiescents est appelé *Automate de suspension*. Il consiste simplement à ajouter une boucle étiquetée δ sur chaque état quiescent.

Definition 5 (Automate de suspension Δ)**.** *Soit l'IOLTS* $M = (Q^M, A^M, \longrightarrow_M, q_0^M)$*, on définit l'automate de suspension* $\Delta(M) = (Q^M, A^M \cup \{\delta\}, \longrightarrow_{\Delta(M)}, q_0^M)$ *avec* $\longrightarrow_{\Delta(M)} = \longrightarrow_M \cup \{q \longrightarrow_\delta q | q \in quiescent(M)\}$.

La figure 2 (gauche) montre l'automate de suspension $\Delta(S)$ correspondant à l'IOLTS S de la figure 1. La partie droite illustre sa déterminisation.

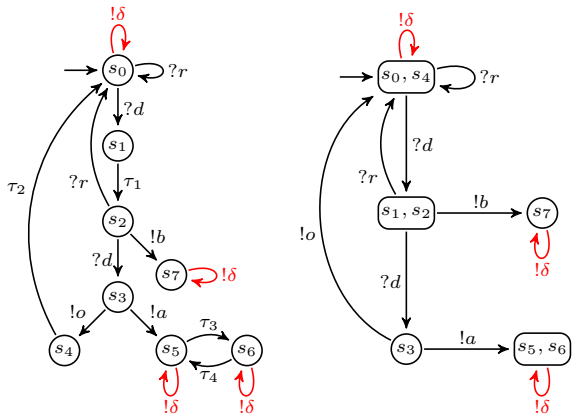


Figure 2. l'automate de suspension $\Delta(S)$ et sa déterminisation $det(\Delta(S))$

Les traces qui tiennent compte des blocages sont appelées *traces suspendues*, notées $STraces$. Il s'agit en fait des traces de l'automate de suspension :

$$STraces(M) \triangleq Traces(\Delta(M)).$$

D. Relation de conformité

Il existe beaucoup de relations de conformité dans la littérature. Il peut s'agir par exemple de relations d'équivalence (c'est le cas pour les machines de Mealy) ou encore de préordres. [11] propose une comparaison de différentes relations. Dans le cadre des IOLTS, c'est la relation **ioco** ([8]) qui s'est le plus répandu, car elle permet notamment de prendre en compte les entrées, les sorties et les blocages pour définir la conformité. L'idée intuitive est que pour toute trace suspendue commune à la spécification et à l'IUT, les sorties (y compris les blocages) de l'IUT doivent être incluses dans celles de la spécification. Formellement :

Definition 6 (ioco). *Soit* S *un IOLTS et* IUT *un IOLTS complet en entrées et compatible avec* S*,*

$$IUT \,\mathbf{ioco}\, S \triangleq \forall \sigma \in STraces(S), Out(\Delta(IUT) \text{ after } \sigma) \subseteq Out(\Delta(S) \text{ after } \sigma).$$

La figure 3 illustre la relation **ioco**. Sur cet exemple, on a IUT_1 **ioco** S (les sorties de IUT_1 sont toutes incluses dans S), IUT_2 **ioco** S (l'entrée $?b$ depuis l'état initial ne fait pas partie des traces de $\Delta(S)$), $\neg(IUT_3$ **ioco** $S)$ (la sortie $!z$ après $?a$ n'est pas autorisée par S), et $\neg(IUT_4$ **ioco** $S)$ (le blocage après $?a$ n'est pas autorisé par S).

E. Génération non-déterministe des tests

A partir d'une spécification S, il est possible de définir un IOLTS le plus général possible permettant de détecter la non conformité d'une implémentation par rapport à S pour la relation **ioco**. Il s'agit du *testeur canonique*. Il est construit à

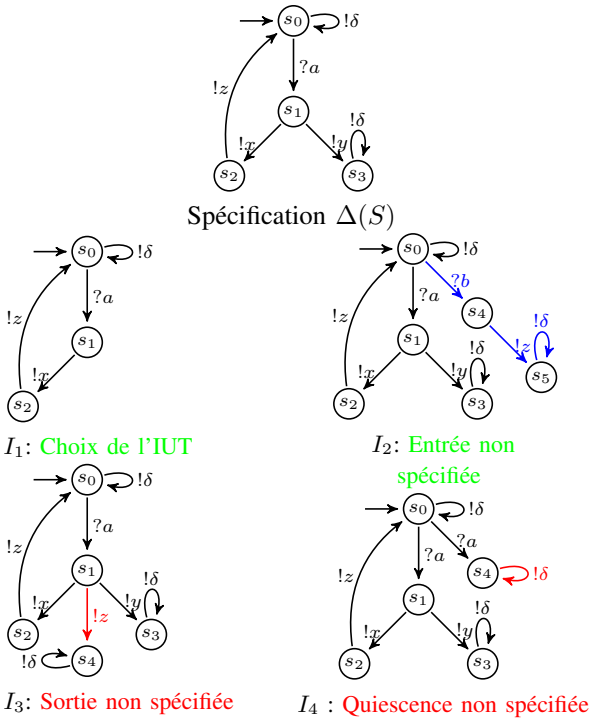


Figure 3. La relation **ioco**

partir de $det(\Delta(S)) = (Q^d, A^d, \longrightarrow_d, q_0^d)$, et d'un nouvel état **Fail**. Il est défini par $Can(S) = (Q^c, A^c, \longrightarrow_c, q_0^c)$ tel que :

- $Q^c = Q^d \cup \{\textbf{Fail}\}$ et $q_0^c = q_0^d$
- $A^c = A_I^c \cup A_O^c$ avec $A_I^c = A_O^d$ et $A_O^c = A_I^d$ les entrées du testeur sont les sorties de S et réciproquement.
- $\longrightarrow_c = \longrightarrow_d \cup \{q \xrightarrow{a}_c \textbf{Fail} \mid q \in Q^d, a \in A_I^c \wedge \neg(q \xrightarrow{a}_d)\}$, toutes les sorties non spécifiées (i.e. les entrées du point de vue testeur) mènent à **Fail**.

Le testeur canonique correspondant à S de la figure 1 est décrit figure 4. Pour simplifier les notations par la suite, les états ont été renommés.

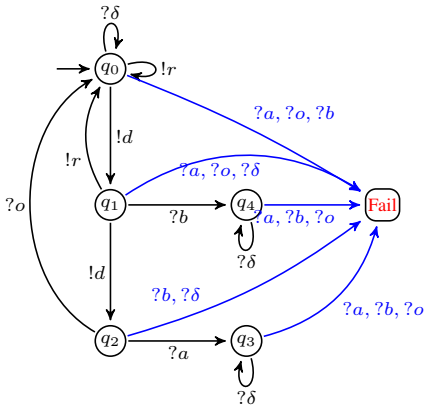


Figure 4. Le testeur canonique $Can(S)$

En notant $Traces_{Fail}$ l'ensemble de traces menant à **Fail**, il est possible de montrer que

$$IUT \textbf{ ioco } S \Leftrightarrow STracesIUT \cap Traces_{Fail}(Can(S)) = \emptyset.$$

Afin de décrire les interactions entre le testeur et l'IUT, on utilise un *cas de test*, qui est en fait un IOLTS muni d'états particuliers permettant d'identifier les verdicts. On pourrait penser à utiliser directement le testeur canonique, mais il est en général trop compliqué, et il ne permet pas de choisir quelle entrée appliquer sur l'IUT en cas de possibilités multiples. Le cas de test doit être capable de déterminer le verdict pour toute sortie de l'IUT ce qui nécessite une complétion sur toutes les entrées de l'alphabet. En revanche, comme le testeur contrôle les entrées de l'IUT, il n'est pas nécessaire de permettre plusieurs choix au niveau des entrées. Nous verrons par la suite comment générer ces cas de test. Ensuite, il reste à exprimer l'exécution d'un cas de test sur une IUT, ce qui permet d'identifier les traces menant aux différents verdicts. Cette exécution est modélisée à partir de la notion de composition parallèle entre deux automates.

Definition 7 (Cas de test)**.** *Un cas de test est un IOLTS déterministe* $(Q^{TC}, A^{TC}, \longrightarrow_{TC}, t_0^{TC})$ *qui comporte un ensemble d'états puits verdicts :* **Pass**, **Fail** *et* **Inconc** *de* Q^{TC} *tel que :*

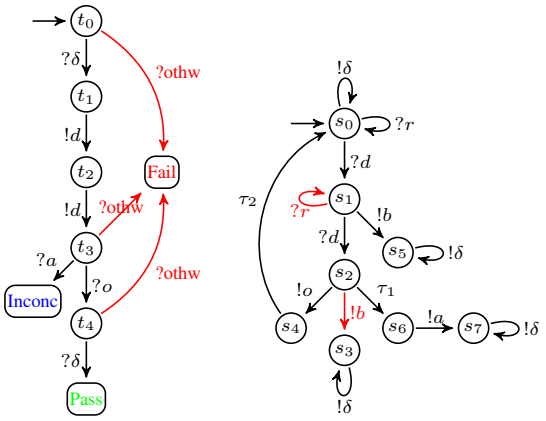
- $A^{TC} = A_I^{TC} \cup A_O^{TC}$ *avec* $A_O^{TC} = A_I^S$ *(TC envoie des entrées de S), et* $A_I^{TC} = A_O^S \cup \{\delta\}$ *(TC reçoit les sorties de S et les blocages).*
- TC *est contrôlable, i.e. il n'a jamais le choix entre plusieurs sorties ou entre des entrées et des sorties :* $\forall q \in Q^{TC}, (\exists a \in A_O^{TC}, q \xrightarrow{a}_{TC} \Rightarrow \forall b \in A^{TC}, (b \neq a \Rightarrow q \not\xrightarrow{b}_{TC}))$
- *Tous les états permettant une entrée, hormis les états verdicts, sont complets en entrée :* $\forall q \in Q^{TC}, (\exists a \in A_I^{TC}, q \xrightarrow{a}_{TC} \Rightarrow \forall b \in A_I^{TC}, q \xrightarrow{b}_{TC}$.

Soit une implémentation IUT et un cas de test TC. L'exécution de TC sur IUT, notée $TC \,||\, \Delta(IUT) = (Q^{TC} \times Q^{IUT}, A^{IUT}, \longrightarrow_{TC||\Delta(IUT)}, (q_0^{IUT}, t_0^{TC})))$ est la composition parallèle de TC et IUT avec synchronisation sur les actions visibles communes. La définition précise se trouve dans [9]. Il est possible de montrer que l'exécution d'un cas de test sur une implémentation n'est jamais bloquée. La figure 5 donne un exemple de cas de test $TC1$ et d'exécution sur une implémentation IUT.

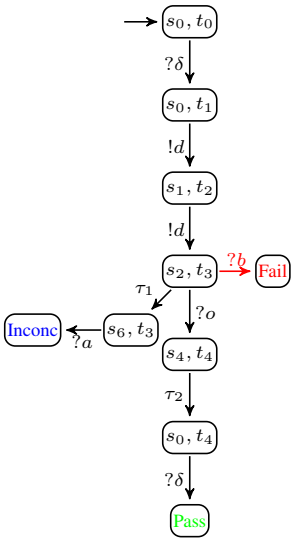
Ainsi, on considère que l'IUT n'est pas conforme s'il existe une exécution de $TC||\Delta(IUT)$ qui mène à **Fail**. Dans ce cas, on notera $TC\ fails\ IUT$. Notons qu'il s'agit d'une possibilité de rejet, car d'autres exécutions peuvent parfois produire des verdicts **Pass** ou **Inconc**. Par la suite, on considérera une *suite de tests* comme un ensemble de *cas de test*.

Afin de s'assurer de la pertinence d'une suite de test, on cherche à établir un lien entre celle-ci et la relation de conformité. Pour cela, on définit les notions suivantes. Soit $\mathcal{I}$ l'ensemble des implémentations compatibles avec la spécification S.

- Un cas de test TC est dit *non-biaisé* si seulement les implémentations non conformes sont rejetées, i.e. $\forall IUT \in \mathcal{I}, IUT \textbf{ ioco } S \Rightarrow \neg(TC\ fails\ IUT)$. Une suite



Cas de test $TC1$ Implémentation $\Delta(IUT)$



Exécution $TC1||\Delta(IUT)$

Figure 5. Exemple de cas de test et d'exécution

de tests est *non biaisée* si tous ses cas de test sont non biaisés.

- Une suite de tests TS est dite *exhaustive* pour S et **ioco** si toutes les implémentations non conformes sont détectées, i.e. $\forall IUT \in \mathcal{I}, \neg(IUT \,\mathbf{ioco}\, S) \Rightarrow \exists TC \in TS, TC \; fails \; IUT$.
- Une suite de tests est *complète* si elle est *non-biaisée* et *exhaustive*.

Dans l'idéal, on va donc chercher à générer des suites de test complètes. En pratique, compte tenu du grand nombre de possibilités, on va fournir des algorithmes de génération de cas de test qui sont non-biaisés, mais seulement exhaustifs en considérant tous les cas de test possibles produits par cet algorithme, ce qui n'est pas réaliste en pratique.

La façon la plus simple pour générer une suite de tests complète, c'est de partir du testeur canonique. Il suffit donc d'extraire des tests parmi tous les comportements décrits par $Can(S)$. L'algorithme est décrit dans [8], nous allons juste présenter les grandes lignes. Il propose à chaque étape de choisir (aléatoirement) entre les trois possibilités suivantes (supposons qu'on se trouve à l'état P) :

- On s'arrête en émettant le verdict **Pass**
- On choisit d'émettre une entrée x de l'IUT (si possible), et on rappelle récursivement cette fonction sur P *after* x
- On choisit d'écouter les sorties de l'IUT et d'exprimer un verdict **Fail** en cas de sortie non autorisée. En cas de sortie autorisée y, on rappelle récursivement cette fonction sur P *after* y.

Notons que cet algorithme génère une suite de test non biaisée et exhaustive (pour tous les cas de test possibles). Il n'y a pas d'états *Inconc* qui seront utilisés avec les objectifs de test. La figure 6 montre deux possibilités de cas de tests générés par cet algorithme.

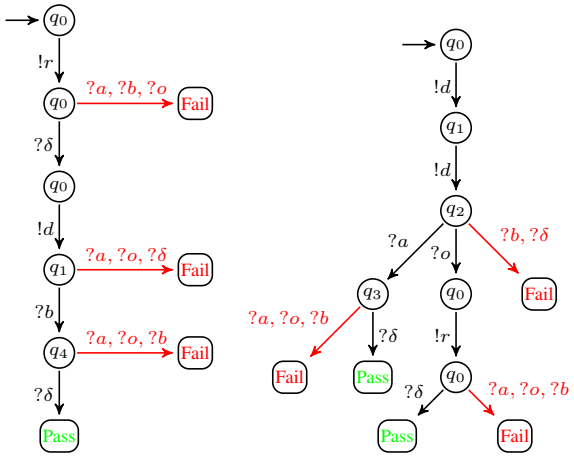


Figure 6. Deux exemples de cas de test générés de façon non déterministe

F. Génération par objectif de test

Dans l'algorithme de génération précédent, il n'est pas possible de cibler un comportement particulier que l'on souhaite tester. Pour résoudre cette faiblesse, [9] propose de générer les tests à partir d'un objectif de test, lui même sous forme d'IOLTS. Cette approche permet de limiter le nombre de cas de tests. Elle se décompose en plusieurs étapes que nous allons détailler par la suite :

1) construction de l'automate de suspension de S puis déterminisation $(det(\Delta(S)))$
2) construction du produit synchronisé visible $PS^{VIS} = Can(S) \times TP$
3) élagage
4) contrôlabilité et sélection de test

Dans un premier temps, il faut exprimer les comportements ciblés à l'aide d'un objectif de test. Notons que les états Accept décrivent les comportements que l'on souhaite privilégier, et les états Refuse servent à décrire les comportements que l'on ne souhaite pas tester (et en aucun cas les comportements erronés!). C'est ce qui permet de limiter l'exploration de la spécification.

Definition 8 (Objectif de test (Test Purpose)). *Un objectif de test est un IOLTS* $TP = (Q^{TP}, A^{TP}, \longrightarrow_{TP}, q_0^{TP})$ *déterministe et complet (i.e.* $\forall q \in Q^{TP}, \forall a \in A^{TP}, q \xrightarrow{a}_{TP}$*) muni de deux ensembles d'états puits disjoints* Accept^{TP} *et* Refuse^{TP} *de* Q^{TP} *tel que* $A^{TP} = A^S_{VIS} \cup \{\delta\}$ *(les actions visibles et les blocages de* S*). Notons que les états puits sont aussi complets.*

Afin de simplifier les notations, la complétude sera faite de façon implicite à l'aide du symbole $*$. En pratique, cela permet de décrire des comportements partiels, et de les compléter facilement ensuite.

La figure 7 (droite) montre un exemple d'objectif de test. Celui-ci stipule que l'on souhaite vérifier qu'après l'entrée d'un $?r$, le système va bien émettre un $!o$. Toutefois, nous ne souhaitons pas tester les comportements où une deuxième entrée $?r$ serait appliquée avant l'émission de $!o$. La partie gauche de la figure rappelle simplement $Can(S)$ qui nous servira dans l'exemple par la suite.

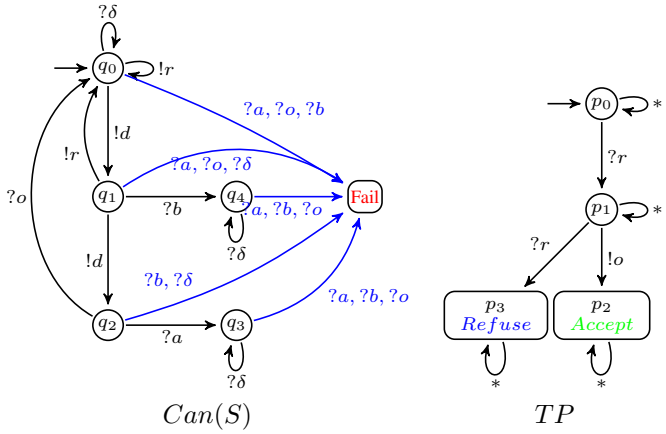


Figure 7. Rappel de $Can(S)$ et exemple d'objectif de test TP

Pour sélectionner les comportements de $Can(S)$ reconnus par TP, on utilise le produit synchrone :

Definition 9 (Produit Synchrone). *Soit* $M_1 = (Q^{M1}, A, \longrightarrow_{M1}, q_0^{M1})$*, et* $M_2 = (Q^{M2}, A, \longrightarrow_{M2}, q_0^{M2})$ *deux IOLTS. Le produit synchrone de* M_1 *et* M_2 *est l'IOLTS* $M_1 \times M_2 = (Q^{M1} \times Q^{M2}, A, \longrightarrow, q_0^{M1} \times q_0^{M2})$ *avec* $\longrightarrow$ *défini par :* $(q_{M1}, q_{M2}) \xrightarrow{a} (q'_{M1}, q'_{M2}) \Leftrightarrow (q_{M1} \xrightarrow{a}_{M1} q'_{M1}) \wedge (q_{M2} \xrightarrow{a}_{M2} q'_{M2})$

Si on considère les deux IOLTS $Can(S) = (Q^c, A^S_{VIS} \cup \{\delta\}, \longrightarrow_c, q_0^c)$ et $TP = (Q^{TP}, A^S_{VIS} \cup \{\delta\}, \longrightarrow_{TP}, q_0^{TP})$, on va noter $PS_{VIS} = Can(S) \times TP$, dans lequel les états acceptants et refusants sont définis par : $\mathsf{Accept}_{VIS} = (Q^c - \{Fail\}) \times \mathsf{Accept}_{TP}$ et $Fail_{VIS} \times Q^{TP}$. La figure 8 montre un exemple de produit synchronisé.

Il est possible de montrer que

$Traces_{accept}(PS^{VIS}) = STraces(S) \cap Traces(TP)$ et $Traces_{fail}(PS^{VIS}) = Traces_{fail}(Can(S))$.

Ainsi, PS^{VIS} permet d'intégrer à la fois les états Accept et de donner un verdict *Fail* en cas de comportement erroné. Il reste encore à sélectionner un cas de test parmi les différentes

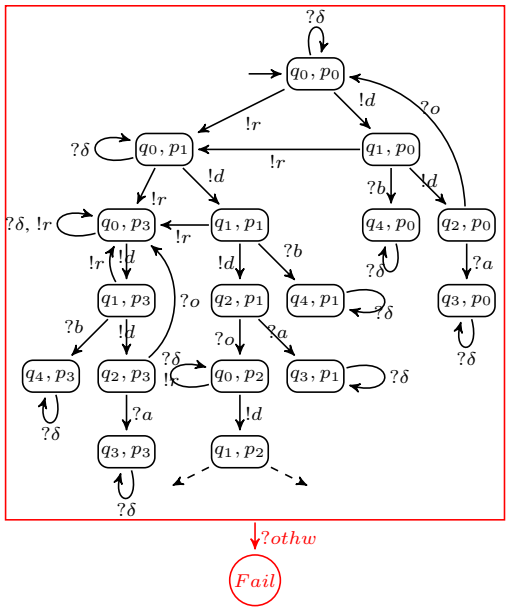


Figure 8. Le produit synchronisé $Can(S) \times TP$

possibilités offertes par PS^{VIS}. Pour cela, on va commencer par l'"élaguer" en supprimant certaines branches inutiles. C'est ce qu'on nomme le CTG pour *Complete Test Graph*. Sans rentrer dans les détails (voir [10]), l'élagage consiste à ne conserver que le premier état Accept atteint dans un chemin (celui-ci devient *Pass*), et à supprimer les autres états Accept. Ensuite, on identifie l'ensemble des états co-accessibles depuis les états *Pass*, noté $coreach(Pass)$. Puis, pour chaque état q de PS^{VIS} :

- si $q \in coreach(Pass)$ ou $q \in \{Fail\}$, q est conservé
- si q est successeur d'un état de $coreach(Pass)$ par une sortie (i.e. une entrée du point de vue testeur), q est conservé et devient *Inconc*. En effet, cette transition est non contrôlable.
- tous les autres états (et transitions) sont supprimés.

Le CTG correspondant à la figure 8 est donné figure 9.

Ensuite, il reste à extraire des cas de test en réglant notamment les *problèmes de contrôlabilité*. En effet, dans un cas de test, il ne doit y avoir qu'une sortie à la fois (i.e. une seule entrée pour l'IUT), car le testeur contrôle l'action qu'il souhaite envoyer, contrairement aux sorties. La figure 10 montre un exemple de cas de test issu de la figure 9. Remarquons que l'ensemble des cas de test que peut produire cet algorithme est *complet* (i.e. *exhaustif* et *non-biaisé*).

III. Extension des IOLTS dans le cadre temporisé

Dans cette section, nous allons présenter essentiellement les résultats de [12]. La raison de ce choix est que ces travaux sont une extension naturelle des méthodes de test sur les IOLTS avec la relation **ioco** présentées dans la section précédente, et qu'ils sont représentatifs des difficultés rencontrées dans ce genre de problèmes. Il existe effectivement d'autres approches, comme [13] ou [14], mais la démarche de base est similaire.

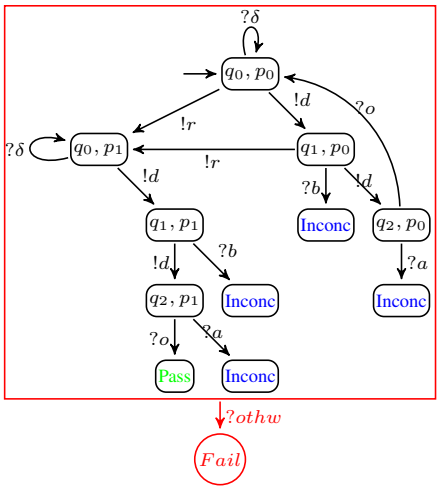


Figure 9. Un graphe de test complet (CTG)

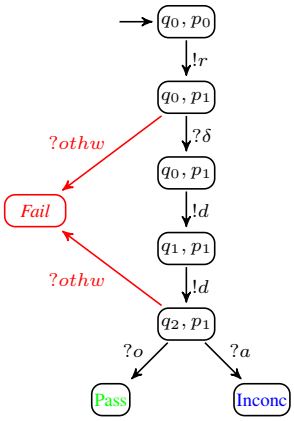


Figure 10. Un exemple de cas de test

C'est le cas aussi de l'extension temporisée d'UPPAAL ([15], [16]). On pourra trouver une comparaison de ces différentes méthodes dans [17]. [18] propose une approche assez similaire mais à base de FSM étendues temporisées qui combine la gestion des variables et le temps.

A. Définitions

Le modèle utilisé dans ces travaux est une extension des automates temporisés d'Alur et Dill ([6]) avec des entrées / sorties, et la possibilité d'urgences sur les transitions. Soit un ensemble fini d'actions Act partitionné en deux sous-ensembles disjoints, Act_I l'ensemble des entrées et Act_O celui des sorties. On définit aussi une *action non observable* $\tau \notin Act$. On note $Act_\tau = Act \cup \{\tau\}$. Le modèle utilisé est le TAIO défini par :

Definition 10 (TAIO)**.** *Un TAIO (Timed Automaton with Inputs and Outputs) est un quintuplet* $A = (Q, q_0, X, Act_\tau, E)$ *tel que :*

- Q *est un ensemble fini de* **localités**,
- $q_0 \in Q$ *est la* localité initiale,
- X *est un ensemble fini d'***horloges**,
- $Act_\tau = Act_I \cup Act_O \cup \{\tau\}$ *l'ensemble des* **actions**.
- E *est l'ensemble des* **transitions** (q, q', ψ, r, d, a) *avec*
  - q, $q' \in Q$ *les localités* **source** *et* **destination**
  - ψ *la* **garde** *i.e. une conjonction de contraintes* $x \# c$, $x \in X$, $c \in \mathbb{N}$, $\# \in \{<, \leq, \geq, >\}$,
  - $r \subseteq X$*: l'ensemble des horloges à réinitialiser*
  - $d \subseteq \{lazy, delayable, eager\}$ *la* **deadline**,
  - $a \in Act_\tau$ *l'***action**

Les transitions de type $eager$ ne sont pas autorisées avec des gardes de la forme $x > c$. Si un TAIO ne contient aucune τ transition, on dit qu'il est *observable*.

Afin de décrire précisément le comportement d'un TAIO A, il est nécessaire d'utiliser une sémantique formelle. Il s'agit ici d'un IOLTS infini appelé TIOLTS (Timed IOLTS) défini par :

- S_A un ensemble infini d'**états** $s = (q, v)$ avec $q \in Q$ une localité, $v : X \to \mathbb{R}^+$ une valuation d'horloges,
- $s_0^A = (q_0, \vec{0})$ l'état initial ($\vec{0}$ est la valuation d'horloges qui les affecte toutes à 0),
- T_d les **transitions discrètes:** $(q, v) \xrightarrow{a} (q', v')$ ssi $\exists (q, q', \psi, r, d, a) \in E, v \models \psi \wedge v' = reset(r)$ dans v (i.e. v' est la valuation obtenue à partir de v en réinitialisant toutes les horloges de r).
- T_t les **transitions temporelles:** $(q, v) \xrightarrow{t} (q, v+t)$ pour $t \in \mathbb{R}^+$ ssi il n'existe pas de transition $(q, q', \psi, r, d, a) \in E$ telle que $d = delayable \wedge \exists 0 \leq t_1 < t_2 \leq t, v + t_1 \models \psi \wedge v + t_2 \not\models \psi$ ou $d = eager \wedge v \models \psi$.

Un exemple de TAIO est donné figure 11.

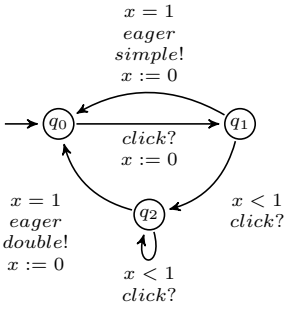


Figure 11. Un exemple de TAIO décrivant un double-clic

Pour un ensemble d'actions Act, on note

- $RT(Act)$ l'ensemble des *séquences temps-réel* $(Act \cup \mathbb{R}^+)^*$ muni de la séquence vide ϵ.

Etant donné $Act' \subseteq Act$ et $\rho \in RT(Act)$, on notera

- $P_{Act'}(\rho)$ la *projection* de ρ sur Act' en conservant les délais (i.e. on supprime les actions de ρ qui ne sont pas dans Act'). Par exemple, si $\rho = a\ 8\ b\ 1\ c\ 3$, alors $P_{\{a,c\}}(\rho) = a\ 8\ 1\ c\ 3$ c'est à dire $a\ 9\ c\ 3$.

Pour une séquence ρ, on définit

- $time(\rho)$ comme la somme des délais de ρ. Par exemple $time(a\ 8\ b\ 1c\ 3) = 12$ et $time(\epsilon) = 0$.

- De façon usuelle, un état $s \in S_A$ est dit atteignable s'il existe une séquence temps-réel permettant de l'atteindre depuis l'état initial, i.e. $\exists \rho \in RT(Act) : s_0^A \xrightarrow{\rho} s$.

Soit un TAIO A sur Act_τ. De façon similaire aux IOLTS, on dit que

- A est *complet en entrées* si $\forall s \in Reach(A), \forall a \in Act_I, s \xrightarrow{a}$, i.e. il accepte toute entrée à tout moment.
- Il est *déterministe* si $\forall s, s', s'' \in Reach(A), \forall a \in Act_\tau, s \xrightarrow{a} s' \wedge s \xrightarrow{a} s'' \Rightarrow s' = s''$,
- A est *non bloquant* s'il n'empêche pas le temps de s'écouler, i.e. $\forall s \in Reach(A), \forall t \in \mathbb{R}^+, \exists \rho \in RT(Act_O \cup \{\tau\}) : time(\rho) = t \wedge s \xrightarrow{\rho}$;
- on définit l'ensemble des *traces temporisées observables* par $TTraces(A) = \{P_{Act}(\rho) | \rho \in RT(Act_\tau) \wedge s_0^A \xrightarrow{\rho}\}$.

Remarquons que ces notions sont définies sur la sémantique du TAIO.

B. Relation de conformité

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. C'est ce qui explique pourquoi la relation est définie sur la sémantique.

Pour un TAIO A et une séquence observable $\sigma \in RT(Act)$, on définit A *after* σ l'ensemble des états de A atteignables par une séquence temporisée ρ dont la projection sur les actions observables est σ, i.e.

$$A \text{ after } \sigma \triangleq \{s \in S_A | \exists \rho \in RT(Act_\tau) : s_0^A \xrightarrow{\rho} s \wedge P_{Act}(\rho) = \sigma\}.$$

Pour un état $s \in S_A$ on définit $elapse(s)$ comme l'ensemble de tous les délais possibles sans observer aucune action en s, i.e.

$$elapse(s) = \{t > 0 | \exists \rho \in RT(\{\tau\}) : time(\rho) = t \wedge s \xrightarrow{\rho}\}.$$

Comme dans le cadre de **ioco**, il est nécessaire de définir aussi l'ensemble des sorties observables dans un état donné $s \in S_A$, y compris l'écoulement du temps :

$$out(s) = \{a \in Act_O | s \xrightarrow{a}\} \cup elapse(s).$$

Cette notion ce généralise ensuite pour un ensemble d'états S :

$$out(S) = \bigcup_{s \in S} out(s).$$

Nous avons maintenant tous les éléments pour définir la relation de conformité. L'idée est que toutes les sorties (y compris l'écoulement du temps) possibles en un état de l'IUT après avoir observé une séquence de A_S doivent être autorisées (i.e. spécifiées) par A_S. On suppose que la spécification du système est un TAIO A_S non bloquant, et que l'implémentation peut être modélisée par un TAIO A_{IUT} inconnu, mais non bloquant et complet en entrées. La relation de conformité est donc définie par :

$$A_{IUT} \textbf{ tioco } A_S \triangleq \forall \sigma \in TTraces(A_S), out(A_{IUT} \text{ after } \sigma) \subseteq out(A_S \text{ after } \sigma).$$

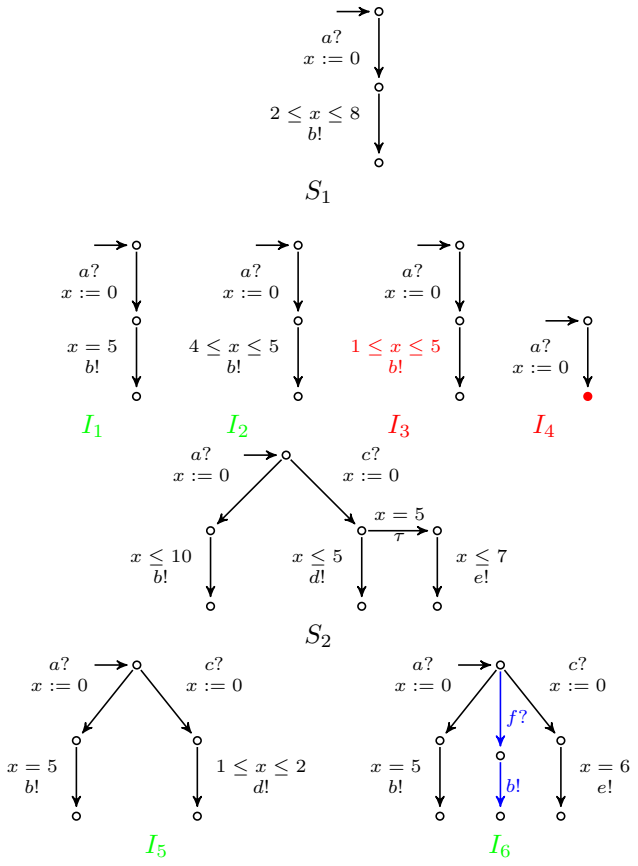


Figure 12. Exemples pour illustrer **tioco**

La figure 12 illustre la relation **tioco** sur quelques exemples. Dans celle-ci on considère que les transitions avec sortie sont *delayable*, et que les transitions avec entrée sont *lazy*. On constate que I_1 **tioco** S_1 et I_2 **tioco** S_1 car après réception de $a?$, $b!$ est émis dans les deux cas dans un temps qui est bien entre $x = 2$ et $x = 8$. En revanche, $\neg(I_3$ **tioco** $S_1)$ car $out(I_3 \text{ after } a\,1) = (0;4] \cup \{b!\}$ et $\neg(I_4$ **tioco** $S_1)$ car $out(I_4 \text{ after } a\,1) = (0;\infty]$ alors que $out(S_1 \text{ after } a\,1) = (0;7]$. Par ailleurs, I_5 **tioco** S_2 car toutes les sorties de I_5 sont autorisées par S_2 et enfin I_6 **tioco** S_2 car la trace passant par $f?$ n'est pas spécifiée dans S_2.

C. Test analogique

Nous allons maintenant voir comment générer des cas de test à partir d'un TAIO pour la relation **tioco** . Dans cette partie, nous allons considérer le temps dense, i.e. on va considérer que le testeur est muni d'une horloge, et qu'il peut mesurer les instants de façon exacte. Cette approche est la plus simple et la plus intuitive, mais nous verrons qu'elle s'avère difficilement réalisable en pratique. On parle donc de *testeur analogique* car il est muni d'une *horloge analogique*.

Pour une spécification A_S sur l'alphabet Act_τ, un *cas de test analogique* est une fonction $T : RT(Act) \longrightarrow Act_I \cup \{wait, pass, fail\}$, qui pour chaque séquence $\rho \in Act$ précise ce que le testeur doit effectuer :

- soit émettre une entrée,

- soit attendre (et écouter),
- soit émettre un verdict.

Sans rentrer dans les détails, le testeur doit satisfaire certaines conditions, notamment le fait d'atteindre un verdict en un temps borné. Notons qu'en fait T définit un LTS dont les états sont les séquences $\rho \in RT(Act)$. Il est parfois possible de représenter T à l'aide d'un TAIO (mais pas toujours).

L'exécution d'un test T sur l'implémentation A_{IUT} est défini comme la composition parallèle des LTS définis par T et A_{IUT} (noté $A_{IUT}||T$), en synchronisant les transitions de même étiquette. Notons qu'en cas d'implémentation non déterministe, il est possible que le verdict atteint ne soit pas le même sur plusieurs exécutions. On dit que A_{IUT} passe le test, noté

$$A_{IUT}\ passes\ T \triangleq \textit{Fail} \notin Reach(A_{IUT}||T).$$

On généralise cette notation pour une suite de tests $\mathcal{T}$. On considère que $\mathcal{T}$ est *non biaisée* si

$$\forall A_{IUT}, A_{IUT}\ \textbf{tioco}\ A_S \Rightarrow A_{IUT}\ passes\ \mathcal{T}$$

et qu'elle est *exhaustive*[3] si

$$\forall A_{IUT}, A_{IUT}\ passes\ \mathcal{T} \Rightarrow A_{IUT}\ \textbf{tioco}\ A_S.$$

Nous allons maintenant décrire l'algorithme de génération proposé dans [12]. Celui-ci va calculer à tout moment dans quel état (symbolique) on se trouve lors de la génération à partir de A_S. Pour cela, il est nécessaire de définir les *successeurs* : soit un ensemble d'états P d'un TAIO A_S, $a \in Act$ et $t \in \mathbb{R}^+$. On définit

- les *successeurs discrets* : $dsucc(P,a) = \{s' \mid \exists s \in P, s \xrightarrow{a} s'\}$
- et les *successeurs temporels* : $tsucc(P,t) = \{s' \mid \exists s \in P, \exists \rho \in RT(\tau), time(\rho) = t \wedge s \xrightarrow{\rho} s'\}$.

L'algorithme de génération est inspiré de celui de TorX ([8]) sur les IOLTS. En partant de l'état $P_0 = tsucc(\{s_0^{A_S}, 0\})$, le testeur choisit entre les trois règles suivantes :

- attendre : si le testeur reçoit une action b après un délai t, alors $P := dsucc(tsucc(P,t), b)$; ou si aucune action n'est reçue après un délai t, alors $P := tsucc(P,t)$. Si $P = \emptyset$ le test émet *Fail*.
- émettre une action : pour toute entrée a, si $dsucc(P,a) \neq \emptyset$, émettre a, et $P := dsucc(P,a)$
- arrêter le test et émettre le verdict *Pass*

Il est possible de montrer que les tests obtenus sont non-biaisés et exhaustifs (pour tous les cas de test possibles). [12] montre aussi que ces tests sont *stricts*, i.e. qu'ils détectent la non conformité au plus tôt.

Un exemple de test généré par cet algorithme est donné figure 13, sous forme de fonction et sous forme de TAIO.

D. Test digital

En pratique, il est difficile de dater de façon précise les événements car l'horloge utilisée est discrète. [12] propose

[3]Selon les communautés, on parle de test complet au lieu d'exhaustif. C'est le cas de [12]. Cette différence peut induire une confusion dans les notations de ce papier. Par souci de cohérence, nous avons conservé les notations précédentes

$$\begin{array}{lcl} T(\epsilon) & = & a! \\ T(a!) & = & wait \\ T(a!t) & = & wait, \forall t \leq 8 \\ & = & fail, \forall t > 8 \\ T(a!tb?) & = & fail, \forall t \leq 2 \\ & = & pass, \forall t \in [2,8] \end{array}$$

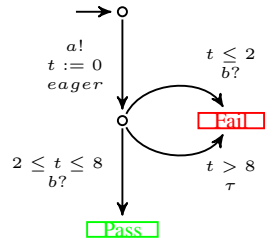


Figure 13. Un exemple de test analogique et son TAIO associé

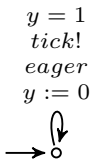


Figure 14. L'horloge digitale

donc de mettre en place un test *digital* où le testeur utilise une horloge périodique sous forme d'automate pour mesurer le temps : il s'agit de l'*automate de tick* (noté $Tick$) qui est en fait simplement un TAIO qui émet un $tick$ toutes les unités de temps (voir figure 14).

De façon similaire au test analogique, pour une spécification A_S sur l'alphabet Act_τ en ajoutant une nouvelle action $tick$ n'appartenant pas à Act_τ, un test *digital* pour A_S est une fonction $D : (Act \cup \{tick\})^* \longrightarrow Act_I \cup \{wait, pass, fail\}$ qui va observer les actions d'entrées, de sorties, et les $tick$. Comme le testeur analogique, il définit aussi un LTS. Cette fois, l'exécution du test est définie par l'exécution parallèle $A_{IUT}||Tick||D$.

Les tests seront générés à partir de $A'_S = A_S||Tick$. En effet, ce produit permet d'identifier toutes les entrées et sorties, y compris les $tick$ périodiques. Il est aussi nécessaire de définir une nouvelle sorte de successeur (dit non observable) pour un ensemble d'états P de A'_S :

$$usucc(P) = \{s' \mid \exists s \in P, \exists \rho \in RT(\{\tau\}), s \xrightarrow{\rho} s'\}$$

qui représente l'ensemble des états successeurs de P par une séquence d'actions non observables. Il est important de noter que cet ensemble est toujours borné en temps, car la séquence en question finira forcément par être interrompue par un $tick$.

Il est maintenant possible de définir l'algorithme de génération, qui peut être utilisé à la volée ou offline. Le résultat se présente comme un LTS, dont l'état initial est $P_0 = \{s_0^{A'_S}\}$. Ensuite, on choisit l'une des règles suivantes :

- réception : $\forall b \in Act_O \cup \{tick\}$, si $P' = dsucc(usucc(P), b) \neq \emptyset$ alors $P \xrightarrow{b} P'$, sinon $P \xrightarrow{b} fail$

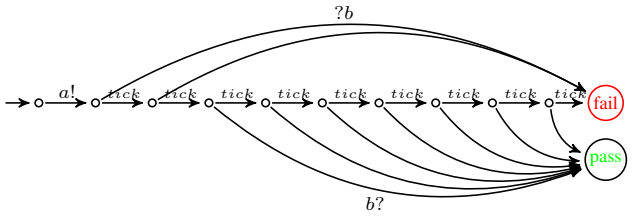


Figure 15. Un exemple de test digital

- émission : s'il existe une entrée $a \in A_I$ telle que $P'' = dsucc(tsucc(S, 0), a) \neq \emptyset$, alors $P \xrightarrow{a} P''$.
- arrêter le test et émettre le verdict *Pass*

On remarque qu'en cas d'émission, le testeur doit émettre instantanément l'action d'entrée.

La figure 15 montre un exemple de test digital obtenu à partir de cet algorithme sur S_1.

Il est possible de montrer que les tests générés son non biaisés. En revanche, il ne sont si stricts, ni exhaustifs. Cela est du au manque de précision de l'horloge.

IV. Cas des systèmes à flot de données : test à base de VDTA

Dans cette section, nous allons nous intéressons à des systèmes temporisés réactifs à flot de données. Ces systèmes interagissent avec leur environnement et vont réagir en fonction d'un ensemble d'événements. Les résultats sont essentiellement tirés de [19], [20] et [21]. Prenons l'exemple d'une commande *bi-manuelle* qui est un dispositif pour démarrer une machine dangereuse, et dont le cahier des charges est : *La commande est constituée de deux boutons (gauche (L) et droit (R)) et d'un contrôleur du démarrage et de l'arrêt de la machine (S). Initialement, la machine est arrêtée; elle démarre lorsque les deux boutons sont appuyés dans un intervalle de temps inférieur à* 1 *seconde. Lorsque qu'un seul bouton est appuyé et que le second n'est pas appuyé* 1 *seconde après le premier, la machine ne démarre pas tant que les deux boutons ne sont pas relâchés. Quand la machine est activée, elle s'arrête dès qu'un bouton est relâché. Le processus de démarrage est réinitialisé lorsque les deux boutons sont relâchés.* Le chronogramme de la figure 16 donne un exemple des comportements possibles, en notant S la sortie.

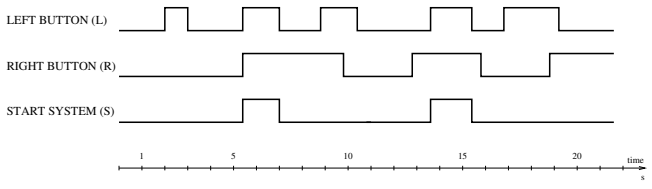


Figure 16. Quelques comportements de la commande bi-manuelle

Il est possible d'utiliser un $TAIO$ pour modéliser ce genre de systèmes. Toutefois, compte tenu des particularités comme la simultanéité des signaux, leur persistance, ainsi que l'aspect immédiat des réactions, [21] propose un autre modèle, dit *VDTA (Variable Driven Timed Automaton)* pour décrire ce genre de systèmes, et la génération de tests associée. Il s'agit en fait d'une sorte d'extension avec temps continu des modèles Lustre (cités précédemment) en utilisant les principes vus dans les automates temporisés. Ce modèle à horloges se base sur des transitions urgentes, et ce sont les variables (des entrées et des sorties) qui régissent le comportement de l'automate : les gardes portent sur les variables et les horloges, et chaque fois qu'une garde est vraie, la transition doit immédiatement être franchie. Ainsi, il est possible dans certains cas de franchir plusieurs transitions successives en temps zéro (comme en Esterel / Lustre par exemple). Notons que ce modèle ne prend pas en compte les transitions internes explicites.

A. Variable Driven Timed Automata

1) Définitions: Soit un ensemble donné de variables V, chaque variable $V_i \in V$ dans un domaine (possiblement infini) $Dom(V_i)$ dans $\mathbb{N}$, $\mathbb{Q}^+$ ou $\mathbb{R}^+$. Pour un ensemble de variables V, une affection est un n-uplet $\Pi_{i\in[1..n]}(\{V_i\} \times (Dom(V_i) \cup \{\perp\}))$ et on note $A(V)$ l'ensemble des affectations de variables pour V. Soit une valuation $v = (v_1, \cdots, v_n)$ de V et $A \in A(V)$, on définit les valuations $v[A]$ tel que : $v[A](V_i) = c$ si $(V_i, c) \in A$ et $c \neq \perp$, et sinon $v[A](V_i) = v_i$. Intuitivement, (V_i, c) de A doit affecter c à V_i si c est une constante de $Dom(V_i)$, sinon elle reste inchangée. $Var(A)$ correspond à l'ensemble des variables de V qui sont mises à jour par A, Id_V est l'affection identité, i.e. qui laisse les variables de V inchangées. On note $\mathcal{G}(V)$ l'ensemble des *contraintes* sur les variables, défini comme une combinaison de contraintes simples de la forme $V_i \bowtie c$ avec $V_i \in V$, $c \in Dom(V_i)$ et $\bowtie \in \{<, \leq, =, \geq, >\}$. Etant donné $G \in \mathcal{G}(V)$ et une valuation $v \in Dom(V)$, on écrit $v \models G$ quand $G(v) \equiv true$. On définit la projection de façon usuelle[4] : $Proj_{V_i}(G) \in \mathcal{G}(V_i)$ la projection de G sur V_i. Un VDTA est donc défini ainsi :

Definition 11. *Un* VDTA (Variable Driven Timed Automaton) *est un 7-uplet* $\mathcal{A} = \langle L, X, I, O, \ell^0, G^0, \Delta_\mathcal{A}\rangle$, *où* L *est un ensemble fini de localités,* X *est un ensemble fini d'horloges,* I *et* O *deux ensembles disjoints de variables d'entrées et de sorties,* $\ell^0 \in L$ *est la localité initiale,* $G^0 \in \mathcal{G}(I, O)$ *la condition initiale (une contrainte sur* $I \cup O$*), et* $\Delta_\mathcal{A} \subseteq L \times \mathcal{G}(I, O, X) \times A(O) \times 2^X \times L$ *est la* relation de transition.

Dans une transition $\langle \ell, G, A, \mathcal{X}, \ell'\rangle \in \Delta_\mathcal{A}$ (souvent écrite $\ell \xrightarrow{G,A,\mathcal{X}} \ell'$): $G \in \mathcal{G}(I, O, X)$ est une combinaison booléenne d'éléments de $\mathcal{G}(I)$, $\mathcal{G}(O)$ et $\mathcal{G}(X)$; $A \in A(O)$ est une affectation sur des variables de sortie, et $\mathcal{X} \in 2^X$ est un ensemble d'horloges réinitialisées au passage de la transition.

L'ensemble I des variables d'entrées représente les variables dans lesquelles l'environnement (e.g. le testeur) va écrire des valeurs. Il peut écrire n'importe quelle valeur du domaine à n'importe quel moment. L'ensemble O des variables de sortie représente les variables qui sont mises à jour par le système

[4]Attention, le projection usuelle n'est pas la même que celle de [12] définie dans la section précédente. Cette dernière conservait les délais, ce qui n'est pas le cas ici à moins que ça ne soit explicitement précisé.

au passage d'une transition. Le testeur peut les observer mais ne peut pas les contrôler. Par ailleurs, on considère que toutes les transitions sont urgentes, i.e. la transition doit absolument être franchie au moment où sa garde est vraie.

2) Sémantique et notations: La sémantique du VDTA est assez proche de celle utilisée pour les transitions *eager* des TAIO ([12]) ou encore de celle de IF ([22]). Un *état* d'un VDTA est de la forme (ℓ, i, o, x) où $\ell \in L$ est une localité, i, o et x sont respectivement les *valuations* des variables d'entrées, sorties et horloges. Une valuation est simplement une fonction qui renvoie les valeurs des variables. Si $A \in A(I)$ est une affectation sur les variables d'entrée, $i[A]$ change la valeur des variables d'entrée selon cette affectation. Soit x une valuation d'horloges, $\mathcal{X}$ un sous-ensemble des horloges, et $\delta \in \mathbb{R}^+$ un délai, la valuation $x + \delta$ ajoute δ à chaque valeur d'horloge, et $x[\mathcal{X} \leftarrow 0]$ met à zéro dans x les horloges de $\mathcal{X}$.

Definition 12. *La sémantique d'un VDTA $\mathcal{A}$ est un système de transition $[\![\mathcal{A}]\!] = \langle S_{\mathcal{A}}, s^0, \Sigma, \rightarrow\rangle$ où $S_{\mathcal{A}} = L \times Dom(I) \times Dom(O) \times \mathbb{R}^+$ est l'ensemble (infini) des Etats, $s^0 = (\ell^0, i^0, o^0, x^0)$ est l'état initial avec x^0 est la valuation d'horloges où elles sont toutes à 0 et (i^0, o^0) est la seule solution de G^0; $\Sigma = A(I) \cup A(O) \cup \mathbb{R}^+$ l'ensemble (infini) d'actions; et $\rightarrow$ est la relation de transition régie par les trois règles suivantes :*

- **T1** *mise à jour de sortie :* $(\ell, i, o, x) \xrightarrow{A} (\ell', i, o[A], x[\mathcal{X} \leftarrow 0])$ *s'il existe* $(\ell, G, A, \mathcal{X}, \ell') \in \Delta_{\mathcal{A}}$ *tel que* $(i, o, x) \models G$,
- **T2** *mise à jour d'entrée :* $(\ell, i, o, x) \xrightarrow{A} (\ell, i[A], o, x)$ *avec* $A \in A(I)$ *si* $\forall (\ell, G, A', \mathcal{X}, \ell') \in \Delta_{\mathcal{A}}$, $(i, o, x) \not\models G$.
- **T3** *transition temporelle :* $(\ell, i, o, x) \xrightarrow{\delta} (\ell, i, o, x + \delta)$ *avec* $\delta > 0$ *si pour tout* $\delta' < \delta$, *pour toute transition symbolique* $(\ell, G, \mathcal{X}', \ell') \in \Delta_{\mathcal{A}}$, *on a* $(i, o, x + \delta') \not\models G$.

La sémantique considère des *transitions discrètes (T1 et T2)* et des *transitions temporelles (T3)*. Les transitions discrètes concernent la mise à jour des variables de sorties (T1) ou d'entrées (T2). Les transitions temporelles marquent l'écoulement du temps. Les transitions de mise à jour de sortie (T1) doivent être tirées immédiatement dès que la contrainte est satisfaite. Les transitions de mise à jour d'entrées (T2) sont tirées par l'environnement. Cette sémantique donne priorité aux transitions de mise à jour de sortie. En effet, les autres transitions ne peuvent être tirées que lorsque qu'aucune autre garde n'est satisfaite, i.e. de façon équivalente si aucune transition de mise à jour des sorties n'est possible.

La figure 17 donne un exemple de VDTA correspondant à la spécification précédente.

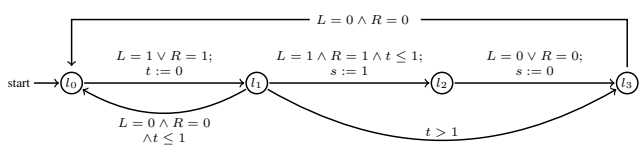


Figure 17. VDTA décrivant la commande bi-manuelle

Etant donné un état $s = (l, i, o, x) \in S$ de $[\![\mathcal{A}]\!]$, une action $a \in \Sigma$, une séquence $\sigma = a_1.a_2.\ldots.a_{k-1}.a_k$ de Σ^* on définit :

- $Out(s) \triangleq o$ donne accès à la valeur des sorties de $[\![\mathcal{A}]\!]$ à l'état s.
- $s \xrightarrow{a}$ s'il existe s' tel que $s \xrightarrow{a} s'$; sinon on écrit $s \not\xrightarrow{a}$
- $s \xrightarrow{\sigma} s'$ s'il existe $\{s_i\}_{i=1..k-1}$ tel que $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots.\xrightarrow{a_{k-1}} s_{k-1} \xrightarrow{a_k} s'$
- $s \xrightarrow{\sigma}$ s'il existe s' tel que $s \xrightarrow{\sigma} s'$.
- une *exécution (run)* est une séquence alternant états et actions : $r = s_0 b_1 s_1 \cdots b_n s_n$ dans $S.(\Sigma.S)^*$ tel que $\forall i \geq 0, s_i \xrightarrow{b_{i+1}} s_{i+1}$.
- $Run(s, [\![\mathcal{A}]\!])$ dénote l'ensemble des exécutions qui peuvent être appliquées sur $[\![\mathcal{A}]\!]$ en partant de l'état s,
- $Run([\![\mathcal{A}]\!]) = Run(s^0, [\![\mathcal{A}]\!])$.
- la *trace* $\rho(r)$ d'une exécution $r = s_0 b_1 s_1 \cdots b_n s_n$ est donnée par la séquence $\rho(r) = Proj_{\Sigma}(r) = b_1 \cdots b_n \in \Sigma^*$.
- $Tr([\![\mathcal{A}]\!]) = \{\rho(r) | r \in Run([\![\mathcal{A}]\!])\}$ est l'ensemble des traces générées par $\mathcal{A}$.

Voici deux exemples d'exécutions possibles du VDTA de la figure 17 (on lira sous le format $(loc., (L, R), S, t)$). On rappelle que la notation Id_O signifie qu'aucune variable de sortie n'est affectée (c'est l'identité).

- $(\ell_0, (0,0), 0, 0) \xrightarrow{L:=1} (\ell_0, (1,0), 0, 0) \xrightarrow{Id_O} (\ell_1, (1,0), 0, 0) \xrightarrow{0.3} (l_1, (1,0), 0, 0.3)$
- $(\ell_0, (0,0), 0, 0) \xrightarrow{L:=1, R:=1} (\ell_0, (1,1), 0, 0) \xrightarrow{Id_O} (\ell_1, (1,1), 0, 0) \xrightarrow{S:=1} (\ell_2, (1,1), 1, 0)$

3) Déterminisme, stabilité: Il est possible que plusieurs gardes soient vraies en même temps dans un état. C'est un cas de non déterminisme. Ainsi on considère qu'un VDTA $\mathcal{A}$ est *déterministe*

- si G_0 est satisfaite par au plus une valuation (i^0, o^0); et
- s'il existe $\ell \xrightarrow{G_1, A_1, \mathcal{X}_1} \ell_1$, $\ell \xrightarrow{G_2, A_2, \mathcal{X}_2} \ell_2$ avec $\ell_1 \neq \ell_2$, alors $G_1 \cap G_2$ est insatisfiable.

Intuitivement, certains états vont être traversés en temps nul. En effet, lorsqu'une exécution atteint un état et qu'à ce moment là, une des gardes de cet état est vraie, alors la transition doit être immédiatement franchie. On dira que ce genre d'état est *non stable*. Plus précisément :

> un état s de $[\![\mathcal{A}]\!]$ est dit *stable* si pour tout $A \in A(O)$, $s \not\xrightarrow{A}$.

Dans la figure 17, les états $(\ell_0, (1,1), 0, 0)$ and $(\ell_1, (1,1), 0, 0)$ ne sont pas stables alors que $(\ell_1, (1,0), 0, 0)$ est stable. Une *exécution stable* est une exécution qui termine sur un état stable. On dira qu'un VDTA est stable si ne contient pas de boucle d'états instables. Par la suite, on ne considérera que des VDTA stables.

B. Relation de conformité

Dans cette partie, nous allons définir la relation de conformité entre deux VDTA. Celle-ci est assez similaire à la relation **tioco** de [12]. L'idée principale est de considérer que tous les comportements observables de l'implémentation

doivent être autorisés par la spécification, ce qui signifie : (1) l'implémentation n'est pas autorisée à changer la valeur d'une variable de sortie à un instant non autorisé, (2) l'implémentation ne peut omettre un changement de valeur d'une variable de sortie s'il est requis par la spécification. Comme il est possible que les valeurs de sortie changent plusieurs fois en temps nul, on ne considérera l'observation des sorties que sur les états stables.

Etant donné un état stable s, nous allons donc nous intéresser aux états stables que l'implémentation peut atteindre (un singleton si le VDTA est déterministe) à partir de s après l'exécution d'une entrée $A_i \in A(I)$. Soit deux états stables $s, s' \in S$, $A_i \in A(I)$ et $\delta \in \mathbb{R}^+$ on écrit :

- $s \overset{A_i}{\Longrightarrow} s'$ s'il existe une séquence $\sigma \in A(O)^*$ telle que $s \xrightarrow{A_i} s'' \xrightarrow{\sigma} s'$, i.e. s' est un état stable qui peut être atteint depuis s après mise à jour des variables d'entrée avec A_i en ne passant que des transitions en temps zéro.
- $s \overset{\delta}{\Longrightarrow} s'$ s'il existe une séquence non observable $\sigma \in (\{Id_O\} \cup \mathbb{R}^+)^*$ telle que $s \xrightarrow{\sigma} s'$ et $\delta = \sum_{\delta_i \in Proj_{\mathbb{R}}(\sigma)} \delta_i$ (δ est la somme des délais de σ), i.e. s' est un état stable qui peut être atteint en laissant δ unités de temps s'écouler sans mise à jour de sortie observable.

Le *système de transition temporisé observable* $Obs(\mathcal{A}) = (S, s^0, A(I) \cup \mathbb{R}^+, \Longrightarrow)$ est généré inductivement depuis $[\![\mathcal{A}]\!]$ en partant depuis s^0 (supposé stable), et en utilisant les deux règles précédentes.

On définit aussi

- $ObsRun(\mathcal{A}) \triangleq Run(Obs(\mathcal{A}))$
- $ObsTr(\mathcal{A}) \triangleq Tr(Obs(\mathcal{A}))$.

Précisons que $ObsTr(\mathcal{A})$ est une séquence d'entrées et de délais. Finalement, on définit pour $s, s' \in S$, et une séquence temps réel d'entrées $\alpha \in (A(I) \cup \mathbb{R}^+)^*$,

- s *Safter* $\alpha = \{s' \mid s \overset{\alpha}{\Longrightarrow} s'\}$
- $\mathcal{A}$ *Safter* $\alpha = s^0$ *Safter* α.

Par exemple, sur la figure 17, $(\ell_0, (0,0), 0, 0)$ *Safter* $\{0.3.(L := 1; R := 1)\} = \{(\ell_2, (1,1), 1, 0)\}$.

Il est maintenant possible de définir la relation de conformité **tvco** (timed variable conformance relation).

Definition 13 (tvco). *Soit Imp et $\mathcal{A}$ deux VDTA compatibles.*

$$Imp \textbf{ tvco } \mathcal{A} \triangleq \forall \sigma \in ObsTr(\mathcal{A}),\ Out(Imp \textit{ Safter } \sigma) \subseteq Out(\mathcal{A} \textit{ Safter } \sigma)$$

Rappelons qu'Out renvoie les valeurs des variables de sortie. Intuitivement, la relation a pour objectif de vérifier si les valeurs des variables de sortie de l'implémentation sont correctes après application d'une séquence stabilisée d'entrées ou de délais. Le testeur peut observer les sorties de l'implémentation et peut modifier les entrées ou laisser le temps s'écouler. Remarquons qu'ici les détections d'états bloquants ne sont pas prises en compte.

C. Génération par objectif de test

Dans cette partie, nous allons donner les idées principales concernant la génération des cas de test, mais dans un souci de simplification, nous allons volontairement ignorer certains détails, notamment concernant l'analyse de co-accessibilité d'un état qui n'est pourtant pas triviale.

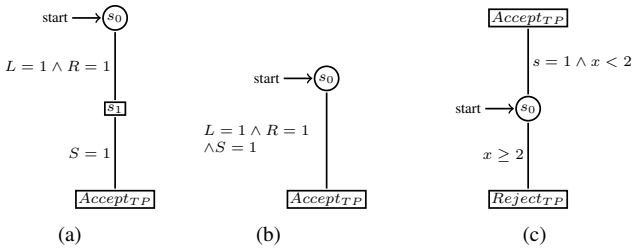


Figure 18. Objectifs de test

Nous allons décrire ici les grandes lignes de l'algorithme de génération des tests à base d'objectif de test (Test Purpose). Il existe aussi une approche non déterministe ([21]) qui est similaire à l'approche de [8] présentée ci-dessus. Soit une spécification $\mathcal{A}$, nous allons définir l'objectif de test TP. Il s'agit en fait d'un VDTA *non intrusif* pour la spécification, i.e. il n'est autorisé ni à réinitialiser des horloges de la spécification, ni à affecter de nouvelles valeurs aux variables de sortie de la spécification. En revanche, il est muni de ses propres horloges.

Definition 14. *Un* objectif de test *TP d'une spécification $\mathcal{A} = \langle L, X, I, O, l^0, G^0, \Delta_{\mathcal{A}} \rangle$ est un VDTA déterministe $TP = \langle S, X \cup X', I, O, s^0, G^0, \Delta_{TP} \rangle$ tel que :*

- *S est un ensemble fini de localités muni des localités $Accept_{TP}$ et éventuellement $Reject_{TP}$;*
- *s^0 est la localité initiale;*
- *I, O et X sont respectivement les variables d'entrées, de sorties, et d'horloges de la spécification,*
- *$G^0 \in \mathcal{G}(I, O)$ est la* condition initiale *de TP et $\mathcal{A}$;*
- *X' est l'ensemble des* horloges privées *de TP, avec $X' \cap X = \emptyset$; et*
- *$\Delta_{TP} \subseteq S \times \mathcal{G}(I, O, X, X') \times Id_O \times 2^{X'} \times S$ est la relation de transition.*

Comme pour la spécification, on suppose que les gardes de $\mathcal{G}(I, O, X, X')$ sont des combinaisons booléennes d'éléments dans $\mathcal{G}(I)$, $\mathcal{G}(O)$, $\mathcal{G}(X)$ et $\mathcal{G}(X')$. Remarquons que TP est autorisé à observer les états (les valeurs des variables) de $\mathcal{A}$. La figure 18 montre des exemples d'objectifs de test pour la commande bi-manuelle.

La génération se fait en deux étapes. Par la suite, on supposera que la spécification est déterministe.

Etape 1 : Produit entre la spécification $\mathcal{A}$ et l'objectif de test TP. Pour cela, on définit le produit synchrone :

Definition 15 (Produit synchrone)**.** *Soit $\mathcal{A} = \langle L, X, I, O, l^0, G^0, \Delta_{\mathcal{A}} \rangle$ une spécification et un objectif de test $TP = \langle S, X' \cup X, I, O, s^0, G^0, \Delta_{TP} \rangle$, le produit synchrone de $\mathcal{A}$ et TP est le VDTA $\mathcal{A} \times TP = \langle L \times S, X \cup X', I, O, (l^0, s^0), G^0, \Delta_{\mathcal{A} \times TP} \rangle$ où $\Delta_{\mathcal{A} \times TP}$ est construit par les règles suivantes ($\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$):*

$\mathcal{R}_1$: $(l, s) \xrightarrow{G \wedge G_s, A, \mathcal{X}} (l', s)$ *avec* $G_s = \bigwedge_{G' \in G_{TP}(s)} \neg G'$ *ssi il*

existe $l \xrightarrow{G,A,\mathcal{X}} l'$

$\mathcal{R}_2$: $(l,s) \xrightarrow{G_l \wedge G', Id_O, \mathcal{X}'} (l,s')$ *avec* $G_l = \bigwedge_{G' \in G_{\mathcal{A}}(l)} \neg G'$ *ssi il existe* $s \xrightarrow{G, Id_O, \mathcal{X}'} s'$

$\mathcal{R}_3$: $(l,s) \xrightarrow{G \wedge G', Id_O, \mathcal{X} \cup \mathcal{X}'} (l',s')$ *ssi il existe* $l \xrightarrow{G,A,\mathcal{X}} l'$ *et* $s \xrightarrow{G', Id_O, \mathcal{X}'} s'$

Avec ces règles, une transition de mise à jour de sorties peut donc se faire soit exclusivement sur la spécification ($\mathcal{R}_1$), soit exclusivement sur l'objectif de test ($\mathcal{R}_2$) ou bien simultanément sur les deux ($\mathcal{R}_3$) (en cas d'intersection des gardes). Il est possible de montrer que $Tr(\mathcal{A}) = Tr(\mathcal{A} \times TP)$ et $ObsTr(\mathcal{A}) = ObsTr(\mathcal{A} \times TP)$. Par ailleurs, les localités de la forme $(l, Accept_{TP})$ sont marquées comme acceptantes (Acc).

Etape 2 : Sélection de cas de test symboliques et exécution

Afin de trouver les séquences permettant d'atteindre un état Accept, il est nécessaire d'effectuer un analyse de co-accessibilité pour atteindre cet état. Pour cela, il faut calculer l'ensemble des prédécesseurs d'un état. Dans le cas des VDTA, pour un ensemble P d'états, on a

$$Pre(P) = Pre_o(P) \cup Pre_i(P) \cup Pre_t(P),$$

avec $Pre_o(P)$ les prédécesseurs avant mise à jour d'une sortie, $Pre_i(P)$ idem pour les entrées, et $Pre_t(P)$ les prédécesseurs avant l'écoulement du temps. Ensuite, il reste à calculer $Pre^*(P)$. Toutefois, il est possible que cet algorithme ne termine pas, et il est donc nécessaire de calculer les prédécesseurs sur des états symboliques. Ainsi, on construit $RG(\mathcal{A} \times TP)$ qui est une variante du graphe des régions d'Alur et Dill (voir [6]) dans lequel les localités sont construites comme des classes d'équivalence pour le temps. Cette construction permet d'obtenir un nombre fini d'états[5]. Ensuite, on marque comme états $Pass$ l'ensemble des localités dont la première composante est de type Accept. Puis, il est nécessaire de calculer $CoReach(Pass)$ sur $RG(\mathcal{A} \times TP)$ qui permet d'obtenir les contraintes adéquates pour atteindre $Pass$. Remarquons qu'il est possible d'utiliser une approche à base de *zones temporelles*, i.e. des unions de régions afin d'améliorer certains calculs sur les prédécesseurs.

L'exécution du cas de test consiste pour chaque localité à sélectionner une entrée ou un délai qui satisfasse les contraintes calculées précédemment. Une implémentation Imp passe un cas de test ssi tous les tests mènent au verdict $pass$. Il est possible de montrer que tous les cas de test qu'il est possible de générer avec cette approche sont non biaisés et exhaustifs (pour tous les cas possibles).

V. Conclusion

Dans ce document, nous avons présenté quelques approches pour le test de systèmes réactifs, et notamment dans un cadre temporisé. Nous nous sommes concentrés sur le test à base de système de transitions étiquetées (plus particulièrement les IOLTS), et leurs extensions temporisées (TAIO, VDTA) qui se fondent sur la théorie d'Alur et Dill. Nous avons notamment évoqué les notions de conformité, et les particularités des cas de test (non biais, exhaustivité). Les perspectives de recherche sont importantes dans ce domaine, notamment pour prendre en compte de façon efficace la gestion du temps et des variables affectées avec des opérations mathématiques complexes. Si des premiers travaux commencent à apparaître sur ce genre de systèmes, les outils de test en revanche sont encore assez peu développés.

[5]Cette construction étant trop volumineuse et pas le sujet essentiel de ce document, nous avons choisi de ne pas mettre d'exemple associé, voir [20] pour plus de détails.

Remerciements

Merci à Thierry Jéron pour son aide précieuse et les documents qui m'ont aidé à rédiger ce support.

References

[1] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-Based Testing of Reactive Systems*, ser. Lecture Notes in Computer Science. Springer Verlag, 2005, vol. 3472.

[2] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - a survey," *Proc. of the IEEE*, vol. 84, pp. 1090–1123, 8 1996.

[3] A. Petrenko, "Fault model-driven test derivation from finite state models: Annotated bibliography," in *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, ser. MOVEP '00. Springer-Verlag, 2000, pp. 196–205.

[4] L. D. Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon, "Lutess: A specification-driven testing environment for synchronous software," in *International Conference on Software Engineering*, 1999, pp. 267–276.

[5] P. Raymond, X. Nicollin, N. Halbwachs, and D. Waber, "Automatic testing of reactive systems, madrid, spain," in *Proceedings of the 1998 IEEE Real-Time Systems Symposium, RTSS'98*. IEEE Computer Society Press, December 1998, pp. 200–209.

[6] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[7] T. Jéron, "Génération de tests pour les systèmes réactifs et temporisés," September 2009, ecole d'Eté Temps-Réel, Télécom ParisTech, Paris.

[8] J. Tretmans, "Test generation with inputs, outputs, and repetitive quiescence," *Software–Concepts and Tools*, vol. 17, pp. 103–120, 1996.

[9] C. Jard and T. Jéron, "Tgv: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *Software Tools for Technology Transfer (STTT)*, 10 2004.

[10] T. Jéron, "Contribution à la génération automatique de tests pour les systèmes réactifs," 2004, habilitation à Diriger des Recherches - Université de Rennes 1.

[11] J. Tretmans, "Repetitive Quiescence in Implementation and Testing (Extended Abstract)," in *Formale Beschreibungstechniken für verteilte Systeme*, ser. GMD-Studien, A. Wolisz, I. Schieferdecker, and A. Rennoch, Eds., no. Nr. 315, GI/ITG-Fachgespräch. St. Augustin: GMD, 1997, pp. 23–37.

[12] M. Krichen and S. Tripakis, "Black-box conformance testing for real-time systems," in *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004*, ser. Lecture Notes in Computer Science, vol. 2989. Springer, 2004, pp. 109–126.

[13] B. Nielsen and A. Skou, "Automated Test Generation from Timed Automata," in *Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, ser. Lecture Note in Computer Science, T. Margaria and W. Yi, Eds., vol. 2031. Springer, april 2001, pp. 343–357.

[14] L. B. Briones and E. Brinskma, "A test generation framework for quiescent real-time systems," in *4th International Workshop on Formal Approaches to TEsting of Software (FATES 2004), Linz, Austria*, September 2004.

[15] M. Mikucionis, K. G. Larsen, and B. Nielsen, "T-uppaal: Online model-based testing of real-time systems," in *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*. IEEE Computer Society, 2004, pp. 396–397.

[16] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using uppaal," in *Formal Methods and Testing*, ser. Lecture Notes in Computer Science, R. Hierons, J. Bowen, and M. Harman, Eds., vol. 4949. Springer Berlin / Heidelberg, 2008, pp. 77–117.

[17] J. Schmaltz and J. Tretmans, "On conformance testing for timed systems," in *Formal Modeling and Analysis of Timed Systems (FORMATS'08)*, 2008, pp. 250–264.

[18] M. Núñez and I. Rodríguez, "Conformance testing relations for timed systems," in *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 3997. Springer, 2005, pp. 103–117.

[19] O. Nguena-Timo and A. Rollet, "Test selection for data-flow reactive systems based on observations," in *7th Workshop on Advances in Model Based Testing A-MOST 2011, Berlin, Germany*. IEEE, 03 2011, p. 8p.

[20] O. Nguena-Timo, H. Marchand, and A. Rollet, "Automatic test generation for data-flow reactive systems with time constraints," in *22nd IFIP International Conference on Testing Software and Systems (ICTSS10), (ex Testcom/Fates), Natal, Brazil (short paper)*, 11 2010, p. 6p.

[21] O. Nguena-Timo and A. Rollet, "Conformance testing of variable driven automata," in *8th IEEE International Workshop on Factory Communication Systems Communication in Automation (WFCS 2010), May 18-21, 2010, Nancy, France*, 05 2010, p. 8p.

[22] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, L. Mounier, and J. Sifakis, "If: An intermediate representation for sdl and its applications," in *World Congress on Formal Methods (1)*, 1999, pp. 307–327. [Online]. Available: citeseer.nj.nec.com/bozga99intermediate.html