# Génération de tests pour les systèmes réactifs et temporisés

Thierry Jéron
INRIA Rennes Bretagne-Atlantique, Rennes, France.
*Thierry.Jeron@irisa.fr*
*http ://www.irisa.fr/prive/jeron/*

## Abstract

*L'objectif de ce cours est d'introduire les principes du test de conformité basé sur des modèles pour les systèmes réactifs, ainsi que les techniques de génération automatique de tests correspondantes. Dans un premier temps, nous présentons une théorie du test sur des modèles simples de systèmes de transitions ainsi que les algorithmes de génération de tests pour ces modèles, basés sur une analyse de co-accessibilité. Nous évoquons ensuite l'extension de ces algorithmes à des modèles manipulant des données qui reposent alors sur une analyse approchée. Enfin, nous expliquons comment étendre la théorie du test et les algorithmes de génération au modèle des automates temporisés.*

## 1 Introduction

Le test de conformité est un type de test dit *boîte noire* où une implémentation sous test (IUT pour *implemention under test*) est comparée au comportement attendu décrit par une spécification de référence, par l'expérimentation de comportements appelés *cas de test*. Ce type de test est très courant dans la pratique industrielle quand on cherche à vérifier expérimentalement qu'un système répond bien à ses exigences. Il est maintenant reconnu que l'utilisation de modèles permet d'améliorer la qualité et le coût des tests, en systématisant voire automatisant leur génération, leur exécution et l'analyse des verdicts qu'ils produisent. Nous nous focalisons ici sur le test de systèmes réactifs, c'est à dire de systèmes qui réagissent à leur environnement, que l'on trouve dans beaucoup d'applications critiques. Dans ce type d'application, les exigences de sûreté du logiciel imposent des méthodes de vérification et de test rigoureuses. Les méthodes formelles tentent de répondre à ces exigences par la mise à disposition de modèles mathématiques précis du logiciel et de techniques algorithmiques de vérification et de test.

L'objectif de ce cours est de donner un aperçu de méthodes formelles utilisables pour le test de conformité de systèmes réactifs. Cet aperçu n'est évidemment pas exhaustif et se focalise sur certaines techniques permettant d'aborder des modèles différents, dans une certaine continuité. On trouvera dans [BJK+05] une description plus large d'un ensemble de techniques de génération de tests basées sur des modèles. Pour trois types de modèles de systèmes réactifs, nous visiterons des théories du test, des algorithmes de génération, et les propriétés des tests produits. Le premier modèle est un modèle très simple de systèmes de transitions à entrées sorties, les ioLTS pour lequel le test de conformité consiste à vérifier l'enchainement correct des actions. Pour ce modèle, nous verrons deux algorithmes de génération : un algorithme *à la volée* où les tests sont calculés pendant leur exécution, et un algorithme *hors-ligne* où les tests sont sélectionnés en fonction d'un objectif de test décrivant des comportements d'intérêt pour le test en recourrant à une analyse de co-accessibilité. Le deuxième modèle, les systèmes de transition symboliques à entrées/sorties ou ioSTS, plus expressif, permet de décrire de façon plus succincte des ioLTS infinis, par l'utilisation de variables, gardes et affectations. Le prix à payer pour la génération de tests par un objectif de test, est que l'analyse de co-accessibilité est approchée. Enfin, nous examinerons la génération de tests pour un modèle d'automates temporisés, partiellement observables, permettant de décrire des systèmes sujets à des contraintes temps-réel.

## 2 Génération de tests pour des IOLTS

Cette section présente une théorie du test de conformité pour le modèle de systèmes de transitions à entrées et sorties (ioLTS) permettant la description de systèmes non-déterministes et partiellement observables, une théorie du test basée sur la notion de relation de conformité qui définit l'ensemble des implémentations conformes à une spécification, ainsi que deux algorithmes de génération, l'un fondé sur un choix non-déterministe de comportements à tester, l'autre fondé sur une sélection par des objectifs de test qui décrivent de façon abstraite des comportements d'intérêt.

Dans la littérature sur le test de conformité fondé sur les modèles, les machines de Mealy tiennent également une place importante (voir le survey [LY96]. La théorie du test se fonde sur la notion de *modèle de faute* décrivant les fautes possibles des implémentations, la génératon de tests recherchant une couverture ehaustive du modèle de

faute par des techniques issues de la reconnaissance de machines.

### 2.1 Modèle des IOLTS

**Définition 2.1** *Un système de transitions à entrées-sorties (IOLTS) est un quintuplet* $M = (Q, \Lambda_?, \Lambda_!, \mathcal{T}, \rightarrow, Q_0)$ *où*

- $Q$ *est un ensemble dénombrable d'états,* $Q_0 \subseteq Q$ *étant l'ensemble d'états initiaux.*
- $\Lambda = \Lambda_? \cup \Lambda_! \cup \mathcal{T}$ *est l'alphabet d'actions, partitionné en entrées (notées* $?a$*), sorties (notées* $!x$*), actions internes inobservables (notées* $\tau_i$*), on notera* $\Lambda_{VIS} = \Lambda \setminus \mathcal{T} = \Lambda_? \cup \Lambda_!$ *l'ensemble des actions observables,*
- $\rightarrow \subseteq Q \times \Lambda \times Q$ *est la relation de transition,*

Dans la suite on considérera des IOLTS sans $\tau$-divergence, c'est à dire ne possédant pas de séquence infinie de transitions étiquetées par $\tau$ traversant une infinité d'états. La figure 1 représente un IOLTS fini.

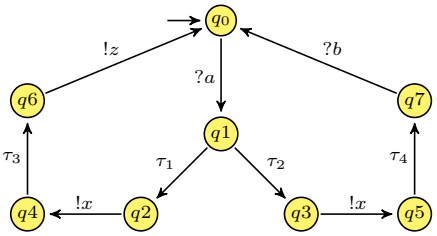


FIGURE **1. Exemple d'IOLTS**

**Notations :** Soit $\lambda_{(i)} \in \Lambda$ des actions, $a_{(i)} \in \Lambda_{\text{VIS}}$ des actions visibles, $\tau_{(i)} \in \mathcal{T}$ des actions internes, $\sigma \in {\Lambda_{\text{VIS}}}^*$ une séquence d'actions visibles, $q, q', q_{(i)} \in Q$ des états. On note $q \xrightarrow{\lambda} q'$ pour $(q, \lambda, q') \in \rightarrow$. De plus $q \xrightarrow{\lambda}$ est parfois utilisé à la place de $\exists q' : q \xrightarrow{\lambda} q'$. On note aussi $q \xrightarrow{\lambda_1 \ldots \lambda_n}_{\text{M}} q' \triangleq \exists q_0, \ldots, q_n : q = q_0 \xrightarrow{\lambda_1} q_1 \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_n} q_n = q'$.

Une *exécution* de $M$ est une suite $\rho = q_0 \xrightarrow{\lambda_0} q_1 \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{n-1}} q_n$ telle que $q_0 \in Q_0$. L'ensemble des exécutions de $M$ est noté $Runs(M) \subseteq Q_0.(\Lambda.Q)^*$. Une *séquence* $\mu$ est la projection $proj_\Lambda(\rho)$ d'une exécution sur les actions. Le langage généré par $M$ est $L(M) = proj_\Lambda(Runs(M)) \subseteq \Lambda^*$. Une *traces* est la projection $\sigma = proj_{\Lambda_{\text{VIS}}}(\rho) = a_1.a_2 \ldots a_k$ d'une exécution $\rho$ sur les actions visibles $\Lambda_{\text{VIS}}$. L'ensemble des traces de $M$ est noté $Traces(M) = proj_{\Lambda_{\text{VIS}}}(Runs(M)) \subseteq \Lambda_{\text{VIS}}^*$.

Un IOLTS $M = (Q, \Lambda, \rightarrow, Q_0)$ doté d'un ensemble d'états marqués $X \subseteq Q$ peut être considéré comme un automate observateur. On parle alors de ses exécutions acceptées $\rho = q_0 \xrightarrow{\lambda_0} q_1 \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{n-1}} q_n \in Runs(M)$ tel que $q_n \in X$. Une séquence acceptée est la projection sur $\Lambda^*$ d'une exécution acceptée et une trace acceptée, la projection sur $\Lambda_{\text{VIS}}^*$ d'une exécution acceptée. On notera $Runs_X(M) \subseteq Runs(M) = Runs_Q(M)$ l'ensemble des exécutions acceptées en $X$, $L_X(M) = proj_\Lambda(Runs_X(M)) \subseteq L(M) = L_Q(M)$ l'ensemble des séquences acceptées, et $Traces_X(M) = proj_{\Lambda_{\text{VIS}}}(Runs_X(M)) \subseteq Traces(M) = Traces_Q(M)$ l'ensemble des traces acceptées.

La sémantique de traces de $M$ induit une relation $\Rightarrow \subseteq Q \times \Lambda_{\text{VIS}}^* \times Q$

- $q \overset{\varepsilon}{\Rightarrow} q' \triangleq q = q' \vee q \xrightarrow{\tau_1.\tau_2 \cdots \tau_n} q'$ et
- pour $a \in \Lambda_{\text{VIS}}$, $q \overset{a}{\Rightarrow} q' \triangleq \exists q_1, q_2 : q \overset{\varepsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\varepsilon}{\Rightarrow} q'$ où $\varepsilon$ dénote la séquence vide de $\Lambda_{\text{VIS}}^*$.
- pour $\sigma = a_1 \cdots a_n \in \Lambda_{\text{VIS}}^*$, $q \overset{\sigma}{\Rightarrow} q' \triangleq \exists q_0, \ldots, q_n : q = q_0 \overset{a_1}{\Rightarrow} q_1 \cdots \overset{a_n}{\Rightarrow} q_n = q'$ et $q \overset{\sigma}{\Rightarrow} \triangleq \exists\, q' : q \overset{\sigma}{\Rightarrow} q'$.

On note $q \text{ after } \sigma \triangleq \{q' \in Q \mid q \overset{\sigma}{\Rightarrow} q'\}$ et pour $P \subseteq Q$, $P \text{ after } \sigma \triangleq \bigcup_{q \in P} q \text{ after } \sigma$. $M \text{ after } \sigma \triangleq Q_0 \text{ after } \sigma$ dénote l'ensemble d'états où $M$ peut résider après avoir observé $\sigma$ depuis l'état initial.

Sur l'exemple de la figure 1, on a $q_1 \overset{\varepsilon}{\Rightarrow} q_2$ et $q_1 \overset{\varepsilon}{\Rightarrow} q_3$, $q_1 \overset{!x}{\Rightarrow} q_4$, $q_1 \overset{!x}{\Rightarrow} q_6$, $q_1 \overset{!x}{\Rightarrow} q_5$, $q_1 \overset{!x}{\Rightarrow} q_7$, $q_1 \overset{!x.!z}{\Rightarrow} q_0$. On a également $q_0 \text{ after } ?a.!x = \{q_4, q_5, q_6, q_7\}$, $q_0 \text{ after } ?a.!z = \emptyset$ et $\{q_2, q_3\} \text{ after } !x = \{q_4, q_5, q_6, q_7\}$. Enfin $Traces(S) = \{\varepsilon, ?a, ?a.!x, ?a.!x.!z, ?a.!x.?b, \ldots\}$.

### 2.2 Propriétés et opérations sur les IOLTS

Le modèle des IOLTS permet de spécifier des systèmes avec des sorties non-contrôlées, i.e. une même entrée permettant d'activer plusieurs sorties possibles. Il permet également de décrire des systèmes non-déterministes.

**Définition 2.2** *Un IOLTS* $M$ *est* déterministe *s'il n'a aucune action interne (*$\mathcal{T} = \emptyset$*) et* $\forall q, q', q" \in Q, \forall a \in \Lambda_{VIS}$, $q \xrightarrow{a} q' \wedge q \xrightarrow{a} q" \implies q' = q"$.

Pour un IOLTS fini non-déterministe $M = (Q, \Lambda_?, \Lambda_!, \mathcal{T}, \rightarrow, Q_0)$ ($Q$ est fini), il est possible de construire un IOLTS déterministe $det(M)$ de même ensemble de traces. $det(M) = (2^Q, \Lambda_?, \Lambda_!, \rightarrow_{det}, Q_0 \text{ after } \varepsilon)$ où l'ensemble d'états est $2^Q = \mathcal{P}(Q)$, l'ensemble des parties de $Q$, et la relation de transition $\xrightarrow{a}_{\text{det}}$ est définie comme suit : pour $P, P' \in 2^Q$ et $a \in \Lambda^{\text{VIS}}$, $P \xrightarrow{a}_{\text{det}} P'$ si $P' = P \text{ after } a$.

Soit un IOLTS $M = (Q, \Lambda_?, \Lambda_!, \mathcal{T}, \rightarrow, Q_0)$, un état $q \in Q$ et un sous-alphabet d'actions visibles $A \subseteq \Lambda_{\text{VIS}}$. On dira que $q$ est *fortement $A$-complet* si toute action de $A$ est tirable dans $q$ : $\forall \lambda \in A, q \xrightarrow{\lambda}$. $q$ est *faiblement $A$-complet* si toute action de $A$ est tirable après d'éventuelles actions internes : $\forall \lambda \in A, q \overset{\lambda}{\Rightarrow}$. Enfin $M$ est fortement (resp. faiblement) $A$-complet si tout état de $Q$ est fortement (resp. faiblement) $A$-complet. Ces deux notions coincident pour les IOLTS déterministes.

**Blocages :** Dans la pratique du test, un testeur interagit avec l'implémentation sous test et en observe les sorties, mais aussi les blocages, par l'utilisation de temporisateurs (*timeouts*). L'usage est d'armer un temporisateur sur l'attente d'un événement, et de faire l'hypothèse que s'il arrive à échéance, l'entrée n'est plus attendue. Le test doit donc différencier les blocages prévus

par la spécification de ceux qui ne le sont pas. On distingue trois types de blocages : les deadlocks : aucune action tirable, i.e. $q \in deadlock(M) \triangleq \Gamma(q) = \emptyset$ où $\Gamma(q) \triangleq \{a \in \Lambda \mid q \xrightarrow{a}\}$ est l'ensemble des actions tirables en $q$ ; les livelocks : boucle d'actions inobservables, i.e. $q \in livelock(M) \triangleq \exists \tau_1, \ldots, \tau_n, q \xrightarrow{\tau_1 \cdots \tau_n} q$ ; et les blocages de sortie : seules des entrées sont possibles $q \in outputlock(M) \triangleq \Gamma(q) \subseteq \Lambda_?$. Or la sémantique de traces qui sert de base au test ne permet pas de conserver ces blocages. Il est donc nécessaire de l'expliciter par l'ajout, dans chaque état de blocage, d'une transition étiquetée par une nouvelle action notée $\delta$, considérée comme une sortie car observable. Cette opération est appelée *suspension*. L'IOLTS suspendu de $M = (Q, \Lambda_?, \Lambda_!, \mathcal{T}, \rightarrow, q_0)$ est l'IOLTS $\Delta(M) = (Q, \Lambda_?, \Lambda_! \cup \{\delta\}, \mathcal{T}, \rightarrow_\delta, q_0)$ où $\rightarrow_\delta = \rightarrow \cup \{(q, \delta, q) \mid q \in quiescent(M)\}$ avec $quiescent(M) = deadlock(M) \cup livelock(M) \cup outputlock(M)$.

Pour une spécification $M$, le comportement observable de $M$ du point de vue du test, et qui servira de base à la définition de la conformité est alors $STraces(M) \triangleq Traces(\Delta(M)) = Traces(det(\Delta(M)))$.

### 2.3 Relation de conformité

La *relation de conformité* définit formellement l'ensemble des implémentations conformes à une spécification. Le choix d'une telle relation dépend de la puissance des tests, leur pouvoir d'observation et de discernement entre implémentations. La relation **ioco** définie par Tretmans est particulièrement bien adaptée pour les IOLTS. Intuitivement, une implémentation $I$ est conforme à sa spécification $S$ si toutes les sorties et blocages de $I$ après une trace suspendue de $S$ sont bien spécifiés dans $S$

**Définition 2.3** *Soient $S = (Q^s, \Lambda_?, \Lambda_!, \mathcal{T}^s, \rightarrow_s, Q^s_0)$ une spécification et $I = (Q^I, \Lambda_?, \Lambda_!, \mathcal{T}^I, \rightarrow_I, Q^I_0)$ une implémentation de même interface.*

$$I \textbf{ ioco } S \triangleq \forall \sigma \in STraces(S), out(\Delta(I) \textit{ after } \sigma) \subseteq out(\Delta(S) \textit{ after } \sigma)$$

*où $out(P) \triangleq \Gamma(P) \cap \Lambda^\delta_!$ est l'ensemble des sorties/blocages dans $P$.*

**Exemple :** la figure 2 illustre la relation **ioco** pour la spécification $S$ et quatre implémentations. On a $I_1$ **ioco** $S$, $I_1$ restreignant ses sorties en $s_1$ par rapport à $S$ et $I_2$ **ioco** $S$ car une entrée supplémentaire (ici $?b$ en $s_0$) est autorisée par **ioco** qui ne restreint que les sorties. Par contre $\neg(I_3$ **ioco** $S)$ car la sortie $!z$ n'est pas spécifiée. De même $\neg(I_4$ **ioco** $S)$ car le blocage $\delta$ n'est pas permis dans $S$ après la trace $?a$.

Une caractérisation alternative de **ioco** est la suivante :

$$I \textbf{ ioco } S \iff STracesI \cap [STracesS \cdot \Lambda^\delta_! \setminus STracesS] = \emptyset$$

L'ensemble $STracesS \cdot \Lambda^\delta_! \setminus STracesS$ est exactement l'ensemble de traces minimales non-conformes : les traces

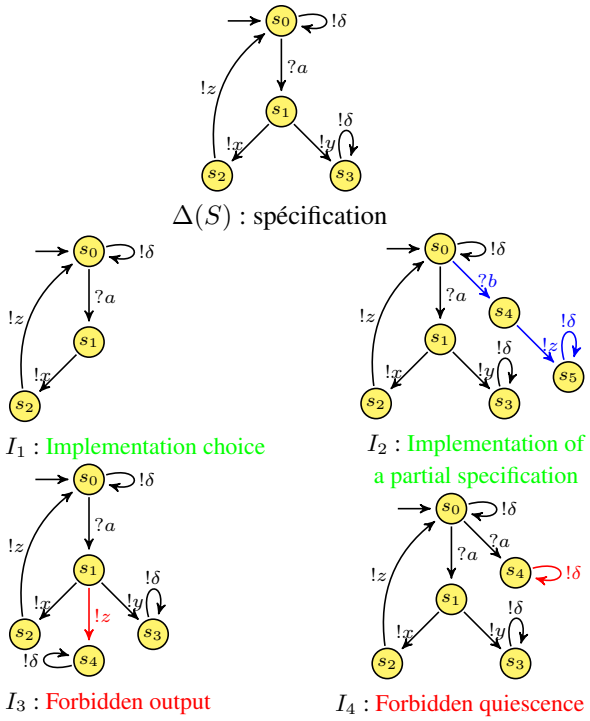


FIGURE **2. Illustration de ioco**

suspendues de $S$ prolongées par une sortie non-spécifiée. Cette caractérisation met également en exergue le fait que la conformité à $S$ est une propriété de sûreté de $I$. Cette propriété se décrit naturellement par un observateur appelé *testeur canonique* défini comme suit :

Le *testeur canonique* de $S$ est l'IOLTS $Can(S) = (Q^c, \Lambda^c_? = \Lambda^\delta_!, \Lambda^c_! = \Lambda_?, \rightarrow_c, q^c_0,)$ muni de l'ensemble d'états $\{\textbf{Fail}\} \subseteq Q^c$, et construit depuis $det(\Delta(S)) = (Q^d, \Lambda_?, \Lambda^\delta_!, \rightarrow^d, q^d_0)$ comme suit :

- $Q^c \triangleq Q^d \cup \{\textbf{Fail}\}$, où **Fail** $\notin Q^d$ est un nouvel état, et $q^c_0 \triangleq q^d_0$
- $\Lambda^c_? = \Lambda^\delta_!$ et $\Lambda^c_! = \Lambda_?$ i.e. les sorties du testeur canonique sont les entrées de $S$ et inversement,
- $\rightarrow_c \triangleq \rightarrow_d \cup \{q \xrightarrow{a}_c \textbf{Fail} \mid q \in Q_d, a \in \Lambda^\delta_! = \Lambda^c_? \wedge \neg(q \xrightarrow{a}_d)\}$, i.e. pour toute sortie non spécifiée, une transition portant l'entrée correspondante mène à **Fail** (qu'on notera de façon générique $q \xrightarrow{?othw}_c$ **Fail**).

L'ensemble des traces reconnues par $Can(S)$ dans **Fail** est
$Traces_{Fail}(Can(S)) = STraces(S).\Lambda^\delta_! \setminus STraces(S)$.
Par conséquent on a :
$I$ **ioco** $S \iff STraces(I) \cap Traces_{\text{Fail}}(Can(S)) = \emptyset$.
Le testeur canonique peut être vu comme le testeur le plus général permettant de détecter la non-conformité vis à vis de $S$, pour la relation **ioco**. Il servira donc naturellement de base pour la construction des tests.

**Example :** La figure 3 détaille l'ajout des blocages dans $\Delta(S)$, puis la déterminisation $det(\Delta(S))$, et enfin le calcul du testeur canonique $Can(S)$.

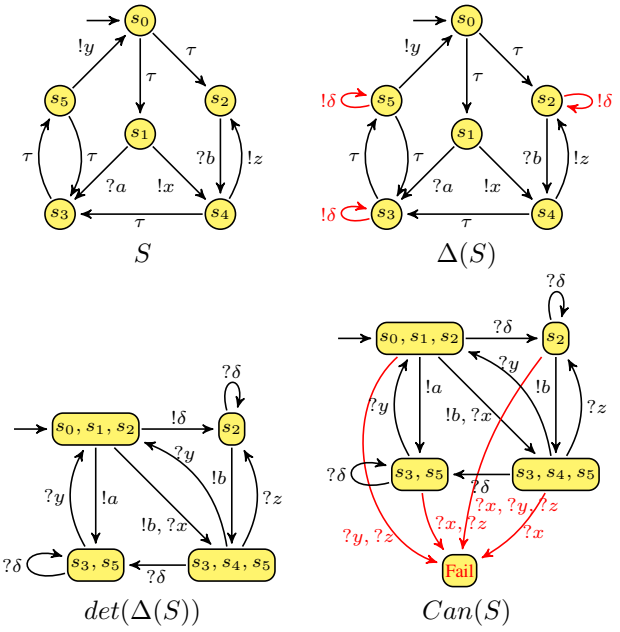


FIGURE **3. Construction du testeur canonique**

### 2.4 Les cas de tests et leur exécution

Dans ce papier, nous considérons des cas de tests très généraux, sous la forme d'IOLTS. Plus précisément un *cas de test* pour $S$ est un IOLTS déterministe $TC = (Q^{TC}, \Lambda^{TC}_?, \Lambda^{TC}_!, \rightarrow_{TC}, q^{TC}_0)$ tel que :

- $\Lambda^{TC}_! = \Lambda_?$ et $\Lambda^{TC}_? = \Lambda^{\delta}_! = \Lambda_! \cup \{\delta\}$ (inversion des entrées/sorties)
- $TC$ est muni d'un ensemble d'états puits ($q$ tq $\forall a \in \Lambda^{TC}, \neg(q \xrightarrow{a})$) representant les *verdicts*. En général **Verdicts** = **Pass** $\cup$ **Fail** $\cup$ **Inconc** $\subseteq Q^{TC}$
- les états de $TC$, excepté les **Verdicts**, sont $\Lambda^{TC}_?$-*complets* i.e. $TC$ est toujours prêt à recevoir une entrée de $\Lambda^{TC}_? = \Lambda^{\delta}_!$.

L'exécution d'un cas de test sur une implémentation $I$ se modélise par la composition parallèle de $TC$ et $\Delta(I)$ où les actions communes de $\Lambda^{\delta}_{VIS}$ se synchronisent. Plus précisément $TC\|\Delta(I) = (Q^{TC} \times Q^{I}, \Lambda^{I}, \rightarrow_{TC\|\Delta(I)}, (q^{TC}_0, q^{I}_0))$ où $\rightarrow_{TC\|\Delta(I)}$ est définie par les règles :

$$\frac{tc \xrightarrow{a}_{TC} tc' \quad q \xrightarrow{a}_{\Delta(I)} q' \quad a \in \Lambda^{\delta}_{VIS}}{(tc, q) \xrightarrow{a}_{TC\|\Delta(I)} (tc', q')}$$

$$\frac{q \xrightarrow{\tau}_{\Delta(I)} q' \quad \tau \in \mathcal{T}^{I}}{(tc, q) \xrightarrow{\tau}_{TC\|\Delta(I)} (tc, q')}$$

On a alors $Traces(TC\|\Delta(I)) = Traces(TC) \cap STraces(I)$. Comme $I$ est faiblement $\Lambda_?$-complet et les états de $TC$ (excepté ceux de **Verdicts**) sont $\Lambda^{\delta}_!$-complet, on peut montrer que $TC\|\Delta(I)$ n'est jamais bloqué, sauf dans les états de **Verdicts** de $TC$.

On dit qu'un cas de test $TC$ *peut rejeter* l'implémentation $I$ et on notera $TC$ $fails$ $I$ si une exécution de $TC\|\Delta(I)$ mène au verdict **Fail**. En d'autres termes, d'après la propriété surles traces de $TC\|\Delta(I)$ :
$TC$ fails $I \triangleq STraces(I) \cap Traces_{\text{Fail}}(TC) \neq \emptyset$.

Les implémentations étant non-contrôlables, $TC$ $fails$ $I$ ne signifie pas que l'exécution de $TC$ sur $I$ produit nécessairement le verdict **Fail**, d'autres exécutions du même cas de test sur la même implémentation peuvent produire **Pass** ou **Inconc**, comme dans l'exemple de la figure 4.

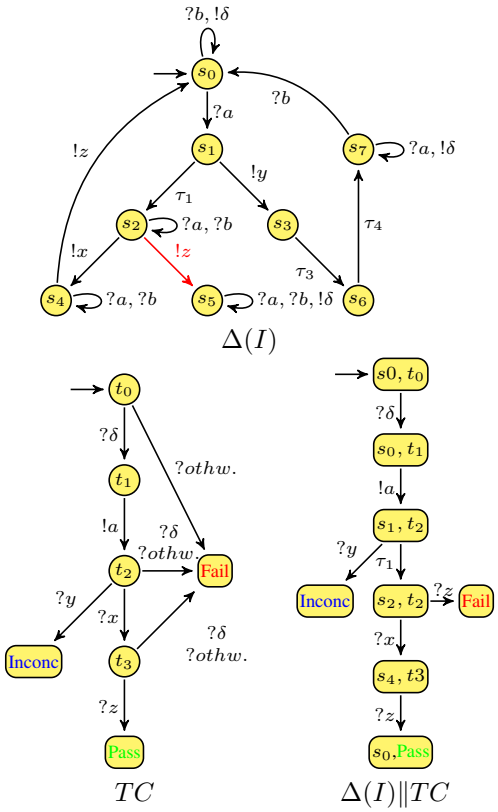


FIGURE **4. Execution de test**

**Propriétés des cas de tests :** Les cas de test doivent satisfaire des propriétés de cohérence entre les verdicts qu'ils produisent lors de l'exécution sur une implémentation et la relation **ioco** : le rejet par un test doit impliquer la non-conformité. Inversement, on souhaiterait que le non-rejet par un ensemble de tests implique la conformité. Ces notions sont formalisées par les notions de *correction* et d'*exhaustivité*.

**Définition 2.4** *$TC$ est* correct *pour $S$ et* **ioco** *s'il ne peut rejeter que des implémentations non-conformes :*
$TC$ *sound*$/S \triangleq \forall\, I, [TC$ *fails* $I \Rightarrow \neg(I$ **ioco** $S)]$.
*Une suite de tests (ensemble de cas de tests) $TS$ est correcte si tous ses cas de tests sont corrects.*
*Une suite de tests $TS$ est* exhaustive *pour $S$ et* **ioco** *si pour toute implémentation non-conforme, il existe un cas de test $TC \in TS$ qui* peut la rejeter *:*
$TS$ *exhaustive* $\triangleq \forall I, [\neg(I$ **ioco** $S) \Rightarrow [\exists TC \in TS, TC$ *fails* $I]]$.

*Une suite de test est dite* complète *si elle est correcte et exhaustive.*

La correction équivaut à $\forall\ I, [I$ **ioco** $S \Rightarrow \neg(TC$ fails $I)]$. En utilisant la caractérisation de **ioco** :
$I$ **ioco** $S \iff STraces(I) \cap Traces_{\text{Fail}}(Can(S)) = \emptyset$
et la définition du rejet par un test :
$TC$ fails $I \iff STraces(I) \cap Traces_{\text{Fail}}(TC) \neq \emptyset$
on peut alors en déduire que
$TC$ sound$/S \iff Traces_{\text{Fail}}(TC) \subseteq Traces_{\text{Fail}}(Can(S))$
et pour une suite $TS$,
$TS$ sound$/S \iff \bigcup_{TC \in TS} Traces_{\text{Fail}}(TC) \subseteq Traces_{\text{Fail}}(Can(S))$.

De même l'exhaustivité équivaut à $\forall I, [[\forall TC \in TS, \neg(TC$ fails $I)] \Rightarrow I$ **ioco** $S]$)
et on peut montrer que
$TS$ exhaustive$/S \iff Traces_{\text{Fail}}(Can(S)) \subseteq \bigcup_{TC \in TS} Traces_{\text{Fail}}(TC)$.

On a alors $TS$ complète $/$**ioco**$, S \iff \bigcup_{TC \in TS} Traces_{\text{Fail}}(TC) = Traces_{\text{Fail}}(Can(S))$. Donc en particulier $TS = \{Can(S)\}$, si on considère $Can(S)$ comme un test, est une suite complète. Mais $Can(S)$ ne peut pas être considéré comme un test, il ne contrôle pas le choix sur ses sorties, et il est trop complexe, d'où la nécessité de sélectionner un suite de tests.

### 2.5 Génération de tests

Le but de la génération de tests est de générer automatiquement une suite de tests à partir de la spécification. On aimerait bien sûr que celle-ci soit complète. La correction est facile à obtenir, par contre en général l'exhaustivité ne peut pas être atteinte par une suite finie. Cependant une exhaustivité à la limite peut être atteinte : l'ensemble infini de tous les tests que peut générer un algorithme peut être exhaustif. Dans cette sous-section, nous présentons deux algorithmes de génération de tests ayant ces propriétés. Le premier est un algorithme non-déterministe dû à Tremans [Tre96]. Le second sélectionne les tests par des objectifs de tests formalisant les comportements que l'on souhaite tester.

#### 2.5.1 Génération non-déterministe

L'algorithme de génération non-déterministe peut être vu comme un dépliage du testeur canonique $Can(S)$ en un IOLTS $T$. Soit donc $Can(S) = (Q^c, \Lambda^c_?, \Lambda^\delta_!, \Lambda^c_! = \Lambda_?, \rightarrow_c, q^c_0, )$ muni de l'ensemble d'états $\{Fail\} \subseteq Q^c$. Un test est un IOLTS $T = (Q^T, \Lambda^c_?, \Lambda^c_!, \rightarrow_T, q^T_0, )$ où $Q^T \subseteq Q^c$ et $q^T_0 = q^c_0$. Les états de $Q^T$ et les transitions sont construits récursivement à partir de $q^T_0$ comme suit : soit $q \in Q^T$ un état et $Tr_?(q) = \{q \xrightarrow{x}_c q' \in \rightarrow_c | \ x \in \Lambda_?\}$ $Tr_!(q) = \{q \xrightarrow{a}_c q' \in \rightarrow_c | \ a \in \Lambda_!\}$. Si $q \notin Fail \cup Pass$, choisir de façon non-déterministe entre

- choisir une émission i.e. choisir $q \xrightarrow{a}_c q'$ dans $Tr_!(q)$, et $rel_T := \rightarrow_T \cup \{q \xrightarrow{a}_c q'\}$ et $Q_T := Q_T \cup \{q'\}$.
- garder toutes les réceptions i.e. $\rightarrow_T := \rightarrow_T \cup Tr_?(q)$ et $Q_T := Q_T \cup \{q' \mid q \xrightarrow{x}_c q' \in \Gamma_I(q)\}$.
- arrêter et émettre le verdict $Pass$ i.e. $Pass := Pass \cup \{q\}$.

Cet algorithme est implémenté à la volée dans l'outil TorX [BFd+96], c'est à dire que le test est calculé pendant son exécution. Pour ceci $Can(S)$, donc $\Delta(S)$ et $det(\Delta(S)))$ sont aussi calculés à la volée.

**Exemple :** La figure 5 décrit la génération de deux cas de tests à partir du testeur canonique de la figure 3.

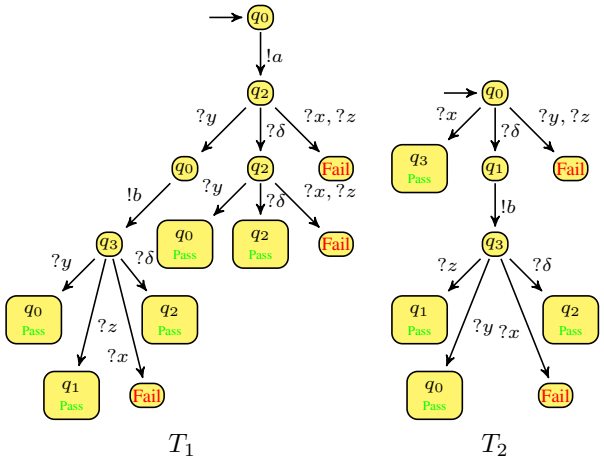


FIGURE **5. Generation non-deterministe de tests**

On peut démontrer que la suite de tests $TS$ composée de tous les tests que peut produire l'algorithme est complète. En effet, $TS$ est constituée de tous les dépliages finis et contrôlables (pas de choix entre une émission et une autre action) de $Can(S)$. On a donc $\bigcup_{TC \in TS} Traces_{\text{Fail}}(TC) = Traces_{\text{Fail}}(Can(S))$.

#### 2.5.2 Sélection par objectif de test

L'algorithme précédent utilisé à la volée est très utile pour découvrir rapidement des non-conformités. Celui que nous présentons maintenant, version simplifiée de l'implémentation de l'outil TGV [JJ04], permet de cibler des comportements partculiers, par exemple les comportements suspects, ou encore un ensemble de tests couvrant un critère structurel. Pour cela, il utilise des objectifs de tests qui spécifient des comportements particuliers à tester sous la forme d'un observateur. Le principe consiste à sélectionner les comportements de $Can(S)$ reconnus par l'objectif de test.

**Définition 2.5** *Un objectif de test est un LTS* $TP = (Q^{TP}, \Lambda^\delta_{VIS}, \rightarrow_{TP}, q^{TP}_0)$, *déterministe et complet pour* $\Lambda^\delta_{VIS}$ $(\forall q, \forall a \in \Lambda^\delta_{VIS}, q \xrightarrow{a}_{TP})$,
*muni d'un ensemble d'états puits* $Accept^{TP}$ *(puits :* $\forall a \in \Lambda^\delta_{VIS}, Accept \xrightarrow{a}_{TP} Accept$*).*

$TP$ étant déterministe et défini sur l'alphabet observable $\Lambda^{\delta}_{VIS}$, on note $Traces_{\text{Accept}}(TP) = L_{\text{Accept}}(TP)$ le langage accepté, qui est clos par extension : tout prolongement d'un mot accepté est accepté. Puisque $TP$ est complet on a $Traces(TP) = L(TP) = (\Lambda^{\delta}_{\text{VIS}})^*$. Un objectif de test $TP$ peut être vu comme un observateur d'une propriété d'accessibilité. On peut également le voir comme la négation d'une propriété de sûreté si on interpète $Accept$ comme un ensemble d'états à éviter et en interprétant également les verdicts différemment. L'objectif de test sert à sélectionner les comportements de $Can(S)$ de sorte à générer des tests qui soient à la fois :

- des observateurs de non-conformité : $Traces_{\text{Fail}}(TC) \subseteq Traces_{\text{Fail}}(Can(S))$
- des observateurs d'accessibilité : $Traces_{\text{Pass}}(TC) \subseteq Traces_{\text{Accept}}(TP)$

**Exemple :** La figure 6 décrit un objectif de test pour le testeur canonique $Can(S)$ de la figure 3. On note l'étiquetage de l'état $p_3$ par $Refuse$. Ces états sont utilisés par l'algorithme de TGV pour décrire des comportements qu'on souhaite éviter, mais nous n'en tiendrons pas compte ici. L'étiquette $*$ sur une transition signifie le complémentaire sur l'alphabet des actions portées par les autres transitions sortantes, et permet donc de rendre complet aisément un objectif.

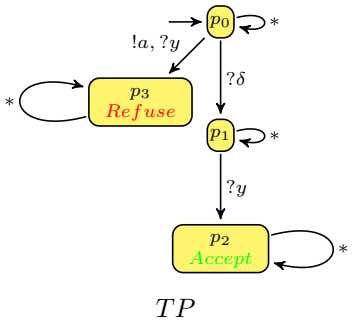


FIGURE **6. Objectif de test**

Le but de la sélection étant de sélectionner les comportements de $Can(S)$ reconnus par $TP$, donc l'intersection des traces, l'opération standard pour ce faire est le produit synchrone.

**Définition 2.6** *Soit $M_1 = (Q^1, A, \rightarrow_1, q^1_0)$ muni de $F_1$, et $M2 = (Q^2, A, \rightarrow_2, q^2_0)$ muni de $F_2$ deux (IO)LTS de même alphabet $A$.*
*Le produit synchrone de $M_1$ et $M_2$ est l'(IO)LTS $M_1 \times M_2 = (Q^1 \times Q^2, A, \rightarrow, (q^1_0, q^2_0))$ muni de $F_1 \times F_2$ où $\rightarrow$ est définie par la règle :*

$$\frac{q_1 \xrightarrow{a}_1 q'_1 \quad q_2 \xrightarrow{a}_2 q'_2}{(q_1, q_2) \xrightarrow{a} (q'_1, q'_2)}$$

On a alors $L(M_1 \times M_2) = L(M_1) \cap L(M_2)$ et $L_{F_1 \times F_2}(M_1 \times M_2) = L_{F_1}(M_1) \cap L_{F_2}(M_2)$.

Soit $Can(S) = (Q^c, \Lambda^{\delta}_{\text{VIS}}, \rightarrow_c, q^c_0, )$ muni de $Fail \subseteq Q^c$ et $TP = (Q^{\text{TP}}, \Lambda^{\delta}_{\text{VIS}}, \rightarrow_{\text{TP}}, q^{\text{TP}}_0)$ muni de $Accept_{\text{TP}} \subseteq Q^{\text{TP}}$, les deux IOLTS étant munis du même alphabet $\Lambda^{\delta}_{\text{VIS}}$.

On note $PS^{\text{VIS}} = Can(S) \times TP$ et on considère les deux ensembles d'états accepteurs suivants :

- $Accept_{\text{VIS}} = Q_c \setminus \{\textbf{Fail}\} \times Accept_{\text{TP}}$ et
- $Fail_{\text{VIS}} = \{Fail\} \times Q^{\text{TP}}$

$TP$ étant complet, on a $Traces(TP) = (\Lambda^{\delta}_{\text{VIS}})^*$, donc

- $Traces(PS^{\text{VIS}}) = Traces(Can(S))$
- $Traces_{\text{Accept}}(PS^{\text{VIS}}) = STraces(S) \cap Traces_{\text{Accept}}(TP)$
- $Traces_{\text{Fail}}(PS^{\text{VIS}}) = Traces_{\text{Fail}}(Can(S))$

**Exemple :** La figure 7 représente la partie initiale du produit synchrone $PS^{\text{VIS}} = Can(S) \times TP$ pour $Can(S)$ décrit à la figure 3 et $TP$ décrit à la figure 6.

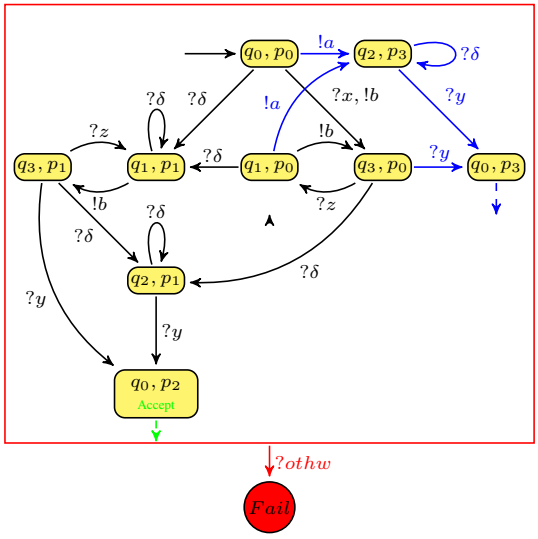


FIGURE **7. Produit synchrone** $PS^{\text{VIS}} = Can(S) \times TP$

Le calcul de $PS^{\text{VIS}}$ a permis d'étiqueter par $Accept_{\text{VIS}}$ les états de $Can(S)$. Il s'agit maintenant d'extraire de $PS^{\text{VIS}}$ un cas de test $TC$ dont les comportements acceptés sont ceux de $PS^{\text{VIS}}$, et qui ne contiennent pas, ou au moins détecte les comportements dont les prolongements ne sont pas acceptés. On a donc besoin de calculer l'ensemble des états de $PS^{\text{VIS}}$ co-accessibles de $Accept_{\text{VIS}}$, i.e. l'ensemble des états depuis lesquels $Accept_{\text{VIS}}$ est accessible. Pour un ensemble d'états $P$ d'un IOLTS $M$, l'ensemble des états co-accessibles de $P$ est défini par le plus petit point fixe : $coreach(P) = \mu X.P \cup pre(X) = \bigcup_{i \geq 0} pre^i(P)$

où $pre(P) = \{q' \in Q^{\text{M}} \mid \exists b \in \Lambda, \exists q \in P,\ q' \xrightarrow{b}_{\text{M}} q\}$.

Soit $PS^{\text{VIS}}(Q^{\text{VIS}}, \Lambda^{\delta}_{\text{VIS}}, \rightarrow_{\text{VIS}}, q^{\text{VIS}}_0)$, muni de $Accept_{\text{VIS}}$ et $Fail_{\text{VIS}}$. Le *graphe de test complet* est l'IOLTS $CTG = (Q^{\text{VIS}}, \Lambda^{\delta}_{\text{VIS}}, \rightarrow_{\text{CTG}}, q^{\text{VIS}}_0)$ muni des ensembles d'états de verdict **Pass** $\triangleq Accept_{\text{VIS}}$, **Inconc** $\subseteq Q^{\text{VIS}}$ et **Fail** $=$ **Fail**$_{\text{VIS}}$,
où **Inconc** et $\rightarrow_{\text{CTG}}$ sont définis par les règles :

$$\frac{q \in coreach(Accept_{\text{VIS}}) \qquad q \xrightarrow{\alpha}_{\text{VIS}} q' \qquad q' \in coreach(Accept_{\text{VIS}}) \cup \{\textbf{Fail}_{\text{VIS}}\} \qquad \alpha \in \Lambda^{\delta}_{\text{VIS}}}{q \xrightarrow{\alpha}_{\text{CTG}} q'}$$

$$\frac{q \in coreach(Accept_{\text{VIS}}) \qquad q \xrightarrow{\alpha}_{\text{VIS}} q' \qquad q' \notin (coreach(Accept_{\text{VIS}}) \cup \{\textbf{Fail}_{\text{VIS}}\} \qquad \alpha \in \Lambda_{\text{VIS}?} = \Lambda^{\delta}_{!}}{q \xrightarrow{\alpha}_{\text{CTG}} q' \qquad q' \in \textbf{Inconc}}$$

La première règle conserve les transitions dont l'origine est co-accessible d'$Accept^{\text{VIS}}$ et la destination est soit co-accessible, soit $Fail$. La seconde règle concerne les transitions dont la destination n'est pas co-accessible, et conserve uniquement les entrées du testeur (les sorties du système). En effet, les sorties étant contrôlables, elles peuvent être coupées, alors que sur une entrée non-contrôlable, on détecte qu'on est sorti des états co-accessibles en émettant le verdict **Inconc**.

**Exemple :** la figure 8 représente le résultat du calcul du graphe de test complet à partir du produit synchrone $PS^{\text{VIS}}$ représenté figure 7.

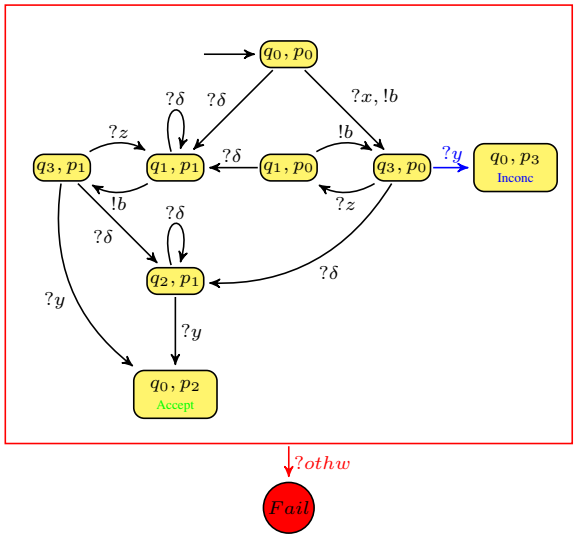


FIGURE **8. Graphe de test complet** $CTG$

On notera que les états de $CTG$ peuvent posséder à la fois des entrées et des sorties. Or dans un test, on souhaite en général distinguer les états d'entrée et de sortie, et sur les états de sortie, faire un choix. Une opération d'élagage peut permettre de supprimer ces choix et d'extraire des cas de tests "contrôlables", i.e. ayant soit une sortie, soit des entrées.

**Exemple :** deux cas de tests obtenus par élagage sont représentés Figure 9.

On peut démontrer que la suite de test infinie que l'algorithme peut générer est correcte et exhaustive.

La preuve de correction repose sur le fait que les tests sont obtenus depuis $Can(S)$ par produit et sélection, d'où la propriété $Traces_{\text{Fail}}(TC) \subseteq Traces_{Fail}(Can(S))$.

Pour l'exhaustivité, si $\neg(I$ **ioco** $S)$ alors par définition $\exists \sigma \in STraces(S), \exists x \in \Lambda^{\delta}_{!}, x \in Out(\Delta(I) \text{ after } \sigma) \wedge$

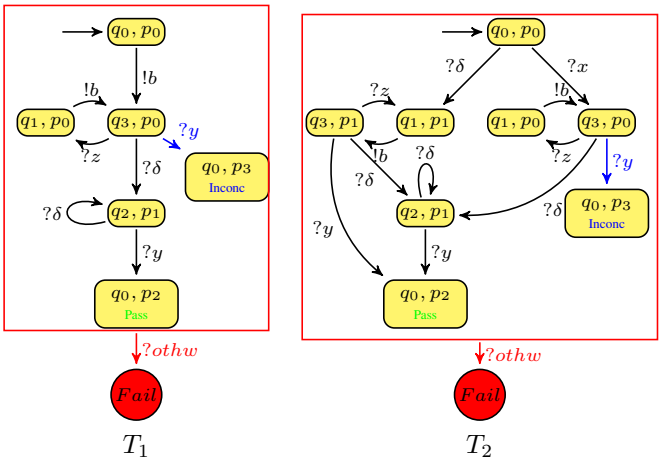


FIGURE **9. Cas de test produits**

$x \notin Out(\Delta(S) \text{ after } \sigma)$. On a $Out(\Delta(S) \text{ after } \sigma) \neq \emptyset$, puisque si ce n'était pas le cas, on aurait ajouté une action $\delta$. Soit donc $y$ tel que $y \in Out(\Delta(S) \text{ after } \sigma)$, et $\sigma' = \sigma.y(\in STraces(S))$. On construit un objectif de test $TP$ de langage accepté $\sigma'$. Par définition, $CTG$ after $\sigma.x \in$ **Fail**. Il suffit alors d'élaguer $CTG$ en un cas de test $TC$ tq $\sigma.x \in Traces(TC)$ d'où $TC$ after $\sigma.x \in$ **Fail**. Donc $TC$ fails $IUT$.

## 3 Génération de tests pour des IOSTS

Dans la section2, nous avons présenté des techniques de génération de tests pour des modèles d'IOLTS finis. Ces modèles peuvent être produits par des spécifications de plus haut niveau, dans des langages tels que SDL, Lotos, UML, etc. Or ces langages peuvent produire des modèles infinis. Des techniques "à la volée" comme TorX, ou paresseuses comme TGV, peuvent quand même générer des tests pour des modèles infinis, pour peu que le degré de branchement des IOLTS soit fini, et pas trop grand. Dans cette section, nous allons examiner une technique qui permet de générer des tests pour le modèle des IOSTS (pour *Input Output Symbolic Transition Systems*) qui sont des automates étendus avec variables, gardes et affectations et dont les actions d'entrées et sorties portent des paramètres. Leur sémantique sont des IOLTS infinis, à branchement infini [JJRZ05].

La relation de conformité portant sur la sémantique IOLTS, on conserver la même relation **ioco**. Les tests produits seront aussi des IOSTS qui peuvent être vus comme des programmes de tests, dont les paramètres d'actions ne sont pas instanciés. C'est en effet lors de l'exécution que les paramètres de sorties (émises vers le système) seront instanciés par la résolution de contraintes sur les gardes et que les gardes portant sur les paramètres d'entrées (reçues du système) seront vérifiées. La génération s'effectue par sélection par un objectif de test (aussi un IOSTS) et suit le même principe général que dans le cas des IOLTS, c'est à dire fondé sur le calcul des états co-accessibles

d'états accepteurs. Cependant, la co-accessibilité dans ces modèles est indécidable, ce qui amène à calculer une sur-approximation de ces états co-accessibles, de façon symbolique, par des techniques d'interprétation abstraite.

Il existe d'autres travaux sur la génération de tests pour des modèles similaires, comme [CIvdPS05] qui repose sur l'abstraction par des ioLTS, la génération de tests sur ces ioLTS, et leur concrétisation dans les ioSTS, [LG02] qui utilise des techniques de résolution de contraintes, ou [FTW04] qui utlise la génération à la volée.

### 3.1 Le modèle des IOSTS et sa sémantique IOLTS

**Définition 3.1** *Un IOSTS est un quadruplet* $\mathcal{M} = (V, \Theta, \Sigma, T)$ *où :*

- $V = V_i \cup V_x$ *est l'ensemble de* variables*, partitionné en variables internes* $V_i$ *et externes* $V_x$*,*
- $\Theta \subseteq \mathcal{D}_{V_i}$ *: est la* condition initiale *qui a une solution unique dans* $\mathcal{D}_{V_i}$*.*
- $\Sigma = \Sigma_? \cup \Sigma_!$ *: est un alphabet fini d'*actions *portant des* paramètres de communication *de type* $\mathrm{sig}(a)$*.*
- $T$ *est un ensemble fini de* transitions symboliques $t = (a, \vec{p}, G, A)$ *où*
  - $a \in \Sigma$ *est l'action,*
  - $\vec{p}$ *est un tuple de* paramètres de communication*, de portée locale à* $t$*,*
  - $G \subseteq \mathcal{D}_V \times \mathcal{D}_{\mathrm{sig}(a)}$ *: est la* garde*, exprimée dans une théorie dans laquelle la satisfiabilité est décidable,*
  - $A : \mathcal{D}_V \times \mathcal{D}_{\mathrm{sig}(a)} \to \mathcal{D}_{V_i}$ *est une affectation.*

La notion de variable interne/externe permet d'utiliser le même modèle pour décrire les spécifications et les objectifs de tests. Les variables internes sont les variables ordinaires, les variables externes sont des variables modifiées par un autre modèle, utilisées par les objectifs de tests pour parler des variables de la spécification. Le modèle n'utilise pas de localité explicite, cependant les localités seront modélisées par une variable $pc$ de type énuméré. Par soucis de simplicité, nous ne considérons pas ici d'action interne. Celles-ci peuvent être prises en compte tant qu'il n'existe pas de boucle d'action interne.

**Exemple :** pour illustrer cette section, nous utiliserons un exemple simple de controleur d'un ascenseur décrit par la figure 10. Dans cet exemple, $h$ est un paramètre entier spécifiant la hauteur de l'immeuble, $c, g$ et $pc$ sont des variables entières, $c$ étant l'étage courant, $g$ l'étage à atteindre, et $pc$ la localité. prenant ses valeurs dans $\{Wait, Move, End\}$. L'ascenseur possède une entrée $Target?$ et les sorties $Up!$, $Down!$, $Stop!$, $Break!$ ; $p$ est le paramètre de communication de $Target?$, $Up!$, $Down$ et $Stop$. Initialement, $c = 0$ et $g = 0$ et la localité est $pc = Wait$. Sur réception de $Target(p)$, avec $0 \leq p \leq h$, la valeur de $p$ est affectée à $g$, l'étage à atteindre, et la nouvelle localité est $Move$. Dans $Move$, soit on est arrivé à l'étage demandé ($c = g$) et on émet le message $Stop!(p)$ avec la valeur $p = c$, soit $c < g$, on émet

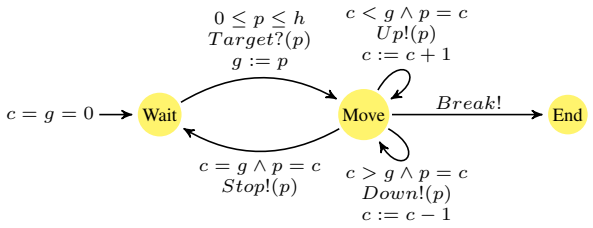


FIGURE **10. IOSTS d'un controleur d'ascenseur**

$Up!(p)$ avec $p = c$ et l'ascenseur monte en incrémentant $c$, soit $c > g$, on émet $Down!(p)$ avec $p = c$ et l'ascenseur descend en décrémentant $c$.

La sémantique d'un ioSTS $\mathcal{M} = (V, \Theta, \Sigma, T)$ est un ioLTS $[\![\mathcal{M}]\!] = (Q, Q_0, \Lambda, \to)$ où :

- $Q = \mathcal{D}_V$ est un ensemble (infini) d'*états* ;
- $Q^0 = \{\vec{\nu} = \langle \vec{\nu}_i, \vec{\nu}_x \rangle \mid \vec{\nu}_i \in \Theta \wedge \vec{\nu}_x \in \mathcal{D}_{V_x}\}$ : est l'ensemble d'*états initiaux* ;
- $\Lambda = \{\langle a, \vec{\pi} \rangle \mid a \in \Sigma \wedge \vec{\pi} \in \mathcal{D}_{\mathrm{sig}(a)}\}$ : est l'ensemble d'*actions valuées* partitioné en entrées et sorties $\Lambda = \Lambda_? \cup \Lambda_!$ ;
- $\to$ est la *relation de transition* définie par la règle :

$$\frac{(a, \vec{p}, G, A) \in T \quad \vec{\nu} = \langle \vec{\nu}_i, \vec{\nu}_x \rangle \in \mathcal{D}_V \quad \vec{\pi} \in \mathcal{D}_{\mathrm{sig}(a)} \quad \vec{\nu}' = \langle \vec{\nu}'_i, \vec{\nu}'_x \rangle \in \mathcal{D}_V \quad G(\vec{\nu}, \vec{\pi}) \quad \vec{\nu}'_i = A(\vec{\nu}, \vec{\pi})}{\vec{\nu} \xrightarrow{\langle a, \vec{\pi} \rangle} \vec{\nu}'}$$

Cette sémantique IOLTS des IOSTS permet de réutiliser les notions introduites pour les IOLTS, comme les exécutions, les séquences, les traces, etc. On notera en particulier $STraces(\mathcal{M})$ pour $STraces([\![\mathcal{M}]\!])$.

Pour l'exemple de l'ascenseur, $\langle pc = Wait, g = 0, c = 0 \rangle \xrightarrow{Target?(3)} \langle Move, 3, 0 \rangle \xrightarrow{Up!(0)} \cdots \langle Move, 3, 1 \rangle \xrightarrow{Up!(1)} \langle Move, 3, 2 \rangle \xrightarrow{Up!(2)} \langle Move, 3, 0 \rangle \xrightarrow{Stop!(3)} \langle Wait, 3, 0 \rangle$ est une exécution, $Target?(3).Up!(0).Up!(1).Up!(2).Stop!(3)$ est une trace.

Nous nous restreindrons dans cet exposé à des IOSTS déterministes où $\mathcal{M} = (V, \Theta, \Sigma, T)$ est *déterministe* si pour toute action $a \in \Sigma$, et toute paire de transitions $t_1 = (a, \vec{p}, G_1, A_1)$ et $t_2 = (a, \vec{p}, G_2, A_2)$ portant la même action, la conjonction des gardes $G_1 \wedge G_2$ est insatisfiable. Les IOSTS déterministes forment une sous-classe stricte des IOSTS, et il n'est donc pas toujours possible de déterminiser un IOSTS. Cependant, une heuristique permet de déterminiser la sous-classe des IOSTS à *lookhead* borné [JMR06].

Comme dans le cas des IOLTS, on considère comme observable les entrées, sorties et blocages. Une opération de suspension syntaxique permet de transformer un IOSTS en un nouvel IOSTS où les blocages sont explicités, les seuls blocages étant ici les deadlocks et les blocages de sorties (pas de livelock car pas d'action interne) : la *suspension* de $\mathcal{M} = (V, \Theta, \Sigma, T)$ est L'IOSTS

$\Delta(\mathcal{M}) = (V, \Theta, \Sigma^\delta, T_\delta)$ où :
- $\Sigma^\delta = \Sigma^\delta_! \cup \Sigma_?$ avec $\Sigma^\delta_! = \Sigma_! \cup \{\delta\}$,
- $T_\delta = T \cup \{\langle \delta, G_\delta, Id_V \rangle\}$ avec

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

Le comportement observable considéré pour le test sera $STraces(\mathcal{M}) \triangleq Traces(\Delta(\mathcal{M}))$.

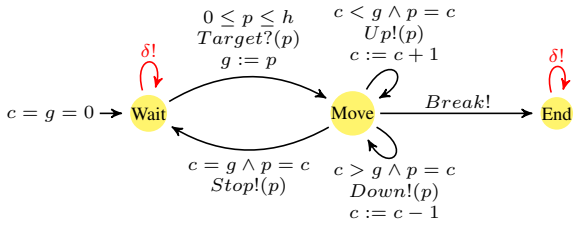


FIGURE **11. IOSTS suspendu de l'ascenseur**

### 3.2 Relation de conformité pour les IOSTS

Nous supposons que la spécification est un ioSTS déterministe $\mathcal{S} = (V^\mathcal{S}, \Theta^\mathcal{S}, \Sigma, T^\mathcal{S})$, avec $\Sigma = \Sigma_! \cup \Sigma_?$ et $V^\mathcal{S}_x = \emptyset$ (toutes les variables sont internes). Sa sémantique est un IOLTS $[\![\mathcal{S}]\!] = S = (Q, Q^0, \Lambda, \rightarrow)$ avec $\Lambda = \Lambda_! \cup \Lambda_?$.

On suppose que l'implémentation est un ioLTS inconnu et $\Lambda_?$-complet $I = (Q_I, Q^0_I, \Lambda_! \cup \Lambda_?, \rightarrow_I)$.

Un cas de tests qu'on souhaite générer est un ioSTS $\mathcal{TC} = (V^{TC}, \Theta^{TC}, \Sigma^{TC}, T^{TC})$, avec $\Sigma^{TC}_? = \Sigma_!$, $\Sigma^{TC}_! = \Sigma_?$ muni d'une variable Verdict avec $\mathcal{D}_{verdict} = \{\mathtt{none}, \mathtt{fail}, \mathtt{pass}, \mathtt{inconc}\}$. Il sera déterministe, $\Sigma^{TC}_?$-complet dans tout état où Verdict $= \mathtt{none}$. Sa sémantique est un IOLTS $[\![\mathcal{TC}]\!] = TC = (Q^{TC}, q^{TC}_0, \Lambda^{TC}, \rightarrow_{TC})$ muni des ensembles d'états de verdict *Fail* $=$ (Verdict $=$ `fail`), *Pass* $=$ (Verdict $=$ `pass`), *Inconc* $=$ (Verdict $=$ `inconc`).

La relation de conformité choisie est identique à **ioco** :

$I$ **ioco** $\mathcal{S} \triangleq \forall \sigma \in Straces(S)$,
$out(\Delta(I) \text{ after } \sigma) \subseteq out(\Delta(S) \text{ after } \sigma)$

qui se caractérise aussi par

$I$ **ioco** $\mathcal{S} \iff STraces(I) \cap [STraces(\mathcal{S}) \cdot \Lambda^\delta_! \setminus STraces(\mathcal{S})] = \emptyset$.

Cette caractérisation mène à la construction d'un testeur canonique reconnaissant $STraces(\mathcal{S}) \cdot \Lambda^\delta_! \setminus STraces(\mathcal{S})$, décrit par un ioSTS construit par transformation syntaxique de $Delta(\mathcal{S})$.

Le *testeur canonique* de $\mathcal{S} = (V^\mathcal{S}, \Theta^\mathcal{S}, \Sigma, T^\mathcal{S})$ est l'ioSTS $Can(\mathcal{S}) = (V^{Can}, \Theta^{Can}, \Sigma^{Can}, T^{Can})$ tel que :
- $V^{Can} = V^\mathcal{S} \cup \{\mathsf{Verdict}\}$ où $\mathcal{D}_{\mathsf{Verdict}} = \{\mathtt{none}, \mathtt{fail}\}$
- $\Theta^{Can} = \Theta^\mathcal{S} \wedge \mathsf{Verdict} = \mathtt{none}$ ;
- $\Sigma^{Can}_? = \Sigma^\delta_!$ and $\Sigma^{Can}_! = \Sigma_?$ (les alphabets sont inversés par rapport à $\Delta(\mathcal{S})$)
- $T^{Can} = T^{\Delta(\mathcal{S})} \cup T^{new}$ où $T^{new}$ est défini par la règle :

$$\frac{a \in \Sigma^\delta_! = \Sigma^{Can}_? \quad G_a = \bigwedge_{(a,\vec{p},G,A)\in T^{\Delta(\mathcal{S})}} \neg G}{\left[\, a(\vec{p}) \ :\ G_a(\vec{\nu}, \vec{p}) \,?\, \mathsf{Verdict}' := \mathtt{fail} \,\right] \in T^{new}}$$

On a alors $Traces_{Fail}(Can(\mathcal{S})) = STraces\mathcal{S} \cdot \Lambda^\delta_! \setminus STraces\mathcal{S}$

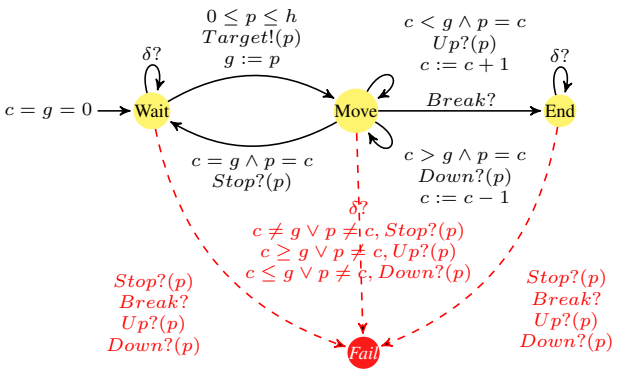


FIGURE **12. Testeur canonique de l'ascenseur**

**Exemple :** le testeur canonique de l'ascenseur est décrit par la figure 12.

L'exécution d'un test $\mathcal{TC}$ sur une implémentation $I$ se modélise par la composition parallèle de $TC = [\![\mathcal{TC}]\!]$ et $\Delta(I)$ comme dans le cas des ioLTS. En découlent les mêmes définitions d'échec d'un test sur une implémentation, de correction et d'exhasutivité, ainsi que leurs conditions nécessaires et suffisantes en termes de traces reconnues en $Fail$ :

$TS$ correcte iff
$\bigcup_{TC\in TS} Traces_{Fail}(TC) \subseteq Traces_{Fail}(Can(\mathcal{S}))$
$TS$ exhaustive iff
$\bigcup_{TC\in TS} Traces_{Fail}(TC) \supseteq Traces_{Fail}(Can(\mathcal{S}))$

### 3.3 Sélection de tests

Nous décrivons maintenant un algorithme de génération de tests qui, comme dans le cas des ioLTS, est basé sur la sélection par un objectif de test. La différence avec le cas des ioLTS est que l'objectif est décrit par un ioSTS, et que le calcul exact des états co-accessibles est impossible, ce qui amène à calculer une sur-approximation par interprétation abstraite.

**Définition 3.2** *Un objectif de test est un ioSTS déterministe* $\mathcal{TP} = (V^{TP}, \Theta^{TP}, \Sigma^\delta, T^{TP})$ *tel que :*
- $V^{TP}_x = V^S_i$ *ce qui signifie que* $\mathcal{TP}$ *peut observer les variables (internes) de* $\mathcal{S}$ *;*
- $V^{TP}_i \cap V^S_i = \emptyset$ *avec* $pc^{TP} \in V^{TP}_i$ *et* `accept` $\in \mathcal{D}_{pc^{TP}}$. Accept $\triangleq (pc^{TP} = \mathtt{accept})$.
- $\mathcal{TP}$ *est* complet *excepté dans* `accept` *:* $\forall a \in \Sigma^\delta$, $pc^{TP} \neq \mathtt{accept} \Rightarrow \bigvee_{(a,\vec{p},G,A)\in T^{TP}} G = true$.

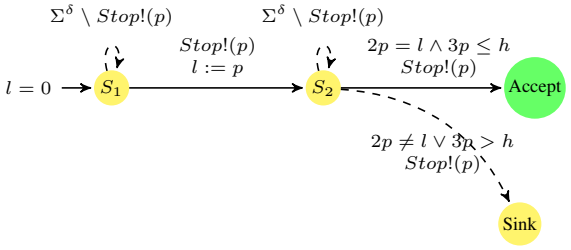


FIGURE **13. Objectif de test pour l'ascenseur**

**Exemple :** la figure 13 décrit un objectif de test pour l'ascenseur. L'idée est de sélectionner les comportements dans lesquels un premier arrêt est effectué à un certain étage $l$, puis un second arrêt à un étage $p$ qui soit la moitié de $l$ et inférieur au tiers de $h$. Cet objectif a peu d'intérêt réel, mais sert à illustrer certaines caractéristiques de l'algorithme de génération.

La première étape consiste à construire syntaxiquement le produit synchrone entre le testeur canonique $Can(\mathcal{S})$ et l'objectif $\mathcal{TP}$, ce qui permet de marquer les comportements de $can(\mathcal{S})$ acceptés par $\mathcal{TP}$.

Le produit synchrone est l'ioSTS $\mathcal{P} = Can(\mathcal{S}) \times \mathcal{TP} = (V^P, \Theta^P, \Sigma^{Can}, T^P)$ où :

- $V^P = V_i^P \cup V_x^P$, avec $V_i^P = V_i^{Can} \cup V_i^{TP}$ et $V_x^P = \emptyset$ ;
- $\Theta^P(\langle \vec{v}^{Can}, \vec{v}^{TP} \rangle) = \Theta^{Can}(\vec{v}^{Can}) \wedge \Theta^{TP}(\vec{v}^{TP})$ ;
- $T^P$ est défini par la règle suivante :

$$\frac{[\,a(\vec{p}) \,:\, G^c(\vec{v}^c, \vec{p})\,?\,(\vec{v}_i^c)' := A^c(\vec{v}^c, \vec{p})\,] \in T^{Can} \quad [\,a(\vec{p}) \,:\, G^t(\vec{v}^t, \vec{p})\,?\,(\vec{v}_i^t)' := A^t(\vec{v}^t, \vec{p})\,] \in T^{TP}}{[\,a(\vec{p}) \,:\, G^c(\vec{v}^c, \vec{p}) \wedge G^t(\vec{v}^t, \vec{p})\,? \quad (\vec{v}_i^c)' := A^c(\vec{v}^c, \vec{p}), (\vec{v}_i^t)' := A^t(\vec{v}^t, \vec{p})\,] \in T^P}$$

Finalement $\mathcal{P}'$ est l'ioSTS obtenu par ajout de Verdict := `pass` aux transitions ayant l'affectation $pc' :=$ `accept`.

**Exemple :** la figure 14 représente le produit synchrone de $Can(\mathcal{S})$ et $\mathcal{TP}$ pour l'exemple de l'ascenseur.

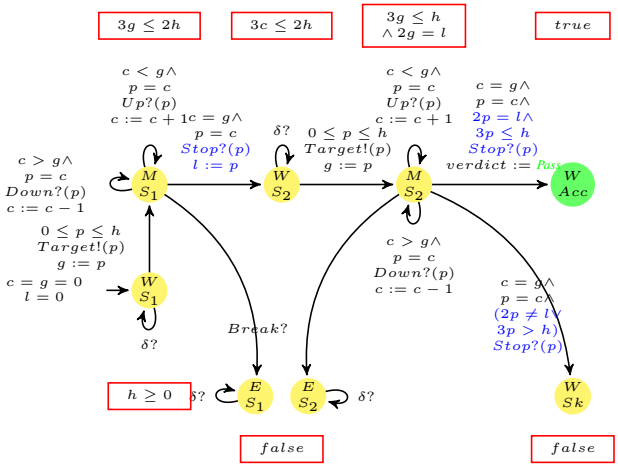


FIGURE **14. Produit synchrone** $Can(\mathcal{S}) \times \mathcal{TP}$

La génération d'un test à partir de $\mathcal{P}'$ s'effectue par calcul de l'ensemble des états co-accessibles de $Accept$. Cependant, il est en général impossible de calculer cet ensemble de façon exacte, l'accessibilité étant indécidable sur ce type de modèle. On s'appuie donc sur un calcul approché.

Soit $pre(A)(X)(\vec{v}, \vec{p}) \triangleq \exists \vec{v'} : X(\vec{v'}) \wedge \vec{v'} = A(\vec{v}, \vec{p})$ la précondition de $X$ par l'affectation $A$, i.e., l'ensemble des états depuis lesquels l'affectation $A$ mène en $X$. Soit $pre^\alpha(A)(X)(\vec{v}, \vec{p}) \supseteq pre(A)(X)(\vec{v}, \vec{p})$ une sur-apparoximation de cet ensemble.

Soit $coreach(Pass) = \text{lfp}(\lambda X.Pass \cup pre(X))$ l'ensemble des états co-accessibles de *Pass* et $coreach^\alpha$ une sur-approximation de $coreach(Pass)$.

$pre^\alpha(A)(coreach^\alpha)$ est alors une condition nécessaire pour atteindre $coreach(Pass)$ par l'affectation $A$, donc $\neg pre^\alpha(A)(coreach^\alpha)$ est une condition suffisante pour quitter $coreach(Pass)$ par l'affectation $A$. Ces deux conditions vont servir à renforcer les gardes afin de calculer un cas de test depuis $\mathcal{P}'$.

Le cas de test obtenu pour $\mathcal{S}$ et $\mathcal{TP}$ est l'ioSTS $\mathcal{TC} = (V^{P'}, \Theta^{P'}, \Sigma^{Can}, T^{\mathcal{TC}})$ où $T^{\mathcal{TC}}$ est défini à partir de $\mathcal{P}'$ par les trois règles :

$$\text{(Select output)} : \frac{(a, \vec{p}, G, A) \in T^{P'} \quad a \in \Sigma_!^{Can} \quad G' = pre^\alpha(A)(coreach^\alpha)}{(a, \vec{p}, G \wedge G', A) \in T^{\mathcal{TC}}}$$

$$\text{(Fail)} : \frac{(a, \vec{p}, G, A) \in T^{P'} \quad a \in \Sigma_?^{Can} \quad A_{\mathsf{Verdict}} = \mathsf{Verdict}' := \texttt{fail}}{(a, \vec{p}, G, A) \in T^{\mathcal{TC}}}$$

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

où $A'$ défini par $\begin{cases} A'_{\mathsf{Verdict}} = \mathsf{Verdict}' := \texttt{inconc}, \\ A'_v = A_v \text{ for } v \neq \mathsf{Verdict}, \end{cases}$

La première règle concerne les sorties, et permet de ne conserver que les transitions de sorties qui satisfont la condition nécessaire de rester dans $coreach^\alpha$. La seconde règle préserve les transitions d'entrée menant au verdict `fail`. Enfin la dernière règle sépare les autres transitions d'entrée en deux transitions : celles qui satisfont la condition nécessaire de rester dans $coreach^\alpha$ et les autres qui mènent au verdict `inconc`. La figure 15 décrit une vue schématique du renforcement des gardes.

**Exemple :** les rectangles près des localités de la figure 14 détaillent l'analyse de co-accessibilité. Le calcul du cas de test résultant pour l'exemple de l'ascenseur est illustré à la figure 16.

A l'issue de cette étape, le cas de test peut encore être simplifié, mais sans modification de sa sémantique, en effectuant une analyse d'accessibilité (rectangles associées aux localités de la figure 16 permettant de supprimer des transitions dont les gardes sont non satisfiables dans la sur-approximation de la localité de départ. On obtient alors le cas de test de la figure 17.

La suite de tests infinie composée des tests produits par l'algorithme de génération symbolique est correcte et ex-

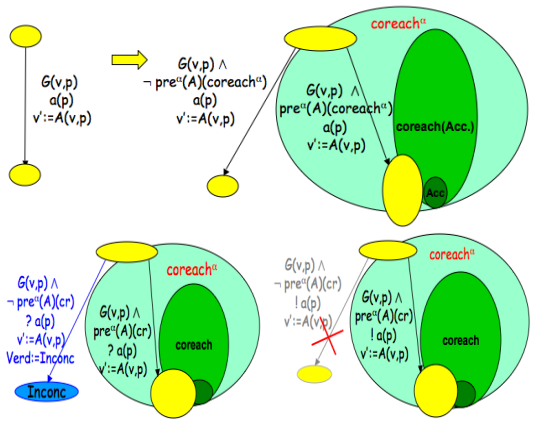


FIGURE **15. Vue schematique du renforcement des gardes**

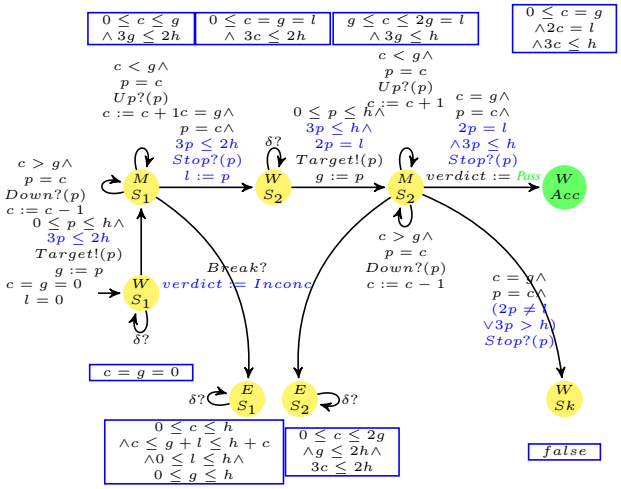


FIGURE **16. Test resultant de l'analyse de co-accessibilite approchee**

haustive. Comme dans le cas des IOLTS, la démonstration de la correction utilise le fait que le test est construit depuis $Can(\mathcal{S})$ et n'ajoute aucun verdict $Fail$. L'exhaustivité se démontre par la construction d'un objectif de test ciblant une non-conformité.

On peut alors se demander ce que l'on perd par une analyse de co-accessibilité approchée. En fait, cette analyse permettant de cibler les comportements acceptés par l'objectif, plus l'abstraction est précise (exacte à la limite), moins on conservera de comportements ne permettant pas d'atteindre l'objectif.

Les tests produits par l'algorithme de génération sont des programmes de test, possédant des variables, des gardes, des affectations. En particulier les paramètres des sorties du cas de test ne sont pas instanciées, mais caractérisées par des gardes. Le choix de valeurs de ces paramètres nécessite la résolution des gardes. Les entrées nécessitent seulement de vérifier les gardes. La figure 18

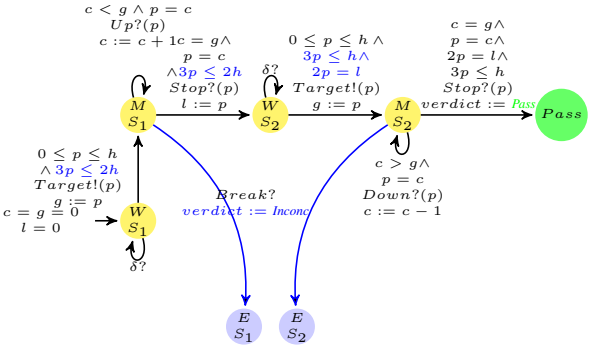


FIGURE **17. Cas de test final**

décrit deux exécutions du test généré pour une valeur de $h = 10$, dans le cas d'une abstraction par polyhèdres. Dans la première exécution, la garde sur la sortie $Target!$ a plusieurs solutions, la résolution produit par exemple l'étage but $p = 4$. Si l'implémentation est conforme, l'ascenseur monte, puis s'arrête en 4. La résolution de la garde pour la deuxième sortie $Target!$ produit 2, unique solution. L'ascenseur doit donc descendre, puis s'arrêter à l'étage 2. Pour la deuxième exécution, le choix d'une solution de la garde sur la première sortie produit 5. L'ascenseur monte jusqu'à 5 et s'arrête. La garde de la deuxième sortie $Target!$ n'a pas de solution entière, et on observe alors $?\delta$, ce qui produit le verdict *Inconc*. On remarque en effet que l'analyse n'est pas exacte : l'analyse polyhédrique ne permet pas d'inférer que le premier but doive être pair pour pouvoir ensuite aller à la moitié de la hauteur. Une analyse de congruence serait nécessaire dans ce cas.

## 4 Génération de tests pour des automates temporisés

Dans cette section, nous nous intéressons à la génération de tests pour des automates temporisés, qui permettent de modéliser des systèmes soumis à des contraintes temporelles. Nous nous focalisons sur les travaux de Krichen et Tripakis [KT04], parce qu'ils nous paraissent assez représentatifs des travaux sur de tels modèles, et étendent de façon naturelle les travaux introduits en section 2. En effet, la théorie du test sous-jacente se base sur la relation de conformité **tioco** qui étend **ioco**au cas temporisé. Les modèles considérés sont non-déterministes et partiellement observables, ce qui pose des problèmes de déterminisation. Enfin, deux types de tests sont considérés : des tests analogiques qui permettent de mesurer le temps précisément, et des tests numériques qui mesurent le temps par une horloge périodique, moins précis que les premiers, mais implémentables en pratique.

Il existe de nombreux travaux sur la génération de tests pour des modèles temporisés. Ceux-ci se fondent sur des modèles similaires, et des variations sur les relations de conformité. Citons par exemple [NS03] qui considère une sous-classe déterminisable d'automates temporisés,

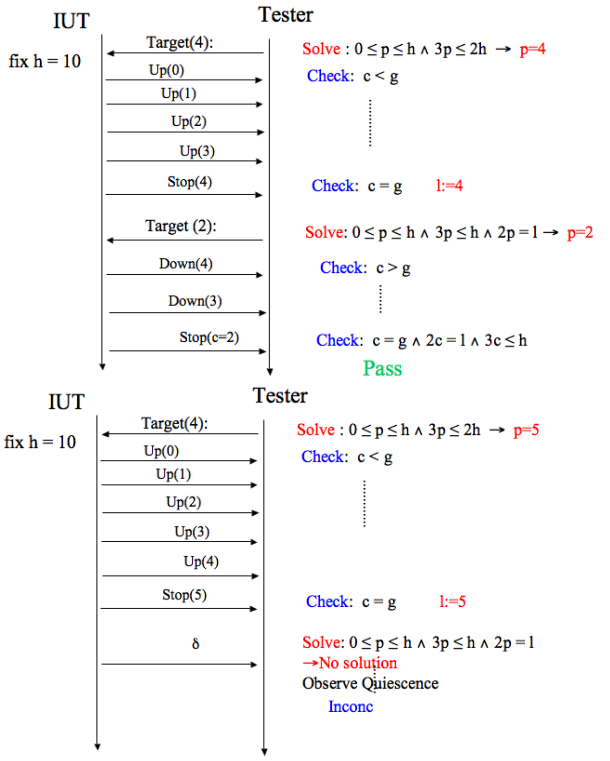


FIGURE **18. Execution des tests**

ou [BB05] qui utilise une variante de la relation de conformité utilisée ici. On trouvera une comparaison de ces relations dans [ST08].

### 4.1 Modèle des automates temporisés

**Définition 4.1** *Un automate temporisé à entrée sortie (TAIO pour* Timed automata with Inputs and Outputs*) est un quintuplet* $A = (Q, q_0, X, Act_\tau, E)$ *où*

- $Q$ *: est un ensemble fini de* localités, $q_0 \in Q$ *étant la* localité initiale,
- $X$ *est un ensemble fini d'*horloges,
- $Act_\tau = Act_? \cup Act_! \cup \{\tau\}$ *est l'ensemble d'*actions, *partitionné en entrées/sorties/actions internes.*
- $E$ *: est un ensemble de* transitions $(q, q', \psi, r, d, a)$ *avec*
  - $q$, $q' \in Q$ *les localités* d'origine *et* destination,
  - $\psi$ *est la* garde *i.e. une conjonction de contraintes de la forme* $x \# c$, $x \in X$, $c \in \mathbb{N}$ *où* $\# \in \{<, \leq, \geq, >\}$,
  - $r \subseteq X$ *est l'ensemble des horloges* réinitialisées,
  - $d \subseteq \{lazy, delayable, eager\}$ *est une* échéance,
  - $a \in Act_\tau$ *est l'*action.

**Exemple :** la figure 19 représente un TAIO spécifiant un double click.

La sémantique d'un TAIO $A$ est un IOLTS temporisé (TIOLTS pour *Timed Input Ouptut LTS*) $(S, s_0, Act_\tau \cup \mathbb{R}^+, T_d, T_t)$ tel que

- $S$ est l'ensemble infini d'*états* de la forme $s = (q, v)$ où $q \in Q$ est une localité et $v : X \to \mathbb{R}^+$ est une *valuation des horloges*. L'état initial est $s_0 = (q_0, \vec{0})$ où $\vec{0}$ est la valuation où toutes les horloges sont initialisées à 0.
- $T_d$ est un ensemble de *transitions discrètes* telles que $(q, v) \xrightarrow{a} (q', v')$ si $(q, q', \psi, r, d, a) \in E$, $v \models \psi$, et $v' = reset(r)\ in\ v$ est la valuation identique à $v$ sauf pour les horloges de $r$ remises à 0.
- $T_t$ est un ensemble de *transitions temporelles* telles que $(q, v) \xrightarrow{t} (q, v + t)$, $t \in \mathbb{R}^+$ si il n'existe aucune transition $(q, q', \psi, r, d, a) \in E$ tq
  $d = delayable \wedge \exists 0 \leq t_1 < t_2 \leq t, v + t_1 \models \psi \wedge v + t_2 \not\models \psi$
  or $d = eager \wedge v \models \psi$ i.e. une transition *eager* est tirée aussitôt que sa garde devient vraie, et une transition *delayable* doit être tirée entre le moment où sa garde est satisfate et le moment où sa garde n'est plus satisfaite, les transitions *lazy* étant les moins prioritaires.

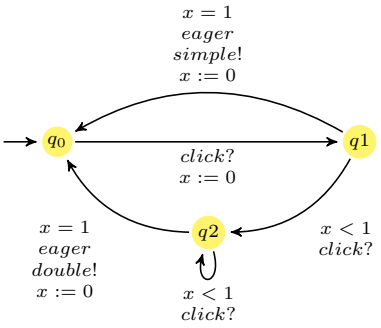


FIGURE **19. Exemple de TAIO**

L'ensemble $RT(ACT_\tau) = (Act_\tau \cup \mathbb{R}^+)^*$ est l'ensemble des séquences temporisées, par exemple $\rho = 2\ click\ .5\ click\ double$ est une séquence du TAIO du double click. $time(\rho)$ est la somme des délais d'une séquence, ici $time(\rho) = 2.5$. On note $proj_{Act}(\rho)$ la projection de la séquence $\rho$ sur l'alphabet observable $Act = Act_\tau \setminus \{\tau\}$ et préservant les délais e.g. $proj_{Act}(a\ 1\ \tau\ 2\ a\ 3) = a\ 3\ a\ 3$.

On dira qu'un état $s$ est *accessible* si $\exists \rho \in RT(Act_\tau), s_0^A \xrightarrow{\rho} s$. $Reach(A)$ dénote l'ensemble des états accessibles de $A$. Enfin l'ensemble des *traces temporisées* est défini par $TTraces(A) = \{proj_{Act}(\rho) \mid \rho \in RT(Act_\tau) \wedge s_0^A \xrightarrow{\rho}\}$.

Un TAIO $A$ est *input-complet* si tout input est accepté dans tout état : $\forall s \in Reach(A), \forall a \in Act_?, s \xrightarrow{a}$.

Un TAIO $A$ est *déterministe* si $\forall s, s' \in Reach(A), \forall a \in Act_\tau, s \xrightarrow{a} s' \wedge s \xrightarrow{a} s" \Rightarrow s' = s"$. Notons que la classe des TAIO déterministe est un sous-ensemble strict de l'ensemble des TAIO, et qu'il est donc en général impossible de déterminiser un TAIO. Il existe des sous-classes déterminisables, comme les Event Recording Automata où chaque événement est associé à une horloge remise à zéro à chaque occurrence de l'action correpondante [AFH99].

Un TAIO $A$ est dit *non-bloquant* si $\forall s \in$

$Reach(A), \forall t \in \mathbb{R}^+, \exists \rho \in RT(Act_! \cup \{\tau\}), time(\rho) = t \wedge s \xrightarrow{\rho}$, i.e. $A$ n'empêche pas le temps de progresser par l'attente d'une entrée.

### 4.2 Théorie du test pour les TAIO

Nous introduisons ici la relation de conformité **tioco** pour les TAIO qui étend la relation **ioco** au cas temporisé. Nous commençons par quelques définitions.

Pour une séquence observable $\sigma \in RT(Act)$, on note $A\ after\ \sigma \triangleq \{s \in S_A \mid \exists \rho \in RT(Act_\tau), s_0^A \xrightarrow{\rho} s \wedge proj_{Act}(\rho) = \sigma\}$ l'ensemble des états où $A$ peut résider après l'observation de $\sigma$ depuis l'état initial.

On note $elapse(s) = \{t > 0 \mid \exists \rho \in RT(\{\tau\}), time(\rho) = t \wedge s \xrightarrow{\rho}\}$ l'ensemble des délais possibles depuis $s$ avant l'observation d'une action observable.

Enfin, $out(s) = \{a \in Act_! \mid s \xrightarrow{a}\} \cup elapse(s)$ est l'ensemble des sorties et des délais observables en $s$.

**Définition 4.2** *Soit $S$ un TAIO non-bloquant, $I$ une implémentation inconnue, modélisable par un TAIO non-bloquant, input-complet.*

$I$ **tioco** $S \triangleq$ $\forall \sigma \in TTraces(S)$, $out(I\ after\ \sigma) \subseteq out(S\ after\ \sigma)$

*i.e. les sorties et délais de $I$ observables après une trace temporisée de $S$ doivent être spécifiés dans $S$.*

**Exemple :** la figure 20 illustre la relation **ioco**. On a $I_1$ **tioco** $S_1$ et $I_2$ **tioco** $S_1$, les gardes sur $b!$ dans $I_1$ et $I_2$ étant des restrictions sur la garde correspondante dans $S_1$. Par contre $\neg(I_3$ **tioco** $S_1)$ car $out(I_3\ after\ a\ 1) = (0,4] \cup \{b\} \not\subseteq (0,7] = out(S_1\ after\ a\ 1)$ et $\neg(I_4$ **tioco** $S_1)$ car $out(I_3\ after\ a\ 1) = (0,\infty) \not\subseteq (0,7]$. Pour la spécification $S_2$, on a $I_5$ **tioco** $S_2$, la sortie $e!$ n'étant plus tirable, et les gardes sur les autres sorties étant restrictives. Et $I_6$ **tioco** $S_2$, l'entrée $f?$ ayant été rajoutée, sans conséquences sur la conformité.

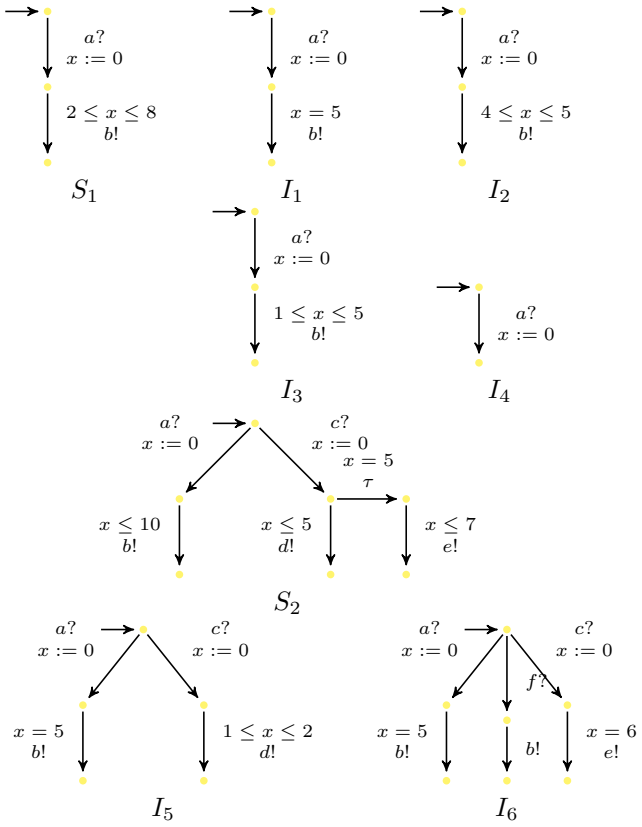


FIGURE **20. Illustration de tioco**

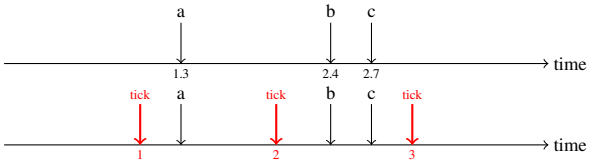


FIGURE **21. Difference entre tests analogiques et numeriques**

### 4.3 Génération de tests

On se propose maintenant de générer des tests à partir d'une spécification sous forme de TAIO, et pour la relation **tioco**. On suppose que le testeur est muni d'une horloge lui permettant de mesurer l'écoulement du temps. On distingue alors deux types de tests suivant la précision de l'horloge : les tests analogiques qui permettent de mesurer le temps de façon précise pour un temps dense (figure 21, haut), et les tests numériques qui mesurent le temps grâce à une horloge périodique en comptant le nombre de $tick$ produits par l'horloge (figure 21, bas).

#### 4.3.1 Génération de tests analogiques

L'algorithme de génération de tests analogiques repose sur le calcul symbolique d'ensembles d'états représentant la vision du testeur après une observation (action ou délai) depuis un ensemble d'états, i.e., effectuant une déterminisation pas à pas. Pour celà, nous définissons deux fonctions

- $dsucc(Q,a) = \{s' \mid \exists s \in Q, s \xrightarrow{a}_s{}' \}$ est l'ensemble des états accessibles depuis un ensemble d'états $Q$ du TAIO $S$ par l'observation de l'action observable $a$.

- $tsucc(Q,t) = \{s' \mid \exists s \in Q, \exists \rho \in RT(\tau), time(\rho) = t \wedge s \xrightarrow{\rho}_s{}' \}$ est l'ensemble des états accessibles depuis $Q$ par l'observation d'un délai $t$.

L'algorithme de génération est un algorithme à la volée, où les actions du testeur sont calculées pendant l'exécution. Initialement, le testeur calcule $Q_0 := tsucc(\{s_0^S\}, 0)$. Pour un état courant $Q$, le testeur choisit parmi les actions suivantes

- soit il décide de recevoir une action.
  - si $?a$ est reçu après le délai $t$, $Q$ est mis à jour par $S := dsucc(tsucc(Q,t), !a)$ ;
  - si aucun événement n'est reçu pendant un délai $t$,

$Q$ devient $Q := tsucc(Q, t)$ ;
si $Q = \emptyset$ le verdict fail est émis ;
- soit $dsucc(Q, ?b) \neq \emptyset$ pour une action $?b$ et il décide d'émettre l'action $b!$ et $Q$ est mis à jour par $Q := dsucc(Q, ?b)$
- soit il décide d'arrêter le test émet le verdict pass

**Propriétés des tests analogiques** Comme dans le cas des IOLTS et des IOSTS, l'exécution des tests se modélise par la composition parallèle du test et de l'implémentation. Cette composition parallèle se synchronise ici à la fois sur les actions communes et sur le temps, en supposant que les horloges des composants sont initialisées en même temps et qu'elles ont la même vitesse.

L'échec d'un test $T$ sur une implémentation $I$ se caractérise alors par $I \ fails \ T \triangleq fail \in Reach(I \| T)$

On peut démontrer que les tests générés sont corrects : $I$ fails $T \Rightarrow \neg(I$ **tioco** $S)$. La démonstration consiste à vérifier que tout verdict fail est émis sur une émission non spécifiée après une trace temporisée de $S$. Dans le cas de **ioco** pour les ioLTS et les ioSTS, la démonstration reposait sur la construction du testeur canonique. Pour les TAIO, on ne sait pas construire de testeur canonique en général puisqu'il faudrait déterminiser la spécification.

Les tests générés sont stritcts au sens suivant : $\neg(I \| T) \ tioco \ S \Rightarrow I$ fails $T$. C'est à dire que les tests détectent les non-conformité au plus tôt. Cette propriété est également vraie pour les tests générés pour **ioco** dans le cas des ioLTS et des ioSTS.

Enfin la suite infinie de tests générés est exhaustive : si $\neg(I$ **tioco** $S)$ alors il existe une exécution de l'algorithme qui produit le verdict fail. La démonstration consiste à guider l'algorithme de génération par une séquence non-conforme de $I$.

### 4.3.2 Génération de tests numériques

En pratique, les tests analogiques ne sont pas implémentables. Le temps est toujours mesuré par une horloge discrète qui ne permet par de dater de façon exacte les événements. Pour simplifier, on supposera ici que le testeur mesure le temps par une horloge périodique, appelé *automate de tick*, comme sur la figure 22, ici de périodicité 1. Le test peut alors être vu comme un ioLTS $D$ dont les actions sont étiquetées par les actions observables et les ticks. Il se synchronise avec l'implémenation sur ses actions communes et avec les ticks de l'automate de tick, l'implémentation et l'automate de ticks se synchronisant sur le temps. L'exécution se modélise donc par $I \| Tick \| D$.

Le test de conformité consiste alors à comparer $I \| Tick$ avec $S \| Tick$. La génération de tests s'appuie donc sur le produit $S \| Tick$.

L'algorithme nécessite l'introduction d'une nouvelle fonction de mise à jour
$usucc(Q) = \{s' \mid \exists s \in Q, \exists \rho \in RT(\{\tau\}), s \xrightarrow{\rho} s'\}$ pour un ensemble d'états $Q$ de $S \| Tick$, est l'ensemble

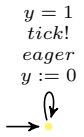


FIGURE **22. Automate de tick de periodicite** 1

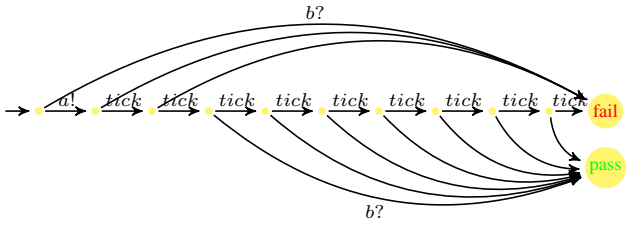


FIGURE **23. Cas de test numerique**

des successeurs d'états de $Q$ par une séquence d'actions inobservables. On remarque que les séquences en question sont de durée bornée car l'automate de Tick est de périodicité bornée.

L'algorithme de génération construit un IOLTS $D$, mais une exécution à la volée est également possible. Initialement on calcule $Q_0 := \{s_0^{S \| Tick}\}$, l'état initial de $S \| Tick$ ; Pour un ensemble d'états courant $S$, on choisit de façon non-déterministe l'une des actions suivantes :

- recevoir : $\forall a \in Act_! \cup \{tick\}$, si $Q' = dsucc(usucc(Q), a) \neq \emptyset$, ajouter la transition $Q \xrightarrow{a} Q'$, sinon ajouter la transition $Q \xrightarrow{a} fail$
- émettre : si $\exists b \in A_?$ tel que $Q" = dsucc(tsucc(Q, 0), b) \neq \emptyset$ ajouter la transition $Q \xrightarrow{a} Q"$ (noter qu'une émission se produit dès l'entrée dans l'état courant $Q$).
- arrêter et émettre le verdict Pass.

**Exemple :** la figure 23 représente un cas de test pour la spécification $S_1$ de la figure 20.

**Propriétés des tests numériques** Les tests numériques générés sont corrects : $I$ fails $D \Rightarrow \neg(I$ **tioco** $S)$.

Par contre ils ne sont ni stricts, ni complets en général. En effet, l'imprécision des horloges permet à une implémentation non-conforme de ne pas être détectée, par exemple si une réception se produit trop tôt ou trop tard, mais dans l'intervalle correct entre deux ticks par rapport à $S \| Tick$. La correction et la propriété d'être strict peuvent être obtenus si la spécification est *discrétisable* i.e. équivalente à un automate à temps discret. On pourra alors trouver un automate de ticks pour lequel la précision est suffisante pour rejeter toute implémentation non-conforme.

## 5 Conclusion

Nous avons introduit la théorie du test et des techniques de génération de tests pour trois modèles de systèmes réactifs, non-temporisés et énumérés dans les IOLTS, ou avec variables, gardes et affectations dans les IOSTS, et enfin temporisés dans les TAIO. Ceta aperçu, bien que partiel, a permis de montrer les problématiques sous-jacentes, comme les problèmes liés à l'observation partielle, à l'approximation dans l'analyse dans le cas des IOSTS, à l'implémentabilité et la précision des tests pour des horloges. Nous n'avons pas abordé les problèmes de couverture, où on cherche à couvrir les comportements des systèmes par des tests, et qui reposent en général sur des critères structurels. Nous n'avons pas non plus abordé les techniques de génération basées sur la résolution de contraintes utilisés dans plusieurs outils. Enfin il existe d'autres travaux intéressants sur d'autres modèles de systèmes réactifs comme les systèmes hybrides pour lesquels les recherches sont encore à leurs débuts. Il reste donc beaucoup de directions de recherche possibles pour étendre ces travaux, en améliorer l'efficacité, et les rendre disponibles dans des outils utilisables sur des systèmes réels.

## Références


[AFH99] R. Alur, L. Fix, and T. A. Henzinger. Event-clock automata : A determinizable class of timed automata. *Theoretical Computer Science*, 211 :253–273, 1999.

[BB05] E. Brinksma and L. B. Briones. A test generation framework for quiescent real-time systems. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, Revised Selected Papers*, number 3395 in LNCS. Springer, 2005.

[BFd+96] A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, and S. Mauw. Formal test automation : a simple experiment. In *International Workshop on the Testing of Communicating Systems (IWTCS'99)*, pages 179–196, 1996.

[BJK+05] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems : Advanced Lectures*, volume 3472 of *LNCS*. Springer, 2005.

[CIvdPS05] J. R. Calamé, N. Ioustinova, J. van de Pol, and N. Sidorova. Data abstraction and constraint solving for conformance testing. In *Proc. of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005), Taipei, Taiwan*, pages 541–548. IEEE Computer Society, December 2005.

[FTW04] L. Frantzen, J. Tretmans, and T. Willemse. Test generation based on symbolic specifications. In *4th International Workshop on Formal Approaches to Testing of Software (FATES 2004), Linz, Austria*, volume 3395 of *LNCS*. Springer-Verlag, 2004.

[JJ04] C. Jard and T. Jéron. Tgv : theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6, octobre 2004.

[JJRZ05] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *$11^{th}$ Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), Edinburgh, Scottland*, volume 3440 of *LNCS*. Springer, april 2005.

[JMR06] T. Jéron, H. Marchand, and V. Rusu. Symbolic determinisation of extended automata. In *4th IFIP International Conference on Theoretical Computer Science, 2006, Santiago, Chile*. SSBM (Springer Science and Business Media), August 2006.

[KT04] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In S. Graf and L. Mounier, editors, *11th International SPIN Workshop, Barcelona, Spain,*, number 2989 in LNCS. Springer, 2004.

[LG02] G. Lestiennes and M.-C. Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *13th International Symposium on Software Reliability Engineering (ISSRE'02), Annapolis, Maryland*. IEEE Computer Society Press, 2002.

[LY96] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. *Proceedings of the IEEE*, 84(8), 1996.

[NS03] B. Nielsen and A. Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer, STTT*, 5, 2003.

[ST08] J. Schmaltz and J. Tretmans. On conformance testing for timed systems. In F. Cassez and C. Jard, editors, *6th International Conference, FORMATS 2008, Saint Malo, France*, number 5215 in LNCS, pages 15–17. Springer, 2008.

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