RAPPORT DE RECHERCHE

L R I

MODELISATION ET TEST DE SYSTEMES COMPORTANT DES ACTIONS PRIORITAIRES

LESTIENNES G / GAUDEL M C

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

03/2006

Rapport de Recherche N° 1434

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

Modélisation et test de systèmes comportant des actions prioritaires

Grégory Lestiennes, Marie-Claude Gaudel

March 15, 2006

Il existent de nombreux formalismes permettant la modélisation de systèmes à entrées-sorties. Le choix du modèle utilisé pour décrire un système et les propriétés relatives à celui-ci dépendent du type de systèmes que l'on considère. L'un des points les plus débattus en ce qui concerne la modélisation des systèmes à entrées-sorties est la gestion de l'acceptation des entrées par le système. En effet, si certains systèmes sont réceptifs (input enabled) [13, 18, 19, 21, 4], d'autres le sont pas, i.e. ils sont capables de refuser des entrées dans certains états [8, 7, 2, 16, 15, 12]. Selon les approches, la non réceptivité des systèmes est traitée de différentes façons.

Dans cet article, nous proposons de classifier les systèmes, et de présenter les modèles utiliser pour les décrire. Cette classification mettra en évidence des systèmes pour lesquels peu de modèles existent. Ces systèmes sont ceux dont certaines actions, telles que des alarmes ou des interruptions, sont prioritaires. Le modèle des SyncCharts [1] de la famille des StateCharts [6] permet de définir des priorités entre actions. Ainsi dans un même état, chacune des actions possibles a une priorité différente. Le système spécifié est de ce fait totalement déterministe. Il n'est cependant pas toujours évident ni sensé de définir une priorité entre deux actions quelconques d'un système. De plus, lorsque l'on s'intéresse à la génération de test, la multiplicité du nombre de priorités fait exploser le nombre de tests à envisager. C'est pourquoi nous proposerons un modèle non déterministe distinguant simplement deux niveaux de priorité.

Dans la seconde partie de cet article, nous nous intéresserons au test de systèmes comportant des priorités. Cela nécessitera la modélisation de l'environnement dans lequel le système doit évoluer. Afin de tester la priorité des actions, ce modèle devra entre autres prendre en compte le fait que plusieurs actions peuvent être déclenchées simultanément.

1 Modèles existants

Dans cette section, nous proposons un aperçu de différents modèles utilisés dans le test de conformité pour représenter les systèmes à entrées-sorties, et donnons leurs spécificités. Le modèle de base est le modèle des LTS pour *Labeled Transition System*.

Définition 1 *Un LTS est défini par un tuple* $<S,L,T,s_0>$ *où:*

- S *est un ensemble fini non vide d'états;*
- L *est un ensemble fini d'actions;*
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ *est une relation de transition;* τ *modélise les actions internes non observables du système.*
- $s_0 \in S$ *est l'état initial.*

Ce modèle permet de décrire simplement les comportements d'un système. Les états d'un LTS représentent les états du systèmes, et ses transitions étiquetées représentent les actions possibles faisant évoluer le système d'un état à un autre.

Afin de modéliser de façon plus précise les systèmes à entrées-sorties, un modèle distinguant les entrées des sorties a été défini. Ce modèle est le modèle des $IOTS$ signifiant *Input Output Transition Systems*. L'ensemble des actions d'un $IOTS$ est divisé en deux sous-ensembles disjoints, l'un contenant les actions d'entrée et l'autre contenant les actions de sortie. La définition d'un $IOTS$ est la suivante:

Définition 2 *Un IOTS est un tuple* $<S,L=L_I \cup L_U,T,s_0>$ *où:*

- S *est un ensemble fini non vide d'états;*
- L *est un ensemble fini d'actions composé de 2 sous-ensembles disjoints:*
  L_I *est l'ensemble des actions d'entrée*
  L_U *est l'ensemble des actions de sortie*
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ *est une relation de transition;*
- $s_0 \in S$ *est l'état initial.*

De plus, les $IOTS$ possèdent la propriété suivante:

$$\forall s \in S, \forall a \in L_I, \exists s' \in S \mid (s, a, s') \in T \; ou \; (s, \tau, s') \in T$$

Cette propriété est la propriété de réceptivité qui implique que dans tout état, le système modélisé doit accepter n'importe laquelle de ses entrées. La réceptivité est une propriété de certains systèmes qui peuvent ainsi être modélisés à l'aide d'$IOTS$. Cependant en pratique, les systèmes ne sont pas tous réceptifs, et des modèles ont été définis qui permettent de décrire de tels systèmes de façon réaliste.

Nous avons tout d'abord le modèle des $MIOTS$ pour $Multi - Input\ Output\ Transition\ System$. L'ensemble des actions d'entrée et l'ensemble des actions de sortie d'un $MIOTS$ peuvent être partitionnés en plusieurs sous-ensembles disjoints. Dans tout état d'un $MIOTS$, toutes les actions d'un sous-ensemble sont soit possibles, soit refusées. Au niveau des entrées, on retrouve donc une certaine notion de la réceptivité dans le sens où dès qu'une action d'entrée est possible dans l'une des partitions, toutes le sont.

Ce modèle sert à modéliser des systèmes ayant des interfaces distribuées dont les canaux de communication peuvent être désactivés. C'est le cas pour un distributeur dans lequel on ne peut insérer de carte lorsqu'une autre est déjà à l'intérieur: le 'canal' utilisé pour insérer une carte est désactivé, par contre, celui correspondant aux touches du clavier ne l'est pas.

Nous avons ensuite le modèle des $RI-OLTS$ ($Restrictive\ Input$ - $Output\ Labeled\ Transition\ System$). Celui-ci reprend la notion de refus des entrées, mais sans faire de partition au sein des ensembles d'actions. Trois types d'actions d'entrées sont distingués : dans tout état, une entrée est soit possible, soit impossible, soit non spécifiée. Lorsque l'on modélise un système par un $RI - OLTS$, le fait de laisser des entrées non spécifiées permet de se limiter à la description des entrées possibles et impossibles. Les entrées non spécifiées sont quant à elles laissées à des choix d'implémentation.

Le dernier modèle dont nous parlerons ici est celui des $Team\ Automata$. Il s'agit d'un modèle très général dont le but est de décrire des systèmes à base de composants. Un $Team\ Automaton$ est défini comme le produit synchronisé de plusieurs automates composants. Le but est ici de modéliser le fait que les entrées de l'un des composants peuvent être les sorties des autres. Le type de modèle initialement utilisé pour décrire les $Team\ Automata$ s'apparente à celui des LTS, i.e. il est non réceptif, mais distingue tout de même les entrées, les sorties et les actions internes. Cependant, le concept des $Team\ Automata$ peut être repris en utilisant d'autres modèles pour décrire les composants pouvant notamment inclure la réceptivité [3].

2 Modélisation des systèmes et de leur environnement en vue du test

L'exécution d'un test consiste à faire interagir un processus de test avec le système.

Tout comme les systèmes eux-mêmes, les environnement dans lesquels ces systèmes vont évoluer, doivent être modélisés. Ces environnements possèdent certaines propriétés et selon le cas, l'un ou l'autre des modèles proposés dans la section précédente est plus adapté au type d'environnement considéré. La modélisation de l'environnement va permettre de définir des processus de test ayant les mêmes propriétés que celui-ci, en particulier le fait qu'il soit réceptif ou non doit être pris en compte dans le modèle. Il est ainsi possible de tester un système dans des conditions réalistes.

Plusieurs architectures de test peuvent être envisagées. Dans la plupart des cas, l'exécution d'un test est faite par mise en parallèle synchrone d'un processus de test avec le système. Une autre architecture de test, asynchrone celle-ci, consiste à mettre en parallèle le système avec deux processus, l'un fournissant des entrées au système et l'autre recueillant ses sorties.

Dans cette section, nous présentons différents contextes de test existants dépendants des systèmes considérés.

2.1 Test de systèmes réceptifs

Dans certains travaux effectués par Tretmans [18, 19, 21], le système est modélisé par un $IOTS$ et l'exécution d'un test est synchrone. Le modèle utilisé pour représenter les tests est celui des LTS. Du point de vue de la réceptivité, le système est donc réceptif, et l'environnement ne l'est à priori pas. On retrouve le même type de modèles avec Brinksma dans [4] pour les systèmes temporisés. Le système y est modélisé par un $Timed\ Input\ Output\ Transition\ System$ et les tests sont des $Timed\ Labeled\ Transition\ System$. Selon la façon dont les tests sont écrits, on peut, à l'aide de LTS (ou de $TLTS$), modéliser un environnement réceptif ou non.

Lors de l'exécution d'un test, pour qu'une action ait lieu, il faut que le processus de test et le système se synchronisent sur cette action. Le fait que les tests ne soit pas réceptifs implique que ceux-ci peuvent à priori empêcher le système de produire des sorties pour l'obliger à accepter une entrée. C'est d'ailleurs le cas dans [18, 19, 21]. Lorsque le test propose une entrée au système, il n'accepte aucune sortie de la part de celui-ci. Par contre lorsqu'il attend une sortie de la part du système, il s'attend à recevoir n'importe laquelle de ses sorties.

L'une des caractéristiques principales des tests non réceptifs est qu'ils peuvent diriger le déroulement de l'exécution du système: en effet, ces tests peuvent imposer certaines actions d'entrée en refusant les sorties que le système tenterait de produire. Cela a pour avantage de lever une partie du non déterminisme lié à l'exécution des tests. Il faut cependant pouvoir assurer que l'environnement dans lequel le système doit être implanté est effectivement non réceptif car sinon les résultats des tests pourraient être biaisés : bien que le système soit par définition réceptif, lorsque celui-ci veut produire une sortie et que l'environnement veut fournir une entrée, il est possible que les sorties du système passent avant ses entrées. Le scénario envisagé par un test qui force une entrée à être acceptée n'est dans ce cas pas réalisable sur le système dans son environnement final.

Lorsque l'on considère des environnements réceptifs, comme le fait Brinksma dans [4], les tests doivent donc accepter toute sortie du système, même lorsqu'ils lui proposent une entrée. Cela implique qu'il faudra parfois passer plusieurs fois un même test pour savoir si une entrée peut ou non être exécutée dans un état du système. De cette façon on peut vérifier lors des tests que certaines entrées peuvent avoir lieu. Par contre, comme dans tout test de systèmes non déterministe, le nombre d'exécutions de test étant borné, on ne peut pas être certain que celles qui n'ont pas lieu sont impossibles. Le non déterminisme dû au fait que les tests peuvent accepter les sorties du système alors qu'ils tentent de lui fournir une entrée peut engendrer des phénomènes de famine aussi bien en entrée du système qu'en sortie. La notion de priorité des actions prend ici tout son sens. Selon les systèmes que l'on considère, on devra privilégier tout ou partie des sorties aux entrées ou inversement, ou encore un mélange de ces solutions selon l'état du système dans lequel on se trouve. Nous revenons sur la notion de priorité en section 3.

2.2 Test de systèmes non réceptifs

2.2.1 Architecture asynchrone

Dans d'autres travaux de Tretmans [20], et dans ceux plus récents de Petrenko [15, 9], une autre architecture de test que celle consistant à synchroniser directement le processus de test et le système a été proposée. Les systèmes y sont également modélisés par des $IOTS$, mais l'architecture de test est basée sur le test à distance avec communications asynchrones. Cette architecture est appelée *queue testing*. L'interaction entre le test et l'implémentation se fait par l'intermédiaire de deux files d'attente FIFO, l'une stockant les entrées fournies à l'implémentation, et l'autre stockant ses sorties. Ainsi,

l'implémentation peut consommer ses entrées quand elle le "souhaite" en consultant son buffer en entrée. Le test et l'implémentation peuvent donc respectivement fournir une entrée vers la file d'entrée et produire une sortie vers la file de sortie, au même moment. Le système est réceptif par définition, puisqu'il est modélisé par un $IOTS$, mais l'environnement ne peut pas le forcer à accepter des entrées, celles-ci étant stockées dans la file en entrée. De cette façon, le système peut choisir de ne pas accéder à cette file afin de produire des sorties.

Pour que cette modélisation de l'environnement soit valable, il faut que le futur environnement d'exécution du système possèdent également des files d'attente en entrée et en sortie.

Bien que l'hypothèse de réceptivité soit adaptée à certains systèmes, notamment dans des domaines tels que celui des protocoles de communication, elle ne l'est pas toujours. En effet de nombreux systèmes refusent certaines de leurs entrées. Le concept de refus comporte deux aspects: on peut considérer qu'une entrée est refusée si son apparition elle même est impossible. Ce cas concerne plus particulièrement les actions physiquement impossibles (cache empêchant l'insertion d'une carte/de pièces, clavier bloqué...). Le second aspect concerne les entrées impossibles d'un point de vue logiciel. Dans ce cas l'action d'entrée elle-même a lieu, mais les effets attendus suite à cette action ne se produisent pas car cette action a été refusée. C'est le cas lorsque l'on tente d'activer une option grisée dans les menus déroulants d'une application ou lorsqu'un buffer plein reçoit et ignore un nouveau message.

Dans la section suivante, nous présentons les modèles existants permettant de définir des systèmes non réceptifs.

2.2.2 Architecture synchrone

2.2.2.1 Modélisation du système à l'aide de MIOTS

Dans [8, 7], Tretmans et Heerink modélisent le système et l'environnement à l'aide des $MIOTS$. Les processus de tests sont donc également modélisés par des $MIOTS$. À chaque étape, soit les processus de test proposent une entrée au système et vérifient si celle-ci est ou non acceptée tout en refusant les sorties du système, soit ils observent les sorties produites sur l'une des partitions des actions de sortie du système et refusent les sorties venant des autres partitions.

2.2.2.2 Modélisation du système à l'aide de RI-OLTS

Dans l'approche que nous avons proposée [12], le système est modélisé

par un $RI - OLTS$ et les tests sont des $IOTS$. L'environnement considéré est donc réceptif vis à vis des sorties du système. Ainsi, lorsqu'une entrée est proposée au système, il se peut que celui-ci produise une sortie avant que l'entrée puisse être prise en compte. Tout comme dans le cadre de test de Brinksma [4], on doit alors passer un test plusieurs fois avant de pouvoir déterminer si une action d'entrée est possible dans un état du système.

La réceptivité des processus de test que l'on trouve dans [4, 12] nous confronte au problème de priorité entre actions: lorsque le processus de test souhaite soumettre une entrée au système et que celui-ci souhaite produire une sortie, l'action qui sera finalement exécutée n'est pas définie. Or il existe des cas pour lesquels on voudrait être sûr que l'une ou l'autre de ces actions est prioritaire. C'est le cas lorsque l'on est en présence de systèmes comportant des alarmes ou des interruptions. Dans la section suivante, nous allons voir comment de telles actions peuvent prises en compte dans le modèle du système.

3 Modélisation de systèmes avec priorités

Le problème de la priorité ne se limite pas au conflit pouvant se produire lorsque l'environnement soumet une entrée au système sous test, et que ce même système tente de produire une sortie ou d'effectuer une action interne. En effet, un problème du même ordre se pose en cas d'arrivées simultanées de plusieurs entrées. Des priorités sont donc à définir aussi bien entre actions d'entrée et actions de sortie (ou actions internes), qu'entre actions d'entrée. En ce qui concerne le conflit entre sorties, on peut considérer que le système gère ses sorties correctement et ne génère pas ce type de conflit. De même on considère que actions internes et actions de sortie ne sont pas en concurrence.

3.1 Priorité des actions entre elles

Pour lever totalement l'ambiguïté relative à l'action devant être exécutée lorsque plusieurs actions sont proposées simultanément, il faudrait que, dans un même état, chaque action ait une priorité différente. En effet, à partir du moment où deux actions ont la même priorité, on ne peut pas définir celle qui doit être exécutée. Cependant, il n'est pas toujours évident ni sensé de définir une priorité entre deux actions lors de la spécification d'un système. Il paraît raisonnable de considérer des modèles avec un nombre réduit de niveaux de priorité autorisant la spécification d'actions de même priorité

dans un même état[1]. Le modèle décrit dans cette section distingue deux niveaux de priorité ce qui permet de modéliser une large classe de systèmes. Il peut être généralisé à un nombre quelconque de priorités.

Par ailleurs, les actions prioritaires peuvent varier d'un état à l'autre. En effet, une action correspondant par exemple à une annulation peut n'être prioritaire que dans certains états d'un système et interdite dans d'autres. On ne peut donc pas séparer les ensembles d'actions en deux sous-ensembles, l'un prioritaire et l'autre non, de telle façon que cette séparation soit valable pour tout état. Il faut attribuer une priorité à chacune des actions dans chacun des états du système.

3.2 Modèle

Le modèle que l'on propose est celui des $RI-OLTS$ avec priorités. Dans chaque état, chacune des actions spécifiées est définie comme prioritaire ou non.

Définition 3 *Un $RI-OLTS$ avec priorités est un 7-uplet $(Q, q_0, \mathcal{I}, \mathcal{U}, T_{NP}, T_P, T_\times)$ où:*

- *Q est un ensemble fini d'états,*
- *$q_0 \in Q$ est l'état initial,*
- *$\mathcal{I}$ est l'ensemble des actions d'entrée observables,*
- *$\mathcal{U}$ est l'ensemble des actions de sortie observables,*
- *$T_{NP} \subseteq Q \times (\mathcal{I} \cup \mathcal{U} \cup \{\tau\}) \times Q$ est l'ensemble des transitions d'actions non prioritaires,*
- *$T_P \subseteq Q \times (\mathcal{I} \cup \mathcal{U} \cup \{\tau\}) \times Q$ est l'ensemble des transitions d'actions prioritaires,*
- *$T_\times \subseteq Q \times \mathcal{I}$ est l'ensemble des transitions d'entrée impossibles,*

De plus, dans un premier temps, on considérera que l'on a les propriétés suivantes sur les ensembles de transitions:

1. $\forall (q, a) \in T_\times, \nexists\ q' \mid (q, a, q') \in T_{NP} \cup T_P$
2. *T_{NP} et T_P sont disjoints*

[1]Cela implique que nos modèles sont non déterministes, ce qui correspond à des cas que l'on rencontre en pratique.

Ces deux propriétés formalisent le fait que dans un même état, une action ne peut avoir plusieurs statuts (prioritaire, non prioritaire et interdite).

On donne ici les définitions des ensembles d'actions prioritaires et non prioritaires dans un état $q \in Q$. L'action spéciale δ est utilisée pour spécifier la quiescence d'un état; Elle modélise le fait qu'aucune action de sortie ne peut être exécutée dans l'état considéré.

Définition 4 *Soit* $R=(Q, q_0, \mathcal{I}, \mathcal{U}, T_{NP}, T_P, T_\times)$ *un* $RI-OLTS$ *avec priorités.*

On note $Prio(q) = \{a \in \mathcal{I} \cup \mathcal{U} \cup \{\delta\} | \ \exists q' \in Q, (q, a, q') \in T_P\}$ *l'ensemble des actions prioritaires dans l'état* q*,*

et $NonPrio(q) = \{a \in \mathcal{I} \cup \mathcal{U} \cup \{\delta\} | \ \exists q' \in Q, (q, a, q') \in T_{NP}\}$ *l'ensemble des actions non prioritaires dans l'état* q*.*

Soit $S=(Q, q_0, \mathcal{I}, \mathcal{U}, T_{NP}, T_P, T_\times)$ un $RI-OLTS$ avec priorités, nous utilisons les notations suivantes :

$q \xrightarrow{a} q'$ signifie que $(q, a, q') \in T_P \cup T_{NP}$.

$q \xrightarrow{a}$ signifie qu'il existe un état q' tel que $(q, a, q') \in T_P \cup T_{NP}$.

$q \xrightarrow{a} \times$ où $a \in \mathcal{I}$, signifie que $(q, a) \in T_\times$. L'action a est dite *impossible* dans l'état q.

$q \overset{a}{\not\longrightarrow}$ signifie qu'il n'existe pas d'état q' tel que $(q, a, q') \in T_P \cup T_{NP}$.

Classiquement, les actions d'entrée sont marquées d'un '?' et les actions de sortie sont marquées d'un '!'. Afin d'inclure la notion de priorité, ces symboles seront doublés lorsque l'action considérée est prioritaire. Ainsi, les notations suivantes sont utilisées dans un $RI-OLTS$ avec priorités:

$a?$: Action d'entrée a non prioritaire

$a??$: Action d'entrée a prioritaire

$a!$: Action de sortie a non prioritaire

$a!!$: Action de sortie a prioritaire

τ : Action interne non prioritaire

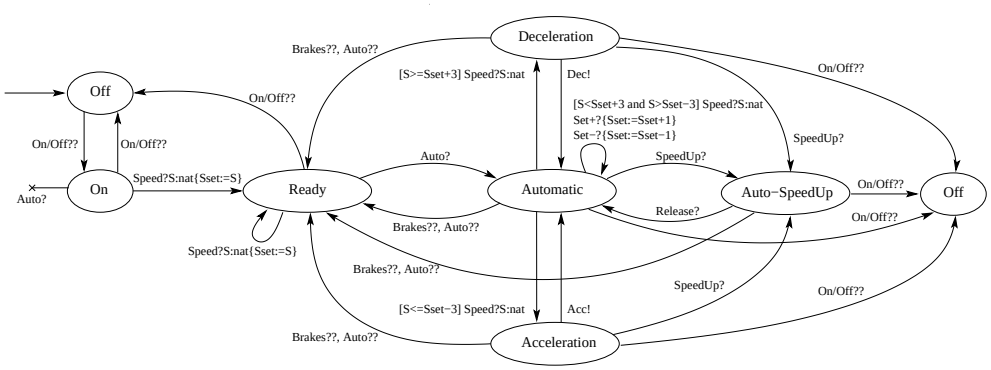
$\tau\tau$: Action interne prioritaire

3.3 Exemple

Nous présentons ici l'exemple d'un régulateur de vitesse modélisé à l'aide d'un $RI-OLTS$ symbolique avec priorités. Le fait que le modèle soit symbolique implique que ses actions peuvent être conditionnées par des gardes et peuvent comporter des symboles au lieu d'être totalement instanciées.

Des informations complémentaires sur les modèles symboliques comme les $EFSM$ ou les $IOSTS$ peuvent être trouvées dans [10, 17].

Le système interagit avec l'utilisateur et le véhicule.



Le système a 8 entrées:

- On/Off : Mise en marche et arrêt du régulateur (Entrée utilisateur)
- Set+ : Augmentation de la vitesse désirée (Entrée utilisateur)
- Set- : Diminution de la vitesse désirée (Entrée utilisateur)
- Auto : Activation du régulateur (Entrée utilisateur)
- Brakes : Activation de la pédale de freins (Entrée utilisateur, prioritaire quand elle est spécifiée)
- Speed(Nat) : Vitesse actuelle (Entrée véhicule)
- SpeedUp : Activation de la pédale d'accélération (Entrée utilisateur)
- Release : Désactivation de la pédale d'accélération (Entrée utilisateur)

et 2 sorties prioritaires lorsqu'elles sont spécifiées:

- Acc : Commande d'accélération (Sortie véhicule)
- Dec : Commande de décélération (Sortie véhicule)

Le système a également un paramètre

- Sset : Vitesse réglée

Le comportement du régulateur est le suivant:

Une fois que le régulateur a été mis en marche à l'aide de l'entrée On/Off, il faut attendre que la vitesse courante soit connue pour pouvoir activer le régulateur. La vitesse courante est obtenue par le biais de l'entrée $Speed$ et mémorisée dans le paramètre $Sset$.

Par la suite, tant que l'entrée $Auto$ n'est pas fournie pour activer le régulateur, la vitesse enregistrée dans $Sset$ est régulièrement mise à jour selon la vitesse courante.

Lorsque le régulateur est activé l'utilisateur peut augmenter ou diminuer la vitesse réglée à l'aide des entrées $Set+$ et $Set-$. Le régulateur continue de recevoir périodiquement la vitesse courante via l'entrée $Speed$. Si l'écart

entre celle-ci et la vitesse réglée dépasse strictement 2kms/h, une commande d'accélération (Acc) ou de décélération (Dec) est activée. Si l'utilisateur accélère(entrée $SpeedUp$) quand le régulateur est activé, le contrôle de la vitesse par le régulateur est suspendu jusqu'à ce que l'utilisateur relâche la pédale d'accélération (entrée $Release$).

Quand le régulateur est activé, il peut être désactivé à tout moment à l'aide du frein (entrée $Brakes$) ou de l'entrée $Auto$.

Quand le régulateur est en marche, il peut être éteint à tout moment à l'aide de l'entrée On/Off.

Notons que l'on aurait pu envisager un modèle plus précis avec trois niveaux de priorité permettant par exemple de définir laquelle des actions On/Off??, $Brakes$?? ou $Auto$?? est la plus prioritaire.

3.4 Test de la priorité

Que le futur environnement soit ou non réceptif, il nous faut des tests pouvant fournir simultanément plusieurs entrées au système en acceptant éventuellement ses sorties, afin de tester la priorité des actions. Un nouveau type de modèle est nécessaire pour décrire ces processus de test.

Le modèle que nous proposons pour les tests est celui des $CIOLTS$ pour Concurrent Input Output Labeled Transition System. Ce modèle permet de spécifier le fait que plusieurs actions sont activées en même temps. Un $CIOLTS$ est un tuple $(Q, q_0, \mathcal{I}, \mathcal{U}, T, T_\times)$ où:

Q est un ensemble d'états fini, q_0 est l'état initial, $\mathcal{I}$ est l'ensemble des actions d'entrée observables du $CIOLTS$, $\mathcal{U}$ est l'ensemble de ses actions de sortie observables, $T_\times$ l'ensemble des transitions d'entrée interdites et $T \in (Q \times \mathcal{P}(\mathcal{I} \cup \mathcal{U}) \times Q) \cup (Q \times \{\delta\} \times Q)$ est l'ensemble des transitions portant sur un ensemble non vide d'actions.

Les notations utilisées pour les actions d'entrées-sorties sont les notations classiques avec '?' et '!'. Les actions n'ont ici pas de priorité puisqu'elles peuvent avoir lieu simultanément. Afin de tester les actions interdites, on reprend la notation utilisée dans [12]: $\tilde{a}$ correspond au refus de l'action a.

Nous considérons une architecture de test synchrone. L'exécution d'un test consiste donc en sa mise en parallèle avec le système. Celui-ci étant modélisé par un $RI-OLTS$ avec priorités, et l'environnement étant modélisé par un $CIOLTS$, il nous faut donner un opérateur de parallélisation définissant le comportement d'une exécution en parallèle entre un $RI-OLTS$ et un $CIOLTS$.

Définition 5 *Soit C un $CIOLTS$ dont l'ensemble d'états contient q_C, et soit R un RI-OLTS avec priorités dont l'ensemble d'états contient q_R. L'ensemble*

des actions d'entrée (I) (resp. de sortie (U)) de R est le égal à l'ensemble des actions de sortie (U) (resp. d'entrée (I)) de C. L'opérateur de parallélisation $||$ entre un $CIOLTS$ et un $RI-OLTS$ avec priorités est défini comme suit:

- $q_R \xrightarrow{a} q'_R, q_C \xrightarrow{a_1,...,a_n} q'_C, a \in \{a_1, ..., a_n\} \cup Prio(q_R) \ \vdash q_R||q_C \xrightarrow{a} q'_R||q'_C$
- $q_R \xrightarrow{a} q'_R, q_C \xrightarrow{a_1,...,a_n} q'_C, a \in \{a_1, ..., a_n\} \cup NonPrio(q_R),$
  $\nexists a' \in \{a_1, ..., a_n\} \cap Prio(q_R) : q_R \xrightarrow{a'} \ \vdash q_R||q_C \xrightarrow{a} q'_R||q'_C$
- $q_C \xrightarrow{\delta} q'_C, \forall a \in \mathcal{I} \cup \mathcal{U} : q_C \not\xrightarrow{a} \ ou \ q_R \not\xrightarrow{a} \ \vdash q_R||q_C \xrightarrow{\delta} q_R||q'_C$
- $q_R \xrightarrow{a} \times, q_C \xrightarrow{a_1,...,a_n} q'_C, a \in \{a_1, ..., a_n\},$
  $\nexists a' \in \{a_1, ..., a_n\} \cap Prio(q_R) : q_R \xrightarrow{a'} \ \vdash q_R||q_C \xrightarrow{\tilde{a}} q_R||q'_C$

L'opérateur de parallélisation est compliqué du fait qu'on ne sait pas dans quel état du système on se trouve après avoir fourni plusieurs entrées simultanément. On ne sait pas quelle est celle qui a effectivement été exécutée.

Le nombre de tests possibles augmente de façon exponentielle avec le nombre d'actions d'entrée du système puisque, dans tout état, chacune des partitions de l'ensemble d'entrée peut donner lieu à un test. Si l'on veut tester de manière exhaustive une entrée particulière parmi les n entrées d'un système, il faut tester les 2^{n-1} partitions contenant cette entrée. Des opérations de sélection sont alors nécessaires. On peut par exemple ne tester que les ensembles à deux éléments contenant l'action que l'on souhaite tester. On peut aussi imaginer ne tester que la partition contenant toutes les entrées.

A titre d'exemple, nous donnons en Figure 1 quatre des tests linéaires permettant de tester l'action prioritaire *Brakes* pour le régulateur de vitesse. Ces tests ne sont pas réceptifs.

Si l'on souhaite obtenir des tests réceptifs, il faut compléter les tests avec les sorties du système. Un exemple d'un tel test est donné Figure 2.

4 Conclusion

Dans cet article, nous avons donné une classification des modèles permettant de décrire des systèmes et leurs environnements. Nous avons proposé un nouveau modèle afin d'inclure la notion d'actions prioritaires de certains systèmes lors de la modélisation et du test. Nous avons vu que le test d'actions prioritaires nécessite une architecture particulière dans le sens où il faut être capable de fournir simultanément plusieurs actions d'entrée. Par ailleurs le fait de pouvoir activer plusieurs entrées simultanément provoque une explosion du nombre de tests possibles. Comme toujours lorsque le nombre

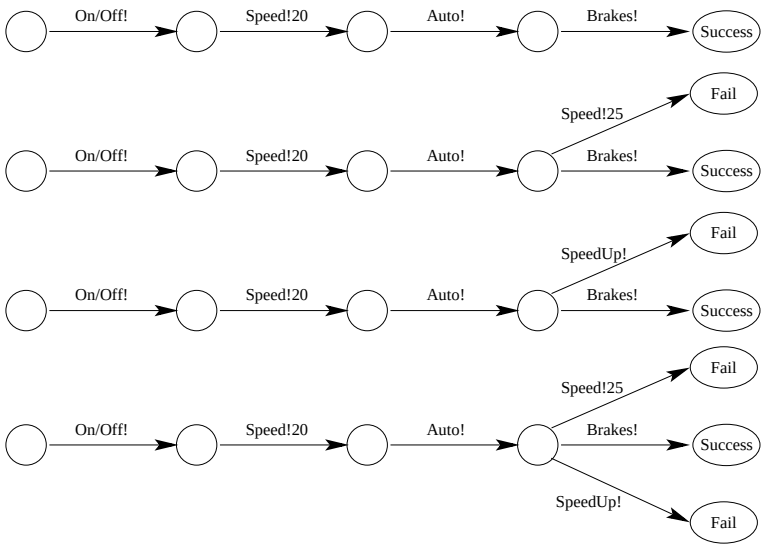


Figure 1: Exemple de tests non réceptifs

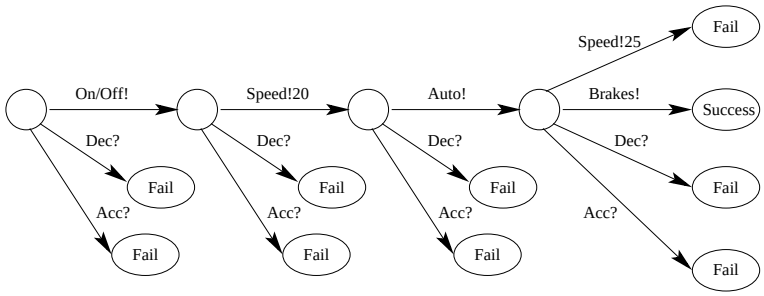


Figure 2: Exemple de test réceptif

exhaustif de tests est infini ou trop important pour être utilisé tel quel, il faut avoir recours à des méthodes de sélection. Des méthodes de sélection basées sur la couverture des transitions symboliques [11] peuvent être envisagées. On peut également s'inspirer des approches utilisées pour le test combinatoire [14, 5]. Celui-ci vise à tester un sous ensemble des combinaisons possibles des paramètres d'entrée d'un système. Dans notre cas, les combinaisons seraient basées sur le nombre d'actions d'entrée que l'on déclenche simultanément à la fin des tests linéaires.

References

[1] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC.

[2] M. Beek and J. Kleijn. Team automata satisfying compositionality, 2003.

[3] M.H. ter Beek and J. Kleijn. Modularity for Teams of I/O Automata. *Information Processing Letters*, 95(5):487–495, 2005.

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

[5] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.*, 23(7):437–444, 1997.

[6] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[7] L. Heerink. *Ins and Outs in Refusal Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1998.

[8] Lex Heerink and Jan Tretmans. Refusal testing for classes of transition systems with inputs and outputs. In *FORTE*, pages 23–38, 1997.

[9] J. Le Huo and A. Petrenko. On testing partially specified IOTS through lossless queues. In *TestCom*, pages 76–94, 2004.

[10] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A surve y. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[11] G. Lestiennes. *Contributions au test de logiciel basé sur des spécifications formelles*. PhD thesis, Université de Paris-Sud-Orsay, France, 2005.

[12] Grégory Lestiennes and Marie-Claude Gaudel. Test de systèmes réactifs non réceptifs. In *Journal Européen des Systèmes Automatisés – Modélisation des Systèmes Réactifs (MSR'05)*, pages 255–270, 2005.

[13] Nancy Lynch and Mark Tuttle. An Introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[14] Robert Mandl. Orthogonal latin squares: an application of experiment design to compiler testing. *Commun. ACM*, 28(10):1054–1058, 1985.

[15] A. Petrenko and N. Yevtushenko. Queued testing of transition systems with inputs and outputs. In *In Proceedings of the Workshop on Formal Approaches to Testing Of Software (Fates'02)*, pages 79–93, 2002.

[16] Alexandre Petrenko, Nina Yevtushenko, and Jia Le Huo. Testing transition systems with input and output testers. In *TestCom*, pages 129–145, 2003.

[17] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *International Conference on Integrating Formal Methods (IFM'00)*, LNCS 1945, pages 338–357. Springer Verlag, November 2000.

[18] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. In T. Margaria and B. Steffen, editors, *LNCS*, volume 1055, pages 127–146. Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96), 1996.

[19] J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baeten and S. Mauw, editors, *LNCS*, volume 1664, pages 46–65. CONCUR'99 - 10th Int. Conference on Concurrency Theory, 1999.

[20] Jan Tretmans and Louis Verhaard. A queue model relating synchronous and asynchronous communication. In *PSTV*, pages 131–145, 1992.

[21] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Component based testing with ioco. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Testing of Software (FATES)*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer-Verlag, 2004. Full report: TR–CTIT–03–34, University of Twente.